2007

Power Amplifier Linearization Implementation Using A Field Programmable Gate Array

Abilash Menon
University of Massachusetts Amherst

POWER AMPLIFIER LINEARIZATION IMPLEMENTATION USING A FIELD PROGRAMMABLE GATE ARRAY

A Thesis Presented

by

ABILASH MENON

Submitted to the Graduate School of the

University of Massachusetts Amherst in partial fulfillment

of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL AND COMPUTER ENGINEERING

September 2007

Department of Electrical and Computer Engineering

POWER AMPLIFIER LINEARIZATION IMPLEMENTATION USING A FIELD PROGRAMMABLE GATE ARRAY

A Thesis Presented

by

ABILASH MENON

Approved as to style and content by:

_____

Dennis Goeckel, Chair

_____

Russel Tessier, Member

_____

Wayne Burleson, Member

_____

Christopher .V. Hollot, Department Head

Electrical and Computer Engineering

To my Parents and Teachers

ABSTRACT

POWER AMPLIFIER LINEARIZATION IMPLEMENTATION USING A FIELD PROGRAMMABLE GATE ARRAY

September 2007

ABILASH MENON

B.S., E&C,. KERALA UNIVERSITY, TRIVANDRUM

M.S.E.C.E, UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Dr Dennis Goeckel

The emphasis on higher data rates, spectral efficiency and cost reduction has driven the field towards linear modulation techniques such as quadrature phase shift keying (QPSK), quadrature amplitude modulation (QAM), wideband code division multiple access (WCDMA), and orthogonal frequency division multiplexing (OFDM). The result is a complex signal with a non-constant envelope and a high peak-to-average power ratio. This characteristic makes these signals particularly sensitive to the intrinsic nonlinearity of the RF power amplifier (PA) in the transmitter. The nonlinearity will generate intermodulation (IMD) components, also referred to as out-of-band emission or spectral re-growth, which interfere with adjacent channels. Such distortion, or so called Adjacent Channel Interference (ACI), is strictly limited by FCC and ETSI regulations. Meanwhile, the nonlinearity also causes in-band distortion which degrades

the bit error rate performance. Typically, the required linearity can be achieved either by reducing power efficiency or by using linearization techniques.

For a Class-A PA, simply "backing off" the input power level can improve linearity; however, for high peak to average power ration (PAPR) signals, this normally reduces the power efficiency down to 10% while increasing heat dissipation up to 90%. When considering the vast number of base stations that wireless operators need to account for, increasing power consumption, or in other words, power back-off is not a viable tradeoff. Therefore, amplifier linearization has become an important technology and a desirable alternative to backing-off an amplifier in modern communications systems.

In this work, a novel adaptive algorithm is presented for predistorter linearization of power amplifiers. This algorithm uses Pade-Chebyshev polynomials and a QR decomposition followed by back substitution to find the pre-distorter coefficients. This algorithm is implemented on a Field Programmable Gate Array (Stratix 1S80).The implementation provides improved linearization and also runs the algorithm fast enough so that the adaptive part can be done quickly. Yet another challenge was the integration of a transmitter, receiver and this adaptive algorithm into a single FPGA chip and its communication with a base station. The work thus presents a novel pre-distortion implementation technique using an FPGA and a soft processor (Nios 2) which provides significant intermodulation distortion suppression.

TABLE OF CONTENTS

Page

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION

Nonlinear amplification yields intermodulation distortion (IMD) products and results in unacceptable spectral regrowth in the adjacent channels. Modern communications systems have been designed to take advantage of the high spectrum efficiency offered by complex modulation schemes such as quadrature amplitude modulation (QAM).But highly linear amplification is required for complex modulation formats. In particular , such schemes are far more susceptible to distortion than were the relatively simple modulation schemes of the past. Besides causing intersymbol interference (ISI) which raises the bit error rate, distortion can spread the transmitted spectrum, making it difficult to comply with FCC regulations. Therefore all components in such a system must be highly linear. Unfortunately the system power amplifier (PA) must be operated in the nonlinear region close to saturation in order to exhibit power efficiency. To achieve highly linear amplification, special linearization techniques are usually employed. The three main linearization methods that are used are : the predistortion method, the feedback method, and the feed forward method.

Linearization has a number of advantages. It provides spectral efficiency which helps in the use of sophisticated modulation techniques and high speed data transmission. It also enables the use of class AB, B or high efficiency Doherty amplifiers instead of the conventional class-A amplifier, which requires more power than the former three. Thus linearization lowers the overall cost and, with technological

development, it is easy to upgrade the firmware and software to accommodate new features. Thus it is worth investigating techniques that provide linear amplification.

The efficiency of power amplifiers (PAs) may be improved by using predistortion (PD). In this work, a new scheme is proposed where the PD functions are estimated based on an adaptive algorithm. The memoryless part of the predistorter uses a type of orthogonal polynomial – Padé Chebyshev and a QR-decomposition recursive least square (QRD-RLS) update algorithm. This improves the system robustness and adaptation speed and can be used for a wide range of modulation schemes.

The work implements the entire algorithm and the predistorter into a single chip (Field Programmable Gate Array).The transmitter and receiver chain is included in the chip along with the predistorter. A programmable Tx/Rx chain frequency response correction is also provided. The system would transmit at intermediate frequency(IF) , and there would be no in-phase/quadrature(I/Q) mismatch and balance problem.

A Nios 2 soft processor is the platform used for the adaptive algorithm as it provides the flexibility to change parameters on the run. The other modules are written in hardware description language and interfaced with Nios 2.The challenge is to build the whole system together and provide fast and accurate results.

Figure 1.1 shows an example plot of power amplifier input vs. output. The ideal curve for a power amplifier is a linear curve (straight green line in the figure).But due to the non-linearity present in the PA, the actual curve would not be linear .This is shown as the blue curve in the figure. The linearization method followed in this thesis applies an inverse function, called the pre-distortion curve .By applying an inverse function , the overall PA can be made linear.

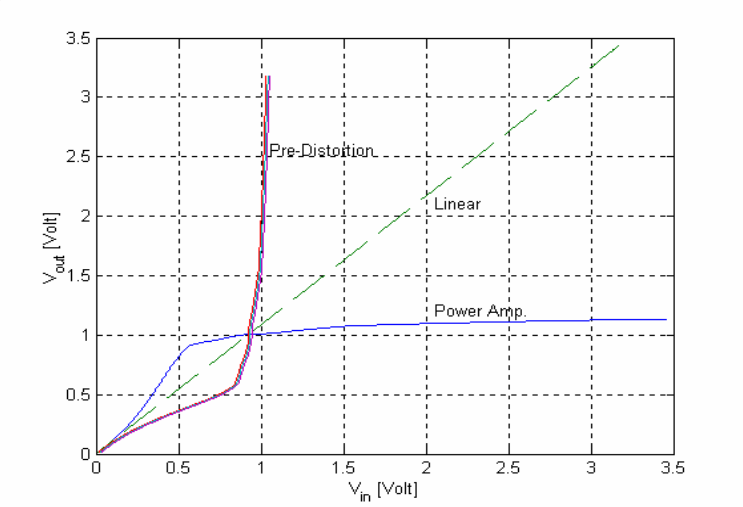

Figure 1.1: Amplifier Linearization

Some preliminary results were obtained regarding the implementation details of the adaptive algorithm in Nios 2. The algorithm can be implemented using fixed point or floating point arithmetic, each having its own limitations. These initial results gave a starting point for the final implementation, which provides a new adaptive algorithm that includes the above mentioned features. Its implementation details are presented in this work. To our knowledge ,this is the first implementation of this type of algorithm and pre-distorter in a single chip.

The rest of the document is organized as follows: In Chapter 2, the existing PA linearization techniques are reviewed, and their shortcomings are discussed. In Chapter 3, the predistorter method adopted for the new scheme is proposed and the details of the adaptive algorithm are explained. The proposed architecture for the transmitter and receiver chain and also the key hardware specifications are stated in Chapter 4.The hardware and software implementation details are discussed in the Chapter 5.The results

and observations are discussed in Chapter 6.Finally a summary is provided in Chapter 7.

CHAPTER 2

BACKGROUND

2.1 Power Amplifier Linearization Schemes

There are a variety of RF power amplifier linearization schemes. To date,no single scheme dominates for general-purpose use. The best scheme to use depends on many parameters such as efficiency, complexity, modulation scheme, bandwidth, Adjacent Channel Interference (ACI) specification, and dynamic range. A brief summary of some commonly know RF power amplifier linearization schemes are given below. All linearization methods are limited in their maximum correctable range, which is the region of power output level near the onset of saturation.

2.1.1 Boot up Bias

The simplest and most obvious way to improve the linearity is to increase the power amplifier bias points, i.e. drive the amplifier toward Class-A operation. This is equivalent to reducing the input power level of the power amplifier.As a result, the power amplifier will operate in the small signal linear region and the corresponding out-of-band emission level will decrease. This brute force method comes with a price of lowering the overall efficiency of the power amplifier , while reducing the total RF output power. This can be a fast fix for some applications due to its simplicity.

2.1.2 Dynamic Bias

As discussed in Section 2.1.1, simply increasing the DC bias for a Class-A amplifier is an inefficient way to linearize a power amplifier. However, if the bias level

can adaptively change with the input envelope of the RF signal so that the power amplifier dissipates as little power as possible while it maintains a reasonable out-of-band emission level, such a technique could be very practical. The two diagrams of commonly used open loop and close loop dynamic bias networks are shown in Figure 2.1 and Figure 2.2 respectively.

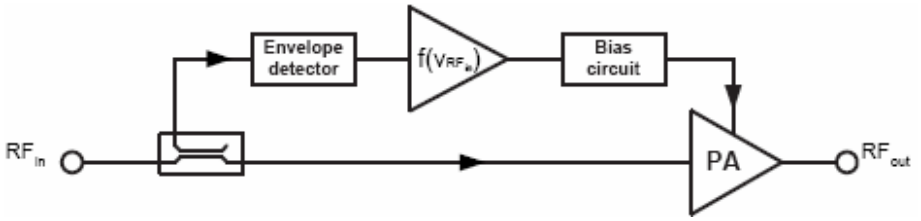

Figure 2.1 : Diagram of Open Loop Dynamic Bias

According to the literature [2-3], the amplifier's 1-dB compression point can be bumped a few dB by using the dynamic bias method. This method requires a fast speed wideband envelope detector and a DC-DC converter with high current capability, which is currently a challenge for the power supply industry. The performance of a dynamic bias system could be corrupted by undesired phase distortion occurring when relatively large changes in the bias level happen at a higher power level. Although this problem could be improved by simultaneously adapting a phase feedback loop [4], this adds another dimension of complexity , which is nontrivial in an RF application.

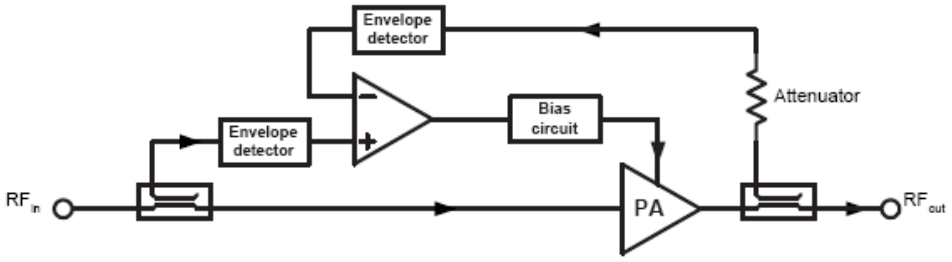

Figure 2.2 : Diagram of Close Loop Dynamic Bias

2.1.3 RF Feedback

Another simple way to perform linearization is to use feedback techniques which adopt the principle of operational amplifiers. For RF amplification, however, many stages are normally required to get enough gain, which reduces the overall efficiency since each stage uses power. More importantly, the delay per RF amplifier stage will cause instability if global feedback is used. Hence, not many practical applications employ RF feedback as a linearization approach.

2.1.4 Baseband Envelope Feedback

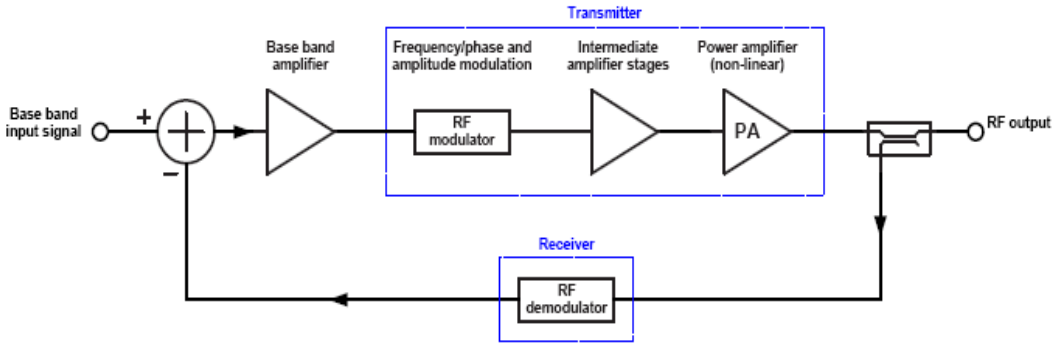

Figure 2.3 : Diagram of Baseband Feedback

The RF feedback technique requires the components in the feedback path to operate at a higher frequency band or large bandwidth situation. As shown in Figure

2.3, the main amplifier can also be linearized by feeding back the baseband signal rather than the RF signal. First of all, the baseband signal is modulated onto the RF carrier and amplified by the power amplifier, and then the power amplifier output is taken, demodulated and fed back to the input to predistort the input of the high gain baseband amplifier such that the output of the main amplifier is linearized. The demodulator is assumed to be linear and distortion free at the bandwidth of interest. In order to maintain system stability, the loop bandwidth must be within the MHz range. Therefore, the main disadvantage of this system is the narrow bandwidth and, in some cases , complexity [1].

2.1.5 Polar Feedback

The polar feedback technique overcomes the fundamental inability of envelope feedback to correct for AM-PM distortion effects. It is a baseband feedback scheme where the envelope- and phase-feedback functions operate independently as show in Figure 2.4. Polar feedback scheme provides relatively high efficiency since the power amplifier can operate completely nonlinearly, and this method will be robust since it has both forms of feedback. Since both amplitude and phase are corrected in the polar feedback system, variations in temperature, load, and manufacturing should be mitigated.

The key disadvantage of polar feedback lies in the generally different bandwidths required for the amplitude and phase feedback paths. This usually leads to a different level of improvement of the AM-AM and AM-PM characteristics and a poorer

overall performance than that is achievable from an equivalent Cartesian-loop transmitter.

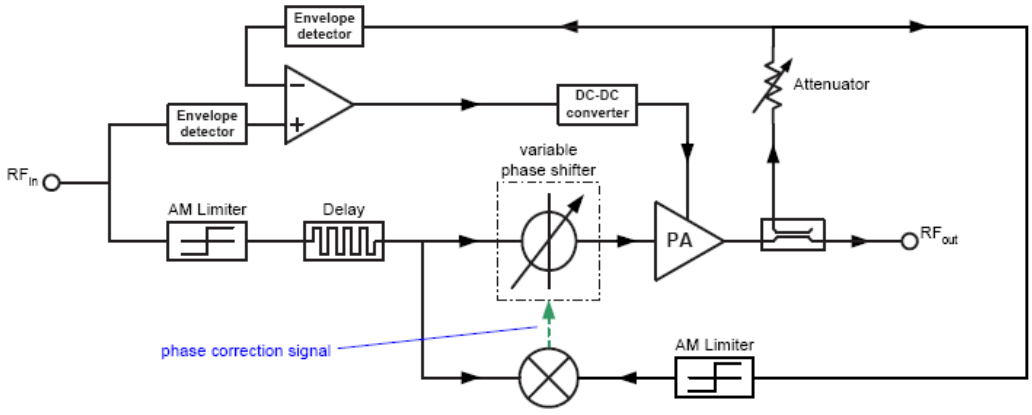

Figure 2.4: Diagram of Polar Feedback

A good example of the difference occurs with a standard two-tone test, which causes the phase-feedback path to cope with a discontinuity at the envelope minima. In general, the phase bandwidth must be five to ten times the envelope bandwidth, which limits available loop gain for a given delay. For a narrowband application, the improvement in two-tone IMD is typically around 30 dB [5].

2.1.6 Cartesian Feedback

Cartesian Feedback was first proposed by Petrovic [6]. The fundamental idea is to I-Q modulate the carrier before passing it to a nonlinear but efficient RF power amplifier as shown in Figure 2.5. The forward path of the system consists of the main control loop gain and compensation filters, a synchronous I-Q modulator, and the antenna acting as an output load. The feedback path obtains a portion of the transmitter

output via an RF coupler, the signal from which is then synchronously demodulated and fed back to perform the linearization. The loop control characteristics are established by the gain and the compensation filters. The level of intermodulation distortion reduction is essentially dominated by the loop gain, and the compensation allows the stability and behavior of the system to be controlled. Synchronization between the modulator and demodulator is obtained by splitting a common RF carrier. Due to RF path differences in the forward and feedback paths, a phase adjuster is necessary to maintain the correct relationship between the input signals and feedback signals.

Cartesian Feedback can automatically compensate for drifts in amplifier nonlinearities due to temperature and power supply variations. However, this technique is only conditionally stable and the setting of the adjuster with the aim of maintaining stability is one of the key problems. Amplifier nonlinearities also affect stability as does excessive baseband phase shift. Another limiting factor in this system is the nonlinearities of the down converting mixers [1]. But the main disadvantage of this scheme is the narrow bandwidth that is somewhat inherent in baseband feedback systems.

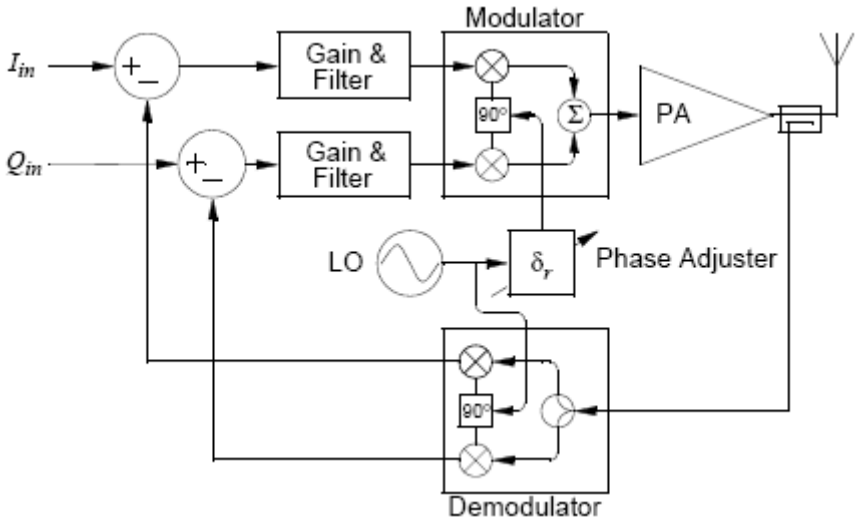

Figure 2.5 : Diagram of Cartesian Feedback

2.1.7 Envelope Elimination and Restoration (EER)

The EER linearization method was first proposed by Khan [7]. Figure 2.6 shows the block diagram of the prototype implementation of a closed loop version of the EER linearization scheme. As shown in Figure 2.6, the envelope of the RF input is first eliminated by a limiter to generate a constant amplitude phase signal. At the same time, the magnitude information is extracted by an envelope detector. The magnitude and phase information are amplified separately and then recombined to restore the desired RF output via a high efficiency switched-mode RF power amplifier. A feedback path from the RF output of the power amplifier to the input of the switching power supply guarantees amplitude tracking between the RF input and RF output waveforms.

The key advantage of EER approach is that the RF PA always operates in an efficient switched mode. That is why the EER system can linearize the switched-mode RF PA without compromising its efficiency.

There are a few disadvantages. Normally, the restoration is accomplished via biasing the power amplifier's drain voltage. As the drain voltage is varied to correct the output amplitude of the power amplifier, the phase varies also. Too much unintended phase modulation increases spectral regrowth above specifications.

Another typical disadvantage of EER is the slowness of the envelope restoration feedback loop. Practically, EER only has on the order of 20-30 dB of dynamic range. Even when the bias level to the power amplifier is zero, some AC power bleeds through.

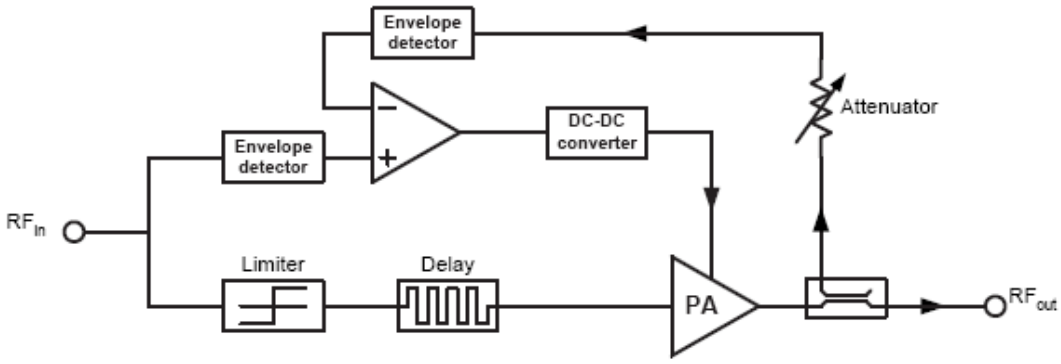

Figure 2.6: Diagram of Envelope Elimination and Restoration

2.1.8 Adaptive Feed-forward

As with most linearization methods, the feed-forward technique is not a new idea. It was invented as means of distortion reduction in telephone repeaters by Black in 1923 [8]. This technique is usually applied directly at RF and the block diagram of an

adaptive feed forward scheme is shown in Figure 2.7. Such an architecture has been used successfully to linearize many power amplifiers.

The principle of feed forward can be described as follows: A non-distorted signal goes into the power amplifier and also into a variable gain/phase amplifier in Canceler #1. The adaptive system samples the power at point "A" and tweaks the gain and phase of Canceler #1 such that the power at point A is minimized. When the power is minimized, only the distortion from the power amplifier remains at point "A". This distortion then passes through Canceler #2 which has its gain and phase adaptively adjusted to minimize the total power at point B. The only way to minimize the power at point B is to cancel the distortion from the power amplifier.

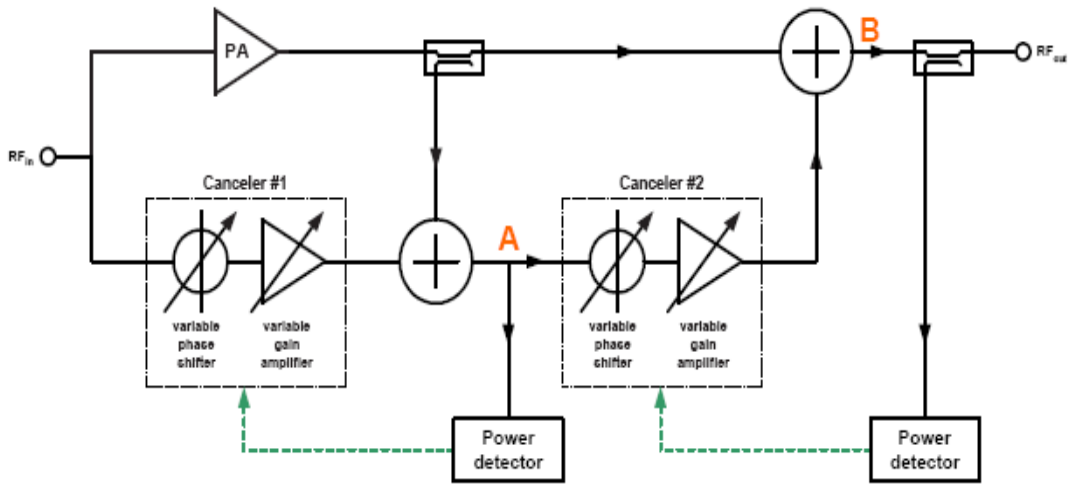

Figure 2.7: Diagram of Adaptive Feed Forward

Feed forward linearization can deliver reasonable linearization performance (20 dB-40 dB improvement) over relatively wide bandwidths (3 MHz-50 MHz) and has the

advantage of inherent stability [9-10]. However, there are a couple of underlying assumptions that must be true for this scheme to work. First of all, it is assumed that the power amplifier generates the dominant non-linearity. Additionally, the Canceler #2 amplifier must be linear and must have a high enough output power capability to overcome the loss through the output coupler. The efficiency of the feed forward system is reduced by the power consumption of the Canceler #2 amplifier. Amplitude and phase matching is a problem since amplifier characteristics tend to drift with temperature and time, and also vary with manufacturing tolerances. Adaptive techniques can enable the performance of the system to be maintained despite these effects; therefore, A DSP processor has to be used at this point to implement the adaptive algorithm[11-12].

2.1.9 Predistortion Method

As shown in Figure 2.8, the basic concept of a pre-distortion system involves the insertion of a nonlinear element prior to the RF power amplifier such that the combined transfer characteristic of both is linear. From a mathematical point of view, if G[•] is the mathematical model of a power amplifier, the pre-distorter F[•] is such a function that enables H[F(V_i)] to be a linear function of the Input Vi, for example, if G(V_p)=K*V_p^3, then F(V_i)= $V_i^{(1/3)}$ and Vo =H[F(V_i)]=K V_i.

Predistortion can be accomplished at either RF or baseband. The practical operational bandwidths of most RF pre-distortion techniques is similar to, or greater than, those of feed-forward, and the RF pre-distortion techniques can be easily combined with other linearization methods to obtain higher efficiency and linearity than

with only one linearization method. The degree of cancellation is dominated by memory effects in the PA, the gain and phase flatness of the pre-distorter and the RF power amplifier itself.

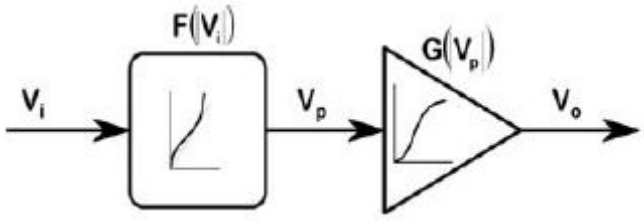

Figure 2.8: Diagram of Pre-distorter Concept

Although better performance can be achieved with more complex forms of RF pre-distortion such as Adaptive Parametric Linearization (APL®), which is capable of multi-order correction [13], a digital pre-distorter is more flexible with better correction and adaptation capability for industry application. The considerable flexibility and processing power now available from DSP devices allows users to update the required pre-distortion characteristic easily to achieve maximum correction while maintaining the system performance as the environmental changes, such as the temperature and device characteristics that drift over time.

2.2 Digital Predistortion

Digital predistortion can operate with analog-baseband, digital-baseband, analog-IF, digital-IF, or analog-RF input signals. Digital-baseband and digital-IF processing are most commonly used by most engineers and scientists. Although many versions of digital predistorters has been developed in the past several years and could be categorized with respect to many criteria, two main groups can be distinguished One of them is a look-up-table (LUT) predistorter while the other one is a parametric predistorter with an analytical formulation (such as Volterra kernel based predistorter). The overall performance of digital predistortion is dominated by both the structure of the predistorter itself and the adaptive algorithm. The speed and the complexity of the predistorter adaptation is one of the crucial problems for their practical implementation. These two issues along with some detailed description of digital predistortion fundamental knowledge are addressed in the coming sections.

2.2.1 Magnitude and Phase Mismatch and Signal Cancellation

The goal of power amplifier linearization is to cancel the distortion components while improving the overall power efficiency. The distortion components are deterministic signals that vary with the instantaneous amplitude and modulation frequency of the signal. A study of signal cancellation shows that good cancellation performance places very tight requirements on the amplitude and phase match between the distortion components of the amplifier and signal components generated in the predistorter. This cancellation is demonstrated in Figure 2.9. The power of the residual

IMD component can be calculated using the cosine rule, and the required matching for a given degree of cancellation is shown as

$$\textbf{Cancellation} = 10 \log [1 - (A + \Delta A/A) \cos(\Delta\varphi) + (1 + \Delta A/A)\, 2]$$

$\Delta\varphi$ and ΔA are the phase and amplitude errors, respectively. Numerical values are shown in Figure 2.9. For example, to achieve a 25 dB reduction in IMD components, the phase error cannot be bigger than 2-3 degrees and a gain matching $\Delta A/A$ (flatness) must be better than 0.25 dB (3%) over the entire signal and IMD band [13].

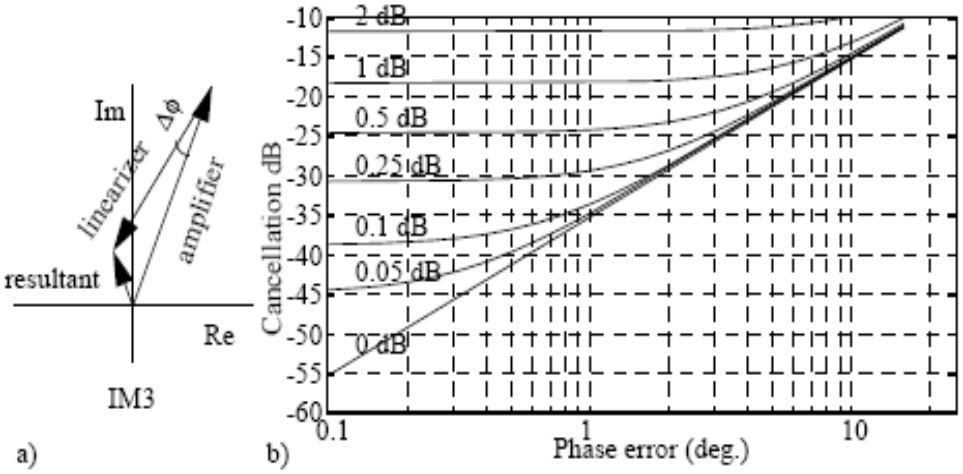

Figure 2.9 : Principle of Distortion Cancellation.

In practice, the limiting factor is nearly always the bandwidth over which a given accuracy can be obtained plus the system noise level, especially the close-in phase noise performance. The details will be discussed in a later section.

2.2.2 Direct Learning Adaptive Digital Predistortion Algorithm

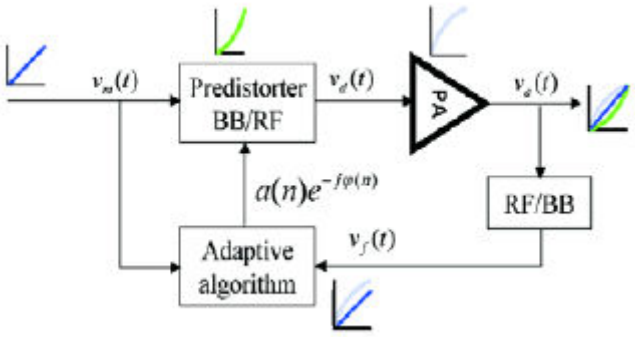

Figure 2.10 : Direct Learning Architecture of Digital Predistorter

Figure 2.10 illustrates a block diagram of a direct learning predistortion algorithm. At the start of the predistortion session, the complex gains $a(n)e^{-j\varphi(n)}$ are normalized to unity "1". After that, for each input baseband sample $v_m(n)$, the predistorter gain and phase $a(n)e^{-j\varphi(n)}$ is generated using an error signal that is based on the difference between the power amplifier output distorted baseband sample $v_f(n)$ and its corresponding undistorted input sample $v_m(n)$. The predistorter gain and phase are set such that the overall combination response of the PA and the predistorter becomes a linear system. This means that the predistorter is actually acting as an inverse PA nonlinearity pre-equalizer.

For the memoryless case, a look-up table of predistorter gain values then can be stored for every possible input envelope value of $v_m(n)$. The table entries then become a sample-by-sample complex scaling of the modulation before it is sent to the PA. This scaling will then cancel the undesired nonlinear response of the PA. To do this, a mathematical algorithm is used to update this table based on a snapshot record of both the input and the output of the PA. This method has widely been used. However its effectiveness in inverse equalizing the PA deteriorates when the PA suffers electrical or

electro-thermal memory. Due to the fact that the same input sample no longer has a single distinct inverse value at such a situation, it is no longer possible to predict the inverse of the PA. Furthermore, the update algorithm convergence condition strongly depends on the system noise level, particularly the close-in phase noise of the ADC/DAC clocks and local oscillator clocks of the RF up/down converters. Therefore, the LUT values may not be optimal values to maximize the intermodulation cancellation.

2.2.3 Indirect Learning Adaptive Digital Predistortion Algorithm

The indirect learning concept evolves from a multilayer neural network controller [15]. Figure 2.11 shows the typical indirect learning structure modified for the predistorter identification application. Basically, there are two mathematically identical predistorters, the transmit predistorter and training predistorter, excited by different input signals. The feedback path labeled “training predistorter” is scaled by the reciprocal of the gain of the power amplifier. The actual transmit predistorter is an exact copy of the feedback path and its output feeds into the power amplifier. Ideally, the algorithm will converge when the error energy is minimized, i.e. the power amplifier is linearized.

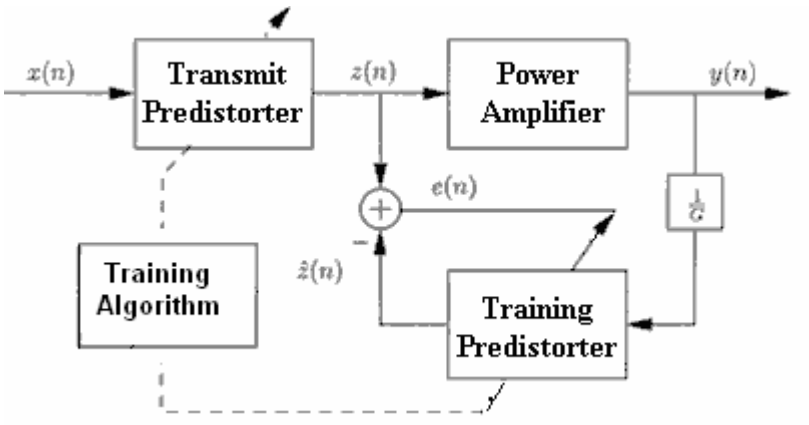

Figure 2.11: Indirect Learning Architecture of Digital Predistorter

The convergence of the algorithm is based on the assumption that the PA nonlinearity is invertible and its characteristics do not change rapidly over time. In most case, such changes in power amplifier characteristics are due to temperature drift, aging, etc., which have long time constants. Thus, it can be automatically adapted if the updating rate is fast enough compared to the drifting time constant. For the indirect learning predistorter architecture, the training branch can process the data offline after gathering a block of data samples, which lowers the processing requirements of the predistortion system. Once the predistorter identification algorithm has converged, the new set of parameters are plugged into the transmit predistorter, which can be implemented using a commercially available DSP, application specific integrated circuits (ASICs) or field programmable gate arrays (FPGAs).If the power amplifier characteristics are fairly stable over time, once the predistorter coefficients have been found , the setup in Figure 2.11 can even be run in open loop mode. In other words, the feedback path can be temporarily shut down to save energy dissipated in the training

branch until changes in the power amplifier characteristics require a predistorter coefficient update.

The algorithm for this work will be based on a digital pre-distorter implementation. An adaptive algorithm based on the indirect learning technique is used for the digital pre-distorter. This implementation thus has a training pre-distorter and a transmit pre-distorter. The details of the digital pre-distorter and the algorithm will be given in the next chapter.

CHAPTER 3

ALGORITHM

A digital predistorter based linear transmitter hardware and firmware implementation is developed using commercially available components. The predistortion algorithm has been refined and the corresponding firmware also been validated. The principle and design details of the digital predistortion hardware platform for commercial applications up to Mbps transmit rate using cost effective commercial components will also be discussed. The main aim of the design is to capture the minimum system level requirements and architecture of the design which is capable of linearizing power transmissions up to Mbps total transmit rate in a cost effective fashion. The design reuses an existing PA-1 linearization hardware platform.

The memoryless part of the digital predistorter is based on rational Chebyshev polynomial (the so-called Chebyshev-Padé representation) and a QR-decomposition recursive least square (QRD-RLS) update algorithm. Such a methodology potentially improves the system robustness and adaptation speed and can be used for a wide range of modulation schemes. It can be applied for TDMA communication which is a challenge for power amplifier linearization due to the bursty nature of the system over time. For the conventional polynomial approach, when the order of the polynomial is larger, the regression matrix in the least squares coefficient estimation is ill-conditioned and causes numerical instability. However, an orthogonal polynomial can be used to improve the numerical stability, and this is the greatest advantage of using an orthogonal polynomial for such applications. Furthermore, all orthogonal polynomial sequences have a number of elegant and fascinating properties. The recurrence relation

of the first kind of Chebyshev polynomial has been found to be very attractive and convenient for digital implementation to meet the power amplifier linearization requirement. Therefore, an introduction to the Chebyshev polynomial and Chebyshev-Padé representation will be helpful to understand the entire predistorter design for the linear transmitter.

3.1 Chebyshev Polynomial

Chebyshev polynomials are one type of orthogonal polynomials which are especially easy to generate using Gram-Schmidt orthonormalization. Although the orthogonal polynomial cannot carry more information than the same order conventional polynomial, orthogonal polynomials have very useful properties in the solution of mathematical and physical problems. Just as Fourier series provide a convenient method of expanding a periodic function in a series of linearly independent terms, orthogonal polynomials provide a natural way to solve, expand, and interpret solutions to many types of important differential equations need to be solved in practical engineering applications.

There are two different types of Chebyshev polynomial : the Chebyshev polynomial of the first kind and the Chebyshev polynomial of the second kind respectively.The Chebyshev polynomial used in our design evolved from the conventional first kind of Chebyshev polynomial [16]. A degree n from modified Chebyshev polynomial is denoted as $T_n(x)$, and is defined through the following explicit formula

$$\mathbf{T_n(x) = \cos(n \cdot \arccos(x))}$$

They also satisfy the recurrence relations:

$$T_0(x) = 1$$

$$T_1(x) = x$$

$$T_2(x) = 2x \cdot T_1(x) - T_0(x)$$

$$T_3(x) = 2x \cdot T_2(x) - T_1(x)$$

$$\cdots\cdots$$

$$T_{n+1}(x) = 2x \cdot T_n(x) - T_{n-1}(x)$$

The above polynomials are orthogonal in the interval [-1, 1]. As can be seen from Figure 3.1, the polynomial $T_n(x)$ has n zeros and n+1 extrema (maxima and minima) where all of the maxima have value "1" and the minima value "-1'. This property makes Chebyshev polynomial attractive in polynomials approximation and digital scaling and implementation. Chebyshev polynomials are not necessarily more accurate than some other approximating polynomials of the same order N, but they can be truncated to a polynomial of lower degree in a very graceful way that does yield the "most accurate" approximation of degree N-1.

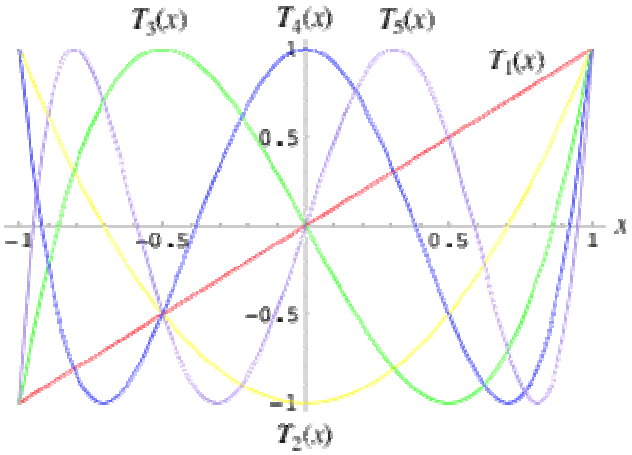

Figure 3.1 : Chebyshev Polynomials T_1 through T_6

3.2 Chebyshev Padé Approximation

A Padé rational approximation to f(x) on [a, b] is the quotient of two polynomials $P_n(x)$ and $Q_m(x)$ of degrees n and m, respectively. We use the notation $R_{n,m}(x)$ to denote this quotient:

$$R_{n,m}(x) = P_n(x) / Q_m(x)$$

The method is attributed to the French mathematician Henri Eugène Padé (1863-1953), and requires that f(x) and its derivatives be continuous at x = 0. The Padé approximation is able to achieve substantially higher accuracy than the optimal polynomial approximation with the same number of coefficients. Moreover, it can follow curves that are not essentially polynomial such as tan(x), a Heaviside (step) function and the practical complex gain characteristic of the Doherty power amplifier. These might not even have a suitable uniform polynomial approximation at all.

One disadvantage of the Padé approximation is the stability issue due to the poles of the denominator. Another drawback of the Padé approximation is the fact that finding the Padé approximation is not as straightforward as finding a polynomial approximation, but this can be done elegantly via a Chebyshev polynomials transformation [16]. Therefore, the rational Chebyshev polynomial is selected to approximate the memoryless part of the digital predistorter for our project , and stability is also well controlled by specially attention to the tuning algorithm. Obviously, the predistorter is very flexible and can be configured as a general Chebyshev polynomial by setting the constant term of the denominator coefficient to unit ‘1’ and rest of them

to zero. The order or the degree of the rational approximation can be easily programmable as well if the higher order term coefficients are set to zeros.

3.3 Memoryless Digital Predistorter in Complex Domain

In the digital envelope domain, the complex input sample of the digital predistorter is represented as x(n) = I(n)+j*Q(n), where I(n) is in-phase part while the Q(n) is quadrature part, and then the complex output of the memoryless part of digital baseband predistorter used in our design is written as follows:

$$y(n) = x(n) \sum_{k=0}^{N} A_k T_k(n)(|x(n)|^{2k}) \qquad n = 0,1,2\Lambda$$

(3.3.1)

$$y(n) = \frac{\sum_{k=0}^{N} A_k \bullet T_k(n)(|x(n)|^{2k})}{1 + \sum_{m=1}^{L} B_m \bullet T_m(n)(|x(n)|^{2m})} x(n) \qquad n = 0,1,2\Lambda$$

(3.3.2)

Equations (3.3.1) and (3.3.2) are called the Chebyshev representation and Chebyshev Padé representation respectively. The Chebyshev polynomials $T_K(n)$ used in the above equations are modified versions of Chebyshev polynomial discussed in section 3.1 by substituting the original variable x with $|x(n)|^2$ and shifting the interval

over which the polynomials are orthogonal from [-1, 1] to [Umin, Umax]. The entire set of modified Chebyshev polynomials is listed as:

$$
\begin{aligned}
&T_0(|x(n)|^2) = 1 \\
&T_1(|x(n)|^2) = \frac{2}{U_{\max} - U_{\min}} \bullet |x(n)|^2 - \frac{U_{\max} + U_{\min}}{U_{\max} - U_{\min}} \\
&\quad \text{M} \\
&T_{k+1}(|x(n)|^2) = 2 \bullet T_1(|x(n)|^2) \bullet T_k(|x(n)|^2) - T_{k-1}(|x(n)|^2) \qquad k \geq 1
\end{aligned}
\tag{3}
$$

The A_K and B_K in (3.3.1) and (3.3.2) are the complex coefficients that need to be adaptively identified to keep the power amplifier linearized over time. From the algorithm implementation point of view, only (3.3.2) needs to be implemented, and (3.3.1) can be treated as special case of (3.3.2). Although the Chebyshev Padé representation based memory predistorter needs a denominator part and is more expensive for hardware implementation and costs more resource to maintain the update, it is more powerful and can linearize more sophisticated power amplifier more efficiently than a Chebyshev representation. The Chebyshev representation based digital predistorter, however, can be used to linearize Class-A/AB power amplifiers when adaptation speed is the critical requirement. Practically, (3.3.2) always gives us no worse cancellation than that given by (3.3.1).

3.4 Coefficients Sensitivity Analysis of Digital Predistorter

The above digital pre-distorter can achieve up to 70dBc adjacent channel power ratio (ACPR) for QAM64 and SAM 150K modulation waveform. To have good

distortion cancellation, the following sensitivity analysis shows that the phase and magnitude error must be within a strict level.

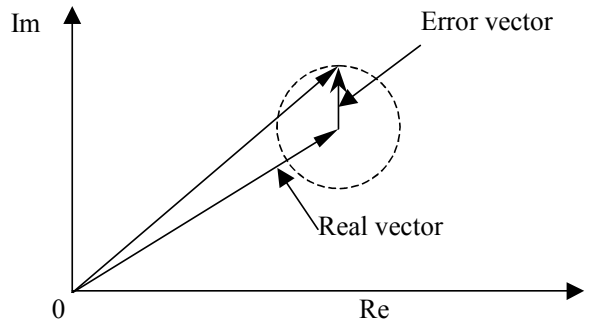

Figure 3.2 : The Error Vector Applied to the Complex Coefficients

To analyze the sensitivity to errors in the pre-distorter coefficients, an error vector with fixed magnitude (e.g 0.1dB relative to the real vector) and random phase uniformly changed from 0 to 360^{o} is applied to every coefficient of the predistorter model as shown in Figure 13, then ,a statistical simulation is performed 500 times with different random seed. The simulation results show that a 0.1dB magnitude error in all of the predistorter coefficients can cause a maximum ±0.80dB magnitude error and ±2.5 degree phase error in the complex gain of the digital predistorter. By referring to the distortion cancellation plot shown in Figure2.9(b), the maximum intermodulation cancellation will be roughly limited to 17dB.Ofcourse it is believed that more error in the coefficient will cause even less intermodulation cancellation.

3.5 Adaptive Algorithm

Several adaptive algorithms have been used for different types of application. The underlying metric of the adaptive algorithm is the least mean square based criteria.

Although a least square based nonlinear optimization can offer more flexibility and better cancellation performance, only QR-decomposition based recursive least square (QRD-RLS) will be addressed here due to the limitations of the hardware implementation.

3.6 QR Decomposition

Mathematically, any matrix A can be written as

$$\mathbf{A = QR} \qquad \textbf{(3.6.2)}$$

where R is an upper triangular matrix and Q is an orthogonal matrix. An orthogonal tensor Q satisfies the necessary and sufficient conditions of $Q^TQ = I$, and determinant of Q = 1. Equation (5.1) is called the QR decomposition.

For a square matrix A, the simultaneous equations A x = b can be solved by the QR decomposition as

$$\mathbf{A\ x = (QR)\ x = b} \qquad \textbf{(3.6.3)}$$

Then, with

$$\mathbf{y = Q^T\ b} \qquad \textbf{(3.6.4)}$$

Solve the triangular system of equations

$$\mathbf{R\ x = y} \qquad \textbf{(3.6.5)}$$

The QR decomposition for a square matrix, if carried out by Householder transformation, is two times more expensive than the LU decomposition(a matrix decomposition which writes a matrix as the product of a lower and upper triangular

matrix). The QR decomposition is always stable while the LU decomposition is stable only with complete pivoting [16].

For a rectangular matrix A of size m x n ($m \geq n$) with full rank, the QR decomposition produces

$$Q = \begin{bmatrix} Q_1 & Q_2 \end{bmatrix} \quad and \quad r = \begin{bmatrix} R_1 \\ 0 \end{bmatrix} \qquad \textbf{(3.6.6)}$$

Q is an m x m matrix and R is a m x n matrix, where the n columns of Q_1 form the orthonormal basis of the range space of A, and the m-n columns of Q_2 form the orthonormal basis of the null space of A^T. R_1 is an n x n matrix, and the lower part of the R matrix is a zero matrix of size (m-n) x n.

In the over determined full rank least squares problem, the residual of a rectangular matrix A with right-hand side vector, b, and the solution, x, is written as

$$\mathbf{r = A\,x - b} \qquad \textbf{(3.6.7)}$$

The least mean square for variable x can be obtained as [19-20]:

$$x=\left[A^T A\right]^{-1} A^T b = A^{-g} b\,, \qquad \textbf{(3.6.8)}$$

where $A^{-g} = \left[A^T A\right]^{-1} A^T$ is called the generalized inverse. On the other hand, the projection of vector b (of size m) into a lower dimensional range space of A (of size n, with m >n) gives the minimum length of the Euclidean norm of r.

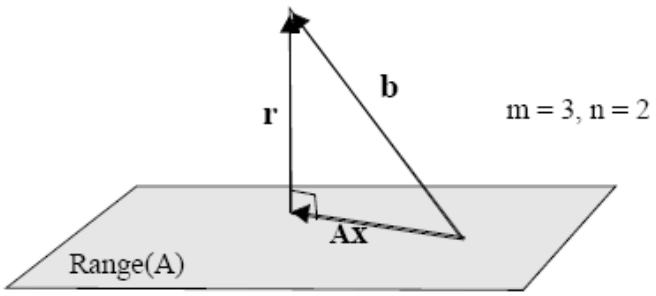

Figure 3.3: Projection into Range Space of A Gives the Minimum Length of "r"

Since r and the range of A are perpendicular to each other, every column of A is orthogonal to r ; therefore,

$$\mathbf{A^T r = 0 \text{ (orthogonal property)}} \qquad \mathbf{(3.6.9)}$$

Substituting equation (3.6.7) into (3.6.8), yields

$$\mathbf{A^T (A x - b) = A^T A x - A^T b = 0} \qquad \mathbf{(3.6.10)}$$

This is called the normal equation obtained from the range space projection [19].

One way to tackle the least squares problem is to first obtain A^TA and A^Tb, and then solve the system of equations. Since A^TA is symmetrical, the Cholesky decomposition can be used to solve the problem efficiently. However, the process to get A^TA is sometimes problematic. Round-off errors accumulated in the multiplication of the two matrices, A^TA, may corrupt the information in the original A matrix." A robust way to remedy this is to use the QR decomposition for the least squares solution of A. Consider the square of the residual norm as

$$\|r\|_2^2 = \|Ax\text{-}b\|_2^2 \qquad \mathbf{(3.6.11)}$$

An orthogonal transformation of (3.6.10) with Q^T should not change the length of the residual, thus,

$$\|r\|_2^2 = \|Ax-b\|_2^2 = \|Q^T AX-Q^T b\|_2^2 \quad \textbf{(3.6.12)}$$

where

$$Q^T A=R=\begin{bmatrix} R_1 \\ 0 \end{bmatrix} \quad \text{and} \quad Q^T b=b'=\begin{bmatrix} b_1' \\ b_2' \end{bmatrix} \quad \textbf{(3.6.13)}$$

The sub-matrix R_1 and sub-vector have sizes of n x n and n, respectively, and the null matrix and the sub-vector have sizes of (m-n) x n and (m-n), respectively. Therefore, (3.6.11) becomes

$$\|r\|_2^2 = \|Ax-b\|_2^2 = \|Q^T Ax-Q^T b\|_2^2 = \|R_1 x-b_1'\|_2^2 + \|b_2'\|_2^2 \quad \textbf{(3.6.14)}$$

In (3.6.13), the squares of residual norm is minimized with respect to x if we set

$$R_1 x-b_1'=0 \quad \textbf{(3.6.15)}$$

Therefore, after we have done the QR decomposition, A = QR, the least squares solution can be found by first obtaining b' = Q^Tb, then, solving (3.6.14) for x.

3.7 QR-Decomposition Based Recursive Least Square

The computational complexity has to be reduced considerably in order to increase the practical applicability of solving the above linear equations. Many algorithms have been reported over the last decade [21]. The QRD-RLS algorithms is numerically more robust than the standard LMS, RLS and Kalman Filter algorithm and is more suitable for power amplifier linearization application. The method is based upon orthogonal triangularization of the input data matrix using QR decomposition. Here, we

briefly describe the concept of the QR Decomposition-based Recursive Least Squares (QRD-RLS) method for the predistorter adaptation application.

The general case of the recursive least squares minimization problem is based on an adaptive linear combiner. Let $A(k) \in £^{M}$ be a vector of observations taken from M data signals at sample time n. Using a linear combination of the signals $A_m[k]$ (m=1,L M), a desired signal b[k] is to be estimated at the same time instant. Thereby, the goal is to minimize the sum of exponentially weighted squared errors,

$$\min_{x(k)} \sum_{i=0}^{k} \beta^{k-i} \left| b(i) - A^{T}(i) g x(i) \right|^{2} \qquad \textbf{(3.7.1)}$$

The so-called "forgetting factor" $0 \le \beta \le 1$ is commonly used to discount old data from the computations (exponential down dating), in order to provide a certain tracking capability when the system operates in a non-stationary environment. This is equivalent to determining the weight vector x[k] which minimizes the l_2-norm of the vector of error residuals e[k],

$$\|e(k)\| = \sqrt{e^{H}(k) e(k)} \qquad \textbf{(3.7.2)}$$

With the data matrix

$$X[k] @ \begin{pmatrix} x^{T}[1] \\ \\ x^{T}[k] \end{pmatrix} \in £^{k \times M} \qquad \textbf{(3.7.3)}$$

and the weighing matrix

$$W[k] @ \text{diag}(\sqrt{\beta}^{k-1}, \sqrt{\beta}^{k-2}, \text{L } 1) \in ¡^{k \times k} \qquad \textbf{(3.7.4)}$$

Since the Euclidean vector norm is invariant with respect to unitary (orthogonal) transformations Q[k], we apply the QRD to transform the weighted input data matrix W[k]X[k] into an upper triangular matrix $R(k) \in \pounds^{M\times M}$:

$$Q[k]e[k] = \begin{pmatrix} b_1[k] \\ b_2[k] \end{pmatrix} - \begin{pmatrix} R[k] \\ 0 \end{pmatrix} x[k] \qquad \textbf{(3.7.5)}$$

As can be seen from the above equation, the minimum norm condition for the error residual e[k] is obtained when

$$R[k]X[k]=b_1[k] \qquad \textbf{(3.7.6)}$$

This is the least squares solution for the adaptive linear combiner. Since the matrix R[k] is upper triangular, the weight vector x[k] can be derived very simply by a process of back-substitution.

The entire algorithm is summarized in Table 3.1, and the triangular system of equations can be updated on a sample by sample basis. The unitary update transformation $\hat{Q}[k]$ represents a sequence of M complex Givens rotations, consisting of a phase compensation term $G(j\varphi_m)$ time a real Givens rotation $G(j\theta_m)$, which operate on two rows of the matrix at a time and they are define as:

$$\hat{Q}[k] = \underbrace{\begin{pmatrix} \cos\theta_m & \sin\theta_m \\ -\sin\theta_m & \cos\theta_m \end{pmatrix}}_{G(\theta_m)} \underbrace{\begin{pmatrix} 1 & 0 \\ 0 & \exp(-j\varphi_m) \end{pmatrix}}_{G(\varphi_m)} \qquad \textbf{(3.7.8)}$$

where the rotation angle θ_m and φ_m are chosen to cancel the complex value

$$\theta_m = \arctan \frac{|A(m)|}{|R(m-1)|} \quad \textbf{(3.7.9)}$$

$$\varphi_m = \arctan \frac{\mathrm{Im}(R(m-1))}{\mathrm{Re}(R(m-1))} \quad \textbf{(3.7.10)}$$

The algorithm for complex givens rotation is summarized in the table given below.

Initialization

$R[0] = \sqrt{\delta}\, I_{M\times M} \in$ ¡ $^{M\times M}$ with $0 \le \delta \le 1$, U[0]=$0^{M\times 1}$

For k =1, 2, 3,…..n

$$\hat{Q}[k] \bullet \begin{pmatrix} A[k] & U[k] \\ \sqrt{\beta}R[k-1] & \sqrt{\beta}U[k-1] \end{pmatrix} = \begin{pmatrix} R[k] & U[k] \\ 0_{1\times M} & e[k] \end{pmatrix}$$

where $\hat{Q}(k) = \hat{Q}_M(k) \mathrm{L}\ \hat{Q}_1(k) \in$ £ $^{M\times M}$ with

$$\hat{Q}(k) = \begin{pmatrix} \cos\theta_m[k] & \mathrm{L} & \sin\theta_m[k]\exp(-j\varphi_m[k]) & \mathrm{L} \\ \mathrm{M} & I_{M-m} & \mathrm{M} & \mathrm{M} \\ -\sin\theta_m[k] & \mathrm{L} & \cos_m[k]\exp(-j\varphi_m[k] & \mathrm{L} \\ \mathrm{M} & \mathrm{M} & \mathrm{M} & \mathrm{I}_{m-1} \end{pmatrix}$$

Table 3.1 : QRD-RLS Algorithm Based on Complex Givens Rotation

In the real hardware, the complex Givens rotation will be implemented use co-ordinate rotation digital computer (CORDIC) algorithm..

3.8 CORDIC Algorithm

An efficient parallel triangular systolic processor array realization of the QR decomposition based RLS (QRD-RLS) algorithm using Givens rotations was introduced in [21]. The systolic array is controlled by a uniform cyclic clock and it executes plane rotations to annihilate certain elements of the input signal matrix. Commonly the computation of rotation angles requires either square roots and divisions or trigonometric functions, which is time-consuming and thus not applicable for hardware implementation. To solve the problem, the famous CORDIC (Coordinate Rotation Digital Computer) algorithm [23] has been introduced to perform the two-dimension vector rotation instead of the conventional Givens rotations. The main idea underlying this algorithm is to do phase shifting through a series of "micro rotations" using a fixed set of elementary rotation angles. Through a proper choice of the elementary angles all computations can be implemented efficiently in FPGA/ASIC using a sequence of shift and add/subtract operations. Generally, a look-up-table holding the elementary rotation angles is set up in advance to perform the phase shifting replacing the trigonometric functions exploited in the Givens rotations.

The basic idea underlying the CORDIC scheme is to carry out vector ("macro") rotations by an arbitrary rotation angle θ via a series of b+1 "micro-rotations" using a fixed set of predefined elementary angles α_j.

$$\theta = \sum_{j=0}^{b} \delta_j \alpha_j, \quad \delta_j \in \{-1, +1\} \tag{3.8.1}$$

This leads to a representation of the rotation angle θ in terms of the rotation coefficients δ_j. If the elementary angles are defined as

$$\delta_j = \arctan(2^{-j}),\ j \in \mathfrak{I} = \{0,1,2,\mathrm{L}\ b\} \qquad \textbf{(3.8.2)}$$

It follows that, an unscaled μ-rotation $G_\mu(\delta_j)$ can be performed via two shift-add operations, which are easily realized in hardware:

$$\begin{pmatrix} x_{j+1} \\ y_{j+1} \end{pmatrix} = \underbrace{\begin{pmatrix} 1 & \tan(\alpha_j) \\ -\tan(\alpha_j) & 1 \end{pmatrix}}_{G_\mu(\alpha_j)} \cdot \begin{pmatrix} x_j \\ y_j \end{pmatrix} = \begin{pmatrix} 1 & \delta_j 2^{-j} \\ -\delta_j 2^{-j} & 1 \end{pmatrix} \cdot \begin{pmatrix} x_j \\ y_j \end{pmatrix} \qquad \textbf{(3.8.3)}$$

The final result is obtained with a precision of b bits ($\overset{b}{=}$) after the execution of b+1 unscaled μ-rotations (CORDIC iterations) and a multiplication with the scaling factor $K = \prod_{j=0}^{b} \frac{1}{\sqrt{1+2^{-2j}}}$ (scaled rotation $G_s(\theta)$):

$$\begin{pmatrix} x_{out} \\ y_{out} \end{pmatrix} = K \cdot \overset{b}{\underset{j=0}{\mathrm{C}}} G_\mu(\alpha_j) \cdot \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} \overset{b}{=} G(\theta) \cdot \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} \qquad \textbf{(3.8.4)}$$

The multiplication with the constant factor K can also be decomposed into a sequence of simple shift-add operations which are often performed in a series of additional scaling iterations.

The CORDIC has two modes of operation called “vectoring”, to compute the magnitude and phase of a vector.

$$\begin{pmatrix} x_{out} \\ y_{out} \end{pmatrix} = \begin{pmatrix} \mathrm{sign}(x_{in}) \cdot \sqrt{x_{in}^2 + y_{in}^2} \\ 0 \end{pmatrix} \qquad \textbf{(3.8.5)}$$

$$\theta_{out} = -\arctan \frac{y_{in}}{x_{in}} \qquad \textbf{(3.8.6)}$$

where the vector $(x_{in}, y_{in})^T$ is rotated to the x-axis, with "rotation"

$$\begin{pmatrix} x_{out} \\ y_{out} \end{pmatrix} = \begin{pmatrix} \cos\theta_{in} & -\sin\theta_{in} \\ \sin\theta_{in} & \cos\theta_{in} \end{pmatrix} \cdot \begin{pmatrix} x_{in} \\ y_{in} \end{pmatrix} \quad \textbf{(3.8.7)}$$

$$\theta_{out} = \theta_{in} \quad \textbf{(3.8.8)}$$

When the vector $(x_{in}, y_{in})^T$ is rotated by the angle θ_{in}, the Givens rotation in (5.21) can be carried out using the CORDIC Algorithm in rotation mode, whereas the determination of the rotation angle according to (5.22) is accomplished using the CORDIC in vector mode. Table3-2 shows the details of the16 bit CORDIC algorithm.

Initialization:

$x_0 = x_{in}/2$, $y_0 = y_{in}/2$
add all-zero guard bit extension

for $j = 0 \ldots 18$ do:

$x_{j+1} = x_j - \sigma_j y_j 2^{-s_j} + \gamma_j x_j 2^{-s_j}$
$y_{j+1} = y_j + \sigma_j x_j 2^{-s_j} + \gamma_j y_j 2^{-s_j}$

where
$\sigma_j = -\text{sign}(x_j) \cdot \text{sign}(y_j) = \sigma_{out}$ in Vec. mode and
$\sigma_j = \sigma_{in}$ in Rot. mode
$\{s_j | j = 0, 1, \ldots, 18\} = \{0, 1, 2, 3, 4, 5, 6, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17\}$
$\{\gamma_j | j = 0, 1, \ldots, 18\} = \{0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1\}$

Correction cycle:

if $x_{19} < 0$ then $x_{20} = x_{19} + 1$ else $x_{20} = x_{19}$
if $y_{19} < 0$ then $y_{20} = y_{19} + 1$ else $y_{20} = y_{19}$
erase guard bits

$x_{in} = x_{20}$, $x_{out} = x_{20}$, $y_{out} = y_{20}$

Table 3.2: The flow of the16 bit CORDIC Algorithm

3.9 Apply the QRD-RLS to the Chebyshev Padé Based Predistorter

To adopt the QRD-RLS adaptive algorithm for power amplifier linearization, some necessary modification has to be done. First of all, the mathematically nonlinear predistorter equation has to be rewritten into a compact linear format. Secondly, some special dynamic scaling operation has to be done to guarantee the robustness and convergence over the different modulation schemes. The higher the chip rate and bigger the peak to average ratio, the better control required on the dynamic scaling. This technique along with the proposed orthogonal predistorter architecture will be implemented in hardware.

In order to get the coefficient A_k and B_m, the equation (3.9.1) can be rewritten as

$$x(n)\sum_{k=0}^{N} A_k T_k(n)(|x(n)|^{2k}) - y(n)\sum_{m=1}^{L} B_m T_m(n)(|x(n)|^{2m}) = y(n) \quad \textbf{(3.9.1)}$$

Moreover, it can be written into a compact matrix for as follows for n+1 sampling data input.

$$\begin{bmatrix} Num & Den \end{bmatrix} g \begin{bmatrix} A_1 \\ M \\ A_N \\ B_1 \\ M \\ B_L \end{bmatrix} = \begin{bmatrix} y(0) \\ y(1) \\ y(3) \\ y(4) \\ M \\ y(n) \end{bmatrix} \quad \textbf{(3.9.2)}$$

where

$$\text{Num} = \begin{bmatrix} x(0)T_0(x(0)) & x(0)|x(0)|^2 gT_1(x(0)) & L & x(0)|x(0)|^{2N} gT_N(x(0)) \\ x(1)T_0(x(1)) & x(1)|x(1)|^2 gT_1(x(1)) & L & x(1)|x(1)|^{2N} gT_N(x(1)) \\ M & M & O & M \\ x(n)T_0(x(n)) & x(n)|x(n)|^2 gT_1(x(n)) & L & x(n)|x(n)|^{2N} gT_N(x(n)) \end{bmatrix}$$

(3.9.3)

and

$$\text{Den} = \begin{bmatrix} -y(0)|x(0)|^2 gT_1(x(0)) & -y(0)|x(0)|^4 gT_2(x(0)) & L & -y(0)|x(0)|^{2N} gT_L(x(0)) \\ -y(1)|x(1)|^2 gT_1(x(1)) & -y(1)|x(1)|^4 gT_2(x(1)) & L & -y(1)|x(1)|^{2N} gT_L(x(1)) \\ M & M & O & M \\ -y(n)|x(n)|^2 gT_1(x(n)) & -y(n)|x(n)|^4 gT_2(x(n)) & L & -y(n)|x(n)|^{2N} gT_L(x(n)) \end{bmatrix}$$

(3.9.4)

Equation (3.9.2) is a linear combiner format; therefore, the corresponding coefficients can be obtained via the QRD-RLS adaptive algorithm described earlier. A special case when the denominator coefficients B_m are forced to zeros, equation (3.9.2) can be simplified with less unknowns. In this case, the QRD-RLS algorithm could use less hardware resources. A user changeable generic variable will be necessary in the HDL code to solve this problem. If a digital filter based memory part is added into the predistorter, a similar process has to be done in order to use QRD-RLS algorithm to find the corresponding coefficients.

CHAPTER 4

ARCHITECTURE

4.1 Digital Pre-Distorter

The digital predistorter hardware and firmware is developed using commercial available components. The predistortion algorithm has been refined and the corresponding firmware also validated .The principle and design details of the digital predistortion hardware platform for the commercial application up to Mbps transmit rate using cost effective commercial components will also be discussed. The main aim of the design is to capture the minimum system level requirements and architecture of the design which is capable to linearize power transmitter up to Mbps total transmit rate in a cost effective fashion. The design reuses the existing PA-1 linearization hardware platform. The proposed architecture for the receiver and transmitter chain are given below.

4.2 IF Section

The IF at 70Mhz passes through the ADC .The IF is under sampled at the ADC which is clocked at 100Mhz.The output of the ADC passes through a digital Quad Demodulator which converts the IF into I and Q signals. These I-Q samples are then passed through a Low Pass Filter and a Decimator. The output I-Q samples of the decimator will be at 22.5Mhz.These will then pass through a channel compensation block and then into the Rx processing block at 22.5Mhz.The pre-distorter update algorithm obtains the I-Q samples from the modulation source as well as from the Rx

chain. This block calculates the new set of coefficients and updates the predistorter block in the Tx chain.

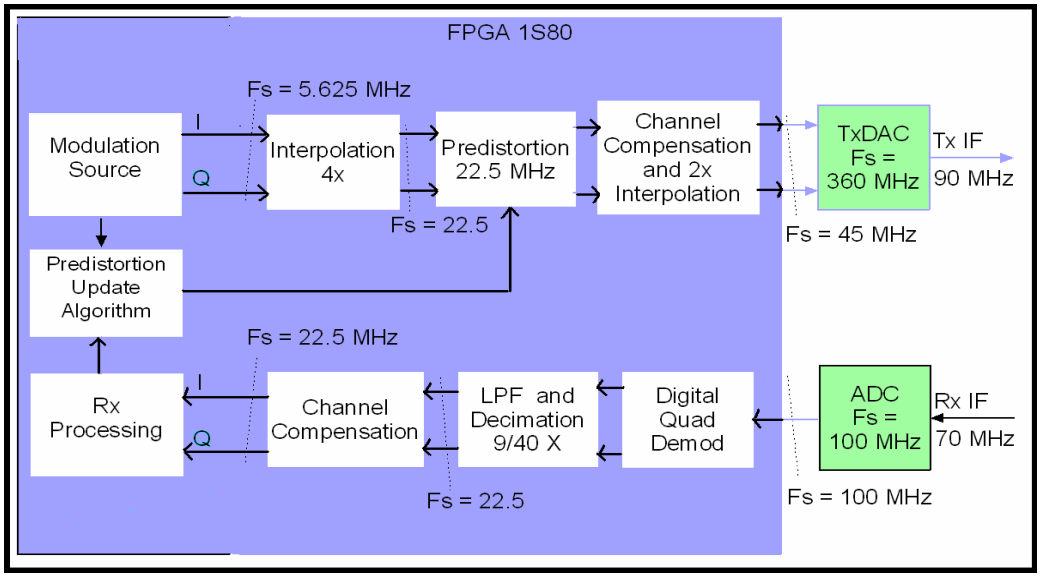

Figure 4.1 : Signal Processing Block Diagram

The I-Q samples produced in the modulator source are at 5.625 MHz. Hence this passes through an interpolator which up samples them to 22.5Mhz.This is then passed through a channel compensator and an interpolator to get the final I-Q samples at 45Mhz.This is then passed to the transmitter DAC which generates IF at 90 Mhz.As shown in the diagram, the Tx and Rx signal processing algorithms are implemented in the FPGA .The Rx ADC and Tx DAC are interfaced to the FPGA.

4.3 RF Section

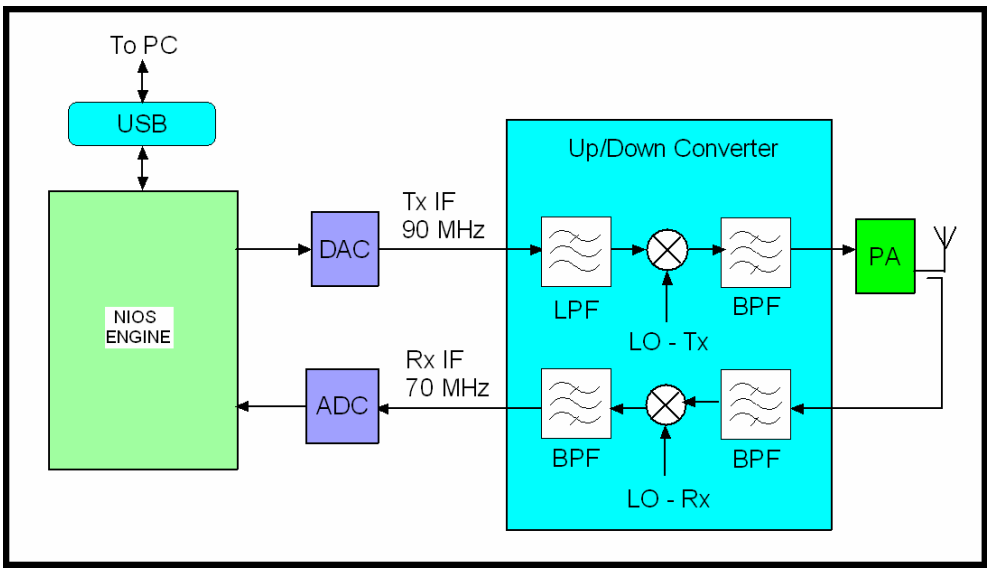

Figure 4.2: RF Section Block Diagram

The IF from the Tx at 90 MHz is first passed through a low pass filter and then upconverted to the required frequency and passed through a band pass filter into the power amplifier. The PA amplifies the RF signal and then transmits it out through the antenna.The output of the power amplifier is fed back into the receiver. This is done to provide the output samples to the predistorter using the adaptive algorithm which requires both the input and output samples of the PA. In the Rx chain, the fed back RF is down converted to IF at 70 MHz and passed through a band pass filter.

4.4 Key Hardware Specifications

Hardware linearity, signal-to-noise ratio and spurious-free dynamic range are critical to the digital predistortion-based power amplifier linearization techniques.

Hence, the key components have to be selected properly. Due to the higher peak-to-average ratio of realistic OFDM and SAM/QAM signals, the dynamic range plays a major role while evaluating the selected components.

In order to the meet most commercial application requirements for the system dynamics, the digital control attenuator combined with a low distortion amplifier is adopted in the current design. Since the high speed ADCs and DACs are operated in differential mode to gain higher speed, lower noise and higher dynamic performance, all the mixers and the LNA will employ a differential operation mode to best utilize such an advantage. Some key hardware specifications for both the transmitter and receiver chain have been listed below.

4.4.1 Rx ADC's

The Rx ADC is the dominant part on receive side. For the total bandwidth requirement, some ADCs from Linear Technology have reasonably good performance and pin compatible features within the same family. They also provide optional internal dither and a data output randomizer. This ADC family can support input undersampling IF up to 500MHz.Model LTC2204 is currently chosen for this design. It has the following characteristics.

- Sample Rate: 65Msps/40Msps
- 79dB SNR and 100dB SFDR (2.25V Range)
- SFDR >83dB at 170MHz (1.5VP-P Input Range)
- PGA Front End (2.25VP-P or 1.5VP-P Input Range)
- 700MHz Full Power Bandwidth S/H

- Optional Internal Dither
- Optional Data Output Randomizer
- Single 3.3V Supply
- Power Dissipation: 530mW/470mW
- Optional Clock Duty Cycle Stabilizer
- Out-of-Range Indicator
- Pin Compatible Family: 105Msps: LTC2207 (16-Bit)
- 80Msps: LTC2206 (16-Bit)
- 65Msps: LTC2205 (16-Bit)
- 40Msps: LTC2204 (16-Bit)

Another alternative option from Linear Technology is the 14-bit LTC2246 or 14- bit LTC2296. LTC2246/LTC2296 can provide competitive performance at the frequency range of interest, but the LTC2296 is a dual 14bit ADC which would give some flexibility if we plan to do an RF in/RF out linear transmitter.

4.4.2 Tx DAC's

The test results on current power amplifier linearization systems indicate that the resolution of the transmitter digital-to-analog converter puts a limit on the dynamic range. The three DAC's that fit this design are the AD9779 from Analog, the MAX5895 from Maxim and the DAC5687 from TI semiconductor respectively. After some evaluation, the AD9779 from Analog Device has been selected for this design and has the following performance.

The AD9779 is a dual 16-bit high performance, high frequency DAC that provides a sample rate of 1 GSPS, permitting multi-carrier generation up to its Nyquist frequency. It is part of a pin-compatible family, complemented by the 14-bit AD9778 and 12-bit AD9776 that allows performance to be traded off for cost. All three products include features optimized for direct conversion transmit applications, including complex digital modulation and gain and offset compensation. The DAC outputs are optimized to interface seamlessly with analog quadrature modulators such as the AD8349. A serial peripheral interface (SPI) provides for programming many internal parameters and also enables read-back of status registers. The output current can be programmed over a range of 10mA to 30mA. The AD977X family is manufactured on an advanced 0.18 μm CMOS process and operates from 1.8 V and 3.3 V supplies for a total power consumption of less than 1 W. It is supplied in a 100-lead QFP package. The other features are as follows.

- Ultra-low Noise and Intermodulation Distortion (IMD) enable high quality synthesis of wideband signals from baseband to high intermediate frequencies.
- Single-ended CMOS interface supports a maximum input rate of 300 MSPS with 1x interpolation.
- Manufactured on a CMOS process, the AD9779 uses a proprietary switching technique that enhances dynamic performance.
- The current outputs of the AD9779 can be easily configured for various single-ended or differential circuit topologies.

4.4.3 Tx/Rx IF Amplifier

The MAX2055 high-performance, digitally controlled, variable-gain, differential analog-to-digital converter (ADC) driver/amplifier (DVGA) is designed for use from 30MHz to 300MHz in base station receivers. The device integrates a digitally controlled attenuator and a high-linearity single-ended-to-differential output amplifier, which can either eliminate an external transformer, or can improve the even-order distortion performance of a transformer-coupled circuit, thus relaxing the requirements of the anti-alias filter preceding an ADC. Targeted for ADC driver applications to adjust gain either dynamically or as a one-time channel gain setting, the MAX2055 is ideal for applications requiring high performance. The attenuator provides 23dB of attenuation range with ±0.2dB accuracy. The MAX2055 is available in a thermally enhanced 20-pin TSSOP-EP package and operates over the -40°C to +85°C temperature range. The typical feature of MAX2055 is as follows:

- 30MHz to 300MHz Frequency Range
- Single-Ended-to-Differential Conversion
- -3dB to +20dB Variable Gain
- 40dBm Output IP3 (at All Gain States and 70MHz)
- 2nd Harmonic -76dBc
- 3rd Harmonic -69dBc
- Noise Figure: 5.8dB at Maximum Gain
- Digitally Controlled Gain with 1dB Resolution and ±0.2dB Accuracy

- Adjustable Bias Current

Alternatively, the LT5514 is a programmable gain amplifier (PGA) with bandwidth extending from low frequency (LF) to 850MHz. It consists of a digitally controlled variable attenuator, followed by a high linearity amplifier. The amplifier is configured with two identical transconductance amplifiers, hard wired in parallel with individual dedicated enable pins. When both amplifiers are enabled (Standard mode), the LT5514 offers an OIP3 of +47dBm (at 100MHz). Power dissipation can be reduced when a single amplifier is enabled (Low Power mode). Four parallel digital inputs control the gain over a 22.5dB range with 1.5dB step resolution. An on-chip power supply regulator/filter helps isolate the amplifier signal path from external noise sources. The LT5514's open-loop architecture offers stable operation for any practical load conditions, including peaking free AC response when driving capacitive loads, and excellent reverse isolation. The LT5514 may be operated broadband, where the output differential RC time constant sets the bandwidth, or it may be used as a narrowband driver with the appropriate output filter. Output IP3 at 100MHz: 47dBm. Here is the summary of the device features:

- Maximum Output Power: 21dBm
- Bandwidth: LF to 850MHz
- Propagation Delay: 0.8ns
- Maximum Gain: 33dB
- Noise Figure: 7.3dB (Max Gain)
- Gain Control Range: 22.5dB

- Gain Control Step: 1.5dB
- Gain Control Settling Time: 500ns
- Output Noise Floor: –134dBm/Hz (Max Gain)
- Reverse Isolation: –80dB
- Single Supply: 4.75V to 5.25V
- Low Power Mode
- Shutdown Mode
- Enable/Disable Time: 1μs
- Differential I/O Interface

Comparing the MAX2055 from Maxim and the LT5514 from Linear Technology, the MAX2055 has a smaller attenuation step size and lower noise figure, but less IP3(3^{rd} order intercept).The MAX2055 is used in this design.

4.4.4 Tx/Rx Mixer

The HMJ1 from WJ Communication is the commonly used Tx/Rx mixer for existing power amplifier linearization systems. The HMJ1 is a high dynamic range, GaAs FET mixer. This active FET realizes a typical third order intercept point of +39 dBm at an LO drive level of +17 dBm and a DC bias of 3.0V. The HMJ1 comes in a low cost, J-lead package. Typical applications include frequency up/down conversion, modulation and demodulation for receivers and transmitters used in cellular communications systems. With the higher LO requirement, a critical IF band pass filter

is necessary to reject the LO leaking. In this version of the hardware we try to find some alternative mixer to replace the existing mixers.

For the Tx side, the LT5521 is a very high linearity mixer optimized for low distortion and low LO leakage applications from linear technology. The chip includes a high speed LO buffer with single-ended input and a double-balanced active mixer. The LT5521 requires only –5dBm LO input power to achieve excellent distortion and noise performance while reducing external drive circuit requirements. The LO buffer is internally 50W matched for wideband operation. According to the data sheet, with a 250MHz input, a 1.7GHz LO and a 1.95GHz output frequency, the mixer has a typical IP3 of +24.2dBm, –0.5dB conversion gain and a 12.5dB noise figure. The LT5521 offers exceptional LO-RF isolation, greatly reducing the need for output filtering to meet LO suppression requirements. The device is designed to work over a supply voltage range from 3.15V to 5.25V. The highlighted the feature of this device is as follows:

- Wideband Output Frequency Range up to 3.7GHz
- +24.2dBm IIP3 at 1.95GHz RF Output
- Low LO Leakage: –42dBm
- Integrated LO Buffer: Low LO Drive Level
- Single-Ended LO Drive
- Wide Single Supply Range: 3.15V to 5.25V
- Double-Balanced Active Mixer

- Shutdown Function
- 16-Lead (4mm x 4mm) QFN Package

For the Rx side, the LT5527 active mixer is optimized for high linearity, wide dynamic range downconverter applications. The IC includes a high speed differential LO buffer amplifier driving a double-balanced mixer. Broadband, integrated transformers on the RF and LO inputs provide single ended 50Ω interfaces. The differential IF output allows convenient interfacing to differential IF filters and amplifiers, or is easily matched to drive 50Ω single-ended, with or without an external transformer. The RF input is internally matched to 50Ω from 1.7GHz to 3GHz, and the LO input is internally matched to 50Ω from 1.2GHz to 5GHz. The frequency range of both ports is easily extended with simple external matching. The IF output is partially matched and usable for IF frequencies up to 600MHz. The LT5527's high level of integration minimizes the total solution cost, board space and system-level variation. The typical highlighted features are as follows:

- 50Ω Single-Ended RF and LO Ports
- Wide RF Frequency Range: 400MHz to 3.7GHz*
- High Input IP3: 24.5dBm at 900MHz, 23.5dBm at 1900MHz
- Conversion Gain: 3.2dB at 900MHz, 2.3dB at 1900MHz
- Integrated LO Buffer: Low LO Drive Level
- High LO-RF and LO-IF Isolation
- Low Noise Figure: 11.6dB at 900MHz, 12.5dB at 1900MHz

- Very Few External Components
- Enable Function
- 4.5V to 5.25V Supply Voltage Range
- 16-Lead (4mm × 4mm) QFN Package

All these components are shown in the figure below. All these components are placed on the same board as that of the FPGA and the corresponding connections made.

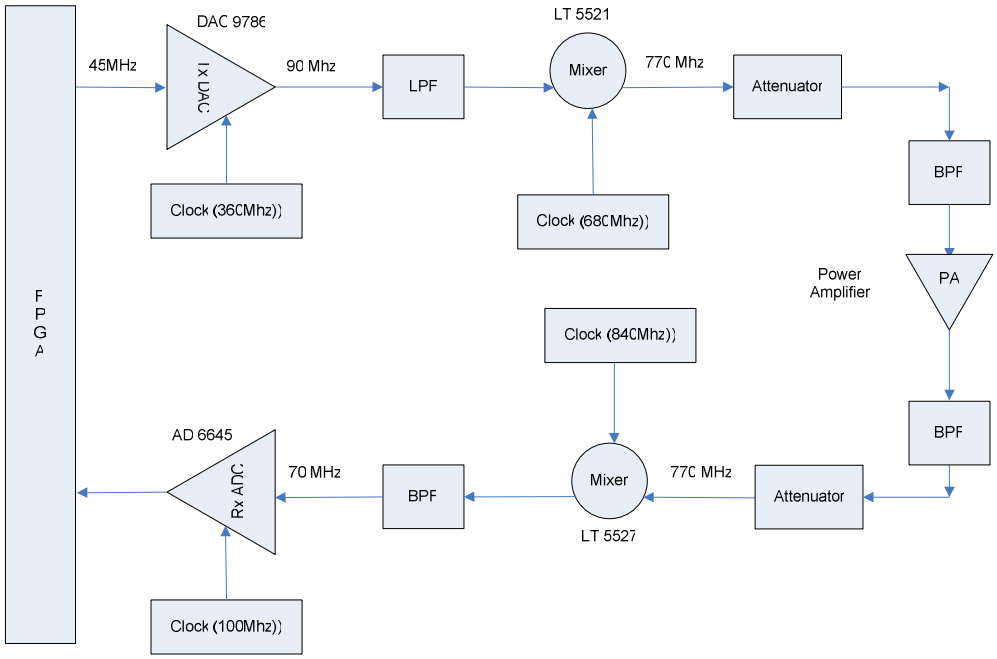

Figure 4.3 : Block Diagram showing the components

The output from the FPGA will have I/Q samples at 45 MHz as shown in Figure 4.3. These samples are then fed to the TX DAC 9786, which is clocked at 360Mhz.The output of the DAC would be an IF at 90Mhz.This is then passed through a low pass

filter and then fed to the Tx mixer operating at 680MHz.The output of the mixer is passed through a band pass filter to yield the 770Mhz signal for transmission. This is then fed to the Tx attenuator, which can be tuned to a particular value specified by the user. The attenuator is important as it is required to protect the PA .The output of the attenuator is then fed to the power amplifier.

The feedback path includes the output of the PA at 770MHz.It is then fed to the Rx attenuator. The attenuator can be set to a particular value by the user. This is important as excess power might damage the other components in the feedback path. The output of the attenuator is then fed to the Rx mixer, operating at 840Mhz.This gives an IF of 70 MHz, which is fed to the input of the Rx ADC AD6645, which is clocked at 100Mhz.The output of the ADC goes to the FPGA input.

CHAPTER 5

IMPLEMENTATION DETAILS

5.1 Hardware Details

5.1.1 Interpolation 2x Filter

To meet the 45MHz input data rate of the up-converter, the pre-distorted signal must be interpolated by two times. Interpolation can generate extra points in between the original samples. When a signal is interpolated, zeros are inserted between data points and the data is filtered to remove spectral components that were not present in the original signal.

The architecture of an interpolation 2x filter is shown below. It's a polyphase finite impulse response (FIR) filter with two separate sets of coefficients (C_0, C_2, C_4, …) and (C_1, C_3, C_5, …).

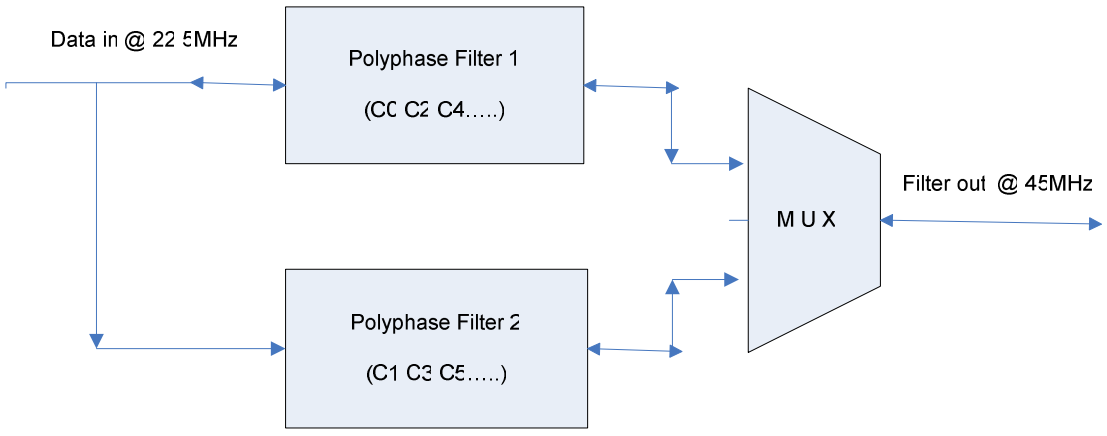

Figure 5.1: Interpolation 2x FIR (69 tap)

These polyphase filters are implemented by an FIR compiler provided by Altera Corp. The FIR compiler can provide multiple implementation options, given the filter coefficients. The table below lists the different implementation options.

	Throughput	**Speed Required**	**Resource Required**
Fully Parallel	1/1 cycle	22.5 MHz	5,300 LEs with 136 M512Ks
Fully Serial	1/16 cycles	360 MHz	700 LEs with 20 M512Ks
4 Multi-Bit Serial	1/4 cycles	90 MHz	2,200 LEs with 80 M512Ks

Table 5.1: Poly phase 2 FIR with 39 16-bit Coefficients, 16-bit input & output

As observed in the table, the 4 Multi-Bit Serial Units implementation has been chosen for it‘s good trade-off between the speed and resource requirement. A clock rate of 90MHz can be easily achieved and 1,100 Logic Elements (LE) only take less than 2% of logic resources of the whole FPGA.

Therefore, the interpolation 2x filter will run at 90Mhz, and the total resource is about 2,200*2 = 4,400 LEs and 80*2 = 160 M512K memory blocks (I and Q channel runs simultaneously).

5.1.2 Demodulation Equation

During the up converter chain, the baseband complex signal X(t) = I+jQ will be modulated by transfer function e^{jwt} = cos(w_1t)+jsin(w_1t), and the lower band will be rejected, so the output modulated signal becomes Y(t) = I*cos(w_1t)+jQ*sin(w_1t). Here, w_1 is 360Mhz, the Intermediate Frequency (IF) of up-converter.

After the down converter chain, the feedback signal R(t) sampled at 100MHz has to be demodulated to I and Q signal: I= R(t)*cos(w_2t); Q= R(t)*sin(w_2t) . Here, w_2 is 70MHz, the IF frequency of down-converter. Both the I and Q signal have to be passed through a low-pass filter and data-rate transferring filter to get the 22.5M samples/s sampled data.

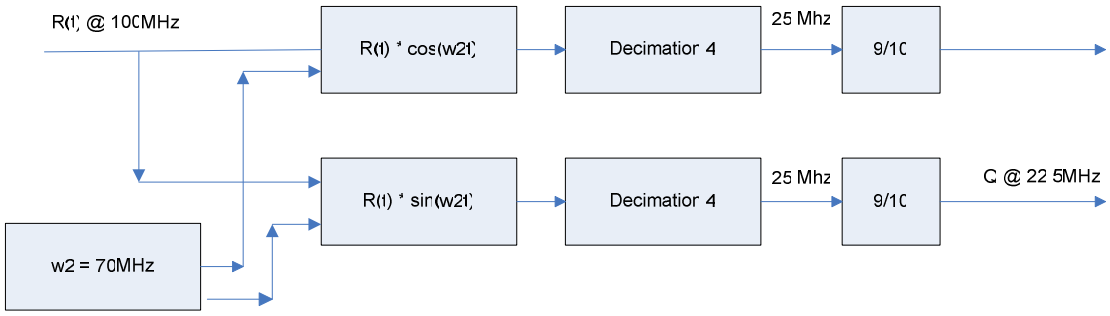

Figure 5.2: Demodulation Structure

5.1.3. Decimation 4x Low Pass Filter

To obtain the I and Q signal at 22.5M samples/s sample rate, a low pass decimation 4x filter is applied which can filter the high frequency and get a output baseband signal sampled at 25MHz. In general, decimation removes redundant data

points. To decimate a signal, a low-pass filter is also required to remove spectral components that are not present at the low sample rate.

The architecture of the decimation 4x filter is described in Figure 6. This is similar to the interpolation 2x filter described before. It's also a polyphase FIR filter, but it has four separate sets of coefficients (C_0, C_2, C_4, …) … and (C_3, C_5, C_7, …).

The decimation 4 x filters can also be implemented by the FIR compiler. For a better tradeoff between speed and area, the 4 Multi-Bit Serial Units Option has been selected to implement the filter, which will run at 100MHz and take about 5,500 LEs. With two channels, the total resources required for the decimation 4x filters will be doubled.

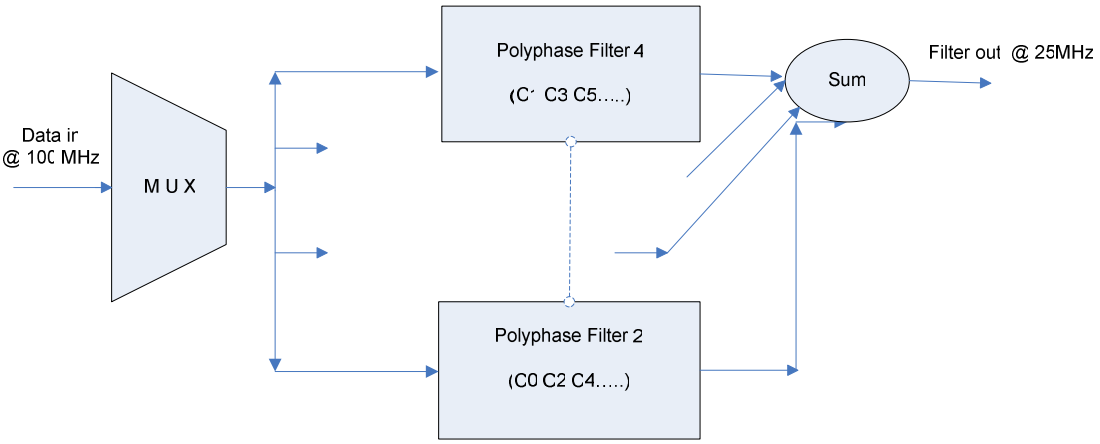

Figure 5.3: Decimation 4 times Filter (99 tap)

5.1.4. Implementation of 9/10x filter

To finally get 22.5M samples/s sample rate, the sampled data from the decimation 4x filter has to go through another 9/10x filter. The 9/10x filter can be treated as being interpolated 9 times first, then decimated by 10 times.

Considering there are a total of 99 taps, each poly phase filter will only have 11 coefficients. The table below compares the two different implementations of this 9x interpolation.

From the table, it can be seen that the coefficient-reload architecture is more efficient. In general, we only need one 11-tap filter, and every cycle, a new set of coefficients are reloaded. Since the output of the filter is still at 25M samples/s sample rate, a Dual Clock First In First Out (FIFO) buffer is used, which has a write clock of 25MHz and a read clock of 22.5MHz.A write enable signal is generated which will discard one write every 10 cycles .So finally we can get a 22.5M samples/s sampled data stream out of the FIFO since 25 * 9/10 = 22.5M.

	Throughput	**Speed Required**	**Poly-phase Filters Required**	**Resource Required**
Coefficient-Reload	1 cycle	25 MHz	1	4,600 LEs 6 M4Ks
4 Multi-Bit Serial	4 cycles	100 MHz	9	5,200 LEs with 108 M4Ks

Table 5.2 : Polyphase 9 FIR with 99 16-bit Coefficients, 16-bit input & output

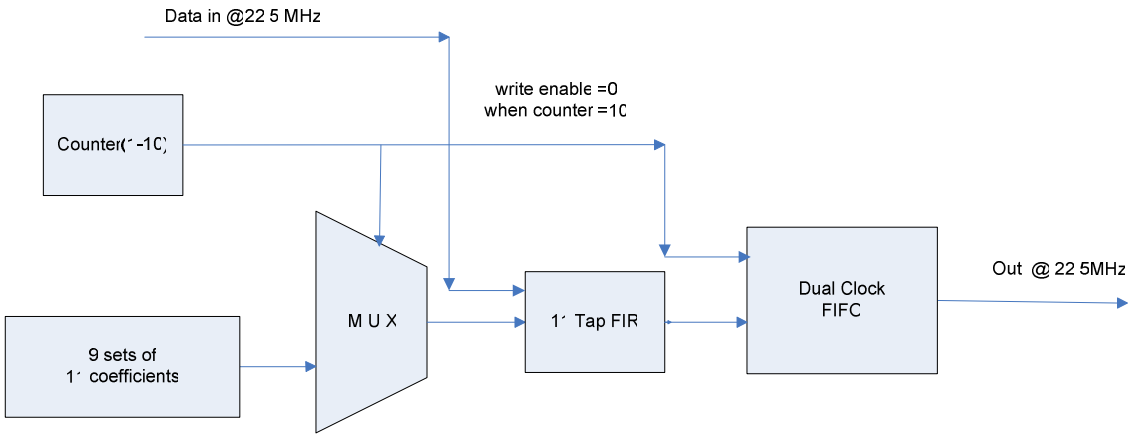

Figure 5.4 : 9/10x filter

The 9/10x filter is also implemented by the FIR compiler. In the current system, the I and Q channel both run 9/10x filtering at 25MHz. So the total logic resource usage will be about 9,200 LEs.

5.1.5 Firmware for theNios Interface

This is the firmware developed for the address decoding of both read/write operations in Nios 2 software. The Nios 2 software would issue a read/ write command specifying the corresponding address. This address is passed through the Avalon read/write interface and is decoded in the firmware. The firmware uses 12 bit address and 32 bit data. The firmware mainly deals with 2 kinds of operations :

1) **Write Operation**

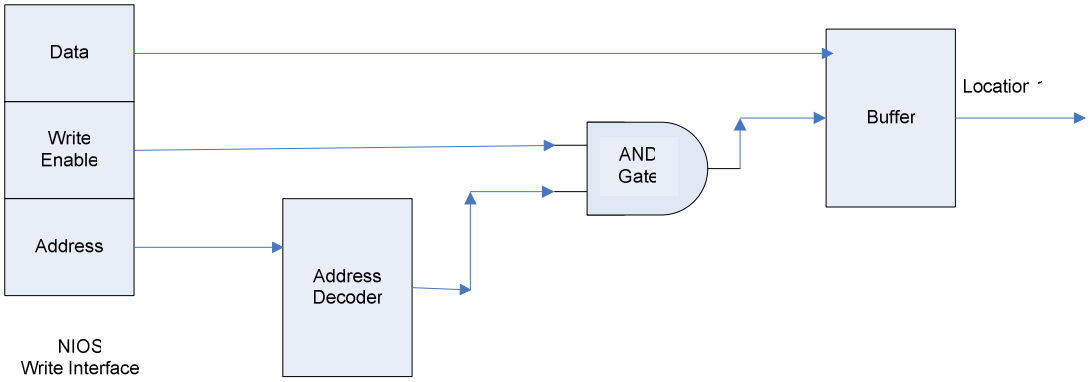

Figure 5.5 : Write Operation

As shown above, during a write operation in Nios , the write-enable signal goes high and hence the data can be send to the corresponding location. The output of the address decoder is enabled by the write enable signal using an AND gate. This is important as the same address could be issued for a read operation also, in which case the write-enable signal would be low. The data goes to the input of a buffer which is enabled by the output of the AND gate. The buffer output thus sends the data to the corresponding location.

2) Read Operation

The read operation is also handled in a similar fashion as above. During a read operation in Nios, the read-enable signal goes high and the data can be read from the corresponding location. The address is decoded, the output of which is enabled by the read enable signal using an AND gate. The data to be read, goes to the input of a buffer

which is enabled by the output of the AND gate. The output of the buffer sends the data to Nios.

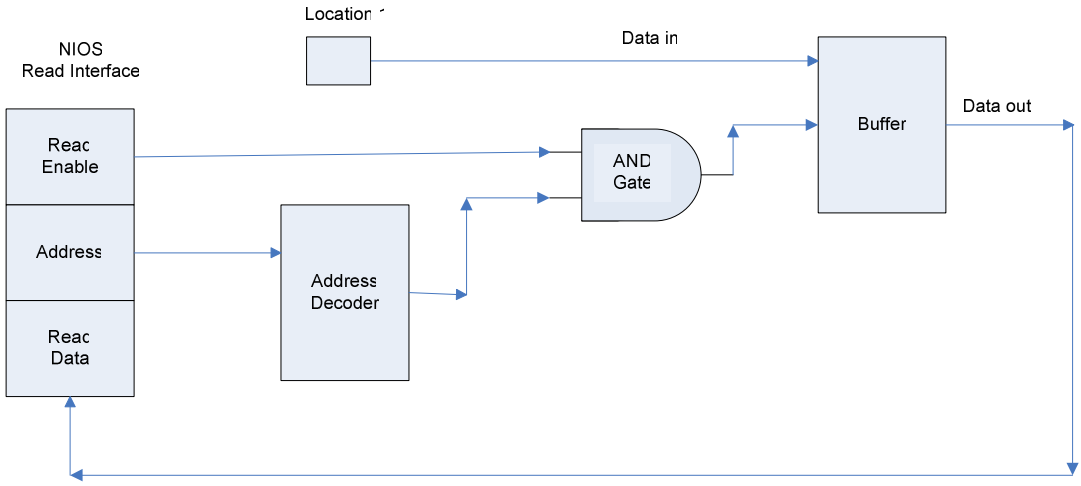

Figure 5.6: Read Operation

The firmware thus helps to set the Tx and Rx status signals. Each status will have an associated address. Thus Nios can read/write the status signals from the FPGA through the firmware.

The firmware also provides communication to various hardware devices like the PLL, DAC, and attenuator. The corresponding pins from these hardware are connected to the FPGA pins. Thus the firmware would send the appropriate signals to the corresponding FPGA pins to communicate with these hardware devices. Thus the read/write commands in Nios establish communication with these external peripherals through this firmware.

5.1.6 Buffer for the samples

This module is used to capture the samples before and after the amplification. These samples are used by Nios for training the pre-distorter. Once the training starts, I/Q samples fed to the input of the PA and the samples obtained from the PA output after demodulation, are both buffered. Each buffer stores 1000 samples at a time. These are then read by Nios and the pre-distortion algorithm applied to find the required coefficients. These coefficients are then updated in the pre-distorter block by Nios.

The output can be monitored on a spectrum analyzer. If the output seems linear, these coefficients can be used. But if the output starts developing inter-modulation products, then the training can be started again. The user can start the pre-distorter training step anytime from the user interface.

5.2 Software Details

The pre-distortion algorithm is written in C++ and executed in the Nios 2 soft processor. The details of the algorithm were discussed in Chapters 2 & 3.The program operates on 1000 I/Q samples and produces the coefficients. These coefficients are then updated in the pre-distorter block. Besides this algorithm, some drivers were written to interface the external peripherals like the PLL, DAC and the attenuator with Nios 2. A firmware was also developed for this purpose (Nios_Interface) which was described earlier.

The first step in the software flow is to initialize the PLL, DAC and the attenuator. This is done by function calls to the respective driver programs written for each of these peripherals. The default value for the attenuator is 62 for Rx and 62 for Tx which corresponds to an attenuation of 31dB each. These values can be specified by the user through the user interface from the PC.

In this application, 9 numerator and 1 denominator coefficients are used. The denominator coefficient is 1.Hence only the 9- numerator coefficients need to be found in this case. So initially the coefficient a0 is set to 1 and all others are set to 0. After the training phase, these coefficients will be updated by Nios. This is done by sending the values to the firmware along with the address. The firmware does the address decoding and updates the values.

The peak amplitude values can also be specified by the user and can be set by Nios. Alpha and beta values are initially set to 1 and 0 respectively. These are used for aligning the Tx and Rx samples. So initially all the samples are multiplied by 1.These values are later updated by Nios. The status signals for both Tx and Rx are also set by Nios. The type of modulation can also be specified by the user.

The drivers written to interface the external peripherals are described below. These were written taking into account the data-sheet specification of each peripheral. The driver details are explained below.

The figure below shows how the driver is interfaced to the external peripherals. The driver is a software program written in C++ and executed in Nios 2.It communicates with the firmware which was described before. This firmware is

implemented in the FPGA using Verilog HDL.The firmware interacts with the external peripheral, which is soldered on the same board as that of the FPGA..

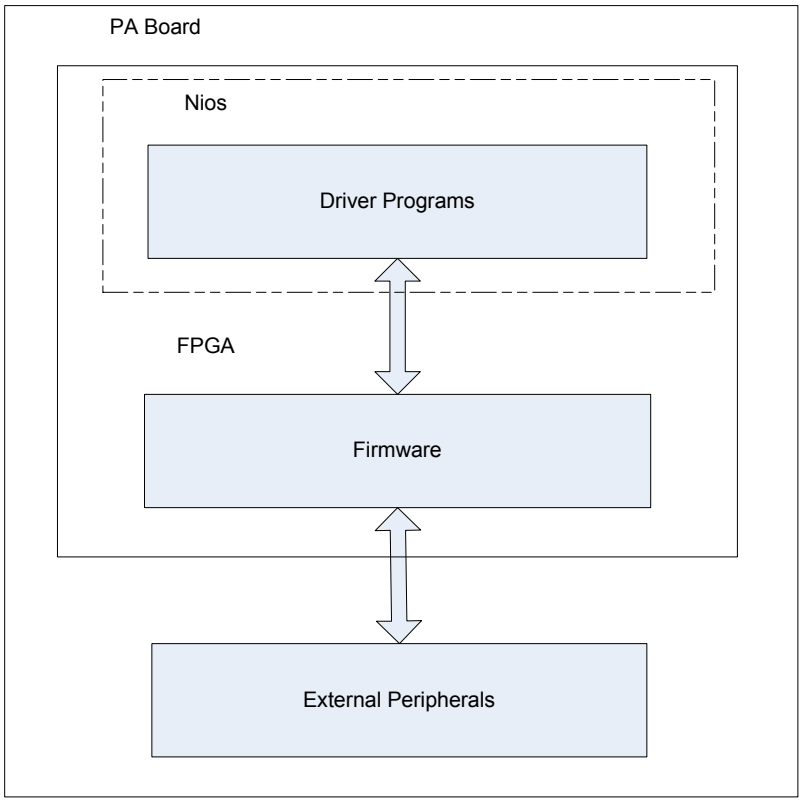

Figure 5.7: Driver and Firmware

The driver programs are written to interface with 3 peripherals - PLL, attenuator and DAC. The details of the drivers are given in the following section.

5.2.1 Driver for PLL

AD9786 from Analog Devices is used as the PLL. To communicate with the PLL, a driver and firmware is developed. The following functions are used in the program. Their functionality is explained below.

Set up :PLLSetup()

The function PLL_Setup is used to set the initial values of the PLL's. An option is also given for inverted board (in case the board has an inverted clock, this bit needs to be set).Function PLL init() is used to initialize the PLL_LE values. This is done with the help of PLL_write function which writes data to the PLL's. It is actually a hex number sequence indicating the status.

Write Function : PLL_write(int PllNum, int val, int verbose)

Each write function would do the following.

1) Initially write the value for the corresponding LE value high or low for data to be clocked in. The Nios write command is used for this :

IOWR_NIOS(PLL_WR,PllRegImage);

2) It then provides a small delay to compensate for the slow response of the PLL by calling the function: PllDelay((int) PLL_DELAY_CNT)

The values are written from MSB to LSB. Each value is written at the rising edge of a clock. As software generated clock is used in this case. Initially the value is written .Then the clock value is changed to 1, so that the previous value will be read in during the rising edge of the clock. After each write, there will be a delay due to the slow response of the PLL. Then the clock is changed to 0 .

Thus the data is written during the rising edge of each clock. Then all the PLL_LE lines are set and this value is written in.

Initialization function: PllInit(int verbose)

This function would call PLL_write function thrice for initializing each PLL. To initialize each PLL to the desired frequency , three writes are required. It is actually a hex number sequence indicating the status. So the 4 PLL's are initialized so that they operate at frequencies 100 MHz, 360 MHz, 680 MHz and 840 MHz respectively.

The PLL Header file pll.h has the following values as shown in the table below.

Variable	**Value**
PLL_SDAT	**0x00000002**
PLL_SCLK	**0x00000001**
PLL_LE0	**0x00000004**
PLL_LE1	**0x00000008**
PLL_LE2	**0x00000010**
PLL_LE3	**0x00000020**
PLL_LE4	**0x00000040**
PLL_LE5	**0x00000080**
PLL_LE_ALL	**(PLL_LE0|PLL_LE1|PLL_LE2|PLL_LE3|PLL_LE4|PLL_LE5)**

Table 5.3 : PLL Mask values

These are the masks used to set different values in the PLL. These are set according to the specifications given in the data sheet of the PLL.

5.2.2 Driver for DAC

The firmware remains the same as before which does the address decoding and read/write to the peripherals.

Set up :fastDACSetup()

The function fastDACSetup is used to set the initial values of the DAC. It is used to set the filter for interpolation. Here a value of 8x is chosen. Then the channel data rate is also set. The following options are also set by this function : modulation using fs/4 , rejecting lsb ,using i and q for processing and real o/p to be routed to DAC. The function

FastDacPutCtrl (int,int) was used to set these values. All these values are actually hex numbers indicating their status specified in the DAC header file.

Write Function : void FastDacPutCtrl (int reg, int val)

The FastDAC has an SPI interface. The driver is designed to take care of this. There are 2 phases to communication – the instruction byte and the data transfer byte. The instruction byte indicates the type of operation- read/write while the data transfer bye indicates the data to be transferred. Each phase requires eight clock cycles. So a

total of 16 clock cycles are required for a single communication cycle. It is implemented as follows:

AD9786_WRITE indicates that it is a write operation.AD9786_BYTES1 indicates the number of bytes to be written. The value in “reg” indicates the register address and “val” indicates the value. The value in “val” is put as the 2nd byte as it is during the write phase. This is done using the following statement. Thus serword contains the instruction byte as well as the data transfer byte.

serword = ((AD9786_WRITE | AD9786_BYTES1 | reg) << 8) | val;

The chip-select was initially set high. So 1 to 0 initializes the instruction cycle. Then 16 clock cycles are used to feed the value in “serword” to the DAC. At each clock cycle, 1 bit is written. The clock bit is toggled from 0 to 1 to 0 each time The register is indicated by the reg value in the instruction cycle.Finally the chip select value is set to high.

Initialization function: FastDacInit (void)

This is called to set the initial values of the DAC. Initially the chip is reset briefly. For this initially FDACR_RESET is set. Along with it FDACR_CSB and FDACR_SCLK bits are also set. A small delay is provided to compensate for the response time of the DAC. Then the reset bit is set to 0 and the value written to the firmware. Thus the chip is reset for a brief time. The SCLK and CSB bits remain high at this time.

Variable	Value
FAST_DAC_BUF	**0x38000000**
FAST_DAC_CTRL	**0x38100005**
FDACC_ACTIVE	**0x00008000**
FAST_DAC_RAW	**0x38100008**
FDACR_CSB	**0x00000001**
FDACR_SCLK	**0x00000002**
FDACR_RESET	**0x00000004**
FDACR_SDI2DAC	**0x00000008**
FDACR_SDO2TS	**0x00000010**

Table 5.4 : Masks for DAC

The fastDAC header file pll.h has the values given in the table.

5.2.3 Driver for attenuator

The firmware remains the same as before which does the address decoding and read/write functions to the peripherals. In the driver for attenuator , only 1 function is used. This function is used to set the attenuator values of the receiver as well as the transmitter.

void AttenuatorSendValues (int msb, int lsb)

This function is used to set the values of the attenuator in transmitter and receiver side. The “msb” value represent the attenuator value in the Rx chain and the “lsb” value indicates the Tx chain attenuation.

Attenuation = msb/2 dB for Rx

= lsb/2 dB for Tx

The Rx and Tx attenuation values are accepted through the function and stored in a variable called “outval’. Each bit of this value is written starting from the MSB during each clock cycles. Hence 16 clock cycles are required to set one set of attenuator values. The attenuator header file attenuator.h has the following values

Variable	**Value**
ATTENUATOR_ACLK	**0x00000002**
ATTENUATOR_ADAT	**0x00000001**

Table 5.5 : Masks for attenuator

5.2.4 Calibration

Once the algorithm is up and running fast enough, the whole system should be calibrated properly for the proper functioning. Calibration is the process of introducing variable delays in both the Tx and Rx chain so that the whole system works intact without any lag. In order to calculate the delays involved in the chain, the auto-

correlation of the input and output samples will be taken and the position of the peak gives the delay.

The system operates at different frequencies. So this offset must take into consideration the frequency requirement of all the blocks before deciding on the digital delay that will be given to the Tx and Rx chain by Nios 2. Calibration will be done while the system boots up and it is just a one time effort unless the system component or wiring changes. This is also a very challenging part of the design.

Once the input and output samples are captured, the auto-correlation of these 2 sets of samples are taken and the delay measured. Digital delays are given in both the Tx and Rx chain so that the samples will be aligned before the pre-distortion algorithm is applied.

CHAPTER 6

RESULTS

6.1 Initial Approach

The initial approach was to study the tradeoffs in implementing an application using fixed point arithmetic (8.24) and floating point arithmetic in a Nios 2 soft processor. In order to study these effects, it is required to implement the application in both floating point and also in fixed point format. The application chosen for the initial approach is Pade Chebyshev algorithm.

This study involves the trade offs involved in the conversion between the two formats .It also provides an opportunity to find out whether these conversions perform faster if written in the C++ IDE of Nios2 platform or whether the custom instructions (hardware) written in verilog will prove to be better.

This work requires the development of two major modules and one sub module. The 2 major modules are modules for the conversion of the fixed point (8.24) to floating point and vice versa. The sub-module is a fixed point (8.24) multiplier. This is required because the c++ software (also in Nios 2) does not support fixed point arithmetic. The fixed point addition and subtraction operations can be done as regular integer operations, having an imaginary binary point in between. But multiplication requires tracking of this binary position , and hence a separate module needs to be constructed.

This work assumes that there will not be any overflow or underflow in the fixed point operations involved in the application as the maximum value of any operation in

this application can be represented in 8.24 format. The operations used by the application involve only addition, subtraction and multiplication.

The fixed point number format represented in binary from has three parts as described earlier - sgn_A int_A .$frac_A$ where they represent the sign , integer portion and fractional portion respectively. The floating point number in binary form also has three parts sgn_B exp_B .$mantis_B$ denoting the sign , exponent and mantissa respectively.

6.1.1 Fixed Point [8.24] to Floating point conversion

```
Fixed2Float( )
{
  Read the input in the 8.24 format (sgn_A int_A .frac_A)
  Assign the sign bit sgn_A to sgn_B .
  if (sgn_A =1 ) then
      int_A .frac_A = ~int_A .frac_A + 1; // take the 2's complement
  end if
  Find the occurrence of the first "1" in the bits int_A .frac_A staring from left.
  if (first "1" detected in int_A ) then
      shift-right int_A .frac_A , by ( 7- pos_int) bits
      exp_B =127(bias) + (7- pos_int)
  else if ( first "1" detected in frac_A )
      shift-left int_A .frac_A , by ( pos_frac) bits
      exp_B =127(bias) – pos_frac
  else
      exp_B = 0
  end if
  Assign the first 23 bits of frac_A to mantis_B
```

Figure 6.1 : Algorithm for fixed point to floating point conversion

An algorithm [Fig 6.1] for the conversion of fixed point (8.24 format) to floating point conversion was derived from the basic concepts involved in the conversion from one format to another. Though some previous works [25] & [26]

mentioned these number format conversions, this specific format conversion did not seem to be addressed by them

These conversions were first implemented in software (c++) using type casting operations, which would convert between fixed point to floating point using type cast operations provided by the c++ compiler. The c++ function implemented for a fixed to floating point conversion using type casting function is illustrated in Fig 6.2.

The modules developed were eventually used for the Pade Chebyshev polynomial determination as mentioned earlier. Initially both these functions were executed in a microprocessor (Intel Pentium 4, 3GHz) for different test vectors. The accuracy and speed of these two functions were compared.

```
int Fixed2float( int y )
{
 float r;//final float value
 unsigned int x;
 x=y; //assign to unsigned int
 if(y<0)
      {
         y=-(pow(2,32)-x
        r=(float)y / ( 1 << frac );
     }
 Else  //if not negative
    {
       r=(float)y / ( 1 << frac );
    }

  return r ;//return the value

}
```

Figure 6.2 : C++ sub-routine for fixed point to floating point conversion

The same function was then executed in a Nios 2 soft processor (cyclone EPC2C35F672C6).Nios 2 IDE provides a c++ programming interface. So both these functions were ported to the Nios 2 platform and the output compared in terms of both accuracy and speed performance.

Nios 2 IDE also provides an option to import custom instructions written in verilog or vhdl . So the algorithm [Fig 6.1] was also written in verilog using Quartus II software from Altera. This was then imported as a custom instruction in the Nios 2 platform and executed. This gave the third set of data points in terms of speed and accuracy.

The three data sets were compared and the best approach was noted. It was found that the algorithm [Fig 6.1] gave better performance in terms of speed compared to the type-casting algorithm[Fig 6.2] when executed in both the processor as well as in Nios2. It also turned out as expected that the implementation using the custom instructions written in verilog gave the best results .

6.1.2 Floating Point to Fixed point conversion

Floating point to fixed point conversion would result in a oss of resolution. So the dynamic range of the application needs to be determined before the conversion takes place.

As described earlier, this work would use an application that determines Pade Chebyshev polynomials .The maximum value of any operation in this application can be represented by 7 integer bits. So the 8.24 format will be sufficient for its

representation in fixed point. The application also requires only 24 bits of accuracy in the fractional part.

```
   Float2Fixed( )
{
    Read the input in the IEEE floating point format (sgn_B exp_B .mantis_B)
    Assign the sign bit  sgn_B to sgn_A. .
   Assign mantis_B  to frac_A from right to left
   Set the last bit of int_A  to 1  // so that it isin  normalized form before shifting
    if (exp_B > 127) then
      left shift the int_A .frac_A bits by (exp_B -127)
    else
      right shift the bits int_A .frac_A by (127- exp_B)
    end if
     if (sgn_A =1 ) then
        int_A .frac_A = ~int_A .frac_A + 1; // take the 2's complement
    end if


}
```

Figure 6.3 : Algorithm for floating point to fixed point conversion

```
int Float2fixed( float val )
{
    unsigned int ret=0;//unsigned int
   int inter;//signed int
   inter=(int)floor( val * ( 1 << frac ));//convert to int

          if(inter<0)//If negative
            {
              ret= pow(2,32)-abs(inter);//2's complement
              inter=ret;//convert to signed
            }
    ret=inter;
    return ret;

}
```

Figure 6.4 : C++ sub-routine for floating point to fixed point conversion

This conversion was also implemented in software (c++) using type casting operations as before which would convert between floating point to fixed point using type cast operations provided by the c++ compiler. The c++ function implemented for a fixed to floating point conversion using type casting function is illustrated in Fig 6.4.

These modules were also used for the implementation of the Pade Chebyshev polynomial determination. The accuracy and speed of the function [Fig 6.4] was compared with that of the software function developed using the algorithm [Fig 6.2] described earlier. The functions were first executed in a microprocessor (Intel Pentium 4, 3GHz) .Then they would be ported to the Nios 2 platform (cyclone 2c25) and the output compared in terms of both accuracy and speed performance.

The algorithm [Fig 6.3] was also written in verilog using Quartus II software from Altera .This was then imported as a custom instruction in the Nios 2 platform and executed. The results obtained from these experiments were compared. The algorithm [Fig 6.3] was found to be faster than the algorithm [Fig 6.4] which used type-casting in both software and Nios 2 platform. In Nios2, as observed in the previous experiment , the implementation using custom instructions(hardware) turned out to be the best in this case also.

6.1.3 Fixed point multiplication (8.24 format multiplication)

This module was developed to enable fixed point (8.24) multiplications. Neither C++ nor Nios 2 provides an explicit module for its implementation. This requires the creation of this module. This module multiplies two 8.24 formats and gives the output also in 8.24 format. Here it is assumed that the application used in the work has a

maximum value that can be represented using 8.24 formats. So the lower bits are rounded off and the output is provided in 8.24 format.

As a next step, the Pade Chebyshev algorithm was executed in software using 2 different modules. The first module was implemented using floating point operations only .The second module used only fixed point operations. The results were noted in both the cases. The same modules were then executed in a Nios 2 platform and the results compared

```
int fixedMult (int  a , int b)
    {
      c= a*b; //c will have 64 bits
     round c at 24 binary points
     c=c>>24
    Assign the 64th bit to the 32nd bit //this is the sign bit
    Assign the last 32 bits of c to the variable "result"
    return result

    }
```

Figure 6.5 **:** Algorithm for 8.24 multiplications

The next section describes in details the experiments that were performed using these algorithms and the results obtained for the same. This study would thus help in understanding the trade offs involved in executing fixed point operations and floating point operations in a Nios 2 soft processor. This would also provide a basis for the whole algorithm implementation.

6.2 Preliminary Results and Inferences

The suggested design was executed in a Nios2 soft processor as well as in a microprocessor (Intel Pentium 4) with different test vectors. The application would

perform operations on 65535 different test vectors. The test vectors are actual I and Q values that were recorded .

6.2.1 Experiments using C++ software in micro-processor

The first step was to execute the conversion algorithms in software (c++) to convert (65535 * 4) test vectors and the time they take to complete was noted. The following table was drawn from this experiment..

Experiments in microprocessor	**Time to complete (ms)**	**Norm Error**
Fixed point to floating point conversion [Fig 17]	**4.531**	**0**
Fixed point to floating point conversion [Fig 18] (using type casting operation provided by compiler)	**4.953**	**0**
Floating point to fixed point conversion [Fig 19]	**2.657**	**0.4005**
Floating point to fixed point conversion [Fig 20] (using type casting operation provided by compiler)	**7.484**	**0.4646**

Table 6.1 : Experiments using software (c++) in microprocessor .

This experiment did not have anything to do with FPGAs. Nevertheless this experiment gave an insight into the speed and accuracy of the algorithms .The root means square error of the outputs gave an estimate of the accuracy provided by these functions

Expected Result

Both the algorithms [Fig 6.1] and [Fig 6.2] were expected to give the same rms error as they are both expected to give similar results. However, it would be interesting to note this result for algorithm [Fig 6.2] as it uses the compiler options. For the same reason, the latter [Fig 6.2] is expected to be faster. This is because the sequential instruction flow of c++ language may make the algorithm slower. Similar inferences were drawn for the other 2 algorithms [Fig 6.3] & [Fig 6.4].

Observations

The fixed point to floating point conversions using algorithm [Fig 6.1] turned out to be faster than the type-casting algorithm [Fig 6.2] .The error analysis shows that both are very accurate in the conversions. These conversions were accurate because the dynamic range of the test vectors was within the limit that could be represented by the 8.24 format.

The floating point to fixed point conversions using the algorithm [Fig 6.3] proves to be much faster than the type-casting algorithm [Fig 6.4].The conversion error for both these conversions are in the same range and hence they are both good in terms of the accuracy.

Inference

It could be seen that both the modules involving the type casting operations do not perform well in terms of speed. This can be due to the inherent overhead associated with these conversions. Moreover each of these type-cast modules does an exponent calculation which might also consume a lot of cycles.

From the above experiment, it is clear that the modules implemented without type-cast operations perform very well in terms of speed and accuracy. So the modules [Fig 6.3] and [Fig 6.4] perform the fastest conversions. The results also indicate that the casting techniques provided by the compiler are not as efficient as they could be for this type of conversion.

This experiment thus forms a base for the rest of the experiments. So it can be inferred that the same modules will prove to be the best in terms of speed even in Nios 2.

6.2.2 Experiments using Nios 2 IDE

The next step was to port these algorithms to the Nios 2 IDE. These algorithms were executed in the Altera DE 2 board. Here it was tested using a fewer number of test vectors.(120 *4)(This was because only a small amount of on-chip memory was initially assigned)Similar readings were taken for this experiment also.

This experiment uses the c++ compiler that comes with the Nios 2 IDE for execution. A comparison of these algorithms gave the best possible module for both number format conversions in the Nios 2 platform.

Experiments in Nios2 IDE	**Time to complete (ms)**	**Root Mean Square Error**
Fixed point to floating point conversion [Fig 17]	**16.041**	**0**
Fixed point to floating point conversion [Fig 18] (using type casting operation provided by compiler)	**89.193**	**0**
Floating point to fixed point conversion [Fig 19]	**8.068**	**0.4005**
Floating point to fixed point conversion [Fig 20] (using type casting operation provided by compiler)	**104.148**	**0.4646**

Table 6.2 Experiments in Nios 2 IDE using c++ code

Expected Result

The results expected were not the same as deduced for the experiment 4.1. Here the accuracy for the 1st two experiments was expected to be same as before. But the 1st experiment was expected to be faster than the second. This is because the conversion algorithm used in the 2nd expt. [Fig 6.4] uses a floating point division operation. Floating point division operation is expected to consume a lot of cycles in Nios 2 platform. So it would be interesting to see if the 2nd algorithm would perform faster. The other experiments were expected to give similar results as the earlier ones.

Observations

The results obtained for this experiment clearly shows that the modules [Fig 6.1] & [Fig 6.3] will perform faster than the other modules. The type-cast algorithms will consume more time as observed in the previous experiment. The graph shown below illustrates this.

Inference

As seen in the earlier experiment it could be seen that both the modules involving the type casting operations do not perform well in terms of speed. This can be due to the inherent overhead associated with these conversions. Moreover each of these type-cast modules does an exponent calculation which might also consume a lot of cycles.

From the above experiment, it is clear that the modules implemented without type-cast operations perform very well in terms of speed and accuracy in Nios 2

platform as well.. So the modules [Fig 6.1] and [Fig 6.3] perform the fastest conversions here.

An important observation in Table 4 compared to Table 3 is that the modules [Fig 6.1] & [Fig 6.3] are much faster than the other modules in Nios 2 IDE compared to the microprocessor. In the microprocessor, the fixed2float module with no type-cast has comparable run-time with that of the module with type-cast. In case of the reverse conversion, the float2fixed module without type-cast is almost 3 times faster than the other one.

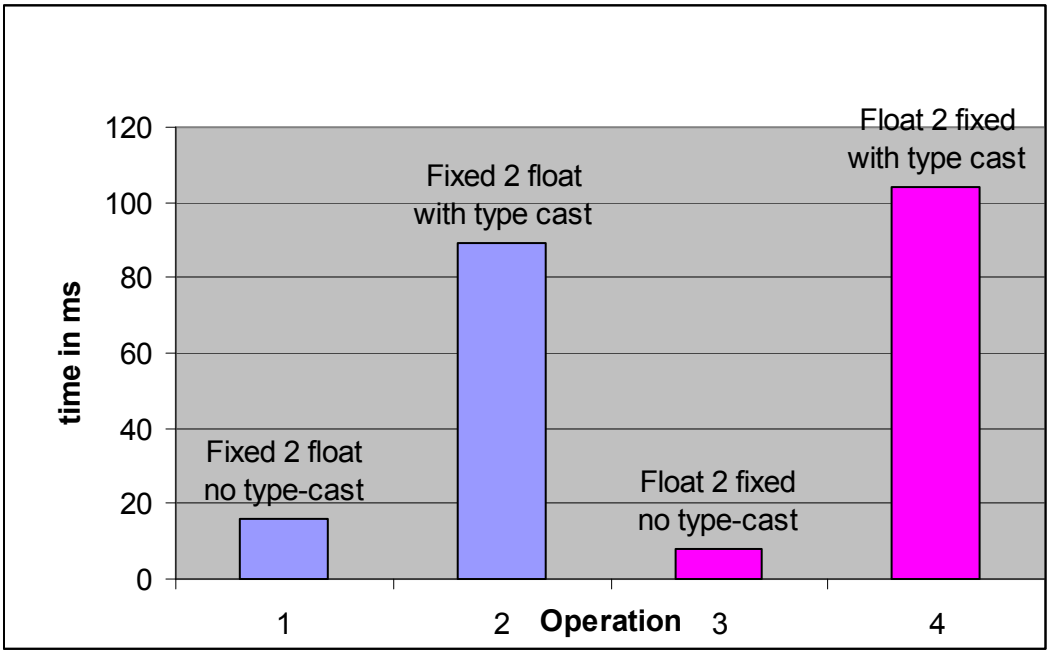

Figure 6.6 : Graph showing the run-times for different conversions (in Nios 2)

It can be observed here that the fixed2float module with no type-cast is 5 times faster than the module with type-cast. In case of the reverse conversion, the float2fixed module without type-cast is almost 12 times faster than the other one. This increase in

speed is due to the fact that the type-cast conversions and the exponent calculation consumes more cycles in Nios 2 as compared to the microprocessor. Hence the difference in speed up is observed.

Thus it can be inferred from this experiment that the modules that can be used for number format conversions in Nios 2 are [Fig 6.1] & [Fig 6.2] as they provide the best results in terms of speed and accuracy.

6.2.3 Experiments in Nios 2 IDE using custom instructions

The previous experiment gives the best modules for number format conversions in terms of both speed and accuracy. Nios 2 provides an option to write custom instruction for certain functions ,that is , implement the logic in hardware. Now the algorithms [Fig 6.1] & [Fig 6.2] were written in **verilog** using the Quartus II software and were imported as custom instructions in Nios 2 platform.(hardware). These were then executed for the same set of input vectors (120 *4) as before.

Experiments in Nios2 IDE using custom instructions	**Time to complete (ms)**	**Root Mean Square Error**
Fixed point to floating point conversion [Fig 3.1]	**1.067**	**0**
Floating point to fixed point conversion [Fig 3.3]	**1.103**	**0.4005**

Table 6.3 Experiments in Nios 2 IDE using custom instructions

Expected Result

The above set of experiments is expected to give the best performance in terms of accuracy and speed. This is because the behavioral model of the verilog version can be optimized to provide better performance than the c++ code.

Observations

The results show that the implementation using custom instructions executes much faster than the previously determined best modules (implemented in c++ in Nios 2).The fixed to float conversion is 16 times faster than it's c++ counterpart and the reverse conversion in custom implementation is 8 times faster. The conversion error remains the same in both cases.

The graph shown below clearly shows the increase in the execution speed obtained while using the custom instructions.

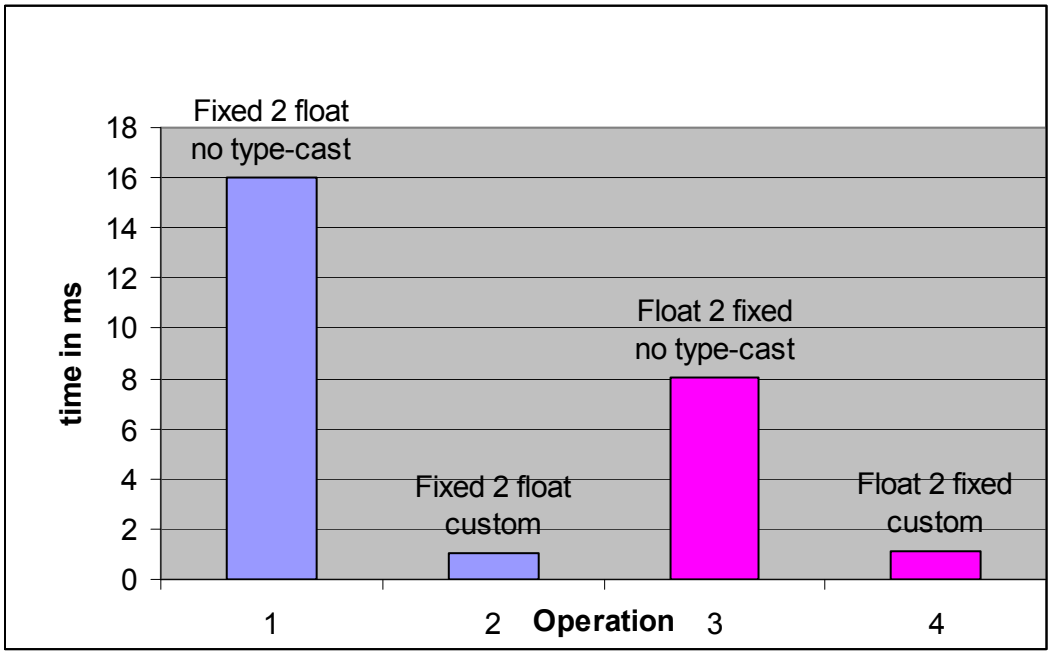

Figure 6.7 : Graph showing the run-times for different conversions (in Nios 2)

Inference

The hardware implementation seems to be the best implementation in terms of speed and accuracy. It seems to be much faster than the software implementation in Nios 2.This is particularly true because of the specialization provided by the hardware implementation.

These modules were be implemented in hardware to perform their respective conversions. The hardware is thus tuned to the application. Hence it performs much faster than a software implementation. The error remains the same - as the same functions are being ported to hardware. Hence the specialization provided by the hardware executes these functions much faster as compared to their software implementation.

Thus from the above set of experiments it is clear that the conversion algorithms using the custom implementation provides the best results in terms of speed and accuracy and hence would be used in the subsequent algorithm implementation. These experiments thus gave an insight into the speed and accuracy of all the different implementations and also helped to decide the best module for the number format conversions.

6.2.4 Floating point and Fixed point operations

The main focus of this work is to find the trade offs between fixed point and floating point implementation of a Pade Chebyshev polynomial determination in the Nios 2 soft processor. For this, the following modules were developed :

(A) Module having only floating point operations.

It receives the test vectors as floats and computes the Pade Chebyshev polynomials and returns the output also in floating point format.

(B) Module having only fixed point operations (conversion algorithms in hardware)

It receives the input vectors in floating point format. These vectors are then converted to fixed point format and the fixed point operations performed (the fixed point multiplication module is used here) and the results converted back to floating point. The modules used for the number format conversion were derived from the experiments performed before.The module which provides the best performance in terms of speed was used here.(custom instruction implementation-(hardware)) .

These modules were executed in Nios2 platform as well in a microprocessor and the results noted.

6.2.5 Experiments in microprocessor (Intel Pentium 3Ghz)

Initially these modules were executed in the microprocessor. The results are shown below .

Experiments in microprocessor	**Time to complete (ms)**	**Root Mean Square Error**
Module A	**0.046**	**0**
Module B	**1.89**	**1.2596e-007**

Table 6.4 : Experiments in microprocessor using modules A & B

Expected Result

When executed in a microprocessor, module A is expected to perform faster. This is because the conversion algorithms implemented using the c++ code in module B may consume more time. So the module A having only floating point operations may perform faster. The accuracy of module A is of course expected to be the best.

Observations

From the table shown above, it could be clearly seen that the module A executes faster than the module B. This means that floating point implementation is faster than the fixed point implementation in microprocessors. The root mean square error of module B is very small (almost negligible) .So module B produces almost accurate results, but turns out to be slower.

Deductions

The above experiment shows that the floating point module provided by c++ executes very fast. So the pade chebyshev algorithm for 65536 test vectors were executed in 0.046ms .But on the other hand, module B (fixed point) turns out to be slower. This is mainly due to the extra overhead (time) incurred in terms of the fixed to floating point conversions ,adjustments required in multiplication module and the reverse conversions back to fixed point. Presumably Pentium also has special hardware to perform the floating point arithmetic.

Thus it can be inferred that an algorithm implemented in floating point algorithm will definitely prove to be faster than a fixed point implementation in a

microprocessor. This is because the specialized multipliers in the processor can execute the floating point operations quite fast. So a fixed point implementation is not quite required in this case. (refers only to addition, subtraction and multiplication operations)

6.2.6 Experiments in Nios2 processor (Atlera Cyclone II EP2C35F672C6)

These modules were then executed in a Nios2 processor (50Mz). Here a set of 500 test vectors were given as input to the pade chebyshev algorithm. Module B was implemented using the fastest conversion obtained from the previous experiments. We found that the implementation using custom instructions was the best in terms of speed and accuracy for both the conversions. So module B was implemented using the custom instructions for the conversions .But the multiplication module and the algorithm was implemented in c++ that could be used in the Nios 2 IDE.

Experiments in Nios2 IDE	**Time to complete (ms)**	**Root Mean Square Error**
Module A	**825.993**	**0**
Module B	**6296.673**	**1.5625e-008**

Table 6.5 : Experiments in Nios 2 IDE using modules A & B

Expected Result

The results obtained using the Nios 2 platform was expected to be in favor of module B. Module B was expected to be faster than module A, though the latter might

be better in terms of accuracy. This was expected because floating point operations consume a lot of cycles in Nios 2 hardware.

Observations

The results obtained for this experiment was not as expected. Module A seemed to be much faster than module B. Module A executes 7 times faster than module B. It can again be seen that the root mean square error of module B is almost negligible.

Deductions

Module B turns out to be much slower in spite of using the custom instructions (hardware implementation) of the conversion algorithms. This can be traced to the overhead caused by the multiplication module. This module was implemented using four 16 bit multiplications and shift operations. Thus each fixed point multiplication would result in 4 different multiplications which would definitely slow down the module. This can also be the reason for the earlier software version of the module B to be much slower than module A.

The bottleneck detected, the next step was to change this module also to a custom instruction implementation (hardware).The module was also modified to reflect the algorithm [Fig 21] to make it faster without including a lot of multiplications. Hence a new module was developed – Module C.

(C) Module having only fixed point operations (with multiplication module also in hardware)

It receives the input vectors in floating point format. These vectors are then converted to fixed point format and the fixed point operations performed (the fixed point multiplication module is used here) and the results converted back to floating point. The modules used for the number format conversion were derived from the experiments performed earlier. The module which provides the best performance in terms of speed was used here.

6.2.7 Experiments in Nios2 processor with Module C.

These modules were then executed in a Nios2 processor (50Mz).As performed earlier, a set of 500 test vectors were given as input to the pade chebyshev algorithm.

Experiments in Nios2 IDE	**Time to complete (ms)**	**Root Mean Square Error**
Module C	**154.772**	**1.5625e-008**

Table 6.6 : Experiments in Nios 2 IDE using module C

Expected Result

Module C is expected to run faster than the modules A &B. This is because this module uses 3 custom instruction implementations (hardware).This is bound to increase its execution speed.

Observations

The results obtained clearly indicate module C executes faster than both modules A&B. It can be seen that module C executes 5 times faster than module A and 40 times faster than module B. It can also be seen that the rms error in this case is also very small. This can thus be considered as the most efficient module for the algorithm implementation in Nios II processor.

Deductions

The final results clearly show the need to use the fixed point implementation of the pade chebyshev algorithm. Here the speed improvement comes at the cost of additional hardware. The modules for conversions and multiplication were implemented in hardware in a Nios 2 processor. This brought about a drastic improvement in performance- almost 40 times faster than module B while maintaining the same accuracy.

The graph shown above [Fig 6.8] clearly gives the big picture. It can be seen that module B executes the slowest in Nios 2 while module C being the fastest. The module A (the all floating point module) seems to be faster than module B but slower than module C for any finite number of inputs. Thus for any finite number of inputs, module C would provide the best results.

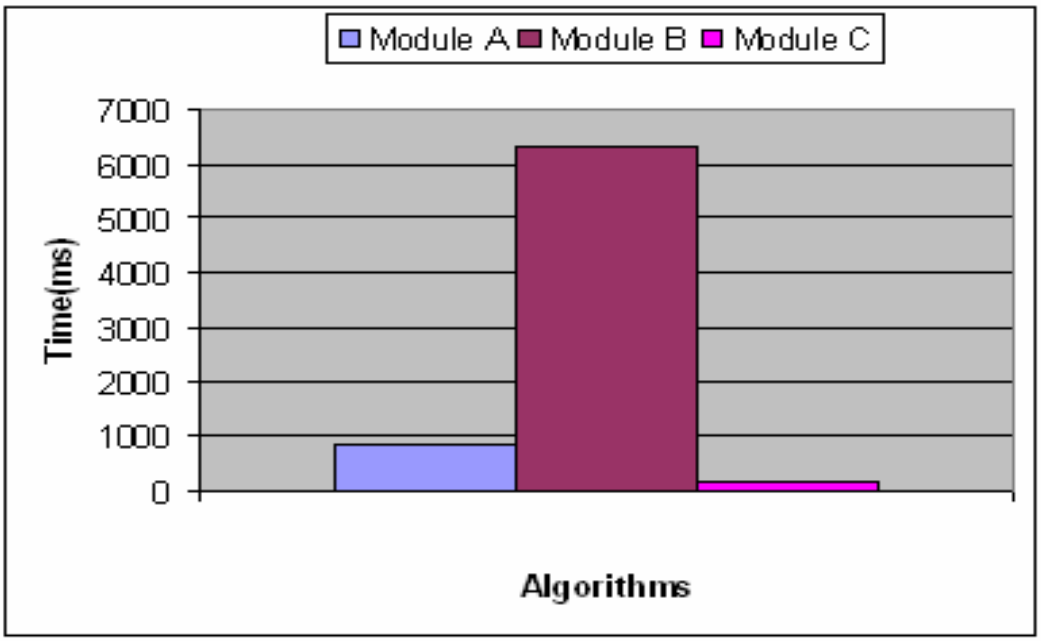

Figure 6.8 : Graph showing the different algorithms vs. time for completion

Module C proves to be faster than the floating point implementation because of the inherent specialization. In module C, the conversion functions and multiplications were implemented in hardware. This provides specialization, which increases the execution speed. These algorithms also produced results with negligible error.

The results also indicate that as the number of inputs increase the run-time of each module increases linearly. But the run-time of module B increase at a faster rate compared the other two. In any case, module C will always execute faster than the floating point implementation for the same number of inputs.

6.3 Trade offs

There are some tradeoffs that should be looked into as a result of the experiments. There are 2 tradeoffs.

Speed vs Accuracy

This is not evident in these experiments because only multiplication, addition and subtraction operation were being used in the algorithm. But it could be seen that as division and square root operations are used, the precision requirement may not be adequately represented by the fixed point formats. So the error would go up. But for faster execution, fixed point implementation is inevitable in Nios 2 platform. So there will always be a trade-off between speed and accuracy in this case. In such situations the dynamic range of the output and the admissible error needs to be initially determined. Thus it depends on the type of application being considered.

Speed vs Hardware

An important tradeoff that can be observed in these experiments is the speed vs hardware requirements. It was found that the module C executes faster than the module B(all floating point).But this comes at the expense of additional hardware .The conversion algorithms and the multiplication module were implemented in hardware.

Increased hardware reduces the space that could be occupied by other components in the chip. It can also lead to increase in power which is not desirable in most cases. The table below shows the FPGA resources required by module C.

FPGA Resources	Number of elements
Total Logic Elements	**1037**
Total Registers	**4**
Embedded Multiplier 9-bit elements	**8**

Table 6.7 : FPGA resources used by module C

This hardware implementation provided specialization and resulted in faster execution. It was found that the float2fixed module required 305 LEs and the fixed2float module required 96 LEs respectively for the hardware implementation. But both these modules were found to be much faster than their software counterparts. Hence the specialization comes at the expense of extra hardware.

This shows that, the decision to convert into hardware needs to be taken very carefully. If the hardware implementation brings about significant increase in speed, then it is beneficial to do so. But if it does not, then it would be better to stay with the software implementation. The algorithm implemented in this work brings about

a speed increase a factor 5. Hence it was feasible to go for the hardware implementation of the 3 functions.

Thus any algorithm that needs to be executed in Nios 2 needs to address these issues in the implementation phase. This work addressed both these issues and was able to provide a fixed point implementation which was significantly faster than the corresponding floating point implementation.

6.4 Implications of the results

The initial results describes the tradeoffs involved in executing an application using fixed point arithmetic and floating point arithmetic in Nios 2 soft processor. The same implementation was also done using a micro-processor and results were compared.

The first stage of the work was to determine the best possible conversion algorithms for both fixed point (8.24) to floating point format and vice versa .This was executed in a microprocessor as well in the Nios 2 processor. In both cases, it was found that the type-cast functions were slower than the non-type cast functions.

These function were then implemented in hardware and called as custom instructions in Nios 2.These were found to be much faster then the software implemented functions. The float to fixed conversion was 8 times faster than the corresponding software module and the reverse conversion was 16 times faster. This speed up comes from the specialization obtained from the hardware implementation. But this comes at the cost of extra hardware (LEs).

The next stage explores the trade off between floating point and fixed point operations in Nios 2 and the microprocessor. The first two experiments shows that

floating point implementation provides faster execution than the corresponding fixed point implementation.(in Nios 2 and the microprocessor).The slower speed of the fixed point algorithm was traced to the multiplication module which had overheads. This was also then transformed into hardware.

At this stage it was found that the custom implementation of the fixed point algorithm is much faster than the corresponding floating point algorithm in Nios 2.This experiment shows that fixed point operations in Nios 2 executes 5 times faster than the floating point operations even though the conversion algorithms have to be executed in the former.

Thus the work presented here provides the best implementation method of one of the modules to be implemented in Nios 2 hardware. The intermediate results obtained will be useful during the implementation of the other algorithms in Nios 2. It was found that fixed point implementation of the algorithm executes much faster then the corresponding floating point. This result will be very useful for the implementation of a variety of time-critical applications.

This increased speed comes at the expense of additional hardware. But the small hardware requirement provides a large increase in execution speed with very good accuracy. This trade –off definitely proves to be beneficial. Here the specialization of the hardware resulted in an increase in the execution of the algorithm.

6.5 Speed comparison of the final pre-distortion algorithm

The whole predistortion algorithm is divided into mainly three software modules in Nios 2, namely the pade-chebyshev module, the QRD module and the back-substitution module. The pade –chebyshev module calculates the polynomials for each

I-Q sample. The QRD module performs givens rotation and builds the matrix for each sample .Finally back substitution is performed on the final matrix obtained after the operation of the first two modules on all samples. All these modules together constitute the pre-distortion algorithm.

From the preliminary results obtained ,we can conclude that fixed point implementation of an algorithm using custom instructions in Nios 2 would definitely provide improvement in terms of speed as compared to floating point implementation. Thus these algorithms can be implemented using custom instructions, in fixed point. In the fixed point implementation, the algorithm is not completely implemented as fixed point. Only the pade-chebyshev module and QRD module is implemented in fixed point, while the givens rotation and back substitution modules are both implemented in floating point. The floating point implementation of these modules is done using custom instructions.

The fixed point implementation using custom instructions might prove to be faster than the floating point implementation. But the former implementation suffers from many conversions back and forth from fixed point to floating point format. It is to be seen whether these intermediate conversions would slow down the whole algorithm.

The algorithm can also be implemented using floating point custom instructions. This can be faster than the fixed point implementation as it does not involve any intermediate number format conversions. Hence, the algorithm can be implemented in 3 different ways. These can be divided into 3 different modules.

a) Module X

This module contains the floating point implementation of the pre-distortion algorithm. The floating point operations are done in software(Nios 2).

b) Module Y

Here Pade-chebyshev and QRD modules are implemented using fixed point custom instructions. The givens rotation and back substitution module are implemented using floating point custom instructions.

c) Module Z

Here the pre-distortion algorithm is implemented using floating point custom instructions.

6.6 Experiment in Nios 2 IDE using modules X and Y

The modules X and Y are initially compared. The time taken by each module to process 1000 samples is noted. The accuracy of the final output is also noted.

Experiments in Nios2 IDE	**Time to complete (s)**	**Root Mean Square Error**
Module X	**15.325**	**0**
Module Y	**0.49428**	**0.0387**

Table 6.8: Experiments using modules X and Y

Expected Result

The floating point custom instructions are expected to provide accurate results of course. But the speed of these operations would be slow as the floating point operations using the software library consumes more time as seen in the previous experiments. So module Y is expected to be faster, which has fixed point implementation using custom instructions.

Observations

It could be seen that module Y is almost 30 times faster than module X. The accuracy of the output for the module Y is also very good. It only has a root mean square error of 0.03.Thus module Y is more efficient than module X for the implementation of the pre-distortion algorithm.

Deductions

The improvement in speed for module Y comes from using custom instructions for the fixed point operations in the Pade-chebyshev and QRD modules. The floating point operations in the remaining 2 modules also used custom instructions. Thus, the use of custom instruction has brought about a significant improvement in the execution speed of the algorithm.

But if we look at the resources required to implement the modules, module Y requires more FPGA resources as compared to module X, which uses only the software libraries. The table below shows the resources used by module Y.

FPGA Resources	Number of elements
Total Logic Elements	**2432**
Total Registers	**488**
Embedded Multiplier 9-bit elements	**43**

Table 6.9 : FPGA resources used by module Y

6.7 Experiment in Nios 2 IDE using modules Y and Z

Here, all the floating point operations are implemented using custom instructions. If the whole algorithm is implemented using floating point custom instructions, there will be no loss of accuracy. So the next step is to compare the speed and accuracy of the algorithm implemented using floating point custom instructions and the one using fixed point custom instructions.

Experiments in Nios2 IDE	Time to complete (ms)
Module Y	**494.28**
Module Z	**183.390**

Table 6.10 : Experiments using modules Y and Z

Expected Result

The floating point custom instructions are expected to provide accurate results of course. But the speed of these operations need to be determined as they depend on the speed of the available floating point units .If each floating point operation takes only one clock cycle , then the speed will be comparable to that of the fixed point implementation , in which case , the final design will use the floating point custom instructions. But Nios 2 itself has a lot of overheads in the form of data fetch, register load etc. So it would be interesting to see which module would take the upper hand.

Observations

It could be seen that module Z is almost 2.7 times faster than module Y. The accuracy of the output for the module Z is comparable to that of the original floating

point operations (Module X).Thus module Z seems to be more efficient than module Y for the implementation of the pre-distortion algorithm in terms of speed and accuracy.

Deductions

The improvement in speed for module Z comes from the fact that there are no intermediate conversions back and forth to different number formats in this implementation.(everything is in floating point format).But in module Y, fixed point operations are used n the Pade-chebyshev and QRD modules ,while floating point operations in the remaining 2 modules. So the intermediate number format conversions are required. This causes it be slower than module Z.

Experiments in Nios2 IDE	**Resources**
Module Y	**Custom fixed to float conversion unit** **Custom float to fixed conversion unit** **Custom fixed point multiplication unit** **Custom floating point operations unit**
Module Z	**Custom floating point operations unit**

Table 6.11 : FPGA resources used by module Y and module Z

Even in terms of FPGA resources, the module Z requires only fewer resources as compared to module Y. The resources used by both the modules are given in the table below.

Thus it is clear that module Z is more efficient than module Y in terms of FPGA resources also. Hence we can conclude that the pre-distortion algorithm can be implemented very efficiently using the module Z, where custom floating point operations are used. The module Z is almost 78 times faster than module X also .But this improvement in speed comes at the expense of increased FPGA resources.

6.8 Experiments involving the whole system

The pre-distortion algorithm is implemented in Nios 2 and the whole system is built into the FPGA as described in chapter 4.The output of the power amplifier with and without pre-distortion is observed in a power spectral analyzer and the output saved.

The power amplifier used in this case is ZHL-42 from Mini circuits. It operates over a wide range of frequencies varying from 700 MHz to 4.2GHz.It has a high gain of 30dBm and medium high power of 28dBm.The output of the amplifier was monitored using a power spectral analyzer.

FPGA Resources	Number of elements
Total Logic Elements	**32,435**
Total Registers	**26950**
DSP block 9-bit elements	**94**

Table 6.12 : FPGA resources for the hardware implementation

The hardware details described in chapter 4 are implemented on a Stratix 1S80 FPGA. The FPGA resources required for this implementation is noted in the table.

The FPGA resources given in the above table does not include the resources required for the algorithm implementation. So the total amount of FPGA resources required for the whole implementation should also include the ones required for the algorithm implementation in Nios 2 also. This varies according to the type of implementation.

Two types of inputs are used in this experiment. The inputs used here are a two-tone signal and a 64-QAM signal. The response of the power amplifier before and after pre-distortion is recorded.

6.9 Experiment using two-tone signal as input

In this experiment, a two-tone signal at 770 MHz is given as the input to the power amplifier. The output of the amplifier before and after pre-distortion algorithm is applied is recorded. The figure below shows the output recorded on a power spectrum analyzer.

This is a two-tone signal with center frequency of 770Mhz.The plot in green indicates the output of the PA before pre-distortion and the red plot shows the output after pre-distortion. It has almost 35dB cancellation of the inter modulation components. All the inter modulation products are at 65dBc below the fundamental tone after pre-distortion is applied.

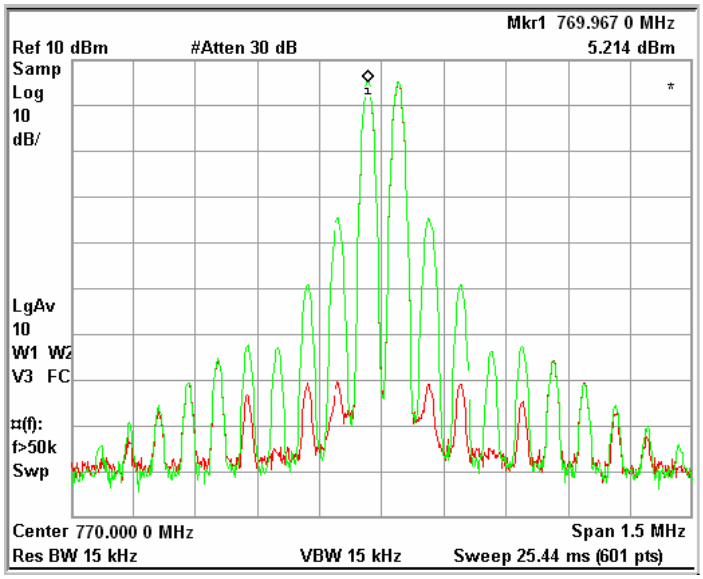

Figure 6.9: Output of 2-tone signal before and after pre-distortion

Experiment using two-tone signal	**Intermodulation products(in dB)**
Before Pre-distortion	**-25**
After Pre-distortion	**-60**

Table 6.13: Output power of IM products

The table above clearly shows the cancellation brought about by the pre-distortion algorithm. Thus it can be seen that the pre-distortion algorithm cancels the inter-modulation components.

6.10 Experiment using 64-QAM as input

Here a 64-QAM signal is fed as input to the power amplifier. The output of the amplifier before and after pre-distortion algorithm is applied is recorded in this case also. The figure 6.8 below shows the output recorded on a power spectrum analyzer.

The plot in green shows the output of power amplifier before pre-distortion, while the pink plot indicates the output after pre-distortion.

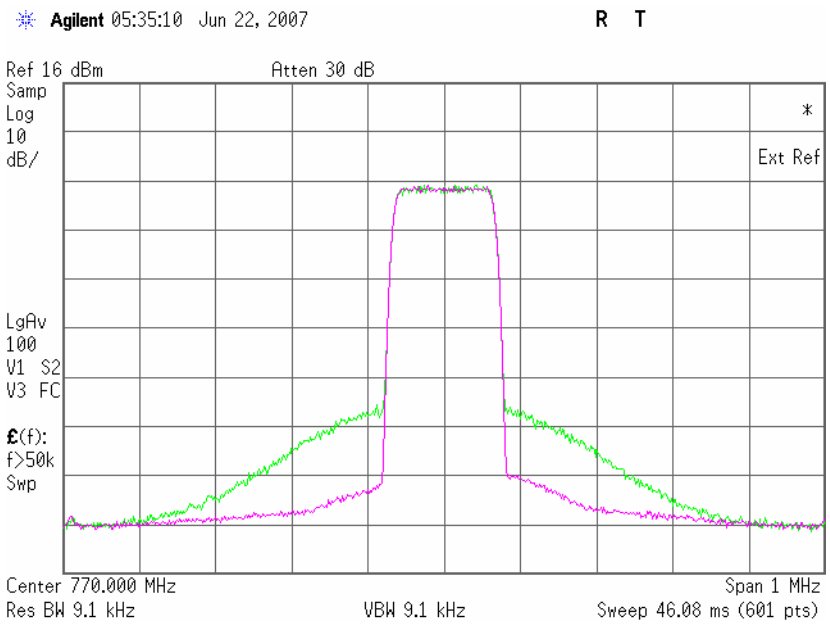

Figure 6.10: Output of 64- QAM signal before and after pre-distortion

Experiment using 64-QAM	**Intermodulation products(in dB)**
Before Pre-distortion	**-50**
After Pre-distortion	**-65**

Table 6.14: Output power of IM products

The figure above shows a 64-QAM signal with center frequency of 770Mhz.The plot in green indicates the output of the PA before pre-distortion and the pink plot shows the output after pre-distortion. It has almost 15dB cancellation of the inter modulation components. All the inter modulation products are at 60dBc below the fundamental tone after pre-distortion is applied. The table also gives the output power of the intermodulation products before and after pre-distortion is applied.

The results clearly indicate the effect of the pre-distortion algorithm on the power amplifier output. Thus it could be seen that the pre-distortion algorithm implemented in Nios 2 soft processor cancels the intermodulation components .Hence the output of the power amplifier becomes almost linear. This can be verified for different kinds of signals using different types of power amplifiers.

CHAPTER 7

CONCLUSION AND FUTURE WORK

The thesis gives insight into the existing PA linearization techniques. It discusses in detail the various types of linearization techniques available and also the problems associated with each of them.

Most of the existing linearization methods were reviewed and the digital pre-distorter method was chosen to be the starting platform. A novel adaptive algorithm is proposed which would converge faster and would provide better results than the current linearization techniques. The proposed algorithm and its implementation details are discussed. The hardware resources required for its implementation and the challenges involved are also explained.

Preliminary work and the results obtained from the same are discussed .The results helped to find the most efficient implementation technique for the pre-distortion algorithm .The algorithm implemented using floating point custom instructions is found to be the best in terms of speed.

The whole system consists of the transmitter, receiver and the adaptive algorithm integrated into a single FPGA chip. The FPGA and the external peripherals are all placed on the same board. The algorithm gives very good performance. For a two-tone signal at 770MHz, it provides 35dB cancellation of the inter-modulation products. Similarly for 64-QAM at 770 MHz, it provides 15dB cancellation of the inter-modulation components.

The work is novel as no one has tried to implement the adaptive algorithm work completely on an FPGA. The linearization algorithm has been implemented on a soft processor – Nios 2, which has not been attempted before. Most of the linearization algorithms are performed on powerful DSP processors which provide fast and accurate results, but at the cost of power and money. This thesis is an attempt to implement a new adaptive linearization technique on an FPGA and provide better results.

Future work can focus on introducing some kind of parallelism into the algorithm. An approach would be convert one of the modules into a verilog module which can process in parallel while the rest of the operations can be done in Nios 2.This may speed up the whole algorithm, but can take up some FPGA resources as well. This tradeoff requires more investigation.

The results from these experiments could be gathered to derive a method by which any general algorithm or a class of algorithms can be assigned a fixed point or floating point number format for their implementation. Hopefully a theoretical formulation of this problem would be a nice way to classify different algorithms.

.

BIBLIOGRAPHY

1. Stephen Bruss, April 23 2003, “Linearization Methods”
2. Steve C. Cripps, “RF Power Amplifiers for Wireless Communications”, Artech House, 1999.
3. Steve C. Cripps, “Advanced Techniques in RF Power Amplifier Design”, Artech House, 2002.
4. T. Sowlati, Y. Greshishchev, and A. Salama, “Phase Correcting Feedback System for Class E power Amplifier”, IEEE J. Solid-State Circuits, vol. 32, pp. 544–550, April 1997.
5. Frederick H. Raab, Peter Asbeck, etc, “RF and Microwave Power Amplifier and Transmitter Technologies”, November 2003, High Frequency Electronics.
6. V. Petrovic, “Reduction of Spurious Emission from Radio Transmitters by Means of Modulation Feedback”, in Proceedings of IEE Conference on Radio Spectrum Conservation Techniques, September 1983, pp. 44-49.
7. L. Kahn, “Single-sided Transmission by Envelope Elimination and Restoration,” Proc. IRE, July 1952, pp. 803–806
8. Harold S. Black, “Translating System”, U.S Patent No. 1 686 792, October 1928.
9. P. B. Kenington, “Power Amplification Techniques for Linear TDMA 10 Base stations”, in Proceedings of GLOBECOM ’92, Orlando, USA, December 1992, pp. 74-78.

10. R. D. Stewart and F. F. Tusubira, “Feedforward Linearisation of 950MHz Amplifiers”, IEE Proceedings, vol. 135, Pt. H, No. 5, October 1988, pp. 347-350.

11. James K. Cavers, “Convergence Behavior of an Adaptive Feedforward Linearizer”, in Proceedings of the 44th IEEE Vehicular Technology Conference, Stockholm, Sweden, VTC-94, June 1994, pp. 499-503.

12. Peter B. Kenington, Mark A. Beach, Andrew B. and Joseph P. McGeehan, “Apparatus and Method for Reducing Distortion in Amplification”, PCT Patent No. WO 91/16760, April 1991.

13. Morris KA & McGeehan (2000), “Gain and Phase Matching Requirements of Cubic Predistortion Systems”, IEEE Electronics Letters, 36(21), 1822-1824.

14. Mohamed K Nezami, “Fundamentals of Power Amplifier Linearization Using Digital Predistortion”, Sept 2004, High Frequency Electronics.

15. D. Psaltis, A. Sideris, and A. A. Yamamura, “A multilayer Neural Network Controller,” IEEE Contr. Syst. Mag., pp. 17–21, Apr. 1988.

16. Willianm H. Press etc, “Numerical Recipes”, ISBN 0-521-30811-9, Cambridge University Press, Cambridge, New York New Rochelle Melbourne Sydney.

17. Holly Q. He and Mike Faulkner, “Performance of Adaptive Predistortion with Temperature”, 5th International Symposium on Signal Processing and its Applications, ISSPA’99, Brisbane, Australia, 22-25 August, 1999.

18. Slim Boumaiza and Fadhel M. Ghannouchi, “Thermal Memory Effects Modeling and Compensation in RF Power Amplifiers and Predistortion Linearizer”, IEEE Transactions on MTT, Vol. 51, No 12, Dec. 2003.

19. D. G. Luenberger, 1969, “Optimization by Vector Space Methods”, John Wiley & Sons, Inc., p. 55.

20. B. N. Datta, 1995, “Numerical Linear Algebra, and Applications” Brooks/Cole Publishing Company, pp. 333-4, and pp.337-8.

21. Haykin, S. [1996] Adaptive Filter Theory, 3rd Ed. (Prentice-Hall Inc., New Jersey).

22. Volder, J. E. [1959], “The CORDIC Trigonometric Computing Technique”, IRE Trans. on Electronic Computer 8, pp. 330-334.

23. Bruno Haller, etc, “Efficient Implementation of Rotation Operations for High Performance QRD-RLS Filtering”, Published in Proc. ASAP’97, Zurich, Switzerland, July 14-6, 1997, pp. 162-174.

24. F. Charot and V. Messe. A Flexible Code GenerationFramework for the Design of Application SpecificProgrammable Processors . In 7th internationalworkshop on Hardware/Software Codesign,CODES’99, Rome, Italy, May 1999.

25. Ki II Kum,Jiang Kang , Wonyong Sun, A floating point to fixed point C-converter for Fixed-point Digital Signal Processors

26. Arnaud Massiani,Fabienne Nouvel. MC-CDMA system using fixed-point interference cancellation and single user detection,In 2004 IEEE 5th workshop on SPAWC

27. H. Keding, M. Willems, M. Coors, and H. Meyr.FRIDGE: A Fixed-Point Design And Simulation Environment. In Design, Automation and Test in urope 1998 (DATE-98), 1998.

28. S. Kim, K. Kum, and S. Wonyong. Fixed-Point Optimization Utility for C and C++ Based Digital Signal Processing Programs. IEEE Transactions on Circuits and Systems II, 45(11), November 1998.

29. S. Kim and W. Sung. A Floating-point to Fixed-point Assembly program Translator for the TMS 320C25. IEEE Trans. Circuits and Systems, November 1994.

30. www.wikipedia.com