2013-01-10

Background Calibration of a 6-Bit 1Gsps Split-Flash ADC

Anthony Crasso
Worcester Polytechnic Institute

Background Calibration of a 6-Bit 1Gsps Split-Flash ADC

by

Anthony Crasso

A Thesis
Submitted to the Faculty
of the
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements for the
Degree of Master of Science
in
Electrical and Computer Engineering
by

_____

January 2013

APPROVED:

_____

Professor John McNeill, Major Advisor

_____

Professor D. Richard Brown

_____

Professor Stephen Bitar

Abstract

In this MS thesis, a redundant flash analog-to-digital converter (ADC) using a "Split-ADC" calibration structure and lookup-table-based correction is presented. ADC input capacitance is minimized through use of small, power efficient comparators; redundancy is used to tolerate the resulting large offset voltages. Correction of errors and estimation of calibration parameters are performed continuously in the background in the digital domain. The proposed flash ADC has an effective-number-of-bits (ENOB) of 6-bits and is designed for a target sampling rate of 1Gs/s in 180nm CMOS. The calibration algorithm described has been simulated in MATLAB and an FPGA implementation has been investigated.

Acknowledgements

I would like to take a section to thank everyone that supported this thesis. There are many ways in which I was supported. These included financially, grammatically, emotionally and ultimately academically.

First and for most I would like to thank Professor McNeill for his advice and teaching throughout this design process. He was very understanding of my programming ability and was very open to my creativity. Without his original idea of the split-ADC and his guidance, none of this would be possible.

A few students that I would like to thank are first Robbie D'Angelo for introducing me to my first independent study with professor McNeill and establishing this connection that has made all this possible. My MQP partners Karen Anundson and K Thet for the amazing experience of working with two of the smartest and hardest working people I know. Our MQP report was great document and our project was even better. Without our report as a starting point for this thesis, I'd be still writing this thesis a couple of years from now. Lastly I would like to thank the students in my the NECAMSID lab Rabeeh Majidi and Jianping Gong. These students supported my idea and are in the process of implementing a flash ADC so that this calibration can be used in a real circuit. Rabeeh was also co-auther of the ISCAS paper published on this calibration and again, without that work I'd still be writing this thesis.

For funding I would like to thank the ECE department for allowing me to TA my favorite analog courses and to Analog Devices for funding the lab. I don't know what I'd do if I was in more debt right now.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Analog to digital converters are vital to many modern systems that require the integration of analog signals with digital systems. These applications can range from music recording to communications applications to medical instrumentation. [7] These converters are implemented using a variety of architectures, sizes and speeds. The demand for smaller, faster, lower power converters has led to the investigation of alternative ADC design techniques. As CMOS technologies improve and smaller process sizes lead to an increase in the implementation of digital signal processing, the potential for digital correction and calibration of ADCs has emerged. [14]

With the advance of CMOS technology, high speed and low power, analog-to-digital converters with high effective number of bits (ENOBs) are in demand. Flash ADCs as fast low resolution analog-to-digital converters are typically used in wireless receivers and high density disk drives [13, 16]. In comparison to other types of analog-to-digital converters, the simple analog structure of flash ADCs makes them useful in deep sub micron CMOS. Working in a deep sub micron process has the advantage of high speed but at the price of increased variation and device mismatch, decreasing the ADC effective number of bits (ENOB). Especially in flash ADCs, device mismatch causes offset error in each comparator, affecting differential and integral non-linearity (DNL and INL) of the ADC and degrading ENOB performance. One method of recovering ENOB is to improve matching by increasing device size. However this approach imposes area and power consumption costs. Several

methods have been proposed in the literature such as averaging and digitally controlled trimming [16] to mitigate the effects of comparator offsets.

Redundancy has been shown to be an effective method of yield improvement in IC designs [6]. Comparator redundancy has the advantage of tolerating the large comparator offsets associated with small device sizes necessary to reduce input capacitance and provide the high speed flash ADC with acceptable fan-in. Examples of redundant flash ADCs can be seen in [6, 21, 16, 9]. Each of these have different way to use the information by creating more trip points and this design is yet another.

In this paper digital background calibration of a redundant flash ADC is done using the split ADC structure while all redundant comparators are used to raise the effective number of bits. Since all the comparators are used, the difficulty associated with edge effects is reduced. The assignment of raw comparators output to ADC codes is performed using a look up table (LUT) which is updated continuously in the background to tolerate comparator threshold variation due to effects such as temperature drift.

The scope of this work is included the design, simulation and implementation of the calibration algorithm. The correction of an Flash ADC is first explored using a look-up table. The rest of the paper focuses on implementing the calibration algorithm to produce the values used in the look up table. Additional sections highlight possible implementations of the flash ADC and a FPGA implementation of the algorithm.

Chapter 2

Background

Analog-to-digital converters (ADCs) provide a link between the analog signals of the real world and the world of digital signal and data processing. Figure 2.1 shows the basic concept of an analog to digital converter: a continuous analog signal input is converted to a discrete digital signal at the output. This digital output can then be processed by a digital system such as a processor or an FPGA.

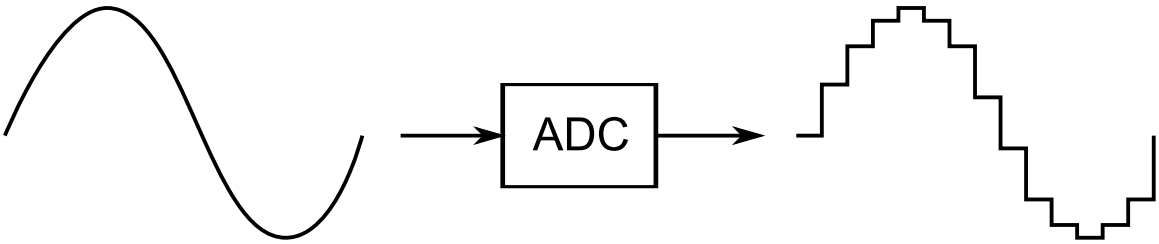

Figure 2.1: Analog-to-Digital Converter

The rapid growth and improvement of digital processing systems has led to more processing being implemented in the digital domain. Decreasing process sizes mean increased numbers of logic gates in a given space. The computational power of a digital system increases with the number of logic gates. Digital processing can often offer advantages in design flexibility. An FPGA, for example, allows for digital hardware designs to be reconfigured to suit changing system needs. In order to take advantage of these digital processing systems, however, real world analog signals must be converted into digital signals. As its

name implies, an analog to digital converter fills this need.

ADCs can be designed with a variety of architectures depending on the requirements for the device. Some of these architectures also include calibration methods to improve the ADC's performance. This section will introduce ADC concepts and architecture types and calibration.

2.1 Sampling

One of the fundamental parts of an analog to digital converter is a sampling component. In order for a continuous time analog signal to be converted to a discrete time signal, the analog signal must be sampled in time. Figure 2.2 shows a signal v_{in} being sampled every time T_s. Equivalently, the signal is being sampled at a frequency f_s. Ideally, the sampled input will be a series of impulses, shown in Figure 2.3, with time spacing T_s and an amplitude determined by the value of the input signal at time nT_s, where n is an integer.

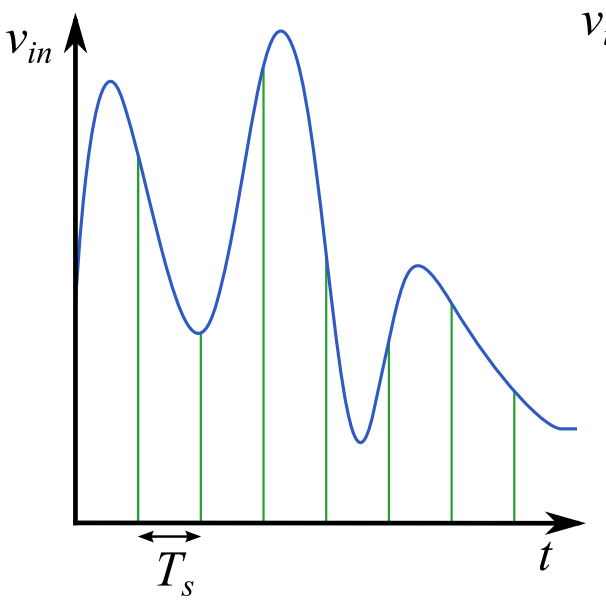

Figure 2.2: Continuous Signal

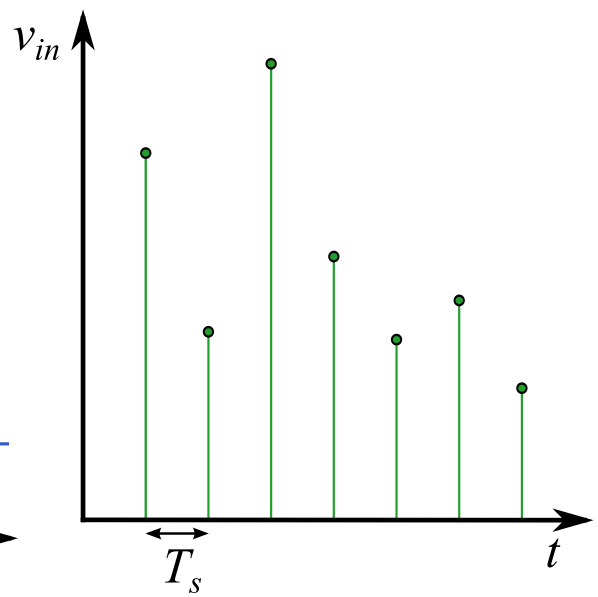

Figure 2.3: Discrete Time Sampled Signal

Choosing a sampling frequency to ensure that the sampled signal contains sufficient information about the original signal and prevents aliasing can be done based on the Nyquist-

Shannon sampling theorem. That is, if the sampling frequency

$$f_s > 2B_{signal} \tag{2.1}$$

then the signal can be fully recovered. This holds as long as the samples are not restricted to discrete y values as they are in a digital signal. The discrete behavior of the y values introduces errors due to quantization [8].

Analog-to-digital converters can be categorized into two major categories based on their sampling frequencies: oversampling and Nyquist converters.

2.1.1 Oversampling Converter

Oversampling converters are characterized by a sampling frequency much higher than the Nyquist rate. This high sampling rate causes larger spacing in the signal spectrum, ideally preventing the overlap of samples in the spectrum that leads to aliasing effects. These converters are typically used when high accuracy is required and a reduction in the effects of aliasing is desired, such as in band limited signals like music. The design trade-off for the accuracy is a lower throughput. These converters also require a large number of samples to perform a single conversion. [7]

2.1.2 Nyquist Converters

Nyquist converters can process signals up to one half of the sampling frequency. This is in accordance with the Nyquist theorem that the sampling frequency must be at least twice the bandwidth of the signal in order to recover the information from the original signal. That is

$$f_s = 2 * Bandwidth_{inputsignal} \tag{2.2}$$

These converters have higher throughput than oversampling converters. The trade-off made for this speed is a reduced accuracy. Some Nyquist converters are high speed with what is considered to be low to medium accuracy, such as flash or pipeline ADCs. Other Nyquist converters fall into the middle range for both speed and accuracy, such as successive approximation converters (SAR) and cyclic converters. These converters tend to be a good

compromise between slow oversampling converters and less accurate options such as flash converters. [20]

2.2 Quantization

Quantization is also necessary for analog-to-digital converters. Quantization is the process of assigning certain ranges of values from a continuous signal range to discrete values. This assignment creates quantization errors. A quantization error is the difference between the quantized value and the original signal. In Figure 2.4, the original signal v_{in} is shown in blue. If a sample of this signal is taken at time T_1, it would be quantized to n_2, as shown in Figure 2.5. The difference between the sample of v_{in} and its quantized value n_2 is indicated by the black bar.

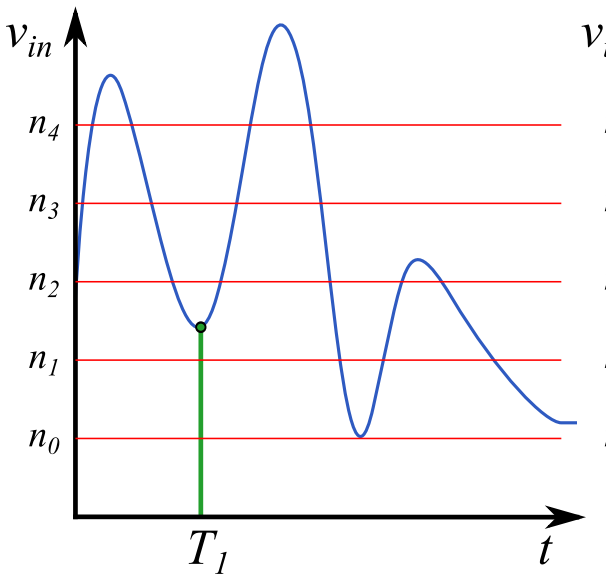

Figure 2.4: Quantization Sample

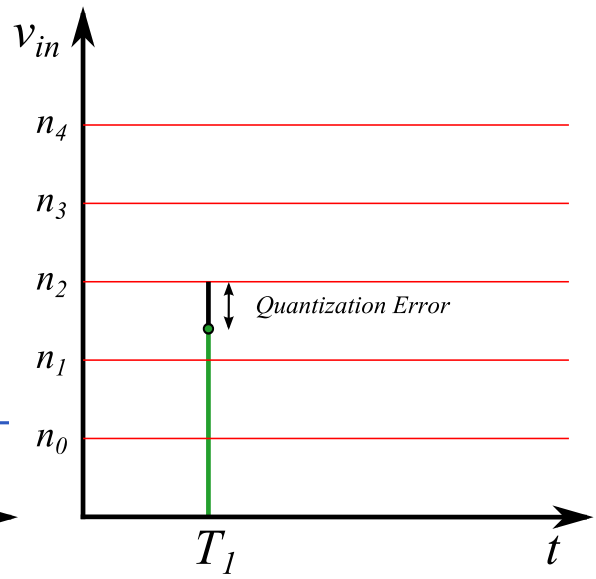

Figure 2.5: Quantization Error

Quantization errors are directly related to the resolution of the ADC. An ADC that needs an accuracy within a very small margin of error is going to need more quantization levels. More levels require a larger number of digital bits to encode all the information. Higher resolution often comes at the cost of converter speed, so converters need to be optimized for required speeds and resolutions. This optimization depends greatly on the

type of architecture chosen for the ADC design.

2.3 Classification of ADCs

Analog-to-digital converters are often divided into three major categories based on converter speed and accuracy. Table 2.1 was adapted from [15]

Low Speed, High Accuracy	Medium Speed, Medium Accuracy	High Speed, Low Accuracy
Integrating	Successive Approximation	Flash
Oversampling	Algorithmic	Two-Step
		Pipeline
		Time-Interleaved

Table 2.1: Classification of ADCs [15]

A selection of ADC architecture types with their respective sampling rate and resolution ranges can be seen in Figure 2.6.

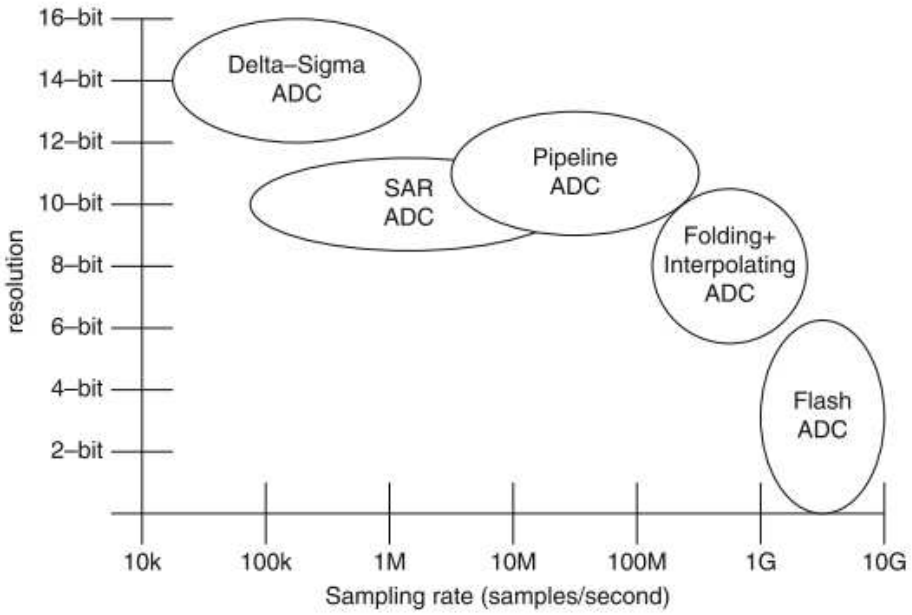

Figure 2.6: ADC Architecture Comparison [1]

2.3.1 Low Speed, High Accuracy

Some converters that are characterized by low speed and high accuracy include the integrating ADC and the sigma-delta oversampling ADC. Integrating converters are slow and their conversion times are proportional to the input voltage. Integrating ADCs require, in general, 2^Nclock cycles for N bits of resolution. A higher resolution means a slower conversion time [8].

2.3.2 Moderate Speed, Moderate Accuracy

Other converters can be categorized by moderate speed and moderate accuracy. Successive-approximation register ADCs and cyclic ADCs are both included in this classification of converters. [8]

SAR ADC

The SAR architecture algorithm is often described as being similar to a binary search algorithm. One common analogy for a binary search is looking for specific information on a page of a book. The searcher does not know the correct page and can only ask the book's owner "yes or no" questions. The search would begin by starting at the center of the book and asking if the page being searched for is a higher number than the current page. If it is, then divide the upper half of the book in half and ask the same question for the new halves until there is only one page left. The decisions algorithm for a SAR converter is shown in Figure 2.7.

The SAR ADC follows a similar algorithm that compares input voltages and reference voltages to determine a digital output value. The main advantage that a SAR design offers is the use of only a few analog components, particularly the use of only one comparator, that results in a compact area and simpler design. The trade-off for this space is made in the maximum sampling rate. A converter with a sampling rate f_s would require the comparator, DAC and SAR logic, shown in Figure 2.8, to operate at Nf_s.

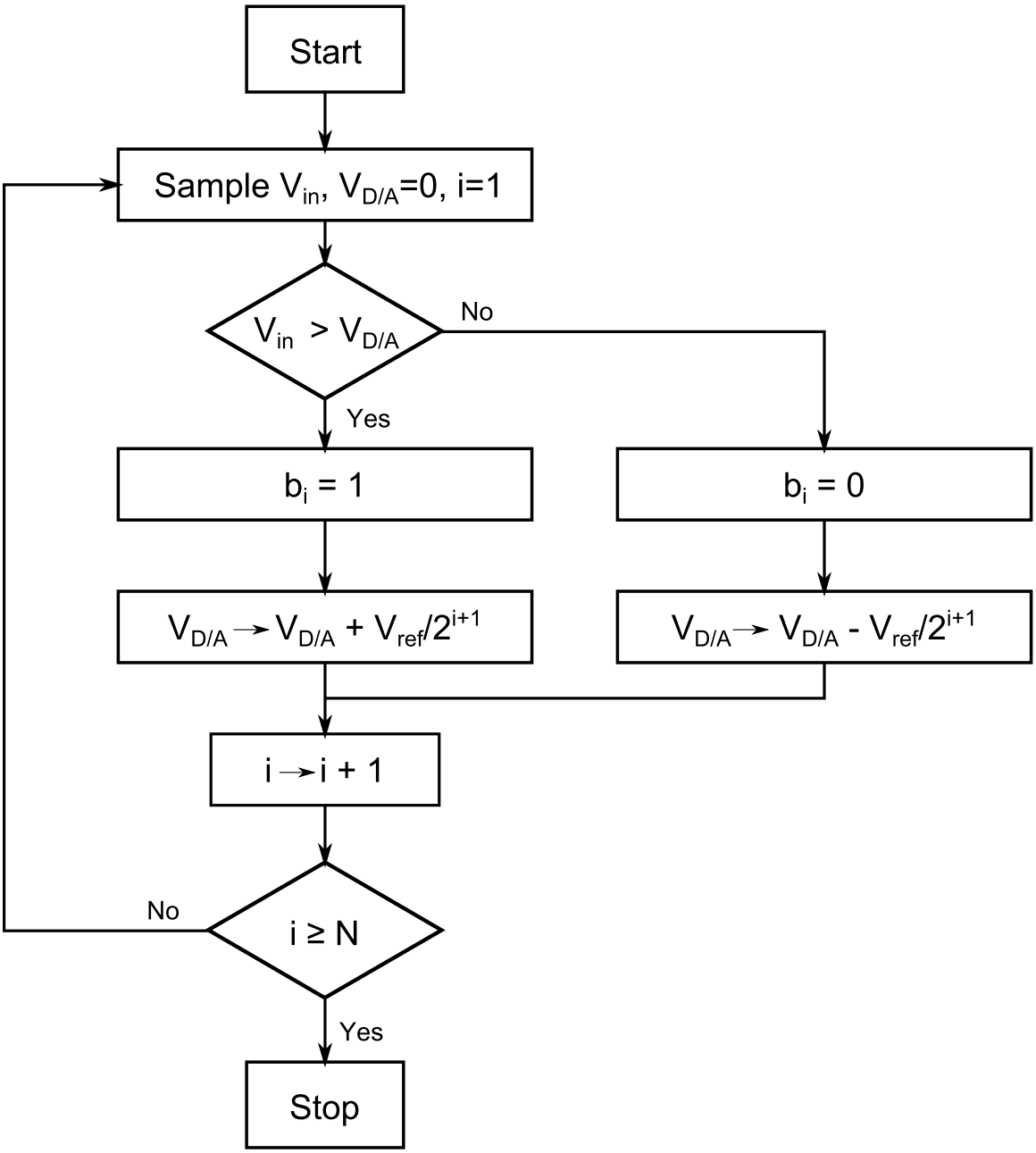

Figure 2.7: SAR Algorithm [5]

Cyclic ADC

Cyclic ADCs also fall into this middle category. A cyclic ADC (also known as an algorithmic ADC) operates similarly to the SAR ADC. In a cyclic ADC, however, it is not

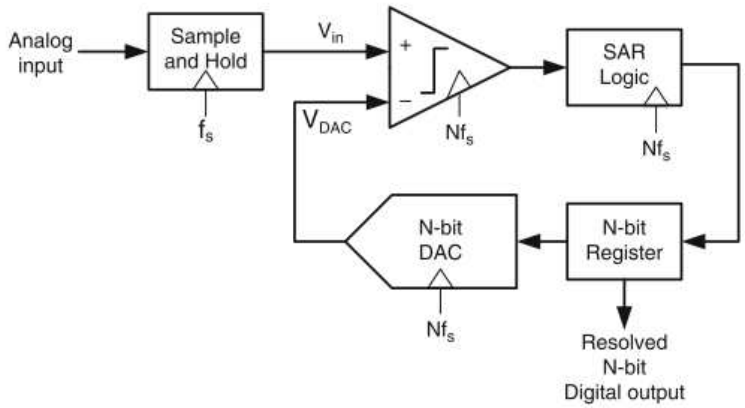

Figure 2.8: SAR ADC Topology [1]

the reference voltage that changes, but rather the residue is put through a gain stage and amplified. In a cyclic converter, the input is sampled and compared to a threshold voltage. A 1-bit digital output is generated and the residue generated by subtracting the output of the DAC from the original input is fed back into the sample and hold circuit. The cycle repeats for the same number of cycles as desired bits. The high level block diagram of a cyclic ADC is shown in Figure 2.9.

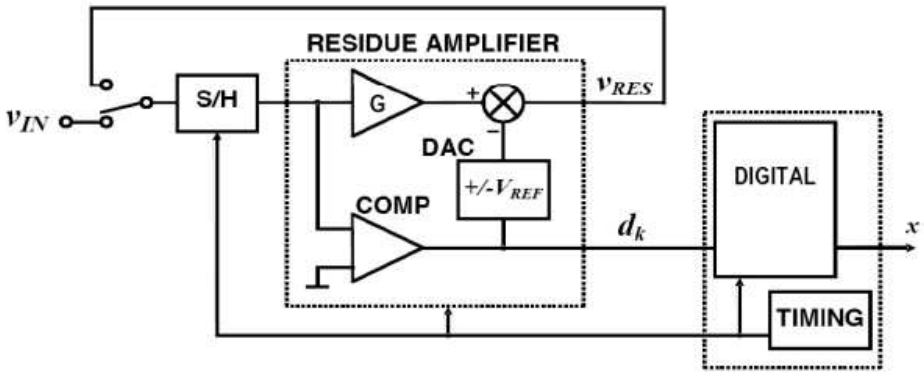

Figure 2.9: Cyclic ADC Block Diagram[20]

2.3.3 High Speed

Some converters that can be categorized as high speed and low accuracy converters are two-step, time-interleaved and flash. Pipeline ADCs can be low, moderate or high accuracy.

Flash

A flash ADC can be compared to a ruler (Figure 2.10). A ruler maps an infinite precision value length to finite precision value (e.g. 4mm). A flash ADC uses comparators to perform a similar function.

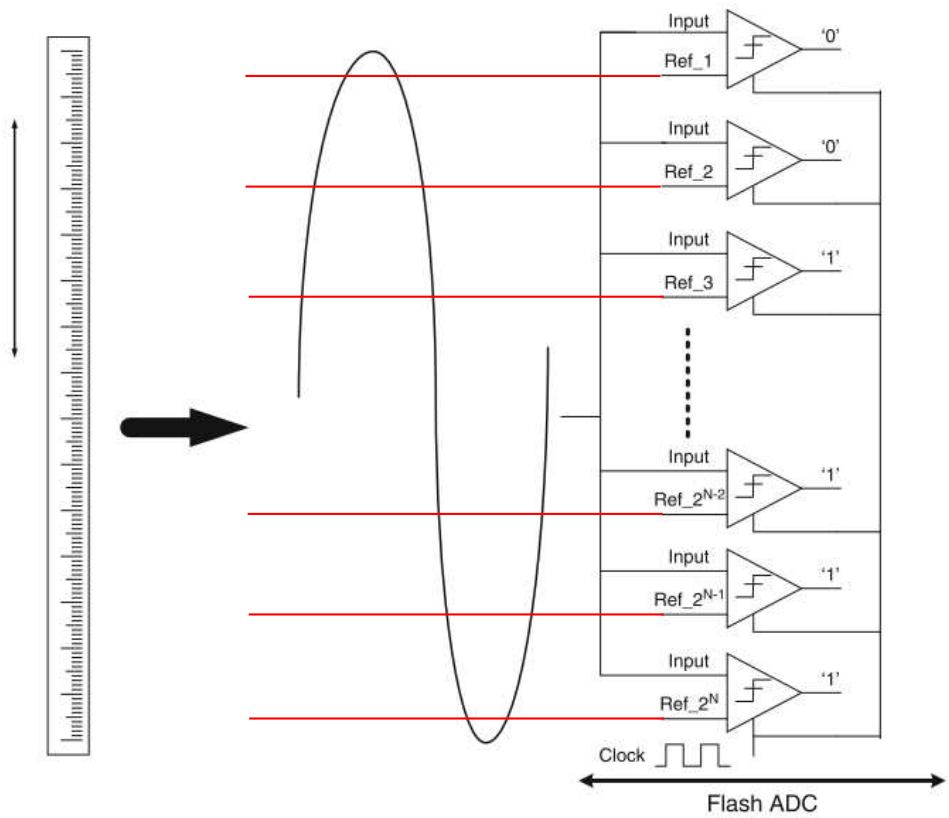

Figure 2.10: Ruler Flash Analogy [1]

A flash converter compares input (infinite precision value) to a number of fixed references to determine a binary output (finite precision value). The output of the comparators is in thermometer code. In this example, if the input is higher than the reference, the ther-

mometer bit is one, otherwise it is zero. This thermometer code must then be translated into the equivalent binary value. The number of reference levels can be expressed as:

$$Reflvls = 2^N \tag{2.3}$$

where N is the accuracy for the ADC. A flash ADC that has 16 comparison levels will have an accuracy of 4 bits. From this relationship, it can be observed that the number of comparators required will increase exponentially compared to the increase in desired resolution. Because of this, flash ADCs are usually used in low resolution applications. The main advantage that flash converters offer is speed. Flash comparators have the potential for conversion to take only one clock cycle. Flash ADCs are often included in the design of other ADC architectures such as a pipeline[1].

Pipeline

Pipeline ADCs are also high speed ADCs and can be capable of resolving medium to high resolutions. [1] These ADCs work by converting a signal from analog to digital in stages. Each stage converts a portion of the output resolution. The first stage converts the most significant bits (MSB) and the subsequent stages convert less significant bits until the least significant bits (LSB) are converted. The overall general architecture of a pipeline ADC is shown in Figure 2.11. Each stage has a similar structure, shown exploded in Figure 2.11. Each block contains a sample and hold block to sample the analog signal. This feeds into a small flash converter that resolves n-bits. This n-bit output is fed back through a DAC and the binary value is subtracted from the original input signal to generate a residue voltage.

Figure 2.12 shows a common implementation of a pipeline ADC stage. Typically, the DAC, summer, gain stage and sample and hold are implemented together in one block called a multiplying digital-to-analog converter (MDAC). The residue voltage is amplified and input into the next stage of the pipeline until the desired number of bits have been resolved.

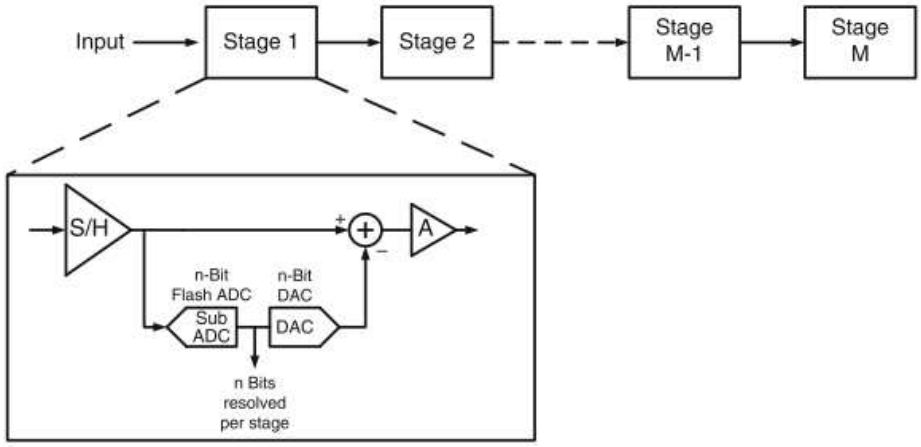

Figure 2.11: General Pipeline ADC Architecture[1]

2.4 Performance Metrics

Performance metrics are needed in order to evaluate the performance of the ADC in this work.

2.4.1 Effective Number of Bits (ENOB)

The effective number of bits of an ADC is one measure to compare different ADC designs. The ENOB is characterized by the equation

$$ENOB = \frac{SINAD - 1.76}{6.02} \tag{2.4}$$

where SINAD is the signal to noise and distortion ratio [1]. The 6.02 term converts decibels to bits and the 1.76 is due to the quantization error in an ideal converter. The equation for the SNDR is

$$SINAD = 20log_{10}\frac{Signal(volts, RMS)}{Noise + Harmonics(Volts, RMS)} \tag{2.5}$$

.

2.4.2 Figure of Merit (FOM)

A Figure of Merit (FOM) used to compare analog-to-digital converters is defined as

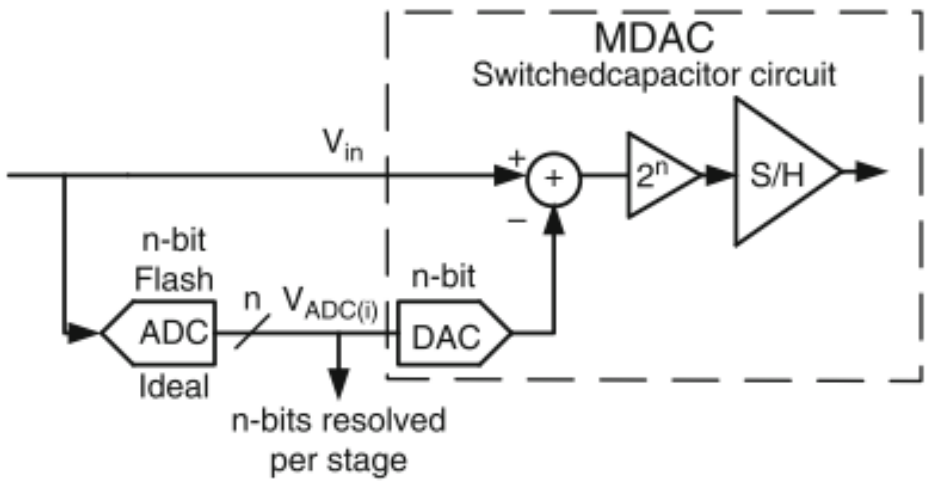

Figure 2.12: Pipeline Stage with MDAC [1]

$$FOM = \frac{Power}{(2^{ENOB})(f_s)} \tag{2.6}$$

The FOM takes into account the power consumption of the ADC, the ENOB, and the sampling frequency f_s. A lower FOM indicates better ADC performance based on these parameters. Lower power consumption, higher ENOB and a higher sampling frequency all contribute to a lower FOM. All three of these design characteristics require design tradeoffs with one another. Increasing the sampling frequency f_s will accommodate an increased signal bandwidth, relaxes filtering requirements and can sometimes relax resolution requirements; however, increasing the sampling speed results in increased power consumption. Increasing resolution will accommodate an improved dynamic range and relax filtering requirements, but can result in increased power consumption. If trying to optimize for power consumption, compromises need to made in the design to sampling speed or resolution.

2.4.3 Differential Non-Linearity (DNL)

When the step size of an ADC's output is not equal to the ideal step size, the ADC is said to have differential nonlinearity. The DNL measurement for an ADC is classified based on amount of least significant bit (LSB) values that the actual transfer function deviates

from the ideal transfer function. If the DNL is greater than 1 LSB, a non-monotonic transfer function will cause missing codes. Figure 2.13 shows the deviation of a real transfer function from the ideal.

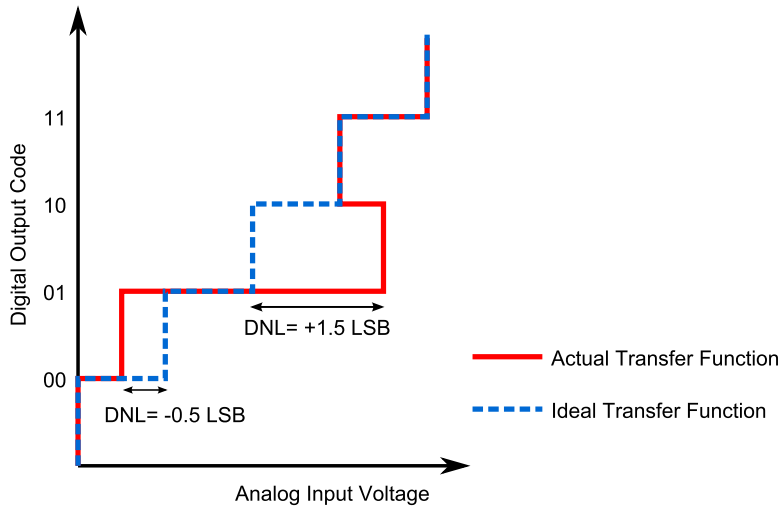

Figure 2.13: DNL

2.4.4 Integral Non-Linearity (INL)

The transfer function of an ideal ADC can be represented by a best fit line, typically either an endpoint fit or a least squares fit. An ADC that exhibits integral non-linearity will have a transfer function that is not a perfect line. The maximum difference between the actual and ideal transfer characteristic is the INL. This concept is illustrated in Figure 2.14.

2.5 Calibration

Non linearity in the gain stage is a common error in pipeline ADC designs. This error is caused by capacitor mismatch and low DC operation amplifier gain, but the exact error is typically not known by the designer. In order to compensate for this non linearity, various calibration techniques are used.

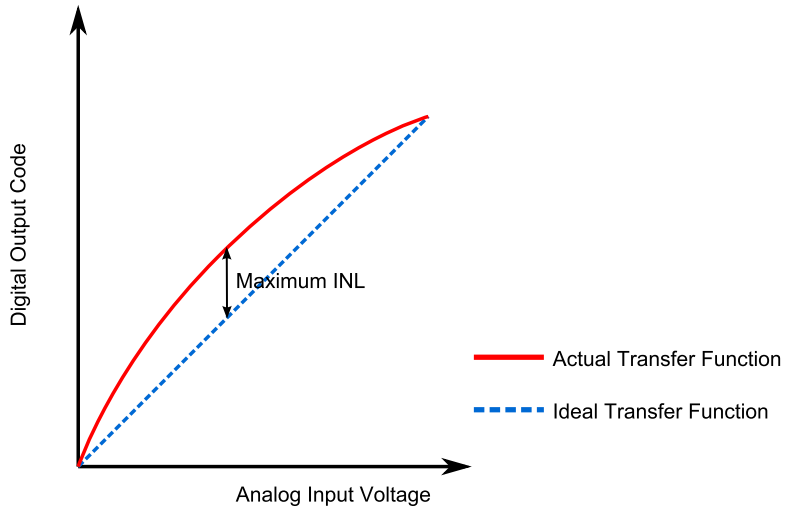

Figure 2.14: INL

2.5.1 Foreground Calibration

In a foreground calibration scheme, the unknown errors are estimated by interrupting the operation of the ADC and then injecting a known signal. The expected output is compared to the actual output to measure the error [18]. Once the error is acquired, Least Mean Square (LMS) algorithms can be used to correct for the error.

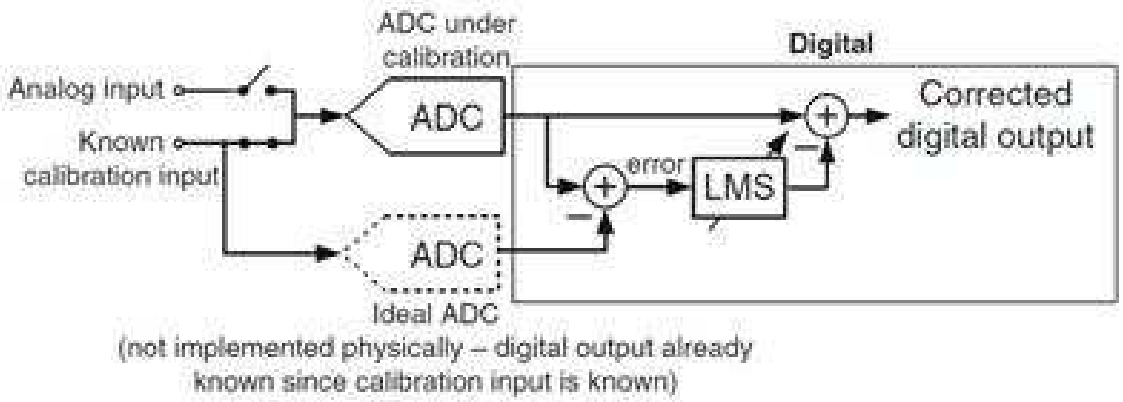

Figure 2.15: Foreground Calibration Block Diagram

As shown in Figure 2.15, analog input signal is fed into the actual ADC and a known

signal is fed into the ideal ADC. Since it is impossible to implement an ideal ADC, this component is simulated digitally. Another digital component is used to calculate the error between the actual output and the ideal output. This same digital component will then correct the digital output for this calculated error. The main advantage of using foreground calibration is that one can achieve the corrected digital output in a few clock cycles. However, the operation of the ADC is interrupted during calibration. This interruption is impractical in some applications.

2.5.2 Background Calibration

Background calibration technology can correct errors of ADC circuits without interrupting the operation of the ADC. Methods of background calibration can be analog or digital and have a variety of implementations.

Bootstrapped Digital Calibration

The bootstrapped digital calibration scheme is one of the famous calibration methods as it can reduce the calibration convergence time [24],[5]. In this case, the ADC is utilized to calibrate the DAC and vice verse. Bootstrapped digital calibration includes analog circuits in the part of the calibration process to more accurately track the voltage and current samples. The addition of these analog circuits increases the overall power consumption of the ADC.

An accurate, constant gain and signal dependent gain are required for bootstrapped calibration [5]. These two gains are known, however, so an initial estimate of the gain values is required. The estimation of the constant gain is then updated 1024 times depending on the measured positive and negative thresholds of the residue characteristic curve. After updating the constant gain estimate, we need to update the signal dependent gain. To update the signal dependent gain, the linear and nonlinear ADC transfer characteristics are used. The signal dependent gain is updated 256 times. The number of update times, 1024 and 256 are selected analytically, but they are only the minimum number of times required for convergence [5]. As the two gain values are repeatedly fed into the ADC and DAC of each stage, the gains are constantly being updated, and eventually these two values will

converge, resulting in successful calibration.

1.5-bit Stage ADC Architecture

A 1.5-bit stage architecture uses two approximately symmetrical analog voltage levels to produce an implementation with increased bandwidth and redundancy between stages. The 1.5-bit stage pipeline ADC architecture achieves greater bandwidth by using a lower inter stage gain [23]. Due to this low gain requirement, we can realize low cost production and higher speed. Each stage generates an output of two bits in which the bits can only have the value of 00, 01, and 10. The output is determined by comparators at two symmetrical decision levels that make up a sub-ADC block of a pipeline architecture. Because of the following gain of 2, these two levels must be within the range of $\pm\frac{V_{REF}}{2}$, where $\pm V_{REF}$ are the maximum and minimum values of the signal. The choice of these reference value is not highly critical in the design, but because they must lie within the range of $\pm\frac{V_{REF}}{2}$ the decision values are often chosen to be $\pm\frac{V_{REF}}{4}$. These decision levels are designated as +1, 0, and -1 and are used in an implementation called a Redundant Signed Digit (RDS). The redundancy comes from the 0.5 bit overlap between stages. When the stages are summed, the carryover from the previous stage creates a redundancy and error correction.

Redundancy

Similar to the concept of 1.5-bit stages in pipeline ADCs, redundancy is used in flash ADCs. Instead of having 2 to the N -1 comparators, 2 times this or more is used.

An example is in [16] where more comparators are used as a means to have a better chance to have a more correct trip point. Reassignment is done to pick the best comparator and only the best trip point it used and the other comparators are turned off. A graphic showing how the redundancy is used is shown below in Figure 2.16. This is done with foreground calibration. It has advantages in power savings from turning off the unused comparators but in a sense wastes the die area because of that.

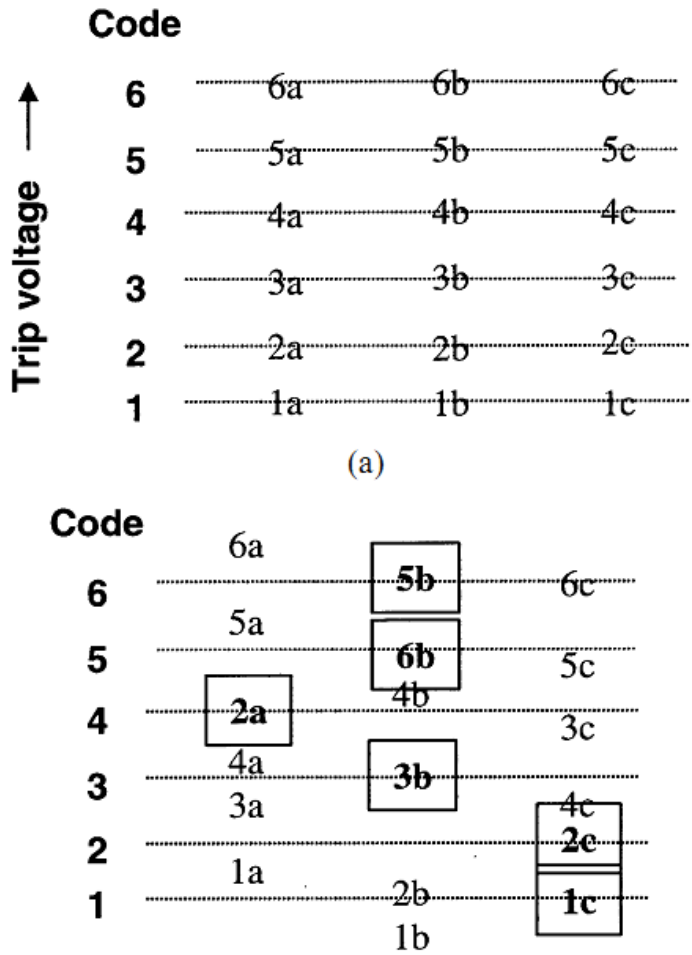

Figure 2.16: Pick Best Comparator Correction

Stochastic

Another way to make use of redundancy for calibration is to have a very large number of trip points and no reference ladder. If not reference ladder is used to set trip points and digital circuits at minimum size are used, trip points can be randomly distributed. To make this work two groups of these comparators are used, each with a shift from zero, the sum can create a largely linear region. Figure 2.17 illustrates this concept. A shift means instead of using a inverter with a trip point in the middle of the two rails, use one slightly larger than mid scale and one lower. This is a very interesting way of using all the information

from the redundancy used, with no digital calibration algorithm or analog changes. [21]

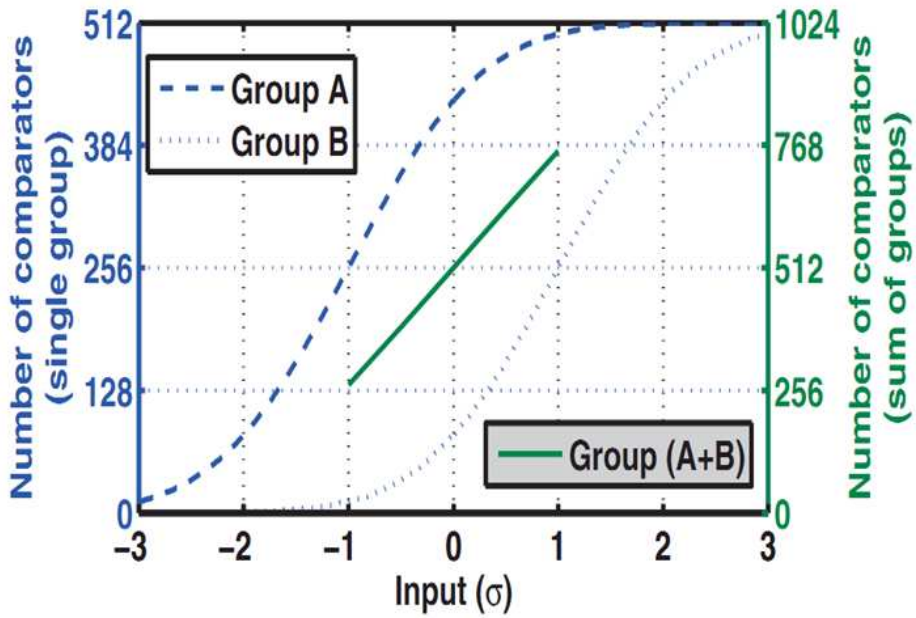

Figure 2.17: Stochastic Flash

Murmann's residue gain error correction

Murmann's residue gain error correction calibration method starts with adding a logic block to the output of the sub ADC block in each stage of the pipeline ADC [10]. This logic block provides two different residue characteristics that generate Figure 2.18.

The distance between one residue plot versus the other can show the linearity of the ADC. In this case, h_1 represents the ideal distance while h_2 represents the nonlinear distance. The goal is to apply an adaptive routine to correct the error between h_1 and h_2 such that error will converge to zero.

First, the probability density function of the residue characteristic is calculated to estimate h_1 and h_2 by using a random number generator. With the estimations of h_1 and h_2, we can calculate the error. Then, the LMS algorithm is applied to force the error to zero.

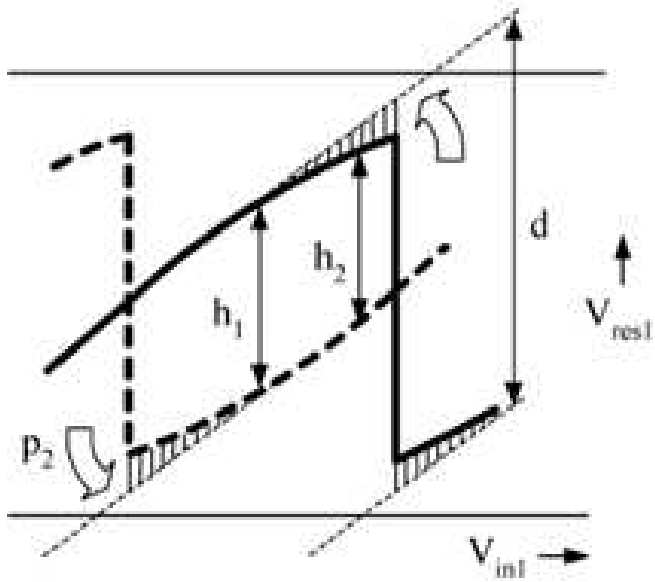

Figure 2.18: Difference Between Two Residue Characteristics

Once this is achieved that one can adjust the parameters p_1 and p_2 to force the output of Stage 1 and the backend stage to be linear. The main advantage of this technique is that it can achieve low power consumption.

Split ADC Architecture

The split ADC architecture is known for being able to calibrate residue gain error over a short period of time [2]. It can also digitally correct DAC errors in pipeline ADCs. In the split ADC architecture, there are two ADCs with the same resolution. The only difference between them is the residue transfer characteristic. Those two ADCs are placed in parallel and are applied with the same input signal. The following diagram shows how the split ADC architecture is used in a pipeline ADC.

As shown in the Figure 2.20, the same V_{in} is input into the two split ADCs, ADC A and ADC B. However, the outputs of the two ADCs are different for every input due to their different residue transfer functions. The difference between the outputs of those two ADCs is the error of the residue amplifier. Using this difference, the adaptive error cancellation

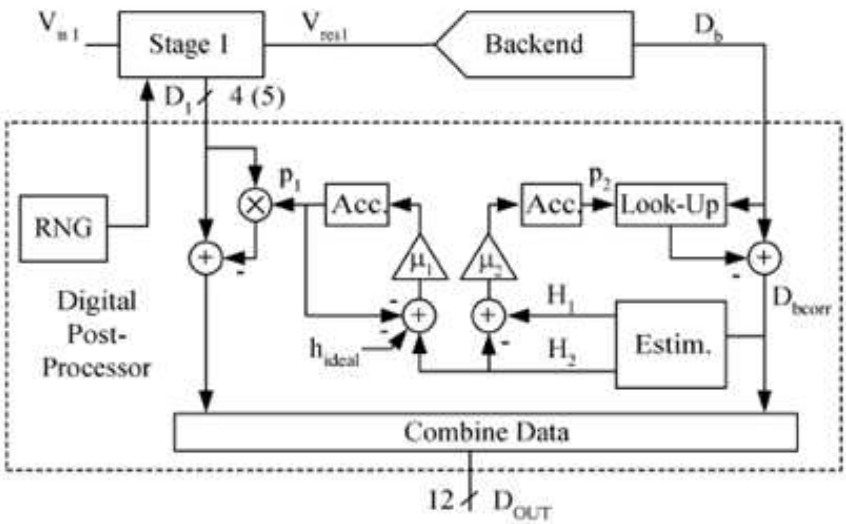

Figure 2.19: Error correction algorithms by using PDF

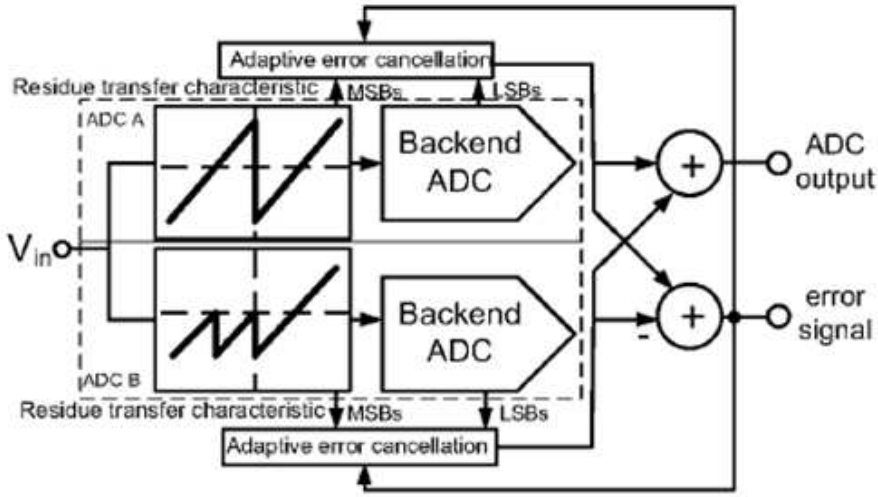

Figure 2.20: Split-ADC utilized in a pipeline ADC

can be processed to correct the residue amplifier gain error. Then the outputs from each of the adaptive error cancellation block are added to get the final output of the pipeline ADC.

In [2], to implement a 12-bit pipeline ADC, the authors incorporated two stages in each of the split ADCs in their design. The first stage consists of a 4-bit pipeline stage and

the second stage consists of a single 10-bit flash ADC. In this work, only the first stage is calibrated and the second stage does not need to be calibrated. Even though the goal is to implement a 12-bit ADC, they included two extra bits to achieve more accuracy in error correction. A different residue transfer characteristic in the two ADCs in the split ADC architecture can be acquired by offsetting one residue transfer characteristic curve with respect to the other [2]. Due to the residue amplifier gain errors, the slopes of back end codes will not be similar. Therefore, the difference between the outputs of two ADCs is not equal to zero. By using that difference we can adapt a corrective term to fix the residue amplifier errors, which would also calibrate the DAC's non linearity.

Another split pipeline ADC architecture is described in [14]. In this paper, similar to the previous design, the ADC is split into two identical ADCs, processing the same input but producing different outputs as shown in Figure 2.21.

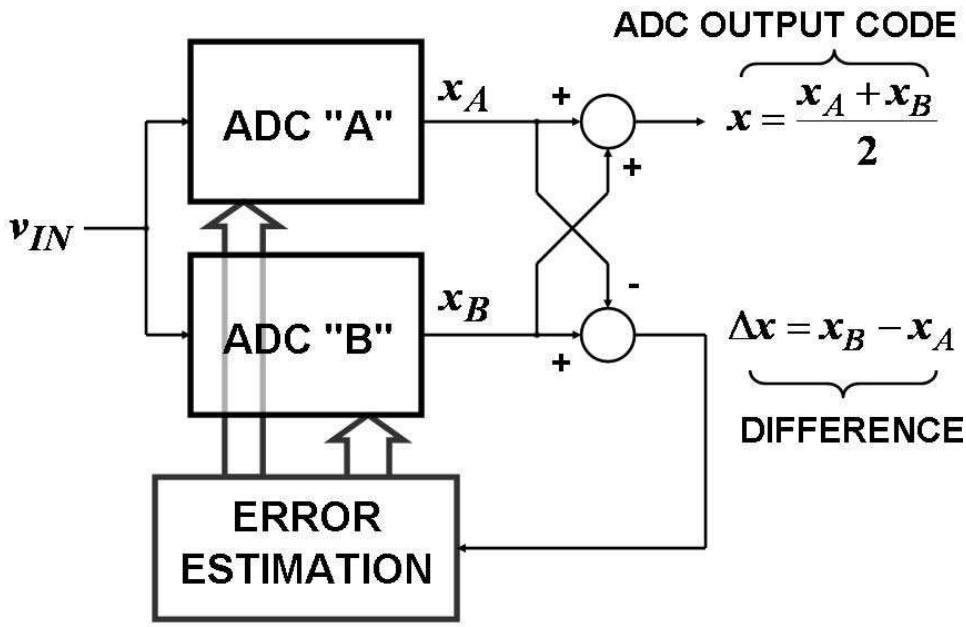

Figure 2.21: Split ADC Architecture

The average of the two outputs becomes the output of the ADC. The difference between the two outputs is used to calibrate the ADC. If the difference between two outputs is zero, there is no error and the ADC is calibrated perfectly. If the difference is nonzero, that

difference is used to adapt the error corrective term and update the calibration parameters in each ADC to achieve an error of zero. Finally, the advantage of the split architecture is its fast calibration convergence. [14]

The paper [21] with the stochastic ADC is an example of a split flash ADC. If we look at the figure below of the Split-ADC comparison chart, it can be seen that most of the ADC using this architecture are in the moderate speed, moderate accuracy range. There is a large gap in the high speed low resolution ADC.

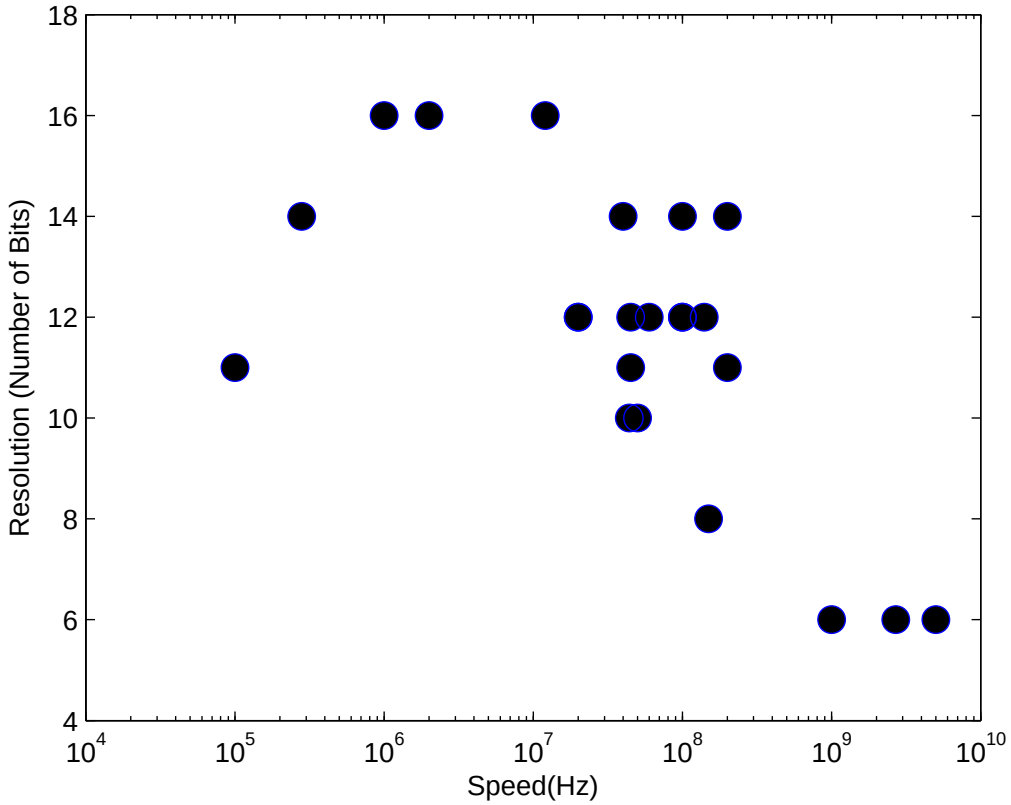

Figure 2.22: Split-ADC Comparision

2.5.3 Flash ADC Calibration

In flash ADCs, the importance of calibration can be shown by the following graph. It shows how many effective number of bits are improved in 6-bit flash ADCs. This sets

performance goals for calibration designs

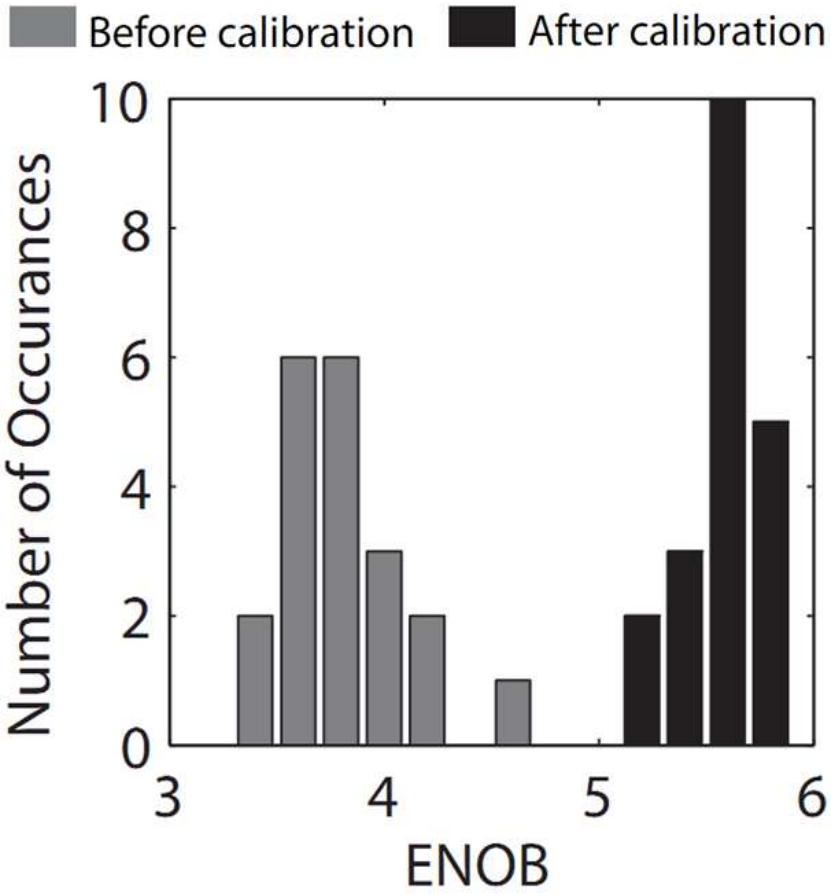

Figure 2.23: Flash ADC Calibration

Chapter 3

System Level Design

To perform a system level design we will define the specifications, provide a system block diagram and take a closer look at the blocks that need to be designed to implement the ADC and the calibration system.

3.1 Design Specifications

The analog to digital converter was designed to be part of a self-calibrating split-ADC. The basic specifications for the design are outlined in Table 3.1.

Specifications	
Circuit Type	Integrated Circuit
Maximum Size	$1mm^2$
Process Type	0.18um
Resolution	6 bits
Throughput	1 GS/s
Power	1mW
Other Specs	Fully Differential: $1V_{pp}$

Table 3.1: Flash ADC Specifications

This Flash ADC was designed for the 0.18um Jazz Semiconductor process. This process

allows for a 1.8V supply voltage. Because of the small amount of headroom allowed by this supply voltage, the signal swing the ADC can handle is $1V_{pp}$. This differential signal reduces second order distortion in the sample and hold circuit and doubles the input range from that possible in a single ended system.

The ADC will resolve up to 6-bit accuracy with a speed of 1GS/s. Since this is a flash ADC, it will be able to achieve a very high conversion rate. This is because there is no multiple stages and each result is obtained in a single set of comparators outputs. To achieve 6-bits of resolution the flash ADC must have 2^{N-1} comparators. This means the ADC needs 63 comparators. The output of these comparators then need to be summed to obtain the digital results.

Several aspects of the design reduce power consumption. Large flash ADCs consume high amounts of power since the number of comparators required roughly doubles for each additional bit of resolution. Since this flash ADC is mainly comparators. A large effort will put into minimizing the power of the comparator. The design of this flash ADC does not require a high accuracy output, so the comparator can be simplified and use small device sizes can be used, therefore minimizing the power consumed. Other options will be considered to minimize the power consumption.

3.2 System Block Diagram

The overall system block diagram of the ADC is shown below in Figure 3.1.

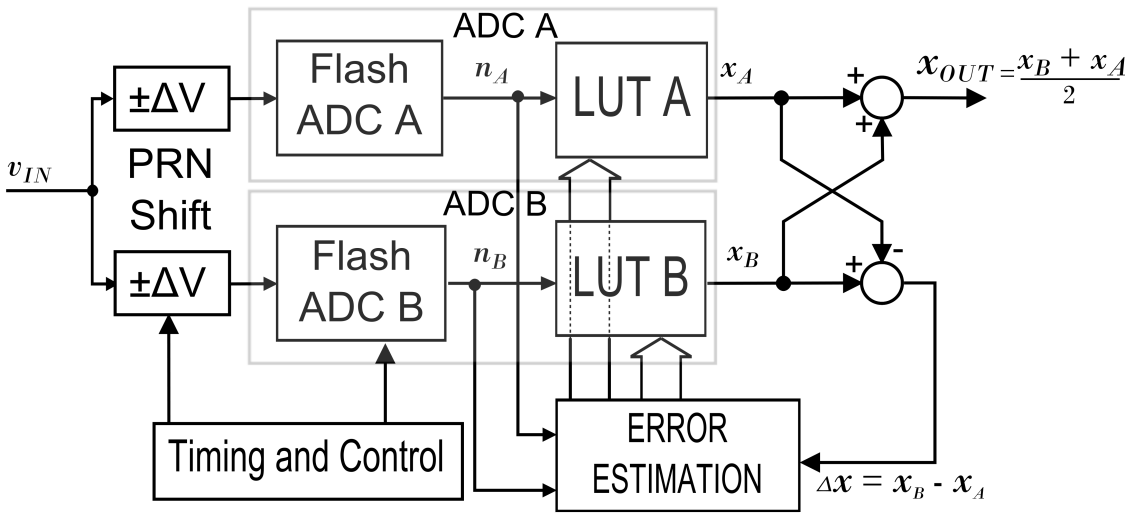

Figure 3.1: System Block Diagram

This ADC was implemented for use in a split architecture that includes two separate flash ADCs. Something needs to be added to create an intentional difference for calibration purposes when both ADCs are incorrect. This is created by adding a shift at the input. The output of each stage is connected to the digital calibration block. The calibration algorithm feeds digital correction information back into the system based on the difference between the outputs of the two ADCs. The scope of this project included the design of the calibration system. Circuit designs will be proposed and briefly explored.

3.2.1 Detailed Block Diagram

The block diagram of the flash ADC is shown below in Figure 3.2.

Each flash ADC is made up of three parts as shown above; resistive ladder, comparators and a digital adder. The signal can be sampled by using a clocked comparator or clocking the digital adder. The resistor ladder creates a reference that is compared to the input signal. The ladder should create as as many references as there are comparators. This is done with 2^N number of resistors. After the values are compared to the input, the outputs of the comparators are counted by the adder. The output is then sent to digital correction block and the calibration block.

Not shown in the block diagram above are a few key blocks that are fundamental to circuit operation. These blocks are the bias circuitry for the comparator, the output drivers for each of the digital decisions and from the comparator to the digital, and a timing block to control the timing of the switches.

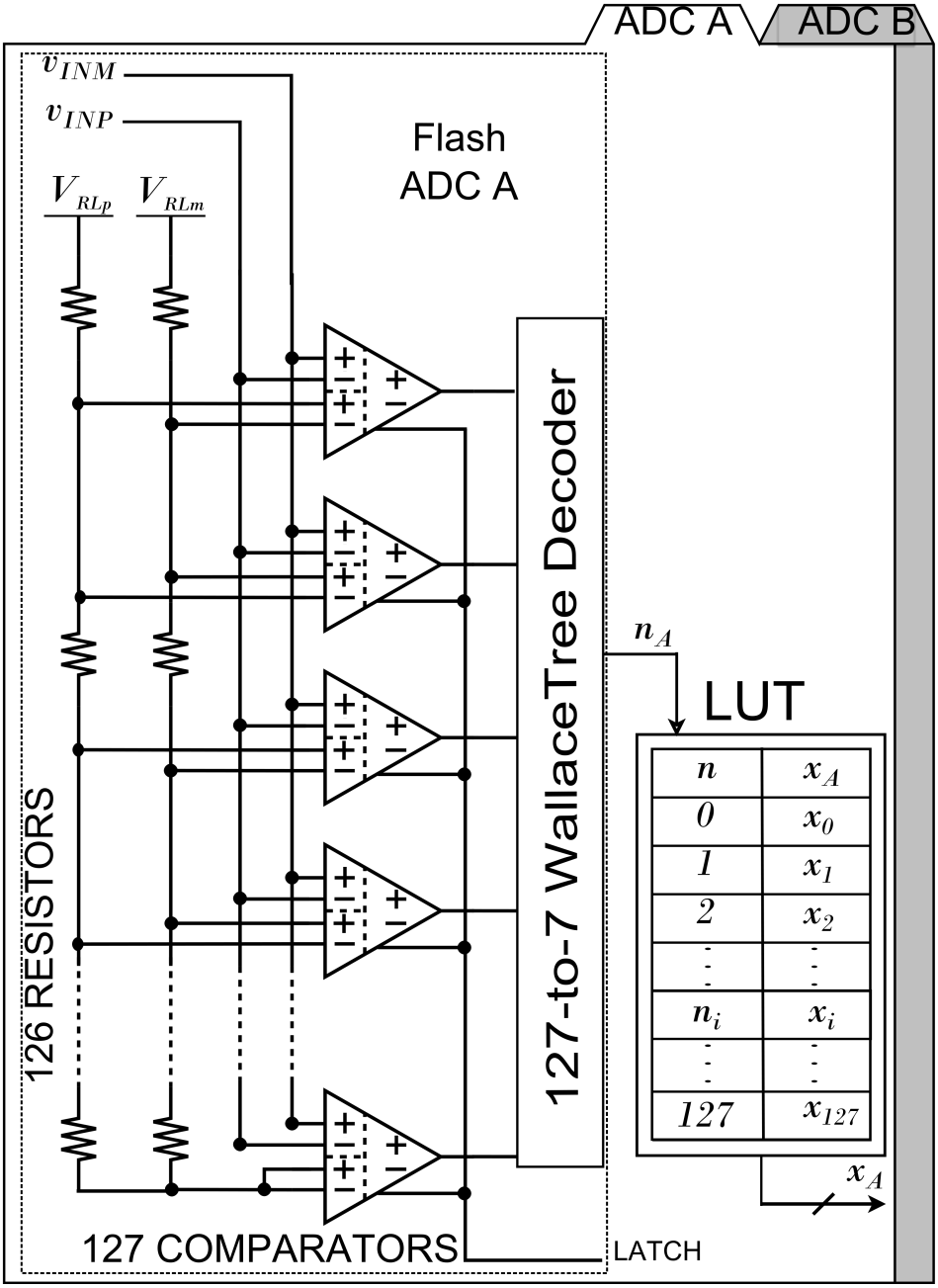

Figure 3.2: Detailed Block Diagram

Chapter 4

Correction

Analog to digital converters tend to need some sort of calibration in order to achieve a high number of effective bits and reduce it's errors. Calibration needs some way to correct the detected imperfections of the ADC. Depending on the calibration, different types of correction is used. This section will detail how correction can be done and how it will be done in this split flash ADC.

4.1 The Drive

As process sizes decreases, the need for calibration increases and therefore ways to correct the ADCs does too. One major topology to correct non-linearity in ADCs is redundancy. Just by adding twice the amount of comparators and averaging the two results can create better ADC performance. Just this concept validates the use of the split calibration since this averages two ADCs at the output. This will be verified in the calibration section.

4.2 Derivation

From looking at the ways other than averaging to use the redundant information, it was concluded to attempt to use a combination of the approaches. Our correction will use a 4X redundancy. This is the same as [16] but instead of only picking the best comparators, we will use the information from all of them to obtain a corrected output. Also similar to [21]

Raw (n)	Corrected(x_A)
1	x_0
2	x_1
2	x_2
.	.
.	.
.	.
127	x_{126}

Table 4.1: Look-Up table

we will use an intentional difference between the adcs. Instead of using this as a means to make the sum zero, we will use it as a reference for the calibration.

4.3 Look-up table correction

The topology that will be used to correct the output of our ADC is a look-up table. The look-up table will be used to match the raw output codes to corrected values determined by calibration. The calibration to determine the values will be discussed in the next chapter.

The raw codes are mapped to the corrected one by collecting information about each code and then using that code to index a table. The table has stored in it the value that should be used instead of the raw code. The implementation of this is similar to memory. Where the address is based on the raw output of the ADC and the corrected value is the number stored in memory.

4.3.1 Resolution of table

One major design challenge when using a look-up table is determining the size and resolution of the number used in the table. Some of it is decided in regards to the calibration type and the rest is a fight for area. If the flash ADC is small and there is room for a large resolution LUT. In the end there will be a trade of between size and accuracy.

There are ways to keep the resolution of the look-up table small. If you only update the table when an integer value should be changed you can keep the resolution of the table entries as big as the ADC itself. Determining when you should make a integer value change can be done with a simple counter. But if this counter needs to count to a large value and you need to make 2^N of these counters, it can take up a lot of space. A way to minimize that space and number of counters is by having one larger resolution counter and another look-up table that is storing the last number in the count. This should be considered for the system if space is a problem.

In previous split calibration designs a look-up table resolution was found optimal at 2^8. This was not verified but can be reviewed in a preview thesis based on a cyclic ADC. [22]

4.3.2 Ideal calibration with Look-up table

In order to look at the behavior of the LUT in terms of a flash ADC, an ideal calibration is implemented. This calibration is a simulated foreground calibration. It essentially inputs a ramp signal to detect where all the randomly distributed trip points are. The raw codes are then centered with the ideal code and the look up table is populated with that ideal code. This is difficult to even implement even in MATLAB. To do the centering we counted how many times this code was sampled and then found the middle. So if there was a count of 5 we would say the center was 2.5 and the number would be rounded up or down.

If we look at the results in Figure 4.1 the ideally calibrated flash ADC, we see that the DNL and INL are minimized. The goal of this work is try to approach this type of result without the use of MATLAB or a foreground approach such as a ramped input.

In order to see how well the LUT can correct even the worst flash ADC, a graph of the sigma deviation of a normal distribution vs the overall achieved effective number of bits. Figure 4.2 shows that even after a ten LSB of deviation on the simulated flash ADC, a 6.5 bits of ENOB can be achieved and 5.7bits truncated to 6bits. The simulation is of an 8-bit converter which is the same number of trip points as if you implement this calibration with 7-bits per side as planned. This is with a maximum resolution LUT and an ideal calibration. The calibration used will most likely not be able to come close to this. However it does say that even if the raw flash adc is very bad, a look-up table will be able to make it a much

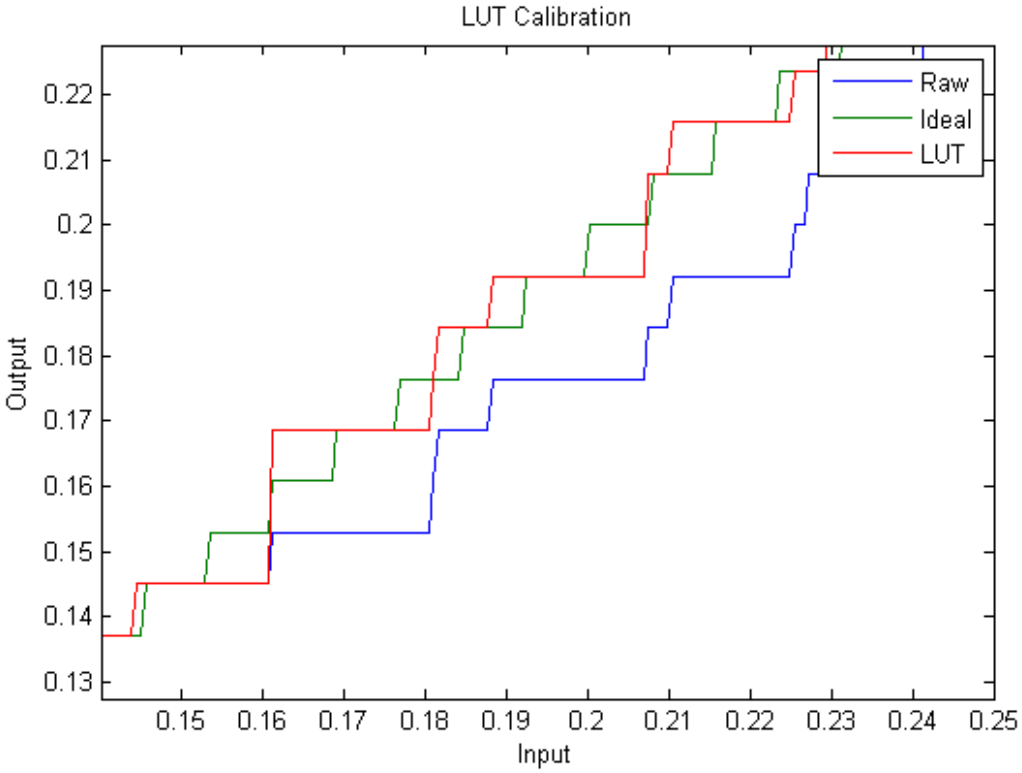

Figure 4.1: Ideal LUT Results

better converter.

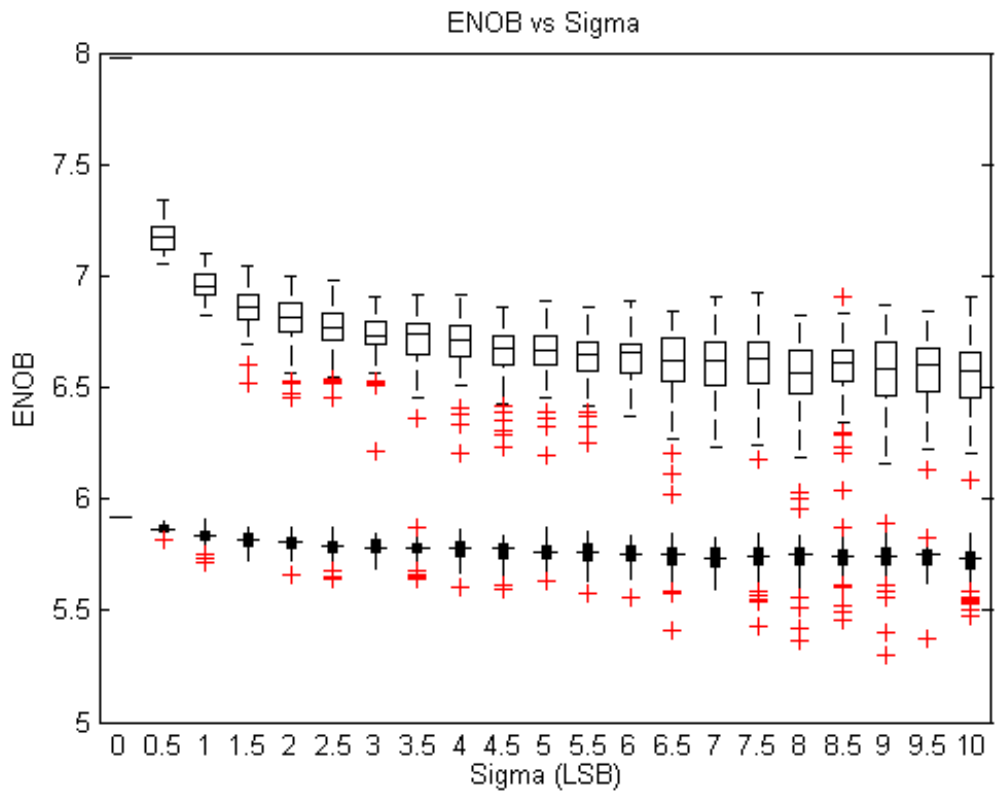

Figure 4.2: ENOB VS Sigma

Chapter 5

Calibration

5.1 Redundant Flash ADC

Figure 3.2 in the System level design showed a block diagram of the flash ADC designed for this work. Each of the "A" and "B" ADCs is composed of 127 comparators, for a redundancy factor [16, 9] of $R = 2$ compared with the $2^6 - 1$ comparators required for a 6b ADC with no redundancy. To tolerate non monotonic comparator outputs caused by large threshold variation, the raw digital output n is simply the number of comparators with a logic "high" output. Each of the n_A, n_B is realized with a Wallace tree decoder. To correct the DNL and INL errors due to threshold variation, the raw code n is used as the index to a LUT which provides the corrected output code x. In the ideal case, each entry x_i in the look-up table corresponds to the best fit code for the range of analog input voltages corresponding to each raw code n_i. Note that the digital precision of the x_i can be greater than the number of bits in ni to avoid quantization effects in correction and calibration.

5.2 Split ADC Structure

Figure 3.1 shows the split ADC concept [11, 12] applied to the design of this flash ADC. The ADC from Figure 3.2 is used for each of the "A" and "B" ADCs in Figure 3.1, for an overall redundancy factor of $R = 4$. The overall ADC output code x_{OUT} is the average of the individual output codes x_A and x_B . To enable background calibration, a small pseudo

random voltage shift $\pm\Delta V$ is introduced in the analog buffer at each ADC input. The $\pm\Delta V$ shift is derived from the ADC reference voltage, and for an ideal converter would cause a known shift in output code of $\pm\Delta C$. Since the $\pm\Delta V$ is equal in magnitude but opposite in sign for the two channels, the shift cancels in the averaging process and the output code x_{OUT} is unaffected.

As shown in Figure 3.1, the difference Δx between the x_A and x_B outputs provides information for the background calibration process. If both ADC look-up tables were calibrated correctly, the Δx would be equal to $\pm 2\Delta C$ LSB corresponding to the (known) shift ΔV which was introduced in each analog input. Any difference in Δx from the expected $\pm 2\Delta C$ LSB value provides information needed to update the x_A and x_B values in the LUTs corresponding to each of the n_A and n_B raw codes. As the input exercises the ADC inputs over their signal range, information is accumulated to calibrate the LUTs for all entries used. The advantage of using the split ADC is in the differencing operation, which removes the unknown input from the background calibration signal path [11, 12]. The following section describes the correction and calibration process in more detail.

5.3 System Overview

5.3.1 Digital Correction

To model the errors that need to be corrected and calibrated in this system, consider an example in which an input voltage is applied with a $-\Delta V$ shift in the A path and a $+\Delta V$ shift in B. Raw codes n_{iA} and n_{jB} from the A and B ADCs are mapped through the respective LUTs to produce corrected codes x_{iA} and x_{jB}:

$$\begin{aligned} n_{iA} &\xrightarrow{\text{LUT“A”}} x_{iA} = x - \Delta\, C + \epsilon_{iA} \\ n_{jB} &\xrightarrow{\text{LUT“B”}} x_{jB} = x + \Delta\, C + \epsilon_{jB} \end{aligned} \tag{5.1}$$

In (1), we model each of the x_{iA} and x_{jB} outputs as being composed of the ideal output x corresponding to the original unshifted analog input, the $\pm\Delta C$ code shift, and errors ε_{iA}

and ε_{jB} in the ith and jth locations of the A and B LUTs respectively. For the ADC output x_{OUT}, averaging the individual outputs in (1) gives

$$x_{OUT} = \frac{x_{iA} + x_{jB}}{2} = x + \frac{1}{2}(\varepsilon_{iA} + \varepsilon_{jB}) \tag{5.2}$$

As indicated earlier, the shift cancels and we are left with the ideal correct output x and an error component due to the errors in the LUTs. The calibration process to be described in the following section is an iterative procedure that drives the LUT errors ε_{iA} and ε_{jB} to zero, thereby ensuring accuracy of the digital output code x_{OUT}.

5.3.2 Calibration

There are several possible methods for obtaining the LUT used for correction. One possibility is to use a foreground approach of applying a known signal, using a ramp or DAC, and determining a best fit LUT for the outputs observed. As quality of the calibration signal is increased, the accuracy of the LUT can be made as precise as necessary. Disadvantages of this approach include the need to generate the calibration signal, as well as taking the ADC offline whenever calibration is required.

A novel aspect of this work is the background approach in which the errors are estimated iteratively. The background calibration accommodates any variations in comparator thresholds that may occur over time or temperature. The algorithm estimates the LUT errors based on the information provided by the difference of the outputs. Taking the difference of the outputs in (1) gives

$$\Delta x = x_{jB} - x_{iA} = \varepsilon_{jB} - \varepsilon_{iA} + 2\Delta C \tag{5.3}$$

From (3) we see that the (unknown) input signal is canceled from the calibration path, leaving only the known shift and the errors ε_{iA} and ε_{jB} we need to determine. To the extent that Δx differs from the target value of $\pm 2\Delta C$, we know there is a nonzero error in either or both of ε_{iA} and ε_{jB}. The purpose of the pseudo random analog shift is to provide additional information over multiple conversions that allows unambiguous determination of errors in the LUT. Without the shift, in the case of a DC input, there would be no way to

assign the error from the observed Δx to ε_{iA} or ε_{jB}. We can keep track of all errors in the A and B LUTs with 127-element vectors ε_{iA} and ε_{jB}B; with this notation we can write (3) as

$$\Delta x = \overbrace{[\overbrace{0 \ldots 0 \; -1 \ldots 0}^{\text{“}A\text{” } LUT} \vdots \overbrace{0 \ldots 0 \; +1 \ldots 0}^{\text{“}B\text{” } LUT}]}^{ASSIGNMENT \;\; \hat{W}} \overbrace{\begin{bmatrix} \varepsilon_{0A} \\ \vdots \\ \varepsilon_{iA} \\ - \\ \varepsilon_{0B} \\ \vdots \\ \varepsilon_{jB} \end{bmatrix}}^{\hat{e}} + 2\Delta C \tag{5.4}$$

The assignment vector has a -1 entry corresponding to the ith location in the A LUT, and a +1 entry for the jth location in the B LUT. Over many conversions, we can accumulate a matrix of information relating the Δx values to codes in the LUTs:

$$\overbrace{\begin{bmatrix} \vdots \\ \Delta x \\ \vdots \end{bmatrix}}^{\hat{d}} = \overbrace{\begin{bmatrix} 0 \ldots -1 \ldots 0 \vdots 0 \ldots +1 \ldots 0 \\ 0 \; -1 \, 0 \ldots 0 \vdots 0 \ldots 0 \; +1 \; 0 \\ \vdots \qquad\qquad \vdots \\ 0 \ldots -1 \ldots 0 \vdots 0 \ldots +1 \ldots 0 \end{bmatrix}}^{\hat{W}} \overbrace{\begin{bmatrix} \varepsilon_{0A} \\ \vdots \\ \varepsilon_{iA} \\ - \\ \varepsilon_{0B} \\ \vdots \\ \varepsilon_{jB} \end{bmatrix}}^{\hat{e}} + \overbrace{\begin{bmatrix} \vdots \\ 2\Delta C \\ \vdots \end{bmatrix}}^{\hat{s}} \tag{5.5}$$

Rather than solve the matrix equation in (5) exactly, the iterative technique in [7] is used.

5.3.3 Analog Shift

The analog shift is implemented as shown in Figure 5.1 using a source follower structure biased by current sources. The ΔIS current which is added to one of the branches of the source follower provides the appropriate voltage shift. The shift need not be instantaneous

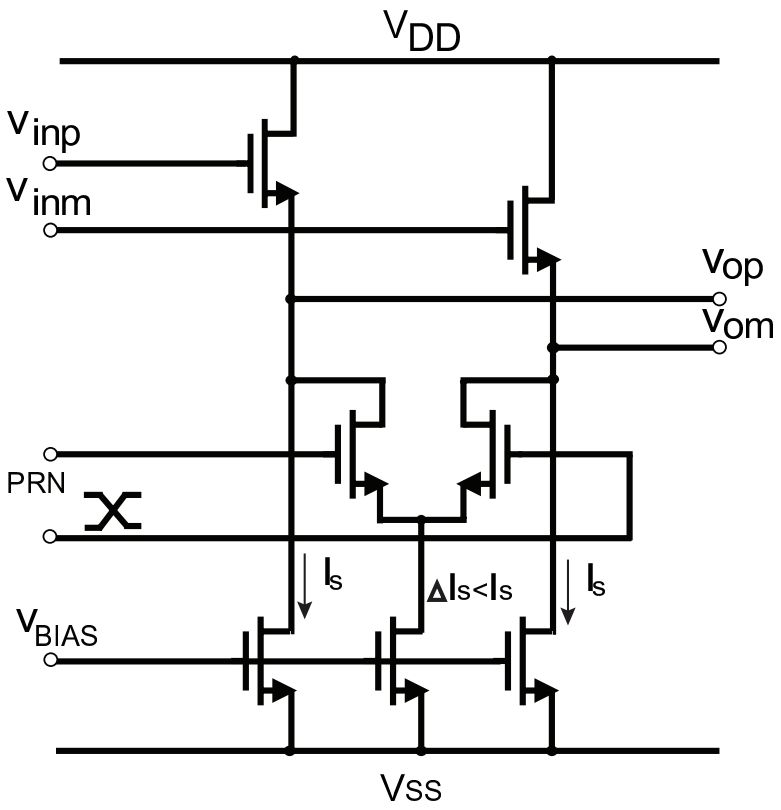

Figure 5.1: Analog Shift Circuit

as long as it is symmetric; samples from the transition region when ΔV has not reached its full value are discarded from the calibration data. The size of the ΔV shift is subject to an optimization tradeoff: too large a shift consumes excessive signal range, while too small a shift does not provide sufficient information for calibration. Numerical simulations show acceptable performance with a shift corresponding to a ΔC of 2-4 LSBs.

Another way to implement the shift is to change the voltage rails on the reference ladder. This is seen the the following paper [19]. This paper was found after completion of our algorithm but provides great motivation for a few improvements to our design. To implement the rail voltages changing, one possible way is to make two different rails. That way you can switch each ADC to a different rail creating that difference between them. At all times only both rails would be in use and there would not be a time where one rail would have to handle the load of two flash ADCs. The switching in the rail may cause some noise

on the reference and might cause a comparator to switch when it is not intended to.

5.3.4 Error Estimation

The mathematical development proceeds as in [12]. Formally, beginning with $d = We+s$ in (5), we subtract s from each side and premultiply by the transpose of W to obtain

$$\hat{W}^T(\hat{d} - \hat{s}) = \hat{W}^T\hat{W}\hat{e} \tag{5.6}$$

Since W is a very sparse matrix filled with only ± 1 for nonzero values, the product of $\hat{W}^T\hat{W}$ results in a diagonally dominant square matrix. If the matrix were purely diagonal, then its inverse would be easy to compute exactly as the inverse of the diagonal elements. Since, as in [12], we only need an approximate solution for the iterative least mean squares (LMS) procedure, we multiply by a factor μ to obtain estimates of the LUT errors:

$$\hat{e} = \mu\hat{W}^T(\hat{d} - \hat{s}) \tag{5.7}$$

The LMS factor μ is chosen to be a power of 2 so (7) can be easily computed as a shift in the digital hardware. The choice of μ also affects the dynamics of the iteration convergence; for stable convergence μ should be chosen smaller than the inverse of the largest diagonal element of $\hat{W}^T\hat{W}$. The $\hat{W}^T(\hat{d} - \hat{s})$ data can be accumulated on a conversion-by-conversion basis and requires the same number of memory locations as the vector. A block diagram of the calibration algorithm is shown in Figure 5.2. The calibration portion on the left side is performed after the system collects a set of data over a large number (of order 1000s) of conversion cycles.

5.4 Simulation Results

The full split ADC system was simulated behaviorally using MATLAB with the system parameters shown in Table 5.1. All results are reported at the 6b level. The comparator threshold variation value σ was estimated from circuit-level simulation and process specifications.

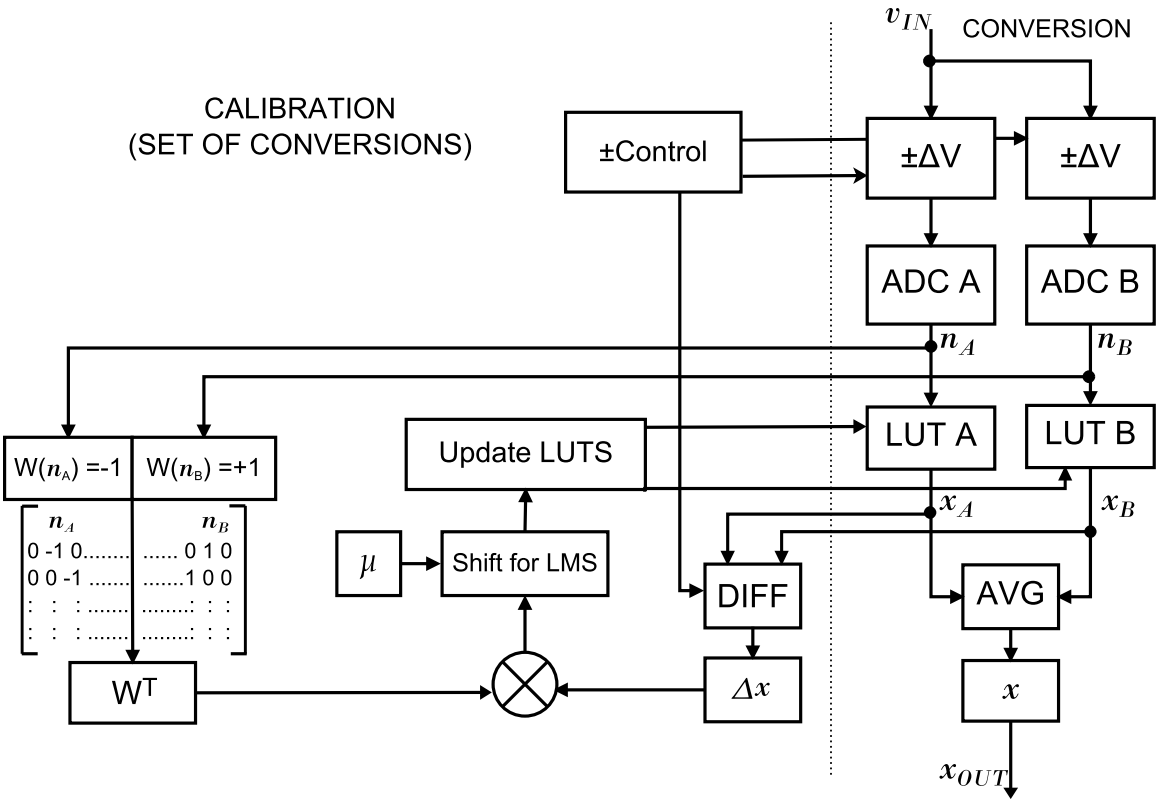

Figure 5.2: Calibration Block Diagram

Figures 5.4 and 5.3 show ADC differential non linearity (DNL) and integral non linearity (INL) of the system before and after calibration. DNL improves from $+1.53/-1.00$ to $+.85/-.90$ LSB; INL improves from 2.56 to 1.52 LSB pk-pk. The adaptation transient of the ADC for different μ values is shown in Figure 5.5. So that the detailed performance of the calibration algorithm can be seen, corrected code outputs are reported in 12b precision rather than truncated to 6 bits. For the $\mu = 2^{-21}$ case, convergence to ENOB > 6 is seen within 2E+9 conversions. At 1GSps, this corresponds to less than 2 seconds to converge to what would be quantization-limited accuracy. As is typical of LMS systems, faster convergence is seen for smaller μ, subject to stability and accuracy tradeoffs.

To find the ENOB numbers shown in the graphs above, the signal to noise and distortion ratio (SINAD) was calculated. The SINAD was calculated by taking the FFT of a signal out of the corrected ADC and taking the ratio of the value of the signal to the noise and

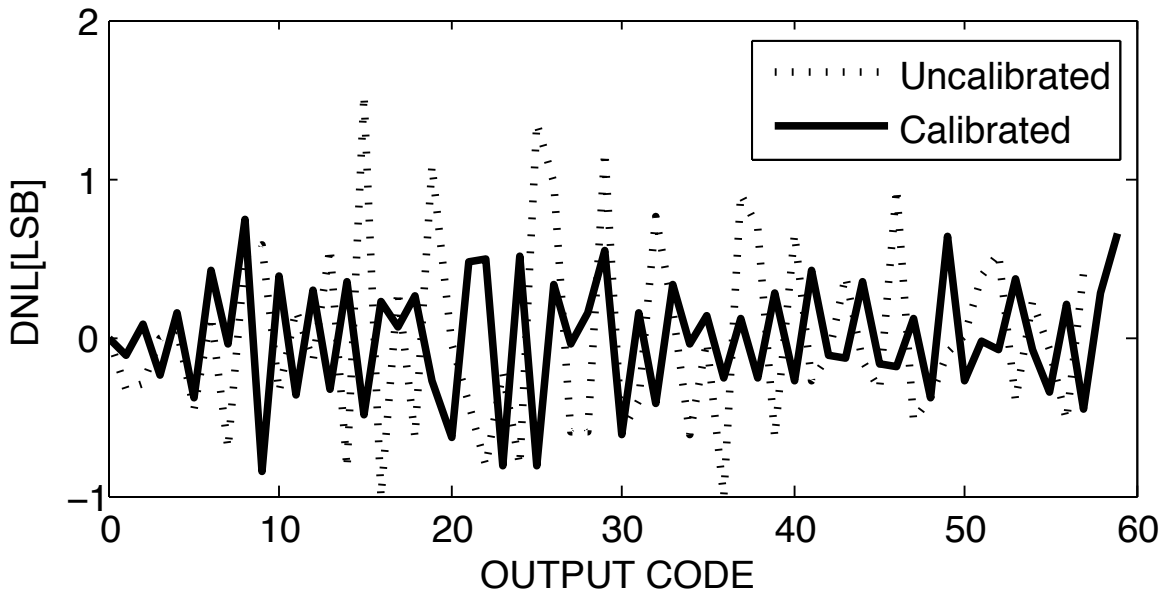

Figure 5.3: Calibrated and Uncalibrated DNL

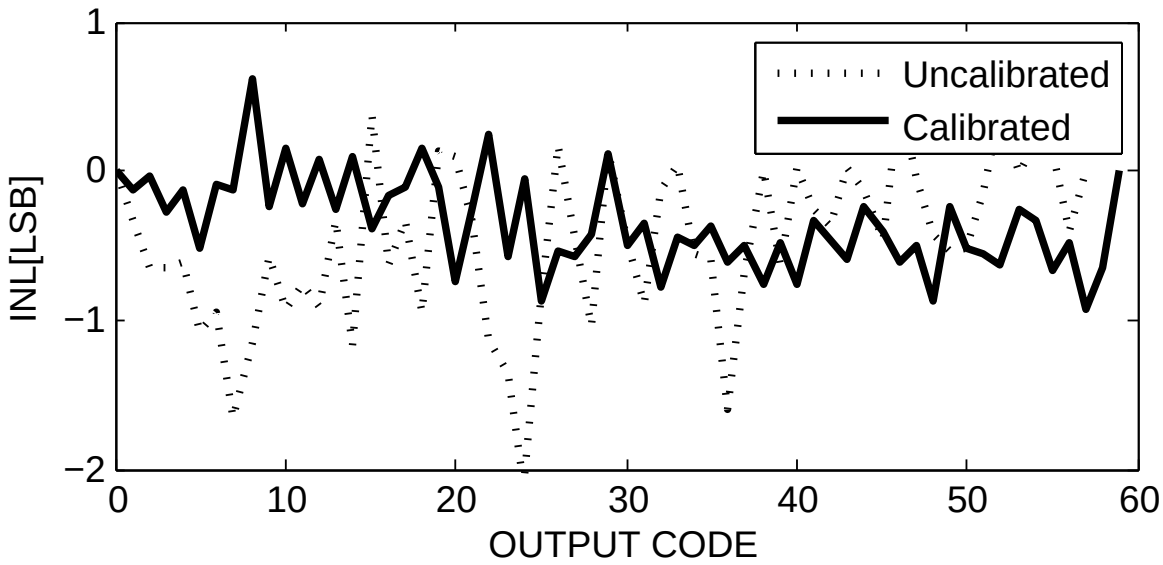

Figure 5.4: Calibrated and Uncalibrated INL

distortion.

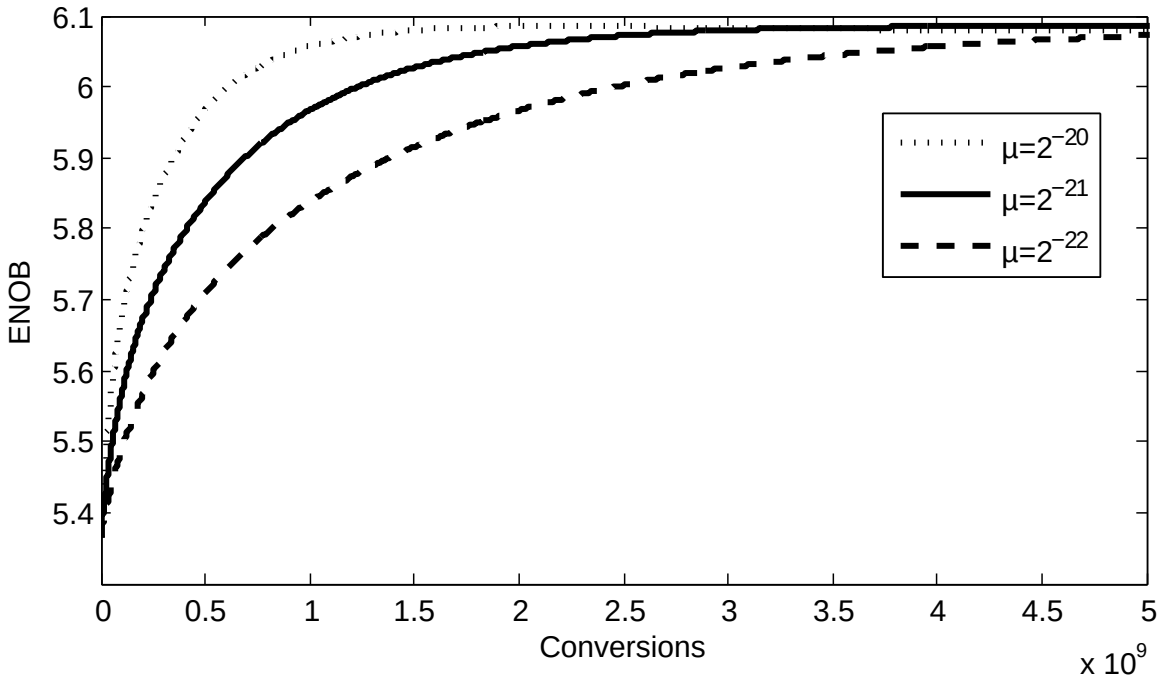

Figure 5.5: Calibration Convergence

5.4.1 Shift Values

The intentional difference created at the input of our calibration is a critical variable for design. This value determines how many bits apart the two inputs should be. If we have shift value very high, we can obtain more information about the ADC at DC or with a slow changing input. With a small shift you will have less information at DC and in general. During simulation of this calibration, it was seen that a larger value also had disadvantages. This is because at really large or really small input values, you will end up saturated the output to the LSB or the MSB. This will make some of your information useless and can corrupt you LSB and MSB of your ADC. There is also some dependency on your ADC error when picking your shift value. Larger shift values tend to work better with a larger distribution of trip points.

Figure 5.6 above shows different calibration convergence with different shift values. There is clearly an optimum solution show at 1.5 LSB. The lower shift value takes much longer to change the look-up table but at least never seems to start degrading the corrected

Table 5.1: System Simulation Parameters

PARAMETER		VALUE
LMS Parameter	μ	2^{-21}
Analog shift value	ΔV	3.5 LSBs
Intial Error Estimate	ε	0
Threshold variation standard deviation	σ	5 LSBs
Total Number of Comparators		254
Effective Number of Bits (ENOB)		6.1
INL(after calibration)		1.52 LSB pk-pk
DNL(after calibration)		$+.85/-.90$ LSB

values. The larger shifts do end up decreasing. This is most likely due to the over and under MSB/LSB threshold values. One interesting thing is that there seem so be some speed increase in calibration with a larger shift. This is due to the more information obtained per calibration cycle.

In order to reduce the effects of the saturated outputs, there was consideration of implementing some intelligence into the calibration. This was to try to determine when the converters output is not valid information. To do this, you could see if two samples resolved the same input but was not two times the shift value at the output, you can discard the information. Also you can set limits to the look-up-table and say that nothing can be above the MSB and nothing can be lower than the LSB. This is easily implemented and will be included in the calibration. Trying to determine when the converter was shifting to an out of ranged value, proved to be difficult without know the input. To see if it is worth pursuing this addition intelligence to the background calibration, a simulation was done to compare the limiting. The results of that are shown in Figure 5.7. If a small enough value of a shift is use, this detection is not needed but if a larger shift is desired, this would have to be done. One thing to consider, is that sense a larger shift makes the calibration go faster the simulation with the larger shift are starting to decrease the ENOB faster.

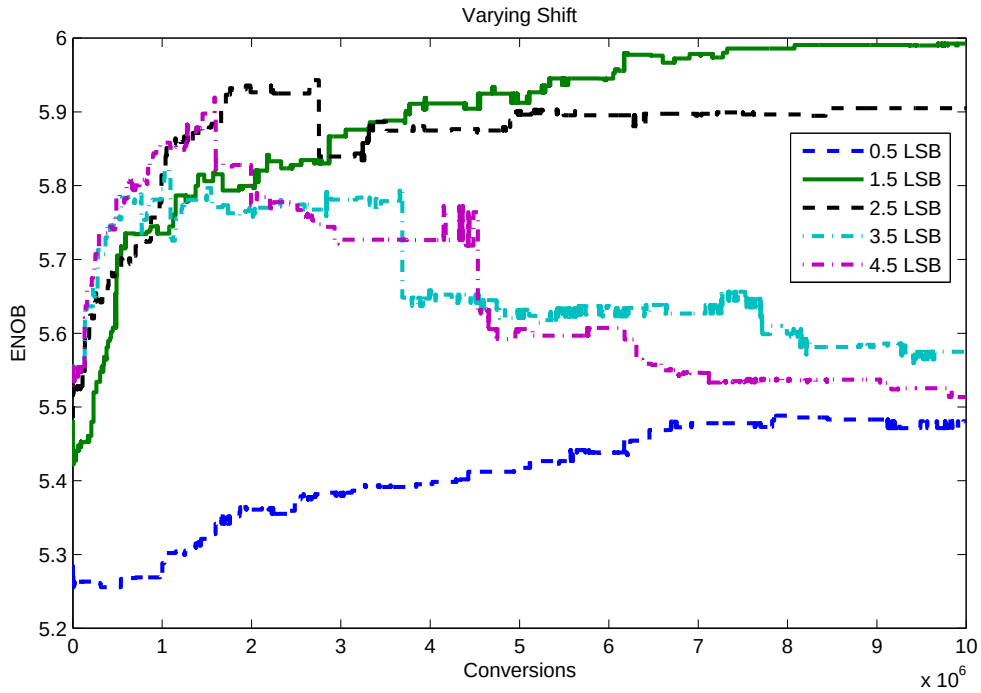

Figure 5.6: ENOB VS Shift Value

5.4.2 Calibration With Different inputs

A concern with the design of all calibrations is how well it does with different input signals. Though application can provide approximate input signal guidelines, a good calibration should handle all types. To verify this quality of the calibration, the calibration was done with three different inputs, random, sine and ramp input. Initial prediction of the results would be that the ramp would create the best look-up table because all values would be covered if the ramp was slow enough. It is also expect that a random input would take longer due to the fact that it may take time for all values to occur often enough to give calibration information. The expected results are verified in the plot in Figure 5.8. The plot shows that sine is in the middle of the two and that all depends on the frequency of the sine wave relative to the sampling frequency.

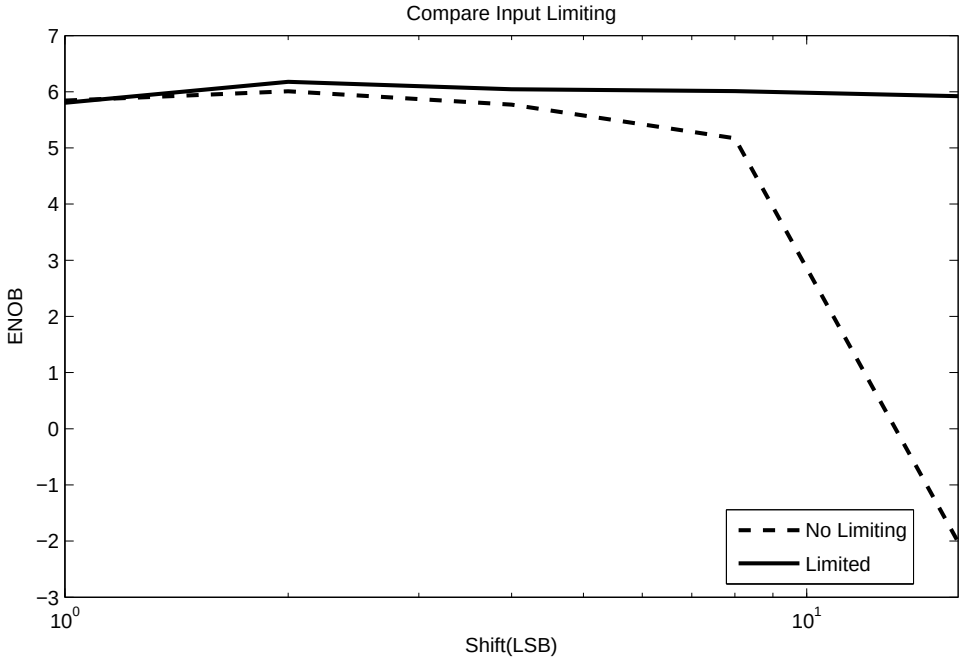

Figure 5.7: Comparing Ideal Limiting

5.4.3 LUT Truncation

Since most of the simulation for this calibration was done in MATLAB, floating point math was used. Though you can implement this in a digital system, the space you need is very large. Not to mention that the output of your ADC will be integer numbers. To avoid the floating point complications a simplification of the algorithm was thought of, and to see the effect of the integer number output, a simple rounding was done in MATLAB. Figure 5.9 shows a full precision output and a truncated output. There is about a 0.3 ENOB difference. So if the system can take in more than 6 bits, more information about the input can be found.

5.5 Calibration Simplification

The calibration outlined so far is developed in MATLAB using mathematical theory. In order to make the calibration more suited for an FPGA, a simplification needs to be

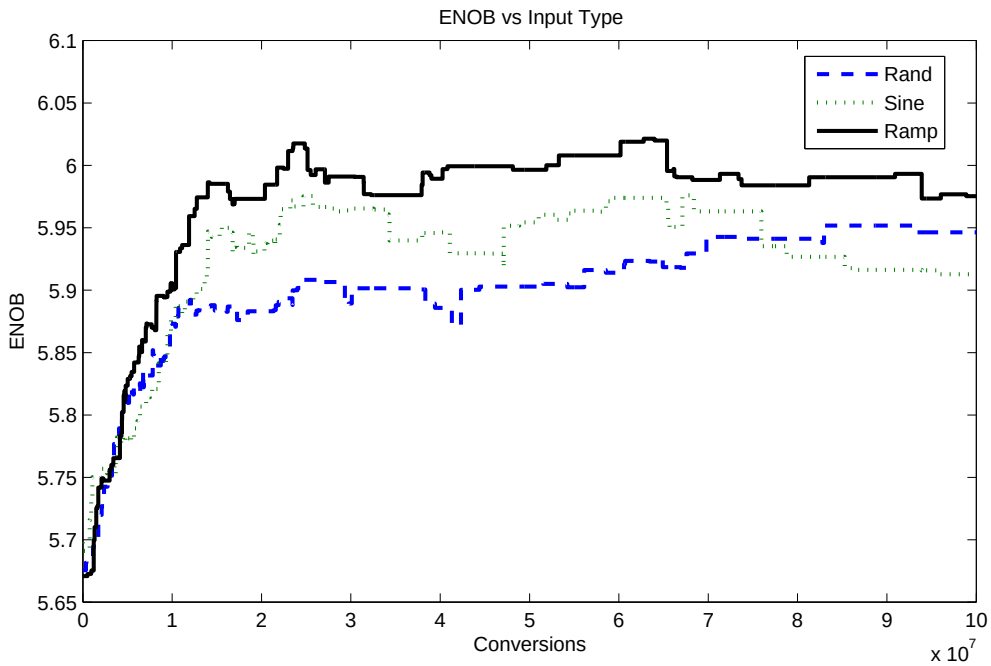

Figure 5.8: Comparing Different Input Signals

thought of and considered.

If we evaluate what is actually being done in this calibration we can determine a simplification. The easiest way of looking at the calibration is just collecting errors and applying a small fraction of them to the Look-up table. The matrix solution is just one way of collecting the information and really only adds additional computation that has to be done. If we just create a second table that just stores the sum of the errors for each raw code, we can take a small fraction of that table and apply it to the real look-up table. The block diagram of this simplification is show below in Figure 5.10. The summation block is where you'd accumulate the delta x data each conversion.

A way to keep the math fixed point integer math, you can accumulate the differences by incrementing a counter to a number proportional to the mu term. Once this counter reaches a curtain value, that location in the look up table is corrected. This may have some advantage because you wouldn't be doing your next conversions math based on the last conversions change to the LUT, which the matrix solution already does. Since the value of

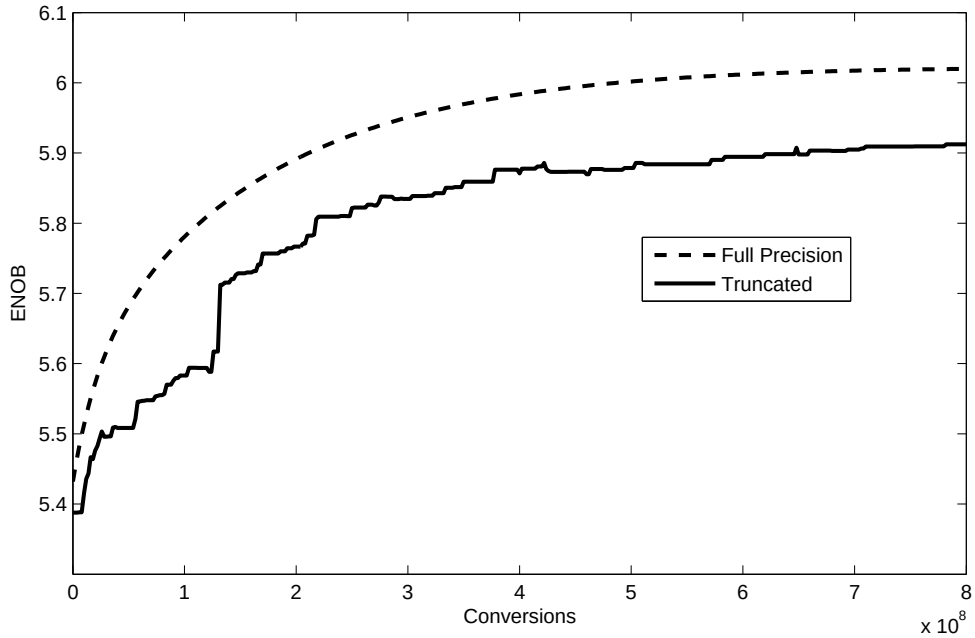

Figure 5.9: Truncated Signal

mu is very small, this is advantage hasn't been seen.

The results of the calibration are the same. The MATLAB code is simplified and porting the simplification to an FPGA is greatly simplified. So is the size of the circuitry needed to implement the calibration. The LUT will still be large but not changing the LUT for every error term used can also reduce the resolution of the LUT. The LUT is reduced by having a lower resolution but the counters may just be an even trade off. Since the FPGA generally is not that big this optimization may not be needed. But if a non FPGA version was to be implemented the counting idea could be used.

5.6 Resolution of the ADC

The goal of our design was to achieve a high resolution with small inaccurate circuits and a calibration algorithm to improve it. Since the simulation was not done with circuit simulation or silicon results, a random distribution of trip points were used. The calibration was done assuming a very high sigma variation and also was based off of prior art where

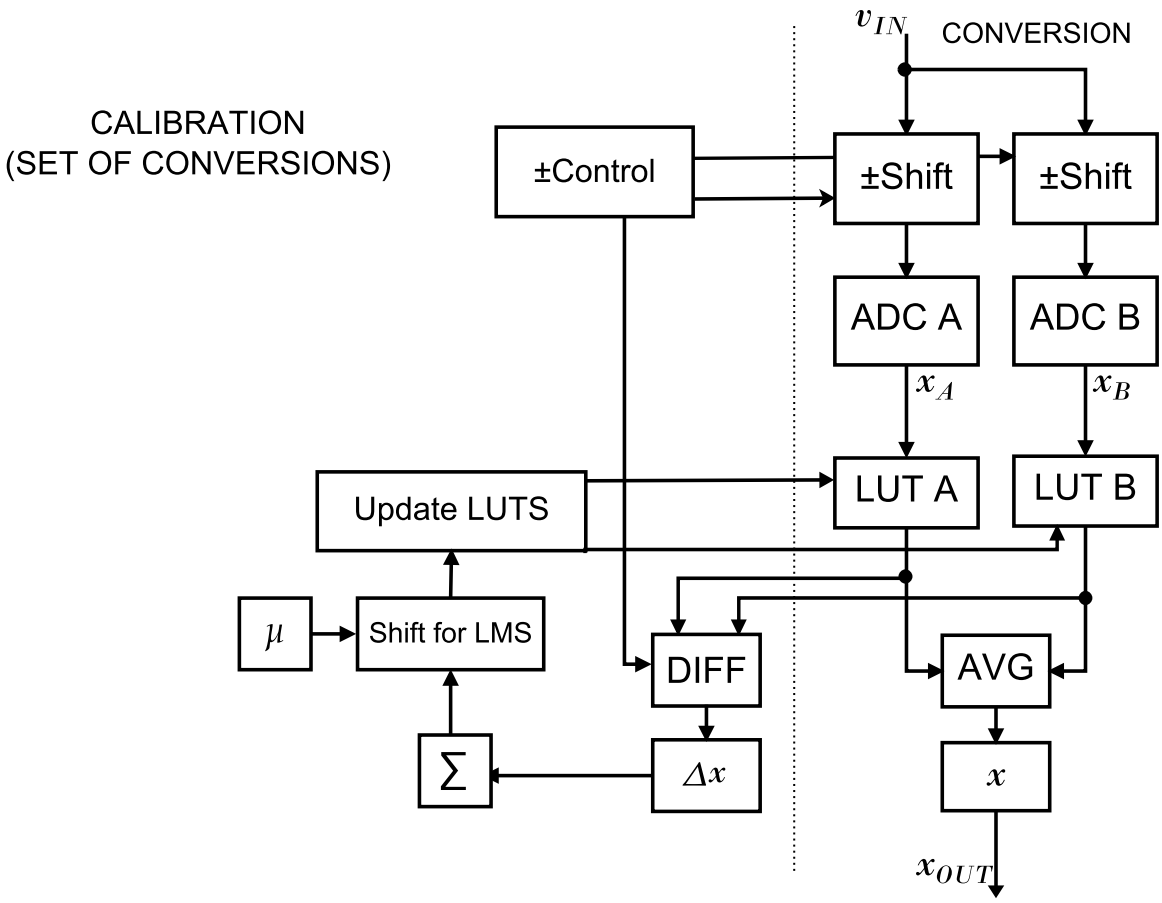

Figure 5.10: Simplified Calibration

4x redundancy was used. The question was posed, if our design is not that bad since we moved from a 45nm process to 180nm, can we get away with only 6-bits per side.

The simulation was recoded with this question in mind and the following results were obtained in Figure 5.11. It can be seen that the algorithm achieves 5.2 number of effective bits, where as with 7-bits we got 6.1bits. Obviously more comparators mean more trip points and more information to calibrate with. There are two interesting things to note on in Figure 5.11. First is the large improvement from averaging the two ADCs. If we predict what the outcome of the averaged ADC, you may think there should be an improvement of one ENOB. One important thing to consider is that, though there are enough comparators to make a 1 ENOB improvement, not all comparators will create additional useful trip points. Two 6-bit flashes with ENOB of 4.6 averages out to 5.2 ENOB. The total resolution

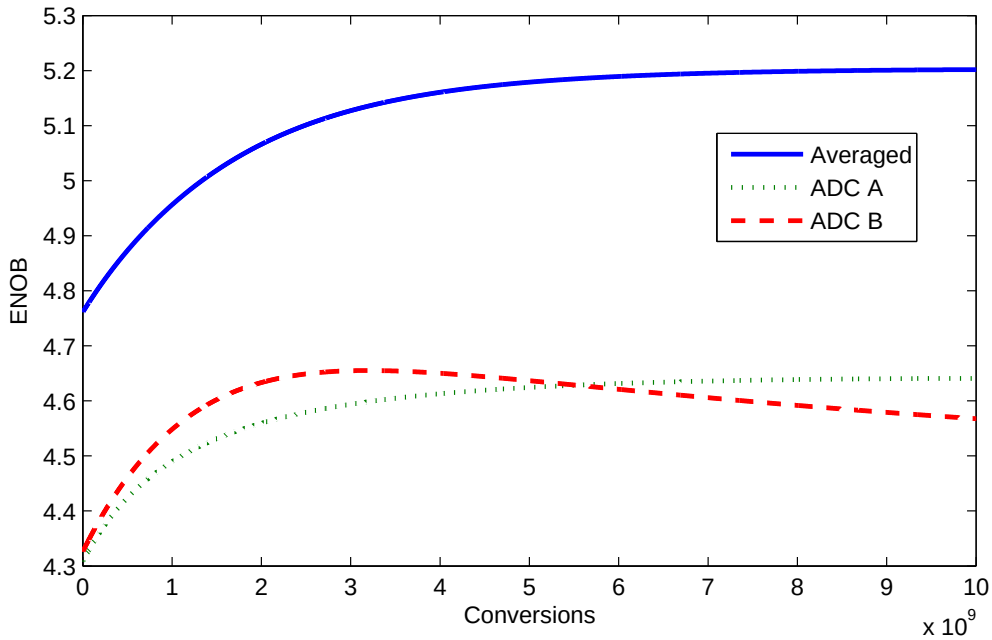

Figure 5.11: 6-bit Per Side Calibration

improvement is 0.9 ENOB, with two thirds of this coming from redundancy. The second thing to notice how ADC A does not converge. This means that there are improvements that can be made to this calibration setup for the simulation and might also mean corrections need to be made to calibration technique. This can also explain the need for the out of range limiting. Even more interesting about this plot is that even though the ADC A is decreasing and ADC B is not, overall the averaged output is not decreasing. So even thought ADC A is getting worse, it may still be helping to improve the overall ADC.

Chapter 6

Calibration Implementation

The simplification of the calibration was proven in MATLAB, and the algorithm was attempted in a FPGA. This portion of the project required a refresh on FPGA programming in VHDL/Verilog. It was important to research this in order to determine if it was possible to include this algorithm on chip and fully background. The more thought that was put into it, better, less FPGA required solutions were thought of. The code discussed here was not completely verified, but this serves as a starting point for a possible completely separate thesis project or as a starting point for someone to finish their investigation of the flash circuit implementation.

6.1 FPGA Implementation

The design of FPGA implementation was separated into a few parts not too dissimilar to how this thesis is presented. First there is the main section that declares variables, sets up clocks and instantiates the other parts of the code. The other blocks are calibration, correction, and RAM. The RAM is where the look-up table is stored. The calibration is where the difference, accumulation, and LMS operations are done. The correction block serves as the connection between the two, correcting the RAM values based on the calibration data.

The block diagram in Figure 6.1 illustrates the flow of the FPGA code and another way of looking at the simple approach to this calibration. Also included in the code in

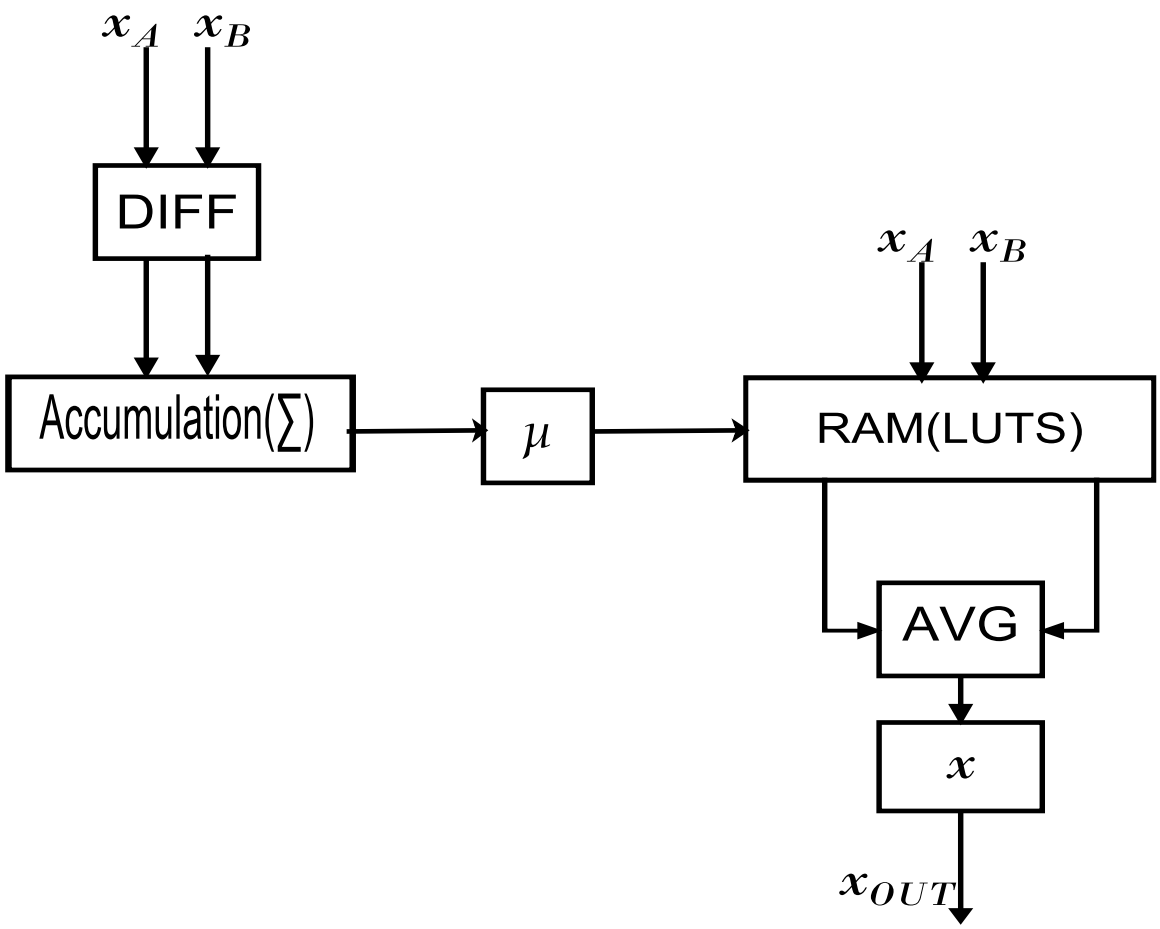

Figure 6.1: FPGA Block Diagram

the appendix is an attempt to use the FPGA input buffers as comparators. This will be discussed in the next chapter.

6.2 Synthesis

The FPGA code that was written does compile and synthesized. Test benches should be written to test the calibration and compare it to the MATLAB simulation, but going beyond the point of determining the feasibility of the calibration in a FPGA is beyond the scope of this thesis.

The number of four input LUTs used in this design was 335. There were 175 flip flops

used. This turned out to be a lot smaller than was originally predicted. It would have been interesting to synthesis this to the gate level and approximate the area that this calibration would consume on an IC.

6.3 No FPGA

After some thought about the complexity of the FPGA code written and the simplification discussed in the last section, more investigation was done to determine if the calibration could be done without an FPGA.

One portion of the circuit that is impossible to removed is the look-up table used for correction. The information has to be stored somewhere, but it may be possible to only store the difference between the raw answer and the corrected answer.

The idea behind having no FPGA is to take the raw output and find the difference of the two outputs and detect which is positive. If we add one to a counter for the channel that is larger, and subtract one from the counter that is negative we can accumulate the difference in a scaled fashion. One the things that is not every efficient or practical about the calibration is the resolution of the look up table to do the math accurately. This enables us to never depend on large resolution number which can make the FPGA size very big. One other difference in this new idea is that I would only be changing the look-up table when a major change is done do the channel and not just after a certain number of conversions. This has some advantages. This new method is explored in MATLAB to see the difference in results. There ideally should have no difference at the output.

To do this without an FPGA, there are a few circuits that will be needed. First we need an up/down counter. This counter will have to be big in order to simulate the mu value picked. But this technique might make not go slow enough to get good calibration results. This counter will count to a large number, once this value is reached it will send a bit high to another counter. This counter will be then what is used to input into an adder/subtractor with the raw codes. For doing the differencing we will need a digital comparator or big subtractor.

Chapter 7

Circuit Implementation

The design of the circuit for this ADC was not the focus of this masters thesis. This was covered in other student's in the labs Ph.D thesis and could be something I complete in the future. Compiled here are design considerations and possible designs that could be used.

7.1 Flash ADC

A typical flash ADC has a resistor ladder to set a reference for a comparator to determine if the input is larger than it or not. This leads to major parts, a references creator or control and device to compare the into to that reference. The typical implementation of a flash ADC is shown below in Figure 7.1.

7.2 Comparators

The comparator in our ADC does not need to be very accurate do to the calibration algorithm presented. Because of this, you can think of it's requirements as completely relaxed in terms of accuracy. This means that the speed and power can be optimized. The goal for this circuit is 1Gs/s so speed is a big consideration. Since the calibration algorithm can use additional comparators to achieve an great ENOB, power is something that should be minimized.

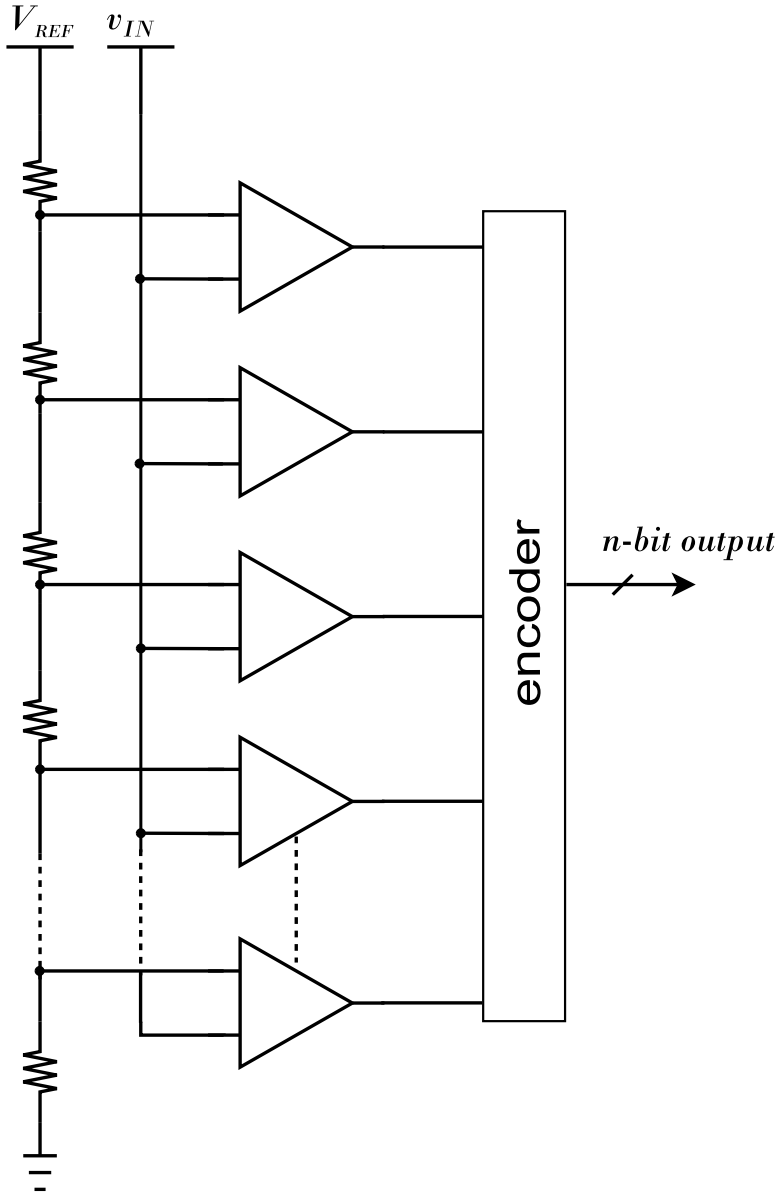

Figure 7.1: Basic Flash ADC

7.2.1 Comparator Theory

The most basic ideal comparator, shown in Figure 7.2a, compares two voltage signals and the output indicates whether V_1 is greater than V_2. The input-output characteristic of an ideal comparator can be seen in Figure 7.2b.

The comparator is comprised of a preamplifier and an analog latch as shown in Figure

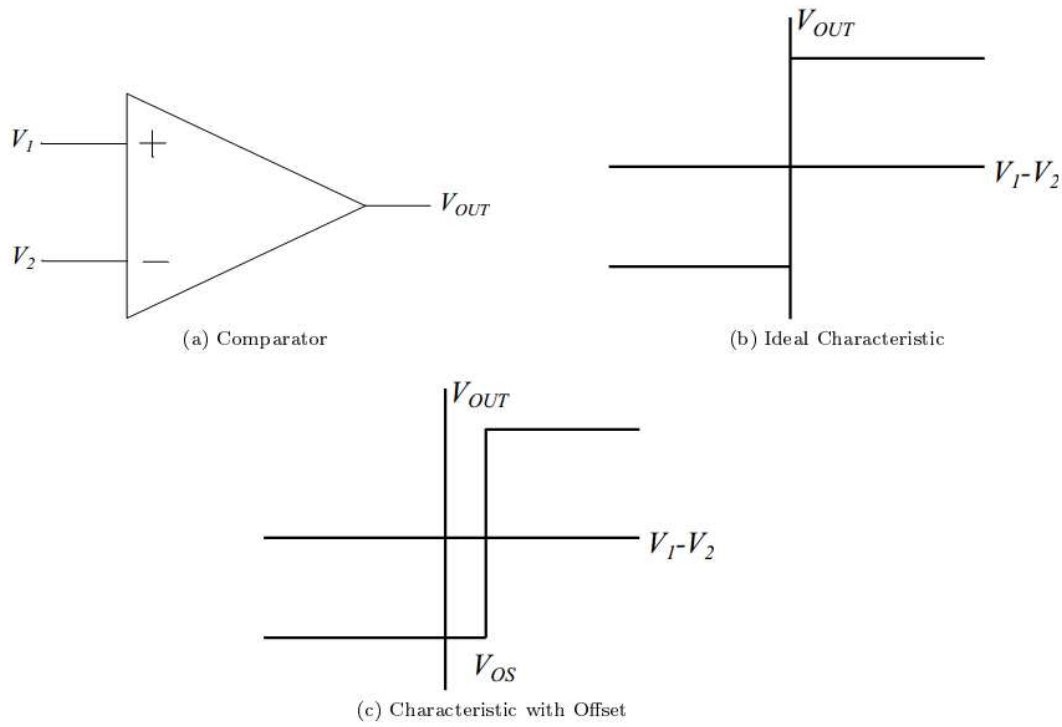

Figure 7.2: Ideal Comparator Characteristic

7.3.

Analog Latch

The analog latch drives its output to one rail or the other depending on the input it receives from the preamplifier. The basic latch is two connected inverters as shown in Figure 7.4a. This configuration will have the input output characteristic shown in Figure 7.4b. The points of stability and metastability occur at the intersection of the input-output curves of the two inverters. Figure 7.4b shows two stable points, one high (V_H) and one low (V_L), and a metastable point in the middle V_{Meta}. The difference between stability and metastability in a latch can be visualized like a ball balanced at the peak of a hill versus a ball at the foot of a hill. The ball at the foot of the hill needs to be moved up the hill to reach another point of stability or metastability, while the ball balanced on the peak requires only a small push to fall to a stable point. Likewise, to move from $V_L$to V_H or vice verse, a significant

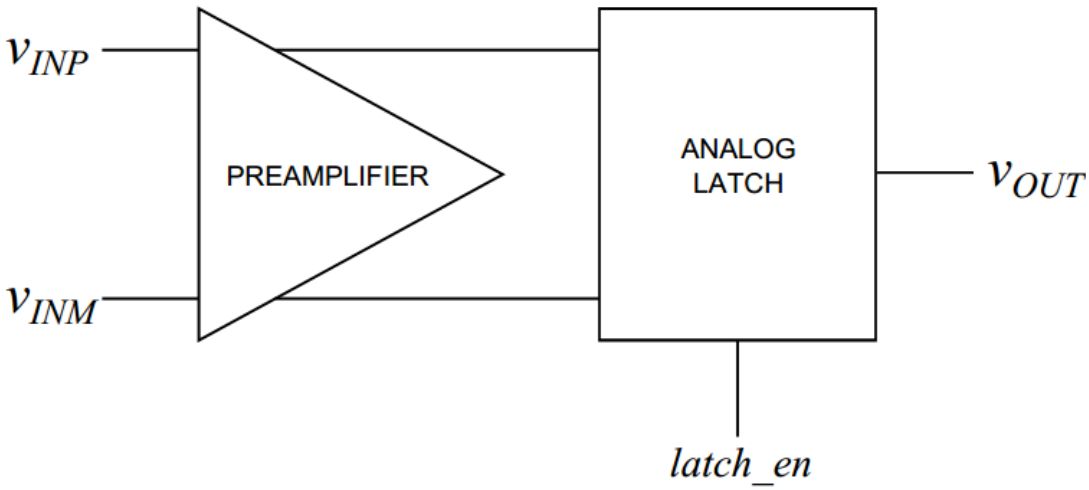

Figure 7.3: Comparator Block Diagram

change must be applied, however, only a small voltage variation is required to cause the metastable point to "fall" to one stable point or another. If $V_A$and V_B are shorted together with a switch as shown in Figure 7.4c, then the two voltages will both be pulled to the metastable point. Input switches are used to apply voltages that create an imbalance that will push the latch into one of its stable states. Once this imbalance is established, the switch shorting the latch into its metastable state is released and the latch output is driven either high or low.

Comparators are implemented a number of ways. The topology described above can be used as a model for most comparators. For this design we may want to use something simple that could be implemented easily in a small amount of space. A trend in very small processes is to use inverters as amplifiers, in this case as the preamplifier. Interesting enough, an inverter can also be thought of as comparator. The comparator should be clocked and differential so a basic inverter might need to be expanded to meet those needs.

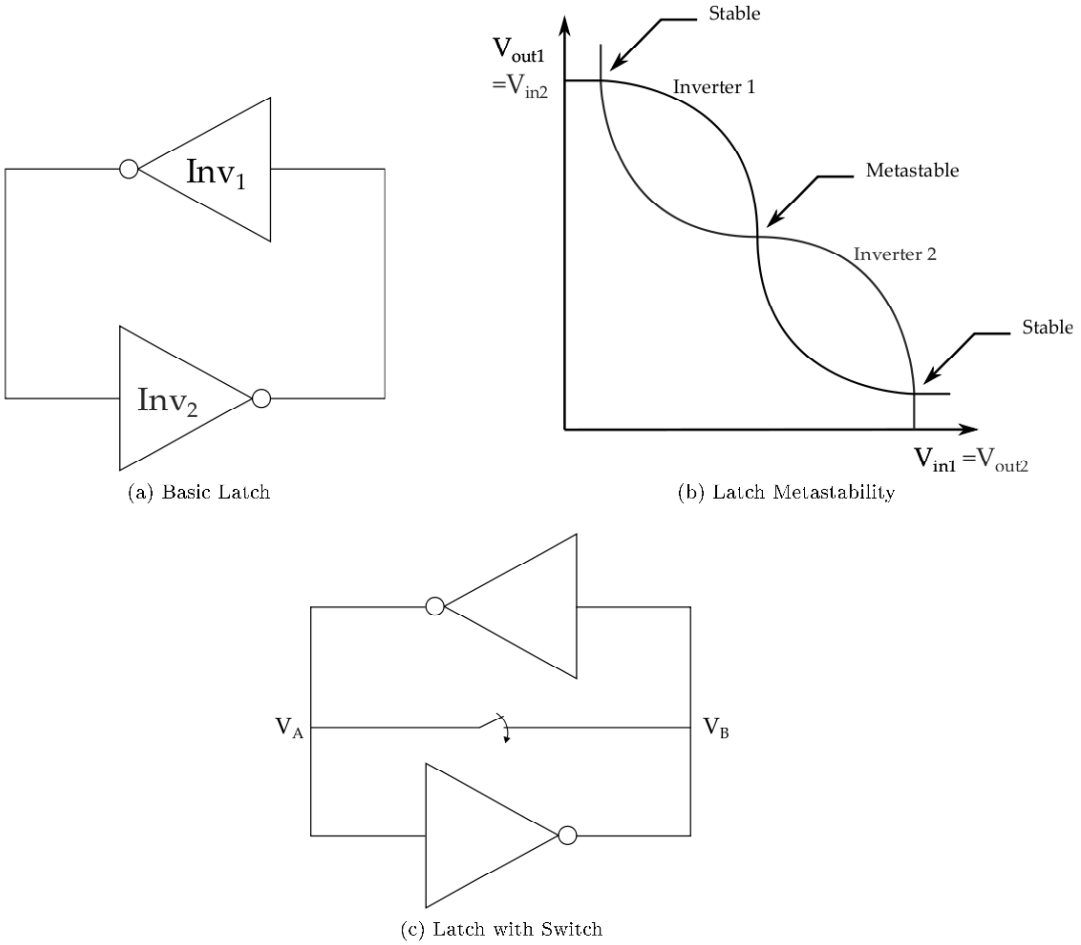

Figure 7.4: Latch

7.3 Inverter Comparator

The concept of a simple inverter as a comparator is interesting. Using sizing alone you can distribute the trip points for the flash ADC. A simple inverter is only single ended. Figure 7.5 shows the simple inverter with increased gain by having active loads supplied by a bias voltage. The simulation results in Figure 7.6 show how changing the device size and the bias voltages can change the trip point of the inverter over a large range.

Figure 7.7 shows the evolution of a differential version of the using the inverter as a comparator, starting with a basic differential pair and combining the complimentary invert as the input. This is a self bias differential receiver.[4] This tackled the problem of making

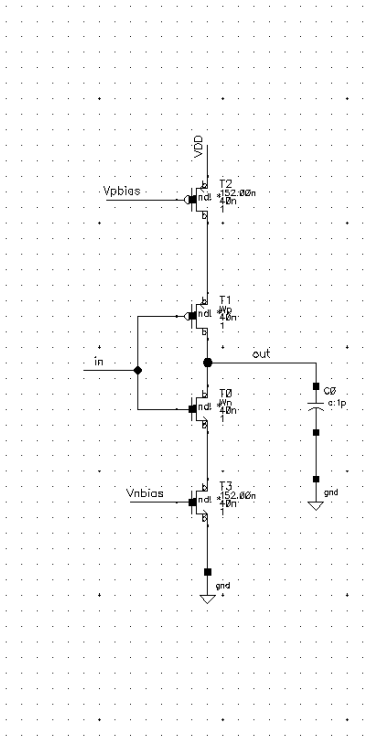

Figure 7.5: Inverter

the inverter differential and last we need to find out how to clock this design. One option is to just reset the inputs after the latch is clocked using a transmission gate to short the inputs to each other.

The variation in capacitance of the inputs to the comparator could be a problem, but the fact that the resistance seen at each input is also changing, you just need to design for the worst case scenario. The ultimate biggest resistance and biggest capacitance would create the slowest comparator decision.

7.4 References

References are as stated before normally made from a resistive ladder. The resistive ladder ends up constantly drawing current and can be wasteful due to that aspect. It could also be done using a MOS divider or capacitor divider for lower power consumption. The

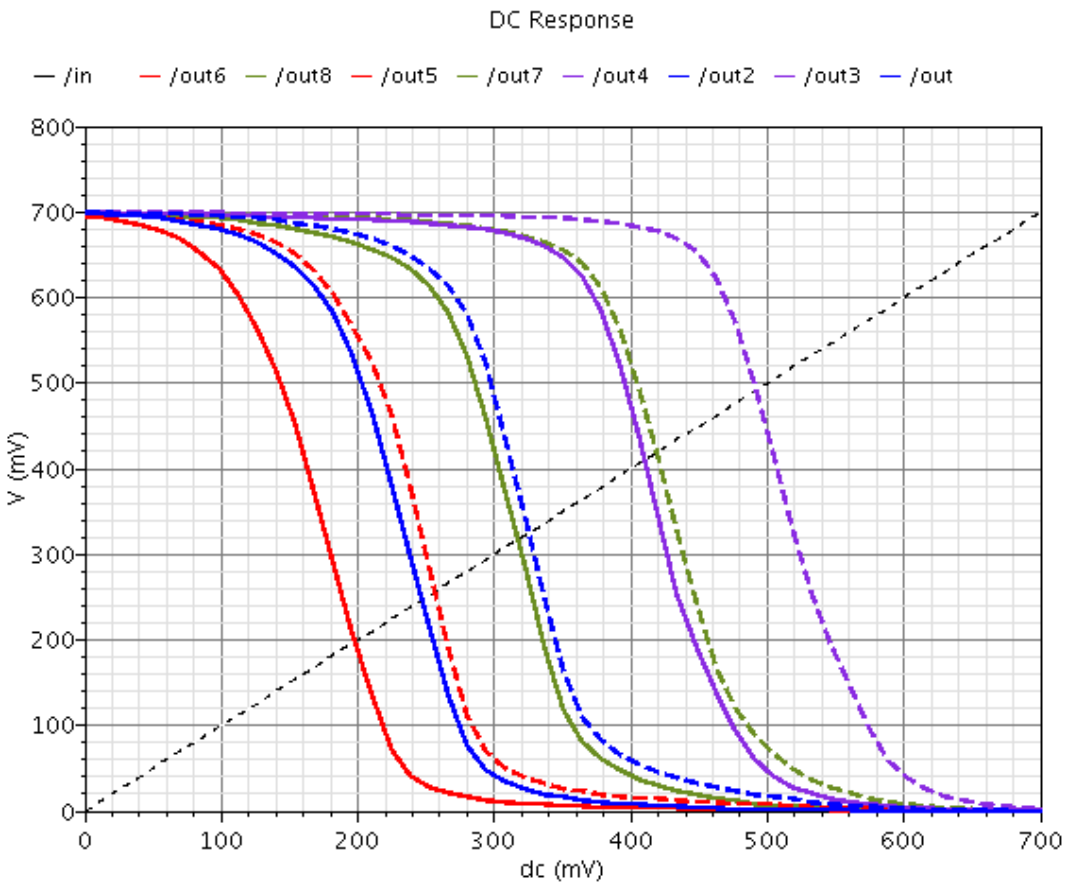

Figure 7.6: Inverter Simulation

problem with the last two is that the input capacitance would be come even larger and possibly limit the speed of the circuit. Also capacitors tend to be larger than resistors in terms of area and you have to consider the divider created with the input to the comparator.

The best way to change the threshold may be by changing the sizes of devices like explained in the comparator theory section. For example you can change the thresholds of an inverter by make the PMOS or the NMOS bigger. For this topology you are limited by the resolution of the process on how many different sizes and combinations you have to create different thresholds. Another disadvantage is having to layout many different sized blocks and limiting the amount of copy and paste that can be done in layout. But if thats the cost for extreme reduction of area and power for the reference ladder, this approach

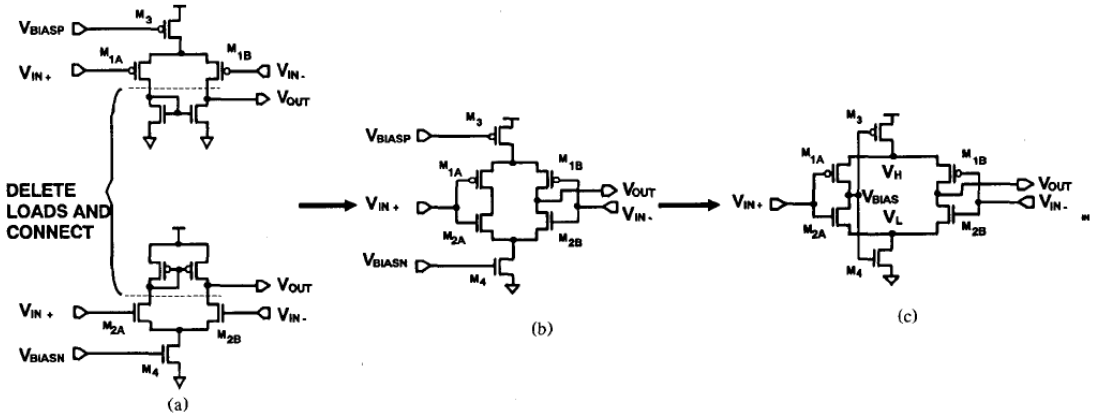

Figure 7.7: Evolution of the Self-Bias Differential Receiver[4]

might be worth investigating more. Another disadvantage of this approach is that each input can be a different capacitance if we distribute based on sizing. This means that each comparator may take longer to make a decision, but you would again design for your worst case and set the limit on your sizing based on speed.

7.5 Logic

The last block of the Flash ADC is the logic that samples the comparator output and puts the stream of 1s into a format readable by an FPGA or micro processor. A typical flash ADC designed for linearity and accurate output without calibration might have thermometer code to binary converter. This will convert the raw outputs into a binary number. This design is normally assuming no bubble codes and all trip points designed in a row. Bubble codes are codes in the thermometer string that should be 1 but are zero due to comparator offset. This ADC will most likely be bubble codes and other errors in our output, we will use an adder to count the ones. The adder that will be used will be a Wallace tree adder. Unlike typical Flash ADC encoders, this technique will offer the error correction and suppression without the use of addition NAND gates handle the bubble codes[3]. This design may be slow and need pipe-lining to improve it's performance. Figure 7.8 shows a circuit diagram

of a Wallace tree adder.

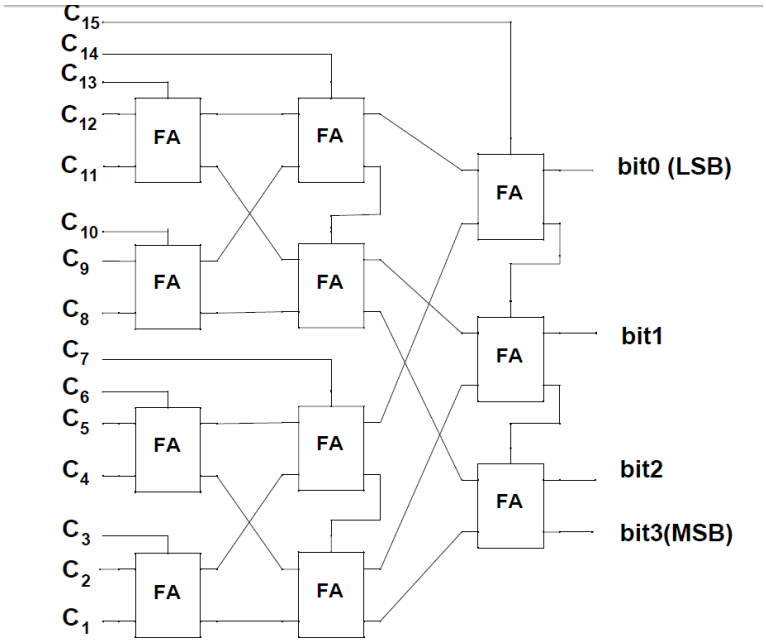

Figure 7.8: Wallace Tree Adder [3]

7.6 FPGA ADC

The easiest way to implement that logic for our ADC would be by using an internal FPGA. This way we could have the raw output of the ADC feed directly to the FPGA which then can add the signal, perform the algorithm, output the corrected and calibrated code without having to go to an external FPGA. Since we are limited by the small die area that we have, this will not happen soon. But if future funding was given and more chips could be fabricated after our first test chip, this is the way to go.

After reading the paper about the fully digital stochastic flash ADC[21], an attempt was made at implementing this design all in an FPGA. One interesting thing is that many FPGAs use input buffers that are similar to the differential inverter design discussed in the section about comparators. It was observed that the variation of trip points, without being able to size the devices and depending only on process variation, was too small to make a

converter that was very useful. Initially it was thought to be hard to implement the shift, but if we have a constant shift and the FPGA could allow multiple rails, the shift can be done similar to how it was implemented in [21].

Chapter 8

Conclusions and Future Recommendations

8.1 Conclusions

This paper has presented all digital background calibration of a redundant flash ADC suitable for for aggressively scaled CMOS technologies. Implementation using the split-ADC calibration technique minimizes analog complexity and enables purely background calibration. Four times redundant comparators are used and correction is realized using a look-up table which is continuously calibrated in the background using a split calibration architecture. Simulation results show the proposed algorithm has the ability to reach performance comparable to previous work without requiring additional silicon area, a precise signal source, or offline calibration.

Presented in the thesis was a calibration algorithm using the concept of a split-ADC with a flash analog to digital converter. The idea of the split-ADC was first published in [11]. This novel idea has been applied to many converters and now can be applied to a Flash ADC. Each ADC needs its own version of this calibration due to the fact that each have their own shortcomings and each need to collect the information provided about those shortcomings in a different way.

The major results of the calibration have been presented here and also in the published

paper based on this calibration [17]. The results presented are based on redundancy and therefore have 4 times the number of comparators than a normal 6-bit ADC. This calibration can be implemented with whatever number of comparators that are needed based on the desired effective number of bits. It can also use with any type of flash ADC architecture as long as there is some version of a ones counted output to allow for the maximum amount of information to be collected from the ADC and the calibration scheme to work correctly.

8.2 Future Recommendations

For this project, an implementation of the flash ADC was not completed but ideas are presented here to encourage continued work on this project in the direction of a very fast and inaccurate ADC. This can create an overall Flash ADC with true background calibration at over a giga sample per second and minimize the area and power by allowing a less accurate architecture and comparator.

I would also recommend looking more into the calibration in terms of limiting bad corrections. A few ideas concerning this are as follows: a dynamic least mean square term, full scale limiting, and other means in which to control the calibration. Currently the the approach is simple and effective. I do believe with some additional complexity in the calibration there may be a way to reach the full potential of the look-up table correction by controlling the calibration more. This may add cost in terms of FPGA complexity and size but in the end might minimize the amount of comparators needed to reach closer to a full 6-bit accuracy. The last thing that should be considered is interleaving the flash ADC and calibrating interleaving errors and possibly making use of the additional information created by the increased redundancy.

8.3 Closing

Before working on this project, my appreciation for digital was minimal and my focus was on all analog design. From the project I learned a lot in terms of programming in FPGAs and MATLAB, and ultimately learned the extreme usefulness that digital circuits

and code can provide to a complex and inaccurate analog system.

Appendix A

MATLAB CODE

```
 %% Inverse Operation to find errors
%% MS THESIS
%% Calibration keys
points = 1024;                %% points per calibration cycle
calibpoints = 10000000;          %% How many times to do calibration loop
Vref = 2;                   %% +/- 0.7V
numbits = 6;                  %% Per ADC on each side
tempsig=1.5;                 %% Sigma Variation in terms of LSB
shift = 1.5;                 %% Shift +/- to input
LMSN = 26  ;                    %% LMS mu value 1/2^LMSN
numtrips = 2^numbits -1;      %% Number of trip points to generate
LSB = Vref/numtrips;          %% the least sig bit
```

```
fs = 10000;
freq = 10000*(13/points);
T = 1/fs;
time=0:1/fs:(points-1)/(fs);
input =Vref*rand(1,points)-Vref/2;                    %% RANDOM input UNI
inputsine = (Vref/2)*sin(2*pi*time*freq);             %% SINE
inputramp=(0:Vref/(points-1):Vref)-Vref/2;            %% RAMP
calib = repmat(tempsig,1,calibpoints);                %% Vector f Sigma
SINADLUTAv = zeros(calibpoints,1);                    %% Collecting SINAD
ENOBLUTAv = zeros(calibpoints,1);                     %% Collecting ENOB
SINADLUTB = zeros(calibpoints,1);                    %% Collecting SINAD
ENOBLUTB = zeros(calibpoints,1);                     %% Collecting ENOB
SINADLUTC = zeros(calibpoints,1);                    %% Collecting SINAD
ENOBLUTC = zeros(calibpoints,1);                     %% Collecting ENOB
Timee = zeros(calibpoints,1);
%LUT = zeros(128, calibpoints);
%% Declare original LUTs
LUT_changingA = (0:1:numtrips)';                      %% Look up table A
LUT_changingB = (0:1:numtrips)';                      %% Look up table B
sigma = LSB*tempsig;                                  %% Sigma value V
sig = 1;
%% Generate Errors and trips
errorsA = LSB + sigma.*randn(numtrips,1);             %% Error vector A
errorsB = LSB + sigma.*randn(numtrips,1);             %% Error vector B
```

```
REF_trips = (-Vref/2+LSB/2:LSB:Vref/2-LSB/2)';          %% Ideal trips
tripsA =REF_trips + errorsA;                            %% Error trip A
tripsB = REF_trips + errorsB;                           %% Error trip A

%% Implement Flash ADCs with different inputs, A/B
%% flash ADC A/B RANDOM
outsA = repmat(input,numtrips,1)>repmat(tripsA,1,points);
%% sum # of ones, divide by # of comparators, mult by voltage ref
outputA = ((sum (outsA,1))/numtrips)*Vref -Vref/2;
%% sum only
outputsumA = sum(outsA,1);
outsB = repmat(input,numtrips,1)>repmat(tripsB,1,points);
outputB = ((sum (outsB,1))/numtrips)*Vref -Vref/2;
outputsumB = sum(outsB,1);

%% flash ADC for cal with ramp A/B
outsrampA = repmat(inputramp,numtrips,1)>repmat(tripsA,1,points);
outputsumrampA = sum(outsrampA,1);
outputrampA = ((sum (outsrampA,1))/numtrips)*Vref -Vref/2;
outsrampB = repmat(inputramp,numtrips,1)>repmat(tripsB,1,points);
outputsumrampB = sum(outsrampB,1);
outputrampB = ((sum (outsrampB,1))/numtrips)*Vref -Vref/2;

%% flash ADC for cal with sine A/B
```

```
outssineA = repmat(inputsine,numtrips,1)>repmat(tripsA,1,points);
outputsumsineA = sum(outssineA,1);
outputsineA = ((sum (outssineA,1))/numtrips)*Vref -Vref/2;
outssineB = repmat(inputsine,numtrips,1)>repmat(tripsB,1,points);
outputsumsineB = sum(outssineB,1);
outputsineB = ((sum (outssineB,1))/numtrips)*Vref -Vref/2;
outrawavg = (outputsumsineA+outputsumsineB);
outrawavge = (outrawavg/numtrips)*Vref-Vref/2;

%% Ideal flash ADC with ramp
outsideal = repmat(inputramp, numtrips,1)>repmat(REF_trips,1,points);
outidealsum = sum(outsideal,1);
outideal = (sum(outsideal,1)/numtrips)*Vref-Vref/2;
%% Ideal flash ADC with sine
outsidealsine = repmat(input, numtrips,1)>repmat(REF_trips,1,points);
outidealsumsine = sum(outsidealsine,1);
outidealsine = (sum(outsidealsine,1)/numtrips)*Vref;

%% Vectors for vectorizing
even = 2:2:points;
odd = 1:2:points;
full = 1:points;
%% Create shift input using sine input
inputshift  = input;
```

```
inputshiftm = input;
inputshifta = input;
inputshift(even)=inputshifta(even)+shift*LSB;
inputshiftm(even)=inputshifta(even)-shift*LSB;
inputshift(odd)=inputshifta(odd)-shift*LSB;
inputshiftm(odd)=inputshifta(odd)+shift*LSB;

%% Shifted output values
outsshiftA = repmat(inputshiftm,numtrips,1)>repmat(tripsA,1,points);
outputsumshiftA = sum(outsshiftA,1);
outputshiftA = ((sum (outsshiftA,1))/numtrips)*Vref -Vref/2;
outsshiftB = repmat(inputshift,numtrips,1)>repmat(tripsB,1,points);
outputsumshiftB = sum(outsshiftB,1);
outputshiftB = ((sum (outsshiftB,1))/numtrips)*Vref -Vref/2;

tic
%% OVERALL calibration loop
    for test = calib,

        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        %%%%CALIBRATION%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        differenceshift=zeros(points,1);
        invforerrshift = zeros(points,(numtrips+1)*2);
```

```
        %% Take even differences from LUT value and add back known shift
        differenceshift(even)= LUT_changingB(outputsumshiftB(even)+1)
            -LUT_changingA(outputsumshiftA(even)+1)-shift*2;
        differenceshift(odd)= LUT_changingB(outputsumshiftB(odd)+1)
            -LUT_changingA(outputsumshiftA(odd)+1)+shift*2;

        %% Build weighted matrix with +1 for B -1 for A
%         invforerrshift(full,outputsumshiftB(full)+129)=1;
%         invforerrshift(full,outputsumshiftA(full)+1)=-1;
        %% Trying to vectorize
%         for i= 1:points,
%             invforerrshift(i,outputsumshiftB(i)+129)=1;
%             invforerrshift(i,outputsumshiftA(i)+1)=-1;
%         end

        %%%%TRAVIS HELP%%%%%
        %% Vector Version
        B=[];
        A=[];
        Bp=[];
        Ap=[];
        column=128;
        invforerrshift2 =zeros(1,128*points);
```

```
Bp=65:column:points*column;
Ap=1:column:points*column;

B=Bp +(outputsumshiftB);
A=Ap +(outputsumshiftA);
%invforerrshift2 =zeros(points,256);
invforerrshift2(B) = 1;
invforerrshift2(A) = -1;
invforerrshift= reshape(invforerrshift2,128,points)';
%invforerrshift2 =zeros(points,256);
%invforerrshift2[B]=1;
%invforerrshift2(1:points,B) = 1;

%invforerrshift2=reshape(array,points,256);

%% Take transpose of invforerr
transforerr = invforerrshift';

%% Multiply the differences by the transposed weight matrix
atransdiffer = transforerr*differenceshift;

%% Multiply the LMS factor mu by  the difference and weighted
%% This creates a small fraction of the estimated errors
errortrans = (1/(2^LMSN))*atransdiffer;
```

```
ErrortransA= (errortrans(1:(numtrips+1)));
ErrortransB= (errortrans((numtrips+2):(numtrips+1)*2));

%% Each loop subtract erros from the look up tables
LUT_changingA = LUT_changingA-ErrortransA;
LUT_changingB = LUT_changingB-ErrortransB;

%% Corrected Outputs
OUTLUTshiftA = (LUT_changingA(outputsumrampA+1)/numtrips)*Vref-Vref/2;
OUTLUTshiftB = (LUT_changingB(outputsumrampB+1)/numtrips)*Vref-Vref/2;
OUTLUTshiftsineA = (LUT_changingA(outputsumsineA+1)/numtrips)*Vref-Vref/2;
OUTLUTshiftsineB = (LUT_changingB(outputsumsineB+1)/numtrips)*Vref-Vref/2;

%% Averaged outputs
AVG = (LUT_changingA(outputsumsineA+1)+LUT_changingB(outputsumsineB+1));  %%outputsumsineA
AVG2 = (LUT_changingA(outputsumsineA+1)+LUT_changingB(outputsumsineB+1))/2;
ONEA = LUT_changingA(outputsumsineA+1);
ONEB = LUT_changingB(outputsumsineB+1);
AVGr = round(AVG);
AVGr2 = round(AVG2);
OUTLUTavg = (AVG/(numtrips*2))*Vref-Vref/2;
OUTONEA = (ONEA/(numtrips))*Vref-Vref/2;
OUTONEB = (ONEB/(numtrips))*Vref-Vref/2;
OUTLUTavgmax = (AVG/max(AVGr))*Vref-Vref/2;
```

```
OUTLUTavgr = (AVGr/(numtrips*2))*Vref-Vref/2;
AVG6 = bitshift(AVGr,-1);
numtrips6 = bitshift(numtrips,-1);
OUTLUTavg6 = (AVG6/numtrips6)*Vref-Vref/2;
AVGramp = (LUT_changingA(outputsumrampA+1)+LUT_changingB(outputsumrampB+1))/2;
AVGramp = round(AVGramp);
OUTLUTavgramp = (AVGramp/numtrips)*Vref-Vref/2;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% FFT Work%%
%% FFT OUTavg
sAv = fft(OUTLUTavg); %%OUTLUTlmssineA
SAv = 20*log10(abs(sAv));
aAv = SAv(1:length(OUTLUTavg)/2);
dfAv = fs/(length(SAv));
f1Av = 0:dfAv:dfAv*(length(aAv)-1);

%% Determine power spectrum
spectPv=(abs(sAv)).*(abs(sAv));
Pdcv = sum(spectPv(1));  %% dont need, differential
Psv = max(spectPv(1:points/2));
Pallv = sum(spectPv(1:points/2));
Pnv = Pallv-Psv-Pdcv;
```

```
SINADLUTAv(sig)=10*log10(Psv/Pnv);
ENOBLUTAv(sig) = (SINADLUTAv(sig) - 1.76)/6.02;

sB = fft(OUTONEA); %%OUTLUTlmssineA
SB = 20*log10(abs(sB));
aB = SB(1:length(OUTONEA)/2);
dfB = fs/(length(SB));
f1B = 0:dfB:dfB*(length(aB)-1);

%% Determine power spectrum
spectPB=(abs(sB)).*(abs(sB));
PdcB = sum(spectPB(1));  %% dont need, differential
PsB = max(spectPB(1:points/2));
PallB = sum(spectPB(1:points/2));
PnB = PallB-PsB-PdcB;
SINADLUTB(sig)=10*log10(PsB/PnB);
ENOBLUTB(sig) = (SINADLUTB(sig) - 1.76)/6.02;
%sig = sig +1;
%LUT(1:128,sig) = LUT_changingA(1:128);
Timee(sig) =toc;

sC = fft(OUTONEB); %%OUTLUTlmssineA
SC = 20*log10(abs(sC));
aC = SC(1:length(OUTONEB)/2);
```

```
        dfC = fs/(length(SC));
        f1C = 0:dfC:dfC*(length(aC)-1);

        %% Determine power spectrum
        spectPC=(abs(sC)).*(abs(sC));
        PdcC = sum(spectPC(1));  %% dont need, differential
        PsC = max(spectPC(1:points/2));
        PallC = sum(spectPC(1:points/2));
        PnC = PallC-PsC-PdcC;
        SINADLUTC(sig)=10*log10(PsC/PnC);
        ENOBLUTC(sig) = (SINADLUTC(sig) - 1.76)/6.02;
        sig = sig +1;
        %LUT(1:128,sig) = LUT_changingA(1:128);
        %Timeee(sig) =toc;
    end

%% PlotEnob vs converions
conversion = (1:points:points*calibpoints)';
figure(1)
plot(conversion,ENOBLUTAv , conversion, ENOBLUTB, conversion, ENOBLUTC)


%DNL_INL_SIN
% dnl and inl ADC output
```

```
% input y contains the ADC output
% vector obtained from quantizing a
% sinusoid
% Boris Murmann, Aug 2002
% Bernhard Boser, Sept 2002
% histogram boundaries
AVGrr = bitshift(AVGr,-1);
y=AVGr;
minbin=min(y);
maxbin=max(y);
% histogram
h = hist(y, minbin:maxbin);
% cumulative histogram
ch = cumsum(h);
% transition levels found by:
T = -cos(pi*ch/sum(h));
% linearized histogram
hlin = T(2:end) - T(1:end-1);
% truncate at least first and last
% bin, more if input did not clip ADC
trunc=2;
hlin_trunc = hlin(1+trunc:end-trunc);
% calculate lsb size and dnl
lsb= sum(hlin_trunc) / (length(hlin_trunc));
```

```
dnlcal= [0 hlin_trunc/lsb-1];
misscodes = length(find(dnlcal<-0.99));
% calculate inl
inlcal= cumsum(dnlcal);
codes6bitcal = (0:1:(length(hlin_trunc)));
% figure(6)
% plot(codes6bit,inlcal)
% xlabel('OUTPUT CODE')
% ylabel('INL[LSB]')
% figure(7)
% plot(codes6bit,dnlcal)
% xlabel('OUTPUT CODE')
% ylabel('DNL[LSB]')

outrawavg =round(outputsumsineB);
%outrawavg = bitshift(outrawavg,-1);
yx = outrawavg;
%DNL_INL_SIN
% dnl and inl ADC output
% input y contains the ADC output
% vector obtained from quantizing a
% sinusoid
% Boris Murmann, Aug 2002
% Bernhard Boser, Sept 2002
```

```
% histogram boundaries
minbin=min(yx);
maxbin=max(yx);
% histogram
h = hist(yx, minbin:maxbin);
% cumulative histogram
ch = cumsum(h);
% transition levels found by:
T = -cos(pi*ch/sum(h));
% linearized histogram
hlin = T(2:end) - T(1:end-1);
% truncate at least first and last
% bin, more if input did not clip ADC
trunc=2;
hlin_trunc = hlin(1+trunc:end-trunc);
% calculate lsb size and dnl
lsb= sum(hlin_trunc) / (length(hlin_trunc));
dnl= [0 hlin_trunc/lsb-1];
misscodes = length(find(dnl<-0.99));
% calculate inl
inl= cumsum(dnl);
codes6bit = (0:1:(length(hlin_trunc)));
figure(6)
plot(codes6bit,inl,codes6bitcal,inlcal,'LineWidth',2,'LineStyle',':','Color',[0 0 0]);
```

```
xlabel('OUTPUT CODE')
ylabel('INL[LSB]')
figure(7)
plot(codes6bit,dnl,codes6bitcal,dnlcal,'LineWidth',2,'LineStyle',':','Color',[0 0 0]);
xlabel('OUTPUT CODE')
ylabel('DNL[LSB]')

figure(8)
plot(inputramp, outputrampA, inputramp,outputrampB,
inputramp,outideal,inputramp,OUTLUTavgramp)
xlabel ('Input')
ylabel('Output')
title('Ideal,Best,Avg Input/Ouputs');
axis([-1 1 -1 1]);


%%%% 7 bits per side

figure(9)
plot(conversion,ENOBLUTAv)
```

Appendix B

UPGRADED MATLAB

```
 %% Inverse Operation to find errors
%% MS THESIS
%% Calibration keys
points = 1024;                %% points per calibration cycle
calibpoints = 50000;          %% How many times to do calibration loop
Vref = 2;                   %% +/- 0.7V
numbits = 7;                  %% Per ADC on each side
tempsig=2.6;                 %% Sigma Variation in terms of LSB
shift = 4;                 %% Shift +/- to input
LMSNF = 20;                     %% LMS mu value 1/2^LMSN
LMSN = 20;
LMSND = 22;
```

```
numtrips = 2^numbits -1;      %% Number of trip points to generate
LSB = Vref/numtrips;          %% the least sig bit
fs = 10000;
freq = 10000*(13/points);
T = 1/fs;
time=0:1/fs:(points-1)/(fs);
%input =(Vref+.1)*rand(1,points)-(Vref+.1)/2;                    %% RANDOM input UNI
input =(Vref)*rand(1,points)-(Vref)/2;                    %% RANDOM input UNI
inputsine = (Vref/2)*sin(2*pi*time*freq);              %% SINE
inputramp=(0:Vref/(points-1):Vref)-Vref/2;             %% RAMP
calib = repmat(tempsig,1,calibpoints);                 %% Vector f Sigma
SINADLUTAv = zeros(calibpoints,1);                     %% Collecting SINAD
ENOBLUTAv = zeros(calibpoints,1);                      %% Collecting ENOB
SINADLUTB = zeros(calibpoints,1);                     %% Collecting SINAD
ENOBLUTB = zeros(calibpoints,1);                      %% Collecting ENOB
rmserror = zeros(calibpoints,1);
Timee = zeros(calibpoints,1);
LUT = zeros(128, calibpoints);
%% Declare original LUTs
LUT_changingA = (0:1:numtrips)';                       %% Look up table A
LUT_changingB = (0:1:numtrips)';                       %% Look up table B
LUT_changedA = (0:1:numtrips)';                       %% Look up table A
LUT_changedB = (0:1:numtrips)';                       %% Look up table B
LUT_changedAD = (0:1:numtrips)';                       %% Look up table A
```

```
LUT_changedBD = (0:1:numtrips)';                    %% Look up table B
LUT_sumA = zeros(numtrips+1,1);                    %% Look up table A
LUT_sumB = zeros(numtrips+1,1);                    %% Look up table B
LUT_changedAF = int32(LUT_changedA);                    %% Look up table A
LUT_changedBF = int32(LUT_changedB);                    %% Look up table B
LUT_changedAF = bitsll(LUT_changedAF,26);                    %% Look up table A
LUT_changedBF = bitsll(LUT_changedBF,26);
LUT_sumAF = int32(LUT_sumA);                    %% Look up table A
LUT_sumBF = int32(LUT_sumB);                    %% Look up table B
sigma = LSB*tempsig;                    %% Sigma value V
sig = 1;
%% Generate Errors and trips
errorsA = LSB + sigma.*randn(numtrips,1);                    %% Error vector A
errorsB = LSB + sigma.*randn(numtrips,1);                    %% Error vector B
REF_trips = (-Vref/2+LSB/2:LSB:Vref/2-LSB/2)';                    %% Ideal trips
%tripsA =REF_trips + errorsA;                    %% Error trip A
%tripsB = REF_trips + errorsB;                    %% Error trip A

%% Implement Flash ADCs with different inputs, A/B
%% flash ADC A/B RANDOM
outsA = repmat(input,numtrips,1)>repmat(tripsA,1,points);
%% sum # of ones, divide by # of comparators, mult by voltage ref
outputA = ((sum (outsA,1))/numtrips)*Vref -Vref/2;
%% sum only
```

```
outputsumA = sum(outsA,1);
outsB = repmat(input,numtrips,1)>repmat(tripsB,1,points);
outputB = ((sum (outsB,1))/numtrips)*Vref -Vref/2;
outputsumB = sum(outsB,1);

%% flash ADC for cal with ramp A/B
outsrampA = repmat(inputramp,numtrips,1)>repmat(tripsA,1,points);
outputsumrampA = sum(outsrampA,1);
outputrampA = ((sum (outsrampA,1))/numtrips)*Vref -Vref/2;
outsrampB = repmat(inputramp,numtrips,1)>repmat(tripsB,1,points);
outputsumrampB = sum(outsrampB,1);
outputrampB = ((sum (outsrampB,1))/numtrips)*Vref -Vref/2;

%% flash ADC for cal with sine A/B
outssineA = repmat(inputsine,numtrips,1)>repmat(tripsA,1,points);
outputsumsineA = sum(outssineA,1);
outputsineA = ((sum (outssineA,1))/numtrips)*Vref -Vref/2;
outssineB = repmat(inputsine,numtrips,1)>repmat(tripsB,1,points);
outputsumsineB = sum(outssineB,1);
outputsineB = ((sum (outssineB,1))/numtrips)*Vref -Vref/2;
outrawavg = (outputsumsineA+outputsumsineB);
outrawavge = (outrawavg/(numtrips*2))*Vref-Vref/2;

%% Ideal flash ADC with ramp
```

```
outsideal = repmat(inputramp, numtrips,1)>repmat(REF_trips,1,points);
outidealsum = sum(outsideal,1);
outideal = (sum(outsideal,1)/numtrips)*Vref-Vref/2;
%% Ideal flash ADC with sine
outsidealsine = repmat(inputsine, numtrips,1)>repmat(REF_trips,1,points);
outidealsumsine = sum(outsidealsine,1);
outidealsine = (sum(outsidealsine,1)/numtrips)*Vref-Vref/2;

%% Vectors for vectorizing
even = 2:2:points;
odd = 1:2:points;
full = 1:points;
%% Create shift input using sine input
inputshift  = input;%sine;
inputshiftm = input;%sine;
inputshifta = input;%sine;
inputshift(even)=inputshifta(even)+shift*LSB;
inputshiftm(even)=inputshifta(even)-shift*LSB;
inputshift(odd)=inputshifta(odd)-shift*LSB;
inputshiftm(odd)=inputshifta(odd)+shift*LSB;

%% Shifted output values
outsshiftA = repmat(inputshiftm,numtrips,1)>repmat(tripsA,1,points);
outputsumshiftA = sum(outsshiftA,1);
```

```
outputshiftA = ((sum (outsshiftA,1))/numtrips)*Vref -Vref/2;
outsshiftB = repmat(inputshift,numtrips,1)>repmat(tripsB,1,points);
outputsumshiftB = sum(outsshiftB,1);
outputshiftB = ((sum (outsshiftB,1))/numtrips)*Vref -Vref/2;

tic
%% OVERALL calibration loop
    for test = calib,

        input =(Vref)*rand(1,points)-(Vref)/2;                    %% RANDOM input UNI
        inputshift  = input;%sine;
        inputshiftm = input;%sine;
        inputshifta = input;%sine;
        inputshift(even)=inputshifta(even)+shift*LSB;
        inputshiftm(even)=inputshifta(even)-shift*LSB;
        inputshift(odd)=inputshifta(odd)-shift*LSB;
        inputshiftm(odd)=inputshifta(odd)+shift*LSB;

        %% Shifted output values
        outsshiftA = repmat(inputshiftm,numtrips,1)>repmat(tripsA,1,points);
        outputsumshiftA = sum(outsshiftA,1);
        outputshiftA = ((sum (outsshiftA,1))/numtrips)*Vref -Vref/2;
        outsshiftB = repmat(inputshift,numtrips,1)>repmat(tripsB,1,points);
        outputsumshiftB = sum(outsshiftB,1);
```

```
    outputshiftB = ((sum (outsshiftB,1))/numtrips)*Vref -Vref/2;

    %% FPGA stuff
    differenceagain = zeros(points,1);
    differenceagainF = int32(differenceagain);
    differenceagainF = bitsll(differenceagainF,26);  %% not needed
    shiftshift = bitsll(shift,26) ;
    LUT_sumA = zeros(numtrips+1,1);                       %% Look up table A
    LUT_sumB = zeros(numtrips+1,1);                       %% Look up table B
    LUT_sumAa = zeros(numtrips+1,1);                      %% Look up table A
    LUT_sumBa = zeros(numtrips+1,1);

    LUT_sumAaD = zeros(numtrips+1,1);                     %% Look up table A
    LUT_sumBaD = zeros(numtrips+1,1);
    LUT_sumAF = int32(LUT_sumA);                        %% Look up table A
    LUT_sumBF = int32(LUT_sumB);                        %% Look up table B
    LUT_sumAaF = int32(LUT_sumAa);                      %% Look up table A
    LUT_sumBaF = int32(LUT_sumBa);
    LUT_sumAD = zeros(numtrips+1,1);                      %% Look up table A
    LUT_sumBD = zeros(numtrips+1,1);                      %% Look up table B

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%%%CALIBRATION%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
differenceshift=zeros(points,1);
invforerrshift = zeros(points,(numtrips+1)*2);

%% Take even differences from LUT value and add back known shift
differenceshift(even)= LUT_changingB(outputsumshiftB(even)+1)
    -LUT_changingA(outputsumshiftA(even)+1)-shift*2;
differenceshift(odd)= LUT_changingB(outputsumshiftB(odd)+1)
    -LUT_changingA(outputsumshiftA(odd)+1)+shift*2;
differenceagain(even)= LUT_changedB(outputsumshiftB(even)+1)
    -LUT_changedA(outputsumshiftA(even)+1)-shift*2;
differenceagain(odd)= LUT_changedB(outputsumshiftB(odd)+1)
    -LUT_changedA(outputsumshiftA(odd)+1)+shift*2;
differenceagainF(even)= LUT_changedBF(outputsumshiftB(even)+1)
    -LUT_changedAF(outputsumshiftA(even)+1)-shiftshift*2;
differenceagainF(odd)= LUT_changedBF(outputsumshiftB(odd)+1)
    -LUT_changedAF(outputsumshiftA(odd)+1)+shiftshift*2;
%%% might need to fix this line
%% what I really want to do it sum the difference for each output sum
for i=1:points
    LUT_sumAa(outputsumshiftA(i)+1) =
          LUT_sumAa(outputsumshiftA(i)+1) - differenceagain(i);
    LUT_sumBa(outputsumshiftB(i)+1) =
          LUT_sumBa(outputsumshiftB(i)+1) + differenceagain(i);
    LUT_sumAaF(outputsumshiftA(i)+1) =
```

```
            LUT_sumAaF(outputsumshiftA(i)+1) - differenceagainF(i);
        LUT_sumBaF(outputsumshiftB(i)+1) =
            LUT_sumBaF(outputsumshiftB(i)+1) + differenceagainF(i);

        %% need to make changes to more than just one entry of the LUT
        %% should detect if even or odd to know if I should add up to end
        %% or add down to the begining
        if mod(i,2) ==0, %% odd
            LUT_sumAD(1:outputsumshiftA(i)+1) =
                LUT_sumAD(1:outputsumshiftA(i)+1) - differenceagain(i);
            LUT_sumBD(outputsumshiftB(i)+1:end) =
                LUT_sumBD(outputsumshiftB(i)+1:end) + differenceagain(i);
        else %% even
            LUT_sumAD(outputsumshiftA(i)+1:end) =
                LUT_sumAD(outputsumshiftA(i)+1:end) - differenceagain(i);
            LUT_sumBD(1:outputsumshiftB(i)+1) =
                LUT_sumBD(1:outputsumshiftB(i)+1) + differenceagain(i);

        end
    end
    LUT_sumA = (1/(2^LMSNF))*LUT_sumAa;
    LUT_sumB = (1/(2^LMSNF))*LUT_sumBa;
    LUT_changedA = LUT_changedA-LUT_sumA;
    LUT_changedB = LUT_changedB-LUT_sumB;
```

```
        LUT_sumAaD = (1/(2^LMSND))*LUT_sumAD;
        LUT_sumBaD = (1/(2^LMSND))*LUT_sumBD;
        LUT_changedAD = LUT_changedAD-LUT_sumAaD;
        LUT_changedBD = LUT_changedBD-LUT_sumBaD;

        LUT_sumAF = (1/(2^LMSNF))*LUT_sumAaF;
        LUT_sumBF = (1/(2^LMSNF))*LUT_sumBaF;
        LUT_changedAF = LUT_changedAF-LUT_sumAF;
        LUT_changedBF = LUT_changedBF-LUT_sumBF;
        LUT_FIXEDA = bitsra(LUT_changedAF,26);
        LUT_FIXEDB = bitsra(LUT_changedBF,26);

        %% Build weighted matrix with +1 for B -1 for A
%         invforerrshift(full,outputsumshiftB(full)+129)=1;
%         invforerrshift(full,outputsumshiftA(full)+1)=-1;
        %% Trying to vectorize
%         for i= 1:points,
%             invforerrshift(i,outputsumshiftB(i)+129)=1;
%             invforerrshift(i,outputsumshiftA(i)+1)=-1;
%         end

        %%%%TRAVIS HELP%%%%%
        %% Vector Version
```

```
B=[];
A=[];
Bp=[];
Ap=[];
column=(numtrips+1)*2;
invforerrshift2 =zeros(1,(numtrips+1)*2*points);
Bp=17:column:points*column;
Ap=1:column:points*column;

B=Bp +(outputsumshiftB);
A=Ap +(outputsumshiftA);
%invforerrshift2 =zeros(points,256);
invforerrshift2(B) = 1;
invforerrshift2(A) = -1;
invforerrshift= reshape(invforerrshift2,(numtrips+1)*2,points)';
%invforerrshift2 =zeros(points,256);
%invforerrshift2[B]=1;
%invforerrshift2(1:points,B) = 1;

%invforerrshift2=reshape(array,points,256);

%% Take transpose of invforerr
transforerr = invforerrshift';
```

```
%% Multiply the differences by the transposed weight matrix
atransdiffer = transforerr*differenceshift;

%% Multiply the LMS factor mu by  the difference and weighted
%% This creates a small fraction of the estimated errors
errortrans = (1/(2^LMSN))*atransdiffer;
ErrortransA= (errortrans(1:(numtrips+1)));
ErrortransB= (errortrans((numtrips+2):(numtrips+1)*2));

%% Each loop subtract erros from the look up tables
LUT_changingA = LUT_changingA-ErrortransA;
LUT_changingB = LUT_changingB-ErrortransB;

%% Corrected Outputs
OUTLUTshiftA = (LUT_changingA(outputsumrampA+1)/numtrips)*Vref-Vref/2;
OUTLUTshiftB = (LUT_changingB(outputsumrampB+1)/numtrips)*Vref-Vref/2;
OUTLUTshiftsineA = (LUT_changingA(outputsumsineA+1)/numtrips)*Vref-Vref/2;
OUTLUTshiftsineB = (LUT_changingB(outputsumsineB+1)/numtrips)*Vref-Vref/2;
OUTLUTshiftAF = (LUT_changedA(outputsumrampA+1)/numtrips)*Vref-Vref/2;
OUTLUTshiftBF = (LUT_changedB(outputsumrampB+1)/numtrips)*Vref-Vref/2;
OUTLUTshiftsineAF = (LUT_changedA(outputsumsineA+1)/numtrips)*Vref-Vref/2;
OUTLUTshiftsineBF = (LUT_changedB(outputsumsineB+1)/numtrips)*Vref-Vref/2;

%% Averaged outputs
```

```
        AVG = (LUT_changingA(outputsumsineA+1)+LUT_changingB(outputsumsineB+1))/2;
        AVGF = (LUT_changedAD(outputsumsineA+1)+LUT_changedBD(outputsumsineB+1))/2;
        %%outputsumsineA
        AVGr = round(AVG);
        OUTLUTavg = (AVG/(numtrips))*Vref-Vref/2;
        OUTLUTavgF = (AVGF/(numtrips))*Vref-Vref/2;
        OUTLUTavgr = (AVGr/(numtrips))*Vref-Vref/2;
        %%AVG6 = bitshift(AVGr,1);
        %%numtrips6 = bitshift(numtrips,-2);
        %OUTLUTavg6 = (AVG6/numtrips6)*Vref-Vref/2;
        AVGramp = (LUT_changingA(outputsumrampA+1)+LUT_changingB(outputsumrampB+1))/2;
        AVGramp = round(AVGramp);
        OUTLUTavgramp = (AVGramp/numtrips)*Vref-Vref/2;

        AVGrampD = (LUT_changedAD(outputsumrampA+1)+LUT_changedBD(outputsumrampB+1))/2;
        AVGrampD = round(AVGrampD);
        OUTLUTavgrampD = (AVGrampD/numtrips)*Vref-Vref/2;

%         AVGramp = (LUT_changingA(outputsumrampA+1)+LUT_changingB(outputsumrampB+1))/2;
%         AVGramp = round(AVGramp);
%         OUTLUTavgramp = (AVGramp/numtrips)*Vref-Vref/2;
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        %% FFT Work%%
        %% FFT OUTavg
```

```
sAv = fft(OUTLUTavg); %%OUTLUTlmssineA
SAv = 20*log10(abs(sAv));
aAv = SAv(1:length(OUTLUTavg)/2);
dfAv = fs/(length(SAv));
f1Av = 0:dfAv:dfAv*(length(aAv)-1);

%% Determine power spectrum
spectPv=(abs(sAv)).*(abs(sAv));
Pdcv = sum(spectPv(1));  %% dont need, differential
Psv = max(spectPv(1:points/2));
Pallv = sum(spectPv(1:points/2));
Pnv = Pallv-Psv-Pdcv;
SINADLUTAv(sig)=10*log10(Psv/Pnv);
ENOBLUTAv(sig) = (SINADLUTAv(sig) - 1.76)/6.02;

sB = fft(OUTLUTavgF); %%OUTLUTlmssineA
SB = 20*log10(abs(sB));
aB = SAv(1:length(OUTLUTavgF)/2);
dfB = fs/(length(SAv));
f1B = 0:dfB:dfB*(length(aB)-1);

%% Determine power spectrum
spectPB=(abs(sB)).*(abs(sB));
PdcB = sum(spectPB(1));  %% dont need, differential
```

```
        PsB = max(spectPB(1:points/2));
        PallB = sum(spectPB(1:points/2));
        PnB = PallB-PsB-PdcB;
        SINADLUTB(sig)=10*log10(PsB/PnB);
        ENOBLUTB(sig) = (SINADLUTB(sig) - 1.76)/6.02;
        rmserror(sig) = 20*log(abs(std(inputsine(1:1000)-OUTLUTavg(1:1000)')));
        sig = sig +1;
       %% LUT(1:128,sig) = LUT_changingA(1:128);
        Timee(sig) =toc;
    end

%% PlotEnob vs converions
conversion = (1:points:points*calibpoints)';
figure(1)
plot(conversion,ENOBLUTAv , conversion, ENOBLUTB)


% %DNL_INL_SIN
% % dnl and inl ADC output
% % input y contains the ADC output
% % vector obtained from quantizing a
% % sinusoid
% % Boris Murmann, Aug 2002
% % Bernhard Boser, Sept 2002
```

```
% % histogram boundaries
% AVGr = bitshift(AVGr,-2);
% y=AVGr;
% minbin=min(y);
% maxbin=max(y);
% % histogram
% h = hist(y, minbin:maxbin);
% % cumulative histogram
% ch = cumsum(h);
% % transition levels found by:
% T = -cos(pi*ch/sum(h));
% % linearized histogram
% hlin = T(2:end) - T(1:end-1);
% % truncate at least first and last
% % bin, more if input did not clip ADC
% trunc=2;
% hlin_trunc = hlin(1+trunc:end-trunc);
% % calculate lsb size and dnl
% lsb= sum(hlin_trunc) / (length(hlin_trunc));
% dnlcal= [0 hlin_trunc/lsb-1];
% misscodes = length(find(dnlcal<-0.99));
% % calculate inl
% inlcal= cumsum(dnlcal);
% codes6bitcal = (0:1:(length(hlin_trunc)));
```

```
% % figure(6)
% % plot(codes6bit,inlcal)
% % xlabel('OUTPUT CODE')
% % ylabel('INL[LSB]')
% % figure(7)
% % plot(codes6bit,dnlcal)
% % xlabel('OUTPUT CODE')
% % ylabel('DNL[LSB]')
%
% outrawavg =round(outrawavg);
% outrawavg = bitshift(outrawavg,-2);
% yx = outrawavg;
% %DNL_INL_SIN
% % dnl and inl ADC output
% % input y contains the ADC output
% % vector obtained from quantizing a
% % sinusoid
% % Boris Murmann, Aug 2002
% % Bernhard Boser, Sept 2002
% % histogram boundaries
% minbin=min(yx);
% maxbin=max(yx);
% % histogram
% h = hist(yx, minbin:maxbin);
```

```
% % cumulative histogram
% ch = cumsum(h);
% % transition levels found by:
% T = -cos(pi*ch/sum(h));
% % linearized histogram
% hlin = T(2:end) - T(1:end-1);
% % truncate at least first and last
% % bin, more if input did not clip ADC
% trunc=2;
% hlin_trunc = hlin(1+trunc:end-trunc);
% % calculate lsb size and dnl
% lsb= sum(hlin_trunc) / (length(hlin_trunc));
% dnl= [0 hlin_trunc/lsb-1];
% misscodes = length(find(dnl<-0.99));
% % calculate inl
% inl= cumsum(dnl);
% codes6bit = (0:1:(length(hlin_trunc)));
% figure(6)
% plot(codes6bit,inl,codes6bitcal,inlcal,'LineWidth',2,'LineStyle',':','Color',[0 0 0]);
% xlabel('OUTPUT CODE')
% ylabel('INL[LSB]')
% figure(7)
% plot(codes6bit,dnl,codes6bitcal,dnlcal,'LineWidth',2,'LineStyle',':','Color',[0 0 0]);
% xlabel('OUTPUT CODE')
```

```
% ylabel('DNL[LSB]')

figure(8)
plot(inputramp, outputrampA, inputramp,outputrampB,inputramp,
outideal,inputramp,OUTLUTavgramp, inputramp, OUTLUTavgrampD)
xlabel ('Input')
ylabel('Output')
title('Ideal,Best,Avg Input/Ouputs');
axis([-1 1 -1 1]);

rmserror(1:1000) = 20*log(abs(inputsine(1:1000)-OUTLUTavg(1:1000)'));
rmserror(1:1000) = 20*log(abs(inputsine(1:1000)-outrawavge(1:1000)));
```

Appendix C

FPGA CODE

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    10:24:28 10/25/2011
// Design Name:
// Module Name:    inverter
// Project Name:
// Target Devices:
// Tool versions:
// Description:
```

```
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module inverter(
    input [15:0] inputs1,
    input [15:0] inputs2,
    output [3:0] outputcount1,
    output [3:0] outputcount2,
    output [15:0] output1,
    output [15:0] output2,
    output [6:0] seven_seg,
    output [3:0] anode,
    input clk,
    input rst,
    output flash_ce,
    output sram_oe,
    output sram_we,
    output sram_ub,
    output sram_lb,
```

```
    output sram_ce,
    output [22:0] sram_addr,
    output [7:0] sram_data

    );
// intialize array
   reg clk_250HZ;
   reg clk_25M;
   reg clk_25000;
   parameter MEM_SIZE = 1024;
   reg [3:0] mem1 [0:MEM_SIZE -1];
   reg [3:0] mem2 [0:MEM_SIZE -1];
   integer k;
   reg [3:0] T1;
   reg [3:0] T2;
   reg [4:0] Tall;
   reg [15:0] outputs2;
   reg [20:0]counter_250;
   reg [20:0]counter_25000;
   reg [22:0] addr;
   reg [4*100-1:0] collectT1;
   reg [4*100-1:0] collectT2;
   wire start;
   wire rw;
```

```
// wire[15:0] inv1;
// wire [15:0] inv2;
// wire[15:0] invv1;
// wire [15:0] invv2;
   wire [15:0] value;
   wire [7:0] datain1;
   wire [7:0] dataout1;
   wire [128:0] LUTT1;
   wire [128:0] LUTT2;
   integer i;
   integer conv;


   initial
      begin
            T1 =4'b0;
            T2 =4'b0;
            counter_250 = 21'd0;
            counter_25000 = 21'd0;
            addr = 23'b00000000000000000000000;
            //start =1'b0;
            //rw =1'b0;
      end
```

```
always @ (posedge clk, posedge rst)
begin
   if(rst)
      begin
      clk_250HZ <= 0;
      counter_250<=21'd0;
      end
   else if (counter_250 ==21'd100000)
      begin
      clk_250HZ <= ~clk_250HZ;
      counter_250<=21'd0;
      end
   else
      counter_250 <= counter_250 +1;
end

always @ (posedge clk, posedge rst)
begin
   if(rst)
      begin
      clk_25000 <= 0;
      counter_25000<=21'd0;
      end
```

```
        else if (counter_25000 ==21'd1000)
110        begin
           clk_25000 <= ~clk_25000;
           counter_25000<=21'd0;
           end
        else
115        counter_25000 <= counter_25000 +1;
     end

     always @(posedge rst, posedge clk)
     begin
120     if(rst)
           clk_25M <=0;
        else
           clk_25M <= ~clk_25M;
     end
125
  //instantiate disp
  displayv2 disp1 (
  .value(value),
  .seven_seg(seven_seg),
130 .mux_clk(clk_250HZ),
  .anode(anode));
```

```
//entity sram is
//  Port ( sram_oe          :  out std_logic;
//       sram_we         :  out std_logic;
//       sram_ub         :  out std_logic;
//       sram_lb         :  out std_logic;
//       sram_ce         :  out std_logic;
//
//         sram_addr     :  out std_logic_vector(22 downto 0);
//       sram_data       :  inout std_logic_vector(7 downto 0);
//
//       clock           :  in std_logic; --
//       reset           :  in std_logic; --aborts last operation
//       ready           :  out std_logic; --ready to start operation when high
//       start           :  in std_logic; --starts operation if high
//       rw              :  in std_logic; --read/write control. if '1' read else write
//       addr            :  in std_logic_vector(22 downto 0);
//       datain          :  in std_logic_vector(7 downto 0); --data to be written
//       dataout         :  out std_logic_vector(7 downto 0)); --data to be read
//end sram;

   // instantiate sram cntrl
   sram sram (
         .sram_oe(sram_oe),
         .sram_we(sram_we),
```

```
        .sram_ub(sram_ub),
        .sram_lb(sram_lb),
        .sram_ce(sram_ce),

        .sram_addr(sram_addr),
        .sram_data(sram_data),

        .clock(clk),
        .reset(rst),
        .ready(ready),
        .start(start),
        .rw(rw),
        .addr(addr),
        .datain(datain1),
        .dataout(dataout1));


/* initial
   begin
   for (k = 0; k < MEM_SIZE - 1; k = k + 1)
   begin
       mem[k][3:0] = 0;
   end
   end*/
```

```
// assign anode = anpwnd;
// assign inv1 = ~inputs1;
// assign output1 = inv1;
// assign inv2 = inputs2;
// assign output2= inv2;


   always @( posedge clk)
   begin
   T1= 4'b0;
   T2= 4'b0;
   Tall = 5'b00000;
   for( i =0; i<16; i =i+1) begin
      T1 = T1+inputs1[i];
      T2 = T2+inputs2[i];
      Tall = Tall +inputs1[i] + inputs2[i];
   end
   end

   // might need to make batch correction too. for cal...
   //instantiate Correction
   correction correction (
```

```
        .T1(T1),
        .T2(T2),
        .clk(clk),
        .LUTT1(LUTT1),
        .LUTT2(LUTT1),
        .OUT1(outputcount1),
        .OUT2(outputcount2)
);

/// instantiate Calibration
calibration calibration(
        .T1(T1),
        .T2(T2),
        .clk(clk),
        .clk_25000(clk_25000),
        .LUTT1(LUTT1),
        .LUTT2(LUTT2),
        .OUT1(OUT2),
        .OUT2(OUT2),
        .OUTavg(OUTavg)
        );


//assign outputcount2 = T2;
```

```
   //assign outputcount1 = T1;
   assign value[15:0] = {8'b0,outputcount1,outputcount2};

   assign datain1 [7:0] = {3'b0,Tall[4:0]};
   //assign dataout1 [7:0]= {T1,T2};
   assign flash_ce =1'b1;

   //start  <= '1' WHEN (read0 = '1' OR write0 = '1') AND ready = '1' else '0';
   //rw        <= '1' WHEN read0 = '1' else '0' WHEN write0 = '1' else 'Z';

   assign start = ready ? 1:0;
   assign rw = 0;



   always @(posedge clk_250HZ)
   begin
      addr= addr+1'b1;
   end




// always @ (inputs2)
// begin
```

```
//    outputs2<=inputs2;
// end

// always @ (posedge clk)
// begin
//    outputs2<=inputs2;
// end
//



/* assign invv1 = output1;
   assign invv2 = output2;*/

//
// assign outputcount1[3:0] = T1;
//
// always @(outputcount1)
//    for( conv=0; conv<MEM_SIZE-1; conv=conv+1)
//    begin
//       mem1[conv] <= outputcount1;
//    end
//
// assign inv2 = ~inputs2;
// assign output2 = inv2;
```

```
//
// always @( inv2)
//     for( i =0; i<16; i =i+1) begin
//        T2 = T2+inv2[i];
//     end
//
// assign outputcount2[3:0] = T2;
//
// always @(outputcount2)
//     for( conv=0; conv<MEM_SIZE-1; conv=conv+1)
//     begin
//        mem2[conv] <= outputcount2;
//     end
280
285
290
endmodule
295
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
300
entity sram is
```

```
  Port ( sram_oe        :  out std_logic;
         sram_we        :  out std_logic;
         sram_ub        :  out std_logic;
         sram_lb        :  out std_logic;
         sram_ce        :  out std_logic;

         sram_addr      :  out std_logic_vector(22 downto 0);
         sram_data      :  inout std_logic_vector(7 downto 0);

         clock          :  in std_logic; --
         reset          :  in std_logic; --aborts last operation
         ready          :  out std_logic; --ready to start operation when high
         start          :  in std_logic; --starts operation if high
         rw             :  in std_logic; --read/write control. if '1' read else write
         addr           :  in std_logic_vector(22 downto 0);
         datain         :  in std_logic_vector(7 downto 0); --data to be written
         dataout        :  out std_logic_vector(7 downto 0)); --data to be read
end sram;

architecture Behavioral of sram is
   TYPE state_type IS(sREADY, sSTART, sWRITE1, sWRITE2,
   sWRITE3, sREAD1, sREAD2, sREAD3, sREAD4, sREAD5, sREAD6);
   signal current_state, next_state : state_type;
   signal dataout_temp : std_logic_vector(7 downto 0);
```

```
begin

state_memory: process (clock, reset)
begin
   if reset = '1' then
      current_state <= sREADY;
   elsif clock ' EVENT and clock = '1' then
      current_state <= next_state;
   end if;
end process state_memory;

next_state_logic: process(current_state, start, rw)
begin
   CASE current_state IS
      WHEN sREADY =>
                     if start = '1' then
                        next_state <= sSTART;
                     else
                        next_state <= sREADY;
                     end if;
      WHEN sSTART =>
                     if rw = '1' then
                        next_state <= sREAD1;
```

```
                else
                   next_state <= sWRITE1;
                end if;
        WHEN sWRITE1 =>
                next_state <= sWRITE2;
        WHEN sWRITE2 =>
                next_state <= sWRITE3;
        WHEN sWRITE3 =>
                next_state <= sREADY;

        WHEN sREAD1 =>
                next_state <= sREAD2;
        WHEN sREAD2 =>
                next_state <= sREAD3;
        WHEN sREAD3 =>
                next_state <= sREAD4;
        WHEN sREAD4 =>
                next_state <= sREAD5;
        WHEN sREAD5 =>
                next_state <= sREAD6;
        WHEN sREAD6 =>
                next_state <= sREADY;
```

```
   END CASE;
end process next_state_logic;

sram_addr <= "ZZZZZZZZZZZZZZZZZZZZZZ" when current_state = sREADY else addr;

ready    <= '1' when current_state = sREADY else '0';
sram_ce  <= '1' when current_state = sREADY else '0';

sram_oe  <= '1' when current_state = sREAD1 else
            '1' when current_state = sREAD2 else
            '1' when current_state = sSTART else
            '1' when current_state = sREAD3 else '0';

sram_we  <= '0' when current_state = sWRITE2 else
            '0' when current_state = sWRITE3 else '1';

sram_ub  <= '1';
sram_lb  <= '0';

sram_data <= datain     WHEN current_state = SWRITE2 else
             datain     WHEN current_state = SWRITE3 else
             "ZZZZZZZZ";
```

```
dataout_temp   <= sram_data      WHEN current_state = sREAD5 else
                  sram_data      WHEN current_state = sREAD4 else
                  dataout_temp   WHEN rw = '1' else
                  "ZZZZZZZZ";



dataout <= dataout_temp;


end Behavioral;



--------------------------------------------------------------------------------
-- Company: WPI
-- Engineer: Anthony Crasso
-- Box: 30
-- Create Date:    12:46:29 03/21/2011
-- Design Name: display decoder generic
-- Module Name:    display - Behavioral
-- Description: 16 bit input to 4 seven segment displays
--
```

```
--
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity displayv2 is
    Port ( value : in  STD_LOGIC_VECTOR (15 downto 0);
           seven_seg : out  STD_LOGIC_VECTOR (6 downto 0);
           mux_clk : in  STD_LOGIC;
           anode : out  STD_LOGIC_VECTOR(3 downto 0)
);
end displayv2;

architecture Behavioral of displayv2 is
```

```
constant zero   : std_logic_vector(6 downto 0) := "1000000";
constant one    : std_logic_vector(6 downto 0) := "1111001";
constant two    : std_logic_vector(6 downto 0) := "0100100";
constant three  : std_logic_vector(6 downto 0) := "0110000";
constant four   : std_logic_vector(6 downto 0) := "0011001";
constant five   : std_logic_vector(6 downto 0) := "0010010";
constant six    : std_logic_vector(6 downto 0) := "0000010";
constant seven  : std_logic_vector(6 downto 0) := "1111000";
constant eight  : std_logic_vector(6 downto 0) := "0000000";
constant nine   : std_logic_vector(6 downto 0) := "0010000";
constant A      : std_logic_vector(6 downto 0) := "0001000";
constant B      : std_logic_vector(6 downto 0) := "0000011";
constant C      : std_logic_vector(6 downto 0) := "1000110";
constant D      : std_logic_vector(6 downto 0) := "0100001";
constant E      : std_logic_vector(6 downto 0) := "0000110";
constant F      : std_logic_vector(6 downto 0) := "0001110";

signal disp     : std_logic_vector(3 downto 0) := "0000";
alias disp0     : std_logic_vector(3 downto 0) IS value (15 downto 12);
alias disp1     : std_logic_vector(3 downto 0) IS value (11 downto 8);
alias disp2     : std_logic_vector(3 downto 0) IS value (7 downto 4);
alias disp3     : std_logic_vector(3 downto 0) IS value (3 downto 0);
```

```
begin
  --display the count value on the seven segments
  seven_segment_decoder_process: process(disp)
  begin
    case disp is
      when "0000" => seven_seg <= zero;
      when "0001" => seven_seg <= one;
      when "0010" => seven_seg <= two;
      when "0011" => seven_seg <= three;
      when "0100" => seven_seg <= four;
      when "0101" => seven_seg <= five;
      when "0110" => seven_seg <= six;
      when "0111" => seven_seg <= seven;
      when "1000" => seven_seg <= eight;
      when "1001" => seven_seg <= nine;
      when "1010" => seven_seg <= A;
      when "1011" => seven_seg <= B;
      when "1100" => seven_seg <= C;
      when "1101" => seven_seg <= D;
      when "1110" => seven_seg <= E;
      when "1111" => seven_seg <= F;
      when others => seven_seg <= zero;
    end case;
  end process seven_segment_decoder_process;
```

```
--count anodes
anode_count_process: process(mux_clk,disp0,disp1,disp2,disp3,disp)
   variable anode_count : integer range 0 to 3;
begin
   if rising_edge(mux_clk) then
      if anode_count = 3 then
         anode_count := 0;
         disp <= disp3;
         anode <= "1110";
      elsif anode_count = 2 then
         anode_count := anode_count + 1;
         disp <= disp2;
         anode <= "1101";
      elsif anode_count = 1 then
         anode_count := anode_count + 1;
         disp <= disp1;
         anode <= "1011";
      else
         anode_count := anode_count + 1;
         disp <= disp0;
         anode <= "0111";
      end if;
   end if;
```

```
   end process anode_count_process;
end Behavioral;



`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    11:30:47 12/19/2011
// Design Name:
// Module Name:    correction
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
```

```
////////////////////////////////////////////////////////////////////////////////
module correction(
    input [3:0] T1,
    input [3:0] T2,
    input      clk,
    input [128:0] LUTT1,
    input [128:0] LUTT2,
    output [3:0]  OUT1,
    output [3:0]  OUT2,
    output [3:0]  OUTavg
    );


   reg [7:0] temp1;
   reg [7:0] temp2;
   reg [7:0] LUTun1 [15:0];
   reg [7:0] LUTun2 [15:0];
   integer i;
   integer b,a;



   // make 4 bit portions out of flattened LUT
   // this might be sloww.....
```

```
565    always @( posedge clk)
       begin

       for( i =0; i<15; i =i+1) begin
          temp1 = LUTT1[(i*8+7)+:8 ];
570       temp2 = LUTT2[(i*8+7)+:8 ];
          LUTun1[i] = temp1;
          LUTun2[i] = temp2;
       end
       end
575
       // Correct the value by indexing the look up table
       assign OUT1 = LUTun1[T1]/16;
       assign OUT2 = LUTun2[T2]/16;

580    assign OUTavg = (OUT1 + OUT2)>>1;




    endmodule
585



    `timescale 1ns / 1ps
```

```
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    10:06:04 12/20/2011
// Design Name:
// Module Name:    calibration
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module calibration(
    input [3:0] T1,
    input [3:0] T2,
    input      clk,
    input      clk_25000,
```

```
output [128:0] LUTT1,
output [128:0] LUTT2,
input [3:0]   OUT1,
input [3:0]   OUT2,
input [3:0]  OUTavg
);


reg [7:0] diff;
reg [7:0] LUTun1 [15:0];
reg [7:0] LUTun2 [15:0];
reg [128:0] LUTTT1;
reg [128:0] LUTTT2;
reg [7:0] temp1;
reg [7:0] temp2;
reg [7:0] OUTbig1;
reg [7:0] OUTbig2;
reg [7:0]  shift;
reg odd;
integer i;
integer k;

initial
begin
```

```
    odd = 1;
    shift =8'b00000010;
  //for( i =0; i<16; i =i+1) begin

  //end
  end

  // every clock cycle store the difference
  // and acumulate differnce in LUT
  always @ (posedge clk)
  begin
  OUTbig1 = OUT1<<4;
  OUTbig2 = OUT2<<4;
    if (odd ==1)
       diff = OUTbig2-OUTbig1+shift;
    else
       diff = OUTbig2-OUTbig1-shift;// add in shift value and also maybe do shift of value here
  // EVEN AND ODD shifts (COUNT WITH IF STATEMENT)
  //diff = diff>>4;
  LUTun1[T1] = LUTun1[T1] +diff; // should put in temp
  LUTun2[T2] = LUTun2[T2] -diff;
  odd =~odd;
  //Dont shift till full output
  end
```

```
    // NOW DO SHIFT OF LUT

      //do flatten every cal cycle.
    // flatten LUTun
      // make 4 bit portions out of flattened LUT
   // this might be sloww.....
   always @( posedge clk_25000)
   begin

   for( i =0; i<16; i =i+1) begin

      temp1 = LUTun1[i];
      temp2 = LUTun2[i];
      LUTTT1[(i*4+3)+:4] = temp1;
      LUTTT2[(i*4+3)+:4] = temp2;
   end
   end

   assign LUTT1 = LUTTT1;
   assign LUTT2 = LUTTT2;

// /////////////// first try
//  initial
```

```
//  begin
//  for( i=0; i<100; i=i+1)
//  begin
//    W[i] = 0;
//    Wt[i] = 0;
//  end
//  end
//
//
// always @( posedge clk)
// begin
// if (k<100) begin
//    collectT1[k] = T1;
//    collectT2[k] = T2;
// need to figure out how to synronize with using it in the calibration part
//    collectDIFF[k] = OUT2-OUT1;
//    k = k+1;
// end
// else begin
//    k=0;
// end
// end
//
//
```

```
//
//  // Generate W matrix
//  always @( posedge clk)
//  begin
//  for( i=0; i<100; i=i+1)
//  begin
//  //W[i][collectT1[i]]=-1;
//  //W[i][collectT2[i]+16]=1;
//  Wt[collectT1[i]][i] = -1;
//  Wt[collectT1[i]][i] = 1; // Cheap inverses
//  end
//  end
//
//
//  // Then multiply  Differences
//  // Then shift to do division
//  // Subtract from LUT



endmodule
```

Bibliography

[1] Imran Ahmed, *Pipelined adc design and enhancement technique*, 2010.

[2] Imran Ahmed and David A. Johns, *Dac nonlinearity and residue gain error correction in a pipelined adc using a split-adc architecture*, Research in Microelectronics and Electronics 2006, Ph. D. (2006).

[3] Syed Masood Ali, Rabin Raut, and Mohamad Sawan, *Digital encoders for high speed flash-adcs: Modeling and comparison*, Circuits and Systems, 2006 IEEE North-East Workshop on, june 2006, pp. 69 –72.

[4] M. Bazes, *Two novel fully complementary self-biased cmos differential amplifiers*, Solid-State Circuits, IEEE Journal of **26** (1991), no. 2, 165 –168.

[5] Paul J. Hurst Carl R. Grace and Stephen H. Lewis, *A 12-bit 80-msample/s pipelined adc with bootstrapped digital calibration*, IEEE Journal of Solid-State Circuits **40** (2005), no. 5, 1038–1046.

[6] D. C. Daly and A. P. Chandrakasan, *A 6-bit, 0.2 v to 0.9 v highly digital flash adc with comparator redundancy*, IEEE J.Solid-State Circuits **44** (2009), no. 11, 3030–3038.

[7] Analog Devices, *Analog-digital conversion handbook*, third ed., 1985.

[8] Charles Gammal Devin Auclair and Fitzgerald Huang, *12b 100msps pipeline adc with open-loop reisdue amplifier*, 2008.

[9] C. Donovan and M. Flynn, *'a 'digital' 6-bit adc im0.25μm cmos*, Custom Integrated Circuits ConC (2001).

[10] Echere Iroaga and Boris Murmann, *A 12-bit 75-ms/s pipelined adc using incomplete settling*, IEEE Journal of Solid-State Circuits **42** (2007), no. 4.

[11] M. Coln J. McNeill and B. Larivee, *'split-adc' architecture fordeterministic digital background calibration of a 16b 1ms/s adc*, IEEE J.Solid-State Circuits **40** (2005), no. 12, 2437–2445.

[12] ———, *'split adc' calibration for all-digital correction of time-interleaved adc errors*, IEEE Trans. Circuits Syst. II **56** (2009), no. 5, 344–348.

[13] J.-O. Plouchart J. Proesel, G. Keskin and L. Pileggi, *'an 8-bit 1.5 gs/s flash adc using post-manufacturing statistcal selection*, IEEE Custom Integrated Circuits Conf. (2010), 1–4.

[14] Sanjeev Goluguri John A. McNeill and Abhilash Nair, *"split-adc" digital background correction of open-loop residue amplifier nonlinearity errors in a 14b pipeline adc*, (2007).

[15] David A. Johns and Ken Martin, *Analog integrated circuit design*, 1997.

[16] C. Donovan M. Flynn and L. Sattler, *Digital calibration incorporating redundancy of flash adcs*, IEEE Trans. Circuits Syst. II: Analog Digit. Signal Process **50** (2003), no. 5, 205–213.

[17] Rabeeh Majidi, Anthony Crasso, and John A. McNeill, *Digital background calibration of redundant split-flash adc in 45nm cmos*, Circuits and Systems (ISCAS), 2012 IEEE International Symposium on, may 2012, pp. 1271 –1274.

[18] Boris Murmann and Bernhard E. Boser, *Digitally assisted pipeline adcs, theory and implementation*, Boston, Massachusetts, 2004.

[19] Y. Nakajima, A. Sakaguchi, T. Ohkido, N. Kato, T. Matsumoto, and M. Yotsuyanagi, *A background self-calibrated 6b 2.7 gs/s adc with cascade-calibrated folding-interpolating architecture*, Solid-State Circuits, IEEE Journal of **45** (2010), no. 4, 707 –718.

[20] Shant Orchanian, *Split non-linear cyclic analog-to-digital converter*, Master's thesis, Worcester Polytechnic Institute, 2010.

[21] D. Knierim S. Weaver, B. Hershberg and U.-K. Moon, *"a 6b stochastic flash analog-to-digital converter without calibration or reference ladder*, IEEE Asian Solid-State Circuits Conf. (2008).

[22] Hattie Spetla, *Split cyclic analog to digital converter using a nonlinear gain stage*, Master's thesis, Worcester Polytechnic Institute, 2009.

[23] Dave Treleaven, *1.5-bit stages in pipelined adc*, May 2006.

[24] Li Zhang, *The calibration technique for pipelined adc*, 2008 International Conference on MultiMedia and Information Technology (2008).