All Digital, Background Calibration for Time-Interleaved and Successive Approximation Register Analog-to-Digital Converters

by

Christopher Leonidas David

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

Christopher Leonidas David

May 2010

APPROVED:

John McNeill

Professor John McNeill, Major Advisor

Donald R. Brown

Professor Donald R. Brown

Michael Coln

Doctor Michael Coln

Abstract

The growth of digital systems underscores the need to convert analog information to the digital domain at high speeds and with great accuracy. Analog-to-Digital Converter (ADC) calibration is often a limiting factor, requiring longer calibration times to achieve higher accuracy. The goal of this dissertation is to perform a fully digital background calibration using an arbitrary input signal for A/D converters. The work presented here adapts the cyclic "Split-ADC" calibration method to the time interleaved (TI) and successive approximation register (SAR) architectures.

The TI architecture has three types of linear mismatch errors: offset, gain and aperture time delay. By correcting all three mismatch errors in the digital domain, each converter is capable of operating at the fastest speed allowed by the process technology. The total number of correction parameters required for calibration is dependent on the interleaving ratio, M. To adapt the "Split-ADC" method to a TI system, 2M+1 half-sized converters are required to estimate 3(2M+1) correction parameters. This thesis presents a 4:1 "Split-TI" converter that achieves full convergence in less than 400 000 samples.

The SAR architecture employs a binary weight capacitor array to convert analog inputs into digital output codes. Mismatch in the capacitor weights results in non-linear distortion error. By adding redundant bits and dividing the array into individual unit capacitors, the "Split-SAR" method can estimate the mismatch and correct the digital output code. The results from this work show a reduction in the non-linear distortion with the ability to converge in less than 750 000 samples.

To My Family,

My Friends,

and My Lee Anne

Acknowledgements

It has been a great privilege to be a student of Professor John McNeill over the past six years. Over this period of time, I have worked on various projects including the Split-Interleaved and Split-SAR work on which this thesis is based. In each of these projects, he has provided me with invaluable guidance and insight in the field of analog integrated circuit design. His encouragement and assistance helped me through the most challenging aspects of my work. The annual trips to the International Sold-State Circuits Conference provided a first-hand glimpse at leading edge research in analog IC design. These trips inspired his graduate students to pursue a level of excellence and professional standard to match his own.

I would also like to thank my committee members, Professor D. Rick Brown and Dr. Michael Coln for their expertise and review of this work. Professor Brown's suggestions on algorithmic development and Dr. Coln's comments on circuit design were an important contribution to the thesis. In addition, I would like to thank Dr. Michael Coln's High Speed Converter group at Analog Devices, Inc. for their feedback on both the discrete and integrated circuit designs.

This work was carried out as part of the New England Center for Analog and Mixed Signal Design at WPI under the direction of Professor McNeill. The two Split-ADC projects were supported in part by NSF Award 0523996 and by Analog Devices, Inc.

This thesis represents research work that was carried out in collaboration with many colleagues at the Analog Lab in the Electrical and Computer Engineering Department at WPI, to which I owe my gratitude. Rosa Croughwell provided the layout design work for the integrated circuit version of the 4:1 Split-Interleaved architecture, to which I applied my calibration research. Chilann Chan, a colleague and a friend, developed the Split-SAR algorithm which was applied to ADC chip I worked on. Cody Brenneman was my partner throughout all of the trials and tribulations of the Split-SAR integrated circuit design. I could not have completed the project in such a timely and efficient manner without his help. I would like to thank my other colleagues in the lab, many of whom are also my friends, Tsai Chen, Ali Ulas Ilhan, Thomas Liechti, Chengxin Liu, Shant Orchanian, Altin Pelteku,

Alihusain Sirohiwala and Hattie Spetla.

In addition to the Analog Lab, there were several friends and colleagues in the Digital Lab and Machine Vision Lab who lent both moral and technical support. I would like to thank Jorge Alejandro, Vincent Amendolare, Matthew Campbell, Andrew Cavanaugh, Vivek Varshney and Benjamin Woodacre. Their guidance and friendship is greatly appreciated. Professor Fred Looft has been a great benefit as both the head of the Electrical and Engineering Department at WPI as well as my sailplane instructor.

Throughout my entire education, my family has been with me through the good time and the bad. Without their support, this thesis would not have been possible. Finally, I would like to thank my girlfriend Lee Anne Gerardi, for all of her love as well as her knowledge of the English language. Her encouragement and generous assistance in improving the readability of the thesis will not be forgotten.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

One of the most common themes in electrical engineering is the transmission, storage and processing of data. Early forms of this handled data in the analog domain, by manipulating signals with analog components such as resistors, capacitors, inductors, and amplifiers. This can be found in examples such as early analog oscilloscopes used for test and measurement. Figure 1.1 shows a block diagram of an analog oscilloscope which uses only components in the analog domain such as attenuators, amplifiers and delay circuits [1]. These types of oscilloscopes, while essential for analog circuit evaluation, had limited abilities in signal processing, analysis, and display abilities. The Cathode Ray Tube (CRT) could only display the input signal in real-time and didn't have the ability to store an input waveform for post-analysis [1].

With the advent of digital computing, there has been a tremendous shift of signal processing and storage from analog to the digital domain. To extend the example of the oscilloscope, the quantization and Data Acquisition (DAQ) of digital sampling oscilloscopes allowed for more advanced analysis [1]. Figure 1.2 shows how an input signal is minimally processed and filtered before being sampled and quantized by an Analog-to-Digital Converter (ADC). The digitized waveform can then be processed, sorted, analyzed and displayed entirely in the digital domain [1]. Both analog and digital scopes are capable of simple analysis such as measuring peak voltage (V_{pk}) and frequency. However, digital scopes can perform statistical analyses and generate frequency spectrum plots of analog

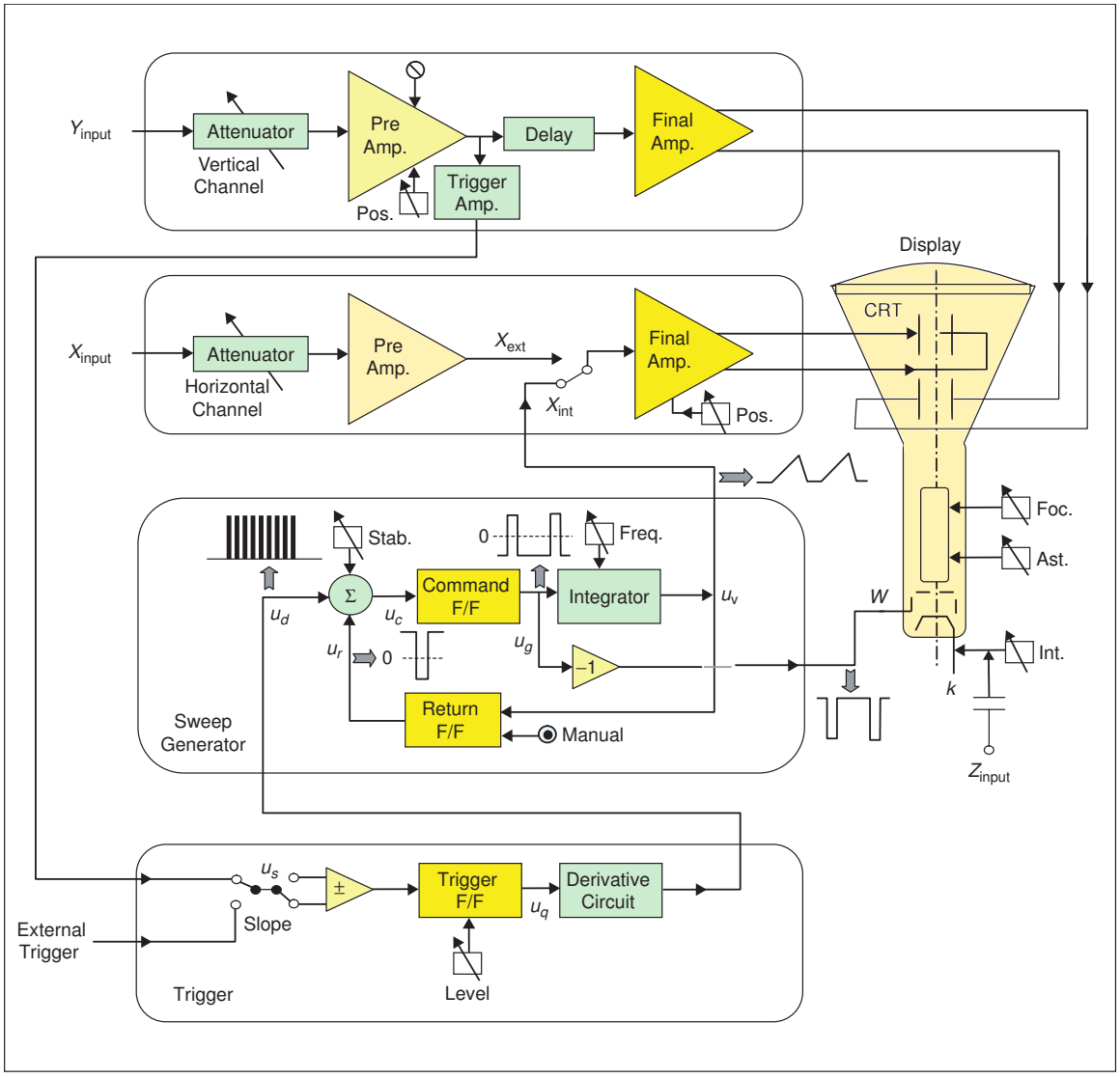

Figure 1.1: Block Diagram of an Analog Oscilloscope

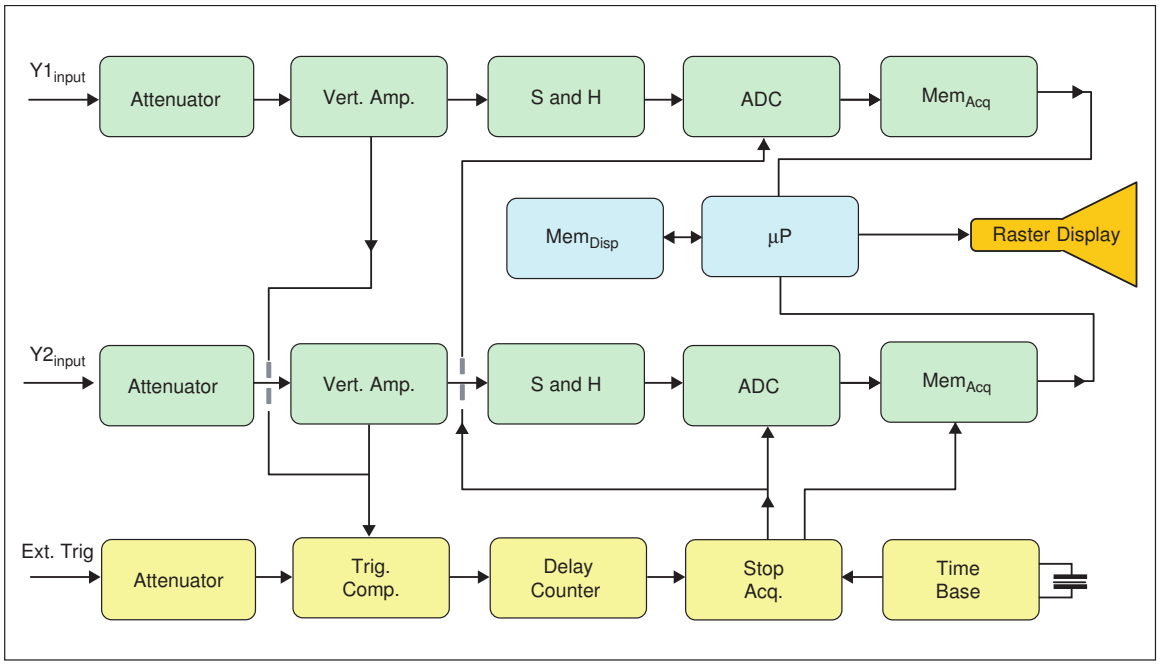

Figure 1.2: Block Diagram of a Digital Sampling Oscilloscope

waveforms [1].

When analog signals are transferred to the digital domain, the performance of the Digital Signal Processing (DSP) is limited by the accuracy of the Analog to Digital Converter (ADC) used. As the amount of data bandwidth has increased over the years, so has the speed, resolution and overall precision required of ADCs. Unfortunately, physical limitations and variations in the semiconductor fabrication process introduces sources of error into the A/D converters [2]. In order to increase their performance, much research has been done to calibrate out and correct for these inherent errors. The work presented here looks into two types of converter, time interleaved and Successive Approximation Register ADCs. These two A/Ds each have their own types of unique errors that must be corrected.

The Time Interleaved (TI) A/D is a technique that uses multiple converters to increase the throughput of the data. The interleaving method has some flexibility because it uses existing ADC architectures. It is capable of achieving faster conversion speeds than what a given technology would allow for a single converter [3]. Interleaving converters are often used in applications such as instrumentation. Typical TI systems use M sub-ADCs that each operate at 1/M the master sampling rate, f_S. The M digital output codes are then

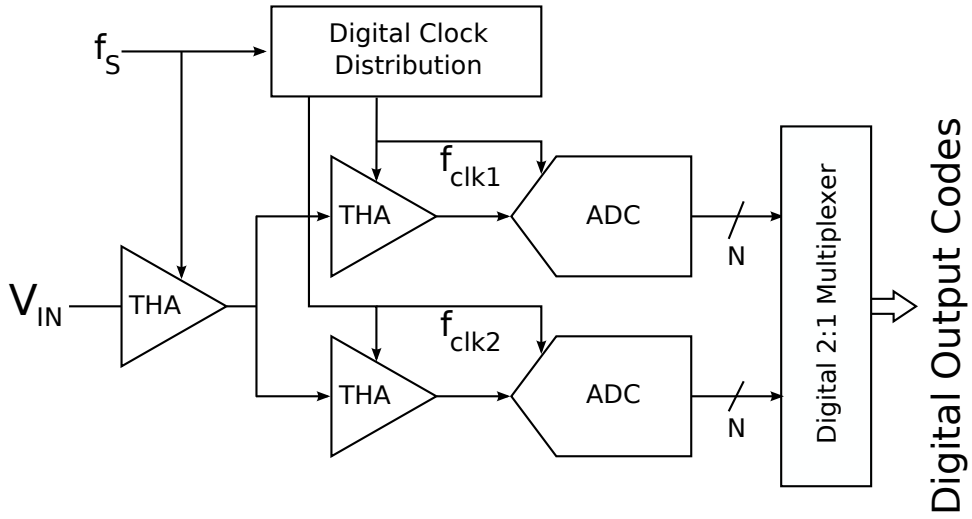

Figure 1.3: Block Diagram of a 2:1 Time-Interleaved ADC

multiplexed into a single stream of digital codes. A 2:1 example of which is shown in Figure 1.3 [2]. Mismatch between the sub-ADCs produces three types of errors, offset, gain, and aperture time delay which affect all interleaved A/Ds. These errors are often dealt with using a combination of analog and digital calibration techniques [2, 4, 3].

Successive Approximation Register ADCs use a binary search algorithm with a charge redistribution network to convert an analog input into a digital output. Use of capacitors in a binary weighted redistribution network help the SAR achieve moderate to high resolutions with relatively low power [5, 6, 7]. The mismatch between these capacitors leads to nonlinearity errors in the digital output code. The traditional technique for correcting this error source is to use a calibration DAC to offset the charge mismatch [7, 8]. This calibration still requires a method for measuring ratio errors of the binary weighted capacitors.

The goal of this work is to apply a self-calibrating algorithm to both the time interleaved and SAR converters. This algorithm uses the Split-ADC method introduced in [9, 10]. Most recent calibration techniques are performed using a digital logic, however they often require foreground operation or statistical methods to achieve full convergence. The Split-ADC performs the calibration and correction completely in the digital domain, while operating in the background and achieving a deterministic convergence rate. The original Split-ADC algorithm was used to estimate only two correction parameters. Both the time interleaved and the SAR converters require a much larger number of correction parameters to be found.

The majority of this research focused on adapting the Split-ADC method to calibrate the multiple parameters needed for these two architectures.

This thesis is organized as follows; Chapter 2 describes the performance metrics used to characterize ADCs. It then goes on to introduce the different types of architectures, their trade-offs, and the applications they are best suited for. Chapter 3 describes the interleaved and SAR architectures in more detail, as well as error sources and their affects on the output code. Chapter 4 discusses the work done using previous calibration techniques, including the Split-ADC method and its advantages over other methods. Chapter 5 presents what work has been to adapt the Split-ADC algorithm to a time interleaved converter and the results from simulation. Chapter 6 reviews the Split-SAR algorithm presented in [11], the design of the integrated circuit used for this research and the results from the fabricated chip. Finally, Chapter 7 concludes the research work presented here and provides possible paths for future designs.

Chapter 2

Background

A wide variety of analog to digital converters exist with each type suited for different applications. Section 2.1 describes the characterization of ADCs, in order to determine which ADC architecture to use for a given application. Section 2.2 describes different ADC architectures and their trade-offs in characteristics.

2.1 ADC Characterization

There are several characteristics of A/D converters that measure their performance. These are useful in determining which type of ADC to use for a target application. This section will discuss the parameters of conversion speed and resolution, integral and differential nonlinearity (INL and DNL), total harmonic distortion (THD), and signal to noise and distortion ratio (SNDR).

2.1.1 Sampling Rate and Resolution

The sampling rate of an ADC is used to determine the number of analog to digital code conversions that are completed within a specific period. Typically, this is measured in samples per second, such as 1 MS/sec showing that 1 million samples are collected within one second. This sampling rate is equal to the master sampling clock frequency for Nyquist rate ADCs. For these converters, the clock frequency must be greater than twice the highest frequency component in the analog input signal [12].

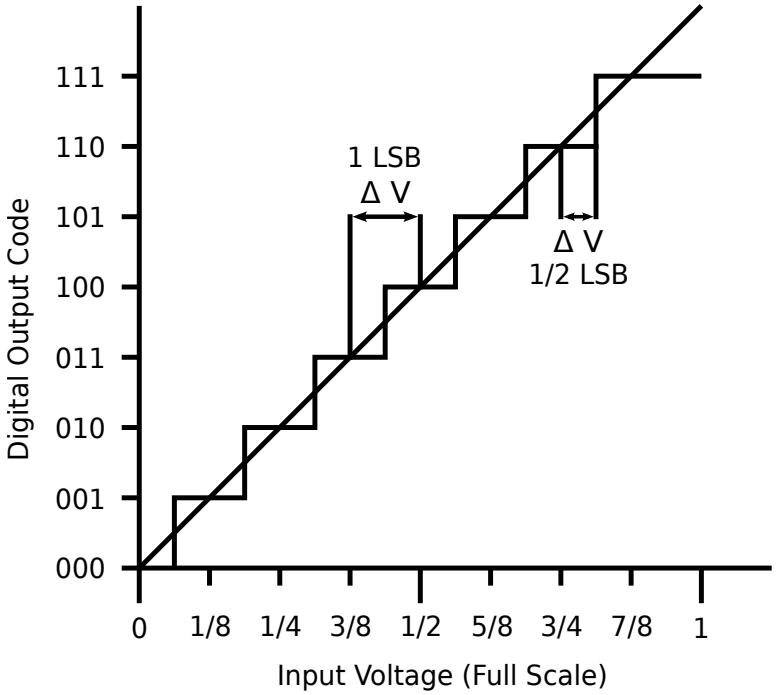

Figure 2.1: Ideal ADC Transfer Characteristic

The resolution of an analog to digital converter describes how many quantization levels the ADC can represent. Since the output of an ADC is in binary format, the resolution is given in powers of 2. For example, a 10-bit A/D converter can represent an analog signal using 2^{10} or 1024 quantization levels. As a general rule with Nyquist rate ADCs, there is a trade-off between sampling rate and converter accuracy and resolution. As the sampling rate increases, the converter resolution must be reduced in order to complete the conversion in a shorter period of time [12].

A single quantization level is the smallest analog voltage level that an ADC can resolve. This is called the Least Significant Bit, or LSB. The analog voltage value of the LSB is determined by the ADC resolution and the full scale voltage range, V_{FS}:

$$1 \text{ LSB} = \frac{V_{FS}}{2^N} \tag{2.1}$$

A transition level is the point from one quantization level to the next. The distance between two consecutive transition points is equal to a single quantization level or 1 LSB. Some input voltage changes are small enough to occur between the transition levels without crossing

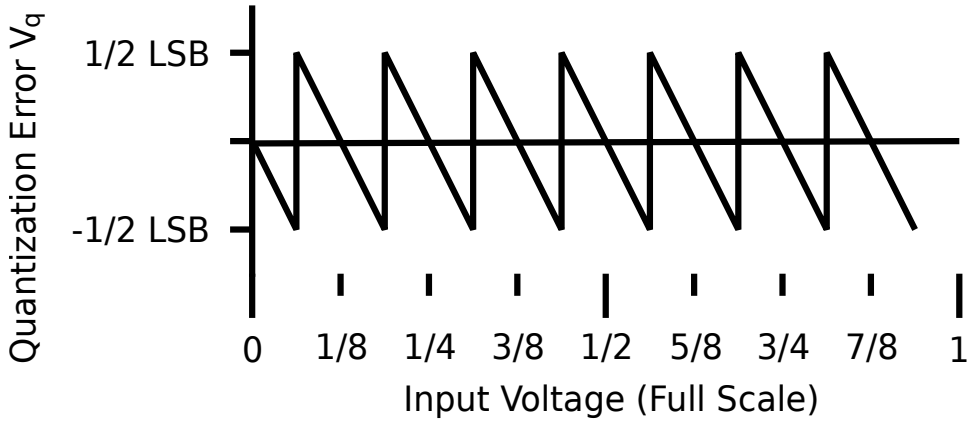

Figure 2.2: ADC Quantization Error Characteristic

them. These small changes are not detectable by the converter. This finite limit is referred to as the quantization error or noise in the A/D.

If an ideal transfer function of an ADC is compared to an ideal analog ramp, there is a difference of $\pm\frac{1}{2}$ LSB. This quantization noise signal has an average value of zero, but its Root Mean Squared (RMS) value can be shown as:

$$(\text{RMS})V_q \approx \frac{1 \text{ LSB}}{\sqrt{12}} = \frac{V_{FS}}{2^N\sqrt{12}} \tag{2.2}$$

[12, 13]

The ideal dynamic range of an A/D is the maximum Signal-to-Noise Ratio (SNR), or the ratio from the largest to the smallest analog signal the converter can represent. The SNR for signals with a uniform distribution from 0 to V_{FS} can be shown as a ratio of V_{FS} to 1 LSB [12, 5, 13].

$$\text{Dynamic Range (dB)} = 20\log\left(\frac{V_{FS}}{1 \text{ LSB}}\right) = 20\log\left(2^N\right) \approx 6.02N \tag{2.3}$$

However, since most input signals are sinusoidal with a non-uniform distribution, the maximum SNR is more commonly represented as the ratio of the AC power of the input to 1 LSB [5].

$$\text{Dynamic Range (dB)} = 20\log\left(\frac{V_{FS}/(2\sqrt{2})}{1 \text{ LSB}}\right) \approx 6.02N + 1.76 \tag{2.4}$$

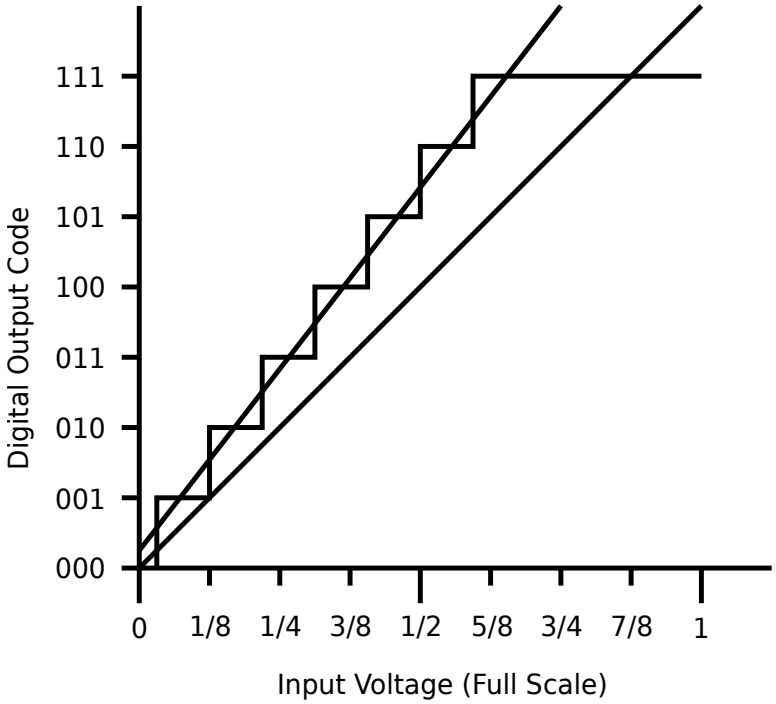

Figure 2.3: ADC Transfer Function with Offset and Gain Error.

2.1.2 Nonidealities and Nonlinearity

The ideal analog to digital converter has a linear transfer function within $\pm\frac{1}{2}$ LSB as previously shown in Figure 2.1. Real converters produce errors due to non ideal devices and component mismatch. The most common types of errors are offset, gain, and nonlinearity. Offset error represents a fixed difference between the output and input signals. Gain error (or scaling factor) occurs when the slope of the real transfer function is different from the ideal transfer function. These two errors can be calibrated in hardware or software using $y = Ax + b$.

In an ideal case, each quantization step of a converter is equal to 1 LSB. However, in a real converter this is not always the case and any deviation from 1 LSB is measured as Differential Nonlinearity (DNL). The deterministic approach to measuring this error is defined as

$$\mathrm{DNL}[i] = \frac{V_{i+1} - V_i}{V_{LSB}} - 1 \tag{2.5}$$

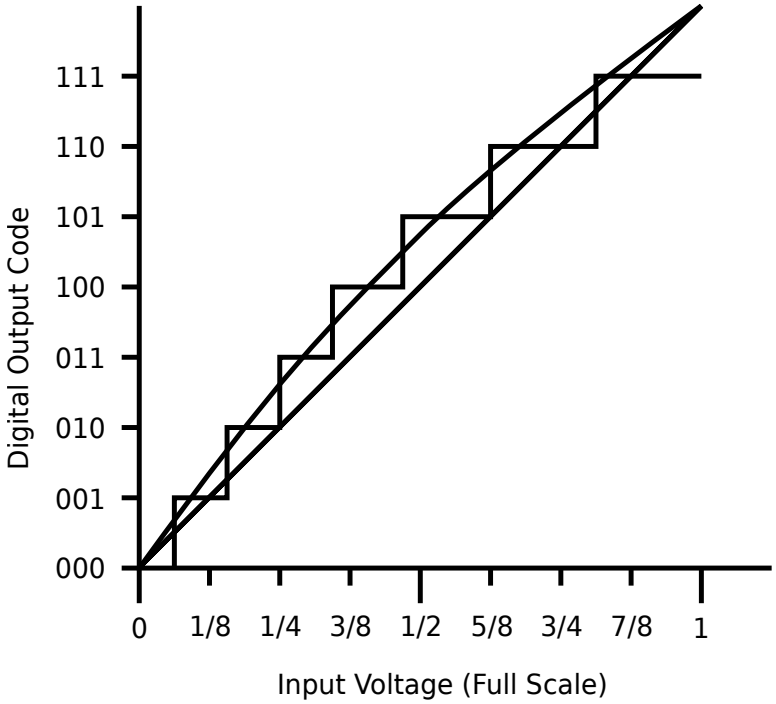

Figure 2.4: ADC Transfer Function with Nonlinearity.

where i is the quantization code level [13]. For example, if the output changes levels when the analog input increases by only 3/4 LSB, then the DNL for that transition is $-1/4$ LSB. Each transition has an associated DNL value that can be greater than or equal to -1 as seen in Equation 2.5. In the case where the DNL is -1, the transition never occurs. When the DNL exceeds +1, the quantization level is skipped completely, which results in a missing code [12, 13].

A different approach to finding the DNL of a converter is to perform a statistical analysis using an input signal with a known probability density function (pdf). The most common input signal used is a sinewave which has an ideal pdf of

$$f_S(v) = \frac{1}{\pi\sqrt{V_{pk}^2 - v^2}} \tag{2.6}$$

where v represents the input voltage and V_{pk} represents the amplitude [14]. By integrating the pdf over all code levels or bins, the ideal probability value for each bin can be calculated

as $P(i)$.

$$P[i] = \frac{1}{\pi}\left[\arcsin\left(\frac{V_{FS} \times (i - 2^{N-1})}{2^N V_{pk}}\right) - \arcsin\left(\frac{V_{FS} \times (i - 1 - 2^{N-1})}{2^N V_{pk}}\right)\right] \tag{2.7}$$

Equation 2.7 requires the code level, i, the resolution of the A/D, N, the full scale voltage for the converter, V_{FS}, and the amplitude of the sinewave, V_{pk}. The DNL for each code can be found by comparing the deviation between the measured histogram for each code of the converter, $MP[i]$, to the ideal histogram $P[i]$ [14].

$$\text{DNL}[i] \text{ (LSB)} = \frac{MP[i]}{P[i]} - 1 \tag{2.8}$$

The summation of DNL measurements is referred to as the Integral Nonlinearity (INL). Using the individual DNL measurements from Equation 2.5 or 2.8, the INL can be shown as the sum of the DNL errors.

$$\text{INL}[k] = \sum_{i=0}^{k} \text{DNL}[i] \tag{2.9}$$

This can be shown as a nonlinear transfer function of the A/D as shown in Figure 2.4. A plot of the INL can be made by taking the difference of the converter transfer function from the best linear fit of the two endpoints.

While individual DNL errors may be less than 1/2 LSB, the summation can result in INL error greater than 1/2 LSB. This is important to note because a 16-bit ADC that is still capable of resolving an LSB voltage change may have an overall accuracy of the converter less than 16-bit [15]. Figure 2.4 shows how the individual DNL errors can result in a large INL error.

Nonlinearity errors result in harmonic distortion in the frequency domain. This is measured as Total Harmonic Distortion (THD) which is the sum of the power of the harmonics divided by the power of the fundamentals. The THD measurement can be added to the SNR value to calculate the Signal to Noise and Distortion Ratio (SNDR). This SNDR value provides one type of measurement of the overall accuracy of the analog to digital converter.

The Effective Number of Bits (ENOB) is another method of measuring the overall

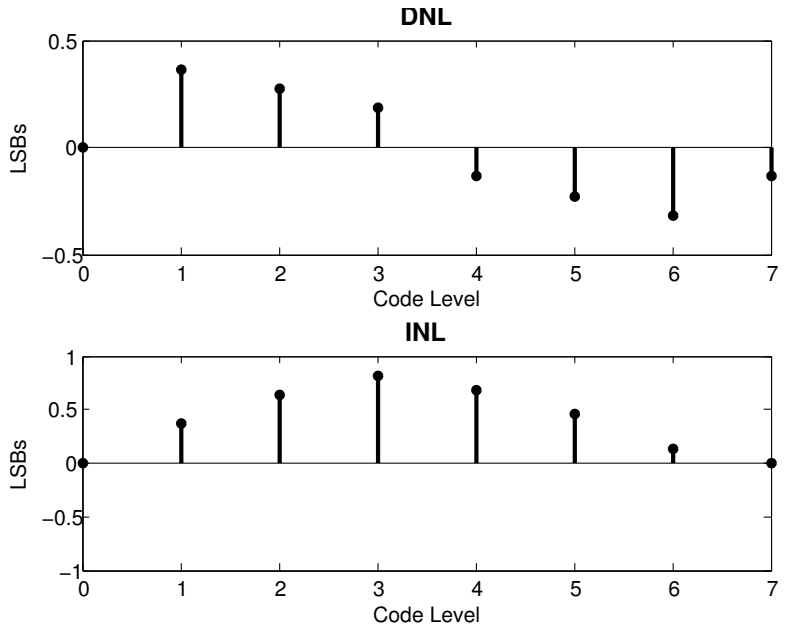

Figure 2.5: ADC DNL and INL

accuracy of an A/D. Using the ideal SNR Equation 2.4 the ENOB can be calculated from the measured SNDR.

$$\text{ENOB} = \frac{\text{SNDR} - 1.76}{6.02} \tag{2.10}$$

The overall accuracy presented in this format makes it easier to compare the real ADC's performance to its ideal resolution.

Aperture delay error is one more type of nonideality. There is always a delay from the time the ADC is told to capture a new sample to the time the converter acquires the sample. This delay is composed of a finite, deterministic delay and a varying, random delay. The finite delay component is due to such factors as signal path length and digital circuit propagation. The random component is called clock jitter, which is caused by phase noise in the sample clock generator circuit. The clock jitter component is a concern for all ADCs as too much jitter can affect the noise floor of the converter. The maximum clock jitter requirement is dependent on both the quantization level and the input signal frequency.

$$\Delta t = \frac{1}{2^N \pi f_o} \tag{2.11}$$

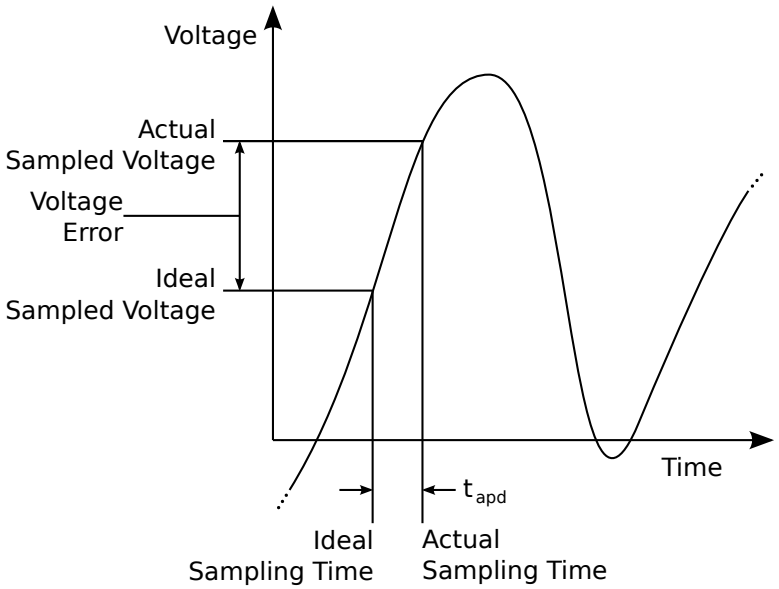

Figure 2.6: The Effects of Aperture Delay on the Sampled Voltage

The fixed deterministic component of aperture delay does not affect most converters. While clock jitter is a varying delay, the deterministic component of aperture delay does not affect the absolute sample period, since most ADCs are time invariant. The one exception to this is a time interleaved architecture. A time interleaved architecture is made of multiple ADC channels to create a single, high-speed converter system. Any difference in the signal paths between each ADC channel in a TI system will result in different fixed aperture delays. This results in a master sampling period that changes based on which ADC channel is used. The effects of this fixed delay on a TI architecture will be discussed further in Chapter 3.

2.2 ADC Architectures

Nyquist rate converters are designed using a variety of different architectures. Each type of architecture design has its advantages and disadvantages which makes them suitable for different applications. As stated earlier, Nyquist rate architectures have a common trade-off between sampling rate and resolution. This makes architectures with high speeds and low resolutions well suited for applications such as video imaging, while high resolution paired with slower speeds are more suited for measurement systems.

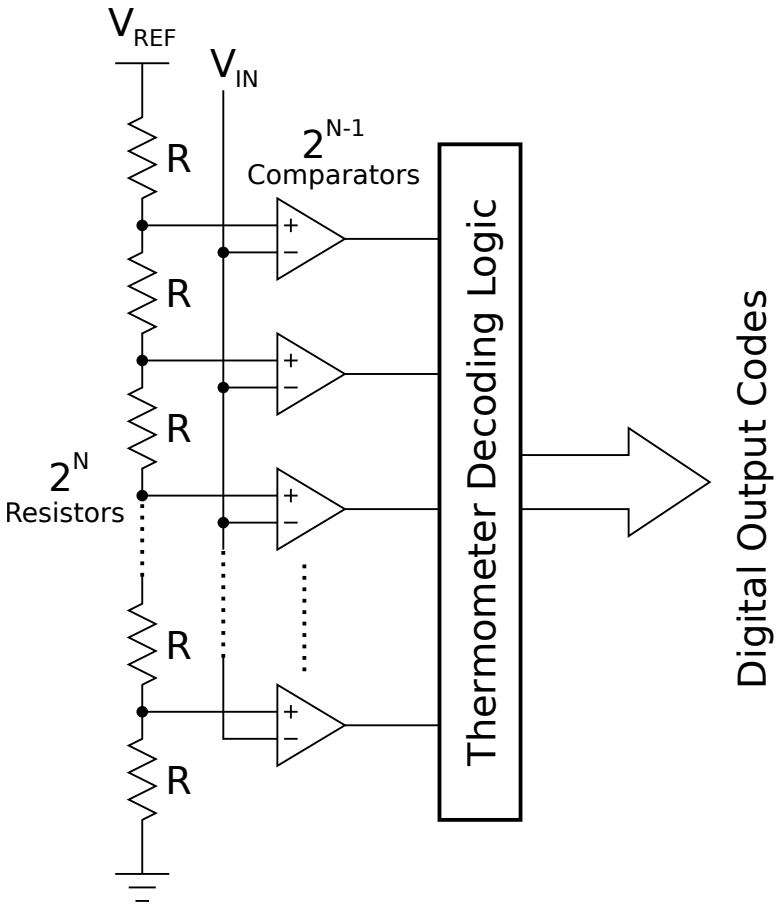

Figure 2.7: Block Diagram of a Flash ADC

2.2.1 Flash Architecture

The flash ADC architecture is one of the more commonly used converters [2, 7]. The basic design of the flash A/D uses a comparator for every quantization level in order to compare an input voltage with a series of reference voltages. These voltages are typically generated using a single reference input and a voltage divider resistor ladder. The analog input signal is then fed to the negative input of each converter while the reference voltage is fed to the positive input. This results in a thermometer code output, where all comparators with reference voltages greater than the input voltage will output a logic '1' and vice versa. A digital decoder circuit is required to translate the thermometer code into a binary weighted output code [5, 2].

This type of setup requires $2^N - 1$ comparators, making a high-resolution flash ADC

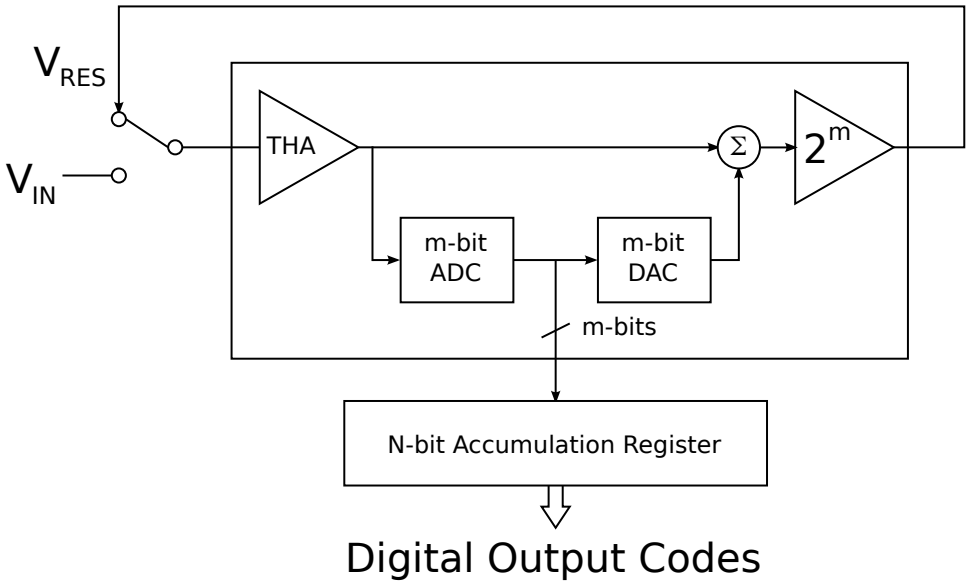

Figure 2.8: Block Diagram of a Cyclic ADC

impractical. For example, an 8-bit converter would require 255 comparators, placing large demands on both die area and power consumption. However, the converter speed is directly dependent on the propagation delay from the analog input signal to the output of the comparators (and subsequent thermometer to binary logic). For this reason, normal flash ADCs are used in applications such as video imaging and wide band radio transceivers where speed is preferred over resolution [15, 16, 6].

2.2.2 Cyclic Architecture

While the flash ADC is useful on its own, there are several architectures that use the flash to achieve higher resolutions. The cyclic ADC uses a low resolution flash converter with a negative feedback loop that operates over multiple internal cycles. The block diagram in Figure 2.8 shows the function of the cyclic converter [12].

For the first cycle, the frontend multiplexer switch selects the analog input voltage to be held on the track and hold circuit. This voltage held on the track and hold circuit is quantized by a small resolution, m-bit flash converter while simultaneously fed to a subtracter circuit. The m-bit digital code from the flash is then sent to a m-bit DAC. The analog voltage from the DAC is subtracted from the analog voltage held by the track and

hold circuit which, generates a small residue voltage. This small voltage is then amplified with a gain of 2^m to produce a larger residue voltage, V_{RES}. For the next cycle, the frontend switch is set to sample V_{RES} on the track and hold circuit. This process from the previous cycle is repeated for a total of n cycles. Using this method, the converter generates n number of m-bit digital codes to create a single N-bit output code [12, 5]. In more recent systems, there is at least a 1-bit overlap in the m-bit codes between cycles for redundancy. To achieve this overlap, the amplifier stage has a gain of less than 2^m (typically $\approx 1.8^m$) [17]. The result is a N-bit converter output code that is less than $n \times m$.

The primary advantage of the cyclic ADC over a straight flash converter is that it uses less comparators and achieves a higher resolution. For an N-bit cyclic ADC, the number of comparators required is only $2^m - 1$. This requires less die area and power than an equivalent N-bit flash converter. In addition, a full resolution of more than 8-bits can be achieved by increasing the number of cycles used for each conversion. The main drawback for this architecture is the converter speed. Since a single conversion requires multiple cycles, the sample period is dependent on the signal propagation times the number of cycles performed. This makes the cyclic ADC well suited to moderate speed, moderate resolution applications that require a small die area and low power [5, 17].

2.2.3 Pipelined Architecture

The advantages of the cyclic ADC are improved upon by the pipelined architecture. A pipelined A/D breaks up a single conversion into multiple stages, similar to the way a pipelined microprocessor breaks up a single instruction execution. In this setup, the feedback loop from the cyclic ADC is removed and multiple copies or stages are used in a daisy chain array as shown in Figure 2.9 [2].

For example, the first stage samples the input signal with a track and hold circuit without the frontend selection switch. The path is the same as a cyclic ADC up until output of the residue amplifier. Instead of sending the V_{RES} voltage back to the beginning of the system, the voltage is sent to another stage that looks similar to the first stage. The frontend track and hold amplifier at the beginning of the first stage is now free to capture the next analog input sample while the second stage THA can sample the residue voltage

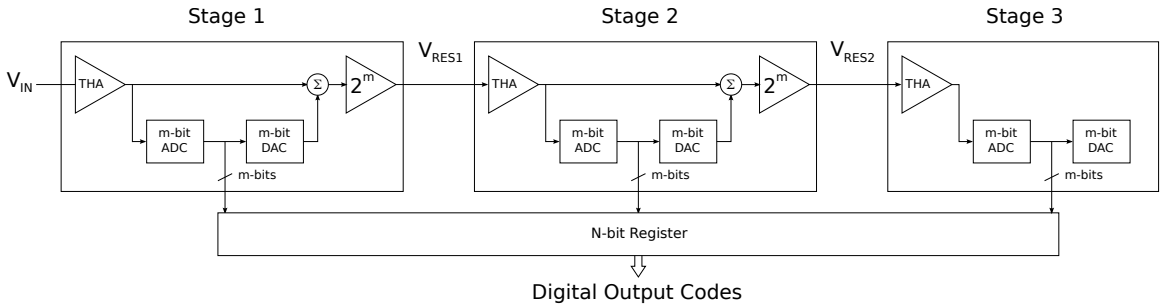

Figure 2.9: Block Diagram of a Pipelined ADC

from the first stage. The delay time from input to output for a N stage pipeline converter is about the same as a N cycle ADC. The sampling rate is only limited by the slowest stage in the pipeline converter as opposed to the sum of all stages. The pipeline converter is capable of moderate to high speed sample rates and moderate resolutions [7]. This architecture is common in such applications as Charge Coupled Device (CCD) imaging and ultrasonic medical imaging [2].

2.2.4 SAR Architecture

Unlike the previous architectures the Successive Approximation Register (SAR) ADC does not use a flash sub-converter. The SAR instead uses a full resolution DAC and one comparator to perform a binary search algorithm [5, 6]. The SAR iterates through each bit of the DAC to find an analog voltage that is approximately equal to the sampled input voltage. Most SARs use a binary weighted charge balancing architecture for the D/A since capacitors are easier to match than resistors and offer better noise performance. A comparator is used to determine if the DAC voltage is greater or less than the original input voltage. The decision of the comparator is used to determine the next bit in the DAC code. Once the SAR has reached the last bit, the complete digital word is used as the ADC's output.

Figure 2.10 shows the first couple of steps in the successive approximation process for a unipolar, charge balancing SAR converter [5]. For the first step, the analog input voltage is sampled onto all of the capacitors in the switch-cap network while the comparator is reset to its threshold voltage (ground in this case). During the hold mode, the comparator output

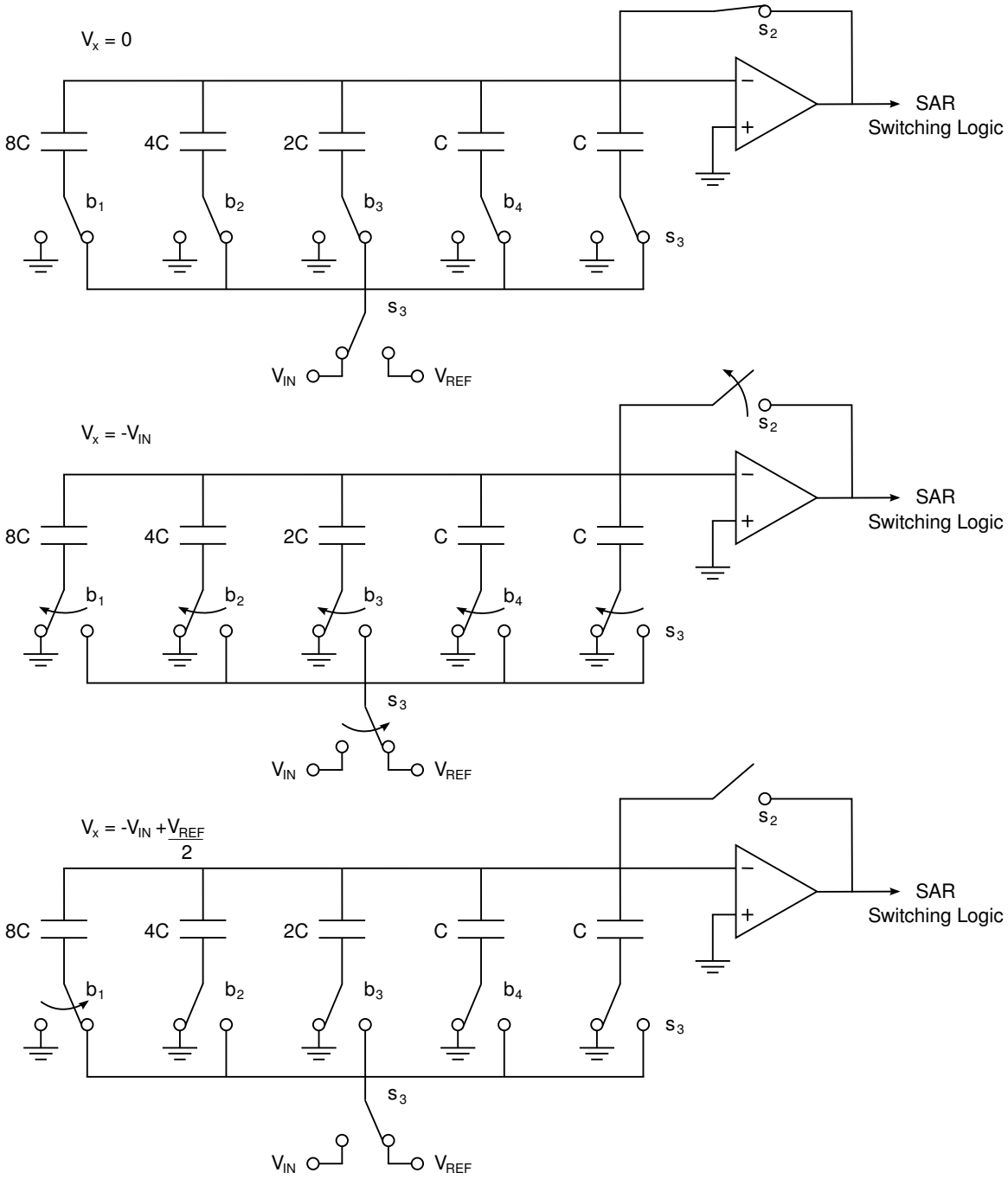

Figure 2.10: Basic Operation of a SAR ADC

is released and all capacitors are switched to ground. This forces the DAC output voltage (V_x) to go to negative V_{IN}. At the same time, s_1 is switched to V_{REF}. The next step is the start of the bit-cycling mode, which is where the binary search takes place. In the scenario shown in Figure 2.10 the comparator output during the hold mode was a logic '1' decision. Using this information, the first capacitor which represents the MSB is switched to V_{REF}. This changes the DAC voltage to $-V_{IN} + \frac{V_{REF}}{2}$ resulting in a new comparator decision for the next DAC bit. This process is repeated until all DAC bits have been set and sent as the A/D digital output code [7, 6, 5].

Iterating through the sample, hold, and bit cycling modes requires a longer sample period than flash, cyclic, and pipeline ADCs [6]. This makes the SAR converter a moderate speed ADC, but it is capable of higher resolutions than the flash based architectures. In the cyclic and pipeline converter, a residue voltage must be amplified by a fixed gain amplifier before being sent to the next cycle or stage. This amplifier circuit will add noise to the residue voltage with a cumulative effect for each cycle or stage. This places a practical limit on the number of cycles or stages used, thereby restricting the full A/D resolution to moderate levels. The noise in a charge balancing, successive approximation converter is limited by the capacitor DAC network and the preamp for the comparator. Charge balancing SAR converters are more power efficient that flash or pipeline converters due to the low number of active components and the elimination of a resistor ladder [7, 6]. This makes the SAR converter well suited for both low power and higher resolution applications such as portable instrumentation and data acquisition [2, 7].

2.2.5 Time Interleaved Architecture

The time interleaved architecture is a system that uses multiple full resolution ADCs running on a subdivided master sample clock [2, 3]. For a given M:1 time interleaved A/D, there are M converters which operate at sample rate of $\frac{f_S}{M}$. Figure 2.11 shows the operation of a 2:1 interleaved converter. Two separate ADCs operate at an individual sample rate of $\frac{f_S}{2}$. One converter processes the odd samples of the input while the other converter processes the even samples. This allows for the system sample rate, f_S, to be twice the maximum sample rate of the two sub-converters [3].

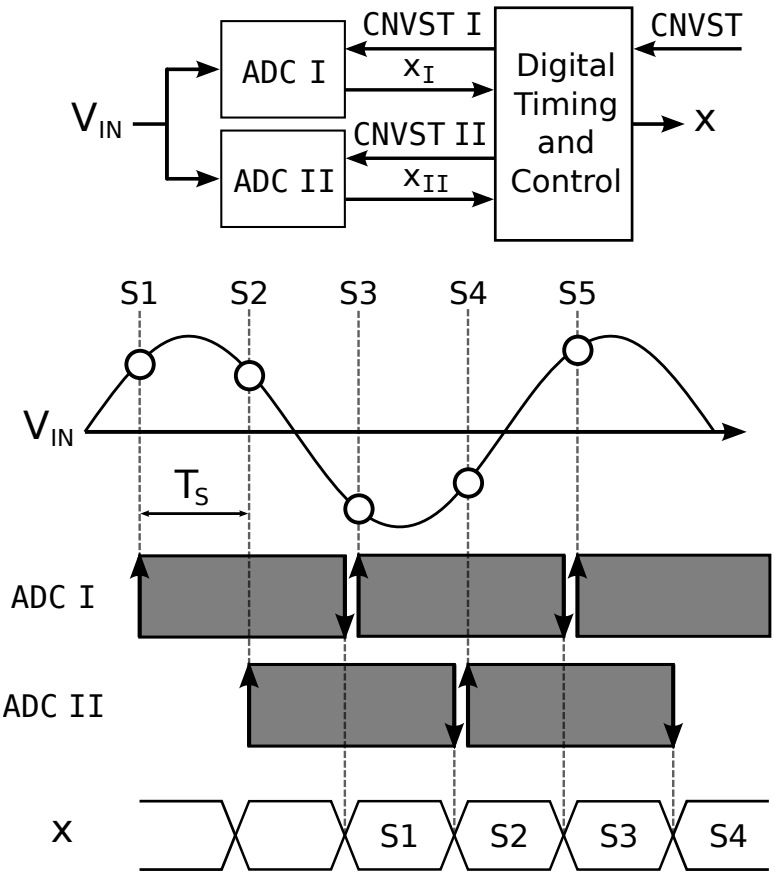

Figure 2.11: Block Diagram of a Time Interleaved ADC

The 2:1 example can be extended to any M:1 interleaved system that can operate at M times the sample rate of its M interleaved converters. The primary advantage of a time interleaved system is the increase in speed without sacrificing resolution. This makes them well suited for applications that require high speed and high resolution converters such as instrumentation and measurement. Unfortunately, linear errors that were trivial in a single converter system present a new set of problems in the interleaved architecture.

As discussed earlier gain and offset errors can be easily calibrated out for a single converter [2, 3]. In an interleaved system, the individual converters have each have their own unique gain and offset errors. The mismatch in these errors between each converter channel can no longer be modeled as $y = Ax+b$. The third linear mismatch error is the deterministic component of aperture delay. As previously mentioned, a fixed delay in the sample time of a single converter does not affect the output code. However, in the interleaved architecture, mismatch in signal path lengths leads to differences in the aperture delay between each sub-converter channel [4, 3]. Since the delay is no longer fixed for each sample time, this results in a varying sample period. The effects of gain, offset, and aperture delay channel mismatch will be discussed further in Chapter 3.

Chapter 3

Interleaved and SAR Architectures

The two ADC architectures that are the focus of this research are the time interleaved and successive approximation register converters. Both architectures can provide high resolution samples with moderate to high speed conversions. However, as previously discussed, each type has unique non-idealities which can limit the accuracy of the ADC. Section 3.1 describes the interleaved ADC and the effects of gain, offset and aperture delay on the output signal. Section 3.2 describes the charge balancing, SAR converter and the effects of capacitor mismatch.

3.1 The Time Interleaved Converter

As discussed earlier, the time interleaved converter is an effective architecture for achieving high speed conversions using M full resolution subconverters. Each Nyquist-rate subconverter operates at a rate of f_S/M, allowing the full system to run M times faster than the maximum sample rate of a single subconverter. However, differences between each subconverter channel leads to errors in a TI system. As device technology scales deeper into the nanoscale region, physically matching subconverter channels becomes a critical issue. Because of this, the error sources in a TI structure have been studied to determine their cause and how to minimize their effects.

The three main error contributions due to channel mismatch are offset, gain and aperture or timing delay. The effects of all three errors on the digital output code for a single channel

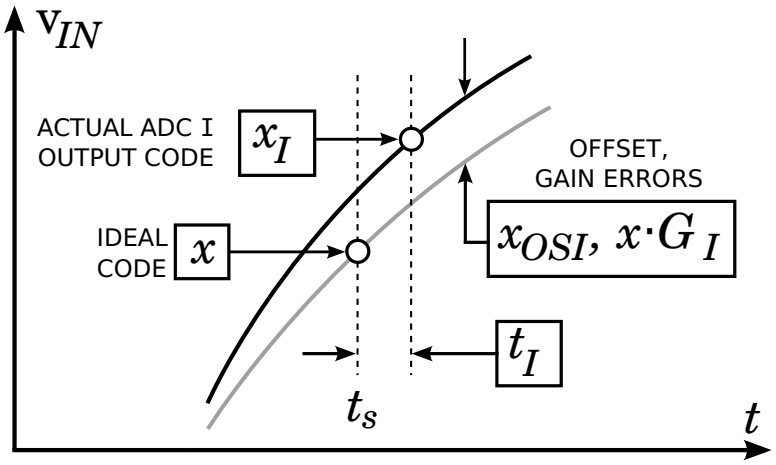

Figure 3.1: Effects of time interleaved channel mismatch on an input signal

can be modeled as (3.1).

$$x_I = x + x_{OSI} + xG_I + \dot{x}t_I \tag{3.1}$$

The three error coefficients for channel 1 are x_{OSI} as the offset error, G_I as the gain error, and t_I for the aperture delay. The ideal output code is represented by x, while $\dot{x}$ represents the derivative and x_I represents the non-ideal output code from channel 1. Using Equation (3.1), one can see how the effects of gain mismatch are proportional to the magnitude of the input signal and the effects of aperture delay are proportional to the derivative.

Figure 3.1 shows a graphical example of (3.1). The solid black curve represents the signal with only offset and gain distortion. The delay in sampling time can be seen as difference in V_{IN} that is dependent on the derivative of the input signal as described by (3.1).

To understand the effects of this channel mismatch, the time interleaved converter should be analyzed in the frequency domain. The work presented in [4] quantifies the relationship between the effects of the channel mismatch and the error spurs in a FFT.

3.1.1 Offset Mismatch Error

The magnitude of the offset mismatch spurs can be found in (3.2). The location of the spurs in the frequency domain is dependent on the frequency components of the input signal and the interleaving ratio of the TI converter. For a 4:1 system, the offset spurs would occur

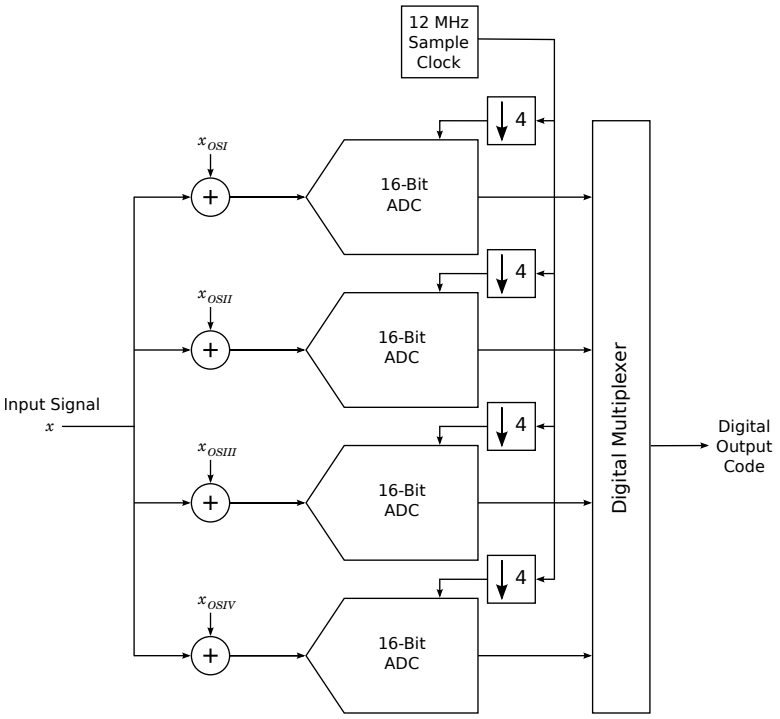

Figure 3.2: Block Diagram of 4:1 Time Interleaved Converter with Offset Error Introduced

at $f_S/4$ and $f_S/2$.

$$\text{Offset Spur} = 20 \log \left(\frac{\Delta OS}{V_{FS}} \right) \tag{3.2}$$

Equation (3.2) shows how the spurs are proportional to the offset mismatch, ΔOS (in volts), with relation to the full scale voltage, V_{FS}.

A simulation of a 4:1 TI system with only offset channel mismatch provides a clear example of its effect on the output spectrum. For this setup, all gain and aperture delay errors are set to zero and the offset mismatch is limited to $\pm 0.1\%$. While the mean or system offset is 215 μV, it has no effect on the image spurs. A block diagram of the configuration is shown in Figure 3.2. A 317 kHz, full scale input sinewave is fed to four A/D converters. A unique offset voltage is added to the signal before the input stage of each ADC. All four converters have a 16-bit resolution and operate at 3 MHz each to produce a master sampling frequency of 12 MHz.

The simulation results are presented in Figure 3.3. The spectrum plot is normalized to

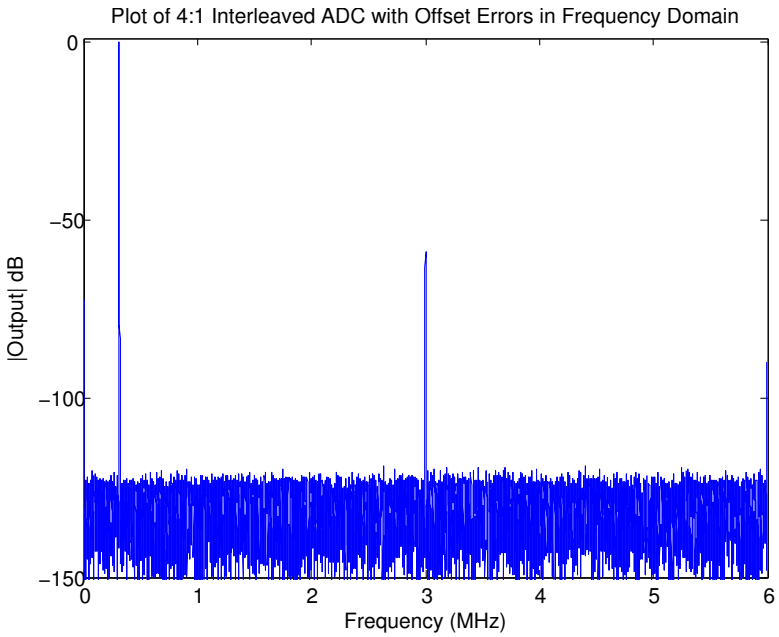

Figure 3.3: 4:1 Time Interleaved Output Spectrum with Offset Errors

the 317 kHz input sinewave at 0 dB. The plot shows two errors spurs at 3 MHz and 6 MHz. The Spurious Free Dynamic Range is the ratio of the input signal to the largest distortion or error spur. In this simulation the SFDR is 58.81 dB from the input signal at 317 kHz to the offset error spur at 3 MHz.

3.1.2 Gain Mismatch Error

The effects of gain mismatch error on the output spectrum of a TI system is described by (3.3). In a 4:1 time interleaved converter, the image spurs due to gain mismatch error would occur at $f_S/2 - f_{in}$ and $f_S/4 \pm f_{in}$ [4]. The gain mismatch error is the full scale voltage ratio between two channels as defined by (3.4).

$$\text{Gain Spur} = 20\log\left(\frac{G_{err}}{2 \cdot V_{FS}}\right) \tag{3.3}$$

$$G_{err} = \left|1 - \frac{V_{FSA}}{V_{FSB}}\right| \tag{3.4}$$

This simulation of a time interleaved system with a 4:1 ratio used a gain mismatch error

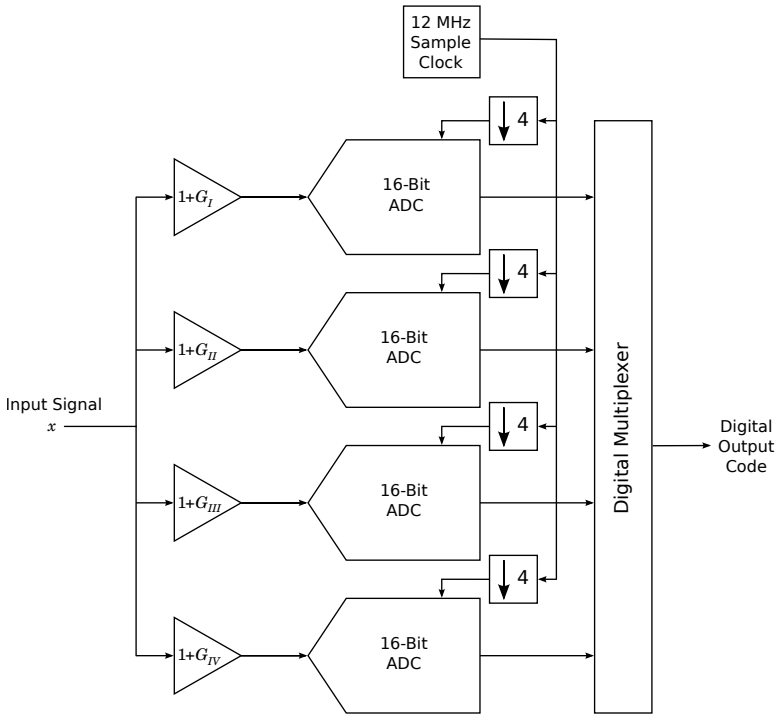

Figure 3.4: Block Diagram of 4:1 Time Interleaved Converter with Gain Error Introduced

of $\pm 0.2\%$ and all other errors set to zero. The sampling rate used was 12 MHz with an input signal of 317 kHz. The sinewave is fed to four gain stages where each stage has a unique gain of 1 ± 0.002. The gain stage outputs are then fed to their respective 16-bit converters.

The simulation results showed a system gain error of -617 mV. The image spurs can be seen in Figure 3.5 at 2683 kHz, 3317 kHz and 5683 kHz. The largest spurs are the two at 2683 kHz and 3317 kHz which gives a SFDR of 65.15 dB.

3.1.3 Aperture Delay Mismatch Error

The effect of aperture delay mismatch error on the output spectrum can be represented by (3.5). A 4:1 TI system has image spurs in the same locations as those due to gain mismatch error. When both gain and aperture delay mismatch errors are present, the Root Mean Square value is shown by (3.6) [4]. The magnitude of the image spurs due to aperture delay mismatch are dependent on both the time delay and input frequency.

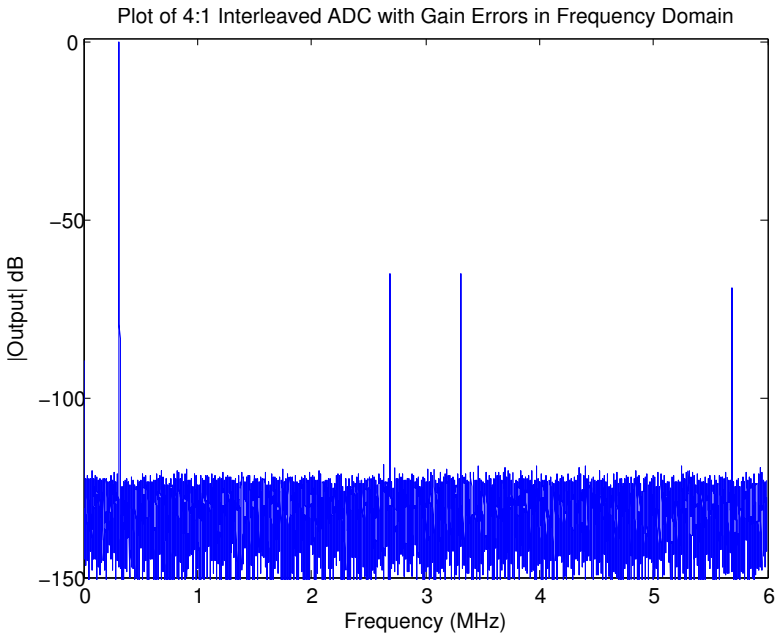

Figure 3.5: 4:1 Time Interleaved Output Spectrum with Gain Errors

$$\text{Aperture Delay Spur} = 20\log\left(\frac{2\pi f_{in}\Delta t_{apd}}{2}\right) \tag{3.5}$$

$$\text{Gain \& Aperture Delay Spurs} = 20\log\left(\sqrt{\left(\frac{G_{err}}{2}\right)^2 + \left(\frac{2\pi f_{in}\Delta t_{apd}}{2}\right)^2}\right) \tag{3.6}$$

A simulation of a 4:1 TI converter was used to show the effect of aperture delay. The setup for this simulation was similar to the one used for gain mismatch. A single input signal was fed to four individual delay stages. Each stage had a delay between 0 and 50 ps with a mean aperture delay error of 23 ps. The delayed signals were then sent to their respective, 16-bit converters. As before, the sampling rate was 12 MHz, but this time the input frequency was set to 3175 MHz. As previously mentioned, the effects of aperture delay are proportional to the frequency of the input signal, as seen in (3.5). Therefore, an input with a higher frequency was used to show the effect of aperture delay for signals close to Nyquist.

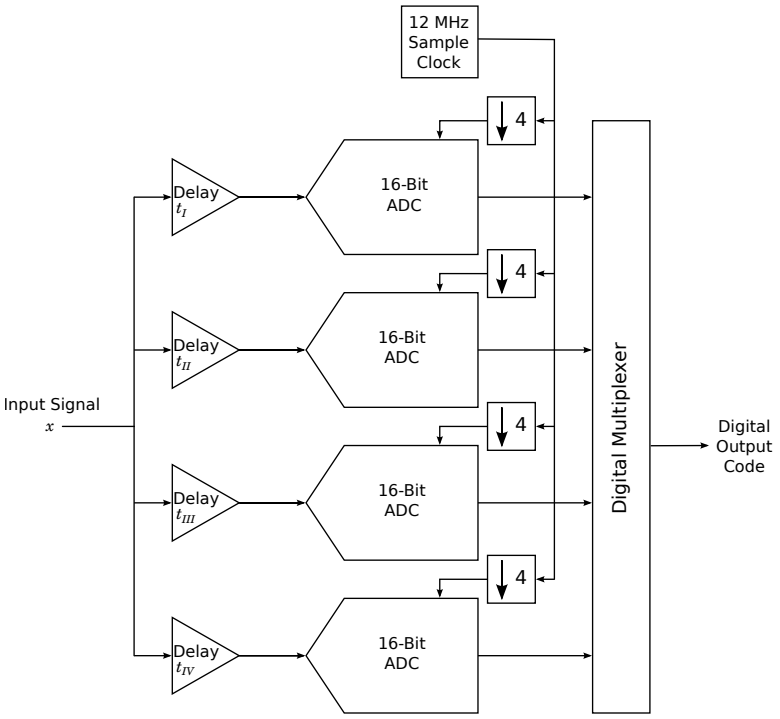

Figure 3.6: Block Diagram of 4:1 Time Interleaved Converter with Aperture Delay Error Introduced

Figure 3.7 shows the output spectrum of the simulated TI converter with spurs at 175 kHz, 2825 kHz and 5825 kHz. The largest spur at 2825 kHz produces a SFDR of approximately 62.29 dB.

The total power of all image spurs due to offset, gain, and aperture delay can be found using the Root Sum Square (RSS) as shown in (3.7).

$$\text{Total Spurs} = 20\log\left(\sqrt{\sum_{i=1}^{M}\left(\frac{\Delta OSi}{V_{FS}}\right)^2+\sum_{i=1}^{M}\left(\frac{G_{erri}}{2}\right)^2+\sum_{i=1}^{M}\left(\frac{2\pi f_{in}\Delta t_{apdi}}{2}\right)^2}\right) \quad (3.7)$$

For a given M:1 interleaved converter, the square of each mismatch error for each, i^{th} channel is summed before taking the square root. The total power is used to determine how much calibration is necessary to reduce the errors enough for an N-bit converter. For example, using the SNDR from (2.4), a 16-bit converter can have a maximum SNDR of 98 dB. Therefore, the total power of all spurs due to mismatch error after calibration must be less than -98 dB.

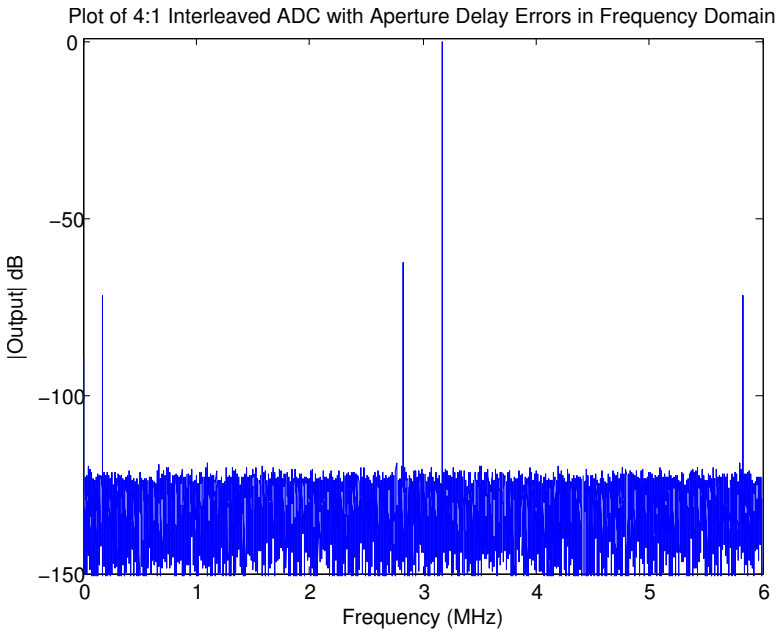

Figure 3.7: 4:1 Time Interleaved Output Spectrum with Aperture Delay Errors

3.1.4 Bandwidth Mismatch Error

Channel bandwidth mismatch is a type of nonlinear error in TI converters. An in-depth analysis of effects due to this error is presented in [18]. The T/H circuit in each channel can be approximated as a first-order system in the frequency domain as

$$H_k(j2\pi f) = 1/(1 + jf/f_{c(i)}) \tag{3.8}$$

where $f_{c(i)}$ is the mismatch of the bandwidth in each channel ($i = 1, 2, \cdots, M$). The output signal has two error components due to bandwidth mismatch, AC gain mismatch G_i and AC phase mismatch θ_i [18].

$$G_i = \frac{1}{\sqrt{1 + (f_{in}/f_{c(i)})}} \tag{3.9}$$

$$\theta_i = -\arctan(f_{in}/f_{c(i)}) \tag{3.10}$$

It is important to note that AC gain and phase depend on bandwidth mismatch and the input frequency f_{in}, unlike the linear gain and aperture delay mismatch. [18] presents a

detailed analysis of bandwidth mismatch in a 4:1 time interleaved system. It shows that the SNR in a 4:1 is

$$\text{SNR (dB)} = 20\log\frac{B_{s4}^2}{B_{n1}^2 + B_{n2}^2 + B_{n3}^2} \tag{3.11}$$

where

$$B_{s4} = \sqrt{B_{sc}^2 + B_{ss}^2} \quad B_{n1} = \sqrt{B_{n1c}^2 + B_{n2s}^2} \quad B_{n2} = \sqrt{B_{n2c}^2 + B_{n2s}^2} \quad B_{n3} = \sqrt{B_{n3c}^2 + B_{n3s}^2}$$

and the the values B_{sc}, B_{ss}, B_{n1c}, B_{n1s}, B_{n2c}, B_{n2s}, B_{n3c} and B_{n3s} are defined below [18].

$$\begin{aligned} B_{sc} = \frac{1}{4}\Big(&+ (G_1+G_3)\cos(\theta)\cos(\theta_{13}) + (G_2+G_4)\cos(\theta)\cos(\theta_{24}) \\ &- (G_1-G_3)\sin(\theta)\sin(\theta_{13}) + (G_2-G_4)\sin(\theta)\sin(\theta_{24})\Big) \end{aligned} \tag{3.12}$$

$$\begin{aligned} B_{ss} = \frac{1}{4}\Big(&- (G_1+G_3)\sin(\theta)\cos(\theta_{13}) + (G_2+G_4)\sin(\theta)\cos(\theta_{24}) \\ &- (G_1-G_3)\cos(\theta)\sin(\theta_{13}) - (G_2-G_4)\cos(\theta)\sin(\theta_{24})\Big) \end{aligned} \tag{3.13}$$

$$\begin{aligned} B_{n1c} = \frac{1}{4}\Big(&- (G_1+G_3)\sin(\theta)\sin(\theta_{13}) + (G_2+G_4)\cos(\theta)\sin(\theta_{24}) \\ &+ (G_1-G_3)\cos(\theta)\cos(\theta_{13}) - (G_2-G_4)\sin(\theta)\cos(\theta_{24})\Big) \end{aligned} \tag{3.14}$$

$$\begin{aligned} B_{n1s} = \frac{1}{4}\Big(&- (G_1+G_3)\cos(\theta)\sin(\theta_{13}) + (G_2+G_4)\sin(\theta)\sin(\theta_{24}) \\ &- (G_1-G_3)\sin(\theta)\cos(\theta_{13}) + (G_2-G_4)\cos(\theta)\cos(\theta_{24})\Big) \end{aligned} \tag{3.15}$$

$$\begin{aligned} B_{n2c} = \frac{1}{4}\Big(&+ (G_1+G_3)\cos(\theta)\cos(\theta_{13}) - (G_2+G_4)\cos(\theta)\cos(\theta_{24}) \\ &- (G_1-G_3)\sin(\theta)\sin(\theta_{13}) - (G_2-G_4)\sin(\theta)\sin(\theta_{24})\Big) \end{aligned} \tag{3.16}$$

$$\begin{aligned} B_{n2s} = \frac{1}{4}\Big(&- (G_1+G_3)\sin(\theta)\cos(\theta_{13}) - (G_2+G_4)\sin(\theta)\cos(\theta_{24}) \\ &- (G_1-G_3)\cos(\theta)\sin(\theta_{13}) + (G_2-G_4)\cos(\theta)\sin(\theta_{24})\Big) \end{aligned} \tag{3.17}$$

$$B_{n3c} = \frac{1}{4}\Big(-(G_1+G_3)\sin(\theta)\sin(\theta_{13}) - (G_2+G_4)\cos(\theta)\sin(\theta_{24}) + (G_1-G_3)\cos(\theta)\cos(\theta_{13}) + (G_2-G_4)\sin(\theta)\cos(\theta_{24})\Big) \tag{3.18}$$

$$B_{n3s} = \frac{1}{4}\Big(-(G_1+G_3)\cos(\theta)\sin(\theta_{13}) - (G_2+G_4)\sin(\theta)\sin(\theta_{24}) - (G_1-G_3)\sin(\theta)\cos(\theta_{13}) - (G_2-G_4)\cos(\theta)\cos(\theta_{24})\Big) \tag{3.19}$$

The values G_1 through G_4 and θ_1 and θ_4 are defined in 3.9 and 3.10 respectively. The values θ, θ_{13} and θ_{24} in (3.12) through (3.19) are defined as

$$\theta = \frac{1}{4}(\theta_1 - \theta_2 + \theta_3 - \theta_4) \quad \theta_{13} = \frac{1}{2}(\theta_1 - \theta_3) \quad \theta_{24} = \frac{1}{2}(\theta_2 - \theta_4). \tag{3.20}$$

To determine how much bandwidth mismatch is acceptable for an N-bit converter, (2.4) must be used with (3.11). Again, for a 16-bit converter the SNDR must be better than 98 dB, however, the SNDR due to bandwidth mismatch is dependent on the bandwidth of each channel and the input frequency f_{in}. Therefore, a numerical analysis of bandwidth mismatch can show SNDR performance over a range of input frequencies. Figure 3.8 shows SNDR performance versus channel bandwidth mismatch for multiple input frequencies. To perform this analysis a set of restrictions were placed on the variables f_{c1}, f_{c2}, f_{c3} and f_{c4}. The channel bandwidths were uniformly distributed with a mean bandwidth $f_{c\mu}$, of 50 MHz and a standard deviation σ_{f_c}, determined by

$$\sigma_{f_c} = \sqrt{\frac{(f_{c1}-f_{c\mu})^2 + (f_{c2}-f_{c\mu})^2 + (f_{c3}-f_{c\mu})^2 + (f_{c4}-f_{c\mu})^2}{4}}. \tag{3.21}$$

The master sampling frequency was 12 MHz and four input frequencies were tested, 1.2 MHz, 6 MHz, 12 MHz, and 18 MHz. According to the plot in Figure 3.8, a 16-bit TI ADC with an average bandwidth of 50 MHz and an input frequency near Nyquist $f_S/2$, can tolerate a maximum bandwidth mismatch of $\pm 3\%$.

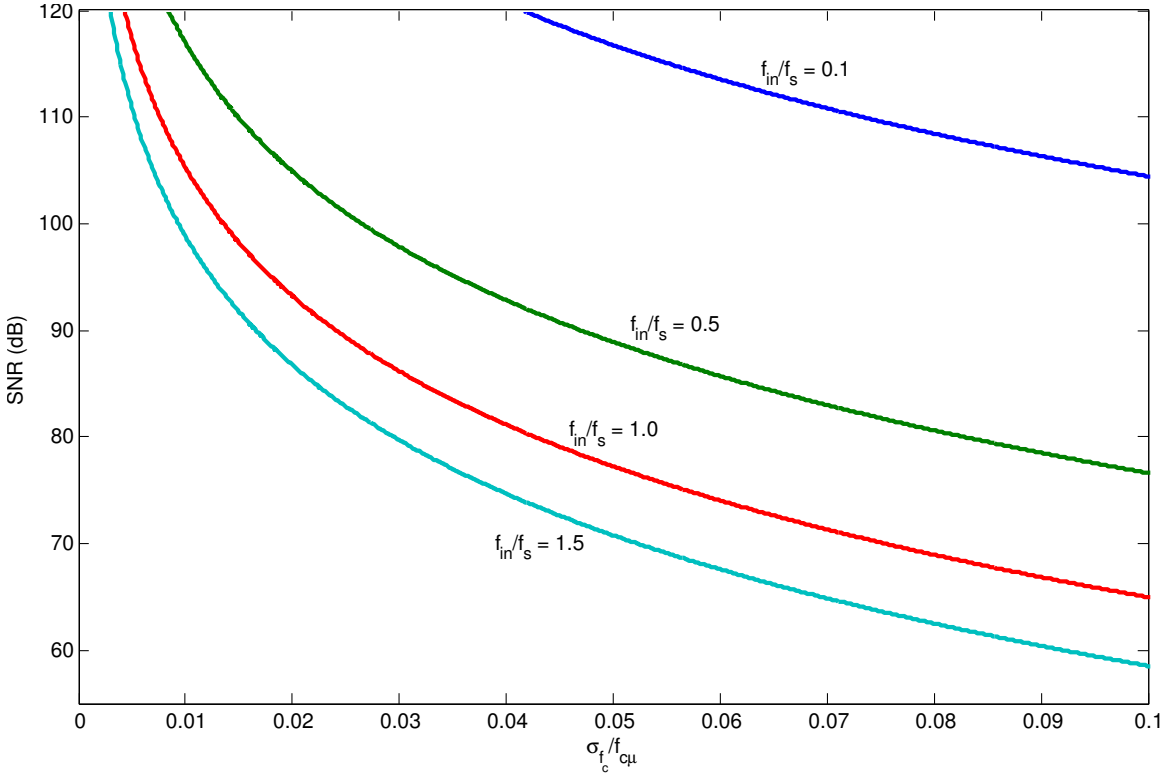

Figure 3.8: Plot of Bandwidth Mismatch for Multiple Input Frequencies

3.2 The SAR Converter

Successive approximation ADCs provide high resolution samples at moderate conversion speeds with moderate circuit complexity [5]. The SAR converter requires only one precision analog comparator, a DAC network with a sample and hold circuit, as well as supporting digital logic for the binary search algorithm.

One of the more common designs for a successive approximation ADCs uses a differential, charge balancing DAC network. The benefits of using capacitors in the DAC circuit is two fold. In an IC layout, capacitors are better than resistors at both device matching as well as noise performance [12, 5]. The use of a switched-capacitor DAC also eliminates the need for a separate sample and hold circuit. A fully differential, charge balancing SAR is presented as an example, in the following section. Section 3.2.2 describes the non-idealites and errors that are found in this type of A/D.

3.2.1 Operation of a Differential SAR

The basic operation of a differential SAR is similar to the operation of the single-ended converter described in Section 2.2.4. Two switched-cap DAC networks are used, one positive and one negative. Figure 3.9 shows a block diagram of the differential converter with the two DAC circuits. Instead of comparing a single DAC voltage to a fixed common-mode DC voltage, the positive DAC voltage is compared to the negative side. This allows for the sampling of a differential input while making use of a high Common Mode Rejection Ratio (CMRR). The differential output of the comparator is fed into digital logic that makes the appropriate capacitor selection in the DACs. With each bit decision, the differential voltage is driven to zero, balancing the charge between the positive and negative DACs.

Figure 3.10 shows the first three steps of the differential SAR converter. There are a few differences between the single-ended and differential circuits. During the sample phase, the capacitors are switched to their respective input voltages. In this case, the comparator is fully differential and there is no closed-loop feedback for the sample mode. Instead, the two DAC voltages are shorted to a common mode voltage. For the hold phase, the capacitor network is switched to the common mode while the differential DAC voltage is allowed to

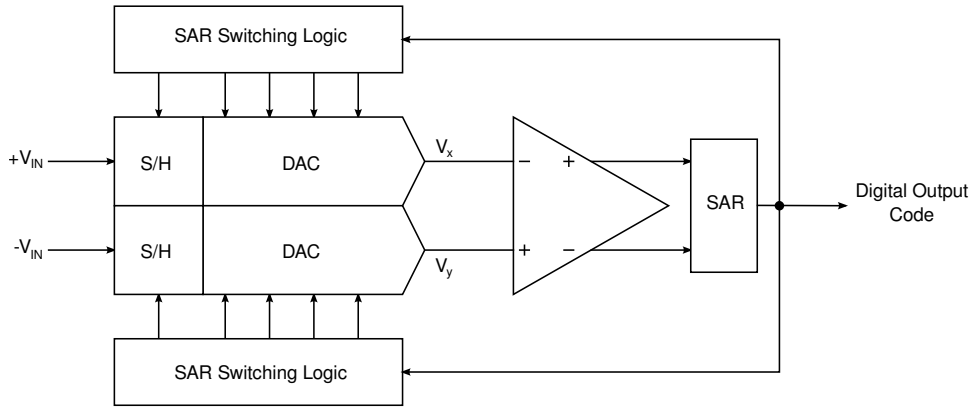

Figure 3.9: Block Diagram of a Differential SAR

Table 3.1: Table of DAC Voltages for a Single Conversion

Cycle	V_x **(V)**	V_y **(V)**	**Bit Decision**
Sample	0.900	0.900	N/A
Hold	1.375	0.425	+1
Bit 1	0.925	0.875	+1
Bit 2	0.700	1.100	-1
Bit 3	0.8125	0.9875	-1
Bit 4	0.8688	0.9313	N/A

settle to the sampled differential input voltage. For the bit cycling mode, the comparator decision is used to switch the capacitors used for that bit in each DAC.

To better illustrate the operation for a full conversion, Figure 3.11 shows the voltage waveforms from both DACs for a 4-bit SAR converter. The voltage reference used for this example was 1.8 V with a 0.9 V common mode and a differential input voltage of +0.95 V. The two DAC voltages, V_P and V_N are brought to 0.9 V for the sample mode, while the input side of the capacitors are shorted to the differential input voltage. For the hold mode, V_P is allowed to settle to 1.375 V while V_N is allowed to settle 0.425 V. At this point, the magnitude of the differential DAC voltage is equal to the magnitude of the sampled input voltage. The comparator then makes a +1 decision, switching bit one in the negative DAC to 0 V and the positive DAC to 1.8 V. This forces the differential DAC voltage to settle to a new value, 0.05 V. Once the DACs have enough time to settle, the comparator makes a new decision of +1, bringing V_P to 0.7 V and V_N to 1.1 V. Table 3.1 describes the remaining bit decisions and DAC voltages for the entire conversion cycle.

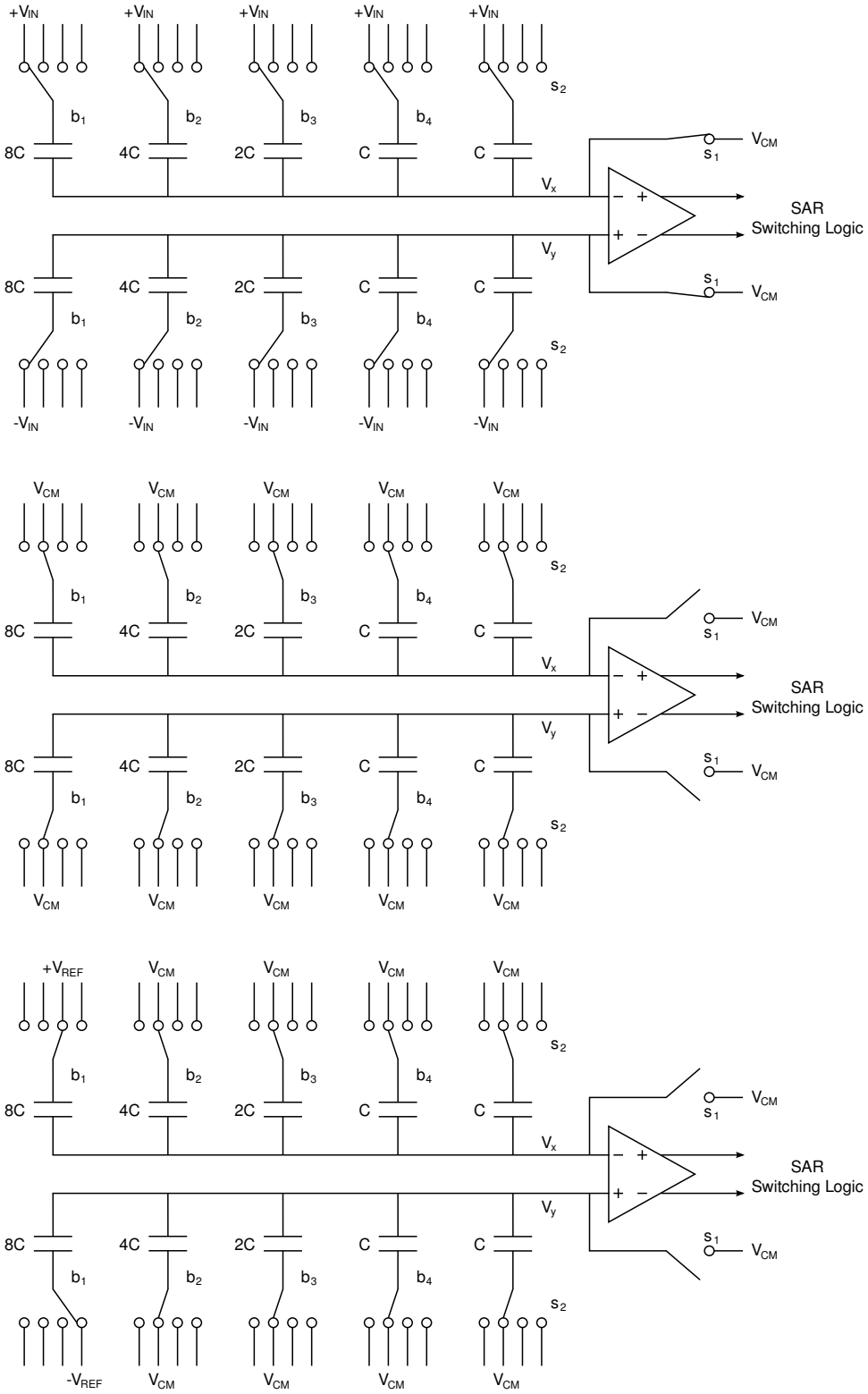

Figure 3.10: Basic Circuit Operation of a Differential SAR

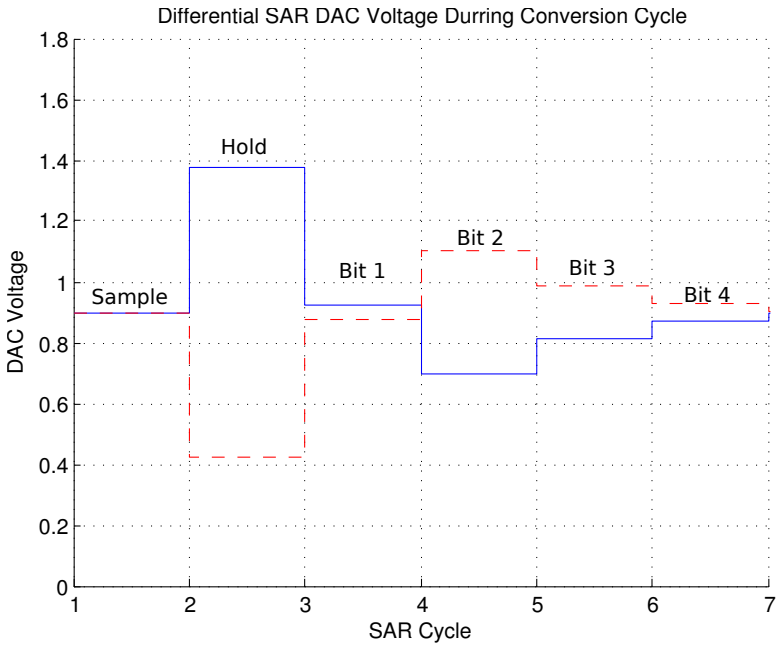

Figure 3.11: DAC Voltage Waveform of the Differential SAR

In the ideal case, the DAC voltages quickly settle to the correct values and the comparator always makes the right decision. In addition, the capacitor weights would always be perfectly matched to powers of 2. Unfortunately, this is never the case in the real world, and non-idealities with the comparator, DAC settling time, and capacitor mismatch results in output errors.

3.2.2 Non-Idealities in a SAR Converter

The two main types of errors in a SAR A/D are incorrect comparator decisions and nonlinearity. The sources for these errors can range from capacitor mismatch, voltage reference noise and distortion, to insufficient DAC settling time. These types of errors are difficult to account for outside of the SAR, so extra circuitry is usually added to correct for these errors internally.

3.2.3 Incorrect Comparator Decisions

Noise or distortion on the reference voltage side of the DAC network can result in an incorrect decision. The charge balancing network is a capacitively loaded circuit that

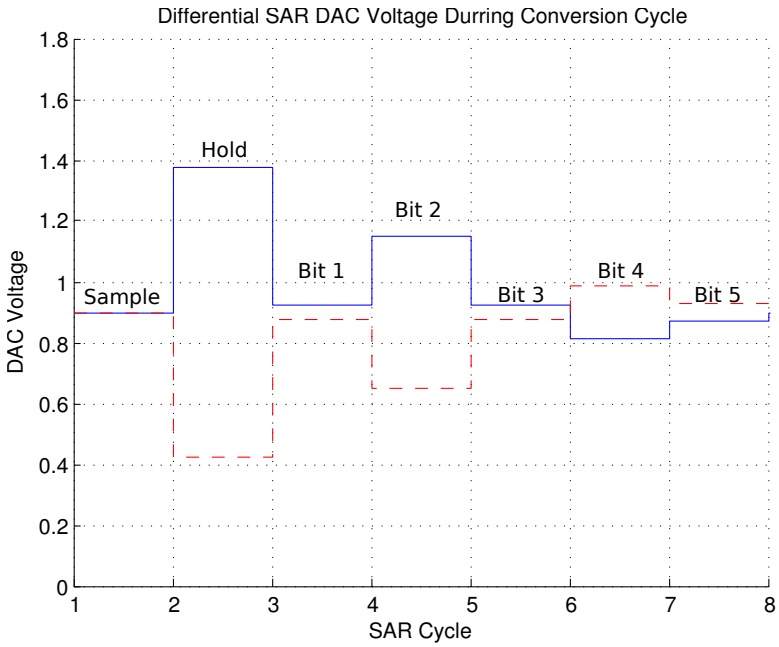

Figure 3.12: DAC Voltage Waveform of the SAR with Redundant Bit and Recovery

changes voltage in sharp steps. When the differential DAC voltage changes after a bit decision, there is a finite amount of time required to allow for the voltage to settle from ringing. If the differential voltage does not settle in time, the comparator might make the wrong the decision and tell the DAC to switch in the wrong direction. Once this occurs, it is no longer possible to drive the differential DAC voltage to within one half LSB. This will also result in a digital output code that does not match with the sampled input voltage. A bad decision at the beginning of the conversion with one of the first couple of bits would result in a larger error than a bad decision with one of the last few bits.

In order to correct for this kind of error, the DAC network must be expanded to allow for the ability to recover from a wrong decision. The classic method of doing this is to add redundant bits to the charge balancing circuit. Using the four bit SAR example, a fifth bit decision can be added in between bits two and three. This redundant bit would have the same weight as the second bit and allow for a recover if a mistake is made with either the first or second bit decision. Figure 3.12 and Table 3.2 shows how the ADC can recovery from an early, incorrect bit decision.

In this simulation, the differential voltage input is still +0.95 V, but the comparator

Table 3.2: Table of DAC Voltages for a SAR Conversion with Decision Recovery

Cycle	V_x **(V)**	V_y **(V)**	**Bit Decision**
Sample	0.900	0.900	N/A
Hold	1.375	0.425	+1
Bit 1	1.120	0.875	-1
Bit 2	1.150	0.650	+1
Bit 3	0.925	0.875	+1
Bit 4	0.8125	0.9875	-1
Bit 5	0.8688	0.9313	N/A

makes a wrong decision with the second bit. Without the redundant bit, the final digital output would be 1011 which results in an equivalent analog voltage of 0.787 V. Using the redundant bit with a weight equal to the second bit, the new output code is 10110 with an equivalent analog voltage of 1.0125 V. As one can see, the fifth bit does not add to the resolution of the A/D, but it does help correct for a decision error in the first two bits. For higher resolution converters, the number of redundant bits can be increased to correct for multiple bad decisions.

3.2.4 Capacitor Weight Mismatch

In a SAR converter, the second type of error, nonlinearity, is due to the capacitor size mismatch in the DAC network. An ideal SAR DAC has a binary weighted capacitor network where each successive capacitor is exactly one half the size of the previous capacitor. When the weights are not exact binary multiples of each other, the transfer characteristic of the converter is nonlinear. This effect is most obvious with transition at the Most Significant Bit. The weight of the MSB should be one LSB greater than the sum of the weights for the remaining bits. However, the mismatch between the MSB and the remaining bits is usually the largest.

In the example shown in Figure 3.13, a discontinuity can be seen when the code changes from 0111 to 1000. Table 3.3 shows both the ideal and non-ideal weights for a 4-bit A/D. Using this table the combined weights of the last three bits is 0.7782 V while the weight of the MSB is 0.9135 V resulting in a weight mismatch of 0.1353 V.

The nonlinearity due to capacitor size mismatch can be easily measured and character-

Table 3.3: Table of Ideal and Non-Ideal DAC Weights

Ideal Weights	**Non-Ideal Weights**
0.9000 V	0.9135 V
0.4500 V	0.4478 V
0.2250 V	0.2212 V
0.1125 V	0.1092 V

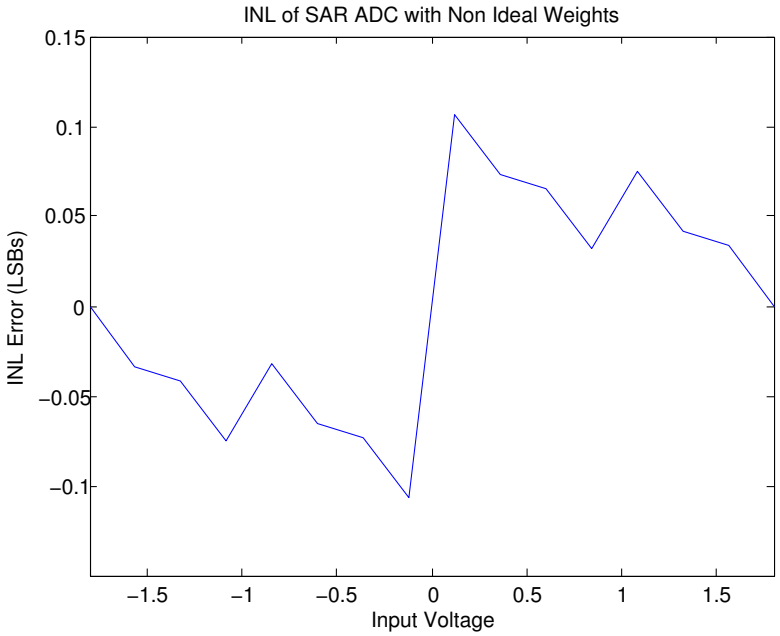

Figure 3.13: INL of Non-Ideal SAR Converter

ized externally. There are several methods used to measure DNL and INL for an A/D. The most straight forward method requires a computer controlled, precision voltage generator and a Digital Acquisition unit. The computer steps through all 2^N quantization levels, while DAQ captures the output code for each sample. This allows for a direct measurement of the nonlinearity errors. However, it is difficult to precisely iterate through all quantization levels of a high resolution ADC. More often, a statistical method is used by feeding a sinusoidal input to the A/D and plotting a histogram of the output as discussed in Section 2.1.2. After several million samples, the output histogram is compared to the ideal histogram of the sinewave input. Deviations from the ideal histogram curve translate to DNL and the INL can be calculated from the DNL. [14]

Measuring and quantifying the nonlinearity error due to capacitor mismatch can pro-

vide information to be used in correcting the output. Typically these measurements are fed back to a calibration circuit within the SAR converter. This eliminates the need for external circuitry to compensate for non-ideal weights. There are several calibration and correction techniques used with both TI and SAR converters, each with their own benefits and drawbacks. These methods and the method used for this research are discussed in more detail in the following chapter.

Chapter 4

ADC Calibration

As integrated circuit technology scales down to deep sub-micron sizes, device mismatching and increased variability introduce more challenges to analog circuit design. These difficulties have a large impact on the accuracy and distortion of all analog to digital converters. Therefore, much effort has gone into calibrating and correcting the outputs of A/Ds using combinations of analog and digital techniques.

4.1 Time Interleaved ADC Calibration

There are several techniques for reducing errors found in TI converters. One method can reduce the errors within the desired frequency range and increases the signal-to-noise-and-distortion ratio (SNDR). An example of this is the use of digital filters at the output of the converter to shift or shape the spectrum of the overall signal [19, 20, 21]. Another method involves calibrating the ADC to remove or reduce the cause of the errors and thus remove their effects. There are various forms of converter calibration, each with their own advantages and disadvantages. The main goal for this research is to perform converter calibrations as quickly and as accurately as possible. Therefore, it is necessary to ensure that a deterministic and digital background calibration technique is used.

In the past, analog circuit techniques have been used for calibration, but suffered the penalty of adding complex circuitry. In the analog domain, an increase in circuit complexity results in a larger penalty of die area, power, and noise [22, 9]. Common analog

techniques include variable signal delay stages or a reference ramp signal generator [23, 24]. These components can be very sensitive to noise and interference, producing invalid error estimates.

The work in [24] uses digital techniques to generate timing delay estimates which are discussed later. While the calibration estimates are done in the digital domain, this converter implements adjustable delay lines for each channel in the analog domain to correct for aperture delay mismatch. Another analog technique for correcting aperture delay mismatch is a distributed Sample and Hold network [25, 26]. A front-end S/H circuit operates at the full speed of the interleaved converter, f_S, to capture every sample. In order to reduce loading of the master S/H circuit, each channel has its own S/H circuit placed before the sub-ADC. By reducing the load on the front-end S/H circuit, the converter can run at a higher speed without any delay mismatch [25]. The additional S/H networks introduce extra analog complexity and the top speed of the master S/H is still limited by the process technology. An analog technique for offset error is shown in [21]. This work introduces a TI structure that adds a PRN chopping circuit to the analog front-end of each channel for offset estimation (correction is done in the digital domain).

[27] describes a 2:1 interleaved pipeline converter. This novel technique uses a single first stage that runs at the full sampling rate, f_S, but uses two interleaved subsequent stages that each operate at $f_S/2$. The ADC uses two tack and hold circuits that run at the full speed sampling rate, f_S. One T/H drives the first stage 2.8-bit flash A/D. The output of the first A/D is fed to two MDACs that use the second T/H to generate the two, interleaved residue paths. While the two T/H allow for a higher throughput rate, it adds analog complexity and the need to design for time delay mismatch. The ADC performs gain and offset error calibration in the digital domain. However, the offset error corrections that are calculated are then fed into offset cancellation circuits inside each pipeline stage. This introduces another layer of analog complexity to the calibration and correction hardware.

One of the advantages of technology scaling is the ability to use redundancy as a method for calibrating interleaved converters [26, 24]. The highly interleaved converter in [26] achieved a 36:1 ratio by implementing 42 ADCs on the chip. Instead of directly calibrating out all of the mismatch errors for each sub-ADC, the 6 worst performing channels were

eliminated in order to improve the system level performance. While this method does not use any additional power as those six sub-ADCs are turned off, it does require additional space for both analog and digital circuitry. This method also has the disadvantage of requiring foreground calibration to determine which six channels must be turned off.

Another redundant method which is similar to this author's work described in Chapter 5 compares the different ADC channels with each other [24]. In [24] two redundant ADC channels are added to a 16:1 interleaved converter. One of the channels is chosen as the reference channel. For some conversions, this reference channel is run simultaneously with another channel. A calibration loop observes the long-term RMS and mean values and attempts to make them identical for each channel pairing by making gain and offset adjustments in the digital domain [24]. While this calibration is similar to the one used in Chapter 5, there are two main differences. As mentioned earlier, [24] uses variable delay lines to correct for timing delay in the analog domain. In addition, the use of two redundant, full sized A/Ds adds both die area and power to the circuit design.

Digital calibration takes advantage of the scalability and improved performance of digital circuits [28, 9]. These circuits are more robust and less sensitive to noise from the system, which allows for accurate estimation and correction. Unfortunately, previous digital calibration techniques have not been both background and deterministic. Statistical calibration techniques usually require known inputs and long calibration times [29, 30, 31].

As mentioned earlier, [21] used a front-end PRN chopping circuit to calibrate offset error. The digital outputs from each channel are then fed into their respective accumulators. The output of each channel's accumulator is subtracted from the channel's digital code. The residual is then fed back into the accumulator in a feed-back loop. This forces the average accumulator inputs to be zero. Since the chopped signal is white noise, offset error would show up as DC spurs. The feed-back loop with the accumulators would drive the mean to zero, removing the offset error. There are two draw backs to this approach; it requires the additional chopping circuits and it takes a long time to converge.

To calibrate gain error, [21] upsamples the digital code by inserting zeros. The signal is then processed through a Finite Impulse Response (FIR) filter to remove the spurs due to gain error. By removing the spurs using filters, a portion of the usable spectrum is lost.

This method reduces the frequency range to less than Nyquist since the higher frequencies are filtered out with the gain spurs.

For aperture time delay, the offset and gain digital output codes are sent to an adaptive timing calibration system. Two adaptive FIR filters are used to remove the spurs due to delay. The filter coefficients are updated from a feed-back loop containing a phase detector and an accumulator. This method has two main drawbacks that were brought up with the gain and the offset calibration. The filtering reduces the usable spectrum, similar to the gain correction, while the accumulator feed-back loop is a statistical technique that takes a large number of samples to converge.

The foreground calibration techniques used in the past do not track with changes in the ADC non-idealities during operation such as temperature or supply voltage drift [22, 25, 26]. Both [25] and [26] use a distributed S/H network to avoid aperture delay error, but the remaining errors due to mismatch are calibrated using foreground techniques. [25] uses traditional gain and offset calibration techniques with a known test signal. Once enough data has been collected for each channel, the errors are then subtracted out of the digital code and multiplexed to generate the system output. As was previously mentioned, [26] performs a foreground calibration by characterizing all 42 channels in the converter. Once this is done, the six channels with the worst performance characteristics are then shutdown, while the remaining 36 are used for interleaving. Fully characterizing 42 ADCs requires a long amount of time. In addition, the performance of the remaining ADCs may still change over time, reducing the overall system performance.

Background calibration allows for error tracking while the ADC is operating. As the errors in the converters change over time, the correction system can track with those changes. This eliminates the need to take the converter off-line in the course of its operation [32, 33]. Using this technique, a constant stream of calibrated output data is achieved with no gaps or breaks. The research presented in [34] is able to calibrate out gain and offset error in the background. Timing delay is handled by using the traditional method of a front-end, full speed S/H circuit. A slow, highly accurate reference ADC is used to compare with each interleaved channel. An adaptive Least Mean Squares (LMS) loop is used to adjust the gain and offset for each channel in the digital domain until the mismatch error is eliminated.

Unfortunately, this calibration method requires the addition of this slow ADC that must be designed to operate with greater accuracy than each of the interleaved sub-ADCs. This reference ADC still places a burden on both die area and power consumption.

4.2 SAR A/D Calibration

Most calibration methods of the SAR converters are designed to correct for errors due to the effects of capacitor mismatch in the DAC. One technique used is to modify the DAC network itself to provide the ability to correct for mismatch. An example of this is shown in [35], where extra capacitors can be switched in or out of the DAC network to "trim" out the mismatch. A common method of reducing the die area of the charge scaling DAC is the use of bridging or coupling capacitors in series with banks of binary weighted capacitors. Figure 4.1 shows the comparison of an 8-bit SAR DAC with one bank of straight binary weighted capacitors versus two banks of parallel capacitors separated by a fractional coupling capacitor in series. In theory, the total capacitance of the lower bank on the left side should be equal to the LSB of the upper bank on the right side. Unfortunately, the matching is very dependent on the size of the coupling capacitor and the parasitics at that node. Therefore, additional calibration is required to achieve the correct matching between the two banks [35].

The 8-bit SAR presented in [35] uses two banks separated by a single coupling capacitor. The DAC uses some matching techniques such as the placement of dummy capacitors surrounding the upper bank to improve matching. In order to compensate for process variations with the size and parasitics of the coupling capacitor, additional "trim" capacitors are added. The upper bank uses an additional unit capacitor while an adjustable capacitor is added to the lower bank. To calibrate the DAC, the appropriate trim settings are calculated in the foreground by attempting to make the total charge value of the lower bank equal to the LSB of the upper bank. Throughout this calibration mode, the LSB of the upper bank is compared to the total value of the lower bank. The output of the comparator determines which side is greater and the appropriate adjustments are made to the trim capacitors. This process is repeated until the final values are found and the trim settings are stored in

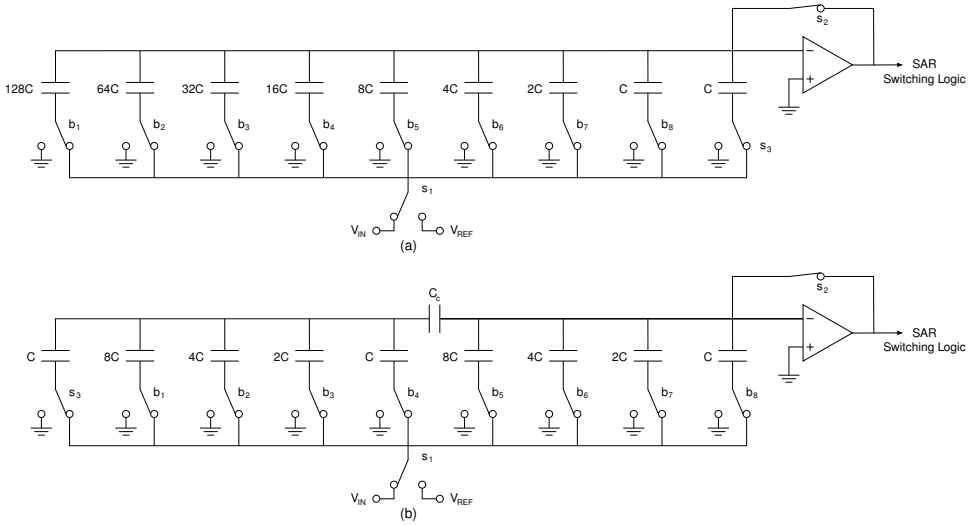

Figure 4.1: (a) An 8-bit SAR with straight binary weighted DAC
(b) An 8-bit SAR with a Coupling Capacitor DAC

memory for normal SAR operation.

This technique is highly dependent on the accuracy of the comparator for calibration. Therefore, it is necessary for the comparator to have a small offset error in order for the calibration algorithm to work. In order to minimize additional analog complexity, [35] uses a two-step timing delay to adjust for offset error. Additional transistors are added for course and fine adjustment for the time delay. During the regenerative phase, additional charge is injected into the nodes of the comparator to set the compensation voltage. The regenerative transistor pair amplifies the input-referred offset voltage. A longer time delay results in a higher gain, allowing the small offset to be canceled by the compensation voltage [35].

While [35] does require additional analog circuitry, another method proposed in [36] does not. This method uses a non-binary weighted DAC in a unique way to "learn" the correct capacitor weights. The SAR in [36] accomplishes this by using a Linear Feedback Shift Register and digital feedback loop to calculate the individual capacitor weights and a single offset weight. At the start of a calibration loop, the LFSR generates a PRN vector of the capacitors used to sample the reference voltage. The current set of weights and offset are then applied to the digital result, D_a. Then the normal SAR operation is performed and the weights are applied again to generate a new digital decision, D_b. The new weight values are calculated by taking the old weight values and adding the weighted difference between

D_a and D_b [36]. In fact, the digital feedback method and the use of the PRN is similar to the calibration presented in 6. However, both [35] and [36] require a foreground calibration, requiring the converter to be taken offline periodically to correct for errors dependent upon environmental changes.

There are several examples of using non-binary weighted ADCs to overcome the effects of mismatch error. [37] uses non-binary weighted capacitors with the addition of redundant bits to overcome non-linearity errors. This work was able to compensate for nonlinearity of up to 65 LSBs. A later example of this is described in [38], where the non-binary weighted SAR is used with the addition of digital signal processing. The main drawback of this method is the limitation on the amount of linearity that can be corrected [37]. It is important to note that the additional digital processing in [38] allowed for calibration of a higher resolution SAR converter.

Background calibration provides the benefit of tracking and correcting these errors over time. The background method used in [39] was developed for both pipeline and SAR converters. It corrects for errors in these converters with the addition of offset comparators and additional offset voltage references. Similar to some of the work described in the previous section, this work also takes advantage of redundant circuitry. The calibration method "swaps out" an extra stage in the pipeline converter in order to generate the correction data. This can also be performed in a SAR converter by swapping out capacitors or groups of capacitors. The use of extra capacitors is similar to the research presented in Chapter 6. The additional analog circuitry increases the power and die area requirements more than an all digital technique.

One common form of SAR calibration is to use an extra capacitive calibration DAC to correct for charge imbalance in the SAR DAC [40]. This method is used in [8] with the addition of digital logic to self-correct the SAR DAC. A calibration DAC with a range of ± 16 LSBs and a step size of 1/4 LSB is used to correct charge imbalance due to offset and linearity error. For the calibration cycle, all capacitors are discharged to V_{CM} and the input is switched to a known measurement signal. The main DAC is held at V_{CM} while the calibration DAC operates as a normal SAR DAC to search for the error voltage across the entire dynamic range of the converter. Once these errors are found, the binary

code from the calibration DAC is stored in a digital memory block. The converter is then switched to normal conversion mode and begins sampling the input signal. Data from the calibration memory block are then fed to the calibration DAC to correct for voltage error in each conversion.

The calibration performed in [8], like [35, 36], still requires an extra reference signal during a separate calibration mode. The ideal method would not require any additional reference signal or converter to generate the appropriate weight values for the SAR DAC. It would also perform the weight calculation in the background without the need to take the system offline or interrupt the input signal.

One technique that performs digital, background calibration is presented in [41]. This SAR uses linear equalization to match the digital output codes of the SAR with that of a slower, more accurate reference ADC. The SAR DAC uses non-binary weighted capacitors with a radix of 1.84 and two redundant bit decisions, which is similar to [37]. Reducing the capacitor radix to less than 2 prevents the problem of missing decision levels and works with the digital calibration and correction algorithm [41]. The main SAR A/D operates at a sample rate of f_S while the accurate reference ADC operates at a speed of f_S/M. The raw digital output of the SAR A/D is decimated by a factor M and equalized with the digital output of the reference ADC. The tap values for the linear equalization adaptive filter are updated using an LMS loop that runs at the sample rate of the main SAR A/D. The undecimated raw bit stream from the main SAR is corrected using the information from the latest linear equalization and sent as the system output. While this calibration technique does operate in the background using mostly digital hardware, its main drawback is that an extra reference ADC must be used to generate calibration data.

The sub-radix 2 technique used in [41] was modified and presented again in [42] to achieve a fully digital, background calibration method for a 12-bit SAR. This newer technique is similar to the Split-ADC architecture described in [9] and used in this work. The SAR in [41] performs two successive approximation conversions for each sample taken. In addition to the redundant capacitors used in [41], the SAR has an additional perturbation capacitor, C_Δ. This capacitor decision is switched to positive V_{ref} for the first conversion and negative V_{ref} for the second conversion. This generates two unique, weighted digital codes, x_+ and

Table 4.1: Comparison of Previous Calibration Techniques

	Time-Interleaved ADC	SAR ADC	Split-ADC [9]
Background	[24] [21] [34]	[38] [39]	✓
All Digital	[26] [25] [21]	[36] [38] [8] [42]	✓
Deterministic	[24] [26] [25] [34]	[35] [8] [42]	✓
Arbitrary	[24] [21] [34]		✓

x_-, that contain the known perturbation offset. If the difference Δx between those codes minus the perturbation offset is non-zero, then some error exists in the current values for the weights and perturbation. The stream of Δx data is fed into an LMS loop that is used to estimate the correct weights and perturbation values.

The perturbation calibration used in [42] has similar advantages as the Split-ADC calibration [9], which is described in the next section. Both techniques operate in the background without the need for a known analog signal or an extra reference ADC. They both perform the calibration and correction entirely in the digital domain to take advantage of scaling in deep sub-micron technologies. However, since the perturbation method must perform two conversions for each sample during calibration, it must operate at half of its full speed. In addition, the use of the extra perturbation capacitor requires a dynamic input signal and will not converge with an input at DC. The Split-ADC can operate with any input signal and maintain its maximum sampling rate for the entire operation as opposed to previous methods shown in Table 4.1. Therefore, the Split-ADC calibration method was chosen as the calibration technique to be applied to the SAR and interleaved converters in this work.

4.3 The Split-ADC Architecture

The Split-ADC architecture presented in [9] is a fully digital, deterministic, background self-calibration method for a cyclic ADC. Performing both the calibration and correction in the digital domain takes advantage of CMOS scaling. The background self-calibration eliminates the need for using a known benchmark signal and allows the converter to continue operating without being taken offline. Finally, the deterministic nature of the split architecture refers to a short time constant necessary to achieve full calibration.

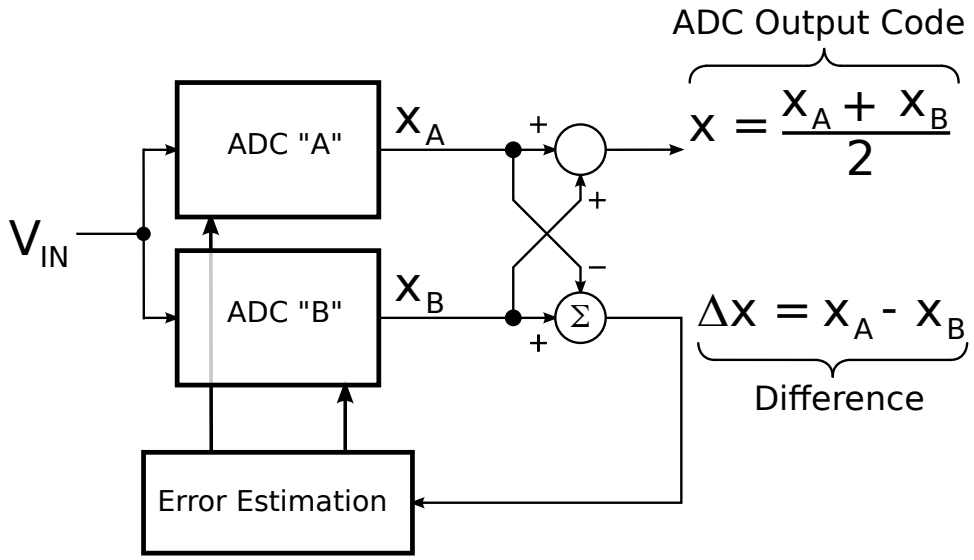

Figure 4.2: Block Diagram of Split-ADC Architecture

A system overview of the Split-ADC architecture is presented in Figure 4.2. The fundamental structure of the Split-ADC is composed of two independent converters that sample the same input signal V_{IN}. The individual outputs of the two A/Ds x_A and x_B are used to calibrated the two converters.

The system uses the two signals x_A and x_B to generate an average x, which is used as the digital output code, as well as a difference Δx, which is used for calibration. The Δx code is used to estimate the error in the current set of correction parameters for each ADC. This error estimate is used to update the correction parameters in the background. The Δx is driven to zero as the correction parameters are updated each time, reducing the estimated error. The calibration process is performed using digital CMOS logic, entirely in the background with a Least Mean Squares loop. The individual output codes x_A and x_B are digitally corrected in the foreground with the latest correction parameters. As Δx approaches zero, the average output signal x converges to the correct code.

The Split-ADC calibration described in [9] was applied to a cyclic A/D converter to correct for the gain parameter of the amplifiers used in the two ADCs. The fixed gain parameter for the amplifiers is a linear error correction; the analog amplifiers were designed to be highly linear. The Δx values were used to estimate the gain correction parameters in a Loop-Up Table (LUT). The error estimation loop was able to achieve full convergence in

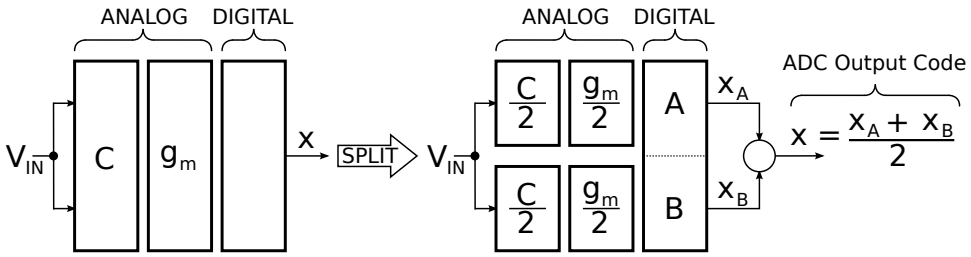

Figure 4.3: Splitting an ADC into two halves

less than 10,000 samples. Using a sampling rate of 1 MS/s, the convergence time constant is fast enough to track with any parameter variations while operating [9].

The primary trade-off when using the Split-ADC calibration structure is the need for two independent converters. However, [9] demonstrates how it is possible to use two ADCs that are half the size of the original, but still achieve the same performance with a minimal addition of digital circuitry. This is achieved by taking the average of the two output codes, x_A and x_B. With a single A/D, the size and complexity of the analog circuitry is the limiting factor with regards to die area and power. The capacitor sizes are chosen so that the $\sqrt{kT/C}$ noise performance is less than 1 LSB. If the capacitor and subsequent transistor sizes are reduced by 2, then the noise level will increase by $\sqrt{2}$. However, by taking the average of the two outputs, the noise level decreases by $\sqrt{2}$, back to the same performance level as the original, full sized converter. Figure 4.3 shows how a single ADC is split into two halves with equivalent power and die area usage.

For the analog circuitry in the single converter, the power, P, is assumed to be proportional to g_m. Using the Split-ADC method, the g_m in the analog circuitry is half the amount of original, single ADC. Therefore, the total power for the Split-ADC is shown as (4.1).

$$P_{Total} \propto \frac{g_m}{2} + \frac{g_m}{2} = g_m \tag{4.1}$$

(4.1) shows how the total power of the two, half-sized ADCs is equivalent to the single, full-sized ADC. The bandwidth f_T, of the single ADC is assumed to be proportional to g_m/C. When the single converter is split into two halves, the bandwidth of one half-sized

A/D is represented by (4.2).

$$f_T \propto \frac{g_m}{2} \cdot \frac{2}{C} = \frac{g_m}{C} \tag{4.2}$$

As such, the bandwidth of the Split-ADC is equal to the bandwidth of the single, full-sized converter.

The Split-ADC calibration method requires significant adaptation to be applied to different architectures. Chapters 5 and 6 discuss how to apply the Split-ADC method to the time interleaved and SAR converters, respectively.

Chapter 5

Split-Interleaved ADC

The previous section described the benefits of the Split-Cyclic ADC architecture over previous calibration methods for A/D. However, the work presented in [9] had less error coefficients than the amount required for a Split-Interleaved converter. The research presented in this chapter discusses the digital error correction for a TI converter and how the Split-ADC method can be adapted to estimate the correction parameters.

5.1 Digital Error Correction

Section 3.1 discussed the three channel mismatch error coefficients in an interleaved A/D, offset, gain, and aperture delay. From Equation (3.1), one can see that the difference between the ideal output code x, and the real output code, x_I is the sum of the three error components, x_{OSI}, xG_I, and $\dot{x}t_I$.

In order to get the correct output code, all three error components must be subtracted from the real code. This can be done in the digital domain using estimated correction parameters, $\hat{x}_{OSI}$, $\hat{G}_I$, and $\hat{t}_I$ representing the error coefficients.

$$\hat{x} = x_I - \hat{x}_{OSI} - x_I\hat{G}_I - \dot{x}\hat{t}_I \tag{5.1}$$

The corrected output code $\hat{x}$ will be equal to the ideal output code x, once the estimated correction parameters are equal to the real error coefficients.

Correcting the aperture delay error in the digital domain has an additional advantage over traditional methods using a high-speed track and hold analog circuit prior to the sub-ADCs. This front-end circuit runs at the master sample rate f_S and each sub-ADC converts the sample held by the main track and hold circuit. While this does eliminate the issue of aperture delay mismatch between channels, the device technology must be fast enough for the track and hold circuit run at f_S. This results in the sub-ADCs running at speeds less than what the device technology is capable of. By eliminating the front-end circuit and correcting the aperture delay in the digital domain, the individual sub-ADCs can be run at the maximum speed that the technology allows.

Estimating the correction parameters directly can be difficult; so instead, the error in the correction parameters can be estimated. This error in the correction parameters can be shown by plugging (3.1) into (5.1) to yield (5.2).

$$\hat{x} = x + \underbrace{x_{OSI} - \hat{x}_{OSI}}_{\varepsilon_{OSI}} + x_I \underbrace{\left(G_I - \hat{G}_I\right)}_{\varepsilon_{GI}} + \dot{x}_I \underbrace{\left(t_I - \hat{t}_I\right)}_{\varepsilon_{tI}} \tag{5.2}$$

By estimating the error in the correction parameters, the ideal values of $\hat{x}_{OSI}$, $\hat{G}_I$, and $\hat{t}_I$ can be calculated. Using this information, the Split-ADC method is used to estimate ε_{OSI}, ε_{GI}, and ε_{tI}.

5.2 Split-ADC and the Time Interleaved Converter

In the original Split-Cyclic ADC, a single, full-sized A/D was split into two half-sized ADCs. The basic concept of the Split-TI converter is to split each channel into half-sized, sub-ADCs. In order to generate estimates for all mismatch errors, every channel must be compared with each other. Figure 5.1 shows how the Split-ADC is applied to a 2:1 interleaved converter. In this configuration, there are 2M+1, or 5 sub-ADCs allowing one A/D to be "swapped out" for each master sample period, T_S. For example, at sample S3, ADCs 1 and 2 have just completed the conversion for S1, while ADCs 3 and 4 are still busy with S2. This leaves ADCs 1, 2 and 5 available to convert sample S3. ADCs 1 and 5 are chosen to free up sub-ADC 2 to be paired with sub-ADC 3 on the next sample conversion.

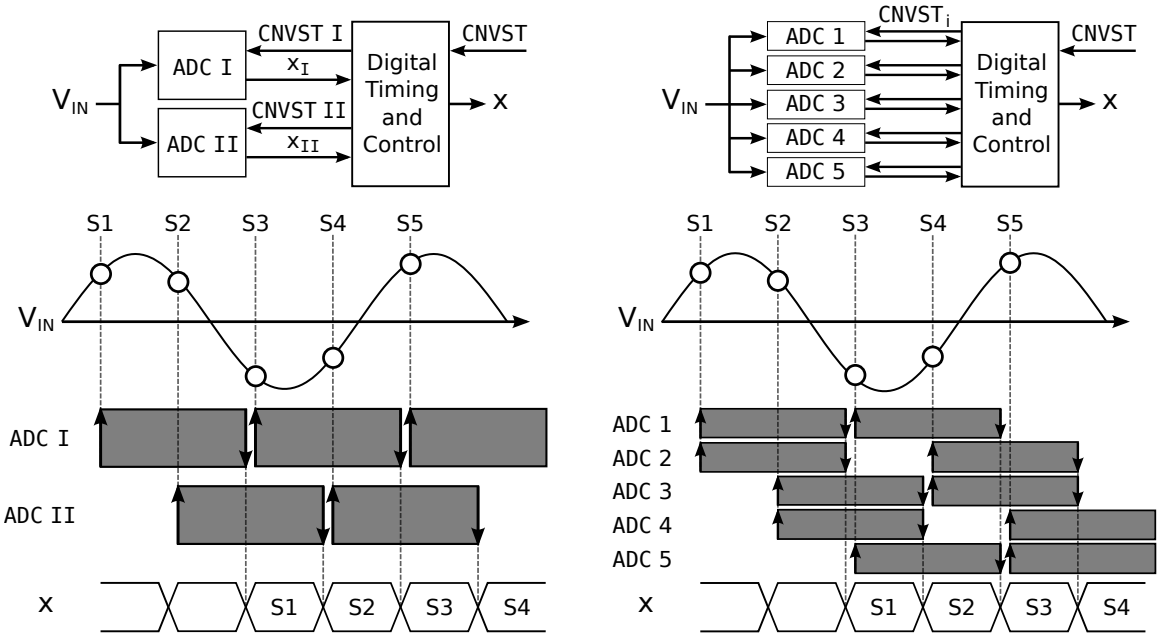

Figure 5.1: Block Timing Diagram of Split 2:1 TI ADC

This allows for all possible pair permutations to be used, generating a unique Δx for each pair.

The addition of the one extra half-sized sub-ADC channel does add some die area, an increase of 1/2M or 1/4 for a 2:1 interleaved converter. However, as the interleaving ratio M, increases, the fractional excess die area required decreases. Power usage will remain the same since since only 2M half-sized, sub-ADCs are operating at any given point in time.

The Δx values from each sub-ADC pair represent the channel mismatch error. For example, the difference in the corrected outputs of sub-ADCs 1 and 2 can be shown using (5.2) as (5.3).

$$\Delta x_{1;2} = \hat{x}_1 - \hat{x}_2 = (\varepsilon_{G_2} - \varepsilon_{G_1})\, x + \varepsilon_{x_{OS2}} - \varepsilon_{x_{OS1}} + (\varepsilon_{t_{APTD2}} - \varepsilon_{t_{APTD1}})\frac{dx}{dt} \tag{5.3}$$

As the correction parameters approach the ideal values, the error in the correction parameters goes to zero. As this error decreases, so does the Δx for the corresponding sub-ADC pair. This demonstrates one of the fundamental concepts of the Split-ADC architecture, as the calibration converges to the correct parameters, the Δx output goes to zero. However, while the Δx is non-zero, the error in the correction parameters can be solved for

using Δx data.

To demonstrate this calibration, consider a TI architecture with only offset error. By removing gain and aperture delay error, the example of the Δx for sub-ADCs 1 and 2 from (5.3) can be reduced to (5.4).

$$\Delta x_{1;2} = x_1 - x_2 = \varepsilon_{x_{OS2}} - \varepsilon_{x_{OS1}} \tag{5.4}$$

If all ten possible pairing combinations for a 2:1 TI converter are used, the resulting linear equation becomes (5.5).

$$\overbrace{\begin{bmatrix} \Delta x_{1;2} \\ \Delta x_{3;4} \\ \Delta x_{5;1} \\ \Delta x_{2;3} \\ \Delta x_{4;5} \\ \Delta x_{1;3} \\ \Delta x_{2;4} \\ \Delta x_{3;5} \\ \Delta x_{1;4} \\ \Delta x_{2;5} \end{bmatrix}}^{\mathbf{\Delta}} = \overbrace{\begin{bmatrix} +1 & -1 & 0 & 0 & 0 \\ 0 & 0 & +1 & -1 & 0 \\ -1 & 0 & 0 & 0 & +1 \\ 0 & +1 & -1 & 0 & 0 \\ 0 & 0 & 0 & +1 & -1 \\ +1 & 0 & -1 & 0 & 0 \\ 0 & +1 & 0 & -1 & 0 \\ 0 & 0 & +1 & 0 & -1 \\ +1 & 0 & 0 & -1 & 0 \\ 0 & +1 & 0 & 0 & -1 \end{bmatrix}}^{\mathbf{S}} \overbrace{\begin{bmatrix} \varepsilon_{OS1} \\ \varepsilon_{OS2} \\ \varepsilon_{OS3} \\ \varepsilon_{OS4} \\ \varepsilon_{OS5} \end{bmatrix}}^{\mathbf{e}} \tag{5.5}$$

The +/- 1 matrix, **S**, shows which sub-ADC channel was used for a given conversion. The sign shows the sub-ADC's contribution to the Δx value. In this format, the estimated errors in the correction parameters **e** can be solved for using the data in $\mathbf{\Delta}$ and **S**.

While the format presented in 5.5 does allow for the solution of **e**, a more condensed and intuitive format can be used. The total error component for a given channel is represented by the sum of all of the Δx values that use that specific channel. For example, the error contribution from channel 1 can be shown as (5.6).

$$\Delta x_{1;2} + \Delta x_{1;3} + \Delta x_{1;4} + \Delta x_{1;5} = 4\varepsilon_{OS1} - \varepsilon_{OS2} - \varepsilon_{OS3} - \varepsilon_{OS4} - \varepsilon_{OS5} \tag{5.6}$$

Here, the rows in $\mathbf{S}$ corresponding to channel 1 are summed, inverting the sign when indicated by column 1. When this is done for every channel, the resulting matrix equation looks like

$$\overbrace{\begin{bmatrix} \Delta x_{1;2} + \Delta x_{1;3} + \Delta x_{1;4} + \Delta x_{1;5} \\ \Delta x_{2;1} + \Delta x_{2;3} + \Delta x_{2;4} + \Delta x_{2;5} \\ \Delta x_{3;1} + \Delta x_{3;2} + \Delta x_{3;4} + \Delta x_{3;5} \\ \Delta x_{4;1} + \Delta x_{4;2} + \Delta x_{4;3} + \Delta x_{4;5} \\ \Delta x_{5;1} + \Delta x_{5;2} + \Delta x_{5;3} + \Delta x_{5;4} \end{bmatrix}}^{\mathbf{S^T\Delta}} = \overbrace{\begin{bmatrix} 4 & -1 & -1 & -1 & -1 \\ -1 & 4 & -1 & -1 & -1 \\ -1 & -1 & 4 & -1 & -1 \\ -1 & -1 & -1 & 4 & -1 \\ -1 & -1 & -1 & -1 & 4 \end{bmatrix}}^{\mathbf{S^TS}} \overbrace{\begin{bmatrix} \varepsilon_{OS_1} \\ \varepsilon_{OS_2} \\ \varepsilon_{OS_3} \\ \varepsilon_{OS_4} \\ \varepsilon_{OS_5} \end{bmatrix}}^{\mathbf{e}} \tag{5.7}$$

Note that matrix $\mathbf{S^TS}$ is singular. This is due to the fact that the absolute error in each channel cannot be calculated with Δx values. However, the goal of this work is to correct the channel mismatch error, not the absolute error. Therefore, a constraint is added to the system to force the calibration algorithm to solve for only mismatch error. The sum of the errors is set to zero by adding a row to both the $\mathbf{S^T\Delta}$ and $\mathbf{S^TS}$.

$$\begin{bmatrix} \vdots \\ \mathbf{S^T\Delta} \\ \vdots \\ 0 \end{bmatrix} = \overbrace{\begin{bmatrix} 4 & -1 & -1 & -1 & -1 \\ -1 & 4 & -1 & -1 & -1 \\ -1 & -1 & 4 & -1 & -1 \\ -1 & -1 & -1 & 4 & -1 \\ -1 & -1 & -1 & -1 & 4 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}}^{\mathbf{S^TS}} \overbrace{\begin{bmatrix} \varepsilon_{OS_1} \\ \varepsilon_{OS_2} \\ \varepsilon_{OS_3} \\ \varepsilon_{OS_4} \\ \varepsilon_{OS_5} \end{bmatrix}}^{\mathbf{e}} \tag{5.8}$$

By redistributing the average error constraint to all channels as was done for Equation

5.7, the final result is linear system with a $\mathbf{S^T S}$ matrix with coefficients along the diagonal.

$$\mathbf{S^T \Delta} = \overbrace{\begin{bmatrix} 5 & 0 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 & 0 \\ 0 & 0 & 5 & 0 & 0 \\ 0 & 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 0 & 5 \end{bmatrix}}^{\mathbf{A}} \overbrace{\begin{bmatrix} \varepsilon_{OS_1} \\ \varepsilon_{OS_2} \\ \varepsilon_{OS_3} \\ \varepsilon_{OS_4} \\ \varepsilon_{OS_5} \end{bmatrix}}^{\mathbf{e}} \tag{5.9}$$

The linear equation (5.9) can easily be solved, reducing the digital hardware requirements for calibration. In fact, since $\mathbf{A}$ is a diagonal matrix, it does not need to be stored in memory. Each ε_{OSi} can be solved for using the sum of the Δx values in $\mathbf{S^T \Delta}$ and a fixed division by a power of 2, further simplifying the necessary digital hardware.

Using (5.3), the linear system described by (5.9) can be expanded to include all three mismatch errors and 2M+1 sub-ADCs.

$$\overbrace{\begin{bmatrix} \vdots \\ \Delta x_{i;j} \\ \vdots \end{bmatrix}}^{\mathbf{\Delta}} = \overbrace{\begin{bmatrix} \vdots & \vdots & \vdots \\ \mathbf{S} & \hat{\mathbf{x}} & \hat{\hat{\mathbf{x}}} \\ \vdots & \vdots & \vdots \end{bmatrix}}^{\mathbf{B}} \overbrace{\begin{bmatrix} \varepsilon_{OS(i)} \\ \varepsilon_{G(i)} \\ \varepsilon_{t(i)} \end{bmatrix}}^{\mathbf{e}} \tag{5.10}$$

From (5.3), the coefficients matrix consists of +/- 1 for the offset, +/- $\hat{x}$ for the gain, and +/- $\hat{\hat{x}}$ for the aperture delay. The number of rows in $\mathbf{e}$ is dependent upon the number of sub-ADCs.

One important difference between Equations 5.9 and 5.10 is that matrix $\mathbf{B}$ is no longer just the Selection matrix, $\mathbf{S}$. The values in the Selection matrix are restricted to just three integers, -1, 0 +1, but matrix $\mathbf{B}$ contains the input signal x, and its estimated derivative $\hat{\hat{x}}$. These values can be scaled to be contained from -1 to +1, but they may take on any non-integer value. This requires more digital complexity to divide by floating point, high precision coefficients. In addition, numerical precision error may be introduced when dividing by a small coefficient that's close to 0. To reduce the digital hardware complexity,

the sgn of the coefficients is multiplied by the Δx values and summed for each error value.

$$[\mathrm{sgn}\,(\mathbf{B})]^{\mathbf{T}} \overbrace{\begin{bmatrix} \vdots \\ \Delta x_{i;j} \\ \vdots \end{bmatrix}}^{\mathbf{\Delta}} = \overbrace{[\mathrm{sgn}\,(\mathbf{B})]^{\mathbf{T}}\,\mathbf{B}}^{\mathbf{C}} \overbrace{\begin{bmatrix} \varepsilon_{OS(i)} \\ \varepsilon_{G(i)} \\ \varepsilon_{t(i)} \end{bmatrix}}^{\mathbf{e}} \tag{5.11}$$

Multiplying the $\mathbf{B}$ matrix by $[\mathrm{sign}\,(\mathbf{B})]^{\mathbf{T}}$ creates the coefficient matrix $\mathbf{C}$ where each three sub-matrices are diagonally dominant, similar to matrix $\mathbf{A}$ from 5.9. Essentially the sgn() function indicates whether the size of the error in the correction parameters is negative or positive. Instead of solving for the $\mathbf{e}$ directly, the errors can be found iteratively by using the sgn() function to take steps in the direction towards the solution.

The x values for the gain coefficients are easy to find; the average of the two corrected outputs from the Split-ADC which are used as the system output can also serve as the coefficients. The $\hat{\dot{x}}$ values for the aperture delay coefficients are supposed to be the derivative of the input x. However, it is difficult to know the exact derivative of the input for every sample, therefore, a digital estimate is used for $\hat{\dot{x}}$.

5.2.1 Digital Derivative Estimate

In the ideal case, the true value of dx/dt would be used for the aperture delay coefficients in 5.11. Since the true value can't be used, a digital estimate using the finite difference equation is used. The simplest form is the second order accurate $O(h^2)$, central finite difference method described in (5.12) [43].

$$\hat{\dot{x}}[n] = \frac{\hat{x}[n+1] - \hat{x}[n-1]}{2} \tag{5.12}$$

Using this derivative estimate does introduce a delay of one sample, but a single conversion latency is a small delay. As the input signal increases in frequency the effects of aperture delay increase since the magnitude of the derivative increases. In this situation, second order accuracy may not be sufficient and a higher order estimate must be used. A fourth

order accurate approximation $O(h^4)$, is presented in (5.13) with increased latency [43].

$$\hat{\dot{x}}[n] = \frac{-\hat{x}[n+2] + 8\hat{x}[n+1] - 8\hat{x}[n-1] + \hat{x}[n-2]}{12} \tag{5.13}$$

The main trade-off with choosing the order for the difference equation is accuracy versus latency. For inputs with low frequencies, a low order estimate can be used, and for inputs with frequencies near Nyquist, a higher order approximation is required. Figure 5.2 shows the comparison of different orders of the difference equation estimate used for error correction. This graph shows the RMS error of an input signal with a 50 ps delay introduced. The error is plotted in units of LSBs for a 16-bit converter with a sampling rate of 12 MS/sec. The solid block line shows that a signal without any delay correction will be accurate within one LSB for low frequencies. The corrected signal using a second order accurate difference method maintains an error less than one LSB for frequencies below 70% of Nyquist. The error is less than one LSB for all frequencies below Nyquist for correction using a fourth order accurate method.

5.2.2 The LMS Calibration

As was previously mentioned, solving for the error estimates exactly would require complex digital hardware. To eliminate the need for an exact solution and relax the hardware requirements, a Least Means Square adaptive filter technique is used. Figure 5.3 shows a block diagram of the LMS calibration and the digital error correction. The right hand side shows the path from input to output. The two sub-ADC outputs for a single conversion are digitally corrected using the current parameters. Their differences, average, and derivative estimates are then stored in the estimation matrix for calibrating the current correction parameters.

The LMS loop shown on the left hand side, operates continuously in the background, updating the error estimates ($\varepsilon_{OS(i)}$, $\varepsilon_{G(i)}$, and $\varepsilon_{t(i)}$) and correction parameters ($\hat{x}_{OS(i)}$, $\hat{G}_{(i)}$, and $\hat{t}_{(i)}$) every 128 samples. The number of samples needs to be sufficiently large as to make sure that all possible channel pairing combinations have been used. This is dependent on the interleaving factor M, as more pair combinations will require more samples to calculate

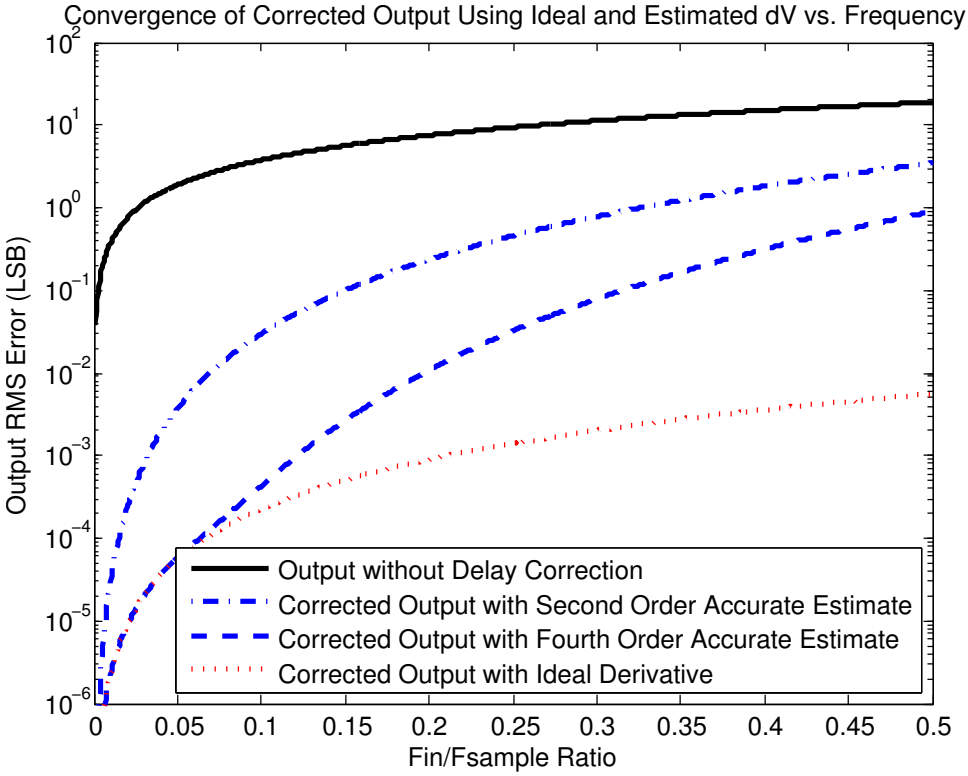

Figure 5.2: Comparison of Derivative Estimates used for Error Correction

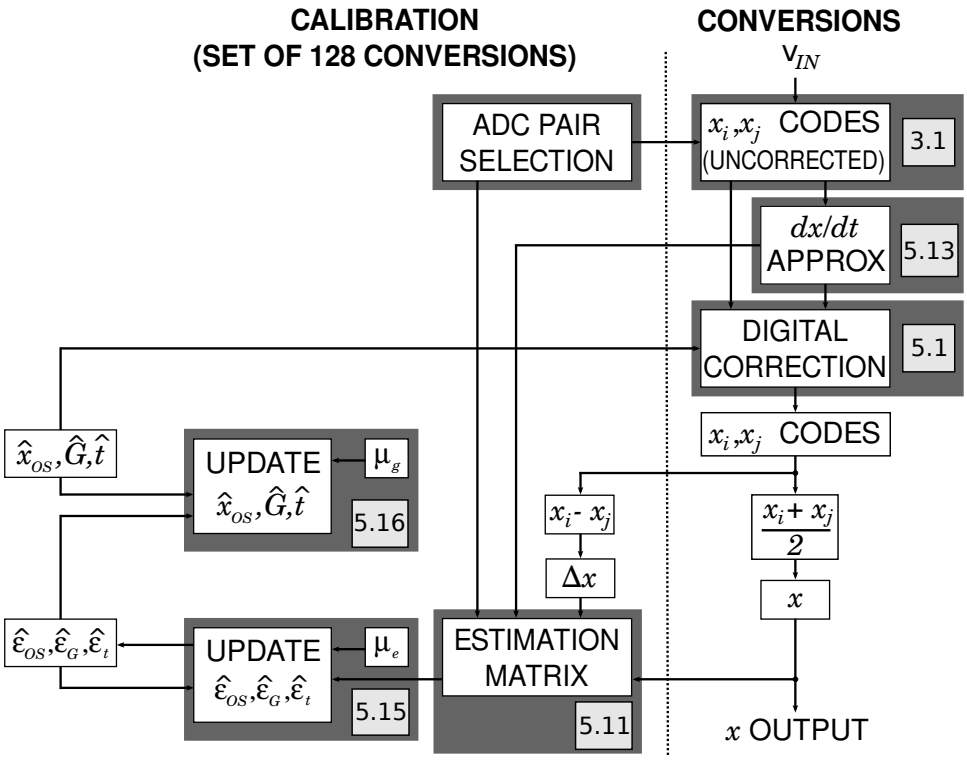

Figure 5.3: Error Estimation Algorithm and Correction

the new correction parameters.

Similar to matrix **A** in (5.9), empirical results have shown that coefficient sub-matrices **C** are diagonally dominant [44]. Using this information and the properties of the LMS adaptive filter algorithm, the error matrix **e** can be solved for by iteratively using a fixed step parameter μ_e.

$$\begin{bmatrix} \varepsilon_{OS(i)} \\ \varepsilon_{G(i)} \\ \varepsilon_{t(i)} \end{bmatrix} = \mu_e \left[\text{sgn}\left(\mathbf{B}\right)\right]^{\mathbf{T}} \begin{bmatrix} \vdots \\ \Delta x_{i;j} \\ \vdots \end{bmatrix} \tag{5.14}$$

This iterative method removes the need to store matrix **C** and avoid calculations. Once the new errors are found after 128 conversions, the old errors are updated within the LMS loop. By averaging the error solutions over time, the effects of any incorrect solutions are

reduced. The errors are updated with the inner LMS loop using (5.16).

$$\begin{aligned}
\varepsilon_{OS(i)}^{(new)} &= (1-\mu_e)\,\varepsilon_{OS(i)}^{(old)} + \mu_e\left(\left[\mathrm{sgn}\left(\mathbf{S}\right)\right]^{\mathbf{T}}\boldsymbol{\Delta}\right) \\
\varepsilon_{G(i)}^{(new)} &= (1-\mu_e)\,\varepsilon_{G(i)}^{(old)} + \mu_e\left(\left[\mathrm{sgn}\left(\mathbf{x}\right)\right]^{\mathbf{T}}\boldsymbol{\Delta}\right) \\
\varepsilon_{t(i)}^{(new)} &= (1-\mu_e)\,\varepsilon_{t(i)}^{(old)} + \mu_e\left(\left[\mathrm{sgn}\left(\hat{\mathbf{x}}\right)\right]^{\mathbf{T}}\boldsymbol{\Delta}\right)
\end{aligned} \tag{5.15}$$

Once the updated error values are found using the inner LMS loop, the outer loop is used to update the correction parameters. The estimated digital parameters are updated using the LMS step parameter μ_g in (5.17).

$$\begin{aligned}
x_{OS(i)}^{(new)} &= x_{OS(i)}^{(old)} - \mu_g\varepsilon_{OS(i)} \\
G_{(i)}^{(new)} &= G_{(i)}^{(old)} - \mu_g\varepsilon_{G(i)} \\
t_{(i)}^{(new)} &= t_{(i)}^{(old)} - \mu_g\varepsilon_{t(i)}
\end{aligned} \tag{5.16}$$

The two LMS parameters μ_e and μ_g are chosen so that the calibration loop converges quickly while still remaining stable. To maintain stability, μ_e is greater than μ_g while μ_e is chosen to be as large as possible to keep the convergence time constant small. Once the estimated correction parameters have converged, the corrected digital output codes are within one LSB of the ideal output code.

5.3 Simulation Results

To demonstrate the Split-Interleaved converter, the calibration method was applied to a 4:1 TI ADC in a simulation. The converter was modeled after the AD7621, a 16-bit SAR A/D from Analog Devices. The simulations were performed in two stages. First, a MATLAB model of the 4:1 converter was used to test and develop the calibration algorithm. For the second stage, the simulation used a mixture of SPICE and HDL models from an IC layout of nine AD7621 while the calibration algorithm was performed in MATLAB.

Table 5.1: Behavioral Simulation Parameters

PARAMETER		VALUE
Subchannel ADC	Resolution	16b
	Sample Rate	3 MSps
	SNR	90.2 dB
	INL	±1LSB
	DNL	±0.5LSB
	Bandwidth	50 MHz ±1%
Error Range	Offset	±0.1% FSR
	Gain	±0.2% FSR
	Aperture Delay	±50 ps
LMS Parameter	μ_g	1/1024
	μ_e	1/32
Internal Digital Precision		24b
Interleaved ADC	Interleaving	4:1
	Sample Rate f_S	12 MSps
	SNDR Uncorrected	34.2 dB
	SNDR Corrected	92.9 dB

5.3.1 MATLAB Simulation

The parameters for the MATLAB simulation were chosen based on the performance of the AD7621 in anticipation for the IC layout. The setup used nine 16-bit ADCs that each ran at 3 MS/sec, generating a master sample rate of 12 MS/sec. The nonlinearity of the sub-ADCs was restricted to be less than +/-1 LSB while the SNR was 90.2 dB. A full list of the simulation parameters is given in Table 5.1.

As Figure 5.1 shows, for each new sample there are always three sub-ADCs available to chose from. The previous section described how the LMS loop accumulated 128 samples of Δx to gather enough pair combinations. Iterating through a repeating set of all pair combinations does help provide enough information to solve for all errors. However, the repeating pattern of pair selections produces the spurs similar to the ones seen in a normal TI ADC without calibration. Figure 5.4 shows the FFT of the uncorrected digital output of the 4:1 converter with a repeating sequence of channel pair selections.

Simulation results have shown that a problem can occur with multi-tone inputs with the error spurs present. If one of the fundamentals of the input frequency overlaps with one

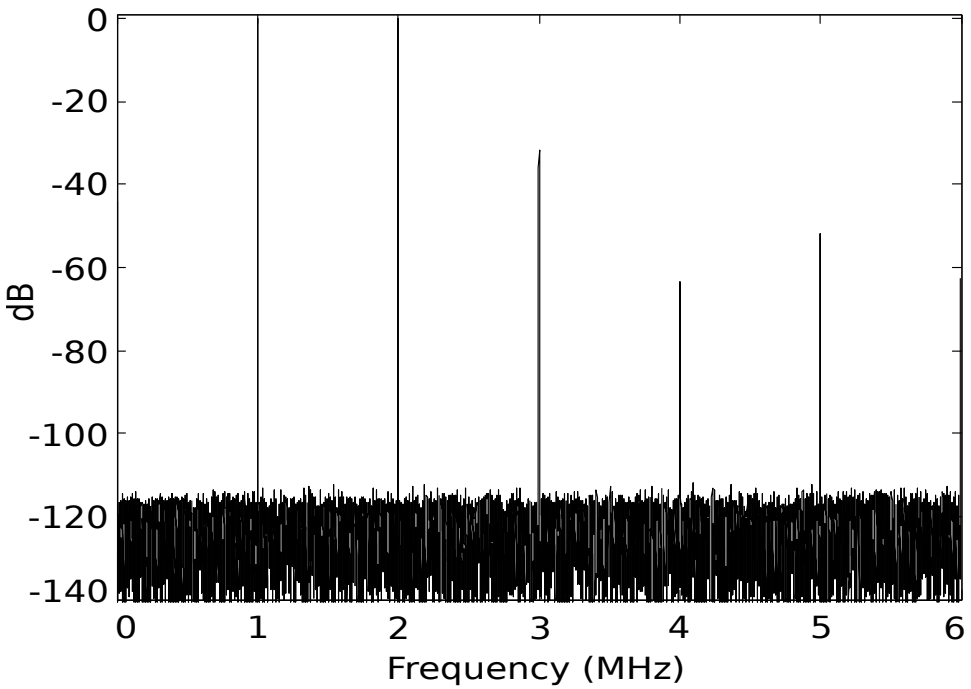

Figure 5.4: Uncorrected Interleaved ADC Output

of the error spurs, it is difficult to distinguish the difference between the ideal input signal and the error. When this occurs, the calibration algorithm does not always converge to a final state. Therefore, a pseudo-random channel pair selection is used to spread out the error spurs, similar to the work presented in [19]. For every conversion, two out of the three available sub-ADCs are chosen using a PRN generator. Instead of iterating through a set of 128 known pairs, the new set is composed of a permutation of shuffled channel pairs. Even though there is no guarantee that all 36 unique pairs of sub-ADCs have been selected, the relaxed requirements of the LMS loop takes adjusts for this. If a few pairs are missing, the LMS algorithm averages out the occasional bad error estimate. Figure 5.5 shows the FFT of the uncorrected output with the pair shuffling introduced. Even without error correction, the shuffling alone greatly improves the SFDR of the output signal by spreading the errors across all frequencies.

Finally, once the calibration loop is turned on and allowed to converge, the FFT of the fully corrected output is displayed in Figure 5.6. The SFDR is 104.8 dB and the total SNDR was 92.9 dB, 2.7 dB above the SNDR of the individual sub-ADCs. This is consistent

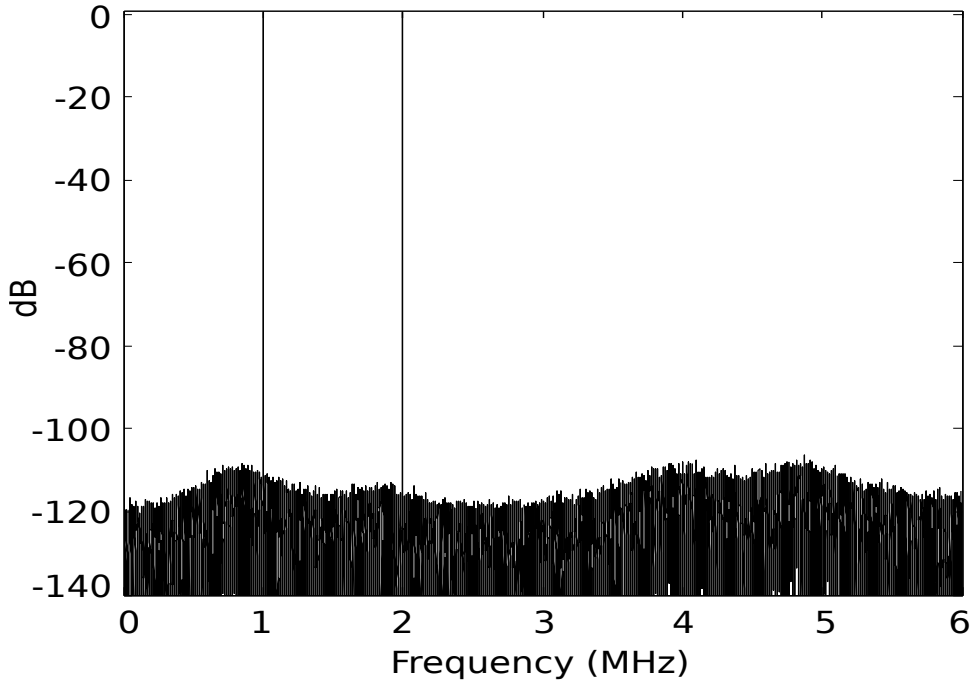

Figure 5.5: Uncorrected Interleaved ADC Output with Pair Shuffling

with the explanation previously given that averaging the two individual outputs for each conversion would decrease the noise level by 3 dB. Using the calculation for the ENOB given by (2.10), the Effective Number of Bits is approximately 15.14. It is important to note that the ENOB of a converter must always be less than the ideal number of bits due to all non-idealities.

The two error spurs seen at 3 and 4 MHz are due to the nonlinearity of the individual channels. Since this calibration method is designed to correct for linear error mismatch, the performance is still limited by the INL and DNL of the sub-ADCs. It is important to note that bandwidth mismatch can also contribute to these error spurs, imposing another limit to maximum performance of the Split-Interleaved method. As described in 3.1.4, the analysis performed in [18] shows how bandwidth mismatch is dependent on two factors, the average bandwidth and the input frequency. For a mean bandwidth of 50 MHz and input frequencies near Nyquist, the maximum tolerable bandwidth mismatch can be $\pm 3\%$. However, the input signal is bandwidth limited by a first order, low pass filter at 16 MHz. The numerical analysis for an input frequency at 1.5 of f_S or 18 MHz shows that the

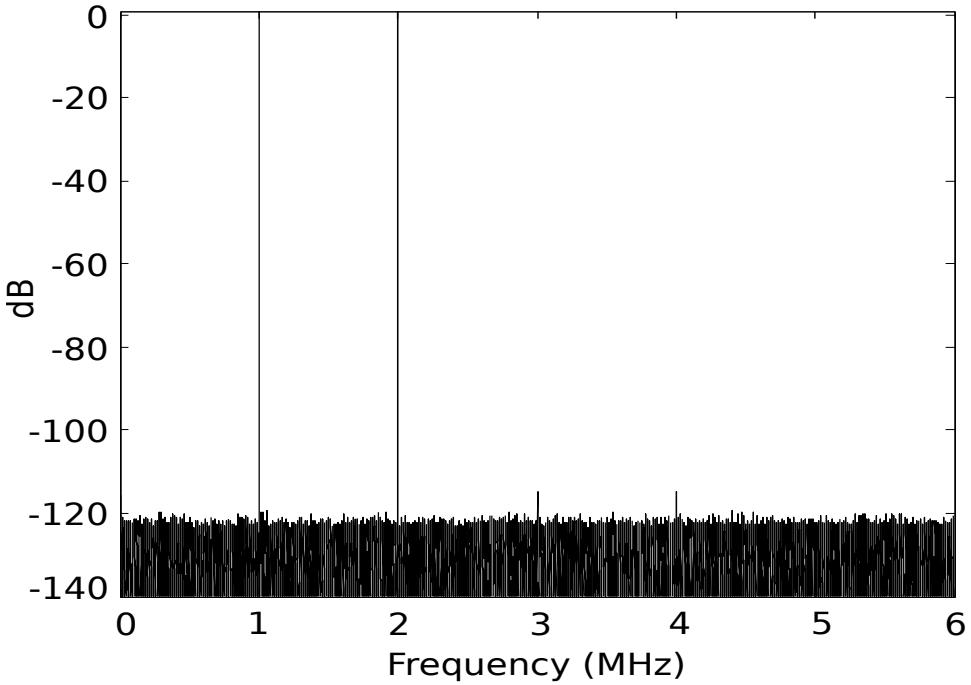

Figure 5.6: Fully Corrected Interleaved ADC Output

maximum bandwidth mismatch in this 16-bit system is ±1%.

Simulations were also run to test the convergence rate based on the type of input signal and the LMS parameters. Using the parameters shown in Table 5.1, several types on input signals were tested. Figure 5.7 shows the convergence rate with four different input signals. For three input signals, a multi-tone sinewave and two DC inputs, the calibration loop achieves full convergence in less than 100 000 samples. The random input signal takes 400 000 samples, however, with the master sample of 12 MS/sec, the time elapsed is only 33 ms. Even with the random input, all signals have a convergence time that is fast enough to track with most error variations due to environmental changes.

Figure 5.8 shows the convergence rate due to LMS parameter adjustments. As the step size is increased, the calibration loop reaches its final state at a faster rate. Increasing the step size can also translate to a longer recovery time if a bad error estimate is calculated. Therefore, a compromise is chosen by using $\mu_e = 2^{-5}$ and $\mu_g = 2^{-10}$.

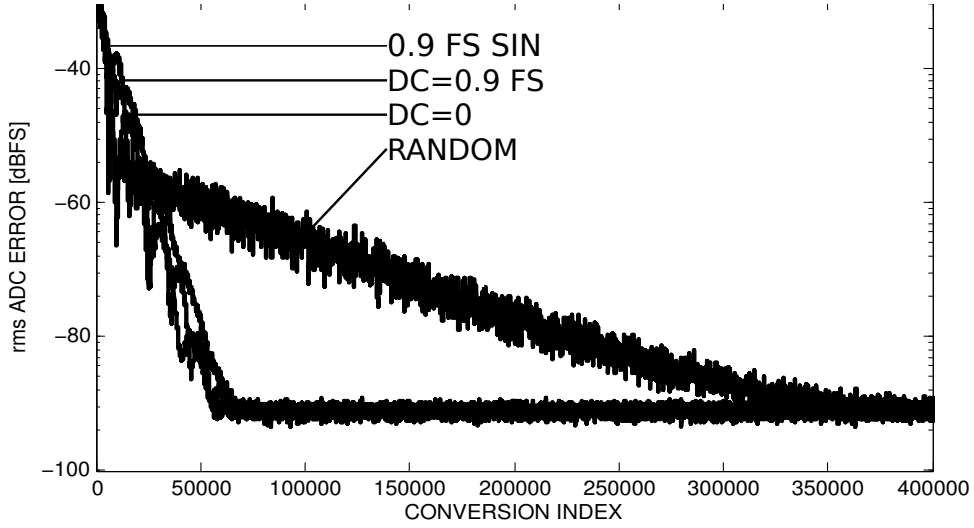

Figure 5.7: Calibration Convergence for Sinusoidal, DC, and Random Input Signals

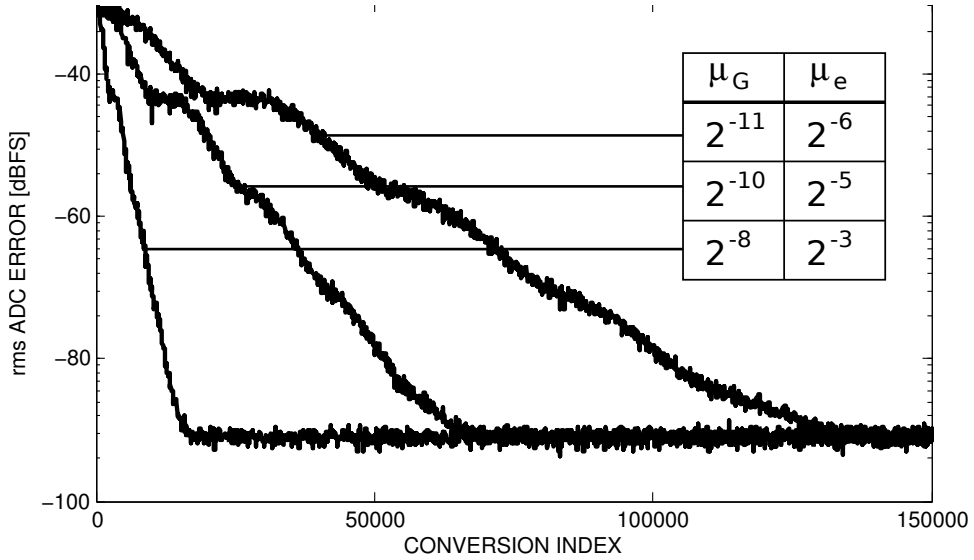

Figure 5.8: Calibration Convergence for Different LMS Parameters

5.3.2 IC Layout Simulation

With the Split-Interleaved concept successfully demonstrated using a complete MATLAB simulation, an IC was made by Rosa Croughwell to implement this calibration [45]. The following simulation uses the SPICE and HDL models from that layout to perform the calibration and correction in MATLAB. The IP cores for nine AD7621s were laid out in a $0.25um$ process with added digital logic for timing and I/O. This layout was extracted to a combination SPICE and HDL model to be tested with the calibration algorithm. Each sub channel was assigned an offset error of +/- 0.1%, a gain error of +/-0.2%, and an aperture time delay error of +/-50 ps. The input signal used was a 2V peak-to-peak sinewave with 1024 sample points. The overall sampling rate was 12 MS/s (each sub channel pair operating at 3 MS/s) which was used to determine the input signal frequency with (5.17).

$$f_{in} = \left(\frac{NumberofCycles}{1024} \right) f_{sample} \tag{5.17}$$

Using 12 cycles of the input sinewave, the frequency used was 140.625 kHz.

The actual calibration algorithm was performed in MATLAB. The nonideal output of the model from the IC layout was fed into the algorithm with 512K samples of the simulated input sinewave. This signal was created by replicating the 1024 samples 500 times. Figure 5.9 shows an FFT of the output sinewave before calibration. Notice the nonideal signal spectrum does not have the frequency spurs normally associated with channel mismatch errors due to the PRN pairing. In Figure 5.10, the spectrum of the corrected signal is shown after the calibration routine has achieved full convergence. Here the corrected waveform's performance matches the ideal signal without any errors.

The output spectrum with PRN pairing prior to calibration convergence had a SNDR of 53.12 dB and a SFDR of 62.58 dB. After calibration has achieved full convergence, the corrected output signal had a SNDR of 99.07 dB and a SFDR of 116.04 dB.

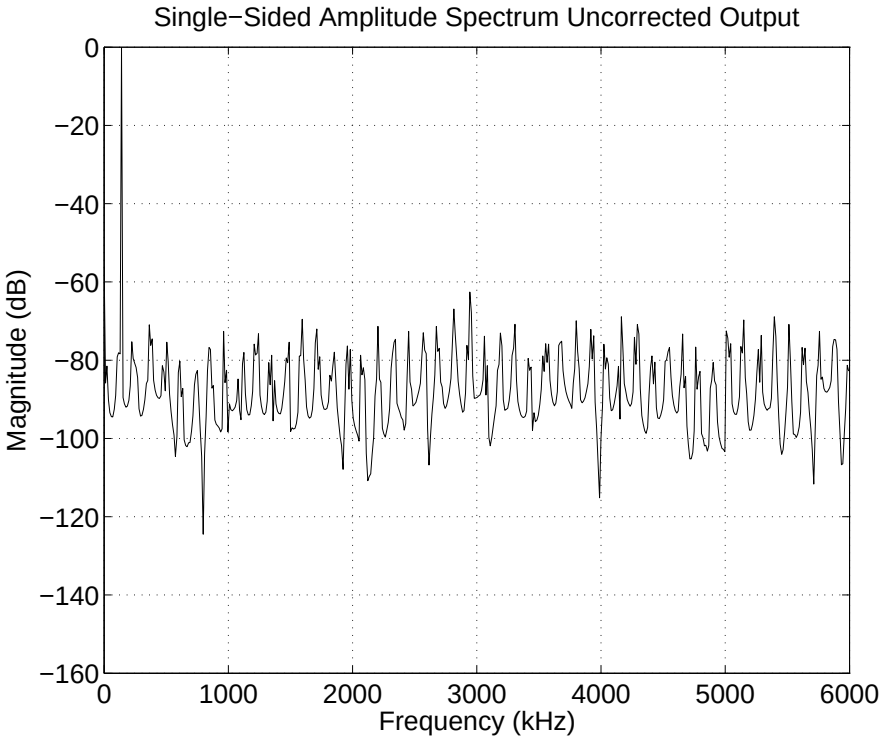

Figure 5.9: FFT of Uncorrected Sine Wave $f_{in} = 140.625kHz$

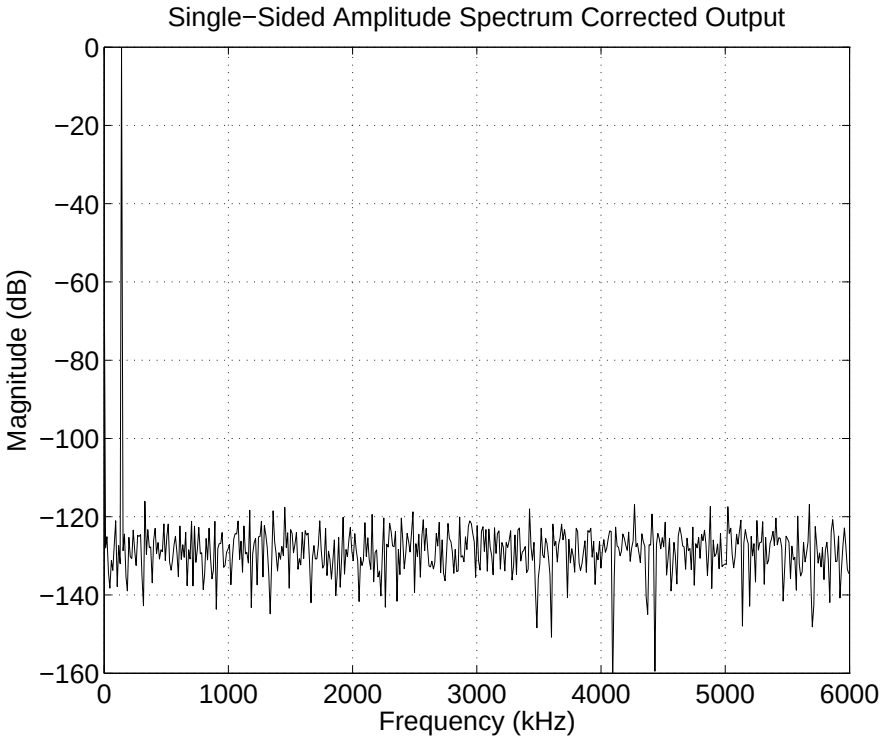

Figure 5.10: FFT of Corrected Sine Wave $f_{in} = 140.625kHz$

Chapter 6

Split-SAR ADC

Chapter 3 discussed capacitor mismatch as the main source of error in successive approximation register ADCs. Most calibration methods attempt to correct the weights of the individual bit decisions. The Split-ADC method can be applied to estimate for and apply the capacitor weights to the bit decisions for the digital output. In order to solve for this capacitor mismatch the traditional SAR architecture must be adapted to use the Split-ADC calibration. Section 6.1 reviews the application of the Split-ADC method to the SAR converter. Section 6.2 describes the design of the Split-SAR integrated circuit and Section 6.3 presents the results.

6.1 Applying the Split-ADC to the SAR Converter

The thesis work presented by Chilann Chan in [11] described the development and simulation of the Split-SAR calibration architecture. The goal of using the Split-ADC structure with a SAR converter is to calculate the capacitor weight mismatch of the charge balancing DAC. As such, the number of calibration parameters is directly proportional to the resolution of the A/D. Similar to the time interleaved converter, the SAR converter must be modified to the accommodate the Split-ADC method. These changes to the converter are made in the charge balancing DAC network and the digital domain in order to calibrate out the capacitor mismatch. Figure 6.1 shows the block diagram of the entire Split-SAR including both the integrated circuit hardware and the off-chip digital logic.

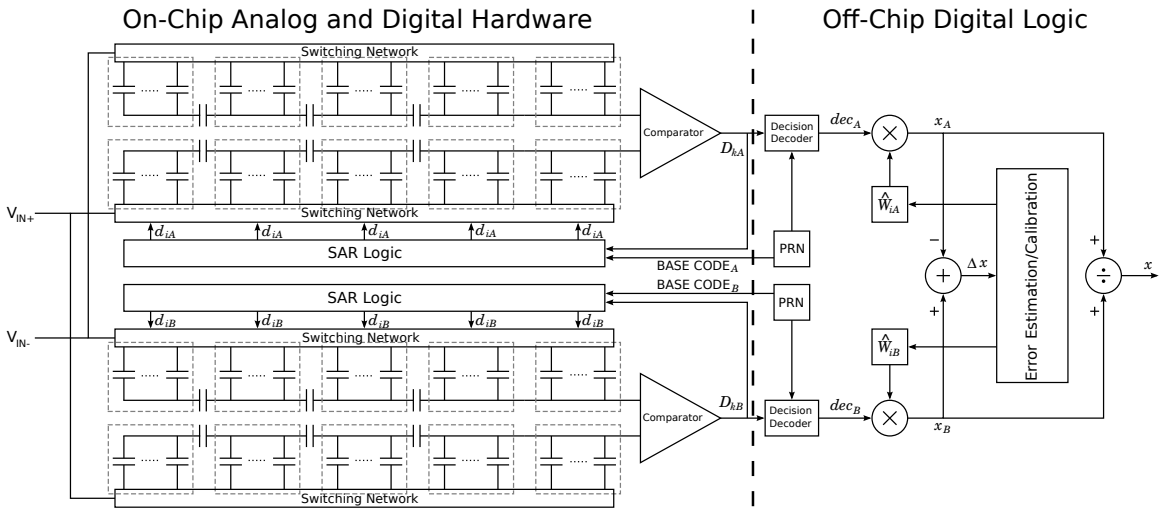

Figure 6.1: Split-SAR Block Diagram

6.1.1 Modifying the SAR DAC Network

The 4-bit SAR example presented in Section 3.2.3 contained a redundant bit to correct for a bad comparator decision. To extend this example to a 16-bit converter, a total of 20-bits are used, where every fourth bit is a redundant bit. Distributing redundant bits across the DAC network allows the SAR to compensate for a bad decision though out the entire conversion cycle.

The 4-bit example shows how the charge scaling network uses binary weighted capacitor sizes. Each bit decision is a binary multiple of a fixed unit capacitor size C, where the LSB decision is equal to one C. This unit capacitor is useful for scaling a design for different resolutions and technology professors. As stated earlier, the capacitor sizes in an A/D are limited by the noise performance $\sqrt{kT/C}$. The unit capacitor is sized so that this noise level is less than one LSB. The use of a unit capacitor can help reduce mismatch error in layout. By constructing the physical DAC capacitors out of unit capacitor blocks, the IC fabrication process yields better matched capacitors [12].

When creating a 16-bit weighted DAC out of unit capacitors, it is impractical to create a MSB weight out of 65 536 unit caps. As described in Section 4.2 it is necessary to use a charge dividing network by placing coupling capacitors in series with banks of parallel, binary weighted caps. For this SAR converter, all 20-bits are divided into five banks of 4-

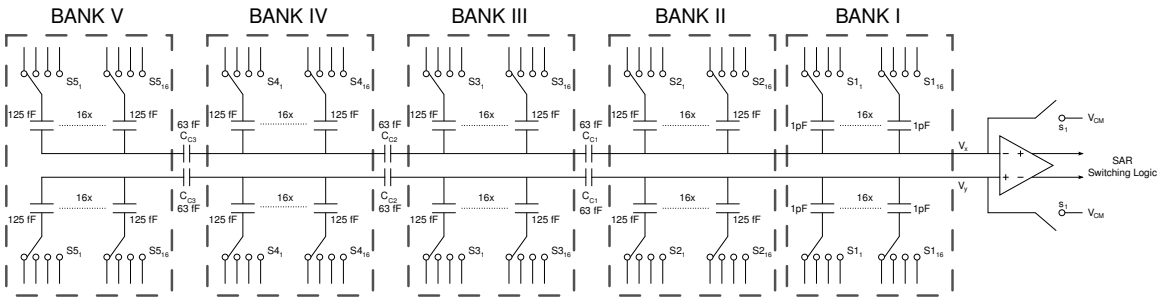

Figure 6.2: Split-SAR DAC Network

bits. Each bank consists of three binary weighted capacitors and one redundant bit. Figure 6.2 shows one side of the 20-bit, differential SAR DAC with three coupling caps dividing the last four banks. The first bank containing the top three MSB decisions uses a unit capacitor weight of 1 pF. The remaining four banks use a unit capacitor weight of 125 fF. The coupling capacitors are sized so that each successive bank is half the weight of the previous one. This eliminates the need for unreasonably sized capacitors.

Like the Split-Interleaved converter, in order to solve for capacitor mismatch, the unit capacitors must be compared with each other. To accomplish this, extra unit caps are added to allow for swapping out a different unit capacitor every conversion. As shown in Figure 6.2 each bank consists of 16 unit capacitors giving a total of 80 unit capacitors for each side of the differential charge balancing network. To apply the Split-ADC calibration, one unit cap is swapped out in each conversion to permutate through 16 capacitor combinations.

Traditional SAR A/Ds use bit decisions that consist of a fixed set of unit capacitors. The novel content of the Split-SAR converter is the ability to have all 80 unit capacitors in each bank be individually selectable. This means each bit decision in a bank can use different combinations of unit capacitors. To facilitate this ability, a 4-bit base code is used for each bank in the DAC. The base code selects one of the 16 unit caps to be swapped out and the remaining 15 to be used as the four bit decisions. Figure 6.3 demonstrates how the base code selects one of 16 unit capacitor combinations.

The base code sets up a selection matrix $\mathbf{S}$, to distribute the four bit decisions, D_k through D_{k+4}, among the 16 capacitors in each bank. The result is a 16 element vector $\mathbf{d}$, of ± 1 bank segment decisions. For example, with a base code of 3, the first bank of segment

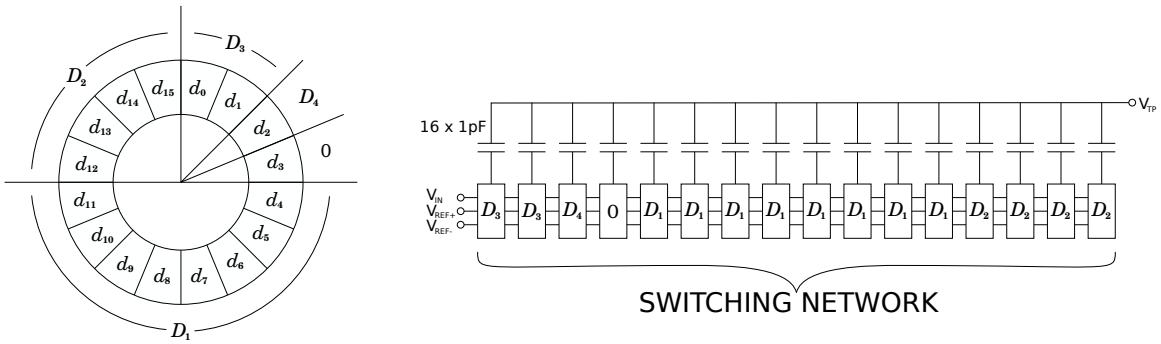

Figure 6.3: Split-SAR Bank Capacitor Selection

decisions $\mathbf{d_I}$, are determined by (6.1).

$$\overbrace{\begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \\ d_4 \\ d_5 \\ d_6 \\ d_7 \\ d_8 \\ d_9 \\ d_{10} \\ d_{11} \\ d_{12} \\ d_{13} \\ d_{14} \\ d_{15} \end{bmatrix}}^{\mathbf{d_I^T}} = \overbrace{\begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}}^{\mathbf{S}} \overbrace{\begin{bmatrix} D_1 \\ D_2 \\ D_3 \\ D_4 \end{bmatrix}}^{\mathbf{D_I}} \tag{6.1}$$

Since the base code is 3, the segment decision d_3 equals 0 so the unit capacitor C_3 is swapped out. The next eight segment decisions, d_4 – d_{11}, are assigned bit decision D_1, and switch bank capacitors C_4 – C_{11} to appropriate V_{REF}. The second bit decision, D_2 is assigned to

the following four segment decisions, d_{12} – d_{15}, to switch the capacitors C_{12} – C_{15}. Finally segment decisions d_0 and d_1 are assigned bit D_3 and segment $d2$ is assigned D_4, selecting switches C_0 – C_1 and C_2 respectively.

6.1.2 Split-SAR Error Calibration

To fully implement the Split-ADC algorithm, two separate Split-SAR A/Ds are placed on the same chip. The average of the two outputs are used as the converter output while the differences are used to estimate the correct unit cap weights. The digital output code for each conversion uses the current set of DAC weights. After a certain amount of conversions are completed, the current DAC weights are updated with a new estimate of the error in those weights.

The previous section described how each Split-SAR sub-ADC consists of 80 unit capacitor selections instead of just the 20 bit decisions a regular SAR would use. These decisions **dec**, are divided up by the five banks and grouped into segment decisions **d**. Each segment represents the four bit decisions translated by the base code and distributed among the 16 capacitors. For the example used in (6.1) where bank segment 1 was assigned the base code 3, the decision distribution would look like

$$\mathbf{d_I} = [D_3 D_3 D_4 0 D_1 D_1 D_1 D_1 D_1 D_1 D_1 D_1 D_2 D_2 D_2 D_2] \tag{6.2}$$

Each conversion consists of five segments to form one decision vector, **dec**.

$$\mathbf{dec} = [\mathbf{d_I d_{II} d_{III}} d_{IV} d_V] \tag{6.3}$$

Note that the individual capacitor mismatch in the final two banks is not sufficient enough to have a significant impact on linearity. Therefore, only the total capacitor decisions are

stored for segments 4 and 5. The result is

$$d_m = \overbrace{\begin{bmatrix} 8 & 4 & 2 & 1 \end{bmatrix}}^{\mathbf{S}} \overbrace{\begin{bmatrix} D_k \\ D_{k+1} \\ D_{k+2} \\ D_{k+4} \end{bmatrix}}^{\mathbf{D_m}} \tag{6.4}$$

where d_m and D_k – D_{k+4} is d_{IV} and D_{13} – D_{16} for bank 4 and d_V and D_{17} – D_{20} for bank 5 respectively. This forces the capacitor values in the last two banks to be treated as binary-weighted capacitors.

The final decision vector **dec**, contains a total of 50 elements. The first 48 elements are ± 1 for segment unit capacitor decisions and a 0 for each unused capacitor. The last two elements are the sums of the capacitor decisions for the last two banks. To generate the final output code, the decisions are multiplied by the estimated weights $\hat{\mathbf{W}}$.

$$\hat{x} = \underbrace{[\mathbf{d_I d_{II} d_{III}} d_{IV} d_V]}_{\mathbf{dec}} \cdot \hat{\mathbf{W}} \tag{6.5}$$

The first 48 estimated weights represent the actual weight of each unit capacitor in the first 3 banks. The last two weights represent a scaling factor for the binary-weighted sums of the last two banks.

The Split-ADC calibration algorithm is used to update the estimated unit capacitor weights in the SAR DAC. The left hand side of the block diagram in Figure 6.4 shows the single LMS loop used to estimate the DAC weights. On the right hand side, two independent decisions, D_A and D_B are generated by the SAR A and SAR B. The PRN base selection is used to decode the bit decisions into segmented unit capacitor decisions, $\mathbf{dec_A}$ and $\mathbf{dec_B}$. These are multiplied by their current estimated weights, $\hat{\mathbf{W}}_\mathbf{A}$ and $\hat{\mathbf{W}}_\mathbf{B}$, to generate the two codes $\hat{x}_A$ and $\hat{x}_B$. The average of the codes are used as the system output $\hat{x}$, while the difference, Δx is fed to the estimation matrix to generate the next set of estimated weights.

The Split-SAR algorithm works on the basis that the Δx represents the nonlinearity error in both SAR DACs. By accumulating enough Δx values to estimate the nonlinearity

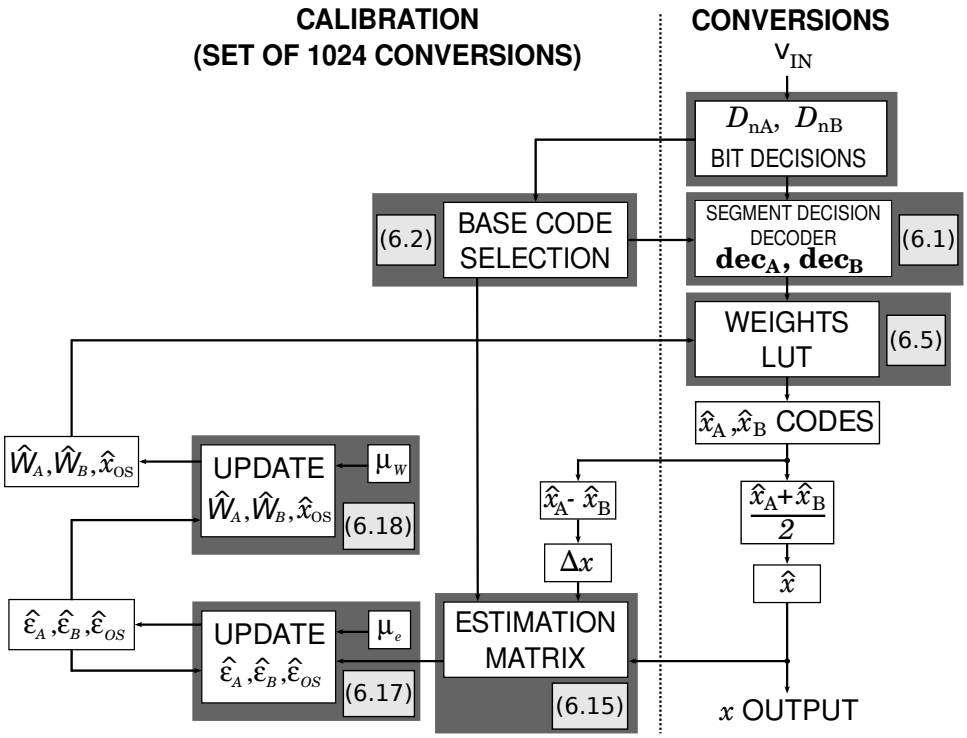

Figure 6.4: Split-SAR Error Estimation Algorithm and Correction

and correcting the error, the Δx values should go to zero. However, when using two parallel SAR converters, additional error components are introduced to the system, offset and gain mismatch. The fixed offset mismatch can be represented by (6.6).

$$\begin{aligned} \hat{x}_A &= \hat{x} - \frac{\hat{x}_{OS}}{2} \\ \hat{x}_B &= \hat{x} + \frac{\hat{x}_{OS}}{2} \end{aligned} \tag{6.6}$$

Since gain mismatch is represented by a multiplication factor, it can be incorporated into the estimated weights values. The offset mismatch, the estimated weights, and the decisions vectors $\mathbf{dec_A}$ and $\mathbf{dec_B}$ can be expressed in (6.7).

$$\begin{aligned} \hat{x}_A &= \overbrace{[\mathbf{d_I d_{II} d_{III}} d_{IV} d_V]}^{\mathbf{dec_A}} \cdot \hat{\mathbf{W}}_\mathbf{A} - \frac{\hat{x}_{OS}}{2} \\ \hat{x}_B &= \overbrace{[\mathbf{d_I d_{II} d_{III}} d_{IV} d_V]}^{\mathbf{dec_B}} \cdot \hat{\mathbf{W}}_\mathbf{B} + \frac{\hat{x}_{OS}}{2} \end{aligned} \tag{6.7}$$

The estimated weights can be represented as a combination of the ideal weights and the error in the estimated weights.

$$\begin{aligned}\hat{\mathbf{W}}_\mathbf{A} &= \mathbf{W}_\mathbf{A} + \varepsilon_\mathbf{A} \\ \hat{\mathbf{W}}_\mathbf{B} &= \mathbf{W}_\mathbf{B} + \varepsilon_\mathbf{B}\end{aligned} \tag{6.8}$$

The estimated offset mismatch can be represented as the sum of the ideal offset and error in the estimated offset.

$$\hat{x}_{OS} = x_{OS} + \varepsilon_{OS} \tag{6.9}$$

From (6.8) and (6.9), the output codes can be represented as

$$\hat{x}_A = \mathbf{dec_A} \cdot \overbrace{[\mathbf{W_A} + \varepsilon_\mathbf{A}]}^{\hat{\mathbf{W}}_\mathbf{A}} - \overbrace{\left(\frac{x_{OS}}{2} + \frac{\varepsilon_{OS}}{2}\right)}^{\hat{x}_{OS}/2} \tag{6.10}$$

$$\hat{x}_B = \mathbf{dec_B} \cdot \overbrace{[\mathbf{W_B} + \varepsilon_\mathbf{B}]}^{\hat{\mathbf{W}}_\mathbf{B}} + \overbrace{\left(\frac{x_{OS}}{2} + \frac{\varepsilon_{OS}}{2}\right)}^{\hat{x}_{OS}/2} \tag{6.11}$$

The difference, Δx can be shown to represent error in the current estimated weights by substituting in (6.10) and (6.11) for the two SAR codes, $\hat{x}_A$ and $\hat{x}_B$ respectively.

$$\begin{aligned}\Delta x &= \hat{x}_B - \hat{x}_A \\ &= \left(\underbrace{\mathbf{dec_B} \cdot \mathbf{W_B}}_{x_B} - \underbrace{\mathbf{dec_A} \cdot \mathbf{W_A}}_{x_A}\right) + \mathbf{dec_B} \cdot \varepsilon_\mathbf{B} - \mathbf{dec_A} \cdot \varepsilon_\mathbf{A} + \varepsilon_{OS}\end{aligned} \tag{6.12}$$

As both SAR ADCs are converting the same input at the same time, the ideal output codes x_A and x_B should be equal, leaving only the error components as shown in (6.13).

$$\Delta x = \hat{x}_B - \hat{x}_A = \mathbf{dec_B} \cdot \varepsilon_\mathbf{B} - \mathbf{dec_A} \cdot \varepsilon_\mathbf{A} + \varepsilon_{OS} \tag{6.13}$$

The calibration loop can solve for the error in current weights by collecting 1024 Δx

and solving the linear equation

$$\underset{\substack{\uparrow \\ 1024 \\ \text{Samples} \\ \downarrow}}{\overbrace{\begin{bmatrix} \Delta x_1 \\ \vdots \\ \Delta x_{1024} \end{bmatrix}}^{\mathbf{\Delta}}} = \overbrace{\begin{bmatrix} -\mathbf{dec_{A(1)}} & \mathbf{dec_{B(1)}} & 1 \\ \vdots & \vdots & \vdots \\ -\mathbf{dec_{A(1024)}} & \mathbf{dec_{B(1024)}} & 1 \end{bmatrix}}^{\mathbf{R}} \overbrace{\begin{bmatrix} \varepsilon_{\mathbf{A}} \\ \varepsilon_{\mathbf{B}} \\ \varepsilon_{OS} \end{bmatrix}}^{\mathbf{e}} \tag{6.14}$$

Similar to the matrix manipulation performed in Section 5.2.2 and due to the relaxed requirements of the LMS algorithm, the error in the weights can be approximated as

$$\overbrace{\begin{bmatrix} \varepsilon_{\mathbf{A}} \\ \varepsilon_{\mathbf{B}} \\ \varepsilon_{OS} \end{bmatrix}}^{\mathbf{e}} = \mu_e [\text{sgn}(\mathbf{R})]^{\text{T}} \overbrace{\begin{bmatrix} \Delta x_1 \\ \vdots \\ \Delta x_{1024} \end{bmatrix}}^{\Delta} \tag{6.15}$$

By removing the need to store $\mathbf{R}$, the digital calculations are simplified with a multiplication by a sgn() and summation of Δx values. After 1024 samples have been acquired the new errors are calculated with the inner LMS loop using the parameter μ_e.

$$\begin{aligned} \varepsilon_{A(i)}^{(new)} &= (1-\mu_e)\,\varepsilon_{A(i)}^{(old)} + \mu_e \left([\text{sgn}\,(\mathbf{R})]^{\mathbf{T}}\,\mathbf{\Delta} \right) \\ \varepsilon_{B(i)}^{(new)} &= (1-\mu_e)\,\varepsilon_{B(i)}^{(old)} + \mu_e \left([\text{sgn}\,(\mathbf{R})]^{\mathbf{T}}\,\mathbf{\Delta} \right) \\ \varepsilon_{OS(i)}^{(new)} &= (1-\mu_e)\,\varepsilon_{OS(i)}^{(old)} + \mu_e \left([\text{sgn}\,(\mathbf{R})]^{\mathbf{T}}\,\mathbf{\Delta} \right) \end{aligned} \tag{6.16}$$

Once the updated error values are found using the inner LMS loop, the outer loop is used to update the weights. The estimated weights are updated using the LMS step parameter μ_W in (6.17).

$$\begin{aligned} \hat{\mathbf{W}}_{\mathbf{A}}^{(\mathbf{new})} &= \hat{\mathbf{W}}_{\mathbf{A}}^{(\mathbf{old})} - \mu_W \varepsilon_{\mathbf{A}} \\ \hat{\mathbf{W}}_{\mathbf{B}}^{(\mathbf{new})} &= \hat{\mathbf{W}}_{\mathbf{B}}^{(\mathbf{old})} - \mu_W \varepsilon_{\mathbf{B}} \\ \hat{x}_{OS}^{(new)} &= \hat{x}_{OS}^{(old)} - \mu_W \varepsilon_{OS} \end{aligned} \tag{6.17}$$

This LMS loop requires two step parameters that directly affect the convergence time constant of the calibration algorithm. If the step sizes are too large, the algorithm may diverge, making it necessary to select a compromise between speed and instability.

6.2 Split-SAR IC Design

The work presented in [11] described a circuit design for the Split-SAR converter using the TSMC 0.25μm CMOS process. The original design included a schematic level simulation with Verilog-A modules for the digital timing and control blocks. This work adapts the Split-SAR IC design to the Jazz Semiconductor 0.18μm CMOS process. The majority of the analog circuit design and layout, including the comparator and switched capacitor network, was performed by Cody Brenneman. The work presented here will describe the digital hardware design for the timing and control of the Split-SAR DACs.

6.2.1 Split-SAR IC Overview

To maintain the original performance requirements from [11] the basic structure and functionality was preserved. The block diagram in Figure 6.5 represents a one of the SAR converters. This structure is very similar to a traditional SAR, with the exception of the additional DAC capacitor selection lines and the base number decoder. The serial to parallel base code block reads in the PRN serial base codes from the FPGA and sends them to the base decoder logic. The comparator decision D_k, is fed to the base decoder which selects the appropriate unit capacitors in the DAC network using the base codes.

The integrated circuit layout was organized to provide separation between the analog and digital domains while reducing interconnect complexity and propagation delay. Figure 6.6 shows the basic floor plan for the IC layout and the division between analog and digital domains. The area budget for each SAR converter is 700 μm by 800 μm for a total area of 0.56 mm^2. The bottom and top halves of the differential SAR DAC are placed on the left and right sides of the layout, respectively. This allows for the main analog signal path to enter from the top and pass down the center of the layout to the comparator at the bottom. The switches for the unit capacitors in the DAC network connect the bottom plates to one

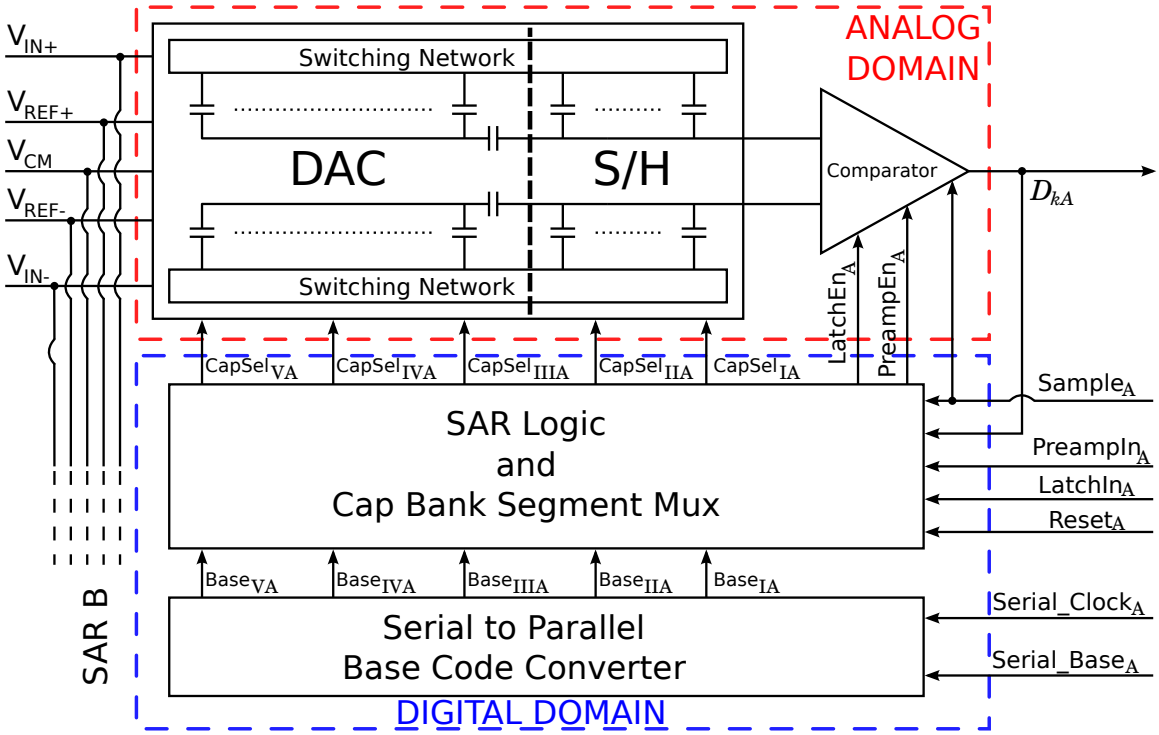

Figure 6.5: Split-SAR Circuit Block Diagram

of several analog voltage nodes. The digital signals that control these switches must not be allowed to overlap, otherwise the bottom plates may be shorted to multiple voltage sources at the same time. Therefore, a non-overlap logic block is used for each unit capacitor switch in the DAC. These blocks are placed right next to each of the analog switches in the circuit layout to minimize delay. The timing and control logic is placed at the bottom of the layout to keep it separate from the analog components. The switch control signals from the main control block are routed along the outside to the non-overlap blocks.

6.2.2 Split-SAR Analog Circuit

The analog domain consists of two main circuit blocks, the DAC switching network and the comparator. The top-level analog circuit diagram in Figure 6.2 shows the signal interconnect between the two blocks while Figure 6.6 shows their placement on the chip. Note that the MSB banks are placed at the top of the layout with the comparator towards the bottom. This is the opposite of what the circuit diagram depicts, with the MSB bank

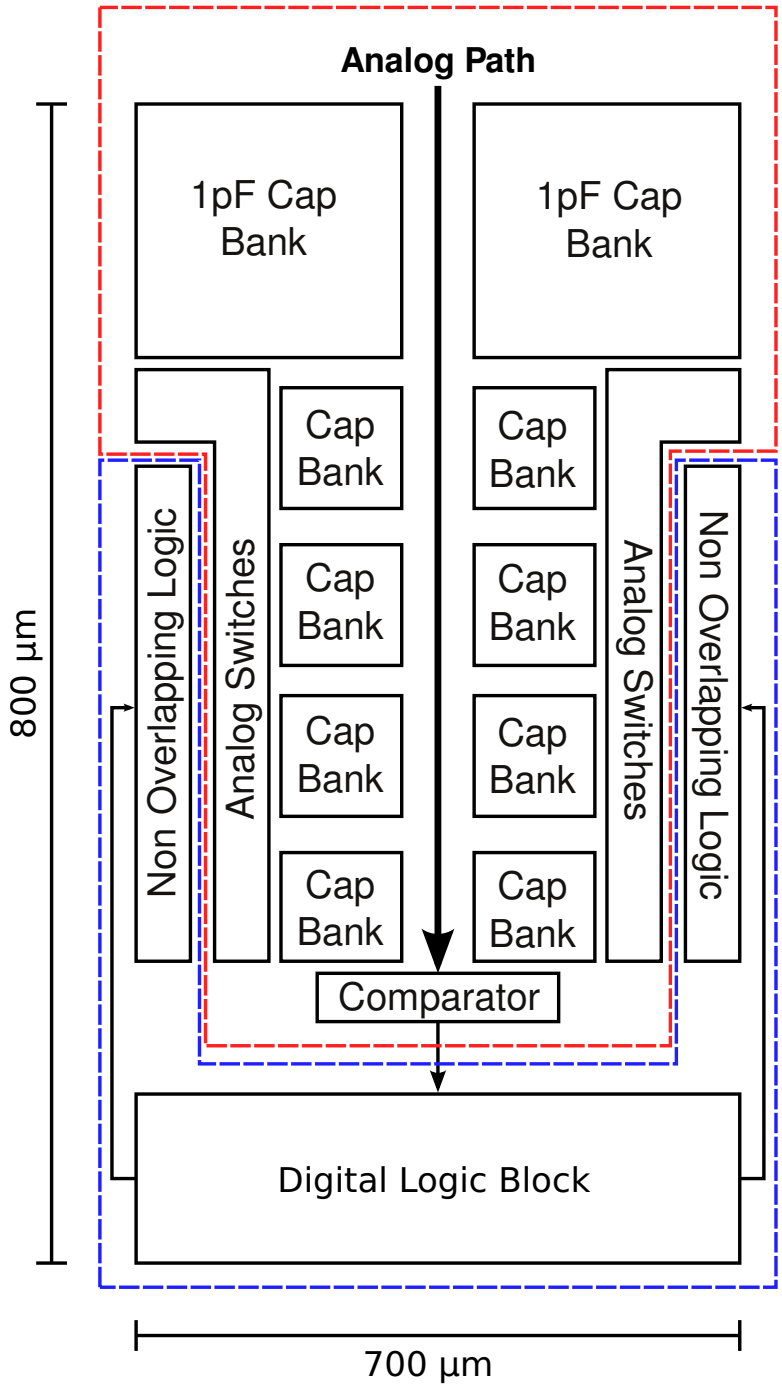

Figure 6.6: Split-SAR Layout Floor Plan

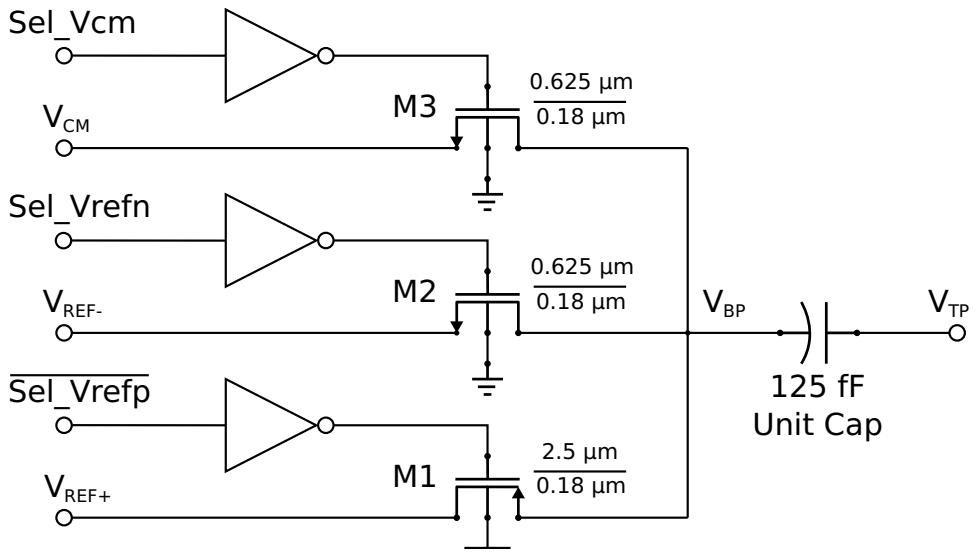

Figure 6.7: Switching Circuit for 125 fF Capacitors

placed next to the comparator. In the layout, the analog input signal enters at the top of the chip. Therefore, the input sampling switches are placed as close to the input source as possible to reduce effects from parasitics. Since only the first bank samples the input, the placement order of the capacitor banks in the IC layout is reversed from the order depicted in Figure 6.2.

Split-SAR DAC Switching Network

As stated earlier, the novel work in this SAR converter is the ability to individually select each unit capacitor in the DAC. To accomplish this, each capacitor has its own switching circuit controlled by digital signals. Capacitor banks two through five are made up of 125 fF capacitors that can switch the bottom plate between three voltage sources, V_{REF+}, V_{CM}, and V_{REF-}. The circuit diagram shown in Figure 6.7 uses a PMOS transistor for the V_{REF+} switch, and NMOS transistors for the V_{CM} and V_{REF-} switches.

The digital signals that control theses switches are generated by digital circuits that use minimum size transistors resulting in a high impedance output. The large gate sizes on the analog switches, M1 through M3, present a large capacitive load to the high impedance logic output. In order to compensate for this capacitive load, a tapered buffer is used to

lower the output impedance and prevent the digital logic from being loaded down. The tapered buffer consists of one inverter that uses transistors twice the minimum size followed by another inverter that uses transistors four times the minimum size.

Capacitor bank one uses 1 pF unit capacitors for the first four MSBs of the charge scaling DAC. These capacitors are not only larger than the capacitors in banks two through five, but they are also used to sample the analog input voltage. The 1 pF unit capacitor switching circuit, shown in Figure 6.8, is used to switch the bottom plate between the four voltage sources, V_{REF+}, V_{CM}, V_{REF-}, and V_{IN}. The circuit uses the same configuration as the one for the 125 fF capacitors with the addition of a transmission gate for the V_{IN} switch. Since the most critical phase of the conversion cycle is sampling the input, a transmission gate is used to eliminate the threshold voltage effects when using a single pass-transistor. Another difference with the 1 pF switching circuit is the use of larger transistor sizes due to the large unit capacitor.

Split-SAR Comparator

The comparator is a key component in a successive approximation converter. The comparator takes in the analog DAC voltage and converts it to a digital bit decision for the SAR. The comparator is made up of two sub components, a preamplifier and an analog latch. The goal of the preamplifier is to increase the smallest DAC voltage beyond the offset voltage of the latch. The latch used in this SAR converter has an approximate offset voltage of 20 mV. The smallest DAC voltage that needs to be resolved is one half LSB as expressed by (6.18).

$$\frac{1}{2}\ \text{LSB} = \frac{3.6V}{2^{17}} = 27.5\ \mu V \tag{6.18}$$

Therefore, the minimum required gain for the preamplifier is 20 μV/27.5 μV or a gain of 730. While it is possible to design a single stage amplifier with a gain of 730, the high impedance node would reduce the speed of the amplifier. In order to achieve necessary speed, the preamplifier is broken into four stages with a gain of 5.2 each.

The circuit for each stage of the preamplifier, shown in Figure 6.9, consists of a differen-

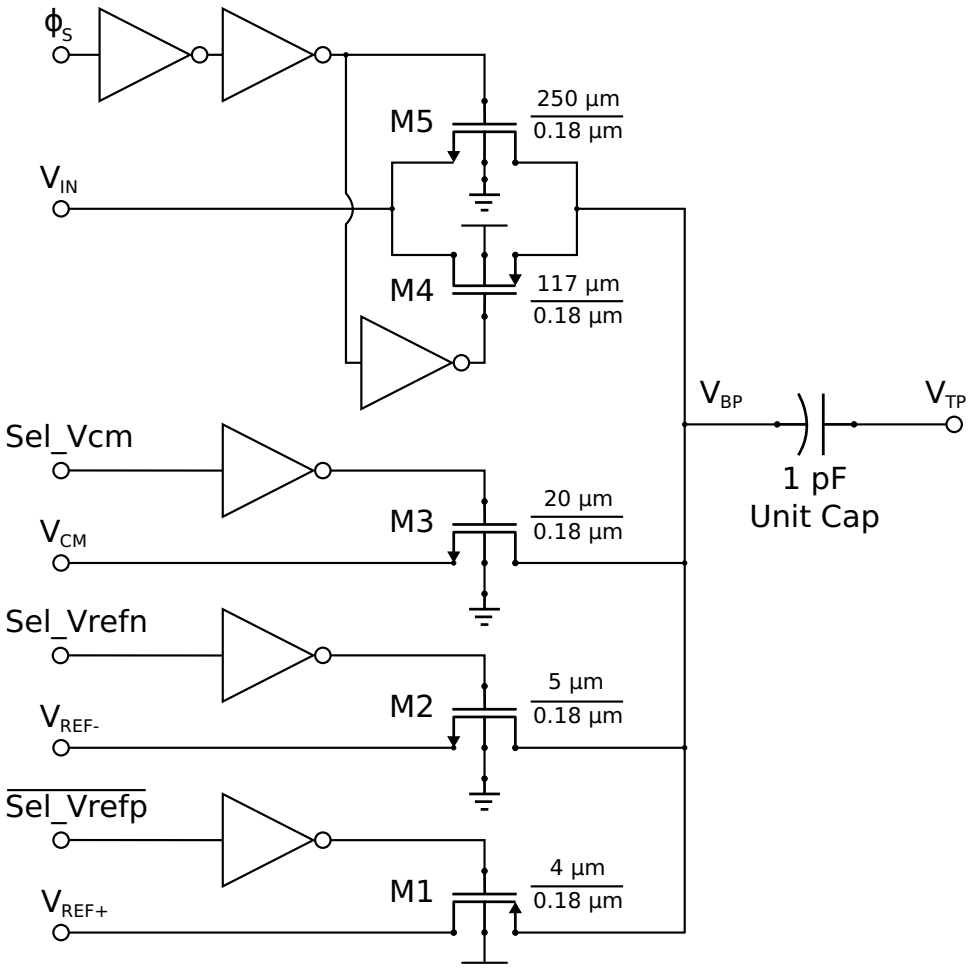

Figure 6.8: Switching Circuit for 1 pF Capacitors

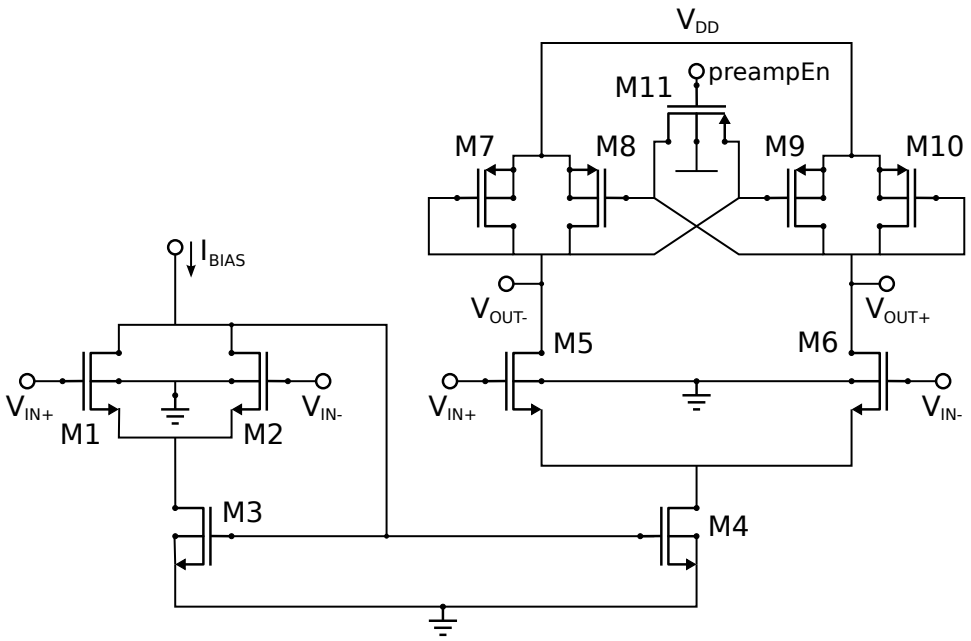

Figure 6.9: Preamplifier Single Stage Schematic

Table 6.1: Transistor Sizes for the Preamplifier Stages

Stage	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10	M11
1	$\frac{6\mu m}{180nm}$	$\frac{6\mu m}{180nm}$	$\frac{3\mu m}{1\mu m}$	$\frac{3\mu m}{1\mu m}$	$\frac{6\mu m}{180nm}$	$\frac{6\mu m}{180nm}$	$\frac{11\mu m}{0.3\mu m}$	$\frac{10.5\mu m}{0.3\mu m}$	$\frac{10.5\mu m}{0.3\mu m}$	$\frac{11\mu m}{0.3\mu m}$	$\frac{20\mu m}{180nm}$
2-4	$\frac{3\mu m}{180nm}$	$\frac{3\mu m}{180nm}$	$\frac{3\mu m}{1\mu m}$	$\frac{3\mu m}{1\mu m}$	$\frac{6\mu m}{180nm}$	$\frac{6\mu m}{180nm}$	$\frac{3.25\mu m}{0.3\mu m}$	$\frac{2.75\mu m}{0.3\mu m}$	$\frac{2.75\mu m}{0.3\mu m}$	$\frac{3.25\mu m}{0.3\mu m}$	$\frac{20\mu m}{180nm}$

tial amplifier with an enable switch, M11. When the stage is disabled, the PMOS transistor M11 is turned on, pulling the differential output nodes to a common mode. When the stage is turned on, the PMOS transistor turns off, releasing the differential output nodes and allowing them to settle to a final output voltage.

The final component in the comparator circuit is the analog, regenerative latch shown in Figure 6.10. The latch is essentially two, cross-coupled inverters whose outputs are tied together with PMOS transistors. When the PMOS transistors are turned on, the outputs of the inverters are brought together and the latch is placed into a tracking mode. At the same time, transistors M3 and M4 are also turned on, equalizing the current through both inverters. When a differential, analog input signal is placed at the inputs to the preamplifiers M1 and M2, it starts to create a small current imbalance. Once the switches are turned off and the inverter outputs are released, the small current imbalance forces one inverter output

high and the opposite inverter output low. This final logic decision indicates whether or not the analog DAC voltage was positive or negative. The decision is fed to the digital logic to be stored in the SAR and switch the next set of capacitors to their appropriate reference voltage.

6.2.3 Split-SAR Digital Logic

A more detailed block diagram of the digital logic for the Split-SAR is shown in Figure 6.11. The main digital logic block shown at the bottom of the floor plan contains the internal timing and control logic, the array of twenty 2-bit registers, and all of the base code logic. The timing and control logic takes in the three external timing signals, sample, latch_In, and preamp_In. This timing and control block then distributes these timing signals to control the comparator circuit and when to switch the DAC capacitors during the conversion cycle.

A complete hardware description of the main digital logic block is presented in Appendix C. The hierarchical structure of the logic block is presented in Figure 6.12. The top level consists of five blocks, the timing and control logic block, the SAR array block, the base decoder block, and two capacitor selection multiplexers. The control block generates the three main timing signals, ϕ_S, latchEn, and preampEn.

The timing diagram of the three signals is shown in Figure 6.13. A full conversion cycle is designed to take a sampling period T_S, of 1 μs. The first 200 ns of the cycle are used for the sample and hold modes for SAR converter, where the control signal ϕ_S is active. During this time, the first bank of capacitors sample the input voltage while the remaining capacitors are tied to the common mode voltage. The bit cycling mode takes up the remaining 800 ns of the sampling period. Each bit decision is given 40 ns to settle on a final value. The first 20 ns are allotted for digital propagation from the last bit decision and to allow the DAC voltage to settle after the capacitors have been switched. The next 10 ns has only the preampEn signal active to enable the pre-amplifiers in the front-end of the comparator. These preamps amplify the differential DAC voltage to the rails. For the remaining 10 ns, both the preampEn and latchEn are activated to keep the preamps enabled along with the analog latch. The latch is the output stage of the comparator and sets the comparator decision for that bit. The analog latch output is sent to the digital register array which

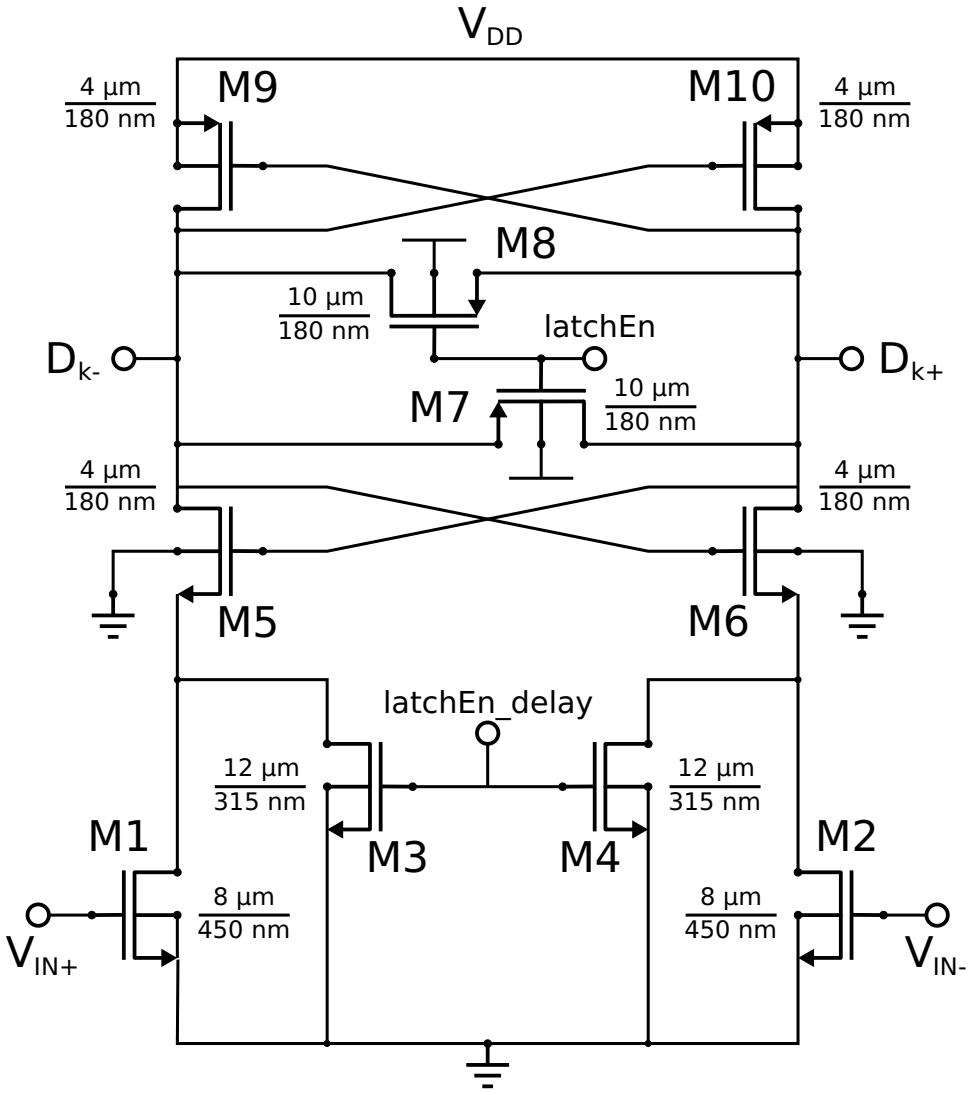

Figure 6.10: Preamplifier Regenerative Latch

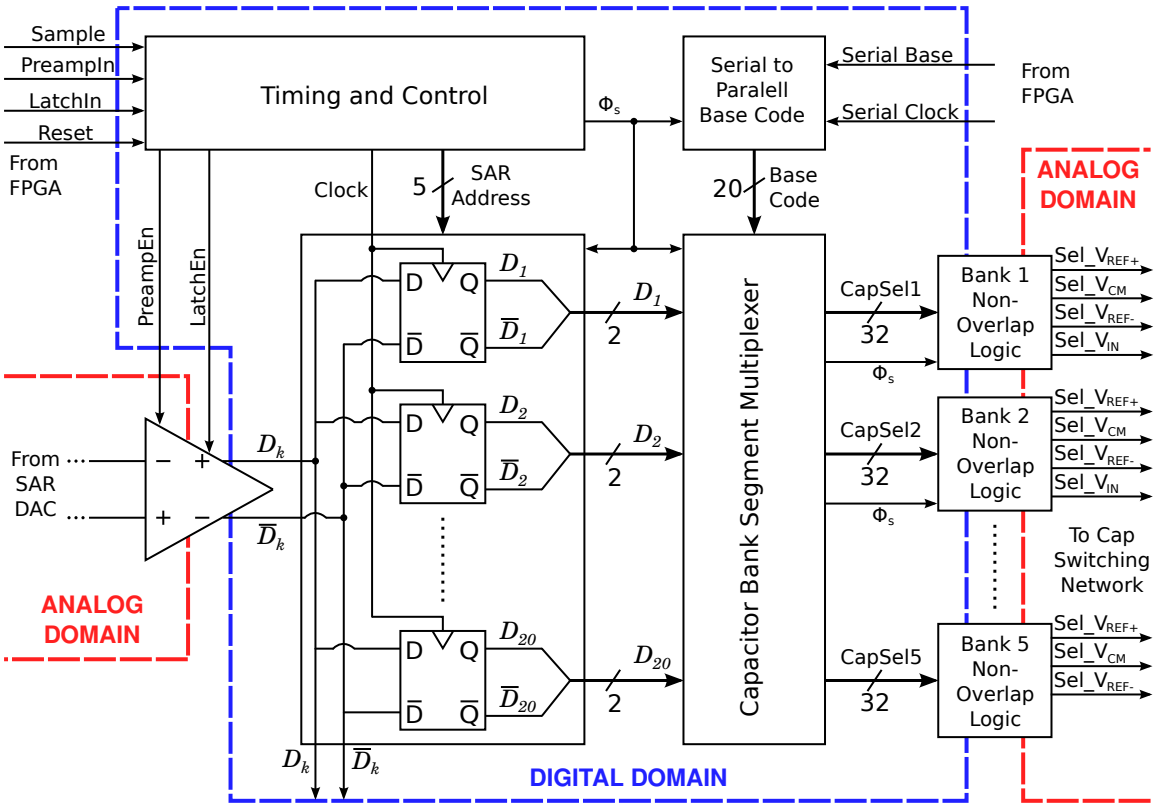

Figure 6.11: Digital Block Diagram for the Split-SAR

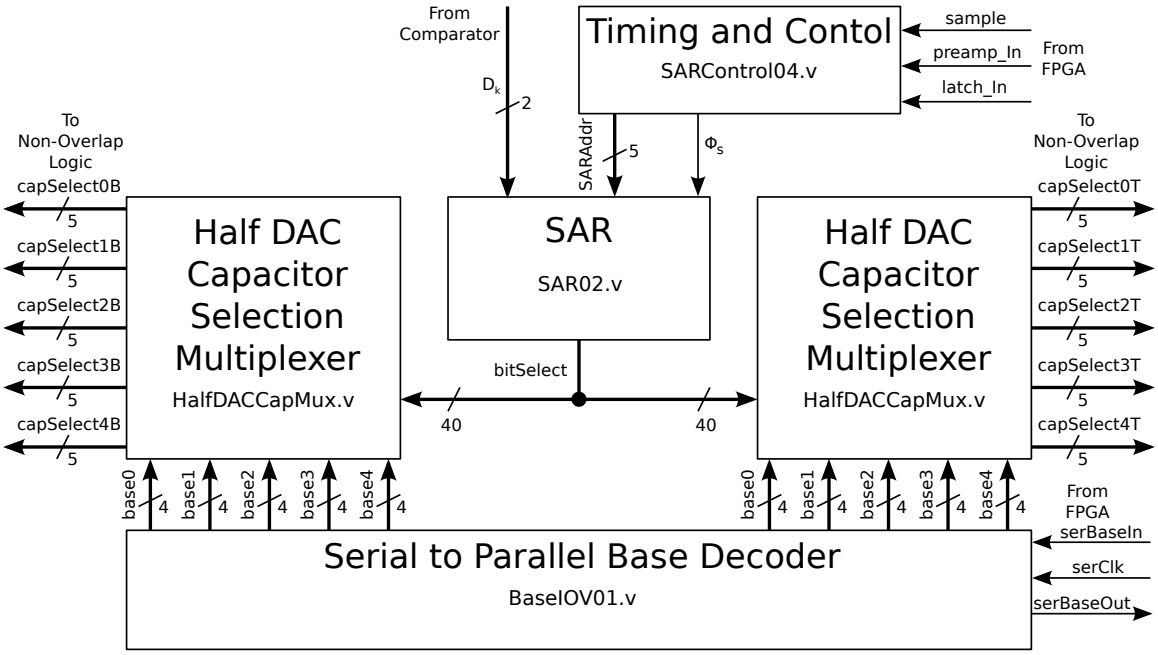

Figure 6.12: Digital Circuit Hierarchy Block Diagram

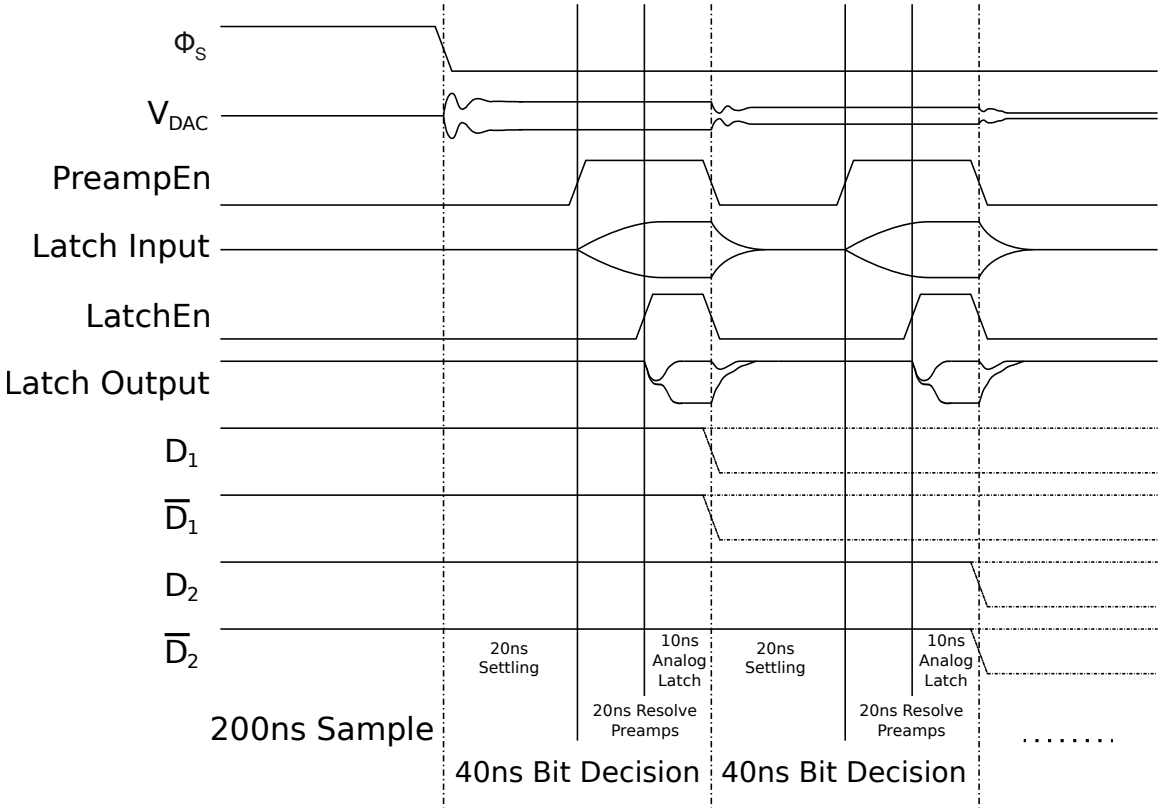

Figure 6.13: Timing Diagram of Control Signals

eventually gets used to select the appropriate capacitors in the DAC.

The array of 2-bit registers store each of the comparator decisions for a single conversion. At the start of the cycle during the sample mode, the forty registers are reset to 11, the signal to select the common mode voltage in the DAC. When the comparator makes a decision, it outputs either a 10 for a +1 or a 01 for a -1. This decision is stored in the appropriate bit register for base decoding.

The base decoding logic block serves two functions, the serial to parallel transformation of the base code input and the decoding of the bit decisions to individual unit capacitor selections. As stated earlier, the two input signals that are fed into this block are the 20-bit serial sequence of the five base codes and the accompanying serial clock. This serial code is fed into the chip during the preceding conversion and then stored as five, 4-bit base codes at the start of the current cycle. The five base codes are fed into the capacitor selection multiplexers, which are used to control the capacitor switches.

The selection multiplexers perform the base decoding logic by using the comparator bit decisions and 4-bit base codes to select the appropriate unit capacitors in the DAC network. The multiplexers are divided into five blocks, one for each capacitor bank. Each block takes in one 4-bit base code and four bit decisions stored in the SAR array. The multiplexer splits up the four bit decisions into sixteen unit capacitor decisions using the base code. The outputs of the multiplexer block are then routed from the main digital circuit to the non-overlap logic blocks next to the capacitor switches.

The logic diagram for the non-overlap blocks is shown in Figure 6.14. There are two types of non-overlap blocks used for the DAC circuit. The first bank of capacitors (the 1 pF block) can connect to one of four analog voltage nodes; the input signal voltage for sampling, V_{REF+}, common mode, or V_{REF-}. The remaining banks of 125 fF capacitors can only switch between three voltages; V_{REF+}, common mode, or V_{REF-}. There are five or seven signals required to connect the non-overlap blocks to the analog switches. However, only two or three signals are used to connect the rest of the SAR digital logic to the non-overlap blocks. Placing the non-overlap blocks next to the analog switches, rather than integrating them into the SAR logic block, has the added advantage of reducing the number of switching signals to be routed across the chip.

The digital block does not contain the logic necessary for generating the PRN sequence of base codes, applying the digital calibration and error correction. These functions were left to be performed off-chip to allow for flexibility while developing the algorithm. This allows the calibration and error correction to be prototyped in MATLAB and a development FPGA. The results from the final IC fabrication and MATLAB analysis are presented in the next section.

6.3 Split-SAR Results

The final Split-SAR layout was fabricated on a chip containing another design project. The entire Split-SAR circuitry used up 1.2 mm^2, not including the pad ring. A die photo of the Split-SAR layout is displayed in Figure 6.15. The digital and analog blocks are outlined in the photo to demonstrate the floor plan used throughout the IC design process. The

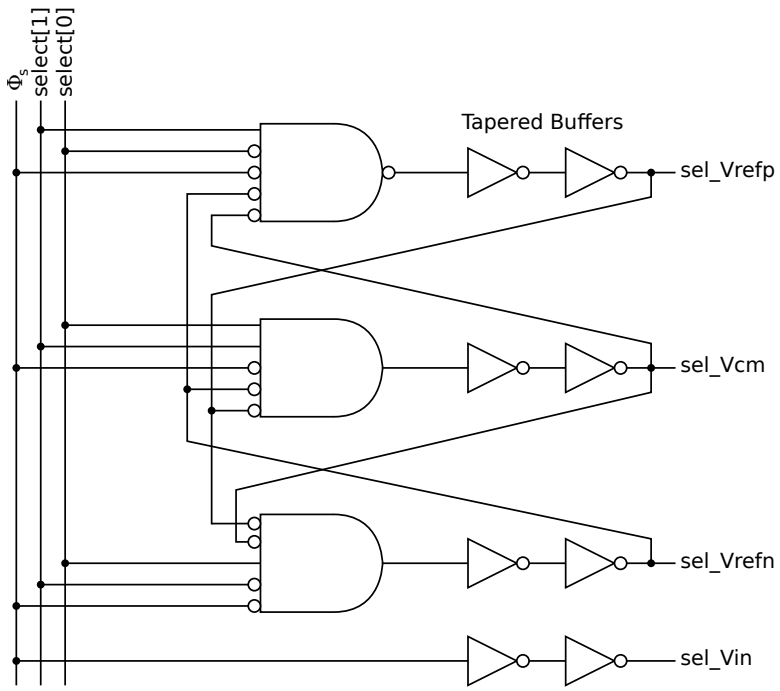

Figure 6.14: Non-Overlap Logic Block

separation of the analog and digital domains in the layout floor plan allows for grouping the analog pins together at the top of the chip while the digital pins are kept on the sides. This provides for easy isolation of the analog and digital signals during evaluation.

For evaluation, a Tektronix AFG3021B function generator provides the analog test signal for the ADC. Since the generator uses a 14-bit DAC to generate the waveform, the analog signal is filtered using a bandpass from 100 Hz through 20 kHz. The filter reduces the harmonic distortion and quantization noise level to achieve 16-bit performance. The signal is sent through a single-end-to-differential-end (SE-DE) circuit before being routed to the Split-SAR ADC. A FPGA is used to provide timing and control signals, including the PRN base codes for selecting the unit capacitors. The output of the SAR ADC is connected to a digital logic analyzer capable of acquiring up to approximately 104 000 samples for calibration and analysis.

The acquired data block is replicated 100 times before being processed by the split calibration algorithm. Figure 6.16 shows how the algorith convergence uses different input signals ranging from DC to full-scale. This demonstrates the ability to process various input

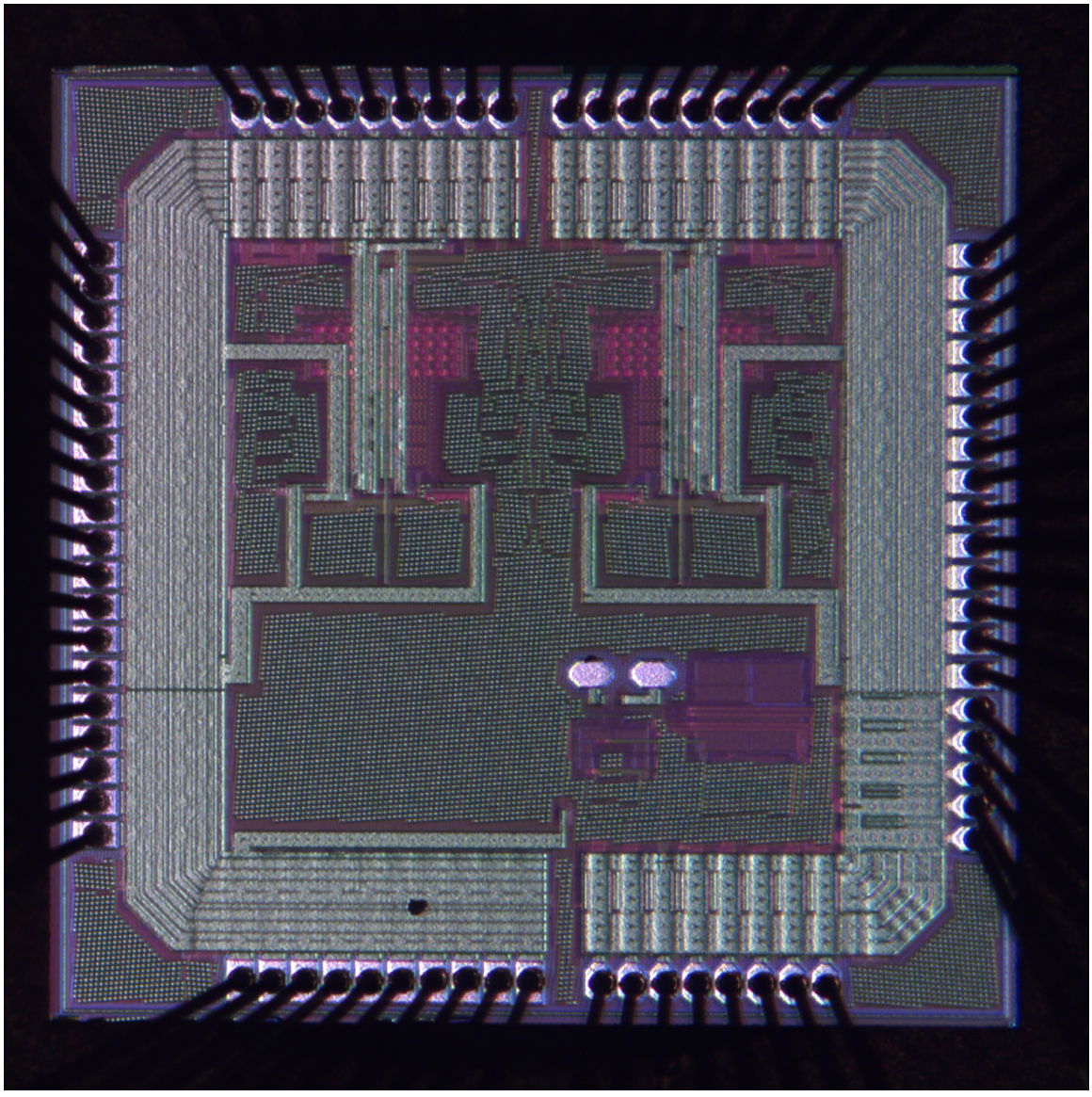

Figure 6.15: Die Photo of Split-SAR

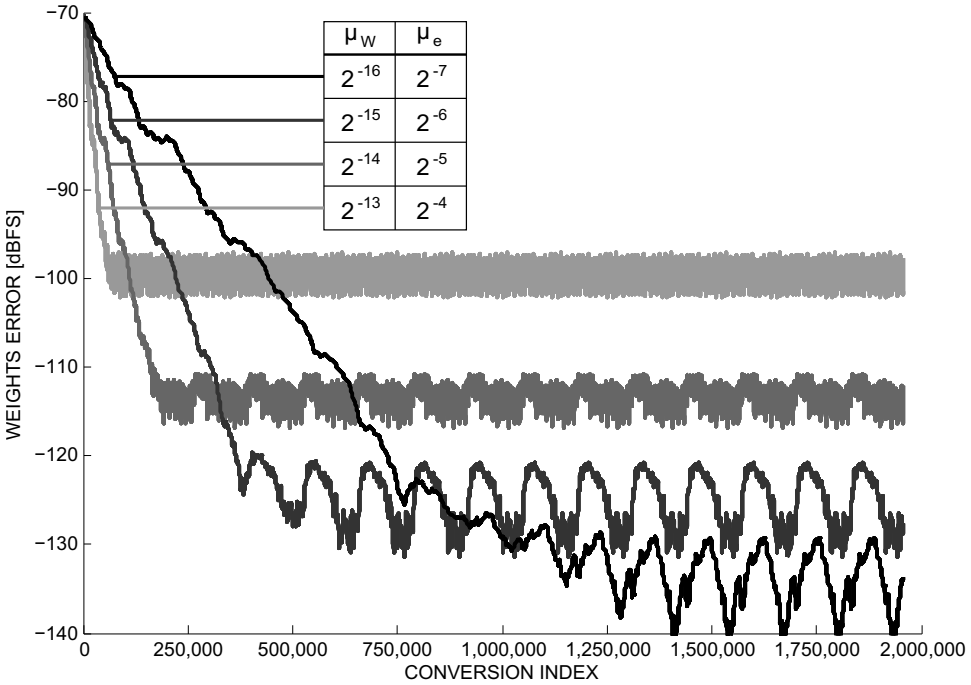

Figure 6.16: Split-SAR Algorithm Convergence with Different Waveforms

signals regardless of the waveform used. The ability to calculate new weights using a DC input is an improvement of [42], which requires a varying signal. Adjusting the LMS loop parameters produces different rates and levels of convergence. Similar to the parameters using the Split-Interleaved converter, the loop in the Split-SAR algorith has a trade-off between speed and accuracy. The effects of different LMS parameters is shown in Figure 6.17. A faster rate of convergence results in a higher level of error in the estimated weights and vice versa. To achieve a 16-bit level of accuracy, the error level must remain below -100 dB.

For the purposes of testing, a sinewave of 18 kHz is used with the converter sample rate set to 100 kS/sec. The output spectrum of the signal prior to calibration is shown in Figure 6.18. The SNR is 59.09 dB with a THD of -70.41 dB for a total SNDR of 59.10 dB. After the algorithm is allowed to achieve full convergence, the spectrum in Figure 6.19 shows a small improvement. The SNR increases by 4 dB to 63.05 dB, while the THD decreases by 2 dB to -72.5 dB. The total SNDR of the calibrated output is 63.07 dB, an improvement of 4 dB. From 2.10, the ENOB of the calibrated output is a small improvement from 9.5 bit

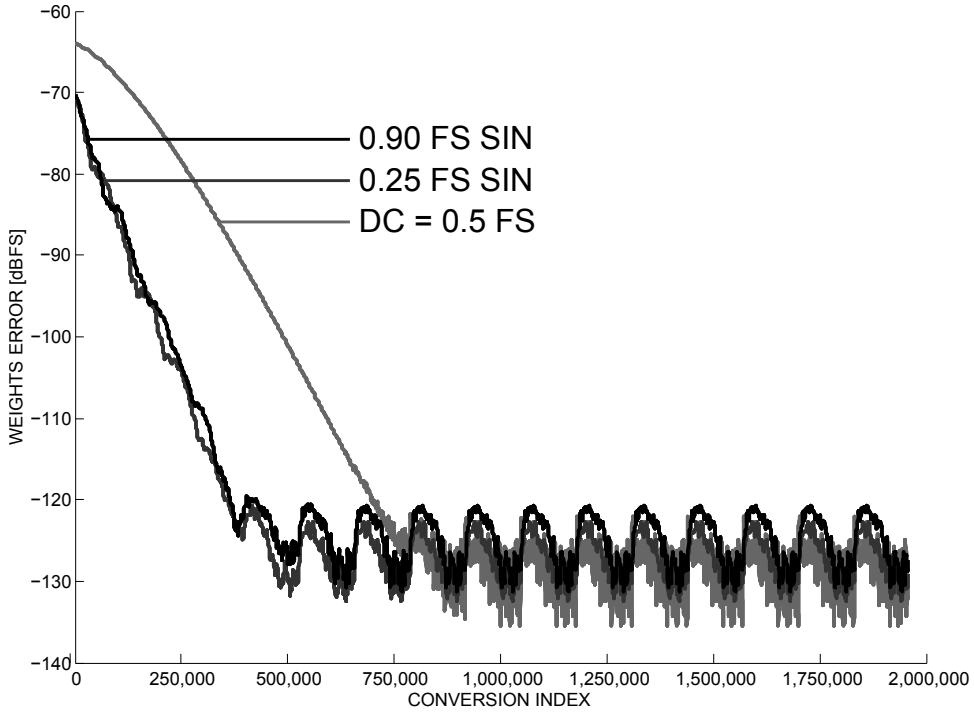

Figure 6.17: Split-SAR Algorithm Convergence with Different LMS Parameters

to 10.2 bits, which is much less than the desired 15 to 16 bit range.

It is clear from the results that the noise level is the limiting factor to the overall performance of the SAR converter. In early tests, it was discovered that the gain in the preamplifier was unstable at the target bias current. The transconductance in the M8 and M9 PMOS transistors is very sensitive to variations in the drain current, resulting in an oscillating output. This causes the latch to make bad decisions and produce stuck output codes. To achieve a functional converter, the bias current was lowered until the oscillations were reduced and the stuck codes were eliminated. Unfortunately, lowering the bias current decreases the gain on the preamplifier, thereby increasing the output referred noise level. This results in a lowered SNR performance as seen in the previous figures.

An additional concern is the large third order harmonic seen in both Figures 6.18 and 6.19. While there is a slight improvement from -70.41 dB to -72.54 dB, the third harmonic remains the dominant in both the uncorrected and corrected case. The effects of this third order harmonic is evident in the INL plot shown in Figure 6.20. The plot represents a linearity test by sampling 100 DC values and measuring the difference between the output

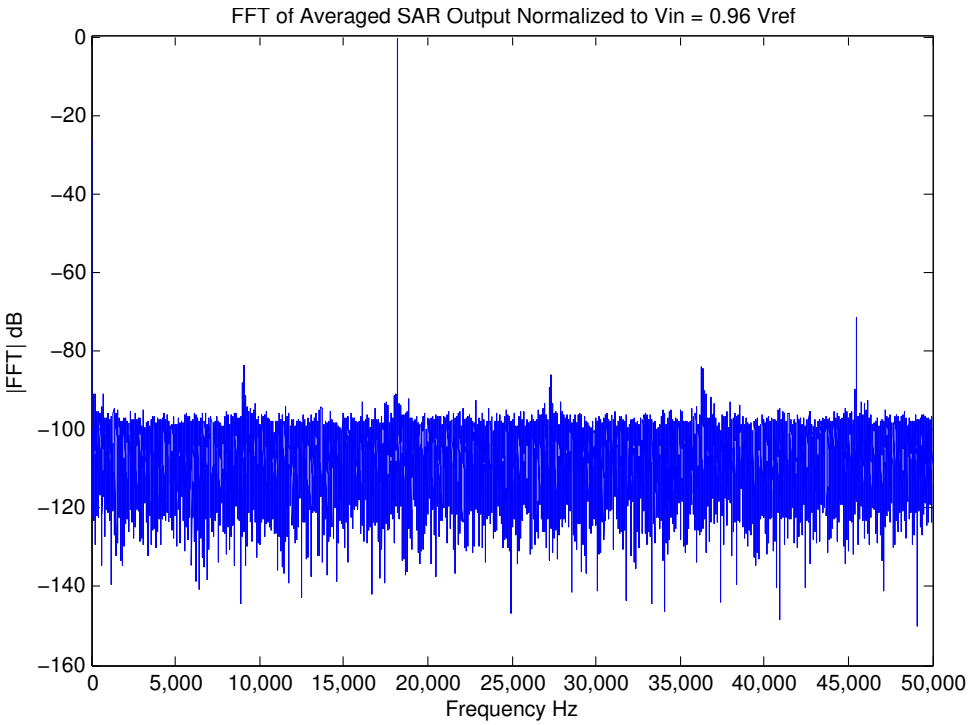

Figure 6.18: FFT of Split-SAR Output before Calibration

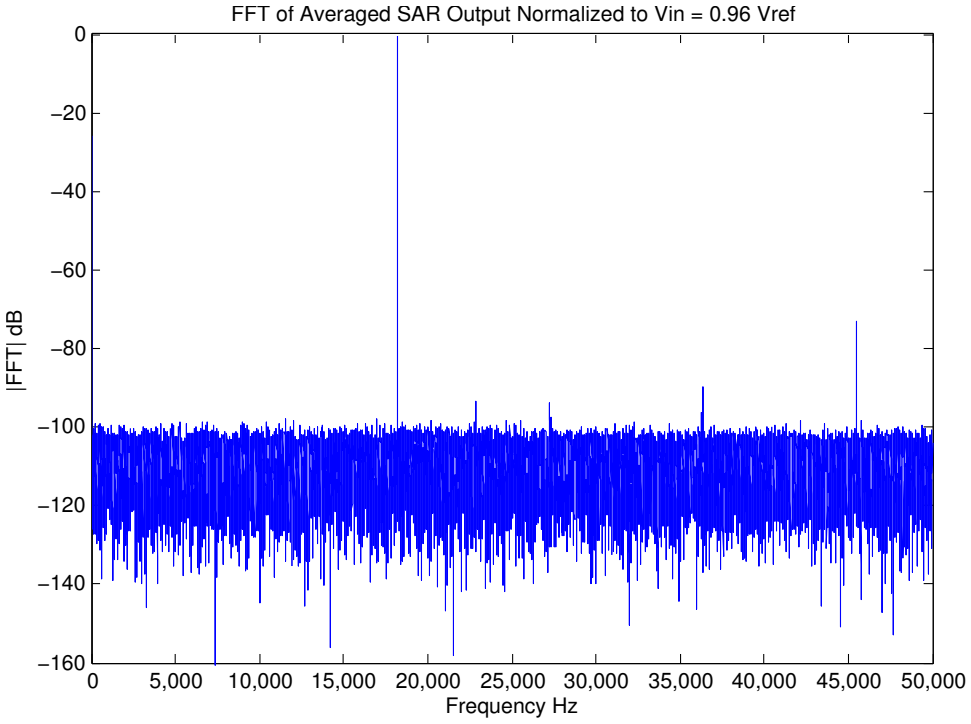

Figure 6.19: FFT of Split-SAR Output before Calibration

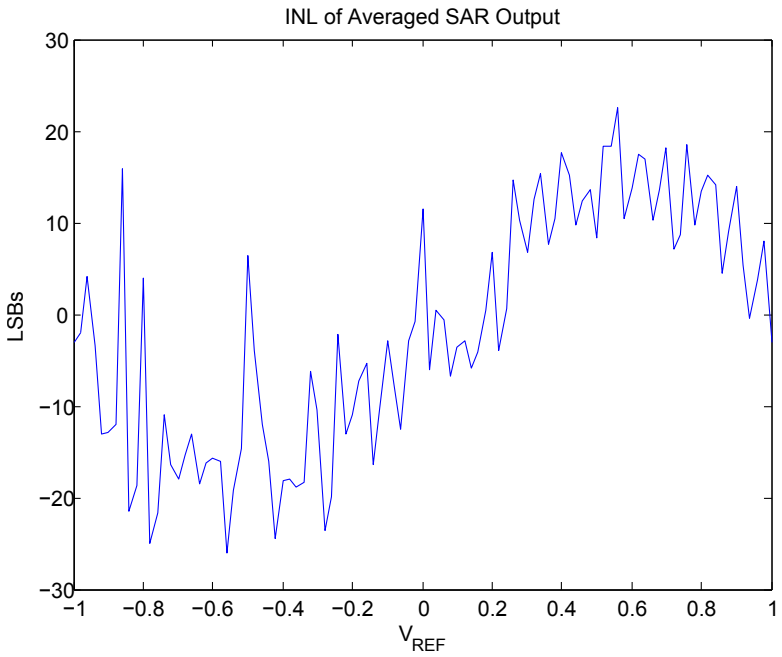

Figure 6.20: 100 Point INL of Calibrated Split-SAR

codes and the analog input. The result shows a clear third order effect across the entire range of the converter with a +25/-25 LSB error. The INL plot in Figure 6.21 shows the linearity of the converter with the third order harmonic removed. This shows a linear performance of +10/-12 LSB with not including the few outliers. The source of this third order distortion can be attributed to a combination of capacitor mismatch and nonlinearity in the input sampling switches. When the input signal is sampled onto the capacitors in the charge DAC, these switches are turned on by the digital control lines. When the switches are turned off, the charge built up on the gates must be removed. This can result in a phenomenon known as charge injection, where some to the gate charge is leaked into the sampling capacitors. This creates non-linearity distortion during the sample and hold phase of the conversion process. Unfortunately, this error can not be removed by calibrating out capacitor mismatch, limiting the performance of the SAR ADC.

Overall, the split-SAR method does show some improvement in ADC performance by estimating capacitor weights. In fact, it has been shown that the algorithm can still converge in the presence of noise and distortion due to sampling. Unfortunately, due to the limitations as a result of other non-ideal effects, the final calibrated SAR converter has a performance

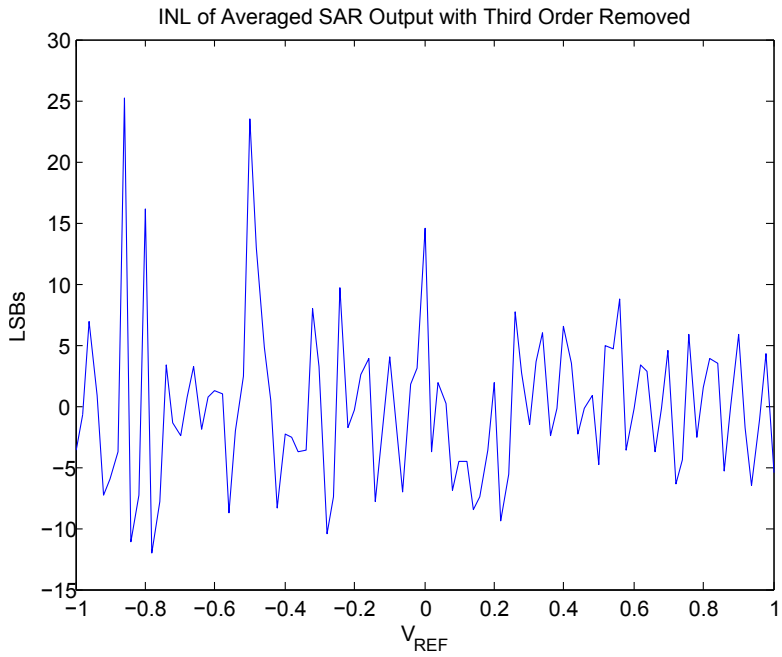

Figure 6.21: 100 Point INL of Calibrated Split-SAR with Third Order Effect Removed

level 5 bits lower than expected. It is important to note, however, that the algorith is successful in estimating capacitor weights that improve the operation of the converter.

Chapter 7

Conclusions

The application of the Split-ADC calibration method to the TI and SAR architectures was presented in this thesis. Chapters 2 and 3 provided a detailed background on the two architectures and the nature of the errors associated with them. Chapter 4 presented previous calibration techniques and their respective advantages and disadvantages. It went on to review the Split-ADC method for calibration as well as the three advantages over previous techniques. Chapters 5 and 6 described the research involved with adapting the Split-ADC method to the time interleaved and SAR architecture, respectively.

While the concept of the Split-ADC as a calibration method for cyclic converters was demonstrated in [9, 10], the adaptation to different architectures was non-trivial. The Split-Interleaved or Splinta calibration architecture was shown using simulations in both MATLAB and layout. The findings from this research was additionally presented in [10]. The research and development of the Split-SAR architecture was presented in [11] however, this was done in MATLAB and schematic simulations only. The work presented here extended the Split-SAR design by implementing it in an IC fabricated in the Jazz Semiconductor 0.18μm CMOS process. The Split-ADC algorithm was applied off-chip to demonstrate the concept in practice.

7.1 Future Work

Although much research was done to show the Splinta and Split-SAR concepts, there are several paths that can be followed to extend the work presented here. Since the Split-Interleaved architecture uses any type of Nyquist rate sub-ADCs, it can be implemented on either a PCB or IC. Research is currently being done on implementing either a 4:1 or 3:1 Splinta architecture using discrete AD7621 ADCs on a printed circuit board. This method constructing a Splinta system may not be the most efficient in terms of resources such as power and area, but it has the advantage of being less costly to implement than on an IC. This can allow for faster prototyping and direct hardware modification setup. An IC version of the Splinta was attempted with the help of Rosa Croughwell and Analog Devices. Unfortunately, the process used was different than the one that the original AD7621s were made in. As a result, the capacitor mismatch was much greater in the Splinta, more than what the calibration DAC could handle. This meant that the converters on the chip could not have their non-linearity calibrated out before attempting the Splinta operation.

The Split-SAR was implemented on chip, however, a few problems were noted during the testing. The design of the preamplifier stages did not fully account for instability which resulted in bad comparator decisions and stuck codes. Correcting for this after fabrication by reducing the preamplifier gain decreased the noise performance of the ADC. The non-linearity effects due to sampling also presented a problem by introducing a third order harmonic that was not a result of capacitor mismatch. Despite these problems, it is the belief of this author that the Split-SAR method is capable of calibrating capacitor mismatch with a redesign of the integrated circuit.

Glossary

V_{FS}	Full-Scale Voltage
V_{pk}	Peak Voltage
f_S	Sample Frequency
A/D	Analog-to-Digital Converter
ADC	Analog-to-Digital Converter
CCD	Charge Coupled Device
CMOS	Complementary Metal Oxide Semiconductor
CMRR	Common Mode Rejection Ratio
CRT	Cathode Ray Tube
D/A	Digital-to-Analog Converter
DAC	Digital-to-Analog Converter
DAQ	Data Acquisition
DE	Differential End
DNL	Differential Non-Linearity
DSP	Digital Signal Processing
ENOB	Effective Number of Bits
FFT	Fast Fourier Transform

FS	Full Scale
FSR	Full Scale Range
HDL	Hardware Description Language
IC	Integrated Circuit
INL	Integral Non-Linearity
IP	Intellectual Property
LFSR	Linear Feedback Shift Register
LMS	Least Mean Squares
LSB	Least Significant Bit
LUT	Lookup Table
MOSFET	Metal Oxide Semiconductor Field Effect Transistor
MS/sec	Mega-Samples per second
MSB	Most Significant Bit
MSps	Mega-Samples per second
NMOS	N-Channel Metal Oxide Semiconductor
PMOS	P-Channel Metal Oxide Semiconductor
PRN	Pseudo Random Number
RMS	Root Mean Square
RSS	Root Sum Square
SAR	Successive Approximation Register
SE	Single End

SFDR	Spurious Free Dynamic Range
SNDR	Signal-to-Noise-and-Distortion Ratio
SNR	Signal-to-Noise Ratio
THA	Track and Hold Amplifier
THD	Total Harmonic Distortion
TI	Time Interleaved

Appendix A

Split-Interleaved MATLAB Code

./source_code/splinta/adlec_figs_V01d.m

```
% adlec_figs.m

% makes figures used in ADLEC presentation

% First run setup
 %multi_ADC_setup05a

 % Then quantizer
%quantizer01

% Then actually correct errors
 %multi_ADC_5ptderivEst_cor01a
%

% Plot of rms variation of output difference vs conversion
 %Nchunks=floor(size(VoutCor,2)/100);
%RMS_errors=zeros(1,100*Nchunks);

%for ek=1:Nchunks
 %    RMS_errors((100*(ek-1)+1):100*ek)=std(Vin((100*(ek-1)+1):100*ek)-VoutCor((100*(ek-1)+1):100*ek));
%end

figure(5);

semilogy(RMS_errors_10_5,'LineWidth',2); hold on;
 semilogy(RMS_errors_11_6,'LineWidth',2);
semilogy(RMS_errors_8_3,'LineWidth',2); hold off;
 %semilogy(RMS_errors_10_5,'LineWidth',2)
%semilogy(RMS_errors_12_7,'LineWidth',2)
 set(gca,'Xlim',[0 150000])
set(gca,'Ylim',[1e-5 31.6e-3])
 Xlabel(['CONVERSION INDEX'],'FontSize',10,'Fontweight','normal')
Ylabel(['rms ADC ERROR [dBFS]'],'FontSize',10,'Fontweight','normal')
 set(gca,'YTickMode','manual')
%set(gca,'YTick',{1e-5; 1e-4; 1e-3; 1e-2; 1e-1})
```

```
set(gca,'YTickLabel',{'-80';'-60';'-40';'-20'})
set(gca,'XTickMode','manual')
%set(gca,'XTick',{0; 50000; 100000; 150000; 200000})
set(gca,'XTickLabel',{'0';'50000';'100000';'150000'})
```

./source_code/splinta/fft_win_intl_an02.m

```
%************************************************************************
% FFT Anlaysis With Windowing for Intl Simulation vers. 01
% fft_win_intl_an01.m
% 06.11.06
% Compute, normalize and plot the FFTs with Blackman windowing of Corrected
% and Uncorrected Vout.
% This program was designed for the the purposes of presentation.
%
%************************************************************************

numCycles = 11;                 % Number of cycles for the sinewave
Td1=1/fd1*1E6;                  % Get Scaled Period T1
Td2=1/fd2*1E6;                  % Get Scaled Period T2
Ts=1/fs*1E6;                    % Get Scaled Period Ts
cycSmpl = (Td1/Ts)*(Td2/Ts);    % Get the number of samples per cycle

% Try and get an integral number of cycles
totalSmpls = 4*128*round(numCycles*cycSmpl);

lng_Vout=length(VoutCor)-Ncoef;     % Get the number of samples for Vout
%t_end=t(1:totalSmpls);          % Setup time for the last 16 cycles
% Setup Vout for the last 16 cycles

% Get Samples of Fully Corrected Ouput after Full Convergence
VoutCor_end=VoutCor(lng_Vout-totalSmpls+1:lng_Vout);
% Get Samples of G and OS Corrected Ouput (No AptD) after Full Convergence
VoutBad_end=VoutBad(lng_Vout-totalSmpls+1:lng_Vout);
% Get Sample of Uncorrected Output (Before Convergence)
VoutBad_start=VoutBad(1:totalSmpls);

Vin_end=Vin(lng_Vout-totalSmpls+1:lng_Vout);


% Process FFT of Vin
VIN=abs(fft(Vin_end));           % FFT of Vin
VIN_dB=20*log10(VIN);          % Convert to dB
maxVIN=max(VIN_dB);            % Normalize to zero dB
VIN_n=VIN_dB-maxVIN;
%VIN_c=max(VIN_n,-200);         % Clip at -100 dB

% Use a Windowing technique to reduce spectral leakage
w=blackman(length(VIN));
VINw=abs(fft(Vin_end.*w',1*totalSmpls)); % FFT of Vin w/ Window
VINw_dB=20*log10(VINw);          % Convert to dB
maxVINw=max(VINw_dB);            % Normalize to zero dB
VINw_n=VINw_dB-maxVINw;
VINw_c=max(VINw_n,-200);         % Clip at -200 dB

% Process FFT of Vout With No Correction (Before Convergence)
VOUT1=abs(fft(VoutBad_start));      % FFT of Vout
VOUT1_dB=20*log10(VOUT1);       % Convert to dB
maxVOUT1=max(VOUT1_dB);         % Normalize to zero dB
VOUT1_n=VOUT1_dB-maxVOUT1;
%VOUT1_c=max(VOUT1_n,-100);      % Clip at -100 dB
```

```
% Use a Windowing technique to reduce spectral leakage
w=blackman(totalSmpls);
VOUT1w=abs(fft(VoutBad_start.*w',1*totalSmpls)); % FFT of Vin w/ Window
VOUT1w_dB=20*log10(VOUT1w);          % Convert to dB
maxVOUT1w=max(VOUT1w_dB);            % Normalize to zero dB
VOUT1w_n=VOUT1w_dB-maxVOUT1w;
VOUT1w_c=max(VOUT1w_n,-200);         % Clip at -200 dB

% Process FFT of VoutRaw G and OS Corrected Ouput (No AptD) after Full
% Convergence
VOUT1b=abs(fft(VoutBad_end));      % FFT of Vout
VOUT1b_dB=20*log10(VOUT1b);       % Convert to dB
maxVOUT1b=max(VOUT1_dB);         % Normalize to zero dB
VOUT1b_n=VOUT1b_dB-maxVOUT1b;
%VOUT1_c=max(VOUT1_n,-100);      % Clip at -100 dB

% Use a Windowing technique to reduce spectral leakage
w=blackman(totalSmpls);
VOUT1bw=abs(fft(VoutBad_end.*w',1*totalSmpls)); % FFT of Vin w/ Window
VOUT1bw_dB=20*log10(VOUT1bw);          % Convert to dB
maxVOUT1bw=max(VOUT1bw_dB);            % Normalize to zero dB
VOUT1bw_n=VOUT1bw_dB-maxVOUT1bw;
VOUT1bw_c=max(VOUT1bw_n,-200);         % Clip at -200 dB

% Process FFT of VoutCor after Full Convergence
VOUT2=abs(fft(VoutCor_end));  % FFT of Vout
VOUT2_dB=20*log10(VOUT2);       % Convert to dB
maxVOUT2=max(VOUT2_dB);         % Normalize to zero dB
VOUT2_n=VOUT2_dB-maxVOUT2;
%VOUT2_c=max(VOUT2_n,-100);      % Clip at -100 dB

% Use a Windowing technique to reduce spectral leakage
w=blackman(length(VOUT2));
VOUT2w=abs(fft(VoutCor_end.*w',1*totalSmpls)); % FFT of Vin w/ Window
VOUT2w_dB=20*log10(VOUT2w);          % Convert to dB
maxVOUT2w=max(VOUT2w_dB);            % Normalize to zero dB
VOUT2w_n=VOUT2_dB-maxVOUT2w;
VOUT2w_c=max(VOUT2w_n,-200);         % Clip at -200 dB

% Plot Vin
figure(1);                      % Plot Vin in time and frequency
f=fs*(0:length(VINw_n)/2-1)/length(VINw_n);
plot(f,VINw_n(1:floor(length(VINw)/2)),'Color','black');           % Plot Vin in frequency
xlabel('Frequency (Hertz)'); ylabel('|VIN| dB');
title('Plot of Vin in Frequency Domain');
axis([0 fs/2 -140 1]);
set(gca,'XTickMode','manual')
set(gca,'XTickLabel',{'0';'1';'2';'3';'4';'5';'6'})

% Plot VoutRaw
figure(2);
f=fs*(0:length(VOUT1w_n)/2-1)/length(VOUT1w_n);
plot(f,VOUT1w_n(1:floor(length(VOUT1w)/2)),'Color','black');         % Plot Vout in frequency
xlabel('Frequency (Hertz)'); ylabel('|VOUT| dB');
title('Plot of Vout Before Callibration in Frequency Domain');
axis([0 fs/2 -140 1]);
```

```
set(gca,'XTickMode','manual')
set(gca,'XTickLabel',{'0';'1';'2';'3';'4';'5';'6'});

% Plot VoutRaw
figure(3);
f=fs*(0:length(VOUT1bw_n)/2-1)/length(VOUT1bw_n);
plot(f,VOUT1bw_n(1:floor(length(VOUT1bw)/2)),'Color','black');         % Plot Vout in frequency
xlabel('Frequency (Hertz)'); ylabel('|VOUT| dB');
title('Plot of Vout After Gain and Offset Callibration in Frequency Domain');
axis([0 fs/2 -140 1]);
set(gca,'XTickMode','manual')
set(gca,'XTickLabel',{'0';'1';'2';'3';'4';'5';'6'});

% Plot VoutCor
figure(4);
f=fs*(0:length(VOUT2w_n)/2-1)/length(VOUT2w_n);
plot(f,VOUT2w_n(1:floor(length(VOUT2w)/2)),'Color','black');         % Plot Vout in frequency
xlabel('Frequency (Hertz)'); ylabel('|VOUT| dB');
title('Plot of Fully Corrected Vout in Frequency Domain');
axis([0 fs/2 -140 1]);
set(gca,'XTickMode','manual')
set(gca,'XTickLabel',{'0';'1';'2';'3';'4';'5';'6'});

%Subplot
% figure(5);
% subplot(1,4,1);
% f=fs*(0:length(VINw_n)/2-1)/length(VINw_n);
% plot(f,VINw_n(1:floor(length(VINw)/2)),'Color','black');           % Plot Vin in frequency
% xlabel('Frequency (Hertz)'); ylabel('|VIN| dB');
% title('Plot of Vin in Frequency Domain');
% axis([0 fs/2 -140 1]);
% set(gca,'XTickMode','manual')
% set(gca,'XTickLabel',{'0';'1';'2';'3';'4';'5';'6'});
%
% subplot(1,4,2);
% f=fs*(0:length(VOUT1w_n)/2-1)/length(VOUT1w_n);
% plot(f,VOUT1w_n(1:floor(length(VOUT1w)/2)),'Color','black');         % Plot Vout in frequency
% xlabel('Frequency (Hertz)'); ylabel('|VOUT| dB');
% title('Plot of Vout Before Callibration in Frequency Domain');
% axis([0 fs/2 -140 1]);
% set(gca,'XTickMode','manual')
% set(gca,'XTickLabel',{'0';'1';'2';'3';'4';'5';'6'});
%
% subplot(1,4,3);
% f=fs*(0:length(VOUT1bw_n)/2-1)/length(VOUT1bw_n);
% plot(f,VOUT1bw_n(1:floor(length(VOUT1bw)/2)),'Color','black');         % Plot Vout in frequency
% xlabel('Frequency (Hertz)'); ylabel('|VOUT| dB');
% title('Plot of Vout After Gain and Offset Callibration in Frequency Domain');
% axis([0 fs/2 -140 1]);
% set(gca,'XTickMode','manual')
% set(gca,'XTickLabel',{'0';'1';'2';'3';'4';'5';'6'});
%
% subplot(1,4,4);
% f=fs*(0:length(VOUT2w_n)/2-1)/length(VOUT2w_n);
% plot(f,VOUT2w_n(1:floor(length(VOUT2w)/2)),'Color','black');         % Plot Vout in frequency
% xlabel('Frequency (Hertz)'); ylabel('|VOUT| dB');
% title('Plot of Fully Corrected Vout in Frequency Domain');
```

```
% axis([0 fs/2 -140 1]);
% set(gca,'XTickMode','manual')
% set(gca,'XTickLabel',{'0';'1';'2';'3';'4';'5';'6'});
```

./source_code/splinta/Intl_Error_Examples01a.m

```
%*************************************************************************
% Interleaved Errors Examples Vers. 01
% 2006.11.06
%
% This program sets up a 4:1 interleaving example using Intl_Error_Setup01
% and generates the FFT plots using fft_1tone_an01.
%
%
% Modified to include all errors together CLD 2008.10.26
%
%*************************************************************************
close all;
clear all;

Intl_Error_Setup01a

% Setup input vector for FFT Analysis
Vin_fft=Vinq;
fft_1tone_an01;                 % Perform FFT Analysis
% Plot FFT of Qunatized Vin with Noise using Blackman Window
figure(1);                      % Plot Vin in time and frequency
f=fs*(0:length(VIN_n)/2-1)/length(VIN_n);
plot(f,VINw_n(1:floor(length(VIN_n)/2)));           % Plot Vin in frequency
xlabel('Frequency (Hertz)'); ylabel('|VIN| dB');
title('Plot of Vin in Frequency Domain');
axis([0 fs/2 -150 1]);

% Setup Vout Gain Error vector for FFT Analysis
Vin_fft=VoutGq;
fft_1tone_an01;                 % Perform FFT Analysis
% Plot FFT of Qunatized Vout with G Error and Noise using Blackman Window
figure(2);                      % Plot Vin in time and frequency
f=fs*(0:length(VIN_n)/2-1)/length(VIN_n);
plot(f,VINw_n(1:floor(length(VIN_n)/2)));           % Plot Vin in frequency
xlabel('Frequency (Hertz)'); ylabel('|VIN| dB');
title('Plot of Vout with Offset Errors in Frequency Domain');
axis([0 fs/2 -150 1]);

% Setup Vout Offset Error vector for FFT Analysis
Vin_fft=VoutOSq;
fft_1tone_an01;                 % Perform FFT Analysis
% Plot FFT of Qunatized Vout with OS Error and Noise using Blackman Window
figure(3);                      % Plot Vin in time and frequency
f=fs*(0:length(VIN_n)/2-1)/length(VIN_n);
plot(f,VINw_n(1:floor(length(VIN_n)/2)));           % Plot Vin in frequency
xlabel('Frequency (Hertz)'); ylabel('|VIN| dB');
title('Plot of Vout with Gain Errors in Frequency Domain');
axis([0 fs/2 -150 1]);

% Setup Vout Aperture Delay Error vector for FFT Analysis
Vin_fft=VoutAPTDq;
fft_1tone_an01;                 % Perform FFT Analysis
% Plot FFT of Qunatized Vout with APT Error and Noise using Blackman Window
figure(4);                      % Plot Vin in time and frequency
f=fs*(0:length(VIN_n)/2-1)/length(VIN_n);
```

```
plot(f,VINw_n(1:floor(length(VIN_n)/2)));           % Plot Vin in frequency
xlabel('Frequency (Hertz)'); ylabel('|VIN| dB');
title('Plot of Vout with Delay Errors in Frequency Domain');
axis([0 fs/2 -150 1]);

% Setup Vout All Errors vector for FFT Analysis
Vin_fft=VoutALLq;
fft_1tone_an01;                 % Perform FFT Analysis
% Plot FFT of Qunatized Vout with APT Error and Noise using Blackman Window
figure(5);                      % Plot Vin in time and frequency
f=fs*(0:length(VIN_n)/2-1)/length(VIN_n);
plot(f,VINw_n(1:floor(length(VIN_n)/2)));           % Plot Vin in frequency
xlabel('Frequency (Hertz)'); ylabel('|VIN| dB');
title('Plot of Vout with Offset, Gain and Aperture Delay Errors in Frequency Domain');
axis([0 fs/2 -140 1]);
```

./source_code/splinta/Intl_Error_Setup01a.m

```
%*************************************************************************
% Interleaved Errors Examples Setup Vers. 01
% 2006.11.06
%
% This program sets up a 4:1 interleaving example with individual Gain,
% Offset, and Aperture Delay.
%
% Uses function quantizerFcn01.m
%
%*************************************************************************

% Modified for two tones

M = 4;                      % Number of Interleaved ADCs
nsamples=500*1024+4;
fs=12E+6;                   % Sampling Frequency
fd1=1E+6;                  % Fundemental Frequency
fd2=2E+6;                  % Fundemental Frequency
Vin_amp=0.4;                % Vin Amplitude
t=(0:nsamples-1)./fs;       % sample time vector
lngVout = nsamples;

%Hardcoded  Errors
E_g1=0.012;                 % ADC_1 Gain Error
E_os1=0.011;                % ADC_1 Offset Error
t_apd1=0.02E-9;             % ADC_1 Aperture Delay
E_g2=-0.005;                 % ADC_2 Gain Error
E_os2=-0.003;               % ADC_2 Offset Error
t_apd2=-0.035E-9;           % ADC_2 Aperture Delay
E_g3=-0.01;                 % ADC_3 Gain Error
E_os3=-0.008;               % ADC_3 Offset Error
t_apd3=0.072E-9;            % ADC_3 Aperture Delay
E_g4=-0.007;                 % ADC_4 Gain Error
E_os4=0.005;                % ADC_4 Offset Error
t_apd4=-0.027E-9;           % ADC_4 Aperture Delay

EG=[E_g1; E_g2; E_g3; E_g4];
EOS=[E_os1; E_os2; E_os3; E_os4;];
EAPTD=[t_apd1;t_apd2;t_apd3;t_apd4;];
realErrors = [EG; EOS; EAPTD];


% Add Gaussian noise to the input signal
QNoise=16;
%Vnoise=zeros(1,lngVout);            % Zero Noise
%Vnoises=zeros(M,lngVout);           % Zero Noise
Vnoise=(1/2^(QNoise-1)).*randn(1,nsamples);     % 16-bit Gaussian Noise
Vnoises=(1/2^(QNoise-1)).*randn(M,nsamples);    % 16-bit Gaussian Noise

% Input Voltage Signal (assumes 1V reference)
%Vin=Vin_amp*sin((2*pi)*fd1*t)+Vnoise;
Vin=Vin_amp*sin((2*pi)*fd1*t)+Vin_amp*sin((2*pi)*fd2*t)+Vnoise;

%Initialize memory for matrices
```

```
%Vins=repmat(Vin_amp*sin((2*pi)*fd1*t),M,1)+Vnoises;
  Vins=repmat(Vin_amp*sin((2*pi)*fd1*t)+Vin_amp*sin((2*pi)*fd2*t),M,1)+Vnoises;
OUT_G=zeros(M,lngVout);
  OUT_OS=zeros(M,lngVout);
OUT_APTD=zeros(M,lngVout);

VoutG=zeros(1,lngVout);
  VoutOS=zeros(1,lngVout);
VoutAPTD=zeros(1,lngVout);

% Create MxNSamples Matrices with Individual G and OS Errors
  OUT_G = Vins.*repmat((1+EG),1,lngVout); % MxN Output Matrix w/ G Errors
OUT_OS = Vins+repmat(EOS,1,lngVout);     % MxN Output Matrix w/ OS Errors
  % Create MxNSamples Matrices with Individual APTD Errors
OUT_APTD(1,:)=(Vin_amp*sin((2*pi)*fd1*(t+t_apd1))+Vin_amp*sin((2*pi)*fd2*(t+t_apd1)))+Vnoises(1,:);
  OUT_APTD(2,:)=Vin_amp*sin((2*pi)*fd1*(t+t_apd2))+Vin_amp*sin((2*pi)*fd2*(t+t_apd2))+Vnoises(2,:);
OUT_APTD(3,:)=Vin_amp*sin((2*pi)*fd1*(t+t_apd3))+Vin_amp*sin((2*pi)*fd2*(t+t_apd3))+Vnoises(3,:);
  OUT_APTD(4,:)=Vin_amp*sin((2*pi)*fd1*(t+t_apd4))+Vin_amp*sin((2*pi)*fd2*(t+t_apd4))+Vnoises(4,:);

  OUT_ALL(1,:)=(Vin_amp*sin((2*pi)*fd1*(t+t_apd1))+Vin_amp*sin((2*pi)*fd2*(t+t_apd1)))*(1+EG(1))+EOS(1)+Vnoises(1,:);
OUT_ALL(2,:)=Vin_amp*sin((2*pi)*fd1*(t+t_apd2))+Vin_amp*sin((2*pi)*fd2*(t+t_apd2))*(1+EG(2))+EOS(2)+Vnoises(2,:);
  OUT_ALL(3,:)=Vin_amp*sin((2*pi)*fd1*(t+t_apd3))+Vin_amp*sin((2*pi)*fd2*(t+t_apd3))*(1+EG(3))+EOS(3)+Vnoises(3,:);
OUT_ALL(4,:)=Vin_amp*sin((2*pi)*fd1*(t+t_apd4))+Vin_amp*sin((2*pi)*fd2*(t+t_apd4))*(1+EG(4))+EOS(4)+Vnoises(4,:);

% Set up the interleaving order
  Intl_Order=repmat([1:4],1,floor(nsamples/4));
adjustment=0:4:floor(4*lngVout)-1;
  Intl_Order=Intl_Order+adjustment;

  % Get the final outputs
VoutG=OUT_G(Intl_Order);
  VoutOS=OUT_OS(Intl_Order);
VoutAPTD=OUT_APTD(Intl_Order);
  VoutALL=OUT_ALL(Intl_Order);

  % Quantize the final output
Vinq=quantizerFcn01(Vin);
  VoutGq=quantizerFcn01(VoutG);
VoutOSq=quantizerFcn01(VoutOS);
  VoutAPTDq=quantizerFcn01(VoutAPTD);
VoutALLq=quantizerFcn01(VoutALL);
```

./source_code/splinta/LPF_Test03.m

```
% Low Pass Filter Example
fs=12E6;
fT=1/fs;

for (j=1:M)
    RC = ((rand*.02)+0.99)*3.18E-9;
    a = fT / (RC + fT);
    for i=2:length(Vout(j,:))
        Vout(j,i) = a * Vout(j,i) + (1-a) * Vout(j,i-1);
    end
end
```

./source_code/splinta/multi_ADC_5ptderivEst_cor01b.m

```
%************************************************************************
% Iterative Correction Algorithm for the Multi Interleaved ADC Vers. 03
% 2006.11.06
%
% This program builds up the deltaX values for finding the gain, offset
% and apperture delay, errors in the Multi Intlerleaved, Split ADC
% architecture.  A coefficient matrix is also built for testing and
% debugging purposes.
%
% This program also computes the RMS Error between the Ideal and Corrected
% output.
%
% For use with the multi_ADC_setup03,4,5.
%
%************************************************************************

% increased my to 1/3 to try to reduce oscillations - JM, ADLEC2006

% Estimation Loop Parameters
% mx is the step size in the Gain and Offset error estimation
%mxrecip=32; % mu_e
mxrecip=2^4; % mu_e
mx=1/mxrecip;   %Step size of aproaching the Estimated Error

%myrecip=1024; %mu_G
myrecip=2^10; %mu_G
my=1/myrecip;
mtrecip=2^8;
mt=1/mtrecip;

Ncoef=128;       % Number of conversions used to build up matrices
jacobLeng=floor(nsamples/Ncoef);    %Number of main loops

% Initialize all Matrices to make room in Memory and save time
VoutA=zeros(1,(jacobLeng-1)*Ncoef+1);
VoutB=zeros(1,(jacobLeng-1)*Ncoef+1);
VoutBad=zeros(1,(jacobLeng-1)*Ncoef+1);
VoutCorA=zeros(1,(jacobLeng-1)*Ncoef+1);
VoutCorB=zeros(1,(jacobLeng-1)*Ncoef+1);
VoutCor=zeros(1,(jacobLeng-1)*Ncoef+1);

%EATPD=[t_apd1;t_apd2;t_apd3;t_apd4;t_apd5].*10E6;   % Real Aperture coefficients

Eg_est=zeros(M,1);         % Initialize all Error Estimates to zero
Eg_eps=zeros(M,1);
Eos_est=zeros(M,1);
Eos_eps=zeros(M,1);
Etpd_est=zeros(M,1);       % Initialize all Error Estimates to zero
Etpd_eps=zeros(M,1);       % Initialize Error in the Estimate to zero

jacobLeng=floor(nsamples/Ncoef);
RMS_Convergence=zeros(1,jacobLeng-1);
tempRMS_0=zeros(1,Ncoef-2);
tempRMS=zeros(1,Ncoef);
ADC_Convergence=zeros(1,jacobLeng-1);
```

```
tempADC_0=zeros(1,Ncoef-2);
tempADC=zeros(1,Ncoef);
j=1;
% Initialize the coeficients and bins matrices to zero
Eos_coef=zeros(Ncoef,M);    % Initialize Offset Error Coefficients matrix
Eg_coef=zeros(Ncoef,M);     % Initialize Gain Error Coefficients matrix
Etpd_coef=zeros(Ncoef,M);
Eos_bins=zeros(1,M);
Eg_bins=zeros(1,M);
Etpd_bins=zeros(1,M);
deltaX=zeros(Ncoef+3,1);

% Fill up the first two samples of the uncorrected vectors using Vout
VoutA(1:3)=[Vout(Apick(1),1) Vout(Apick(2),2), Vout(Apick(3),3)];
VoutB(1:3)=[Vout(Bpick(1),1) Vout(Bpick(2),2), Vout(Bpick(3),3)];
VoutBad(1:3)=(VoutA(1:3)+VoutB(1:3))/2;
deltaX(1:2)=VoutB(1:2)-VoutA(1:2);

for (i=3:Ncoef)
    k=(Ncoef*(j-1)+i);      % Generate the proper index for the Apick and
                            % Bpick matrices to keep track of ADC A and
                            % ADC B
    VoutA(k+1)=Vout(Apick(k+1),k+1)-Eos_est(Apick(k+1));    % Correct for G and OS
    VoutA(k+1)=VoutA(k+1)/(1+Eg_est(Apick(k+1)));
    VoutB(k+1)=Vout(Bpick(k+1),k+1)-Eos_est(Bpick(k+1));
    VoutB(k+1)=VoutB(k+1)/(1+Eg_est(Bpick(k+1)));

    VoutA(k+2)=Vout(Apick(k+2),k+2)-Eos_est(Apick(k+2));    % Correct for G and OS
    VoutA(k+2)=VoutA(k+2)/(1+Eg_est(Apick(k+2)));
    VoutB(k+2)=Vout(Bpick(k+2),k+2)-Eos_est(Bpick(k+2));
    VoutB(k+2)=VoutB(k+2)/(1+Eg_est(Bpick(k+2)));
    %VoutA(k+1)=Vout(Apick(k+1),k+1);
    %VoutB(k+1)=Vout(Bpick(k+1),k+1);
    VoutBad(k+1:k+2)=(VoutA(k+1:k+2)+VoutB(k+1:k+2))/2;
    %deltaConv=(VoutBad(k+1)-VoutBad(k-1))/2;    % Get Average Delta Conversion
    deltaConv=(VoutBad(k+1)-VoutBad(k-1))*(2/3)+...
        (VoutBad(k-2)-VoutBad(k+2))*(1/12);      % Get Average Delta Conversion
    VoutCorA(k)=VoutA(k)-Etpd_est(Apick(k))*deltaConv;
    VoutCorB(k)=VoutB(k)-Etpd_est(Bpick(k))*deltaConv;
    VoutCor(k)=(VoutCorA(k)+VoutCorB(k))/2;     % Get Average Corected Output
    deltaX(i)=(VoutCorB(k)-VoutCorA(k));        % Get difference between
                                                % corrected outputs

    Eos_coef(i,:)=pmmat(:,k)';
    Eg_coef(i,:)=VoutCor(k)*pmmat(:,k)';
    Etpd_coef(i,:)=deltaConv*pmmat(:,k)';       % Collect Coefficients
    Eos_bins=(sign(Eos_coef(i,:))*deltaX(i))+Eos_bins;
    Eg_bins=(sign(Eg_coef(i,:))*deltaX(i))+Eg_bins;
    Etpd_bins=(sign(Etpd_coef(i,:))*deltaX(i))+Etpd_bins;
    tempRMS_0(i)=Vin(k)-VoutCor(k);
    tempADC_0(i)=Vin(k)-VoutCor(k);
end

E_coef=[Eos_coef Eg_coef Etpd_coef];
E_coef=[E_coef; [ones(1,M),zeros(1,2*M)]; [zeros(1,M),ones(1,M),zeros(1,M)];...
    [zeros(1,2*M),ones(1,M)]];
```

```
% Calculate and track the Error in the Esitmate
Eos_eps=(1-mx)*Eos_eps+Eos_bins'*mx;
Eg_eps=(1-mx)*Eg_eps+Eg_bins'*mx;
Etpd_eps=(1-mx)*Etpd_eps+Etpd_bins'*mx;

Eos_eps_track(:,j)=Eos_eps;
Eg_eps_track(:,j)=Eg_eps;
Etpd_eps_track(:,j)=Etpd_eps;

% Calculate and track the Estimate
Eos_est=my.*Eos_eps+Eos_est;
Eg_est=my.*Eg_eps+Eg_est;
Etpd_est=mt.*Etpd_eps+Etpd_est;

Eos_est_track(:,j)=Eos_est;
Eg_est_track(:,j)=Eg_est;
Etpd_est_track(:,j)=Etpd_est;

RMS_Convergence(1)=sum(tempRMS_0.^2)/length(tempRMS_0);

jacobLeng=floor(nsamples/Ncoef);

for (j=2:jacobLeng-1)

    % Initialize the coeficients and bins matrices to zero
    Eos_coef=zeros(Ncoef,M);    % Initialize Offset Error Coefficients matrix
    Eg_coef=zeros(Ncoef,M);     % Initialize Gain Error Coefficients matrix
    Etpd_coef=zeros(Ncoef,M);
    Eos_bins=zeros(1,M);
    Eg_bins=zeros(1,M);
    Etpd_bins=zeros(1,M);
    deltaX=zeros(Ncoef+3,1);

    for (i=1:Ncoef)
        k=(Ncoef*(j-1)+i);      % Generate the proper index for the Apick and
                                % Bpick matrices to keep track of ADC A and
                                % ADC B
        VoutA(k+1)=Vout(Apick(k+1),k+1)-Eos_est(Apick(k+1));    % Correct for G and OS
        VoutA(k+1)=VoutA(k+1)/(1+Eg_est(Apick(k+1)));
        VoutB(k+1)=Vout(Bpick(k+1),k+1)-Eos_est(Bpick(k+1));
        VoutB(k+1)=VoutB(k+1)/(1+Eg_est(Bpick(k+1)));
        VoutA(k+2)=Vout(Apick(k+2),k+2)-Eos_est(Apick(k+2));    % Correct for G and OS
        VoutA(k+2)=VoutA(k+2)/(1+Eg_est(Apick(k+2)));
        VoutB(k+2)=Vout(Bpick(k+2),k+2)-Eos_est(Bpick(k+2));
        VoutB(k+2)=VoutB(k+2)/(1+Eg_est(Bpick(k+2)));
        %VoutA(k+1)=Vout(Apick(k+1),k+1);
        %VoutB(k+1)=Vout(Bpick(k+1),k+1);
        VoutBad(k+1:k+2)=(VoutA(k+1:k+2)+VoutB(k+1:k+2))/2;
        %deltaConv=(VoutBad(k+1)-VoutBad(k-1))/2;    % Get Average Delta Conversion
        deltaConv=(VoutBad(k+1)-VoutBad(k-1))*(2/3)+...
        (VoutBad(k-2)-VoutBad(k+2))*(1/12);      % Get Average Delta Conversion
        VoutCorA(k)=VoutA(k)-Etpd_est(Apick(k))*deltaConv;
        VoutCorB(k)=VoutB(k)-Etpd_est(Bpick(k))*deltaConv;
        VoutCor(k)=(VoutCorA(k)+VoutCorB(k))/2;     % Get Average Corected Output
        deltaX(i)=(VoutCorB(k)-VoutCorA(k));        % Get difference between
                                                    % corrected outputs

```

```
        Eos_coef(i,:)=pmmat(:,k)';
171     Eg_coef(i,:)=VoutCor(k)*pmmat(:,k)';
        Etpd_coef(i,:)=deltaConv*pmmat(:,k)';       % Collect Coefficients
173     Eos_bins=(sign(Eos_coef(i,:))*deltaX(i))+Eos_bins;
        Eg_bins=(sign(Eg_coef(i,:))*deltaX(i))+Eg_bins;
175     Etpd_bins=(sign(Etpd_coef(i,:))*deltaX(i))+Etpd_bins;

177     tempRMS(i)=Vin(k)-VoutCor(k); %Get difference for RMS Convergence
        tempADC(i)=Vin(k)-VoutCor(k); %Get difference for RMS Convergence
179 end

181 E_coef=[Eos_coef Eg_coef Etpd_coef];
    E_coef=[E_coef; [ones(1,M),zeros(1,2*M)]; [zeros(1,M),ones(1,M),zeros(1,M)];...
183     [zeros(1,2*M),ones(1,M)]];    % Coefficient matrix with averaging

185 % Calculate and track the Error in the Esitmate
    Eos_eps=(1-mx)*Eos_eps+Eos_bins'*mx;
187 Eg_eps=(1-mx)*Eg_eps+Eg_bins'*mx;
    Etpd_eps=(1-mx)*Etpd_eps+Etpd_bins'*mx;
189
    Eos_eps_track(:,j)=Eos_eps;
191 Eg_eps_track(:,j)=Eg_eps;
    Etpd_eps_track(:,j)=Etpd_eps;
193
    % Calculate and track the Estimate
195 Eos_est=my.*Eos_eps+Eos_est;
    Eg_est=my.*Eg_eps+Eg_est;
197 Etpd_est=mt.*Etpd_eps+Etpd_est;

199 Eos_est_track(:,j)=Eos_est;
    Eg_est_track(:,j)=Eg_est;
201 Etpd_est_track(:,j)=Etpd_est;

203 % Compute RMS Error
    RMS_Convergence(j)=sum(tempRMS.^2)/Ncoef;
205 end
```

./source_code/splinta/multi_ADC_setup06.m

```
%************************************************************************
% Multi Interleaved ADCs with Random Iteration Matrix Setup Vers. 05
% 06.04.04
% This program simulates nine non-ideal ADC (without quantization) in an
% interleaved setup.  This interleaved setup uses overlapping ADCs for
% digital error correction later on.  The program sets up 3*M error
% parameters; M gain errors, M offset errors, and M aperture delay errors.
%
% Gaussian Noise Added. No quantization (Handled by quantizer01.m)
%
%************************************************************************

% changed to set sum of tpd errors to zero - JM  ADLEC 2006

clear all;
close all;
intlRatio=4;                % Ratio of ADC Interleaving (x:1)
M=2*intlRatio+1;            % Number of ADCs required for interleaving
nsamples=500*1024+4;
fs=12E+6;                    % Sampling Frequency
%fd1=0.1*fs*.5;              % Fundemental Frequency 1
%fd2=0.4*fs*.5;              % Fundemental Frequency 2
fd1=1e6;                   % Fundemental Frequency 1
fd2=2e6;                   % Fundemental Frequency 2
Vin_amp=0.40;                % Vin Amplitude
t=(0:nsamples-1)./fs;       % sample time vector
lngVout = nsamples;
DC = 0.0;

%************************************************************************
% Code for randomly generating Errors
% The following was an attempt to randomly generate the errors removing all
% hardcoded values and giving the simulation a more realistic performance.
% Initial tests did not work becasue of scaling issues with the apperture
% delay.  The code should be fixed to make this work, but it has not been
% tested fully at this time.

%Perc_EG=0.05;    % Gain Error Percent (1=100%)
%Perc_EOS=0.1;    % Offset Error Percent (1=100%)
%Perc_Apt=0.001; % Aperture Error Percent (1=100%)

%EG=Perc_EG+2*Perc_EG*rand(1,M);     % Generate Random Gan Errors
%EG=EG'-mean(EG);                     % Set mean = 0
%EOS=Perc_EOS+2*Perc_EOS*rand(1,M);  % Generate Random Gan Errors
%EOS=EOS'-mean(EOS);                  % Set mean = 0
%EATPD=Perc_Apt*fs+2*fs*Perc_Apt*rand(1,M);% Generate Random Gan Errors
%EATPD=EATPD'-mean(EATPD);            % Set mean = 0
%************************************************************************

%Hardcoded  Errors
E_g1=0.045;                 % ADC_1 Gain Error
E_os1=0.035;                % ADC_1 Offset Error
t_apd1=0.05E-9;             % ADC_1 Aperture Delay
E_g2=-0.03;                 % ADC_2 Gain Error
E_os2=-0.094;               % ADC_2 Offset Error
```

```
t_apd2=-0.05E-9;           % ADC_2 Aperture Delay
E_g3=-0.01;                % ADC_3 Gain Error
E_os3=-0.008;              % ADC_3 Offset Error
t_apd3=0.02E-9;            % ADC_3 Aperture Delay
E_g4=-0.06;                % ADC_4 Gain Error
E_os4=0.024;               % ADC_4 Offset Error
t_apd4=-0.02E-9;           % ADC_4 Aperture Delay
E_g5=0.055;                % ADC_5 Gain Error
E_os5=0.043;               % ADC_5 Offset Error
t_apd5=-0.03E-9;           % ADC_5 Aperture Delay
E_g6=-0.038;               % ADC_6 Gain Error
E_os6=-0.013;              % ADC_6 Offset Error
t_apd6=-0.01E-9;           % ADC_6 Aperture Delay
E_g7=-0.027;               % ADC_7 Gain Error
E_os7=0.052;               % ADC_7 Offset Error
t_apd7=0.04E-9;            % ADC_7 Aperture Delay
E_g8=0.049;                % ADC_8 Gain Error
E_os8=-0.031;              % ADC_8 Offset Error
t_apd8=-0.01E-9;           % ADC_8 Aperture Delay
E_g9=0.016;                % ADC_9 Gain Error
E_os9=-0.008;              % ADC_9 Offset Error
t_apd9=0.01E-9;           % ADC_9 Aperture Delay

%realErrors = [EG; EOS; EATPD];
EG=[E_g1; E_g2; E_g3; E_g4; E_g5; E_g6; E_g7; E_g8; E_g9];
EOS=[E_os1; E_os2; E_os3; E_os4; E_os5; E_os6; E_os7; E_os8; E_os9];
EATPD=[t_apd1;t_apd2;t_apd3;t_apd4;t_apd5;t_apd6;t_apd7;t_apd8;t_apd9];
realErrors = [EG; EOS; EATPD];

% Add Gaussian noise to the input signal
QNoise=16;
% Modify to make SNR some number of dB
SNR=90;   % Published SNR of AD7621
QNoise=(SNR/(20*log10(2)))-0.5;
%Vnoise=zeros(1,lngVout);             % No Noise
%Vnoises=zeros(M,lngVout);            % No Noise
Vnoise=(1/2^(QNoise+1)).*randn(1,nsamples);     % 16-bit Gaussian Noise
Vnoises=(1/2^(QNoise+1)).*randn(M,nsamples);    % 16-bit Gaussian Noise
Vnoise1=(1/2^(QNoise+1)).*randn(1,nsamples);    % 16-bit Gaussian Noise
Vnoise2=(1/2^(QNoise+1)).*randn(1,nsamples);    % 16-bit Gaussian Noise
Vnoise3=(1/2^(QNoise+1)).*randn(1,nsamples);    % 16-bit Gaussian Noise
Vnoise4=(1/2^(QNoise+1)).*randn(1,nsamples);    % 16-bit Gaussian Noise
Vnoise5=(1/2^(QNoise+1)).*randn(1,nsamples);    % 16-bit Gaussian Noise
Vnoise6=(1/2^(QNoise+1)).*randn(1,nsamples);    % 16-bit Gaussian Noise
Vnoise7=(1/2^(QNoise+1)).*randn(1,nsamples);    % 16-bit Gaussian Noise
Vnoise8=(1/2^(QNoise+1)).*randn(1,nsamples);    % 16-bit Gaussian Noise
Vnoise9=(1/2^(QNoise+1)).*randn(1,nsamples);    % 16-bit Gaussian Noise

%Initialize memory for matrices
Vins=zeros(M,lngVout);
OUT=zeros(M,lngVout);
Vout=zeros(M,lngVout);

lng_Vin=length(Vins);        % Length of Input Voltage
```

```
% Input Voltage (assumes 1V reference)
Vin=DC+Vin_amp*sin((2*pi)*fd1*t)+Vin_amp*sin((2*pi)*fd2*t)+Vnoise;

% PRN Input Signal Setup
% Vin = zeros(1,lng_Vin);
% PRN_Phase = zeros(1,90);
% PRN_Freq = zeros(1,90);
% for (i=1:90)
%     PRN_Phase(i) = 4*rand(1)-2;
%     PRN_Freq(i) = 1E+6*rand(1)+1e+6;
%     Vin=0.01*sin(((2*pi*PRN_Freq(i)*t)+(PRN_Phase(i)*pi)))+Vin;
% end
% Vin = Vin + Vnoise;
%
% % ADC Input with Aperture Delay
% for (i=1:M)
%     for (j=1:90)
%         Vins(i,:)=0.01*sin(((2*pi*PRN_Freq(j)*t)+EATPD(i)...
%             +(PRN_Phase(j)*pi)))+Vins(i,:);
%     end
%     Vins(i,:) = Vins(i,:) + Vnoises(i,:);
% end

% Setup Intlereaved Pattern Matrix
% Initialize Matrix sizes in memory to increase execution speed
pmmat=zeros(M,lngVout);
intlvdSetup=zeros(M,lngVout);
pathTrack=zeros(1,lngVout);
Apick=zeros(1,lngVout);
Bpick=zeros(1,lngVout);
%ADCTimer=[0 1 1 2 2 3 3 4 4];   %Setup Counter to keep track of ADCs
ADCTimer=zeros(1,M);   %Setup Counter to keep track of ADCs
for (k=1:(M-1)/2)
    ADCTimer(2*k:2*k+1)=k;
end
TimerDecr=ones(1,M);            %Initialize Timer Decrement Vector

%comboMat=triu(ones(M),1);

% Random Path Generation is done by choosing 2 out of the 3 available ADCs
% for the next conversion cycle.  ADCTimer keeps track of the available
% ADCs and the number of conversions before they become available again.
% comboMat is a Matrix that keeps track of which ADCs have already been
% chosen.  This allows for keeping the number of repeated choices to a
% minimum and generating full paths with every possible combination in
% them.
for (i=1:lngVout)
    choices=find(ADCTimer<2);       %Get Possible Choices
    newChoice=ceil(3*rand(1));      %Generate Random Number
    switch newChoice                %Choose 2 out of the 3 possible ADCs
        case {3}                    %based on the Random Number Generated.
            ADCA=1;
            ADCB=2;
        case {2}
            ADCA=1;
            ADCB=3;
        case {0 , 1}
```

```
            ADCA=2;
171             ADCB=3;
        otherwise
173             disp('Failed');
                break;
175 end
    ADCA=choices(ADCA);
177 ADCB=choices(ADCB);

179% The following code was used to try and minimize repeated pari
   % combinations, however, it was eventually shown that this was not
181% necessary.  This code was removed to try and simplify the PRN technique
   % for the eventual FPGA implementation.
183%  if (comboMat(ADCA,ADCB)==1)
   %      comboMat(ADCA,ADCB)=0;
185%      %i=i+1;
   %  elseif (comboMat(ADCA,ADCB)==0)
187%      ADCA=1;
   %      ADCB=2;
189%      ADCA=choices(ADCA);
   %      ADCB=choices(ADCB);
191%      if (comboMat(ADCA,ADCB)==1)
   %          comboMat(ADCA,ADCB)=0;
193%          %i=i+1;
   %      else
195%          ADCA=1;
   %          ADCB=3;
197%          ADCA=choices(ADCA);
   %          ADCB=choices(ADCB);
199%          if (comboMat(ADCA,ADCB)==1)
   %              comboMat(ADCA,ADCB)=0;
201%              %i=i+1;
   %          else
203%              ADCA=2;
   %              ADCB=3;
205%              ADCA=choices(ADCA);
   %              ADCB=choices(ADCB);
207%              if (comboMat(ADCA,ADCB)==1)
   %                  comboMat(ADCA,ADCB)=0;
209%                  %i=i+1;
   %              else
211%                  newChoice=ceil(3*rand(1));      %Generate Random Number
   %                  switch newChoice
213%                      case {3}
   %                          ADCA=1;
215%                          ADCB=2;
   %                      case {2}
217%                          ADCA=1;
   %                          ADCB=3;
219%                      case {0 , 1}
   %                          ADCA=2;
221%                          ADCB=3;
   %                      otherwise
223%                          disp('Failed');
   %                          break;
225%                  end
   %                  ADCA=choices(ADCA);
```

```
%                   ADCB=choices(ADCB);
%               end
%           end
%       end
%   end

%   if (isempty(find(comboMat))==1)
%       comboMat=triu(ones(M),1);
%   end

    ADCTimer=ADCTimer-TimerDecr;    %Decrement the timer
    ADCTimer(ADCA)=(M-1)/2;              %Set 1st ADC in ADCTimer to 4
    ADCTimer(ADCB)=(M-1)/2;              %Set 2nd ADC in ADCTimer to 4
    Apick(i)=ADCA;
    Bpick(i)=ADCB;
    intlvdSetup(((i-1)*M)+Apick(i))=1;          %Setup 'A' ADC pattern
    pmmat(((i-1)*M)+Apick(i))=-1;               %Setup P/M 'A' ADC pattern for -1
    intlvdSetup(((i-1)*M)+Bpick(i))=1;          %Setup 'B' ADC pattern
    pmmat(((i-1)*M)+Bpick(i))=1;                %Setup P/M 'B' ADC pattern for +1

    for (j=1:M)
        % ADC Input with Aperture Delay
        Vins(j,i)=DC+Vin_amp*sin((2*pi)*fd1*(t(i)+EATPD(j)))...
        +Vin_amp*sin((2*pi)*fd2*(t(i)+EATPD(j)))+Vnoises(j,i);
    end

    % Calculate the latest input samples with Gain and Offset Errors
    OUT(:,i) = Vins(:,i).*(1+EG)+EOS;    % MxN Output Matrix

    % Generate the next output sample
    Vout(:,i)=OUT(:,i).*intlvdSetup(:,i);
    %*********************************************************************

end
```

./source_code/splinta/NonLinQuantizer03.m

```
%************************************************************************
% 5 Interleaved ADCs Quantizer Vers. 01
% 2009.01.22
%
% This program quantizes the Vout vector with a nonlinear performance.
% This applies a PRN INL to each ADC.
%
% For use with the multi_ADC_setup06
%
%************************************************************************

numBits = 24;           %Number of Bits for quantization
LSB = 2^-19;
LSB2 = 2^-10.3;
Vout_old=Vout;          %Store the old, unquantized values
qVout=Vout*2^numBits-1; %Start quantizing the values
for (i=1:M)
    qVout(i,:)=qVout(i,:).*(1+(((rand/2)+0.5)*LSB)*sin(qVout(i,:)/(2^numBits)*2*pi));
end
qVout=round(qVout);
qVout=min(qVout,2^numBits-1);
qVout=max(qVout,-2^numBits-1);
qVout=qVout/(2^numBits-1);
Vout=qVout;             %Store the new, quantized values for correction

Vin_old = Vin;
Vin = Vin.*(1+LSB2*sin(Vin/(2^10.3)*2*pi));
```

./source_code/splinta/quantizerFcn01.m

```
function Vout=quantizerFcn01(Vin)
%************************************************************************
% Quantizer Function with One Input
% 06.11.06
%
% This a function version of the older Quantizer01 program.  This function
% reads in an input vecotr, quantizes it to the number of bits hardcoded in
% this function, then returns a quantized output.  Future versions of this
% function may include an argument for setting the quantization level.
%
% Inputs:
%   Vin = The input vector to be quantized
%
% Outputs:
%   Vout = The output vector containing quantized values of the input
%       vecotr Vin.
%
% Parameter:
%   numBits = The number of bits that determines the quantization levels
%       for the output vector.  Future versions may set this paramter up to
%       be an input argument to this function.
%
%************************************************************************

numBits = 16;           %Number of Bits for quantization


qVout=Vin*2^numBits-1; %Start quantizing the values
qVout=round(qVout);
qVout=min(qVout,2^numBits-1);
qVout=max(qVout,-2^numBits-1);
qVout=qVout/(2^numBits-1);
Vout=qVout;             %Store the new, quantized values for correction
```

./source_code/splinta/SNR_V02.m

```
numpt = length(VIN_n);
fin=find(VIN_n(1:numpt/2)>-1);
spectP=(abs(VIN)).*(abs(VIN));
span = 3;
Pdc=sum(spectP(1));
Ps=sum(spectP([fin(1)-span:span+fin(1), fin(2)-span:span+fin(2)]));
Pn=sum(spectP(1:numpt/2))-Pdc-Ps;
SNR_IN=10*log10(Ps/Pn)

numpt = length(VOUT1w_n);
fin=find(VOUT1w_n(1:numpt/2)>-1);
spectP=(abs(VOUT1w)).*(abs(VOUT1w));
span = 3;
Pdc=sum(spectP(1));
Ps=sum(spectP([fin(1)-span:span+fin(1), fin(2)-span:span+fin(2)]));
Pn=sum(spectP(1:numpt/2))-Pdc-Ps;
SNR_UNCOR=10*log10(Ps/Pn)

numpt = length(VOUT2w_n);
fin=find(VOUT2w_n(1:numpt/2)>-1);
spectP=(abs(VOUT2w)).*(abs(VOUT2w));
span = 3;
Pdc=sum(spectP(1));
Ps=sum(spectP([fin(1)-span:span+fin(1), fin(2)-span:span+fin(2)]));
Pn=sum(spectP(1:numpt/2))-Pdc-Ps;
SNR_COR=10*log10(Ps/Pn)
```

./source_code/splinta/TestNonLin02.m

```
multi_ADC_setup06;
LPF_Test03;
NonLinQuantizer03;
multi_ADC_5ptderivEst_cor01b;
fft_win_intl_an02;
```

Appendix B

Split-SAR MATLAB Code

./source_code/split_sar/annotate_fft.m

```
% Annotate Plots
 % Basic Info
figure(21);
 text( 500, -5, ['Vref = ' num2str(Vref) ' V']);
text( 500, -10, ['Vin = ' num2str(Vfrac) ' Vref']);
 text( 500, -15, ['Vin = ' num2str(Vfrac*Vref) ' V']);
text( 500, -20, ['Vin = ' num2str(20*log10(Vfrac)) ' dB']);
 text( 500, -25, ['Vin Freq = ' num2str(fin_A) ' Hz']);

 text( f(fh2i), -40, 'Second Harmonic', 'HorizontalAlignment', 'center');
text( f(fh2i), -45, ['Freq = ' num2str(f(h2A_i)) ' Hz'], 'HorizontalAlignment', 'center');
 text( f(fh2i), -50, [num2str(h2A) ' dB'], 'HorizontalAlignment', 'center');

 text( f(fh3i), -40, 'Third Harmonic', 'HorizontalAlignment', 'center');
text( f(fh3i), -45, ['Freq = ' num2str(f(h3A_i)) ' Hz'], 'HorizontalAlignment', 'center');
 text( f(fh3i), -50, [num2str(h3A) ' dB'], 'HorizontalAlignment', 'center');

 text( f(fh4i), -40, 'Fourth Harmonic', 'HorizontalAlignment', 'center');
text( f(fh4i), -45, ['Freq = ' num2str(f(h4A_i)) ' Hz'], 'HorizontalAlignment', 'center');
 text( f(fh4i), -50, [num2str(h4A) ' dB'], 'HorizontalAlignment', 'center');

 text( f(fh5i), -40, 'Fifth Harmonic', 'HorizontalAlignment', 'center');
text( f(fh5i), -45, ['Freq = ' num2str(f(h5A_i)) ' Hz'], 'HorizontalAlignment', 'center');
 text( f(fh5i), -50, [num2str(h5A) ' dB'], 'HorizontalAlignment', 'center');

 text( f(fh7i), -55, 'Seventh Harmonic', 'HorizontalAlignment', 'center');
text( f(fh7i), -60, ['Freq = ' num2str(f(h7A_i)) ' Hz'], 'HorizontalAlignment', 'center');
 text( f(fh7i), -65, [num2str(h7A) ' dB'], 'HorizontalAlignment', 'center');

 % Figure 2 for SAR B
figure(22);
 text( 500, -5, ['Vref = ' num2str(Vref) ' V']);
text( 500, -10, ['Vin = ' num2str(Vfrac) ' Vref']);
 text( 500, -15, ['Vin = ' num2str(Vfrac*Vref) ' V']);
text( 500, -20, ['Vin = ' num2str(20*log10(Vfrac)) ' dB']);
```

```
text( 500, -25, ['Vin Freq = ' num2str(fin_A) ' Hz']);
37
text( f(fh2i), -40, 'Second Harmonic', 'HorizontalAlignment', 'center');
39text( f(fh2i), -45, ['Freq = ' num2str(f(h2B_i)) ' Hz'], 'HorizontalAlignment', 'center');
text( f(fh2i), -50, [num2str(h2B) ' dB'], 'HorizontalAlignment', 'center');
41
text( f(fh3i), -40, 'Third Harmonic', 'HorizontalAlignment', 'center');
43text( f(fh3i), -45, ['Freq = ' num2str(f(h3B_i)) ' Hz'], 'HorizontalAlignment', 'center');
text( f(fh3i), -50, [num2str(h3B) ' dB'], 'HorizontalAlignment', 'center');
45
text( f(fh4i), -40, 'Fourth Harmonic', 'HorizontalAlignment', 'center');
47text( f(fh4i), -45, ['Freq = ' num2str(f(h4B_i)) ' Hz'], 'HorizontalAlignment', 'center');
text( f(fh4i), -50, [num2str(h4B) ' dB'], 'HorizontalAlignment', 'center');
49
text( f(fh5i), -40, 'Fifth Harmonic', 'HorizontalAlignment', 'center');
51text( f(fh5i), -45, ['Freq = ' num2str(f(h5B_i)) ' Hz'], 'HorizontalAlignment', 'center');
text( f(fh5i), -50, [num2str(h5B) ' dB'], 'HorizontalAlignment', 'center');
53
text( f(fh7i), -55, 'Seventh Harmonic', 'HorizontalAlignment', 'center');
55text( f(fh7i), -60, ['Freq = ' num2str(f(h7B_i)) ' Hz'], 'HorizontalAlignment', 'center');
text( f(fh7i), -65, [num2str(h7B) ' dB'], 'HorizontalAlignment', 'center');
```

./source_code/split_sar/DC_linearity_test_V02.m

```
% DC_linearity_test_V02
% DC Lineartiy Test
% Imports pre-estimated weights, applies them to the pre-loaded DC points
% and finds the difference between the corrected and measured DC values.
% This version leaves out some of the values in each DC level to account
% for filtering out "bad" decisions
%

import_measured_DC_file('../../DC_levels_V2/Chip1_Measured_DC_Levels.csv');
%load '../Estimated_Weights_sine_10.04.08.01.mat';
%load '../Estimated_Weights_DC_10.04.08.01.mat';
x_A_hat = D_A_bits_filtered * W_A_hat_final;
x_B_hat = D_B_bits_filtered * W_B_hat_final;

conv_DC_levels_A = zeros(1,101);
conv_DC_levels_B = zeros(1,101);

LSB = 3.6/2^16;

for i=1:101;
    conv_DC_levels_A(i) = mean(x_A_hat((i-1)*1310+1:i*1310-256));
    conv_DC_levels_B(i) = mean(x_B_hat((i-1)*1310+1:i*1310-256));
end

INL_A = conv_DC_levels_A-Measured_SAR_A';
INL_A = INL_A - mean(INL_A);
slope_A = linspace(INL_A(end),INL_A(1),101);
INL_A = INL_A + slope_A;
INL_A = INL_A/LSB;

INL_B = conv_DC_levels_B-Measured_SAR_B';
INL_B = INL_B - mean(INL_B);
slope_B = linspace(INL_B(end),INL_B(1),101);
INL_B = INL_B + slope_B;
INL_B = INL_B/LSB;

conv_DC_levels_avg = (conv_DC_levels_A+conv_DC_levels_B)/2;
Measures_SAR_avg = (Measured_SAR_B' + Measured_SAR_A')/2;
INL_avg = conv_DC_levels_avg - Measures_SAR_avg;
INL_avg = INL_avg - mean(INL_avg);
slope_avg = linspace(INL_avg(end),INL_avg(1),101);
INL_avg = INL_avg + slope_avg;
INL_avg = INL_avg/LSB;

x_ax = linspace(-1,1,101);

figure(1);
plot(x_ax,INL_A);
title('INL of SAR A');
ylabel('LSBs');
xlabel('V_{REF}');
%axis([-1 1 -25 25]);

figure(2);
plot(x_ax,INL_B);
```

```
title('INL of SAR B');
ylabel('LSBs');
xlabel('V_{REF}');
%axis([-1 1 -25 50]);

figure(3);
plot(x_ax,INL_avg);
title('INL of Averaged SAR Output');
ylabel('LSBs');
xlabel('V_{REF}');
%axis([-1 1 -25 25]);

% Fit third order polynomial and remove from averaged INL
p_inl = polyfit(x_ax,INL_avg,3);
f_inl = polyval(p_inl,x_ax);
INL_avg_no_3rd = INL_avg - f_inl;

figure(4);
%plot(x_ax,INL_avg); hold on;
%plot(x_ax,f_inl); hold off;
plot(x_ax,INL_avg_no_3rd);
title('INL of Averaged SAR Output with Third Order Removed');
ylabel('LSBs');
xlabel('V_{REF}');
```

./source_code/split_sar/dual_ADC_data_import_no_limit_No_DOutP_B.m

```
%Import data:  Final data will be in actual_data, manipulate from there
%  Auto-generated by MATLAB on 09-Sep-2009 11:28:02

% Import the file
newData1 = importdata(filename);

% Break the data up into a new structure with one field per column.
colheaders = genvarname(newData1.colheaders);
for i = 1:length(colheaders)
    dataByColumn1.(colheaders{i}) = newData1.data(:, i);
end

% Create new variables in the base workspace from those fields.
vars = fieldnames(dataByColumn1);
for i = 1:length(vars)
    assignin('base', vars{i}, int8(dataByColumn1.(vars{i})));
end

clear vars; clear newData1; clear i; clear dataByColumn1; clear colheaders;

actual_data_A = DataOutP_A - DataOutN_A;
num_decisions = floor(length(actual_data_A)/20)*20;
actual_data_A = actual_data_A(1:num_decisions);
base_A = BaseOut_A(1:num_decisions);

DataOutP_B = int8(not(DataOutN_B));
actual_data_B = DataOutP_B - DataOutN_B;
actual_data_B = actual_data_B(1:num_decisions);
base_B = BaseOut_B(1:num_decisions);
```

./source_code/split_sar/fft_sine_V02.m

```
% Creating and plotting FFT
 % Generate Window Function for x_A
window = blackman(length(sinewave_A));

% Apply windowing for SAR A Output and normalize
 blackman_sine_A_fft = abs(fft(sinewave_A.*window));
blackman_sine_A_fft_dB = 20*log10(blackman_sine_A_fft);
 FFT_A = blackman_sine_A_fft_dB - max(blackman_sine_A_fft_dB);
FFT_A_2 = FFT_A(1:floor(length(FFT_A)/2));

% Plot and label FFT for SAR A
 f = linspace(0, fs/2, length(FFT_A_2));
figure(21);
 plot(f,FFT_A_2);
xlabel('Frequency Hz');
 ylabel('|FFT| dB');
title(['FFT of SAR A Output Normalized to Vin = ' num2str(Vfrac) ' Vref']);
 ylim([-160 0]);

 % Generate Window Function for x_B
window = blackman(length(sinewave_B));

% Apply windowing for SAR B Output and normalize
 blackman_sine_B_fft = abs(fft(sinewave_B.*window));
blackman_sine_B_fft_dB = 20*log10(blackman_sine_B_fft);
 FFT_B = blackman_sine_B_fft_dB - max(blackman_sine_B_fft_dB);
FFT_B_2 = FFT_B(1:floor(length(FFT_B)/2));

% Plot and label FFT for SAR B
 f = linspace(0, fs/2, length(FFT_B_2));
figure(22);
 plot(f,FFT_B_2);
xlabel('Frequency Hz');
 ylabel('|FFT| dB');
title(['FFT of SAR B Output Normalized to Vin = ' num2str(Vfrac) ' Vref']);
 ylim([-160 0]);

 % Generate Window Function for x_avg
window = blackman(length(sinewave_avg));

% Apply windowing for SAR avg Output and normalize
 blackman_sine_avg_fft = abs(fft(sinewave_avg.*window));
blackman_sine_avg_fft_dB = 20*log10(blackman_sine_avg_fft);
 FFT_avg = blackman_sine_avg_fft_dB - max(blackman_sine_avg_fft_dB);
FFT_avg_2 = FFT_avg(1:floor(length(FFT_B)/2));

% Plot and label FFT for SAR B
 f = linspace(0, fs/2, length(FFT_avg_2));
figure(23);
 plot(f,FFT_avg_2);
xlabel('Frequency Hz');
 ylabel('|FFT| dB');
title(['FFT of Averaged SAR Output Normalized to Vin = ' num2str(Vfrac) ' Vref']);
 ylim([-160 0]);
```

./source_code/split_sar/filter_bad_decs.m

```
% Filter out "Bad Decisions" containing 15 in the last bank

% Find "bad" decisions
bad_A = find(D_A_bits(:,50) == 15);
bad_B = find(D_B_bits(:,50) == 15);

% Combine the two, sort the indeces (not necessary) and remove duplicates
bad_decs = [bad_A; bad_B];
bad_decs = sort(bad_decs);
bad_decs = unique(bad_decs);

% Delete the bad entries
D_A_bits_filtered = D_A_bits;
D_A_bits_filtered(bad_decs,:) = [];
D_B_bits_filtered = D_A_bits;
D_B_bits_filtered(bad_decs,:) = [];
```

./source_code/split_sar/harmonic_locations.m

```
% Find local harmonics based on known sampling frequency and sinewave input
% frequency

% Get the single sinewave input frequency
%fin_A = interp1(FFT_A,f,0,'nearest');
fin_A = f(find(FFT_A == max(FFT_A)));

% Get Second Harmonic
[fh2, fh2i] = min(abs(f-(2*fin_A)));
h2A = max(FFT_A(fh2i-50:fh2i+50));
h2A_i = find(FFT_A==h2A);
h2B = max(FFT_B(fh2i-50:fh2i+50));
h2B_i = find(FFT_B==h2B);

% Get Third Harmonic
[fh3, fh3i] = min(abs(f-(fs-3*fin_A)));
h3A = max(FFT_A(fh3i-50:fh3i+50));
h3A_i = find(FFT_A==h3A);
h3B = max(FFT_B(fh3i-50:fh3i+50));
h3B_i = find(FFT_B==h3B);

% Get Fourth Harmonic
[fh4, fh4i] = min(abs(f-(fs-4*fin_A)));
h4A = max(FFT_A(fh4i-500:fh4i+500));
h4A_i = find(FFT_A==h4A);
h4B = max(FFT_B(fh4i-500:fh4i+500));
h4B_i = find(FFT_B==h4B);

% Get Fifth Harmonic
[fh5, fh5i] = min(abs(f-(fs-5*fin_A)));
h5A = max(FFT_A(fh5i-500:fh5i+500));
h5A_i = find(FFT_A==h5A);
h5B = max(FFT_B(fh5i-500:fh5i+500));
h5B_i = find(FFT_B==h5B);

% Get Seventh Harmonic
[fh7, fh7i] = min(abs(f-(1.5*fs-7*fin_A)));
h7A = max(FFT_A(fh7i-500:fh7i+500));
h7A_i = find(FFT_A==h7A);
h7B = max(FFT_B(fh7i-500:fh7i+500));
h7B_i = find(FFT_B==h7B);
```

./source_code/split_sar/import_measured_DC_file.m

```
function import_measured_DC_file(fileToRead1)
%IMPORTFILE(FILETOREAD1)
%  Imports data from the specified file
%  FILETOREAD1:  file to read

%  Auto-generated by MATLAB on 08-Apr-2010 15:25:32

% Import the file
newData1 = importdata(fileToRead1);

% Break the data up into a new structure with one field per column.
colheaders = genvarname(newData1.colheaders);
for i = 1:length(colheaders)
    dataByColumn1.(colheaders{i}) = newData1.data(:, i);
end

% Create new variables in the base workspace from those fields.
vars = fieldnames(dataByColumn1);
for i = 1:length(vars)
    assignin('base', vars{i}, dataByColumn1.(vars{i}));
end
```

./source_code/split_sar/reshape_dual_data_shuffled_banks1_3_V03.m

```
% Reshape the actual data into the 20bit by N conversion array
D_A_bits = reshape_shuffled_bank1_3_data_fcn(actual_data_A, base_A);
D_B_bits = reshape_shuffled_bank1_3_data_fcn(actual_data_B, base_B);


% Get the data output using the starting weights
x_A = D_A_bits * W_A - OS_init/2;
x_B = D_B_bits * W_B + OS_init/2;
```

./source_code/split_sar/reshape_shuffled_bank1_3_data_fcn.m

```
function [segmented_data] = reshape_shuffled_bank1_3_data_fcn(data_in, base);
%************************************************************************
% Function: reshape_shuffled_bank1_3_data_fcn
% [segmented_data] = reshape_shuffled_bank1_3_data_fcn(data_in, base)
% This function takes in two vectors from the Split SAR, a decisions vector
% and a base vector, both stored as 8-bit integers consisting of -1, 0 or 1
% This function then reshapes the data vector and uses the base codes to
% construct a segmented_data matrix (also 8-bit signed integers) that
% breaks down the top three bank decisions into individual segments.
% The result is a Nx50 matrix, where N is the number of samples.  The
% first 48 colums are the segmented decisions from banks 1 through 3.
% The last two are the summed decisions of banks 4 and 5.  Banks 4 and 5
% are weighted as powers of 2.
%
% Inputs:
%   data_in = An 8-bit signed integer vector of -1s and 1s of each bit
%       decision from the SAR.
%
%   base = An 8-bit signed integer vector of 0s and 1s representing the
%       base code used for each bank.
%
% Outputs:
%   segmented_data = a Nx50 8-bit signed integer matrix of the segmented
%       decisions from the SAR
%
%************************************************************************

%reshape the actual data into the 20bit by number of samples conversion array
num_samples = length(data_in)/20;
D = reshape(double(data_in), 20, num_samples)';

% Divide up individual bank decisions
D_bank1_bits = D(:,1:4);
D_bank2_bits = D(:,5:8);
D_bank3_bits = D(:,9:12);
D_bank4_bits = D(:,13:16);
D_bank5_bits = D(:,17:20);

% Weight the banks 4 and 5 decisions by powers of 2 (8 4 2 1)
D_bank4_decs = D_bank4_bits .* repmat([8 4 2 1], num_samples, 1);
D_bank5_decs = D_bank5_bits .* repmat([8 4 2 1], num_samples, 1);

% Sum banks 4 and 5 powers of 2 decisions
D_bank4_sum = sum(D_bank4_decs,2);
D_bank5_sum = sum(D_bank5_decs,2);

base_bits = reshape(base, 20, num_samples)';
% Get the Base Bits for Bank 1
bank1_bits = base_bits(:,1:4);
% Get the Base Bits for Bank 2
bank2_bits = base_bits(:,5:8);
% Get the Base Bits for Bank 3
bank3_bits = base_bits(:,9:12);
% Set up Base Code Weights
base_code_weights = [1; 2; 4; 8];
```

```
% Calculate the value of each base code for bank 1
bank1_codes = double(bank1_bits) * base_code_weights;
% Calculate the value of each base code for bank 2
bank2_codes = double(bank2_bits) * base_code_weights;
% Calculate the value of each base code for bank 3
bank3_codes = double(bank3_bits) * base_code_weights;

% Initialize the segment matrix with the expanded bank 1 segments
segmented_data = zeros(num_samples,50);

% Convention: Base Number indiactes the position of the unused unit cap
% For example: Base Code = 0 gives the following segment
% 0 d1 d1 d1 d1 d1 d1 d1 d1 d2 d2 d2 d2 d3 d3 d4
% Base Code = 6 gives the following segment
% d2 d2 d2 d3 d3 d4 0 d1 d1 d1 d1 d1 d1 d1 d1 d2

% Set up the full segment matrix using the bank1 base codes
for i = 1:num_samples;
%for i = 1:1;
    % Start with expanded segment with the unused cap in position 0
    bank1_segment = [0 repmat(D_bank1_bits(i,1),1,8),...
        repmat(D_bank1_bits(i,2),1,4),...
        repmat(D_bank1_bits(i,3),1,2) D_bank1_bits(i,4)];

    bank2_segment = [0 repmat(D_bank2_bits(i,1),1,8),...
        repmat(D_bank2_bits(i,2),1,4),...
        repmat(D_bank2_bits(i,3),1,2) D_bank2_bits(i,4)];

    bank3_segment = [0 repmat(D_bank3_bits(i,1),1,8),...
        repmat(D_bank3_bits(i,2),1,4),...
        repmat(D_bank3_bits(i,3),1,2) D_bank3_bits(i,4)];

    % Then shift the bank segments based on the bank base code number
    bank1_segment = circshift(bank1_segment, [0 bank1_codes(i)]);
    bank2_segment = circshift(bank2_segment, [0 bank2_codes(i)]);
    bank3_segment = circshift(bank3_segment, [0 bank3_codes(i)]);

    segmented_data(i,:) = [bank1_segment bank2_segment bank3_segment,...
        D_bank4_sum(i) D_bank5_sum(i)];
end
```

./source_code/split_sar/run_dual_data_without_DOutP_B_v01.m

```
clear all;
close all;

%filename = './Modified_ref_buffers/Tri2Hz_base0_SAR_A_DOutP_B_Off_100k.09.10.15.01.txt'
filename = './Modified_ref_buffers/Tri2Hz_base0_SAR_A_only_100k.09.10.15.01.txt'

single_ADC_data_import_v02;

actual_data = actual_data_A;
figure(1);
reshape_single_data_fixed_LSBs;
title('Triangle Wave with Ideal Weights for SAR A');
calculate_weights_v02;
figure(2);
INL_calculation;
title('INL with Fixed Base for SAR A');

% DataOutP_B = int8(not(DataOutN_B));
% actual_data_B = DataOutP_B - DataOutN_B;
% actual_data_B = actual_data_B(1:2097140);
% actual_data = actual_data_B;
% figure(3);
% reshape_single_data_fixed_LSBs;
% title('Triangle Wave with Ideal Weights for SAR B');
% calculate_weights_v02;
% figure(4);
% INL_calculation;
% title('INL with Fixed Base for SAR B');
```

./source_code/split_sar/run_sine_test_v02.m

```
% Split SAR Sinewave Test


sinewave_A = D_A_bits * W_A;
sinewave_B = D_B_bits * W_B;

% sinewave_A = D_A_bits * W_A_hat_final;
% sinewave_B = D_B_bits * W_B_hat_final;

sinewave_avg = (sinewave_A+sinewave_B)/2;

fft_sine_V02;

SNDR_V02;

% harmonic_locations;
% annotate_fft;
```

./source_code/split_sar/run_split_SAR_test_v02.m

```
% Split SAR Delta X Calibration Test
close all;
clear all;

fs = 100E3;
Vref = 1.8;
Vfrac = 0.96;

%filename = '../Bias_changes/Sine18kHz_1.8Vpk_0.9Vvcm_SARAB_shuffled_10.01.29.01.txt';
%filename = '../Noise_testing/Sine18kHz_FS_500uA_SARAB_shuffled_10.03.30.01.txt';
%filename = '../Noise_testing/FG_0.9Vdc_500uA_SARAB_shuffled.10.31.03.01.txt';
%filename = '../Noise_testing/Sine18kHz_quarter_scale_430uA_SARAB_shuffled_10.31.03.01.txt';
filename = '../DC_levels_V2/101_DC_Levels_500uA_SARAB_shuffled_10.31.03.01.txt';

%weightsfile = 'SAR_Chip1_Vcm0.92_DAC_Weights_2010.01.20.txt';
%weightsfile = 'SAR_Chip1_fixed_bias_Vcm0.9_DAC_shuffled_Weights_2010.02.03.txt';

%import_weights_file;

%Setup Initial Weights
hs = 1.8;  % Half-Scale voltage
init_seg1_weights = repmat(hs/2^4,16,1);
init_seg2_weights = repmat(hs/2^7,16,1);
init_seg3_weights = repmat(hs/2^10,16,1);
other_seg_weights = [hs/2^13; hs/2^16];

init_weights = [init_seg1_weights; init_seg2_weights;...
    init_seg3_weights; other_seg_weights];

%Initial offset
OS_init = 0;

W_A = init_weights;
W_B = init_weights;


dual_ADC_data_import_No_DOutP_B;

reshape_dual_data_shuffled_banks1_3_V03;
```

./source_code/split_sar/run_split_SAR_test_v03.m

```
% Split SAR Delta X Calibration Test
close all;
clear all;

fs = 100E3;
Vref = 1.8;
Vfrac = 0.96;

filename = '../Bias_changes/Sine18kHz_1.8Vpk_0.9Vvcm_SARAB_shuffled_10.01.29.01.txt';
%filename = '../Noise_testing/Sine18kHz_FS_500uA_SARAB_shuffled_10.03.30.01.txt';
%filename = '../Noise_testing/Grounded_SARAB_shuffled_10.04.08.01.txt';
%filename = '../Noise_testing/FG_0.9Vdc_500uA_SARAB_shuffled.10.31.03.01.txt';
%filename = '../Noise_testing/FG_1.8Vdc_no_bias_adjustment_SARAB_shuffled.10.30.03.01.txt';
%filename = '../Noise_testing/Sine18kHz_quarter_scale_430uA_SARAB_shuffled_10.31.03.01.txt';

%filename = '../../Chip2/Sine18kHz_FS_SARAB_shuffled_10.04.07.01.txt';
%filename = '../../Chip25/Sine18kHz_FS_SARAB_shuffled_10.04.07.01.txt';
%filename = '../../Chip7/Sine18kHz_FS_SARAB_shuffled_10.04.07.01.txt';

%weightsfile = 'SAR_Chip1_Vcm0.92_DAC_Weights_2010.01.20.txt';
%weightsfile = 'SAR_Chip1_fixed_bias_Vcm0.9_DAC_shuffled_Weights_2010.02.03.txt';

%import_weights_file;

%Setup Initial Weights
hs = 1.8;  % Half-Scale voltage
init_seg1_weights = repmat(hs/2^4,16,1);
init_seg2_weights = repmat(hs/2^7,16,1);
init_seg3_weights = repmat(hs/2^10,16,1);
other_seg_weights = [hs/2^13; hs/2^16];

init_weights = [init_seg1_weights; init_seg2_weights;...
    init_seg3_weights; other_seg_weights];

%Initial offset
OS_init = 0;

W_A = init_weights;
W_B = init_weights;


dual_ADC_data_import_no_limit_No_DOutP_B;

reshape_dual_data_shuffled_banks1_3_V03;

splitcal_V02i;
%splitcal_V02j2;

run_sine_test_v02;
```

./source_code/split_sar/SNDR_V02.m

```
% Calculate SNDR for SAR A
numpt = length(blackman_sine_A_fft);
% Find all frequencies that are NOT the input signal
fin=find(FFT_A(1:floor(numpt/2))>-1);
% Set span "window" for number of bins around the input frequency
span = 25;
% Get maximum value of noise floor
noise_level = max(FFT_A(50:250));
% Find bins with levels above noise floor
fhd = find(FFT_A(50:floor(numpt/2)) > (noise_level+2)) + 49;
% Remove levels associated with input signal to get only the harmonic bins
fhd = fhd( find(fhd > fin + span));

% Get total spectral power
spectP=(abs(blackman_sine_A_fft)).*(abs(blackman_sine_A_fft));

Pdc=sum(spectP(1:1+span));
Ps=sum(spectP(fin-span:fin+span));
Phd=sum(spectP(fhd));
Pn=sum(spectP(1:floor(numpt/2)))-Pdc-Ps-Phd;
SNR_SAR_A = 10*log10((Ps)/Pn)
THD_SAR_A = 10*log10((Phd)/Ps)
SNDR_SAR_A=10*log10((Ps+Pdc)/Pn)
%%
% Calculate SNDR for SAR B
numpt = length(blackman_sine_B_fft);
% Find all frequencies that are NOT the input signal
fin=find(FFT_B(1:floor(numpt/2))>-1);

% Set span "window" for number of bins around the input frequency
span = 25;
% Get maximum value of noise floor
noise_level = max(FFT_B(50:250));
% Find bins with levels above noise floor
fhd = find(FFT_B(50:floor(numpt/2)) > (noise_level+2)) + 49;
% Remove levels associated with input signal to get only the harmonic bins
fhd = fhd( find(fhd > fin + span));

% Get total spectral power
spectP=(abs(blackman_sine_B_fft)).*(abs(blackman_sine_B_fft));

Pdc=sum(spectP(1:1+span));
Ps=sum(spectP(fin-span:fin+span));
Phd=sum(spectP(fhd));
Pn=sum(spectP(1:floor(numpt/2)))-Pdc-Ps-Phd;
SNR_SAR_B = 10*log10((Ps)/Pn)
THD_SAR_B = 10*log10((Phd)/Ps)
SNDR_SAR_B=10*log10((Ps+Pdc)/Pn)

% Calculate SNDR for SAR avg
numpt = length(blackman_sine_avg_fft);
% Find all frequencies that are NOT the input signal
fin=find(FFT_avg(1:floor(numpt/2))>-1);

% Set span "window" for number of bins around the input frequency
```

```
  span = 25;
% Get maximum value of noise floor
  noise_level = max(FFT_avg(50:250));
% Find bins with levels above noise floor
  fhd = find(FFT_avg(50:floor(numpt/2)) > (noise_level+2)) + 49;
% Remove levels associated with input signal to get only the harmonic bins
  fhd = fhd( find(fhd > fin + span));

  % Get total spectral power
spectP=(abs(blackman_sine_avg_fft)).*(abs(blackman_sine_avg_fft));
  %spectP=(abs(FFT_avg)).*(abs(FFT_avg));

  Pdc=sum(spectP(1:1+span));
Ps=sum(spectP(fin-span:fin+span));
  Phd=sum(spectP(fhd));
Pn=sum(spectP(1:floor(numpt/2)))-Pdc-Ps-Phd;
  SNR_SAR_avg = 10*log10((Ps)/Pn)
THD_SAR_avg = 10*log10((Phd)/Ps)
  SNDR_SAR_avg=10*log10((Ps+Pdc)/Pn)
```

./source_code/split_sar/splitcalplot_SAR_AB_V01a.m

```
% splitcalplot

% plot results of split cal
figure(1)
plot(delta_x_track')
title('delta x track')

figure(2)
plot(OS_hat_track)
title('offset')

% Plot SAR A Weights and Errors

figure(3)
plot(eps_A_hat_track')
title('SAR A Weight Errors (Epsilon hat)')

figure(4)
plot((W_A_hat_track(:,:)-repmat(W_A_hat_final,1,num_iters))')
title('SAR A Weight Error')

figure(5)
semilogy(abs((W_A_hat_track(:,1:(end-1))-repmat(W_A_hat_final,1,num_iters-1))'))
title('SAR A Weight Error')

figure(6)
semilogy(abs((W_A_hat_track(:,1:(end-1))-repmat(W_A_hat_final,1,num_iters-1))'), ...
    'k','linewidth',1)
title('SEGMENT WEIGHT ERROR')
ylim([1e-9 3e-3])
%xlim([0 2000])
grid on

figure(7)
hold off
semilogy(abs((W_A_hat_track(:,1:(end-1))-repmat(W_A_hat_final,1,num_iters-1))'), ...
    'Color',[.5 .5 .5 ],'linewidth',1)
hold on
semilogy(std((W_A_hat_track(:,1:(end-1))-repmat(W_A_hat_final,1,num_iters-1))), ...
    'k','linewidth',2)
title('SEGMENT WEIGHT ERROR')
ylim([1e-9 3e-3])
%xlim([0 2000])
grid on

figure(8)
plot(filt_eps_A_hat_track(1:16,:)')
title('SAR A Filtered Epsilons 1-16')

figure(9)
plot(std(filt_eps_A_hat_track(1:16,:)'))
title('std SAR A Filtered Epsilons 1-16')

figure(10)
plot(W_A_hat_track(1:16,:)','k','linewidth',2)
```

```
title('SEGMENT WEIGHTS')
  %xlim([0 1000])

  figure(11)
subplot(5,3,1)
  plot(W_A_hat_track(1:16,:)')
title('A weights 1-16')
  subplot(5,3,4)
plot(W_A_hat_track(17:32,:)')
  title('block 2')
subplot(5,3,7)
  plot(W_A_hat_track(33:48,:)')
title('block 3')
  subplot(5,3,10)
plot(W_A_hat_track(49:50,:)')
  title('BLOCKS 4, 5')
subplot(5,3,13)
  plot(OS_hat_track)
title('offset')
  subplot(5,3,2)
plot(eps_A_hat_track(1:16,:)')
  title('A epsilon 1-16')
subplot(5,3,5)
  plot(eps_A_hat_track(17:32,:)')
title('A epsilon 17-32')
  subplot(5,3,8)
plot(eps_A_hat_track(33:48,:)')
  title('A epsilon 33-48')
subplot(5,3,11)
  plot(eps_A_hat_track(49:50,:)')
title('A epsilon 49,50')
  subplot(5,3,14)
plot(eps_os_hat_track')
  title('Epsilon Offset')
subplot(5,3,3)
  plot(filt_eps_A_hat_track(1:16,:)')
title('A filt epsilon 1-16')
  subplot(5,3,6)
plot(filt_eps_A_hat_track(17:32,:)')
  title('A filt epsilon 17-32')
subplot(5,3,9)
  plot(filt_eps_A_hat_track(33:48,:)')
title('A filt epsilon 33-48')
  subplot(5,3,12)
plot(filt_eps_A_hat_track(49:50,:)')
  title('A filt epsilon 49,50')
subplot(5,3,12)
  plot(filt_eps_os_hat_track')
title('Filtered Epsilon Offset')


  % Plot SAR B Weights and Errors


figure(12)
  plot(eps_B_hat_track')
title('SAR B Weight Errors (Epsilon_hat)')
```

```
figure(13)
plot((W_B_hat_track(:,:)-repmat(W_B_hat_final,1,num_iters))')
title('SAR B Weight Error')

figure(14)
semilogy(abs((W_B_hat_track(:,1:(end-1))-repmat(W_B_hat_final,1,num_iters-1))'))
title('SAR B Weight Error')

figure(15)
semilogy(abs((W_B_hat_track(:,1:(end-1))-repmat(W_B_hat_final,1,num_iters-1))'), ...
    'k','linewidth',1)
title('SEGMENT WEIGHT ERROR')
ylim([1e-9 3e-3])
%xlim([0 2000])
grid on

figure(16)
hold off
semilogy(abs((W_B_hat_track(:,1:(end-1))-repmat(W_B_hat_final,1,num_iters-1))'), ...
    'Color',[.5 .5 .5 ],'linewidth',1)
hold on
semilogy(std((W_B_hat_track(:,1:(end-1))-repmat(W_B_hat_final,1,num_iters-1))), ...
    'k','linewidth',2)
title('SEGMENT WEIGHT ERROR')
ylim([1e-9 3e-3])
%xlim([0 2000])
grid on

figure(17)
plot(filt_eps_B_hat_track(1:16,:)')
title('SAR A Filtered Epsilons 1-16')

figure(18)
plot(std(filt_eps_B_hat_track(1:16,:)'))
title('std SAR A Filtered Epsilons 1-16')

figure(19)
plot(W_B_hat_track(1:16,:)','k','linewidth',2)
title('SEGMENT WEIGHTS')
%xlim([0 1000])

figure(20)
subplot(5,3,1)
plot(W_B_hat_track(1:16,:)')
title('B weights 1-16')
subplot(5,3,4)
plot(W_B_hat_track(17:32,:)')
title('block 2')
subplot(5,3,7)
plot(W_B_hat_track(33:48,:)')
title('block 3')
subplot(5,3,10)
plot(W_B_hat_track(49:50,:)')
title('BLOCKS 4, 5')
subplot(5,3,13)
plot(OS_hat_track)
```

```
title('offset')
    subplot(5,3,2)
plot(eps_B_hat_track(1:16,:)')
    title('B epsilon 1-16')
subplot(5,3,5)
    plot(eps_B_hat_track(17:32,:)')
title('B epsilon 17-32')
    subplot(5,3,8)
plot(eps_B_hat_track(33:48,:)')
    title('B epsilon 33-48')
subplot(5,3,11)
    plot(eps_B_hat_track(49:50,:)')
title('B epsilon 49,50')
    subplot(5,3,14)
plot(eps_B_hat_track')
    title('B epsilon offset')
subplot(5,3,3)
    plot(filt_eps_B_hat_track(1:16,:)')
title('B filt epsilon 1-16')
    subplot(5,3,6)
plot(filt_eps_B_hat_track(17:32,:)')
    title('B filt epsilon 17-32')
subplot(5,3,9)
    plot(filt_eps_B_hat_track(33:48,:)')
title('B filt epsilon 33-48')
    subplot(5,3,12)
plot(filt_eps_B_hat_track(49:50,:)')
    title('B filt epsilon 49,50')
subplot(5,3,12)
    plot(filt_eps_os_hat_track')
title('Filtered Epsilon Offset')
```

./source_code/split_sar/splitcal_V02i.m

```
% Split SAR Calibration Algorithm Vers. 2 I
% Use iterative method without permutation of data blocks
% (Strongly) Force average of weights to be constant
% Track Weights and Errors after each block (of block_size)

% Initialize variables
num_weights = length(init_weights);

% Get the starting average of the total weights (A and B);
mean_total_init_weights = mean(init_weights);

% Calculated Epsilon Values
eps_A_hat = zeros(num_weights,1);
eps_B_hat = zeros(num_weights,1);
eps_os_hat = 0;

% Filtered Epsilon Values
% Since direct pinv method is used in this version, the filtered epsilon
% values will be equal to the calculated epsilon values
filt_eps_A_hat = zeros(num_weights,1);
filt_eps_B_hat = zeros(num_weights,1);
filt_eps_os_hat = 0;

W_A_hat = init_weights;
W_B_hat = init_weights;
OS_hat = 0;

% Setup Run Lengths

% data_set_length is the length of the importated data set
data_set_length = length(D_A_bits);

num_sets = 10;

data_length = num_sets * data_set_length;

% Permutation would go here
% For now, just use iterations;

dec_A = zeros(data_length, num_weights);
dec_B = zeros(data_length, num_weights);

for k = 1:num_sets;
    seq = 1:data_set_length;
    %seq = randperm(data_set_length);
    dec_A( ((k-1)*data_set_length)+1:(k*data_set_length), :) = ...
        D_A_bits(seq,:);
    %seq = randperm(data_set_length);
    dec_B( ((k-1)*data_set_length)+1:(k*data_set_length), :) = ...
        D_B_bits(seq,:);
end

% Calculate total run length and number of iterations

num_runs = 10;           % Set the number of runs used with this data set
```

```
56
  run_length = data_length;    % Get total run length
58
  block_size = 2^8;      % Set block size
60
  % Get total number of iterations used
62num_iters = floor(data_length/block_size) * num_runs;

64% Initialize variables used to keep track of Weights and Epsilons (errors)

66% Track calculated epsilons
  eps_A_hat_track = zeros(num_weights,num_iters);
68eps_B_hat_track = zeros(num_weights,num_iters);
  eps_os_hat_track = zeros(1,num_iters);
70
  % Track filtered epsilons
72filt_eps_A_hat_track = zeros(num_weights,num_iters);
  filt_eps_B_hat_track = zeros(num_weights,num_iters);
74filt_eps_os_hat_track = zeros(1,num_iters);

76% Track estimated weights
  W_A_hat_track = zeros(num_weights,num_iters);
78W_B_hat_track = zeros(num_weights,num_iters);
  OS_hat_track = zeros(1,num_iters);
80
  % Track delta_x values
82% Force avg error set to zero (the reason for the +2)
  %delta_x_track = zeros(block_size + 2,num_iters);
84delta_x_track = zeros(block_size + 1,num_iters);

86% Set LMS loop parameters
  mu_e = 2^-6;
88mu_w = 2^-15;

90
  % Initialize iteration count;
92iter_cntr = 1;

94for run_cntr = 1:num_runs;
      for i = 1:block_size:run_length-block_size;
96
          % First apply current weights and offsets to find x_hat values
98
          dec_A_i = dec_A(i: (i+block_size) -1,:);
100       dec_B_i = dec_B(i: (i+block_size) -1,:);

102       x_A_hat = dec_A_i * W_A_hat - OS_hat/2;
          x_B_hat = dec_B_i * W_B_hat + OS_hat/2;
104
          % Then calculate delta_x and record those values
106       % Force average error to be zero
          delta_x = x_B_hat - x_A_hat;
108       delta_x = [delta_x; 0];

110       delta_x_track(:,iter_cntr) = delta_x;

112       % Generate the estimation matrix using the decisions
```

```
        % Force the sum of the errors to be zero
        est_mat = [ -dec_A_i dec_B_i ones(block_size,1)];
        est_mat = [est_mat; block_size*ones(1,2*num_weights) 0];

        % Calculate estimated errors using pinv method
        %[epsilons] = pinv(est_mat) * delta_x;
        epsilons = sign(est_mat') * delta_x;

        eps_A_hat = epsilons(1:num_weights);
        eps_B_hat = epsilons(num_weights+1:2*num_weights);
        eps_os_hat = epsilons(end);

        % Record calculated epsilons
        eps_A_hat_track(:,iter_cntr) = eps_A_hat;
        eps_B_hat_track(:,iter_cntr) = eps_B_hat;
        eps_os_hat_track(iter_cntr) = eps_os_hat;

        % Filter epsilons with LMS loop

        filt_eps_A_hat = (1-mu_e)*filt_eps_A_hat + mu_e * eps_A_hat;
        filt_eps_B_hat = (1-mu_e)*filt_eps_B_hat + mu_e * eps_B_hat;
        filt_eps_os_hat = (1-mu_e)*filt_eps_os_hat + mu_e * eps_os_hat;

        % Record calculated epsilons
        filt_eps_A_hat_track(:,iter_cntr) = filt_eps_A_hat;
        filt_eps_B_hat_track(:,iter_cntr) = filt_eps_B_hat;
        filt_eps_os_hat_track(iter_cntr) = filt_eps_os_hat;

        % Get new wieghts using old weights
        W_A_hat = W_A_hat - mu_w * filt_eps_A_hat;
        W_B_hat = W_B_hat - mu_w * filt_eps_B_hat;

        % Force the average total weights to remain constant
        % (But allow A and B Weights to have different gains)
        W_hat = [W_A_hat; W_B_hat];
        W_mean_gain = mean_total_init_weights/mean(W_hat);
        W_A_hat = W_A_hat * W_mean_gain;
        W_B_hat = W_B_hat * W_mean_gain;

        OS_hat = OS_hat - mu_w * filt_eps_os_hat;

        W_A_hat_track(:,iter_cntr) = W_A_hat;
        W_B_hat_track(:,iter_cntr) = W_B_hat;
        OS_hat_track(:,iter_cntr) = OS_hat;

        % Increase the Iteration Counter
        iter_cntr = iter_cntr + 1;
    end

    Run_Count = run_cntr
end

% Get the sums of the calculated epsilons
sum_eps_A_hat = sum(eps_A_hat_track);
sum_eps_B_hat = sum(eps_B_hat_track);

% Get the sums of the filtered epsilons
```

```
170 sum_filt_eps_A_hat = sum(filt_eps_A_hat_track);
    sum_filt_eps_B_hat = sum(filt_eps_B_hat_track);
172
    W_A_hat_final = mean(W_A_hat_track(:,(end-100):end),2);
174 W_B_hat_final = mean(W_B_hat_track(:,(end-100):end),2);
    OS_hat_final = mean(OS_hat_track(:,(end-100):end),2);
```

./source_code/split_sar/splitcal_V02j2.m

```
% Split SAR Calibration Algorithm Vers. 2 J
% Use iterative method without permutation of data blocks
% (Strongly) Force average of weights to be constant
% This is broken down by each bank of weights in an attempt to fix
% DC convergence
% Track Weights and Errors after each block (of block_size)

% Initialize variables
num_weights = length(init_weights);

% Get the starting average of the total weights (A and B);
mean_init_weights_bank_12 = mean(init_weights(1:32));
mean_init_weights_bank_345 = mean(init_weights(33:50));

% Calculated Epsilon Values
eps_A_hat = zeros(num_weights,1);
eps_B_hat = zeros(num_weights,1);
eps_os_hat = 0;

% Filtered Epsilon Values
% Since direct pinv method is used in this version, the filtered epsilon
% values will be equal to the calculated epsilon values
filt_eps_A_hat = zeros(num_weights,1);
filt_eps_B_hat = zeros(num_weights,1);
filt_eps_os_hat = 0;

W_A_hat = init_weights;
W_B_hat = init_weights;
OS_hat = 0;

% Setup Run Lengths

% data_set_length is the length of the importated data set
data_set_length = length(D_A_bits);

num_sets = 10;

data_length = num_sets * data_set_length;

% Permutation would go here
% For now, just use iterations;

dec_A = zeros(data_length, num_weights);
dec_B = zeros(data_length, num_weights);

for k = 1:num_sets;
    seq = 1:data_set_length;
    %seq = randperm(data_set_length);
    dec_A( ((k-1)*data_set_length)+1:(k*data_set_length), :) = ...
        D_A_bits(seq,:);
    %seq = randperm(data_set_length);
    dec_B( ((k-1)*data_set_length)+1:(k*data_set_length), :) = ...
        D_B_bits(seq,:);
end

```

```
% Calculate total run length and number of iterations

num_runs = 15;           % Set the number of runs used with this data set

run_length = data_length;    % Get total run length

block_size = 2^12;      % Set block size

% Get total number of iterations used
num_iters = floor(data_length/block_size) * num_runs;

% Initialize variables used to keep track of Weights and Epsilons (errors)

% Track calculated epsilons
eps_A_hat_track = zeros(num_weights,num_iters);
eps_B_hat_track = zeros(num_weights,num_iters);
eps_os_hat_track = zeros(1,num_iters);

% Track filtered epsilons
filt_eps_A_hat_track = zeros(num_weights,num_iters);
filt_eps_B_hat_track = zeros(num_weights,num_iters);
filt_eps_os_hat_track = zeros(1,num_iters);

% Track estimated weights
W_A_hat_track = zeros(num_weights,num_iters);
W_B_hat_track = zeros(num_weights,num_iters);
OS_hat_track = zeros(1,num_iters);

% Track delta_x values
% Force avg error set to zero (the reason for the +2)
%delta_x_track = zeros(block_size + 2,num_iters);
delta_x_track = zeros(block_size + 1,num_iters);

% Set LMS loop parameters
mu_e = 2^-6;
mu_w = 2^-15;


% Initialize iteration count;
iter_cntr = 1;

for run_cntr = 1:num_runs;
    for i = 1:block_size:run_length-block_size;

        % First apply current weights and offsets to find x_hat values

        dec_A_i = dec_A(i: (i+block_size) -1,:);
        dec_B_i = dec_B(i: (i+block_size) -1,:);

        x_A_hat = dec_A_i * W_A_hat - OS_hat/2;
        x_B_hat = dec_B_i * W_B_hat + OS_hat/2;

        % Then calculate delta_x and record those values
        % Force average error to be zero
        delta_x = x_B_hat - x_A_hat;
        delta_x = [delta_x; 0];
```

```
113         delta_x_track(:,iter_cntr) = delta_x;

115         % Generate the estimation matrix using the decisions
            % Force the sum of the errors to be zero
117         est_mat = [ -dec_A_i dec_B_i ones(block_size,1)];
            est_mat = [est_mat; block_size*ones(1,2*num_weights) 0];
119
            % Calculate estimated errors using pinv method
121         %[epsilons] = pinv(est_mat) * delta_x;
            epsilons = sign(est_mat') * delta_x;
123
            eps_A_hat = epsilons(1:num_weights);
125         eps_B_hat = epsilons(num_weights+1:2*num_weights);
            eps_os_hat = epsilons(end);
127
            % Record calculated epsilons
129         eps_A_hat_track(:,iter_cntr) = eps_A_hat;
            eps_B_hat_track(:,iter_cntr) = eps_B_hat;
131         eps_os_hat_track(iter_cntr) = eps_os_hat;

133         % Filter epsilons with LMS loop

135         filt_eps_A_hat = (1-mu_e)*filt_eps_A_hat + mu_e * eps_A_hat;
            filt_eps_B_hat = (1-mu_e)*filt_eps_B_hat + mu_e * eps_B_hat;
137         filt_eps_os_hat = (1-mu_e)*filt_eps_os_hat + mu_e * eps_os_hat;

139         % Record calculated epsilons
            filt_eps_A_hat_track(:,iter_cntr) = filt_eps_A_hat;
141         filt_eps_B_hat_track(:,iter_cntr) = filt_eps_B_hat;
            filt_eps_os_hat_track(iter_cntr) = filt_eps_os_hat;
143
            % Get new wieghts using old weights
145         W_A_hat = W_A_hat - mu_w * filt_eps_A_hat;
            W_B_hat = W_B_hat - mu_w * filt_eps_B_hat;
147
            % Force the average total weights to remain constant
149         % (But allow A and B Weights to have different gains)
            W_hat_bank12 = [W_A_hat(1:32); W_B_hat(1:32)];
151         W_hat_bank345 = [W_A_hat(33:50); W_B_hat(33:50)];
            W_mean_gain_12 = mean_init_weights_bank_12/mean(W_hat_bank12);
153         W_mean_gain_345 = mean_init_weights_bank_345/mean(W_hat_bank345);
            W_A_hat(1:32) = W_hat_bank12(1:32) * W_mean_gain_12;
155         W_B_hat(1:32) = W_hat_bank12(33:64) * W_mean_gain_12;
            W_A_hat(33:50) = W_hat_bank345(1:18) * W_mean_gain_345;
157         W_B_hat(33:50) = W_hat_bank345(19:36) * W_mean_gain_345;

159         OS_hat = OS_hat - mu_w * filt_eps_os_hat;

161         W_A_hat_track(:,iter_cntr) = W_A_hat;
            W_B_hat_track(:,iter_cntr) = W_B_hat;
163         OS_hat_track(:,iter_cntr) = OS_hat;

165         % Increase the Iteration Counter
            iter_cntr = iter_cntr + 1;
167     end

169     Run_Count = run_cntr
```

```
    end

    % Get the sums of the calculated epsilons
    sum_eps_A_hat = sum(eps_A_hat_track);
    sum_eps_B_hat = sum(eps_B_hat_track);

    % Get the sums of the filtered epsilons
    sum_filt_eps_A_hat = sum(filt_eps_A_hat_track);
    sum_filt_eps_B_hat = sum(filt_eps_B_hat_track);

    W_A_hat_final = mean(W_A_hat_track(:,(end-100):end),2);
    W_B_hat_final = mean(W_B_hat_track(:,(end-100):end),2);
    OS_hat_final = mean(OS_hat_track(:,(end-100):end),2);
```

Appendix C

Split-SAR Hardware Description

./source_code/SARLogicBlockV03.v

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    15:55:50 10/08/2008
// Design Name:
// Module Name:    SARLogicBlockV01
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module SARLogicBlockV03(reset, comp_p, comp_n, sample, serClk, serBaseIn, serBaseOut,
      latch_In, preamp_In, latchEn, preampEn,
      capSelect0T, capSelect1T, capSelect2T, capSelect3T, capSelect4T,
      capSelect0B, capSelect1B, capSelect2B, capSelect3B, capSelect4B);

    input reset;
    input comp_p, comp_n;
    input sample;
    input serClk;
    input serBaseIn;
    input latch_In;
    input preamp_In;
    output latchEn;
    output preampEn;
    output serBaseOut;
```

```
36
    //Top
38   output [31:0] capSelect0T;
    output [31:0] capSelect1T;
40  output [31:0] capSelect2T;
    output [31:0] capSelect3T;
42  output [31:0] capSelect4T;

44  //Bottom
    output [31:0] capSelect0B;
46  output [31:0] capSelect1B;
    output [31:0] capSelect2B;
48  output [31:0] capSelect3B;
    output [31:0] capSelect4B;
50
    //wire sample;
52  wire [3:0] base0;
     wire [3:0] base1;
54   wire [3:0] base2;
     wire [3:0] base3;
56  wire [3:0] base4;
    wire [4:0] SARAddr;
58  wire [39:0] bitSelect;
    wire decStore;
60

62 SARControl04 ControlBlock(reset, sample, latch_In, preamp_In, latchEn, preampEn, decStore, SARAddr);
   SAR02 SARArray(comp_p, comp_n, decStore, SARAddr, sample, bitSelect);
64 BaseIOV01 SerBaseBlock(reset, serBaseIn, sample, serClk, base0, base1, base2, base3, base4, serBaseOut);
   HalfDACCapMux CapMuxB(base0, base1, base2, base3, base4, bitSelect, capSelect0B, capSelect1B,
66              capSelect2B, capSelect3B, capSelect4B);
   HalfDACCapMux CapMuxT(base0, base1, base2, base3, base4, bitSelect, capSelect0T, capSelect1T,
68              capSelect2T, capSelect3T, capSelect4T);

70

72 endmodule
```

./source_code/SARControlV04.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    14:16:06 10/02/2008
// Design Name:
// Module Name:    SARControl03
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module SARControl04(reset, sample, latchEn_Ext, preampEn_Ext, latchEn, preampEn, decStore, SARAddr);

  input reset;
  input sample;
  input latchEn_Ext;
  input preampEn_Ext;

  output latchEn;
  output preampEn;
  output decStore;
  output SARAddr;
  //output rdy;

  // At this time, unused parameters
  //parameter WAIT = 0;
  //parameter SAMPLE = 2'd1;
  //parameter BITCYCLE = 2'd2;


  reg done;
  reg [4:0] SARAddr;  //SARAddr corresponds to bit number

  //Nonoverlap logic for internal LatchEn, PreampEn, and SoreBitDecision Signals
  assign decStore = ~latchEn_Ext;
  assign latchEn = ~decStore;
  assign preampEn = latchEn | preampEn_Ext;

  always @ (posedge preampEn_Ext or posedge reset or posedge sample)
  begin
    if (reset)
    // If Reset signal is asserted return all internal and output
    // signals to their initial states
    begin
      SARAddr <= 5'd20;
      done <= 1;
```

```
56      end
        else if (sample) //If sample, reset SARAddr and clear done signal
58      // this restarts the bit cycling mode
        begin
60        SARAddr <= 5'd20;
          done <= 0;
62      end
        else //Otherwise, preampEn_Ext is active, so check bit cycle status
64      begin
          if (!done) //If not "done" with the bit cycle, check Bit Number
66        begin
            if (SARAddr == 0) //If on the last bit cycle, toggle done
68          begin
              //SARAddr <= 5'd19;
70            done <= 1;
            end
72          else  //Else end of 20 bit cycle, set "done" signal
            begin
74            done <= 0;
            end
76          SARAddr <= SARAddr - 1; //Regardless, alway decrement SARAddr
          // This is used to allow for storing the "21st" bit if necessary
78        // patchEn_Ext and preampEn_Ext wil always trigger a StoreDecision signal
          // As long as SARAddr is >19 (i.e. not 0 through 19) then only the positive
80        // output of the comparator will be stored onto the output data register.
          // The SARegister will remain unchanged, maintaining the Cap DAC switch selections.
82        // By always decrementing the SARAddr counter when preampEn_Ext goes high,
          // the counter is allowed to rollover to 31, preventing the Comp Decision from
84        // being stored into the SAR (it is only stored into the Output Data Register).
          end // End of if (!done) case
86      end // end of else Reset or Smaple case
    end // End of always @ (posedge clk1 or posedge decStore or negedge preamp)
88 endmodule
```

./source_code/SAR02.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    20:32:03 10/01/2008
// Design Name:
// Module Name:    SAR01
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module SAR02(comp_p, comp_n, store, address, preset, bitSelect);
  input comp_p;
  input comp_n;
  input store;
  input [4:0] address;
  input preset;
  output [39:0] bitSelect;

  wire [1:0] comp_in;   // 2-bit Comparator input (pos and neg)
  //wire store_d;

  reg [1:0] SARMem [0:19];  // Memory style register for storing the 20, 2-bit
  // decisions from the output comparator.  This eleminates the need for 20 "store"
  // or clock signals for each 2-bit register for each bit of the converter.
  //reg dataOut;

  assign comp_in[1] = comp_n; // Combine the Comparator pos and neg inputs
  assign comp_in[0] = comp_p; // into one signal
  //assign store_d = store;

  // Assign the 2-bit SARs to their respective Bit Select outputs
  assign bitSelect[1:0] = SARMem[19];
  assign bitSelect[3:2] = SARMem[18];
  assign bitSelect[5:4] = SARMem[17];
  assign bitSelect[7:6] = SARMem[16];
  assign bitSelect[9:8] = SARMem[15];
  assign bitSelect[11:10] = SARMem[14];
  assign bitSelect[13:12] = SARMem[13];
  assign bitSelect[15:14] = SARMem[12];
  assign bitSelect[17:16] = SARMem[11];
  assign bitSelect[19:18] = SARMem[10];
  assign bitSelect[21:20] = SARMem[9];
  assign bitSelect[23:22] = SARMem[8];
  assign bitSelect[25:24] = SARMem[7];
  assign bitSelect[27:26] = SARMem[6];
```

```
56  assign bitSelect[29:28] = SARMem[5];
    assign bitSelect[31:30] = SARMem[4];
58  assign bitSelect[33:32] = SARMem[3];
    assign bitSelect[35:34] = SARMem[2];
60  assign bitSelect[37:36] = SARMem[1];
    assign bitSelect[39:38] = SARMem[0];
62
    always @ (posedge store or posedge preset)
64  begin
      if (preset) // If preset, reset the SAR to 2'b11 (Vcm)
66    begin
        SARMem[0] = 2'b11;
68      SARMem[1] = 2'b11;
        SARMem[2] = 2'b11;
70      SARMem[3] = 2'b11;
        SARMem[4] = 2'b11;
72      SARMem[5] = 2'b11;
        SARMem[6] = 2'b11;
74      SARMem[7] = 2'b11;
        SARMem[8] = 2'b11;
76      SARMem[9] = 2'b11;
        SARMem[10] = 2'b11;
78      SARMem[11] = 2'b11;
        SARMem[12] = 2'b11;
80      SARMem[13] = 2'b11;
        SARMem[14] = 2'b11;
82      SARMem[15] = 2'b11;
        SARMem[16] = 2'b11;
84      SARMem[17] = 2'b11;
        SARMem[18] = 2'b11;
86      SARMem[19] = 2'b11;
        //dataOut <= 0;
88    end
      else
90    begin
        // As long as SARAddr is >19 (i.e. not 0 through 19) then only the positive
92      // output of the comparator will be stored onto the output data register.
        // The SARegister will remain unchanged, maintaining the Cap DAC switch selections.
94      if (((address >= 0) || (address < 20)))
        begin
96        SARMem[address] = comp_in;
        end
98      //dataOut <= comp_p;  // Store the positive Comparator Output
        // into the serial data output register
100   end
    end
102
  endmodule
```

./source_code/HalfDACCapMux.v

```
1 `timescale 1ns / 1ps
  //////////////////////////////////////////////////////////////////////////////////
3 // Company:
  // Engineer:
5 //
  // Create Date:    16:10:00 09/18/2008
7 // Design Name:
  // Module Name:    CapMux_DUT01
9 // Project Name:
  // Target Devices:
11 // Tool versions:
  // Description:
13 //
  // Dependencies:
15 //
  // Revision:
17 // Revision 0.01 - File Created
  // Additional Comments:
19 //
  //////////////////////////////////////////////////////////////////////////////////
21 module HalfDACCapMux(base0, base1, base2, base3, base4, BitSelect, capSelect0, capSelect1,
                capSelect2, capSelect3, capSelect4);
23
      input [3:0] base0;
25   input [3:0] base1;
     input [3:0] base2;
27   input [3:0] base3;
     input [3:0] base4;
29    input [39:0] BitSelect;
     output [31:0] capSelect0;
31   output [31:0] capSelect1;
     output [31:0] capSelect2;
33   output [31:0] capSelect3;
     output [31:0] capSelect4;
35

37  CapMux CapMux0(base0, BitSelect[1:0], BitSelect[3:2], BitSelect[5:4], BitSelect[7:6], capSelect0);
    CapMux CapMux1(base1, BitSelect[9:8], BitSelect[11:10], BitSelect[13:12], BitSelect[15:14], capSelect1);
39  CapMux CapMux2(base2, BitSelect[17:16], BitSelect[19:18], BitSelect[21:20], BitSelect[23:22], capSelect2);
    CapMux CapMux3(base3, BitSelect[25:24], BitSelect[27:26], BitSelect[29:28], BitSelect[31:30], capSelect3);
41  CapMux CapMux4(base4, BitSelect[33:32], BitSelect[35:34], BitSelect[37:36], BitSelect[39:38], capSelect4);

43
  endmodule
```

./source_code/CapMux.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    12:46:00 09/16/2008
// Design Name:
// Module Name:    CapMux01
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module CapMux(base, B0S, B1S, B2S, B3S, capSelect);
    input [3:0] base;
    input [1:0] B0S;
    input [1:0] B1S;
    input [1:0] B2S;
    input [1:0] B3S;
    output [31:0] capSelect;

   reg [1:0] capBank [15:0];  //Memory-Style Bank for storing the 16, 2-bit selects
   //This makes for slightly easier coding by assigning the BnS signal to the
   //apropriate individual Cap switch signal

   assign capSelect[1:0] = capBank[0];
   assign capSelect[3:2] = capBank[1];
   assign capSelect[5:4] = capBank[2];
   assign capSelect[7:6] = capBank[3];
   assign capSelect[9:8] = capBank[4];
   assign capSelect[11:10] = capBank[5];
   assign capSelect[13:12] = capBank[6];
   assign capSelect[15:14] = capBank[7];
   assign capSelect[17:16] = capBank[8];
   assign capSelect[19:18] = capBank[9];
   assign capSelect[21:20] = capBank[10];
   assign capSelect[23:22] = capBank[11];
   assign capSelect[25:24] = capBank[12];
   assign capSelect[27:26] = capBank[13];
   assign capSelect[29:28] = capBank[14];
   assign capSelect[31:30] = capBank[15];

   always @ (base or B0S or B1S or B2S or B3S)
   begin
    case (base)
      4'd0: //Cap0 is dummy cap
      begin
        capBank[0] = 2'b11;
```

```
56          capBank[1] = B0S;
            capBank[2] = B0S;
58          capBank[3] = B0S;
            capBank[4] = B0S;
60          capBank[5] = B0S;
            capBank[6] = B0S;
62          capBank[7] = B0S;
            capBank[8] = B0S;
64          capBank[9] = B1S;
            capBank[10] = B1S;
66          capBank[11] = B1S;
            capBank[12] = B1S;
68          capBank[13] = B2S;
            capBank[14] = B2S;
70          capBank[15] = B3S;
        end
72      4'd1: //Cap1 is dummy cap
        begin
74          capBank[0] = B3S;
            capBank[1] = 2'b11;
76          capBank[2] = B0S;
            capBank[3] = B0S;
78          capBank[4] = B0S;
            capBank[5] = B0S;
80          capBank[6] = B0S;
            capBank[7] = B0S;
82          capBank[8] = B0S;
            capBank[9] = B0S;
84          capBank[10] = B1S;
            capBank[11] = B1S;
86          capBank[12] = B1S;
            capBank[13] = B1S;
88          capBank[14] = B2S;
            capBank[15] = B2S;
90      end
        4'd2: //Cap2 is dummy cap
92      begin
            capBank[0] = B2S;
94          capBank[1] = B3S;
            capBank[2] = 2'b11;
96          capBank[3] = B0S;
            capBank[4] = B0S;
98          capBank[5] = B0S;
            capBank[6] = B0S;
100         capBank[7] = B0S;
            capBank[8] = B0S;
102         capBank[9] = B0S;
            capBank[10] = B0S;
104         capBank[11] = B1S;
            capBank[12] = B1S;
106         capBank[13] = B1S;
            capBank[14] = B1S;
108         capBank[15] = B2S;
        end
110     4'd3: //Cap3 is dummy cap
        begin
112         capBank[0] = B2S;
```

```
        capBank[1] = B2S;
        capBank[2] = B3S;
        capBank[3] = 2'b11;
        capBank[4] = B0S;
        capBank[5] = B0S;
        capBank[6] = B0S;
        capBank[7] = B0S;
        capBank[8] = B0S;
        capBank[9] = B0S;
        capBank[10] = B0S;
        capBank[11] = B0S;
        capBank[12] = B1S;
        capBank[13] = B1S;
        capBank[14] = B1S;
        capBank[15] = B1S;
      end
      4'd4: //Cap4 is dummy cap
      begin
        capBank[0] = B1S;
        capBank[1] = B2S;
        capBank[2] = B2S;
        capBank[3] = B3S;
        capBank[4] = 2'b11;
        capBank[5] = B0S;
        capBank[6] = B0S;
        capBank[7] = B0S;
        capBank[8] = B0S;
        capBank[9] = B0S;
        capBank[10] = B0S;
        capBank[11] = B0S;
        capBank[12] = B0S;
        capBank[13] = B1S;
        capBank[14] = B1S;
        capBank[15] = B1S;
      end
      4'd5: //Cap5 is dummy cap
      begin
        capBank[0] = B1S;
        capBank[1] = B1S;
        capBank[2] = B2S;
        capBank[3] = B2S;
        capBank[4] = B3S;
        capBank[5] = 2'b11;
        capBank[6] = B0S;
        capBank[7] = B0S;
        capBank[8] = B0S;
        capBank[9] = B0S;
        capBank[10] = B0S;
        capBank[11] = B0S;
        capBank[12] = B0S;
        capBank[13] = B0S;
        capBank[14] = B1S;
        capBank[15] = B1S;
      end
      4'd6: //Cap6 is dummy cap
      begin
        capBank[0] = B1S;
```

```
170         capBank[1] = B1S;
            capBank[2] = B1S;
172         capBank[3] = B2S;
            capBank[4] = B2S;
174         capBank[5] = B3S;
            capBank[6] = 2'b11;
176         capBank[7] = B0S;
            capBank[8] = B0S;
178         capBank[9] = B0S;
            capBank[10] = B0S;
180         capBank[11] = B0S;
            capBank[12] = B0S;
182         capBank[13] = B0S;
            capBank[14] = B0S;
184         capBank[15] = B1S;
        end
186     4'd7: //Cap7 is dummy cap
        begin
188         capBank[0] = B1S;
            capBank[1] = B1S;
190         capBank[2] = B1S;
            capBank[3] = B1S;
192         capBank[4] = B2S;
            capBank[5] = B2S;
194         capBank[6] = B3S;
            capBank[7] = 2'b11;
196         capBank[8] = B0S;
            capBank[9] = B0S;
198         capBank[10] = B0S;
            capBank[11] = B0S;
200         capBank[12] = B0S;
            capBank[13] = B0S;
202         capBank[14] = B0S;
            capBank[15] = B0S;
204     end
        4'd8: //Cap8 is dummy cap
206     begin
            capBank[0] = B0S;
208         capBank[1] = B1S;
            capBank[2] = B1S;
210         capBank[3] = B1S;
            capBank[4] = B1S;
212         capBank[5] = B2S;
            capBank[6] = B2S;
214         capBank[7] = B3S;
            capBank[8] = 2'b11;
216         capBank[9] = B0S;
            capBank[10] = B0S;
218         capBank[11] = B0S;
            capBank[12] = B0S;
220         capBank[13] = B0S;
            capBank[14] = B0S;
222         capBank[15] = B0S;
        end
224     4'd9: //Cap9 is dummy cap
        begin
226         capBank[0] = B0S;
```

```
        capBank[1] = B0S;
        capBank[2] = B1S;
        capBank[3] = B1S;
        capBank[4] = B1S;
        capBank[5] = B1S;
        capBank[6] = B2S;
        capBank[7] = B2S;
        capBank[8] = B3S;
        capBank[9] = 2'b11;
        capBank[10] = B0S;
        capBank[11] = B0S;
        capBank[12] = B0S;
        capBank[13] = B0S;
        capBank[14] = B0S;
        capBank[15] = B0S;
      end
      4'd10:  //Cap10 is dummy cap
      begin
        capBank[0] = B0S;
        capBank[1] = B0S;
        capBank[2] = B0S;
        capBank[3] = B1S;
        capBank[4] = B1S;
        capBank[5] = B1S;
        capBank[6] = B1S;
        capBank[7] = B2S;
        capBank[8] = B2S;
        capBank[9] = B3S;
        capBank[10] = 2'b11;
        capBank[11] = B0S;
        capBank[12] = B0S;
        capBank[13] = B0S;
        capBank[14] = B0S;
        capBank[15] = B0S;
      end
      4'd11:  //Cap11 is dummy cap
      begin
        capBank[0] = B0S;
        capBank[1] = B0S;
        capBank[2] = B0S;
        capBank[3] = B0S;
        capBank[4] = B1S;
        capBank[5] = B1S;
        capBank[6] = B1S;
        capBank[7] = B1S;
        capBank[8] = B2S;
        capBank[9] = B2S;
        capBank[10] = B3S;
        capBank[11] = 2'b11;
        capBank[12] = B0S;
        capBank[13] = B0S;
        capBank[14] = B0S;
        capBank[15] = B0S;
      end
      4'd12:  //Cap12 is dummy cap
      begin
        capBank[0] = B0S;
```

```
284         capBank[1] = B0S;
            capBank[2] = B0S;
286         capBank[3] = B0S;
            capBank[4] = B0S;
288         capBank[5] = B1S;
            capBank[6] = B1S;
290         capBank[7] = B1S;
            capBank[8] = B1S;
292         capBank[9] = B2S;
            capBank[10] = B2S;
294         capBank[11] = B3S;
            capBank[12] = 2'b11;
296         capBank[13] = B0S;
            capBank[14] = B0S;
298         capBank[15] = B0S;
        end
300     4'd13:  //Cap13 is dummy cap
        begin
302         capBank[0] = B0S;
            capBank[1] = B0S;
304         capBank[2] = B0S;
            capBank[3] = B0S;
306         capBank[4] = B0S;
            capBank[5] = B0S;
308         capBank[6] = B1S;
            capBank[7] = B1S;
310         capBank[8] = B1S;
            capBank[9] = B1S;
312         capBank[10] = B2S;
            capBank[11] = B2S;
314         capBank[12] = B3S;
            capBank[13] = 0;
316         capBank[14] = B0S;
            capBank[15] = B0S;
318     end
        4'd14:  //Cap14 is dummy cap
320     begin
            capBank[0] = B0S;
322         capBank[1] = B0S;
            capBank[2] = B0S;
324         capBank[3] = B0S;
            capBank[4] = B0S;
326         capBank[5] = B0S;
            capBank[6] = B0S;
328         capBank[7] = B1S;
            capBank[8] = B1S;
330         capBank[9] = B1S;
            capBank[10] = B1S;
332         capBank[11] = B2S;
            capBank[12] = B2S;
334         capBank[13] = B3S;
            capBank[14] = 2'b11;
336         capBank[15] = B0S;
        end
338     4'd15:  //Cap15 is dummy cap
        begin
340         capBank[0] = B0S;
```

```
            capBank[1] = B0S;
342         capBank[2] = B0S;
            capBank[3] = B0S;
344         capBank[4] = B0S;
            capBank[5] = B0S;
346         capBank[6] = B0S;
            capBank[7] = B0S;
348         capBank[8] = B1S;
            capBank[9] = B1S;
350         capBank[10] = B1S;
            capBank[11] = B1S;
352         capBank[12] = B2S;
            capBank[13] = B2S;
354         capBank[14] = B3S;
            capBank[15] = 2'b11;
356       end
          default:
358       begin
            capBank[0] = 2'b11;
360         capBank[1] = B0S;
            capBank[2] = B0S;
362         capBank[3] = B0S;
            capBank[4] = B0S;
364         capBank[5] = B0S;
            capBank[6] = B0S;
366         capBank[7] = B0S;
            capBank[8] = B0S;
368         capBank[9] = B1S;
            capBank[10] = B1S;
370         capBank[11] = B1S;
            capBank[12] = B1S;
372         capBank[13] = B2S;
            capBank[14] = B2S;
374         capBank[15] = B3S;
          end
376     endcase
      end
378 endmodule
```

./source_code/NonOverlapSampleT.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    14:14:38 09/15/2008
// Design Name:
// Module Name:    NonOverlapV01
// Project Name:
// Target Devices:
// Tool versions:
// Description: Provide non-overlapping signals to the capacitor switches to avoid
// turning on multiple switches at once
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module NonOverlapSampleT(sample, select, svin_g, svrefpB_g, svrefn_g, svcm_g,
    svin, svrefpB, svrefn, svcm);
 input [1:0] select; //Cap switch decision
input sample;    //Sample Mode signal (select Vin)
 input svrefpB_g;  //Select Vrefp (after inverters)
 input svrefn_g; //Select Vrefn (after inverters)
 input svcm_g;   //Select Vcm (after inverters)
input svin_g;    //Select Vin (after inverters)
output svin;   //Select Vin (before inverters)
 output svrefpB; //Select Vrefp (before inverters)
 output svrefn;  //Select Vrefn (before inverters)
 output svcm;    //Select Vcm (before inverters)

reg svrefpB, svrefn, svcm, svin;

always @ (sample, select, svin_g, svrefpB_g, svrefn_g, svcm_g)
begin
 if (sample) //If it's time to sample a new Vin
 begin
   svin = 1; //Switch Cap to Vin (ignore race glitch here)
   svcm = 0; //Turn off all other sample switches
   svrefpB = 1;
   svrefn = 0;
 end
 else  //If not sampling Vin, then test select to determine which switch to use
 begin
   svin = 0;
   case (select)
     2'b11:  //Select Vcm, but only if the other sitches are done switching
     begin
       if (svrefpB_g && !svrefn_g && svin_g)
         svcm = 1;
       else
         svcm = 0;
```

```
56          svrefpB = 1;
            svrefn = 0;
58        end
          2'b10:  //Select Vrefp, but only if the other sitches are done switching
60        begin
              if (!svcm_g && !svrefn_g && svin_g)
62            svrefpB = 0;
            else
64            svrefpB = 1;
            svcm = 0;
66          svrefn = 0;
          end
68        2'b01:  //Select Vrefn, but only if the other sitches are done switching
          begin
70          if (!svcm_g && svrefpB_g && svin_g)
              svrefn = 1;
72          else
              svrefn = 0;
74          svcm = 0;
            svrefpB = 1;
76        end
          default:
78        begin
            svcm = 1;
80          svrefpB = 1;
            svrefn = 0;
82        end
        endcase //End of (select) case statement
84    end //End of else statement from if (sample)
    end //End of always block
86

88 endmodule
```

./source_code/NonOverlapSampleB.v

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    14:14:38 09/15/2008
// Design Name:
// Module Name:    NonOverlapV01
// Project Name:
// Target Devices:
// Tool versions:
// Description: Provide non-overlapping signals to the capacitor switches to avoid
// turning on multiple switches at once
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module NonOverlapSampleB(sample, select, svin_g, svrefpB_g, svrefn_g, svcm_g,
    svin, svrefpB, svrefn, svcm);
 input [1:0] select; //Cap switch decision
input sample;    //Sample Mode signal (select Vin)
 input svrefpB_g;  //Select Vrefp (after inverters)
 input svrefn_g; //Select Vrefn (after inverters)
 input svcm_g;   //Select Vcm (after inverters)
input svin_g;    //Select Vin (after inverters)
output svin;   //Select Vin (before inverters)
 output svrefpB; //Select Vrefp (before inverters)
 output svrefn;  //Select Vrefn (before inverters)
 output svcm;    //Select Vcm (before inverters)

reg svrefpB, svrefn, svcm, svin;

always @ (sample, select, svin_g, svrefpB_g, svrefn_g, svcm_g)
begin
 if (sample) //If it's time to sample a new Vin
 begin
   svin = 1; //Switch Cap to Vin (ignore race glitch here)
   svcm = 0; //Turn off all other sample switches
   svrefpB = 1;
   svrefn = 0;
 end
 else  //If not sampling Vin, then test select to determine which switch to use
 begin
   svin = 0;
   case (select)
     2'b11:  //Select Vcm, but only if the other sitches are done switching
     begin
       if (svrefpB_g && !svrefn_g && svin_g)
         svcm = 1;
       else
         svcm = 0;
```

```
56          svrefpB = 1;
            svrefn = 0;
58        end
          2'b01:  //Select Vrefp, but only if the other sitches are done switching
60        begin
              if (!svcm_g && !svrefn_g && svin_g)
62            svrefpB = 0;
            else
64            svrefpB = 1;
            svcm = 0;
66          svrefn = 0;
          end
68        2'b10:  //Select Vrefn, but only if the other sitches are done switching
          begin
70          if (!svcm_g && svrefpB_g && svin_g)
              svrefn = 1;
72          else
              svrefn = 0;
74          svcm = 0;
            svrefpB = 1;
76        end
          default:
78        begin
            svcm = 1;
80          svrefpB = 1;
            svrefn = 0;
82        end
        endcase //End of (select) case statement
84    end //End of else statement from if (sample)
    end //End of always block
86

88 endmodule
```

./source_code/NonOverlapNoSampleT.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    14:14:38 09/15/2008
// Design Name:
// Module Name:    NonOverlapV01
// Project Name:
// Target Devices:
// Tool versions:
// Description: Provide non-overlapping signals to the capacitor switches to avoid
// turning on multiple switches at once
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module NonOverlapNoSampleT(select, svrefpB_g, svrefn_g, svcm_g, svrefpB, svrefn, svcm);
    input [1:0] select; //Cap switch decision
    input svrefpB_g;  //Select Vrefp (after inverters)
    input svrefn_g; //Select Vrefn (after inverters)
    input svcm_g;   //Select Vcm (after inverters)
    output svrefpB; //Select Vrefp (before inverters)
    output svrefn;  //Select Vrefn (before inverters)
    output svcm;    //Select Vcm (before inverters)

    reg svrefpB, svrefn, svcm;

    always @ (select, svrefpB_g, svrefn_g, svcm_g)
    begin
     case (select)
       2'b11:  //Select Vcm, but only if the other sitches are done switching
       begin
         if (svrefpB_g && !svrefn_g)
           svcm = 1;
         else
           svcm = 0;
         svrefpB = 1;
         svrefn = 0;
       end
       2'b10:  //Select Vrefp, but only if the other sitches are done switching
       begin
         if (!svcm_g && !svrefn_g)
           svrefpB = 0;
         else
           svrefpB = 1;
         svcm = 0;
         svrefn = 0;
       end
       2'b01:  //Select Vrefn, but only if the other sitches are done switching
       begin
```

```
56          if (!svcm_g && svrefpB_g)
              svrefn = 1;
58          else
              svrefn = 0;
60          svcm = 0;
            svrefpB = 1;
62        end
          default:
64        begin
            svcm = 1;
66          svrefpB = 1;
            svrefn = 0;
68        end
        endcase
70    end

72
    endmodule
```

./source_code/NonOverlapNoSampleT.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    14:14:38 09/15/2008
// Design Name:
// Module Name:    NonOverlapV01
// Project Name:
// Target Devices:
// Tool versions:
// Description: Provide non-overlapping signals to the capacitor switches to avoid
// turning on multiple switches at once
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module NonOverlapNoSampleT(select, svrefpB_g, svrefn_g, svcm_g, svrefpB, svrefn, svcm);
    input [1:0] select; //Cap switch decision
    input svrefpB_g;  //Select Vrefp (after inverters)
    input svrefn_g; //Select Vrefn (after inverters)
    input svcm_g;   //Select Vcm (after inverters)
    output svrefpB; //Select Vrefp (before inverters)
    output svrefn;  //Select Vrefn (before inverters)
    output svcm;    //Select Vcm (before inverters)

  reg svrefpB, svrefn, svcm;

  always @ (select, svrefpB_g, svrefn_g, svcm_g)
    begin
     case (select)
       2'b11:  //Select Vcm, but only if the other sitches are done switching
       begin
         if (svrefpB_g && !svrefn_g)
           svcm = 1;
         else
           svcm = 0;
         svrefpB = 1;
         svrefn = 0;
       end
       2'b10:  //Select Vrefp, but only if the other sitches are done switching
       begin
         if (!svcm_g && !svrefn_g)
           svrefpB = 0;
         else
           svrefpB = 1;
         svcm = 0;
         svrefn = 0;
       end
       2'b01:  //Select Vrefn, but only if the other sitches are done switching
       begin
```

```
          if (!svcm_g && svrefpB_g)
57          svrefn = 1;
          else
59          svrefn = 0;
          svcm = 0;
61        svrefpB = 1;
        end
63      default:
        begin
65        svcm = 1;
          svrefpB = 1;
67        svrefn = 0;
        end
69    endcase
    end
71

73 endmodule
```

Bibliography

[1] J. M. D. Pereira, "The History and Technology of Oscilloscopes," *IEEE Instrumentation and Measurement Magazine*, vol. 9, pp. 27–35, November 2006.

[2] S. Rapuano, P. Daponte, E. Balestrieri, L. D. Vito, S. J. Tilden, S. Max, and J. Blair, "ADC Parameters and Characteristics," *IEEE Instrumentation and Measurement Magazine*, vol. 8, pp. 44–54, December 2005.

[3] R. Reeder, M. Looney, and J. Hand, "Pushing the State of the Art with Multichannel A/D Converters," *Analog Dialog*, vol. 39, no. 2, pp. 7–10, June 2005.

[4] M. Looney, "Advanced Digital Post-Processing Techniques Enhance Performance in Time-Interleaved ADC Systems," *Analog Dialog*, vol. 37, no. 8, August 2003.

[5] D. Johns and K. Martin, *Analog Integrated Circuit Design.* Hoboken, NJ: John Wiley & Sons, Inc., 1997.

[6] D. A. Rauth and V. T. Randal, "Analog to Digital Conversion," *IEEE Instrumentation and Measurement Magazine*, vol. 8, pp. 44–54, October 2005.

[7] W. Kester, "Which ADC Architecture Is Right for Your Application?" *Analog Dialog*, vol. 39, no. 2, pp. 11–18, June 2005.

[8] Y. Kuramochi, A. Matsuzawa, and M. Kawabata, "A 0.05-mm^2 110-μW 10-b Self-Calibrating Successive Approximation ADC Core in 0.18-μm CMOS," *IEEE Asian Solid-State Circuits Conference*, pp. 224–227, November 2007.

[9] J. McNeill, M. Coln, and B. Larivee, "A Split-ADC Architecture for a Deterministic Digital Background Calibration of a 16b 1MS/s ADC," *ISSCC Dig. Tech. Papers*, pp. 277–278, February 2005.

[10] J. McNeill, M. Coln, D. R. Brown, and B. Larivee, "Digital Background Calibration Algorithm for 'Split ADC' Architecture," *IEEE Transactions on Circuits Systems I*, vol. 56, no. 2, pp. 294–306, February 2005.

[11] C. Chan, "Applying the 'Split-ADC' Architecture to a 16 bit, 1 MS/s differential Successive Approximation Analog-to-Digital Converter," Master's thesis, Worcester Polytechnic Institute, 2008.

[12] A. B. Grebene, *Bipolar and MOS Analog Integrated Circuit Design.* Hoboken, NJ: Wiley - Interscience, 2003.

[13] I. Analog Devices, *Analog-Digital Conversion Handbook.* Englewood Cliifs, NJ: PTR Prentice Hall, 1986.

[14] A. Note, "Histogram Testing Determines DNL and INL Errors," June 2003.

[15] P. C.-W. Yu, "Low-Power Design Techniques for Pipelined Analog-to-Digital Converters," Ph.D. dissertation, Massachusetts Institute of Technology, 1996.

[16] C.-N. Yeh and Y.-T. Lai, "A novel flash analog-to-digital converter," *IEEE International Synposium of Circuits and Systems*, pp. 2250–2253, 2008.

[17] A. Kitagawa, M. Kokubo, T. Tsukada, T. Matsuura, M. Hotta, K. Maio, E. Yamamoto, and E. Imaizumi, "A 10b 3MSamplels CMOS Cyclic ADC," *ISSCC Dig. Tech. Papers*, pp. 280–281, February 1995.

[18] N. Kurosawa, H. Kobayashi, K. Maruyama, Maruyama, H. Sugawara, and K. Kobayashi, "Explicit Analysis of Channel Mismatch Effects in Time-Interleaved ADC Systems," *IEEE Transactions on Circuits and Systems I*, vol. 48, no. 3, pp. 261–271, March 2001.

[19] J. Elbornson, F. Gustafasson, and J. Elkund, "Analysis of Mismatch Effects in a Randomly Interleaved A/D Converter System," *IEEE Transactions on Circuits and Systems I*, vol. 52, pp. 465–476, March 2005.

[20] U. K. Moon and B. S. Song, "Background Digital Calibration Techniques for Pipelined ADC's," *IEEE Transactions on Circuits and Systems II*, vol. 44, pp. 102–109, February 1997.

[21] S. M. Jamal, D. Fu, P. J. Hurst, and S. H. Lewis, "A 10b 120MSample/s Time-Interleaved Analog-to-Digital Converter with Digital Background Calibration," *ISSCC Dig. Tech. Papers*, pp. 1912–1919, February 2002.

[22] Y. Chiu, C. W. Tsang, B. Nikolić, and P. R. Gray, "Least Mean Square Adaptive Digital Background Calibration of Pipelined Analog-to-Digital Converters," *IEEE Transactions on Circuits and Systems I*, vol. 51, pp. 38–40, January 2004.

[23] E. Iroaga, B. Murmann, and L. Nathawad, "A Background Correction Technique for Timing Errors in Time-Interleaved Analog-to-Digital Converters," *IEEE Symposium on Circuits and Systems*, vol. 6, pp. 5557–5560, May 2005.

[24] E. Alpman, H. Lakdawala, L. R. Carley, and K. Soumyanath, "A 1.1V 50mW 2.5GS/s 7b Time-Interleaved C-2C SAR ADC in 45nm LP Digital CMOS," *ISSCC Dig. Tech. Papers*, pp. 76–77, February 2009.

[25] S. Gupta, M. Choi, M. Inerfield, and J. Wang, "A 1GS/s 11b Time-Interleaved ADC in 0.13μm CMOS," *ISSCC Dig. Tech. Papers*, pp. 2360–2369, February 2006.

[26] B. P. Ginsburg and A. P. Chandrakasan, "Highly Interleaved 5b 250 MS/s ADC with Redundant Channels in 65nm CMOS," *ISSCC Dig. Tech. Papers*, pp. 240–241, February 2008.

[27] S. Limotyrakis, S. D. Kulchycki, D. Su, and B. A. Wooley, "A 150MSample/s 8b 71mW Time-Interleaved ADC in 0.18μm CMOS," *ISSCC Dig. Tech. Papers*, pp. 132–133, February 2004.

[28] I. Galton, "Digital Cancellation of D/A Converter Noise in Pipelined A/D," *IEEE Transactions on Circuits and Systems II*, vol. 51, pp. 185–196, March 2000.

[29] H. L. et al., "A 15b 20MS/s CMOS Pipelined ADC with Digital Background Calibration," *ISSCC Dig. Tech. Papers*, pp. 454–455, February 2004.

[30] K. Nair and R. Harjani, "A 96dB SFDR 50MS/s Digitally Enhanced CMOS Pipelined A/D Converter," *ISSCC Dig. Tech. Papers*, pp. 456–457, February 2004.

[31] S. R. et al., "A 14b-Linear Capacitor Self-Trimming Pipelined ADC," *ISSCC Dig. Tech. Papers*, pp. 464–465, February 2004.

[32] D. Chen, Z. Yu, and R. Geiger, "An Adaptive, Truly Background Calibration Method for High Speed Pipeline ADC Design," *IEEE Symposium on Circuits and Systems*, vol. 6, pp. 6190–6193, May 2005.

[33] K. C. Dyer, D. Fu, P. J. Hurst, and S. H. Lewis, "An Analog Background Calibration Technique for Time-Interleaved Analog-to-Digital Converters," *IEEE Journal Of Solid-State Circuits*, vol. 33, pp. 1912–1919, December 1998.

[34] W. Liu, Y. Chang, S.-K. Hsien, B.-W. Chen, Y.-P. Lee, wen Tsao Chen, T.-Y. Yang, G.-K. Ma, and Y. Chiu, "An 600MS/s 30mW 0.013μm CMOS SAR ADC Array Achieving Over 60dB SFDR with Adaptive Digital Equalization," *ISSCC Dig. Tech. Papers*, pp. 82–83, February 2009.

[35] Y. Chen, T. Kuroda, and et. al., "Split Capacitor DAC Mismatch Calibration in Successive Approximation ADC," *IEEE Custom Integrated Circuits Conference*, pp. 279–282, 2009.

[36] J. Gan and J. Abraham, "A Non-Binary Array Calibration Circuit with 22-bit Accuracy in Successive Approximation Analog-to-Digital Converters," *IEEE Circuits and Systems*, vol. 1, pp. 567–570, August 2002.

[37] F. Kuttner, "A 1.2V 10b 20MSample/s Non-Binary Successive Approximation ADC in 0.13μm CMOS," *ISSCC Dig. Tech. Papers*, pp. 136–137, February 2002.

[38] M. Hesener, T. Eichler, A. Hanneberg, D. Herbison, and F. Kuttner, "An 14b 40MS/s Redundant SAR ADC with 480 MHz Clock in 0.13μm CMOS," *ISSCC Dig. Tech. Papers*, pp. 248–249, February 2007.

[39] J. Ingino and B. Wooley, "A Continuously Calibrated 12-b, 10-MS/s, 3.3-V A/D Converter," *IEEE Journal Of Solid-State Circuits*, vol. 33, pp. 1920–1931, December 1998.

[40] K. Tan and et. al., "Error Correction Techniques for High-Performance Differential A/D Converters," *IEEE Journal Of Solid-State Circuits*, vol. 25, pp. 1318–1327, December 1990.

[41] W. Liu and Y. Chiu, "An Equalization-Based Adaptive Digital Background Calibration Technique for Successive Approximation Analog-to-Digital Converters," *IEEE ASICON*, pp. 289–292, October 2007.

[42] W. Liu, P. Huang, and Y. Chiu, "An 12b 22.5/45MS/s 3.0mW 0.059mm^2 CMOS SAR ADC Achieving Over 90dB SFDR," *ISSCC Dig. Tech. Papers*, pp. 380–381, February 2010.

[43] J. C. Strikwerda, *Finite Difference Schemes and Partial Differential Equations*. Pacific Grove, CA: Wadsworth & Brooks, 1989.

[44] J. McNeill, C. David, M. Coln, and R. Croughwell, "'Split-ADC' Calibration for All-Digital Correction of Time-Interleaved ADC Errors," *IEEE Transactions on Circuits Systems II - Express Briefs*, vol. 56, no. 5, pp. 344–348, May 2009.

[45] R. Croughwell, "A 16-b 10Msample/s Split-Interleaved Analog to Digital Converter," Master's thesis, Worcester Polytechnic Institute, 2007.