Applying the "Split-ADC" Architecture to a 16 bit, 1MS/s differential Successive Approximation Analog-to-Digital Converter

by

Chilann, Ka Yan Chan

A Thesis

Submitted to the Faulty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the

Degree of Master of Science

in

Electrical Engineering

April 2008

Approved:

_____

Professor John A. McNeill

Thesis Advisor

_____ _____

Professor Andrew G. Klein

Committee Member

Professor Stephen J. Bitar

Committee Member

Abstract

Successive Approximation (SAR) analog-to-digital converters are used extensively in biomedical applications such as CAT scan due to the high resolution they offer. Capacitor mismatch in the SAR converter is a limiting factor for its accuracy and resolution. Without some form of calibration, a SAR converter can only achieve 10 bit accuracy. In industry, the CAL-DAC approach is a popular approach for calibrating the SAR ADC, but this approach requires significant test time.

This thesis applies the "Split-ADC" architecture with a deterministic, digital, and background self-calibration algorithm to the SAR converter to minimize test time. In this approach, a single ADC is split into two independent halves. The two split ADCs convert the same input sample and produce two output codes. The ADC output is the average of these two output codes. The difference between these two codes is used as a calibration signal to estimate the errors of the calibration parameters in a modified Jacobi method. The estimates are used to update calibration parameters are updated in a negative feedback LMS procedure. The ADC is fully calibrated when the difference signal goes to zero on average.

This thesis focuses on the specific implementation of the "Split-ADC" self-calibrating algorithm on a 16 bit, 1 MS/s differential SAR ADC. The ADC can be calibrated with 10^5 conversions. This represents an improvement of 3 orders of magnitude over existing statistically-based calibration algorithms. Simulation results show that the linearity of the calibrated ADC improves to within ±1 LSB.

Acknowledgements

It has been a great honor for me to be a student of Professor John A. McNeill for the past two years. Over this period of time, I have learned a lot in the area of analog integrated circuit design, and have realized I have a lot yet to learn from this very complex field.

Professor McNeill has been a great advisor. His useful and insightful technical advices have helped me to get through the most difficult phase of this project. I would also like to thank him for making my master experience an enjoyable one. Thank you for widening our eyesight by bringing all of us to the International Solid State Circuit Conference (ISSCC), and diversifying our international food experience with the excellent sushi dinners.

Thanks also to my labmates Chris David, Tsai Chen, Hattie Spetla, Ali Ulas Ilhan for being the good friends they have always been. I would especially like to thank Chris David for sharing his project experience on applying the “Split-ADC” architecture to a 16 bit, 12MS/s interleaved ADC.

I would also like to thank Michael Coln and his group at Analog Devices Inc. for helping with the IC development for this project. They have helped extensively on the SAR converter design, especially on the design of the sample-and-hold circuit and the DAC structure. Without their industrial experience and patient guidance, this project would not have been possible. Thanks also to the National Science Foundation (NSF) for sponsoring this project.

Last, but by all means, not least, I want to thank my family. Thank to my mum for her mental support. Her delicious Cantonese soups have always stayed in my heart and keep me going. Thank to my dad for understanding and supporting my decision of doing a MS. Lastly, thank to all my siblings for being supportive.

Tables of Contents

List of Figures and Tables

1

Introduction

Traditionally, Successive Approximation (SAR) Analog-to-digital converters suffer from the finite matching accuracy of capacitors [1]. Laser-trimming or some forms of calibration are necessary, if ADC's resolution and accuracy of 16 bit and higher has to be achieved. A less expensive way to correct for capacitor mismatch is to develop self-calibration algorithms to be used with the SAR. In 1984, H.S. Lee developed a one-time self-calibrating technique which can be applied to the all MOS charge redistribution SAR A/D converter. He measured and stored the ratio errors of the capacitors on a RAM during the calibration cycle. During the subsequent normal conversion cycle, this information is used with a calibration DAC to correct for the mismatch errors [2]. This method proved to be an effective way to improve the SAR resolution, and many recent subsequent works are based on his approach [3-6]. It is, however, time-consuming to measure and write these capacitor ratio errors into the RAM. In fact, this is a limiting factor in production cost in industry.

Thus, the goal of this work is to develop a self-calibrating algorithm based on the "Split-ADC" architecture introduced in [7, 8] for the SAR ADC to minimize test time. While other self-calibrating algorithms exist to calibrate the SAR ADC, they do so at the expense of increasing the foreground ADC conversion time, or increasing the analog complexity. For instance, [9] makes use of a non-binary capacitor array with the perceptron learning rule to calibrate capacitor weights in the DAC. This digital error correction procedure, however, cannot be used in the background. [10] used an equalization-based adaptive digital background calibration technique to correct for the capacitor ratio mismatch. This approach, however, requires an additional slow-but-accurate reference ADC to compare the error with the filter output.

Self-calibrating techniques of other ADCs have also been investigated. Upon comparison, the split ADC architecture developed in [7, 8] still offers the best solution in calibrating the SAR ADC. The self-calibrating algorithm in this work can be implemented entirely in the digital domain, with no added analog complexity. It operates in the background, and therefore does not affect the speed of foreground operation. Finally, it uses a deterministic approach to track out parameter variations due to environmental influences. Thus, it can calibrate the ADC with a much shorter time, compared to the traditional statistical method. Table 1.1 compares this work with other self-calibrating techniques.

Table 1.1 Comparison with previous work

	SAR specific		Interleaved specific			Pipelined specific			This work
	[9]	[10]	[11]	[12]-[15]	[16]-[20]	[21]-[24]	[25],[26]	[27],[28]	[7, 8]
Deterministic?	√	√					√	√	√
(All)Digital?	√		√		√	√		√	√
Background?		√		√	√	√	√		√

In [7,8], it was shown that the split ADC concept can be applied to the cyclic ADC, with the op-amp gain being the only calibration parameter. In a SAR converter, however, we have significantly more calibration parameters. Thus, the primary challenge of this work is to develop a self-calibrating algorithm to work with the "Split- SAR ADC" architecture that can calibrate many parameters at the same time.

This thesis is organized as follows: Chapter 2 explains the conventional SAR architecture and the modifications we made in the design of the ADC in order to apply the self-calibrating algorithm. Chapter 3 explains how the split SAR architecture works together with the self-calibration algorithm to correct for the calibration parameters at a system level. The whole system is then verified in Matlab. Chapter 4 provides in-depth analyses for the error correction algorithm used in the self-calibrating procedure. Chapter 5 discusses the details of some of the circuit design work necessary to develop the prototype 16 bit, 1 MS/s differential SAR IC in the 0.25um standard CMOS process. Finally, Chapter 6 concludes this thesis and suggests possible future research based on this work.

2

SAR architecture

2.1 Introduction

Successive Approximation A/D converters are one of the most popular approaches for realizing A/D converters. This is because they have relatively quick conversion time, yet moderate circuit complexity [29].

One of the most common architectures used to realize a successive approximation A/D converter is the charge redistribution MOS A/D converter developed by McCreary [30]. With this converter, the sample, hold and bit cycling actions of the ADC can be realized in a single circuit. The use of the binary weighted switched capacitor DAC array also provides better accuracy and linearity than its resistive counterpart [31].

In Section 2.2, we will use the charge redistribution MOS A/D converter to explain the basic operation of a 4-bit differential SAR A/D converter. In Section 2.3, we will review the split ADC architecture. In Section 2.4, we will develop an ideal 16 bit, 1MS/s differential SAR ADC to be used with the split architecture and the error correction algorithm. Simulation results will be provided to show that the proposed design works.

2.2 SAR ADC Review

The top level block diagram of a differential SAR converter is shown in Figure 2.1. It consists of two sample-and-hold circuits(S/H), two DACs, a latched comparator and the necessary SAR logic to update the output of the DAC (Vx and Vy). Vin1, Vin2 are the differential input voltages, and it will be replaced by Vinp and Vinm later.

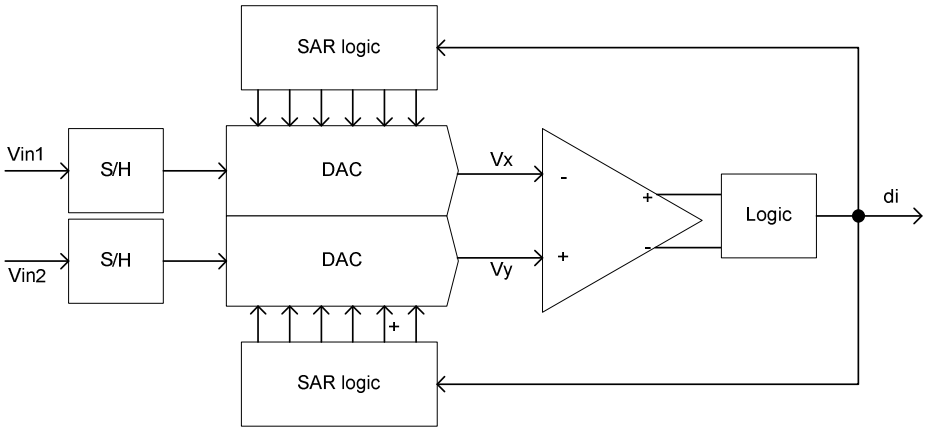


Figure 2.1: Top level block diagram of a differential SAR converter

Figure 2.2 shows the algorithm used to implement the SAR logic. At each conversion, Vin1 and Vin2 are sampled and held onto the DAC . When the bit cycling action starts, the comparator compares Vx and Vy. If Vy is bigger than Vx, the comparator outputs a decision bit 1. Otherwise, the comparator outputs a decision bit -1. Vx and Vy are then updated by (2.1a) and (2.1b). The decision in each cycle can be determined by (2.1c).

$$Vy_i = Vy_{i-1} - \frac{di \cdot (Vref - Vcm)}{2^i} \text{ (2.1a)}$$

$$Vx_i = Vx_{i-1} + \frac{di \cdot (Vref - Vcm)}{2^i} \text{(2.1b)}$$

$$d_i = sign(Vy_{i-1} - Vx_{i-1}) \quad (2.1c)$$

This process is repeated N times for an N bit converter. Eventually, Vx and Vy are driven to within 1/2 LSB of each other.

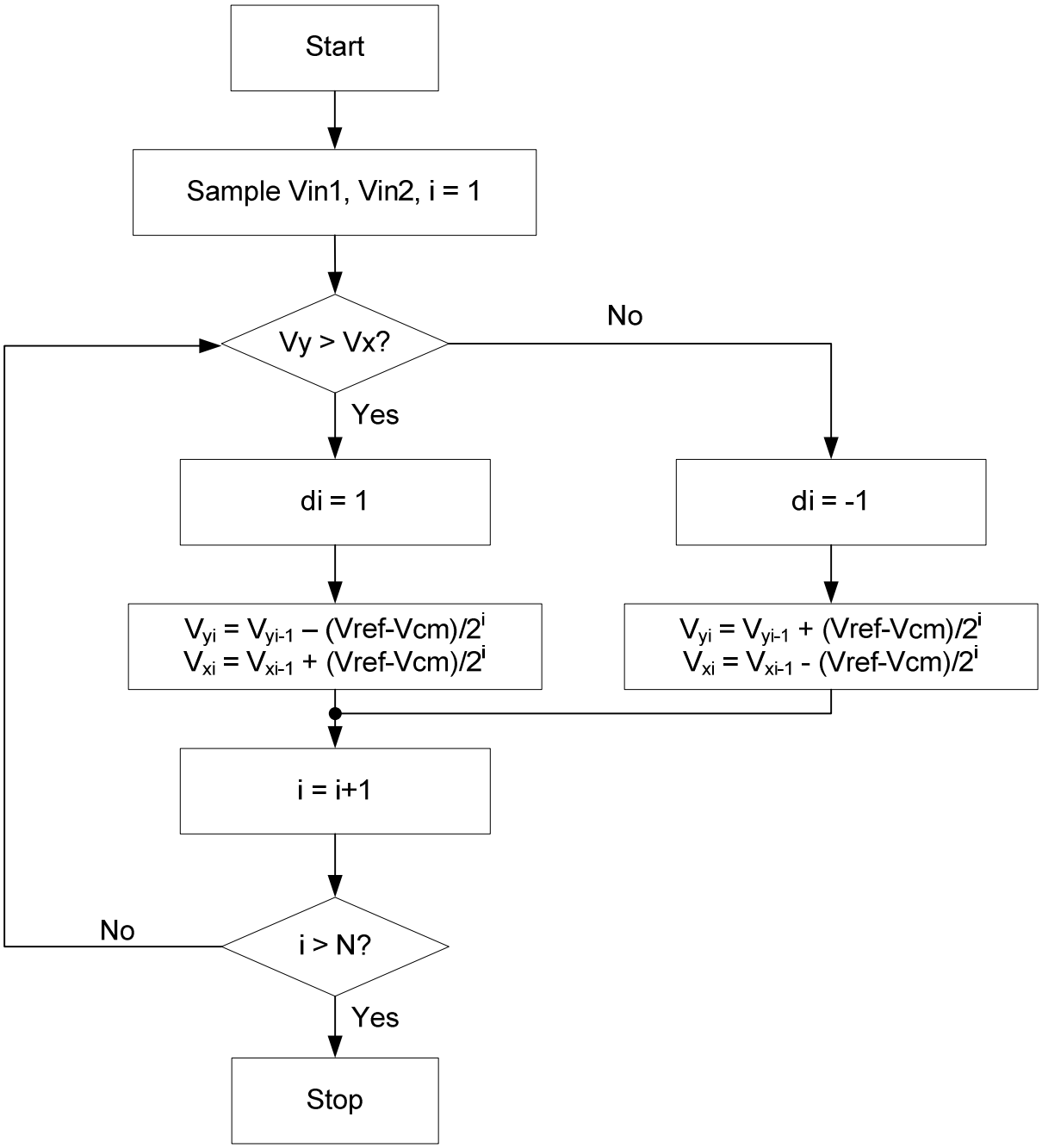


Figure 2.2: Updating the DAC voltage Vx and Vy with the SAR logic

Figure 2.3 shows the sample, hold and bit cycling operations of the differential SAR converter at a circuit level. We demonstrated these operations with a 4 bit converter, where the common mode voltage Vcm is 1.25V, and the positive voltage reference Vp and the negative voltage reference Vm are 2.5V and 0V respectively. The input voltage range for each input is 0V to 2.5V, and this is determined by the 0.25 um process supply. Therefore, the differential input voltage range is ±2.5V. From (3.2), the smallest voltage that the 4 bit converter can resolve is 0.3125V. Note that the comparator is only active during the bit cycling mode.

In the sampling mode, the inputs are being sampled onto all the capacitors. The switches S1 are closed, and S2 and S3 are switched in such a way to sample Vin1 and Vin2. At this instant, the top plate voltages Vx and Vy are both equal to 1.25V (Figure 2.3a).

During the hold mode, S1 and S3 are opened, and all S2 are switched to Vcm (Figure 2.3b). Because of the law of charge conservation, the charges sampled onto the capacitors during the sample mode have to equal to the charges held onto the capacitors during the hold mode. Therefore, Vx and Vy are now equal to

$$Vx = 2.5 - Vin1 \quad (2.2a)$$

$$Vy = 2.5 - Vin2 \quad (2.2b)$$

Next, the bit cycling operation starts (Figure 2.3c). The comparator first compares Vx and Vy. If Vy is bigger than Vx, then the comparator outputs a decision bit 1. Otherwise, the comparator outputs a decision bit -1. Starting with the most significant bit, if the decision is 1, the switches S2 attaching the capacitors labeled 8C will switch to the right. At the top DAC, S3 switches to Vp. At the bottom DAC, S3 switches to Vm. As a result, Vy is reduced by the first binary weight, and Vx is increased by the first binary weight in the first cycle. The opposite happens if the decision bit is -1.

Since 8C is half the total capacitance of the DAC, Vx and Vy are increased/ reduced by (2.5-1.25)/2 = 0.625V in the first bit cycle (2.1). This process is repeated 3 more times, with Vx and Vy being increased/ reduced by half of the previous binary weight each time. Eventually, Vx and Vy converge to within 1/2 LSB of each other.

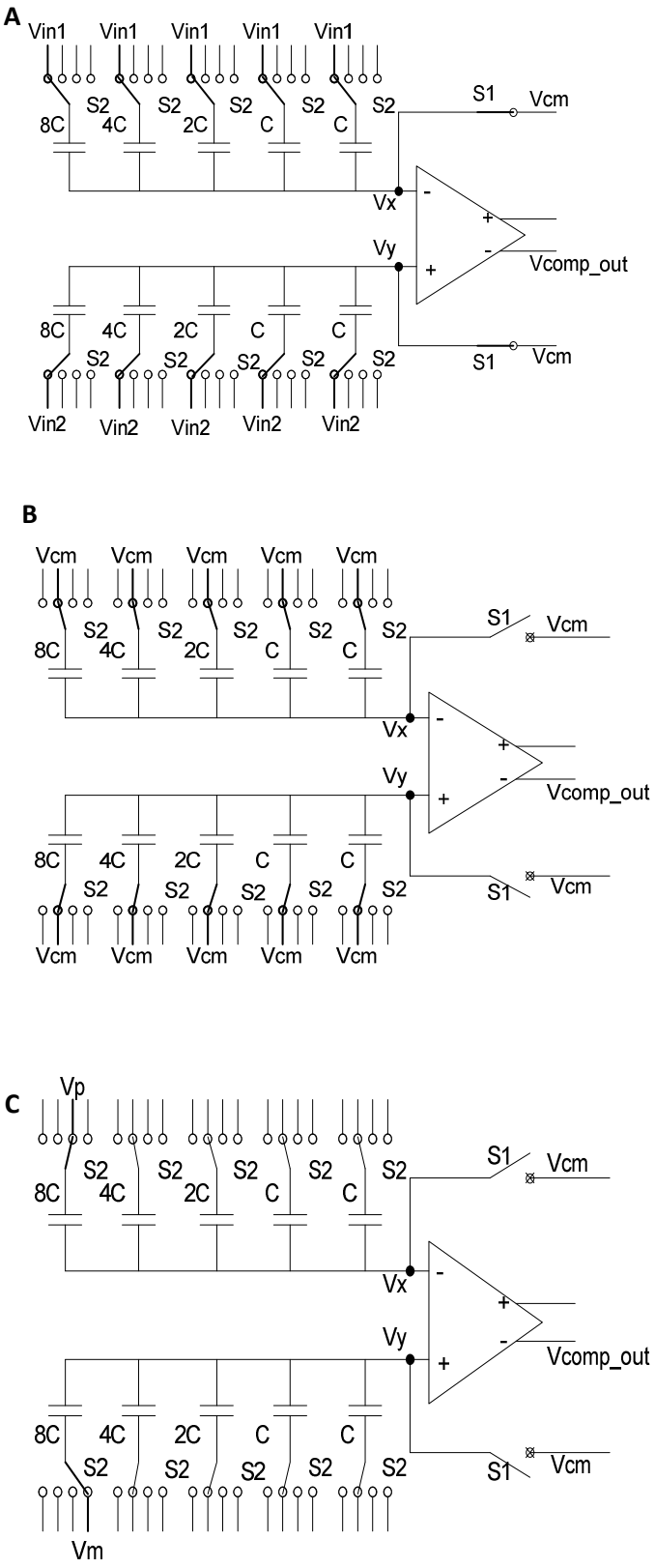


Figure 2.3 Sample, Hold and Bit Cylcing mode

(A) During the sample mode, the inputs are being sampled onto all the capacitors. The switches S1 are closed, and S2 and S3 are switched in such a way to sample Vin1 and Vin2.

(B) During the hold mode, S1 and S3 are opened, and all S2 are switched to Vcm . Because of the law of charge conservation, the charges sampled onto the capacitors during the sample mode have to equal to the charges held onto the capacitors during the hold mode.

(C) During the bit cycling action, the comparator compares Vx and Vy. If Vy is bigger than Vx, then the comparator outputs a decision bit 1. Otherwise, the comparator outputs a decision bit -1. Starting with the most significant bit, if the decision is 1, the switches S2 attaching the capacitors labeled 8C will switch to the right. At the top DAC, S3 switches to Vp. At the bottom DAC, S3 switches to Vm. As a result, Vx is increased by the first binary weight, and Vy is reduced by the first binary weight in the first cycle. The opposite happens if the decision bit is -1.

This process is repeated 3 other times, with Vx and Vy being increased/ reduced by half of the previous binary weight each time. Vx and Vy eventually converge to within ½ LSB of each other.

As an example, suppose Vin1 = 2.5V and Vin2 = 0V. Table 2.1 shows the predicted value of Vx and Vy in the hold mode and bit cycling mode, and the predicted decision output at each cycle using (2.1) and (2.2).

Table 2.1 The predicted value of Vx and Vy, and the output decision during the hold mode and the bit cycling mode

	Vx(V)	**Vy(V)**	**d**
Hold	0	2.5	1
1st cycle	0.625	1.875	1
2nd cycle	0.9375	1.5625	1
3rd cycle	1.09375	1.40625	1
4th cycle	1.171875	1.328125	

The result in Table 2.1 is verified against simulation. Figure 2.4 shows the simulated waveform of Vx and Vy for one conversion. Note that Vx and Vy is driven towards the common mode voltage 1.25 at the end, and that Vx and Vy is within ½ LSB of each other. In general, we expect the DAC voltages Vx and Vy of a N-bit converter possesses a similar waveform.

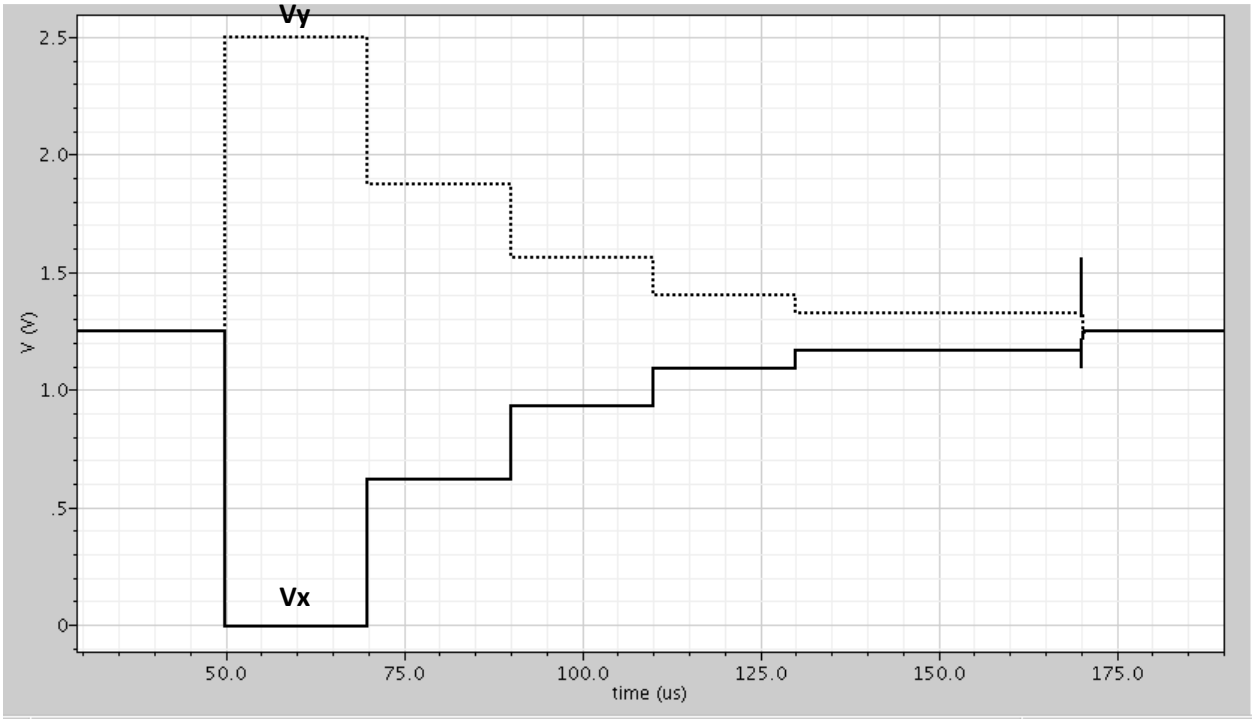


Figure 2.4: Waveform of Vx and Vy for one conversion

2.3 Split SAR ADC architecture

2.3.1 Split ADC Concept Review

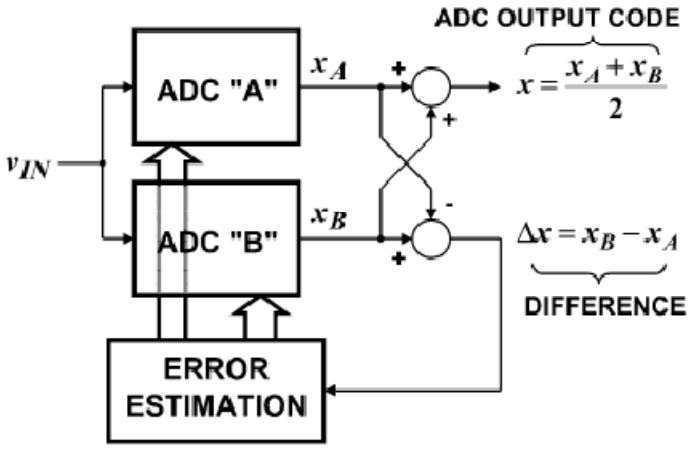


Figure 2.5: "Split ADC" architecture.

The "Split ADC" architecture developed in [7,8] is presented in Fig. 2.5. The "Split ADC" architecture uses two independent and identical ADCs to sample the same input, Vin. Each ADC converts Vin and produces individual outputs x_A and x_B.

On the foreground ADC operation, the average of x_A and x_B , x, is used as the output code of the ADC. In the background self-calibration procedure, the difference between the two output codes, Δx, is used in an error estimation process to adjust the calibration parameters. As Δx is driven to zero, the calibration parameters converge to their correct values. As a result, the output code x also converges to the correct value.

In our case, we use a differential SAR converter as our principal ADC. Since we want to make sure the only way for Δx to be zero is when the calibration parameters are adjusted correctly, we want to ensure the decision paths to arrive at the same output code for ADCA and ADCB are different. In other words, the decisions to reach x_A and x_B should be independent of each other. To do this, we insert redundant bits and use a random process to select the DAC cap segments during the bit cycling process. An added advantage of using randomization is that we can use any input signal for calibration, even for a constant DC signal. Meanwhile, an added advantage of using redundant bits is that it gives the system a second chance to "correct" any previous erroneous decisions, due to the presence of the comparator noise.

The "Spilt ADC" calibration technique can be used for any type of converters. It supports an all-digital, deterministic, background self-calibration procedure. In addition, it has a speed and cost benefit, and a negligible impact on analog complexity and analog die area.

2.3.2 Speed Benefit of the "Split" ADC architecture

The use of Δx in correction enables the subtraction of a large input signal magnitude in the self-calibrating procedure. This in turns provides fast convergence for the calibration parameters as compared to the statistical method such as [21], where the convergence speed in some cases depends upon on the input signal magnitude.

2.3.3 Cost benefit

The preferred tradeoff in submicron CMOS is to move the circuit complexity from the analog domain to digital domain. Since our self-calibrating algorithm can be implemented entirely in the digital domain, we are able to take advantage of CMOS scaling and save on fabrication cost.

2.3.4 Implication for analog complexity

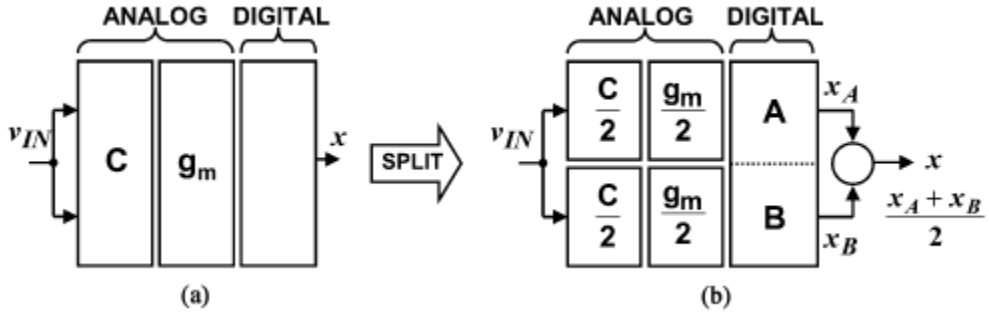


Figure 2.6: Splitting a single ADC into two

Since the "spilt ADC" architecture simply splits the analog area of a single ADC into two independent halves, it is able to maintain the overall area, power, bandwidth and noise performance [7]. Figure 2.6 shows how we split a single ADC into two independent halves. Let's gm represents the area of the active analog circuitries, and C represents the area of passive components such as capacitors and resistors. If we assume that bandwidth f_T is proportional to gm/C, then the bandwidth of one split ADC will be

$$f_t \propto \frac{g_m}{2} \cdot \frac{2}{C} \propto \frac{g_m}{C} \qquad (2.3)$$

Thus, the bandwidth of the split ADC is the same with the single ADC.

If we assume the power P is proportional to gm, then the power of one split ADC is

$$P \propto \frac{g_m}{2} \quad (2.4a)$$

Therefore, the overall power P of the split structure is proportional to gm, where

$$P_{Total} = \frac{g_m}{2}+\frac{g_m}{2} = gm \quad (2.4b)$$

Finally, if we assume the noise N is proportional to $\sqrt{kT/C}$, then the noise of one split ADC is

$$N \propto \sqrt{\frac{kT}{C}} \cdot \sqrt{2} \quad (2.5)$$

However, since the output code is averaged from two ADCs, the overall noise is proportional to $\sqrt{kT/C}$. Thus, the overall bandwidth, power and noise of the split ADCs are the same as a single ADC.

2.4 Modifications of the differential SAR ADC

2.4.1 The new DAC structure

As mentioned in Section 2.3.1, since the split ADC architecture self-calibrating approach requires randomization and the use of redundant bits, one cannot simply extend the basic 4-bit differential SAR structure in Section 2.2 into the required differential 16-bit differential SAR structure.

Based on the SNR and die area constraints, we come up with the proposed DAC structure in Figure 2.7, where the unit capacitor in segment 1 is 1pF, and the unit capacitor in segment 2-5 is 125 fF. The values of the coupling capacitors Cc1, Cc2 and Cc3 are 325.89fF, 321.4fF and 285.71 fF respectively. Chapter 5 shows the detailed design procedure to obtain these values.

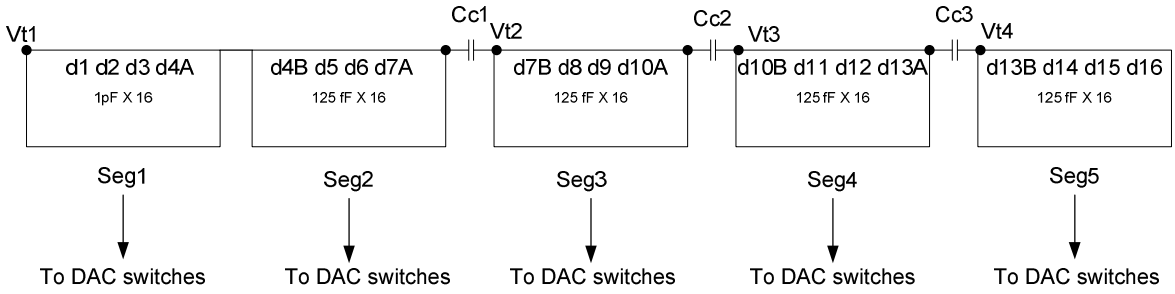


Figure 2.7 Modified DAC structure for the 16-bit differential SAR converter

There are 5 segments in this DAC, and each segment consists of 16 unit capacitors. Cc1, Cc2 and Cc3 are chosen such that the total capacitance of the left segment is 8 times the total capacitance of the right segment. In other words, the MSB of the right segment is equal to the LSB of the left segment. Therefore, each segment is in charge of making 4 decisions, and we have 4 redundant bits.

The use of unit capacitors in each segment allows us to select capacitors randomly to represent the DAC weights. In our randomization process, we generate a random base for each segment so

that we know which capacitor is responsible for making which bit of decision. For example, if the base number generated is 3 for segment 1, then capacitor 3-10 will be used for d1, capacitor 11-14 will be used for d2, capacitor 15-16 will be used for d3, capacitor 1 will be used for d4A and capacitor 2 will be left unused.

2.4.2 DAC weights

To develop an ideal 16-bit differential SAR converter, we need to know the DAC weights corresponding to each bit. We can find out the ideal DAC weights corresponding to each individual capacitor in each segment with the design equations in Section 5.4.2. However, this approach is tedious. Also, if mismatch are present, modifying these equations to find out individual DAC weights become formidable, since each capacitor interact with one another in this network.

To simplify the procedure of finding the DAC weights, we develop the simulation in Figure 2.8. Turning on the positive or negative reference in the DAC correspond to supplying or pulling out 1.25V from the individual capacitor, since the common mode voltage is 1.25V. As an example to find out the ideal DAC weight corresponding to each capacitor (C1-C80), we supply a square wave of 1.25V in magnitude to individual capacitors and see how they change Vtop.

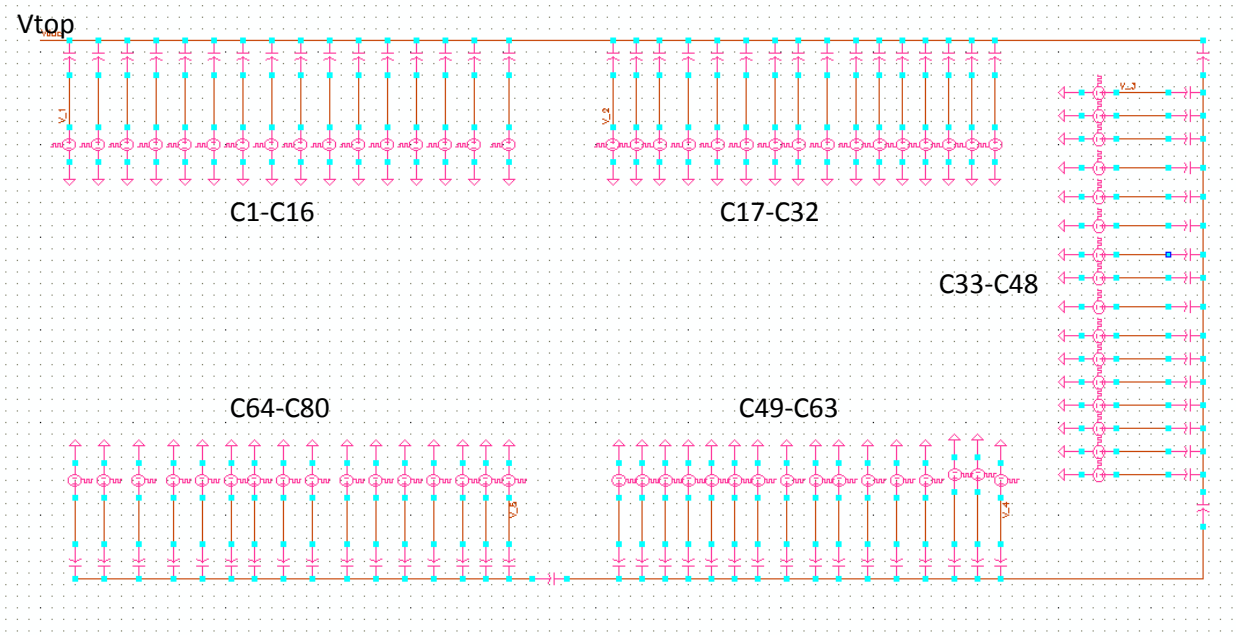


Figure 2.8: Relating capacitor mismatch with DAC weights

Table 2.2 shows the ideal voltage weight each capacitor represents.

Table 2.2 Ideal DAC weights of all capacitors

	Ideal		**Ideal**		**Ideal**
C	Voltage weight	C	Voltage weight	C	Voltage weight
1	6.8361460E-02	31	8.5451830E-03	61	1.3350820E-04
2	6.8361460E-02	32	8.5451830E-03	62	1.3350820E-04
3	6.8361460E-02	33	1.0681480E-03	63	1.3350820E-04
4	6.8361460E-02	34	1.0681480E-03	64	1.3350820E-04
5	6.8361460E-02	35	1.0681480E-03	65	1.6688290E-05
6	6.8361460E-02	36	1.0681480E-03	66	1.6688290E-05
7	6.8361460E-02	37	1.0681480E-03	67	1.6688290E-05
8	6.8361460E-02	38	1.0681480E-03	68	1.6688290E-05
9	6.8361460E-02	39	1.0681480E-03	69	1.6688290E-05
10	6.8361460E-02	40	1.0681480E-03	70	1.6688290E-05
11	6.8361460E-02	41	1.0681480E-03	71	1.6688290E-05
12	6.8361460E-02	42	1.0681480E-03	72	1.6688290E-05
13	6.8361460E-02	43	1.0681480E-03	73	1.6688290E-05
14	6.8361460E-02	44	1.0681480E-03	74	1.6688290E-05
15	6.8361460E-02	45	1.0681480E-03	75	1.6688290E-05
16	6.8361460E-02	46	1.0681480E-03	76	1.6688290E-05
17	8.5451830E-03	47	1.0681480E-03	77	1.6688290E-05
18	8.5451830E-03	48	1.0681480E-03	78	1.6688290E-05
19	8.5451830E-03	49	1.3350820E-04	79	1.6688290E-05
20	8.5451830E-03	50	1.3350820E-04	80	1.6688290E-05
21	8.5451830E-03	51	1.3350820E-04		
22	8.5451830E-03	52	1.3350820E-04		
23	8.5451830E-03	53	1.3350820E-04		
24	8.5451830E-03	54	1.3350820E-04		
25	8.5451830E-03	55	1.3350820E-04		
26	8.5451830E-03	56	1.3350820E-04		
27	8.5451830E-03	57	1.3350820E-04		
28	8.5451830E-03	58	1.3350820E-04		
29	8.5451830E-03	59	1.3350820E-04		
30	8.5451830E-03	60	1.3350820E-04		

2.5 Building an ideal 16-bit 1Ms/s Successive Approximation A/D converter

The operation of a 16-bit differential SAR ADC, with the proposed DAC in Section 2.4.1 implemented, is very similar to the 4-bit differential SAR ADC in Section 2.2, except for 3 differences.

In the 16-bit differential SAR ADC, we only use the first segment for sampling. In addition, during the bit cycling mode, we employed the randomization scheme discussed in Section 2.4.1 to select the DAC weights in each segment. Finally, this ADC can resolve two input voltages within 1 LSB at the 16 bit level, where 1 LSB equals 76 uV.

Figure 2.9 shows the basic DAC cell used in the simulation. The capacitor C_H is the unit capacitor used in a particular segment. If C_H is located in segment 1, it has the value of 1 pF. Otherwise, it has the value of 125 fF. The MOS switches M2 and M3 are used to sample the input voltages. The MOS switch M4 is used to turn on/off the common mode voltage Vcm and is used during the hold mode. The MOS switches M5 and M6 are used to turn on/off the positive reference Vrefp and the negative reference Vrefm respectively during the bit cycling mode. In this DAC cell, every component is ideal. The MOS switches are replaced by the VerilogA ideal switches in Appendix A.

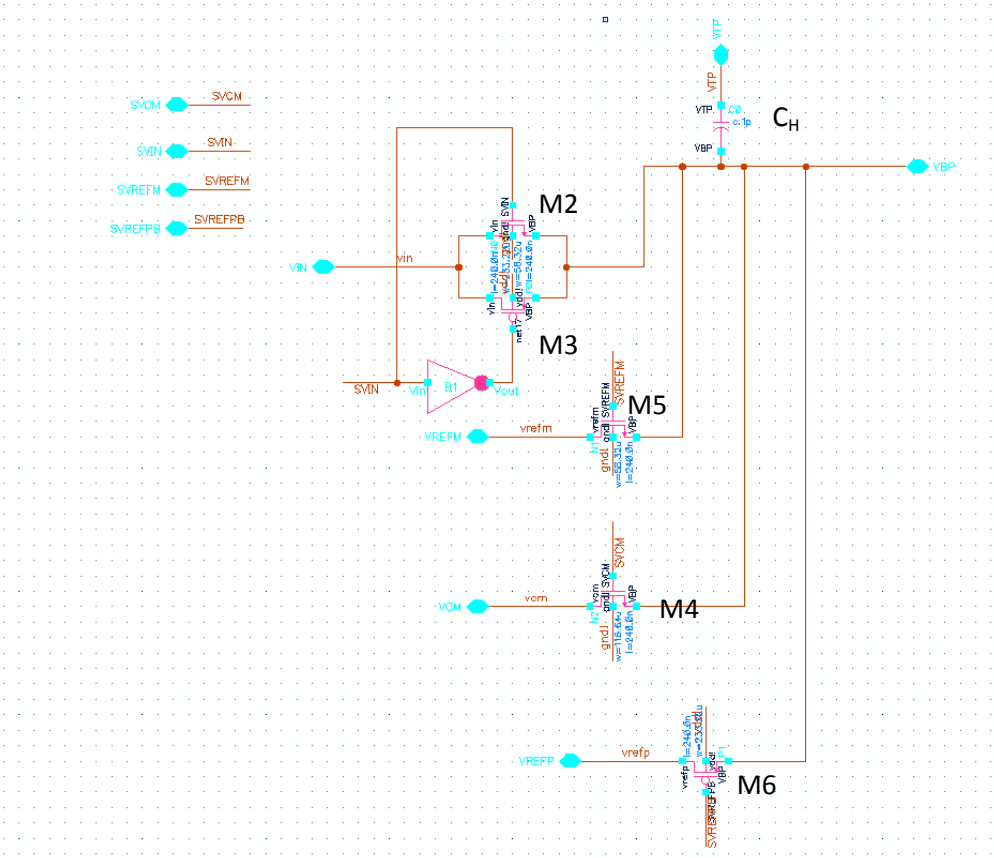


Figure 2.9 Basic DAC cell unit

We connected 16 of these basic cells together to form one segment. They are labeled seg1-seg5 in Figure 2.10.

We used VerilogA to implement random number generators, and the SAR logic necessary to perform the sample, hold and bit cycling action. Based on the random number generated, the DAC switch selector selects the appropriate DAC switches to be used during the sample, hold and bit cycling action. Appendix A details all the cell hierarchy used in the design, and the VerilogA design of each module.

A clock cycle of 40 ns is used in this simulation. For 1 conversion, we will use the first 200 ns to acquire the input voltage, and the remaining 800ns for bit cycling. We supplied an increasing ramp input at Vin1 ranges from 0V to 2.5V, while supplying a decreasing ramp input at Vin2 ranges from 2.5V to 0V. 1000 conversions were made from this differential input, where the sampled voltages range from -2.5V to 2.495V, with an increasing differential step of 5mV.

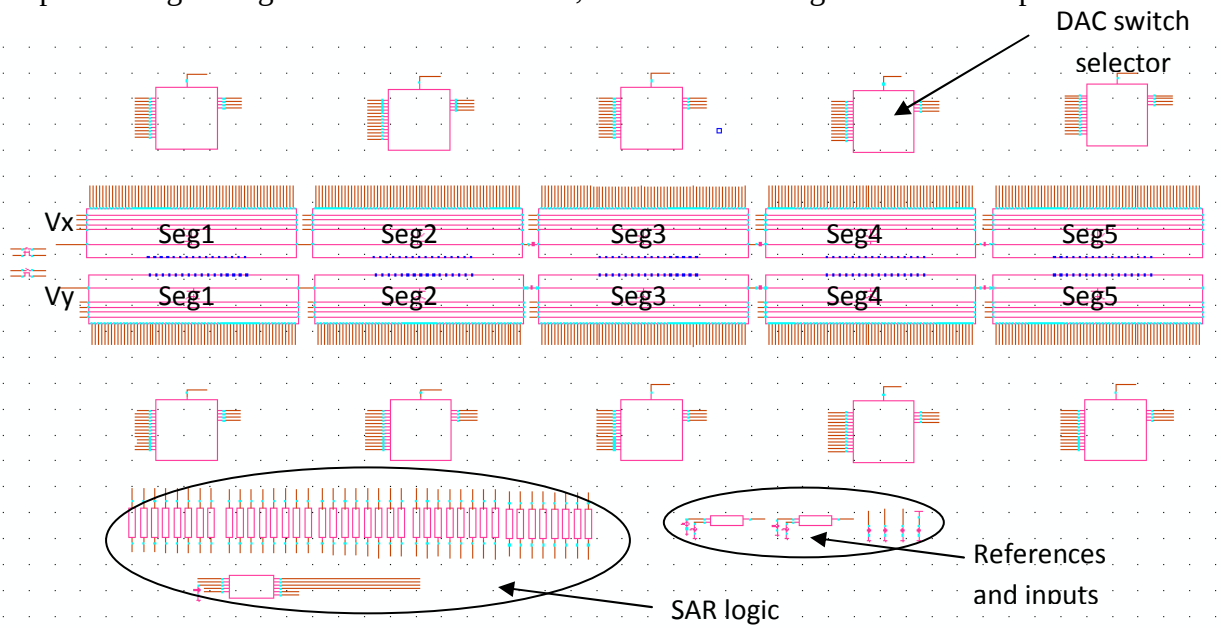


Fig. 2.10 Simulation of the 16-bit differential SAR converter

Figure 2.11 shows that the top plate voltages of the DAC Vx and Vy during the hold mode are linearly proportional to the differential input voltage.

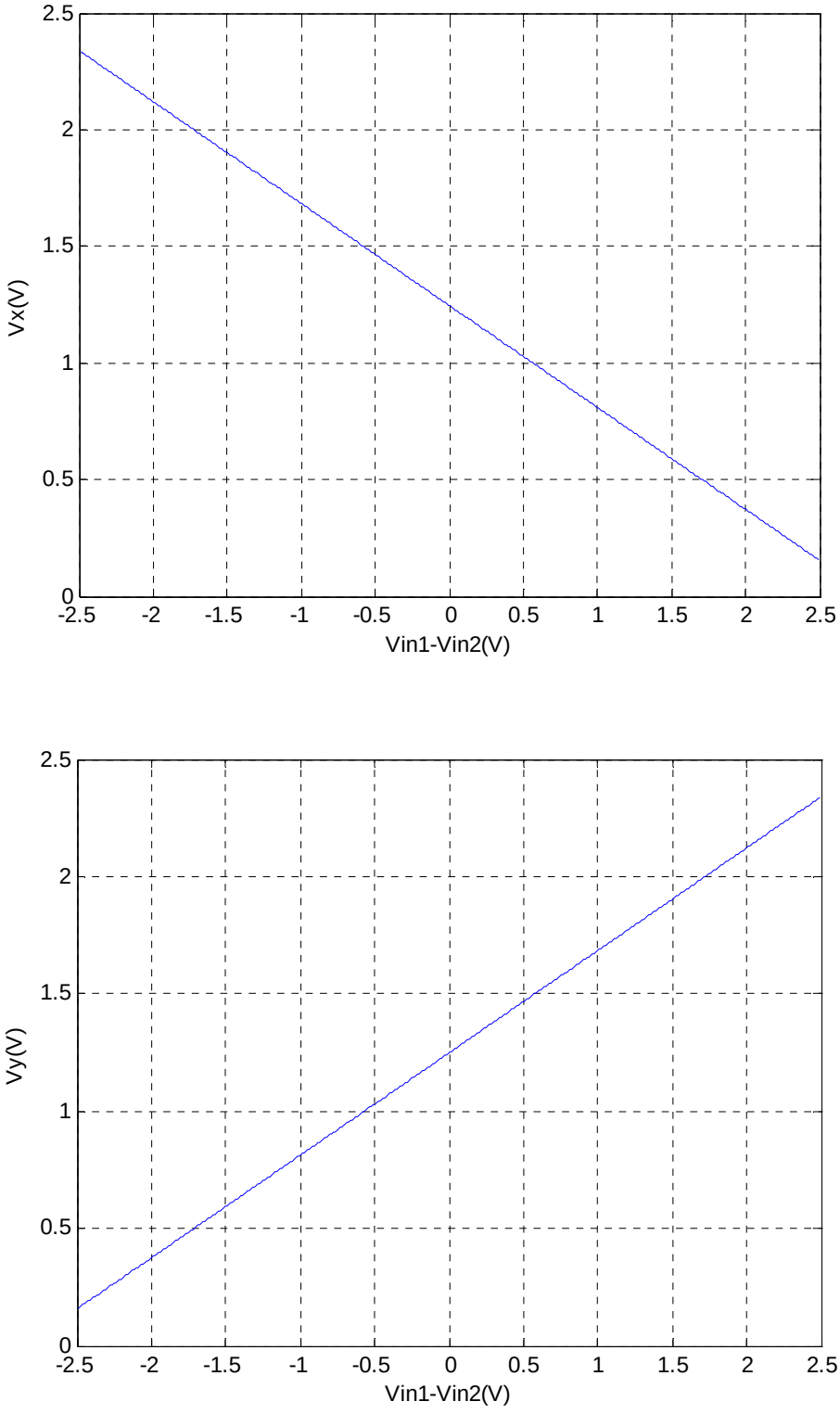


Figure 2.11: The relationship between the differential input and the top plate voltages Vx and Vy during the hold mode

Using the fitline function in Matlab, we found out that we can use (2.6) to predict the value of Vx and Vy during the hold mode.

$$Vx = -0.4367486 \cdot (Vin1 - Vin2) + 1.247626 \quad (2.6a)$$

$$Vy = 0.4367486 \cdot (Vin1 - Vin2) + 1.247626 \quad (2.6b)$$

Furthermore, using the information in Table 2.2, we can deduce the value of Vx and Vy during the bit cycling mode with (2.7).

$$Vy_i = Vy_{i-1} - di \cdot Wi \ (2.7a)$$

$$Vx_i = Vx_{i-1} + di \cdot Wi \ (2.7b)$$

where the weight in each cycle is recorded in Table 2.3.

Although (2.6) is developed for the ideal case, we can modify it for the non-ideal case by changing the fitline parameters. Similarly, we can reuse (2.7), as long as we replaced the ideal weights with the non-ideal weights.

Table 2.3 DAC weights for each cycle in the bit cycling mode

i	W
1	0.54689168
2	0.27344584
3	0.13672292
4	0.06836146
5	0.06836146
6	3.418082e-2
7	1.709041e-2
8	8.545205e-3
9	8.545205e-3
10	4.2726025e-3
11	2.13630125e-3
12	1.068150625e-3
13	1.068150625e-3
14	5.340753125e-4
15	2.6703765625e-4
16	1.33518828125e-4
17	1.33518828125e-4
18	6.67594140625e-5
19	3.33797070313e-5
20	1.66898535156e-5

The values of the redundant bits are highlighted in Table 2.3. The total voltage weights represented by the redundant bits are therefore equal to 78mV (≈1022 LSB). The implication is that as long as the total voltage error caused by the mismatch and noise is less than 78mV, we will always have enough weights to bring the output code to within the allowed output range.

Figure 2.12 shows the change in top plate voltage Vx and Vy for one conversion, where the differential input is 2.5V.

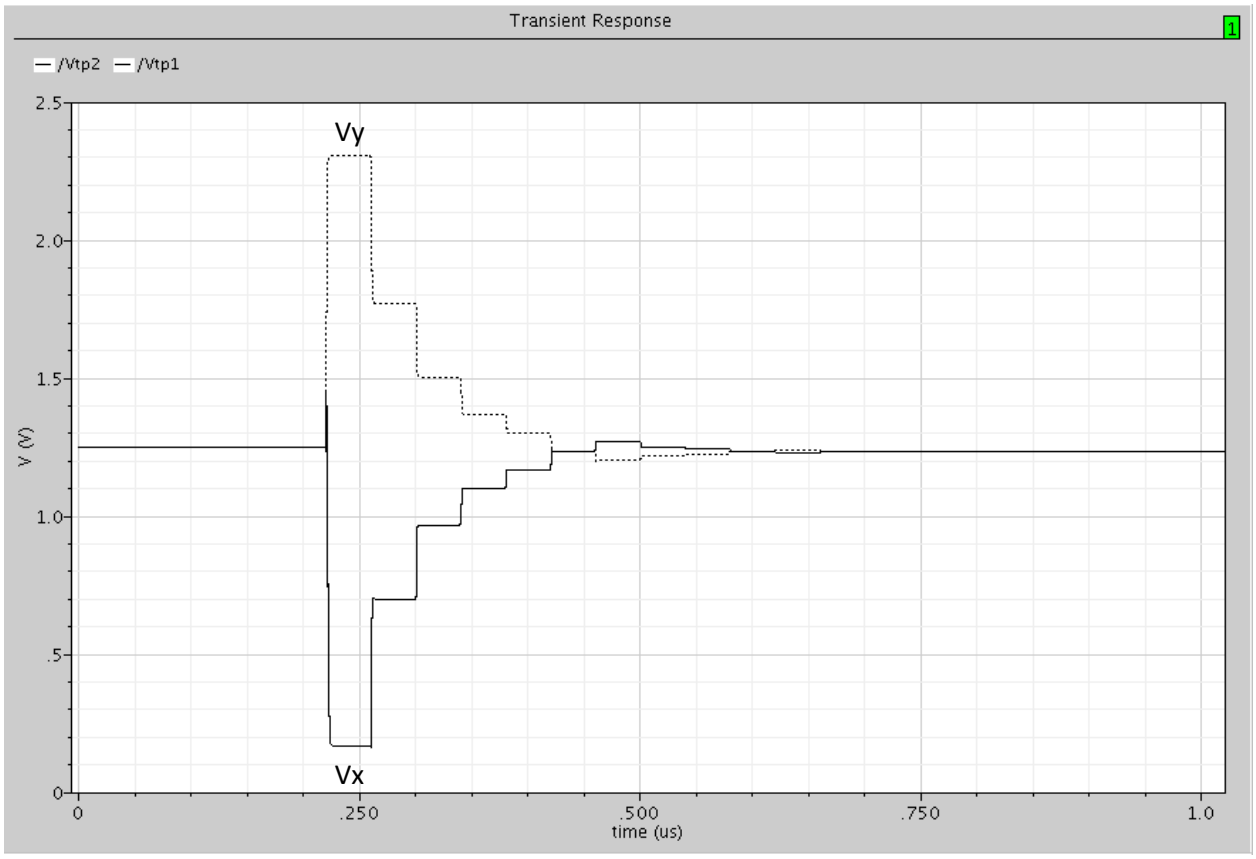


Figure 2.12 The waveform of Vx and Vy during one conversion

Note that the initial value of Vx and Vy are approximately 2.339V and 0.1557 as predicted by (2.6). The 20 decisions from the simulation are -1 -1 -1 -1 -1 1 -1 -1 -1 1 -1 -1 -1 1 -1 -1 -1 1 -1 -1. Using (2.7), we found the value of Vx-Vy to be 3.25015e-05V. This corresponds to the final value Vx-Vy in the simulation.

Figure 2.13 shows the final value of Vx-Vy for all the 1000 conversions. Note that they always converged to within ± 1/2 LSB at the 16-bit level.

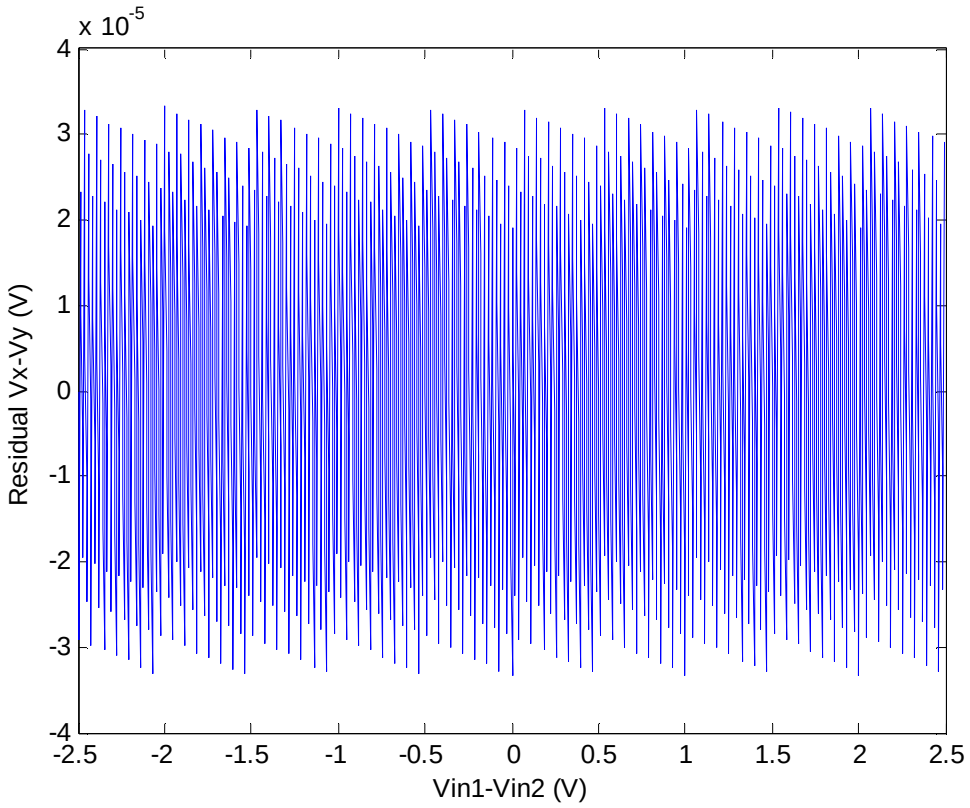


Figure 2.13 Vx-Vy at the end of the 1000 conversions

In Figure 2.14, we also plotted the ADC error ε against the differential input voltage range. The ADC error ε is obtained by (2.8) and (2.9), where $\overline{V}$ is the differential output voltage.

$$\overline{V} = \sum di \cdot Wi \quad (2.8)$$

$$\varepsilon = \overline{V} - (Vin1 - Vin2) \text{ – any systematic gain error} \quad (2.9)$$

Figure 2.14 shows that the error is always within ± 0.5 LSB at the 16 bit level, as expected.

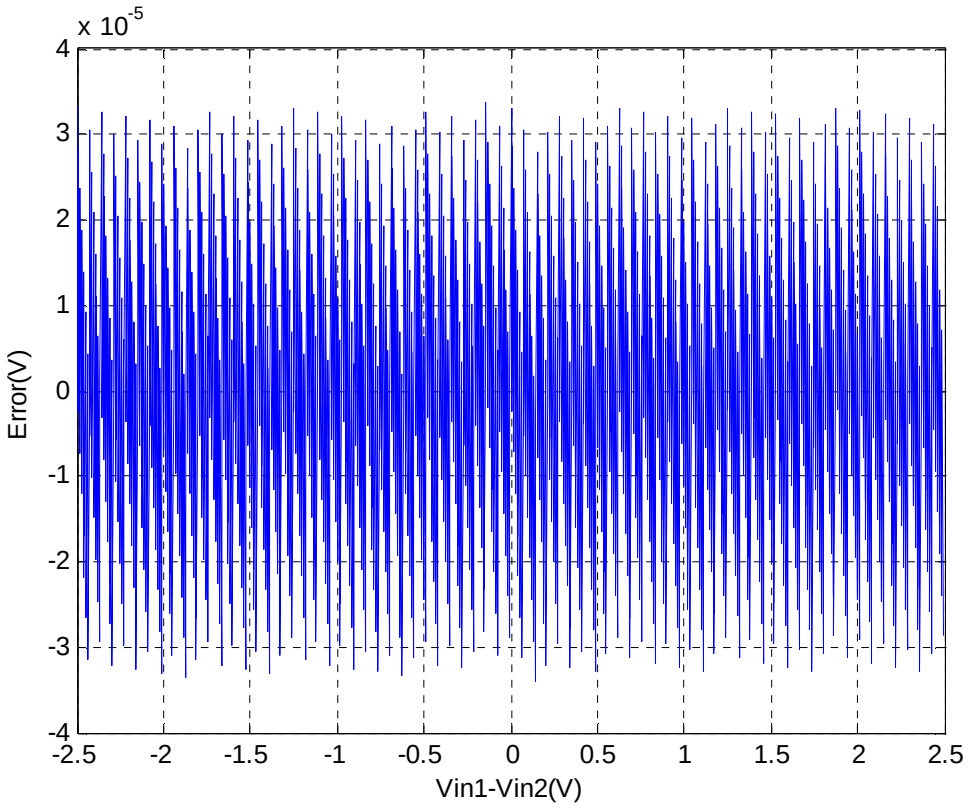


Figure 2.14 The error between the differential input and differential output

2.6 Summary

In this chapter, the split ADC architecture and its benefits are reviewed, and an ideal 16-bit differential SAR converter is developed to be used with the split architecture. Simulation verifies that the ideal 16-bit differential SAR converter works as expected at the system level.

The ideal 16-bit differential SAR converter serves as an important template for the development of the non-ideal Matlab ADC in Section 3.3. In addition, we can check whether the ADC design in Chapter 5 work, by comparing the design results with the simulation results in this chapter.

3

Split SAR ADC Implementation

3.1 Introduction

This chapter starts by explaining how the split ADC architecture and the error correction algorithm work together to calibrate the SAR converter at a system level. Section 3.3 explains how to simulate the whole system in Matlab behaviorally. Section 3.4 describes the performance metrics used to measure the success of the self-calibrating algorithm in calibrating the SAR ADC. Finally, Section 3.5 shows Matlab simulation results for the whole system and Section 3.6 summaries this chapter.

3.2 System Implementation

Figure 3.1 shows the system block diagram of the split SAR ADC implementation. The system consists of 2 main components, the mixed-signal IC and the external FPGA control. The mixed signal IC consists of the "split" SAR ADC and the internal SAR logic, while the external FPGA control consists of the error correction algorithm and digital logic necessary to communicate with the mixed signal IC.

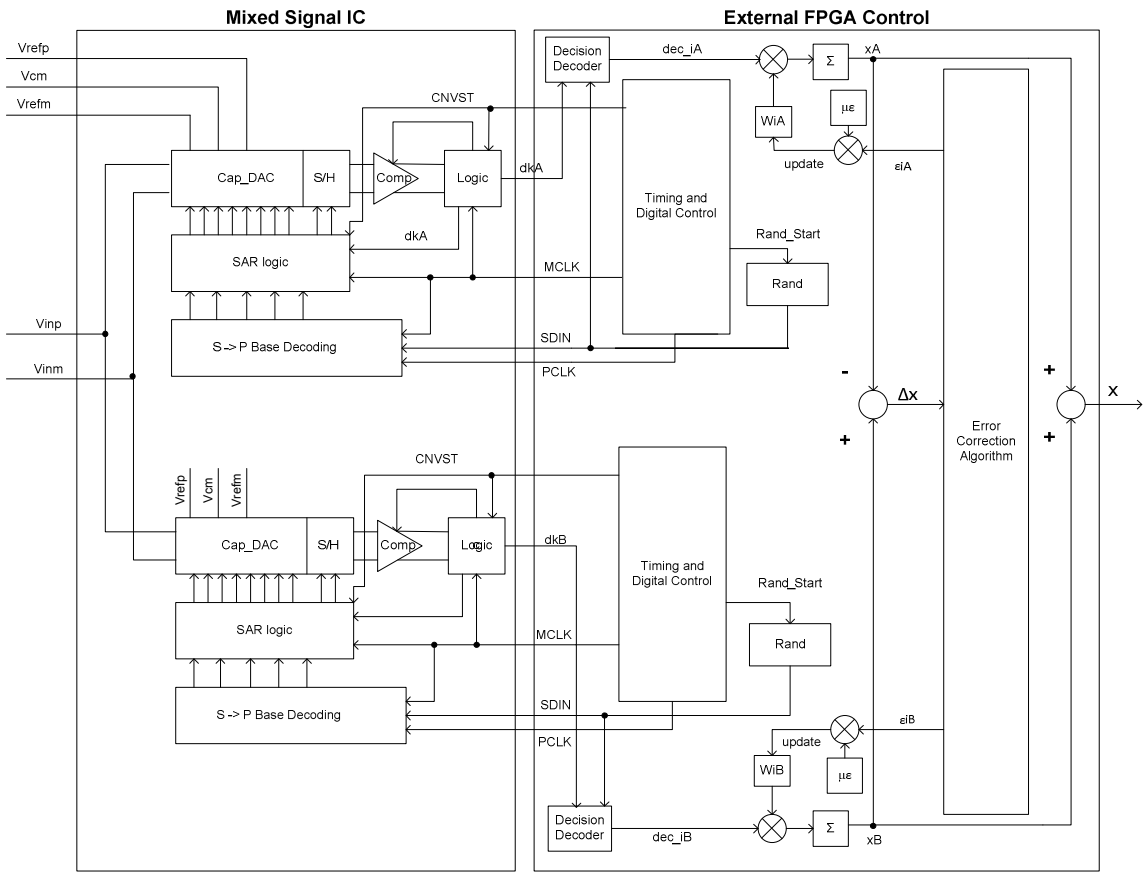


Figure 3-1: System block diagram

3.2.1 Foreground Operation

The mixed signal IC contains two SAR ADCs and their digital controls. The modified SAR converter and its theory of operation are described in details in Section 2.2.In Figure 3-1, the signal MCLK is used as a master clock signal for the ADC conversion. The signal RAND_Start is used to signal the random number generator to start generating random bases for the ADC. The signal SDIN transmits the random bases to the serial to parallel base decoder (S-> P decoding). When 5 bases are accumulated in the S-> P decoder, the PCLK signal is asserted. This signals the SAR logic to start the sample and hold action. The SAR logic serves as a mini state machine and controls the sample/hold and bit cycling action of the ADC. During the bit cycling mode, it also serves as a decoder to control how the capacitor switches in the DAC are used, depending on the comparator decision and the random bases received. At the end of the sample-and-hold action, the signal CNVST is asserted. This turns on the latched comparator, and starts the bit cycling action. Figure 3-2 shows how these signal changes during 1 conversion.

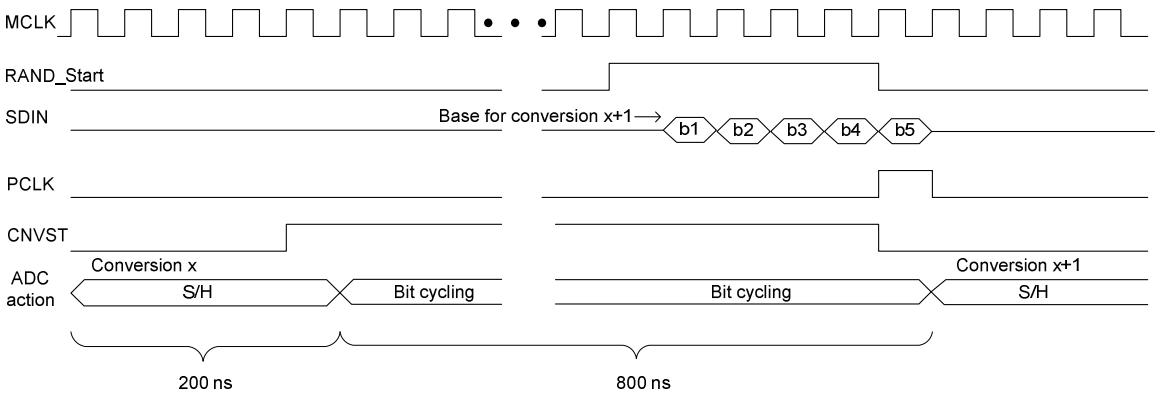


Figure 3-2 Timing diagram

3.2.2 Background self-calibrating algorithm

The external FPGA control is responsible for the following: For each conversion, the decisions accumulated (dkA and dkB) are converted to a sequence of number (dec_{iA} and dec_{iB}) by the decision decoder. This sequence of number indicates how a particular capacitor in the DAC is used during the bit cycling mode. A positive number indicates we added its weight, a negative number indicate we subtracted its weight, and 0 indicate we did not use that particular capacitor at all. This new sequence of number is then multiplied by the estimated voltage weights $\widehat{W}_{iA}$ and $\widehat{W}_{iB}$ stored in the lookup table. They are then accumulated to produce the output code xA and xB.

While xA and xB are averaged to produce the output code x, their difference Δx is used in an error correction algorithm to estimate the errors for $\widehat{W}_{iA}$ and $\widehat{W}_{iB}$ in the L.U.T. One should note that the error correction algorithm operates continuously in the background, and does not interfere with the foreground conversion. When the new error estimates ($\hat{\varepsilon}_{iA}$ and $\hat{\varepsilon}_{iB}$) become ready, the weights in the L.U.T are simply updated by (3.1)

$$\widehat{W}_{iA}^{(new)} = \widehat{W}_{iA}^{(old)} - \mu_e \hat{\varepsilon}_{iA} \quad (3.1a)$$

$$\widehat{W}_{iB}^{(new)} = \widehat{W}_{iB}^{(old)} - \mu_e \hat{\varepsilon}_{iB} \quad (3.1b)$$

The role of the LMS coefficient μ_e is to control the speed of adaptation and the accuracy of the convergence [32, 33]. In general, a bigger μ_e helps the estimated weights converge to the true weights more quickly. However, it also makes the system more susceptible to noise. As a result, the system may overestimate the weights and there is a danger that the solution will diverge instead. A smaller μ_e helps averaging out the noise in the system and provides more accurate weight estimates. However, the tradeoff is that we need longer time for $\widehat{W}_{iA}$ and $\widehat{W}_{iB}$ to reach

their equilibrium. μ_e is chosen to be a number divisible by 2. This is to simplify the digital hardware implementation.

When the estimated errors $\hat{\varepsilon}_{iA}$ and $\hat{\varepsilon}_{iB}$ reach zero (on average), equilibrium is reached and the estimated weights $\widehat{W}_{iA}$ and $\widehat{W}_{iB}$ converge to their true values. As long as the estimates $\hat{\varepsilon}_{iA}$ and $\hat{\varepsilon}_{iB}$ change in successive small steps and point to the right direction, the weight estimates $\widehat{W}_{iA}$ and $\widehat{W}_{iB}$ will converge to their final steady states [8].

3.3 System Behavioral simulation

To reduce the cost and time for implementing the real system, we first want to make sure the system work .To do this, we first used Matlab to simulate the behavior of the mixed signal IC and the external FPGA control in Figure 3.1.

To simulate the operation of the non-ideal 16bit, 1MS/s differential split SAR ADC in Matlab, we can implement the SAR algorithm as shown in Figure 2.2 for two separate MATLAB SAR ADCs. We can then modify (2.6) to predict the top plate voltages Vx and Vy during the hold mode, and modify (2.7) to predict the DAC top plate voltages during the bit cycling mode.

To modify (2.6), we rerun the ideal 16bit SAR ADC Cadence simulation we developed in Section 2.5, with capacitor mismatch added to the Cadence simulation.

As an example to show how this work, we assume there are only mismatch in the first DAC segment of ADCB, where

C1 = 1.01p
C2 = 1.02p
C3 = 0.97p
C4 = 1.03p
C5 = 1.01p
C6 = 0.98p
C7 = 0.95p
C8 = 0.96p
C9 = 1.01p
C10 = 0.99p
C11 = 1.05p
C12 = 0.98p
C13 = 0.97p
C14 = 1.04p
C15 = 0.96p
C16 = 1.02p

We rerun the simulation, and plotted the DAC voltage Vx and Vy at the end of the hold mode, against the differential input voltage. Figure 3.3 shows that their linear relationships remain the same as in Figure 2.11.

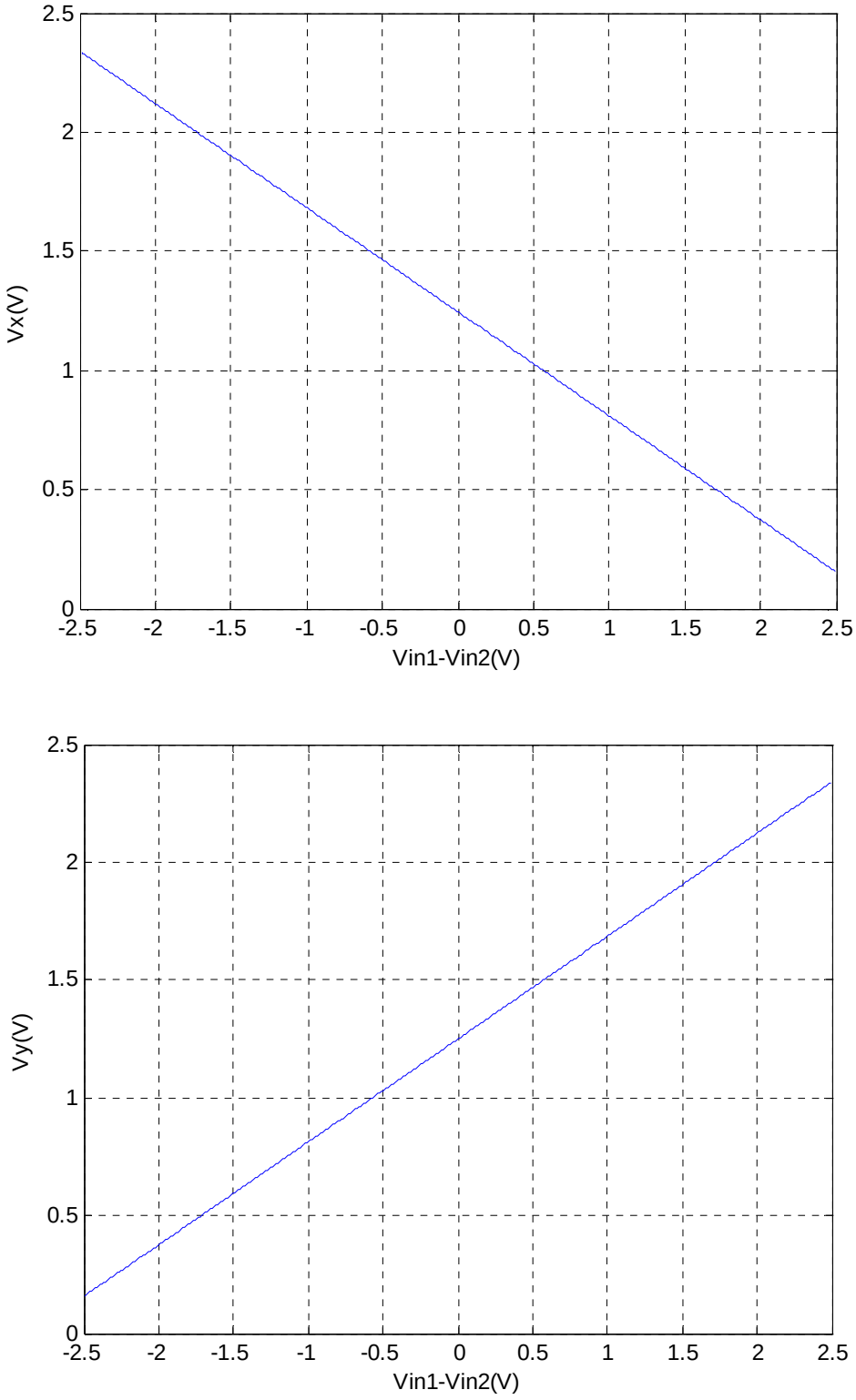


Figure 3.3: The relationship between the differential input and the top plate voltages Vx and Vy during the hold mode

Using linear fitlines for both graphs, we found that the mismatch does not affect the slope and the offset parameters in (2.6). Therefore, we can just use (2.6) to predict the top plate voltages during the hold mode.

To model the top plate voltage during the bit cycling mode, we rerun the simulation in Section 2.4.2 to find the mismatch weights, by changing the value of C1-C16 in the Cadence simulation.

Table 3.1 shows the ideal voltage weight of each capacitor, and the new voltage weight after the mismatch is applied in segment 1. The DAC weight errors $\hat{\varepsilon}_{iB}$, and the ratio of the mismatch weight to the ideal weight for segment 1 are also calculated. Similar Tables for segment2-5 can be found in Appendix B.

Table 3.1 DAC weights in segment 1

	Ideal	**With Mismatch**	**Weight errors εiB**	**Ratio of Mismatch/ Ideal**
C	Voltage weight			
1	6.8361460E-02	6.9234400E-02	8.7294000E-04	1.0127695
2	6.8361460E-02	6.9919890E-02	1.5584300E-03	1.0227969
3	6.8361460E-02	6.6492440E-02	-1.8690200E-03	0.9726597
4	6.8361460E-02	7.0605380E-02	2.2439200E-03	1.0328243
5	6.8361460E-02	6.9234400E-02	8.7294000E-04	1.0127695
6	6.8361460E-02	6.7177930E-02	-1.1835300E-03	0.9826872
7	6.8361460E-02	6.5121460E-02	-3.2400000E-03	0.9526049
8	6.8361460E-02	6.5806950E-02	-2.5545100E-03	0.9626323
9	6.8361460E-02	6.9234400E-02	8.7294000E-04	1.0127695
10	6.8361460E-02	6.7863420E-02	-4.9804000E-04	0.9927146
11	6.8361460E-02	7.1976350E-02	3.6148900E-03	1.0528791
12	6.8361460E-02	6.7177930E-02	-1.1835300E-03	0.9826872
13	6.8361460E-02	6.6492440E-02	-1.8690200E-03	0.9726597
14	6.8361460E-02	7.1290860E-02	2.9294000E-03	1.0428516
15	6.8361460E-02	6.5806950E-02	-2.5545100E-03	0.9626323
16	6.8361460E-02	6.9919890E-02	1.5584300E-03	1.0227969

By applying the weight errors to the Matlab ADC, we successfully simulated a non-ideal 16bit differential SAR converter with capacitor mismatches. We also added a 30uV rms noise during the hold mode and bit cycling mode in the Matlab ADC to simulate the kT/C noise in the real system.

To implement the self-calibrating algorithm in Matlab, we implemented the error correction algorithm described in Chapter 4 to estimate the weight errors $\hat{\varepsilon}_{iA}$ and $\hat{\varepsilon}_{iB}$. For every 128 conversions, we updated the DAC weights $\widehat{W}_{iA}$ and $\widehat{W}_{iB}$ with the LMS loop described in (3.1).

3.4 Performance Metrics

3.4.1 Differential NonLinearity (DNL)

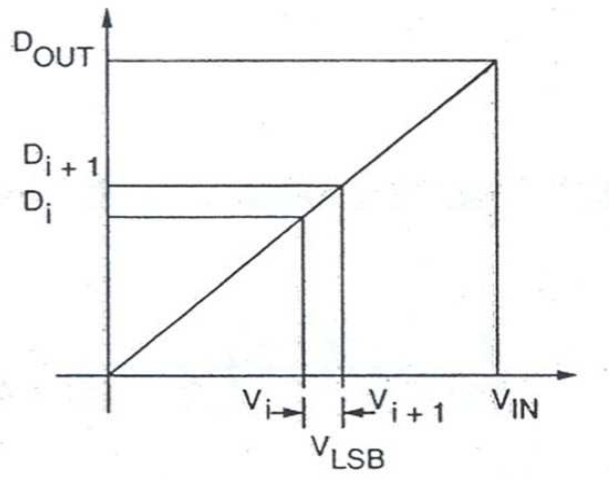


Figure 3.4 Ideal ADC transfer function

The output digital codes of an ideal ADC can be plotted against the analog input voltage as shown in Figure 3.4. For simplicity, we replace the ideal staircase transfer function of the ADC with a straight line. In an ideal ADC, the spacing between the two digital output codes D_i and D_{i+1} equals V_{LSB}, where V_{LSB} is the voltage corresponds to one Least Significant Bit (1 LSB), and it is the smallest voltage that a ADC can resolve [34]. 1 LSB is equal to

$$1\ LSB = V_{LSB} = \frac{Vref}{2^N} \quad (3.2)$$

where $Vref$ is the full-scale analog input voltage, and N is the resolution of the ADC in bits.

The presence of non-idealities in ADC makes the spacing between the two digital output codes either greater or less than 1 LSB. To measure the change in this decision spacing, we define the Differential NonLinearity(DNL) [34]. The DNL for a particular output code is defined as

$$dnl[i] = \left(\frac{V_{i+1} - V_i}{V_{LSB}}\right) - 1 \quad (3.3)$$

where i equals to the value of the digital output Di.

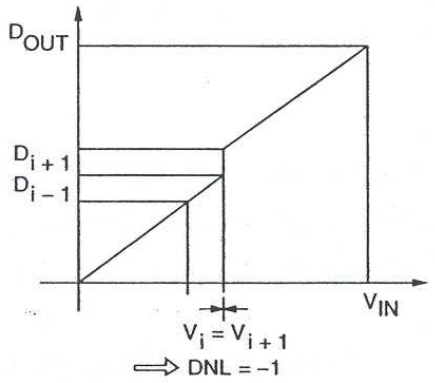


Figure 3.5 A transfer function with a missing code

If the digital output code D_i never appears at the output, the analog decision point $V_{i+1} = V_i$. From (3.3), then, the DNL for Di = -1. This scenario is known as missing code and it is illustrated in Figure 3.5. Note that from (3.3), a DNL of -1 is the worst case negative DNL an ADC can have [34].

On the other hand, if the spacing between D_i and D_{i+1} equals to 2 LSB or more, we have a scenario known as wide code. Since there is no limit as to how far Vi+1 can be from Vi, there is no limit in the positive DNL[34].

3.4.2 Integral NonLinearity (INL)

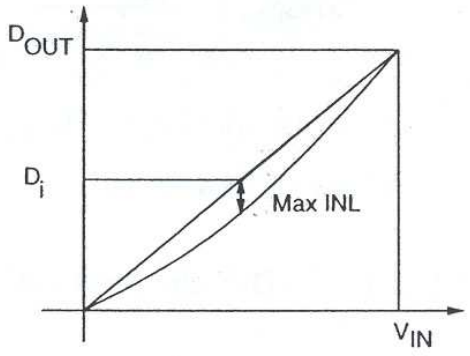


Figure 3.6 A transfer function with large INL

Figure 3.6 plots both the ideal and non-ideal ADC transfer function. The intergral NonLinearity (INL) is a measure of the difference between the actual output codes produced by the ADC and the ideal linear curve [34]. Figure 3.6 shows that the maximum INL occurs at the code Di. INL can be calculated from the cumulative sum of DNL, where

$$INL[i] = \sum_{k=1}^{i} dnl[k] \quad (3.4)$$

Note that it is important to distinguish between DNL and INL. DNL measures how small an input voltage an ADC can resolve, while INL measures the absolute accuracy of an ADC. While an ADC can resolve two input analog voltages to within 1 LSB at the 12 bit level, it may not be accurate to a 12-bit level. In other words, the DNL of a ADC can be smaller than its INL in LSB and vice versa [34].

3.4.3 Characterizing the speed of convergence

The best way to investigate the speed of the self-calibrating algorithm is to observe how the weight error and the average ADC error change with respect to the conversion index. The weight error is the absolute value of the second term of (3.1), and the average ADC error is the average of the difference between the ADC input and the ADC output from the two ADCs. It can be calculated by the second term of (4.17), where its origin will be explained in Section 4.3.2.

$$\widehat{W}_{iA}^{(new)} = \widehat{W}_{iA}^{(old)} - \mu_e \hat{\varepsilon}_{iA} \quad (3.1a)$$

$$\widehat{W}_{iB}^{(new)} = \widehat{W}_{iB}^{(old)} - \mu_e \hat{\varepsilon}_{iB} \quad (3.1b)$$

$$\hat{x} = \frac{\sum dec_{iB} W_{iB} + \sum dec_{iA} W_{iA}}{2} + \frac{\sum dec_{iB}\, \varepsilon_{iB} + \sum dec_{iA}\, \varepsilon_{iA}}{2} \quad (4.17)$$

3.5 Simulation of the whole system

The Matlab code for implementing the whole system in Figure 3.1 can be found in Appendix C. In this example, a μ_e of 2^{13} is used. If the system operates correctly, we would expect the calibration parameters (DAC weights) converge to their correct values, and that the average ADC error will converge to less than 1 LSB. We would also expect to see improvement in the INL/DNL, after the ADC is calibrated. Simulation results show such is the case. Figure 3.7 shows that $\hat{\varepsilon}_{1B}$ converge from its initial magnitude of 10e-4 V to 10e-7 V. Figure 3.8 shows that the average ADC error converges to less than 1 LSB in its steady state. Lastly, Figure 3.9 shows that the self-calibrating algorithm is able to bring the INL/DNL down to ±0.5 LSB.

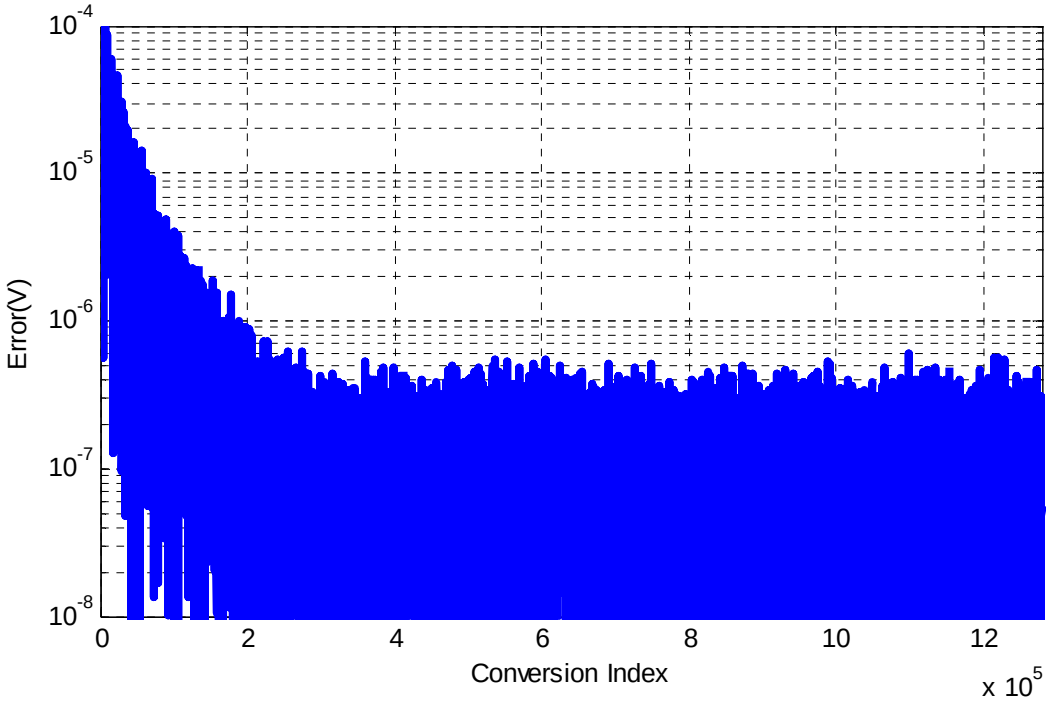


Figure 3-7: Convergence of the weight error of C1

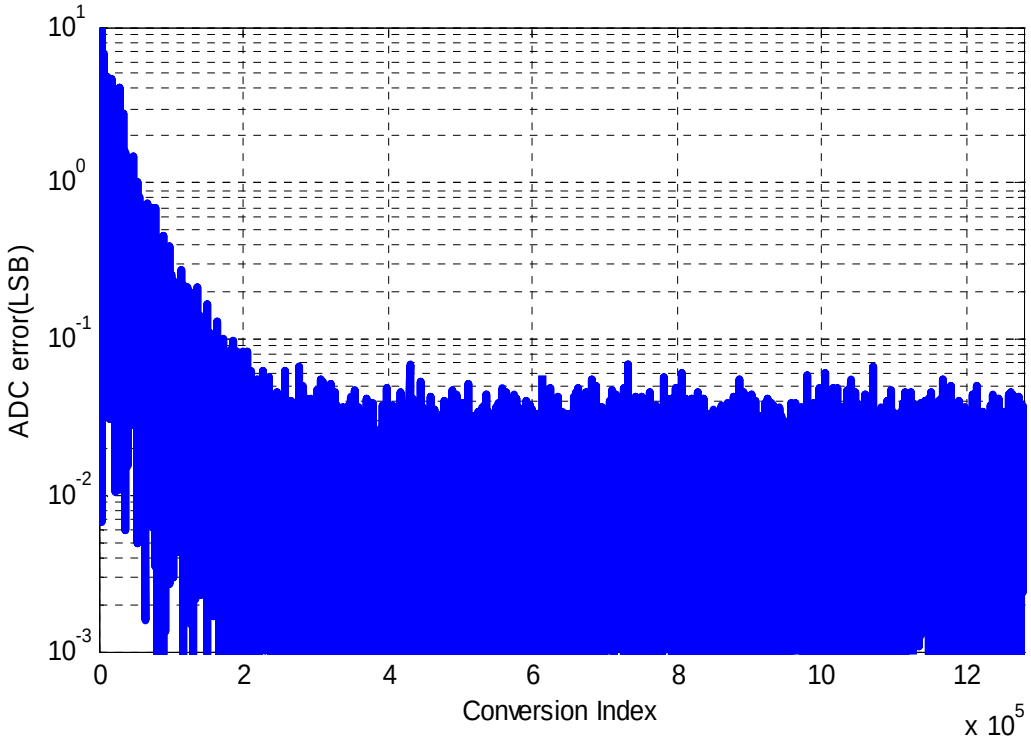


Figure 3-8: Convergence of the ADC error

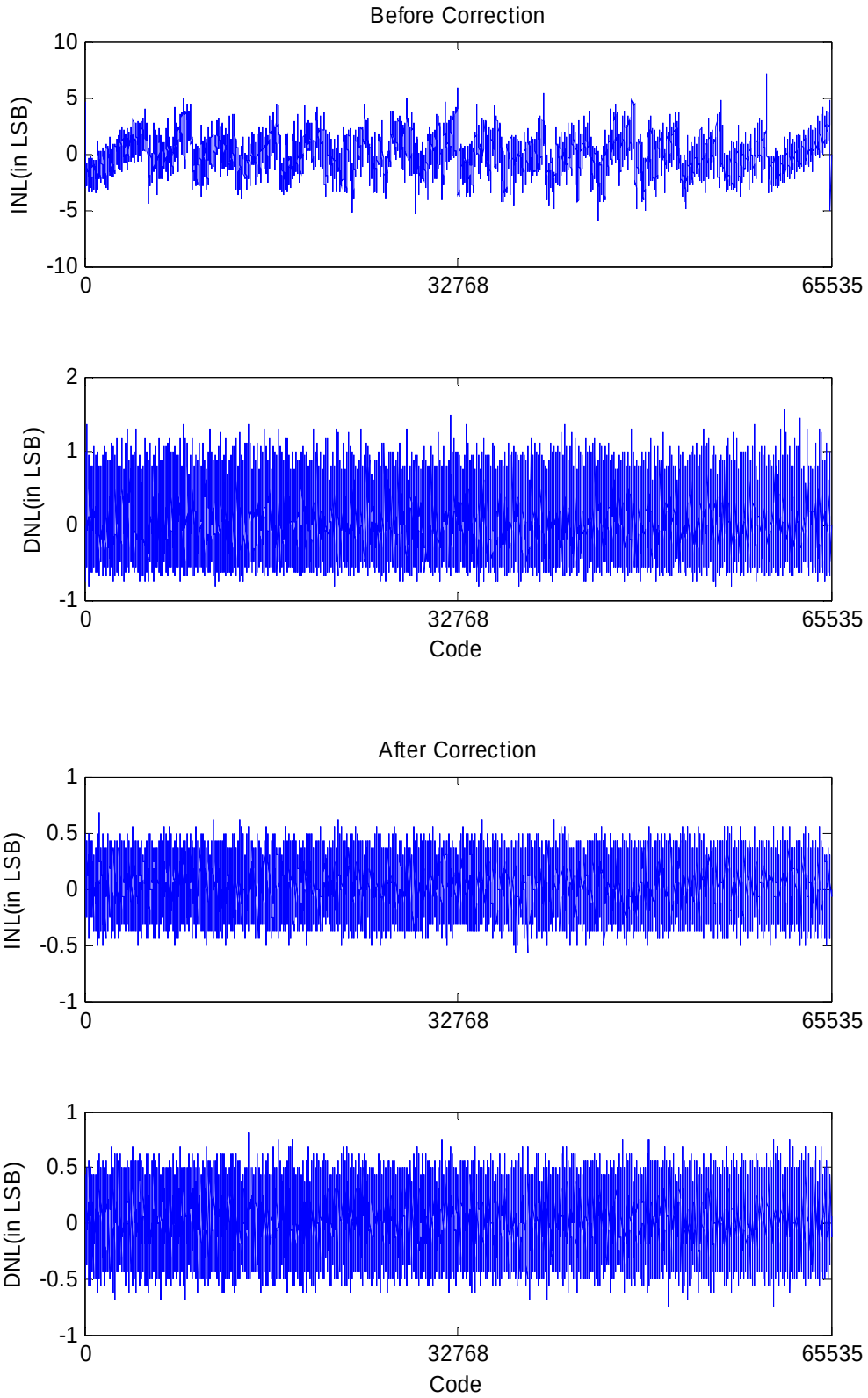


Figure 3-9: INL/DNL improvement

3.6 Summary

This chapter explains how the split ADC architecture work together with the self-calibrating algorithm to calibrate the ADC. Matlab is used to model the system behavior. Simulation results show that the self-calibrating algorithm works. In this example, it calibrates the weight within 400,000 conversions, and improves the INL/DNL of the ADC to within ±0.5 LSB.

4

Error Correction Algorithm

4.1 Introduction

The details of the error correction algorithm used in the "spilt" ADC architecture are explored in this chapter. In Section 4.2, the basic Jacobi Iterative method will be reviewed. In Section 4.3, we developed our error correction algorithm based on the basic Jacobi Iterative method. Section 4.4 shows how the error correction algorithm improves the INL/DNL and the frequency response of the calibrated ADC. It also explores how the choice of different calibration inputs and the adaptive parameter μ_e affect the speed and accuracy of the calibration. Finally, Section 4.5 summarizes this chapter.

4.2 Basic Jacobi Review

The Jacobi iterative method is based on the Jacobi transformation method of matrix diagonalization. It solves a matrix equation on a matrix that has no zeros along its main diagonal [35].

Suppose one want to solve a linear system

$$S.e = d \;\Rightarrow \begin{bmatrix} S_{11} & S_{12} & S_{13} \\ S_{21} & S_{22} & S_{23} \\ S_{31} & S_{32} & S_{33} \end{bmatrix} \begin{bmatrix} e_1 \\ e_2 \\ e_3 \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \end{bmatrix} \quad (4.1)$$

where e_k represents the unknown parameters we want to solve for, d_i represents observations from the system and S_{ij} are the known coefficients that describe the relationship between the unknown parameters e_k and the individual observation $d_i$of the system.

As an example to show how the Jacobi Iterative method works, we show how the first unknown parameter e_1 can be found [8]. Rewriting the first row of the matrix in equation form gives

$$S_{11}e_1 + S_{12}e_2 + S_{13}e_3 = d_1 \quad (4.2)$$

From (4.2), we can see that e_1 can be easily solved, if e_2 and e_3 are known. In the Jacobi Iterative method, however, we extract the new iteration for e_1 from the old iterative values of e_2 and e_3 as in (4.3)

$$e_1{}^{(new)} = \frac{1}{S_{11}}(d_1 - S_{12}e_2{}^{(old)} - S_{13}e_3{}^{(old)}) \quad (4.3)$$

Generalizing equation (4.3), we can solve each unknown e_k for each observation i by

$$e_k{}^{(new)} = \frac{1}{S_{ik}}(d_i - \Sigma_{j\neq k}S_{ij}e_j{}^{(old)}) \quad (4.4)$$

where k in the index of the unknown parameter we want to solve for, and i and j are the row and column indices of the system matrix (4.1).

The process is repeated until the unknown parameters e_k converge to sufficiently accurate values.

In our system, the observation d_i is the difference (Δx) between the two output codes xA and xB of the two ADCs. The unknown parameter e_k is the unknown DAC weight errors $\hat{\varepsilon}_{iA}$,$\hat{\varepsilon}_{iB}$, and the known coefficients are the decisions coming out from the two ADCs. The next section shows how we derive matrix equations for our system in the form of (4.1).

4.3 Error correction algorithm

4.3.1 Derivation of Δx

The output code of each side of the split ADC are accumulated from the product of $dec_{iA,j}$, $dec_{iB,j}$, and the capacitor weight $\widehat{W}_{jA}$ and $\widehat{W}_{jB}$,as shown in (4.5), where i is the conversion index, and j is the DAC weight indices in ADCA and ADCB.

$$\hat{x}_{Ai} = \sum dec_{iA,j} \cdot \widehat{W}_{jA} \quad (4.5a)$$

$$\hat{x}_{Bi} = \sum dec_{iB,j} \cdot \widehat{W}_{jB} \quad (4.5b)$$

$dec_{iA,j}$ and $dec_{iB,j}$ are the known coefficients indicating how the capacitors in the DAC array of ADCA and ADCB are used in each conversion. We briefly explained them in Section 3.2.2. In this section, we provide a more in-depth explanation. Recall that in the ADC operation, we generate a random base for each segment so that we know which capacitor is responsible for making which bit of decision. For example, if the base number generated is 3 for segment 1, then capacitor 3-10 will be used for d1, capacitor 11-14 will be used for d2, capacitor 15-16 will be used for d3, capacitor 1 will be used for d4A and capacitor 2 will be left unused. Thus, if the random bases generated for segment 1-5 for ADCA in conversion 1 are 3, 5, 6, 7, 8 respectively, then

$$dec_{1A,1-50} = [S1\ S2\ S3\ \sum S4\ \sum S5] \quad (4.6a)$$

where S1 records the first 16 known coefficients and is equal to

$$S1 = [d_{4A}\ 0\ d_1\ d_1\ d_1\ d_1\ d_1\ d_1\ d_1\ d_2\ d_2\ d_2\ d_3\ \ d_3] \quad (4.6b)$$

Similarly,

$$S2 = [d_6\ d_6\ \ d_{7A}\ \ 0\ d_{4B}\ d_{4B}\ d_{4B}\ d_{4B}\ d_{4B}\ d_{4B}\ d_{4B}\ d_{4B}\ d_5\ d_5\ \ d_5\ \ d_5\ ]. \quad (4.6c)$$

Following the same rule,

$$S3 = [d_8\ d_9\ \ d_9\ \ d_{10A}\ 0\ \ d_{7B}\ d_{7B}\ d_{7B}\ d_{7B}\ d_{7B}\ d_{7B}\ d_{7B}\ d_{7B}\ d_8\ \ d_8\ \ d_8\ ] \quad (4.6d)$$

$$S4 = [d_{11}\ d_{11}\ \ d_{12}\ \ d_{12}\ d_{13A}\ \ 0\ d_{10B}\ d_{10B}\ d_{10B}\ d_{10B}\ d_{10B}\ d_{10B}\ d_{10B}\ d_{10B}\ \ d_{11}\ d_{11}\ ] \quad (4.6e)$$

$$S5 = [d_{14}\ d_{14}\ \ d_{14}\ \ d_{15}\ d_{15}\ \ d_{16}\ 0\ d_{13B}\ d_{13B}\ d_{13B}\ d_{13B}\ d_{13B}\ d_{13B}\ d_{13B}\ \ d_{13B}\ d_{14}\ ]. \quad (4.6f)$$

Since it is unlikely that the typical mismatch in the individual capacitors in segment 4 and segment5 will have an impact on the output code, we only store the total capacitor weights for segment 4 and segment 5. Hence, we sum up the decisions in S4 and S5 in (4.6a) to indicate how the group weights are used.

The estimates $\widehat{W}_{jA}$ and $\widehat{W}_{jB}$ are related to their true analog capacitor weights W_{jA} and W_{jB} by (4.7), where ε_{jA} and ε_{jB} are the fractional errors of $W_{jA}\ and\ W_{jB}$.

$$\widehat{W}_{jA} = W_{jA} + \varepsilon_{jA} \qquad \widehat{W}_{jB} = W_{jB} + \varepsilon_{jB} \quad (4.7)$$

Substituting (4.7) into (4.5), we get

$$\hat{x}_{Ai} = \sum dec_{iA,j} \cdot (W_{jA} + \varepsilon_{jA}) \quad (4.8a)$$

$$\hat{x}_{Bi} = \sum dec_{iB,j} \cdot (W_{jB} + \varepsilon_{jB}) \quad (4.8b)$$

Subtracting (4.8b) from (4.8a), we have

$$\Delta x_i = \sum dec_{iB,j} W_{jB} - \sum dec_{iA,j}\ W_{jA} + \sum dec_{iB,j}\ \varepsilon_{jB} - \sum dec_{iA,j}\ \varepsilon_{jA} \quad (4.9)$$

Since both ADCs are converting the same input code and W_{jA} , W_{jB} are the true analog capacitor weights, the first two terms must be equal (to within the quantization error)even if their decision paths are different. As a result, they cancel each other, leaving behind the last two terms. Therefore,

$$\Delta x_i = \sum dec_{iB,j}\ \varepsilon_{jB} - \sum dec_{iA,j}\ \varepsilon_{jA} \quad (4.10)$$

Writing (4.10) in the form of (4.1) for 100 conversions, we have

$$\begin{bmatrix} \Delta x_1 \\ \Delta x_2 \\ \vdots \\ \vdots \\ \vdots \\ \Delta x_{100} \end{bmatrix} = \begin{bmatrix} dec_{1B,1} & dec_{1B,2} & \cdots & dec_{1B,50} & -dec_{1A,1} & -dec_{1A,2} \dots & -dec_{1A,50} \\ \vdots & & \ddots & & & \vdots & \\ dec_{100B,1} & dec_{100B,2} & \cdots & dec_{100B,50} & -dec_{100A,1} & -dec_{100A,2} \dots & -dec_{100A,50} \end{bmatrix} \begin{bmatrix} \varepsilon_{1B} \\ \varepsilon_{2B} \\ \vdots \\ \varepsilon_{50B} \\ \varepsilon_{1A} \\ \varepsilon_{2A} \\ \vdots \\ \varepsilon_{50A} \end{bmatrix} \quad (4.11)$$

Therefore, each DAC weight errors ε_{kB} in ADCB can be solved by

$$\varepsilon_{kB}{}^{(new)} = \frac{1}{dec_{iB,k}}(\Delta x_i - \Sigma - dec_{iA,j}\varepsilon_{jA}{}^{(old)} - \Sigma_{j\neq k} dec_{iB,j}\varepsilon_{jB}{}^{(old)}) \quad (4.12a)$$

Similarly, each DAC weight errors ε_{kA} in ADCA can be solved by

$$\varepsilon_{kA}{}^{(new)} = \frac{1}{-dec_{iA,k}}(\Delta x_i - \Sigma dec_{iB,j}\varepsilon_{jB}{}^{(old)} - \Sigma_{j\neq k} - dec_{iA,j}\varepsilon_{jA}{}^{(old)}) \quad (4.12b)$$

4.3.2 Modified Jacobi iterative Method

Solving the DAC weight errors ε_{iA} and $\varepsilon_{iB}$with the basic Jacobi Iterative method in (4.12) poses some difficulties.

1.) To obtain 1 iteration of ε_{iA} and ε_{iB}, we need to do 99 multiplications, 99 subtractions, and 1 non-binary division. This is very time and memory consuming. In addition, non-binary division is hard to implement in hardware.
2.) The iteration stability depends on the value of the diagonal elements of (4.11).If the diagonal elements dominates, then the iteration in (4.12) is stable and is guaranteed to converge [7]. Since the position of the unused capacitor in each segment is random, there is always a chance that one of the diagonal elements in (4.11) is zero. Therefore, the matrix will need to be manipulated to eliminate any zeros in the diagonal elements.
3.) The accuracy of the iteration is very sensitive to the diagonal elements in (4.11). If the kT/C noise causes the diagonal elements to deviate from their supposed value, it can point the estimates of $\varepsilon_{kA}{}^{(new)}$ and $\varepsilon_{kB}{}^{(new)}$in completely wrong directions. Therefore, we better solve an over-determined system with more conversions.

To overcome the problems mentioned above, we modify the basic Jacobi Iterative method and form the new system matrix in (4.13), where Δx, dec_{iB} and dec_{iA} are accumulated from 128 conversions:

$$\begin{bmatrix} \Delta x_1 \\ \Delta x_2 \\ \vdots \\ \vdots \\ \vdots \\ \Delta x_{128} \end{bmatrix} = \begin{bmatrix} dec_{1B,1} & dec_{1B,2} & \cdots & dec_{1B,50} & -dec_{1A,1} & -dec_{1A,2} \dots & -dec_{1A,50} \\ \vdots & & \ddots & & \vdots & & \\ dec_{128B,1} & dec_{128B,2} & \cdots & dec_{128B,50} & -dec_{128A,1} & -dec_{128A,2} \dots & -dec_{128A,50} \end{bmatrix} \begin{bmatrix} \varepsilon_{1B} \\ \varepsilon_{2B} \\ \vdots \\ \varepsilon_{50B} \\ \varepsilon_{1A} \\ \varepsilon_{2A} \\ \vdots \\ \varepsilon_{50A} \end{bmatrix}$$

Δx dec ε

(4.13)

We then perform the following steps to solve for the unknown DAC weights ε_{kA} and ε_{kB}.

1.) Assume you want to solve for ε in row k. Go to column k of *dec* and check for any negative numbers. Negate them to get positive numbers. If you see a zero in any row p in column k of *dec*, change Δx in row p to zero.
2.) Add 128 rows of Δx and *dec* together. This result in the matrix in (4.14), with i indicates the set of 128 conversions, and k indicates the column of *dec* manipulated in step 1 and the index of the error one want to solve for.
3.) If you are solving for ε_{kB}, you can apply $\varepsilon_{kB} = \Delta x_{i,k}$. Alternatively, if you are solving for ε_{kA}, you can apply $\varepsilon_{kA} = \Delta x_{i,k+50}$. This is based on the simplification of (4.15), which will be explained below.
4.) Repeat 1-3 until all 100 ε_k are solved. Multiply results by μ_e and update (3.1).
5.) Wait for the next 128 conversions and repeat the same procedure.

$$\left[\Delta x_{i,k}\right] = \left[S_{i,1}S_{i,2} \ \dots S_{i,k} \dots S_{i,99}S_{i,100}\right] \cdot \begin{bmatrix} \varepsilon_{1B} \\ \varepsilon_{2B} \\ \vdots \\ \varepsilon_{kB} \\ \vdots \\ \varepsilon_{50B} \\ \varepsilon_{1A} \\ \varepsilon_{2A} \\ \vdots \\ \varepsilon_{50A} \end{bmatrix} \quad (4.14)$$

From (4.14), we can solve each error ε_{kA} and ε_{kB} by (4.15), where i is the set of conversions, k is the error index we want to solve for, and j is the index of all error estimates.

$$\varepsilon_{kB}{}^{(new)} = \frac{1}{S_{i,k}}\left(\Delta x_{i,k} - \Sigma_{j\neq k} S_{i,j}\varepsilon_{jB}{}^{(old)} - \Sigma S_{i,j+50}\varepsilon_{jA}{}^{(old)}\right) \text{ (4.15a)}$$

$$\varepsilon_{kA}{}^{(new)} = \frac{1}{S_{i,k+50}}\left(\Delta x_{i,k+50} - \Sigma_{j\neq k} S_{i,j+50}\varepsilon_{jA}{}^{(old)} - \Sigma S_{i,j}\varepsilon_{jB}{}^{(old)}\right) \text{ (4.15b)}$$

The matrix manipulation in step (1) and (2) makes the error ε_k we want to solve for strongly dependent on $\Delta x_{i,k}$. Also, since there are random shufflings in the capacitors during the bit cycling mode, $dec_{iA,j}$ and $dec_{iB,j}$vary in an independent fashion. Therefore, the known coefficients $S_{i,k}$ of other errors vary in an independent fashion. This implies that the contribution of the summation terms in (4.15) to ε_{kA} and ε_{kB} will be small compared to $\Delta x_{i,k}$. Therefore, they can be safely ignored. The coefficient $S_{i,k}$ and $S_{i,k+50}$always stay positive because of the matrix manipulation in step 1. Therefore, solving ε_{kA} and $\varepsilon_{kB}$with (4.16) is justified.

$$\varepsilon_{kB} = \Delta x_{i,k} \quad (4.16a)$$

$$\varepsilon_{kA} = \Delta x_{i,k+50} \quad (4.16b)$$

The modified Jacobi method offers several advantages. Compared to the basic Jacobi Iterative method, it uses less memory resource because we do not need to store all the known coefficients $S_{i,j}$. It is much easier for hardware implementation because only simple negations, additions and division by power of 2 will be implemented. In the worst case scenario, 128 negations, 128 additions and 1 division by 2 are used. This is still much more resource and hardware friendly than implementing the basic Jacobi iterative method because there are no non-binary multiplications and divisions.

The beauty of this modified Jacobi method is that, as long as ε_{iA} and ε_{iB} move in the right directions in successive small steps, the large LMS loop will eventually drive ε_{iA} and ε_{iB} to zero. Thus, there is no need to wait until they converge to sufficiently accurate values before we move on to use the next set of 128 conversions to estimate their new values. Inherently, then, the modified Jacobi method is resistant to random system errors such as the kT/C noise.

Averaging (4.8a) and (4.8b), we obtain the average output code equation (4.17). From (4.17), the average output code x consists of the average output codes $\hat{x}_A$ and $\hat{x}_B$, and the average ADC errors. As the errors ε_{kA} and ε_{kB} converge to zero, Δx in (4.10) converges to zero. The output code x also converge to its real value because the second term in (4.17) vanishes.

$$\hat{x} = \frac{\sum dec_{iB} W_{iB} + \sum dec_{iA} W_{iA}}{2} + \frac{\sum dec_{iB}\, \varepsilon_{iB} + \sum dec_{iA}\, \varepsilon_{iA}}{2} \quad (4.17)$$

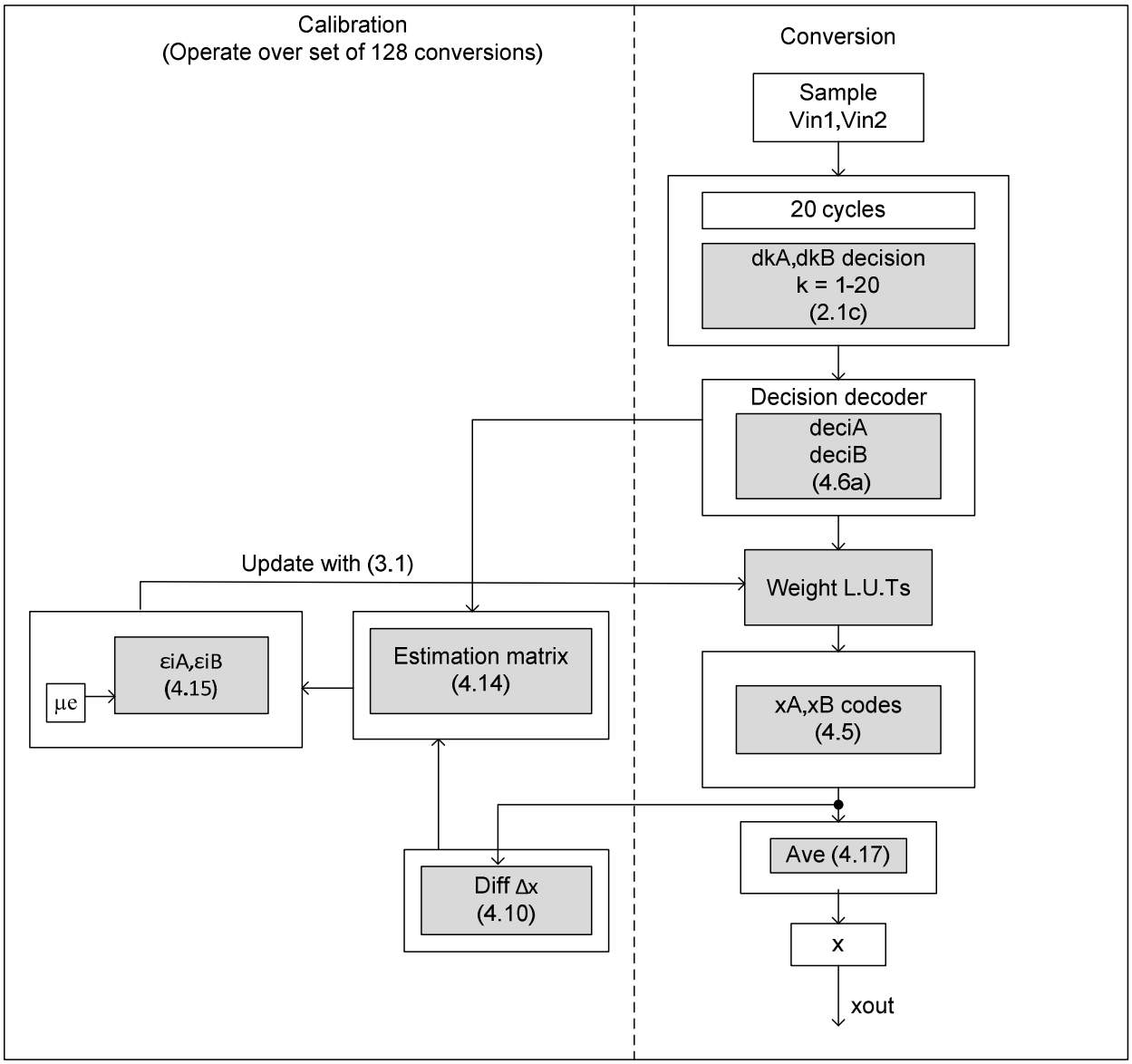


Figure 4.1: Summary of error correction algorithm

Figure 4.1 summarizes the entire calibration procedure, with the equations necessary to use in that step provided. The right hand side of the figure shows events that happen every conversion, while the left hand side of the figure shows events that happen every 128 conversions in order to update the calibration parameters $\widehat{W}_{iA}$ and $\widehat{W}_{iB}$.

During each conversion, the input Vin1, Vin2 is sampled onto both ADCA and ADCB. The comparators determine the decisions in 20 cycles according to (2.1c). Using (4.6a), the decisions dkA and dkB are then converted into dec_{iA} and dec_{iB}. Next, we obtained the output code $\hat{x}_A$ and $\hat{x}_B$ with (4.5). Finally, we averaged $\hat{x}_A$ and $\hat{x}_B$ by means of (4.17) to get the output code $\hat{x}$.

For each conversion, we calculate the difference Δx by (4.10). At the same time, we uses Δx, dec_{iA} and dec_{iB} from 128 conversions in an error estimation matrix defined by (4.14) to find ε_{iA} and ε_{iB}. Finally, we update $\widehat{W}_{iA}$ and $\widehat{W}_{iB}$ in the L.U.T by applying (3.1). This calibration procedure happens in every 128 conversions. It operates in the background and is completely transparent to the foreground operations.

4.4 Simulation results

In this section, we intend to show how the proposed calibration procedure improves the linearity of the ADC. In addition, we will explore how the choice of μ and the type of inputs used for calibration affect the speed and accuracy of the calibration procedure. The frequency response of the output signal, before and after calibration, will also be compared.

Matlab was used to simulate the system level operation and the error correction algorithm, as discussed in Section 3.3 and 3.5. Noise and capacitor mismatch errors were also added to the two ADCs to show how the error correction algorithm copes with these non-idealities. The typical mismatch for the 1pF unit capacitor is ±0.1%, while that for the 125fF unit capacitor is ± 0.2%. These typical mismatch values are estimated based on [36]. In this experiment, random capacitor mismatches about ×100 that of the typical mismatch were added to the capacitors. In addition, kT/C Noise of 30uV rms were added to the system during the sampling and bit cycling mode.

4.4.1 INL/DNL improvement

As long as the total voltage error caused by the noise and mismatch is less than the sum of the redundant bit (1022 LSB), we will always have enough weights to correct for the error in the output code. Figure 4.2a shows the INL/DNL plot of the SAR ADC before the error correction algorithm is applied. Even the mismatch in the DAC is 100 times that of the typical mismatch, we are able to recover its INL/DNL to around 0.5 LSB, once the LMS loop has adapted and converged to its steady state (Figure 4.2b).

Note that the noise and mismatch errors have a much stronger effect on the INLs than the DNLs. The Matlab code for this simulation can be found in Appendix D.

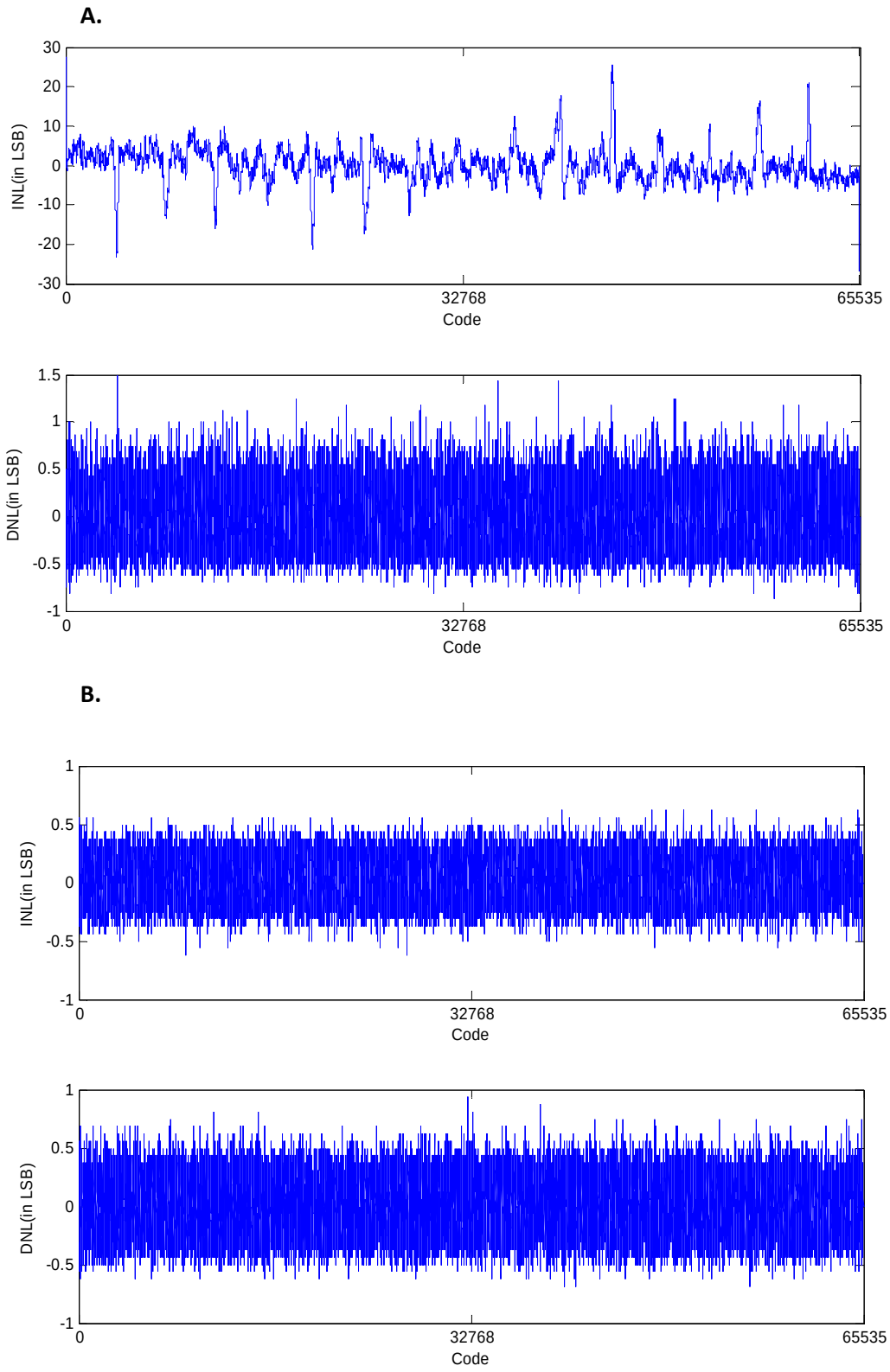


Figure 4.2: (A) INL/DNL plot of the split ADC before correction. (B) INL/DNL plot of the split ADC after correction.

4.4.2 Adaptation for various input signals

In our ADC architecture, a diversity of decision paths are allowed for one input. This is because we have used redundant bits in making decisions, and we chose unit capacitor randomly in the DAC segment to correspond to each bit decision. Therefore, calibration information can be extracted even for a DC input.

We expect to see some variation in convergence speed for different inputs. This is because the dynamics of the matrix iteration depend on the matrix coefficients in (4.13) and (4.14) [37], and the matrix coefficients are in turn determined by the input signals.

In [8], the "Split-ADC" architecture is applied to a 16 bit, 1 MS/s cyclic ADC. In [8], using a DC signal that is close to full scale for calibration poses a challenge. This is because the diversity of decision paths near the positive full range are limited. This in turn provides limited patterns for the matrix coefficients in the estimation matrix in [8] and significantly slows down the convergence.

In this section, we repeated the experiment in [8]. A sine wave, a DC signal (0.1FS and 0.9 FS), and a random signal is used as the calibration input signal. Figure 4.3 shows that both the average ADC error and the weight error of all inputs converge to their steady state at around 600,000 conversions, with a μ of 2^{13}. This contradicts with what we observe in [8].

These results show the advantage of using a differential SAR structure with the calibration algorithm. Since the common voltage of the differential SAR structure is centered at 1.25V, a 0.9FS DC signal will have one of its differential input sitting constantly at 2.375V, and the other differential input sitting constantly at 0.125V. Although the diversity of decision paths is limited at the input of 2.375V, there are many different decision paths for the input at 0.125V. Therefore, the restricted decision paths of one input are compensated by the many decision paths of the other input. This increases the dynamics for the matrix coefficients in (4.13) and (4.14). Therefore, using a full range DC input as a calibration signal no longer poses a challenge in the calibration algorithm.

This experiment proves that the proposed calibration algorithm works for any signal, continuous or not. The calibration time remains relatively the same for all inputs. In our case, with a μ of 2^{13}, the calibration time is about 600,000 conversions. This is more than three orders of magnitude faster than the traditional statistical method, which requires 2^{32} conversions for the calibration parameters to converge to their correct values. The Matlab code for this simulation can be found in Appendix D.

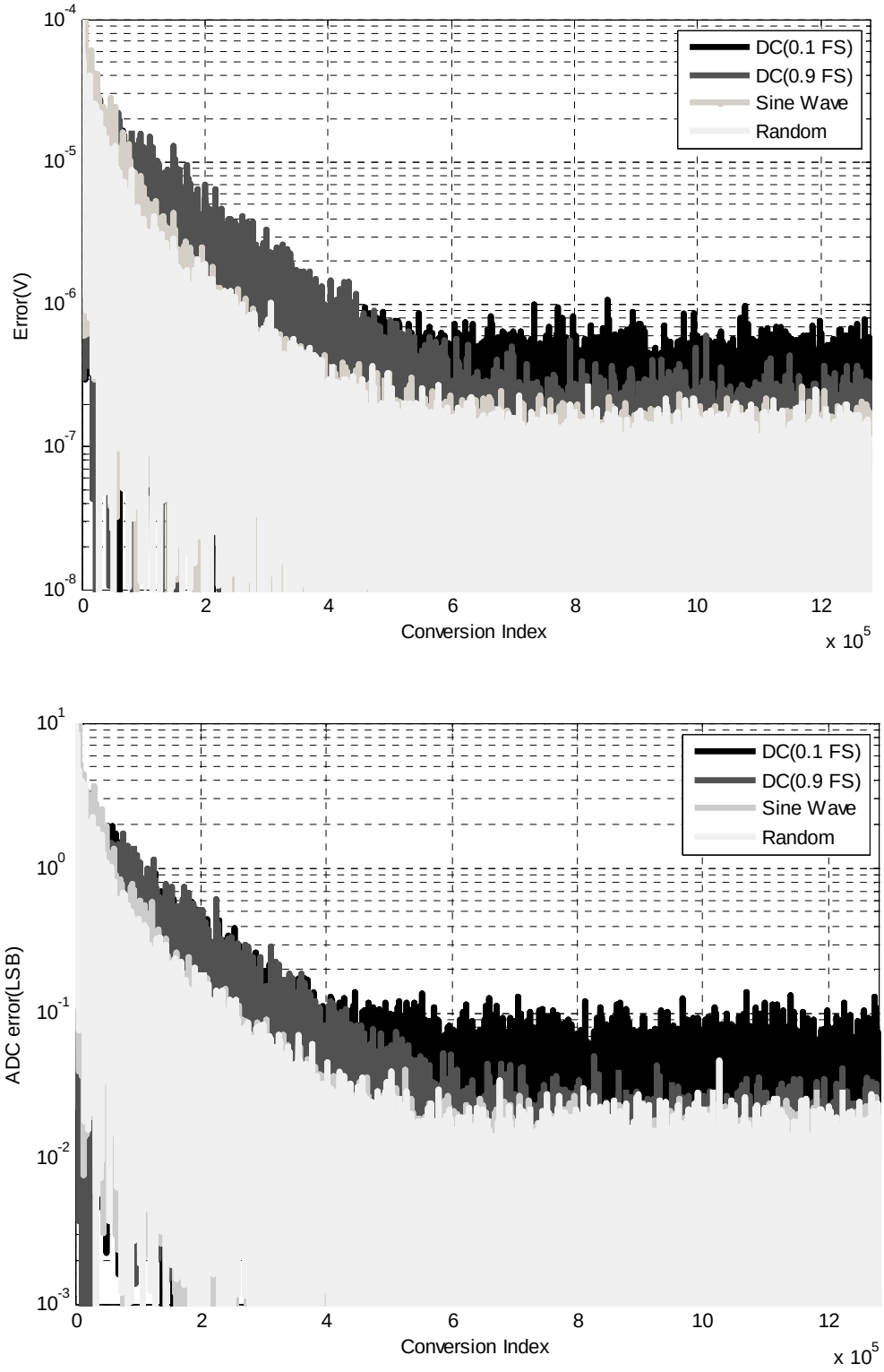


Figure 4.3: (A) The convergence of weight error for different inputs. (B) The convergence of ADC error for different inputs

4.4.3 LMS Parameter Selection

As mentioned in Section 3.3.2, the role of the LMS coefficient μ_e is to control the time constant of the calibration adaptation and is subject to a tradeoff between accuracy and speed of adaptation [32, 33]. In general, while a bigger μ_e help the estimated weights to converge to the true weights more quickly, it also makes the system more susceptible to noise. As a result, the system may overestimate the weights and there is a danger that the solution will diverge instead. A smaller μ_e helps averaging out the noise in the system and provides more accurate weight estimates. However, the tradeoff is that we need longer time for $\widehat{W}_{iA}$ and $\widehat{W}_{iB}$ to reach their equilibrium. μ_e is chosen to be a number divisible by 2. This is to simplify the digital hardware implementation.

Figure 4.4 and Table 4.1 shows the effect of the choice of the LMS coefficient μ_e on the speed and accuracy of the convergence of the weight error and the ADC error.

In our case, the weight error and the ADC error diverge when we use μ_e larger than 2^{-12}. Therefore, using $\mu_e = 2^{-12}$ is a good compromise as it provides fast convergence and a ADC error of less than 1 LSB in the steady state. Decreasing μ_e to 2^{-14} improves the accuracy of the weight estimate and the ADC error slightly, but it significantly slows down the speed of convergence. Increasing $\mu_e = 2^{-16}$ offers no further advantage on speed and accuracy.

In general, if the system noise is comparable to the mismatch error, one should use a smaller μ_e to filter out the noise. On the other hand, if the system noise is small compared to the mismatch, one should use a larger μ_e for faster convergence. The Matlab code for this simulation can be found in Appendix D.

Table 4.1a Tradeoff between convergence and accuracy of weight estimates.

	Convergence Speed (Conversions)	**Standard deviation σ (V)**
$\mu_e = 2^{-12}$	300,000	1.5407e-7
$\mu_e = 2^{-14}$	1,000,000	3.7498e-8
$\mu_e = 2^{-16}$	1,300,000	1.3564e-7

Table 4.1b Tradeoff between convergence and accuracy of ADC error.

	Convergence Speed (Conversions)	**Standard deviation σ (V)**
$\mu_e = 2^{-12}$	300,000	0.0157
$\mu_e = 2^{-14}$	1,000,000	0.0042
$\mu_e = 2^{-16}$	1,300,000	0.0141

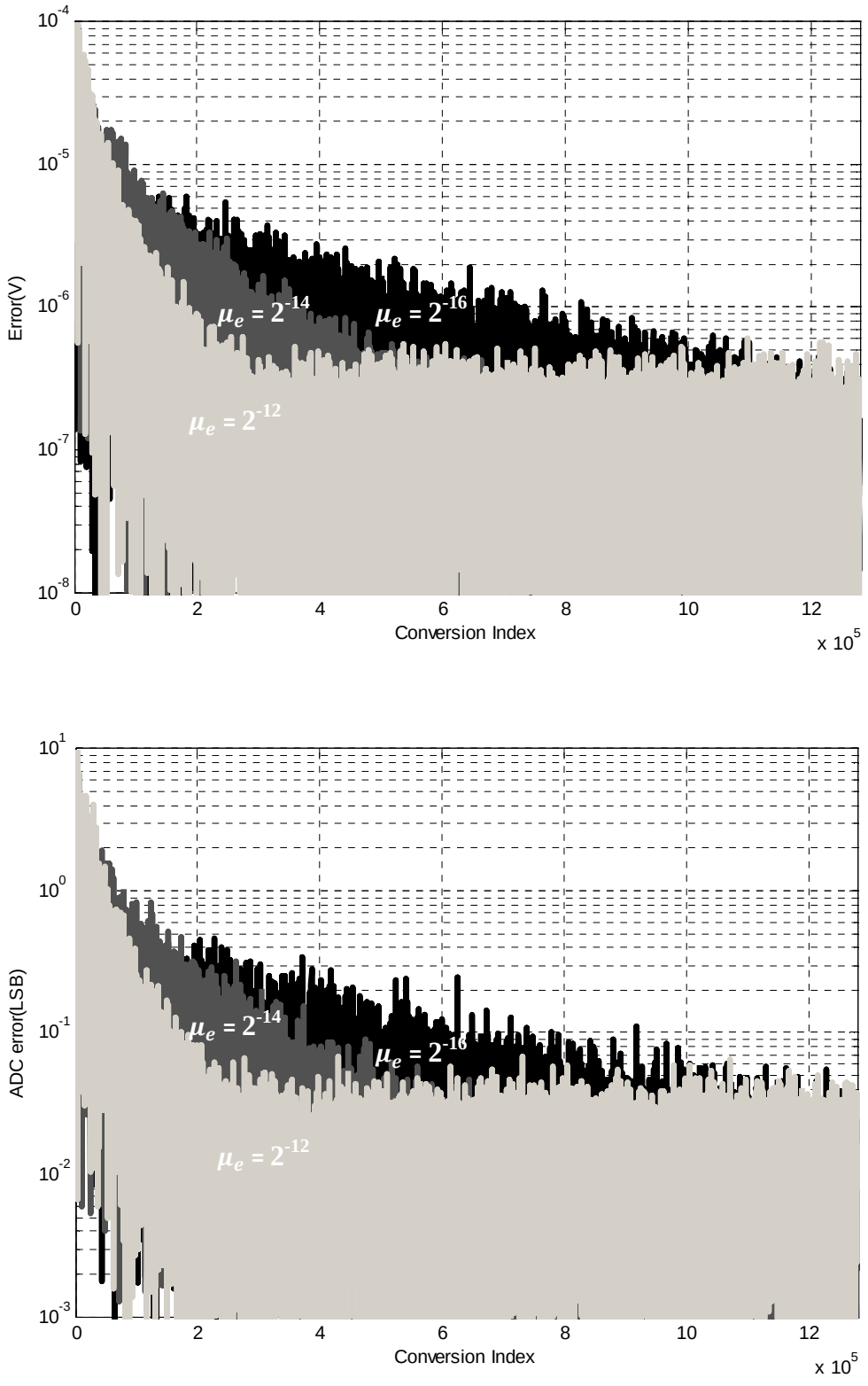


Figure 4.4: (A) Convergence of weight error for different μ_e. (B) Convergence of ADC error for different μ_e

4.4.4 Frequency Response

A 100 kHz sine wave (with 30uV rms noise) is applied to the un-calibrated and calibrated ADC. Figure 4.5 shows the frequency response of the ADC output. Note that we scale the magnitude of the fundamental frequency so that it has a magnitude of 1.

Without calibration, we see a third harmonic at the ADC output code. The noise floor is also much larger than we expected (Fig. 4.5a).

With calibration, the third harmonic at 300kHz is removed. Also, the noise floor is restored to the expected level of -120dB with respect to the fundamental frequency (Figure 4.5b).

The Matlab code used for this simulation can be found in Appendix D.

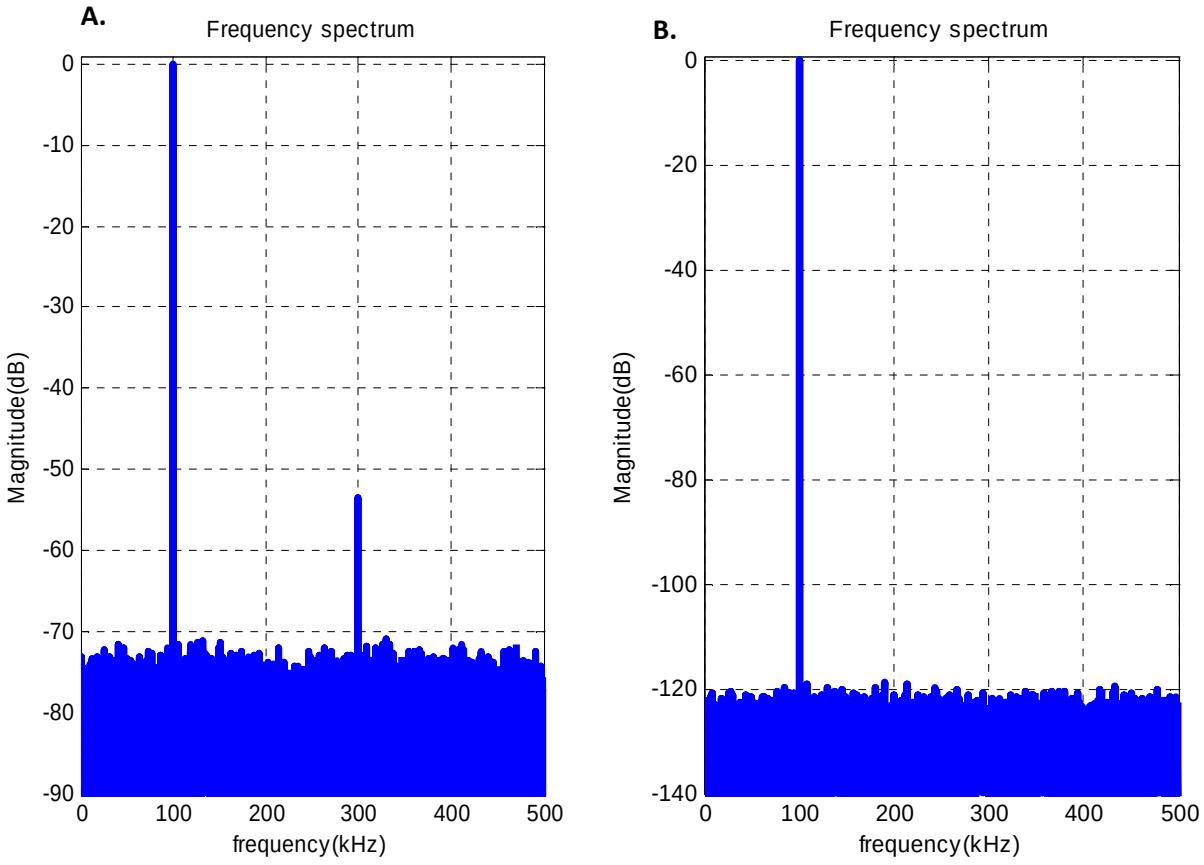


Figure 4.5: Frequency Response for (A) Un-calibrated ADC, and (B) Calibrated ADC

4.5 Summary

In this chapter, the theory of the error correction algorithm is presented. Using Matlab behavioral simulation, we applied the error correction algorithm to the split SAR ADC system in Chapter 3. Simulation results show that the error correction algorithm works with any unknown input, and that the use of the adaptive parameter μ_e allows us to adjust the calibration parameters with the optimal speed and accuracy. The choice of μ_e is sensitive to the relative magnitude between the system noise and the mismatch error. If the system noise is comparable to the mismatch error, one should use a smaller μ_e to filter out the noise. On the other hand, if the system noise is small compared to the mismatch, one should use a larger μ_e for faster convergence.

In general, the error correction algorithm can calibrate the ADC within 10^5-10^6 conversions. It is at least 3 orders of magnitude faster than the traditional statistical method, which requires 2^{32} conversions. There are no limitations on mismatch errors we can correct for. As long as the total voltage error caused by the capacitor mismatches are less than the sum of the redundant bits (1022 LSB), the INL/DNL can be restored to ±0.5 LSB.

5

Circuit Design

5.1 Introduction

This chapter starts by exploring non-idealities in the SAR converter and the SAR ADC IC. In Section 5.3, we set up the design specifications for the required 16-bit, 1 MS/s differential SAR converter to be used with the "Split-ADC" architecture. The prototype design is developed in the 0.25 um standard CMOS process. Section 5.4- 5.7 designed the DAC structure, the tapered buffer, the sample-and-hold circuit (S/H) and the DAC switches according to the specifications in Section 5.3. All these design are based on the ideal 16 bit 1 MS/s differential SAR ADC simulation we developed in Section 2.5. To prove that the prototype design work, we added all the non-idealities and design parameters in the ADC circuit in Section 5.8 and rerun the simulation. Finally, Section 5.9 concludes this chapter by discussing the design results.

5.2 Non-idealities of the 16-bit differential SAR A/D converter

5.2.1 Capacitor mismatch

The accuracy of a SAR ADC is limited by the DAC, and the accuracy of the DAC is in turn limited by how well the capacitor ratios match to each other. In the 16 bit, 1 MS/s differential SAR converter we developed, we have related the capacitor mismatch to the weight error ε_j in Section 3.3, and we have related the weight error ε_j to the output code $\hat{x}_i$ in (4.8).

For convenience, we rewrite (4.8) in (5.1), where the second term in (5.1) represents the voltage error in the output code due to the capacitor mismatch in the DAC.

$$\hat{x}_i = \sum dec_{i,j} \cdot W_j + \sum dec_{i,j} \cdot \varepsilon_j \quad (5.1)$$

We did not intend to improve the capacitor mismatch in the DAC by laser trimming or other advanced layout techniques. It is because the split ADC self-calibrating algorithm can remove the error term in (5.1) in the digital domain, as we have already proven in Chapter 4.

5.2.2 Noise

There are three main noise sources in the proposed 16-bit, 1 MS/s differential SAR converter. Theses noise sources are the kT/C noise from the DAC array, the inherent quantization noise of the ADC and noise from the comparator. Since the comparator is not the scope of this project, we will only analyze how the kT/C noise and the quantization noise affect the ADC.

We use (5.2) to find the kT/C noise from the DAC, where $\sigma_{rms,half}$ is the kT/C noise power from one split ADC, k is the Boltzmann constant 1.38x10-23 JK^{-1} and T is the room temperature in Kelvin [29].

$$\sigma_{rms,half} = \sqrt{\frac{kT}{C}} \quad (5.2)$$

Since the first segment of the proposed DAC in Section 2.4.1 contributed mainly to the kT/C, we put C = 16pF into (5.2). Therefore, the noise power $\sigma_{rms,half}$ of one DAC equals 16uV rms.

For a series of uncorrelated noise powers, the variance of their sums equals to the sum of their variances. Therefore, their total noise power is calculated by (5.3).

$$\sigma_z{}^2 = \sigma_x{}^2 + \sigma_y{}^2 \quad (5.3)$$

Since we have two DACs in the split ADC architecture, the total noise power σ_{rms} due to the DAC is then $\sqrt{2} \cdot 16uV\ rms$ = 22uV.

The quantization noise $V_{Q(rms)}$ [29] can be found by

$$V_{Q(rms)} = \frac{V_{LSB}}{\sqrt{12}} \quad (5.4)$$

Substituting V_{LSB}=76.3uV into (5.4), we get $V_{Q(rms)}$ = 22uV.

Therefore, the total noise power due to the kT/C noise and the quantization noise is 30uV.

5.2.3 Charge injection error

When a MOS transistor is on, it stores charges in its channel. When the MOS transistor is off, these unwanted charges are injected back into the circuit and cause errors in sensitive voltage nodes in the system. This is known as the charge injection error [29].

In this project, the charge injection error comes from sampling switches and the DAC switches. In this section, we will review the mechanisms and the effect of charge injection errors based on Figure 5.1 [34].

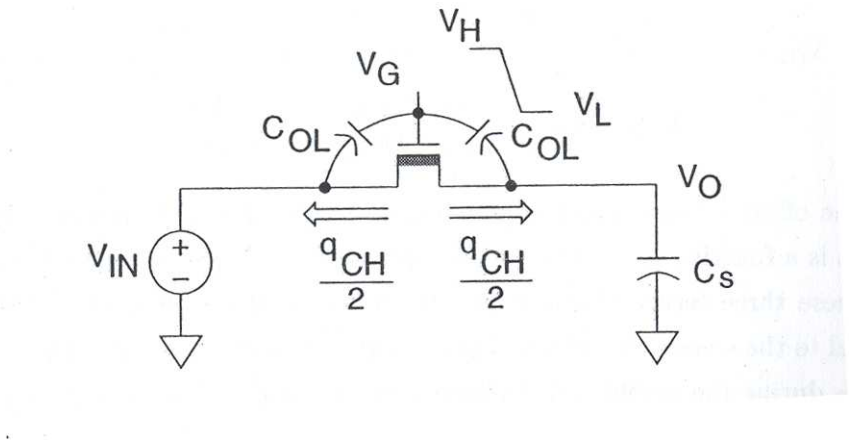


Figure 5.1: Charge injection error

Charge injection error occurs mainly by two mechanisms. The dominate mechanism is due to the channel charge which outflows from the channel region of the transistor to the drain and source junctions [29, 34].

The channel charge of a transistor that has zero VDS is given by

$$q_{CH} = -(V_H - V_{IN} - V_T) \cdot W \cdot L \cdot C_{ox} \quad (5.5)$$

Assuming the falling edge of the clock is very fast, the channel charge is split equally between the drain and the source. Thus, the error introduced at Vo, due to this mechanism, is given by:

$$\Delta V_1 = \frac{q_{CH}}{2Cs} = -\frac{1}{2}\frac{(V_H - V_{IN} - V_T) \cdot W \cdot L \cdot C_{ox}}{Cs} \quad (5.6)$$

The second source of error is the clock feed-through due to the capacitive divider formed by C_{oL} and Cs [34]. The error introduced at Vo due to this mechanism can be found by (5.7). Unless the effective voltage Veff is very small, this charge typically is not the dominant error [29].

$$\Delta V_2 = -(V_H - V_L) \cdot \frac{C_{oL}}{C_{oL} + Cs} \quad (5.7)$$

Considering these two sources of error, the total output voltage will be equal to

$$V_o = V_{IN} + \Delta V_1 + \Delta V_2 \quad (5.8)$$

Simplifying (5.8), we get

$$V_o = V_{IN}(1 - \epsilon) + V_{os} \quad (5.9)$$

where

$$\epsilon = \frac{1}{2} \cdot \frac{W \cdot L \cdot C_{ox}}{Cs} \quad (5.10)$$

and

$$V_{os} = -(V_H - V_L) \cdot \frac{C_{oL}}{C_{oL}+Cs} + \frac{(V_H - V_T) \cdot W \cdot L \cdot C_{ox}}{Cs} \quad (5.11)$$

Thus, the gain error ϵ is caused by a signal-dependent charge injection. The offset error V_{os}, on the other hand, is caused by both the signal independent charge injection and the clock feed-through error [34].

As explained in Figure 2.3, we turned essential MOS switches on and off during the sample, hold and bit cycling mode. Therefore, charge injection errors are continuously injected into the comparator node Vx and Vy, causing the comparator to make erroneous decisions as Vx-Vy approaches zero.

Fortunately, the differential structure of the ADC removes the linear portion of the charge injection error effectively. The structure of the sample-and-hold circuit, and the DAC action, on the other hand, removes most of the non-linear charge injection error, as will be explained in Section 5.6.

5.2.4 Harmonic distortion

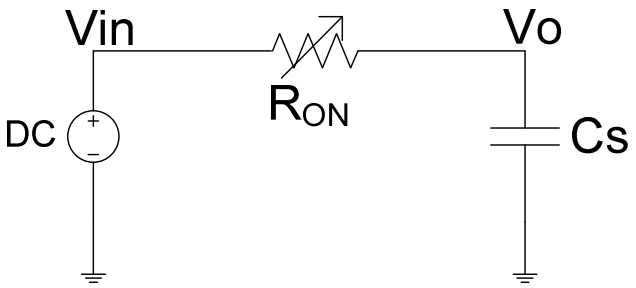


Figure 5.2: The impedance model of a simple sample-and-hold circuit

We used the impedance model of the simple sample-and-hold circuit in Figure 5.2 to explain the effect of harmonic distortion on the sample and hold circuit. In an ideal S/H circuit, the sampled voltage should be equal to the input voltage during the sampling mode. However, the MOS switches used for sampling has a nonlinear ON resistance R_{on} which is given by (5.12)

$$R_{on} = \frac{1}{\mu_n C_{ox} \frac{W}{L} (Vgs - Vth)} \quad (5.12)$$

where μ_n is the mobility of the N-channel MOSFET, Cox is the MOSFET gate capacitance, W and L is the length and width of the MOSFET, Vgs is the gate to source voltage and Vth is the threshold voltage of the MOSFET.

Vgs is the difference between the input voltage Vin and the gate voltage Vg. As Vgs change, the ON resistance R_{on} is also constantly changing, causing the transfer function $\frac{Vo}{Vin}$ of the sample and hold circuit to be non-linear.

We can use a pure sine wave to characterize the harmonic distortion of a sample-and-hold circuit [38]. If a pure sinusoid given as

$$V_{in}(\omega) = V_p \sin(\omega t) \quad (5.13)$$

is applied to the input, then the output of the sample-and-hold circuit with distortion will be

$$V_{out}(\omega) = a_1 V_p \sin(\omega t) + a_2 V_p \sin(2\omega t) + \cdots + a_n V_p \sin(n\omega t) \quad (5.14)$$

Harmonic distortion (HD) for the *i* th harmonic can be defined as the ratio of the magnitude of the *i* th harmonic to the magnitude of the fundamental frequency [38]. For example, the second-harmonic distortion would be given as

$$HD_2 = \frac{a_2}{a_1} \quad (5.15)$$

The total harmonic distortion (THD) is defined as the square root of the ratio of the sum of all of the second and higher harmonics to the magnitude of the fundamental harmonic [38]. Therefore, THD can be expressed as

$$THD = \frac{[{a_2}^2 + {a_3}^2 + \cdots + {a_n}^2]^{\frac{1}{2}}}{a_1} \quad (5.16)$$

In terms of dB, it can be expressed as

$$THD = 10\log\left(\frac{{a_2}^2 + {a_3}^2 + \cdots + {a_n}^2}{{a_1}^2}\right) \quad (5.17)$$

Since we did not consider the non-linearity due to the sample-and-hold circuit in the error correction algorithm in Chapter 4, we need to make the THD in the S/H circuit an insignificant factor in contributing to the output code error $\hat{x}_i$.

5.2.5 Non-idealities in signal source and bond wire

Any external inputs and outputs of the IC will be affected by non-idealities of the external signal sources and bond wires (external non-idealities). A pin used for an analog function tends to be more sensitive to these external non-idealities, because an analog operation is much more sensitive to signal degradation.

In our case, most of the analog operation lies with the DAC and the comparator. Therefore, we want to design the circuit such that the signal pins used for the DAC and comparator operation is

robust to the signal degradation by the external non-idealities. The most effective way to do this is to model these non-idealities and use them in the design process.

The external pin-outs used for the DAC and comparator operation are the differential inputs Vinp and Vinm, and the voltage references Vrefp, Vcm and Vrefm. The external non-idealities are modeled in Figure 5.3, where Rs and Cs are the voltage source's resistance and capacitance respectively, and Rb, Cb and Lb are the resistances, capacitances and inductances of the bond wires respectively.

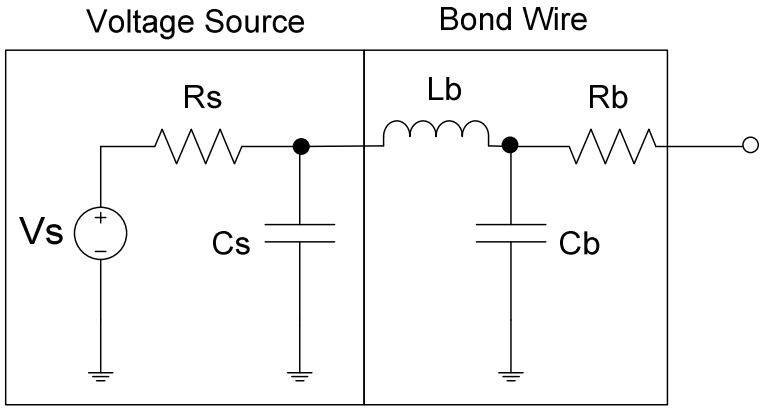


Figure 5.3: Modeling external non-idealities

Typically, Rs has a value ranges from 1mΩ-100mΩ, while Cs has a value of 10uF. Rb, Cb and Lb vary according to the bond wire length and diameter. The data in Appendix E provides the value of Rb, Cb and Lb at different bond wire length, at a fixed bond wire diameter of 1 mil, a die pad pitch at 75 um, a bond pad pitch at 160 um, and a mold compound dielectric constant of 3.9. The following simulation will be based on the bond wire data provided in Appendix E.

5.3 Design Specifications

The 16-bit, 1 MS/s differential SAR converter should be able to resolve two voltages within ± 76.3uV. It should have a linearity within ± 1 LSB, and a differential input voltage range of ±2.5V. It should finish 1 conversion in 1 us.

The kT/C noise from the DAC, the quantization noise of the ADC, and the comparator noise combined together should give a SNR no less than 90 dB.

The sample-and-hold circuit used in the ADC should have an acquisition time of 200ns, charge injection error of much less than 1 LSB, and a total harmonic distortion of -96dB.

The settling time of the DAC switches used during the bit cycling mode should be less than 15 ns, while those used during the hold mode should have a settling time shorter than the duration of the hold mode cycle.

Table 5.1 summarizes the design specifications of the desired 16 bit 1 MS/s differential SAR ADC.

Table 5.1 Design specifications

System Requirement	**Resolution**	76.3 uV
	Linearity	± 1 LSB
	Differential Input voltage range	±2.5V
	Total conversion time	1 us
	SNR	>=90 dB
S/H circuit requirement	**Sample mode acquisition time**	200ns
	Total Harmonic distortion in S/H	>=96 dB
	Charge injection error	<< 1 LSB
DAC switches requirement	**Settling time(Hold mode)**	< duration of hold mode
	Settling time(Bit cycling mode)	<15 ns

5.4 DAC design

5.4.1 A modified DAC structure

To design the DAC, we first estimate the minimum capacitance needed to achieve a SNR of 90dB. The SNR is given by

$$SNR = 20 \cdot \log\left(\frac{Vin_{rms}}{\sigma_{rms}}\right) \quad (5.18)$$

where σ_{rms} is the noise power of the total kT/C noise from both DAC arrays, and Vin_{rms} is the average input power [29]. The differential input signal range of our ADC is ±2.5V, so the average input power Vin_{rms} is 1.77V. Plugging in the required 90dB SNR into (5.18), we obtained σ_{rms} = 56 uV rms.

The kT/C noise of one split DAC, according to (5.3), is equal to 40 uV rms. Using (5.2), we find the minimum capacitance required for one split DAC is 2.6pF.

The smallest reasonable capacitor to be fabricated is 50fF. If we were to use the DAC structure in the basic 4-bit differential SAR converter in Section 2.2, we would need a total capacitance of 3.2768nF for the 16-bit converter. While this satisfies the minimum capacitance requirement for SNR, this occupies too much die area. Therefore, we come up with the alternative DAC structure in Figure 2.7. Figure 5.3 repeats the structure. The total capacitance of this new DAC is around 18pF. This satisfies the minimum capacitance requirement, while providing a reasonable layout area.

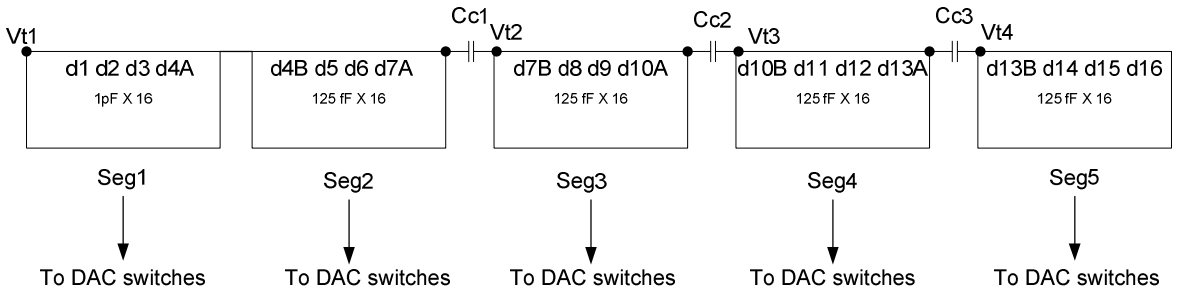


Figure 5.3 Modified DAC structure for the 16-bit differential SAR converter

As mentioned in Section 2.4.1, this new structure allow for the insertion of redundant bits . The use of unit capacitors in each segment also permit random selection of capacitor weights during bit cycling. The coupling capacitors, Cc1, Cc2 and Cc3, are used as attenuating capacitors to save area, and they are chosen such that the total capacitance of the left segment is 8 times the total capacitance of the right segment.

One should note the placement of the capacitors in the DAC. The top plate voltage Vt1 of the DAC is constantly varying during the bit cycling mode, it is therefore very susceptible to parasitic capacitances. Thus, the top plate of the unit capacitors are connected to Vt1,Vt2,Vt3 and Vt4 to reduce the effect of parasitic capacitances.

For the coupling capacitors, the bottom plate of Cc1 is connected to Vt1 because its parasitic capacitance can be easily removed by the DAC action. The bottom plate of Cc2 and Cc3 are connected away from Vt1, since the DAC weights in segment 4 and segment 5 hardly affect the top plate voltages of the DAC.

5.4.2 Deriving analytical expressions to find Cc1, Cc2 and Cc3

To find the value of Cc1, Cc2 and Cc3, we first group the unit capacitors in each segment together. We then supply a voltage source in each segment and find out how they affect the top plate voltage ΔV_{DAC}. We consider 5 cases.

Case 1: The relationship between v1 and ΔV_{DAC}

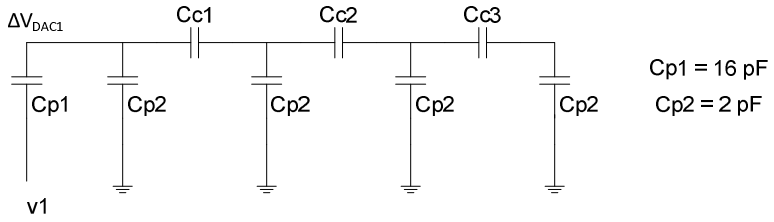


Figure 5.4 Relating v1 and ΔV_{DAC}

$$\Delta V_{DAC1} = \frac{Cp1}{Cp1+Z1} \cdot v1 \quad (5.19)$$

$$Z1 = Cp2 + Cc1 \parallel [Cp2 + Cc2 \parallel \{Cp2 + Cp2 \parallel Cc3\}] \quad (5.20)$$

Case 2: The relationship between v2 and ΔV_{DAC}

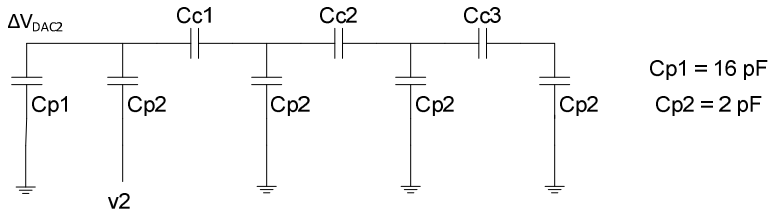


Figure 5.5 Relating v2 and ΔV_{DAC}

$$\Delta V_{DAC2} = \frac{Cp2}{Cp1+Cp2+Z2} \cdot v2 \quad (5.21)$$

$$Z2 = Cc1 \parallel [Cp2 + Cc2 \parallel \{Cp2 + Cp2 \parallel Cc3\}] \quad (5.22)$$

Case 3: The relationship between v3 and ΔV_{DAC}

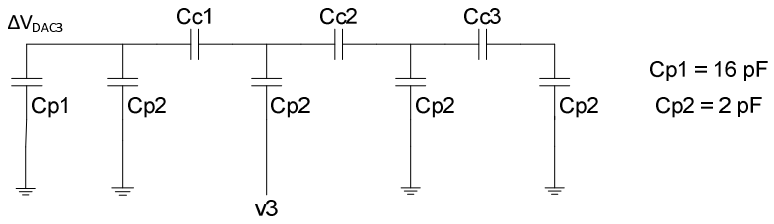


Figure 5.6 Relating v3 and ΔV_{DAC}

$$\Delta V_{DAC3} = \frac{Cc1}{Cc1+Cp1+Cp2} \cdot \frac{Cp2}{Cp2+Z3+Z3''} \cdot v3 \quad (5.23)$$

$$Z3 = Cc2 \parallel [Cp2 + Cp2 \parallel Cc3] \quad (5.24)$$

$$Z3'' = [Cp1 + Cp2] \parallel Cc1 \quad (5.25)$$

Case 4: The relationship between v4 and ΔV_{DAC}

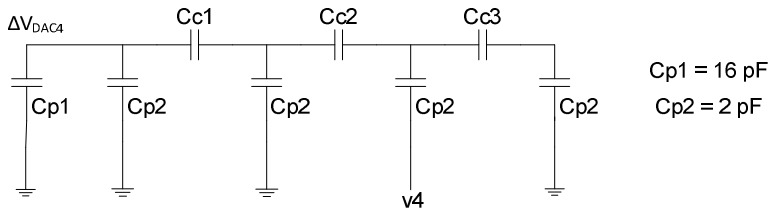


Figure 5.7 Relating v4 and ΔV_{DAC}

$$\Delta V_{DAC4} = \frac{Cc1}{Cc1+Cp1+Cp2} \cdot \frac{Cc2}{Cc2+Z3''+Cp2} \cdot \frac{Cp2}{Cp2+Z4+Z4''} \cdot v4 \quad (5.26)$$

$$Z4 = Cc3 \parallel Cp2 \quad (5.27)$$

$$Z4'' = Cc2 \parallel [Cp2 + Cc1 \parallel \{Cp1 + Cp2\}] \quad (5.28)$$

Case 5: The relationship between v5 and ΔV_{DAC}

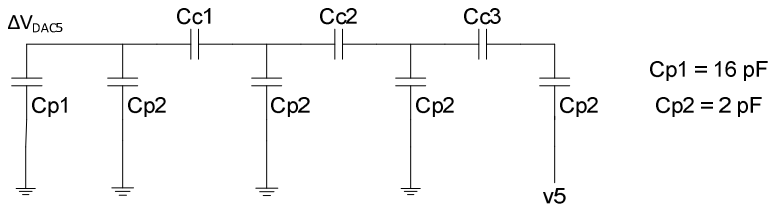


Figure 5.8 Relating v5 and ΔV_{DAC}

$$\Delta V_{DAC5} = \frac{Cc1}{Cc1+Cp1+Cp2} \cdot \frac{Cc2}{Cc2+Z3''+Cp2} \cdot \frac{Cc3}{Cc3+Z4''+Cp2} \cdot \frac{Cp2}{Cp2+Z1''} \cdot v5 \quad (5.29)$$

$$Z1'' = Cc3 \parallel [Cp2 + \{Cc2 \parallel < Cp2 + (Cp1 + Cp2) \parallel Cc1 >\}] \quad (5.30)$$

We know that

$$\frac{\Delta V_{DAC2}}{\Delta V_{DAC3}} = 8 \quad (5.31)$$

$$\frac{\Delta V_{DAC3}}{\Delta V_{DAC4}} = 8 \quad (5.32)$$

$$\frac{\Delta V_{DAC4}}{\Delta V_{DAC5}} = 8 \quad (5.33)$$

Therefore, substituting (5.21),(5.23),(5.26) and (5.29) into (5.31)-(5.33), we get

$$Cc1 = 325.89\ fF$$

$$Cc2 = 321.4\ fF$$

$$Cc3 = 285.71\ fF$$

The Matlab code for solving these equations can be found in Appendix F.

5.5 Tapered buffer design

Large capacitive loads are common in CMOS integrated circuits. These large loads occur both on chip, where high, localized fan-out and long global interconnect lines are common, and off-chip, where highly capacitive chip-to-chip communication lines exist [39]. In order to drive these large capacitive loads at high speed, tapered buffers are needed.

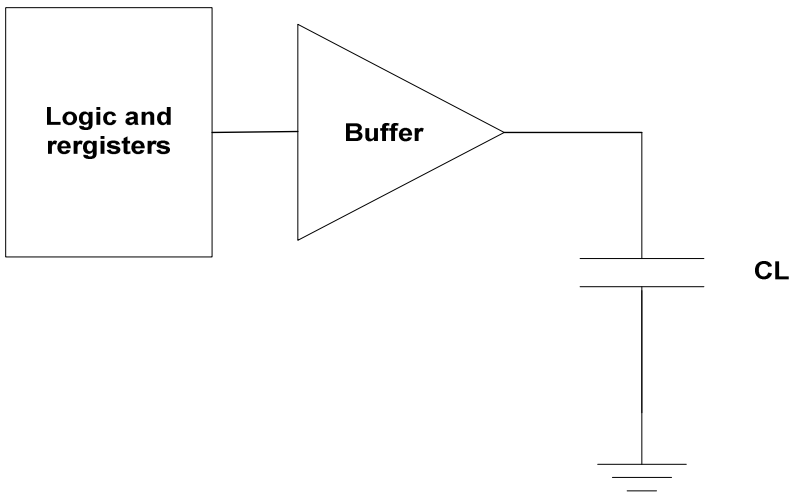


Figure 5.9: Tapered Buffer

Figure 5.9 shows the function of a tapered buffer. It is basically a series of inverters that are used to isolate the high impedance logic lines from the large capacitive load. It provides a high impedance input, so as to prevent the previous logic/registers stage from loading down. Meanwhile, it provides a low impedance output to drive the capacitive loads with sufficient currents [39].

All our clocks and switch signals are driven by tapered buffers. Note that we do not care about optimizing the performance of the tapered buffers, but only scaled them approximately to provide sufficient speeds and a reasonable rise time/ fall time for the logic signals. We will give an example of how the tapered buffers are used in the sample-and-hold ciruit. Figure 5.10 shows the bottom-plate sample-and-hold circuit we used in our ADC design. First, M1 is turned on, followed by M2 and M3 during the sampling mode. During the hold mode, M1 is first turned off, followed by M2 and M3. Figure 5.11 shows its waveform during the sample mode and the hold mode. The reason of why the waveforms φ_{1A}, φ_1 and φ_{1B} follow this particular sequence will be explained in Section 5.6. For now, let's focus on how we build a tapered buffer system for this sample-and-hold circuit.

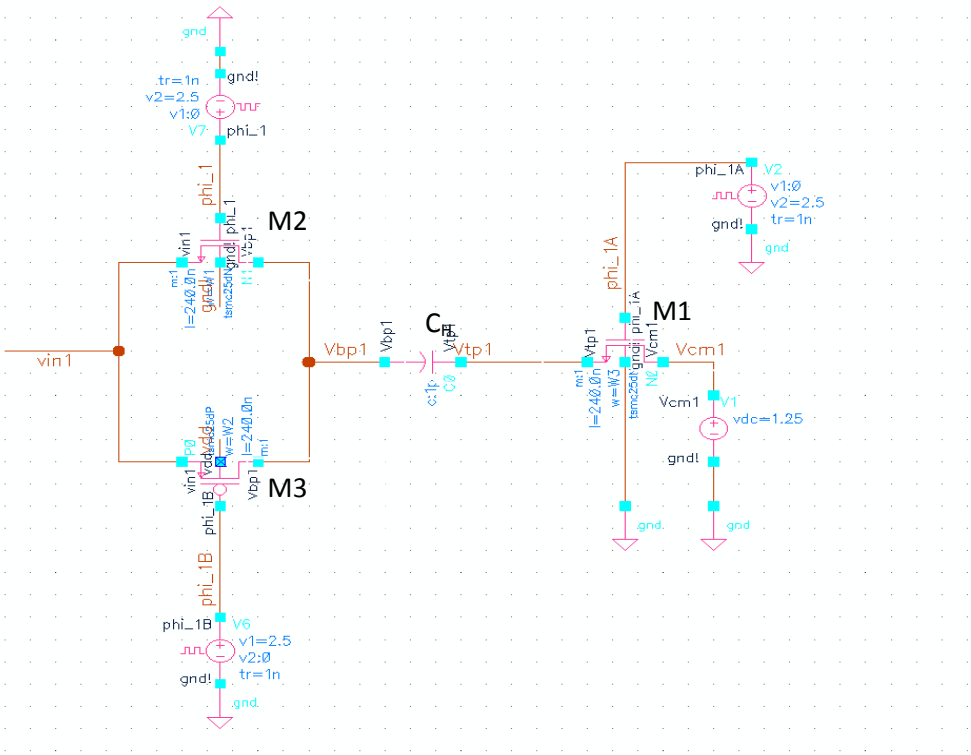


Figure 5.10: The bottom plate sample-and-hold circuit

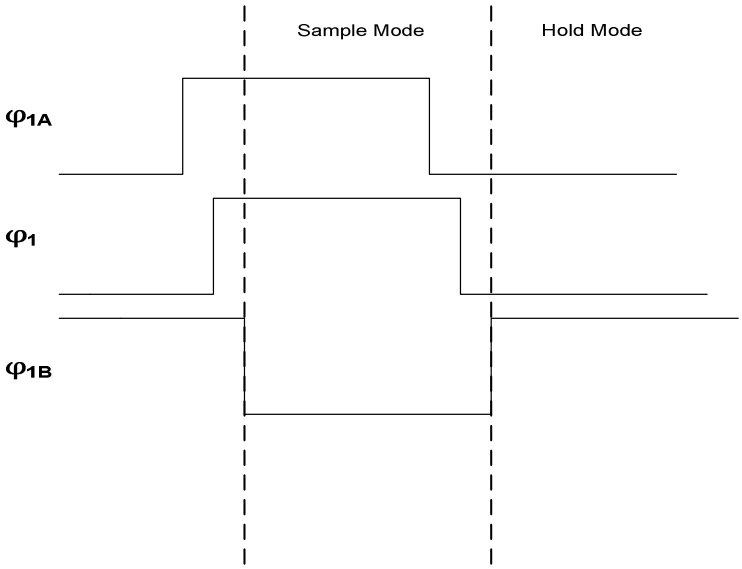


Figure 5.11: Signal waveform controlling the sample-and-hold circuit

In this example, we know that the capacitive load of M3 is 4 times that of M2, and that the capacitive load of M2 is bigger than M1. Based on this information, we scale the tapered buffer such that the inverter in the next stage is 4 times bigger than the inverter in the previous stage. Figure 5.12 shows how the tapered buffer is built for the sample-and-hold circuit in Figure 5.10.

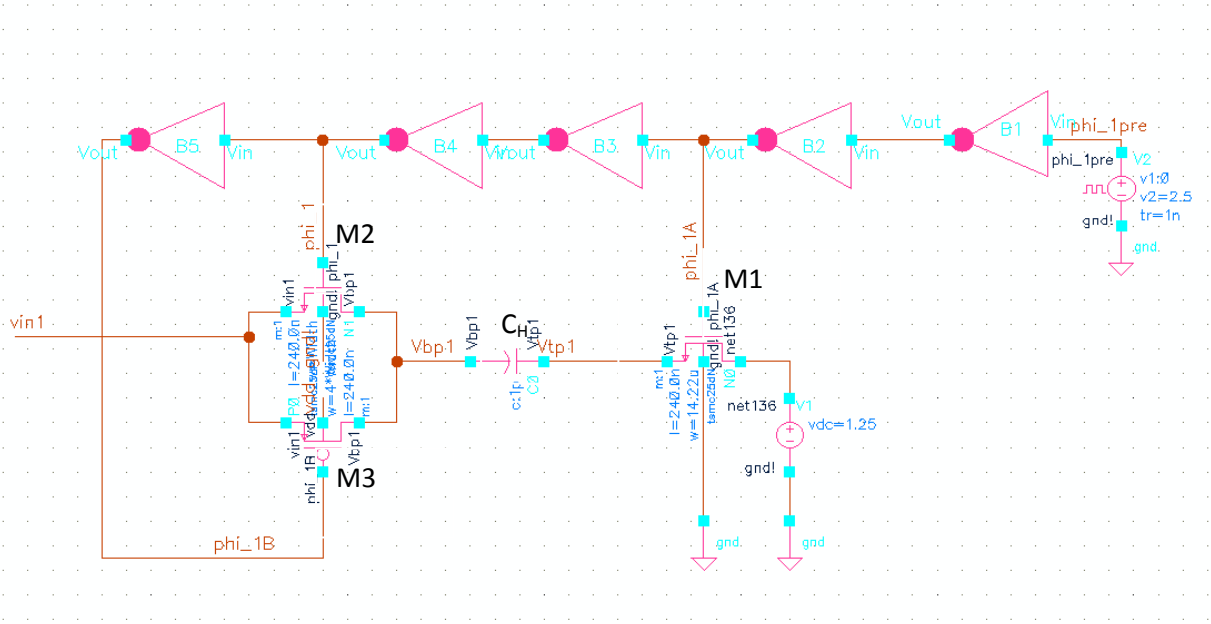


Figure 5.12: Tapered buffer used in the sample-and-hold circuit

Only one signal source φ_{1pre} is needed to drive the system, where its rise time and fall time is set to be 1 ns. The width of the minimum inverter B1 is 0.36um, and the width of M1, M2 and M3 are 233.3u, 933.1u and 113.6u respectively. We used 2 inverters (B1 and B2) to drive the smallest load M1. We used four inverters (B1-B4) to drive the larger load M2, and we used five inverters (B1-B5) to drive the largest capacitive load M3.

Figure 5.13 shows the simulated waveforms φ_{1A}, φ_1 and φ_{1B} during the sample and hold mode. Note that they each have a rise time and fall time of around 200-500ps. Also note that these waveforms follow the timing sequence in Figure 5.11, because of the inherent delay in each inverter.

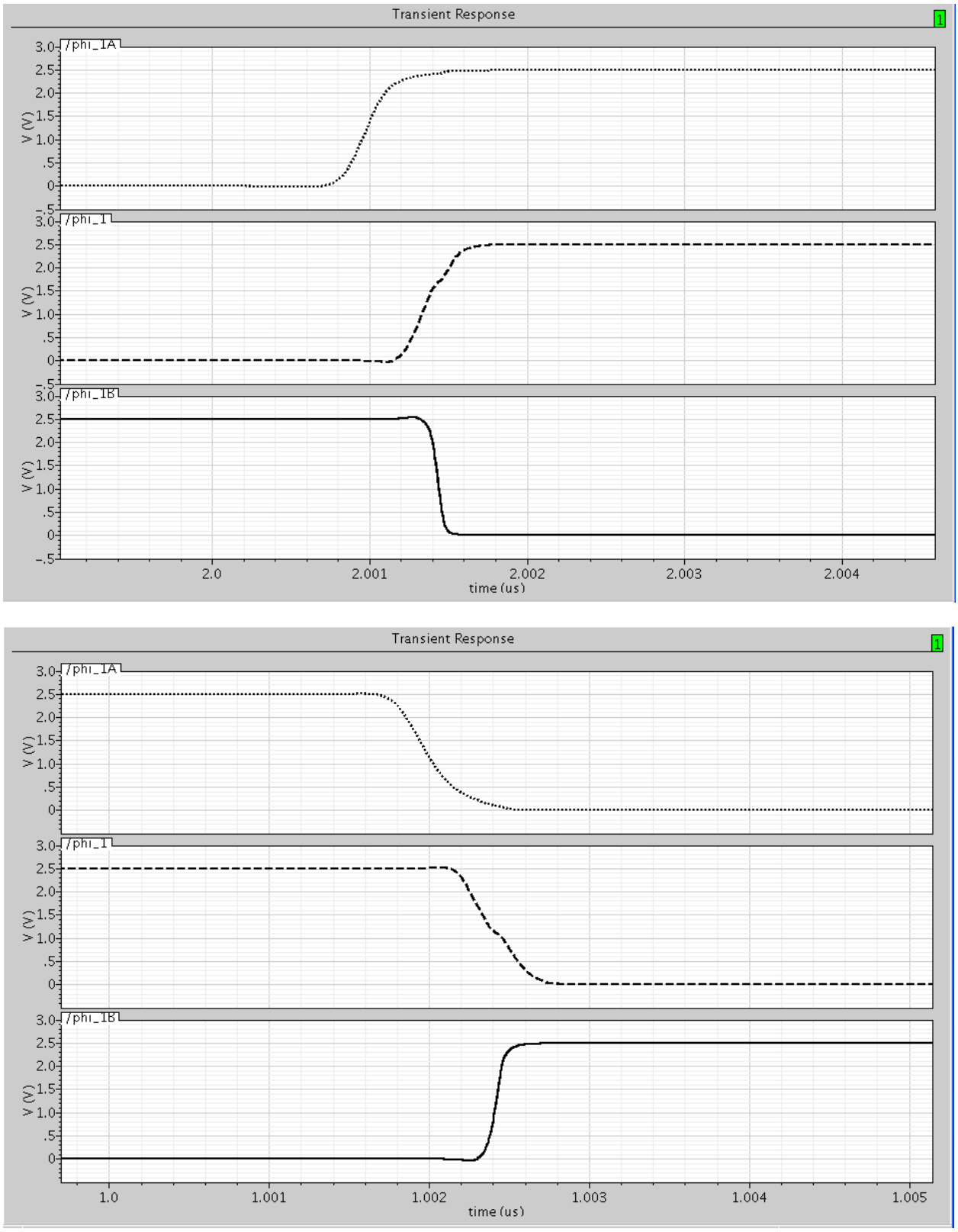


Figure 5.13: (A) Waveform of φ_{1A}, φ_1 and φ_{1B} during sample mode. (B) Waveform of φ_{1A}, φ_1 and φ_{1B} during the hold mode.

5.6 Bottom-Plate Sample-and-Hold circuit

5.6.1 Mechanism to remove charge injection error

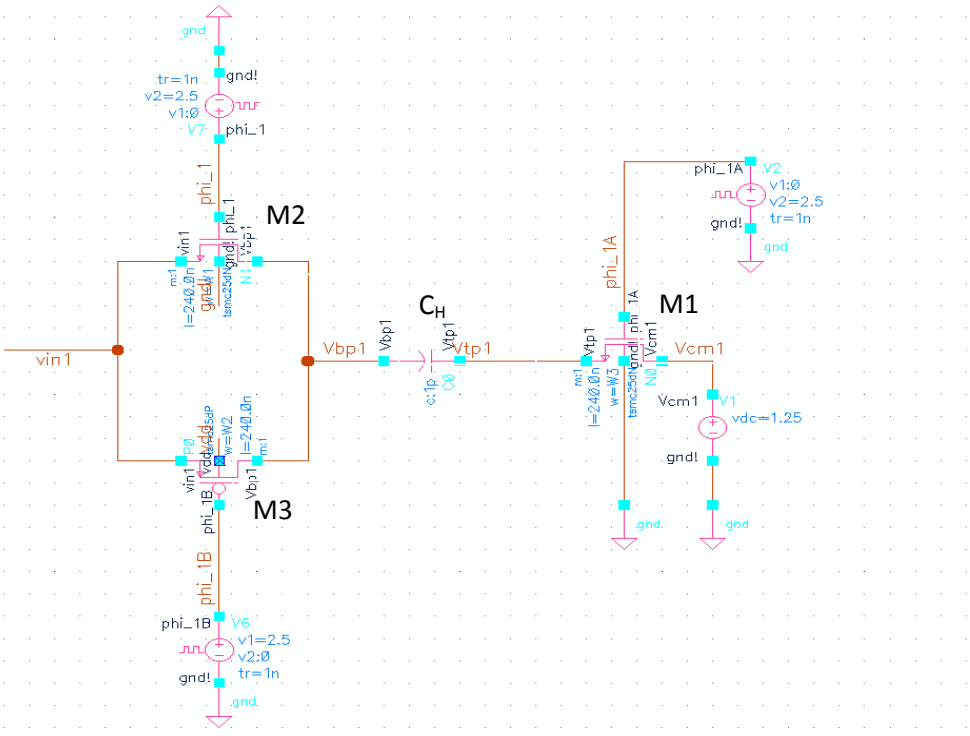


Figure 5.14: Bottom plate sample-and-hold circuit

Figure 5.10 shows the bottom plate sample-and-hold circuit. It is repeated in Figure 5.14 for convenience. Compared to the basic top plate sample-and-hold circuit shown in Figure 5.1, this circuit can effectively remove the charge injection error.

In Section 5.5, we briefly explained that during the sample mode, M1 is first turned on, followed by M2 and M3. Thus, the top plate of C_H equals the common mode voltage Vcm, and the bottom plate of C_H equals Vin.

During the hold mode, M1 is first turned off, followed by M2 and M3. Therefore, C_H holds the voltage Vin-Vcm, plus a voltage error due to the charge injection. Nevertheless, the structure of the bottom plate sample-and-hold circuit and the special switching sequences reduce the charge injection error.

When M1 is turned off, extra charge is injected into the node Vtp1, causing the voltage Vtp1 to change. The charge from M1, however, is signal independent. It can therefore be treated like a common mode error and removed by the differential SAR structure.

When M2 and M3 is turned off, their charges can only go through the low impedance Vin path, because M1 has turned off and make the path to the right a high impedance one. Therefore, the signal dependent charge injection error does not affect the voltage stored on C_H.

Of course, in reality, there are parasitic capacitances on both sides of C_H. Thus, there would be some signal dependent charges, as well as signal independent charges reside on both Vtp1 and Vbp1.

These charges can be further removed by the DAC action. Figure 5.15 shows the bottom-plate sample-and-hold circuit, together with the DAC switches used during the hold and bit cycling action. Vcm, Vrefm and Vrefp represent the common mode voltage, the negative voltage reference and the positive voltage reference of the ADC.

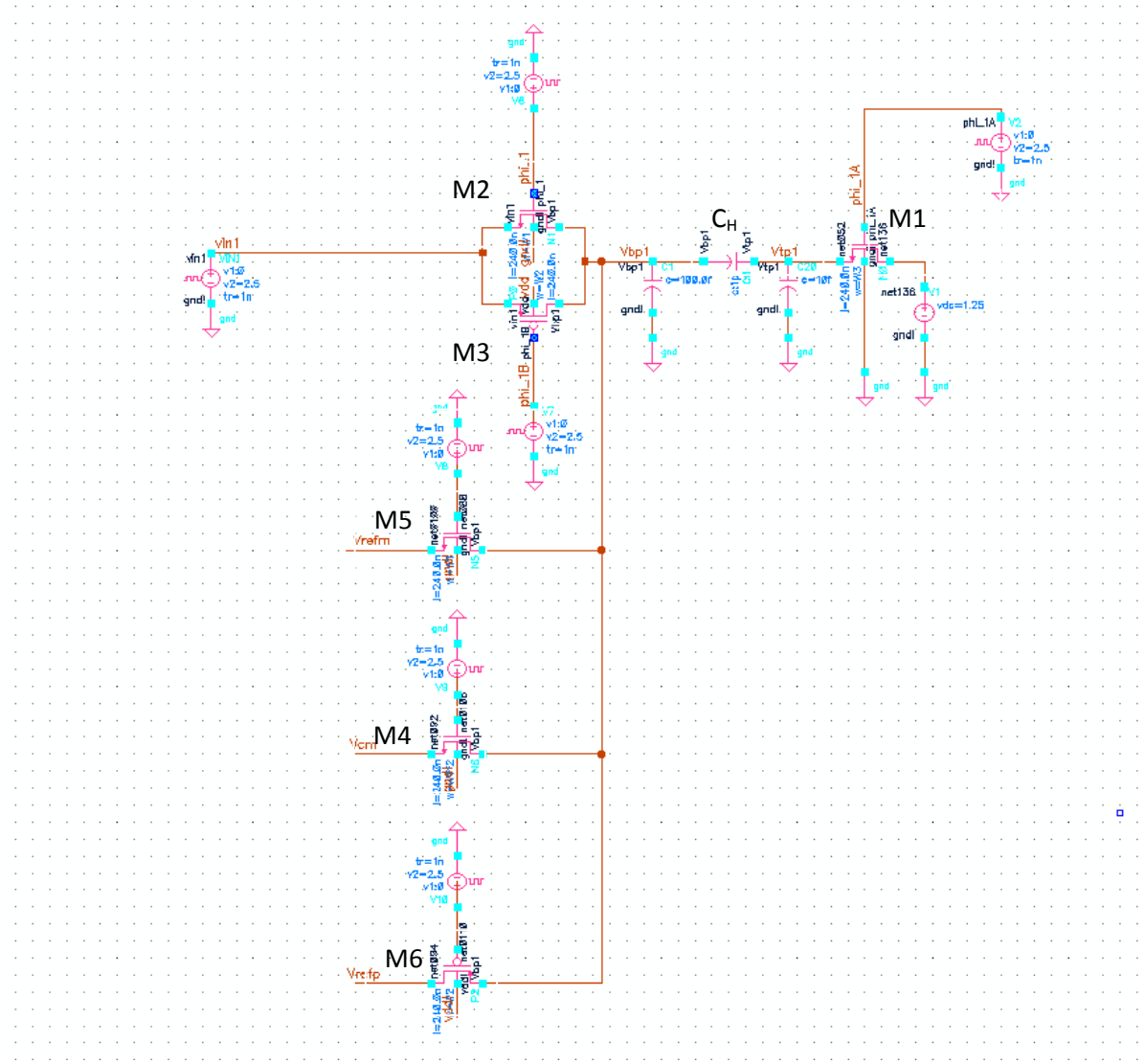


Figure 5.15: The sample-and-hold circuit, together with the DAC switches for a unit capacitor

As explained in the ADC operation in Section 2.2, after we turned off the sampling switches M1, M2 and M3, we turn on M4 during the hold mode. Thus, the non-linear charge from Vbp1 and Vtp1 can escape through the low impedance path Vcm. During the subsequent bit-cycling operation, either M5 or M6 is turned on in each unit capacitor cell. This form a low impedance path Vrefp or Vrefm that will further remove the non-linear charge reside on Vtp. Thus, the top plate Vtp is forced to return to the common mode voltage Vcm by the DAC action eventually.

5.6.2 Bottom-Plate Sample-and-Hold circuit Design

To design a bottom plate sample-and-hold circuit that meets the charge injection specification in Table 5.1, we need to reduce the size of M1, M2 and M3 such that the nonlinear charge reside on the top plate does not affect the outcomes of the comparisons during the bit cycling mode.

The harmonic distortion and acquisition time requirement, however, places contradictive requirement on the size of M1, M2 and M3. The combinational resistance and capacitance of M1, M2 and M3 has to be small enough to meet the 200ns acquisition time requirement. On the other hand, the non-linear resistances of M2 and M3 have to be small enough to meet the -96dB harmonic distortion requirement. This means we should increase the size of M1, M2 and M3.

From preliminary design results, we found that the harmonic distortion is a limiting factor of the sampling switch sizes, because an acquisition time of 200 ns can be easily obtained by adjusting the value of a band-limiting resistor r, and the charge injection error requirement can be easily satisfied as long as the MOS switches are reasonably sized. Therefore, we first determine the sampling switches' sizes based on the harmonic distortion requirement.

Figure 5.16 shows the sample-and-hold circuit formed by one unit capacitor cell. We will use this as a first approximation to find out the value of M1, M2 and M3. This cell is from segment one and has a C_H value of 1 pF. We added the tapered buffer, the DAC switches, the external non-idealities, the parasitic capacitances on both sides of C_H, and the band-limiting resistor r to the circuit to better approximate the harmonic distortion. We supplied a differential sine input that has magnitude of ±2.5V and a frequency of 500 kHz. We then characterized the total harmonic distortion of the sampled differential input voltage (Vbp1-Vtp1)-(Vbp2-Vtp2) with (5.17), where Vbp2 and Vtp2 is the top plate and bottom plate voltage of C_H of the other differential half. Appendix G shows the Matlab code used to find the harmonic distortion from the simulated data.

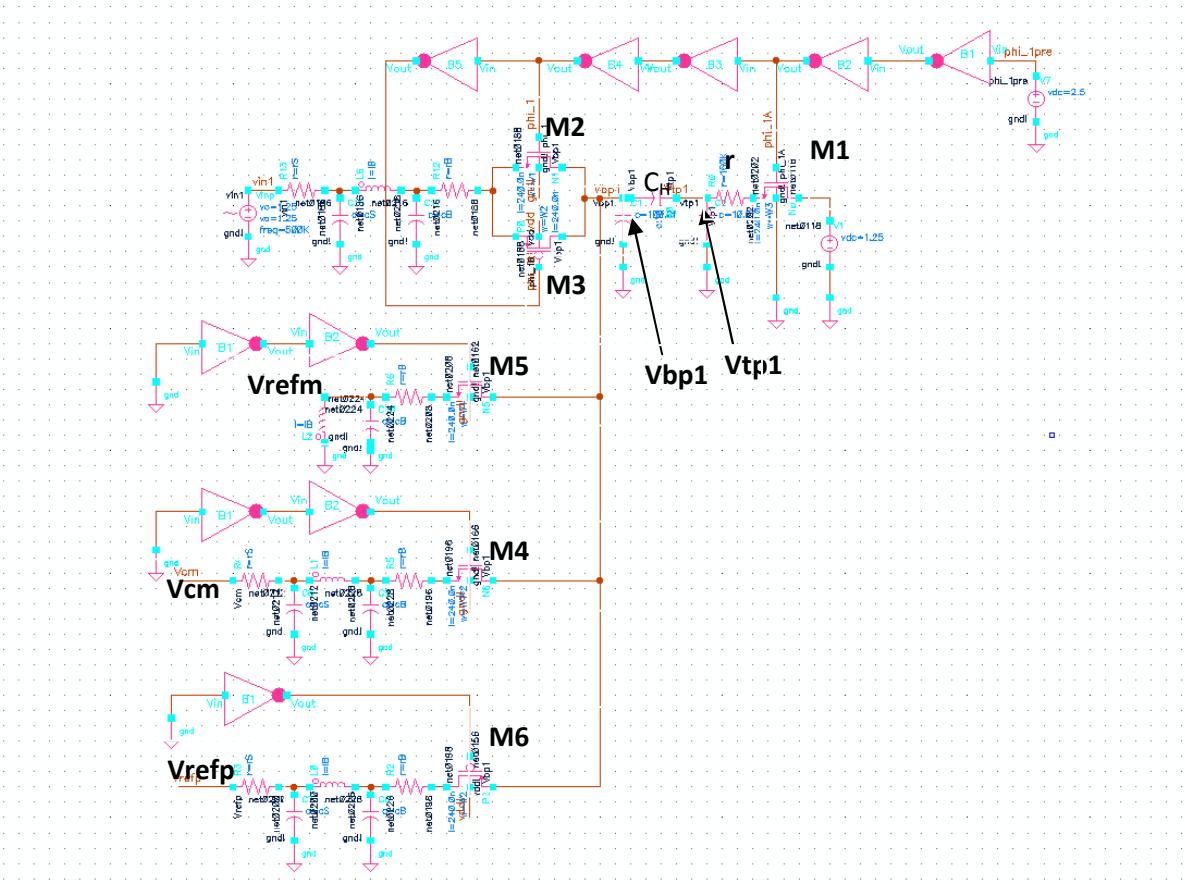


Figure 5.16 Harmonic distortion of one DAC cell

We first allocate a fixed width of 10 um to M1, M2 and M3 respectively. Table 5.2 shows that as we allocate more widths to M2 and M3, the harmonic distortion improve significantly. The size of M2 and M3 are therefore the limiting factor of harmonic distortion.

Table 5.2 Harmonic distortion VS switch sizes of M1, M2 and M3

	Tgate		Top Switch	Total Area	THD
	NMOS(um)	PMOS(um)	NMOS(um)		
	M2	M3	M1	(um^2)	(dB)
W	0.36	1.44	8.2	2.4	-58.63270
W	0.54	2.16	7.3	2.4	-61.79200
W	0.81	3.24	5.95	2.4	-65.04580
W	1.215	4.86	3.925	2.4	-68.33010
W	1.82250	7.29	0.8875	2.4	-71.36940

Knowing that the size of M2 and M3 are the limiting factor of harmonic distortion in the sample-and-hold circuit, we then increase the area of M2 and M3 until a total harmonic distortion of -96 dB is reached. Table 5.3 shows that we need to choose M2 = 233.28um and M3 = 933.12um to achieve a total harmonic distortion of -96 dB. We may need to reduce the size of M1 for the charge injection error reason.

Table 5.3 Increasing M1, M2 and M3 to meet the harmonic distortion requirement

	Tgate		**Top Switch**	**Total Area**	**THD**
	NMOS(um)	**PMOS(um)**	**NMOS(um)**		
	M2	**M3**	**M1**	**(um^2)**	**(dB)**
W	1.82250	7.29	0.8875	2.4	-71.36940
W	3.645	14.58	1.775	4.8	-77.01000
W	7.29	29.16	3.55	9.6	-82.28450
W	14.58	58.32	7.1	19.2	-86.97810
W	29.16	116.64	14.2	38.4	-90.81330
W	58.32	233.28	28.4	76.8	-93.64570
W	116.64	466.56	56.8	153.6	-95.45720
W	233.28	933.12	113.6	307.2	-96.51760

5.7 Designing the DAC switches

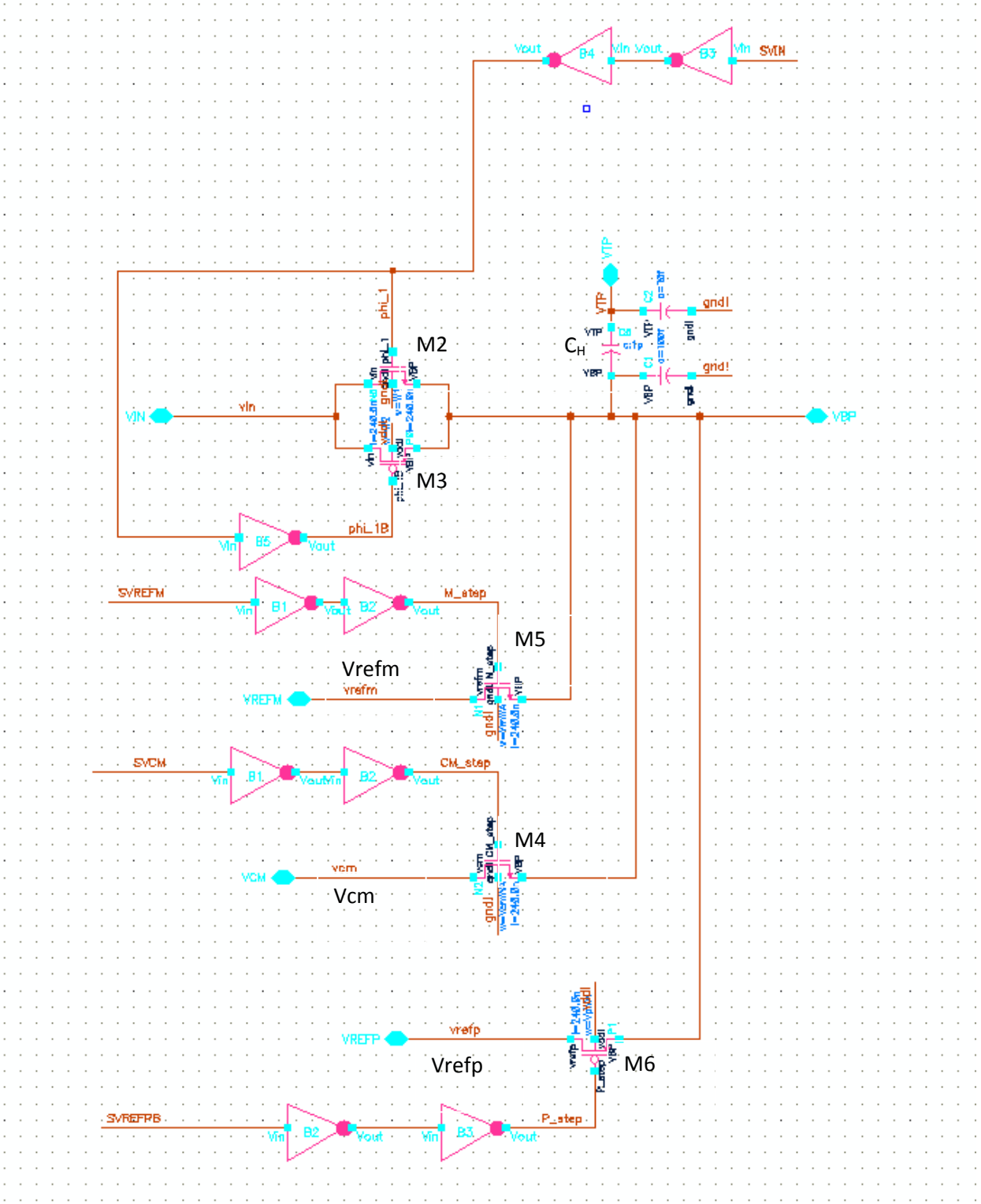


Figure 5.17 DAC cell in segment 1

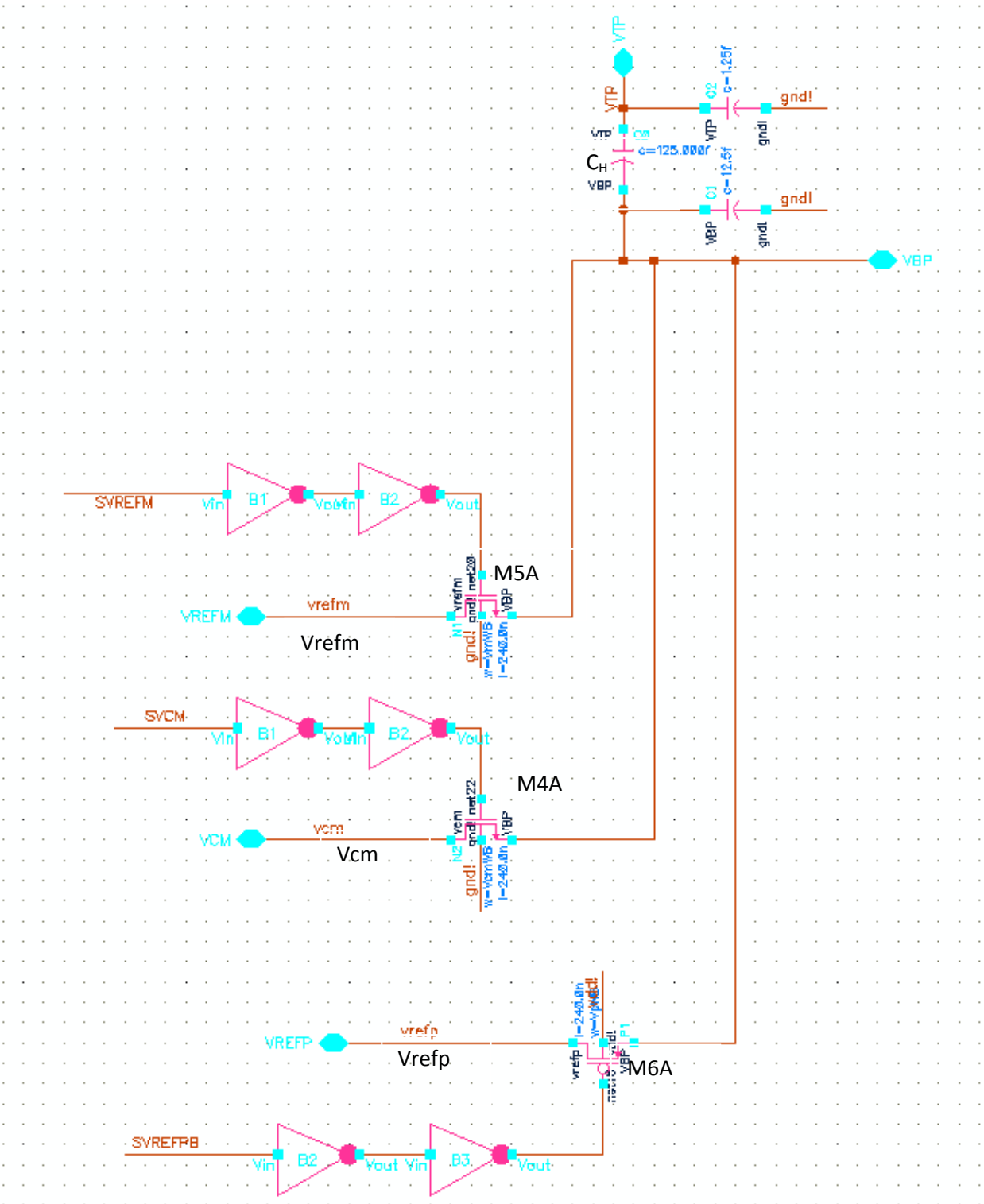


Figure 5.18 DAC cell in segment 2-5

We replaced the ideal DAC cells in the ideal 16-bit, 1 MS/s differential SAR ADC simulation we developed in Section 2.5, with the DAC cells in Figure 5.17 and Figure 5.18. The DAC cell used in segment 1 are shown in Figure 5.17, while the DAC cell used in segment 2-5 are shown in Figure 5.18. From now on, we will call the DAC switches used in segment 1 M4, M5 and M6, and the DAC switches used in segment 2-5 M4A, M5A and M6A. M4/M4A are used during the hold mode, and M5/M5A, M6/M6A are used during the bit cycling mode.

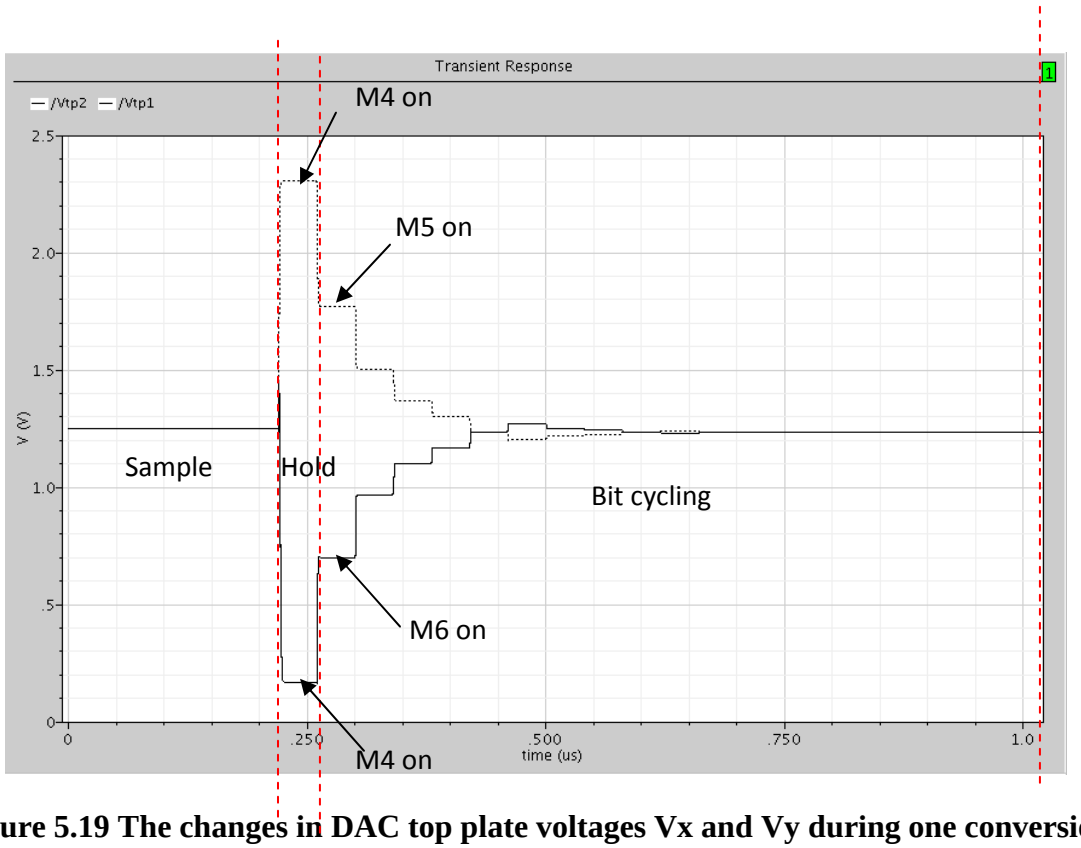


Figure 5.19 The changes in DAC top plate voltages Vx and Vy during one conversion

Figure 2.17 shows the waveform of the DAC top plate voltage Vx and Vy during 1 conversion. We repeated it in Figure 5.19 for convenience. Note that ΔV decreases in each cycle in a binary fashion. Since settling time increases with ΔV, it is most stringent to meet the settling time requirement during the hold mode and the first bit cycling mode.

Based on this observation, we know that as long as we sized M4, M5 and M6 such that it meets the settling time requirement during the hold mode and the first bit cycling mode, we can be assured that the settling requirement is met for the rest of the conversion. Since C_H is segment 2-5 is 8 times smaller than the C_H in segment1, we can automatically size M4A, M5A and M6A such that they are 1/8 of M4, M5 and M6.

The settling time of the DAC switches are strongly affected by the LC oscillation of the bond wires. However, we can use the resistances of the DAC switches M4, M5 and M6 to damp out

these oscillations. On the other hand, the resistances of M4, M5 and M6 should not be too big, because the RC time constant needs to be small enough such that the DAC top plate voltage (Vx and Vy) can reach the next steady state within the duration of the hold mode cycle, and within 15 ns during the bit cycling mode.

Figure 5.20 shows that the effect on the DAC top plate voltages if M4, M5 and M6 are too big. Note that that the LC oscillations affect the comparator decisions during the bit cycling mode.

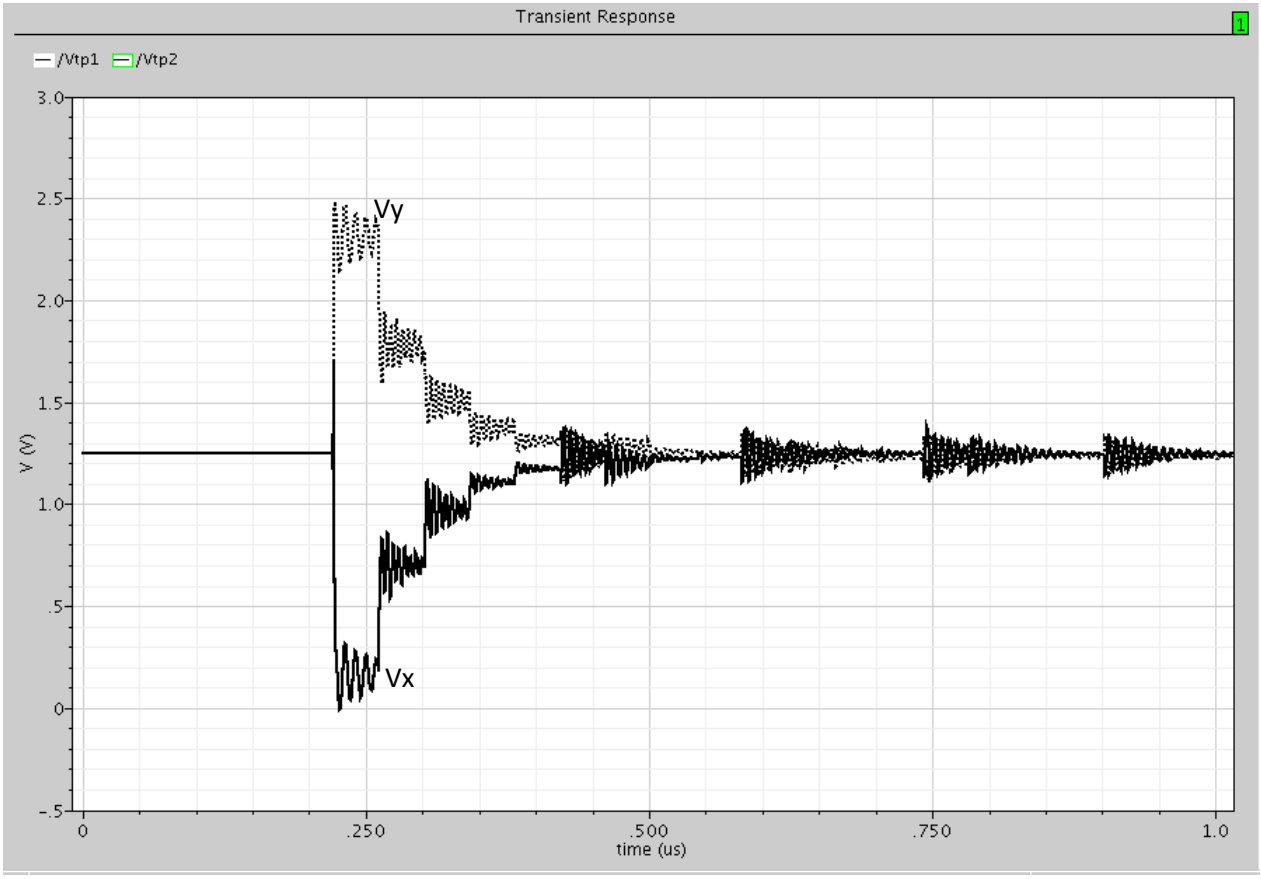


Figure 5.20 LC oscillations from bond wire affecting the DAC action

Figure 5.21 shows the effect on the DAC voltages if we choose DAC switches that are too small. Note that the voltage do not have enough time to reach its steady state due to the big RC time constant.

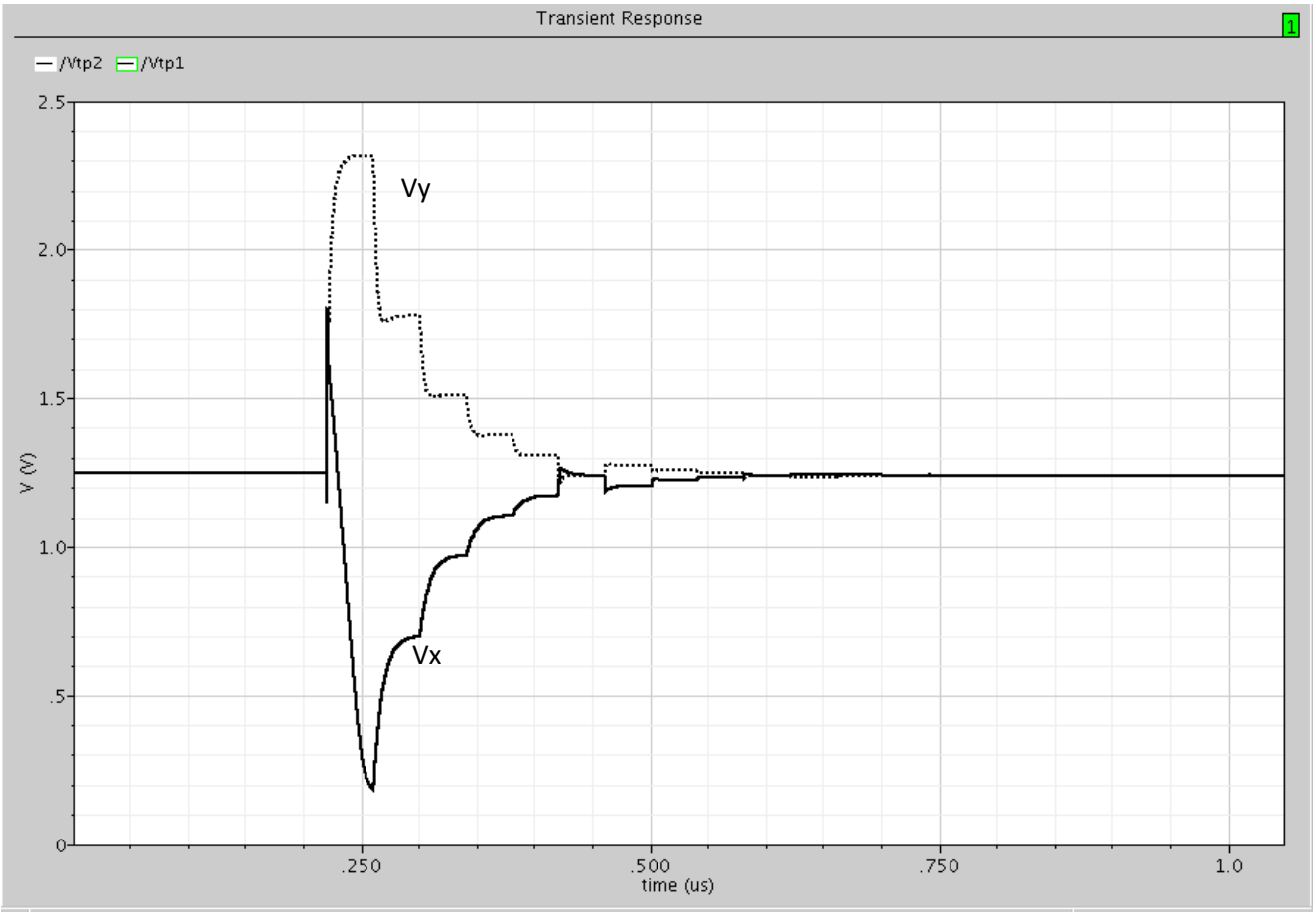


Figure 5.21 RC time constant formed from the bond wire and the DAC switches affecting the DAC action

We need to optimize the size of M4, M5 and M6 to reduce LC oscillation and the RC time constant. Based on the 1mm bond wire data in Appendix E, we optimized the size of M4, M5 and M6 to be 5um, 5um and 20um respectively, giving a settling time in the hold mode to be around 21 ns, and a settling time during the bit cycling action to be around 8.5us.

We also investigate how the length of the bond wire affect the settling time during the hold mode and bit cycling mode, based on the bond wire data in Appendix E. Table 5.4 shows that the settling time during the bit cycling mode changes within $\pm$2ns, as the bond wire length increases from 1 mm to 4 mm. The settling time of the hold mode, however, change as much as $\pm$15ns as the bond wire length increases. These results makes sense, as three switches M2, M3 and M4 that are sensitive to bond wire inductances are switched during the hold mode. On the other

hand, only one DAC switch which is sensitive to bond wire inductance are switched during the bit cycling mode.

The results in Table 5.4 show that we should restrict the bond wire length to 1mm, so that the voltage during the hold mode have enough time to settle within 40 ns(the hold mode duration). If this is not possible, we should allocate more time for the hold mode and the MSB decision making, and allocate less time for the LSB decision making within the 1us conversion.

Table 5.4 The effect of bond wire length on settling time ts

Bond Wire length	ts(hold mode) (ns)	ts for turning on Vm (1^{st} bit cycling) (ns)	ts for turning on Vp (1^{st} bit cycling) (ns)
1mm	20.7	8.8	8.2
2mm	38.9	10.2	10.0
3mm	45.5	9.8	9.8
4mm	41.0	9.7	9.7

5.8 Putting everything together

Using the results from Section 5.4 - Section 5.7, we added the tapered buffer, external non-idealities, C_H parasitics, and the band-limiting resistor r into the ADC simulation shown in Figure 2.10 and Section 2.5.

We found that the harmonic distortion of the sample-and-hold circuit improves, when the sampling DAC cell in Figure 5.17 are connected to others. As a result, we can reduce M2, M3 to 116.64u and 466.56u. Due to the charge injection error reason, we need to reduce M1 to 56.8u. These dimensions give a total harmonic distortion of -96 dB. Reducing M1, M2 and M3 also helps to reduce the LC oscillation during DAC settling.

Due to process variation, the acquisition time may change. Therefore, we want to choose a band-limiting resistor r such that it would give an acquisition time smaller than 200 ns, but still within the same order of magnitude. We chose r = 300 Ω. This gives an acquisition time half of what is required. As a result, the bandwidth and the system noise doubles. This is the preferred tradeoff, however, as the speed requirement is more stringent than the noise requirement in our prototype ADC. Table 5.5 shows the effect of the band-limiting resistor r on the acquisition time.

Table 5.5 The effect of band-limiting resistor r on acquisition time

Band-limiting resistor r Ω	Acquisition time (ns)
0	89.8
100	90.7
300	97.1
500	117.9
700	153.4
900	193.7

Figure 5.22 shows that the residual Vy-Vx is within 1 LSB at the 16-bit level. This shows that the ADC can resolve two voltages at a 16-bit level, and that the charge injection error is effectively removed. Figure 5.23 shows that the ADC error, however, is within 1 LSB at the 15-bit level. Therefore, the ADC only has 15-bit accuracy. This 1 bit reduction in accuracy is due to the top plate parasitic of the unit capacitor and the bond wire parasitic. While we can reduce the effects of bond wire by re-allocating the clock cycles used in the conversion and using shorter bond wires, we cannot do anything about the top plate parasitic of the unit capacitor, except trying to do a good layout to minimize it.

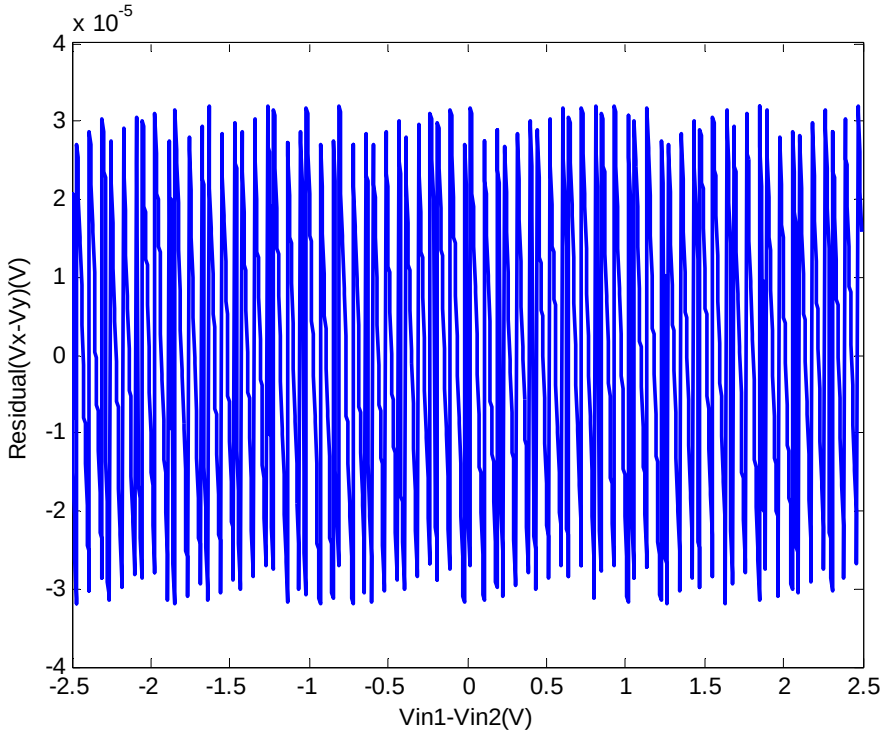


Figure 5.22 Residual error

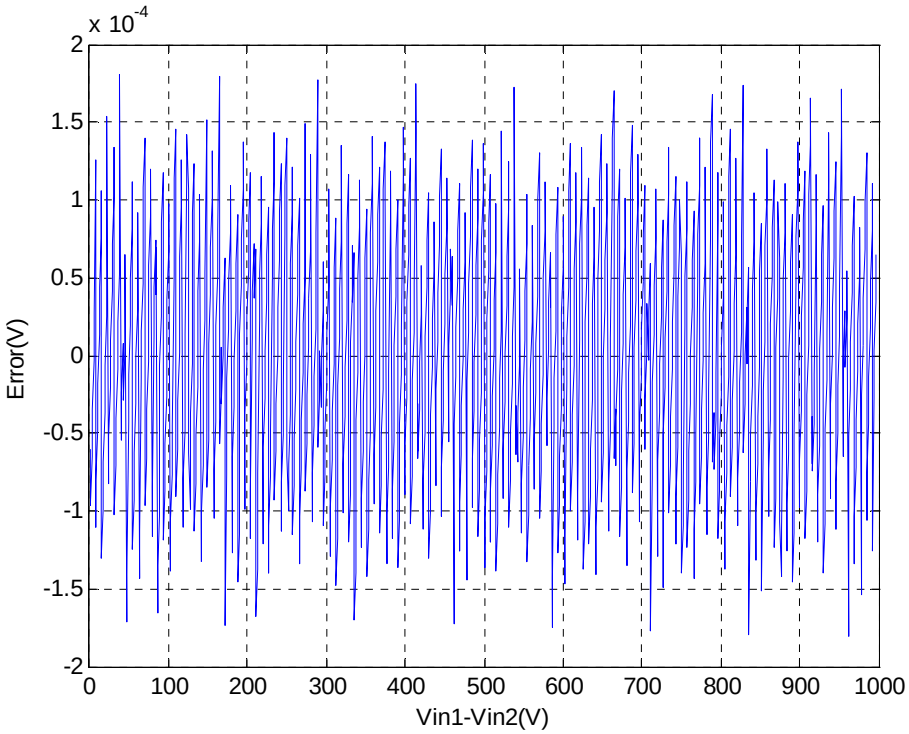


Figure 5.23 ADC error

Table 5.6 summarizes the design parameters we used for the 16 bit, 1 MS/s differential SAR converter to reach the specifications in Section 5.3.

Table 5.6 Design parameters used in the 1 MS/s 16-b differential SAR converter design

Sampling switches	M1	56.8 um
	M2	116.64 um
	M3	466.56 um
DAC switches	M4	5um
	M4A	0.625 um
	M5	5 um
	M5A	0.625 um
	M6	20 um
	M6A	2.5 um
Voltage source parasitics	Rs	1 mΩ
	Cs	10 um
Bond wire parasitics	Rb	125 mΩ-525 mΩ
	Cb	0.08 pF -0.3 pF
	Lb	0.6 nH-4 nH
Min. Width inverter in tapered buffer	Inv_Width	0.36um
Band-limiting resistor r	r	300 Ω
1pF CH top plate parasitics	C_{T1}	10 fF
1pF CH bottom plate parasitics	C_{B1}	100 fF
125 fF CH top plate parasitics	C_{T2}	1.25 fF
125 fF CH bottom plate parasitics	C_{B2}	12.5 fF

5.9 Summary

In this chapter, we successfully designed the DAC, the sample-and-hold circuit, and the DAC switches for the ADC. Sizing the sampling switches M1, M2 and M3 to be 56.8um, 116.64um and 466.46um gives a sample-and-hold circuit that has a harmonic distortion of -96dB and negligible charge injection error. Choosing the DAC switches M4, M5 and M6 to be 5um, 5um and 20um minimize the LC oscillation, as well as the RC time constant resulted from the DAC switches and bond wire parasitics. With the top plate parasitic in the unit capacitor, the designed ADC achieves 16 bit resolution but only 15 but accuracy. Therefore, one needs to be careful in reducing the top plate parasitic during layout.

6

Conclusion

6.1 Summary

This work uses Matlab to verify that conceptually, the "split-ADC" architecture developed in [7,8] can be applied to the SAR converter, even though there are much more parameters to calibrate for in the SAR ADC.

An error correction algorithm is developed for the split SAR ADC architecture. Simulation results show that the calibration algorithm effectively remove the ADC errors due to noise and capacitor mismatch in the digital domain. As long as the total voltage error caused by the capacitor mismatches and noise are less than the sum of redundant bits (1022 LSB), the error correction algorithm can restore the INL/ DNL to within ±0.5 LSB.

The error correction algorithm works for any unknown input signal, whether it is continuous or not. The calibration time remains relatively the same for all inputs. However, using a DC signal for calibration gives more steady state weight error and ADC error.

The use of the adaptive parameter μe allow us to adjust the DAC weights with the optimal speed and accuracy. If the system noise is comparable to the mismatch error, one should use a smaller μe to filter out the noise. If system noise is small compared to the mismatch, one should use a larger μe for faster convergence.

The error correction algorithm can calibrate the ADC within 10^5-10^6 conversions. It is at least 3 orders of magnitude faster than the traditional statistical method, which requires 2^{32} conversions. In addition, frequency response of the calibrated ADC improves.

In applying the error correction algorithm, we also developed an IC design for a 16 bit, 1 MS/s differential SAR converter. Thus far, we have finished the design work on the sample-and-hold circuit and the DAC structure of the ADC. Although the ADC can resolve two input voltages within 1 LSB at the 16 bit level, simulation results show that the top plate parasitics and the bond wire parasitic reduce the accuracy of the ADC to 15 bit.

6.2 Future Work

With the feasibility of the “Split-SAR” architecture verified, the next step is to build the real system in Figure 3-1. On the mixed signal IC side, we have to develop the comparator design and the SAR logic necessary to communicate with the FPGA control. On the FPGA control side, we have implemented the interface necessary to communicate with the mixed signal IC, and we have to work on implementing the error correction algorithm in the FPGA.

References

[1] Z. Zheng at el, "Capacitor Mismatch error cancellation Technique for a successive approcimation A/D converter", *Circuits and Systems*, Vol.2, pg 326-329, May 1999

[2] H.S. Lee, "A Self-Calibrating 15 bit CMOS A/D Converter", *IEEE Journal of Solid-State Circuits*, Vol. SC-19, p 813-819, Dec 1984

[3] H. Neubauer, " A Successive Approximation A/D Converter with 16bit 200kS/s in 0.6um CMOS using Selfcalibration and Low Power Techniques", *Circuits and Systems*, Vol.2, pg 859-862, Sep 2001

[4] Y. Kuramochi, " A 0.05-mm^2 110-uW 10-b Self-Calibrating Successive Approximation ADC Core in 0.18-um CMOS", *Solid-State Circuits Conference*, pg. 224-227, Nov 2007

[5] G. Miller et al, "An 18b 10us Self-Calibrating ADC", *ISSCC*, pg 168-169, Feb 1990

[6] M.F. Wagdy, "Correction of Capacitor Errors During the Conversion Cycle of Self-Calibrating A/D Converters", *IEEE Transactions on Circuits and Systems*, Vol. 37, No. 10, Oct 1990

[7] J. McNeill, M. Coln, B. Larivee, " Split ADC" Architecture for Deterministic Digital Background Calibration of a 16-bit 1-MS/s ADC", *IEEE Journal of Solid-State Circuits*, Vol. 40, No.12, Dec 2005

[8] J. McNeill, M. Coln, B. Larivee, " Digital Background Calibration Algorithm for "Split ADC" Architecture," *In Press*

[9] J. Gan at el, " A Non-Binary Capacitor Array Calibration Circuit with 22-bit accuracy in Successive Approximation Analog-to-Digital Converters", *Circuits and Systems*, Vol.1, pg 567-70, Aug 2002

[10] W. Liu et al, " An Equalization-Based Adaptive Digital Background Calibration Technique for Successive approximation Analog-to-Digital Converters ", *ASIC*, pg 289-292, Oct 2007

[11] M. Looney, "Advanced Digital Post-Processing Techniques Enhance Performance in Time-Interleaved ADC Systems," *Analog Dialog*, Volume 37-08, August 2003

[12] H. Jin, E. Lee, and M.Hassoun, " Time-Interleaved A/D Converter with Channel Randomization, " *IEEE Symposium on Circuits and Systems*, June 1997, pp. 425-428

[13] J. Elbornsson and F. Gustafsson, " Analysis of Mismatch Noise in Randomly Interleaved ADC System," *IEEE International Conference on Acoustics, Speech and Signal Processing*, April 2003, pp. VI227 – VI-280

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

[15] M. Tamba, A. Shimizu, H. Munakata and T. Komuro, " A Method to improve SFDR with Random Interleaved Sampling Method," *IEEE International Test Conference*, Nov. 2001, pp 512-519.

[16] H.Jin and E. Lee, " A Digital-Background Calibration Technique for Minimizing Timing-Error Effects in Time-Interleaved ADC's, " *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, Volume 47, No. 7, July 2000, pp. 603-613

[17] J. Elbornsson, F. Gustafsson, and J. Eklund, "Blind Equalization of Time Errors in a Time-Interleaved ADC System," *IEEE Trasactions on Signal Processing*, April 2005, Volume 53, No. 4, April 2005.

[18] D. Fu, K. Dyer, S. Lewis and P. Hurst, "A Digital-Background Calibration Technique for Time-Interleaved Analog-to-Digital Converters," *IEEE Journal of Solid-State Circuits*, Volume 33, No. 12, Dec. 1998, pp. 1904-1910.

[19] J. Eklund and F. Gustafsson , "Digital Offset Compensation of Time-Interleaved ADC Using Random Chopper Sampling," *IEEE Symposium on Circuits and Systems*, May 2000, pp. 447-450

[20] S. Jamal, D. Fu, N. Chang, P. Hurst and Stephen Lewis, " A 10-bit 120-Msample/s Time Interleaved Analog-to-Digital Converter With Digital Background Calibration, " *IEEE Journal of Solid-State Circuits*, Volume 37, No. 12, Dec. 2002, pp. 1618-1626.

[21] I. Galton, "Digital cancellation of D/A converter noise in pipelined ADCs," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, March 2000, pp. 185-196.

[22] B. Murmann et al., "A 12b 75MS/s Pipelined ADC using open-loop residue amplification, "*International Solid-State Circuits Conference (ISSCC) Digest of Technical Papers*, February, 2003, pp. 328-9.

[23] Liu et al, "A 15b 20MS/s CMOS Pipelined ADC with Digital Background Calibration, "*International Solid-State Circuits Conference (ISSCC) Digest of Technical Papers*, February, 2004, pp. 454-5.

[24] Nair et al., "A 96dB SFDR 50MS/s Digitally Enhanced CMOS Pipelined A/D Converter,"*International Solid-State Circuits Conference (ISSCC) Digest of Technical Papers*, February, 2004, pp. 456-7.

[25] Erdogan et al., "A 12-b Digital-Background-Calibrated Algorithmic ADC with -90-dB THD," *International Solid-State Circuits Conference (ISSCC) Digest of Technical Papers*, Feb, 1999, pp. 316-7.

[26] Chiu et al., "Least mean square adaptive digital background calibration of pipelined ADCs," *IEEE Transactions on Circuits and Systems I: Fundamental Theory andApplications*, Jan. 2004.

[27] H.-S. Lee, "A 12-b 600 ks/s digitally self-calibrated pipelined algorithmic ADC," *IEEE Journal of Solid-State Circuits*, Apr. 1994, pp. 509 -515

[28] A. Karanicolas et al. , "A 15-b 1-MS/s digitally self-calibrated pipeline ADC," *IEEE Journal of Solid-State Circuits*, Dec. 1993, pp. 1207 -1215

[29] David A. Johns, Ken Martin, Analog Integrated Circuit Design, USA: John Wiley & Sons, Inc, 1997.

[30] J.L. McCreary, "ALL-MOS Charge Redistribution Analog-to-Digital Conversion Techniques – Part I", *IEEE Journal of Solid-State Circuits*, Vol. SC-10, p 371-379, Dec 1975

[31] Rosa Crougwell's thesis, pg 37

[32] Y. Chiu, C.W. Tsang, B. Nikolic, P.R. Gray, " Least mean square adaptive digital background calibration of pipelined analog-to-digital converters, " *IEEE Trans. Circuits and Systems I,* vol. 51, no. 1, pp.38-46, Jan. 2004

[33] S.S. Haykin, Adaptive Filter Theory, 3rd. ed. Upper Saddle River, NJ: Prentice-Hall, 1996.

[34] P. Yu, " Low-Power Design Techniques for Pipelined Analog-to-Digital Converters", PHD thesis, May 1996

[35] I. N Bronshtein et al, *Handbook of mathematics*, 3rd ed. New York: Springer-Verlag, p.892, 1997

[36] Pelgrom paper, can't find it!? What is the name of the paper?

[37] G. Golub and C.F. van Loan, "Matrix Computations (Third Edition)." Baltimore, Maryland: John Hopkins University Press, 1996.

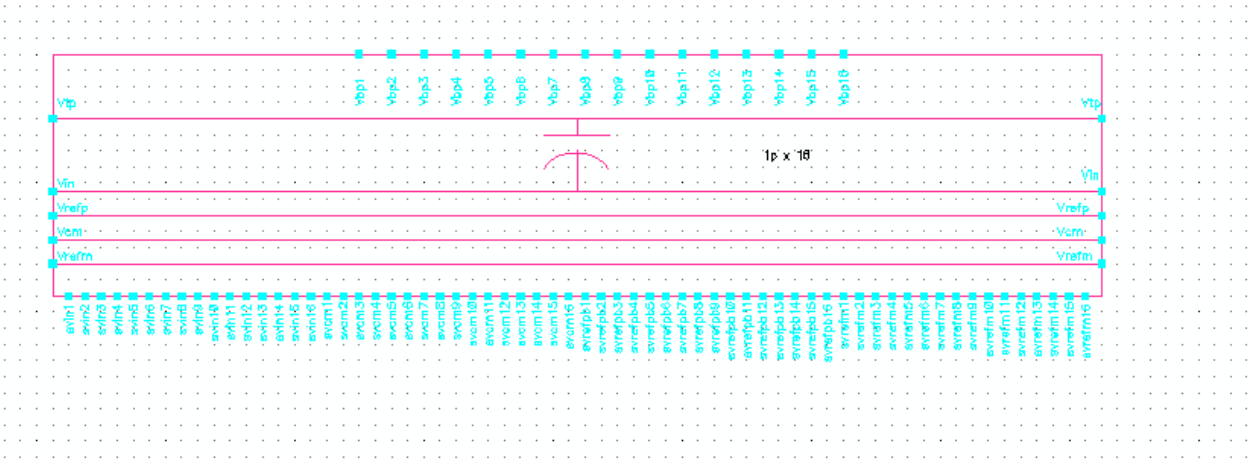
[38] Philip E. Allen, Douglas R. Holberg, CMOS Analog Circuit Design, New York: Oxford University Press, 2002

[39] B.S. Cherkauer at el, "A Unified Design Methodology for CMOS Tapered Buffers", IEEE transactions on VLSI Systems, Vol. 3, No. 1, March 1995

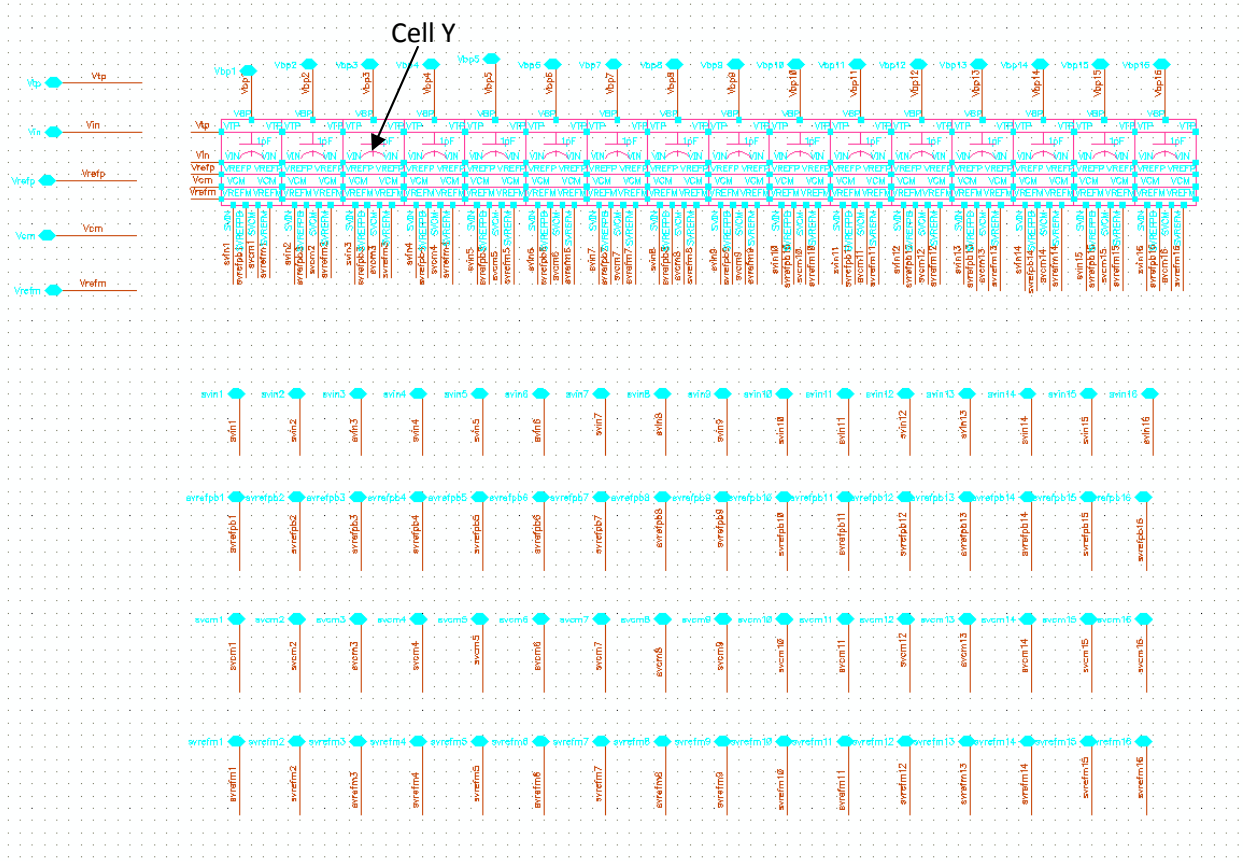
Appendix A

Simulating an ideal 16-bit 1 MS/s differential SAR converter in Cadence

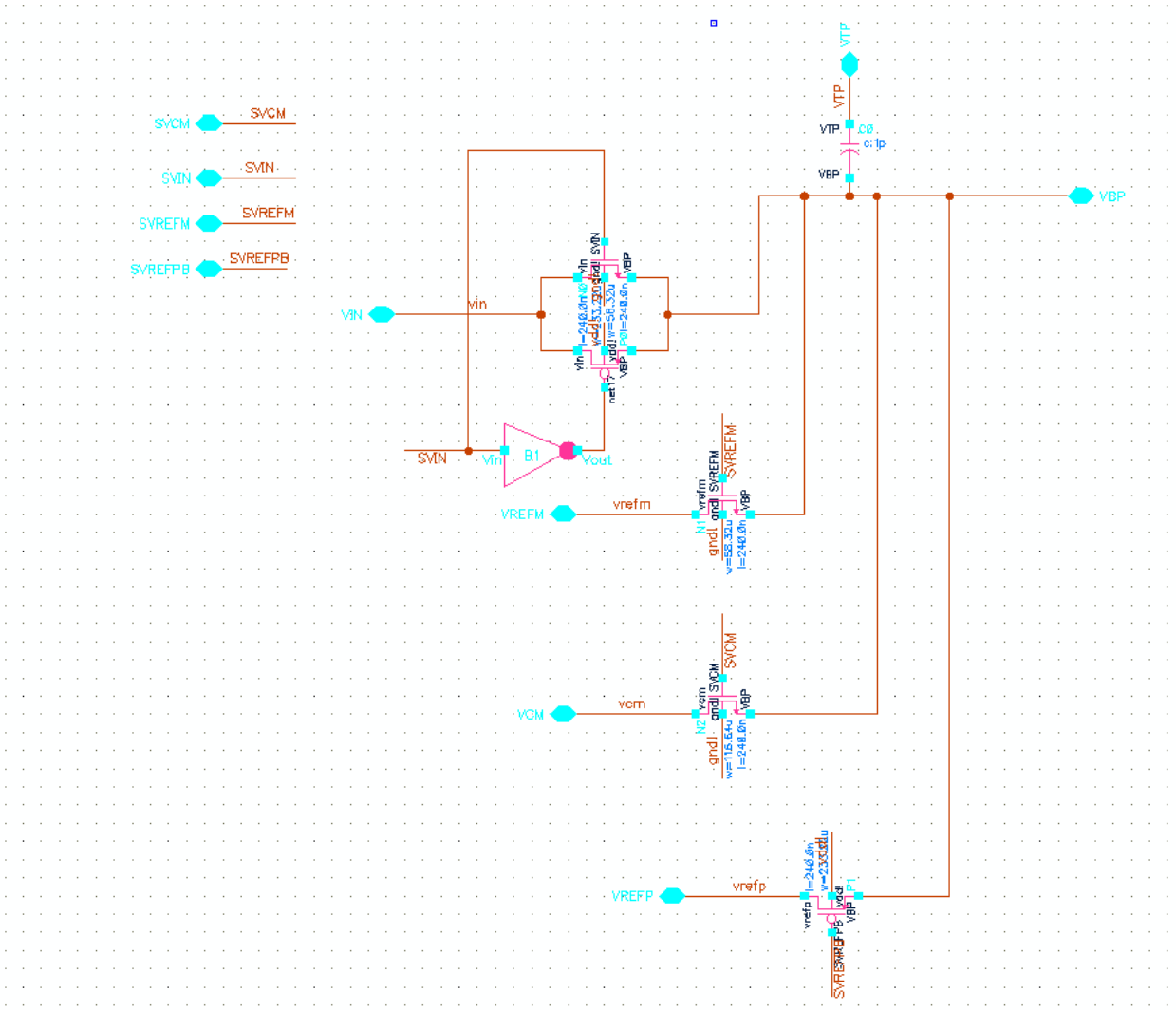
Symbol for Segment X



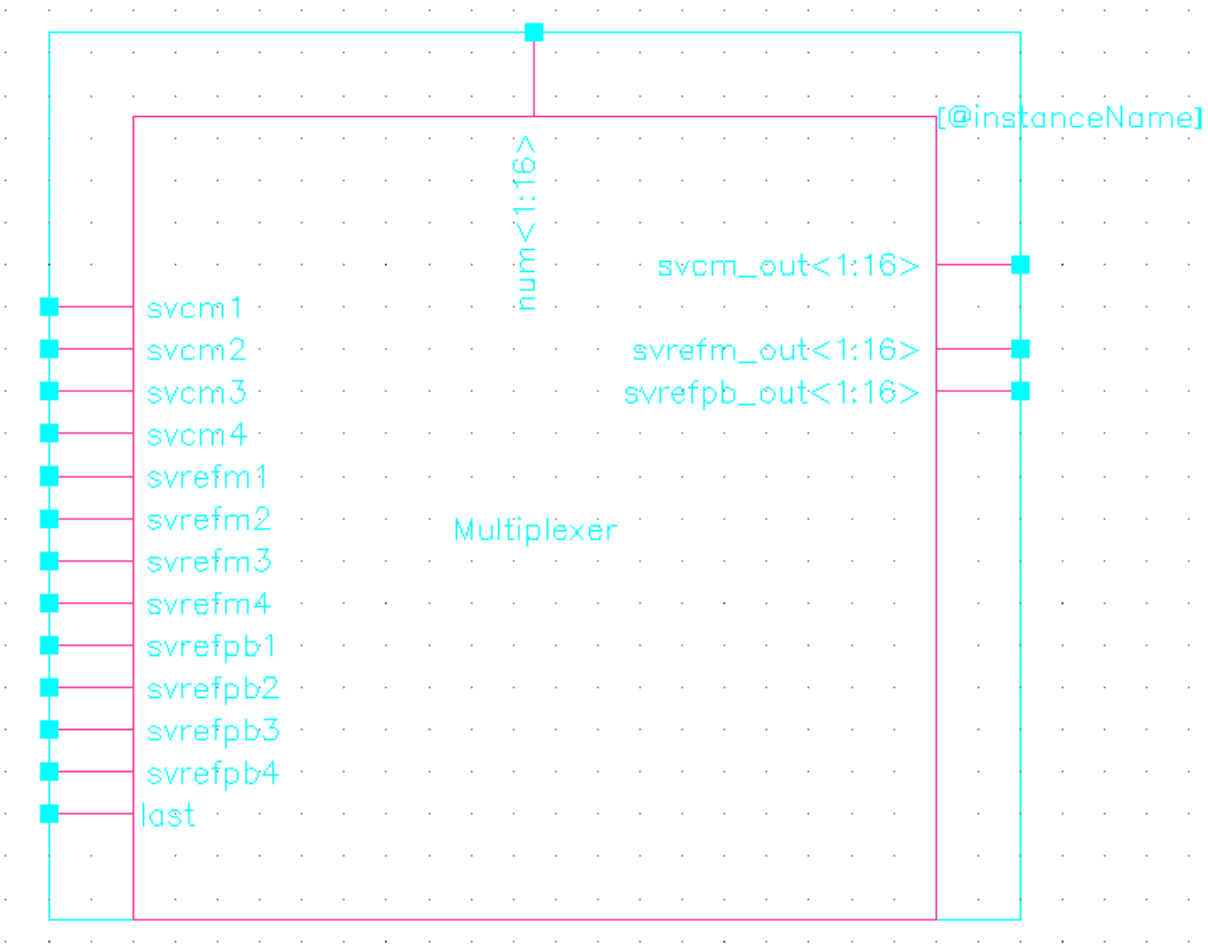
Inside Segment X



Inside Cell Y



Symbol for DAC switch selector



VerilogA Code for DAC switch selector

```
// VerilogA for mylib, multiplexer, veriloga

`include "constants.vams"
`include "disciplines.vams"

module multiplexer_040608(num,svrefpb1,svrefpb2,svrefpb3,svrefpb4,svcm1,
svcm2,svcm3,svcm4,svrefm1,svrefm2,svrefm3,svrefm4,
svrefpb_out,svcm_out,svrefm_out,last);

input [1:16] num;
input svrefpb1,svrefpb2,svrefpb3,svrefpb4;
input svcm1,svcm2,svcm3,svcm4;
input svrefm1,svrefm2,svrefm3,svrefm4,last;
```

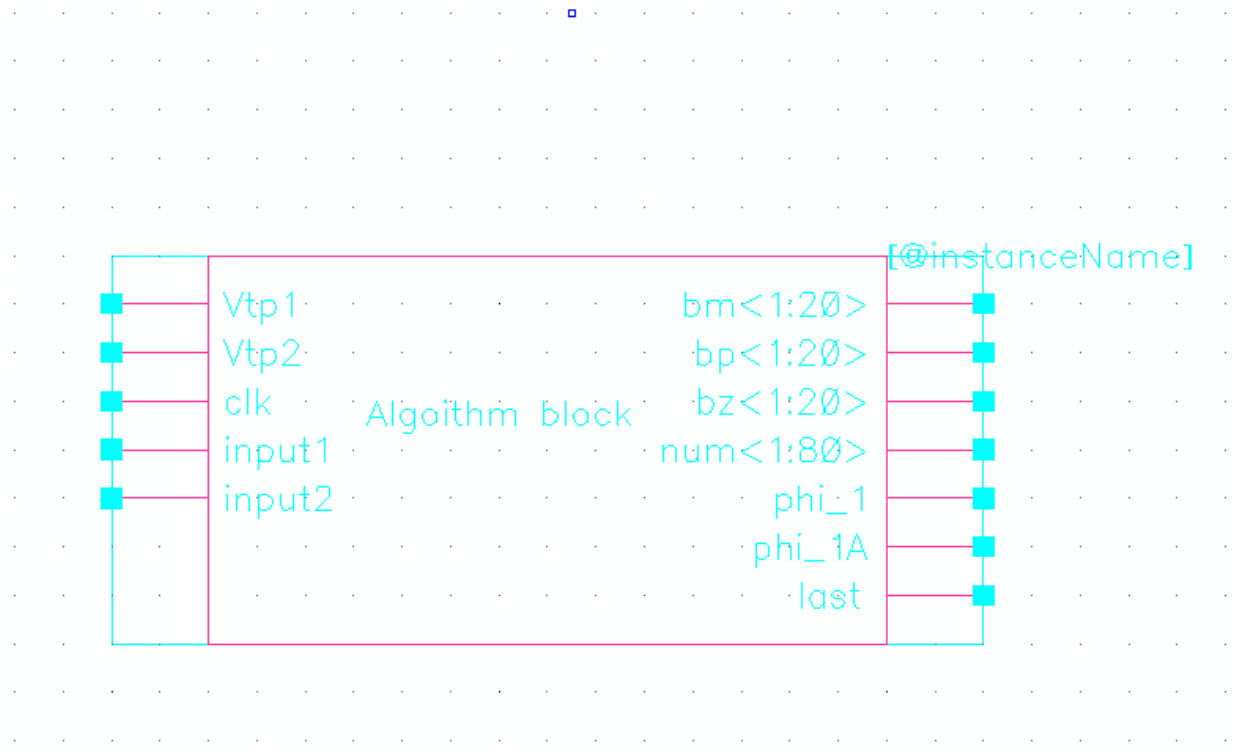
```
output [1:16] svrefpb_out,svcm_out,svrefm_out;

electrical [1:16] num;
electrical svrefpb1,svrefpb2,svrefpb3,svrefpb4;
electrical svcm1,svcm2,svcm3,svcm4;
electrical svrefm1,svrefm2,svrefm3,svrefm4,last;
electrical [1:16] svrefpb_out,svcm_out,svrefm_out;

integer NUM[1:16];
integer i;
real SVREFPB1, SVREFPB2, SVREFPB3, SVREFPB4;
real SVCM1, SVCM2, SVCM3, SVCM4;
real SVREFM1, SVREFM2, SVREFM3, SVREFM4, LAST;
real SVREFPB_OUT[1:16], SVCM_OUT[1:16], SVREFM_OUT[1:16];

analog begin

  NUM[1] = V(num[1]);
  NUM[2] = V(num[2]);
  NUM[3] = V(num[3]);
  NUM[4] = V(num[4]);
  NUM[5] = V(num[5]);
  NUM[6] = V(num[6]);
  NUM[7] = V(num[7]);
  NUM[8] = V(num[8]);
  NUM[9] = V(num[9]);
  NUM[10] = V(num[10]);
  NUM[11] = V(num[11]);
  NUM[12] = V(num[12]);
  NUM[13] = V(num[13]);
  NUM[14] = V(num[14]);
  NUM[15] = V(num[15]);
  NUM[16] = V(num[16]);

  SVREFPB1 = V(svrefpb1);
  SVREFPB2 = V(svrefpb2);
  SVREFPB3 = V(svrefpb3);
  SVREFPB4 = V(svrefpb4);

  SVCM1 = V(svcm1);
```

```
SVCM2 = V(svcm2);
SVCM3 = V(svcm3);
SVCM4 = V(svcm4);

SVREFM1 = V(svrefm1);
SVREFM2 = V(svrefm2);
SVREFM3 = V(svrefm3);
SVREFM4 = V(svrefm4);

LAST = V(last);


 for (i = 1; i <17; i = i+1) begin
  case(1)
    (NUM[i] >=1) && (NUM[i] <= 8):begin
                        SVREFPB_OUT[i] = SVREFPB1;
                                SVCM_OUT[i] = SVCM1;
                                SVREFM_OUT[i] = SVREFM1;
                        end

    (NUM[i]>=9) && (NUM[i] <=12):begin
                        SVREFPB_OUT[i] = SVREFPB2;
                                SVCM_OUT[i] = SVCM2;
                                SVREFM_OUT[i] = SVREFM2;
                        end
    (NUM[i] ==13) || (NUM[i] == 14):begin
                        SVREFPB_OUT[i] = SVREFPB3;
                                SVCM_OUT[i] = SVCM3;
                                SVREFM_OUT[i] = SVREFM3;
                        end
                       NUM[i] == 15:begin
                              SVREFPB_OUT[i] = SVREFPB4;
                                SVCM_OUT[i] = SVCM4;
                                SVREFM_OUT[i] = SVREFM4;
                        end
                       NUM[i] ==16: begin
                        SVREFPB_OUT[i] = 2.5;
                                SVCM_OUT[i] = LAST;
                                SVREFM_OUT[i] = 0;
                        end
```

```
    endcase
    end


V(svrefpb_out[1]) <+ SVREFPB_OUT[1];
V(svrefpb_out[2]) <+ SVREFPB_OUT[2];
V(svrefpb_out[3]) <+ SVREFPB_OUT[3];
V(svrefpb_out[4]) <+ SVREFPB_OUT[4];
V(svrefpb_out[5]) <+ SVREFPB_OUT[5];
V(svrefpb_out[6]) <+ SVREFPB_OUT[6];
V(svrefpb_out[7]) <+ SVREFPB_OUT[7];
V(svrefpb_out[8]) <+ SVREFPB_OUT[8];
V(svrefpb_out[9]) <+ SVREFPB_OUT[9];
V(svrefpb_out[10]) <+ SVREFPB_OUT[10];
V(svrefpb_out[11]) <+ SVREFPB_OUT[11];
V(svrefpb_out[12]) <+ SVREFPB_OUT[12];
V(svrefpb_out[13]) <+ SVREFPB_OUT[13];
V(svrefpb_out[14]) <+ SVREFPB_OUT[14];
V(svrefpb_out[15]) <+ SVREFPB_OUT[15];
V(svrefpb_out[16]) <+ SVREFPB_OUT[16];

V(svcm_out[1]) <+ SVCM_OUT[1];
V(svcm_out[2]) <+ SVCM_OUT[2];
V(svcm_out[3]) <+ SVCM_OUT[3];
V(svcm_out[4]) <+ SVCM_OUT[4];
V(svcm_out[5]) <+ SVCM_OUT[5];
V(svcm_out[6]) <+ SVCM_OUT[6];
V(svcm_out[7]) <+ SVCM_OUT[7];
V(svcm_out[8]) <+ SVCM_OUT[8];
V(svcm_out[9]) <+ SVCM_OUT[9];
V(svcm_out[10]) <+ SVCM_OUT[10];
V(svcm_out[11]) <+ SVCM_OUT[11];
V(svcm_out[12]) <+ SVCM_OUT[12];
V(svcm_out[13]) <+ SVCM_OUT[13];
V(svcm_out[14]) <+ SVCM_OUT[14];
V(svcm_out[15]) <+ SVCM_OUT[15];
V(svcm_out[16]) <+ SVCM_OUT[16];

V(svrefm_out[1]) <+ SVREFM_OUT[1];
```

```
  V(svrefm_out[2]) <+ SVREFM_OUT[2];
  V(svrefm_out[3]) <+ SVREFM_OUT[3];
  V(svrefm_out[4]) <+ SVREFM_OUT[4];
  V(svrefm_out[5]) <+ SVREFM_OUT[5];
  V(svrefm_out[6]) <+ SVREFM_OUT[6];
  V(svrefm_out[7]) <+ SVREFM_OUT[7];
  V(svrefm_out[8]) <+ SVREFM_OUT[8];
  V(svrefm_out[9]) <+ SVREFM_OUT[9];
  V(svrefm_out[10]) <+ SVREFM_OUT[10];
  V(svrefm_out[11]) <+ SVREFM_OUT[11];
  V(svrefm_out[12]) <+ SVREFM_OUT[12];
  V(svrefm_out[13]) <+ SVREFM_OUT[13];
  V(svrefm_out[14]) <+ SVREFM_OUT[14];
  V(svrefm_out[15]) <+ SVREFM_OUT[15];
  V(svrefm_out[16]) <+ SVREFM_OUT[16];

end // analog event end

endmodule
```

Symbol for SAR logic block



VerilogA Code for SAR logic block

```
// VerilogA for mylib, charge_injection_error, veriloga

`include "constants.vams"
`include "disciplines.vams"

module R040608_16_bit_ADC_block_A(clk,
Vtp1,Vtp2,phi_1A,phi_1,bz,bp,bm,num,input1,input2,last);

input clk, Vtp1,Vtp2,input1,input2;
output phi_1A,phi_1,last;
output [1:20] bz,bp,bm;
output [1:80] num;

electrical clk, Vtp1,Vtp2,phi_1A,phi_1,input1,input2,last;
electrical [1:20] bz,bp,bm;
electrical [1:80] num;

integer fs1A,fs2A,fs3A,fs4A,fs5A,fs6A,fs7A,PHI_1A,
BZ[1:20],BP[1:20],BM[1:20],D[1:20],i,next_state, bit,LAST;
integer seed1,seed2,seed3,seed4,seed5,x,NUM[1:80];
real  v1,v2,vin1,vin2,tcheck1,tcheck2,delay;

analog begin
  @(initial_step) begin
     fs1A=$fopen( "file1A.txt");
     fs2A=$fopen("file2A.txt");
     fs3A=$fopen("file3A.txt");
     fs4A=$fopen("file4A.txt");
     fs5A=$fopen("file5A.txt");
     fs6A=$fopen("file6A.txt");
     fs7A=$fopen("file7A.txt");
     seed1 = 231;
     seed2 = 13;
     seed3 = 24;
     seed4 = 36;
     seed5 = 59;
     next_state =0;
     delay = 0.7e-9;
```

```
// generating random number from 1 to 16
NUM[1] = abs($random(seed1)%16)+1;
NUM[17] = abs($random(seed2)%16)+1;
NUM[33] = abs($random(seed3)%16)+1;
NUM[49] = abs($random(seed4)%16)+1;
NUM[65] = abs($random(seed5)%16)+1;

for (i=2;i <17; i= i+1)begin
   NUM[i] = (NUM[1]+(i-1))%16;
      NUM[16+i] = (NUM[17]+(i-1))%16;
      NUM[32+i] = (NUM[33]+(i-1))%16;
      NUM[48+i] = (NUM[49]+(i-1))%16;
      NUM[64+i] = (NUM[65]+(i-1))%16;


      if (NUM[i] == 0) NUM[i] = 16;
      if (NUM[16+i] == 0) NUM[16+i] = 16;
      if (NUM[32+i] == 0) NUM[32+i] = 16;
      if (NUM[48+i] == 0) NUM[48+i] = 16;
      if (NUM[64+i] == 0) NUM[64+i] = 16;

end // for end

PHI_1A = 2.5;
LAST = 0;


for (i=1; i <21; i=i+1)begin
   BP[i]=0;
   BM[i]=0;
end

for (i=1; i <5; i=i+1)begin
   BZ[i]=0;
end


for (i=5; i <21; i=i+1)begin
   BZ[i]=2.5;
end
```

```
     vin1=V(input1);
     vin2=V(input2);


  end


  @(cross(V(clk)-1.25, 1))begin
    v2 = V(Vtp2);
    v1 = V(Vtp1);
    tcheck1 = $abstime;
    $fstrobe( fs1A, "%g %f %f %g %f %d", tcheck1,v2, v1,v2-v1,vin2-vin1, next_state);



    case(1)
      next_state == 0: begin

                    if (tcheck2 > 0)
                              begin
                    $fstrobe( fs2A, "%g %f %g %d %d %d %d %d %d %d %d %d %d %d %d %d
%d %d %d %d %d %d %d",
                                    tcheck2,vin2-vin1,v2-
v1,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], D[16], D[17], D[18], D[19], D[20]);

                              $fstrobe( fs3A, "%g %d %d %d %d %d %d %d %d %d %d %d %d %d
%d %d %d %d", tcheck2, next_state, NUM[1], NUM[2], NUM[3],
                           NUM[4], NUM[5], NUM[6], NUM[7], NUM[8], NUM[9], NUM[10],
NUM[11], NUM[12], NUM[13], NUM[14], NUM[15], NUM[16]);

                              $fstrobe( fs4A, "%g %d %d %d %d %d %d %d %d %d %d %d %d %d
%d %d %d %d", tcheck2, next_state, NUM[17], NUM[18], NUM[19],
                           NUM[20], NUM[21], NUM[22], NUM[23], NUM[24], NUM[25],
NUM[26], NUM[27], NUM[28], NUM[29], NUM[30], NUM[31], NUM[32]);

                              $fstrobe( fs5A, "%g %d %d %d %d %d %d %d %d %d %d %d %d %d
%d %d %d %d", tcheck2, next_state, NUM[33], NUM[34], NUM[35],
```

```
                NUM[36], NUM[37], NUM[38], NUM[39], NUM[40], NUM[41],
NUM[42], NUM[43], NUM[44], NUM[45], NUM[46], NUM[47], NUM[48]);

                $fstrobe( fs6A, "%g %d %d %d %d %d %d %d %d %d %d %d %d %d
%d %d %d %d", tcheck2, next_state, NUM[49], NUM[50], NUM[51],
                NUM[52], NUM[53], NUM[54], NUM[55], NUM[56], NUM[57],
NUM[58], NUM[59], NUM[60], NUM[61], NUM[62], NUM[63], NUM[64]);

                $fstrobe( fs7A, "%g %d %d %d %d %d %d %d %d %d %d %d %d %d
%d %d %d %d", tcheck2, next_state, NUM[65], NUM[66], NUM[67],
                NUM[68], NUM[69], NUM[70], NUM[71], NUM[72], NUM[73],
NUM[74], NUM[75], NUM[76], NUM[77], NUM[78], NUM[79], NUM[80]);
            end


            // generating random number from 1 to 16
                NUM[1] = abs($random(seed1)%16)+1;
                NUM[17] = abs($random(seed2)%16)+1;
                NUM[33] = abs($random(seed3)%16)+1;
                NUM[49] = abs($random(seed4)%16)+1;
                NUM[65] = abs($random(seed5)%16)+1;

                for (i=2;i <17; i= i+1)begin
                  NUM[i] = (NUM[1]+(i-1))%16;
                  NUM[16+i] = (NUM[17]+(i-1))%16;
                  NUM[32+i] = (NUM[33]+(i-1))%16;
                  NUM[48+i] = (NUM[49]+(i-1))%16;
                  NUM[64+i] = (NUM[65]+(i-1))%16;


                  if (NUM[i] == 0) NUM[i] = 16;
                  if (NUM[16+i] == 0) NUM[16+i] = 16;
                  if (NUM[32+i] == 0) NUM[32+i] = 16;
                  if (NUM[48+i] == 0) NUM[48+i] = 16;
                  if (NUM[64+i] == 0) NUM[64+i] = 16;

                end // for end

                PHI_1A = 2.5;
                LAST = 0;
```

```
                next_state = 1;bit =1;

                for (i=1; i <21; i=i+1)begin
              BP[i]=0;
              BM[i]=0;
            end

                for (i=1; i <5; i=i+1)begin
              BZ[i]=0;
            end


            for (i=5; i <21; i=i+1)begin
              BZ[i]=2.5;
            end

                vin1=V(input1);
                vin2=V(input2);

               end

next_state ==1: begin

                next_state = 2;

             end
next_state ==2: begin
             next_state = 3;
               end


next_state ==3: begin
               next_state = 4;
            end
next_state ==4: begin
               next_state = 5;
            end


next_state ==5: begin
```

```
                PHI_1A=0;
                        LAST = 2.5;
                  for (i=1; i <5; i=i+1)
                        begin
                 BZ[i]=2.5;
                        end

                  next_state = 6;
                end

((next_state >5)&&(next_state <26)): begin


                  if (v2-v1>0) begin
                              BM[bit]=2.5;
                              BP[bit]=0;
                                 D[bit] = -1;
                            end
                            else begin
                              BM[bit]=0;
                              BP[bit]=2.5;
                                 D[bit] = 1;
                            end

                            BZ[bit]=0;

                            bit = bit+1;
                            next_state = next_state+1;

                            if (next_state == 26)
                            begin
                              next_state = 0;
                              tcheck2 = $abstime;
                            end

                                    end


  endcase
```

```
 end // cross event end


V(phi_1A) <+ transition(PHI_1A,0,1e-12,1e-12);
V(phi_1)  <+ absdelay(V(phi_1A), 0.2n, 0.2n);
V(bz[1]) <+transition(BZ[1],delay,1e-12,1e-12);
V(bz[2]) <+transition(BZ[2],delay,1e-12,1e-12);
V(bz[3]) <+transition(BZ[3],delay,1e-12,1e-12);
V(bz[4]) <+transition(BZ[4],delay,1e-12,1e-12);
V(bz[5]) <+transition(BZ[5],delay,1e-12,1e-12);
V(bz[6]) <+transition(BZ[6],delay,1e-12,1e-12);
V(bz[7]) <+transition(BZ[7],delay,1e-12,1e-12);
V(bz[8]) <+transition(BZ[8],delay,1e-12,1e-12);
V(bz[9]) <+transition(BZ[9],delay,1e-12,1e-12);
V(bz[10]) <+transition(BZ[10],delay,1e-12,1e-12);
V(bz[11]) <+transition(BZ[11],delay,1e-12,1e-12);
V(bz[12]) <+transition(BZ[12],delay,1e-12,1e-12);
V(bz[13]) <+transition(BZ[13],delay,1e-12,1e-12);
V(bz[14]) <+transition(BZ[14],delay,1e-12,1e-12);
V(bz[15]) <+transition(BZ[15],delay,1e-12,1e-12);
V(bz[16]) <+transition(BZ[16],delay,1e-12,1e-12);
V(bz[17]) <+transition(BZ[17],delay,1e-12,1e-12);
V(bz[18]) <+transition(BZ[18],delay,1e-12,1e-12);
V(bz[19]) <+transition(BZ[19],delay,1e-12,1e-12);
V(bz[20]) <+transition(BZ[20],delay,1e-12,1e-12);
V(last)   <+transition(LAST,delay,1e-12,1e-12);


V(bp[1]) <+transition(BP[1],delay,1e-12,1e-12);
V(bp[2]) <+transition(BP[2],delay,1e-12,1e-12);
V(bp[3]) <+transition(BP[3],delay,1e-12,1e-12);
V(bp[4]) <+transition(BP[4],delay,1e-12,1e-12);
V(bp[5]) <+transition(BP[5],delay,1e-12,1e-12);
V(bp[6]) <+transition(BP[6],delay,1e-12,1e-12);
V(bp[7]) <+transition(BP[7],delay,1e-12,1e-12);
V(bp[8]) <+transition(BP[8],delay,1e-12,1e-12);
V(bp[9]) <+transition(BP[9],delay,1e-12,1e-12);
```

```
V(bp[10]) <+transition(BP[10],delay,1e-12,1e-12);
V(bp[11]) <+transition(BP[11],delay,1e-12,1e-12);
V(bp[12]) <+transition(BP[12],delay,1e-12,1e-12);
V(bp[13]) <+transition(BP[13],delay,1e-12,1e-12);
V(bp[14]) <+transition(BP[14],delay,1e-12,1e-12);
V(bp[15]) <+transition(BP[15],delay,1e-12,1e-12);
V(bp[16]) <+transition(BP[16],delay,1e-12,1e-12);
V(bp[17]) <+transition(BP[17],delay,1e-12,1e-12);
V(bp[18]) <+transition(BP[18],delay,1e-12,1e-12);
V(bp[19]) <+transition(BP[19],delay,1e-12,1e-12);
V(bp[20]) <+transition(BP[20],delay,1e-12,1e-12);


V(bm[1]) <+transition(BM[1],delay,1e-12,1e-12);
V(bm[2]) <+transition(BM[2],delay,1e-12,1e-12);
V(bm[3]) <+transition(BM[3],delay,1e-12,1e-12);
V(bm[4]) <+transition(BM[4],delay,1e-12,1e-12);
V(bm[5]) <+transition(BM[5],delay,1e-12,1e-12);
V(bm[6]) <+transition(BM[6],delay,1e-12,1e-12);
V(bm[7]) <+transition(BM[7],delay,1e-12,1e-12);
V(bm[8]) <+transition(BM[8],delay,1e-12,1e-12);
V(bm[9]) <+transition(BM[9],delay,1e-12,1e-12);
V(bm[10]) <+transition(BM[10],delay,1e-12,1e-12);
V(bm[11]) <+transition(BM[11],delay,1e-12,1e-12);
V(bm[12]) <+transition(BM[12],delay,1e-12,1e-12);
V(bm[13]) <+transition(BM[13],delay,1e-12,1e-12);
V(bm[14]) <+transition(BM[14],delay,1e-12,1e-12);
V(bm[15]) <+transition(BM[15],delay,1e-12,1e-12);
V(bm[16]) <+transition(BM[16],delay,1e-12,1e-12);
V(bm[17]) <+transition(BM[17],delay,1e-12,1e-12);
V(bm[18]) <+transition(BM[18],delay,1e-12,1e-12);
V(bm[19]) <+transition(BM[19],delay,1e-12,1e-12);
V(bm[20]) <+transition(BM[20],delay,1e-12,1e-12);

V(num[1])<+transition(NUM[1],0,1e-12,1e-12);
V(num[2])<+transition(NUM[2],0,1e-12,1e-12);
V(num[3])<+transition(NUM[3],0,1e-12,1e-12);
V(num[4])<+transition(NUM[4],0,1e-12,1e-12);
V(num[5])<+transition(NUM[5],0,1e-12,1e-12);
V(num[6])<+transition(NUM[6],0,1e-12,1e-12);
```

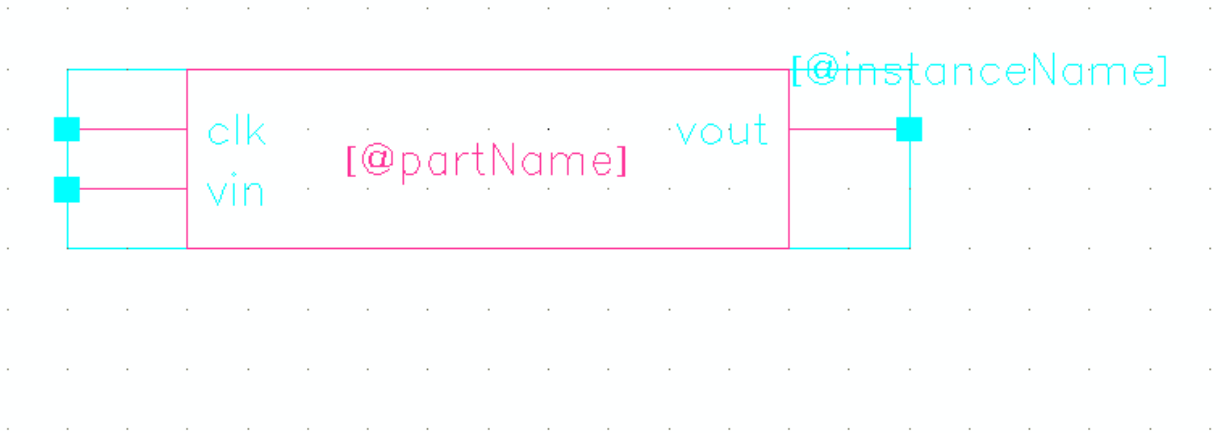
```
V(num[7])<+transition(NUM[7],0,1e-12,1e-12);
V(num[8])<+transition(NUM[8],0,1e-12,1e-12);
V(num[9])<+transition(NUM[9],0,1e-12,1e-12);
V(num[10])<+transition(NUM[10],0,1e-12,1e-12);
V(num[11])<+transition(NUM[11],0,1e-12,1e-12);
V(num[12])<+transition(NUM[12],0,1e-12,1e-12);
V(num[13])<+transition(NUM[13],0,1e-12,1e-12);
V(num[14])<+transition(NUM[14],0,1e-12,1e-12);
V(num[15])<+transition(NUM[15],0,1e-12,1e-12);
V(num[16])<+transition(NUM[16],0,1e-12,1e-12);

V(num[17])<+transition(NUM[17],0,1e-12,1e-12);
V(num[18])<+transition(NUM[18],0,1e-12,1e-12);
V(num[19])<+transition(NUM[19],0,1e-12,1e-12);
V(num[20])<+transition(NUM[20],0,1e-12,1e-12);
V(num[21])<+transition(NUM[21],0,1e-12,1e-12);
V(num[22])<+transition(NUM[22],0,1e-12,1e-12);
V(num[23])<+transition(NUM[23],0,1e-12,1e-12);
V(num[24])<+transition(NUM[24],0,1e-12,1e-12);
V(num[25])<+transition(NUM[25],0,1e-12,1e-12);
V(num[26])<+transition(NUM[26],0,1e-12,1e-12);
V(num[27])<+transition(NUM[27],0,1e-12,1e-12);
V(num[28])<+transition(NUM[28],0,1e-12,1e-12);
V(num[29])<+transition(NUM[29],0,1e-12,1e-12);
V(num[30])<+transition(NUM[30],0,1e-12,1e-12);
V(num[31])<+transition(NUM[31],0,1e-12,1e-12);
V(num[32])<+transition(NUM[32],0,1e-12,1e-12);


V(num[33])<+transition(NUM[33],0,1e-12,1e-12);
V(num[34])<+transition(NUM[34],0,1e-12,1e-12);
V(num[35])<+transition(NUM[35],0,1e-12,1e-12);
V(num[36])<+transition(NUM[36],0,1e-12,1e-12);
V(num[37])<+transition(NUM[37],0,1e-12,1e-12);
V(num[38])<+transition(NUM[38],0,1e-12,1e-12);
V(num[39])<+transition(NUM[39],0,1e-12,1e-12);
V(num[40])<+transition(NUM[40],0,1e-12,1e-12);
V(num[41])<+transition(NUM[41],0,1e-12,1e-12);
V(num[42])<+transition(NUM[42],0,1e-12,1e-12);
V(num[43])<+transition(NUM[43],0,1e-12,1e-12);
```

```
V(num[44])<+transition(NUM[44],0,1e-12,1e-12);
V(num[45])<+transition(NUM[45],0,1e-12,1e-12);
V(num[46])<+transition(NUM[46],0,1e-12,1e-12);
V(num[47])<+transition(NUM[47],0,1e-12,1e-12);
V(num[48])<+transition(NUM[48],0,1e-12,1e-12);

V(num[49])<+transition(NUM[49],0,1e-12,1e-12);
V(num[50])<+transition(NUM[50],0,1e-12,1e-12);
V(num[51])<+transition(NUM[51],0,1e-12,1e-12);
V(num[52])<+transition(NUM[52],0,1e-12,1e-12);
V(num[53])<+transition(NUM[53],0,1e-12,1e-12);
V(num[54])<+transition(NUM[54],0,1e-12,1e-12);
V(num[55])<+transition(NUM[55],0,1e-12,1e-12);
V(num[56])<+transition(NUM[56],0,1e-12,1e-12);
V(num[57])<+transition(NUM[57],0,1e-12,1e-12);
V(num[58])<+transition(NUM[58],0,1e-12,1e-12);
V(num[59])<+transition(NUM[59],0,1e-12,1e-12);
V(num[60])<+transition(NUM[60],0,1e-12,1e-12);
V(num[61])<+transition(NUM[61],0,1e-12,1e-12);
V(num[62])<+transition(NUM[62],0,1e-12,1e-12);
V(num[63])<+transition(NUM[63],0,1e-12,1e-12);
V(num[64])<+transition(NUM[64],0,1e-12,1e-12);

V(num[65])<+transition(NUM[65],0,1e-12,1e-12);
V(num[66])<+transition(NUM[66],0,1e-12,1e-12);
V(num[67])<+transition(NUM[67],0,1e-12,1e-12);
V(num[68])<+transition(NUM[68],0,1e-12,1e-12);
V(num[69])<+transition(NUM[69],0,1e-12,1e-12);
V(num[70])<+transition(NUM[70],0,1e-12,1e-12);
V(num[71])<+transition(NUM[71],0,1e-12,1e-12);
V(num[72])<+transition(NUM[72],0,1e-12,1e-12);
V(num[73])<+transition(NUM[73],0,1e-12,1e-12);
V(num[74])<+transition(NUM[74],0,1e-12,1e-12);
V(num[75])<+transition(NUM[75],0,1e-12,1e-12);
V(num[76])<+transition(NUM[76],0,1e-12,1e-12);
V(num[77])<+transition(NUM[77],0,1e-12,1e-12);
V(num[78])<+transition(NUM[78],0,1e-12,1e-12);
V(num[79])<+transition(NUM[79],0,1e-12,1e-12);
V(num[80])<+transition(NUM[80],0,1e-12,1e-12);
```

end // analog event end

endmodule

Symbol for Staircase generator at input

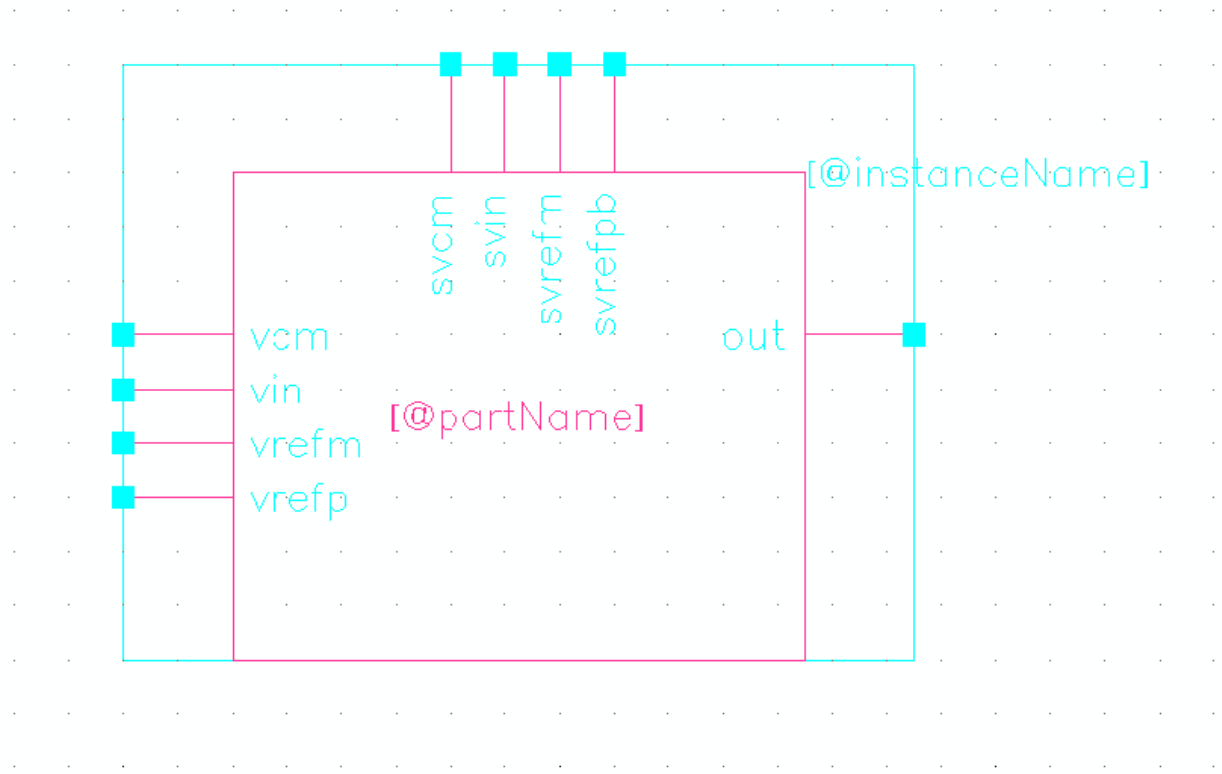


VerilogA Code for staircase generator

```
// VerilogA for mylib, sample and hold, veriloga

`include "constants.vams"
`include "disciplines.vams"

module sample_and_hold_022708_B(clk, vin, vout);
input clk, vin;
output vout;

electrical clk, vin, vout;

//file handle
integer fs1;

real v1;
// v1 is the sample point from the input waveform when the clock edge is falling


analog begin
  @(initial_step) begin
          v1 = 2.5;
```

```
  end

  // Sample at rising edge
  @( cross( V(clk)-1.25, +1) ) begin
       v1 = V(vin);
  end

  // Sample at falling edge
  @( cross( V(clk)-1.25, -1) ) begin
       v1 = V(vin);
  end

     V(vout) <+ transition(v1,0,1e-12,1e-12);

end

endmodule
```

Symbol for Ideal Switch

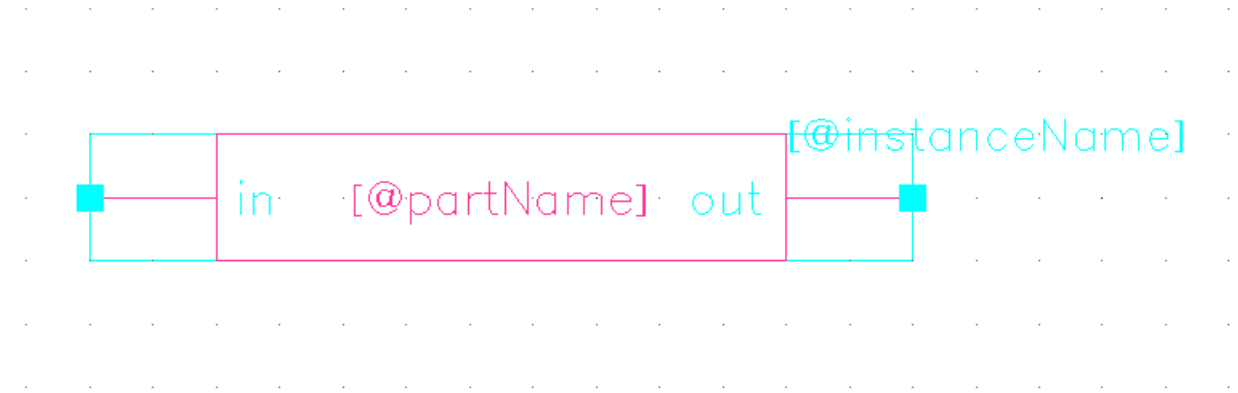


VerilogA Code for Ideal Switch

```
// VerilogA for newlib, switch, veriloga

`include "constants.vams"
`include "disciplines.vams"

module switch(svin, svrefm, svcm, svrefpb, vin,vrefm, vcm,vrefp,out);
input svin, svrefm, svcm, svrefpb, vin,vrefm, vcm,vrefp;
output out;

electrical svin, svrefm, svcm, svrefpb, vin,vrefm, vcm,vrefp,out;
real SVIN, SVREFM, SVCM, SVREFPB, VIN, VREFM, VCM, VREFP, OUT;

analog begin
```

```
SVIN = V(svin);
SVREFM = V(svrefm);
SVCM = V(svcm);
SVREFPB = V(svrefpb);
VIN = V(vin);
VREFM = V(vrefm);
VCM = V(vcm);
VREFP = V(vrefp);

case(1)
  SVIN > 1.25:     OUT = VIN;
  SVREFM > 1.25:   OUT = VREFM;
  SVCM > 1.25:     OUT = VCM;
  SVREFPB < 1.25:  OUT = VREFP;
endcase

V(out) <+ transition(OUT,0,1e-12, 1e-12);

end
endmodule
```

Symbol for Ideal Inverter



VerilogA Code for Ideal Inverter

```
// VerilogA for newlib, inverter, veriloga

`include "constants.vams"
`include "disciplines.vams"
```

```
module inverter(out, in);
output out;
electrical out;
input in;
electrical in;

parameter real vddl = 2.5;
parameter real in_low = 1.24;
parameter real in_high = 1.25;

real out_val;

analog begin

  if (V(in) < in_low) begin
  out_val = vddl;
  end
  else if (V(in) > in_high) begin
  out_val = 0;
  end
  else out_val = -vddl*V(in)/(in_high-in_low)+
      vddl*in_high/(in_high-in_low);

 V(out) <+ transition(out_val,0, 1e-12,1e-12);
 end

endmodule
```

Appendix B

Non-ideal DAC weights in segment 2-5

Table I: DAC weights in segment 2

	Ideal	With Mismatch		
C	Voltage weight		**Weight errors εiB**	**Ratio of Mismatch/ Ideal**
17	8.5451830E-03	8.5686140E-03	2.3431000E-05	1.0027420
18	8.5451830E-03	8.5686140E-03	2.3431000E-05	1.0027420
19	8.5451830E-03	8.5686140E-03	2.3431000E-05	1.0027420
20	8.5451830E-03	8.5686140E-03	2.3431000E-05	1.0027420
21	8.5451830E-03	8.5686140E-03	2.3431000E-05	1.0027420
22	8.5451830E-03	8.5686140E-03	2.3431000E-05	1.0027420
23	8.5451830E-03	8.5686140E-03	2.3431000E-05	1.0027420
24	8.5451830E-03	8.5686140E-03	2.3431000E-05	1.0027420
25	8.5451830E-03	8.5686140E-03	2.3431000E-05	1.0027420
26	8.5451830E-03	8.5686140E-03	2.3431000E-05	1.0027420
27	8.5451830E-03	8.5686140E-03	2.3431000E-05	1.0027420
28	8.5451830E-03	8.5686140E-03	2.3431000E-05	1.0027420
29	8.5451830E-03	8.5686140E-03	2.3431000E-05	1.0027420
30	8.5451830E-03	8.5686140E-03	2.3431000E-05	1.0027420
31	8.5451830E-03	8.5686140E-03	2.3431000E-05	1.0027420
32	8.5451830E-03	8.5686140E-03	2.3431000E-05	1.0027420

Table II: DAC weights in segment 3

	Ideal	**With Mismatch**	**Weight errors εiB**	**Ratio of Mismatch/ Ideal**
C	Voltage weight			
33	1.0681480E-03	1.0710770E-03	2.9290000E-06	1.0027421
34	1.0681480E-03	1.0710770E-03	2.9290000E-06	1.0027421
35	1.0681480E-03	1.0710770E-03	2.9290000E-06	1.0027421
36	1.0681480E-03	1.0710770E-03	2.9290000E-06	1.0027421
37	1.0681480E-03	1.0710770E-03	2.9290000E-06	1.0027421
38	1.0681480E-03	1.0710770E-03	2.9290000E-06	1.0027421
39	1.0681480E-03	1.0710770E-03	2.9290000E-06	1.0027421
40	1.0681480E-03	1.0710770E-03	2.9290000E-06	1.0027421
41	1.0681480E-03	1.0710770E-03	2.9290000E-06	1.0027421
42	1.0681480E-03	1.0710770E-03	2.9290000E-06	1.0027421
43	1.0681480E-03	1.0710770E-03	2.9290000E-06	1.0027421
44	1.0681480E-03	1.0710770E-03	2.9290000E-06	1.0027421
45	1.0681480E-03	1.0710770E-03	2.9290000E-06	1.0027421
46	1.0681480E-03	1.0710770E-03	2.9290000E-06	1.0027421
47	1.0681480E-03	1.0710770E-03	2.9290000E-06	1.0027421
48	1.0681480E-03	1.0710770E-03	2.9290000E-06	1.0027421

Table III. DAC weights in segment 4

	Ideal	**With Mismatch**	**Weight errors εiB**	**Ratio of Mismatch/ Ideal**
C	Voltage weight			
49	1.3350820E-04	1.3387430E-04	3.6610000E-07	1.0027422
50	1.3350820E-04	1.3387430E-04	3.6610000E-07	1.0027422
51	1.3350820E-04	1.3387430E-04	3.6610000E-07	1.0027422
52	1.3350820E-04	1.3387430E-04	3.6610000E-07	1.0027422
53	1.3350820E-04	1.3387430E-04	3.6610000E-07	1.0027422
54	1.3350820E-04	1.3387430E-04	3.6610000E-07	1.0027422
55	1.3350820E-04	1.3387430E-04	3.6610000E-07	1.0027422
56	1.3350820E-04	1.3387430E-04	3.6610000E-07	1.0027422
57	1.3350820E-04	1.3387430E-04	3.6610000E-07	1.0027422
58	1.3350820E-04	1.3387430E-04	3.6610000E-07	1.0027422
59	1.3350820E-04	1.3387430E-04	3.6610000E-07	1.0027422
60	1.3350820E-04	1.3387430E-04	3.6610000E-07	1.0027422
61	1.3350820E-04	1.3387430E-04	3.6610000E-07	1.0027422
62	1.3350820E-04	1.3387430E-04	3.6610000E-07	1.0027422
63	1.3350820E-04	1.3387430E-04	3.6610000E-07	1.0027422
64	1.3350820E-04	1.3387430E-04	3.6610000E-07	1.0027422

Table IV. DAC weights in segment 5

	Ideal	With Mismatch	Weight errors εiB	Ratio of Mismatch/ Ideal
C	Voltage weight			
65	1.6688290E-05	1.6734050E-05	4.5760000E-08	1.0027420
66	1.6688290E-05	1.6734050E-05	4.5760000E-08	1.0027420
67	1.6688290E-05	1.6734050E-05	4.5760000E-08	1.0027420
68	1.6688290E-05	1.6734050E-05	4.5760000E-08	1.0027420
69	1.6688290E-05	1.6734050E-05	4.5760000E-08	1.0027420
70	1.6688290E-05	1.6734050E-05	4.5760000E-08	1.0027420
71	1.6688290E-05	1.6734050E-05	4.5760000E-08	1.0027420
72	1.6688290E-05	1.6734050E-05	4.5760000E-08	1.0027420
73	1.6688290E-05	1.6734050E-05	4.5760000E-08	1.0027420
74	1.6688290E-05	1.6734050E-05	4.5760000E-08	1.0027420
75	1.6688290E-05	1.6734050E-05	4.5760000E-08	1.0027420
76	1.6688290E-05	1.6734050E-05	4.5760000E-08	1.0027420
77	1.6688290E-05	1.6734050E-05	4.5760000E-08	1.0027420
78	1.6688290E-05	1.6734050E-05	4.5760000E-08	1.0027420
79	1.6688290E-05	1.6734050E-05	4.5760000E-08	1.0027420
80	1.6688290E-05	1.6734050E-05	4.5760000E-08	1.0027420

Appendix C

Matlab code used for the system verification

Interface between the mixed signal IC and FPGA

```
clear all;
close all;
clc;

%%%%%%%%%%Overall function%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% This code do several things
%1) Set up the ADC parameters
%2.) Plot ADC error before correction
%3.) Apply error correction algorithm
%4.) Plot ADC error after correction

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%ADC setup%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% N denotes the number of bits of the ADC
N = 20;

 % These are weights got from the perfect 16 bit ADC step size (Book5 pg 54)
W1 = 2*6.836146e-2;
W2 = 2*8.545183e-3;
W3 = 2*1.068148e-3;
W4 = 2*1.335082e-4;
W5 = 2*1.668829e-5;

% This denotes the initial weight used for both ADC_A and ADC_B in the
% digital to analog interface. The decisions from ADC_A will by muliplied
% by the estimated weight WA to get the analog voltage. The same happen for
% ADC_B. We assume we DO NOT KNOW of any error in the ADC itself, so we can
% just estimate them

WA = [W1*ones(16,1);
      W2*ones(16,1);
      W3*ones(16,1);
      W4*ones(16,1);
      W5*ones(16,1)];

WB = [W1*ones(16,1);
      W2*ones(16,1);
      W3*ones(16,1);
      W4*ones(16,1);
      W5*ones(16,1)];


% Initialization for before/after plot
INL = zeros(1,2^16);
DNL = zeros(1,2^16);
dec_A_ramp = zeros(65536*16,50);
dec_B_ramp = zeros(65536*16,50);
INL_after = zeros(1,2^16);
```

```
DNL_after = zeros(1,2^16);

Anum1 = zeros(1,16);
Anum2 = zeros(1,16);
Anum3 = zeros(1,16);
Anum4 = zeros(1,16);
Anum5 = zeros(1,16);

Bnum1 = zeros(1,16);
Bnum2 = zeros(1,16);
Bnum3 = zeros(1,16);
Bnum4 = zeros(1,16);
Bnum5 = zeros(1,16);

% Initialization
dec_A = zeros(128,50);
dec_B = zeros(128,50);
err = zeros(100,1);

resolution = 2.5/(2^16*16);
%VLSB = 2*resolution;

step = 2.5/2^16;


% Define Vin vector. A ramp input is used in this case.
Vin2 = [0: resolution: 2.5-resolution];
Vin1 = [2.5-resolution: -resolution: 0];

 Vsample2 = [0:step:2.5];
 Vsample1 = [2.5:-step:0];



%%%%%%%%%%%%%%%%ADC setup%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%INL/DNL plot before error correction

%Add some noise
for i = 1: length(Vin1)
    Vin1(i) = Vin1(i)+ randn*(30*10^-6);
    Vin2(i) = Vin2(i)+ randn*(30*10^-6);
end

% Convert them
 R013108_ADC_16bit_randomization;
  Vin_ramp1 = Vin1;
  Vin_ramp2 = Vin2;
  new_dA_ramp = new_dA;
  new_dB_ramp = new_dB;
```

```
  Vin_bar = (new_dB_ramp*WB + new_dA_ramp*WA)/2;

  figure(1)
  plot((Vin2-Vin1)', Vin_bar, 'r');
  hold on;
  fita = polyfit((Vin2-Vin1)', Vin_bar, 1);
  Vin_bar_mod = (Vin_bar-fita(2))*1/fita(1);
  plot((Vin2-Vin1)', Vin_bar_mod, 'g');
  axis([-2.5 2.5 -2.5 2.5])

  count = histc((Vin_bar_mod)', ([-5 Vsample2(2:65536)-Vsample1(2:65536)
5]));
  DNL = (count(1:65536)-16)/16;
  INL = cumsum(DNL);

  Code = 0:1:65535;

 figure(2)
 subplot(2,1,1)
 plot(Code, [0, INL(2:65535),0]);
 title('Before Correction');
 xlabel('Code');
 ylabel('INL(in LSB)');

 subplot(2,1,2)
 plot(Code, [0, DNL(2:65535), 0]);
 title('Before Correction');
 xlabel('Code');
 ylabel('DNL(in LSB)');



%%%%%%%%%Plot ADC error before correction
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%Error correction

% Supply a sine input
resolution = 0:1e-3:1280;
Input2 = 1.25*sin(resolution)+1.25;
Input1 = -1.25*sin(resolution)+1.25;

% Add some noise
for i = 1: length(Input2)
    Input1(i) = Input1(i)+ randn*(30*10^-6);
    Input2(i) = Input2(i)+ randn*(30*10^-6);
end

figure(3)
plot(resolution(1:10000),Input2(1:10000), resolution(1:10000),
Input1(1:10000));
```

```
% Rearranging weight
%group cap weight in segment 4 as one
%group cap weight in segment 5 as one
q1 = sum(WA(49:64,1));
q2 = sum(WA(65:80,1));
q3 = sum(WB(49:64,1));
q4 = sum(WB(65:80,1));

% WAside and WBside serve as the updated weight. For each 128 conversions,
% They will be updated

WAside = [WA(1:48); q1; q2];
WBside = [WB(1:48); q3; q4];

for x = 1: 10000

     % This is for sine input only. Using different parts of the sine
     % signal for error correction
     Vin2 = Input2(1,128*(x-1)+1: 128*x);
     Vin1 = Input1(1,128*(x-1)+1: 128*x);

    % Convert using ADC code
    R013108_ADC_16bit_randomization;
    % Apply error correction
    R013108_error_correction;
end

%%%%%%%%%%%%%%%%Error correction
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%Plot ADC error after error correction

% Rearranging decision A and decision B so that they can be easily
% multipled by the "corrected" weights
for i = 1:length(Vin_ramp1)
    t1 = sum(new_dA_ramp(i,49:64));
    t2 = sum(new_dA_ramp(i,65:80));
    dec_A_ramp(i,:) = [new_dA_ramp(i,1:48) t1 t2];

    t3 = sum(new_dB_ramp(i,49:64));
    t4 = sum(new_dB_ramp(i,65:80));
    dec_B_ramp(i,:) = [new_dB_ramp(i,1:48) t3 t4];
end

% Getting the analog output code using the "corrected" weights
Vin_bar_after = (dec_B_ramp*WBside + dec_A_ramp*WAside)/2;


resolution = 2.5/(2^16*16);
Vin_ideal2 = [0: resolution: 2.5];
```

```
Vin_ideal1 = [2.5: -resolution: 0];

Vin_ideal2 = Vin_ideal2(1:2^16*16);
Vin_ideal1 = Vin_ideal1(1:2^16*16);

figure(4)
plot((Vin_ideal2-Vin_ideal1)', Vin_bar_after, 'r');
hold on;
fita = polyfit((Vin_ideal2-Vin_ideal1)', Vin_bar_after, 1);
Vin_bar_after_mod = (Vin_bar_after-fita(2))*1/fita(1);
plot((Vin_ideal2-Vin_ideal1)', Vin_bar_after_mod, 'g');
axis([-2.5 2.5 -2.5 2.5])

count = histc((Vin_bar_after_mod)', ([-5 Vsample2(2:65536)-Vsample1(2:65536)
5]));
DNL_after = (count(1:65536)-16)/16;
INL_after = cumsum(DNL_after);

Code = 0:1:65535;

figure(5)
subplot(2,1,1)
plot(INL_after);
title('After Correction');
xlabel('Code');
ylabel('INL(in LSB)');
axis([0 65535 -1 1]);


subplot(2,1,2)
plot(DNL_after);
title('After Correction');
xlabel('Code');
ylabel('DNL(in LSB)');
axis([0 65535 -1 1]);
```

Modified SAR ADC

```
%%%%%%%%%%%%%%%%%%%%%%%Overall function%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%This code composes of two ADCs. Two ADCs each convert the same input
%into a bunch of decisions, according to their set weights. In this example,
%ADCA's weight is perfect, and zero biased errors are added to first segment
%of ADCB(bottom row).

% Two process contribute to the difference in decisions
%1.) Randomization using rand
%2.) error in ADC_B
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%ADC_A

% %Ideal cap weight, top row(C1-C80)
AWT_seg1 = ones(1,16)*W1*1/2;
AWT_seg2 = ones(1,16)*W2*1/2;
```

```
AWT_seg3 = ones(1,16)*W3*1/2;
AWT_seg4 = ones(1,16)*W4*1/2;
AWT_seg5 = ones(1,16)*W5*1/2;

%Ideal cap weight, bottom row(C1-C80)
AWB_seg1 = ones(1,16)*W1*1/2;
AWB_seg2 = ones(1,16)*W2*1/2;
AWB_seg3 = ones(1,16)*W3*1/2;
AWB_seg4 = ones(1,16)*W4*1/2;
AWB_seg5 = ones(1,16)*W5*1/2;

% Add some error to the bottom row cap in ADC_A, to segment 1,2,3,4,5
% Just like Book 6 page 36, 37 (zero basied error)

%R013108_MismatchA_non_zero;

%Sample Voltage points
S = length(Vin1);

% Initialize Vx and Vy, and vcomp
AVx = zeros(N+1,S);
AVy = zeros(N+1,S);
AVcomp = zeros(N+1,S);
Astore_draw = zeros(S,5);
% Anum1 = zeros(1,16);
% Anum2 = zeros(1,16);
% Anum3 = zeros(1,16);
% Anum4 = zeros(1,16);
% Anum5 = zeros(1,16);

% decision_matrix
dA = zeros(N,length(Vin1));

% reconstructed decision matrix
new_dA = zeros(length(Vin2),80);

% linear fit got from plotting Vx, Vy against Vin in the ideal case
% Assume it does not change much!?

fity = [-0.461439915525947   1.249999999621569];
fitx = [0.461439914252529   1.250000002374065];

% Sample + Hold Mode
% Vy is the bottom row, Vx is the top row
% The linear fit helps to get the first Vx, Vy point. Afterwards, it's all
% up to the weight.
AVx(1,:) = fitx(1)*(Vin2-Vin1)+fitx(2);
AVy(1,:) = fity(1)*(Vin2-Vin1)+fity(2);
AVcomp(1,:) = AVy(1,:)-AVx(1,:);

% Bit cycle mode
  for j = 1:S
```

```
      % Initiating randomization. If num 1 = 8, then num2 = 9. the number
goes in
    % a wheel fashion

    Adraw1 = floor(16*rand)+1;
      for q = 1:16
         Anum1(q) = mod(Adraw1+(q-1),16);
         if Anum1(q) == 0
             Anum1(q) = 16;
         end;
      end;

   Adraw2 = floor(16*rand)+1;
      for q = 1:16
         Anum2(q) = mod(Adraw2+(q-1),16);
         if Anum2(q) == 0
             Anum2(q) = 16;
         end;
      end;

   Adraw3 = floor(16*rand)+1;
      for q = 1:16
         Anum3(q) = mod(Adraw3+(q-1),16);
         if Anum3(q) == 0
             Anum3(q) = 16;
         end;
      end;

   Adraw4 = floor(16*rand)+1;
      for q = 1:16
         Anum4(q) = mod(Adraw4+(q-1),16);
         if Anum4(q) == 0
             Anum4(q) = 16;
         end;
      end;

   Adraw5 = floor(16*rand)+1;
      for q = 1:16
         Anum5(q) = mod(Adraw5+(q-1),16);
         if Anum5(q) == 0
             Anum5(q) = 16;
         end;
      end;

    Astore_draw(j,1:5) = [Adraw1 Adraw2 Adraw3 Adraw4 Adraw5];


    Ab1 = find(Anum1 >=1 & Anum1 <= 8);
    Ab2 = find(Anum1 >=9 & Anum1 <=12);
    Ab3 = find(Anum1>=13 & Anum1 <= 14);
    Ab4a = find(Anum1 == 15);
    Aunused1 = find(Anum1 ==16);

    Ab4b = find(Anum2 >=1 & Anum2 <= 8);
    Ab5 = find(Anum2 >=9 & Anum2 <=12);
```

```
Ab6 = find(Anum2>=13 & Anum2 <= 14);
Ab7a = find(Anum2 == 15);
Aunused2 = find(Anum2 ==16);

Ab7b = find(Anum3 >=1 & Anum3 <= 8);
Ab8 = find(Anum3 >=9 & Anum3 <=12);
Ab9 = find(Anum3>=13 & Anum3 <= 14);
Ab10a = find(Anum3 == 15);
Aunused3 = find(Anum3 ==16);

Ab10b = find(Anum4 >=1 & Anum4 <= 8);
Ab11 = find(Anum4 >=9 & Anum4 <=12);
Ab12 = find(Anum4>=13 & Anum4 <= 14);
Ab13a = find(Anum4 == 15);
Aunused4 = find(Anum4 ==16);

Ab13b = find(Anum5 >=1 & Anum5 <= 8);
Ab14 = find(Anum5 >=9 & Anum5 <=12);
Ab15 = find(Anum5>=13 & Anum5 <= 14);
Ab16 = find(Anum5 == 15);
Aunused5 = find(Anum5 ==16);

% Sum up the weight used for each bit(Top row)
AWT(1) = sum(AWT_seg1(Ab1));
AWT(2) = sum(AWT_seg1(Ab2));
AWT(3) = sum(AWT_seg1(Ab3));
AWT(4) = sum(AWT_seg1(Ab4a));

AWT(5) = sum(AWT_seg2(Ab4b));
AWT(6) = sum(AWT_seg2(Ab5));
AWT(7) = sum(AWT_seg2(Ab6));
AWT(8) = sum(AWT_seg2(Ab7a));

AWT(9)  = sum(AWT_seg3(Ab7b));
AWT(10) = sum(AWT_seg3(Ab8));
AWT(11) = sum(AWT_seg3(Ab9));
AWT(12) = sum(AWT_seg3(Ab10a));

AWT(13)  = sum(AWT_seg4(Ab10b));
AWT(14) = sum(AWT_seg4(Ab11));
AWT(15) = sum(AWT_seg4(Ab12));
AWT(16) = sum(AWT_seg4(Ab13a));

AWT(17)  = sum(AWT_seg5(Ab13b));
AWT(18) = sum(AWT_seg5(Ab14));
AWT(19) = sum(AWT_seg5(Ab15));
AWT(20) = sum(AWT_seg5(Ab16));

% Sum up the weight used for each bit(Bottom row)
AWB(1) = sum(AWB_seg1(Ab1));
AWB(2) = sum(AWB_seg1(Ab2));
AWB(3) = sum(AWB_seg1(Ab3));
AWB(4) = sum(AWB_seg1(Ab4a));

AWB(5) = sum(AWB_seg2(Ab4b));
```

```
AWB(6) = sum(AWB_seg2(Ab5));
AWB(7) = sum(AWB_seg2(Ab6));
AWB(8) = sum(AWB_seg2(Ab7a));

AWB(9)  = sum(AWB_seg3(Ab7b));
AWB(10) = sum(AWB_seg3(Ab8));
AWB(11) = sum(AWB_seg3(Ab9));
AWB(12) = sum(AWB_seg3(Ab10a));

AWB(13)  = sum(AWB_seg4(Ab10b));
AWB(14) = sum(AWB_seg4(Ab11));
AWB(15) = sum(AWB_seg4(Ab12));
AWB(16) = sum(AWB_seg4(Ab13a));

AWB(17)  = sum(AWB_seg5(Ab13b));
AWB(18) = sum(AWB_seg5(Ab14));
AWB(19) = sum(AWB_seg5(Ab15));
AWB(20) = sum(AWB_seg5(Ab16));

for n = 1:N

    dA(n,j) = -sign(AVcomp(n,j)-0+ randn*(30*10^-6));

 % Only d= 1 and d=-1 is allowed. If d = 0, this indicates Vin = Vdac,
 % so a decision bit 1 should be assigned.

    if AVy(n,j)== AVx(n,j)
       dA(n,j) = 1;
    end;

   AVy(n+1,j) = AVy(n,j) + dA(n,j)*AWB(n);
   AVx(n+1,j) = AVx(n,j) - dA(n,j)*AWT(n);
   AVcomp(n+1,j) = AVy(n+1,j) - AVx(n+1,j);


 new_dA(j,Ab1) = dA(1,j);
 new_dA(j,Ab2) = dA(2,j);
 new_dA(j,Ab3) = dA(3,j);
 new_dA(j,Ab4a) = dA(4,j);

 new_dA(j,16+Ab4b) = dA(5,j);
 new_dA(j,16+Ab5) = dA(6,j);
 new_dA(j,16+Ab6) = dA(7,j);
 new_dA(j,16+Ab7a) = dA(8,j);

 new_dA(j,32+Ab7b) = dA(9,j);
 new_dA(j,32+Ab8) = dA(10,j);
 new_dA(j,32+Ab9) = dA(11,j);
 new_dA(j,32+Ab10a) = dA(12,j);

 new_dA(j,48+Ab10b) = dA(13,j);
 new_dA(j,48+Ab11) = dA(14,j);
 new_dA(j,48+Ab12) = dA(15,j);
 new_dA(j,48+Ab13a) = dA(16,j);
```

```
      new_dA(j,64+Ab13b) = dA(17,j);
      new_dA(j,64+Ab14) = dA(18,j);
      new_dA(j,64+Ab15) = dA(19,j);
      new_dA(j,64+Ab16) = dA(20,j);

     end; %(for n loop)


 end; % (for j loop)
%
%%%%%%%%%%%%%%%%%%%%%ADC_A%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%ADC_B

%Ideal cap weight, top row(C1-C80)
BWT_seg1 = ones(1,16)*1/2*W1;
BWT_seg2 = ones(1,16)*1/2*W2;
BWT_seg3 = ones(1,16)*1/2*W3;
BWT_seg4 = ones(1,16)*1/2*W4;
BWT_seg5 = ones(1,16)*1/2*W5;

%Ideal cap weight, bottom row(C1-C80)
BWB_seg1 = ones(1,16)*1/2*W1;
BWB_seg2 = ones(1,16)*1/2*W2;
BWB_seg3 = ones(1,16)*1/2*W3;
BWB_seg4 = ones(1,16)*1/2*W4;
BWB_seg5 = ones(1,16)*1/2*W5;

% Add some error to the bottom row cap in ADC_B, to segment 1,2,3,4,5
% Just like Book 6 page 36, 37 (zero basied error)

R013108_MismatchB;

%Sample Voltage points
S = length(Vin1);

% Initialize Vx and Vy, and vcomp
BVx = zeros(N+1,S);
BVy = zeros(N+1,S);

BVcomp = zeros(N+1,S);
Bstore_draw = zeros(S,5);
% Bnum1 = zeros(1,16);
% Bnum2 = zeros(1,16);
% Bnum3 = zeros(1,16);
% Bnum4 = zeros(1,16);
% Bnum5 = zeros(1,16);
%
% decision_matrix
dB = zeros(N,length(Vin1));
```

```
% reconstructed decision matrix
new_dB = zeros(length(Vin2),80);

% linear fit got from plotting Vx, Vy against Vin in the ideal case
% Assume it does not change much!?

fity = [-0.461439915525947   1.249999999621569];
fitx = [0.461439914252529   1.250000002374065];

% Sample + Hold Mode
% Vy is the bottom row, Vx is the top row
% The linear fit helps to get the first Vx, Vy point. Afterwards, it's all
% up to the weight.
BVx(1,:) = fitx(1)*(Vin2-Vin1)+fitx(2);
BVy(1,:) = fity(1)*(Vin2-Vin1)+fity(2);
BVcomp(1,:) = BVy(1,:)-BVx(1,:);

% Bit cycle mode
  for j = 1:S

      % Initiating randomization. If num 1 = 8, then num2 = 9. the number
goes in
    % a wheel fashion

    Bdraw1 = floor(16*rand)+1;
      for q = 1:16
         Bnum1(q) = mod(Bdraw1+(q-1),16);
         if Bnum1(q) == 0
             Bnum1(q) = 16;
         end;
      end;

   Bdraw2 = floor(16*rand)+1;
      for q = 1:16
         Bnum2(q) = mod(Bdraw2+(q-1),16);
         if Bnum2(q) == 0
             Bnum2(q) = 16;
         end;
      end;

   Bdraw3 = floor(16*rand)+1;
      for q = 1:16
         Bnum3(q) = mod(Bdraw3+(q-1),16);
         if Bnum3(q) == 0
             Bnum3(q) = 16;
         end;
      end;

   Bdraw4 = floor(16*rand)+1;
      for q = 1:16
         Bnum4(q) = mod(Bdraw4+(q-1),16);
         if Bnum4(q) == 0
             Bnum4(q) = 16;
         end;
```

```
   end;

Bdraw5 = floor(16*rand)+1;
   for q = 1:16
      Bnum5(q) = mod(Bdraw5+(q-1),16);
      if Bnum5(q) == 0
          Bnum5(q) = 16;
      end;
   end;

 Bstore_draw(j,1:5) = [Bdraw1 Bdraw2 Bdraw3 Bdraw4 Bdraw5];

 Bb1 = find(Bnum1 >=1 & Bnum1 <= 8);
 Bb2 = find(Bnum1 >=9 & Bnum1 <=12);
 Bb3 = find(Bnum1>=13 & Bnum1 <= 14);
 Bb4a = find(Bnum1 == 15);
 Bunused1 = find(Bnum1 ==16);

 Bb4b = find(Bnum2 >=1 & Bnum2 <= 8);
 Bb5 = find(Bnum2 >=9 & Bnum2 <=12);
 Bb6 = find(Bnum2>=13 & Bnum2 <= 14);
 Bb7a = find(Bnum2 == 15);
 Bunused2 = find(Bnum2 ==16);

 Bb7b = find(Bnum3 >=1 & Bnum3 <= 8);
 Bb8 = find(Bnum3 >=9 & Bnum3 <=12);
 Bb9 = find(Bnum3>=13 & Bnum3 <= 14);
 Bb10a = find(Bnum3 == 15);
 Bunused3 = find(Bnum3 ==16);

 Bb10b = find(Bnum4 >=1 & Bnum4 <= 8);
 Bb11 = find(Bnum4 >=9 & Bnum4 <=12);
 Bb12 = find(Bnum4>=13 & Bnum4 <= 14);
 Bb13a = find(Bnum4 == 15);
 Bunused4 = find(Bnum4 ==16);

 Bb13b = find(Bnum5 >=1 & Bnum5 <= 8);
 Bb14 = find(Bnum5 >=9 & Bnum5 <=12);
 Bb15 = find(Bnum5>=13 & Bnum5 <= 14);
 Bb16 = find(Bnum5 == 15);
 Bunused5 = find(Bnum5 ==16);

 % sum up the weights used for each bit(Top row)
 BWT(1) = sum(BWT_seg1(Bb1));
 BWT(2) = sum(BWT_seg1(Bb2));
 BWT(3) = sum(BWT_seg1(Bb3));
 BWT(4) = sum(BWT_seg1(Bb4a));

 BWT(5) = sum(BWT_seg2(Bb4b));
 BWT(6) = sum(BWT_seg2(Bb5));
 BWT(7) = sum(BWT_seg2(Bb6));
 BWT(8) = sum(BWT_seg2(Bb7a));

 BWT(9)  = sum(BWT_seg3(Bb7b));
 BWT(10) = sum(BWT_seg3(Bb8));
```

```
BWT(11) = sum(BWT_seg3(Bb9));
BWT(12) = sum(BWT_seg3(Bb10a));

BWT(13)  = sum(BWT_seg4(Bb10b));
BWT(14) = sum(BWT_seg4(Bb11));
BWT(15) = sum(BWT_seg4(Bb12));
BWT(16) = sum(BWT_seg4(Bb13a));

BWT(17)  = sum(BWT_seg5(Bb13b));
BWT(18) = sum(BWT_seg5(Bb14));
BWT(19) = sum(BWT_seg5(Bb15));
BWT(20) = sum(BWT_seg5(Bb16));

% sum up the weight used for each bit(Bottom row)
BWB(1) = sum(BWB_seg1(Bb1));
BWB(2) = sum(BWB_seg1(Bb2));
BWB(3) = sum(BWB_seg1(Bb3));
BWB(4) = sum(BWB_seg1(Bb4a));

BWB(5) = sum(BWB_seg2(Bb4b));
BWB(6) = sum(BWB_seg2(Bb5));
BWB(7) = sum(BWB_seg2(Bb6));
BWB(8) = sum(BWB_seg2(Bb7a));

BWB(9)  = sum(BWB_seg3(Bb7b));
BWB(10) = sum(BWB_seg3(Bb8));
BWB(11) = sum(BWB_seg3(Bb9));
BWB(12) = sum(BWB_seg3(Bb10a));

BWB(13)  = sum(BWB_seg4(Bb10b));
BWB(14) = sum(BWB_seg4(Bb11));
BWB(15) = sum(BWB_seg4(Bb12));
BWB(16) = sum(BWB_seg4(Bb13a));

BWB(17)  = sum(BWB_seg5(Bb13b));
BWB(18) = sum(BWB_seg5(Bb14));
BWB(19) = sum(BWB_seg5(Bb15));
BWB(20) = sum(BWB_seg5(Bb16));

for n = 1:N

    dB(n,j) = -sign(BVcomp(n,j)-0 + randn*(30*10^-6));

 % Only d= 1 and d=-1 is allowed. If d = 0, this indicates Vin = Vdac,
 % so a decision bit 1 should be assigned.

    if BVy(n,j)== BVx(n,j)
       dB(n,j) = 1;
    end;

   BVy(n+1,j) = BVy(n,j) + dB(n,j)*BWB(n);
   BVx(n+1,j) = BVx(n,j) - dB(n,j)*BWT(n);
   BVcomp(n+1,j) = BVy(n+1,j) - BVx(n+1,j);
```

```
      new_dB(j,Bb1) = dB(1,j);
      new_dB(j,Bb2) = dB(2,j);
      new_dB(j,Bb3) = dB(3,j);
      new_dB(j,Bb4a) = dB(4,j);

      new_dB(j,16+Bb4b) = dB(5,j);
      new_dB(j,16+Bb5) = dB(6,j);
      new_dB(j,16+Bb6) = dB(7,j);
      new_dB(j,16+Bb7a) = dB(8,j);

      new_dB(j,32+Bb7b) = dB(9,j);
      new_dB(j,32+Bb8) = dB(10,j);
      new_dB(j,32+Bb9) = dB(11,j);
      new_dB(j,32+Bb10a) = dB(12,j);

      new_dB(j,48+Bb10b) = dB(13,j);
      new_dB(j,48+Bb11) = dB(14,j);
      new_dB(j,48+Bb12) = dB(15,j);
      new_dB(j,48+Bb13a) = dB(16,j);

      new_dB(j,64+Bb13b) = dB(17,j);
      new_dB(j,64+Bb14) = dB(18,j);
      new_dB(j,64+Bb15) = dB(19,j);
      new_dB(j,64+Bb16) = dB(20,j);

    end; %(for n loop)


 end; % (for j loop)

%%%%%%%%%%%%%%%%%%%%%%%%%%ADC_B%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Mismatch modeling

```
% Bottom row error, ADCB, seg1
BWB_seg1(1) = BWB_seg1(1)*1.0127695;
BWB_seg1(2) = BWB_seg1(2)*1.0227969;
BWB_seg1(3) = BWB_seg1(3)*0.97265974;
BWB_seg1(4) = BWB_seg1(4)*1.0328243;
BWB_seg1(5) = BWB_seg1(5)*1.0127695;
BWB_seg1(6) = BWB_seg1(6)*0.9826872;
BWB_seg1(7) = BWB_seg1(7)*0.9526049;
BWB_seg1(8) = BWB_seg1(8)*0.9626323;
BWB_seg1(9) = BWB_seg1(9)*1.0127695;
BWB_seg1(10) = BWB_seg1(10)*0.9927146;
BWB_seg1(11) = BWB_seg1(11)*1.0528791;
BWB_seg1(12) = BWB_seg1(12)*0.9826872;
BWB_seg1(13) = BWB_seg1(13)*0.9726597;
BWB_seg1(14) = BWB_seg1(14)*1.0428516;
BWB_seg1(15) = BWB_seg1(15)*0.9626323;
BWB_seg1(16) = BWB_seg1(16)*1.0227969;
```

```
% % Bottom row error, ADCB, seg2
BWB_seg2(1) = BWB_seg2(1)*1.0027420;
BWB_seg2(2) = BWB_seg2(2)*1.0027420;
BWB_seg2(3) = BWB_seg2(3)*1.0027420;
BWB_seg2(4) = BWB_seg2(4)*1.0027420;
BWB_seg2(5) = BWB_seg2(5)*1.0027420;
BWB_seg2(6) = BWB_seg2(6)*1.0027420;
BWB_seg2(7) = BWB_seg2(7)*1.0027420;
BWB_seg2(8) = BWB_seg2(8)*1.0027420;
BWB_seg2(9) = BWB_seg2(9)*1.0027420;
BWB_seg2(10) = BWB_seg2(10)*1.0027420;
BWB_seg2(11) = BWB_seg2(11)*1.0027420;
BWB_seg2(12) = BWB_seg2(12)*1.0027420;
BWB_seg2(13) = BWB_seg2(13)*1.0027420;
BWB_seg2(14) = BWB_seg2(14)*1.0027420;
BWB_seg2(15) = BWB_seg2(15)*1.0027420;
BWB_seg2(16) = BWB_seg2(16)*1.0027420;

% Bottom row error, ADCB, seg3
BWB_seg3(1) = BWB_seg3(1)*1.0027421;
BWB_seg3(2) = BWB_seg3(2)*1.0027421;
BWB_seg3(3) = BWB_seg3(3)*1.0027421;
BWB_seg3(4) = BWB_seg3(4)*1.0027421;
BWB_seg3(5) = BWB_seg3(5)*1.0027421;
BWB_seg3(6) = BWB_seg3(6)*1.0027421;
BWB_seg3(7) = BWB_seg3(7)*1.0027421;
BWB_seg3(8) = BWB_seg3(8)*1.0027421;
BWB_seg3(9) = BWB_seg3(9)*1.0027421;
BWB_seg3(10) = BWB_seg3(10)*1.0027421;
BWB_seg3(11) = BWB_seg3(11)*1.0027421;
BWB_seg3(12) = BWB_seg3(12)*1.0027421;
BWB_seg3(13) = BWB_seg3(13)*1.0027421;
BWB_seg3(14) = BWB_seg3(14)*1.0027421;
BWB_seg3(15) = BWB_seg3(15)*1.0027421;
BWB_seg3(16) = BWB_seg3(16)*1.0027421;
%
% Bottom row error, ADCB, seg4
BWB_seg4(1) = BWB_seg4(1)*1.0027422;
BWB_seg4(2) = BWB_seg4(2)*1.0027422;
BWB_seg4(3) = BWB_seg4(3)*1.0027422;
BWB_seg4(4) = BWB_seg4(4)*1.0027422;
BWB_seg4(5) = BWB_seg4(5)*1.0027422;
BWB_seg4(6) = BWB_seg4(6)*1.0027422;
BWB_seg4(7) = BWB_seg4(7)*1.0027422;
BWB_seg4(8) = BWB_seg4(8)*1.0027422;
BWB_seg4(9) = BWB_seg4(9)*1.0027422;
BWB_seg4(10) = BWB_seg4(10)*1.0027422;
BWB_seg4(11) = BWB_seg4(11)*1.0027422;
BWB_seg4(12) = BWB_seg4(12)*1.0027422;
BWB_seg4(13) = BWB_seg4(13)*1.0027422;
BWB_seg4(14) = BWB_seg4(14)*1.0027422;
BWB_seg4(15) = BWB_seg4(15)*1.0027422;
BWB_seg4(16) = BWB_seg4(16)*1.0027422;

% Bottom row error, ADCB, seg5
BWB_seg5(1) = BWB_seg5(1)*1.0027420;
```

```
BWB_seg5(2) = BWB_seg5(2)*1.0027420;
BWB_seg5(3) = BWB_seg5(3)*1.0027420;
BWB_seg5(4) = BWB_seg5(4)*1.0027420;
BWB_seg5(5) = BWB_seg5(5)*1.0027420;
BWB_seg5(6) = BWB_seg5(6)*1.0027420;
BWB_seg5(7) = BWB_seg5(7)*1.0027420;
BWB_seg5(8) = BWB_seg5(8)*1.0027420;
BWB_seg5(9) = BWB_seg5(9)*1.0027420;
BWB_seg5(10) = BWB_seg5(10)*1.0027420;
BWB_seg5(11) = BWB_seg5(11)*1.0027420;
BWB_seg5(12) = BWB_seg5(12)*1.0027420;
BWB_seg5(13) = BWB_seg5(13)*1.0027420;
BWB_seg5(14) = BWB_seg5(14)*1.0027420;
BWB_seg5(15) = BWB_seg5(15)*1.0027420;
BWB_seg5(16) = BWB_seg5(16)*1.0027420;
```

Error correction algorithm

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%Error Correction Algorithm

% % Initialization
% dec_A = zeros(128,50);
% dec_B = zeros(128,50);
% err = zeros(100,1);


%%% Redistribute decisions from ADCA and ADCB for error correction matrix
for i = 1:128
     t1 = sum(new_dA(i,49:64));
     t2 = sum(new_dA(i,65:80));
     dec_A(i,:) = [new_dA(i,1:48) t1 t2];

     t3 = sum(new_dB(i,49:64));
     t4 = sum(new_dB(i,65:80));
     dec_B(i,:) = [new_dB(i,1:48) t3 t4];

 end

 % Setting up parameters
 mu = 2^13;
 dec = [dec_B -dec_A];
 del_x = dec_B*WBside - dec_A*WAside;

 %%%%% Store the 128 set of decisions and del_x into temp
 temp_dec = dec;
 temp_del_x = del_x;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%Core of error correction algorithm
%%%% It is essentially doing what Book 6 p1 is doing
```

```
%% Small LMS loop for estimating error for each 128 conversions
for j = 1:100
    for i = 1:128
       if dec(i,j) < 0
           temp_dec(i,:) = -temp_dec(i,:);
           temp_del_x(i) = -temp_del_x(i);
       end

       if dec(i,j) == 0
           temp_del_x(i) = 0;
       end

    end

    err(j,1) = sum(temp_del_x)/mu;
    temp_dec = dec;
    temp_del_x = del_x;

end

% Update WB, WA and as a result, delta_x
WBside = WBside - err(1:50);
WAside = WAside - err(51:100);

% Storing and see how each of the 100 weight error evolve
% over the x set of 128 conversions
my_err(1:100,x) = err;
my_weight(1:100,x) = [WBside; WAside];


%%%%%%%%%%%%%End of Error Correction Algorithm
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Appendix D

Matlab Code used for testing performance of the error correction algorithm

INL/DNL improvement

Interface between mixed signal IC and FPGA control

```
clear all;
close all;
clc;

%%%%%%%%%%Overall function%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% This code do several things
%1) Set up the ADC parameters
%2.) Plot ADC error before correction
%3.) Apply error correction algorithm
%4.) Plot ADC error after correction

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%ADC setup%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% N denotes the number of bits of the ADC
N = 20;

 % These are weights got from the perfect 16 bit ADC step size (Book5 pg 54)
W1 = 0.145837866667;
W2 = W1/8;
W3 = W2/8;
W4 = W3/8;
W5 = W4/8;

% This denotes the initial weight used for both ADC_A and ADC_B in the
% digital to analog interface. The decisions from ADC_A will by muliplied
% by the estimated weight WA to get the analog voltage. The same happen for
% ADC_B. We assume we DO NOT KNOW of any error in the ADC itself, so we can
% just estimate them

WA = [W1*ones(16,1);
      W2*ones(16,1);
      W3*ones(16,1);
      W4*ones(16,1);
      W5*ones(16,1)];

WB = [W1*ones(16,1);
      W2*ones(16,1);
      W3*ones(16,1);
      W4*ones(16,1);
      W5*ones(16,1)];


% Initialization for before/after plot
INL = zeros(1,2^16);
DNL = zeros(1,2^16);
dec_A_ramp = zeros(65536*16,50);
```

```
dec_B_ramp = zeros(65536*16,50);
INL_after = zeros(1,2^16);
DNL_after = zeros(1,2^16);

Anum1 = zeros(1,16);
Anum2 = zeros(1,16);
Anum3 = zeros(1,16);
Anum4 = zeros(1,16);
Anum5 = zeros(1,16);

Bnum1 = zeros(1,16);
Bnum2 = zeros(1,16);
Bnum3 = zeros(1,16);
Bnum4 = zeros(1,16);
Bnum5 = zeros(1,16);

% Initialization
dec_A = zeros(128,50);
dec_B = zeros(128,50);
err = zeros(100,1);

resolution = 2.5/(2^16*16);
%VLSB = 2*resolution;

step = 2.5/2^16;


% Define Vin vector. A ramp input is used in this case.
Vin2 = [0: resolution: 2.5-resolution];
Vin1 = [2.5-resolution: -resolution: 0];

 Vsample2 = [0:step:2.5];
 Vsample1 = [2.5:-step:0];



%%%%%%%%%%%%%%%%ADC setup%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%INL/DNL plot before error correction

%Add some noise
for i = 1: length(Vin1)
    Vin1(i) = Vin1(i)+ randn*(30*10^-6);
    Vin2(i) = Vin2(i)+ randn*(30*10^-6);
end

% Convert them
 R013108_ADC_16bit_randomization;
  Vin_ramp1 = Vin1;
  Vin_ramp2 = Vin2;
```

```
  new_dA_ramp = new_dA;
  new_dB_ramp = new_dB;
  Vin_bar = (new_dB_ramp*WB + new_dA_ramp*WA)/2;

  figure(1)
  plot((Vin2-Vin1)', Vin_bar, 'r');
  hold on;
  fita = polyfit((Vin2-Vin1)', Vin_bar, 1);
  Vin_bar_mod = (Vin_bar-fita(2))*1/fita(1);
  plot((Vin2-Vin1)', Vin_bar_mod, 'g');
  axis([-2.5 2.5 -2.5 2.5])

  count = histc((Vin_bar_mod)', ([-5 Vsample2(2:65536)-Vsample1(2:65536)
5]));
  DNL = (count(1:65536)-16)/16;
  INL = cumsum(DNL);

  Code = 0:1:65535;

 figure(2)
 subplot(2,1,1)
 plot(Code, [0, INL(2:65535),0]);
 title('Before Correction');
 xlabel('Code');
 ylabel('INL(in LSB)');

 subplot(2,1,2)
 plot(Code, [0, DNL(2:65535), 0]);
 title('Before Correction');
 xlabel('Code');
 ylabel('DNL(in LSB)');



%%%%%%%%%%Plot ADC error before correction
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%Error correction

% Supply a sine input
resolution = 0:1e-3:1280;
Input2 = 1.25*sin(resolution)+1.25;
Input1 = -1.25*sin(resolution)+1.25;

% Add some noise
for i = 1: length(Input2)
    Input1(i) = Input1(i)+ randn*(30*10^-6);
    Input2(i) = Input2(i)+ randn*(30*10^-6);
end

figure(3)
```

```
plot(resolution(1:10000),Input2(1:10000), resolution(1:10000),
Input1(1:10000));

% Rearranging weight
%group cap weight in segment 4 as one
%group cap weight in segment 5 as one
q1 = sum(WA(49:64,1));
q2 = sum(WA(65:80,1));
q3 = sum(WB(49:64,1));
q4 = sum(WB(65:80,1));

% WAside and WBside serve as the updated weight. For each 128 conversions,
% They will be updated

WAside = [WA(1:48); q1; q2];
WBside = [WB(1:48); q3; q4];

for x = 1: 10000

     % This is for sine input only. Using different parts of the sine
     % signal for error correction
     Vin2 = Input2(1,128*(x-1)+1: 128*x);
     Vin1 = Input1(1,128*(x-1)+1: 128*x);

    % Convert using ADC code
    R013108_ADC_16bit_randomization;
    % Apply error correction
    R013108_error_correction;
end

%%%%%%%%%%%%%%%%Error correction
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%Plot ADC error after error correction

% Rearranging decision A and decision B so that they can be easily
% multipled by the "corrected" weights
for i = 1:length(Vin_ramp1)
    t1 = sum(new_dA_ramp(i,49:64));
    t2 = sum(new_dA_ramp(i,65:80));
    dec_A_ramp(i,:) = [new_dA_ramp(i,1:48) t1 t2];

    t3 = sum(new_dB_ramp(i,49:64));
    t4 = sum(new_dB_ramp(i,65:80));
    dec_B_ramp(i,:) = [new_dB_ramp(i,1:48) t3 t4];
end

% Getting the analog output code using the "corrected" weights
Vin_bar_after = (dec_B_ramp*WBside + dec_A_ramp*WAside)/2;
```

```
resolution = 2.5/(2^16*16);
Vin_ideal2 = [0: resolution: 2.5];
Vin_ideal1 = [2.5: -resolution: 0];

Vin_ideal2 = Vin_ideal2(1:2^16*16);
Vin_ideal1 = Vin_ideal1(1:2^16*16);

figure(4)
plot((Vin_ideal2-Vin_ideal1)', Vin_bar_after, 'r');
hold on;
fita = polyfit((Vin_ideal2-Vin_ideal1)', Vin_bar_after, 1);
Vin_bar_after_mod = (Vin_bar_after-fita(2))*1/fita(1);
plot((Vin_ideal2-Vin_ideal1)', Vin_bar_after_mod, 'g');
axis([-2.5 2.5 -2.5 2.5])

count = histc((Vin_bar_after_mod)', ([-5 Vsample2(2:65536)-Vsample1(2:65536)
5]));
DNL_after = (count(1:65536)-16)/16;
INL_after = cumsum(DNL_after);

Code = 0:1:65535;

figure(5)
subplot(2,1,1)
plot(INL_after);
title('After Correction');
xlabel('Code');
ylabel('INL(in LSB)');
axis([0 65535 -1 1]);


subplot(2,1,2)
plot(DNL_after);
title('After Correction');
xlabel('Code');
ylabel('DNL(in LSB)');
axis([0 65535 -1 1]);
```

Modified SAR ADC

```
%%%%%%%%%%%%%%%%%%%%%%%%Overall function%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%This code composes of two ADCs. Two ADCs each convert the same input
%into a bunch of decisions, according to their set weights. In this example,
%ADCA's weight is perfect, and zero biased errors are added to first segment
%of ADCB(bottom row).

% Two process contribute to the difference in decisions
%1.) Randomization using rand
%2.) error in ADC_B
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%ADC_A

% This is the NON zero-biased cap mismatch, Cc error set
Ctotal_A = 18.2756233; % pF
```

```
weight_factor = 1.25*16/15;

W1_mod = 1/Ctotal_A*weight_factor;
W2_mod = 0.125/Ctotal_A*weight_factor;
W3_mod = W2_mod/7.9837;
W4_mod = W3_mod/8.025;
W5_mod = W4_mod/7.980566;

% %Ideal cap weight, top row(C1-C80)
% AWT_seg1 = ones(1,16)*W1*1/2;
% AWT_seg2 = ones(1,16)*W2*1/2;
% AWT_seg3 = ones(1,16)*W3*1/2;
% AWT_seg4 = ones(1,16)*W4*1/2;
% AWT_seg5 = ones(1,16)*W5*1/2;
%
% %Ideal cap weight, bottom row(C1-C80)
% AWB_seg1 = ones(1,16)*W1*1/2;
% AWB_seg2 = ones(1,16)*W2*1/2;
% AWB_seg3 = ones(1,16)*W3*1/2;
% AWB_seg4 = ones(1,16)*W4*1/2;
% AWB_seg5 = ones(1,16)*W5*1/2;

%Ideal cap weight, top row(C1-C80)
AWT_seg1 = ones(1,16)*W1_mod;
AWT_seg2 = ones(1,16)*W2_mod;
AWT_seg3 = ones(1,16)*W3_mod;
AWT_seg4 = ones(1,16)*W4_mod;
AWT_seg5 = ones(1,16)*W5_mod;

%Ideal cap weight, bottom row(C1-C80)
AWB_seg1 = ones(1,16)*W1_mod;
AWB_seg2 = ones(1,16)*W2_mod;
AWB_seg3 = ones(1,16)*W3_mod;
AWB_seg4 = ones(1,16)*W4_mod;
AWB_seg5 = ones(1,16)*W5_mod;

% Add some error to the bottom row cap in ADC_A, to segment 1,2,3,4,5
% Just like Book 6 page 36, 37 (zero basied error)

R013108_MismatchA_non_zero;

%Sample Voltage points
S = length(Vin1);

% Initialize Vx and Vy, and vcomp
AVx = zeros(N+1,S);
AVy = zeros(N+1,S);
AVcomp = zeros(N+1,S);
Astore_draw = zeros(S,5);
% Anum1 = zeros(1,16);
% Anum2 = zeros(1,16);
% Anum3 = zeros(1,16);
% Anum4 = zeros(1,16);
% Anum5 = zeros(1,16);
```

```
% decision_matrix
dA = zeros(N,length(Vin1));

% reconstructed decision matrix
new_dA = zeros(length(Vin2),80);

% linear fit got from plotting Vx, Vy against Vin in the ideal case
% Assume it does not change much!?

fity = [-0.468749927975927   1.249999974180060];
fitx = [0.468749974761574   1.250000101936904];

% Sample + Hold Mode
% Vy is the bottom row, Vx is the top row
% The linear fit helps to get the first Vx, Vy point. Afterwards, it's all
% up to the weight.
AVx(1,:) = fitx(1)*(Vin2-Vin1)+fitx(2);
AVy(1,:) = fity(1)*(Vin2-Vin1)+fity(2);
AVcomp(1,:) = AVy(1,:)-AVx(1,:);

% Bit cycle mode
  for j = 1:S

      % Initiating randomization. If num 1 = 8, then num2 = 9. the number
goes in
    % a wheel fashion

    Adraw1 = floor(16*rand)+1;
      for q = 1:16
         Anum1(q) = mod(Adraw1+(q-1),16);
         if Anum1(q) == 0
             Anum1(q) = 16;
         end;
      end;

   Adraw2 = floor(16*rand)+1;
      for q = 1:16
         Anum2(q) = mod(Adraw2+(q-1),16);
         if Anum2(q) == 0
             Anum2(q) = 16;
         end;
      end;

   Adraw3 = floor(16*rand)+1;
      for q = 1:16
         Anum3(q) = mod(Adraw3+(q-1),16);
         if Anum3(q) == 0
             Anum3(q) = 16;
         end;
      end;

   Adraw4 = floor(16*rand)+1;
      for q = 1:16
         Anum4(q) = mod(Adraw4+(q-1),16);
         if Anum4(q) == 0
```

```
            Anum4(q) = 16;
        end;
    end;

Adraw5 = floor(16*rand)+1;
    for q = 1:16
        Anum5(q) = mod(Adraw5+(q-1),16);
        if Anum5(q) == 0
            Anum5(q) = 16;
        end;
    end;

 Astore_draw(j,1:5) = [Adraw1 Adraw2 Adraw3 Adraw4 Adraw5];


 Ab1 = find(Anum1 >=1 & Anum1 <= 8);
 Ab2 = find(Anum1 >=9 & Anum1 <=12);
 Ab3 = find(Anum1>=13 & Anum1 <= 14);
 Ab4a = find(Anum1 == 15);
 Aunused1 = find(Anum1 ==16);

 Ab4b = find(Anum2 >=1 & Anum2 <= 8);
 Ab5 = find(Anum2 >=9 & Anum2 <=12);
 Ab6 = find(Anum2>=13 & Anum2 <= 14);
 Ab7a = find(Anum2 == 15);
 Aunused2 = find(Anum2 ==16);

 Ab7b = find(Anum3 >=1 & Anum3 <= 8);
 Ab8 = find(Anum3 >=9 & Anum3 <=12);
 Ab9 = find(Anum3>=13 & Anum3 <= 14);
 Ab10a = find(Anum3 == 15);
 Aunused3 = find(Anum3 ==16);

 Ab10b = find(Anum4 >=1 & Anum4 <= 8);
 Ab11 = find(Anum4 >=9 & Anum4 <=12);
 Ab12 = find(Anum4>=13 & Anum4 <= 14);
 Ab13a = find(Anum4 == 15);
 Aunused4 = find(Anum4 ==16);

 Ab13b = find(Anum5 >=1 & Anum5 <= 8);
 Ab14 = find(Anum5 >=9 & Anum5 <=12);
 Ab15 = find(Anum5>=13 & Anum5 <= 14);
 Ab16 = find(Anum5 == 15);
 Aunused5 = find(Anum5 ==16);

 % Sum up the weight used for each bit(Top row)
 AWT(1) = sum(AWT_seg1(Ab1));
 AWT(2) = sum(AWT_seg1(Ab2));
 AWT(3) = sum(AWT_seg1(Ab3));
 AWT(4) = sum(AWT_seg1(Ab4a));

 AWT(5) = sum(AWT_seg2(Ab4b));
 AWT(6) = sum(AWT_seg2(Ab5));
 AWT(7) = sum(AWT_seg2(Ab6));
 AWT(8) = sum(AWT_seg2(Ab7a));
```

```
AWT(9)  = sum(AWT_seg3(Ab7b));
AWT(10) = sum(AWT_seg3(Ab8));
AWT(11) = sum(AWT_seg3(Ab9));
AWT(12) = sum(AWT_seg3(Ab10a));

AWT(13)  = sum(AWT_seg4(Ab10b));
AWT(14) = sum(AWT_seg4(Ab11));
AWT(15) = sum(AWT_seg4(Ab12));
AWT(16) = sum(AWT_seg4(Ab13a));

AWT(17)  = sum(AWT_seg5(Ab13b));
AWT(18) = sum(AWT_seg5(Ab14));
AWT(19) = sum(AWT_seg5(Ab15));
AWT(20) = sum(AWT_seg5(Ab16));

% Sum up the weight used for each bit(Bottom row)
AWB(1) = sum(AWB_seg1(Ab1));
AWB(2) = sum(AWB_seg1(Ab2));
AWB(3) = sum(AWB_seg1(Ab3));
AWB(4) = sum(AWB_seg1(Ab4a));

AWB(5) = sum(AWB_seg2(Ab4b));
AWB(6) = sum(AWB_seg2(Ab5));
AWB(7) = sum(AWB_seg2(Ab6));
AWB(8) = sum(AWB_seg2(Ab7a));

AWB(9)  = sum(AWB_seg3(Ab7b));
AWB(10) = sum(AWB_seg3(Ab8));
AWB(11) = sum(AWB_seg3(Ab9));
AWB(12) = sum(AWB_seg3(Ab10a));

AWB(13)  = sum(AWB_seg4(Ab10b));
AWB(14) = sum(AWB_seg4(Ab11));
AWB(15) = sum(AWB_seg4(Ab12));
AWB(16) = sum(AWB_seg4(Ab13a));

AWB(17)  = sum(AWB_seg5(Ab13b));
AWB(18) = sum(AWB_seg5(Ab14));
AWB(19) = sum(AWB_seg5(Ab15));
AWB(20) = sum(AWB_seg5(Ab16));

for n = 1:N

    dA(n,j) = -sign(AVcomp(n,j)-0+ randn*(30*10^-6));

 % Only d= 1 and d=-1 is allowed. If d = 0, this indicates Vin = Vdac,
 % so a decision bit 1 should be assigned.

    if AVy(n,j)== AVx(n,j)
       dA(n,j) = 1;
    end;

   AVy(n+1,j) = AVy(n,j) + dA(n,j)*AWB(n);
```

```
       AVx(n+1,j) = AVx(n,j) - dA(n,j)*AWT(n);
       AVcomp(n+1,j) = AVy(n+1,j) - AVx(n+1,j);


      new_dA(j,Ab1) = dA(1,j);
      new_dA(j,Ab2) = dA(2,j);
      new_dA(j,Ab3) = dA(3,j);
      new_dA(j,Ab4a) = dA(4,j);

      new_dA(j,16+Ab4b) = dA(5,j);
      new_dA(j,16+Ab5) = dA(6,j);
      new_dA(j,16+Ab6) = dA(7,j);
      new_dA(j,16+Ab7a) = dA(8,j);

      new_dA(j,32+Ab7b) = dA(9,j);
      new_dA(j,32+Ab8) = dA(10,j);
      new_dA(j,32+Ab9) = dA(11,j);
      new_dA(j,32+Ab10a) = dA(12,j);

      new_dA(j,48+Ab10b) = dA(13,j);
      new_dA(j,48+Ab11) = dA(14,j);
      new_dA(j,48+Ab12) = dA(15,j);
      new_dA(j,48+Ab13a) = dA(16,j);

      new_dA(j,64+Ab13b) = dA(17,j);
      new_dA(j,64+Ab14) = dA(18,j);
      new_dA(j,64+Ab15) = dA(19,j);
      new_dA(j,64+Ab16) = dA(20,j);

     end; %(for n loop)


 end; % (for j loop)
%
%%%%%%%%%%%%%%%%%%%%%ADC_A%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%ADC_B

%Ideal cap weight, top row(C1-C80)
BWT_seg1 = ones(1,16)*1/2*W1;
BWT_seg2 = ones(1,16)*1/2*W2;
BWT_seg3 = ones(1,16)*1/2*W3;
BWT_seg4 = ones(1,16)*1/2*W4;
BWT_seg5 = ones(1,16)*1/2*W5;

%Ideal cap weight, bottom row(C1-C80)
BWB_seg1 = ones(1,16)*1/2*W1;
BWB_seg2 = ones(1,16)*1/2*W2;
BWB_seg3 = ones(1,16)*1/2*W3;
BWB_seg4 = ones(1,16)*1/2*W4;
BWB_seg5 = ones(1,16)*1/2*W5;
```

```
% Add some error to the bottom row cap in ADC_B, to segment 1,2,3,4,5
% Just like Book 6 page 36, 37 (zero basied error)

R013108_MismatchB;

%Sample Voltage points
S = length(Vin1);

% Initialize Vx and Vy, and vcomp
BVx = zeros(N+1,S);
BVy = zeros(N+1,S);

BVcomp = zeros(N+1,S);
Bstore_draw = zeros(S,5);
% Bnum1 = zeros(1,16);
% Bnum2 = zeros(1,16);
% Bnum3 = zeros(1,16);
% Bnum4 = zeros(1,16);
% Bnum5 = zeros(1,16);
%
% decision_matrix
dB = zeros(N,length(Vin1));

% reconstructed decision matrix
new_dB = zeros(length(Vin2),80);

% linear fit got from plotting Vx, Vy against Vin in the ideal case
% Assume it does not change much!?

fity = [-0.468749927975927   1.249999974180060];
fitx = [0.468749974761574   1.250000101936904];

% Sample + Hold Mode
% Vy is the bottom row, Vx is the top row
% The linear fit helps to get the first Vx, Vy point. Afterwards, it's all
% up to the weight.
BVx(1,:) = fitx(1)*(Vin2-Vin1)+fitx(2);
BVy(1,:) = fity(1)*(Vin2-Vin1)+fity(2);
BVcomp(1,:) = BVy(1,:)-BVx(1,:);

% Bit cycle mode
  for j = 1:S

      % Initiating randomization. If num 1 = 8, then num2 = 9. the number
goes in
    % a wheel fashion

    Bdraw1 = floor(16*rand)+1;
      for q = 1:16
         Bnum1(q) = mod(Bdraw1+(q-1),16);
         if Bnum1(q) == 0
             Bnum1(q) = 16;
         end;
```

```
   end;

Bdraw2 = floor(16*rand)+1;
   for q = 1:16
      Bnum2(q) = mod(Bdraw2+(q-1),16);
      if Bnum2(q) == 0
          Bnum2(q) = 16;
      end;
   end;

Bdraw3 = floor(16*rand)+1;
   for q = 1:16
      Bnum3(q) = mod(Bdraw3+(q-1),16);
      if Bnum3(q) == 0
          Bnum3(q) = 16;
      end;
   end;

Bdraw4 = floor(16*rand)+1;
   for q = 1:16
      Bnum4(q) = mod(Bdraw4+(q-1),16);
      if Bnum4(q) == 0
          Bnum4(q) = 16;
      end;
   end;

Bdraw5 = floor(16*rand)+1;
   for q = 1:16
      Bnum5(q) = mod(Bdraw5+(q-1),16);
      if Bnum5(q) == 0
          Bnum5(q) = 16;
      end;
   end;

 Bstore_draw(j,1:5) = [Bdraw1 Bdraw2 Bdraw3 Bdraw4 Bdraw5];

 Bb1 = find(Bnum1 >=1 & Bnum1 <= 8);
 Bb2 = find(Bnum1 >=9 & Bnum1 <=12);
 Bb3 = find(Bnum1>=13 & Bnum1 <= 14);
 Bb4a = find(Bnum1 == 15);
 Bunused1 = find(Bnum1 ==16);

 Bb4b = find(Bnum2 >=1 & Bnum2 <= 8);
 Bb5 = find(Bnum2 >=9 & Bnum2 <=12);
 Bb6 = find(Bnum2>=13 & Bnum2 <= 14);
 Bb7a = find(Bnum2 == 15);
 Bunused2 = find(Bnum2 ==16);

 Bb7b = find(Bnum3 >=1 & Bnum3 <= 8);
 Bb8 = find(Bnum3 >=9 & Bnum3 <=12);
 Bb9 = find(Bnum3>=13 & Bnum3 <= 14);
 Bb10a = find(Bnum3 == 15);
 Bunused3 = find(Bnum3 ==16);

 Bb10b = find(Bnum4 >=1 & Bnum4 <= 8);
```

```
Bb11 = find(Bnum4 >=9 & Bnum4 <=12);
Bb12 = find(Bnum4>=13 & Bnum4 <= 14);
Bb13a = find(Bnum4 == 15);
Bunused4 = find(Bnum4 ==16);

Bb13b = find(Bnum5 >=1 & Bnum5 <= 8);
Bb14 = find(Bnum5 >=9 & Bnum5 <=12);
Bb15 = find(Bnum5>=13 & Bnum5 <= 14);
Bb16 = find(Bnum5 == 15);
Bunused5 = find(Bnum5 ==16);

% sum up the weights used for each bit(Top row)
BWT(1) = sum(BWT_seg1(Bb1));
BWT(2) = sum(BWT_seg1(Bb2));
BWT(3) = sum(BWT_seg1(Bb3));
BWT(4) = sum(BWT_seg1(Bb4a));

BWT(5) = sum(BWT_seg2(Bb4b));
BWT(6) = sum(BWT_seg2(Bb5));
BWT(7) = sum(BWT_seg2(Bb6));
BWT(8) = sum(BWT_seg2(Bb7a));

BWT(9)  = sum(BWT_seg3(Bb7b));
BWT(10) = sum(BWT_seg3(Bb8));
BWT(11) = sum(BWT_seg3(Bb9));
BWT(12) = sum(BWT_seg3(Bb10a));

BWT(13)  = sum(BWT_seg4(Bb10b));
BWT(14) = sum(BWT_seg4(Bb11));
BWT(15) = sum(BWT_seg4(Bb12));
BWT(16) = sum(BWT_seg4(Bb13a));

BWT(17)  = sum(BWT_seg5(Bb13b));
BWT(18) = sum(BWT_seg5(Bb14));
BWT(19) = sum(BWT_seg5(Bb15));
BWT(20) = sum(BWT_seg5(Bb16));

% sum up the weight used for each bit(Bottom row)
BWB(1) = sum(BWB_seg1(Bb1));
BWB(2) = sum(BWB_seg1(Bb2));
BWB(3) = sum(BWB_seg1(Bb3));
BWB(4) = sum(BWB_seg1(Bb4a));

BWB(5) = sum(BWB_seg2(Bb4b));
BWB(6) = sum(BWB_seg2(Bb5));
BWB(7) = sum(BWB_seg2(Bb6));
BWB(8) = sum(BWB_seg2(Bb7a));

BWB(9)  = sum(BWB_seg3(Bb7b));
BWB(10) = sum(BWB_seg3(Bb8));
BWB(11) = sum(BWB_seg3(Bb9));
BWB(12) = sum(BWB_seg3(Bb10a));

BWB(13)  = sum(BWB_seg4(Bb10b));
BWB(14) = sum(BWB_seg4(Bb11));
```

```
    BWB(15) = sum(BWB_seg4(Bb12));
    BWB(16) = sum(BWB_seg4(Bb13a));

    BWB(17)  = sum(BWB_seg5(Bb13b));
    BWB(18) = sum(BWB_seg5(Bb14));
    BWB(19) = sum(BWB_seg5(Bb15));
    BWB(20) = sum(BWB_seg5(Bb16));

    for n = 1:N

        dB(n,j) = -sign(BVcomp(n,j)-0 + randn*(30*10^-6));

     % Only d= 1 and d=-1 is allowed. If d = 0, this indicates Vin = Vdac,
     % so a decision bit 1 should be assigned.

        if BVy(n,j)== BVx(n,j)
           dB(n,j) = 1;
        end;

       BVy(n+1,j) = BVy(n,j) + dB(n,j)*BWB(n);
       BVx(n+1,j) = BVx(n,j) - dB(n,j)*BWT(n);
       BVcomp(n+1,j) = BVy(n+1,j) - BVx(n+1,j);


     new_dB(j,Bb1) = dB(1,j);
     new_dB(j,Bb2) = dB(2,j);
     new_dB(j,Bb3) = dB(3,j);
     new_dB(j,Bb4a) = dB(4,j);

     new_dB(j,16+Bb4b) = dB(5,j);
     new_dB(j,16+Bb5) = dB(6,j);
     new_dB(j,16+Bb6) = dB(7,j);
     new_dB(j,16+Bb7a) = dB(8,j);

     new_dB(j,32+Bb7b) = dB(9,j);
     new_dB(j,32+Bb8) = dB(10,j);
     new_dB(j,32+Bb9) = dB(11,j);
     new_dB(j,32+Bb10a) = dB(12,j);

     new_dB(j,48+Bb10b) = dB(13,j);
     new_dB(j,48+Bb11) = dB(14,j);
     new_dB(j,48+Bb12) = dB(15,j);
     new_dB(j,48+Bb13a) = dB(16,j);

     new_dB(j,64+Bb13b) = dB(17,j);
     new_dB(j,64+Bb14) = dB(18,j);
     new_dB(j,64+Bb15) = dB(19,j);
     new_dB(j,64+Bb16) = dB(20,j);

    end; %(for n loop)


end; % (for j loop)
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%ADC_B%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Error correction algorithm

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%Error Correction Algorithm

% % Initialization
% dec_A = zeros(128,50);
% dec_B = zeros(128,50);
% err = zeros(100,1);


%%% Redistribute decisions from ADCA and ADCB for error correction matrix
for i = 1:128
     t1 = sum(new_dA(i,49:64));
     t2 = sum(new_dA(i,65:80));
     dec_A(i,:) = [new_dA(i,1:48) t1 t2];

     t3 = sum(new_dB(i,49:64));
     t4 = sum(new_dB(i,65:80));
     dec_B(i,:) = [new_dB(i,1:48) t3 t4];

 end

 % Setting up parameters
 mu = 2^13;
 dec = [dec_B -dec_A];
 del_x = dec_B*WBside - dec_A*WAside;

 %%%%% Store the 128 set of decisions and del_x into temp
 temp_dec = dec;
 temp_del_x = del_x;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%Core of error correction algorithm
%%%% It is essentially doing what Book 6 p1 is doing

%% Small LMS loop for estimating error for each 128 conversions
for j = 1:100
    for i = 1:128
       if dec(i,j) < 0
           temp_dec(i,:) = -temp_dec(i,:);
           temp_del_x(i) = -temp_del_x(i);
       end

       if dec(i,j) == 0
           temp_del_x(i) = 0;
       end
```

```
    end

    err(j,1) = sum(temp_del_x)/mu;
    temp_dec = dec;
    temp_del_x = del_x;

end

% Update WB, WA and as a result, delta_x
WBside = WBside - err(1:50);
WAside = WAside - err(51:100);

% Storing and see how each of the 100 weight error evolve
% over the x set of 128 conversions
my_err(1:100,x) = err;
my_weight(1:100,x) = [WBside; WAside];


%%%%%%%%%%%%%%End of Error Correction Algorithm
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Adaptation for various input signal

Interface between mixed signal IC and FPGA control

```
clear all;
close all;
clc;

%%%%%%%%%Overall function%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% This code do several things
%1) Set up the ADC parameters
%2.) Plot ADC error before correction
%3.) Apply error correction algorithm
%4.) Plot ADC error after correction

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%ADC setup%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%N denotes the number of bits of the ADC
N = 20;

%These are weights got from the perfect 16 bit ADC step size (Book5 pg 54)
W1 = 0.145837866667;
W2 = W1/8;
W3 = W2/8;
W4 = W3/8;
W5 = W4/8;

%conv denotes set of conversion
conv = 10000;

%This denotes the initial weight used for both ADC_A and ADC_B in the
%digital to analog interface. The decisions from ADC_A will by muliplied
%by the estimated weight WA to get the analog voltage. The same happen for
%ADC_B. We assume we DO NOT KNOW of any error in the ADC itself, so we can
%just estimate them

WA = [W1*ones(16,1);
      W2*ones(16,1);
      W3*ones(16,1);
      W4*ones(16,1);
      W5*ones(16,1)];

WB = [W1*ones(16,1);
      W2*ones(16,1);
      W3*ones(16,1);
      W4*ones(16,1);
      W5*ones(16,1)];

Anum1 = zeros(1,16);
Anum2 = zeros(1,16);
Anum3 = zeros(1,16);
Anum4 = zeros(1,16);
Anum5 = zeros(1,16);
```

```
Bnum1 = zeros(1,16);
Bnum2 = zeros(1,16);
Bnum3 = zeros(1,16);
Bnum4 = zeros(1,16);
Bnum5 = zeros(1,16);

%Initialization
dec_A = zeros(128,50);
dec_B = zeros(128,50);
dec = zeros(128,100);
del_x = zeros(128,1);

err = zeros(100,1);
my_err = zeros(100,conv);
ADC_error = zeros(128,conv);

mu = 2^13;

%%%%%%%%%%%%%%ADC setup%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%Error correction

%%%%%%%%%%%%Sine Input
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%
%Supply a sine input
resolution = 0:1e-3:1280;
Input2 = 1.25*sin(resolution)+1.25;
Input1 = -1.25*sin(resolution)+1.25;

%Add some noise
for i = 1: length(Input2)
    Input1(i) = Input1(i)+ randn*(30*10^-6);
    Input2(i) = Input2(i)+ randn*(30*10^-6);
end

figure(1)
plot(resolution(1:10000),Input2(1:10000), resolution(1:10000),
Input1(1:10000));

%Rearranging weight
%group cap weight in segment 4 as one
%group cap weight in segment 5 as one
q1 = sum(WA(49:64,1));
q2 = sum(WA(65:80,1));
q3 = sum(WB(49:64,1));
q4 = sum(WB(65:80,1));

%WAside and WBside serve as the updated weight. For each 128 conversions,
%They will be updated

WAside = [WA(1:48); q1; q2];
```

```
WBside = [WB(1:48); q3; q4];

for x = 1: conv

     %This is for sine input only. Using different parts of the sine
     %signal for error correction
     Vin2 = Input2(1,128*(x-1)+1: 128*x);
     Vin1 = Input1(1,128*(x-1)+1: 128*x);

    %Convert using ADC code
    R013108_ADC_16bit_randomization;
    %Apply error correction
    R013108_error_correction;
end

%Storing and see how each of the 100 weight error evolve
%over the x set of 128 conversions
sine_err = my_err;
sine_error = ADC_error;



%%%%%%%%%%%Sine Input
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%DC Input(0.9 FS)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%
%Supply a DC input(0.9 FS)
Input2 = 2.375*ones(1,1280000);
Input1 = 0.125*ones(1,1280000);

%Add some noise
for i = 1: length(Input2)
    Input1(i) = Input1(i)+ randn*(30*10^-6);
    Input2(i) = Input2(i)+ randn*(30*10^-6);
end

figure(2)
plot(1:10000,Input2(1:10000), 1:10000, Input1(1:10000));

%Rearranging weight
%group cap weight in segment 4 as one
%group cap weight in segment 5 as one
q1 = sum(WA(49:64,1));
q2 = sum(WA(65:80,1));
q3 = sum(WB(49:64,1));
q4 = sum(WB(65:80,1));

%WAside and WBside serve as the updated weight. For each 128 conversions,
%They will be updated

WAside = [WA(1:48); q1; q2];
WBside = [WB(1:48); q3; q4];
```

```
for x = 1: conv

     %This is for sine input only. Using different parts of the sine
     %signal for error correction
     Vin2 = Input2(1,128*(x-1)+1: 128*x);
     Vin1 = Input1(1,128*(x-1)+1: 128*x);

    %Convert using ADC code
    R013108_ADC_16bit_randomization;
    %Apply error correction
    R013108_error_correction;
end

%Storing and see how each of the 100 weight error evolve
%over the x set of 128 conversions
DC09_err = my_err;
DC09_error = ADC_error;



%%%%%%%%%%%%DC Input (0.9 FS)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%DC Input(0.1 FS)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%
%Supply a DC input(0.1 FS)
Input2 = 1.375*ones(1,1280000);
Input1 = 1.125*ones(1,1280000);

%Add some noise
for i = 1: length(Input2)
    Input1(i) = Input1(i)+ randn*(30*10^-6);
    Input2(i) = Input2(i)+ randn*(30*10^-6);
end

figure(3)
plot(1:10000,Input2(1:10000), 1:10000, Input1(1:10000));

%Rearranging weight
%group cap weight in segment 4 as one
%group cap weight in segment 5 as one
q1 = sum(WA(49:64,1));
q2 = sum(WA(65:80,1));
q3 = sum(WB(49:64,1));
q4 = sum(WB(65:80,1));

%WAside and WBside serve as the updated weight. For each 128 conversions,
%They will be updated

WAside = [WA(1:48); q1; q2];
WBside = [WB(1:48); q3; q4];
```

```
for x = 1: conv

     %This is for sine input only. Using different parts of the sine
     %signal for error correction
     Vin2 = Input2(1,128*(x-1)+1: 128*x);
     Vin1 = Input1(1,128*(x-1)+1: 128*x);

    %Convert using ADC code
    R013108_ADC_16bit_randomization;
    %Apply error correction
    R013108_error_correction;
end

%Storing and see how each of the 100 weight error evolve
%over the x set of 128 conversions
DC01_err = my_err;
DC01_error = ADC_error;

%%%%%%%%%%%%%%%DC Input (0.1 FS)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%Random
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%
%Supply Random Input
Input2 = 2.4*rand(1,1280000);
Input1 = 2.4-2.4*rand(1,1280000);

%Add some noise
for i = 1: length(Input2)
    Input1(i) = Input1(i)+ randn*(30*10^-6);
    Input2(i) = Input2(i)+ randn*(30*10^-6);
end

figure(4)
plot(1:10000,Input2(1:10000), 1:10000, Input1(1:10000));

%Rearranging weight
%group cap weight in segment 4 as one
%group cap weight in segment 5 as one
q1 = sum(WA(49:64,1));
q2 = sum(WA(65:80,1));
q3 = sum(WB(49:64,1));
q4 = sum(WB(65:80,1));

%WAside and WBside serve as the updated weight. For each 128 conversions,
%They will be updated

WAside = [WA(1:48); q1; q2];
WBside = [WB(1:48); q3; q4];

for x = 1: conv
```

```
     %This is for sine input only. Using different parts of the sine
     %signal for error correction
     Vin2 = Input2(1,128*(x-1)+1: 128*x);
     Vin1 = Input1(1,128*(x-1)+1: 128*x);

    %Convert using ADC code
    R013108_ADC_16bit_randomization;
    %Apply error correction
    R013108_error_correction;
end

%Storing and see how each of the 100 weight error evolve
%over the x set of 128 conversions
random_err = my_err;
random_error = ADC_error;


%%%%%%%%%%%%%%Random
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%Error correction
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%Plot ADC error after error correction

%Plot how error1 evolves over x set of 128 conversion
%error1 in this case is the first cap weight error in ADC_B
conversion = 128*(1:x);
VLSB = 5/2^16;
sine_error = sine_error/VLSB;
DC09_error = DC09_error/VLSB;
DC01_error = DC01_error/VLSB;
random_error = random_error/VLSB;

figure(5)
semilogy(conversion,abs(sine_err(1,:)),conversion,abs(DC09_err(1,:)),conversi
on,abs(DC01_err(1,:)),conversion,abs(random_err(1,:)));
grid;
title('Evolution of weight error over time');
xlabel('Conversion Index');
ylabel('Error(V)');
legend('Sine Wave', 'DC(0.9 FS)','DC(0.1 FS)','Random');

figure(6)
semilogy(conversion, abs(sine_error(1,:)), conversion,
abs(DC09_error(1,:)),conversion, abs(DC01_error(1,:)),conversion,
abs(random_error(1,:)));
grid;
title('ADC error');
xlabel('Conversion Index');
ylabel('ADC error(LSB)');
```

```
legend('Sine Wave', 'DC(0.9 FS)', 'DC(0.1 FS)','Random');

% To calculate standard deviation
% std of weight error
sine_err_std = std(sine_err(1,8000:10000))
DC09_err_std = std(DC09_err(1,8000:10000))
DC01_err_std = std(DC01_err(1,8000:10000))
random_err_std = std(random_err(1,8000:10000))

% std of ADC error(LSB)
sine_error_std = std(sine_error(1,8000:10000))
DC09_error_std = std(DC09_error(1,8000:10000))
DC01_error_std = std(DC01_error(1,8000:10000))
random_error_std = std(random_error(1,8000:10000))


% %%%%%%%%%%%%%%%%Plot ADC error after error correction
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Modified SAR ADC

Same as before

Error Correction Algorithm

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%Error Correction Algorithm

% % Initialization
% dec_A = zeros(128,50);
% dec_B = zeros(128,50);
% err = zeros(100,1);


%%% Redistribute decisions from ADCA and ADCB for error correction matrix
for i = 1:128
     t1 = sum(new_dA(i,49:64));
     t2 = sum(new_dA(i,65:80));
     dec_A(i,:) = [new_dA(i,1:48) t1 t2];

     t3 = sum(new_dB(i,49:64));
     t4 = sum(new_dB(i,65:80));
     dec_B(i,:) = [new_dB(i,1:48) t3 t4];

 end

 % Setting up parameters

 dec = [dec_B -dec_A];
 del_x = dec_B*WBside - dec_A*WAside;
```

```
 %%%%% Store the 128 set of decisions and del_x into temp
 temp_dec = dec;
 temp_del_x = del_x;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%Core of error correction algorithm
%%%% It is essentially doing what Book 6 p1 is doing

%% Small LMS loop for estimating error for each 128 conversions
for j = 1:100
    for i = 1:128
       if dec(i,j) < 0
           temp_dec(i,:) = -temp_dec(i,:);
           temp_del_x(i) = -temp_del_x(i);
       end

       if dec(i,j) == 0
           temp_del_x(i) = 0;
       end

    end

    err(j,1) = sum(temp_del_x)/mu;
    temp_dec = dec;
    temp_del_x = del_x;

end

% Update WB, WA and as a result, delta_x
WBside = WBside - err(1:50);
WAside = WAside - err(51:100);

% Storing and see how each of the 100 weight error evolve
% over the x set of 128 conversions
my_err(1:100,x) = err;
%my_weight(1:100,x) = [WBside; WAside];
ADC_error(1:128,x) = (dec_B*err(1:50)+dec_A*err(51:100))/2;


%%%%%%%%%%%%%End of Error Correction Algorithm
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Effect of different mu

Interface between mixed signal IC and external FPGA control

```
clear all;
close all;
clc;

%%%%%%%%%%Overall function%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% This code do several things
%1) Set up the ADC parameters
%2.) Plot ADC error before correction
%3.) Apply error correction algorithm
%4.) Plot ADC error after correction

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%ADC setup%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% N denotes the number of bits of the ADC
N = 20;

 % These are weights got from the perfect 16 bit ADC step size (Book5 pg 54)
W1 = 0.145837866667;
W2 = W1/8;
W3 = W2/8;
W4 = W3/8;
W5 = W4/8;

% This denotes the initial weight used for both ADC_A and ADC_B in the
% digital to analog interface. The decisions from ADC_A will by muliplied
% by the estimated weight WA to get the analog voltage. The same happen for
% ADC_B. We assume we DO NOT KNOW of any error in the ADC itself, so we can
% just estimate them

WA = [W1*ones(16,1);
      W2*ones(16,1);
      W3*ones(16,1);
      W4*ones(16,1);
      W5*ones(16,1)];

WB = [W1*ones(16,1);
      W2*ones(16,1);
      W3*ones(16,1);
      W4*ones(16,1);
      W5*ones(16,1)];

Anum1 = zeros(1,16);
Anum2 = zeros(1,16);
Anum3 = zeros(1,16);
Anum4 = zeros(1,16);
Anum5 = zeros(1,16);

Bnum1 = zeros(1,16);
Bnum2 = zeros(1,16);
Bnum3 = zeros(1,16);
Bnum4 = zeros(1,16);
```

```
Bnum5 = zeros(1,16);

% Initialization
dec_A = zeros(128,50);
dec_B = zeros(128,50);
dec = zeros(128,100);
del_x = zeros(128,1);

err = zeros(100,1);
my_err = zeros(100,10000);
ADC_error = zeros(128,10000);


%%%%%%%%%%%%%%%%ADC setup%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%Error correction

%%%%%%%%%%%%%%1st mu
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%
% Supply a sine input
resolution = 0:1e-3:1280;
Input2 = 1.25*sin(resolution)+1.25;
Input1 = -1.25*sin(resolution)+1.25;
mu = 2^16;

% Add some noise
for i = 1: length(Input2)
    Input1(i) = Input1(i)+ randn*(30*10^-6);
    Input2(i) = Input2(i)+ randn*(30*10^-6);
end

figure(1)
plot(resolution(1:10000),Input2(1:10000), resolution(1:10000),
Input1(1:10000));

% Rearranging weight
%group cap weight in segment 4 as one
%group cap weight in segment 5 as one
q1 = sum(WA(49:64,1));
q2 = sum(WA(65:80,1));
q3 = sum(WB(49:64,1));
q4 = sum(WB(65:80,1));

% WAside and WBside serve as the updated weight. For each 128 conversions,
% They will be updated

WAside = [WA(1:48); q1; q2];
WBside = [WB(1:48); q3; q4];

for x = 1: 10000
```

```
     % This is for sine input only. Using different parts of the sine
     % signal for error correction
     Vin2 = Input2(1,128*(x-1)+1: 128*x);
     Vin1 = Input1(1,128*(x-1)+1: 128*x);

    % Convert using ADC code
    R013108_ADC_16bit_randomization;
    % Apply error correction
    R013108_error_correction;
end

% Storing and see how each of the 100 weight error evolve
% over the x set of 128 conversions
sine1_err = my_err;
sine1_error = ADC_error;



%%%%%%%%%%%%%%%%1st mu
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%2nd mu
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%
mu = 2^14;

% Rearranging weight
%group cap weight in segment 4 as one
%group cap weight in segment 5 as one
q1 = sum(WA(49:64,1));
q2 = sum(WA(65:80,1));
q3 = sum(WB(49:64,1));
q4 = sum(WB(65:80,1));

% WAside and WBside serve as the updated weight. For each 128 conversions,
% They will be updated

WAside = [WA(1:48); q1; q2];
WBside = [WB(1:48); q3; q4];

for x = 1: 10000

     % This is for sine input only. Using different parts of the sine
     % signal for error correction
     Vin2 = Input2(1,128*(x-1)+1: 128*x);
     Vin1 = Input1(1,128*(x-1)+1: 128*x);

    % Convert using ADC code
    R013108_ADC_16bit_randomization;
    % Apply error correction
    R013108_error_correction;
end
```

```
% Storing and see how each of the 100 weight error evolve
% over the x set of 128 conversions
sine2_err = my_err;
sine2_error = ADC_error;


%%%%%%%%%%%%%%%%2nd mu
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%3nd mu
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%

mu = 2^12;

% Rearranging weight
%group cap weight in segment 4 as one
%group cap weight in segment 5 as one
q1 = sum(WA(49:64,1));
q2 = sum(WA(65:80,1));
q3 = sum(WB(49:64,1));
q4 = sum(WB(65:80,1));

% WAside and WBside serve as the updated weight. For each 128 conversions,
% They will be updated

WAside = [WA(1:48); q1; q2];
WBside = [WB(1:48); q3; q4];

for x = 1: 10000

     % This is for sine input only. Using different parts of the sine
     % signal for error correction
     Vin2 = Input2(1,128*(x-1)+1: 128*x);
     Vin1 = Input1(1,128*(x-1)+1: 128*x);

    % Convert using ADC code
    R013108_ADC_16bit_randomization;
    % Apply error correction
    R013108_error_correction;
end

% Storing and see how each of the 100 weight error evolve
% over the x set of 128 conversions
sine3_err = my_err;
sine3_error = ADC_error;


%%%%%%%%%%%%%%%%3rd mu
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%Error correction
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%Plot ADC error after error correction

%Plot how error1 evolves over x set of 128 conversion
%error1 in this case is the first cap weight error in ADC_B
conversion = 128*(1:x);
VLSB = 5/2^16;
sine1_error = sine1_error/VLSB;
sine2_error = sine2_error/VLSB;
sine3_error = sine3_error/VLSB;

figure(2)
semilogy(conversion,abs(sine1_err(1,:)),conversion,abs(sine2_err(1,:)),conver
sion,abs(sine3_err(1,:)));
grid;
title('Evolution of weight error over time');
xlabel('Conversion Index');
ylabel('Error(V)');
legend('u = 2^-16', 'u = 2^-14','u = 2^-12');

figure(3)
semilogy(conversion, abs(sine1_error(1,:)), conversion,
abs(sine2_error(1,:)),conversion, abs(sine3_error(1,:)));
grid;
title('ADC error');
xlabel('Conversion Index');
ylabel('ADC error(LSB)');
legend('u = 2^-16', 'u = 2^-14','u = 2^-12');

% To calculate standard deviation
% std of weight error
sine1_err_std = std(sine1_err(1,8000:10000))
sine2_err_std = std(sine2_err(1,8000:10000))
sine3_err_std = std(sine3_err(1,8000:10000))

% std of ADC error(LSB)
sine1_error_std = std(sine1_error(1,8000:10000))
sine2_error_std = std(sine2_error(1,8000:10000))
sine3_error_std = std(sine3_error(1,8000:10000))

% %%%%%%%%%%%Plot ADC error after error correction
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%
```

Modified SAR ADC

Same as before

Error Correction

Same as before

Frequency Response

Interface between mixed signal IC and external FPGA

```
clear all;
close all;
clc;

%%%%%%%%%%Overall function%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% This code do several things
%1) Set up the ADC parameters
%2.) Plot ADC error before correction
%3.) Apply error correction algorithm
%4.) Plot ADC error after correction

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%ADC setup%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% N denotes the number of bits of the ADC
N = 20;

 % These are weights got from the perfect 16 bit ADC step size (Book5 pg 54)
W1 = 0.145837866667;
W2 = W1/8;
W3 = W2/8;
W4 = W3/8;
W5 = W4/8;

% This denotes the initial weight used for both ADC_A and ADC_B in the
% digital to analog interface. The decisions from ADC_A will by muliplied
% by the estimated weight WA to get the analog voltage. The same happen for
% ADC_B. We assume we DO NOT KNOW of any error in the ADC itself, so we can
% just estimate them

WA = [W1*ones(16,1);
      W2*ones(16,1);
      W3*ones(16,1);
      W4*ones(16,1);
      W5*ones(16,1)];

WB = [W1*ones(16,1);
      W2*ones(16,1);
      W3*ones(16,1);
      W4*ones(16,1);
      W5*ones(16,1)];


% Initialization for before/after plot

Anum1 = zeros(1,16);
Anum2 = zeros(1,16);
Anum3 = zeros(1,16);
Anum4 = zeros(1,16);
Anum5 = zeros(1,16);
```

```
Bnum1 = zeros(1,16);
Bnum2 = zeros(1,16);
Bnum3 = zeros(1,16);
Bnum4 = zeros(1,16);
Bnum5 = zeros(1,16);

% Initialization
dec_A = zeros(128,50);
dec_B = zeros(128,50);
err = zeros(100,1);
dec_A_no_cal = zeros(10000,50);
dec_B_no_cal = zeros(10000,50);
dec_A_cal = zeros(10000,50);
dec_B_cal = zeros(10000,50);

%%%%%%%%%%%%%%%%ADC setup%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%Frequency Response of Original Signal(With some noise)

% Supply a sine input
resolution = 0:(2*pi/10):1280000*2*pi/10;
Input2 = 1.25*sin(resolution)+1.25;
Input1 = -1.25*sin(resolution)+1.25;

% Add some noise
for i = 1: length(Input2)
    Input1(i) = Input1(i)+ randn*(30*10^-6);
    Input2(i) = Input2(i)+ randn*(30*10^-6);
end

figure(1)
plot(resolution(1:20),Input2(1:20),'*', resolution(1:20), Input1(1:20),'o');

data = Input2(1:10000)-Input1(1:10000);

% Number of points sampled
n = 10000;

% sampling freqency
fs = 1e6;

% Time axis
dt = 1/fs;
T = dt*n;
t = 0:dt:dt*(n-1);

% frequency axis
df = 1/T;
fmax = 1/2*fs;
freq = [-fmax:df:fmax-df];
```

```
Vin_spec=fftshift(abs(fft(data)));
Vin_spec_square = abs(Vin_spec).^2;
spec_max = max(Vin_spec_square);
Vin_spec_square_dB = 10*log10(Vin_spec_square/spec_max);
figure(2)
plot(freq(5000:10000)/1000,Vin_spec_square_dB(5000:10000));
axis([0 500 -140 1]);
xlabel('frequency(kHz)');
ylabel('Magnitude(dB)');
title('Frequency spectrum');

%%%%%%%%%%%%%%%%%Frequency Response with Original Signal(with some noise)%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%Frequency Response(uncalibrated ADC)%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Vin2 = Input2(1:10000);
Vin1 = Input1(1:10000);
R013108_ADC_16bit_randomization;

% Rearranging weight
%group cap weight in segment 4 as one
%group cap weight in segment 5 as one
q1 = sum(WA(49:64,1));
q2 = sum(WA(65:80,1));
q3 = sum(WB(49:64,1));
q4 = sum(WB(65:80,1));

WAside = [WA(1:48); q1; q2];
WBside = [WB(1:48); q3; q4];

for i = 1:length(Vin1)
    t1 = sum(new_dA(i,49:64));
    t2 = sum(new_dA(i,65:80));
    dec_A_no_cal(i,:) = [new_dA(i,1:48) t1 t2];

    t3 = sum(new_dB(i,49:64));
    t4 = sum(new_dB(i,65:80));
    dec_B__no_cal(i,:) = [new_dB(i,1:48) t3 t4];
end

% Getting the analog output code(no calibration)
Vin_bar_after = (dec_B_no_cal*WBside + dec_A_no_cal*WAside)/2;
Vin_after_spec=fftshift(abs(fft(Vin_bar_after)));
Vin_after_spec_square = abs(Vin_after_spec).^2;
spec_max_after = max(Vin_after_spec_square);
Vin_after_spec_square_dB = 10*log10(Vin_after_spec_square/spec_max_after);

figure(3)
plot(freq(5000:10000)/1000,Vin_after_spec_square_dB(5000:10000));
axis([0 500 -90 1]);
xlabel('frequency(kHz)');
ylabel('Magnitude(dB)');
```

```
title('Frequency spectrum');

%%%%%%%%%%Frequency Response(uncalibrated ADC)%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%Frequency Response(calibrated ADC)%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


for x = 1: 10000

     % This is for sine input only. Using different parts of the sine
     % signal for error correction
     Vin2 = Input2(1,128*(x-1)+1: 128*x);
     Vin1 = Input1(1,128*(x-1)+1: 128*x);

    % Convert using ADC code
    R013108_ADC_16bit_randomization;
    % Apply error correction
    R013108_error_correction;
end

Vin2 = Input2(1:10000);
Vin1 = Input1(1:10000);
R013108_ADC_16bit_randomization;

for i = 1:length(Vin1)
    t1 = sum(new_dA(i,49:64));
    t2 = sum(new_dA(i,65:80));
    dec_A_cal(i,:) = [new_dA(i,1:48) t1 t2];

    t3 = sum(new_dB(i,49:64));
    t4 = sum(new_dB(i,65:80));
    dec_B_cal(i,:) = [new_dB(i,1:48) t3 t4];
end

% Getting the analog output code(with calibration)
Vin_cal = (dec_B_cal*WBside + dec_A_cal*WAside)/2;
Vin_cal_spec=fftshift(abs(fft(Vin_cal)));
Vin_cal_square = abs(Vin_cal_spec).^2;
spec_max_cal = max(Vin_cal_square);
Vin_cal_square_dB = 10*log10(Vin_cal_square/spec_max_cal);

figure(4)
plot(freq(5000:10000)/1000,Vin_cal_square_dB(5000:10000));
axis([0 500 -140 1]);
xlabel('frequency(kHz)');
ylabel('Magnitude(dB)');
title('Frequency spectrum');

%%%%%%%%%%Frequency Response(calibrated ADC)%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Modified SAR ADC

Same as before

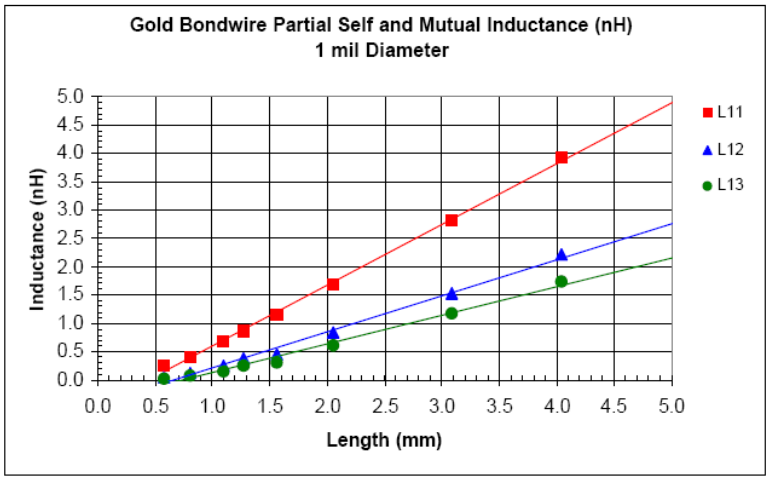
Error correction algorithm

Same as before

Appendix E

Bond Wire Data

3D Bondwire Electrical Modeling Results



Bondwire Diameter = 1 mil

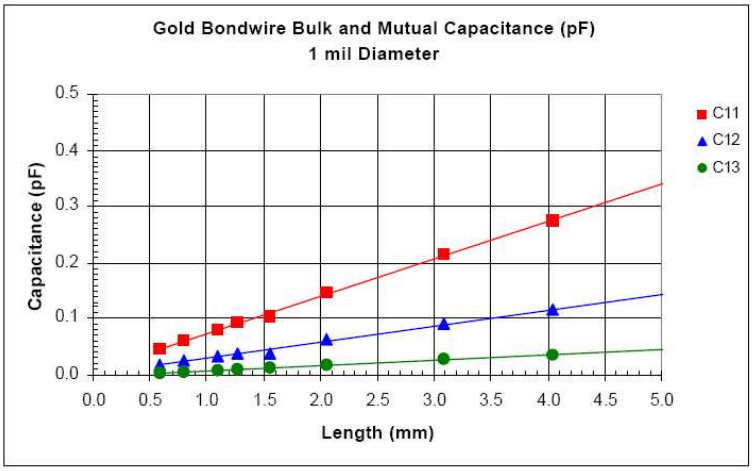
Die Pad Pitch = 75 um

Bond Pad Pitch = 160 um

Mold Compound Dielectric Constant = 3.9

Prepared by Elec. Pkg. Char. Group
7/16/01
Page1

3D Bondwire Electrical Modeling Results



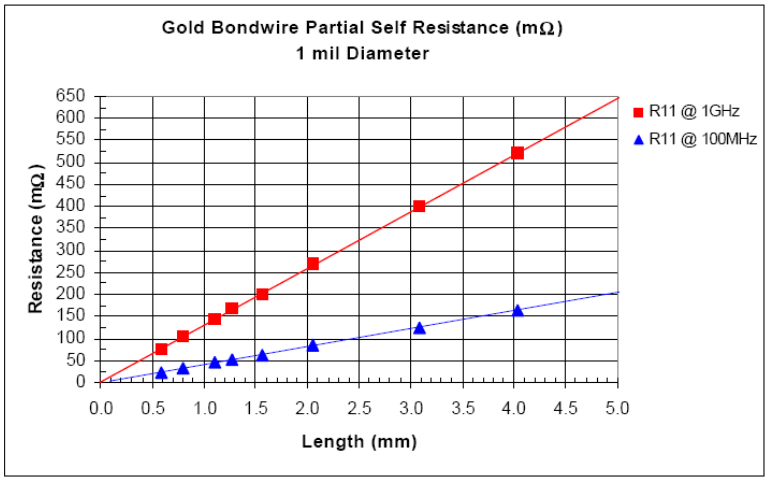
Bondwire Diameter = 1 mil

Die Pad Pitch = 75 um

Bond Pad Pitch = 160 um

Mold Compound Dielectric Constant = 3.9

3D Bondwire Electrical Modeling Results



Bondwire Diameter = 1 mil

Die Pad Pitch = 75 um

Bond Pad Pitch = 160 um

Mold Compound Dielectric Constant = 3.9

Appendix F

Matlab code used for calculating the values of coupling capacitors

```
clear all;
close all;
clc;

Cp1 = 16; % in pF
Cp2 = 2; % in pF

syms Cc2 Cc3 Cc4

Z4 = Cc4*Cp2/(Cc4+Cp2);
Z3 = (Cp2+Z4)*Cc3/(Cp2+Z4+Cc3);
Z2 = (Cp2+Z3)*Cc2/(Cp2+Z3+Cc2);

Z2R = Cp1+Cp2;
Z3R = Z2R*Cc2/(Z2R+Cc2);
Z4R = (Cp2+Z3R)*Cc3/(Cp2+Z3R+Cc3);
Z1R = (Cp2+Z4R)*Cc4/(Cp2+Z4R+Cc4);

eq2 = simplify((Cc2+Cp1+Cp2)*(Cp2+Z3+Z3R)/(Cc2*(Cp1+Cp2+Z2)));
eq3 = simplify((Cc3+Z3R+Cp2)*(Cp2+Z4+Z4R)/(Cc3*(Cp2+Z3+Z3R)));
eq4 = simplify((Cc4+Z4R+Cp2)*(Cp2+Z1R)/(Cc4*(Cp2+Z4+Z4R)));

 EQ2
='(8*Cc4+8+6*Cc3*Cc4+8*Cc3+4*Cc4*Cc2+4*Cc2+Cc3*Cc4*Cc2+2*Cc3*Cc2)/Cc2/(4*Cc4+
4+Cc3*Cc4+2*Cc3)=8';
 EQ3 = '(4*Cc4+4+Cc3*Cc4+2*Cc3)/Cc3/(Cc4+2)=8';
 EQ4 = '(Cc4+2)/Cc4=8';

 [Cc2, Cc3, Cc4] = solve(EQ2,EQ3,EQ4,Cc2,Cc3,Cc4)
```

Appendix G

Matlab code used for calculating harmonic distortion

Appendix 5C Matlab code for calculating harmonic distortion

```
clear all;
close all;
clc;

data = load('ahdli752.txt');

% signal frequency
f = 500000;

% Number of points sampled at one period
N = 10000;

% sampling freqency
fs = 5e9;

% Time axis
dt = 1/fs;
T = dt*N;
t = 0:dt:dt*(N-1);

% frequency axis
df = 1/T;
fmax = 1/2*fs;
freq = [-fmax:df:fmax-df];

figure(1)
plot(t*10^6,data(1:10000,2), 'r*');
xlabel('Time(us)');
ylabel('Voltage(V)');
hold on;
plot(t*10^6,data(1:10000,3), 'b+');
legend('Vin', 'Vhold');
hold off;

Vin_spec=fftshift(abs(fft(data(1:10000,2))));
Vhold_spec =fftshift(abs(fft(data(1:10000,3))));

figure(2)
subplot(2,1,1);
stem(freq,Vin_spec);
xlabel('frequency(Hz)');
ylabel('Magnitude of Vin');
%axis([-2.5e7 0.5e7 0 3]);
title('Frequency spectrum');

subplot(2,1,2);
stem(freq,Vhold_spec);
```

```
%axis([-0.5e7 0.5e7 0 3]);

xlabel('frequency(Hz)');
ylabel('Magnitude of Vhold');

% To calculate THD
Vhold_spec_sqr = Vhold_spec.^2;
loc = find(freq>f,1);
Vh_sqr_sum = sum(Vhold_spec_sqr(loc+1:N,1));
Vf_sqr = Vhold_spec_sqr(loc-1,1);
THD = 10*log10(Vh_sqr_sum/Vf_sqr);
```