Design Techniques for Parallel Pipelined ADC

by
Li Lin

ABSTRACT

The objective of this research is to test the feasibility of a 10-bit, 200MHz parallel pipelined ADC in 0.6μCMOS technology with varies of calibration techniques. We discuss the error sources of a parallel pipelined ADC and their interpretations. We also propose the calibration techniques. At the end, we conclude that a 10-bit, 200MHz parallel pipelined ADC in 0.6μCMOS is feasible with a low jitter(<1ps) clock and the calibration techniques, namely, digital error correction, digital gain calibration, channel gain calibration and channel offset cancellation.

Acknowledgments

I would like to express my sincere appreciation to my advisor Prof. Paul R. Gray. Numerous motivating and instructive discussions with him are the reason for the success of this project. His keen insight into IC design led me into the right direction of the research. I am grateful to his patient guidence and encouragement throughtout the program. I also thank Prof. Robert Meyer for all the discussion and proofreading of my thesis.

I would like to thank my colleagues in Prof. Gray's group. In particular, I thank Dr. Dave Cline, now at Level One, for many long hours with me discussing various issues about pipelined A/D system. His abundant knowledge and experience in A/D saved me much trouble in getting this project off the ground. I also thank Dr. Thomas Cho, now at SF Telecom, for his valuable suggestions and discussions related to this project. My gratitude also goes to Jacques-Christophe Rudell, Arnold Feldman, Andy Abo for reading various drafts of my thesis and their valuable feedback.

I also thank my other colleagues, particularly, Jennie Chen, Keng Leong Fong, Wai Lau, Edwin Chen, Martin Tsai, Tony Lin, Steve Lo, Monolis Terrovitis and others for making my life in Berkeley colorful.

I would like to thank my family for their support during the past two years at UC Berkeley. Their faith in me and never ceasing love are the impetus of my hard work.

This research was supported by SRC contract, DJ-570. I would also like to thank the Department of Education for a GAANN fellowship which provided me financial support for the 1994-95 academic year.

Table Of Contents

1

Introduction

1.1 Motivation

Digital signal processing has been proved over the past decade to be a robust and cost effective way of signal processing. The interface between the real world analog signal and the DSP function block is implemented by A/D converter. The examples of applications of A/D converters are video-imaging systems, personal communication systems, and disk drive read channel[1-17]. An A/D with 40MSamples/sec, 10-bit resolution has been reported[1] Higher speed A/D is required for future application, such as wireless communication. Pipeline is the fastest ADC architecture comparing to flash ADC, subrange ADC, successive approximation ADC and oversampling ADC. Shooting for even higher speed, parallelism is required to be incorporated into pipelined structure.

This work is a feasibility study of a parallel pipeline A/D with 200MHz sampling rate and 10-bit resolution. The most critical problem is the sample/hold block because it sees the full bandwidth of the input signal. The second most critical problem is the clock jitter because clock jitter on the sampling edge degrades the performance to a large degree when the input frequency is high. The third problem is the path mismatching. There are more problems related to the parallel pipeline structure which we will talk about in chapter 3 and chapter 4.

1.2 Thesis Organization

This thesis is divided into five chapters. Chapter 1 introduces the motivations of this work and thesis organization. Chapter 2 describes the architecture of parallel pipeline A/D converter. Chapter 3 discusses the nonidealities in the sample/hold circuit. Chapter 4 discuss the nonidealitites in parallel pipelined ADC. Chapter 5 gives the possible calibration techniques. Chapter 6 gives the simulation results and conclusions.

2

Parallel Pipeline A/D Architectures

2.1 Introduction

In this chapter, several ADC architectures used for high speed and relative high resolution conversion are presented. Pipeline has the advantages of lower power, high throughput, and relatively high resolution. For even higher speed, Parallelism has to be introduced.

2.2 Flash Architectures

A typical flash ADC architecture[17] is shown in Figure 1. By comparing the input signal to a set of reference voltages generated from a resistor string, we get a set of output codes. The number of required comparators, hence the required power consumption of the ADC and hardware cost, is proportional to 2^N(N is the number of bits). The accuracy requirement of the comparators is increased exponentially with N also. This requires autozero to be applied to the comparators when the resolution is greater than 8-bit. Size and complexity of 256 comparators is a problem. Without an input sample/hold circuit, the input signal must drive a large array of comparators. This can be very slow when resolution is 6-bit or greater.

Comparators mismatch is worse when the resolution is larger because the comparators are laid out over a larger area and are more subject to process variation.

If a single sample/hold circuit is connected between input and flash A/D, the throughput is slowed considerably. The S/H circuit also dissipates a lot of power since it drives a large array of comparators. If one S/H circuit is connected to each comparator, matching becomes the problem. In conclusion, flash A/D architecture is most useful at 6-7 bits with bare comparators at 100Msamples/sec or higher.

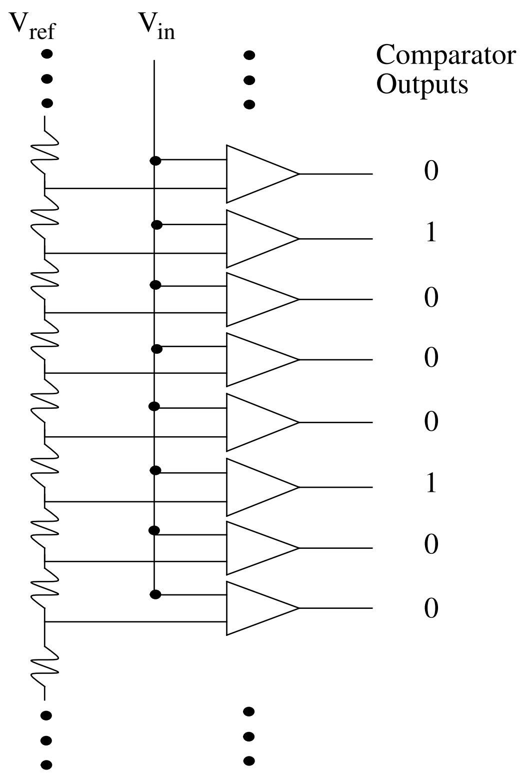


Fig. 1. A Typical Flash Architecture With Reference Generated From A Resistor String

2.3 Two-Step Flash ADC

Two-step flash ADC [16] is developed to reduce the number of comparators considerably. The architecture is shown in Figure 2. ADC1 does N/2 bit coarse quantization on the sampled input signal to get the MSBs. The D/A converts the MSBs to analog signal. This analog signal is subtracted from the sampled input signal. The residue is passed to ADC2, which does fine quantization on the residue and output LSBs. The number of comparators is $2 \times 2^{N/2}$. For 8-bit A/D, two-step flash A/D architecture requires 32 comparators(16 coarse comparators and 16 fine comparators), whereas straight flash A/D requires 256 fine comparators. Less comparators means less hardware cost and less power. The disadvantages of two-step flash are that it requires three full clocks and precision interstage processing. The hardware cost still grows exponentially with the overall number of bits. In conclusion, two-step flash is widely used in 8-bit, 20Mhz video ADCs.

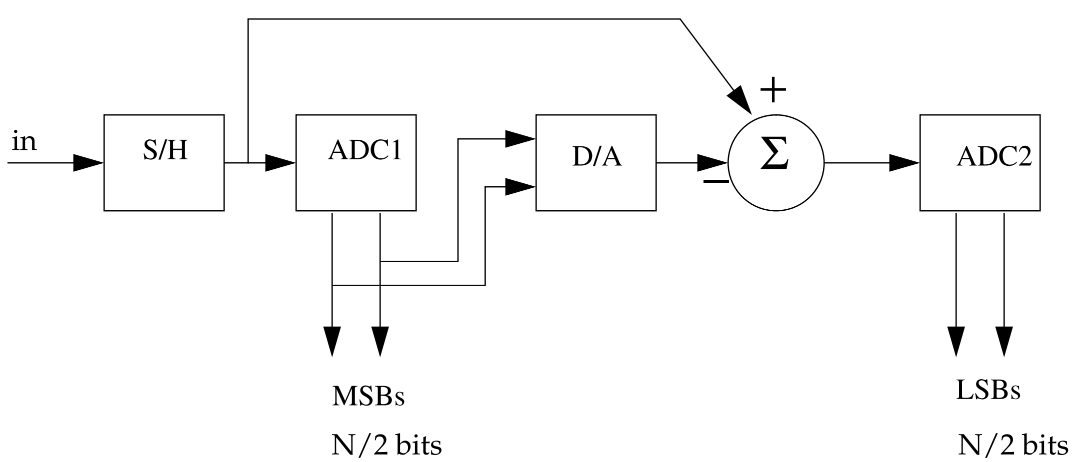


Fig. 2. Two-Step Flash ADC Architecture

2.4 Pipeline Architecture

To further reduce the hardware cost at high resolution, pipeline architecture[1-15] is employed. Figure 3 shows a typical pipeline A/D architecture. There are M number of identical stages, each quantizes k bits. So the overall resolution is Mxk. Each stage samples the output from the previous stage and quantize to k bits digital codes using flash ADC architecture. The codes then are converted to analog signal by k-bit DAC and subtracted from the sampled signal. The residue then is amplified by a gain of $G=2^k$. The output register combines the output bits from each stages and gives the final digital codes. Stage 1 gives MSBs, stage M gives the LSBs. The later the stage is, the less significant the bits it outputs because of the gain factor in each stage.

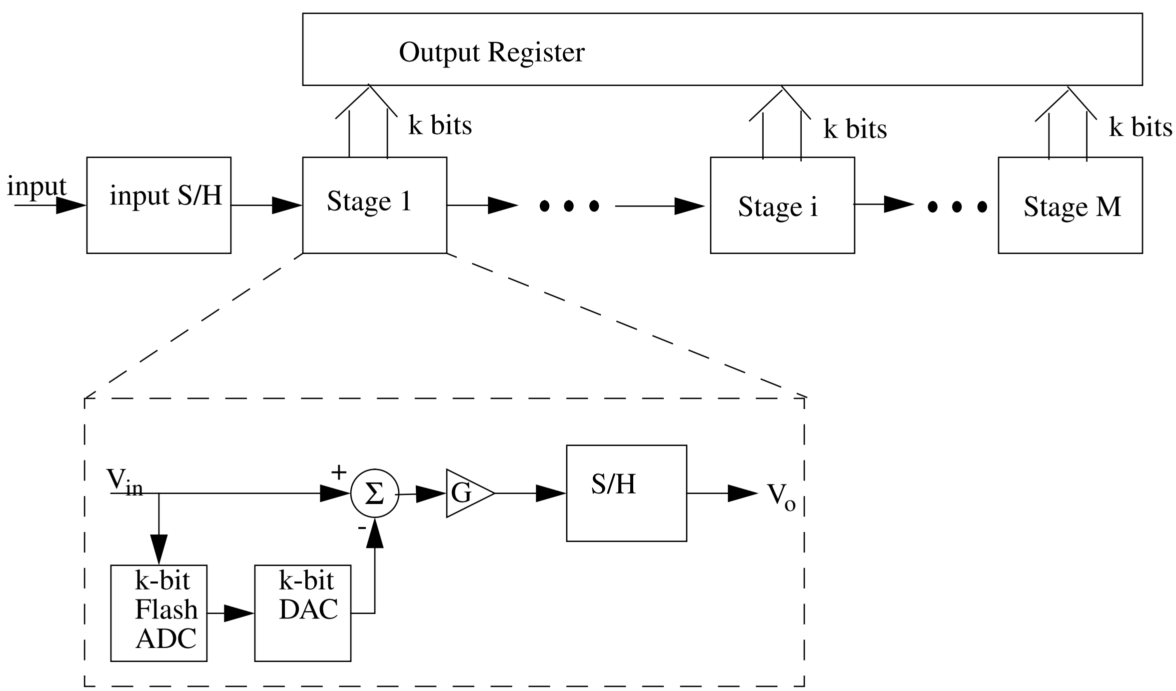


Fig. 3. A Typical Pipeline A/D Architecture

Pipeline architecture offers high degree of concurrence and approximately linear hardware cost with resolution. Assume R is the ratio of per stage hardware to pre-comparator hardware, the total hardware is $M(2^k+R)$. Usually k is a small number compare to the overall resolution. The key advantage of pipeline architecture is that it allows digital error correction and digital gain calibration and power minimization through capacitor scaling. These calibration will be discussed in chapter 4. Pipeline A/D is most popular in high speed (several MHz~50MHz) and high resolution (above 8-bit) application where latency is not a concern.

A typical implementation using switch capacitor circuit is shown in Figure 4. At first, the top plates of both C_S are connected to V_{in} and sampling switch M1 is closed. At the sampling instant, M1 is opened so that the charge on both sampling capacitors is captured. Then the C_F is connected to the output node. The top plate of C_S is connected to V_{ref} or $-V_{ref}$ depending on the output of the coarse flash ADC. Use the conservation of charge at node X and assuming C_S and C_F take the same value C, we can write the following equation:

$$V_{out} = 2^k(V_{in} \pm V_{DAC}) \quad \textbf{(eq 1)}$$

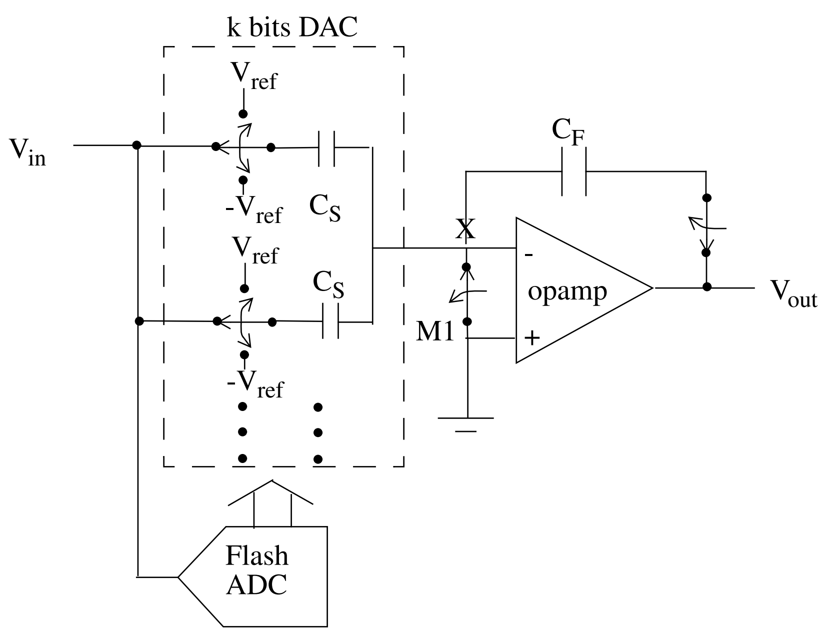


Fig. 4. A Typical Implementation of a Pipeline Stage

Figure 5 shows the transfer function curve of a pipeline stage with k=2,e.g., 2-bit per stage. The slope of the curve is 2^k which is the gain of the stage.

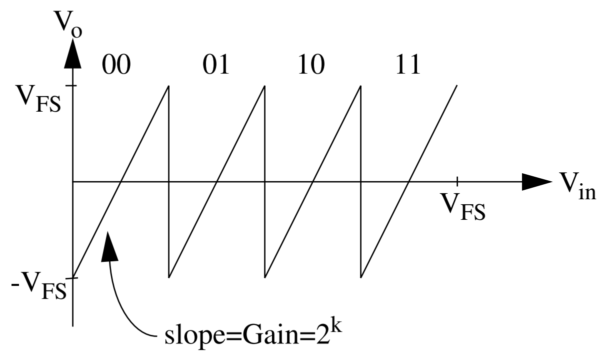


Fig. 5. Transfer function curve of a pipeline stage

2.5 Parallel Pipeline Architecture

When even higher conversion speed is desired, parallelism needs to be applied. Figure 6 shows a system view of the architecture for parallel pipeline A/D. If N channels are used, each channel is operated at frequency of f_S/N, where f_S is the system overall sampling frequency. An analog multiplexor and a digital multiplexor are used at the input and the output of the system. This way throughput of the system is increased by N times at the cost of increasing hardware. Since the DAC reference can be shared by all the channels, the hardware increase in less than the first order linear dependence. Sharing DAC reference is also desired to improve the matching of the channels.

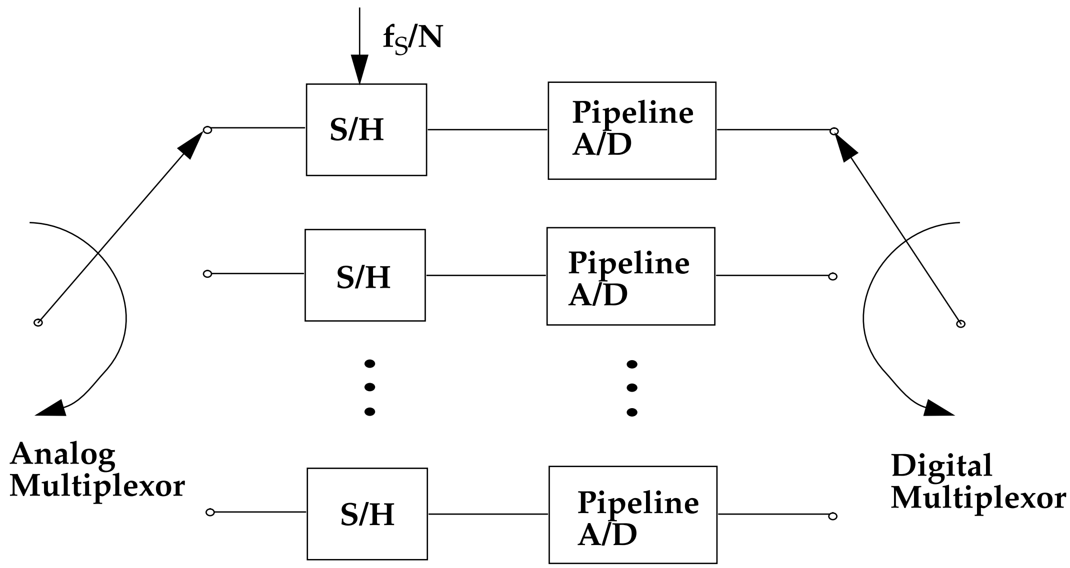


Fig. 6. A System View of Parallel Pipeline A/D Architecture

In Figure 7, an example of the clock scheme of a 4-channel parallel pipelined A/D system is shown. It's based on the concept of time-interleave. The sampling instant of each S/H in each channel is shifted by a quarter of the clock period. The arrows show a full cycle of the operation through the four channels.

The most critical problem that needs to be solved in parallel pipeline structure is how to improve the matching between channels. This directly affect the highest resolution we can achieve. Past work[7] shows 8-bit resolution at 85Mhz sampling rate can be achieved. This work is shooting for 10-bit resolution and 200Mhz sampling rate. Several calibration techniques will be discussed in chapter 5.

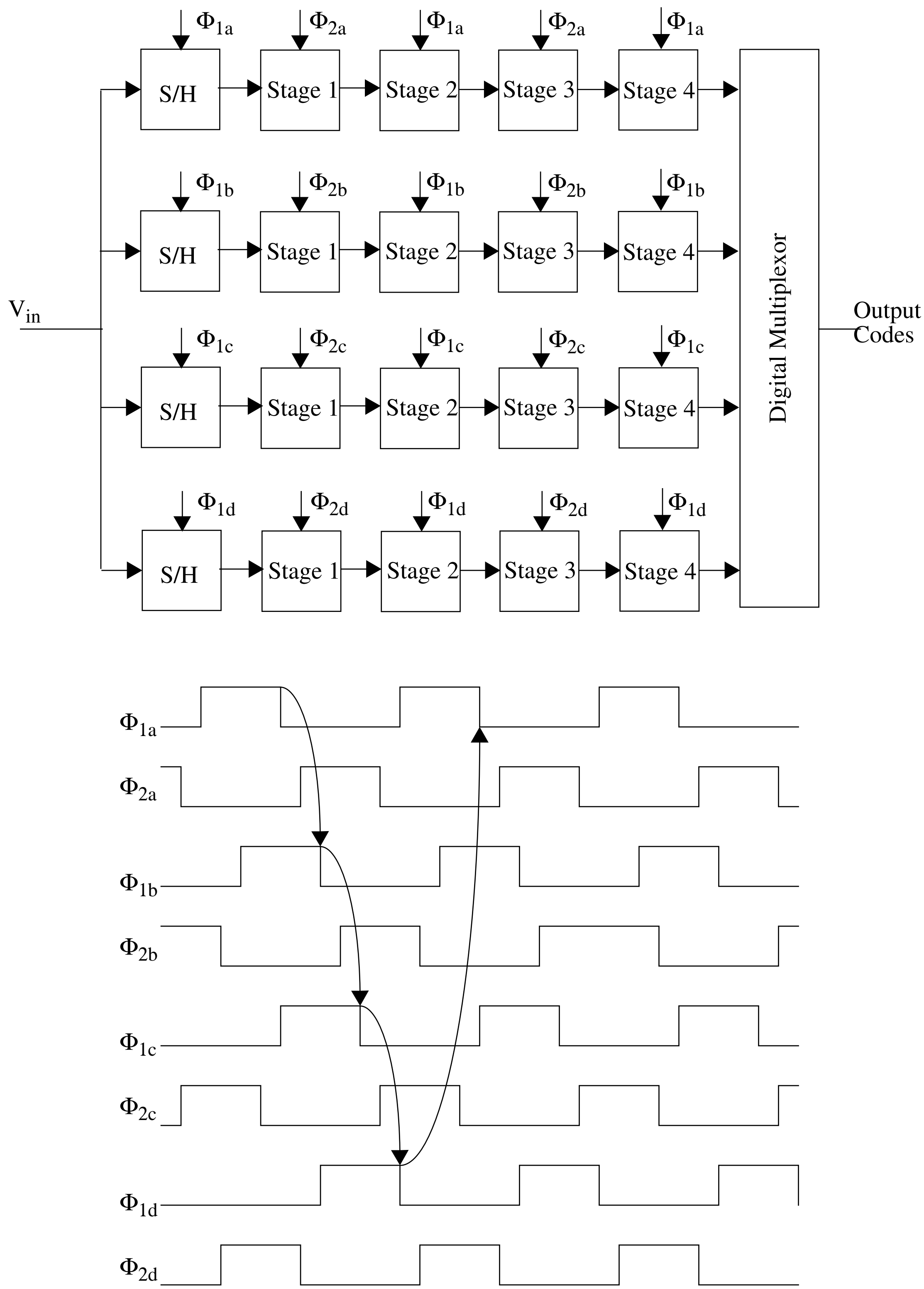


Fig. 7. A four-channel parallel pipeline A/D clocking example

2.6 Oversampling A/D Architecture

Another possible A/D architecture is oversampling A/D, or sigma-delta A/D. This architecture is widely used in high resolution (greater than 10-bit) applications. The sampling frequency is much higher than the input signal frequency because oversampling is utilized. This characteristic determines that most applications of this architecture is in the frequency range less than 5Mhz. This is clearly not a good choice for the application where a couple of hundreds of MHz of the sampling rate is utilized.

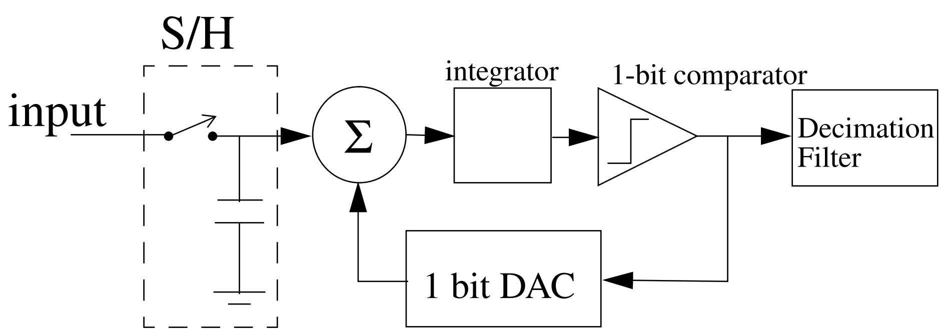


Fig. 8. An oversampling architecture

3

Nonidealities Of Sample/ Hold Circuit

3.1 Sample/Hold Nonidealities

The fundamental accuracy of the A/D converter is limited by the accuracy of the sample/hold circuit[11]. At 40Mhz, 10-bit resolution is easily achieved by a switch-cap circuit with an opamp to hold the voltage. For higher frequency application, careful circuit design is necessary since the sample/hold circuit sees the full bandwidth of the input signal.

3.1.1 Basic Errors of Sample/Hold

The function of a S/H circuit is to track the analog input and to sample the input at an instant and hold that value for the next stage. The simplest implementation of a S/H circuit is a MOS switch with a capacitor load. The circuit is shown in Figure 9. Ideally the circuit accurately tracks the input when the clock is set to track. At the instant of clock is set to hold, the circuit samples the input at that instant and holds it without drooping. The ideal waveform is shown in Figure 10.

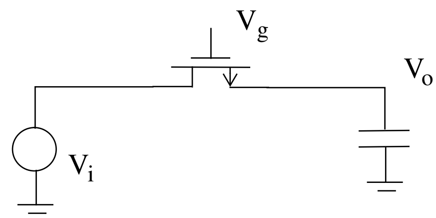


Fig. 9: A Simplest Implementation of S/H

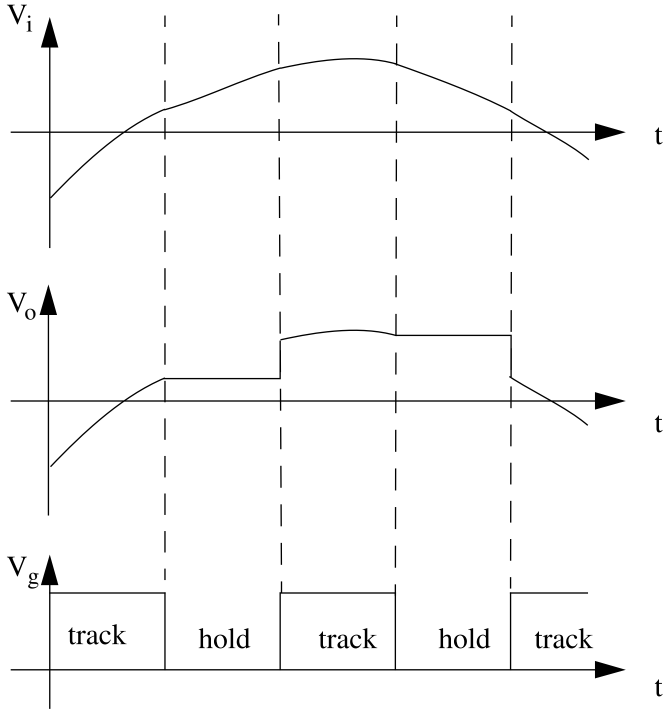


Fig. 10: Waveform of An Ideal S/H Circuit

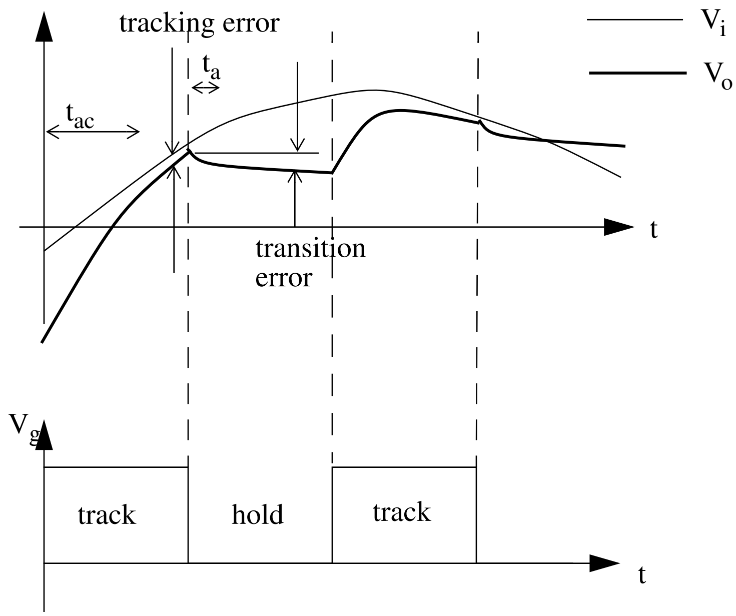


Fig. 11: Waveform of A Real S/H

In reality, the MOS switch has finite resistance that is in series with sampling capacitor C_S. Hence the circuit has finite bandwidth with time constant $R_{on}C_S$ and an acquisition time t_{ac} is required before the circuit can track the input. The error is shown in Figure 11.

The finite bandwidth also causes the circuit tracks the input with a tracking error. The transfer function is equivalent to a linear filter with time constant $\tau = R_{on}C_S$, e.g.,

$$\frac{V_o(s)}{V_i(s)} = \frac{1}{1 + s\tau} \qquad \text{(Eq 1)}$$

The tracking error due to the finite bandwidth of the S/H circuit doesn't affect the linearity of the S/H. But the tracking error due to the nonlinearity of the MOS switch does contribute to the nonlinearity of the S/H. In MOS switch, R_{on} increases nonlinealy with input signal and C_j (junction capacitance of the MOS) decreases nonlinearly with the input signal. So there is a certain degree of cancellation effect on the bandwidth of the S/H.

There is mismatch between the desired sampling instant and the actual sampling instant of the S/H circuit. The mismatch is called aperture time t_a. There are three components contribute to this error. The first component is the fixed aperture time due to clock skew and ?. This only affects the sampling phase and is easy to calibrate out. The second component which is due to the clock jitter greatly affect the performance of the S/H. Assuming a sinusoidal waveform with a frequency f_{in} and an amplitude V_a which is half of the full scale V_{FS} is at the input of the S/H, B is the bits of resolution we want to achieve, we can calculate the jitter requirement as following,

$$V_{in} = V_a \sin 2\pi f_{in} t \tag{Eq 2}$$

The maximum slew rate is:

$$SR_{max} = V_a \times 2\pi \times f_{in} \tag{Eq 3}$$

Then the jitter Δt should be

$$\Delta t \le \frac{\Delta V}{SR_{max}} = \frac{V_{FS}/2^{B+1}}{SR_{max}} = \frac{V_{FS}/2^{B+1}}{V_a \times 2\pi \times f_{in}} = \frac{1}{\pi f_{in}} \times \frac{1}{2^{B+1}} \tag{Eq 4}$$

For an input frequency of 200MHz, a clock jitter of less than 0.78ps is required to achieve 10-bit resolution. This effect puts a stringent requirement on the clock network design.

The third component is due to the fact that the turning off instant of the switch is a function of the input signal magnitude and slew rate. This is the most difficult component to calibrate out.

Transition error occurs due to the channel charge injection when the MOS switch is turned off. The fixed component of the charge injection appears as a DC offset and hence can be easily cancelled out. The signal dependent component of the charge injection is more difficult to calibrate out. But using fully differential approach can eliminate the first order error if the matching is perfect.

The final error droop comes from the leakage of the switch and the storage element. For high frequency application this is not a severe problem.

3.1.2 Design of S/H Circuit

3.1.2.1 Conventional Design of S/H

Conventional Sample/Hold utilizes opamp-based switchcap technique. Figure 12 shows a differential S/H with single sampling switch and the timing diagram of the clock control signals. When Φ_{1-} goes low, M2 and M3 release the common mode voltage of the inputs of opamp. M2 and M3 are chosen to be much smaller than M1 so that the charge injection error is small and decreasing exponentially until M1 is turned off. Then Φ_1 goes low and M1 is turned off. The signal is sampled on the sampling capacitors C_S. The charge injection from M1 is approximately distributed evenly onto two sampling capacitors C_S and will not appear on the output because the differential configuration is employed. Then Φ_{1+} goes low and M5 and M6 are turned off. Since the charge on the bottom plates of the sampling capacitors has no place to go, the charge injection from M5 and M6 doesn't affect the charge on the sampling capacitors, hence no effect on the sampled signal. This is called bottom plate sampling. M5 and M6 are chosen to be large so that the effect of the nonlinearity of the pass transistors on-resistance is minimized. When Φ_{1+} goes low, M9, M10, M11 and M12 are turned off and the output node is released with common mode voltage. Then Φ_2 goes high, M4, M7 and M8 are turned on and the charge captured on the sampling capacitors C_S are transferred onto the C_F.

It can be shown that the percentage of nonlinearity of the S/H circuit is proportional to $\frac{f_{in}}{f_T} \cdot \frac{f_{in}}{BW} \cdot \frac{A}{(V_H - V_L)} \cdot \frac{C_S}{C_T}$, where f_{in} is the input frequency, BW is the bandwidth of the S/H, A is the peak amplitude of input, V_H and V_L are the clock high and low voltages, C_S is the sampling capacitance, C_T is the total capacitance, including C_S, C_F and parasitic capacitance. If the S/H is perfectly matched between the differential paths, then the offset of the output voltage is zero. Otherwise, the offset is inversely proportional to $\frac{f_{in}}{f_T} \cdot \frac{f_{in}}{BW} \cdot \frac{A}{(V_H - V_L)} \cdot \frac{C_S}{C_T}$. [11].

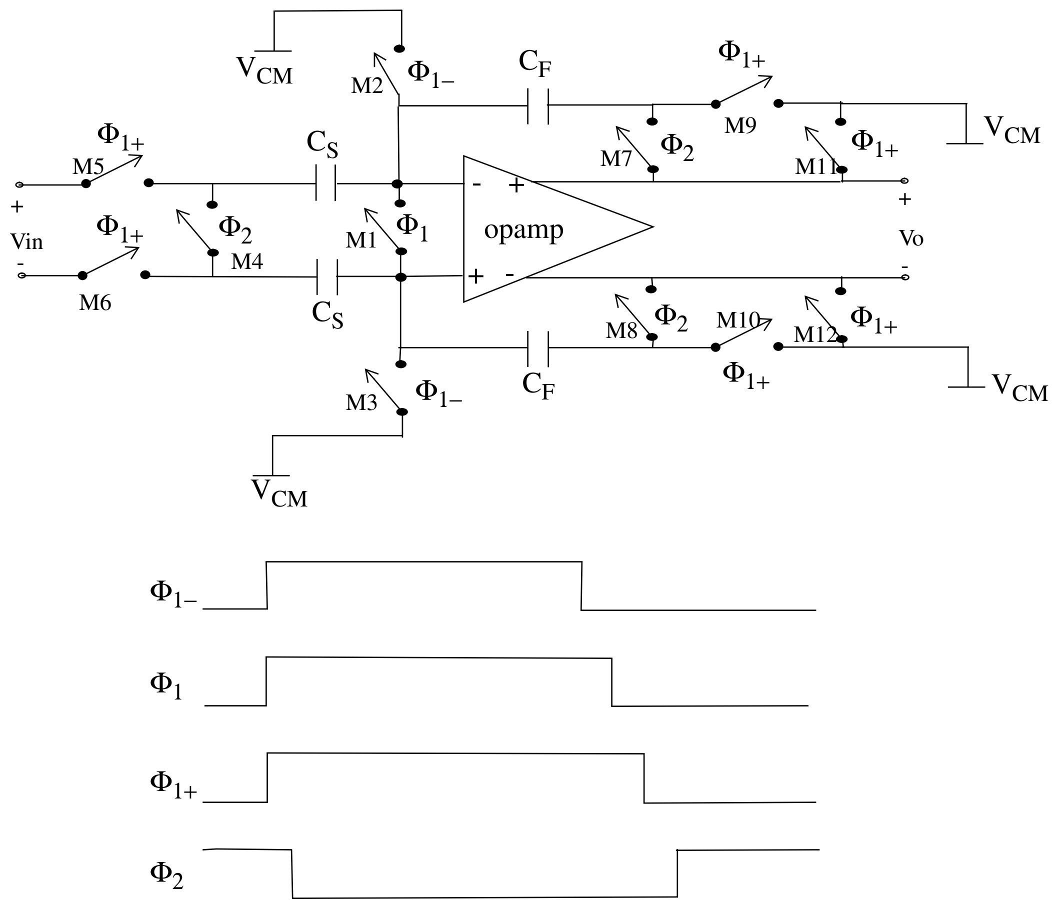


Fig. 12: Differential S/H With Single Sampling Switch And Timing Diagram of The Clock Control Signal

<u>3.1.2.2 Three possible configuration of S/H</u>

The important feature of the S/H circuit for parallel pipeline A/D is that it's fast and has high SNR. The gain of the S/H is not necessary greater than one, which is in the case of the circuit shown in Figure 12. Three possible configuration of S/H are shown in Figure 13. Only single ended version is shown for simplicity.

Circuit A is the simplified version of the S/H just shown in previous section. Circuit is a S/H with single sampling capacitor. Circuit C is a S/H with shared feedback capacitor. For the S/H block in front of each parallel channel of pipeline A/D, the absolute value of the gain is not important. The matching of the gain affects the whole parallel pipeline A/D to a great degree. Circuit B achieve a gain of one independent of the matching between the sampling capacitance of different channels, assuming loop gain Af is large enough. The time constant of circuit B is also much smaller than circuit A and C, hence it settles much faster than circuit A and C. Increasing C_S in circuit B doesn't affect the settling speed very much, but can reduce the kT/C noise lineally. Table 1 shows the comparison of the performance of these three configuration. For the S/H blocks of parallel pipelined A/D, circuit B is chosen for the various reasons mentioned above.

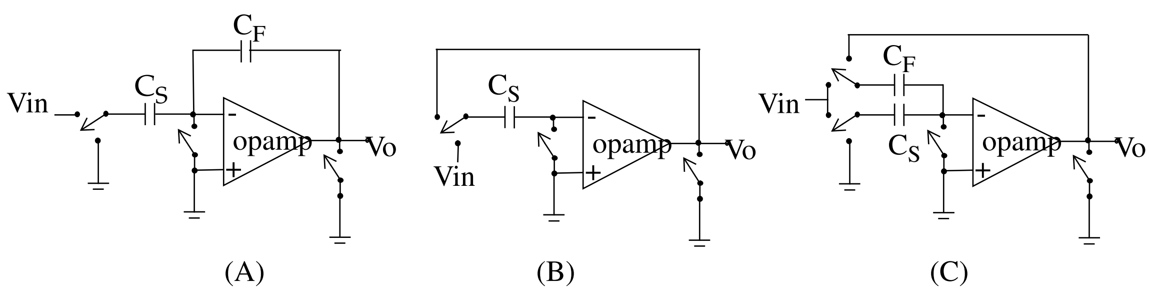


Fig. 13: Three Possible Configurations of Opamp-based SC S/H

Inside each pipeline stage there is a S/H function block. A gain greater than one is necessary to amplify the difference of analog input signal and the coarse digitized signal. Circuit B is no longer suitable for this purpose. We choose circuit C over circuit A for each pipeline stage for the following reasons:

1) For the same amount of gain, circuit C has a larger feedback factor than circuit A, hence settles faster than circuit A.

2) Larger feedback factor means larger loop gain Af, where A is the opamp DC gain. Therefore gain of circuit C is more accurate than gain of circuit A.

3) Circuit C has less kT/C noise than circuit A.

Circuit	A	B	C
Gain	$\frac{C_S}{C_F}\cdot\frac{1}{1+\frac{1}{Af}}$	$\frac{1}{1+\frac{1}{Af}}$	$\left(1+\frac{C_S}{C_F}\right)\cdot\frac{1}{1+\frac{1}{Af}}$
Feedback Factor f	$\frac{C_F}{C_S+C_F+C_g}$	$\frac{C_S}{C_S+C_g}$	$\frac{C_S}{C_S+C_F+C_g}$
τ w/o parasitic	$\frac{C_F C_L + CC + CC}{g_m C_F}$	$\frac{C_L}{g_m}$	$\frac{C_F C_L + C_F C_S + C_S C_L}{g_m C_F}$
τ with parasitic C_g $C_{SP}=C_S+C_g$	$\frac{C_F C_L + C_F C_{SP} + C_S C_L}{g_m C_F}$	$\frac{C_S C_L + C_S C_g + C_g C_L}{g_m C_S}$	$\frac{C_F C_L + C_F C_{SP} + C_{SP} C_L}{g_m C_F}$
kT/C noise w/o parasitic	$\frac{kT}{C_S}$	$\frac{kT}{C_S}$	$\frac{kT}{C_S+C_F}$

Table 1: Comparison of Three Configurations

4

Nonidealities In Parallel Pipelined A/D

4.1 Introduction

There are many error sources in the actual implementation of a parallel pipelined A/D. Basically they can be divided into two groups, systematic error and random error. Systematic error can be taken out completely in theory. One example of a systematic error source is the capacitors mismatching. Although the mismatching of capacitors is random, once the chip is manufactured, the pattern of mismatching is fixed. In another word, the output is deterministic with respect to a certain input. A calibration techniques called digital gain calibration can be used to remove the error due to the capacitors mismatching. We will leave the discussion of calibration techniques to chapter 5. Random error on the other hand is hard to calibrated. But certain techniques can be applied to reduce the random error. One example of random error is the thermal noise. This is the fundamental error in any circuit. Sizing up the sampling capacitor is one of the way to reduce this error.

Error sources of a parallel pipelined A/D can also be divided into two groups based on whether it's related to parallelism.Error sources related to parallelism can be characterized by three major errors, channel offset, channel gain mismatch, and timing mismatch. In this section, we will discuss errors according to these categories. A single pipelined A/D outputs error bits or missing bits due to comparator offsets, stage gain error, nonlinear DAC reference level, nonlinear S/H and not sufficient settling, and thermal noise.

4.2 Error Sources Related To Parallelism

Path mismatch is the fundamental problem in the parallel architecture[8].. In the following three sections we will discuss the sources and the interpretation of the channel offset mismatch, channel gain mismatch and timing mismatch of the parallel architecture. We will assume the system sampling rate is f_s (sampling period $T=1/f_s$) and N channels are used in the parallel structure. Hence the individual channel sampling rate is f_s/N.

4.2.1 Channel Offset

The sources of channel offset mismatch are opamp offset mismatch, MOS switches charge injection mismatch in the S/H of each channel, and the gain mismatch. Channel offset can be modeled as an extra voltage source connected in series with the ideal S/H and pipelined stages as in Figure 14. The value of the offset is the input referred offset of each channel.

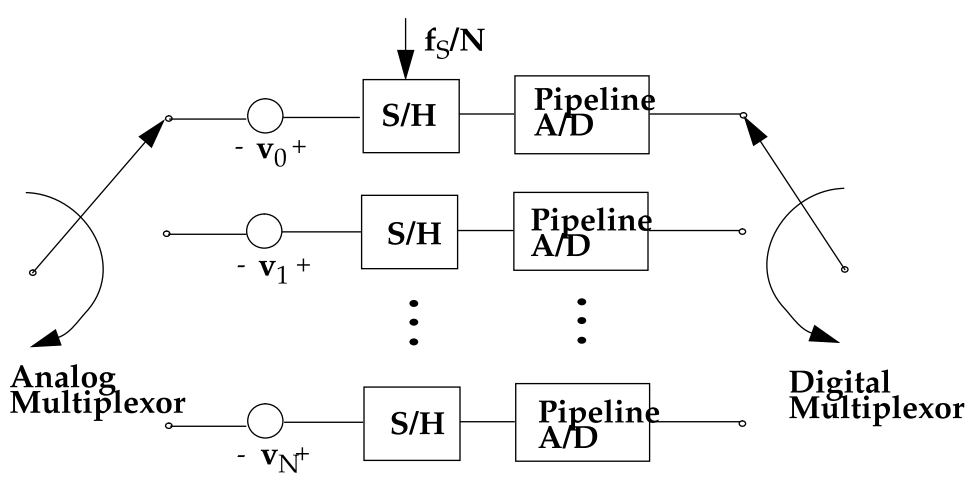


Fig. 14: Model of Channel Offset Mismatch

The offsets are directly added to the outputs, which means the offsets spectrum is directly added to the output spectrum. Assuming the offsets are time invariant, at output node, we see a periodic sequence $\{v_n, n=0,1,2,...\}$ with period N, e.g., $v_n=v_{n+N}$, where N is the total number of channels. Apply Discrete Fourier Transform(DFT) to this periodic sequence, we get the Fourier series coefficients V_k:

$$V_k = \sum_{n=0}^{N-1} v_n \cdot e^{-j\left(2\frac{\pi}{N}\right)kn} \qquad \text{(Eq 5)}$$

Figure 15 shows the time domain and frequency domain representation of the channel offset mismatch and the output spectrum which is the superposition of the ideal output spectrum and the channel offset spectrum. Notice that V_k is also periodic. We see the effect of channel offset mismatch is additive and independent of the input signal level and frequency. This results in tones at multiples of channel sampling rate f_s/N.

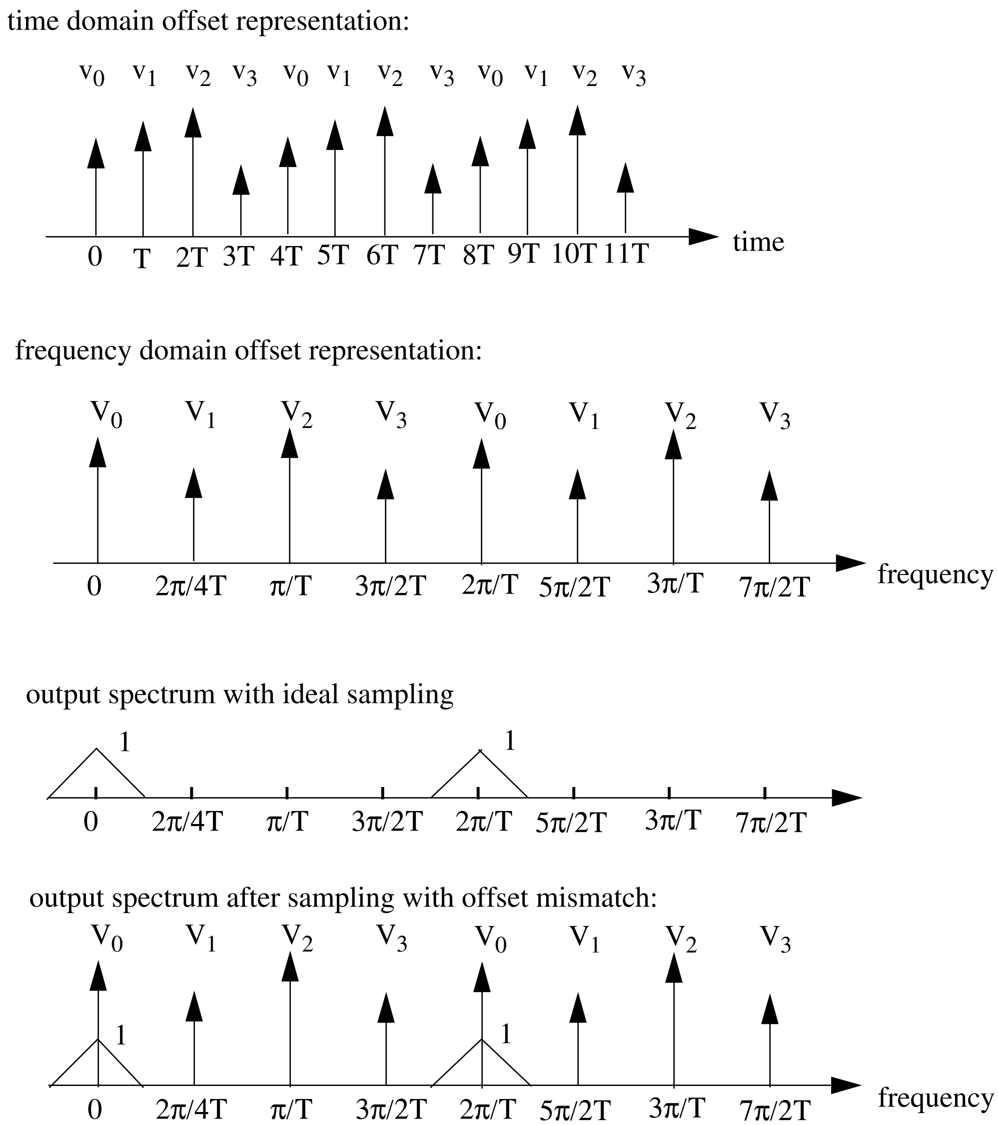


Fig. 15: Time Domain And Frequency Domain Representation of A 4-channel Parallel A/D Offset Mismatch

4.2.2 Channel Gain Mismatch

The major source of gain mismatch is capacitor mismatch. Opamp dc gain mismatch is another source but not a significant one. Channel gain mismatch can be modeled as an extra gain stage connected in series with the ideal S/H and pipelined stages as in Figure 16.

The gain is multiplied to the outputs, which means the spectrum of the gain mismatch is convolved with the output spectrum. Assuming the gain mismatch are time invariant, at output node, we see a periodic sequence $\{g_n, n=0,1,2,...\}$ with period N, e.g., $g_n=g_{n+N}$, where N is the total number of channels. Apply Discrete Fourier Transform(DFT) to this periodic sequence, we get the Fourier series coefficients G_k:

$$G_k = \sum_{n=0}^{N-1} g_n \cdot e^{-j\left(\frac{2\pi}{N}\right)kn} \qquad \text{(Eq 6)}$$

If the input spectrum is $X(j\omega)$, then the output spectrum $Y(j\omega)$ is:

$$Y(j\omega) = \frac{1}{T}\sum_{k=-\infty}^{\infty} G_k X\left(j\left(\omega - \frac{2\pi k}{NT}\right)\right) \qquad \text{(Eq 7)}$$

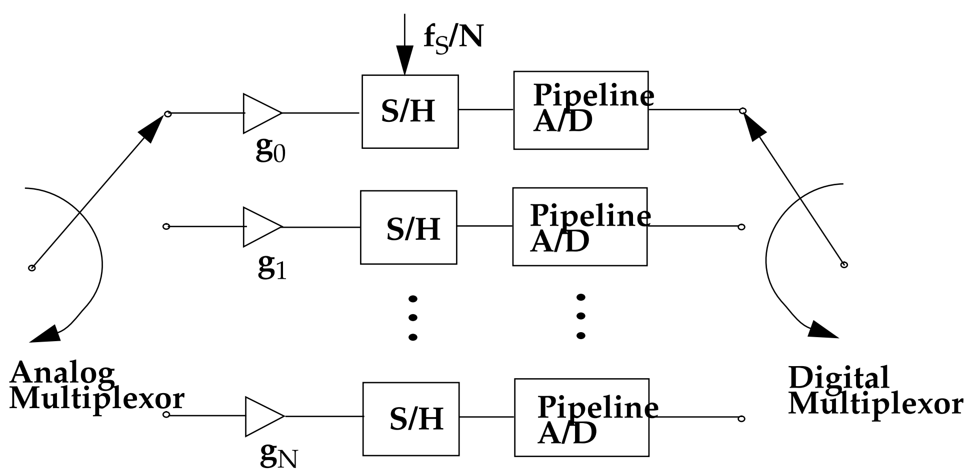


Fig. 16: Model of Channel Gain Mismatch

Figure 17 shows the time domain and frequency domain representation of the channel gain mismatch and the output spectrum which is the convolution of the ideal output spectrum and the channel gain spectrum. Notice that G_k is also periodic. The convolution results in new frequency components at the output spectrum. A special case is that all g_n have the same value, e.g., there is no gain mismatch in the system, then $G_1=1$ and $G_n=0$ for $n>1$.

When the input single side bandwidth is larger than half of the channel sampling rate, e.g., $f_s/2N$, aliases of the input spectrum are produced. These aliasing effect is shown in Figure 18. We can view this process as a down sampler with a factor of N and with amplitude modulation. At the output end, the multiplexor can be viewed as an up sampler with a factor of N. Ideally the aliasing effect can be recovered if the demultiplexor at the input and the multiplexor at the output are synchronous. But if the input single side bandwidth is greater than half of the system sampling rate, e.g., $f_s/2$, the aliasing effect can not be recovered.

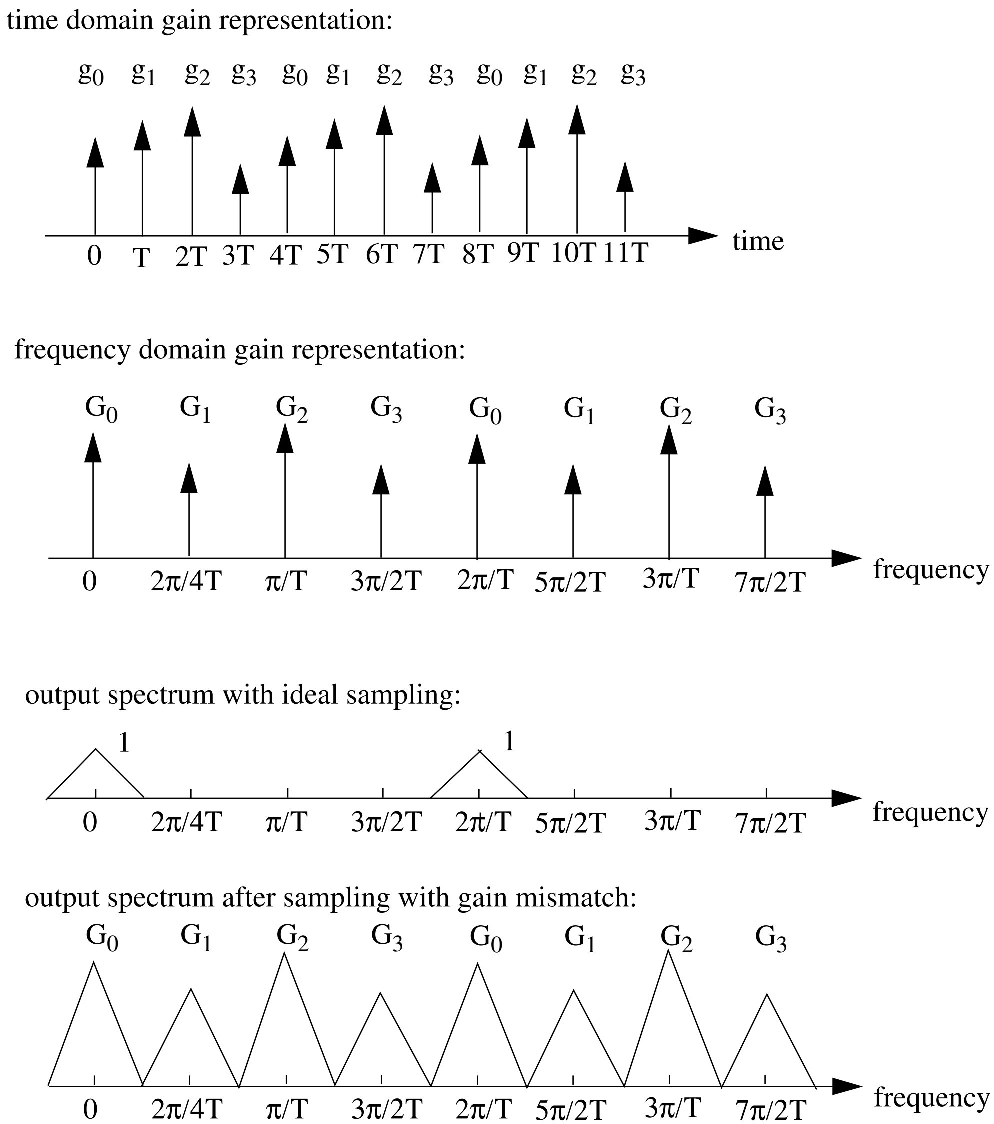


Fig. 17: Time Domain And Frequency Domain Representation of A 4-channel Parallel A/D Gain Mismatch

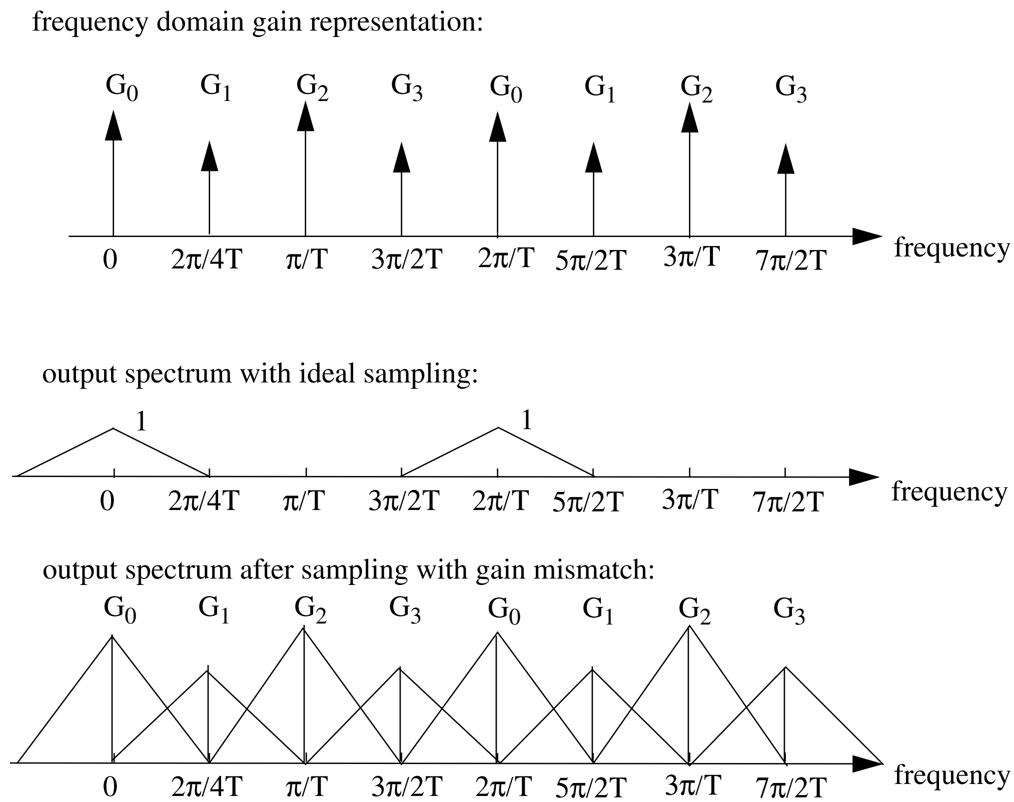


Fig. 18: Aliasing Effect of Gain Mismatch of A 4-channel Parallel A/D

4.2.3 Timing Mismatch

Timing requirement of the S/H blocks in front of each channel is the most critical comparing to the timing requirement of the rest of the pipeline stages. The reason is that the S/H sees the full bandwidth of the input signal whereas the later pipelined stage sees the hold hence slow varying signal from the previous stage. The sources of timing mismatch are the clock generator network, the actual turning off time of the MOS switches in the S/H circuits, and the thermal noise of the electronic devices. Systematic mismatch can be viewed as frequency dependent gain mismatch at the output spectrum. On the other hand, random mismatch like clock jitter can be viewed as white noise added on top of the signal spectrum.

Figure 19 shows the timing instants of an actual parallel pipelined A/D. The ideal sampling instants are at the multiples of T. We will analyze the systematic timing mismatch in frequency domain.

Assuming in channel n the timing offset is Δt_n, then the output spectrum $Y(j\omega)$ in terms of input spectrum $X(j\omega)$ is:

$$Y(j\omega) = \frac{1}{T}\sum_{k=-\infty}^{\infty}\Phi_k(\omega)X\left(j\left(\omega - \frac{2\pi k}{NT}\right)\right) \qquad \text{(Eq 8)}$$

where $\Phi_\kappa(\omega)$ is:

$$\Phi_k(j\omega) = \sum_{n=0}^{N-1} e^{j\left(\omega - \frac{2\pi n}{NT}\right)\Delta t_n} e^{-j\left(\frac{2\pi}{N}\right)nk} \qquad \text{(Eq 9)}$$

Comparing the output spectrum with timing mismatch and the output spectrum with gain mismatch, we find a similar phenomenon that aliases occur at multiples of channel sampling rate $1/Nf_s$. The difference is that in timing mismatch case, the aliases depend on the input frequency whereas in gain mismatch case the aliases do not depend on the input frequency. We can view the timing mismatch as frequency dependent gain mismatch.

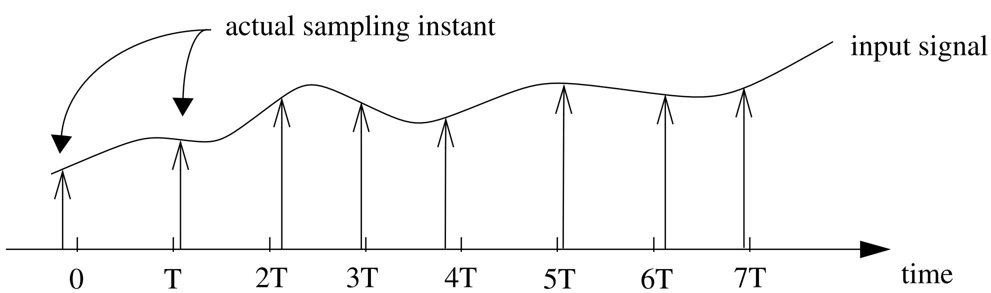


Fig. 19: Timing Mismatch of A 4-channel A/D

Clock jitter can be viewed as white noise. It directly adds to the noise floor and degrades the SNR of the system.

4.3 Error Sources In Single Pipeline

Error sources in single pipeline include comparator offsets, stage gain error, nonlinear DAC reference levels, nonlinearities and insufficient settling in S/H, and kT/C noise. Nonlinearities and insufficient settling effect in S/H is discussed in chapter 3. In this section, we will discuss the other sources and interpretations of the error sources.

4.3.1 Comparator Offsets

The flash ADC in each pipelined stage compares the input of that stage to a set of reference voltages and output the digital code. Figure 20 shows the transfer curve of a pipeline stage with comparator offset. Everything but comparator threshold is assumed to be ideal. We see a wrong decision is made and the amplified residue of this stage is out of the range of the input of next stage. As a result, missing code occurs.

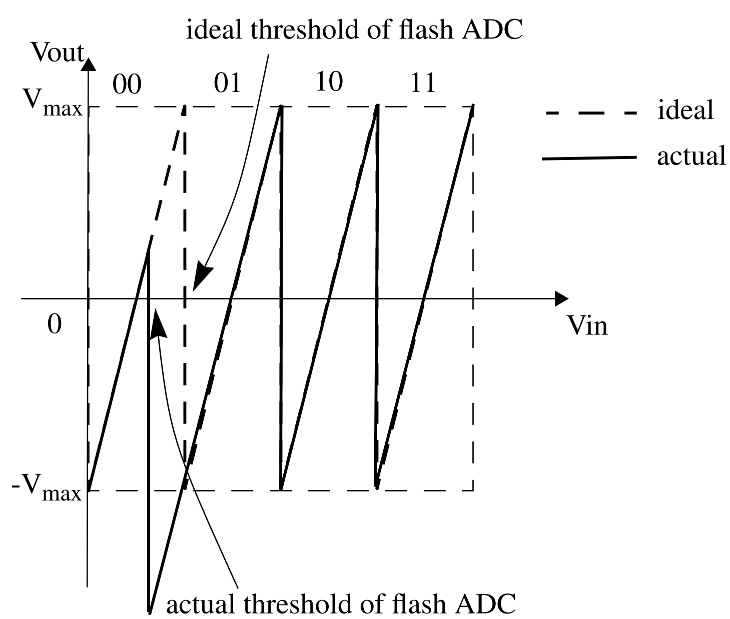


Fig. 20: Transfer Curve of A Pipeline Stage With Comparator Offset

4.3.2 Nonideal DAC Reference Levels

After the input is digitized by the coarse flash ADC, a DAC is used to convert the code to analog signal. The level of the analog signal depends on the level of the DAC reference levels. The shift of DAC reference levels results in out-of-range output and hence missing codes. Figure 21 shows the transfer curve of a pipeline stage with nonideal DAC reference levels. The source of the nonideal DAC reference levels is capacitor mismatch.

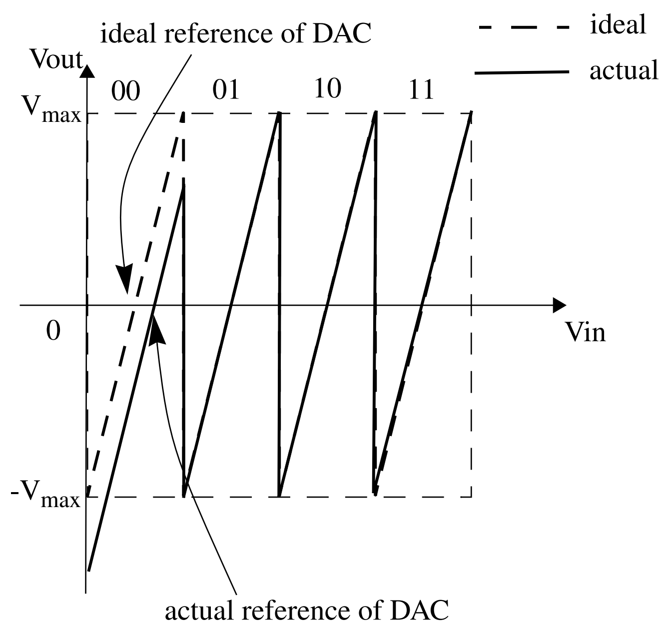


Fig. 21: Transfer Curve of A Pipeline Stage With Nonideal DAC References

4.3.3 Stage Gain Error

The difference of the input analog signal and the output of DAC is amplified by a opamp based switch capacitor circuit. Recall the gain expression in Table 1, the gain is determined by the capacitor ratio and opamp dc gain. Figure 22 shows the transfer curve of a pipelined stage with nonideal residue gain.

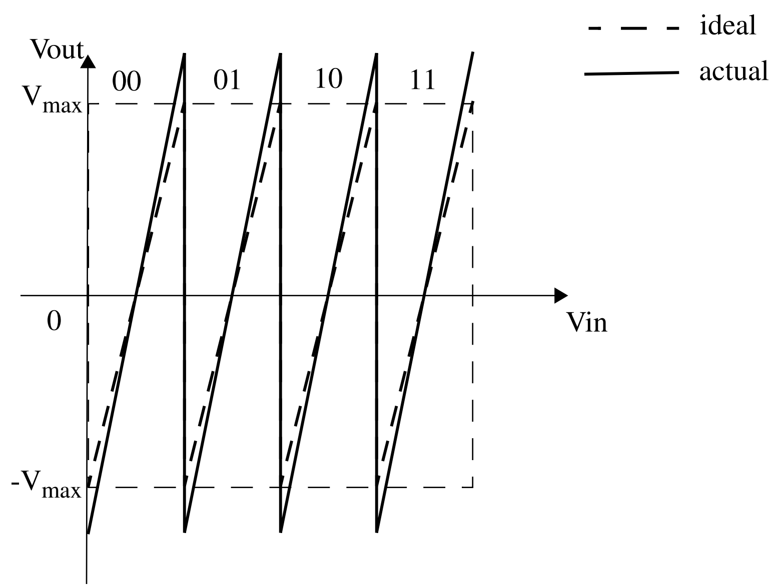
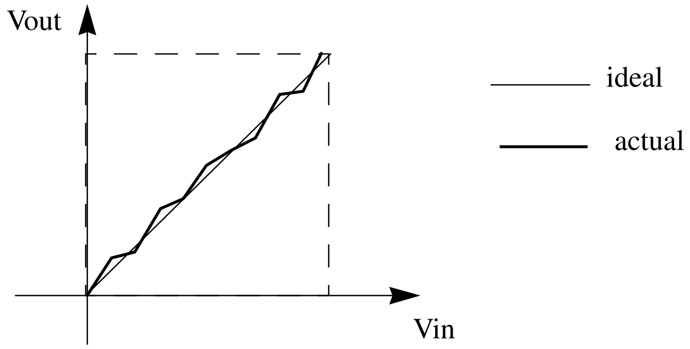


Fig. 22: Transfer Curve of A Pipeline Stage With Nonideal Gain

The effect of opamp dc gain can be neglected if the gain is large enough. Capacitor mismatch is the major source of the gain error. If the stage gain are mismatched in the pipeline, the overall transfer curve is a zigzag curve instead of a straight line in the ideal case.



The capacitor mismatch causes nonideal DAC reference levels and nonideal stage gain. Figure 23 shows the combined effect of them. We can demonstrate this by the simplest special case. Assuming a shared feedback capacitor C_F is used, the sampling capacitance is C_S, and $C_S=(1+\alpha)C_F$, the comparator threshold is ideal, then

$$V_{out} = \left(1+\frac{C_S}{C_F}\right)\left(V_{in}-\frac{C_F}{C_S+C_F}V_{ref}\right) \quad \text{(Eq 10)}$$

$$V_{out} = (2+\alpha)V_{in}-(1+\alpha)V_{ref} \quad \text{(Eq 11)}$$

$V_{out}=0$ when $V_{in}=V_{ref}(1+\alpha)/(2+\alpha)$ and the gain of the transfer function is $2+\alpha$.

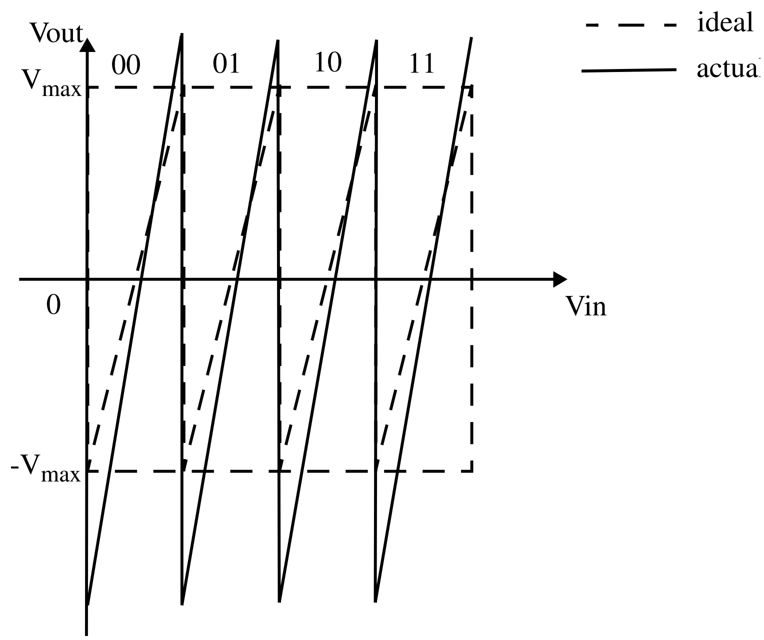


Fig. 23: Transfer Curve of A Pipeline Stage With Capacitor Mismatch

5

Calibration Techniques For Parallel Pipeline A/D

In the previous chapter we discussed various error sources of a parallel pipelined A/D system. If the error is systematic, calibration techniques can be used to improve the linearity and SNR of the system. We will discuss several calibration techniques in this chapter.

5.1 Digital Error Correction Techniques

Comparator threshold offset and nonideal DAC reference levels result in out-of-range transfer curve as shown in Figure 20 and Figure 21. Some output codes are missing when the input of the current stage or output of the previous stage is out of range. One way to deal with this problem is to assign codes for the voltage level outside the range. However, for the implementation of pipelines A/D, it's simpler to reduce the gain and keep the voltage range constant. Digital error correction [13] is the name of the calibration techniques that reduce the gain and keep the voltage range constant with modified coding.

We use a 2-bit/stage pipeline A/D to illustrate how this technique works. Figure 24 shows the comparison of the original coding and digital error correction with modified coding. Table 2 shows the comparison of the DC characteristics between the two. Notice there are four possible codes from each stage in the original coding whereas only three possible codes in the digital error correction case. To get the output code of the whole pipeline A/D, original coding just combine the codes from each stage one next to the other. Five stages are needed for 10-bit output code. The largest code is 1111111111. If digital error correction is applied, the output code is obtained by overlapping one bit between neighboring stages. Nine stages are needed for 10-bit output code. The largest output code is 1111111110.

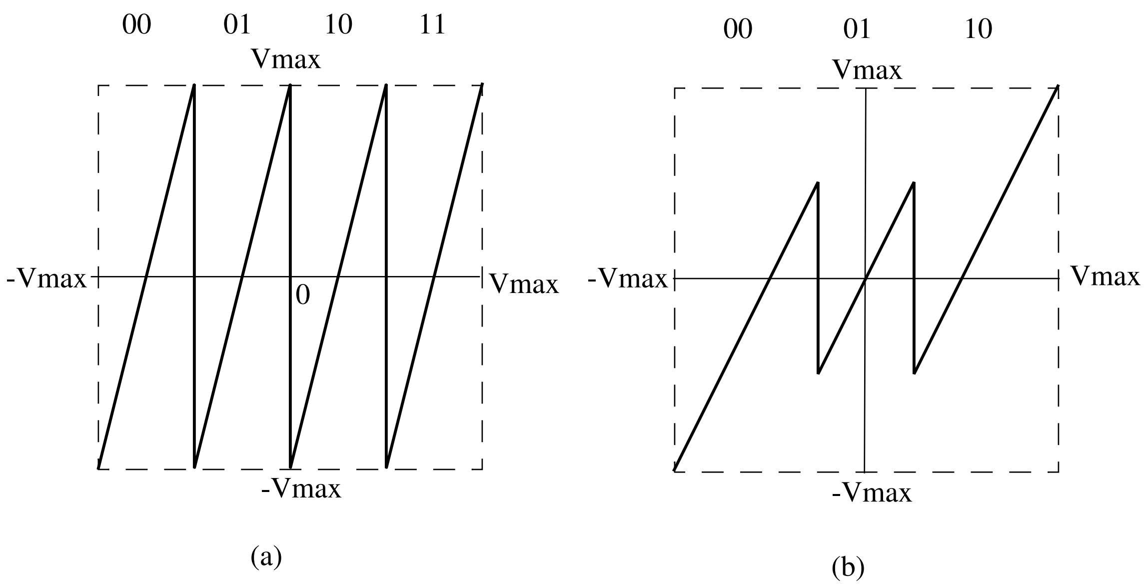


Fig. 24: Transfer Curve of A 2-bit Stage With (a)Original Coding (b)Digital Error Correction With Modified Coding

We miss one largest code by doing digital error correction. But there are many benefits in doing digital error correction. Figure 25 shows the effect of comparator threshold offset and nonideal DAC reference level with digital error correction. From the plot we see that the maximum offset we can tolerate is 1/4 V_{max}. Without the digital error correction, the accuracy of the comparator in the nth stage is required to be $2^{-(n+1)}$. The requirement of the accuracy of the comparators thus is greatly relaxed, hence a dynamic comparator can be used and the power is greatly reduced in expense of digital shifters and adders.

	original coding	digital error correction with modified coding
input range	[-1,1]	[-1,1]
flash ADC threshold levels	-1/2, 0, 1/2	-1/4, 1/4
DAC reference levels	-3/4, -1/4, 1/4, 3/4	-1/2, 1/2
digital codes	00, 01, 10, 11	00, 01, 10
stage gain	4	2
bits/stage	2	2

Table 2: Comparison of original coding and digital error correction with modified coding

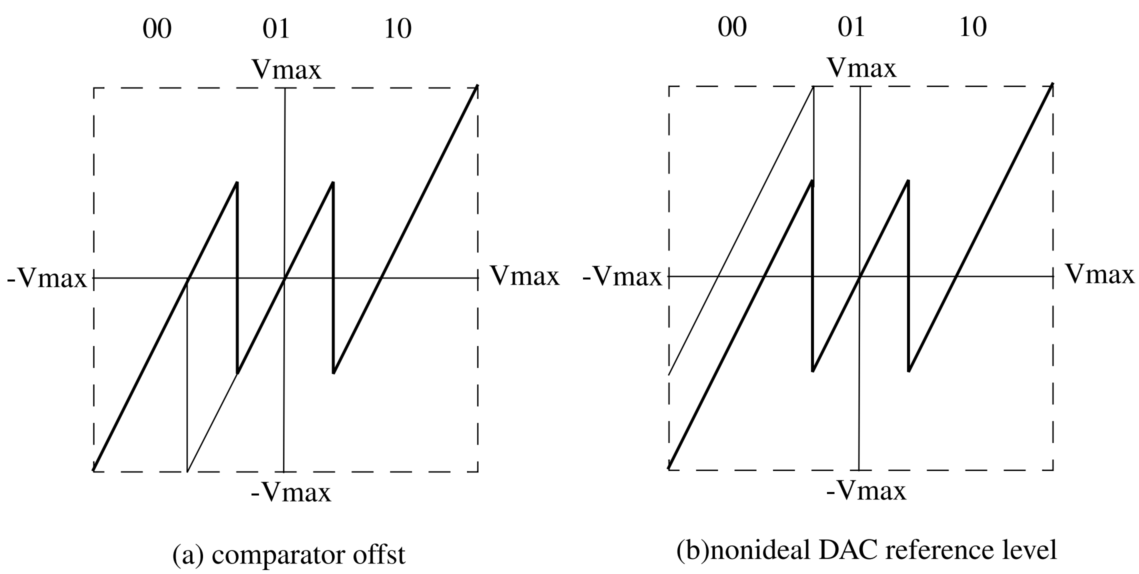


Fig. 25: Effect of Comparator Threshold Offset And Nonideal DAC Reference Level With Digital Error Correction

5.2 Digital Gain Calibration Technique

In an opmap based switch capacitor implementation of a pipeline stage, the function of DAC and amplifying the residue are done by the capacitor array and the opamp. Capacitor mismatch and finite opamp DC gain result in nonideal stage gain and nonlinear DAC reference levels in pipeline A/D. If we can somehow measure the DAC reference levels and store them in a memory, we can recover the ideal output code using the measured reference levels. Digital gain calibration technique [5] is developed based on this idea.

We use a 9-bit pipeline A/D(8 stages, 2-bits/stage with digital error correction) to illustrate how this technique works. Assuming the ideal transfer curve of each stage is as in Figure 24(b), then the transfer function can be written as:

$$V_{out} = \left(1 + \frac{C_S}{C_F}\right)(V_{in} - V_{DAC}) \qquad \text{(Eq 12)}$$

where

$$V_{DAC} = -\frac{C_F}{C_S + C_F} V_{ref} \qquad \text{(Eq 13)}$$

V_{ref} is a DC reference voltage for DAC, usually shared by all the stages to minimize the mismatch. The effective DAC reference level is V_{DAC}, depending on the capacitor value. Ideally we combine the digital output codes of each stage by overlapping 1 bit between the neighboring stages. For example, if each stage outputs 01, then the system output is 011111111, as shown below.

```
0 1                 stage 0
  0 1               stage 1
    0 1             stage 2
      0 1           stage 3
        0 1         stage 4
          0 1       stage 5
            0 1     stage 6
              0 1   stage 7
-----------------
0 1 1 1 1 1 1 1 1
```

Equivalently, we can assign the weight of the code of each stage and do the direct summation for the system output. More specifically, for stage 7, code 00 has weight 0, code 01 has weight 1, code 10 has weight 2; for stage 6, code 00 has weight 0, code 01 has weight 2, code 10 has weight 4; for stage5, code 00 has weight 0, code 01 has weight 4, code 10 has weight 8, ... For the example above, the system output using the direct addition will be $2^0+2^1+2^2+2^3+2^4+2^5+2^6+2^7=255$, equivalent to binary code 01111111.

If capacitors are mismatched, V_{DAC} is no longer ideal, as shown in Figure 26. The weight of the code of each stage no longer equals to the nominal value as assigned above. We apply digital gain calibration to measure the weight of the code of each stage.

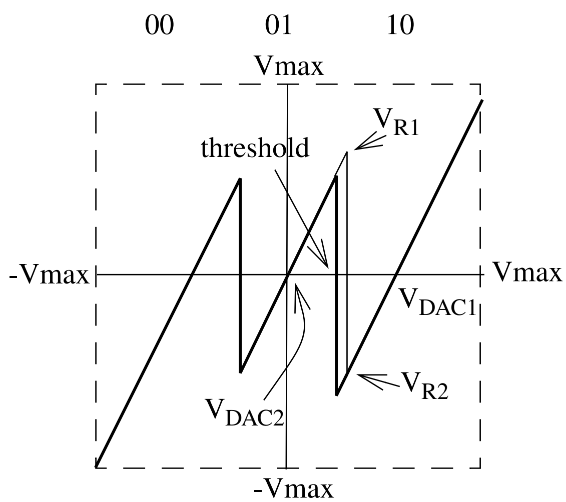


Fig. 26: Transfer Curve of A Pipeline Stage With Digital Gain Calibration

Assuming we are calibrating stage i, the steps are as following:

(1) Let input of stage i to be slightly different from the threshold voltage.

(2) Force the flash ADC to output code 01 and get the output residue V_{R1}.

(3) Use stage i+1, ..., 7 to digitize V_{R1} and get code C_1.

(4) Let the input to be the same as (1), force the flash ADC to output code 10 and get the output residue V_{R2}.

(5) Use stage i+1, ..., 7 to digitize V_{R2} and get code C_2.

(6) Calculate the difference of C_1 and C_2 and store in memory. From the curve we get:

$$V_{R1} - V_{R2} = G \times (V_{DAC1} - V_{DAC2}) \tag{Eq 14}$$

where G is the stage gain. If the code representations of the V_{DAC1} and V_{DAC2} are C_{DAC1} and C_{DAC2}, then

$$C_1 - C_2 = G \times (V_{DAC1} - V_{DAC2}) = G \times \frac{(C_{DAC1} - C_{DAC2})}{G} = C_{DAC1} - C_{DAC2} \tag{Eq 15}$$

So from the measured code C_1 and C_2 we know the difference of the DAC reference codes.

(7) Repeat (1) to (6) for a different threshold voltage for this stage and get the difference of the DAC reference codes C_{DAC1}-C_{DAC0}.

(8) Notice there are three DAC reference levels in one stage. We only measure the difference of them. It turns out that we can arbitrary assign one reference levels and use the two measured differences to calculate the actual references without loosing linearity of the system. Here we assign the code of DAC reference of zero to be 2^{7-i}, i is the stage number.

The complete calibration is done by calibrating from the tail stage (stage 7) to the front stage (stage 0) of the pipeline. The accuracy requirement of the stages is loosen toward the end. It turns out that usually we only need to calibrate the front few stages to improve the DNL of the system substantially. Now we store the calibrated code representations of the DAC references in a memory cell. Later when we digitize the analog input, the flash ADC of each stage output the coarse code functions as an address code to the memory cell and look for the appropriate code representation of the DAC reference of that stage. The search result of each stage is added together to get the final output code. The DNL of the A/D system is greatly improved in expense of digital memories and adders.

5.3 Channel Gain Calibration Techniques

Digital gain calibration improves the DNL of each individual channel. But the linearization process degrades the gain matching between the channels. Figure 27 shows the effect of digital gain calibration on two parallel channels.

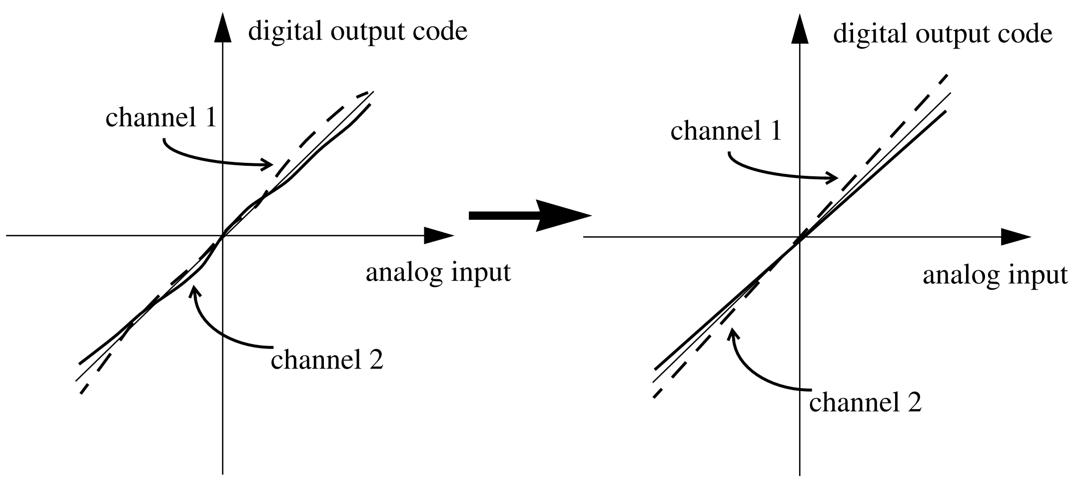


Fig. 27: Effect of Digital Gain Calibration

One solution to this problem is to use N-1 digital multipliers to compensate for the gain mismatch between the channels, using one channel as reference. But a 10-bit multiplier is costly and slow for 200MHz A/D. Usually the gain mismatch is small enough that we can use a N-1 3-bit multipliers to compensate approximately. For example, if channel 1 has gain g_1 and channel 2 has gain g_2, the mismatch is Δ, e.g., $g_2=g_1/(1+\Delta)$, we use channel 1 as reference, then we multiply the output code of channel 2 with $(1+\Delta)$ to compensate for the mismatch:

$$C_2 = (C_{2,\,msb} + C_{2,\,lsb})(1 + \Delta) \qquad \text{(Eq 16)}$$

$$C_2 = C_{2,\,msb} + C_{2,\,lsb} + C_{2,\,msb} \cdot \Delta + C_{2,\,lsb} \cdot \Delta \qquad \text{(Eq 17)}$$

The last term $C_{2,lsb}\Delta$ is small and can be neglected. The only multiplication is $C_{2,msb}\Delta$. This can be done by a 3-bit multiplier.

The other possible solution is to use a look up table and get rid of multiplication completely. The MSB of output code of each channel can be used as address word to a memory cell. The multiplication results $C_{2,msb}\Delta$ are calculated first and stored in the memory cell.

With channel gain calibration and digital gain calibration, the DNL of the system is greatly improved.

5.4 Channel Offset Cancellation Techniques

Channel offset can be calibrated by N-1 adders, using one channel as reference. The procedure is simple. First apply DC ground signal to all the channels and obtain the output code of each channel. Then measure the differences of them and store the differences in a memory. The calibrated system output codes are obtained by compensating the channel offsets using the values stored in memory.

5.5 Calibration Order Issue

The order of applying all the calibration techniques above is important to the performance of the system. When we discuss the calibration techniques above, we assume everything is ideal except the one we are calibrating. This is not true in the real implementation. Take this into consideration, we need to do the calibration in a nested way. More specifically, the sequence of calibration is:

(1) Apply digital gain calibration on each channel. The residue of stage i is digitized using the stage i+1 to the end by overlapping 1 bit of neighboring stages (digital error correction).

(2) Apply channel gain calibration on the N parallel channels, using the stored code representation of the DAC references from digital gain calibration.

(3) Apply channel offset calibration, taking the differences of the codes calibrated by digital error correction, digital gain calibration, and channel gain calibration.

Notice that above calibration techniques do not change circuit elements. Instead, they store the mismatch information in a memory. So the basic A/D is the same as uncalibrated A/D. But when we read the output code, we search the look up table in the memory for the right DAC reference level, gain mismatch factor, and offset factor.

6
Simulation Results

In previous chapters, we discussed the parallel pipelined A/D architecture and the nonidealities associated with it. We also discussed various techniques used to calibrate those nonidealities. In this chapter, we will present the simulation results of a parallel pipelined broadband system with calibration techniques applied.

6.1 S/H Simulation Results

As mentioned in chapter 3, for better matching between channels and larger feedback factor hence faster and more accurate settling, we choose the configuration shown in Figure 28 for the S/H block in front of each channel of the parallel pipeline A/D system. A two-phase nonoverlapping clock scheme is used, where phase Φ_1 has three different edges, one a slight delay of the other, as shown in Figure 28. Φ_{1-} releases the common mode node. Φ_1 samples the data. Φ_{1+} disconnects input with the S/H. Φ_2 transfer the data to output nodes. A multiplexor is used at the output to select the output of four channels at $4xF_c$ rate, where F_c is the channel sampling rate. A cascode stage with a low gain, wideband pre-amplifier (Figure 29) is used to implement the opamp. Only NMOS transistors are in the signal path and the loading capacitance is also the compensation capacitance. This configuration has high gain and high speed. Switches are implemented with NMOS only switches with the gate drive boosted to 5V.

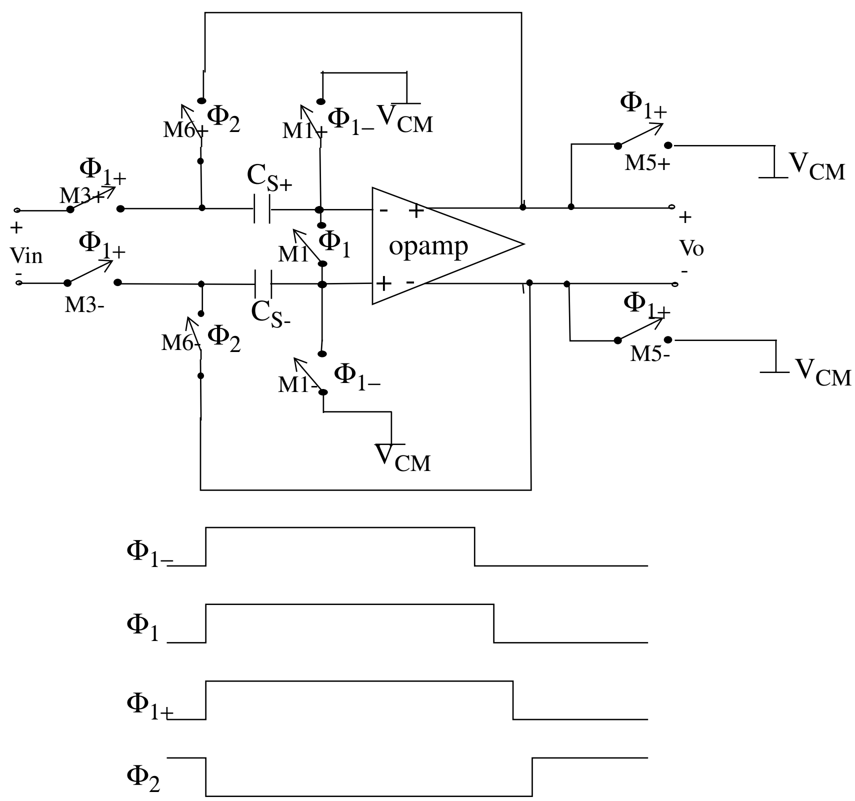


Fig. 28: S/H Configuration

Figure 30 shows the SPICE simulation of the parallel S/H. The first four curves are the output waveforms of the four channels. The last curve is the multiplexed output of the four channels. The input frequency is 95MHz and the system sampling frequency is 200MHz(channel sampling rate is 50MHz).

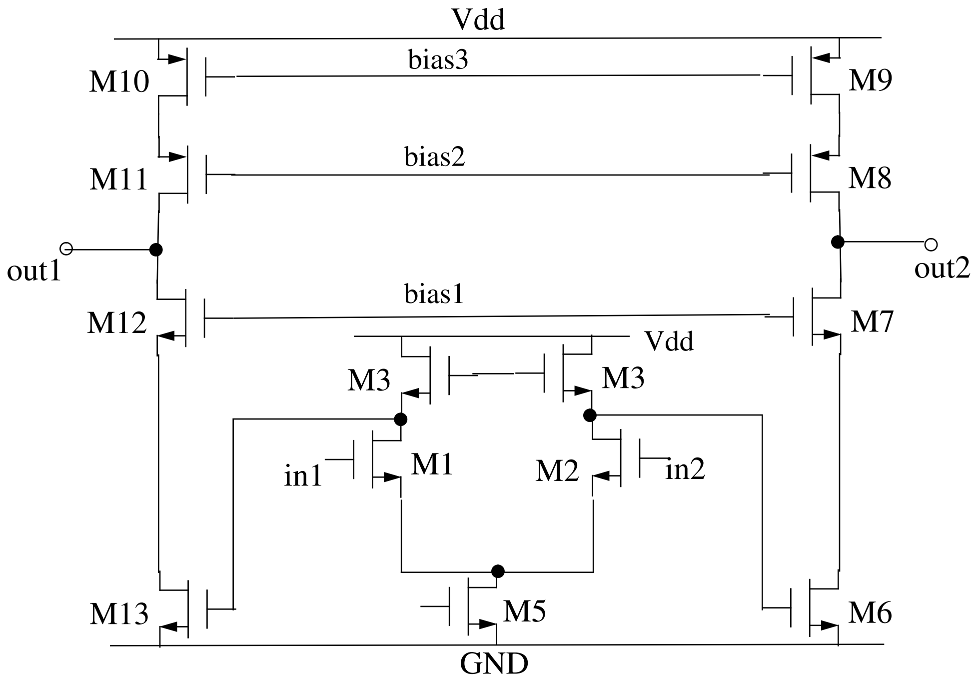


Fig. 29: High-speed High-gain Opamp Configuration

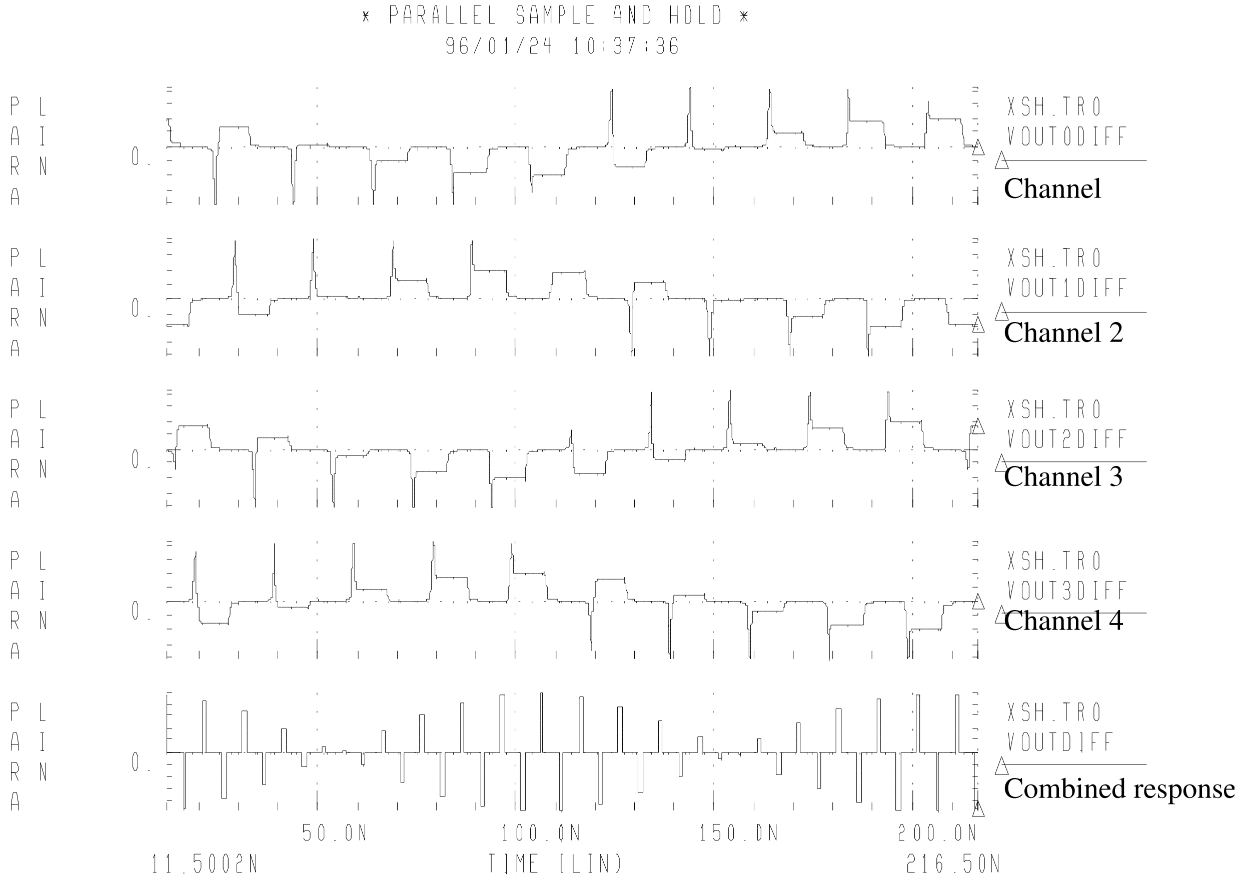


Fig. 30: Parallel Channel Output of A 4-channel A/D S/H Blocks

Figure 31 shows the output spectrum of the 4-channel S/H. We observe a fundamental component at 95MHz which is the input frequency. The third harmonic component is folded back into the baseband. The equivalent frequency is 3x95MHz-200MHz=85MHz. We also observe in the output spectrum a compoent at that frequency. Same for the 5th harmonic component at 5x95MHz-400MHz=75MHz. Even order harmonic distortion is small due the differential nature of this S/H configuration. The non-linearity in the NMOS switches and the non-perfect settling of the opamp and the non-linearity of the opamp gain are the contributions to these harmonic distrotion.

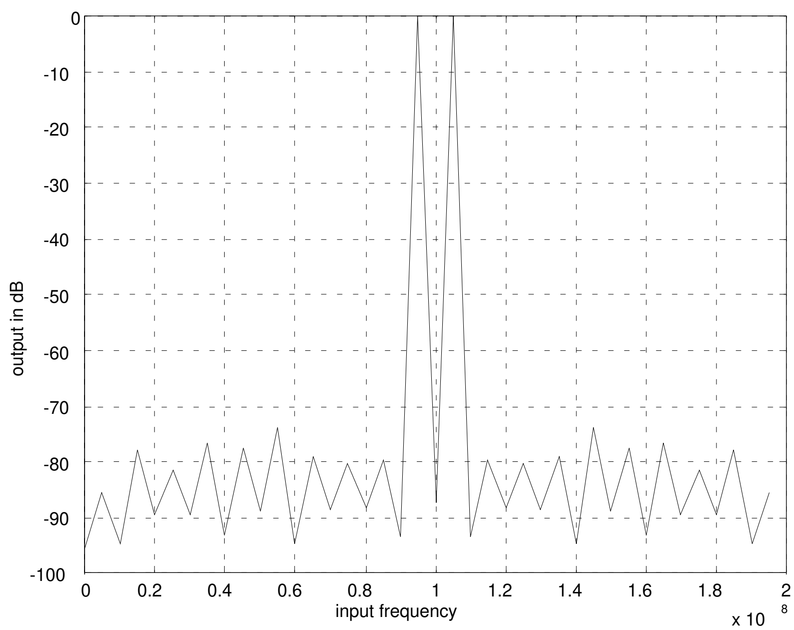


Fig. 31: Output Spectrum of The 4-channel S/H With 95MHz Input

The selected S/H configuration has very little channel gain mismatch associated with the capacitor mismatch. The mismatch between C_{S+} and C_{S-} produces a DC offset in that channel, which can be calibrated out by channel offset cancellation techniques. Conversely, clock jitter is a more severe problem of this S/H circuit since the input is varying close to Nyquist rate of the system. A random number generator written in C code is used to model the clock jitter (Appendix B), where jitter equals the standard deviation used in the random number generator. In SPICE simulation, edges of clock Φ_{1-},Φ_1,Φ_+, and Φ_2 are modulated by the jitter. Jitter on the sampling instant, namely, the falling edge of Φ_1, is the most crucial to the performance of the S/H. The SNR of the entire parallel pipeline system is limited by the S/H block.

Figure 32 shows the SNR of the system vs. clock jitter for three cases, 4-channel with 95MHz input, 8-channel with 95MHz input, and 8-channel with 190MHz input. For 8-channel A/D, the channel sampling rate is still 50MHz, but the system sampling rate is 8x50MHz=400MHz. We observe that when the input frequency is higher, the performance becomes more dependent on the clock jitter.

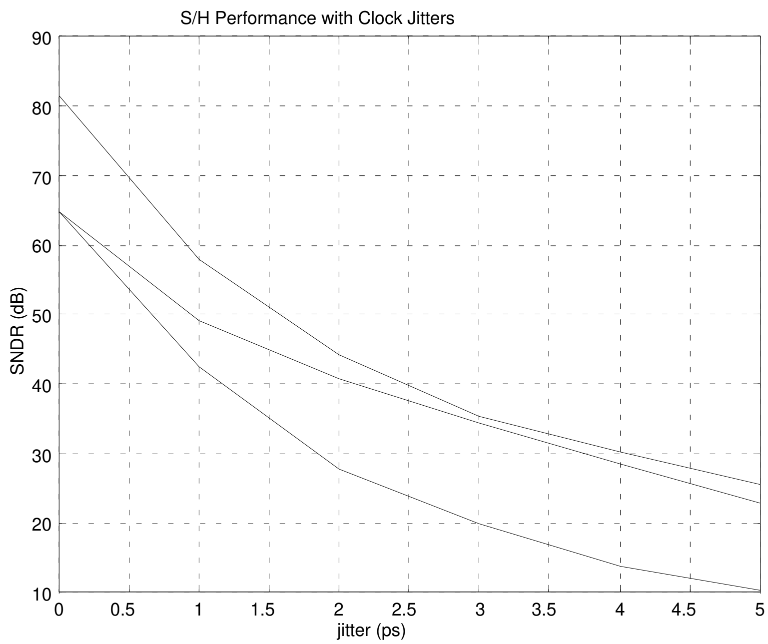


Fig. 32: SNR vs. Clock Jitters For Parallel S/H

6.2 Parallel Pipeline Modeling And Simulation Results

A C program is written to model all the non-idealities mentioned in the previous chapters, including:

1) Capacitor mismatch which causes the channel offset, channel gain mismatch, non-ideal DAC reference levels, and stage gain error.

2) Non-ideal comparator threshold.

3) Clock jitter and skew.

4) Opamp finite gain, offset, and settling time constant.

The multiplexed S/H data simulated with SPICE is used as the input data of the pipeline stages.

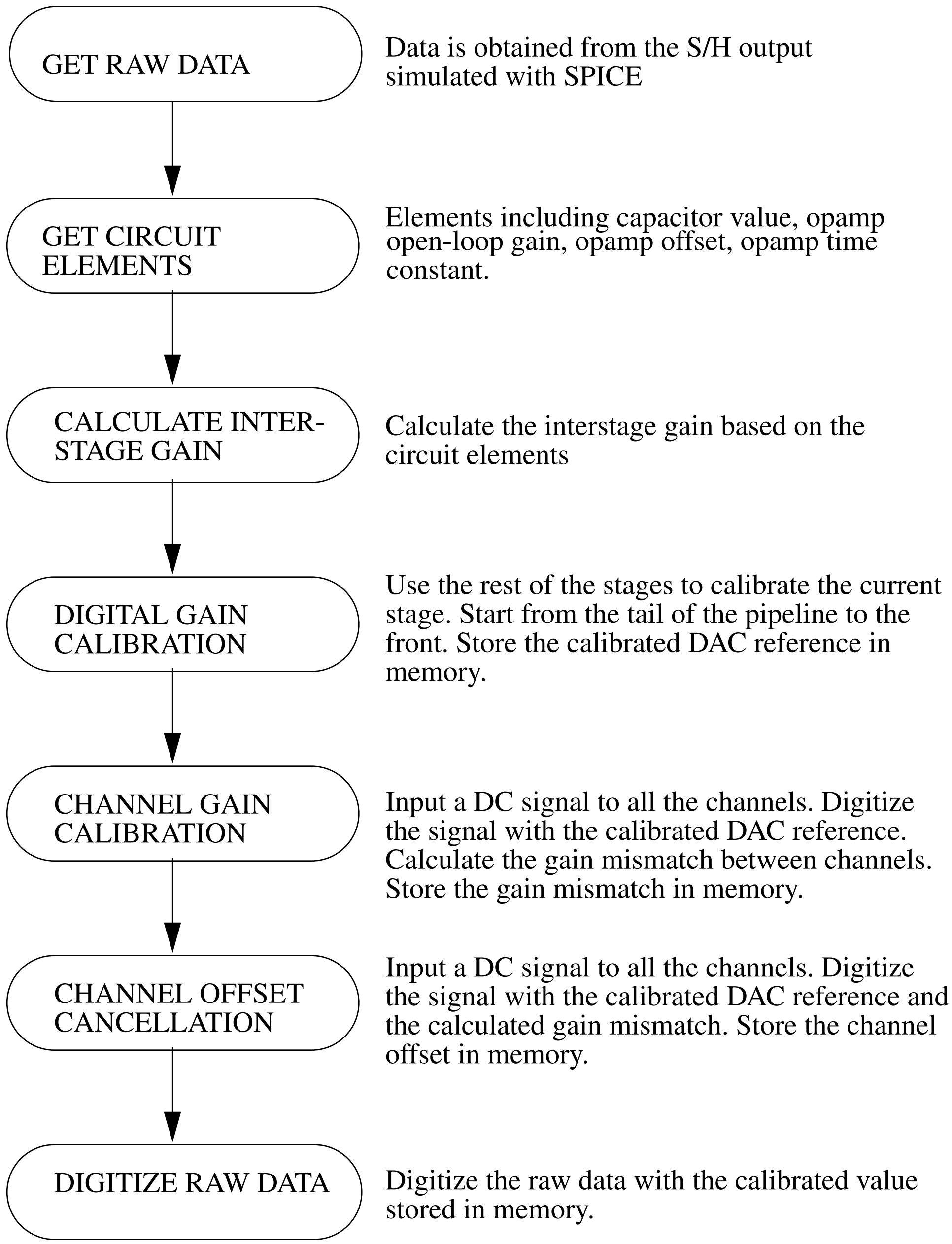


Fig. 33: Flow Diagram of Simulation Modeling

Figure 33 shows the flow diagram of the simulation modeling. Digital error correction is imbedded in each digitization operation because the output code is obtained by overlapping one bit between neighboring stages. The C code is listed in Appendix A.

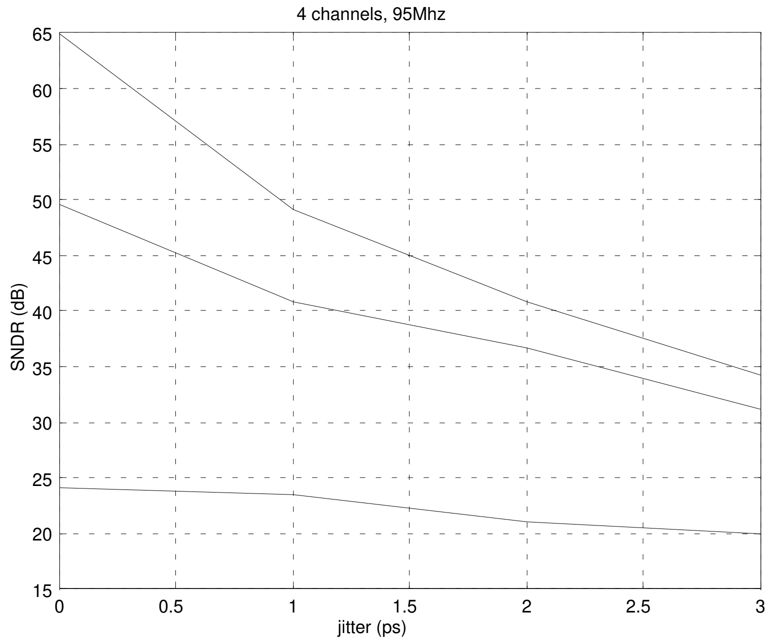


Fig. 34: SNR vs. Clock Jitter for a 4-channel ADC, f_{in}=95MHz

Figure 34 shows the comparison of the performance (total SNDR) of the 4-channel parallel S/H blocks alone, 4-channel parallel S/H with 9 pipeline stages with and without calibration for the 95MHz input. We loose about 10dB from S/H to the entire system with pipeline stages at 1ps jitter because of the non-idealities in the pipelined stages. We see about 15dB improvement of the SNDR at 1ps jitter when the calibration techniques are applied.

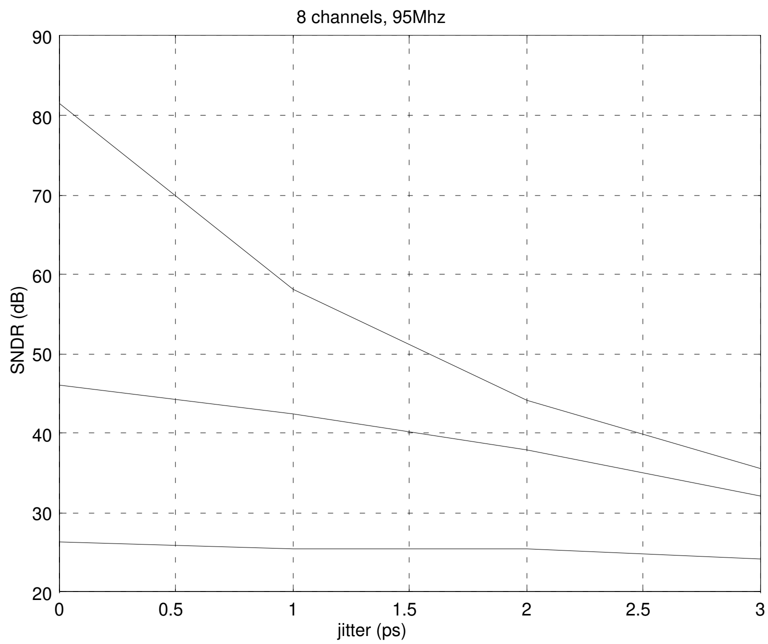


Fig. 35: SNR vs. Clock Jitter for a 8-channel ADC, fin=95MHz

Figure 35 shows the comparison of the performance (total SNDR) of the 8-channel parallel S/H blocks alone, 8-channel parallel S/H with 9 pipeline stages with and without calibration for the 95MHz input. We observe that 8-channel case has a slightly higher SNDR than the 4-channel case for the same input frequency. This is due to fact that the total error energy is approximately the same, but is spread over more channels. Therefore, more channels imply a larger mismatch standard deviation can be tolerated for a given system SNDR.

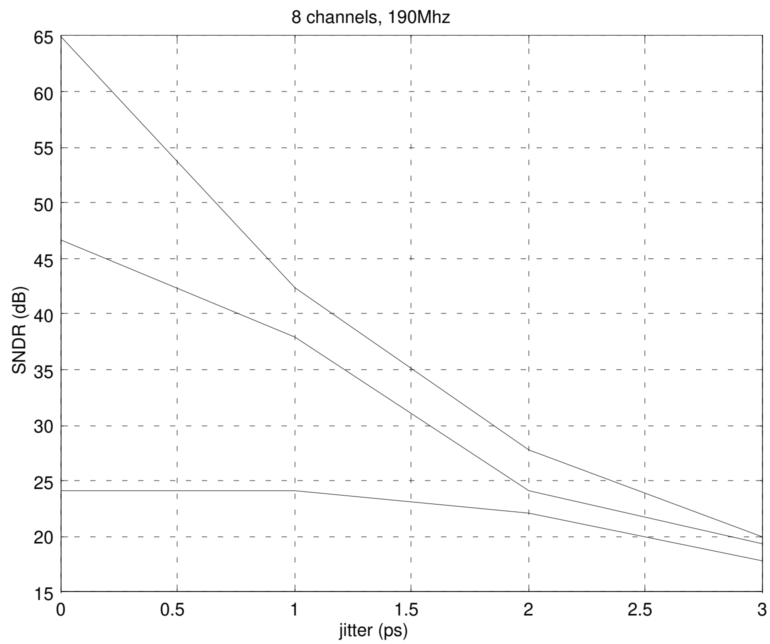


Fig. 36: SNR vs. Clock Jitter for a 8-channel ADC, fin=190MHz

Figure 36 shows the comparison of the performance (total SNDR) of the 8-channel parallel S/H blocks alone, 8-channel parallel S/H with pipeline stages with and without calibration for the 190MHz input. We observe that the slope of the curve, i.e., the dependence of the performance on clock jitter, is larger than both 4-channel and 8-channel cases wtih 95MHz input. When the input frequency is high, the performance is limited by the clock jitters. Calibration techniques do not have significant impact on the system performance when the jitter is larger than 2ps. The problem comes down to how to design a low jitter clock generator and distributing network.

6.3 Conclusion

So far we have discussed various A/D architectures, and conclude that for extreme high speed (> 100MHz) and relatively high resolution (10-bit) applications, parallel pipeline is the most promising architecture. Capacitor mismatch can be calibrated digitally so that channel offset, channel gain mismatch, nonideal DAC reference levels and stage gain error can be reduced dramatically. The calibration techniques include digital error correction, digital gain calibration, channel gain calibration and channel offset cancellation. Nonideal comparator threshold can be calibrated by digital error correction. The circuit configuration as shown in Figure 28 is used to minimize the nonidealities in S/H. The other nonidealities, like clock jitter and thermal noise cannot be calibrated because they are random errors. Careful design of the clock distribution network is required to reduce the jitter to the minimum.

Based on the analysis and simulation so far, we conclude that a 10-bit, 200MHz 4-channel parallel pipelined ADC in 0.6μCMOS technology is feasible with a low jitter(<1ps) clock and the calibration techniques, namely, digital error correction, digital gain calibration, channel gain calibration and channel offset cancellation. Generating such low jitter clock waveform is the greatest challenge in very high speed parallel pipeline ADCs.

Appendix A

```
#include <stdio.h>
#include <math.h>
main()
{
  int   modcode(); /* channel output code with digital error correction, a function */
  void  atod();/* pipeline atod conversion */
  void  oscancel();/* channel offset cancellation */
  void  gaincal();/* digital gain calibration */
  void  changaincal();/* changain calibration */
  intchannels = 4;/* number of channels */
  int   stages = 12;    /* number of stages */
  int i,j,k;/* loop control */
  int   code[12][4];/* digital code output of coarse adc */
  int   codeoffset[4];  /* channel offset correction code, generated by digital offset
cancellation */
  double changain[4];
  int   outcode;/* final output code */
  double out;/* output of each stage */
  double pi = 3.1415122654;

/****** sample/hold block ******/
  int    numsamples=40;/* number of samples */
  double shoffset[4];/* offset from sample and hold */
```

```
  double sample[160];/* sampled data */
  double fs=200.0e6;/* total sampling rate */
  double ts=4/fs;/* channel sampling period */
  double dummy;

/****** coarse ADC block *******/
  double compdelay;/* delay between sampling and comparator sampling */
  double compneg[12][4];/* comparator negative threshold , nominal value -0.25 */
  double comppos[12][4];/* comparator positive threshold , nominal value +0.25 */
  double vref[3];/* reference voltage for dac level */
  int calvref[3][12][4]; /* digital calibrated dac level for stage0 in each channel */

/****** opamp block ********/
  double cs[12][4];/* sampling capacitors */
  double cf[12][4];/* feedback capacitors */
  double copamp[12][4];/* opamp input capacitors */
  double feedback;/* feedback factor determined by cs,cf, and copamp */
  double opampgain[12][4];/* opamp open loop gain */
  double opampoffset[12][4];/* opamp offset values */
  double opamptau[12][4];/* opamp time constant */
  double gain[12][4];/* interstage gain */
  double temp;/* calculation intermedia number */

  FILE *fp_cct;/* circuit data file */
  FILE *fp_sample;/* data from sample/hold */

  fp_cct = fopen("realcct.data", "r");/* open file cct.data */

/************* obtain capacitor values **********/
  for (j=0; j<channels; j++)
   {
    for (i=0; i<stages; i++)
     {
      fscanf(fp_cct, "%lf %lf\n", &cs[i][j], &cf[i][j]);
     }
   }

/*** obtain opamp open loop gain, offset values, input capacitance and time constant
```

```
*****/
  for (j=0; j<channels; j++)
   {
     for (i=0; i<stages; i++)
      {
        fscanf(fp_cct, "%lf %lf %lf %lf\n", &opampgain[i][j], &opampoffset[i][j],
&copamp[i][j], &opamptau[i][j]);
      }
    }

/************* obtain dac levels ****************/
  for (i=0; i<3; i++)
   {
     fscanf(fp_cct, "%lf\n", &vref[i]);
   }
  for (k=0; k<channels; k++)
   {
     for (i=0; i<stages; i++)
      {
        fscanf(fp_cct, "%d %d %d\n",
&calvref[0][i][k],&calvref[1][i][k],&calvref[2][i][k]);
      }
   }

/************* obtain s/h offset and channel gains ****************/
  for (i=0; i<channels; i++)
   {
     fscanf(fp_cct, "%lf %lf\n", &shoffset[i],&changain[i]);
   }

/************* obtain comparator delay and threshold  ****************/
  for (j=0; j<channels; j++)
   {
     for (i=0; i<stages; i++)
      {
        fscanf(fp_cct, "%lf %lf\n", &compneg[i][j], &comppos[i][j]);
      }
   }
```

```
  fclose(fp_cct);

/************** obtain sampled data *************************/
  fp_sample = fopen("sample.data", "r");/* open file sample.data */
  for (i=0;i<numsamples;i++)
   {
/*
     fscanf(fp_sample,"%lf %lf\n",&dummy,&sample[i]);
     sample[i]=sin(2.0*pi*125.0*i/200.0);
     sample[i]=((2.0*(i/4))/numsamples)*4.0-1.0;
     sample[i]=(2.0*i)/numsamples-1.0;
     sample[i]=0.5;
*/
     fscanf(fp_sample,"%lf %lf\n",&dummy,&sample[i]);
   }
  fclose(fp_sample);

/************* calculate interstage gain *******/
  for (j=0; j<channels; j++)
   {
     for (i=0; i<stages; i++)
      {
        feedback = cf[i][j]/(cf[i][j]+cs[i][j]+copamp[i][j]);
        gain[i][j]=(1+cs[i][j]/cf[i][j])*(1-exp(-0.5*ts/opamptau[i][j]));
        gain[i][j]=gain[i][j]/(1+1/(opampgain[i][j]*feedback));
      }
   }

/************* multiple coarse adc conversion ***********/
#ifdef cal

  gaincal(stages,code,calvref,cs,cf,copamp,opampoffset,gain,compneg,comppos,vref);


changaincal(stages,changain,code,calvref,cs,cf,copamp,opampoffset,gain,compneg,co
mppos,vref);
```

```
oscancel(stages,shoffset,codeoffset,changain,code,calvref,cs,cf,copamp,opampoffset,
gain,compneg,comppos,vref);

#endif

  for (j=0; j<numsamples/channels; j++)
   {
     for (k=0; k<channels; k++)
      {
        temp=shoffset[k]+sample[channels*j+k];
        atod(temp,0,stages-1,k,code,calvref,cs,cf,copamp,opampoffset,gain,compneg,
comppos,vref);
        outcode = modcode(0,stages-1,k,code,calvref)/changain[k]-codeoffset[k];
        printf("%d\n", outcode/8);
      } /* end loop k */
   } /* end loop j */

  return(0);
} /* end main */

/************* digital gain calibration  ********/
  void gaincal(stages,code,calvref,cs,cf,copamp,opampoffset,gain,compneg,comppos,
vref)
  int stages;
  int   code[12][4],calvref[3][12][4];
  double  cs[12][4],cf[12][4],copamp[12][4],opampoffset[12][4];
  double  gain[12][4], compneg[12][4],comppos[12][4],vref[3];
{
  int calstage;
  int j,k;
  double temp1,temp2,out,vtest;
  int weight[4];

  for (calstage=3; calstage>=0; calstage=calstage-1)
   {
     for (k=0; k<4; k++)
      {
```

```
        temp1 = cs[calstage][k]/(cs[calstage][k]+cf[calstage][k]);
        temp2 = (cs[calstage][k]+cf[calstage][k]+copamp[calstage][k])/
(cs[calstage][k]+cf[calstage][k]);

        vtest = -0.25;
       /* first measurement */
        out = gain[calstage][k]*(vtest-temp1*vref[0]-temp2*opampoffset[calstage][k]);
        atod(out,calstage+1,stages-1,k,code,calvref,cs,cf,copamp,opampoffset,gain,
compneg,comppos,vref);
        weight[0] = modcode(calstage+1,stages-1,k,code,calvref);

      /* second measurement */
        out = gain[calstage][k]*(vtest-temp1*vref[1]-temp2*opampoffset[calstage][k]);
        atod(out,calstage+1,stages-1,k,code,calvref,cs,cf,copamp,opampoffset,gain,
compneg,comppos,vref);
        weight[1] = modcode(calstage+1,stages-1,k,code,calvref);

        calvref[0][calstage][k] = calvref[1][calstage][k]-(weight[0]-weight[1]);

        vtest = 0.25;
       /* first measurement */
        out = gain[calstage][k]*(vtest-temp1*vref[1]-temp2*opampoffset[calstage][k]);
        atod(out,calstage+1,stages-1,k,code,calvref,cs,cf,copamp,opampoffset,gain,
compneg,comppos,vref);
        weight[2] = modcode(calstage+1,stages-1,k,code,calvref);

      /* second measurement */
        out = gain[calstage][k]*(vtest-temp1*vref[2]-temp2*opampoffset[calstage][k]);
        atod(out,calstage+1,stages-1,k,code,calvref,cs,cf,copamp,opampoffset,gain,
compneg,comppos,vref);
        weight[3] = modcode(calstage+1,stages-1,k,code,calvref);

        calvref[2][calstage][k] = calvref[1][calstage][k]+(weight[2]-weight[3]);

      } /* end loop k */
   } /* end loop clastage */
#ifdef debug
    for (k=0;k<4;k++)
```

```
      {
       for (j=0;j<stages;j++)
        {
          printf("%d %d %d \n",calvref[0][j][k],calvref[1][j][k],calvref[2][j][k]);
        }
       }
#endif
} /* end gaincal */

/************* channel gain calibration ***************/
  void  changaincal(stages,changain,code,calvref,cs,cf,copamp,opampoffset,
gain,compneg,comppos,vref)

  int stages;
  double changain[4];
  int   code[12][4],calvref[3][12][4];
  double  cs[12][4],cf[12][4],copamp[12][4],opampoffset[12][4];
  double  gain[12][4], compneg[12][4],comppos[12][4],vref[3];
{
  int   k,code1,code2;
  double temp;

  for (k=0; k<4; k++)
   {
     temp = 0.5;
     atod(temp,0,stages-1,k,code,calvref,cs,cf,copamp,opampoffset,
gain,compneg,comppos,vref);
     code1=modcode(0,stages-1,k,code,calvref);
     temp = -0.5;
     atod(temp,0,stages-1,k,code,calvref,cs,cf,copamp,opampoffset,
gain,compneg,comppos,vref);
     code2=modcode(0,stages-1,k,code,calvref);
     changain[k]=code1-code2;
   } /* end for k */
  changain[1]=changain[1]/changain[0];
  changain[2]=changain[2]/changain[0];
  changain[3]=changain[3]/changain[0];
  changain[0]=1.0;
```

```
#ifdef debug
  printf("%lf %lf %lf %lf \n",changain[0],changain[1],changain[2],changain[3]);
#endif
}     /* end changaincal */


/************* digital offset cancellation **************/
  void  oscancel(stages,shoffset,codeoffset,changain,code,calvref,cs,cf,
copamp,opampoffset,gain,compneg,comppos,vref)
  int  stages,codeoffset[4];
  double shoffset[4];
  double changain[4];
  int   code[12][4],calvref[3][12][4];
  double  cs[12][4],cf[12][4],copamp[12][4],opampoffset[12][4];
  double  gain[12][4], compneg[12][4],comppos[12][4],vref[3];
{
  int   k,temp1,temp2;
  double temp;

  for (k=0; k<4; k++)
   {
     temp=shoffset[k]+0.5;
     atod(temp,0,stages-1,k,code,calvref,cs,cf,copamp,opampoffset,gain,
compneg,comppos,vref);
     temp1 = modcode(0,stages-1,k,code,calvref)/changain[k];
     temp=shoffset[k]-0.5;
     atod(temp,0,stages-1,k,code,calvref,cs,cf,copamp,opampoffset,gain,
compneg,comppos,vref);
     temp2 = modcode(0,stages-1,k,code,calvref)/changain[k];
     codeoffset[k]=(temp1+temp2)/2;
   } /* end loop k */
  codeoffset[1] = codeoffset[1] - codeoffset[0];
  codeoffset[2] = codeoffset[2] - codeoffset[0];
  codeoffset[3] = codeoffset[3] - codeoffset[0];
  codeoffset[0] = 0;
#ifdef debug
 printf("%d %d %d %d \n", codeoffset[0],codeoffset[1],codeoffset[2],codeoffset[3]);
#endif
```

```
} /* end oscancel */


/************ modified coding with digital error correction *******/
int  modcode(start, stop, k1, code, calvref)
int start;
int stop;
int k1;
int code[12][4];
int calvref[3][12][4];
{
 int i, index,result;
 result = 0 ;
 for (i=start; i<=stop; i++)
  {
    index=code[i][k1];
    result = result + calvref[code[i][k1]][i][k1];
  } /* end loop i */
 return(result);
}

/************* pipeline A to D conversion *************/
  void    atod(input,start,stop,k1,code,calvref,cs,cf,copamp,opampoffset,
gain,compneg,comppos,vref)
  doubleinput;
  intstart,stop,k1;
  int code[12][4],calvref[3][12][4];
  double  cs[12][4],cf[12][4],copamp[12][4],opampoffset[12][4];
  double  gain[12][4], compneg[12][4],comppos[12][4],vref[3];
  {
  inti;
  double  out,temp1,temp2;

  out = input;
  for (i=start; i<=stop; i++)
   {
    temp1 = cs[i][k1]/(cs[i][k1]+cf[i][k1]);
    temp2 = (cs[i][k1]+cf[i][k1]+copamp[i][k1])/(cs[i][k1]+cf[i][k1]);
```

```
    if ( out <= compneg[i][k1] )
     {
       code[i][k1] = 0;
       out = gain[i][k1]*(out-temp1*vref[0]-temp2*opampoffset[i][k1]);
     }
    else if ( (out > compneg[i][k1]) && (out <= comppos[i][k1]) )
     {
       code[i][k1] = 1;
       out = gain[i][k1]*(out-temp1*vref[1]-temp2*opampoffset[i][k1]);
     }
    else
     {
       code[i][k1] = 2;
       out = gain[i][k1]*(out-temp1*vref[2]-temp2*opampoffset[i][k1]);
     }
   } /* end loop i */
} /* end atod */
```

Appendix B

Random Number Generator

```
/*:::::::::::::
  AmathRecipe1.c
  ::::::::::::: */
#include <math.h>
#include "AmathRecipe1.h"
#include <stdio.h>

double ran1(int* idum)
{
   int M1 = 259200;
   int IA1= 7141;
   int IC1= 54773;
   double RM1= (1.0/M1);
   int M2 = 134456;
   int IA2= 8121;
   int IC2= 28411;
   double RM2= (1.0/M2);
   int M3 = 243000;
   int IA3= 4561;
   int IC3= 51349;
```

```
  static long ix1, ix2, ix3;
  static double r[98];
  double temp;
  static int iff=0;
  int j;

  if (*idum<0 || iff==0) {
    iff =1;
    ix1 = (IC1-(*idum)) % M1;
    ix1 = (IA1*ix1+IC1) % M1;
    ix2 = ix1 % M2;
    ix1 = (IA1*ix1+IC1) % M1;
    ix3 = ix1 % M3;
    for (j=1; j<=97; j++) {
      ix1 = (IA1*ix1+IC1) % M1;
      ix2 = (IA2*ix2+IC2) % M2;
      r[j]= (ix1+ix2*RM2)*RM1;
    }
    *idum=1;
  }
  ix1 = (IA1*ix1+IC1) % M1;
  ix2 = (IA2*ix2+IC2) % M2;
  ix3 = (IA3*ix3+IC3) % M3;
  j = 1 + ((97*ix3)/M3);
  if (j>97 || j<1) {
printf("RAN1:  Error this cannot happen.");
exit(1);
}
  temp=r[j];
  r[j] = (ix1+ix2*RM2)*RM1;
  return temp;
}

double gasdev(int *idum)
{
  static int iset=0;
  static double gset;
  double fac,r,v1,v2;
```

```
  if (iset==0) {
    do {
      v1=2.0*ran1(idum)-1.0;
      v2=2.0*ran1(idum)-1.0;
      r=v1*v1+v2*v2;
    } while (r>=1.0);
    fac = sqrt(-2.0*log(r)/r);
    gset = v1*fac;
    iset=1;
    return v2*fac;
  } else {
    iset=0;
    return gset;
  }
}

double grnLimited(double stdev, double max)
{
  static int seed=3;
  double temp;
  do {
    temp = gasdev(&seed);
    temp *= stdev;
  } while(fabs(temp) > max);
  return temp;
}

/*main()
{
double x;
int i=0;
double stdev = 1.0;
double max_abs = 1.0E30;
for (i=0; i<100; i++) {
printf("%d  %lf\n", i, grnLimited(stdev, max_abs));
}
}*/
```

```
/*
:::::::::::::
AmathRecipe1.h
:::::::::::::
*/

#ifndef _AmathRecipe1_H
#define _AmathRecipe1_H

double ran1(int* idum);
double gasdev(int* idum);
double grnLimited(double stdev, double max);

#endif
```

Clock Generator for 4-channel parallel pipeline A/D

```
#include <stdio.h>
#include <math.h>
#include “AmathRecipe1.h”
#include “AmathRecipe1.c”

main()
{
voidclk();
inti;
intchannel, clkindex, cycle;
double time[4][4][800],meastime[4][800];
intpv[4][4][800];

cycle=41;
```

```
/******* channel 0 *************/
printf("Vphi01- phi01- 0 PWL( \n");
clk(0,0,cycle,time,meastime,pv);
for (i=0;i<4*cycle;i++)
 {
   printf("+ %le %d \n",time[0][0][i],pv[0][0][i]);
 }
printf("+ ) \n");

printf("Vphi01 phi01 0 PWL( \n");
clk(0,1,cycle,time,meastime,pv);
for (i=0;i<4*cycle;i++)
 {
   printf("+ %le %d \n",time[0][1][i],pv[0][1][i]);
 }
printf("+ ) \n");

printf("Vphi01+ phi01+ 0 PWL( \n");
clk(0,2,cycle,time,meastime,pv);
for (i=0;i<4*cycle;i++)
 {
   printf("+ %le %d \n",time[0][2][i],pv[0][2][i]);
 }
printf("+ ) \n");

printf("Vphi02 phi02 0 PWL( \n");
clk(0,3,cycle,time,meastime,pv);
for (i=0;i<4*cycle;i++)
 {
   printf("+ %le %d \n",time[0][3][i],pv[0][3][i]);
 }
printf("+ ) \n");

/******* channel 1 *************/
printf("Vphi11- phi11- 0 PWL( \n");
clk(1,0,cycle,time,meastime,pv);
for (i=0;i<4*cycle;i++)
 {
```

```
  printf("+ %le %d \n",time[1][0][i],pv[1][0][i]);
 }
printf("+ ) \n");

printf("Vphi11 phi11 0 PWL( \n");
clk(1,1,cycle,time,meastime,pv);
for (i=0;i<4*cycle;i++)
 {
  printf("+ %le %d \n",time[1][1][i],pv[1][1][i]);
 }
printf("+ ) \n");

printf("Vphi11+ phi11+ 0 PWL( \n");
clk(1,2,cycle,time,meastime,pv);
for (i=0;i<4*cycle;i++)
 {
  printf("+ %le %d \n",time[1][2][i],pv[1][2][i]);
 }
printf("+ ) \n");

printf("Vphi12 phi12 0 PWL( \n");
clk(1,3,cycle,time,meastime,pv);
for (i=0;i<4*cycle;i++)
 {
  printf("+ %le %d \n",time[1][3][i],pv[1][3][i]);
 }
printf("+ ) \n");

/******* channel 2 *************/
printf("Vphi21- phi21- 0 PWL( \n");
clk(2,0,cycle,time,meastime,pv);
for (i=0;i<4*cycle;i++)
 {
  printf("+ %le %d \n",time[2][0][i],pv[2][0][i]);
 }
printf("+ ) \n");

printf("Vphi21 phi21 0 PWL( \n");
```

```
clk(2,1,cycle,time,meastime,pv);
for (i=0;i<4*cycle;i++)
 {
   printf("+ %le %d \n",time[2][1][i],pv[2][1][i]);
 }
printf("+ ) \n");

printf("Vphi21+ phi21+ 0 PWL( \n");
clk(2,2,cycle,time,meastime,pv);
for (i=0;i<4*cycle;i++)
 {
   printf("+ %le %d \n",time[2][2][i],pv[2][2][i]);
 }
printf("+ ) \n");

printf("Vphi22 phi22 0 PWL( \n");
clk(2,3,cycle,time,meastime,pv);
for (i=0;i<4*cycle;i++)
 {
   printf("+ %le %d \n",time[2][3][i],pv[2][3][i]);
 }
printf("+ ) \n");

/******* channel 3 *************/
printf("Vphi31- phi31- 0 PWL( \n");
clk(3,0,cycle,time,meastime,pv);
for (i=0;i<4*cycle;i++)
 {
   printf("+ %le %d \n",time[3][0][i],pv[3][0][i]);
 }
printf("+ ) \n");

printf("Vphi31 phi31 0 PWL( \n");
clk(3,1,cycle,time,meastime,pv);
for (i=0;i<4*cycle;i++)
 {
   printf("+ %le %d \n",time[3][1][i],pv[3][1][i]);
 }
```

```
printf("+ ) \n");

printf("Vphi31+ phi31+ 0 PWL( \n");
clk(3,2,cycle,time,meastime,pv);
for (i=0;i<4*cycle;i++)
 {
   printf("+ %le %d \n",time[3][2][i],pv[3][2][i]);
 }
printf("+ ) \n");

printf("Vphi32 phi32 0 PWL( \n");
clk(3,3,cycle,time,meastime,pv);
for (i=0;i<4*cycle;i++)
 {
   printf("+ %le %d \n",time[3][3][i],pv[3][3][i]);
 }
printf("+ ) \n");

for (i=0; i<cycle; i++)
 {
   for (channel=0; channel<4; channel++)
    {
/*      printf(".meas tran vout%d find par('v(out%d+)-v(out%d-)')
at=%le \n", channel+4*i,channel,channel,meastime[channel][i]); */
    } /* end channel */
 } /* end i */
} /* end main */

/*************** generate clock *****************/
void clk(channel,clkindex,cycle,time,meastime,pv)
intchannel, clkindex, cycle;
doubletime[4][4][800], meastime[4][800];
intpv[4][4][800];

{
inti;
int v1=8, v2=0;
doubletd,jitter;
```

```
double  stdev = 1.0e-12,max_abs = 1.0E30;
double fs=200.0e6,period=4.0/fs;
double tf=0.5e-9,tr=0.5e-9,thigh=7.5e-9,tlow=period-thigh-tf-tr;

td = period*channel/4.0 + 1.0e-9*(clkindex+1);
time[channel][clkindex][0]=0;
if (clkindex != 3)
 {
   pv[channel][clkindex][0]=v1;
   for (i=0; i<cycle; i++)
    {
      jitter=grnLimited(stdev, max_abs);
      time[channel][clkindex][4*i+1]= td+jitter+period*i;
      pv[channel][clkindex][4*i+1]=v1;
      time[channel][clkindex][4*i+2]= td+jitter+period*i+tf;
      pv[channel][clkindex][4*i+2]=v2;
      jitter=grnLimited(stdev, max_abs);
      time[channel][clkindex][4*i+3]= td+jitter+period*i+tf+tlow;
      pv[channel][clkindex][4*i+3]=v2;
      time[channel][clkindex][4*i+4]= td+jitter+period*i+tf+tlow+tr;
      pv[channel][clkindex][4*i+4]=v1;
    } /* end for i */
 } /* if */
else
 {
   pv[channel][clkindex][0]=v2;
   for (i=0; i<cycle; i++)
    {
      jitter=grnLimited(stdev, max_abs);
      time[channel][clkindex][4*i+1]= td+jitter+period*i;
      pv[channel][clkindex][4*i+1]=v2;
      time[channel][clkindex][4*i+2]= td+jitter+period*i+tr;
      pv[channel][clkindex][4*i+2]=v1;
      jitter=grnLimited(stdev, max_abs);
      time[channel][clkindex][4*i+3]= td+jitter+period*i+tr+thigh;
      meastime[channel][i]=time[channel][clkindex][4*i+3];  /* measurement instant */
      pv[channel][clkindex][4*i+3]=v1;
      time[channel][clkindex][4*i+4]= td+jitter+period*i+tr+thigh+tf;
```

```
    pv[channel][clkindex][4*i+4]=v2;
   } /* end for i */
 } /* endif */
} /* end clk */
```

References

[1] T. B. Cho, G. Chien, F. Brianti, and P. R. Gray, "*A Power-Optimized CMOS Baseband Channel Filter and ADC for Cordless Applications,*" *Symposium On VLSI Circuits,* June 1996, in press.

[2] T. B. Cho and P. R. Gray, *"A 10bit, 20MS/s, 35mW Pipeline A/D Converter," Proc. IEEE Custom Integrated Circuits Conference*, May 1994, pp 23.2.1-23.2.4

[3] T. B. Cho and P. R. Gray, *"A 10 b, 20 Msample/s, 35 mW pipeline A/D converter,", IEEE J. Solid-Stage Circuits*, vol.30, pp. 166-172, March 1995

[4] T. B. Cho, *"Low-Power Low-Voltage Analog-to-Digital Conversion Techniques using Pipelined Architectures", Memorandum No. UCB/ERL M95/23*, Electronics Research Laboratory, U. C. Berkeley, April 1995

[5] A. N. Karanicolas, H. S. Lee, and K. L. Barcrania, *"A 15b 1MS/s digitally self-calibrated pipeline ADC", ISSCC Dig. Tech Papers*, Feb. 1993, pp. 60-61

[6] A. N. Karanicolas, Hae-Seung Lee and K. L. Barcrania, *"A 15-b 1-Msample/s digitally self-calibrated pipeline ADC," IEEE J. Solid-Stage Circuits*, vol.28, pp. 1207-1215, Dec. 1993

[7] C. S. G. Conroy, D. W. Cline, and P. R. Gray, *"An 8-bit 85MS/s parallel pipeline A/D converter in 1-μm CMOS", IEEE J. Solid-Stage Circuits*, vol. 28, no. 4, April 1993, pp. 447-454

[8] C. S. G. Conroy, *"A high-speed parallel pipeline A/D converter technique in CMOS", Memorandum No. UCB/ERL M94/9*, Electronics Research Laboratory, U. C. Berkeley, February 1994

[9] D. W. Cline and P. R. Gray, *"A power optimized 13b 5MS/s pipelined analog-to-digital converter in 1.2μm CMOS"*, *Proc. IEEE Custom Integrated Circuits Conf.*, May 1995, pp 219-222

[10] D. W. Cline, *"Noise, Speed and Power Trade-offs in Pipelined Analog to Digital Converters", Memorandum No. UCB/ERL M95/94*, Electronics Research Laboratory, U. C. Berkeley, November 1995

[11] Y. M. Lin, *"Performance Limitations On High-Resolution Video-Rate Analog-Digital Interfaces", Memorandum No. UCB/ERL M90/55*, Electronics Research Laboratory, U. C. Berkeley, June 1990

[12] Y. M. Lin, "*A 13-b 2.5-MHz Self-Calibration Pipelined A/D Converter in 3-μm CMOS," IEEE J. Solid-Stage Circuits*, vol.26, pp. 628-636, April 1991

[13] S. H. Lewis and P. R. Gray, *"A pipelined 5-Msamples/s 9-bit analog-to-digital converter," IEEE J. Solid-State Circuits*, vol. SC-22, pp.954-961, Dec. 1987

[14] S. H. Lewis, *"Video-Rate Analog-to-Digital Conversion Using Pipelined Architectures", Memorandum No. UCB/ERL M87/90*, Electronics Research Laboratory, U. C. Berkeley, November 1987

[15] S. H. Lewis, et al., *"10b 20MS/s analog-to-digital converter", IEEE J. Solid-Stage Circuits*, vol.27, pp. 351-358, March 1992

[16] B. Razavi and B. A. Wooley, "Design techniques for high-speed, high-resolution comparators", IEEE J. Solid-Stage Circuits, vol. 27, no. 12, December 1992, pp. 1916-1926

[17] K. J. McCall, M. J. Demler, and M. W. Plante, "*A 6-bit 125MHz CMOS A/D Converter," in Proc. CICC*, May 1992, pp. 16.8.1-16.8.4