Equation-based Hierarchical Optimization of a Pipeline ADC

by

Tania Khanna
B.S., Electrical and Computer Engineering (2005)
Cornell University

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Master of Engineering in Computer Science and Electrical Engineering
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2008

Author . . . .
Department of Electrical Engineering and Computer Science
January 18, 2008

Certified by . . . .
Joel Dawson
Assistant Professor
Thesis Supervisor

Accepted by . . . .
Terry P. Orlando
Chairman, Department Committee on Graduate Students

Equation-based Hierarchical Optimization of a Pipeline ADC

by

Tania Khanna
B.S., Electrical and Computer Engineering(2005)
Cornell University

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Master of Engineering in Computer Science and Electrical Engineering
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Abstract

In system design, allocation of circuit resources like power and noise budget is a difficult problem. It is difficult to know the optimal distribution of resources because the performance space of each component is not fully characterized. This uncertainty results in an iterative approach with frequent re-design of circuit blocks for different distribution schemes. Equation-based optimization has been shown effective and time efficient in circuit design, but is impractical for systems due to the large number of variables resulting in long solve times. This work shows an equation-based hierarchical optimization strategy suitable for design in deeply scaled CMOS processes. Because it is a hierarchical methodology, it scales gracefully to systems that are much larger than can be handled by known optimization methods.

This thesis matches flat and hierarchical optimizations of a 10-stage pipeline ADC in a 0.18-um process. A pipeline ADC was chosen because it is a system small enough to be handled by a flat optimization, yet large enough to be approached with a hierarchical methodology. This allows a quantitative comparison of the computation resources required by each strategy. In this approach, equation-based optimizations generate the Pareto-optimal surfaces of each pipeline stage. Exploiting the surfaces' gentle nature and amenability to low-order equation fits, they are abstracted to higher levels as representations of the circuit block. Thus, resources are allocated at the system level (such as power dissipation, noise budget, gain, etc.) very rapidly and very efficiently using familiar equation-based optimization strategies. In the end we demonstrate an optimization strategy that takes 25x less time to allocate resources than a traditional, flat methodology.

Keywords: System design, optimization, pipeline analog to digital converters, Pareto-optimal.

Thesis Supervisor: Joel Dawson
Title: Assistant Professor

Acknowledgements

I would first like to thank my family for all of their support and attention. Without my parents' hard work and patience, I would not be who I am. I am amazed every day at how they do it, and it is that effortlessness that I hope to achieve. My siblings have set an excellent example for me in life, and I can only give my thanks for that. My family has been there whenever I needed anything, and their constant support and tolerance of me is priceless.

Secondly, I must thank my advisor, Joel Dawson. He has always given me great advice and direction, especially in recognizing when I need it. In addition to the guidance he has given me, he has always been the optimistic when it has come to my research motivating me to keep going. This work could not have been completed without his help and encouragement.

I am grateful to Sabio Labs, Inc. for their optimization toolbox and constant software support. They have always been willing and prompt in improving the tools for my use. In addition to their generous donation of their software, I would like to thank Mar Hershenson, Sunderarajan Mohan, and Dave Colleran for the donation of their time. Their willingness to share the vast knowledge they've accumulated and their enthusiasm for the topic is immeasurable.

I would also like to thank Ranko Sredojevic, Vladimir Stojanovic, and Willie Sanchez for the many conversations on optimization theory and application. It has been an enlightening experience. Lastly, I extend my thanks to the entire Dawson and Perrott research groups for their company and support on long and short nights.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background information

Much work has been done within the optimization and circuit communities related to the optimization of individual circuit blocks [1, 2]. Both equation-based and simulation-based optimization methods have enjoyed recent success for certain problems [3, 4]. However, the best of these newest methods are still painfully overwhelmed by the sheer size of the design space typical of even modest sized, mixed-signal systems. Examples of such systems abound, including signal paths for RF transmitters and receivers, high-speed serial communications links, and pipelined analog-to-digital converters. Much of today's designs are done by experienced designers who have over time developed the intuition needed to design such complex systems. Employing hierarchy is a natural way to cope with a large number of design variables, and many hierarchical approaches have been explored to simplify system level optimization [5-7]. Still, an efficient technique that brings value to the designer remains elusive.

The result is that many gains and insights reported by the optimization community have yet to impact design practice for large, mixed-signal systems. Current design practice may be summarized as follows. First a system designer breaks down the system into individual blocks, distributing block specifications according to system level equations and circuit design experience. These block specifications are then handed to circuit designers who choose circuit topologies to meet the specifications. The circuit designer then returns a small number of design points for the assigned block back to the system designer who re-evaluates the initial spec distribution. Iterations of this process are repeated until the entire system meets specification. These iterations are necessary because the in the beginning stages of system design, the system designer has to rely on her experience and a discrete number of operating points to characterize the operation of each block. The method applied here allows the system designer to have a quantitative characterization of each block's performance space before allocating system resources.

1.2 Hierarchical optimization

This thesis builds on previous optimization work [8, 9] by decomposing a large system into manageable circuit blocks and adopting a hierarchical, bottom-up (H-BU) approach. To begin, we break the system up into its primitive blocks, and then produce Pareto-optimal [1] surfaces for these blocks. These surfaces can be obtained using a variety of techniques including geometric programming (GP) and evolutionary algorithms (EA) [6] [8]. The H-BU method has been applied to a discrete component receiver segment in a previous work [12], which utilizes GP to generate Pareto-optimal surfaces. However, in this application we veer from GP while still using equation-based techniques with a specialized solver for circuit design from Sabio Labs, Inc. [11] that can handle signomial equations. This deviation allows for more accurate models of the CMOS process than GPs allow for with the benefit of writing more intuitive circuit design equations than EAs allow for.

The tradeoffs between performance metrics (such as gain, noise performance, power, linearity, etc.) are quantitatively modeled as simple monomial functions. Unlike past works [5, 6], we take advantage of the gentle nature of the tradeoffs by using simple monomials, and the result is that the system formulation is less complex and more solver-friendly. Moreover, because the design space is confined for each primitive block to the relatively small slice inhabited by the Pareto surface, we achieve this simplicity without a corresponding severe degradation in accuracy. This work applies the H-BU method on a 10-stage pipeline ADC in a 0.18μm CMOS process with a 100Mbps sampling frequency as proof of concept.

1.3 Thesis organization

Chapter 2 explains the circuit theory of a pipeline ADC by decomposing it into its components. The ADC is also broken down into manageable circuit blocks for the purpose of applying a hierarchical optimization strategy. Finally, design choices for the operational amplifier (op-amp), of great importance in the overall design, are discussed.

Chapter 3 extends the circuit theory and intuition from Chapter 2 to a mathematical program for the pipeline in a flat formulation. The formulation starts from a top-level system design, descends to the pipeline stage topology including an op-amp, and finally to transistor-level design of the op-amp. The focus of Chapter 3 is to understand the circuit theory from a mathematical and hierarchical perspective.

In Chapter 4, the framework from Chapter 3 is used to create a hierarchical formulation of the pipeline. To do this, we abstract a lower level block—in this case a pipeline stage—using low-order equations modeling the tradeoffs between system design variables. Also discussed is the generation of Pareto-optimal surfaces, which are gentle and amenable to fits with the low-order equations. This a non-trivial task because the entire design space of the pipeline stage must be encapsulated in the Pareto-optimal surface.

Chapter 5 shows and compares the optimization results for both the hierarchical and flat formulations. Conclusions and future directions of this work are given in Chapter 6.

Chapter 2

ADC Pipeline

The H-BU methodology can be applied to any system that can be dissected into discrete blocks, and a variety of mathematical programs can be used to implement the method. One such example is the receiver segment [12], which used GP as the technique for optimization. This work retains equation-based optimization techniques, but it veers from GP by allowing signomial [11] expressions in the formulation. The method is applied to a pipeline ADC because it lends itself well to a hierarchical optimization with its many stages, while being practical for a comparison to a flat optimization because of the reasonable total number of design variables.

In the following sections we will explain operation of the pipelined ADC as well as discuss the design choices made from a circuit design perspective, and complete the first step of the H-BU method, which is to decompose the desired system into smaller, more manageable blocks.

2.1. Pipeline operation

The most elementary analog-to-digital converter compares an input voltage with a reference voltage and decides whether the input is higher or lower than that reference. This simple comparison would constitute a 1-bit conversion, however for most applications higher bit resolution is necessary.

To extend the conversion precision we can use a pipelining technique by carrying over to another stage the remainder of the input signal after subtracting either the reference voltage or nothing when the decision bit was high or low respectively. The residue is first multiplied by two (for a 1-bit per stage resolution) so that it is amplified to the full input voltage range and can be passed onto the next stage for the next bit comparison. This pipelining is continued through n-stages to obtain n-bit accuracy. This concept can be extended to k stages with an n bit resolution per stage seen in Figure 2-1. The gain element required for this generalized pipeline is 2^n. This work applies the H-BU method to a 10-stage pipeline with a 1 bit per stage resolution.

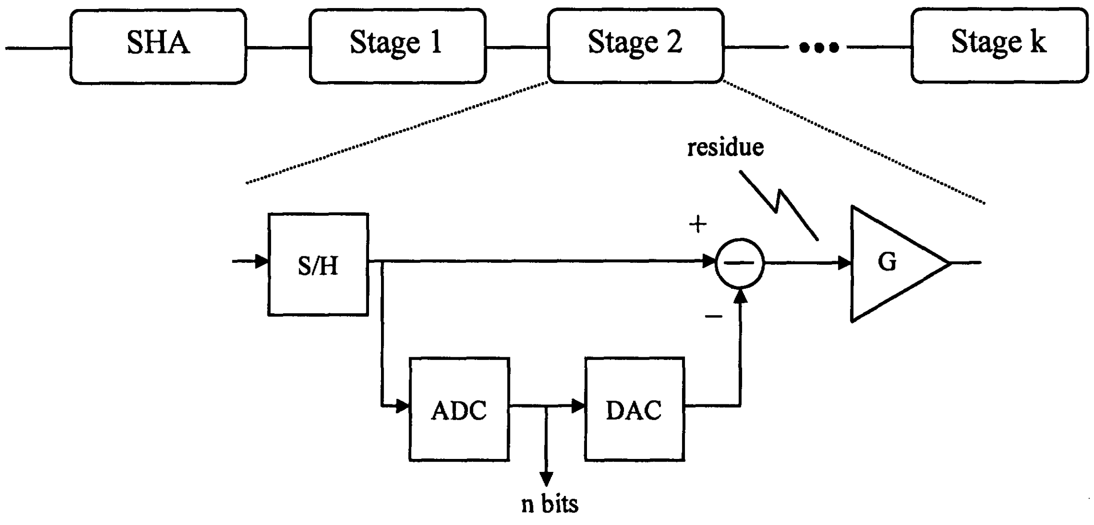


Figure 2-1. Pipeline decomposition and stage operation.

In pipeline ADC design, the limiting factor of power consumption is the operational amplifier (op-amp) because of the necessary gain element. Current design practice scales down the op-amp in later stages to reduce power consumption without reducing bit resolution or speed. The noise requirements in later stages are more lax, thus a smaller, less power hungry op-amp can be used. However, the *optimal* scaling is typically only approximated with rules of thumb, and often it is not worth the designer's time to precisely scale the last few stages. The last uniquely designed stage is therefore cascaded as the remaining stages to fulfill the bit precision specification. The focus of this work is to apply the H-BU method to minimize power by optimally scaling the stages, and compare the optimization results with a flat optimization. In so doing, we will have demonstrated a hierarchical methodology that scales to systems much larger than pipelined ADCs.

2.2. Sample and hold amplifier

The pipeline is fully differential, which results in many circuit benefits including improved SNR, immunity to common mode disturbances, and cancellation of even-order distortion products. We chose an active sample and hold amplifier (SHA) for the input sampling, seen in Figure 2-2. During the sampling phase, the φ_1 switches are closed, and during the hold phase, the φ_2 switches are closed.

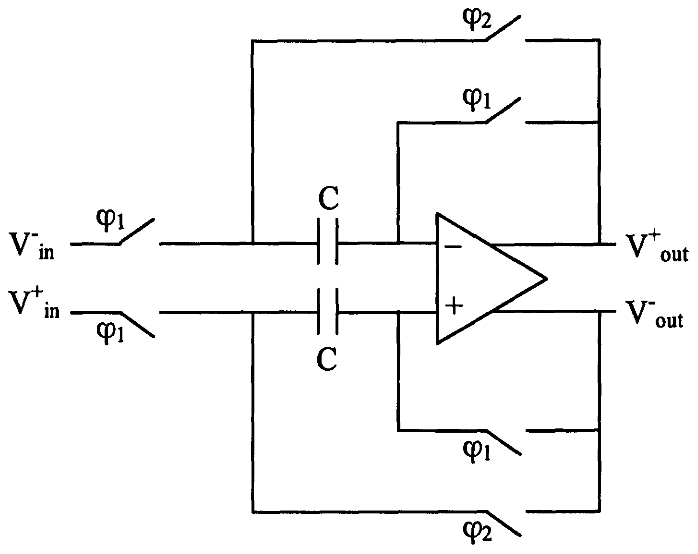


Figure 2-2. Fully differential sample and hold amplifier.

To analyze this active circuit, it is easier to look at the single ended operation. Both phase configurations are seen in Figure 2-3. In Figure 2-3(a), the input voltage is applied across the sampling capacitor, thus storing a charge $Q = CV_{in}$. Also, the output is connected to the inverting input and thus set to ground. When the amplifier switches to the hold phase, we see the configuration in Figure 2-3(b). The input is now disconnected and the output connects to the other side of the sampling capacitor. Because of charge conservation and constant capacitance the voltage across the capacitor must be V_{in}, and therefore V_{out} holds the input voltage.

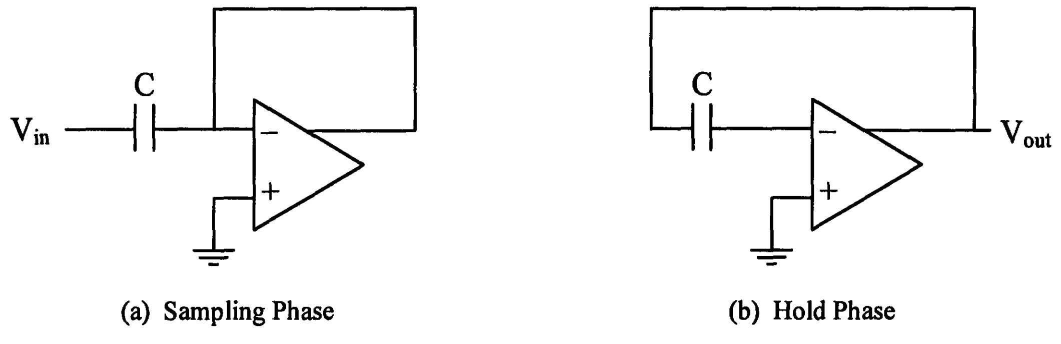


Figure 2-3. Single ended phase configurations for sample and hold amplifier.

2.3. Pipeline stage

Each stage of the pipeline must perform a sample and hold operation, but we can incorporate the gain element into this existing configuration to be more efficient. The pipeline stage is shown in Figure 2-4. With this configuration, there is still a sample phase but the hold phase is replaced with an amplification phase. For convenience, we will still call this the hold phase, but it is implied that in the case of a pipeline stage there is a gain of 2.

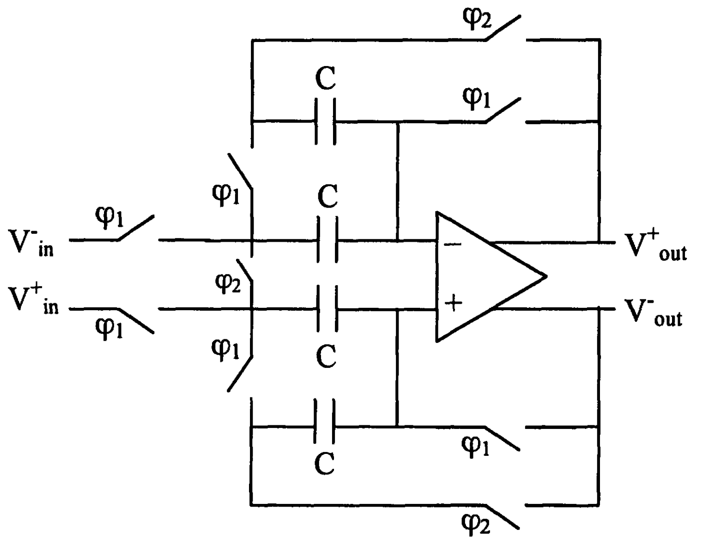
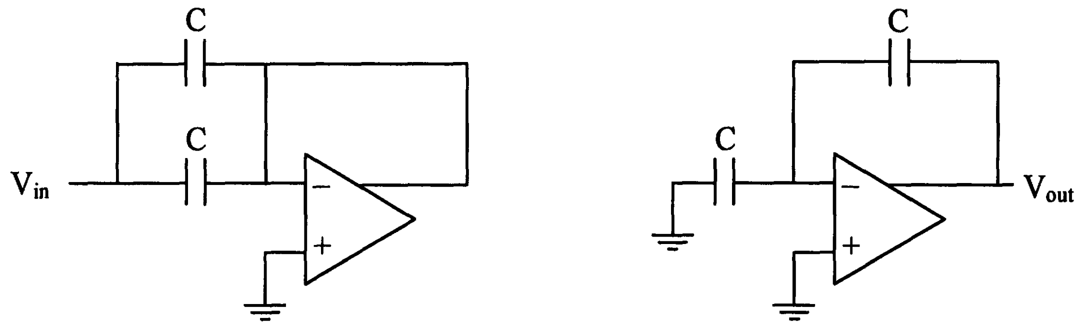


Figure 2-4. Fully differential pipeline stage.

Again, the single ended phase configurations in Figure 2-5 are helpful in analyzing the stage operation. In Figure 2-5(a), the input voltage is applied across the sampling capacitors in parallel, thus storing a charge Q = CV_{in} on each capacitor. Also, the output is connected to the inverting input and thus set to ground. When the phase changes to the hold phase, we see the configuration in Figure 2-5(b). The input is now connected to ground and the sampling capacitors are in series. Because of charge conservation and constant equal capacitance the voltage across each capacitor must be V_{in}, and therefore V_{out} holds $2*V_{in}$.



(a) Sampling Phase (b) Hold (Amplification) Phase

Figure 2-5. Single ended phase configurations for pipeline stage.

A more straightforward approach is to apply Kirchoff's Current Law (KCL) on the inverting terminal of the op-amp, which at the time of switching holds the V_{in}:

$$\begin{aligned} sC\cdot(0-V_{in}) &= sC\cdot(V_{in}-V_{out}) \\ V_{in} &= V_{in}-V_{out} \\ V_{out} &= 2V_{in} \end{aligned}$$

The ADC from figure 2-1 is implemented as a simple comparator, where a sampled voltage larger than the common mode produces a '1' bit and a voltage lower than the common mode produces a '0'. The decision bit then controls switching circuitry as the make-up of the DAC, which subtracts the correct reference voltage to calculate the residual voltage. In this work, we exclude the ADC and DAC from the formulation, because these blocks are not dominant factors in the power consumption of the pipeline, and they are supporting circuitry, which can be designed independently.

We can now construct the pipeline by feeding the analog signal into the sample and hold amplifier and outputs into 10 pipeline stages cascaded together to provide 10 bits of resolution. Thus far we have analyzed how the digital conversion is obtained, but we must characterize the operational amplifier carefully because that is where the majority of the power is consumed.

2.4. Operational Amplifier

The op-amp topology chosen for the pipeline was comprised of two fully differential stages and is shown without biasing circuitry in Figure 2-6. The first stage has a pMOS input differential pair with a folded cascode to achieve high gain by increasing output impedance [13] [14], and the second stage is a differential common source amplifier. Two stages and cascoding were necessary to minimize finite op-amp gain errors in the pipeline, which was crucial for accuracy. If we call the op-amp gain A, the closed loop gain for a pipeline stage can be calculated as follows using Black's formula [15]:

$$V_{out} = \frac{A}{1+(A)\left(\frac{1}{2}\right)} V_{in}$$

$$V_{out} = \frac{2}{1+\left(\frac{2}{A}\right)} V_{in}$$

Looking at the resulting expression, we note that as A approaches infinity the gain approaches 2, the ideal value for perfect residual amplification.

Since the op-amp is fully differential we must actively control the common mode output. Each stage has a node biased using feedback circuitry from that stage's output node. These nodes are marked in Figure 2-6 as V_{cm1} and V_{cm2}. These nodes are important again to minimize op-amp finite gain errors, since they ensure the op-amp is operating with the correct common mode to achieve the highest gain.

One final consideration taken in choosing the op-amp topology is frequency compensation. Because the op-amp is in a negative feedback configuration, we must ensure the stability of the circuit to prevent oscillation. By employing a frequency compensation to split the poles of the op-amp, we push one pole higher in frequency and extend the stable frequency range [13] [14]. A common technique for pole splitting is Miller compensation [13] [14], which is implemented by placing a capacitor between the input and output nodes of the second stage. Though Miller compensation is common and easy to implement, it is not the optimum from the standpoint of increasing closed loop bandwidth. We can do better by using the relatively new technique of cascode compensation [16-18]. To employ cascode compensation, we insert a feedback capacitor from the output of the second stage to the cascode node of the first stage shown in Figure 2-6. In taking advantage of cascode compensation, the op-amp can achieve a greater bandwidth thereby by increasing the pipeline sampling rate.

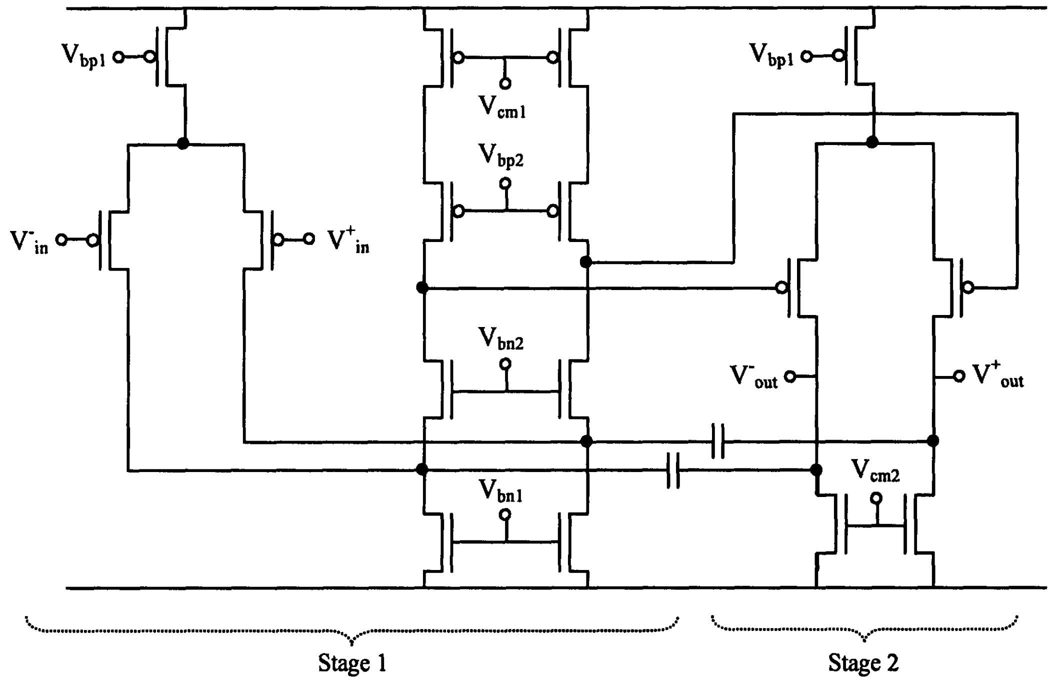


Figure 2-6. Operational amplifier circuit topology.

From this exploration into the circuit theory of the pipeline, we see that hierarchy is useful when analyzing circuits. Much of the theory relies on abstraction of lower level blocks like a pipeline stage or an op-amp. With well-defined topologies, it is possible to mathematically express the performance of each block, and extend hierarchy to the performance tradeoff space of each block as well as in the space of node voltages and currents.

Chapter 3

ADC flat formulation

Before applying the hierarchical method, we first formulate the ADC as a flat optimization problem. This flat formulation is necessary for the H-BU method, since it provides an analytical framework for the hierarchical formulation and Pareto surface generation. However, the proposed method does not require us to actually follow through with a flat *optimization*, which is computationally intensive and thus very time consuming. We only perform a flat optimization in this work by way of comparison, in order to highlight the tremendous speed advantage of a hierarchical optimization.

3.1 System formulation

A simple system level formulation of the full pipeline is

$$\begin{array}{ll} \text{minimize} & \mu_P \cdot P_{total} + \mu_A \cdot A_{total} \\ \text{subject to} & f_{Sample} > f_{Sample,Spec}, \\ & SNR > SNR_{Spec} \end{array}$$

where μ_P and μ_A are normalizing constants for system power and area respectively, f_{Sample} is the sampling frequency, SNR is the signal-to-noise ratio, and $f_{Sample,Spec}$ and SNR_{Spec} are the system specifications that must be met. The system formulation is rather simple and similar to equations system designers already use to approximate system performance. In this work we formalize this, making the case that equation-based optimization is superior to simulation-based optimization at the *system* level.

As discussed in Chapter 1 and 2, the most intuitive way to decompose a pipelined ADC is by its stages seen again in Figure 3-1 with a circuit view of each stage. In a hierarchical design, there are many design variables that are not important to higher levels of hierarchy, such as transistor dimensions, bias currents and voltages, and so on. In generating our Pareto-optimal surfaces, we must choose the variables that *are* important at higher levels in the hierarchy. Or in other words, we must choose the variables that determine the system level specs, which in the case of an ADC are the SNR and sampling frequency. These, in general, will be performance metrics that contribute to the equivalent performance metric at the system level. For example, the power of a given stage is important because it contributes to the overall power consumption of the system. For this problem, we determined that the important system design variables to be abstracted from each stage are: power (*P*); area (*A*); output-referred noise power (*nPO*); settling time constant (τ_{hold}); slew rate (*slewRate*); input capacitance of the sampling stage (*cIS*); output capacitance of the hold stage (*cLH*). In the following sections, we detail the impact of each of these performance metrics on the overall system. The commented top-level system code (ADC_top) is given in Appendix A.1.

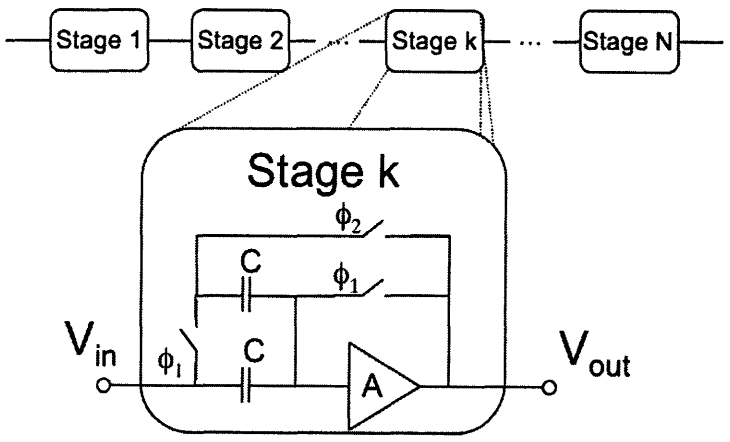


Figure 3-1. Pipeline decomposition with a circuit view of a pipeline stage.

3.1.1. Power and area

The power and area expressions are straightforward and simply a sum of the powers and area of each pipeline stage and the SHA.

$$P_{tot} = P_{SHA} + \sum_{n=1}^{N} P_n, \quad A_{tot} = A_{SHA} + \sum_{n=1}^{N} A_n .$$

In the system level objective function, power and area are weighted with normalization factors, μ_x, so both are held at equal importance during optimization. The circuit designer determines the normalization factors by examination. For example, power and area are on the order of mW and mm^2 respectively resulting in a μ_P=10^3 and μ_A=10^9. This is an example of the designer insight embedded into the formulation.

3.1.2. Signal-to-noise ratio

The dominant noise contributions in the pipeline are due to quantization and thermal noise given as follows:

$$v_{n,tot}^2 = v_{n,quant}^2 + v_{n,thermal}^2 = \left(\frac{LSB}{\sqrt{12}}\right)^2 + \sum_{k=1}^{N} \frac{nPO_k}{2^{2k}},$$

where nPO_k is the output referred noise power of stage k and is explored in section 3.2. The SNR can then be written as

$$SNR = \frac{V_{pp,\max}^2}{8 \cdot v_{n,tot}^2},$$

where $V_{pp,max}$ is the maximum output peak-to-peak voltage.

3.1.3. Sampling frequency

Sampling frequency is constrained by the settling time, t_{Settle}, of each stage, so for a given sampling frequency spec, we can add the following constraint to the system formulation:

$$t_{Settle} = \frac{1}{2 \cdot f_{Sample}}.$$

This constraint ensures that each stage settles within half the sampling period. The settling time of each stage can further be expressed as the time taken for a single ended output to slew across the half the maximum output voltage range, $V_{pp,max}$, plus linear settling time after the slewing:

$$\begin{aligned} t_{Settle,n} &= t_{Settle,slew,n} + t_{Settle,linear,n} \\ &= \frac{1/2 \cdot V_{pp,\max}}{slewRate_n} - \log(precision_n) \cdot \tau_{hold,n} \end{aligned}.$$

where $precision_n$ is the required precision of stage n to achieve the desired bit resolution; $\tau_{hold,n}$ and $slewRate_n$ are the settling time constant and slew rate of stage n respectively. This expression gives us a worst-case calculation for the settling time, and can be constrained

$$t_{settle,n} < t_{settle},$$

for $n=1,...,N$. Therefore, this formulation constrains the worst case settling time such that the sampling frequency specification is met.

3.2 Pipeline stage formulation

The pipeline stage formulation aims to define the system level design variables in terms of the circuit components in the stage, namely the sampling capacitors and op-amp seen in Figure 3-1. In the following sections, we explain the formulation for one pipeline stage, which is expressed in commented code in Appendix A.2.

3.2.1 Power and Area

The power and area of each stage is straightforward:

$$\begin{aligned} P &= P_{Amp} \\ A &= A_{Amp} + 4 \cdot A_{C_{Sample}} \end{aligned}.$$

Here we approximate the total power of stage as the power consumed by the op-amp since it is the dominating power sink. Similarly, the area is defined as the active area comprised of the op-amp and sampling capacitors. The active area of a transistor (A_{Mx}), is calculated as the product of the width (W_{Mx}), length (L_{Mx}), and number of fingers (n_{Mx}):

$$A_{Mx} = W_{Mx} \cdot L_{Mx} \cdot n_{Mx},$$

and the capacitor active area is the product of the capacitance and a constant capacitive density, which is determined by the process. It should be remembered that the stage is fully differential so there are a total of 4 equal sampling capacitors. All other equations assume the half circuit analysis technique in formulating the pipeline operation.

3.2.2 Noise power

For calculating noise in each pipeline stage, we rely on the Sabio models [11] for the noise current in a transistor, which express the noise current as a function of transistor dimensions. Shown in section 3.4.2, we can write an expression for the output noise current of each stage of the op-amp as a function of each transistor noise current and then calculate the output referred thermal noise:

$$nPO = \frac{i^2_{n,Stg1,Amp}}{4 \cdot g_m \cdot f_{FB} \cdot C_{out,Stg1,Amp}} + \frac{i^2_{n,Stg2,Amp}}{4 \cdot g_m \cdot (C_{out,Stg2,Amp} + C_{c,Amp})},$$

where $i^2_{n,Stg1,Amp}$ and $i^2_{n,Stg2,Amp}$ are the square noise current at the output of stage 1 and stage 2 of the op-amp respectively; $C_{out,Stg1,Amp}$ and $C_{out,Stg2,Amp}$ are the node capacitances at the output of stage 1 and stage 2 of the op-amp respectively; and $C_{C,Amp}$ is the compensation capacitor in the op-amp. During the hold mode, the pipeline stage is in feedback and there is an associated feedback factor, f_{FB}, defined as the gain from the output to the input of the stage. This configuration is recalled from Figure 2-5(b) and modified to include the input capacitance of the op-amp, $C_{in,Amp}$, in Figure 3-2. Also recalled is that the node voltage on the inverting terminal is the input voltage sampled during the previous sample phase.

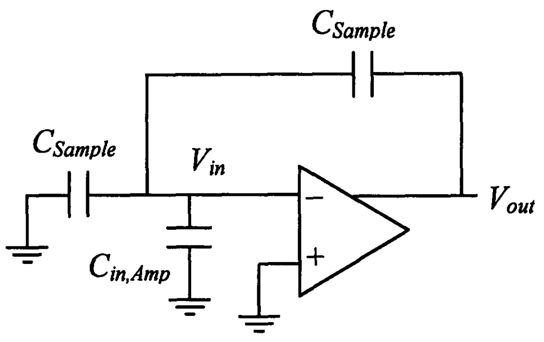


Figure 3-2. Stage in amplification phase with op-amp input capacitance.

Applying KCL on the inverting node of the op-amp, we get

$$s \cdot C_{Sample} \cdot (V_{out} - V_{in}) = s \cdot (C_{Sample} + C_{in,Amp}) \cdot (V_{in})$$
$$C_{Sample} \cdot V_{out} = (2 \cdot C_{Sample} + C_{in,Amp}) \cdot V_{in} \quad .$$
$$f_{FB} = \frac{V_{in}}{V_{out}} = \frac{C_{Sample}}{2 \cdot C_{Sample} + C_{in,Amp}}$$

3.2.3 Settling time constant and slew rate

To fulfill the sampling frequency constraint in section 3.1.3, we need to define the settling time constant, $\tau_{hold,}$ and slew rate, *slewRate*, to express the settling time. By approximating the amplification phase transfer function as a single pole system, we can define the following:

$$\tau_{hold} = \frac{1}{\frac{1}{2} \cdot GBW_{Amp}},$$

where GBW_{Amp} is the gain-bandwidth of the op-amp. Since in hold mode, the op-amp is in a gain of 2 configuration, we divide the *GBW* by a factor of 2. Finally, the slew rate of the stage is simply defined as the op-amp slew rate:

$$slewRate = slewRate_{Amp} \, .$$

The sample and hold amplifier is similarly formulated, except there is only one sampling capacitor. The full sample and hold formulation code can be found in Appendix A.3.

3.3 Interface variables

To ensure that the pipeline stages can interact with each other, we choose to force the interface variables to be compatible. A potential consequence of this choice is a loss in performance, as it requires the designer to circumvent a true, "unpiloted" optimization and impose this discipline. However, the imposition frequently is in line with good design practice, and the result seems to be that no such performance penalty is ultimately paid.

With the ADC we must worry about input and output capacitances of each stage in the two operating modes—hold and sample. It can be shown that the pipeline can be broken down into hold/sample pairs shown in Figure 3-3, and that the input and output capacitance of the hold and sample stages respectively can be ignored since those voltage nodes are virtual grounds and any difference in capacitance will not affect performance.

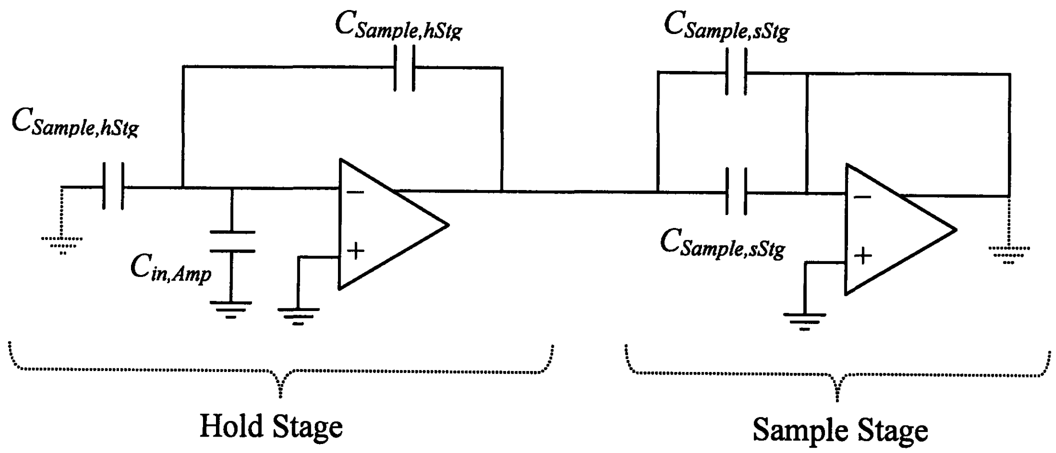


Figure 3-3. Hold/sample pair.

Therefore, for all pairs of consecutive stages

$$cLH = cIS,$$

where *cLH* is the load capacitance seen by the hold stage and *cIS* is the input capacitance of the sample stage. These values determine the load capacitance seen by the op-amp, which affects the design of the op-amp. Therefore, we introduce another variable C_L, which is the load capacitance seen by the op-amp. We can now express the values of these three variables in terms of the sampling capacitors and input capacitance of the hold stage:

$$\begin{aligned} cIS &= 2 \cdot C_{Sample,sStg} \\ cFB &= C_{Sample,hStg} \| (C_{Sample,hStg} + C_{in,Amp}) \\ &= \frac{C_{Sample,hStg} \cdot (C_{Sample,hStg} + C_{in,Amp})}{2 \cdot C_{Sample,hStg} + C_{in,Amp}}, \\ C_L &= cIS + cFB = cLH + cFB \end{aligned}$$

where *cFB* is the capacitance seen looking into the feedback path of the hold stage. With the constraints placed on *cIS*, *cLH*, and C_L, we can ensure that the blocks will interact with each other as expected.

These constraints can be found in the 'connect' codes in Appendix A.4. Also included in this code are the settling time constraint from section 3.1.3 and the noise power calculation mentioned in section 3.2.2. The 'connect' codes are actually part of the top-level system formulation, but for convenience in coding they were included segmented into their own files.

3.4 Operational amplifier formulation

We finally formulate the op-amp, which consists of transistor level design. In Figure 2-6 we see the op-amp circuit diagram, but in Figure 3-4 is the equivalent half circuit diagram including bias circuitry to set node voltages. Using the equivalent half circuit, we can formulate a mathematical program to define all the op-amp parameters used in all higher-level formulations. We now express all op-amp parameters in terms of transistors parameters and other circuit element, which are all defined as a function of the transistor dimensions and biasing by signomial models created by Sabio Labs, Inc. For greater detail, the full op-amp formulation code is in Appendix A.5.

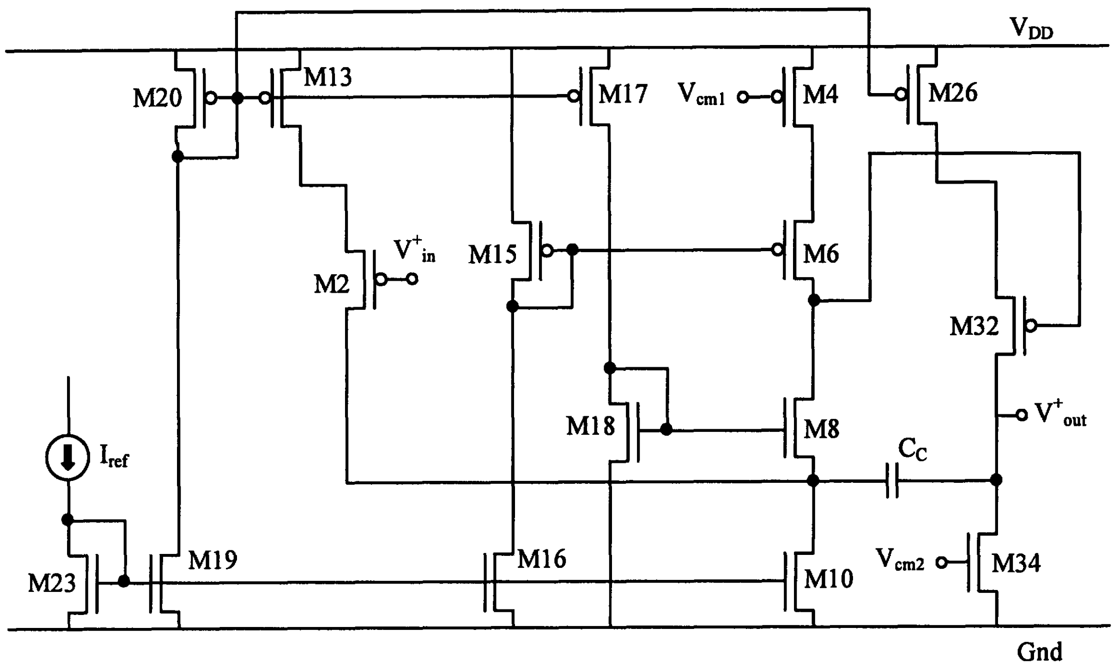


Figure 3-4. Operational amplifier half-circuit with bias circuitry.

3.4.1 Power and area

Power is simply calculate by summing the current flowing from the power supply, V_{DD}, to ground and then multiplied by V_{DD}:

$$P = V_{DD}(i_{bias} + i_{stage1} + i_{stage2}).$$

The op-amp area is expressed as the active area

$$A = 2 \cdot A_{C_C} + \sum_{bias} A + \sum_{core} 2 \cdot A,$$

or as the sum of the transistor areas and capacitor areas, which are now direct function of transistor dimensions and capacitances respectively.

3.4.2 Noise

From higher-level formulations, we know the exact expression for noise we need to extract:

$$\begin{aligned} i_{n,Stg1}^2 &= 2 \cdot (i_{n,M2}^2 + i_{n,M4}^2 + i_{n,M10}^2) \\ i_{n,Stg2}^2 &= 2 \cdot (i_{n,M32}^2 + i_{n,M34}^2) \end{aligned},$$

where $i^2_{n,Stg1}$ and $i^2_{n,Stg2}$ are the square noise currents at the outputs of stage 1 and stage 2 of the op-amp respectively.

3.4.3 Gain-bandwidth and slew rate

The settling time constraints relied on a gain-bandwidth calculation from the op-amp. In true circuit designer fashion, we assume the op-amp is a single pole system, and define the gain-bandwidth as

$$GBW = \frac{g_{m,In}}{C_C}.$$

This assumption is valid because we force constraints on the other poles of the system such that they are high frequency. Therefore the dominant pole is used to define the gain-bandwidth. The imposed constraints are explained in greater detail in section 3.4.4.3.

The other op-amp characteristic needed to calculate settling time is slew rate. We can express the slew rate as the minimum of the slew rates of each stage in the op-amp, by creating a variable *slewRate* and forcing

$$\begin{gathered} slewRate_{Stg1} < slewRate \\ slewRate_{Stg2} < slewRate' \end{gathered}$$

where each stage slew rate is given as

$$\begin{gathered} slewRate_{Stg1} = \frac{i_{M8}}{C_{out,Stg1}} \\ slewRate_{Stg2} = \frac{\frac{1}{2} i_{M26}}{C_{out,Stg2}} \end{gathered}.$$

3.4.4 Functionality constraints

In addition to defining the variables passed onto higher levels of hierarchy, we must ensure the op-amp is operating correctly to rule out any impractical solutions that are still mathematically feasible. In other words, the formulation is not complete without additional constraints to describe the practical design of the op-amp. The following sections detail the functionality constraints, which complete the formulation.

3.4.4.1 Kirchoff's Laws and Circuit biasing

The most basic of functionality constraints is to have Kirchoff's Voltage and Current Laws (KVL and KCL) to hold over all loops and nodes in the op-amp. This would include applying KCL to equate currents of transistors in series and summing currents to zero at intersections of three or more transistors. For example, looking at Figure 3-4 transistors M4, M6, and M8 have equal currents translating to the following constraints:

$$i_{M4} = i_{M6}$$
$$i_{M6} = i_{M8}.$$

Furthermore, the current in M10 is the sum of the currents in M2 and M8:

$$i_{M10} = i_{M8} + i_{M2}.$$

Applying KVL is much more involved, but can be categorized into 6 different applications: 1) setting source-to-bulk voltages, 2) summing voltages in transistor stacks, 3) setting diode connections, 4) gate biasing, 5) stage connection, and 6) input and output common mode. The necessary constraints are rather straightforward and can be found in Appendix A.5; however we include a comment on the common-mode feedback nodes, V_{cm1} and V_{cm2}. These nodes are not defined to and exact voltage, but rather are floating within a range of voltages, because in the practical implementation of the op-amp there would be a common-mode feedback circuit adjusting these voltages. This is another example of a circuit designer mimicking design practice in the formulation of the circuit.

3.4.4.2 Gain

Another functionality constraint is the DC gain of the op-amp. The importance of a high gain was described in Section 2.4, and therefore we impose a minimum constraint to reduce finite-gain errors in the pipeline.

$$Gain > spec.gainDC$$

Op-amp gain is defined as the input transconductance over the output conductance of the op-amp, and we can calculate the gain as the product of the gain of each stage:

$$Gain = Gain_{Stg1} \cdot Gain_{Stg2}$$

$$Gain = \frac{g_{m,In,Stg1}}{g_{out,Stg1}} \cdot \frac{g_{m,In,Stg2}}{g_{out,Stg2}}.$$

For exact definitions of each variable, see Appendix A.5.

3.4.4.3 Phase margin

Also discussed in section 2.4 was frequency compensation in an op-amp, and its importance to ensure stability. We chose to use cascode compensation and from [19] can write the denominator of the op-amp transfer function as

$$D(s) = s^3 d_3 + s^2 d_2 + s d_1 + d_0,$$

where

$$\begin{aligned}
d_3 &= g_{m,M6} \cdot C_C \cdot C_{out,Stg1} \cdot \left(C_{out,Stg2} \left(1 + \frac{C_{casN}}{C_C} \right) + C_{casN} \right) \\
d_2 &= g_{m,M6} \cdot g_{m,M8} \cdot C_{out,Stg1} \cdot \left(C_C + C_{out} \right) \\
d_1 &= g_{m,M6} \cdot g_{m,M8} \cdot g_{m,In,Stg2} \cdot C_C \\
d_0 &= g_{m,M6} \cdot g_{m,M8} \cdot g_{out,Stg1} \cdot g_{out,Stg2}
\end{aligned}.$$

The dominant pole is the ratio

$$\frac{d_0}{d_1} = \frac{g_{out,Stg1} \cdot g_{out,Stg2}}{g_{m,In,Stg2} \cdot C_C},$$

which, when multiplied by the gain, is equal to the gain-bandwidth calculated in 3.4.3. Since we assume a dominant pole, at high frequencies, *D(s)* can be approximated to

$$D'(s) = s^3 d_3 + s^2 d_2 + s d_1,$$

and the non-dominant poles are the zeros of this transfer function. However, we don't care about the location of the poles, but rather the phase shift induced by them.

If we assume t_1 and t_2 to the be the non-dominant poles, we can re-write *D'(s)*:

$$D'(s) = s(1 + st_1)(1 + st_2),$$

where

$$t_1 + t_2 = \frac{d_2}{d_1} \text{ and } t_1 t_2 = \frac{d_3}{d_1}$$

We can simplify our equations by dealing with the tangents of the phase shifts, instead of the actually phase shifts. Let α_1 and α_2 be the phase shifts due to pole t_1 and t_2 respectively, thus

$$\alpha_1 = \tan^{-1}(t_1) \text{ and } \alpha_2 = \tan^{-1}(t_2).$$

Furthermore,

$$\tan(\alpha_1 + \alpha_2) = \frac{\tan(\alpha_1) + \tan(\alpha_2)}{1 - \tan(\alpha_1)\tan(\alpha_2)}$$
$$= \frac{t_1 + t_2}{1 - t_1 t_2}$$

We calculated the tangent of the phase shift caused by the non-dominant poles, however we want to calculate the shift at the unity-gain frequency, therefore the desired value is:

$$\tan(PS) = \frac{GBW(t_1 + t_2)}{1 - t_1 t_2 \cdot GBW^2}.$$

Substituting in for t_1 and t_2, we get

$$\tan(PS) = \frac{GBW\left(\frac{d_2}{d_1}\right)}{1 - \left(\frac{d_3}{d_1}\right) \cdot GBW^2},$$

which we can constrain with an upper limit:

$$\tan(PS) < \tan(30^\circ).$$

Along with the evaluation of the phase margin, we add 'good design practice' constraints to ensure the poles created at the cascode nodes are non-dominant and to prevent peaking. More detail is shown in Appendix A.5. The next section elaborates on 'good design practice'.

3.4.5 Good design practice

It must be stressed that at the core of any formulation is the circuit designer and her circuit analysis, experience and intuition. Therefore we mention the ratio constraints imposed on transistors sharing a gate-to-source voltage. This constraint stems from layout practice, which calls for those transistors to have the same dimensions except for the number of fingers, thereby accounting for layout parasitic effects. The ratio constraints are in essence added for 'good design practice', though all of the formulation is rooted in 'good design practice'.

Chapter 4

ADC hierarchical formulation

At the system level, the hierarchical formulation is identical to the flat formulation; however, instead of instantiating a pipeline stage, the system formulation instantiates a hierarchical module abstracted from the pipeline stage, which encapsulates the system design variable tradeoffs. This section describes the hierarchical module and quantification of the tradeoffs.

4.1 Hierarchical stage module

The system design variables shown in Section 3.1 are treated as independent variables with constraints imposed on them, which describe the tradeoff interactions among them. The constraints are necessary to ensure that our solution is a feasible solution in the design space of the real problem. In other words, the constraints allow only physically attainable designs. To derive these constraints we fit monomial functions to Pareto-optimal surfaces, which were obtained as will be explained in Sections 4.2 and 4.3. This results in the following constraints:

$$power = c_{power} \cdot slewRate^{\alpha_{1,power}} \cdot C_L^{\alpha_{2,power}}$$
$$area = c_{area} \cdot slewRate^{\alpha_{1,area}} \cdot C_L^{\alpha_{2,area}}$$
$$cLH = c_{cLH} \cdot slewRate^{\alpha_{1,cLH}} \cdot C_L^{\alpha_{2,cLH}}$$
$$cIS = c_{cIS} \cdot slewRate^{\alpha_{1,cIS}} \cdot C_L^{\alpha_{2,cIS}}$$
$$nPO = c_{nPO} \cdot slewRate^{\alpha_{1,nPO}} \cdot C_L^{\alpha_{2,nPO}}$$
$$\tau_{hold} = c_{\tau_{hold}} \cdot slewRate^{\alpha_{1,\tau_{hold}}} \cdot C_L^{\alpha_{2,\tau_{hold}}}$$

where the coefficients and exponents, c_x and $\alpha_{i,x}$, were found in the aforementioned monomial fitting. The pipeline stage from the flat formulation can now be replaced with a stage module, which consists of these few constraints and bounds on the system design variables. Code for the hierarchical module (hier_stage) is given in Appendix B.

Using the new hierarchical module, the system formulation is now of this form:

$$
\begin{aligned}
&\text{minimize} \quad P_{tot} + A_{tot} \\
&\text{s. t.} \quad P_{tot} = P_{SHA} + \sum_{k=1}^{N} \mathbf{P_k(slewRate,C_L)} \\
&\qquad A_{tot} = A_{SHA} + \sum_{k=1}^{N} \mathbf{A_k(slewRate,C_L)} \\
&\qquad f_{Sample} > f_{Sample,Spec} \\
&\qquad t_{settle} = \frac{1}{2 \cdot f_{Sample}} \\
&\qquad t_{settle,k}(\boldsymbol{\tau}_{\mathbf{hold}}\mathbf{(slewRate,C_L)},\ slewRate) < t_{settle} \quad \forall\ k = 1 \ldots N \\
&\qquad SNR > SNR_{Spec} \\
&\qquad SNR = \frac{V_{pp,\max}^2}{8 \cdot v_{n,tot}^2} \\
&\qquad v_{n,tot}^2 = \left(\frac{LSB}{\sqrt{12}}\right)^2 + \sum_{k=1}^{N} \frac{\mathbf{nPO_k(slewRate,C_L)}}{2^{2k}} \\
&\qquad \mathbf{cLH_k(slewRate,C_L) = cIS_{k+1}(slewRate,C_L)} \quad \forall\ k = 1 \ldots N-1
\end{aligned}
$$

where the system design variables are now replaced with monomial functions describing the tradeoffs between them shown in bold.

4.2 Pareto-optimal surface generation

The H-BU method is nondiscriminatory on how the Pareto-optimal surfaces are generated. In [20] genetic algorithms are used to generate Pareto surfaces, and in [21] yield-aware Pareto surfaces are obtained. Though the surfaces are generated by simulation-based techniques, they are still well behaved and amenable to monomial fits.

In this thesis, the Pareto-optimal surfaces for a pipeline stage are generated using equation-based optimization by sweeping the specs on the design variables while minimizing power and area. The set of optimal design points makes up the Pareto surface, which spans the entire design space. Once the set is determined, we can perform a log transformation and linear regression on the data points to obtain the monomial fits used in the hierarchical module. The following sections give more detail on this process.

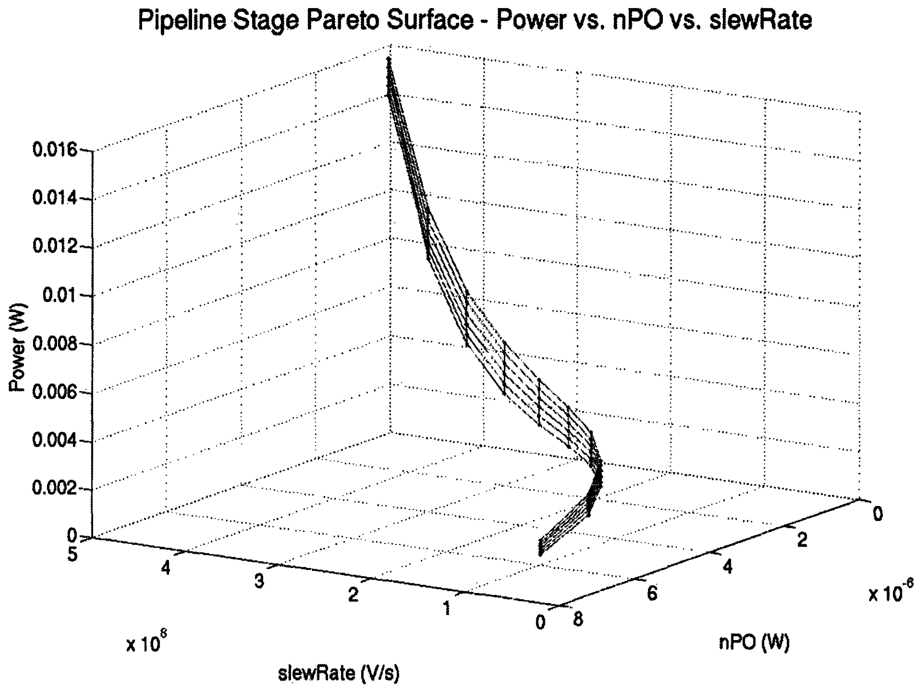
4.2.1 Design space exploration

First, each pipeline stage was formulated as an optimization problem including sampling capacitors and an internally compensated two-stage folded cascode operational amplifier as seen in Figure 3-2. The objective was constructed to minimize the normalized sum of power and area while meeting specs placed on *nPO*, τ_{hold}, *slewRate*, and C_L. Values for *cIS* and *cLH* were chosen such that the objective was minimized and C_L spec was met. A simplified formulation is as follows:

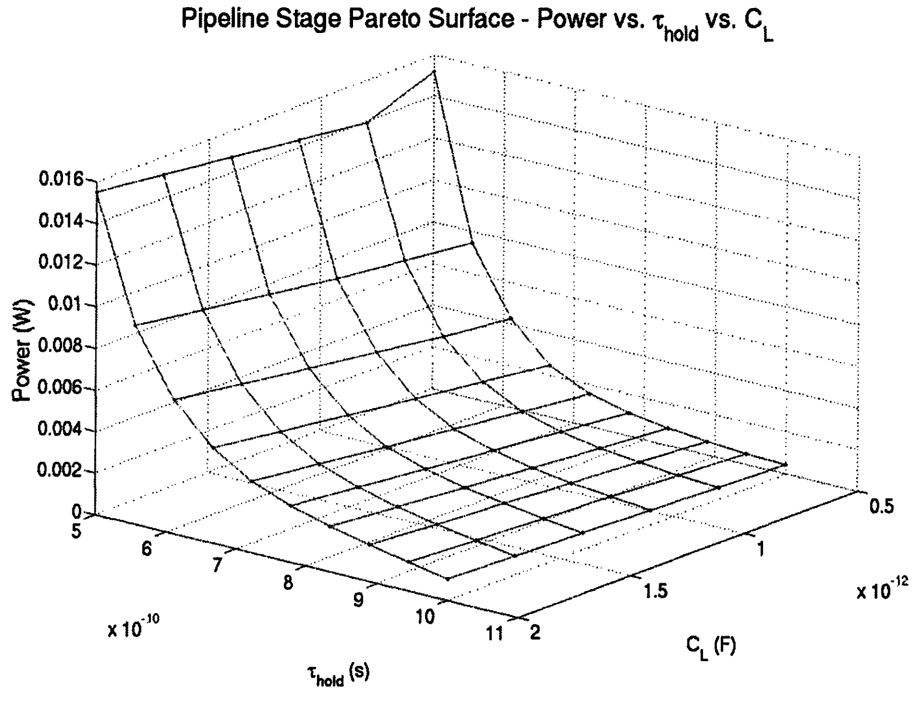
$$\begin{array}{ll} \min & \mu_1 \cdot P + \mu_2 \cdot A \\ s.t. & \text{Stage Functionality} \\ & slewRate > slewRate_{Spec} \\ & t_{hold} < t_{hold,Spec} \\ & nPO < nPO_{Spec} \\ & C_L = C_{L,Spec} \end{array}$$

To generate the Pareto-optimal set, each spec is swept to span the entire practical design space. In sweeping the specs, we increased the spec on *nPO* for increasing specs on *slewRate*, thereby imposing a dependency between the two variables, which mimics the physical problem constraints. This is an example of designer insight improving the efficiency of the H-BU method. The code for generating the Pareto-optimal surfaces is in Appendix C.1 and C.2.

Figure 4-1 shows plots of the Pareto-optimal surface for a pipeline stage. In both figures all four variables, nPO, τ_{hold}, slewRate, and C_L, are swept, and the results are plotted on two different sets of axes. The Pareto surfaces are well behaved and thus apt.



(a) Pipeline stage Pareto surface I showing the design space and tradeoffs between nPO, slewRate, and power for each pipeline stage.



(b) Pipeline stage Pareto surface II showing the design space and tradeoffs between C_L, τ_{hold}, and power for each pipeline stage.

Figure 4-1. Pareto surfaces.

to monomial fitting. This is further evidence that equation-based optimization is suitable at the system level, regardless of the method selected to generate the Pareto surface of each system block

4.2.2 Surface fitting

The Pareto surfaces obtained in Section 4.2.1 are well behaved and lend themselves to low order functions. Therefore, we can then fit a monomial function to describe the tradeoffs. A monomial function in n dimensions is of form

$$f(\mathbf{x}) = c\prod_{i=0}^{n} x_i^{a_i},$$

where $c > 0$ and $a_i \in \mathbf{R}$. Therefore, given a data set

$$\left(x^{(i)}, f^{(i)}\right), \quad i = 1,\ldots,N,$$

where $x^{(i)} \in \mathbf{R}^n$ are positive vectors and $f^{(i)}$ are positive constants, we can perform a logarithmic transformation on the data and use linear regression to solve for c and $a_1,\ldots,a_n$ such that $f(x^{(i)}) \approx f^{(i)}$.

In more detail, we define $y^{(i)} = \log x^{(i)}$ and replace $f(x^{(i)}) \approx f^{(i)}$ with $f(e^{y^{(i)}}) \approx \log f^{(i)}$ giving us

$$\log c + \sum_{k=1}^{n} a_k y_k^{(i)} \approx \log f^{(i)}, \quad i = 1,\ldots,N.$$

Using a least squares method, we can find a solution by minimizing the sum of the squared errors,

$$\sum_{i=1}^{N}\left(\log c + \sum_{k=1}^{n} a_k y_k^{(i)} - \log f^{(i)}\right)^2.$$

The code used to form this fitting is in Appendix C.3 and C.4.

Table 4-1 shows the mean relative errors of each monomial fit used in the hierarchical module in Section 4.1. The relative error is defined as

$$\text{relative error} = \frac{f^{(i)} - c\prod_{k=1}^{n} a_k y_k^{(i)}}{f^{(i)}}.$$

We see that all relative errors are less than 10%, which amplifies the point that the tradeoffs are well behaved and very amenable to simple monomial fits.

TABLE 4-I

MEAN RELATIVE ERRORS FOR MONOMIAL FITS

Design variable	Mean Relative Error (%)
power	3.4
area	1.9
cLH	8.2
cIS	7.9
nPO	3.5
τ_{hold}	5.8

Chapter 5

Optimization results

The pipeline was optimized to achieve a sampling frequency of 100 MHz with an SNR greater than 60 dB. The flat and hierarchical solutions can be found in Table 5-I along with the percentage difference between the solutions. For convenience only power and nPO are shown since these are the most familiar characteristics of a pipeline stage. Also included are the total power comparisons for both variables along with the flat-

TABLE 5-I

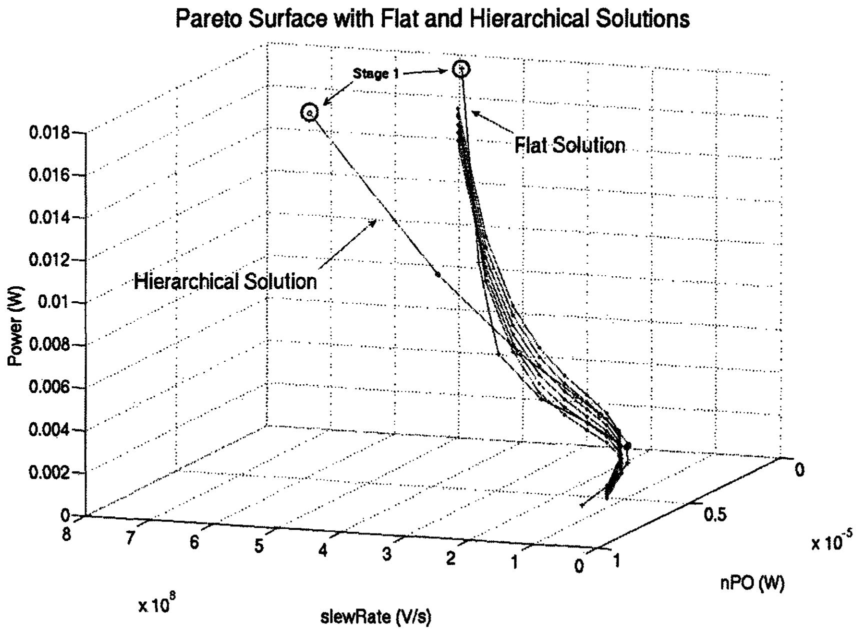
FLAT AND HIERARCHICAL SOLUTIONS

Stage	Power (mW)			nPO (uW)		
	Flat	Hierarchical	% difference	Flat	Hierarchical	% difference
1	17.44	14.83	15	0.05	0.03	30.4
2	8.34	7.62	8.6	0.09	0.08	6.8
3	4.12	4.26	3.5	0.17	0.19	10.2
4	2.19	2.49	13.6	0.31	0.41	34.9
5	1.32	1.78	34.8	0.52	0.65	24.8
6	0.9	1.28	42.3	0.93	1.04	11.7
7	0.69	0.94	36.1	1.52	1.65	8.8
8	0.58	0.92	57.7	2.64	1.73	34.7
9	0.52	0.92	77.9	4.54	1.73	62
10	0.39	0.77	98	6.69	1.48	77.9
Total:	36.49	35.81	1.9	0.156*	0.145*	7.6

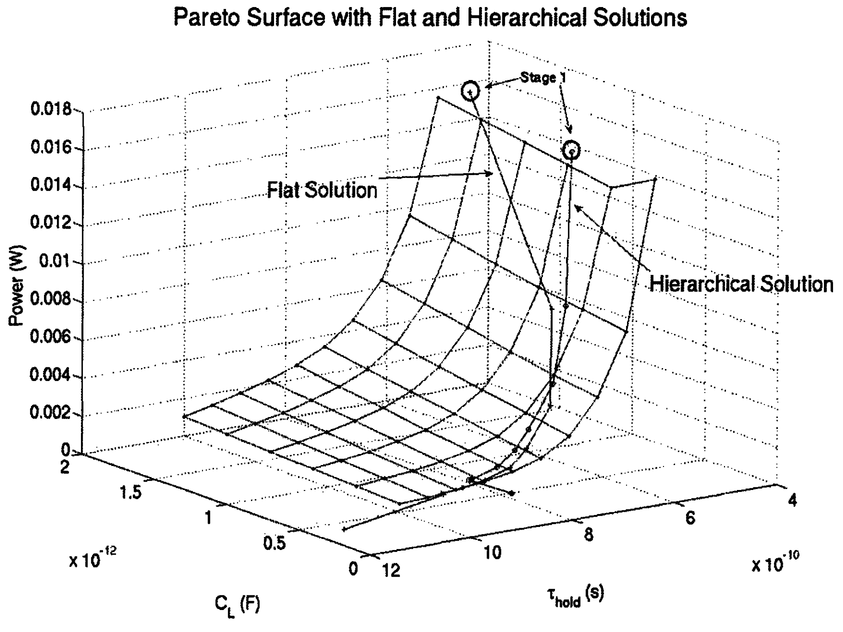
* Noise numbers for individual stages are output-referred noise; noise numbers for the whole system are referred to the system input.

hierarchical percentage difference. The total noise power is calculated as the pipeline input referred noise power contributed by all 10 stages.

Figure 5-1 plots the operating point of each stage in the pipeline. The solution lines plot the operating point of each pipeline stage where stage 1 consumes the most power and stage 10 the least. We can see the stage power decreases along the pipeline, which coincides with current design practice of decreasing stage performance, and thus the power, while still meeting the ADC performance specs. Though the percentage differences cited in Table 5-I are high for some of the performance specs in the later stages, looking at Figure 5-1 we can see that the hierarchical solution trends the same way as the flat solution and follows the design space. Also, the total power and noise power have only a 1.9% and 7.6% difference respectively, indicating that both distributions of power can meet the desired specifications.



(a) System solution plot I. Each solution line plots the operating point of each pipeline stage against the design space from Figure 4-1(a) where stage1 consumes the most power and stage10 the least.



(b) System solution plot II. Each solution line plots the operating point of each pipeline stage against the design space from Figure 4-1(b) where stage1 consumes the most power and stage10 the least.

Figure 5-1. System solution.

We optimized both the flat and hierarchical formulations for an increasing number of stages in the pipeline to demonstrate the speed improvement of a hierarchical formulation over a flat optimization. The optimizations were run on an Intel® Xeon™ 2.80GHz processor, and times can be seen in Figure 5-2. It is clear that the hierarchical optimization is faster than the flat and for 10 stages has a 25x improvement in optimize time.

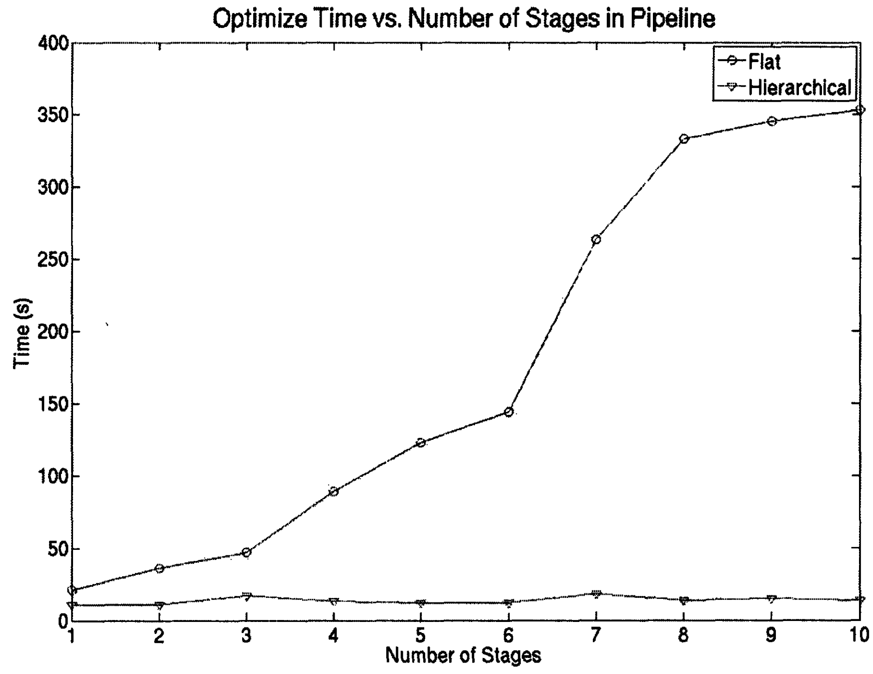


Figure 5-2. Optimize times for the flat and hierarchical optimizations.

This speed up is directly a cause of the slower increase of variables and constraints in the hierarchical formulation. Table 5-2 summarizes the number of variables and constraints in the flat and hierarchical formulations for an increasing number of stages in the pipeline. In the hierarchical case, the number of variables has a rate of increase 17x smaller than the flat case. This is primarily due to the behavioral characterization of the op-amp, so that instead optimizing hundreds of variables, we are only optimizing the performance specs of the op-amp. Similarly, the rate of increase of the number of constraints is 7x smaller.

TABLE II

FLAT AND HIERARCHICAL FORMULATION SIZES

Stages	Flat		Hierarchical	
	Number of Variables	Number of Constraints	Number of Variables	Number of Constraints
10	1563	2060	221	435
9	1421	1873	213	410
8	1279	1686	205	385
7	1137	1499	197	360
6	995	1312	189	335
5	853	1125	181	310
4	711	938	173	285
3	569	751	165	260
2	427	564	157	235
1	285	377	149	210

Chapter 6

Conclusions

System design heavily relies on seasoned designers for their extensive experience and intuition. However, they are not yet able to make system level tradeoffs on a purely quantitative basis. The reason for this is that although optimization methods have been used in circuit design, they are so far not scalable to truly large systems. The H-BU method applied here addresses this need by describing the design space using Pareto-optimal surfaces and abstracting the design tradeoffs to the system level creating a hierarchical optimization of the system. Because the Pareto surfaces are well behaved in nature, we can use low order functions in the abstraction resulting in a low complexity system optimization.

Application of the H-BU method was done to optimize a 10 stage pipelined ADC in a 0.18-um CMOS process and matched it with a flat optimization. Using equation-based optimization, we generated Pareto surfaces encompassing the entire design space of each stage in the pipeline. The surfaces were then quantitatively modeled with monomial fits, which all had less than a 10% relative error. The hierarchical optimization achieved a 25x improvement in optimization time compared to the flat optimization.

The value of the H-BU method lies in the Pareto-optimal surfaces. The surfaces provide a compact global perspective to system designers, and allow them to formulate low complexity system optimizations. We see that the proposed methodology restores the tractability of system-level design problems and is a powerful aid to designers of large, mixed-signal systems.

6.1 Future work

The next step in this work is to simulate both the flat and hierarchical solutions in a circuit simulator like SPECTRE. In doing so, we can show that in both cases the optimizations result in practical designs, meaning that the hierarchical optimization is a valid substitute for the flat optimization. This would require design of the pipeline's supporting circuits including common-mode feedback circuitry and a comparator. The formulation can also be extended to include the supporting circuits as well as models for the switches, which are considered ideal in this work. However, we have early evidence that the inclusion of these details wouldn't result in a significantly different design.

Long-term future work would be to apply the H-BU method to a more complex system like a complete receiver or a wireless link, where a flat optimization is not practical to implement. We can then use the hierarchical method to obtain a design, simulate the design in a circuit simulator, and fabricate an integrated circuit. This would be an example of the entire bottom-up system design process. Because of the generality of the H-BU method, we are not limited to optimizing only the circuits. For example with the wireless link, choice of modulation and coding strategies can be brought into the formulation. Furthermore, the optimization framework can be used to create a Pareto-surface for system performance. Thus given a system topology, the entire performance space of the system can be described instead of one or two design points.

An even longer term goal would be to port the optimization framework—including system formulation, Pareto surface generation, and simulation verification—to a different CMOS technology to show the design flexibility achieved with optimization techniques.

Appendix A

MATLAB flat formulation code

A.1 Top-level system code (ADC_top)

```
function ckt = ADC_top(name,spec)
% system level code

ckt = CKT_begin(name);

% Instantiating stages and SHA
CKT_subckt('SHA','sampleAndHold',spec);
CKT_subckt('stage1','pipelineStage',spec);
CKT_subckt('stage2','pipelineStage',spec);
CKT_subckt('stage3','pipelineStage',spec);
CKT_subckt('stage4','pipelineStage',spec);
CKT_subckt('stage5','pipelineStage',spec);
CKT_subckt('stage6','pipelineStage',spec);
CKT_subckt('stage7','pipelineStage',spec);
CKT_subckt('stage8','pipelineStage',spec);
CKT_subckt('stage9','pipelineStage',spec);
CKT_subckt('stage10','pipelineStage',spec);

% Writing power and area constraints
power = SHA.power + stage1.power + stage2.power + stage3.power ...
    + stage4.power + stage5.power + stage6.power ...
    + stage7.power + stage8.power + stage9.power ...
    + stage10.power ;

area = SHA.area + stage1.area + stage2.area + stage3.area ...
    + stage4.area + stage5.area + stage6.area ...
    + stage7.area + stage8.area + stage9.area ...
    + stage10.area ;

% Writing settling time and noise constraints
CKT_var tSettle;

% Instantiating 'connect' modules which contain settling time, noise, and interface
variable constraints
```

```
% The pipeline is broken into Hold-Sample blacks for circuit isolation and formulation
purposes

% Case where SHA is in Sample mode
CKT_subckt('SampleSHA' , 'connectSample', spec , SHA , 1 , tSettle, 2^(-11), 1);

% Case where SHA is in Hold mode and Stage 1 is in Sample mode
CKT_subckt('HoldSHASample1' , 'connectHoldAndSample', spec , SHA , stage1 , 1, tSettle,
2^(-11));
% Stage 1 is in Hold mode and Stage 2 is in Sample mode
CKT_subckt('Hold1Sample2' , 'connectHoldAndSample', spec , stage1 , stage2 , 2^1,
     tSettle, 2^(-10));
% Following pattern to Stage 9
CKT_subckt('Hold2Sample3' , 'connectHoldAndSample', spec , stage2 , stage3 , 2^2,
     tSettle, 2^(-9));
CKT_subckt('Hold3Sample4' , 'connectHoldAndSample', spec , stage3 , stage4 , 2^3,
     tSettle, 2^(-8));
CKT_subckt('Hold4Sample5' , 'connectHoldAndSample', spec , stage4 , stage5 , 2^4,
     tSettle, 2^(-7));
CKT_subckt('Hold5Sample6' , 'connectHoldAndSample', spec , stage5 , stage6 , 2^5,
     tSettle, 2^(-6));
CKT_subckt('Hold6Sample7' , 'connectHoldAndSample', spec , stage6 , stage7 , 2^6,
     tSettle, 2^(-5));
CKT_subckt('Hold7Sample8' , 'connectHoldAndSample', spec , stage7 , stage8 , 2^7,
     tSettle, 2^(-4));
CKT_subckt('Hold8Sample9' , 'connectHoldAndSample', spec , stage8 , stage9 , 2^8,
     tSettle, 2^(-3));
CKT_subckt('Hold9Sample10' , 'connectHoldAndSample', spec , stage9 , stage10 , 2^9,
     tSettle, 2^(-2));
% Case where Stage 10 is in Hold mode
CKT_subckt('Hold10' , 'connectHold', spec , stage10 , 2^10, tSettle, 2^(-1));
% Ensuring final stage load capacitance meets minimum capacitance requirements
CKT_constraint('final_stage_cload' , stage10.cLoadHold , '==' , spec.minCap);

% Summing noise from each scenario for total noisePower
noisePowerThermal= SampleSHA.noisePower ...
        + HoldSHASample1.noisePower ...
           + Hold1Sample2.noisePower ...
           + Hold2Sample3.noisePower ...
           + Hold3Sample4.noisePower ...
           + Hold4Sample5.noisePower ...
           + Hold5Sample6.noisePower ...
           + Hold6Sample7.noisePower ...
           + Hold7Sample8.noisePower ...
           + Hold8Sample9.noisePower ...
           + Hold9Sample10.noisePower...
        + Hold10.noisePower;

% Writing SNR constraints
LSB=spec.vMax/(2^(spec.numStages*spec.bitsPerStage));
noisePowerQuantization=(LSB/(12^0.5))^2;
noisePowerTot = noisePowerThermal +  noisePowerQuantization;
SNR=(spec.vMax^2/8)/noisePowerTot;

%assume we need to settle within half a period of the sampling frequency
fSample = .5/tSettle;

CKT_save  noisePowerThermal noisePowerQuantization noisePowerTot power area SNR;

% Add spec constraints
```

```
CKT_constraint('SNR' , spec.SNR , '<' , SNR );
CKT_constraint('fSample' , spec.fS , '<' , fSample);

% Add objective
CKT_objective('min', 'power' , 1e3*power + 1e9*area);
```

A.2 Pipeline stage code (pipelineStage)

```
function ckt = pipelineStage(name, spec);
% A stage in a pipeline ADC
% The noise and area of a stage in a pipelined ADC is
% expressed in terms of two design variables:
% C1 Sampling capacitor of the stage, and
% C2 Feedback capacitor of the stage.
% C=C1=C2
%

ckt = CKT_begin(name);

% Instantiating Op-amp and Capacitors
% We only need one capacitor because they are equal
CKT_subckt('C' , 'capADC' , spec);
CKT_subckt('Amp' , 'two_stage_half' , spec);

% Constraining Cap Sizes for circuit design
CKT_constraint('C_Amp.cIn', C.C, '>', 2*Amp.cIn);
CKT_constraint('C_minCap', C.C, '>', spec.minCap);

% Defining power ane area of stage
power = Amp.power;
area = 4 * C.area + Amp.area;
% Feedback factor in hold mode
fFB = C.C/(2*C.C + Amp.cIn);

% Defining time constant and slewRate of stage for settling time constraints
loopGain=2;
tauHold = loopGain/Amp.GBW;
slewRate = Amp.slewRate;

% Calculate node capacitances for interface interaction
CKT_var cLoadHold;
cInSample = 2 * C.C;
cFB = C.C*(C.C + Amp.cIn)/(2*C.C + Amp.cIn);
CKT_constraint('Amp_CL', Amp.CL, '==', cLoadHold +cFB);

% Defining noise of stage
noisePowerOut = (Amp.iNoise1Sq)/(4*Amp.gmIn*Amp.cOutStg1*fFB) +
     (Amp.iNoise2Sq)/(4*Amp.gmIn*(Amp.cOut + Amp.CC));
```

A.3 SHA code (sampleAndHold)

```
function ckt = sampleAndHold(name, spec);
% Sample and hold amplifier
```

```
%

% Instantiating Op-amp and Capacitor
ckt = CKT_begin(name);
CKT_subckt('C' , 'capADC' , spec);
CKT_subckt('Amp' , 'two_stage_half' , spec);

% Constraining Cap Sizes for circuit design
CKT_constraint('C_Amp.cIn', C.C, '>', 2*Amp.cIn);
CKT_constraint('C_minCap', C.C, '>', spec.minCap);

% Defining power ane area of stage
power = Amp.power;
area = 2*C.area + Amp.area;
% Feedback factor in hold mode
fFB = C.C/(C.C + Amp.cIn);

% Defining time constant and slewRate of stage for settling time constraints
loopGain=1;
tauHold = loopGain/Amp.GBW;
slewRate = Amp.slewRate;

% Calculate node capacitances for interface interaction
CKT_var cLoadHold;
cInSample = C.C;
cFB = C.C*Amp.cIn/(C.C + Amp.cIn);
CKT_constraint('Amp_CL', Amp.CL, '==', cLoadHold +cFB);

% Defining noise of stage
noisePowerOut = (Amp.iNoise1Sq)/(4*Amp.gmIn*Amp.cOutStg1*fFB) +
(Amp.iNoise2Sq)/(4*Amp.gmIn*(Amp.cOut + Amp.CC));
```

A.4 'Connect' codes

A.4.1 connectHoldAndSample

```
function ckt = connectHoldAndSample(name , spec , hStg, sStg, gSig, tSettle_var,
precision)
% hStg : hold stage
% sStg : sample stage
% gSig : signal voltage gain
% tSettle : settling time
% precision: settling precision
%

ckt = CKT_begin(name);

% Noise
noisePower = hStg.noisePowerOut/(gSig^2);
CKT_save noisePower;

% Interface capacitance constraint
CKT_constraint('hStg_cLoadHold', hStg.cLoadHold, '==', sStg.cInSample);

% Settling time constraints
tLinearSettling = -log(precision)*hStg.tauHold;
```

```
tSlew = .5*spec.vMax/hStg.slewRate;
tSettle = tLinearSettling + tSlew;
CKT_constraint('tSettle', tSettle, '<', tSettle_var);
```

A.4.2 connectSample

```
function ckt = connectSample(name , spec , sStg, gSig, tSettle_var, precision)
% hStg : hold stage
% sStg : sample stage
% gSig : signal voltage gain
% tSettle : settling time
% precision: settling precision
%

ckt = CKT_begin(name);

% Noise
noisePower = sStg.noisePowerOut/(gSig^2);
CKT_save noisePower;

% Settling time constraints
tLinearSettling = -log(precision)*sStg.tauHold;
tSlew = .5*spec.vMax/sStg.slewRate;
tSettle = tLinearSettling + tSlew;
CKT_constraint('tSettle', tSettle, '<', tSettle_var);
```

A.4.3 connectHold

```
function ckt = connectHold(name , spec , hStg, gSig, tSettle_var, precision)
% hStg : hold stage
% sStg : sample stage
% gSig : signal voltage gain
% tSettle : settling time
% precision: settling precision
%

ckt = CKT_begin(name);

% Noise
noisePower = hStg.noisePowerOut/(gSig^2);
CKT_save noisePower;

% Settling time constraints
tLinearSettling = -log(precision)*hStg.tauHold;
tSlew = .5*spec.vMax/hStg.slewRate;
tSettle = tLinearSettling + tSlew;
CKT_constraint('tSettle', tSettle, '<', tSettle_var);
```

A.5 Operational amplifier code

```
function ckt = two_stage_half(name, spec)
ckt = CKT_begin(name);

% Import process constants and parameters
const = physical_constants;
proc = get_process_params( spec.process);

%------------
% components
%------------
disp('Instantiating components..');
spec.no_ports = 1;
CKT_subckt( {'CA'}, 'capIdeal', spec);
CKT_subckt( {'M10', 'M17', 'M8', 'M34', 'M23', 'M19', 'M16'}, 'xstr_sat_inv', spec,
'nmos');
CKT_subckt( {'M15', 'M20', 'M13', 'M6', 'M2', 'M4', 'M32', 'M26', 'M18'}, 'xstr_sat_inv',
spec, 'pmos');

%---------------
% Ratio devices
%---------------
disp('Creating Ratio topology constraints');
CKT_var r1 r10 r2 r19 r16 r18 -discrete -independent;


% Ratio:M20,M13
%--------------------------
%CKT_ratio(r1, M20.nf, M13.nf);
CKT_constraint( '', M20.nf*r1, '==', M13.nf);
CKT_constraint( 'M20M4_Ratio_r23_9_M4_wf', M13.wf, '==', M20.wf);
CKT_constraint( 'M20M4_Ratio_r23_9_M4_lf', M13.lf, '==', M20.lf);
CKT_constraint( 'M20M4_Ratio_r23_9_M4_count', M13.count, '==', M20.count);

% Ratio:M10,M19
%--------------------------
%CKT_ratio(r1, M19.nf, M10.nf);
CKT_constraint( '', M19.nf*r10, '==', M10.nf);
CKT_constraint( 'M13M23_Ratio_r_M13_wf', M19.wf, '==', M10.wf);
CKT_constraint( 'M13M23_Ratio_r_M13_lf', M19.lf, '==', M10.lf);
CKT_constraint( 'M13M23_Ratio_r_M13_count', M19.count, '==', M10.count);

% Ratio:M16,M19
%--------------------------
%CKT_ratio( r16, M19.nf, M16.nf);
CKT_constraint( '', M19.nf*r16, '==', M16.nf);
CKT_constraint( 'M13M23_Ratio_r_M13_wf', M19.wf, '==', M16.wf);
CKT_constraint( 'M13M23_Ratio_r_M13_lf', M19.lf, '==', M16.lf);
CKT_constraint( 'M13M23_Ratio_r_M13_count', M19.count, '==', M16.count);

% Ratio:M20,M18
%--------------------------
%CKT_ratio( r18, M20.nf, M18.nf);
CKT_constraint( '', M20.nf*r18, '==', M18.nf);
CKT_constraint( 'M20M4_Ratio_r23_9_M4_wf', M18.wf, '==', M20.wf);
CKT_constraint( 'M20M4_Ratio_r23_9_M4_lf', M18.lf, '==', M20.lf);
CKT_constraint( 'M20M4_Ratio_r23_9_M4_count', M18.count, '==', M20.count);
```

```
% Ratio:M23,M19
%--------------------------
%CKT_ratio( r19, M23.nf, M19.nf);
CKT_constraint( '', M23.nf*r19, '==', M19.nf);
CKT_constraint( 'M23M10_Ratio_r23_9_M10_wf', M19.wf, '==', M23.wf);
CKT_constraint( 'M23M10_Ratio_r23_9_M10_lf', M19.lf, '==', M23.lf);
CKT_constraint( 'M23M10_Ratio_r23_9_M10_count', M19.count, '==', M23.count);


% Ratio:M26,M20
%--------------------------
%CKT_ratio( r2, M20.nf, M26.nf);
CKT_constraint( '', M20.nf*r2, '==', M26.nf);
CKT_constraint( 'M23M10_Ratio_r23_9_M10_wf', M26.wf, '==', M20.wf);
CKT_constraint( 'M23M10_Ratio_r23_9_M10_lf', M26.lf, '==', M20.lf);
CKT_constraint( 'M23M10_Ratio_r23_9_M10_count', M26.count, '==', M20.count);


%---------------------
% kcl for internal nets
%---------------------
% This section assumes differential symmetry from above
CKT_constraint('kclM23', M23.ids, '==', spec.iref);

CKT_constraint('kclM20', M20.ids, '==', M19.ids);
CKT_constraint('kclM15', M15.ids, '==', M16.ids);
CKT_constraint('kclM17', M17.ids, '==', M18.ids);

CKT_constraint('kclM13M2', M13.ids, '==', 2*M2.ids);
CKT_constraint('kclM4M6', M4.ids, '==', M6.ids);
CKT_constraint('kclM6M8', M6.ids, '==', M8.ids);
CKT_constraint('kclM10M8M2', M10.ids, '==', M2.ids + M8.ids);
CKT_constraint('kclM26M32', M26.ids, '==', 2*M32.ids);
CKT_constraint('kclM32M34', M32.ids, '==', M34.ids);


%---------------------
% kvl for internal nets
%---------------------
% This section assumes differential symmetry from above
% setting correct vsbs
CKT_constraint('M23vsb', M23.vsb, '==', 0);
CKT_constraint('M19vsb', M19.vsb, '==', 0);
CKT_constraint('M18vsb', M18.vsb, '==', 0);
CKT_constraint('M16vsb', M16.vsb, '==', 0);
CKT_constraint('M13vsb', M13.vsb, '==', 0);
CKT_constraint('M10vsb', M10.vsb, '==', 0);
CKT_constraint('M20vsb', M20.vsb, '==', 0);
CKT_constraint('M4vsb', M4.vsb, '==', 0);
CKT_constraint('M17vsb', M17.vsb, '==', 0);
CKT_constraint('M15vsb', M15.vsb, '==', 0);
CKT_constraint('M26vsb', M26.vsb, '==', 0);
CKT_constraint('M34vsb', M34.vsb, '==', 0);

CKT_constraint('M2vsb', M2.vsb, '==', M13.vds);
CKT_constraint('M6vsb', M6.vsb, '==', M4.vds);
CKT_constraint('M8vsb', M8.vsb, '==', M10.vds);
CKT_constraint('M32vsb', M32.vsb, '==', M26.vds);

% transistor stacks
```

```
CKT_constraint('kvl_stack_M13M2M10', M13.vds + M2.vds + M10.vds, '==', spec.vdd);
CKT_constraint('kvl_stack_M10M8M6M4', M10.vds + M8.vds + M6.vds + M4.vds, '==',
spec.vdd);
CKT_constraint('kvl_stack_M26M32M34', M26.vds + M32.vds + M34.vds, '==', spec.vdd);
CKT_constraint('kvl_stack_M19M20', M19.vds + M20.vds, '==', spec.vdd);
CKT_constraint('kvl_stack_M18M17', M18.vds + M17.vds, '==', spec.vdd);
CKT_constraint('kvl_stack_M16M15', M15.vds + M16.vds, '==', spec.vdd);

% diode connections
CKT_constraint('M23_diode', M23.vgs, '==', M23.vds);
CKT_constraint('M15_diode', M15.vgs, '==', M15.vds);
CKT_constraint('M17_diode', M17.vgs, '==', M17.vds);
CKT_constraint('M20_diode', M20.vgs, '==', M20.vds);

% gate biasing
CKT_constraint('M6_vgs', M6.vgs + M4.vds, '==', M15.vgs);
CKT_constraint('M8_vgs', M8.vgs + M10.vds, '==', M17.vgs);
CKT_constraint('M10_vgs', M10.vgs, '==', M23.vgs);
CKT_constraint('M13_vgs', M13.vgs, '==', M20.vgs);
CKT_constraint('M16_vgs', M16.vgs, '==', M23.vgs);
CKT_constraint('M18_vgs', M18.vgs, '==', M20.vgs);
CKT_constraint('M19_vgs', M19.vgs, '==', M23.vgs);
CKT_constraint('M26_vgs', M26.vgs, '==', M20.vgs);

% M4.vgs and M34.vgs left floating for cmfb, but we still bound them
CKT_constraint('cmfb1Lower', M4.vgs, '>', 0);
CKT_constraint('cmfb1Upper', M4.vgs, '<', spec.vdd);
CKT_constraint('cmfb2Lower', M34.vgs, '>', 0);
CKT_constraint('cmfb2Upper', M34.vgs, '<', spec.vdd);

% stage connections
CKT_var vout1;
CKT_bounds vout1 .1 1.7;
CKT_constraint('outputStg1', M8.vds + M10.vds, '==', vout1);
CKT_constraint('inputStg2', M26.vds + M32.vgs, '==', spec.vdd - vout1);

% input voltage
CKT_constraint('input', M13.vds + M2.vgs, '==', spec.vdd*(1-spec.vcmFactor));

% output voltage
CKT_constraint('nmos_output_bias', M34.vds, '==', spec.vdd*(spec.voutFactor));
CKT_constraint('pmos_output_bias', M26.vds + M32.vds, '==',spec.vdd*(1-spec.voutFactor));


%------
% Power
%------
powerBias = spec.vdd*(spec.iref + M19.ids + M17.ids + M15.ids);
powerStg1 = spec.vdd*(2*M10.ids);
powerStg2 = spec.vdd*(M26.ids);
power = powerBias + powerStg1 + powerStg2;


%------
% Area
%------
areaBias = M16.area + M17.area + M18.area + M19.area + M23.area + M15.area + M20.area;
areaStg1 = M13.area + 2*(M2.area + M4.area + M6.area + M8.area + M10.area);
areaStg2 = M26.area + 2*(M32.area + M34.area);
areaCap = 2*CA.area;
```

```
area = areaBias + areaStg1 + areaStg2 + areaCap;

%
%--------------------------------
% DEFINE SPECIFICATIONS (TOPLEVEL)
%--------------------------------
%------
% Gain
%------
disp('writing small signal constraints ..... ');
gmInStg1 = M2.gm; %differential transconductance
gmIn = gmInStg1;
CKT_var gOutStg1Nmos gOutStg1Pmos;
CKT_constraint('gOutStg1Nmos' , (M2.gds+M10.gds)*M8.gds/M8.gm , '<' , gOutStg1Nmos );
CKT_constraint('gOutStg1Pmos' ,  M4.gds*M6.gds/M6.gm , '<' , gOutStg1Pmos );
gOutStg1 = gOutStg1Nmos + gOutStg1Pmos; %output conductance of the first stage

gmInStg2 = M32.gm;
gOutStg2 = M34.gds + M32.gds + 1/spec.RL; %output conductance of the second stage

gainStg1 = gmInStg1/gOutStg1;
gainStg2 = gmInStg2/gOutStg2;


sblGain = gainStg1*gainStg2;  %DC gain


%------
% Gain-Bandwidth
%------
disp('adding gain-bandwidth constraint');
CKT_var CC -independent;  % Compensation capacitor for stability
CKT_constraint('CC' , CC , '==' , CA.capacitance);

sblGBWHz=1/(2*pi)*gmIn/CC;
GBW = sblGBWHz*2*pi;


% capacitances at nodes
% Introduce variables for the capacitances and define with parasitic capacitances

cIn = M2.cgs + M2.cgd;
CKT_var CL;
cOutStg1 = M8.cdb + M8.cgd + M6.cdb + M6.cgd + M32.cgs + M32.cgd;
cOut = CL + M32.cdb + M32.cgd + M34.cdb + M34.cgd;
cOutCasN = M10.cdb + M10.cgd + M8.cgs + M8.csb + M2.cdb + M2.cgd;
cOutCasP = M4.cdb + M4.cgd + M6.csb + M6.cgs;

% Phase margin qualities are evaulated
% nonDomTauPoles solution of s^2*d3+s*d2+d1=0
% tan(Phase Shift) = GBW*(d2/d1)/(1-GBW^2*d3/d1) = GBW*nonDomTauPoleEff
rd2d1 = cOutStg1*(1+cOut/CC)/gmInStg2;
rd3d1 = cOutStg1*(cOut*(1+cOutCasN/CC)+cOutCasN)/(M8.gm*gmInStg2);
nonDomTauPoleEff = (rd2d1)/(1-GBW^2*rd3d1);
sbltanPhaseShift = GBW*nonDomTauPoleEff;

% constraints for the other non-dominant poles
% Ensure poles from cascode nodes are non-dominant
CKT_constraint('nondom_cOutCasP' , GBW*cOutCasP/M6.gm , '<' , 0.5);
CKT_constraint('nondom_cOutCasN' , GBW*cOutCasN/M8.gm , '<' , 0.5);
```

```
% ensure no peaking
CKT_constraint('no_peaking',8*GBW^2*(1+cOutCasN/CC)*(1+cOut/CC)*(CC*cOutStg1)/(M8.gm*gmIn
Stg2),'<',1);

%------
% noise
%------
disp('adding noise power');
iNoise1Sq = (2*M2.ids_nd+2*M10.ids_nd+2*M4.ids_nd);
iNoise2Sq = (2*M32.ids_nd+2*M34.ids_nd);

%------
% slew rate
%------
disp('adding slew rate constraint');
sblSlewRateStg1 = M8.ids/(cOutStg1);
sblSlewRateStg2 = 0.5*M26.ids/(cOut);

CKT_var slewRate
CKT_constraint('slewRateStg1', sblSlewRateStg1, '>', slewRate);
CKT_constraint('slewRateStg2', sblSlewRateStg2, '>', slewRate);


disp('adding DC gain constraint');
CKT_constraint('gainDCSpec' , spec.gainDC , '<' , sblGain);

disp('adding stability constraint');
%phase margin of 60 degrees or shift of 30 degrees
CKT_constraint('phaseMargin', sbltanPhaseShift, '<', tan(pi/2 - 60/180*pi));
```

Appendix B

Pipeline stage module (hier_stage)

```
function ckt = hier_stage(name, spec);
% An hierarchical stage module%

ckt = CKT_begin(name);

load model_fits;

CKT_var sblArea noisePowerOut cLoadHold cInSample tauHold slewRate CL sblPower;

CKT_constraint('power', sblPower,'==', power_c * sblArea^power_a(1) *
     noisePowerOut^power_a(2) * cInSample^power_a(3) * cLoadHold^power_a(4) *
     tauHold^power_a(5) * slewRate^power_a(6) * CL^power_a(7));

CKT_constraint('area', sblArea,'==', area_c * noisePowerOut^area_a(1) *
     cInSample^area_a(2) * cLoadHold^area_a(3) * tauHold^area_a(4) * slewRate^area_a(5) *
     CL^area_a(6));

CKT_constraint('cLoadHold', cLoadHold,'==', cLoadHold_c * noisePowerOut^cLoadHold_a(1) *
     cInSample^cLoadHold_a(2) * tauHold^cLoadHold_a(3) * slewRate^cLoadHold_a(4) *
     CL^cLoadHold_a(5));

CKT_constraint('cInSample', cInSample,'==', cInSample_c * noisePowerOut^cInSample_a(1) *
     tauHold^cInSample_a(2) * slewRate^cInSample_a(3) * CL^cInSample_a(4));

CKT_constraint('nPO', noisePowerOut, '==', noisePowerOut_c * tauHold^noisePowerOut_a(1) *
     slewRate^noisePowerOut_a(2) * CL^noisePowerOut_a(3));

CKT_constraint('tauHold', tauHold, '==', tauHold_c * slewRate^tauHold_a(1) *
     CL^tauHold_a(2));

CKT_bounds sblPower 100e-6 100e-3;
CKT_bounds sblArea 1e-12 1e-3;
CKT_bounds noisePowerOut 1e-9 1e-3;
CKT_bounds cLoadHold 200e-15 1e-12;
CKT_bounds cInSample 400e-15 2e-12;
CKT_bounds tauHold 1e-12 1e-6;
CKT_bounds slewRate 1e6 1e9;
CKT_bounds CL 200e-15 2e-12;
```

Appendix C

Pareto-optimal surface generation

C.1 Pipeline stage with objective (pipelineStage_objective)

```
function ckt = pipelineStage_objective(name, spec);
% A stage in a pipeline ADC with objective for optimization of single stage
% The noise and area of a stage in a pipelined ADC is
% expressed in terms of two design variables:
% C1 Sampling capacitor of the stage, and
% C2 Feedback capacitor of the stage.
% C=C1=C2
%

ckt = CKT_begin(name);

% Instantiating Op-amp and Capacitors
% We only need one capacitor because they are equal
CKT_subckt('C' , 'capADC' , spec);
CKT_subckt('Amp' , 'two_stage_half' , spec);

% Constraining Cap Sizes for circuit design
CKT_constraint('C_Amp.cIn', C.C, '>', 2*Amp.cIn);
CKT_constraint('C_minCap', C.C, '>', spec.minCap);

% Defining power ane area of stage
power = Amp.power;
area = 4 * C.area + Amp.area;
% Feedback factor in hold mode
fFB = C.C/(2*C.C + Amp.cIn);

% Defining time constant and slewRate of stage for settling time constraints
loopGain=2;
tauHold = loopGain/Amp.GBW;
slewRate = Amp.slewRate;

% Calculate node capacitances for interface interaction
CKT_var cLoadHold;
cInSample = 2 * C.C;
```

```
cFB = C.C*(C.C + Amp.cIn)/(2*C.C + Amp.cIn);
CKT_constraint('Amp_CL', Amp.CL, '==', cLoadHold +cFB);

% Defining noise of stage
noisePowerOut = (Amp.iNoise1Sq)/(4*Amp.gmIn*Amp.cOutStg1*fFB) +
     (Amp.iNoise2Sq)/(4*Amp.gmIn*(Amp.cOut + Amp.CC));

% Bound design variables
CKT_constraint('min_power', power, '>', 1e-9);
CKT_constraint('max_power', power, '<', 1e3);

CKT_constraint('min_area', area, '>', 1e-18);
CKT_constraint('max_area', area, '<', 1e-6);

CKT_constraint('min_cLoadHold', cLoadHold, '>', spec.minCap);
CKT_constraint('max_cLoadHold', cLoadHold, '<', 1e-9);

CKT_constraint('min_cInSample', cInSample, '>', 2*spec.minCap);
CKT_constraint('max_cInSample', cInSample, '<', 1e-9);

% Modify specs with new value for sweeping
CKT_constraint('min_nPO', noisePowerOut, '>', 1e-9);
CKT_constraint('max_nPO', noisePowerOut, '<', 1e-3);
CKT_constraint('nPOSpec', noisePowerOut, '<', spec.lambda1*spec.nPO);

CKT_constraint('min_tauHold', tauHold, '>', 1e-12);
CKT_constraint('max_tauHold', tauHold, '<', 1e-9);
CKT_constraint('tauHoldSpec', tauHold, '<', spec.lambda2*spec.tauHold);

CKT_constraint('min_slewRate', slewRate, '>', 1e6);
CKT_constraint('max_slewRate', slewRate, '<', 1e10);
CKT_constraint('slewRateSpec', slewRate, '>', spec.lambda3*spec.slewRate);

% Minimize power and area
objective = 1e2*power + 1e8*area;
CKT_objective('min', 'objective', objective);
```

C.2 Sweep constraints to get set of Pareto-optimal designs (tradeoff_pipelineStage)

```
% use lambda as weights on the constraints to tighten or loosen them
clear all;
close all;

% create vector of CL values to be swept
CLspecs=CLmin:CLstep:CLmax;

% create arrays to store pareto-optimal points
power_array=[];
area_array=[];
noisePowerOut_array=[];
cLoadHold_array=[];
cInSample_array=[];
tauHold_array=[];
slewRate_array=[];
```

```
CL_array=[];


for k=1:length(CLspecs)
     lambda=lambdaGen; % generates the set of specs to sweep in a 3 row matrix
     load working_specs; %loads default specs

     % modify specs with CL value and correct lambda vector
     specs.CL=CLspecs(k);
     for i=1:size(lambda, 2)
          specs.lambda1=lambda(1,i);
          specs.lambda2=lambda(2,i);
          specs.lambda3=lambda(3,i);

          % optimize pipeline stage
          res=sosRun('pipelineStage_tradeoff', 'specs', specs, 'OpenBrowser', 'off');
          sosAssign(res(1), 'variables', '-force');

          % store optimal result as point on pareto-optimal surface
          if (strcmp(res.status, 'OPTIMAL'))
               power_array=[power_array power];
               area_array=[area_array area];
               noisePowerOut_array=[noisePowerOut_array noisePowerOut];
               cLoadHold_array=[cLoadHold_array cLoadHold];
               cInSample_array=[cInSample_array cInSample];
               tauHold_array=[tauHold_array tauHold];
               slewRate_array=[slewRate_array slewRate];
               CL_array=[CL_array specs.CL];
          end

     end
end

save data
```

C.3 Fit Pareto-optimal data set with monomial (processData)

```
function processData(filename)

% Load data file with pareto-optimal set
load(filename)

ind=1:length(power_array);

% perform monomial fit on power
% power=power_c*area^power_a(1)*noisePowerOut^power_a(2)*cInSample^power_a(3)*
% tauHold^power_a(4)*slewRate^power_a(5)*CL^power_a(6)
grd=[area_array(ind);noisePowerOut_array(ind);cInSample_array(ind);cLoadHold_array(ind);
     tauHold_array(ind);slewRate_array(ind);CL_array(ind)];
[power_c,power_a,power_fit]=monomial_fit(power_array(ind), grd);

% perform monomial fit on area
grd=[noisePowerOut_array(ind);cInSample_array(ind);cLoadHold_array(ind);
     tauHold_array(ind);slewRate_array(ind);CL_array(ind)];
```

```
[area_c,area_a,area_fit]=monomial_fit(area_array(ind), grd);

% perform monomial fit on cLoadHold
grd=[noisePowerOut_array(ind);cInSample_array(ind);tauHold_array(ind);
     slewRate_array(ind);CL_array(ind)];
[cLoadHold_c,cLoadHold_a,cLoadHold_fit]=monomial_fit(cLoadHold_array(ind), grd);

% perform monomial fit on cInSample
grd=[noisePowerOut_array(ind);tauHold_array(ind);slewRate_array(ind);CL_array(ind)];
[cInSample_c,cInSample_a,cInSample_fit]=monomial_fit(cInSample_array(ind), grd);

% perform monomial fit on noisePowerOut
grd=[tauHold_array(ind);slewRate_array(ind);CL_array(ind)];
[noisePowerOut_c,noisePowerOut_a,noisePowerOut_fit]=
     monomial_fit(noisePowerOut_array(ind), grd);

% perform monomial fit on CL
grd=[slewRate_array(ind);CL_array(ind)];
[tauHold_c, tauHold_a, tauHold_fit]=monomial_fit(tauHold_array(ind), grd);

save model_fits_conservative power_c power_a area_c area_a cInSample_c cInSample_a
     cLoadHold_c cLoadHold_a noisePowerOut_c noisePowerOut_a
```

C.4 Monomial best fit code (monomial_fit)

```
function [c,a,fit_data]=monomial_fit(data,grid)
%
% This should do monomial fit for provided
% data to given grid (sample) set ...
%
% Returns weights and exponents
%
% It gives c.*x.^a monomial params and
% [c,a,my_data]=monomial_fit(data,grid)

% extract only positive data
ind=find(data>0);
for jj=1:size(grid,1)
    ind1=find(grid(jj,:)>0);
    ind=intersect(ind,ind1);
end

log_data=log(data(:,ind));
log_grid=log(grid(:,ind));
% we add 1-column to treat c the same way as alpha
effective_grid=[ones(1,size(log_grid,2)); log_grid];

% alpha=sdpvar(size(effective_grid,1),1);

% Performs a best fit on data with minimal error
X=[ones(size(log_grid,2),1) log_grid'];
alpha=X\log_data';

c=exp(alpha(1,:));
a=alpha(2:length(alpha),:);

% rebuilding data and taking care of zeros
```

```
fit_data=c;
for i=1:size(grid,1)
    temp_data=grid(i,:).^a(i);
    fit_data=fit_data.*temp_data;
end
```

Bibliography

[1] S. Boyd and L. Vandenerghe. (2004) Introduction to convex optimization with engineering applications. [Online]. Available: http://www.stanford.edu/~boyd/cvxbook/

[2] E. Zitzler, "Evolutionary algorithms for multi-objective optimization: Methods and applications," PhD thesis, Swiss Federal Institute of Technology, Zurich, Switzerland, Nov. 1999.

[3] J. Zou, D. Mueller, H. Graeb, and U. Schlichtmann, "A *CPPLL* hierarchical optimization methodology considering jitter, power and locking time," *Design Automation Conference*, San Francisco, CA, pp. 19–24. July 2006.

[4] M. Hershenson, S. Boyd, and T. Lee, "Optimal design of a CMOS op-amp via geometric programming," *IEEE Transactions CAD*, vol. 20, pp. 1–21, Jan. 2001.

[5] T. Eekeleart, T. McConaghy, and G. Gielen, "Efficient multiobjective synthesis of analog circuits using hierarchical Pareto-optimal performance hypersurfaces," in *Proceedings of the 42nd annual conference on Design Automation and Test in Europe Conference,* pp. 1070–1075, June 2005.

[6] T. Eekeleart, R. Schoofs, G. Gielen, M. Steyeart, and W. Sansen, "Hierarchical bottom-up analog optimization methodology validated by a delta-sigma A/D converter design for the 802.11a/b/g standard," *Design Automation Conference*, San Francisco, CA, pp. 25–30, July 2006.

[7] F. Bernardinis, P. Nuzzo, and A. Vincentelli, "Robust system level design with analog platforms," *IEEE/ACM ICCAD*, San Jose, CA, Nov. 2006.

[8] M. Hershenson, "Design of pipeline analog-to-digital converters via geometric programming," *IEEE/ACM ICCAD*, pp. 317–324, Nov. 2002.

[9] M. Hershenson, S. Boyd, and T. Lee, "Efficient description of the design space of analog circuits," *IEEE/ACM DAC*, June 2003.

[10] X. Li, J. Wang, L. Pileggi, T.-S. Chen, and W. Chiang, "Performance-centering optimization for system-level and analog design exploration," *IEEE*, pp. 421-428, 2005.

[11] Sabio Home Page. Sabio Labs, Inc. Retrieved 18 Oct. 2007 <http://www.sabiolabs.com>.

[12] W. Sanchez. 2007. A Hierarchical, bottom-up, equation-based optimization design methodology. Masters Thesis, Massachusetts Institute of Technology. 82 p.

[13] B. Razavi. *Design of Analog CMOS Integrated Circuits.* McGraw-Hill, 2000.

[14] D. Johns, K. Martin. *Analog Intgrated Circuit Design.* John Wiley & Sons, Inc. 2004.

[15] T. Lee. *The Design of CMOS Radio-Frequency Integrated Circuits.* Cambridge University Press. 1998.

[16] B. K. Ahuja, "An improved frequency compensation technique for CMOS operational amplifiers," *IEEE Journal of Solid-state Circuits,* vol. sc-18, no. 6, pp. 629-633, 1983.

[17] D. R. Ribner, M. A. Copeland, "Design techniques for cascaded CMOS op amps with improved PSRR and common-mode input range," *IEEE Journal of Solid-state Circuits,* vol. sc-19, pp. 919-925, 1984.

[18] R. Tadeparthy, "An improved frequency compensation technique for low power, low voltage CMOS amplifiers," *IEEE ISCAS,* pp. 497-501, 2004.

[19] M. Yavari, O. Shoaei, F. Svelto, "Hybrid cascode compensation for two-stage

CMOS operational amplifiers," *IEEE ISCAS*, pp. 1565-1568, 2005.

[20] G. Gielen, T. McConaghy, T. Eeckelaert, "Performance space modeling for hierarchical synthesis of analog integrated circuits," *IEEE/ACM DAC*, pp. 881-885, June 2005.

[21] S. K. Tiwary, P. K. Tiwary, R. A. Rutenbar, "Generation of yield-aware pareto surfaces for hierarchical circuit design space exploration," *IEEE/ACM DAC*, pp. 31-36, July 2006.