Circuit Level Synthesis for Delta-Sigma Converters

by

Mark Shane Peng

B.S. Electrical Engineering and Computer Science
University of California at Berkeley (1997)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

January 1999
[February, 1999]

Author .....
Department of Electrical Engineering and Computer Science
January, 1999

Certified by .....
Hae-Seung Lee
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Circuit Level Synthesis for Delta-Sigma Converters

by

Mark Shane Peng

Submitted to the Department of Electrical Engineering and Computer Science
on January, 1999, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

Analog circuit design is a relatively complicated art that requires a high degree of erudition in the field. With this in mind, this thesis presents work on an analog circuit synthesis tool to minimize analog circuit design time. Specifically, the author has designed and implemented a discrete-time, one-bit, oversampling delta-sigma analog-to-digital modulator circuit synthesis tool in MATLAB script. With the parameters of center frequency, loop order, oversampling ratio, and minimum capacitor size, a user can utilize the program to generate a semi-optimized transistor level description of the modulator that can subsequently be used in SPICE. Parlaying Richard Schreier's work on a delta-sigma toolbox for MATLAB, the switched capacitor circuit contains robustly generated differential operational amplifiers and comparators. Furthermore, switched capacitors are scaled for minimal kT/C noise while switch sizes are synchronously adjusted to accommodate these values. Results of the SPICE simulations of the generated circuits compare favorably with the behaviorally predicted results.

Thesis Supervisor: Hae-Seung Lee
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

I would first and foremost acknowledge Professor Hae-Seung Lee for his faith in taking me on as a graduate student. He has helped me in countless ways in providing sagacious guidance, timeless wisdom, and trenchant insights. In working with him, I have truly been enriched.

Next, I would like to acknowledge my family. Although far away, thoughts of them pervade my mind everyday. Strength and equanimity springs from knowledge of their well-being and health. I hope and wish for their reciprocating happiness always.

Then, I would like to acknowledge my fellow research group members who have provided countless insights and solutions to my seemingly incessant and insurmountable problems and musings. Learning is veritably a group effort. Even more, the surroundings of intelligence and brilliance redound to my own aspirations.

Last of all, but not least, I want to thank all of my friends who have been the salve and balm for a seemingly untenable dour life. Through the travails and vicissitudes or elations and euphorias of life, there will always be friends. Friends who comfort, friends to converse, and friends to enjoy.

I must say, "Lucky is my life."

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Analog circuit design has always been a particularly difficult field owing to its complexity. Relationships between parameters and system performance in an analog circuit are numerous and abstruse, if not altogether impossibly unquantifiable. Only an experienced designer with few semi-quantitative equations and much intuition can swiftly and deftly design and build an analog circuit that will perform as desired. Since this requires a high degree of skill and even more experience, completion time is relatively long. Thus, it is not surprising that although the analog portion of a microchip may only occupy 10% of the area, it will typically consume 80-90% of the design time.

With this in mind, it would only be natural that CAD tools would be developed to ameliorate the effects of this problem. However, the design and synthesis tools produced to date have been woefully impractical and lacking in feasibility in comparison to their digital counterparts.

Therefore, this thesis seeks to produce an analog tool that is constrained enough so as to produce usable circuits in a short time. More particularly, this thesis will address the design of delta-sigma analog to digital converters for use by people that do not need or want to understand the inner workings, but need a semi-custom analog-to-digital converter. In the inevitable tradeoff of less performance for increased generality, the work of this thesis uses user parameters in a constrained optimization method to generate a robust switched-capacitor one-bit oversampled delta-sigma modulator.

1.1 Background

To date, the success of analog circuit synthesis has not been auspicious. Either the circuit generated is impractical or it takes an extremely long time to synthesize. And more often than not, the process suffers from both.

Inherent to the problem is that analog circuits are very sensitive to minor changes. Whether it be bias current, capacitor size, process parameters, or transistor width, a small change can produce a dramatic effect if care is not taken. Moreover, it is almost certain that the effect will negatively impact performance as experience shows. Because it is so complicated, most attempts at analog synthesis use iteration to find an acceptable design[1, 2, 3, 4].

The general method taken by these tools is to form a parameter space with all the parameters that can be varied for a design[1]. Then using the multitudinous relationships between parameters and performance characteristics, the programs mechanically iterate by

changing parameters and comparing performance characteristics obtained from simulations. The actual algorithm for iteration can be quite complex and elaborate. Compounding the problem is the criteria for an acceptable solution. As an example, Koza[3] uses "genetic programming", akin to nature's process of evolution, to arrive at a "solution."

Moreover, the success of these tools seems to hinge upon numerical methods rather than the design itself[2, 4]. These programs can take weeks or months on a blazing fast workstation to produce a mediocre design. Innovation continues to come from shortening the simulation time rather than the improving the method of design.

1.2 Thesis Motivation

Realizing that the problem common to these previous attempts at analog circuit synthesis is the generality with which they approach the problem, this thesis chooses to severely constrain the parameter space of acceptable solutions to the problem so a practical solution can be generated quickly. The focus is not to obtain the optimal solution, but a practical solution. In the event that the former is wanted, it would be best to employ a human, experienced designer.

The analog circuit which this thesis will generate is the one-bit delta-sigma oversampling analog to digital modulator. Its choice reflects the fact that the desired resolution is easily adjusted through loop order and oversampling ratio.

Currently, there is a need for analog to digital converters in many systems. Because of the underlying design issues, design of this subsystem could be time consuming when the performance of it is not so critical. In these cases, it would be useful to have a tool that could generate the requisite circuit with some predetermined specifications. That tool is the work of this thesis.

The one-bit delta-sigma oversampling architecture for the analog to digital modulator was chosen because it has many excellent characteristics. First of all, it is extremely robust. The architecture, inherently linear, is highly tolerant of imprecise components. Furthermore, it lends itself nicely to modularity which is crucial to synthesis. The individual blocks of the modulator are comprised of capacitors, operational amplifiers, switches, and comparators, each of which is synthesized and semi-optimized for minimum area, minimum power, and maximum resolution in real-time.

In addition, this tool, written in MATLAB script, uses Richard Schreier's Delta-Sigma 5.0 Toolbox[5] to generate system level descriptions. Schreier's toolbox does not perform circuit level synthesis which is the crux of this thesis.

Also, this tool does not generate a sample and hold front-end or a decimator, both of which would be needed to complete the analog to digital converter.

Finally, this project was simulated using the MOSIS HP 0.5μum process at supplies $\pm$1.65 V with ideal 8.3 MHz clock generators. Obviously, this tool can be made to work at different system parameters with minor adjustments of a few generation blocks in the tool.

1.3 Thesis Organization

This thesis is organized as follows. Chapter two gives an overview of the delta-sigma modulator operation and the process of modulator circuit generation. Chapter three discusses the translation of the chosen loop topology architecture to a circuit level schematic with capacitor values. Chapter four expatiates the switch scaling methodology for each switch

in the generated design. Chapter five describes the generation of the analog subcircuits, namely, the one-bit DAC, the one-bit quantizer, and the operational amplifiers. Chapter six discusses simulation results from behavioral models and actual SPICE simulation of the circuit generated. Chapter seven summarily draws conclusions and points to possible future work on this tool.

Chapter 2

Delta-Sigma Modulator Overview

The basic delta-sigma analog to digital converter system is shown in figure 2-1. The input circuitry is comprised of buffers, filters, and sample and holds. The delta-sigma modulator comes next which already inherently has a sample and hold function because of the switch capacitor circuit implementation. The final piece is the decimator which will filter the 1-bit output into a N-bit output. The 1-bit representation in N-bits is either all the digits are high, or all the digits are low. The focus of this thesis is the middle section, the delta-sigma modulator.

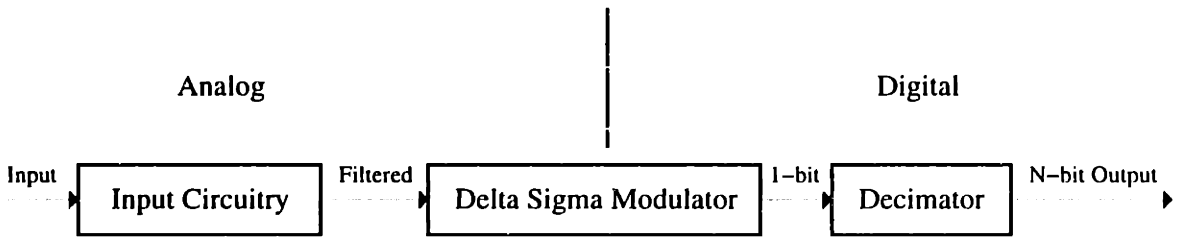


Figure 2-1: Block Diagram of a Delta-Sigma Analog-To-Digital Converter

2.1 Basic Operation and Assumptions of the Delta-Sigma Modulator

The generalized model of a single-bit delta-sigma modulator is shown in figure 2-2. It is essentially a feedback system that converts the analog input into a digital output.

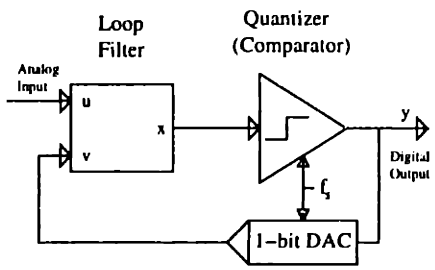


Figure 2-2: Generalized Model of a Single-Bit Δ-Σ Analog-To-Digital Modulator

For simple analysis and insight purposes, we can model the system in the form as seen in figure 2-3 where e is the quantization error. One will notice this looks remarkably like the canonical feedback system with e as the disturbance. Because the analog input and e enter in different places in the loop, the corresponding transfer function to the output y is different. By modulating the noise and signal differently, a designer can, in a sense, separate the quantization error and the input signal into different bands given that the sampling frequency is sufficient (i.e. sampling frequency must be greater than the Nyquist frequency). This is the cornerstone of delta-sigma modulators.

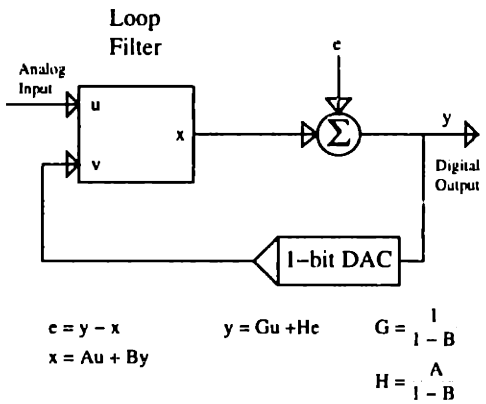


Figure 2-3: Linearized Model of a Single-Bit Delta-Sigma Analog-To-Digital Modulator

However, to exploit this feature, something must be said about the exact shape of the quantization error. Fortunately, it turns out that the quantization noise is in a manageable form. The usual model that describes the quantization noise is the additive white noise approximation which is outlined in detail in Delta-Sigma Data Converters[6, chap. 2].

The real impetus behind the use of delta-sigma modulators is the linearity it can achieve. This is especially true with single-bit quantization because the levels are digitally interpolated perfectly between the low and high levels. The only errors that can arise are a purely linear gain error and offset error which are not critical.

For treatment more in depth of the delta-sigma modulator basic operation, there exists a litany of sources[6, 7, 8, 9] that the reader can refer to.

2.2 Loop Topology Selection

From figure 2-3, we see that the design of a delta-sigma modulator is essentially the design of a loop filter. In fact, all techniques of filter design can be applied here. Popular and oft-used filters such as Butterworth and Chebyshev are common. Thus, it is not surprising that there are standard loop topologies[10].

The chosen topology for this is work is the cascade of resonators with feedback structure (CRFB)[5] or chain of integrators with distributed feedback and distributed feedforward inputs[6, pp. 179-180]. An example of a fifth order topology in block diagram form is shown in figure 2-4.

The main reason for selecting this topology is that it is readily supported in Schreier's tool[5]. Furthermore, it has the capability to be a bandpass or a low pass modulator which will be important for generality. In addition, the structure is extremely modular and easily

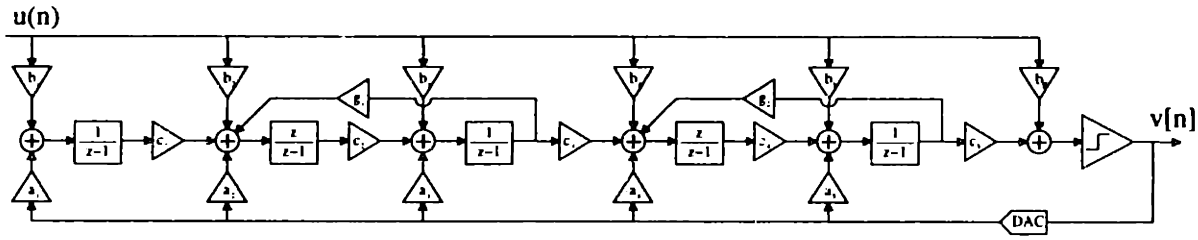


Figure 2-4: Loop Topology: 5th Order Example of Chain of Integrators with Distributed Feedback, Distributed Feedforward Input Paths and Local Resonator Feedbacks.

implemented with operational amplifiers, capacitors, switches, and comparators.

2.3 Generating Parts of the Modulator

The process of generating the modulator is a modular one given the loop filter architecture. Each part of the modulator is generated in the order and in the fashion described below.

2.3.1 Loop Transfer Function Generation

The standard approach to generating the modulator transfer function is to use standard filter design techniques to shape the quantization noise. In Schreier's MATLAB tool[5], the program seems to use an iterative method to place the poles and zeros, stopping when a heuristic criteria and desired performance are satisfied. This is valid as long as the transfer function is stable.

2.3.2 Dynamic Range Scaling

Because of the nonlinear nature and finite signal amplitudes an operational amplifier can handle, the gain of each opamp block must be scaled to achieve optimal dynamic range or maximum allowable input amplitude. The saturation of even one operational amplifier kills the performance of the modulator. Furthermore, these errors can accumulate and will take numerous cycles to recover from. Therefore, this part is a critical step in the process.

Since this is a switched-capacitor filter we can use the standard methods for dynamic range scaling[11]. This entails checking the output amplitude at each operational amplifier and adjusting the capacitors (gain) so that the maximum signal amplitude at each output is the same. The capacitors control the gain, and every capacitor attached to an operational amplifier output is scaled by the same factor to affect gain of that stage, but not the overall transfer function. In other words, if the feedback capacitor of an operational amplifier is scaled by α which corresponds to a gain change of $1/\alpha$, then the input capacitors of the next stage which are connected to this output are also scaled by α so that the gain is recovered. This process is repeated for each operational amplifier in the design.

Fortuitously and propitiously, Schreier's toolbox[5] incorporates this function and is thus used.

2.3.3 Block Diagram to Circuit Translation

Using the block diagram of the given architecture, a switched-capacitor circuit level equivalent can be found. The author chooses to use a completely differential design to reject common mode noise and more importantly, to make complementary signals readily available.

2.3.4 Transfer Function to Capacitor Translation

The tool uses Schreier's toolbox[5] again to transform the generated transfer function into coefficients for the CRFB form. The coefficients specify capacitor ratios and are thus implemented. Realization of capacitor values depend on the minimum size capacitor size. For the first summing node of each modulator, the minimum size capacitor is user specified for kT/C noise. The rest of the capacitors in the subsequent stages are at a minimum size of one hundred femtofarads because the associated noise is indistinguishable from quantization noise and shaped as such.

2.3.5 Switch Scaling

Since the tool uses pass transistor gates as switches, one must make sure the switches are large enough such that the capacitors that they are attached to settle to at least the desired accuracy of the whole converter. The accuracy is determined *a priori* with a simulation from the toolbox. In addition, to be conservative, safety factors are added to ensure the proper operation of the design.

2.3.6 DAC Generation

Because this tool uses single-bit quantization, the digital to analog converter needed is trivial to implement. Depending on the output, the feedback paths will be connected to either the positive or negative reference voltage through another properly sized transistor switch.

2.3.7 Comparator Generation

The comparator needs to settle within the period of a clock phase. Hence, the comparator is scaled depending on the load capacitance. The load capacitance it needs to drive—assuming the decimator input load is small—is just the gate-to-bulk and parasitic capacitances to the switches that make up the one-bit DAC which was generated previously. The generated design of the comparator is also conservative.

2.3.8 Operational Amplifier Generation

Now that all the elements are already sized and functional, the operational amplifiers can be synthesized. The operational amplifier synthesis is based on the capacitor load it must drive since it is crucial that the output settle in the given clock phase. Moreover, it must be able to handle the worst case load of the two clock phases. Again, a safety margin is introduced for a robust design.

2.4 Delta-Sigma Modulator Final Generation

Once all the parts are generated, they can be immediately put in the proper place as specified by the circuit level translation. The output of this program is in the form of a SPICE parsable file. The user can directly simulate this modulator to verify the performance and desired characteristics. The process flow of this generation is shown in figure 2-5.

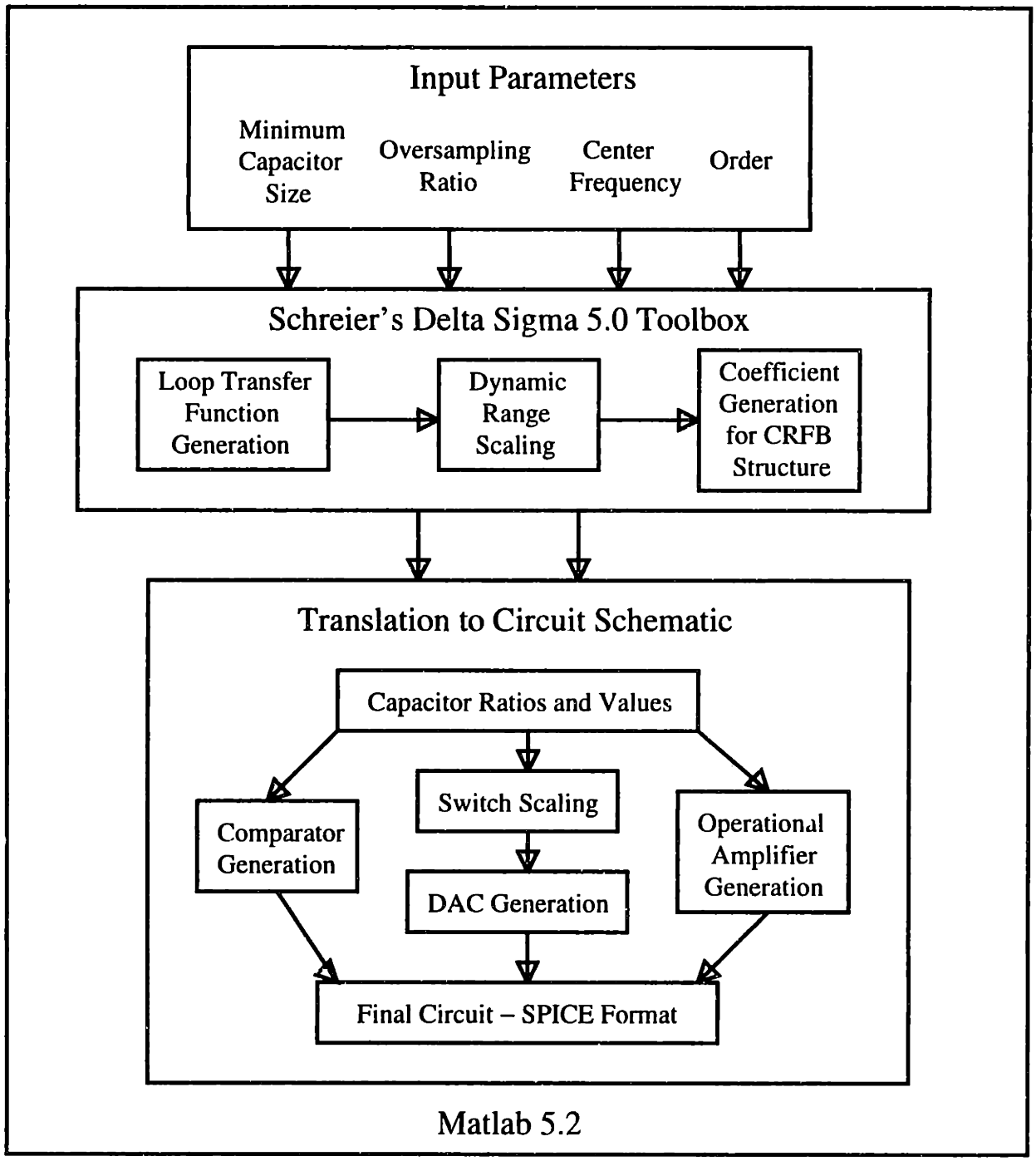


Figure 2-5: Process Flow for Generation of Delta-Sigma Modulator

Chapter 3

Modulator Topology to Circuit Level Translation

From Schreier's MATLAB toolbox[5], the tool of this thesis starts with scaled coefficients for the modulator topology of cascade of resonators with feedback as shown in the example of figure 2-4.

This topology can be generalized for any order by decomposing the architecture into blocks. The most natural and apparent division is to parse the architecture into a start section, cascade sections, and an end section. The start section implements two or three orders of the modulator depending on the order of the modulator. Each cascade section implements two orders of the modulator. Lastly, the end block contains the final feedforward path, the quantizer, and the digital to analog converter. Thus, any order can be realized. For example, a fifth order modulator would be composed of an odd order start section, one cascade section, and an end section. A sixth order modulator would be composed of an even order start section, two cascade sections, and an end section.

Within each section, further decomposition yields stages which implement an order of the loop filter. Each order corresponds to an operational amplifier in a discrete integrator configuration.

3.1 Integration and Summer Configurations

Figure 3-1 shows the implementation of the Delayless Integration which has the transfer function:

$$H(z) = -\frac{C_S}{C_I}\frac{z}{z-1} \tag{3.1}$$

Figure 3-2 shows the implementation of the Non-Inverting Direct Discrete Integration which has the transfer function:

$$H(z) = \frac{C_S}{C_I}\frac{1}{z-1} \tag{3.2}$$

These two are the integrator configurations that this tool needs. Furthermore, the summers are also implemented in the integrator configurations. Addition of sampling capacitors with appropriately phased switches to the summing junction perform this function because of superposition.

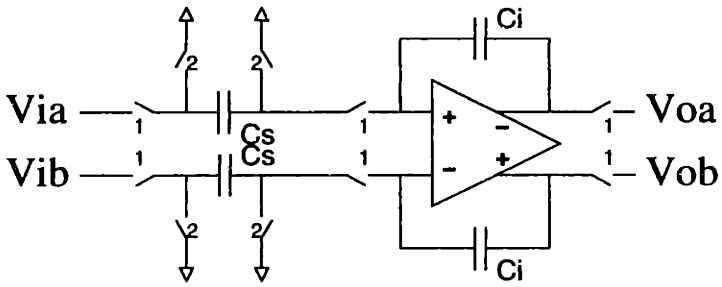


Figure 3-1: Delayless Integration

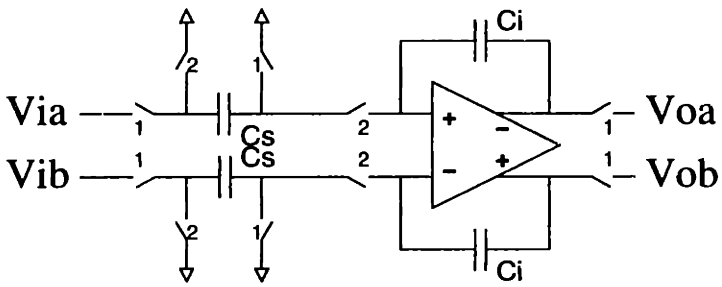


Figure 3-2: Non-Inverting Direct Discrete Integration

3.2 Sections of the Modulator

In figure 3-3, 3-4, 3-5, and 3-6, the sections and their equivalent circuit schematics are shown. The tool translates coefficients into capacitor ratios. Moreover, the tool also translates the summers and integration blocks into opamp circuits as discussed above.

It should be noted that depending on the sign of the coefficient generated by Schreier's tool[5], the connection of some sampling capacitors will be to the complementary signal which is not indicated in the figures. The tool determines the proper connection at run-time.

In figure 3-6, note the transformation of the block diagram which is equivalent if one makes the assumption that the comparator has close to infinite gain. This rearrangement of the block diagram is necessary because it is easier to implement on a circuit level. The summer preceding the quantizer is incorporated in the previous operational amplifier by using a switched-capacitor gain configuration as shown in figure 3-7. The transfer function of this block is:

$$H(z) = -\frac{C_S}{C_I} \tag{3.3}$$

This configuration of the opamp, however, adds extra signal in the local feedback path which must be accounted for through the feedforward coefficient at that node[1].

This transformation will not affect the operation of the modulator greatly. Adams et al.[12] show that lowering the gain in the loop alters the Nyquist Plot by getting the appropriate mapped function closer to encircling the negative one point. This reduces

[1]This is why the note at the bottom of figure 3-6 indicates a change to the previous feedforward coefficient value.

R–Order Modulator Even Order Start Section

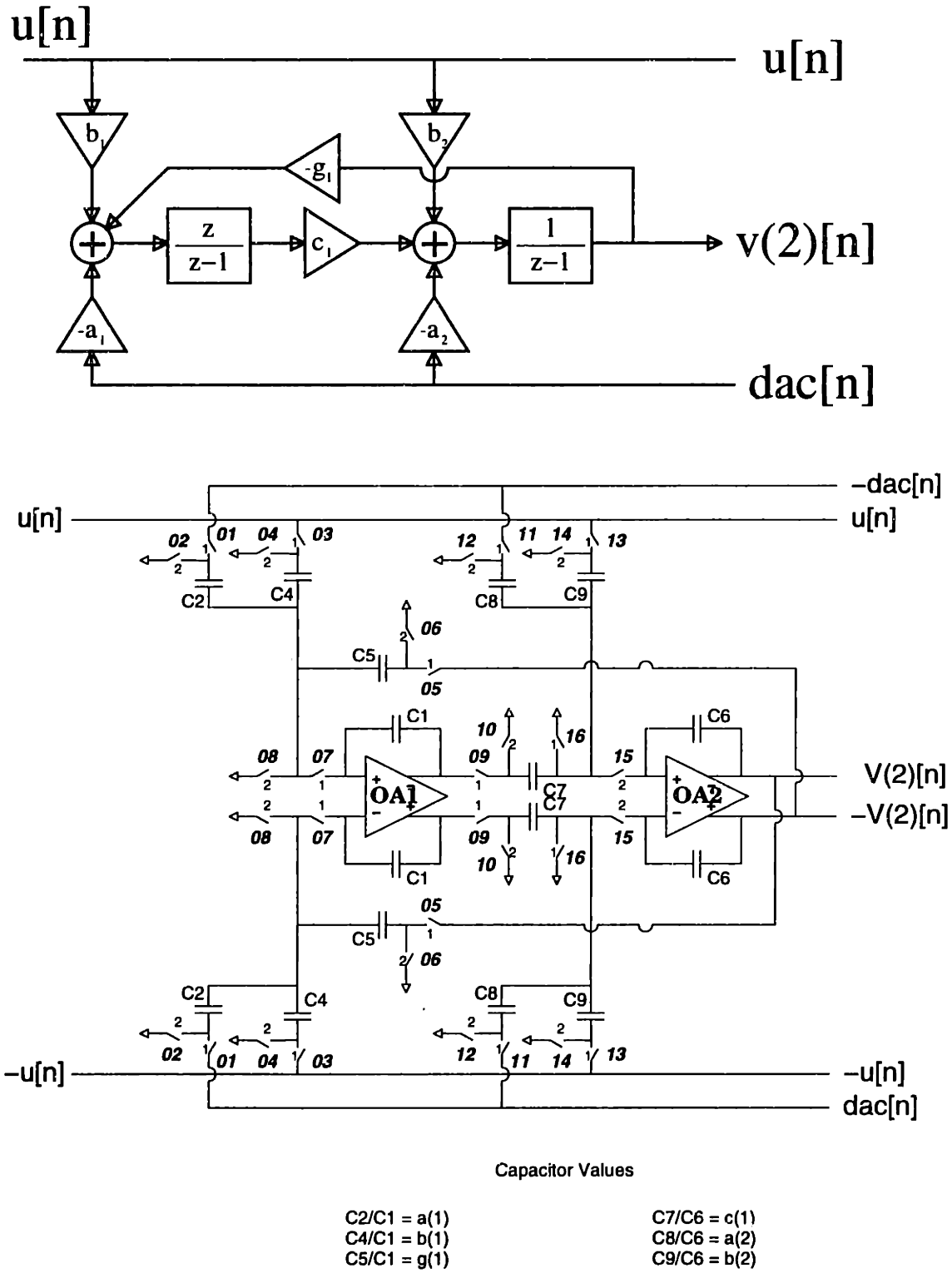
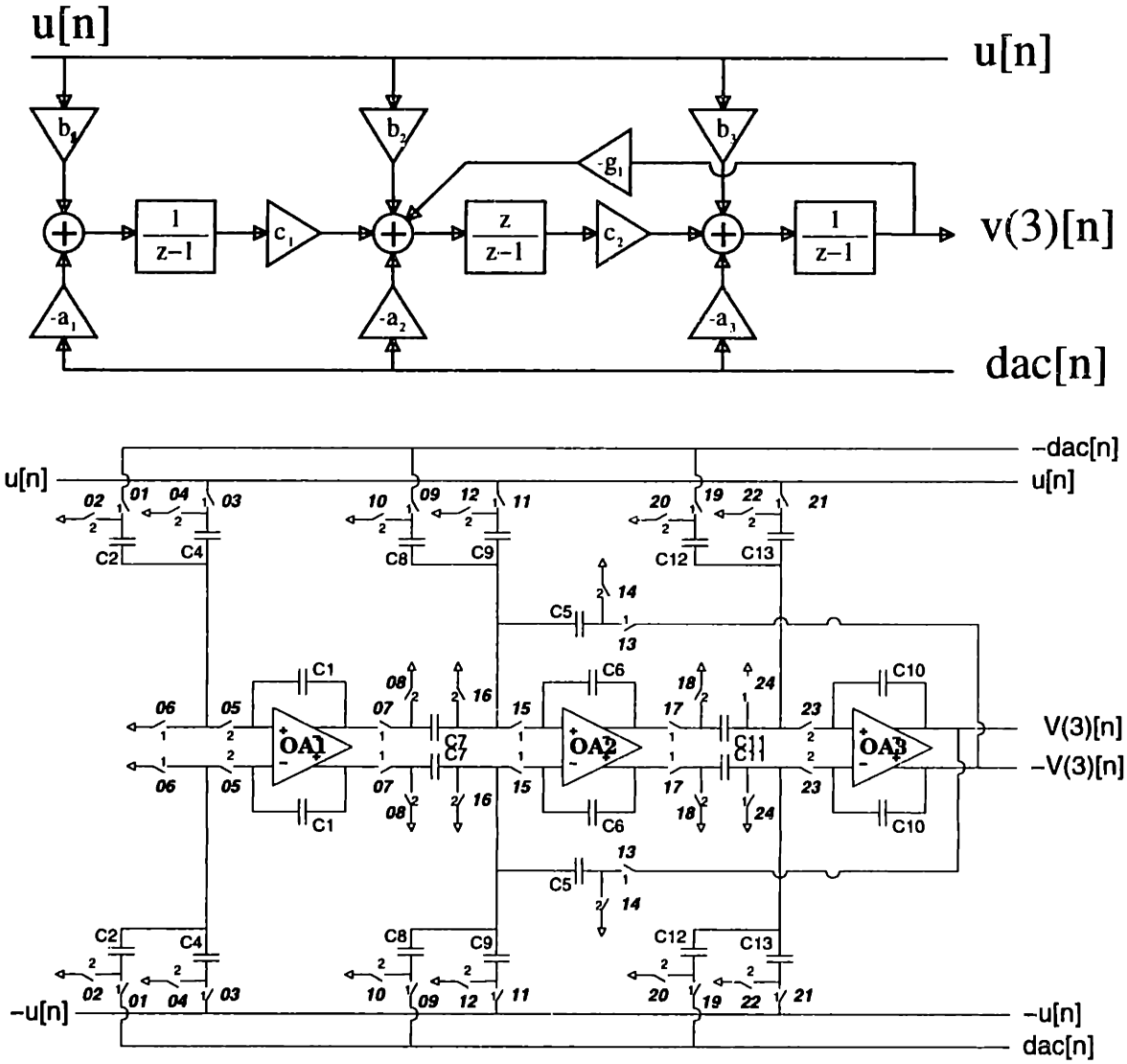


Figure 3-3: Even Order Start Section of R-Order Modulator – Block Diagram and Equivalent Circuit Implementation

R–Order Modulator Odd Order Start Section



Capacitor Values

C2/C1 = a(1)	C7/C6 = c(1)	C11/C10 = c(2)
C4/C1 = b(1)	C8/C6 = a(2)	C12/C10 = a(3)
	C9/C6 = b(2)	C13/C10 = b(3)
	C5/C6 = g(1)	

Figure 3-4: Odd Order Start Section of R-Order Modulator – Block Diagram and Equivalent Circuit Implementation

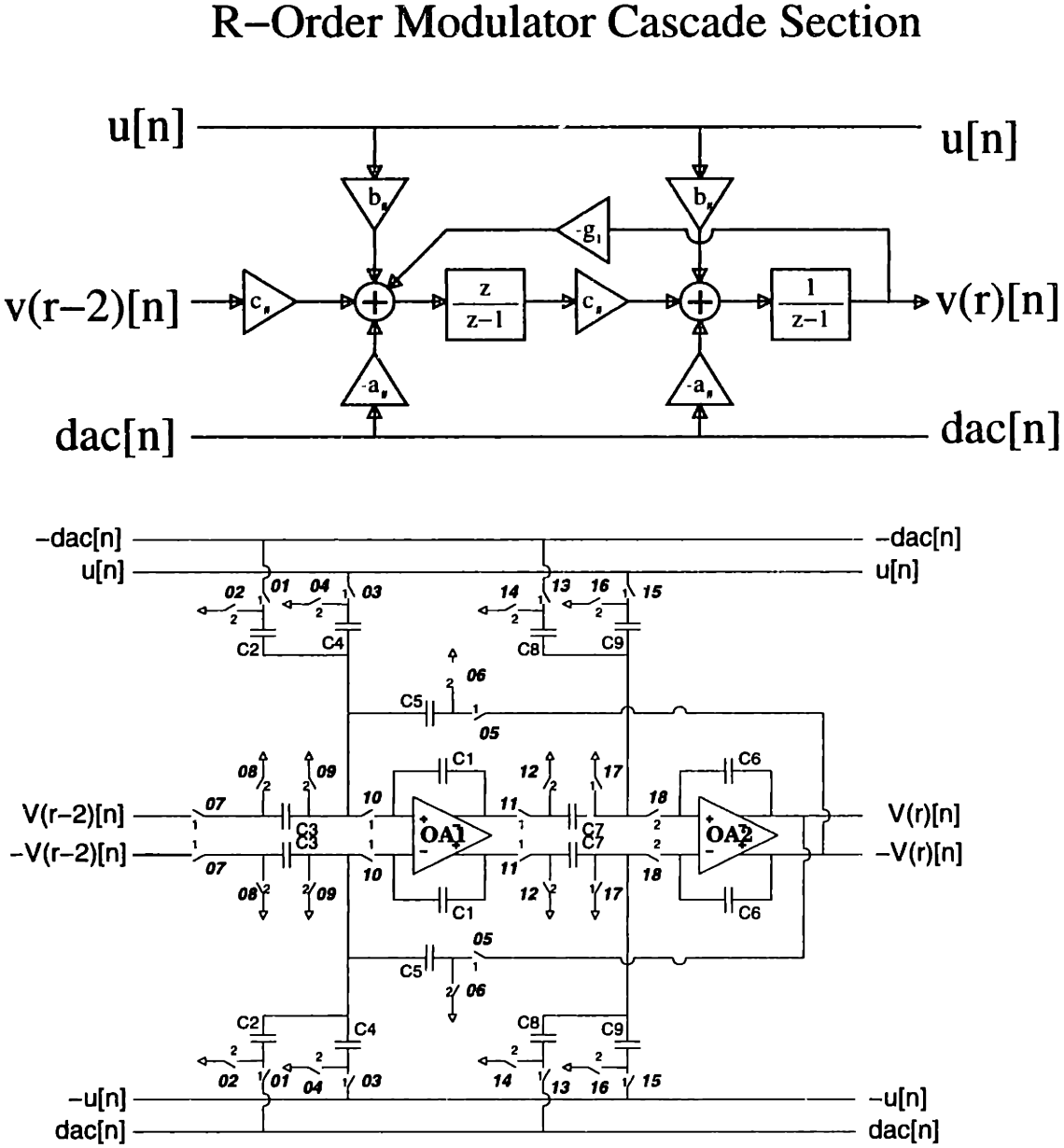


Figure 3-5: Cascade Section of R-Order Modulator – Block Diagram and Equivalent Circuit Implementation - r denotes the order of the section

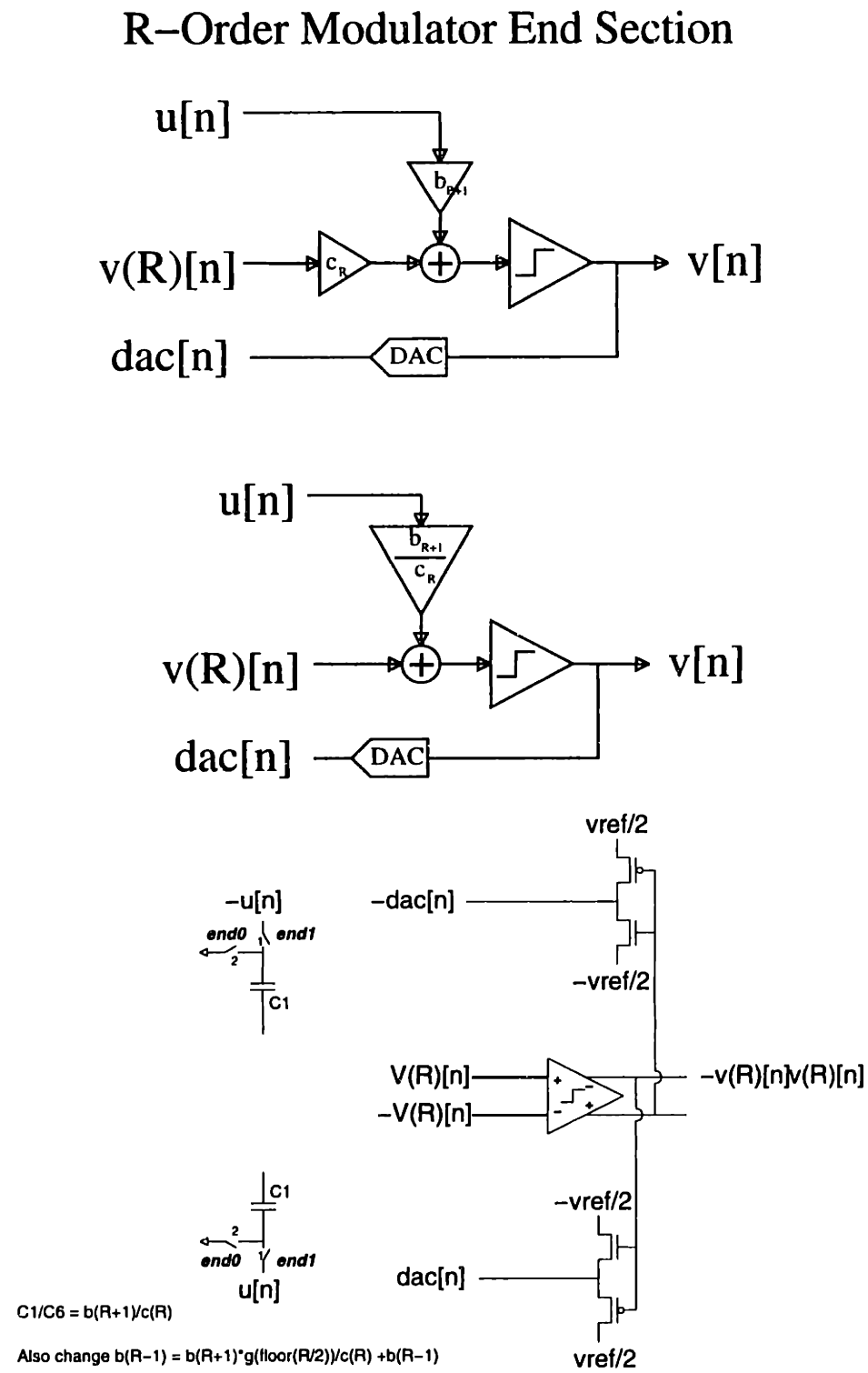


Figure 3-6: End Section of R-Order Modulator – Block Diagram, Transformation and Equivalent Circuit Implementation

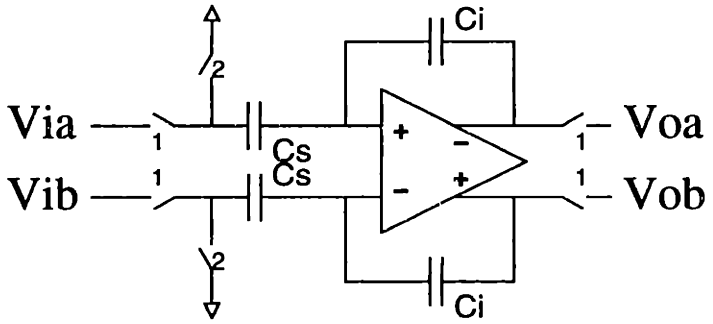


Figure 3-7: Gain Configuration

stability significantly if the change in gain is large. However, the c_R coefficient has been found to be rather insignificant, on the order of 0.1 in most cases. In any case, if this effect is great it will manifest itself in the plots of SNR vs. input amplitude.

3.3 Capacitor Value Determination

Since the tool starts with the coefficients of the CRFB topology, the capacitor ratios are well defined. However, they are only ratios and an implementation needs concrete values. One could use really small values to produce the necessary capacitor ratios in order to save area. However, noise considerations bound the minimum capacitor values. Because of the kT/C noise, the option for the minimum sized capacitor is included. This minimum size capacitor dictates only the minimum size capacitor of the first stage, since noise in this stage does not undergo any shaping. The latter stages, which do undergo shaping, have a minimum capacitor size of 100 femtofarads.

With the minimum value for each stage, the scaling strategy becomes straightforward[11]. The tool starts by assigning the feedback capacitor around each opamp to a value of 1 Farad and assigning the attached sampling capacitor values the coefficient value it is supposed to implement. Then, the tool examines the summing nodes of each operational amplifier. The minimum capacitor attached to that node serves as the scaling factor for the rest of the capacitors attached at that node. Thus, if the smallest sampling capacitor value is 0.5, then that capacitor value now becomes unity and every other capacitor (sampling and feedback) attached to the same summing node are multiplied by two. Once all the capacitors are normalized to 1 F, then the capacitors are multiplied by their respective minimum size capacitor. An example of this scaling method is shown in figure 3-8 and the appropriate scaling is below.

$$
\begin{aligned}
C_{min} &= \min(C_{S1}, C_{S2}, C_{S3}, C_I) \\
C_{S1}^{\dagger} &= \frac{C_{S1}}{C_{min}} \\
C_{S2}^{\dagger} &= \frac{C_{S2}}{C_{min}} \\
C_{S3}^{\dagger} &= \frac{C_{S3}}{C_{min}}
\end{aligned}
$$

$$C_I^{\dagger} = \frac{C_I}{C_{min}}$$

With this strategy, there lies the latent danger of potentially huge capacitors due to a small coefficient. Since small coefficients have small effects on the overall transfer function, coefficients below a threshold of 1e-4 are set to zero and not considered in the minimum capacitor scaling algorithm. Otherwise capacitors in excess of 100 pF are possible.

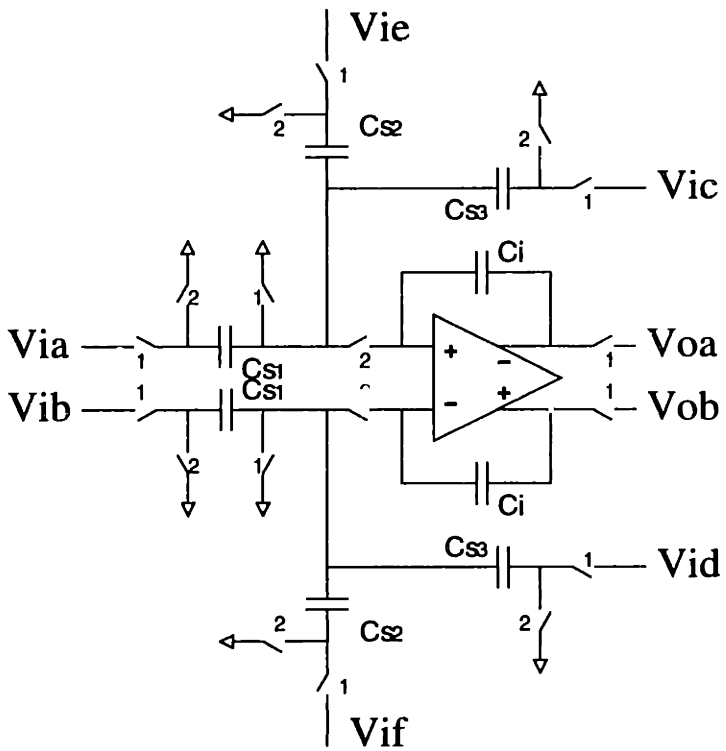


Figure 3-8: Example of Scaling

This process is repeated for every summing junction which will affect every capacitor in the modulator. Furthermore, this does not affect the loop filter transfer function because the capacitor ratios are unchanged.

Chapter 4

Switches

The switches as stated before, are implemented with pass gates as shown in figure 4-1. The complementary configuration with equal geometries for NMOS and PMOS has been chosen to increase signal range[11] and decrease on-resistance variability[13, pp. 213-214].

4.1 Switch Scaling

The tool scales the switches to meet the SNR requirements of the desired delta-sigma modulator. A simulation using Schreier's toolbox[5] gives an estimate of the SNR and this figure, with an appropriate safety factor[1], determines the necessary accuracy and settling time, and hence, the size of the requisite switch[2]. Charge injection of the switch is considered to be minimal. The determining factor for sizing a switch is the settling time constant.

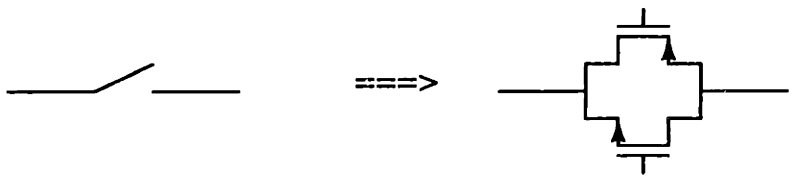


Figure 4-1: Switch Implementation

If one models the switch as a resistor as in figure 4-2, then the settling of the capacitor voltage is first order once the switch closes.

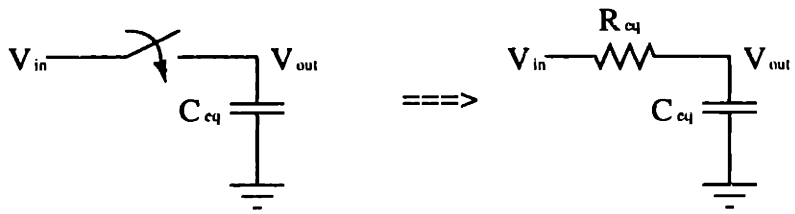


Figure 4-2: Equivalent Modeling Circuit to Determine Switch Size

The well-known time-domain response of this system to a step is:

[1]Through empirical testing and conservative rationale, the safety factor is ten.
[2]Schreier's toolbox[5] simulation of the modulator should yield a SNR that is close to the real value.

Characterization Run	Average On-Resistance	Maximum On-Resistance
n71s	8.86 kΩ	16.5kΩ
n72a	9.91 kΩ	20.0kΩ
n73d	10.1 kΩ	20.0kΩ

Table 4.1: Summary of on-resistances and variability

$$V_{out}(t) = V_{in}(1 - e^{-\frac{t}{R_{eq}C}}) \tag{4.1}$$

The time that this must settle to within the desired accuracy (SNR) is fixed by the clock phase length denoted by t_P, which is 50 nanoseconds[3]. The condition on the size of R_{eq} then is:

$$\frac{1}{R_{eq}} = \frac{\ln(SNR) * C_{eq}}{t_P} \tag{4.2}$$

Since the switch size is inversely proportional to R_{eq}, The required width W_{switch} (NMOS and PMOS) of the switch is:

$$W_{switch} = \frac{\ln(SNR) * C_{eq} * R_{max} * s}{t_P} \tag{4.3}$$

where R_{max} denotes the maximum on-resistance of the minimum size switch and s denotes a safety factor which has conservatively been chosen to be 2. It accounts for parasitic capacitances associated for the switches, resistance non-linearity, etc.

To complete the switch scaling, only R_{max} and C_{eq} need to be found.

4.2 Determination of On-Resistance of a Minimum Size Switch

Simulation of the on-resistances of the switch for three different characterization runs (n71s, n72a, n73d) of the MOSIS HP 0.5 μm process[14] are shown in figures 4-3, 4-4, and 4-5 for different lengths and under different operating conditions. The $1\mu/0.5\mu$ and $10\mu/0.5\mu$ (NMOS and PMOS) switch resistances are shown in each plot in the top row and bottom row, respectively. One terminal of the switch was swept through the supply voltage ($\pm$ 1.65 Volts) while the other was held fixed at -1.65 V, 0.00 V, and +1.65 V as shown in each column, respectively. The current was then measured and the resistance was determined incrementally. Thus, there are discretization steps as seen in the plots.

From the plots, one observes that the on-resistance expectedly scales roughly linearly with the switch size. To avoid small geometry effects, the minimum size or unit switch will be 1μm/0.5μm (NMOS and PMOS).

The average and maximum on-resistances for the minimum size ($1\mu/0.5\mu$) switch are tabulated in table 4.1.

[3]The values must settle to their values during the clock phase length, not the clock period.

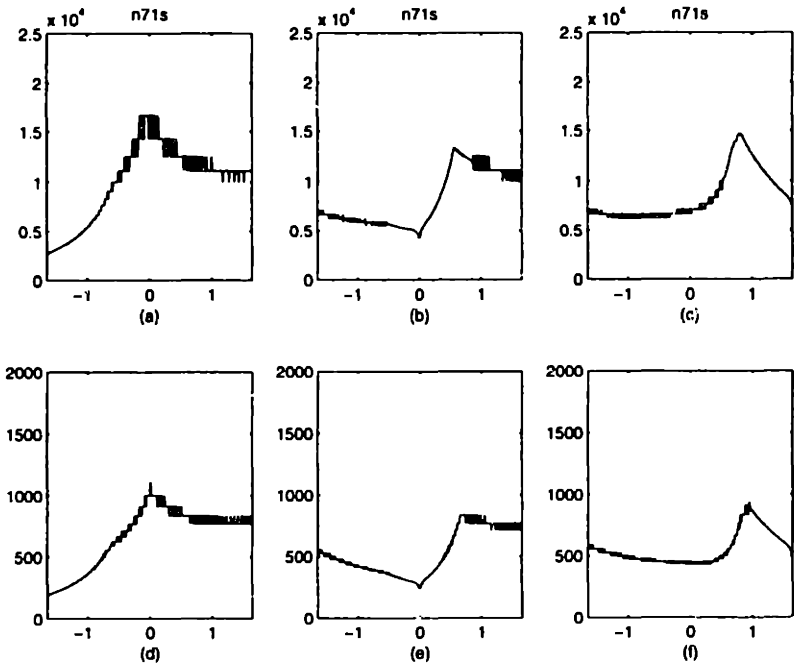


Figure 4-3: Switch On-Resistances of Parametric Test n71s

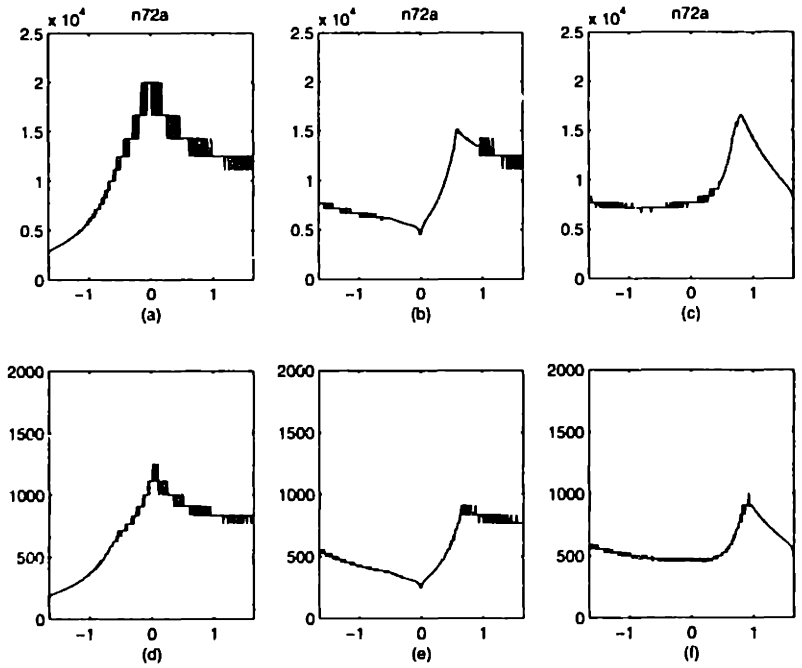


Figure 4-4: Switch On-Resistances of Parametric Test n72a

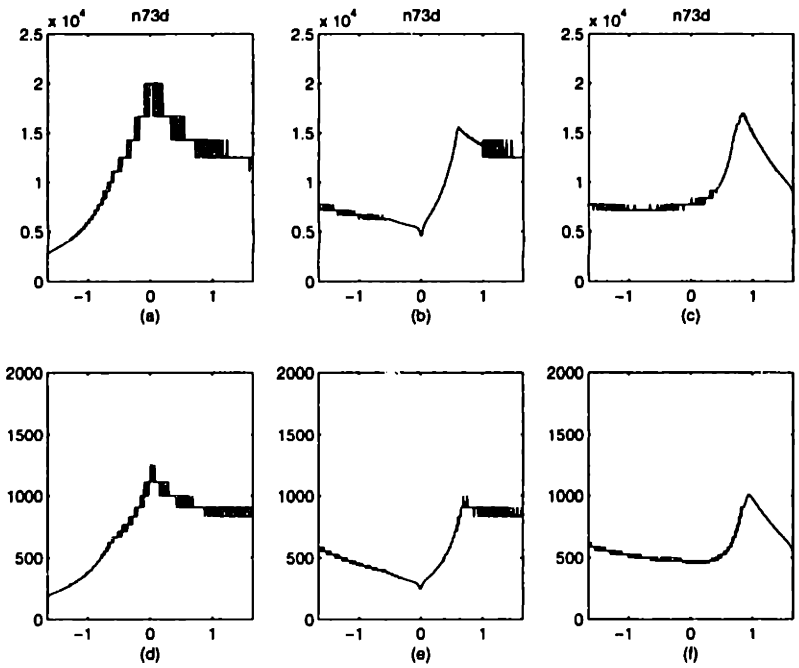


Figure 4-5: Switch On-Resistances of Parametric Test n73d

As a conservative choice, the R_{max} is set as 15 kΩ. and further simulations will use the n73d run as an arbitrary choice for all subsequent simulations.

4.3 Determination of Equivalent Capacitance

In the switched capacitor scheme, the largest capacitance at any node must be determined for both phases of the clock so that the worst case can be accounted for.

The equivalent capacitance that a switch needs to settle is then just the worst case sum of all the capacitors attached to the switch. A conservative calculation of equivalent capacitance that a switch needs to settle is shown in tables 4.2, 4.3, 4.4, and 4.5. The switch numbers and capacitance numbers are the ones as defined in the schematics of chapter 3. (Figures 3-3, 3-4, 3-5, and 3-6 on pages 20, 21, 22, and 23).

Switch	Equivalent Capacitance
01, 02	C_2
03, 04	C_4
05, 06	C_5
07, 08	$C_2 \parallel C_4 \parallel C_5$
09, 10	C_7
11, 12	C_8
13, 14	C_9
15, 16	$C_7 \parallel C_8 \parallel C_9$

Table 4.2: Equivalent Capacitances for Each Switch for Even Order Start Section (Figure 3-3, Page 20)

Switch	Equivalent Capacitance
01, 02	C_2
03, 04	C_4
05, 06	$C_2 \parallel C_4$
07, 08	C_7
09, 10	C_8
11, 12	C_9
13, 14	C_5
15, 16	$C_5 \parallel C_7 \parallel C_8 \parallel C_9$
17, 18	C_{11}
19, 20	C_{12}
21, 22	C_{13}
23, 24	$C_{11} \parallel C_{12} \parallel C_{13}$

Table 4.3: Equivalent Capacitances for Each Switch for Odd Order Start Section (Figure 3-4, Page 21)

Switch	Equivalent Capacitance
01, 02	C_2
03, 04	C_4
05, 06	C_5
07, 08	C_3
09, 10	$C_2 \parallel C_3 \parallel C_4 \parallel C_5$
11, 12	C_7
13, 14	C_8
15, 16	C_9
17, 18	$C_7 \parallel C_8 \parallel C_9$

Table 4.4: Equivalent Capacitances for Each Switch for Cascade Section (Figure 3-5, Page 22)

Switch	Equivalent Capacitance
end0, end1	C_1

Table 4.5: Equivalent Capacitances for Each Switch for End Section (Figure 3-6, Page 23)

Chapter 5

Component Generation and Synthesis

Since the central objective of this thesis is to generate practical designs, the author chose a robust and conservative methodology for synthesis of the delta-sigma modulator components: DACs, comparators, and operational amplifiers. The general idea is to scale everything in unison within each component. This translates into constant width ratios, constant current density, and constant time constants. The loading of a component uniquely determines the scaling factor[1].

This strategy positively ensures the feasibility of these components and avoids all the pitfalls and hazards of general optimization encountered by other analog circuit synthesis tools. In essence, this tool has only one degree of freedom for component generation. The prized benefits are robustness and speed of synthesis.

In the following sections, this general methodology is applied in the context of each component generation process.

5.1 One-Bit Digital To Analog Converter

The one-bit digital to analog converter (DAC) must be synthesized at run-time of the tool because the user's specifications are not always the same and, thus, the load it drives will not always be the same. However, the synthesis for this component is trivial because of the chosen implementation as shown in figure 5-1.

The scaling factor of this DAC is determined by the switches it is connected to. The $(W/L)_{DAC}$ is set to twice the sum widths of all switches that are attached to it. Also the switches connected to the DAC are doubled since they are essentially another switch away from a voltage source. This makes the effective resistance of the switch and DAC switch approximately equal to the original intended switch resistance.

Although the DAC transistors are only connected to a voltage source through one transistor whereas the normal switches are through two transistors (PMOS and NMOS), one end of the DAC transistor is always connected to a reference voltage. It always has the maximum gate-source voltage and has comparable on-resistance[2].

[1] This methodology is analogous to the scaling of the switches.

[2] The complementary device if put in parallel as in the switches would be in cutoff anyhow.

The geometry of the PMOS and CMOS transistors are chosen to be equal for simplicity and consistency with the other switches.

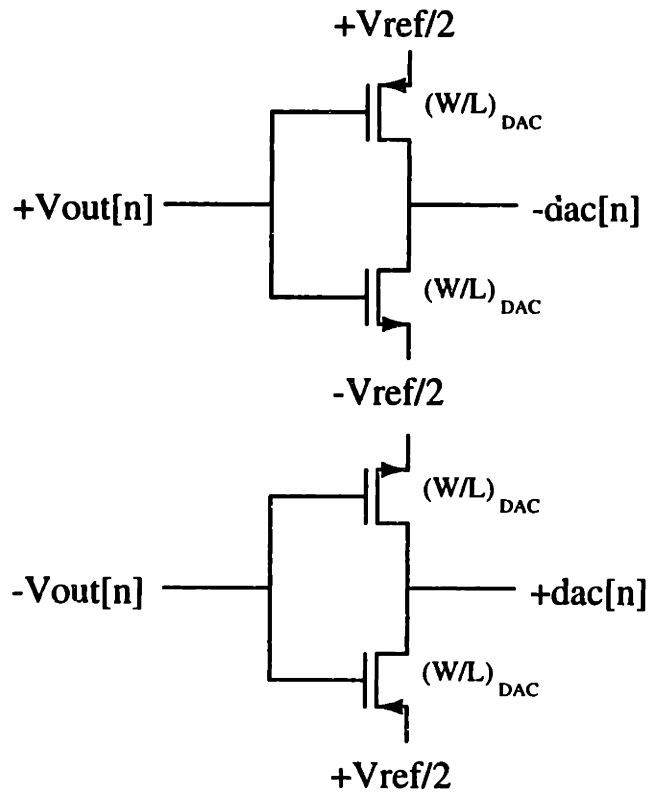


Figure 5-1: Schematic of One-Bit DAC

To protect against extremely small geometry DACs, there is a cutoff for the minimum size DAC. If the DAC transistor widths are below 1.0 μm, then the transistor widths are 1.0 μm.

5.2 Comparator Generation

The comparator/one-bit quantizer needs to be synthesized at run-time as well because of the variable DAC load which it will drive (and perhaps a decimator). However, the performance of the comparator for this modulator is not critical. The only real specification for it is SNR settling (analogous to the switch scaling methodology of chapter 3) within the clock phase. Non-idealities such as offset voltage and hysteresis effects are shaped by the loop filter.

The scaling factor of the comparator is determined by the load which is the DAC[3]. Once that is determined, all transistor widths in the comparator unit design are scaled by that factor. As an example, a scaling factor of 5 would correspond to a load that is five times the nominal load of the unity scale design. Every transistor width in the unity scale comparator would be multiplied by 5. As in the case of the DAC, there is a minimum scale factor of unity. Any scaling factor below this threshold will cause a comparator with unity scaling factor to be generated.

[3]For the complete system, one would need to consider the loading effect of the decimator.

5.2.1 Unit Comparator

This tool uses the unit comparator/one-bit quantizer designed by Yukawa[15] and sized by Brandt, Ferguson, and Rebeschini [6, p. 372]. The unity scale circuit schematic of this sampled, regenerative latch is shown in figure 5-2. The comparator is strobed at the transistor drains to eliminate backgate effects and increase regeneration speed. In addition, an SR latch has been added to minimize effects of comparator indecision. The lengths of the transistors are minimum for greater speed and the widths are determined by conditions needed to make the cross-coupled inverters toggle [13, p. 342].

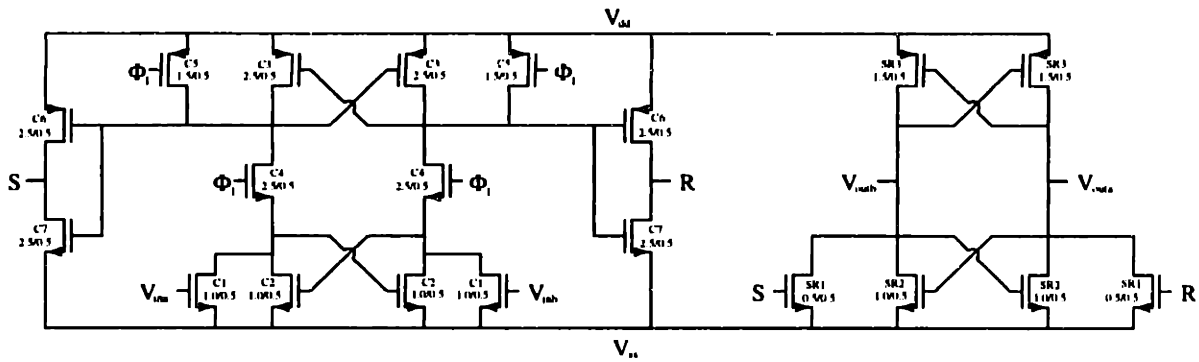


Figure 5-2: Schematic of Unit Comparator/One-Bit Quantizer

This comparator samples the differential input on the leading edge of Φ_1 which is consistent with the phases of the preceding integrators.

Performance of this comparator with a 100 femtofarad capacitor load, the nominal load on each differential output is shown in figure 5-3. The clock phase, Φ_1, of the modulator is high for 50 nanoseconds and the comparator settles to the final value well within that time.

The comparator works for large differential input signals as well as for small differential input signals as it should.

5.2.2 Comparator Load Determination

The load of the comparator is just the input capacitance of the DAC transistors. As an estimate, the input capacitance is just the gate-to-bulk capacitance which is:

$$C_{compload} = 2 * dacsize * 0.5 * 10^{-12} * \kappa_{SiO_2}\varepsilon_0 * s \tag{5.1}$$

where *dacsize* is in microns and s is a safety factor of 2.

Now that the load is determined, the appropriate scaling factor is used to synthesize the comparator.

5.3 Operational Amplifier Generation

Synthesis of the most complicated component, the operational amplifier, is made easy with the sizing methodology. Similar to the previous component generation, the tool starts with a unity scale operational amplifier and then scales everything to keep major parameters of the

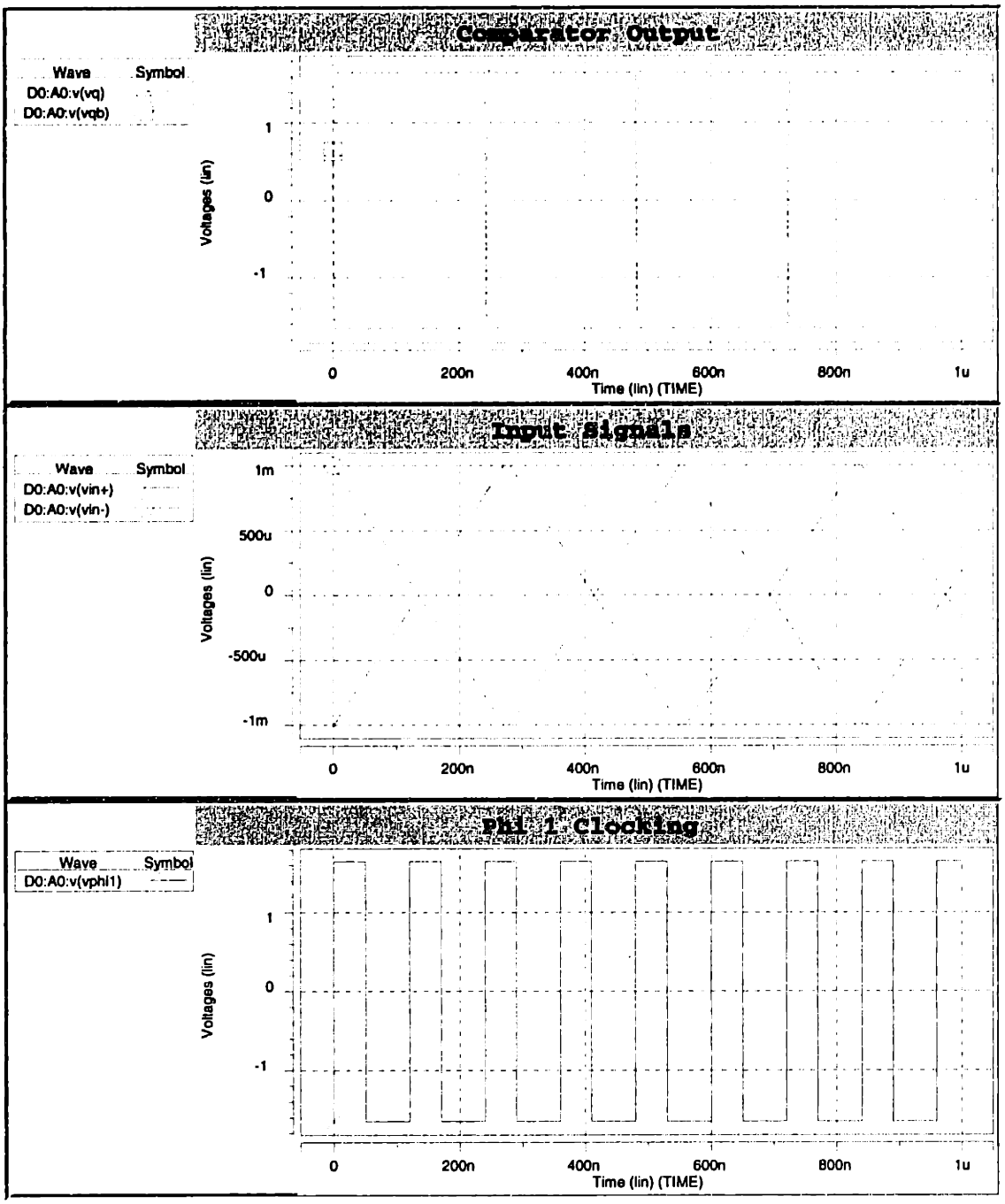


Figure 5-3: Transient Response of Unit Comparator/One-Bit Quantizer

operational amplifier constant. The scaling factor is, again, based on the load capacitance, nominally 100 femtofarads, the operational amplifier must drive. If the scaling factor is below unity, then an operational amplifier with unity scaling factor will be synthesized.

5.3.1 Unit Operational Amplifier

The chosen operational amplifier technology is a MOS fully differential two stage with a cascoded first stage shown in figure 5-4. The opamp design is generic, but robust which is important for scalability.

Opamp Design

Overall, the opamp design is straightforward. All the transistor lengths, except the input pair are twice the minimum length for higher output resistance. The input pair's transconductance is more critical than the output resistance and thus, a shorter length is used. Additionally, this reduces the capacitance load at the input. The drawbacks of this approach are that there will be a larger random offset voltage and greater transconductance mismatch.

The nominal common mode input voltage of the opamp is zero volts which is congruous with the common mode voltages of the delta-sigma modulator.

The first stage of the operational amplifier is cascoded to get higher DC gain. This helps to boost the gain to over 10,000 which is needed to avoid the effects of finite opamp gain in the delta-sigma modulator[4][6, pp. 232-233]. In addition, p-channel inputs are used to optimize slew rate and frequency response[16, pp. 231-232].

The cascode transistors M9-12 are all biased with the same voltage because the output signal swing of the first stage is small due to the second stage amplification. Additionally, an ideal current source for biasing has been assumed.

The tail current source of the first stage is split between two transistors, one which is connected to a bias and the other which is part of the common mode feedback (CMFB) circuitry.

The second stage transistors are large to increase dynamic range, improve phase margin, and increase slew rate.

The well understood Miller capacitance across the second stage compensates this operational amplifier. Lead compensation is also used by placing a resistor in series with the Miller capacitor. Thus, the Miller capacitance causes the dominant pole and the load capacitance causes the non-dominant pole.

Another important feature is that the transistors and capacitors are minimized since this is the smallest design possible. The current and width values border on the frontier of good design. However, the opamp will always be scaled up. Therefore, the design variables of the unit design must be minimized so that the opamps that drive larger loads will not be egregious.

This design results in performance summarized in table 5.1. The performance is not stellar, but conservative.

[4]Finite opamp gain translates into integrator leakage.

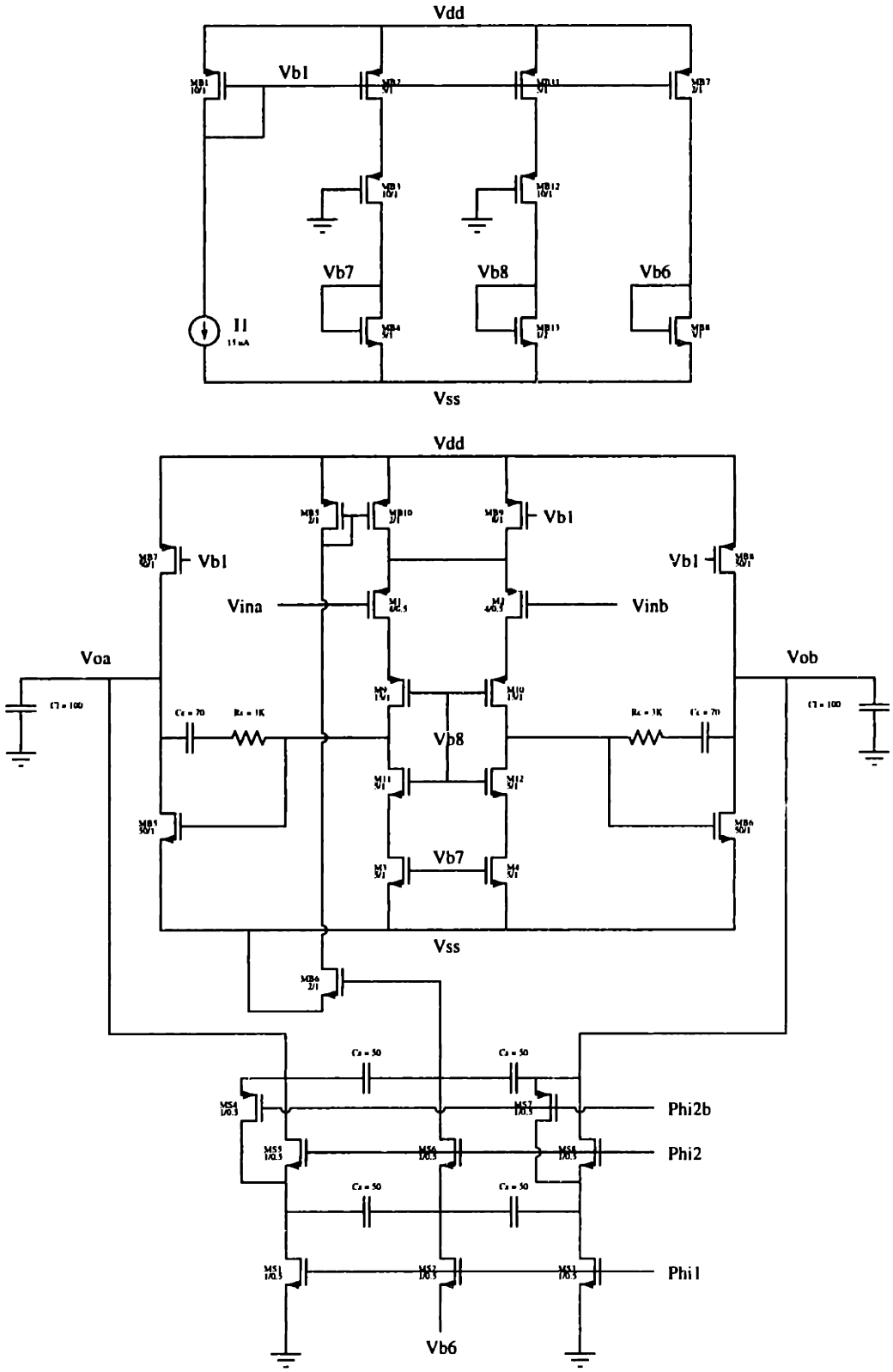


Figure 5-4: Unit Operational Amplifier Schematic. Capacitor values are in femtofarads, resistor values are in ohms, and widths and lengths are in microns.

Performance Characteristic	Value	Units
DC Gain	10,400	—
Output Swing (Gain > 10,000)	-1.4 to 1.4	Volts
Unity Gain Frequency	43	Megahertz
Phase Margin	64	Degrees
Differential Settling Time (-1 Inverting Config.) (100 fF feedback caps) (1.0 diff. input step)	31 to 0.1%	Nanoseconds

Table 5.1: Performance of the Unit Operational Amplifier

CMFB

The CMFB is a standard switched capacitor scheme[17]. The common mode component of the output voltage is sensed with switched capacitors and then fed back to the first stage tail current source through an inverter. The inverter not only generates the necessary inversion for the loop transmission, but also acts as a buffer between the sensing circuitry and the opamp first stage. Furthermore, the CMFB network is stable because it shares the same compensation as the differential path except with a smaller gain because it only controls 20% of the current in the first stage. One may wonder about the validity of the compensation which is a continuous time system while the CMFB is a discrete system, but this view is still approximately valid if the clock period is faster than the dominant time constant of the CMFB. This is in part guaranteed by the lower loop gain which will decrease the system speed as understood in elementary proportional compensation.

The rudiments of how this CMFB works is as follows: (1) During Φ_1 high (Φ_2 low), the output common mode is sampled onto a pair capacitors, and the desired common mode and nominal bias is sample onto another pair of capacitors. (2) When Φ_2 is high (Φ_1 low), the stored charge on these capacitors is combined and fed to the inverter that effects current change in the first stage. Thus, when the the output common mode level is too high, the feedback voltage goes up, the current in the first stage goes down which causes the output voltages to drop. The opposite is true when the output common mode level is too low. An important point to make is that the common mode output voltage will oscillate and not settle to exactly the desired common mode voltage because of the charge injection (affects the fedback voltage) and mismatch between the nominal bias and the actual desired bias set by the feedback loop. This is evident in the transient response.

The capacitors and switches are minimum size. The accuracy and speed of this CMFB loop is not terribly important. The common mode voltage should be stable and slight shifts will not significantly deteriorate overall performance since the differential signal is what is important.

Figure 5-5 shows the common mode transient response. One can see that the switching speed is faster than the loop speed. (i.e. The crossover frequency of the loop transmission is less than the clock frequency.) Also, the slight steady state oscillation of the output as discussed above is observed.

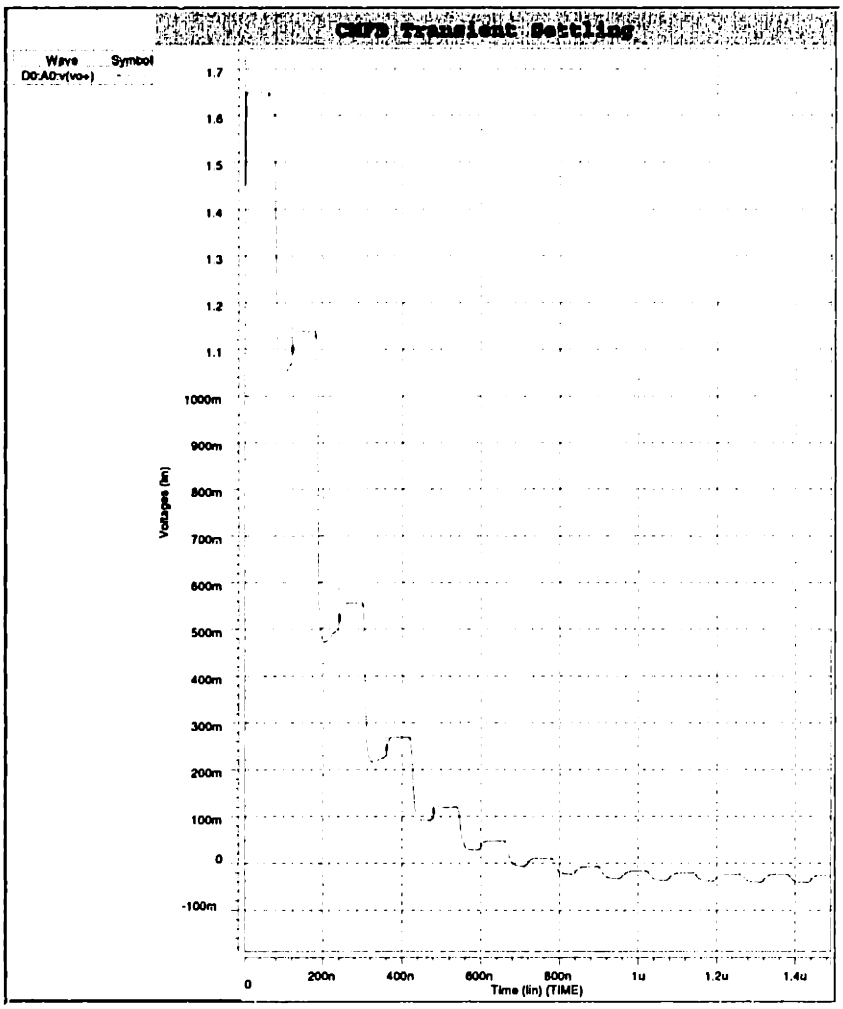


Figure 5-5: Common Mode Feedback Transient Response.

5.3.2 Effects of Opamp Scaling

The scaling factor for opamp scaling is:

$$\alpha = \frac{C_{load}}{100fF} \tag{5.2}$$

To keep all the important characteristics of the operational amplifier constant, the tool must scale all the transistor widths by α, capacitors by α, current sources by α, and resistors by $1/\alpha$. The following equations show the constancy of the DC gain (A_v), unity gain frequency (ω_t), phase margin (PM), and slew rate (SR) with this scaling methodology. The 0 and 1 subscripts denote values of the unit opamp.

$$A_v = G_{m1}R_{o1}G_{m2}R_{o2} = (\sqrt{k\alpha\frac{W_0}{L}\alpha I_0})(\frac{1}{\alpha I_0})(\sqrt{k\alpha\frac{W_1}{L}\alpha I_1})(\frac{1}{\alpha I_1}) = A_{v0} \tag{5.3}$$

$$\omega_t = \frac{G_{m1}}{C_c} = \frac{\sqrt{k\frac{\alpha W_0}{L}\alpha I_0}}{\alpha C_{c0}} = \omega_{t0} \tag{5.4}$$

$$SR = \frac{I}{C_L} = \frac{\alpha I_0}{\alpha C_{L0}} = SR_0 \tag{5.5}$$

For the same phase margin, the tool needs to keep the poles and zeros the same. If the opamp is modeled by:

$$a(s) \approx \frac{(1-\frac{s}{z_1})}{(1-\frac{s}{p_1})(1-\frac{s}{p_2})} \tag{5.6}$$

then,

$$p_1 \approx -\frac{1}{G_{m2}R_{o2}R_{o1}C_c} = -\frac{1}{\alpha G_{m20}\frac{R_{o2}}{\alpha}\frac{R_{o1}}{\alpha}\alpha C_c} = p_{10} \tag{5.7}$$

$$p_2 \approx -\frac{G_{m2}}{C_L} = -\frac{\alpha G_{m20}}{\alpha C_{L0}} = p_{20} \tag{5.8}$$

$$z_1 \approx -\frac{1}{C_c(R_c - \frac{1}{G_{m2}})} = -\frac{1}{\alpha C_c(\frac{R_c}{\alpha} - \frac{1}{\alpha G_{m2}})} = z_{10} \tag{5.9}$$

For verification of this trend, the opamp is synthesized for load capacitors in the range of 100 fF to 10 pF and the characteristics are plotted in figure 5-6 and 5-7. The values stay rather constant after the small geometry effects are sized out. The small geometry effects include narrow channel width effects and low current levels. Thus, one can be confident the opamps in the generated delta-sigma modulator will behave as expected.

5.3.3 Operational Amplifier Load Determination

The load the opamp needs to settle can be determined from the capacitors attached to the output node in figure 3-3, 3-4, 3-5, and 3-6. The worst load for each clock phase must be used since that will dictate the size of the operational amplifier. Conservative estimates of the load as shown in table 5.2 are used.

In the table, C_{next} refers to the capacitance of the next stage. If the next section is a cascade section, then $C_{next} = C_3$. If the next section is the end section, then $C_{next} = C_{comp}$. The comparator input capacitance is estimated as a parallel plate capacitor with

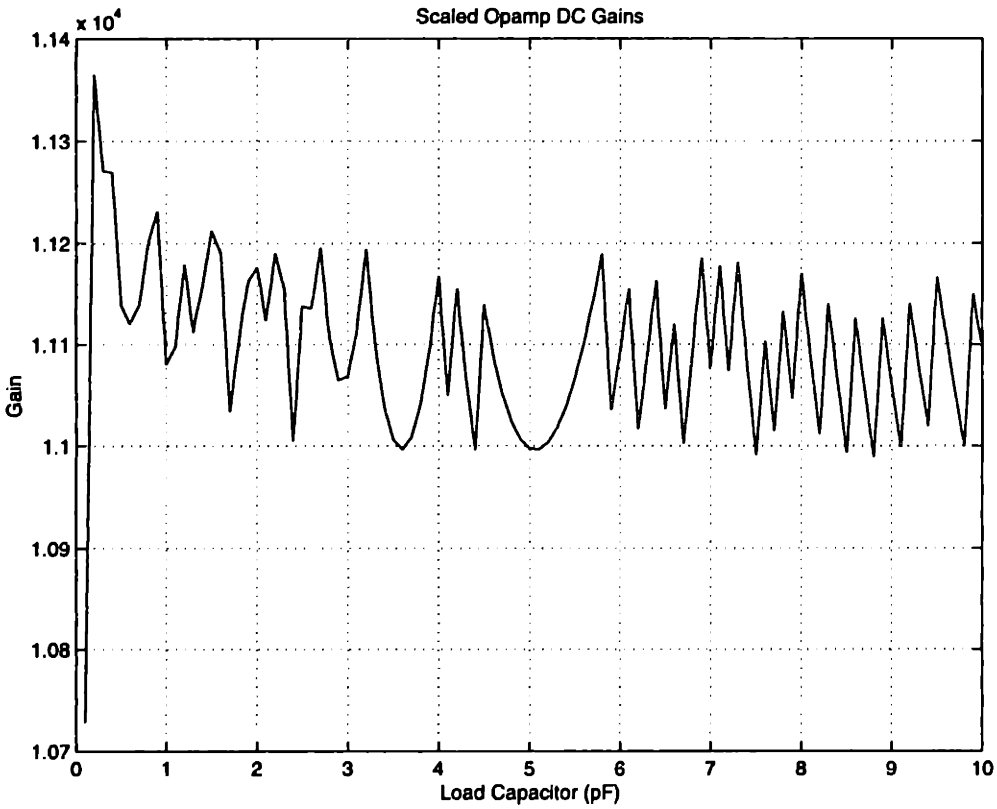


Figure 5-6: DC Gain Versus Load Capacitance of Scaled Opamp.

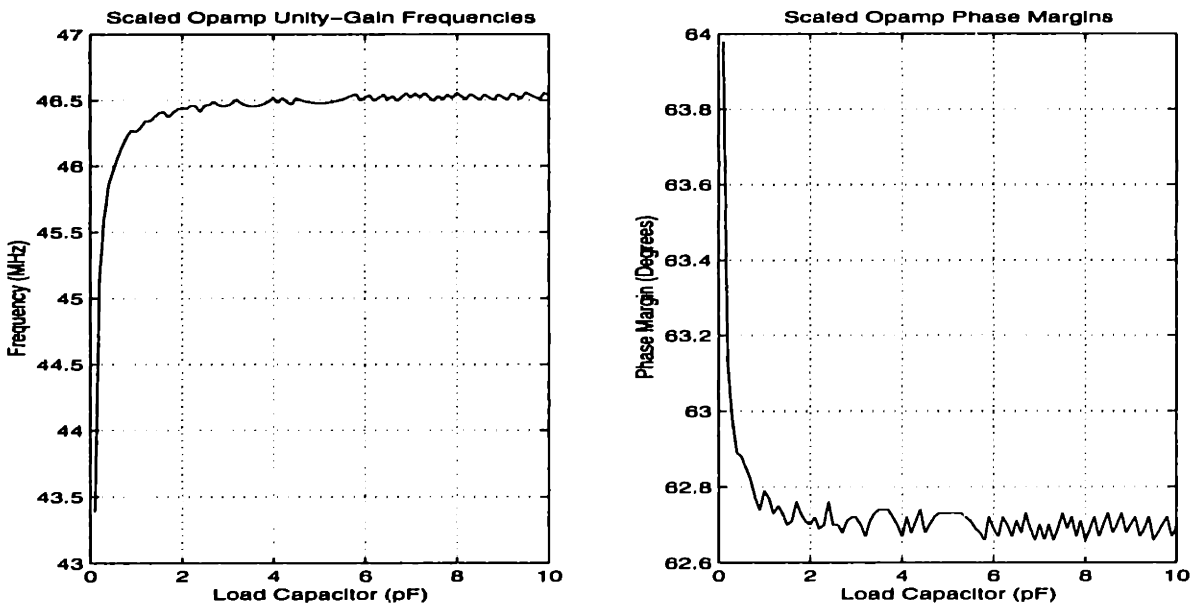


Figure 5-7: Unity Gain Frequency and Phase Margin Versus Load Capacitance of Scaled Opamp.

Even Order Start Section Operational Amplifier	Load To Drive
OA1	$C_1 + C_7$
OA2	$C_5 + C_6 + C_{next}$
Odd Order Start Section Operational Amplifier	Load To Drive
OA1	$C_1 + C_7$
OA2	$C_6 + C_{11}$
OA3	$C_5 + C_{10} + C_{next}$
Cascade Section Operational Amplifier	Load To Drive
OA1	$C_1 + C_7$
OA2	$C_6 + C_{11}$
OA3	$C_5 + C_6 + C_{next}$

Table 5.2: Load Capacitances of Operational Amplifiers to be Generated

the geometry of the input pair.

Thus, all the information for opamp synthesis has been determined and the tool generates the opamps.

Chapter 6

Simulation Results and Discussion

For evidence that the synthesized delta-sigma modulators behave as designed, simulations in SPICE were performed and the output was compared with behavioral simulations from Schreier's MATLAB toolbox[5]. The MOSIS HP 0.5 μm[14], run n73d models, along with ideal clock generators operating at 8.3 MHz were used.

6.1 Fifth Order, 32x Oversampling, Low Pass Delta-Sigma Modulator

For a specific test of the tool, a fifth order, thirty two times oversampling low pass delta-sigma modulator was simulated. Figure 6-1 shows the time domain SPICE output of this modulator. As expected, a windowed average of the modulator output is roughly equal to the input signal[6, p.16].

A better interpretation of how well the modulator performs is seen in the frequency domain. Using a Hanning window which preserves signal power calculations and reduces windowing effects and artifacts[18], the fast Fourier transformed signal is shown in figure 6-2. Only a 2^{11} point FFT is used because the simulation times become prohibitively long. Unfortunately, this also prevents the simulation of large oversampling ratios if accurate SNR figures are desired. This is in fact the reason why the chosen oversampling ratio is relatively low. The generated spectrum is compared to what is produced by Schreier's toolbox's behavioral simulation displayed in figure 6-3. As one can see, the two spectra are very close. The signal input frequency—at 65 kHz, the center of the frequency band of interest—has approximately the same magnitude. The shaped quantization noise is clearly seen as it rises from a low level to high levels out at higher frequencies. Using the quantitative measure of SNR, the behavioral simulation, 63.9 dB, is very close to the SPICE simulation, 62.7.

A salient feature of the two spectra is the magnitudes at low frequencies. Theoretically, the zeros of the loop filter have been placed at 1 in the z-plane and thus, the frequency response should be negative infinity at DC. The effect is manifest in the behavioral simulation. However, in the SPICE simulated output, the noise floor is considerably higher. The reason for this is the numerical noise associated with SPICE simulations. The inaccuracy of simulation moves the zeros off the unit circle and so the attenuation is not as expected. When tolerances are tightened, the noise floor indeed goes down, however, at the expense of increased simulation time. This effect is not noticeable in the SNR figure because of the width of the frequency band of interest. The plots are logarithmic and this noise is but a

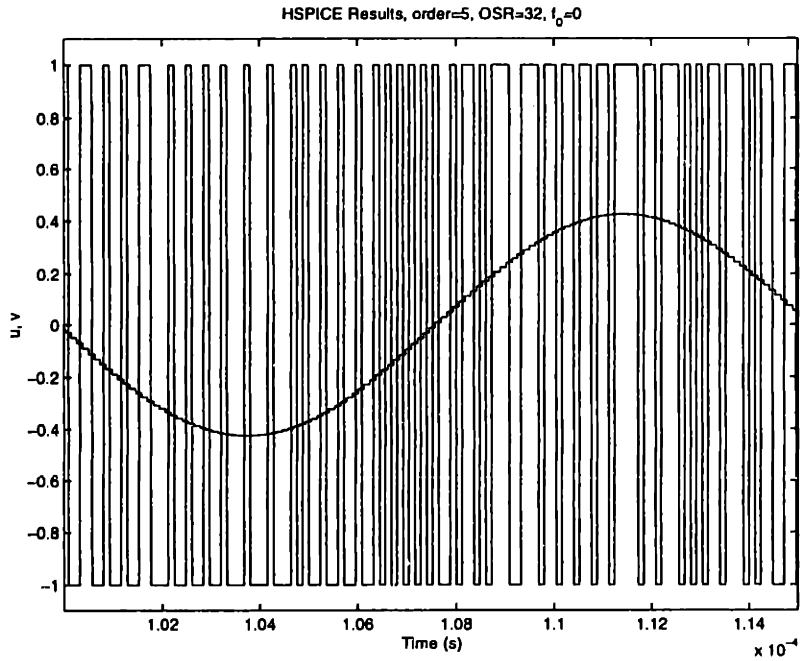


Figure 6-1: Time Domain SPICE Simulation of a 5th Order, 32 Times Oversampling Low Pass Delta-Sigma Modulator. Amplitude is normalized to V_{ref}

small fraction of in-band noise.

The delta-sigma modulator behaves properly given that the signal amplitude is small enough. If the input signal becomes too large, then the modulator—inherently a feedback system—becomes unstable [6, p. 144]. This effect is clearly seen in figure 6-4. There is no noise shaping since the filter is now nonlinear. The internal nodes of the opamp outputs saturate and cause oscillations. The exact input signal amplitude to cause instability is not well described, but can be maximized by performing dynamic range scaling as in the first part of the synthesis. Schreier's tool[5] gives a rough threshold of stability, but it is not exact.

Figure 6-5 shows the SNR versus the input signal amplitude. The dotted line is the rough stability threshold given by Schreier's toolbox[5]. As one can plainly see, the simulations are very close to the behavioral simulation. The differences become larger at low signal amplitudes—simulation accuracy becomes more important—and larger signal amplitudes—starting to reach the brink of instability. Time limitations prevented more data from being gathered.

6.2 Sixth Order, 32X Oversampling, Bandpass Delta-Sigma Modulator

As another test of this tool, a generated sixth order, thirty two times oversampling bandpass delta-sigma modulator is evaluated through simulation as above. The band of interest is around one fourth of the sampling frequency, or 2.08 megahertz in this case. Figures 6-6 and 6-7 display the spectra of the simulated and behavioral bandpass delta-sigma modulator with a sixty percent of full scale input signal amplitude, respectively.

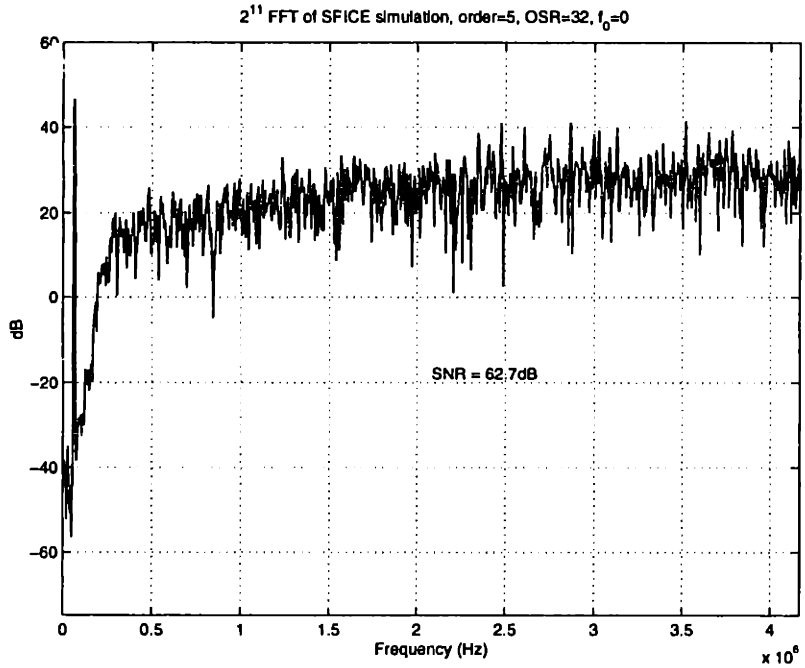


Figure 6-2: Spectrum of SPICE Simulation of a 5th order, 32 Times Oversampling Low Pass Delta-Sigma Modulator

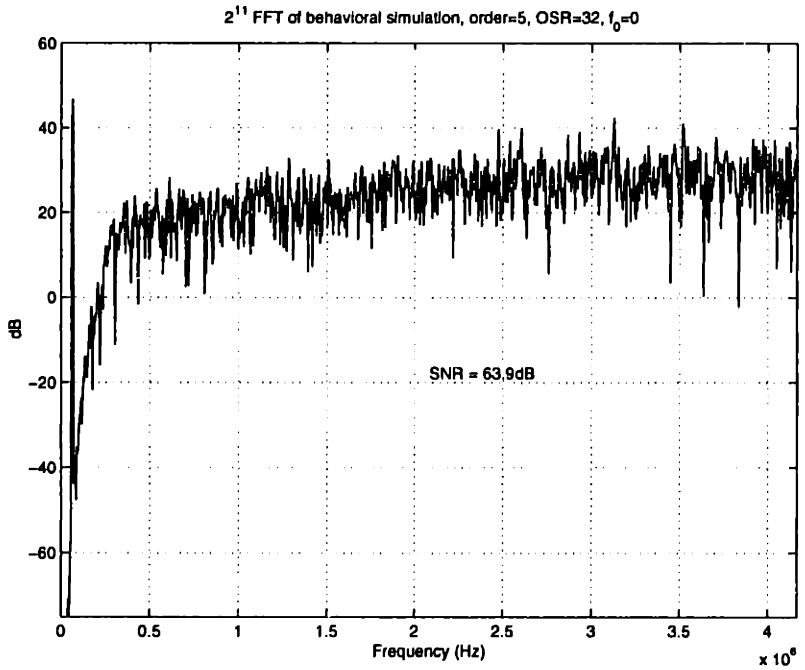


Figure 6-3: Spectrum of Behavioral Simulation of a 5th order, 32 Times Oversampling Low Pass Delta-Sigma Modulator

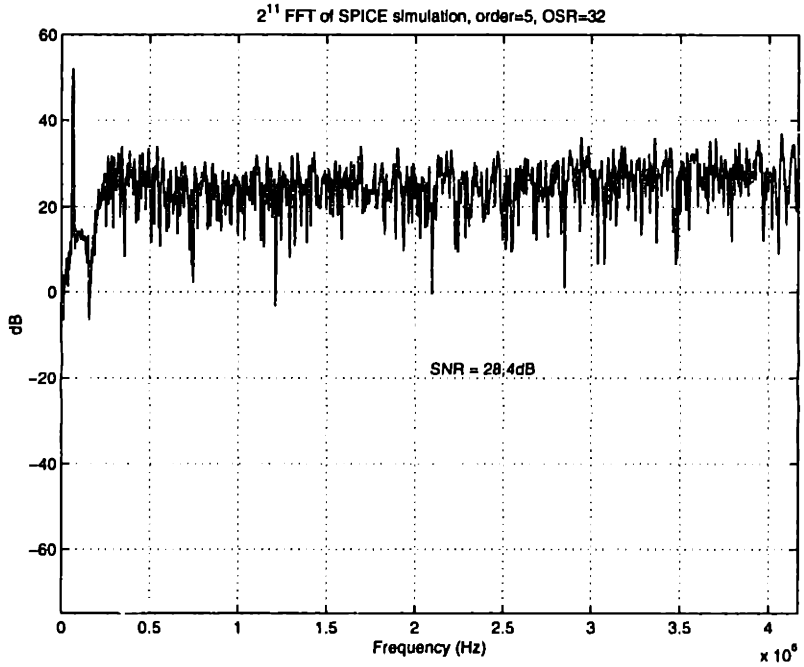


Figure 6-4: Spectrum of Behavioral Simulation of a 5th order, 32 Times Oversampling Low Pass Delta-Sigma Modulator with Unstable Output

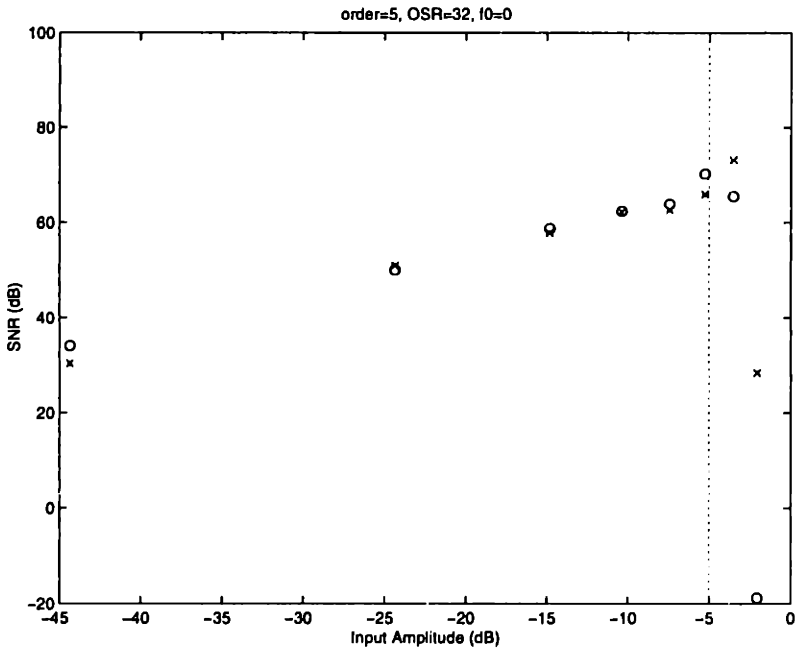


Figure 6-5: Plot of SNR versus Input Amplitude (Normalized to Supply Voltage) for a 5th order, 32 Times Oversampling Low Pass Delta-Sigma Modulator. X–SPICE Simulation, O–Behavioral Simulation

The spectra are very similar. Quantization noise is suppressed at the center frequency. The input signal is clearly seen as it is placed at the center frequency. In the HSPICE spectrum, one again observes the numerical inaccuracy which raises the noise floor around the band of interest. One major discrepancy is the larger tones outside the band of interest which are more closely spaced in the HSPICE simulated spectrum. The origin of this is not known and in any case does not matter because they will eventually be filtered out.

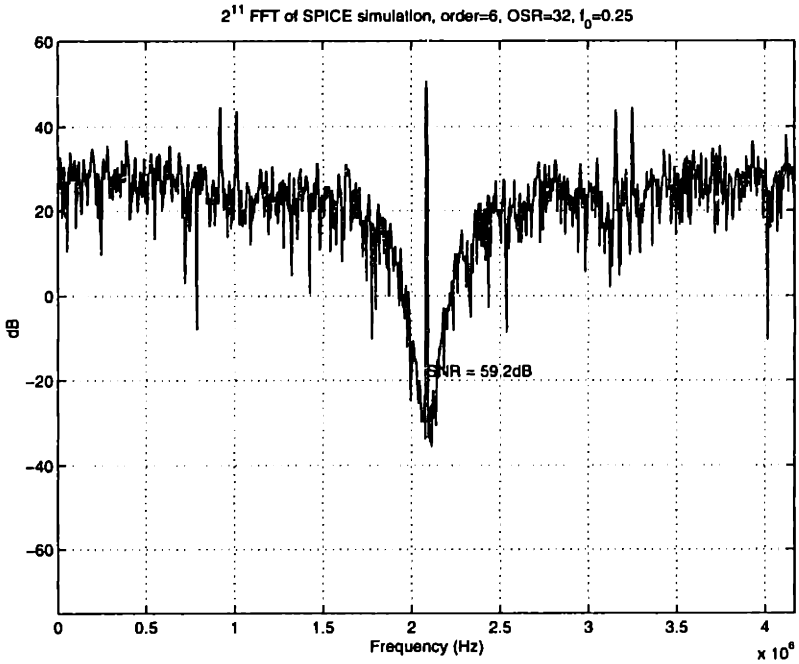


Figure 6-6: Spectrum of SPICE Simulation of a 6th order, 32 Times Oversampling Band Pass Delta-Sigma Modulator

Figure 6-8 shows the SNR versus the input signal amplitude. The dotted line is the rough stability threshold given by Schreier's toolbox[5]. As one can see, the HSPICE simulation results are very close to the behavioral simulation. The behavioral simulation results seems to oscillate a bit while the HSPICE simulated SNR trend is more linear, but no difference is larger than 4 decibels. Simulation accuracy may also contribute to some of the discrepancy.

6.3 Synthesis Statistics

On average, the synthesis of the delta-sigma modulator from user input to the SPICE circuit generation takes an hour on a SPARC Station 10 running MATLAB version 5.2[19].

As a measure of the power and area of the design, table 6.1 gives some measures of the generated designs. The numbers do not seem overly extravagant, and at the same time, they are not pushing the performance envelope, either.

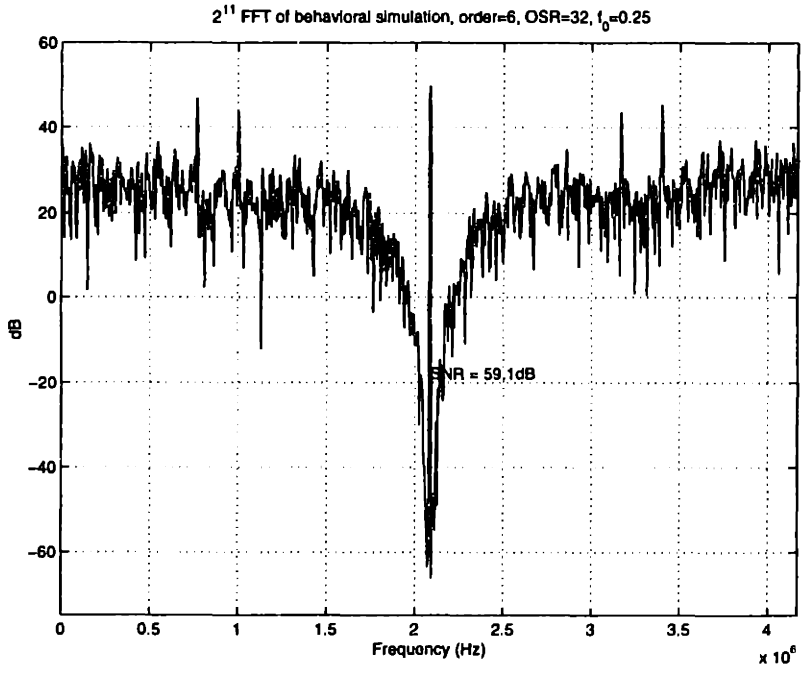


Figure 6-7: Spectrum of Behavioral Simulation of a 6th order, 32 Times Oversampling Band Pass Delta-Sigma Modulator

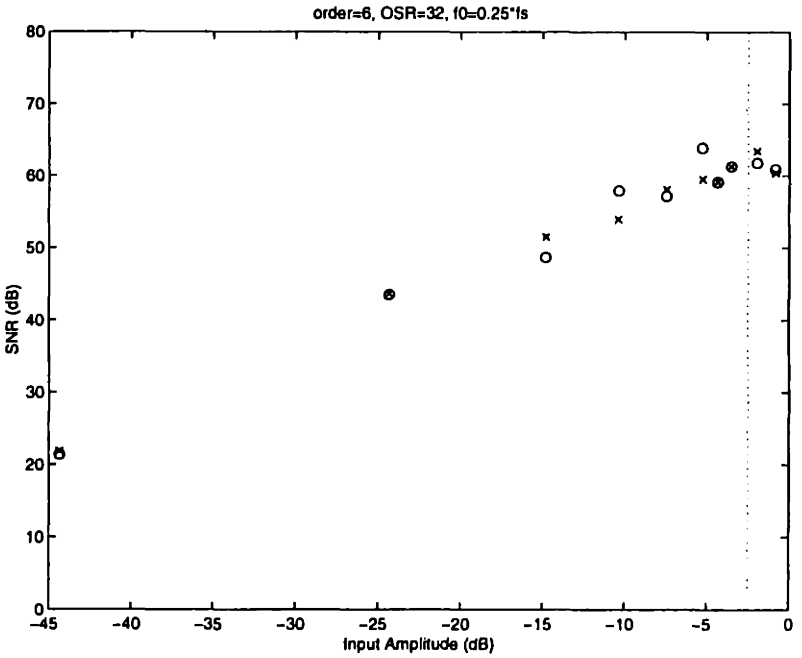


Figure 6-8: Plot of SNR versus Input Amplitude (Normalized to Supply Voltage) for a 6th order, 32 Times Oversampling Band Pass Delta-Sigma Modulator. X–SPICE Simulation, O–Behavioral Simulation.

Metric	5th Order, 32x, Low Pass	6th Order, 32x, Bandpass
Total Capacitance	17.2 pF	34.6 pF
Total Switch Area	$1.29 * 10^{-10} m^2$	$1.81 * 10^{-10} m^2$
Opamp Area	$1.09 * 10^{-8} m^2$	$2.40 * 10^{-8} m^2$
Static Power	22.5 mW	49.5 mW

Table 6.1: Measures of Designs Generated. Note that the area figures are just the transistor widths times the lengths.

Chapter 7

Conclusions and Future Work

As has been shown, this tool synthesizes low pass and bandpass oversampling delta-sigma modulators. The user can specify a oversampling ratio, order, center frequency, and minimum first stage capacitance. This set of specifications allows for maximum generality in generating a desired modulator while ensuring a practical design. In fact, this constrained space is the key behind this tool, turning a possibly intractable problem into a circumscribed and efficient process. The central impetus, indeed, is practicality.

A modification to the delta-sigma modulator for higher SNR ratios is the use of a delayed clock for switches connected to the input sampling capacitors of each stage (operational amplifier). In the results section, the SNR's were of moderate value and the effects of charge injection were not as important. However, at higher SNR's charge injection becomes an issue and steps must be taken to mitigate this effect. One place where its effect will clearly be seen is at the nodes of input sampling capacitors that will be connected to the virtual ground node of an operational amplifier. Using a delayed clock for these switches, that is the switches opening at a later time than the switches connected to the other node of the capacitors in question, would reduce the effects of signal dependent charge injection. By opening these switches later, the charge injection is always the same which only adds an offset. Furthermore, this effect is most pronounced in the first stage since these effects in later stages are shaped by the loop. To be specific, the switches that would be controlled by delayed clocks are switches 08 and 16 in the even order start section, 06, 15, and 24 in the odd order start section, and 09 and 17 in the cascade sections. The use of a delayed clock for these switches should help yield a higher signal to noise ratio. The proper operation of the delta-sigma modulator with this modification has been verified.

Another improvement for the tool is a more aggressive operational amplifier synthesis strategy. Currently, the operational amplifiers are scaled for the load capacitance which has about 80 dB SNR settling accuracy[1]. With the conservative safety margins, the operational amplifiers could probably settle for 100 dB SNR. Clearly, this is not needed for lower resolution modulators and there is an opportunity for power savings.

Also, in the opamp generation, the noise issue of the operational amplifiers should be analyzed more closely.

One useful addition would be the choice of loop filter topologies. Different topologies have different advantages and disadvantages and user based selection would be useful. In this version, the tool only uses the cascade of resonators with local feedback and input

[1]This is for this clock frequency of 8.3 MHz.

feedforward.

Perhaps, the ultimate improvement, or addition rather, is to include a silicon compiler such that the physical design is also generated. A gargantuan task such as this would not be easy, but could be done if the same type of constraint optimization is used. Effectively, such a complete tool would obviate the need of an analog designer for general purpose analog circuits, much the same as digital CAD tools.

However, with "improvements" comes the peril of reverting to the hazards that this tool tries to eschew. Modifications should not needlessly balloon synthesis time or necessarily compromise circuit feasibility. Constrained optimization is still the central idea of this tool and cannot be convoluted or muddled. Otherwise, the whole basis and foundation of this thesis and its work are undermined. As in life, a balance must be maintained.

Appendix A

MATLAB Scripts for Delta-Sigma Modulator Synthesis

The following are the MATLAB[19] scripts used to synthesize the delta-sigma modulators. Together, they are in fact the tool. For these scripts to work, Schreier's tool[5] must be installed.

The first script calls upon the two subsequent scripts as subroutines.

The tool produces a file containing the general structure and files of the necessary opamps and comparators.

A.1 Main Script adcsynth.m

```
% Matlab Script File to Generate A SPICE FILE of a Delta Sigma A2D
% Mark Shane Peng, Started 12/19/97
% Last Updated 29/10/98
% Uses R. Schreier's Delta Sigma Toolbox (needs optimization toolbox)
% Version 15 - CRFB structure for BandPass
% Fixed pass transistors

% Measure time to make
timetoc=cputime;

% Must be set: order, filename2, oversampling ratio, name2

% Default Values
if (exist('OSR','var'));
else OSR = 32,
end;
if (exist('order','var'));
else order = 2,
end;
if (exist('filename2','var'));
else filename2 = 'delsigadc.sp',
end;
if (exist('name2','var'));
else name2 = 'delsigadc',
end;
if (exist('f0','var'));
else f0 = 0;
end;
filename = 'comp.ckt';
cktname = 'comp';
if (exist('cmin','var'));
else cmin = 100; % Minimum Capacitor for kT/C (in fF)
end;

% Error Checking
if (order < 2) 'Error -- order too low'; pause; end;
if (OSR < 2) 'Error -- OSR too low'; pause; end;


% Other Parameters, predetermined
N = 2^11;  % Number of Points for FFT
fB = ceil(N/(2*OSR)); % Normalized bandwidth of modulator (bin)
```

```
fs = 1/120e-9; % Sampling Frequency
if (f0 == 0) f = fs*M/(OSR*2*2)/M ; % 65.104e3; % 9.7656e4; % Low Pass Case
else f = f0*fs; end; % Bandpass case
fn = round(f/fs*N); % Normalized inputfreq (bin)
t = 1205e-8:12e-8:25780e-8; % Time Vector
tp = 50e-9; % Time for which a clock is high
rmax = 15e3; % Ohms
Cox = 3.9*8.85e-12/100e-10; % F/m^2
coeffthres = 1e-4;

% Establish some global variables

n = 0; % For iterating sections on, specifies what order is being processed

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Generate Coefficients for CRFB structure %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Run Schreier's program to get coefficients with dynamic range scaling
H = synthesizeNTF(order,OSR,0,1.5,f0); % Synthesize the coefficients
u = .5*sin(2*pi*f*t); % Simulated time input signal
v = simulateDSM(u,H); % Simulated input signal
spec = fft(v.*hann(N)); % Windowed DFT
minf = fn+1-floor(fB/2) % Band-of-Interest Edges (bin)
maxf = fn+(fB/2) % Band-of-Interest Edges (bin)
if(minf < 0) minf = 0; maxf=0; end;
SNR = calculateSNR(spec(minf:maxf),round(fB/2))
uSNR = 10^(SNR/20)*10 % With safety factor
[a,g,b,c] = realizeNTF(H,'CRFB') % CRFB architecture
ABCD = stuffABCD(a,g,b,c,'CRFB')
[ABCDs,umax] = scaleABCD(ABCD,2,f0,1.0) % Dynamic Range Scaling
[a,g,b,c] = mapABCD(ABCDs,'CRFB')

% Prefilter the output coefficients to remove really small values
a(find(abs(a)<coeffthres)) = 0;
g(find(abs(g)<coeffthres)) = 0;
b(find(abs(b)<coeffthres)) = 0;
c(find(abs(c)<coeffthres)) = 0;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Build the Delta Sigma Modulator Modulator %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Start (First) Section
if (mod(order,2) == 1) % Odd Order First Section

% Make all coefficients positive and store sign in auxiliary structure
% cvalstartsgn = [c1 c2 c4 c5 c6 c7 c8 c9 c10 c11 c12 c13]
'odd order'
n = n+3
cvalstartsgna = 'aaaaaaaaaaaaa';
cvalstartsgnb = 'bbbbbbbbbbbbb';
if (a(1) < 0) cvalstartsgna(2) = 'b'; cvalstartsgnb(2) = 'a'; end;
if (b(1) < 0) cvalstartsgna(3) = 'b'; cvalstartsgnb(3) = 'a'; end;
if (a(2) < 0) cvalstartsgna(7) = 'b'; cvalstartsgnb(7) = 'a'; end;
if (b(2) < 0) cvalstartsgna(8) = 'b'; cvalstartsgnb(8) = 'a'; end;
if (a(3) < 0) cvalstartsgna(11) = 'b'; cvalstartsgnb(11) = 'a'; end;
if (b(3) < 0) cvalstartsgna(12) = 'b'; cvalstartsgnb(12) = 'a'; end;

% Store coefficients in structure and initialize switch structure
% cvalstart = [c1 c2 c4 c5 c6 c7 c8 c9 c10 c11 c12 c13]
% switstart = [1 thru 24] - width in microns
cvalstart = abs([1 a(1) b(1) g(1) 1 c(1) a(2) b(2) 1 c(2) a(3) b(3)]);
switstart = [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1];


elseif (mod(order,2) == 0) % Even Order Start Section
% Make all coefficients positive and store sign in auxiliary structure
% cvalstartsgn = [c1 c2 c4 c5 c6 c7 c8 c0]
'even order'
n = n+2
cvalstartsgna = 'aaaaaaaa';
cvalstartsgnb = 'bbbbbbbb';
if (a(1) < 0) cvalstartsgna(2) = 'b'; cvalstartsgnb(2) = 'a'; end;
if (b(1) < 0) cvalstartsgna(3) = 'b'; cvalstartsgnb(3) = 'a'; end;
if (a(2) < 0) cvalstartsgna(7) = 'b'; cvalstartsgnb(7) = 'a'; end;
if (b(2) < 0) cvalstartsgna(8) = 'b'; cvalstartsgnb(8) = 'a'; end;

% Store coefficients in structure and initialize switch structure
% cvalstart = [c1 c2 c4 c5 c6 c7 c8 c9]
% switstart = [1 thru 18] - width in microns
cvalstart = abs([1 a(1) b(1) g(1) 1 c(1) a(2) b(2)]);
switstart = [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1];

else 'error with building first section';
end;

% Middle Cascaded Sections
while ((n+2) < (order+1))
n = n+2

if (n > order) 'error in cascading', break;
end;
```

```
% Make all coefficients positive and store sign in auxiliary structure
% cvalcascsgn(n,:) = [c1 c2 c4 c5 c6 c7 c8 c9]
'even casc'
cvalcascsgna(n,:) = 'aaaaaaaaa';
cvalcascsgnb(n,:) = 'bbbbbbbbb';
if (a(n-1) < 0) cvalcascsgna(n,2) = 'b'; cvalcascsgnb(n,2) = 'a'; end;
if (b(n-1) < 0) cvalcascsgna(n,4) = 'b'; cvalcascsgnb(n,4) = 'a'; end;
if (a(n) < 0) cvalcascsgna(n,8) = 'b'; cvalcascsgnb(n,8) = 'a'; end;
if (b(n) < 0) cvalcascsgna(n,9) = 'b'; cvalcascsgnb(n,9) = 'a'; end;

% Store coefficients in structure and initialize switch structure
% cvalcasc(n,:) = [c1 c2 c3 c4 c5 c6 c7 c8 c9]
% switcasc(n,:) = [1 thru 18] - width in microns
cvalcasc(n,:) = abs([1 a(n-1) c(n-2) b(n-1) g(floor(n/2)) 1 c(n-1) a(n) b(n)]);
switcasc(n,:) = [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1];
end;

% End ADC Section
% Add last cap, change values, add comparator and dac
% cvalend = [c1]
'end section'
cvalend = [b(n+1)/c(n)];
switend = [1];

% Fix up the capacitor values to accommodate last capacitor
% Case if the order of the converter is only 2
if (n == 2)
cvalstart(3) = b(n-1) + b(n+1)*g(floor(n/2))/c(n);
if (cvalstart(3) > 0) cvalstartsgna(3) = 'a'; cvalstartsgnb(3) = 'b'; end;
if (cvalstart(3) < 0) cvalstartsgna(3) = 'b'; cvalstartsgnb(3) = 'a'; end;
cvalstart(4) = g(floor(n/2));
cvalstart(6) = c(n-1);
cvalstart(7) = a(n);
cvalstart(8) = b(n);
cvalstart = abs(cvalstart);

% Case if the order of the converter is only 3
elseif (n == 3)
cvalstart(8) = b(n-1) + b(n+1)*g(floor(n/2))/c(n);
if (cvalstart(8) > 0) cvalstartsgna(8) = 'a'; cvalstartsgnb(8) = 'b'; end;
if (cvalstart(8) < 0) cvalstartsgna(8) = 'b'; cvalstartsgnb(8) = 'a'; end;
cvalstart(4) = g(floor(n/2));
cvalstart(10) = c(n-1);
cvalstart(11) = a(n);
cvalstart(12) = b(n);
cvalstart = abs(cvalstart);

% General case for order greater than 3
else
cvalcasc(n,4) = b(n-1) + b(n+1)*g(floor(n/2))/c(n);
if (cvalcasc(n,4) > 0) cvalcascsgna(n,4) = 'a'; cvalcascsgnb(n,4) = 'b'; end;
if (cvalcasc(n,4) < 0) cvalcascsgna(n,4) = 'b'; cvalcascsgnb(n,4) = 'a'; end;
cvalcasc(n,5) = g(floor(n/2));
cvalcasc(n,7) = c(n-1);
cvalcasc(n,8) = a(n);
cvalcasc(n,9) = b(n);
cvalcasc(n,:) = abs(cvalcasc(n,:));
end;


% Scale capacitors for minimum capacitor - watch for zero caps
% Find the smallest non-zero capacitor at a summing node and scale all capacitors
% Accordingly
% Final values are in picofarads
n = 0; % Reinitialize global variable

% Odd Order Start Section
if(mod(order,2) == 1)
n = n+3;
c1=1; c2=2; c4=3; c5=4; c6=5; c7=6; c8=7; c9=8; c10=9; c11=10; c12=11; c13=12;
ctemp = [cvalstart(c1) cvalstart(c2) cvalstart(c4)]
ko1 = min(ctemp(find(ctemp)))
cvalstart(c1) = cvalstart(c1)/ko1;
cvalstart(c2) = cvalstart(c2)/ko1;
cvalstart(c4) = cvalstart(c4)/ko1;
ctemp = [cvalstart(c5) cvalstart(c6) cvalstart(c7) cvalstart(c8) cvalstart(c9)]
ko2 = min(ctemp(find(ctemp)))
cvalstart(c5) = cvalstart(c5)/ko2;
cvalstart(c6) = cvalstart(c6)/ko2;
cvalstart(c7) = cvalstart(c7)/ko2;
cvalstart(c8) = cvalstart(c8)/ko2;
cvalstart(c9) = cvalstart(c9)/ko2;
ctemp = [cvalstart(c10) cvalstart(c11) cvalstart(c12) cvalstart(c13)]
ko3 = min(ctemp(find(ctemp)))
cvalstart(c10) = cvalstart(c10)/ko3;
cvalstart(c11) = cvalstart(c11)/ko3;
cvalstart(c12) = cvalstart(c12)/ko3;
cvalstart(c13) = cvalstart(c13)/ko3;
cvalstart = cvalstart*.1*cmin/100; % Change to pF, account for min cap. size

% Even Order Start Section
```

```
else
n = n+2;
c1=1; c2=2; c4=3; c5=4; c6=5; c7=6; c8=7; c9=8;
ctemp = [cvalstart(c1) cvalstart(c2) cvalstart(c4) cvalstart(c5)]
ko1 = min(ctemp(find(ctemp)))
cvalstart(c1) = cvalstart(c1)/ko1;
cvalstart(c2) = cvalstart(c2)/ko1;
cvalstart(c4) = cvalstart(c4)/ko1;
cvalstart(c5) = cvalstart(c5)/ko1;
ctemp = [cvalstart(c6) cvalstart(c7) cvalstart(c8) cvalstart(c9)]
ko2 = min(ctemp(find(ctemp)))
cvalstart(c6) = cvalstart(c6)/ko2;
cvalstart(c7) = cvalstart(c7)/ko2;
cvalstart(c8) = cvalstart(c8)/ko2;
cvalstart(c9) = cvalstart(c9)/ko2;
cvalstart = cvalstart*.1*cmin/100; % Change to pF
end;

% Cascaded Sections
while ((n+2) < (order+1))
n = n+2;
cvals = cvalcasc(n,:)
ctemp = [cvals(1) cvals(2) cvals(3) cvals(4) cvals(5)]
ko1 = min(ctemp(find(ctemp)))
cvals(1) = cvals(1)/ko1;
cvals(2) = cvals(2)/ko1;
cvals(3) = cvals(3)/ko1;
cvals(4) = cvals(4)/ko1;
cvals(5) = cvals(5)/ko1;
ctemp = [cvals(6) cvals(7) cvals(8) cvals(9)]
ko2 = min(ctemp(find(ctemp)))
cvals(6) = cvals(6)/ko2;
cvals(7) = cvals(7)/ko2;
cvals(8) = cvals(8)/ko2;
cvals(9) = cvals(9)/ko2;
cvals = cvals*.1; % Change to pF
cvalcasc(n,:) = cvals
end;

if(order == 3)
cvalend = cvalend/ko3*.1; % Fix last cap.  Always greater than least.
else
cvalend = cvalend/ko2*.1; % Fix last cap.  Always greater than least.
end;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Pass Transistor/Switch Sizing - Cvalues must be in picofarads %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
n = 0; % Reinitialize global variable

% Odd Order Start Section
if (mod(order,2) == 1)
n = n+3;
c1=1; c2=2; c4=3; c5=4; c6=5; c7=6; c8=7; c9=8; c10=9; c11=10; c12=11; c13=12;
switstart(1) = cvalstart(c2);
switstart(2) = cvalstart(c2);
switstart(3) = cvalstart(c4);
switstart(4) = cvalstart(c4);
switstart(5) = cvalstart(c2) + cvalstart(c4);
switstart(6) = cvalstart(c2) + cvalstart(c4);
switstart(7) = cvalstart(c7);
switstart(8) = cvalstart(c7);
switstart(9) = cvalstart(c8);
switstart(10) = cvalstart(c8);
switstart(11) = cvalstart(c9);
switstart(12) = cvalstart(c9);
switstart(13) = cvalstart(c5);
switstart(14) = cvalstart(c5);
switstart(15) = cvalstart(c5) + cvalstart(c7) + cvalstart(c8) + cvalstart(c9);
switstart(16) = cvalstart(c5) + cvalstart(c7) + cvalstart(c8) + cvalstart(c9);
switstart(17) = cvalstart(c11);
switstart(18) = cvalstart(c11);
switstart(19) = cvalstart(c12);
switstart(20) = cvalstart(c12);
switstart(21) = cvalstart(c13);
switstart(22) = cvalstart(c13);
switstart(23) = cvalstart(c11) + cvalstart(c12) + cvalstart(c13);
switstart(24) = cvalstart(c11) + cvalstart(c12) + cvalstart(c13);

% Even Order Start Section
else
n = n+2;
c1=1; c2=2; c4=3; c5=4; c6=5; c7=6; c8=7; c9=8;
switstart(1) = cvalstart(c2);
switstart(2) = cvalstart(c2);
switstart(3) = cvalstart(c4);
switstart(4) = cvalstart(c4);
switstart(5) = cvalstart(c5);
switstart(6) = cvalstart(c5);
switstart(7) = cvalstart(c2) + cvalstart(c4) + cvalstart(c5);
switstart(8) = cvalstart(c2) + cvalstart(c4) + cvalstart(c5);
switstart(9) = cvalstart(c7);
```

```
switstart(10) = cvalstart(c7);
switstart(11) = cvalstart(c8);
switstart(12) = cvalstart(c8);
switstart(13) = cvalstart(c9);
switstart(14) = cvalstart(c9);
switstart(15) = cvalstart(c7) + cvalstart(c8) + cvalstart(c9);
switstart(16) = cvalstart(c7) + cvalstart(c8) + cvalstart(c9);
end;

switstart = log(uSNR)*rmax*switstart*1e-12/tp*2; % Convert to a factor + Safety

% Cascaded Sections
while ((n+2) < (order+1))
n = n+2;
c1=1; c2=2; c3=3; c4=4; c5=5; c6=6; c7=7; c8=8; c9=9;
switcasc(n,1) = cvalcasc(n,c2);
switcasc(n,2) = cvalcasc(n,c2);
switcasc(n,3) = cvalcasc(n,c4);
switcasc(n,4) = cvalcasc(n,c4);
switcasc(n,5) = cvalcasc(n,c5);
switcasc(n,6) = cvalcasc(n,c5);
switcasc(n,7) = cvalcasc(n,c3);
switcasc(n,8) = cvalcasc(n,c3);
switcasc(n,9) = cvalcasc(n,c2) + cvalcasc(n,c3) + cvalcasc(n,c4) + cvalcasc(n,c5);
switcasc(n,10) = cvalcasc(n,c2) + cvalcasc(n,c3) + cvalcasc(n,c4) + cvalcasc(n,c5);
switcasc(n,11) = cvalcasc(n,c7);
switcasc(n,12) = cvalcasc(n,c7);
switcasc(n,13) = cvalcasc(n,c8);
switcasc(n,14) = cvalcasc(n,c8);
switcasc(n,15) = cvalcasc(n,c9);
switcasc(n,16) = cvalcasc(n,c9);
switcasc(n,17) = cvalcasc(n,c7) + cvalcasc(n,c8) + cvalcasc(n,c9);
switcasc(n,18) = cvalcasc(n,c7) + cvalcasc(n,c8) + cvalcasc(n,c9);

switcasc(n,:) = log(uSNR)*rmax*switcasc(n,:)*1e-12/tp*2;
end;

% End Section
switend = cvalend;

switend = log(uSNR)*rmax*switend*1e-12/tp*2;

%%%%%%%%%%%%%%%%%%
% DAC Synthesis %
%%%%%%%%%%%%%%%%%%
n = 0; % Reinitialize global variable
dacsize = 0;

% Add up the switches connected to DAC, and multiply by 2
if (mod(order,2) == 1)
n = n+3;
switstart(1) = switstart(1)*2;
switstart(9) = switstart(9)*2;
switstart(19) = switstart(19)*2;
dacsize = switstart(1) + switstart(9) + switstart(19);
else
n = n+2;
switstart(1) = switstart(1)*2;
switstart(11) = switstart(11)*2;
dacsize = switstart(1) + switstart(11);
end;

for iter = 1:length(switstart); % Minimum size switch
  if(switstart(iter) < 1) switstart(iter) = 1; end;

end;

while ((n+2) < (order+1))
n = n+2;
switcasc(n,1) = switcasc(n,1)*2;
switcasc(n,13) = switcasc(n,13)*2;
dacsize = dacsize + switcasc(n,1) + switcasc(n,13);

for iter = 1:length(switcasc(n,:));
  if(switcasc(n,iter) < 1) switcasc(n,iter) = 1; end;
end;
end;

dacsize = dacsize*1; % Safety Factor

if(dacsize < 1) dacsize = 1; end; % Minimum size for DAC


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% DAC Capacitance Calculation and Comparator Synthesis %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Cdac = 2*dacsize*1e-6*.5e-6*Cox*2.0; % Two for n/pmos, Safety factor, Worst CAse
cload_comp = Cdac*1e15*1; % Convert to FemtoFarads with safety factor
compsyn % Make call to synthesize comparator

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Calculate Load Capacitance and OpAmp Synthesis %
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
cloads = zeros(1,order);     % in pF
cin_comp = k_comp*1e-6*.5e-6*Cox;
c_nextstage = cin_comp*1e12; %  + cvalend;

n = order; % Reinitialize global variable

% Start in reverse order
while(n > 3)
cloads(n) = cvalcasc(n,5) + cvalcasc(n,6) + c_nextstage;
cloads(n-1) = cvalcasc(n,1) + cvalcasc(n,7);
c_nextstage = cvalcasc(n,3);
n = n-2;
end;

if(n == 3)
cloads(n) = cvalstart(4) + cvalstart(9) + c_nextstage;
cloads(n-1) = cvalstart(5) + cvalstart(10);
cloads(n-2) = cvalstart(1) + cvalstart(6);
elseif(n == 2)
cloads(n) = cvalstart(4) + cvalstart(5) + c_nextstage;
cloads(n-1) = cvalstart(1) + cvalstart(6);
else 'Error -- opamp scaling'; pause;
end;

% Make opamps
cloads = cloads*1; % Safety Factor
for iter = 1:order;
filename = sprintf('opamp%d.ckt',iter);
cktname = sprintf('dopamp%d',iter); % Called dopamp#
if(cloads(iter) < 0.1) cload_user = 0.1;
else cload_user = cloads(iter);
end;
opsyn;
end;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Write Complete Delta Sigma Core to a file %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
n = 0; % Reinitialize
fid2 = fopen(filename2,'w');
if fid2 == -1
filename2 = input('File write unsuccessful, Other Output Files: ')
end;

% strin = sprintf('.subckt %s vina vinb vouta voutb vdd vss vphi1 vphi1b vphi2 vphi2b\n',name2);
% fprintf(fid2, strin);

% Include comparator and opamp
fprintf(fid2,'.include comp.ckt\n');

% Include Opamps
for iter = 1:order;
string = sprintf('opamp%d.ckt',iter);
fprintf(fid2,'.include %s\n',string);
end;

% Change the capacitor values to Farads
cvalstart = cvalstart*1e-12; %%%%%%%%%%%%%%%%%%%%%%%%
if(exist('cvalcasc','var')) cvalcasc = cvalcasc*1e-12; end;
cvalend = cvalend*1e-12;

% Change switch/DAC values to Microns
switstart = switstart*1e-6;
if(exist('switcasc','var')) switcasc = switcasc*1e-6; end;
switend = switend*1e-6;
dacsize = dacsize*1e-6;

%%%%%%%%%%%%%%%%%%%%%%%%%%
% Do a little Tabulation %
%%%%%%%%%%%%%%%%%%%%%%%%%%
cload_total = sum(cloads)
if(exist('cvalcasc','var')) cap_total = sum(cvalstart) + sum(sum(cvalcasc));
else cap_total = sum(cvalstart);
end;
cap_total = cap_total + cvalend + cload_total/.1*340e-15
if(exist('switcasc','var')) swit_total = sum(switstart) + sum(sum(switcasc));
else swit_total = sum(switstart);
end;
swit_total = swit_total + switend + 4*dacsize
power_total = cload_total/.1*200e-6*3.3
opamp_area = cload_total/.1*320e-12

% Write the Switch subcircuit
fprintf(fid2,'.subckt switch term1 term2 cntl cntlb vdd vss wid=1e-6\n');
fprintf(fid2,'mswitch1 term1 cntl term2 vss nfet w=''wid'' l=.5u as=''4e-6*wid'' ad=''2e-6*wid'' ps=''8e-6+wid'' pd=4u\n');
fprintf(fid2,'mswitch2 term1 cntlb term2 vdd pfet w=''wid'' l=.5u as=''4e-6*wid'' ad=''2e-6*wid'' ps=''8e-6+wid'' pd=4u\n');
fprintf(fid2,'.ends switch\n\n');

% Write beginning section
if(mod(order,2) == 1)
n = n+3;
```

```
'odd order write'

fprintf(fid2,'x%ds1a vdaca n%dv1a vphi1 vphi1b vdd vss switch wid=%e\n',n,n,switstart(1));
fprintf(fid2,'x%ds1b vdacb n%dv1b vphi1 vphi1b vdd vss switch wid=%e\n',n,n,switstart(1));
fprintf(fid2,'x%ds2a 0 n%dv1a vphi2 vphi2b vdd vss switch wid=%e\n',n,n,switstart(2));
fprintf(fid2,'x%ds2b 0 n%dv1b vphi2 vphi2b vdd vss switch wid=%e\n',n,n,switstart(2));

fprintf(fid2,'x%ds3a vina n%dv2a vphi1 vphi1b vdd vss switch wid=%e\n',n,n,switstart(3));
fprintf(fid2,'x%ds3b vinb n%dv2b vphi1 vphi1b vdd vss switch wid=%e\n',n,n,switstart(3));
fprintf(fid2,'x%ds4a 0 n%dv2a vphi2 vphi2b vdd vss switch wid=%e\n',n,n,switstart(4));
fprintf(fid2,'x%ds4b 0 n%dv2b vphi2 vphi2b vdd vss switch wid=%e\n',n,n,switstart(4));

fprintf(fid2,'x%ds5a n%dv3a n%dv4a vphi2 vphi2b vdd vss switch wid=%e\n',n,n,n,switstart(5));
fprintf(fid2,'x%ds5b n%dv3b n%dv4b vphi2 vphi2b vdd vss switch wid=%e\n',n,n,n,switstart(5));
fprintf(fid2,'x%ds6a 0 n%dv3a vphi1 vphi1b vdd vss switch wid=%e\n',n,n,switstart(6));
fprintf(fid2,'x%ds6b 0 n%dv3b vphi1 vphi1b vdd vss switch wid=%e\n',n,n,switstart(6));

fprintf(fid2,'x%ds7a n%dv5a n%dv6a vphi1 vphi1b vdd vss switch wid=%e\n',n,n,n,switstart(7));
fprintf(fid2,'x%ds7b n%dv5b n%dv6b vphi1 vphi1b vdd vss switch wid=%e\n',n,n,n,switstart(7));
fprintf(fid2,'x%ds8a 0 n%dv6a vphi2 vphi2b vdd vss switch wid=%e\n',n,n,switstart(8));
fprintf(fid2,'x%ds8b 0 n%dv6b vphi2 vphi2b vdd vss switch wid=%e\n',n,n,switstart(8));

fprintf(fid2,'x%ds9a vdaca n%dv8a vphi1 vphi1b vdd vss switch wid=%e\n',n,n,switstart(9));
fprintf(fid2,'x%ds9b vdacb n%dv8b vphi1 vphi1b vdd vss switch wid=%e\n',n,n,switstart(9));
fprintf(fid2,'x%ds10a 0 n%dv8a vphi2 vphi2b vdd vss switch wid=%e\n',n,n,switstart(10));
fprintf(fid2,'x%ds10b 0 n%dv8b vphi2 vphi2b vdd vss switch wid=%e\n',n,n,switstart(10));

fprintf(fid2,'x%ds11a vina n%dv9a vphi1 vphi1b vdd vss switch wid=%e\n',n,n,switstart(11));
fprintf(fid2,'x%ds11b vinb n%dv9a vphi1 vphi1b vdd vss switch wid=%e\n',n,n,switstart(11));
fprintf(fid2,'x%ds12a 0 n%dv9a vphi2 vphi2b vdd vss switch wid=%e\n',n,n,switstart(12));
fprintf(fid2,'x%ds12b 0 n%dv9b vphi2 vphi2b vdd vss switch wid=%e\n',n,n,switstart(12));

fprintf(fid2,'x%ds13a n%dvob n%dv10a vphi1 vphi1b vdd vss switch wid=%e\n',n,n,n,switstart(13));
fprintf(fid2,'x%ds13b n%dvoa n%dv10b vphi1 vphi1b vdd vss switch wid=%e\n',n,n,n,switstart(13));
fprintf(fid2,'x%ds14a 0 n%dv10a vphi2 vphi2b vdd vss switch wid=%e\n',n,n,switstart(14));
fprintf(fid2,'x%ds14b 0 n%dv10b vphi2 vphi2b vdd vss switch wid=%e\n',n,n,switstart(14));

fprintf(fid2,'x%ds15a n%dv7a n%dv11a vphi1 vphi1b vdd vss switch wid=%e\n',n,n,n,switstart(15));
fprintf(fid2,'x%ds15b n%dv7b n%dv11b vphi1 vphi1b vdd vss switch wid=%e\n',n,n,n,switstart(15));
fprintf(fid2,'x%ds16a 0 n%dv7a vphi2 vphi2b vdd vss switch wid=%e\n',n,n,switstart(16));
fprintf(fid2,'x%ds16b 0 n%dv7b vphi2 vphi2b vdd vss switch wid=%e\n',n,n,switstart(16));

fprintf(fid2,'x%ds17a n%dv12a n%dv13a vphi1 vphi1b vdd vss switch wid=%e\n',n,n,n,switstart(17));
fprintf(fid2,'x%ds17b n%dv12b n%dv13b vphi1 vphi1b vdd vss switch wid=%e\n',n,n,n,switstart(17));
fprintf(fid2,'x%ds18a 0 n%dv13a vphi2 vphi2b vdd vss switch wid=%e\n',n,n,switstart(18));
fprintf(fid2,'x%ds18b 0 n%dv13b vphi2 vphi2b vdd vss switch wid=%e\n',n,n,switstart(18));

fprintf(fid2,'x%ds19a vdaca n%dv15a vphi1 vphi1b vdd vss switch wid=%e\n',n,n,switstart(19));
fprintf(fid2,'x%ds19b vdacb n%dv15b vphi1 vphi1b vdd vss switch wid=%e\n',n,n,switstart(19));
fprintf(fid2,'x%ds20a 0 n%dv15a vphi2 vphi2b vdd vss switch wid=%e\n',n,n,switstart(20));
fprintf(fid2,'x%ds20b 0 n%dv15b vphi2 vphi2b vdd vss switch wid=%e\n',n,n,switstart(20));

fprintf(fid2,'x%ds21a vina n%dv16a vphi1 vphi1b vdd vss switch wid=%e\n',n,n,switstart(21));
fprintf(fid2,'x%ds21b vinb n%dv16b vphi1 vphi1b vdd vss switch wid=%e\n',n,n,switstart(21));
fprintf(fid2,'x%ds22a 0 n%dv16a vphi2 vphi2b vdd vss switch wid=%e\n',n,n,switstart(22));
fprintf(fid2,'x%ds22b 0 n%dv16b vphi2 vphi2b vdd vss switch wid=%e\n',n,n,switstart(22));

fprintf(fid2,'x%ds23a n%dv14a n%dv17a vphi2 vphi2b vdd vss switch wid=%e\n',n,n,n,switstart(23));
fprintf(fid2,'x%ds23b n%dv14b n%dv17b vphi2 vphi2b vdd vss switch wid=%e\n',n,n,n,switstart(23));
fprintf(fid2,'x%ds24a 0 n%dv14a vphi1 vphi1b vdd vss switch wid=%e\n',n,n,switstart(24));
fprintf(fid2,'x%ds24b 0 n%dv14b vphi1 vphi1b vdd vss switch wid=%e\n',n,n,switstart(24));

fprintf(fid2,'c1n%da n%dv4a n%dv5a %e\n',n,n,n,cvalstart(1));
fprintf(fid2,'c1n%db n%dv4b n%dv5b %e\n',n,n,n,cvalstart(1));
fprintf(fid2,'c2n%da n%dv1%c n%dv3a %e\n',n,n,cvalstartsgna(2),n,cvalstart(2));
fprintf(fid2,'c2n%db n%dv1%c n%dv3b %e\n',n,n,cvalstartsgnb(2),n,cvalstart(2));
fprintf(fid2,'c4n%da n%dv2%c n%dv3a %e\n',n,n,cvalstartsgna(3),n,cvalstart(3));
fprintf(fid2,'c4n%db n%dv2%c n%dv3b %e\n',n,n,cvalstartsgnb(3),n,cvalstart(3));
fprintf(fid2,'c5n%da n%dv10a n%dv7b %e\n',n,n,n,cvalstart(4));
fprintf(fid2,'c5n%db n%dv10b n%dv7a %e\n',n,n,n,cvalstart(4));
fprintf(fid2,'c6n%da n%dv11a n%dv12a %e\n',n,n,n,cvalstart(5));
fprintf(fid2,'c6n%db n%dv11b n%dv12b %e\n',n,n,n,cvalstart(5));
fprintf(fid2,'c7n%da n%dv6a n%dv7b %e\n',n,n,n,cvalstart(6));
fprintf(fid2,'c7n%db n%dv6b n%dv7a %e\n',n,n,n,cvalstart(6));
fprintf(fid2,'c8n%da n%dv8%c n%dv7b %e\n',n,n,cvalstartsgna(7),n,cvalstart(7));
fprintf(fid2,'c8n%db n%dv8%c n%dv7a %e\n',n,n,cvalstartsgnb(7),n,cvalstart(7));
fprintf(fid2,'c9n%da n%dv9%c n%dv7b %e\n',n,n,cvalstartsgna(8),n,cvalstart(8));
fprintf(fid2,'c9n%db n%dv9%c n%dv7a %e\n',n,n,cvalstartsgnb(8),n,cvalstart(8));
fprintf(fid2,'c10n%da n%dv17a n%dvoa %e\n',n,n,n,cvalstart(9));
fprintf(fid2,'c10n%db n%dv17b n%dvob %e\n',n,n,n,cvalstart(9));
fprintf(fid2,'c11n%da n%dv13a n%dv14a %e\n',n,n,n,cvalstart(10));
fprintf(fid2,'c11n%db n%dv13b n%dv14b %e\n',n,n,n,cvalstart(10));
fprintf(fid2,'c12n%da n%dv16%c n%dv14a %e\n',n,n,cvalstartsgna(11),n,cvalstart(11));
fprintf(fid2,'c12n%db n%dv16%c n%dv14b %e\n',n,n,cvalstartsgnb(11),n,cvalstart(11));
fprintf(fid2,'c13n%da n%dv16%c n%dv14a %e\n',n,n,cvalstartsgna(12),n,cvalstart(12));
fprintf(fid2,'c13n%db n%dv16%c n%dv14b %e\n',n,n,cvalstartsgnb(12),n,cvalstart(12));

fprintf(fid2,'xop1n%d n%dv4b n%dv4a n%dv5a n%dv5b vdd vss vcmfb1n%d vphi1 vphi1b vphi2 vphi2b dopamp%d\n',n,n,n,n,n,n,n,n-2);
fprintf(fid2,'* e1n%d n%dv5b n%dv5a vcvs n%dv4a n%dv4b 1e6 max=2.5 min=-2.5\n',n,n,n,n,n);
fprintf(fid2,'xop2n%d n%dv11b n%dv11a n%dv12a n%dv12b vdd vss vcmfb2n%d vphi1 vphi1b vphi2 vphi2b dopamp%d\n',n,n,n,n,n,n,n,n-1);
fprintf(fid2,'* e2n%d n%dv12b n%dv12a vcvs n%dv11a n%dv11b 1e6 max=2.5 min=-2.5\n',n,n,n,n,n);
fprintf(fid2,'xop3n%d n%dv17b n%dv17a n%dvoa n%dvob vdd vss vcmfb3n%d vphi1 vphi1b vphi2 vpbi2b dopamp%d\n',n,n,n,n,n,n,n,n);
```

```
fprintf(fid2,'* e3n%d n%dvob n%dvoa vcvs n%dv17a n%dv17b 1e6 max=2.5 min=-2.5\n',n,n,n,n,n);

else
n = n+2;
'even order write'

fprintf(fid2,'x%ds1a vdaca n%dv1a vphi1 vphi1b vdd vss switch wid=%e\n',n,n,switstart(1));
fprintf(fid2,'x%ds1b vdacb n%dv1b vphi1 vphi1b vdd vss switch wid=%e\n',n,n,switstart(1));
fprintf(fid2,'x%ds2a 0 n%dv1a vphi2 vphi2b vdd vss switch wid=%e\n',n,n,switstart(2));
fprintf(fid2,'x%ds2b 0 n%dv1b vphi2 vphi2b vdd vss switch wid=%e\n',n,n,switstart(2));

fprintf(fid2,'x%ds3a vina n%dv2a vphi1 vphi1b vdd vss switch wid=%e\n',n,n,switstart(3));
fprintf(fid2,'x%ds3b vinb n%dv2b vphi1 vphi1b vdd vss switch wid=%e\n',n,n,switstart(3));
fprintf(fid2,'x%ds4a 0 n%dv2a vphi2 vphi2b vdd vss switch wid=%e\n',n,n,switstart(4));
fprintf(fid2,'x%ds4b 0 n%dv2b vphi2 vphi2b vdd vss switch wid=%e\n',n,n,switstart(4));

fprintf(fid2,'x%ds5a n%dvob n%dv3a vphi1 vphi1b vdd vss switch wid=%e\n',n,n,n,switstart(5));
fprintf(fid2,'x%ds5b n%dvoa n%dv3b vphi1 vphi1b vdd vss switch wid=%e\n',n,n,n,switstart(5));
fprintf(fid2,'x%ds6a 0 n%dv3a vphi2 vphi2b vdd vss switch wid=%e\n',n,n,switstart(6));
fprintf(fid2,'x%ds6b 0 n%dv3b vphi2 vphi2b vdd vss switch wid=%e\n',n,n,switstart(6));

fprintf(fid2,'x%ds7a n%dv4a n%dv5a vphi1 vphi1b vdd vss switch wid=%e\n',n,n,n,switstart(7));
fprintf(fid2,'x%ds7b n%dv4b n%dv5b vphi1 vphi1b vdd vss switch wid=%e\n',n,n,n,switstart(7));
fprintf(fid2,'x%ds8a 0 n%dv4a vphi2 vphi2b vdd vss switch wid=%e\n',n,n,switstart(8));
fprintf(fid2,'x%ds8b 0 n%dv4b vphi2 vphi2b vdd vss switch wid=%e\n',n,n,switstart(8));

fprintf(fid2,'x%ds9a n%dv6a n%dv7a vphi1 vphi1b vdd vss switch wid=%e\n',n,n,n,switstart(9));
fprintf(fid2,'x%ds9b n%dv6b n%dv7b vphi1 vphi1b vdd vss switch wid=%e\n',n,n,n,switstart(9));
fprintf(fid2,'x%ds10a 0 n%dv7a vphi2 vphi2b vdd vss switch wid=%e\n',n,n,switstart(10));
fprintf(fid2,'x%ds10b 0 n%dv7b vphi2 vphi2b vdd vss switch wid=%e\n',n,n,switstart(10));

fprintf(fid2,'x%ds11a vdaca n%dv9a vphi1 vphi1b vdd vss switch wid=%e\n',n,n,switstart(11));
fprintf(fid2,'x%ds11b vdacb n%dv9b vphi1 vphi1b vdd vss switch wid=%e\n',n,n,switstart(11));
fprintf(fid2,'x%ds12a 0 n%dv9a vphi2 vphi2b vdd vss switch wid=%e\n',n,n,switstart(12));
fprintf(fid2,'x%ds12b 0 n%dv9b vphi2 vphi2b vdd vss switch wid=%e\n',n,n,switstart(12));

fprintf(fid2,'x%ds13a vina n%dv10a vphi1 vphi1b vdd vss switch wid=%e\n',n,n,switstart(13));
fprintf(fid2,'x%ds13b vinb n%dv10b vphi1 vphi1b vdd vss switch wid=%e\n',n,n,switstart(13));
fprintf(fid2,'x%ds14a 0 n%dv10a vphi2 vphi2b vdd vss switch wid=%e\n',n,n,switstart(14));
fprintf(fid2,'x%ds14b 0 n%dv10b vphi2 vphi2b vdd vss switch wid=%e\n',n,n,switstart(14));

fprintf(fid2,'x%ds15a n%dv8a n%dv11a vphi2 vphi2b vdd vss switch wid=%e\n',n,n,n,switstart(15));
fprintf(fid2,'x%ds15b n%dv8b n%dv11b vphi2 vphi2b vdd vss switch wid=%e\n',n,n,n,switstart(15));
fprintf(fid2,'x%ds16a 0 n%dv8a vphi1 vphi1b vdd vss switch wid=%e\n',n,n,switstart(16));
fprintf(fid2,'x%ds16b 0 n%dv8b vphi1 vphi1b vdd vss switch wid=%e\n',n,n,switstart(16));

fprintf(fid2,'c1n%da n%dv5a n%dv6a %e\n',n,n,n,cvalstart(1));
fprintf(fid2,'c1n%db n%dv5b n%dv6b %e\n',n,n,n,cvalstart(1));
fprintf(fid2,'c2n%da n%dv1%c n%dv4b %e\n',n,n,cvalstartsgna(2),n,cvalstart(2));
fprintf(fid2,'c2n%db n%dv1%c n%dv4a %e\n',n,n,cvalstartsgnb(2),n,cvalstart(2));
fprintf(fid2,'c4n%da n%dv2%c n%dv4b %e\n',n,n,cvalstartsgna(3),n,cvalstart(3));
fprintf(fid2,'c4n%db n%dv2%c n%dv4a %e\n',n,n,cvalstartsgnb(3),n,cvalstart(3));
fprintf(fid2,'c5n%da n%dv3a n%dv4b %e\n',n,n,n,cvalstart(4));
fprintf(fid2,'c5n%db n%dv3b n%dv4a %e\n',n,n,n,cvalstart(4));
fprintf(fid2,'c6n%da n%dv11a n%dvoa %e\n',n,n,n,cvalstart(5));
fprintf(fid2,'c6n%db n%dv11b n%dvob %e\n',n,n,n,cvalstart(5));
fprintf(fid2,'c7n%da n%dv7a n%dv8a %e\n',n,n,n,cvalstart(6));
fprintf(fid2,'c7n%db n%dv7b n%dv8b %e\n',n,n,n,cvalstart(6));
fprintf(fid2,'c8n%da n%dv9%c n%dv8a %e\n',n,n,cvalstartsgna(7),n,cvalstart(7));
fprintf(fid2,'c8n%db n%dv9%c n%dv8b %e\n',n,n,cvalstartsgnb(7),n,cvalstart(7));
fprintf(fid2,'c9n%da n%dv10%c n%dv8a %e\n',n,n,cvalstartsgna(8),n,cvalstart(8));
fprintf(fid2,'c9n%db n%dv10%c n%dv8b %e\n',n,n,cvalstartsgnb(8),n,cvalstart(8));

fprintf(fid2,'xop1n%d n%dv6b n%dv5a n%dv6a n%dv6b vdd vss vcmfb1n%d vphi1 vphi1b vphi2 vphi2b dopamp%d\n',n,n,n,n,n,n,n-1);
fprintf(fid2,'* e1n%d n%dv6b n%dv6a vcvs n%dv5a n%dv5b 1e6 max=2.5 min=-2.5\n',n,n,n,n,n);
fprintf(fid2,'xop2n%d n%dv11b n%dv11a n%dvoa n%dvob vdd vss vcmfb2n%d vphi1 vphi1b vphi2 vphi2b dopamp%d\n',n,n,n,n,n,n,n);
fprintf(fid2,'* e2n%d n%dvob n%dvoa vcvs n%dv11a n%dv11b 1e6 max=2.5 min=-2.5\n',n,n,n,n,n);
end;

% Write each cascaded section
while ((n+2) < (order+1))
n = n+2
'even casc write'

fprintf(fid2,'x%ds1a vdaca n%dv1a vphi1 vphi1b vdd vss switch wid=%e\n',n,n,switcasc(n,1));
fprintf(fid2,'x%ds1b vdacb n%dv1b vphi1 vphi1b vdd vss switch wid=%e\n',n,n,switcasc(n,1));
fprintf(fid2,'x%ds2a 0 n%dv1a vphi2 vphi2b vdd vss switch wid=%e\n',n,n,switcasc(n,2));
fprintf(fid2,'x%ds2b 0 n%dv1b vphi2 vphi2b vdd vss switch wid=%e\n',n,n,switcasc(n,2));

fprintf(fid2,'x%ds3a vina n%dv2a vphi1 vphi1b vdd vss switch wid=%e\n',n,n,switcasc(n,3));
fprintf(fid2,'x%ds3b vinb n%dv2b vphi1 vphi1b vdd vss switch wid=%e\n',n,n,switcasc(n,3));
fprintf(fid2,'x%ds4a 0 n%dv2a vphi2 vphi2b vdd vss switch wid=%e\n',n,n,switcasc(n,4));
fprintf(fid2,'x%ds4b 0 n%dv2b vphi2 vphi2b vdd vss switch wid=%e\n',n,n,switcasc(n,4));

fprintf(fid2,'x%ds5a n%dvob n%dv3a vphi1 vphi1b vdd vss switch wid=%e\n',n,n,n,switcasc(n,5));
fprintf(fid2,'x%ds5b n%dvoa n%dv3b vphi1 vphi1b vdd vss switch wid=%e\n',n,n,n,switcasc(n,5));
fprintf(fid2,'x%ds6a 0 n%dv3a vphi2 vphi2b vdd vss switch wid=%e\n',n,n,switcasc(n,6));
fprintf(fid2,'x%ds6b 0 n%dv3b vphi2 vphi2b vdd vss switch wid=%e\n',n,n,switcasc(n,6));

fprintf(fid2,'x%ds7a n%dvoa n%dv4a vphi1 vphi1b vdd vss switch wid=%e\n',n,n-2,n,switcasc(n,7));
fprintf(fid2,'x%ds7b n%dvob n%dv4b vphi1 vphi1b vdd vss switch wid=%e\n',n,n-2,n,switcasc(n,7));
fprintf(fid2,'x%ds8a 0 n%dv4a vphi2 vphi2b vdd vss switch wid=%e\n',n,n,switcasc(n,8));
```

```
fprintf(fid2,'x%ds8b 0 n%dv4b vphi2 vphi2b vdd vss switch wid=%e\n',n,n,switcasc(n,8));

fprintf(fid2,'x%ds9a 0 n%dv5a vphi2 vphi2b vdd vss switch wid=%e\n',n,n,switcasc(n,9));
fprintf(fid2,'x%ds9b 0 n%dv5b vphi2 vphi2b vdd vss switch wid=%e\n',n,n,switcasc(n,9));
fprintf(fid2,'x%ds10a n%dv5a n%dv6a vphi1 vphi1b vdd vss switch wid=%e\n',n,n,n,switcasc(n,10));
fprintf(fid2,'x%ds10b n%dv5b n%dv6b vphi1 vphi1b vdd vss switch wid=%e\n',n,n,n,switcasc(n,10));

fprintf(fid2,'x%ds11a n%dv7a n%dv8a vphi1 vphi1b vdd vss switch wid=%e\n',n,n,n,switcasc(n,11));
fprintf(fid2,'x%ds11b n%dv7b n%dv8b vphi1 vphi1b vdd vss switch wid=%e\n',n,n,n,switcasc(n,11));
fprintf(fid2,'x%ds12a 0 n%dv8a vphi2 vphi2b vdd vss switch wid=%e\n',n,n,switcasc(n,12));
fprintf(fid2,'x%ds12b 0 n%dv8b vphi2 vphi2b vdd vss switch wid=%e\n',n,n,switcasc(n,12));

fprintf(fid2,'x%ds13a vdaca n%dv10a vphi1 vphi1b vdd vss switch wid=%e\n',n,n,switcasc(n,13));
fprintf(fid2,'x%ds13b vdacb n%dv10b vphi1 vphi1b vdd vss switch wid=%e\n',n,n,switcasc(n,13));
fprintf(fid2,'x%ds14a 0 n%dv10a vphi2 vphi2b vdd vss switch wid=%e\n',n,n,switcasc(n,14));
fprintf(fid2,'x%ds14b 0 n%dv10b vphi2 vphi2b vdd vss switch wid=%e\n',n,n,switcasc(n,14));

fprintf(fid2,'x%ds15a vina n%dv11a vphi1 vphi1b vdd vss switch wid=%e\n',n,n,switcasc(n,15));
fprintf(fid2,'x%ds15b vinb n%dv11b vphi1 vphi1b vdd vss switch wid=%e\n',n,n,switcasc(n,15));
fprintf(fid2,'x%ds16a 0 n%dv11a vphi2 vphi2b vdd vss switch wid=%e\n',n,n,switcasc(n,16));
fprintf(fid2,'x%ds16b 0 n%dv11b vphi2 vphi2b vdd vss switch wid=%e\n',n,n,switcasc(n,16));

fprintf(fid2,'x%ds17a 0 n%dv9a vphi1 vphi1b vdd vss switch wid=%e\n',n,n,switcasc(n,17));
fprintf(fid2,'x%ds17b 0 n%dv9b vphi1 vphi1b vdd vss switch wid=%e\n',n,n,switcasc(n,17));
fprintf(fid2,'x%ds18a n%dv9a n%dv12a vphi2 vphi2b vdd vss switch wid=%e\n',n,n,n,switcasc(n,18));
fprintf(fid2,'x%ds18b n%dv9b n%dv12b vphi2 vphi2b vdd vss switch wid=%e\n',n,n,n,switcasc(n,18));

fprintf(fid2,'c1n%da n%dv6a n%dv7a %e\n',n,n,n,cvalcasc(n,1));
fprintf(fid2,'c1n%db n%dv6b n%dv7b %e\n',n,n,n,cvalcasc(n,1));
fprintf(fid2,'c2n%da n%dv1%c n%dv5b %e\n',n,n,cvalcascsgna(n,2),n,cvalcasc(n,2));
fprintf(fid2,'c2n%db n%dv1%c n%dv5a %e\n',n,n,cvalcascsgnb(n,2),n,cvalcasc(n,2));
fprintf(fid2,'c3n%da n%dv4a n%dv5b %e\n',n,n,n,cvalcasc(n,3));
fprintf(fid2,'c3n%db n%dv4b n%dv5a %e\n',n,n,n,cvalcasc(n,3));
fprintf(fid2,'c4n%da n%dv2%c n%dv5b %e\n',n,n,cvalcascsgna(n,4),n,cvalcasc(n,4));
fprintf(fid2,'c4n%db n%dv2%c n%dv5a %e\n',n,n,cvalcascsgnb(n,4),n,cvalcasc(n,4));
fprintf(fid2,'c5n%da n%dv3a n%dv5b %e\n',n,n,n,cvalcasc(n,5));
fprintf(fid2,'c5n%db n%dv3b n%dv5a %e\n',n,n,n,cvalcasc(n,5));
fprintf(fid2,'c6n%da n%dv12a n%dvoa %e\n',n,n,n,cvalcasc(n,6));
fprintf(fid2,'c6n%db n%dv12b n%dvob %e\n',n,n,n,cvalcasc(n,6));
fprintf(fid2,'c7n%da n%dv8a n%dv9a %e\n',n,n,n,cvalcasc(n,7));
fprintf(fid2,'c7n%db n%dv8b n%dv9b %e\n',n,n,n,cvalcasc(n,7));
fprintf(fid2,'c8n%da n%dv10%c n%dv9a %e\n',n,n,cvalcascsgna(n,8),n,cvalcasc(n,8));
fprintf(fid2,'c8n%db n%dv10%c n%dv9b %e\n',n,n,cvalcascsgnb(n,8),n,cvalcasc(n,8));
fprintf(fid2,'c9n%da n%dv11%c n%dv9a %e\n',n,n,cvalcascsgna(n,9),n,cvalcasc(n,9));
fprintf(fid2,'c9n%db n%dv11%c n%dv9b %e\n',n,n,cvalcascsgnb(n,9),n,cvalcasc(n,9));

fprintf(fid2,'xopin%d n%dv6b n%dv6a n%dv7a n%dv7b vdd vss vcmfbin%d vphi1 vphi1b vphi2 vphi2b dopamp%d\n',n,n,n,n,n,n,n-1);
fprintf(fid2,'* e1n%d n%dv7b n%dv7a vcvs n%dv6a n%dv6b 1e6 max=2.5 min=-2.5\n',n,n,n,n,n);
fprintf(fid2,'xop2n%d n%dv12b n%dv12a n%dvoa n%dvob vdd vss vcmfb2n%d vphi1 vphi1b vphi2 vphi2b dopamp%d\n',n,n,n,n,n,n,n);
fprintf(fid2,'* e2n%d n%dvob n%dvoa vcvs n%dv12a n%dv12b 1e6 max=2.5 min=-2.5\n',n,n,n,n,n);
end;

% Write the extra cap and DACs
'end write'
fprintf(fid2,'xcomp n%dvoa n%dvob vouta voutb vdd vss vphi1 comp\n',n,n);
fprintf(fid2,'mdac1a vdaca vouta vdd vdd pfet w=%e l=.5u as=%e ad=%e ps=%e pd=4u\n',dacsize,4e-6*dacsize,2e-6*dacsize,8e-6+dacsize);
fprintf(fid2,'mdac1b vdaca vouta vss vss nfet w=%e l=.5u as=%e ad=%e ps=%e pd=4u\n',dacsize,4e-6*dacsize,2e-6*dacsize,8e-6+dacsize);
fprintf(fid2,'mdac1c vdacb voutb vss vss nfet w=%e l=.5u as=%e ad=%e ps=%e pd=4u\n',dacsize,4e-6*dacsize,2e-6*dacsize,8e-6+dacsize);
fprintf(fid2,'mdac1d vdacb voutb vdd vdd pfet w=%e l=.5u as=%e ad=%e ps=%e pd=4u\n',dacsize,4e-6*dacsize,2e-6*dacsize,8e-6+dacsize);

if (order == 2)
fprintf(fid2,'xends100a 0 nendv1a vphi2 vphi2b vdd vss switch wid=%e\n',switend);
fprintf(fid2,'xends100b 0 nendv1b vphi2 vphi2b vdd vss switch wid=%e\n',switend);
fprintf(fid2,'xends101a vina nendv1a vphi1 vphi1b vdd vss switch wid=%e\n',switend);
fprintf(fid2,'xends101b vinb nendv1b vphi1 vphi1b vdd vss switch wid=%e\n',switend);
fprintf(fid2,'c1n%denda nendv1b n%dv11a %e\n',n,n,cvalend);
fprintf(fid2,'c1n%dendb nendv1a n%dv11b %e\n',n,n,cvalend);
elseif (order == 3)
fprintf(fid2,'xends100a 0 nendv1a vphi2 vphi2b vdd vss switch wid=%e\n',switend);
fprintf(fid2,'xends100b 0 nendv1b vphi2 vphi2b vdd vss switch wid=%e\n',switend);
fprintf(fid2,'xends101a vina nendv1a vphi1 vphi1b vdd vss switch wid=%e\n',switend);
fprintf(fid2,'xends101b vinb nendv1b vphi1 vphi1b vdd vss switch wid=%e\u',switend);
fprintf(fid2,'c1n%denda nendv1b n%dv17a %e\n',n,n,cvalend);
fprintf(fid2,'c1n%dendb nendv1a n%dv17b %e\n',n,n,cvalend);
else
fprintf(fid2,'xends100a 0 nendv1a vphi2 vphi2b vdd vss switch wid=%e\n',switend);
fprintf(fid2,'xends100b 0 nendv1b vphi2 vphi2b vdd vss switch wid=%e\n',switend);
fprintf(fid2,'xends101a vina nendv1a vphi1 vphi1b vdd vss switch wid=%e\n',switend);
fprintf(fid2,'xends101b vinb nendv1b vphi1 vphi1b vdd vss switch wid=%e\n',switend);
fprintf(fid2,'c1n%denda nendv1b n%dv12a %e\n',n,n,cvalend);
fprintf(fid2,'c1n%dendb nendv1a n%dv12b %e\n',n,n,cvalend);
end;

% Add some helpful information
% strin = sprintf('.ends %s\n',name2);
% fprintf(fid2, strin);
fprintf(fid2,'* umax = %e\n',umax);
fprintf(fid2,'* order = %d OSR = %d f0 = %e\n',order,OSR,f0);
status = fclose(fid2);
if status == -1
'Closing was not Successful';
end;
```

```
'Voila!'
cputime-timetoc % Show Time it took
```

A.2 Operational Amplifier Synthesis Script opsyn.m

```
% Matlab Tool for MicroSensor Operational Amplifier Synthesis
% Mark Shane Peng, Started 10/24/97
% Last Updated 01/02/98
% Now implements Fully Differential Opamp
% with higher gain (1st Stage Cascode)

% Program creates a subcircuit file for SPICE - a subcircuit for each stage.

% things that need to be set:  vcmfb, cload_user, filename, x, name

% Default Values
if (exist('vcmfb','var'));
else vcmfb = 0, x = 1,
end;
if (exist('cload_user','var'));
else cload_user = 1,
end;
if (exist('filename','var'));
else filename = 'opamp.sp',
end;
if(exist('cktname','var'));
else cktname = 'opamp',
end;


% Transistor Vector
% m<xtr name> = [<xtr name> <drain> <gate> <source> <body> <N/PMOS> <width> <length>]
% Default units are um
%name = 1;
%wids = 2;
%lens = 3;

% CMOS Transistors
%p = 0;
%n = 1;

% Later Can Change
diffxtrs = [1 4e-6 .5e-6; % 1st Stage
2 4e-6 .5e-6;
3 5e-6 1e-6;
4 5e-6 1e-6;
5 50e-6 1e-6; % 2nd Stage
6 50e-6 1e-6;
7 50e-6 1e-6;
8 50e-6 1e-6;
9 15e-6 1e-6; % Cascode xtrs
10 15e-6 1e-6;
11 5e-6 1e-6;
12 5e-6 1e-6;];
biasxtrs = [1 10e-6 1e-6; % Curr Source Xtr
2 5e-6 1e-6; % 1st Stage load bias
3 10e-6 1e-6;
4 5e-6 1e-6;
5 2e-6 1e-6; % CM Curr Mirr
6 2e-6 1e-6;
7 2e-6 1e-6;
8 3e-6 1e-6;
9 8e-6 1e-6; % Tail current in first stage
10 2e-6 1e-6;
11 5e-6 1e-6; % Cascode Bias
12 10e-6 1e-6;
13 1e-6 2e-6;];

cmfbxtrs = [1 1e-6 .5e-6; % SC CMFB
2 1e-6 .5e-6;
3 1e-6 .5e-6;
4 1e-6 .5e-6;
5 1e-6 .5e-6;
6 1e-6 .5e-6;
7 1e-6 .5e-6;
8 1e-6 .5e-6;];
ccmfb = 50e-15; % F
rcomp = 3e3; % Units are Ohm
ccomp = 70e-15; % Units are F
cload = 100e-15; % Units are F
curr = 15e-6; % Units are in amp

% Start program
'Operational Amplifier Synthesis - constant scaling factors - Differential'
```

```
% Optional runtime stuff
% cload_user = input('Please Input the Load Capacitor (in pF): ')
% filename = input('Output file: ','s')

% Scale by capacitor load: 100f <= cload_user <= 10p
k = cload_user*1e-12/cload;
diffxtrs(:,wids) = diffxtrs(:,wids)*k;
biasxtrs(:,wids) = biasxtrs(:,wids)*k;
cmfbxtrs(:,wids) = cmfbxtrs(:,wids)*k;
rcomp = rcomp/k;
ccomp = ccomp*k;
ccmfb = ccmfb*k;
cload = cload*k;
curr = curr*k;
% end;

% Tack on geometry info.  as=4.0u*w ad=2.0*w ps8.0u+w pd=4.0u -- 0.5um process
for p=1:length(diffxtrs(:,1))
pwid = diffxtrs(p,wids);
odiffxtrs(p,:) = [diffxtrs(p,:) 4e-6*pwid 2e-6*pwid 8e-6+pwid 4e-6];
end;
for p=1:length(biasxtrs(:,1))
pwid = biasxtrs(p,wids);
obiasxtrs(p,:) = [biasxtrs(p,:) 4e-6*pwid 2e-6*pwid 8e-6+pwid 4e-6];
end;
for p=1:length(cmfbxtrs(:,1))
pwid = cmfbxtrs(p,wids);
ocmfbxtrs(p,:) = [cmfbxtrs(p,:) 4e-6*pwid 2e-6*pwid 8e-6+pwid 4e-6];
end;

% Open File
fid = fopen(filename,'w');
if fid == -1
filename = input('File write unsuccessful, Other Output file: ')
end;

% Write it to a SPICE FILE with subckt parameter,
% can then just .include it in analysis one
strin = sprintf('.subckt %s vi+ vi- vo+ vo- vdd vss vcmfb vphi1 vphi1b vphi2 vphi2b\n',cktname);
fprintf(fid, strin);

% Current Source
fprintf(fid, 'il  vb1  vss  %e\n',curr);
fprintf(fid, 'mb%d vb1 vb1 vdd vdd pfet w=%e l=%e as=%e ad=%e ps=%e pd=%e\n', obiasxtrs(1,:));
fprintf(fid, 'mb%d vi1 vb1 vdd vdd pfet w=%e l=%e as=%e ad=%e ps=%e pd=%e\n', obiasxtrs(2,:));
fprintf(fid, 'mb%d vb7 0   vi1 vdd pfet w=%e l=%e as=%e ad=%e ps=%e pd=%e\n', obiasxtrs(3,:));
fprintf(fid, 'mb%d vb7 vb7 vss vss nfet w=%e l=%e as=%e ad=%e ps=%e pd=%e\n', obiasxtrs(4,:));
fprintf(fid, 'mb%d vi2 vb1 vdd vdd pfet w=%e l=%e as=%e ad=%e ps=%e pd=%e\n', obiasxtrs(11,:));
fprintf(fid, 'mb%d vb8 0   vi2 vdd pfet w=%e l=%e as=%e ad=%e ps=%e pd=%e\n', obiasxtrs(12,:));
fprintf(fid, 'mb%d vb8 vb8 vss vss nfet w=%e l=%e as=%e ad=%e ps=%e pd=%e\n', obiasxtrs(13,:));
fprintf(fid, 'mb%d vfb2 vfb2 vdd vdd pfet w=%e l=%e as=%e ad=%e ps=%e pd=%e\n', obiasxtrs(5,:));
fprintf(fid, 'mb%d vfb2 vcmfb vss vss nfet w=%e l=%e as=%e ad=%e ps=%e pd=%e\n', obiasxtrs(6,:));
fprintf(fid, 'mb%d vb6 vb1 vdd vdd pfet w=%e l=%e as=%e ad=%e ps=%e pd=%e\n', obiasxtrs(7,:));
fprintf(fid, 'mb%d vb6 vb6 vss vss nfet w=%e l=%e as=%e ad=%e ps=%e pd=%e\n', obiasxtrs(8,:));
fprintf(fid, 'mb%d vs vb1  vdd vdd pfet w=%e l=%e as=%e ad=%e ps=%e pd=%e\n', obiasxtrs(9,:));
fprintf(fid, 'mb%d vs vfb2 vdd vdd pfet w=%e l=%e as=%e ad=%e ps=%e pd=%e\n', obiasxtrs(10,:));


% Differential Pair Transistors
fprintf(fid, 'm%d vd1 vi- vs vdd pfet w=%e l=%e as=%e ad=%e ps=%e pd=%e\n', odiffxtrs(1,:));
fprintf(fid, 'm%d vd2 vi+ vs vdd pfet w=%e l=%e as=%e ad=%e ps=%e pd=%e\n', odiffxtrs(2,:));
fprintf(fid, 'm%d vd1a vb8 vd1 vdd pfet w=%e l=%e as=%e ad=%e ps=%e pd=%e\n', odiffxtrs(9,:));
fprintf(fid, 'm%d vd2a vb8 vd2 vdd pfet w=%e l=%e as=%e ad=%e ps=%e pd=%e\n', odiffxtrs(10,:));
fprintf(fid, 'm%d vd3 vb7 vss vss nfet w=%e l=%e as=%e ad=%e ps=%e pd=%e\n', odiffxtrs(3,:));
fprintf(fid, 'm%d vd4 vb7 vss vss nfet w=%e l=%e as=%e ad=%e ps=%e pd=%e\n', odiffxtrs(4,:));
fprintf(fid, 'm%d vd1a vb8 vd3 vss nfet w=%e l=%e as=%e ad=%e ps=%e pd=%e\n', odiffxtrs(11,:));
fprintf(fid, 'm%d vd2a vb8 vd4 vss nfet w=%e l=%e as=%e ad=%e ps=%e pd=%e\n', odiffxtrs(12,:));
fprintf(fid, 'm%d vo+ vd2a vss vss nfet w=%e l=%e as=%e ad=%e ps=%e pd=%e\n', odiffxtrs(8,:));
fprintf(fid, 'm%d vo+ vb1 vdd vdd pfet w=%e l=%e as=%e ad=%e ps=%e pd=%e\n', odiffxtrs(8,:));
fprintf(fid, 'm%d vo- vd1a vss vss nfet w=%e l=%e as=%e ad=%e ps=%e pd=%e\n', odiffxtrs(5,:));
fprintf(fid, 'm%d vo- vb1 vdd vdd pfet w=%e l=%e as=%e ad=%e ps=%e pd=%e\n', odiffxtrs(7,:));

% Compensation
fprintf(fid, 'ccomp2 vr2  vo+  %e\n', ccomp);
fprintf(fid, 'rcomp2 vd2a  vr2  %e\n', rcomp);
fprintf(fid, 'ccomp1 vr1  vo-  %e\n', ccomp);
fprintf(fid, 'rcomp1 vd1a  vr1  %e\n', rcomp);

% Common Mode Switch Cap Feedback Network
fprintf(fid, 'ms%d vom1 vphi1 0 vss nfet w=%e l=%e as=%e ad=%e ps=%e pd=%e\n', ocmfbxtrs(1,:));
fprintf(fid, 'ms%d vocm vphi1 vb6 vss nfet w=%e l=%e as=%e ad=%e ps=%e pd=%e\n', ocmfbxtrs(2,:));
fprintf(fid, 'ms%d vop1 vphi1 0 vss nfet w=%e l=%e as=%e ad=%e ps=%e pd=%e\n', ocmfbxtrs(3,:));
fprintf(fid, 'ms%d vom1 vphi2b vo- vdd pfet w=%e l=%e as=%e ad=%e ps=%e pd=%e\n', ocmfbxtrs(4,:));
fprintf(fid, 'ms%d vom1 vphi2 vo- vss nfet w=%e l=%e as=%e ad=%e ps=%e pd=%e\n', ocmfbxtrs(5,:));
fprintf(fid, 'ms%d vocm vphi2 vcmfb vss nfet w=%e l=%e as=%e ad=%e ps=%e pd=%e\n', ocmfbxtrs(6,:));
fprintf(fid, 'ms%d vop1 vphi2b vo+ vdd pfet w=%e l=%e as=%e ad=%e ps=%e pd=%e\n', ocmfbxtrs(7,:));
fprintf(fid, 'ms%d vop1 vphi2 vo+ vss nfet w=%e l=%e as=%e ad=%e ps=%e pd=%e\n', ocmfbxtrs(8,:));
fprintf(fid, 'cm1 vom1 vocm %e\n', ccmfb);
fprintf(fid, 'cm2 vo- vcmfb %e\n', ccmfb);
fprintf(fid, 'cp1 vop1 vocm %e\n', ccmfb);
fprintf(fid, 'cp2 vo+ vcmfb %e\n', ccmfb);
```

```
strin = sprintf('.ends %s\n',cktname);
fprintf(fid, strin);

% Add in vcmfb value for ac,dc analysis if needed
%if(vcmfb(z) ~= 0)
% fprintf(fid, 'vcmfb vcmfb 0 DC=%e\n',vcmfb(z));
%end;
fclose(fid);
```

A.3 Comparator Synthesis Script compsyn.m

```
% Matlab Tool for MicroSensor Comparator Synthesis
% Mark Shane Peng, Started 02/14/98
% Last Updated 02/14/98
% Version 2 - Now retains the scaling factor info in k_comp

% Program creates a subcircuit file for SPICE - a subcircuit for each stage.

% Parameters that need to be set:  cload_comp, filename, name

% Default Values
if (exist('cload_comp','var'));
else cload_comp = 100, % In FemtoFarads
end;
if (exist('filename','var'));
else filename = 'comp.ckt',
end;
if(exist('cktname','var'));
else cktname = 'comp',
end;


% Transistor Vector
% m<xtr name> = [<xtr name> <drain> <gate> <source> <body> <N/PMOS> <width> <length>]
% Default units are um
name = 1;
wids = 2;
lens = 3;

% CMOS Transistors
%p = 0;
%n = 1;

% Default Parameters
% Later Can Change
compxtrs = [1 1e-6 .5e-6; % Regenerative Feedback Comparator
2 1e-6 .5e-6;
3 2.5e-6 .5e-6;
4 2.5e-6 .5e-6;
5 1.5e-6 .5e-6;
6 2.5e-6 .5e-6;
7 2.5e-6 .5e-6;];
rsxtrs = [1 1e-6 .5e-6; % RS Latch
2 .5e-6 .5e-6;
3 1.5e-6 .5e-6;];
cload = 100e-15; % 100 fF
% Start program
'Comparator Synthesis - constant scaling factors - Differential'

% Optional runtime stuff

% Scale by capacitor load: 100f <= cload_user <= 10p
k_comp = cload_comp*1e-15/cload;
if (k_comp < 1) k_comp = 1; end;
compxtrs(:,wids) = compxtrs(:,wids)*k_comp;
rsxtrs(:,wids) = rsxtrs(:,wids)*k_comp;
% cload = cload*k_comp;
% end;

% Tack on geometry info.  as=4.0u*w ad=2.0*w ps8.0u*w pd=4.0u -- 0.5um process
for p=1:length(compxtrs(:,1))
pwid = compxtrs(p,wids);
ocompxtrs(p,:) = [compxtrs(p,:) 4e-6*pwid 2e-6*pwid 8e-6*pwid 4e-6];
end;
for p=1:length(rsxtrs(:,1))
pwid = rsxtrs(p,wids);
orsxtrs(p,:) = [rsxtrs(p,:) 4e-6*pwid 2e-6*pwid 8e-6*pwid 4e-6];
end;

% Open File and Write to it.
fid = fopen(filename,'w');
if fid == -1
filename = input('File write unsuccessful, Other Output file: ')
end;

% Write it to a SPICE FILE with subckt parameter,
```

```
% can then just .include it in analysis one
strin = sprintf('.subckt %s vina vinb voa vob vdd vss vphi1\n',cktname);
fprintf(fid, strin);

% Regenerative Feedback Comparator from Delta Sigma Converters
fprintf(fid, 'mXda vc1 vina vss vss nfet w=%e l=%e as=%e ad=%e ps=%e pd=%e\n', ocomprtrs(1,:));
fprintf(fid, 'mXdb vc2 vinb vss vss nfet w=%e l=%e as=%e ad=%e ps=%e pd=%e\n', ocomprtrs(1,:));
fprintf(fid, 'mXda vc1 vc2 vss vss nfet w=%e l=%e as=%e ad=%e ps=%e pd=%e\n', ocomprtrs(2,:));
fprintf(fid, 'mXdb vc2 vc1 vss vss nfet w=%e l=%e as=%e ad=%e ps=%e pd=%e\n', ocomprtrs(2,:));

fprintf(fid, 'mXda vc3 vc4 vdd vdd pfet w=%e l=%e as=%e ad=%e ps=%e pd=%e\n', ocomprtrs(3,:));
fprintf(fid, 'mXdb vc4 vc3 vdd vdd pfet w=%e l=%e as=%e ad=%e ps=%e pd=%e\n', ocomprtrs(3,:));
fprintf(fid, 'mXda vc3 vphi1 vc1 vss nfet w=%e l=%e as=%e ad=%e ps=%e pd=%e\n', ocomprtrs(4,:));
fprintf(fid, 'mXdb vc4 vphi1 vc2 vss nfet w=%e l=%e as=%e ad=%e ps=%e pd=%e\n', ocomprtrs(4,:));
fprintf(fid, 'mXda vc3 vphi1 vdd vdd pfet w=%e l=%e as=%e ad=%e ps=%e pd=%e\n', ocomprtrs(5,:));
fprintf(fid, 'mXdb vc4 vphi1 vdd vdd pfet w=%e l=%e as=%e ad=%e ps=%e pd=%e\n', ocomprtrs(5,:));

fprintf(fid, 'mXda vos vc3 vdd vdd pfet w=%e l=%e as=%e ad=%e ps=%e pd=%e\n', ocomprtrs(6,:));
fprintf(fid, 'mXda vos vc3 vss vss nfet w=%e l=%e as=%e ad=%e ps=%e pd=%e\n', ocomprtrs(7,:));
fprintf(fid, 'mXdb vor vc4 vdd vdd pfet w=%e l=%e as=%e ad=%e ps=%e pd=%e\n', ocomprtrs(6,:));
fprintf(fid, 'mXdb vor vc4 vss vss nfet w=%e l=%e as=%e ad=%e ps=%e pd=%e\n', ocomprtrs(7,:));

fprintf(fid, 'msrXda vob vos vss vss nfet w=%e l=%e as=%e ad=%e ps=%e pd=%e\n', orsxtrs(1,:));
fprintf(fid, 'msrXdb voa vor vss vss nfet w=%e l=%e as=%e ad=%e ps=%e pd=%e\n', orsxtrs(1,:));
fprintf(fid, 'msrXda vob voa vss vss nfet w=%e l=%e as=%e ad=%e ps=%e pd=%e\n', orsxtrs(2,:));
fprintf(fid, 'msrXdb voa vob vss vss nfet w=%e l=%e as=%e ad=%e ps=%e pd=%e\n', orsxtrs(2,:));
fprintf(fid, 'msrXda vob voa vdd vdd pfet w=%e l=%e as=%e ad=%e ps=%e pd=%e\n', orsxtrs(3,:));
fprintf(fid, 'msrXdb voa vob vdd vdd pfet w=%e l=%e as=%e ad=%e ps=%e pd=%e\n', orsxtrs(3,:));

strin = sprintf('.ends %s\n',cktname);
fprintf(fid, strin);

fclose(fid);
```

Bibliography

[1] R. Harjani. *OASYS: A Framework for Analog Circuit Synthesis.* PhD thesis, Carnegie-Mellon University, 1989.

[2] H. Chang, E. Felt, A. Sangiovanni-Vincentelli, and P. R. Gray. Top-Down, Constraint-Driven Design Methodology Based Generation of a Second Order Sigma-Delta A/D Converter. *Proceedings of IEEE Custom Integrated Circuits Conference*, pages 533–536, May 1995.

[3] J. R. Koza, F. H. Bennett III, D. Andre, and M. A. Keane. Evolution Using Genetic Programming of a Low-Distortion 96 Decibel Operational Amplifier. *Proceedings of the 1997 ACM Symposium on Applied Computing*, pages 207–216, February 1997.

[4] E. S. Ochotta, R. A. Rutenbar, and L. R. Carley. ASTRX/OBLX: Tools for Rapid Synthesis of High-Performance Analog Circuits. *Proceedings of the 31st ACM/IEEE Design Automation Conference*, pages 24–30, 1994.

[5] R. Schreier. The Delta-Sigma Toolbox 5.0, 1997. Toolbox for MATLAB. `ftp://next242.ece.orst.edu/pub/delsig.tar.Z`.

[6] S. R. Norsworthy, R. Schreier, and G. C. Temes, editors. *Delta-Sigma Data Converters: Theory, Design, and Simulation.* IEEE Press, New York, 1997.

[7] J. C. Candy and G. C. Temes, editors. *Oversampling Delta-Sigma Data Converters.* IEEE Press, New York, 1992.

[8] M. W. Hauser. Principles of Oversampling A/D Conversion. *Journal of the Audio Engineering Society*, 39(1/2):3–26, January/February 1991.

[9] B. Zhang. *Delta-Sigma Modulators Employing Continuous-Time Circuits and Mismatch-Shaped DACs.* PhD thesis, Oregon State University, 1996.

[10] K. C.-H. Chao, S. Nadeem, W. L. Lee, and C. G. Sodini. A Higher Order Topology for Interpolative Modulators for Oversampling A/D Converters. *IEEE Transactions on Circuits and Systems*, 37(3):309–318, March 1990.

[11] H. S. Lee. MIT Class 6.775 Lecture Notes. Fall 1997.

[12] R. W. Adams, Jr. P. F. Ferguson, A. Ganesan, A. Volpe S. Vincelette, and R. Libert. Theory and Practical Implementation of a Fifth-Order Sigma-Delta A/D Converter. *Journal of the Audio Engineering Society*, 39(7/8):515–528, July/August 1991.

[13] J. M. Rabaey. *Digital Integrated Circuits.* Prentice Hall, New York, 1996.

[14] MOSIS HP 0.5 μm Process. AMOS14TB BSIM Level 13 SPICE models. `http://www.mosis.org/New/Technical/Testdata/menu-testdata.html`.

[15] A. Yukawa. A CMOS 8-Bit High-Speed A/D Converter IC. *IEEE Journal of Solid-State Circuits*, SC-20(3):775–779, June 1985.

[16] D. A. Johns and K. Martin. *Analog Integrated Circuit Design.* John Wiley and Sons, New York, 1997.

[17] A. N. Karanicolas, K. K. O, J. Y. A. Wang, H.-S. Lee, and R. L. Reif. A High-Frequency Fully Differential BiCMOS Operational Amplifier. *IEEE Journal of Solid-State Circuits*, 26(3):203–208, March 1991.

[18] Ayman U. Shabra. Private Communication with consultation from Richard Schreier.

[19] MATLAB: High-Performance Numeric Computation and Visualization Software, 1997. The Math Works, Inc. Version 5.0.