AN ABSTRACT OF THE THESIS OF

Roger D. Hayward for the degree of Master of Science in Electrical and Computer Engineering presented on April 30, 1992.

Title: Improving a Sampled-Data Circuit Simulator for Delta-Sigma Modulator Design

Abstract Approved:

Richard Schreier

Delta-Sigma Modulator-based Analog-to-Digital converter design is an active area of research. New topologies require extensive simulations to verify their performance. A series of improvements were made to an existing circuit simulation package in order to speed the simulation process for the designer. Various examples of these improvements are presented in typical applications.

Improving a Sampled-Data Circuit Simulator for Delta-Sigma Modulator Design

by

Roger D. Hayward

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Completed April 30, 1992

Commencement June 1992

APPROVED:

Redacted for Privacy

Professor of Electrical and Computer Engineering in charge of major

Redacted for Privacy

Head of department of Electrical and Computer Engineering

Redacted for Privacy

Dean of Graduate School

Date thesis is presented April 30, 1992

Typed by Roger Hayward

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF APPENDIX FIGURES

IMPROVING A SAMPLED-DATA CIRCUIT SIMULATOR FOR DELTA-SIGMA MODULATOR DESIGN

INTRODUCTION

Analog-to-digital converters provide the cornerstone for a variety of modern circuit systems: Instrumentation amplifiers, consumer audio, and telecommunications equipment all require an ADC to provide a digital interface to an analog signal. The art of ADC design has motivated the research community to turn towards a more sophisticated means of obtaining higher precision, and, whenever possible, inherent linearity. The use of Switched-Capacitor topologies[1], and Delta-Sigma Modulators[2], open a new world of design techniques to pursue.

Many advanced topologies used in ADCs require extensive simulation. Many popular programs, such as SPICE[3], are available as general purpose, circuit level, simulation tools. Unfortunately, these tools do not perform well when using switched-circuit topologies. Performance and ease-of-use are both quite poor.

One simulation tool, "gck," developed at University of California, Los Angeles[4], was designed specifically to handle circuits with mixtures of analog and digital circuitry. A clever method for representing switched-circuit topologies was also implemented.

Improvements were made to "gck," in order to provide greater speed, ease of use, and portability. The simulator was ported to the "C" programming language.

Many circuit topologies require the understanding of component matching tolerance. A symbol manager module was added to allow the operator to easily adjust certain components and thereby facilitate these simulations. The concept of a "circuit compiler" greatly improves circuit setup time for systems which require a variety of simulations to be run on an existing topology. Finally, a flexible FFT / DFT module was added to simplify the task of obtaining frequency-domain data from the program.

The addition of these modules allow the program to behave as a stand-alone $\Delta\Sigma$ simulation engine. Using the UNIX™ operating system, the program interfaces easily to existing signal generation and analysis tools.

In this dissertation, the general architecture of the circuit simulator is described. The motivation behind the various improvements to the program are revealed, along with their design strategy. A variety of examples using these additional modules are shown.

BACKGROUND

This chapter describes the aspects associated with ΔΣ/SC circuit simulations, and what makes them unique. The process which "gck" uses in order to simulate a switched capacitor circuit is described.

Much of the research associated with ΔΣ modulator design involves the investigation of various integrator topologies[5]. Typical ΔΣ circuits consist of a variety of capacitors, switches, and an occasional OP AMP. The switches provide a high-level model of the MOS switch, used to vary the topology of the circuit during operation. Two or more switch phases are usually present.

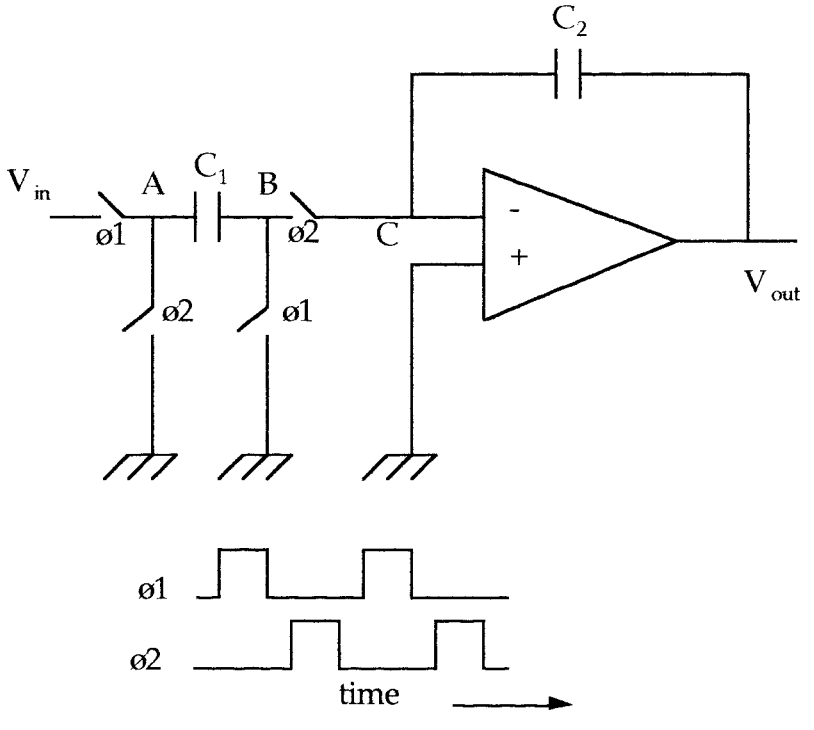

Figure 1. Example SC Integrator Stage

Some ΔΣ modulators utilize multiple-bit ADC and DAC modules, in order to assist in the shaping of the quantization error[6]. It is often convenient to

quickly model an ADC, without having to develop the circuitry which represents it.

The circuit shown in Figure 1 above is a stray-insensitive, non inverting integrator. It emulates an RC network by storing charge in C_1 across constant time intervals. The switches shown are closed during the respective clock cycles. During $\phi 1$, C_1 is charged up to the value at V_{in}. During $\phi 2$, this charge is transferred to C_2. The change in V_{out} becomes

$$\Delta v_{out} = \left(\frac{C_1}{C_2}\right) v_{in}. \tag{1}$$

The z-domain representation of the stage becomes

$$H(z) = \frac{C_1}{C_2} \frac{z^{-1}}{1 - z^{-1}}. \tag{2}$$

Before simulating the stage with analog circuitry, it may be easier to model the above circuit with a simple flow-graph:

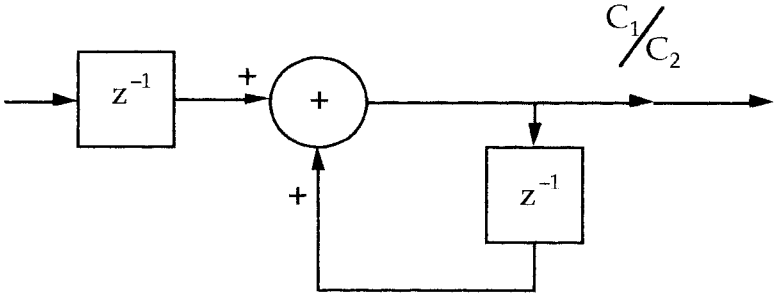

Figure 2. Direct Form I Representation of Integrator Stage

Many circuit packages allow the operator to represent the circuit in an analog manner, as shown in Figure 1. A number of programs are available which can simulate circuits which consist of nothing more than scalars, delay elements, and adders, such as that in Figure 2. However, few systems allow the

operator to easily combine the two circuit representations within the same circuit. Many simulation options are described below.

Coding Your Own Simulation

Many new circuit topologies under investigation are first simulated by preparing a computer program to emulate the circuit topology. Although this certainly can provide results, limitations arise. First, not everyone is a computer programmer. Second, errors associated with the description of the circuit could result in inaccurate results. Finally, cascading this simulation onto other stages of an entire circuit could be cumbersome.

A program by Schreier[2] is currently available for investigating the behavior of a single quantizer $\Delta\Sigma$ modulator with a given noise transfer function. The program is fast, a requirement of any $\Delta\Sigma$ modulator simulation tool, since many simulations involve long input streams. An input sequence of more than 100,000 data points is not unusual. Although this program is very useful in predicting results of the circuit, it doesn't allow the user to represent the circuit's topology or component information.

Mathematical Models for SC Simulations

SADSM[7], "Simulates and Analyzes Delta Sigma Modulators," provides the designer with the ability to investigate properties with a variety of $\Delta\Sigma$ topologies. Although this is a useful tool, new topologies require altering the source code. This does not lend itself well to dealing with custom circuit topologies.

Other custom $\Delta\Sigma$ simulation packages provide the operator with exact behavioral models of switched-capacitors and OP AMPs, but do not allow new

circuit topologies to be exploited[8]. Although these tools have are useful, they do not allow for a "generic" topology to be represented.

Simulating a Generic Circuit Topology

Tools such as SPICE, SABER, SWITCAP, and GCK, allow a generic circuit topology to be described. Each tool comes with its own limitations, however.

Most SC circuits can be simulated with SPICE. However, modeling the generic switch requires controlling voltages to be described. Although the final implementation of the circuit certainly requires driving circuitry for every switch, describing this circuitry requires a great amount of digital circuitry which SPICE would simulate with the same precision as the analog portion of the circuit.

For the example circuit shown above, the circuitry required to drive the two switch phases easily exceeds the size of the circuit under test. SPICE works very well when modeling circuits which rely heavily on device-level parameters. It isn't really the correct tool for the task at hand.

SABER

Analogy, Inc., has a very powerful simulation tool which allows the operator to describe a circuit by developing behavioral models[9].

Although the product is extremely powerful, it is expensive, and reasonably slow. Although SABER simulations are capable of interfacing with many existing tools, this task requires custom interfaces to be written. Even with these "limitations," SABER is an excellent tool for final analysis prior to fabrication of new chips.

SWITCAP

SWITCAP[10] was designed to provide the user the ability to simulate multiple phase, switched-capacitor circuits. The circuit is entered in a manner much like that used by SPICE or GCK. Unfortunately, SWITCAP's syntax checking routines are rather poor. Certain topologies, such as three switches in series (to form a loop), cause internal errors which cause the program to terminate. In addition, limited digital capabilities make representing a multi-bit ADC a cumbersome task.

From this discussion, an argument for finding a simulator that could perform simulations which overcome these problems arise. The UCLA program gck appears to be a very good candidate for handling custom-topology sampled-data systems. Unfortunately, gck is not portable to other platforms due to it's programming language (Pascal), and lacks a strong user interface.

These arguments led to the task of porting gck to the "C" programming language. Additional enhancements performed allow it to become a generic simulation tool for a broad base of sampled-data systems. While still providing the operator the ability to describe the circuit topology, "gckc" fills in the gap between SPICE-level simulations, and $\Delta\Sigma$ modulator simulations which ignore circuit-level topologies completely.

General Control Flow Within GCK

The structure of the gck circuit simulator program involves establishing the circuit topology, setting up a solution, and performing the time-domain simulation. Each of these stages are described below.

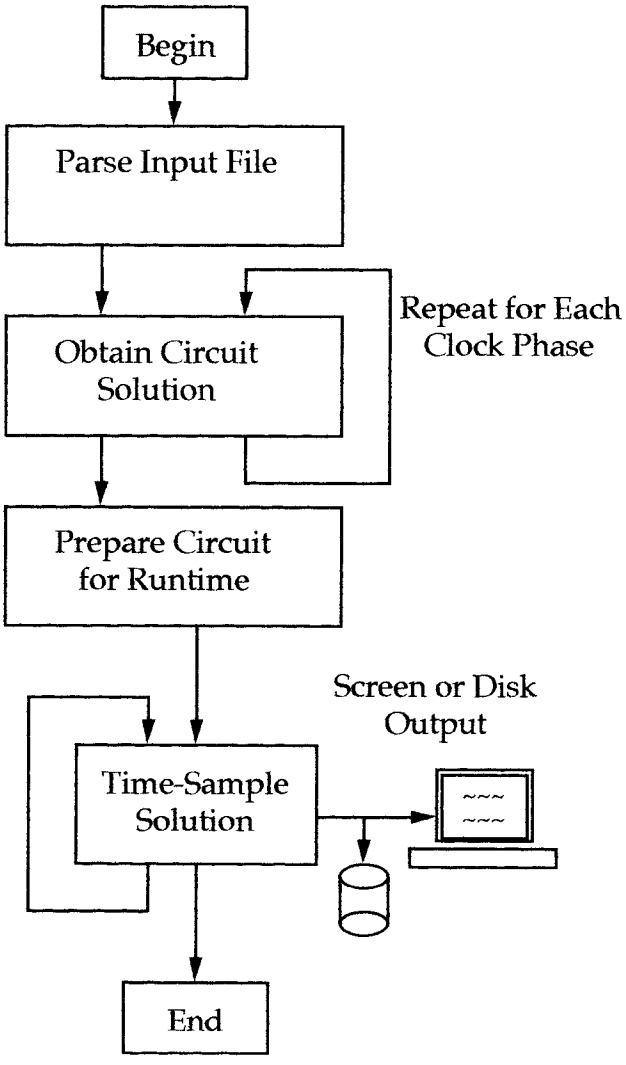

Figure 3. General Flow of GCK Program

Establish Circuit Topology

The circuit topology is described from a standard input file, and additional parameters described on the command line. These both must be parsed, and placed in an appropriate data record.

Initially, memory structures are initialized. This usually involves setting base pointers to NULL. Default control cards are established.

The filename specified when invoking the program is obtained. From there, each line within that file forms a card. Each card describes either a portion of a circuit topology, a simulation control function, or a model description. These are read into individual data structures until the entire parsing routine is complete.

A circuit card usually describes the contents of a branch. For example, the existence of a resistor branch between two nodes is described. A branch number, along with the connecting node numbers, are obtained and stored within a record. This record is attached to a linked list. The linked list forms the complete circuit topology.

If the user specifies a sub-circuit, a new base pointer is established. Circuit cards within the sub-circuit are read and attached onto this new list. When the complete circuit is set up, the base circuit extracts the complete sub-circuit by simply fetching a pointer to the sub-circuit. This is repeated until the complete circuit has been extracted into memory.

Control cards describe details associated with the simulation itself. For example, the .STEP card describes the time interval between circuit simulation

steps. Most of these cards require reading in the value and storing it for later use.

The .SAMPLE and .CLOCK cards describe the pattern which the switches follow during the simulation. The .SAMPLE card allows the user to describe when the output data must be sampled. Since the number of clock phases may be arbitrary, each clock name is stored in a separate record until the circuit solution process is performed.

Obtain a Circuit Solution

Prior to performing the simulation, a variety of structures must be created in order to simulate the circuit in an efficient manner. The two steps performed are setting up structures for all nodes and branches, and then obtaining a solution for the entire circuit.

Once the circuit file is organized by category, a setup function is performed on non-linear circuit devices, such as the quantizer and OP AMP elements. Generic logical elements are established, and all independent sources are initialized.

Obtaining the circuit solution requires establishing the topology for every clock phase, then solving for each of these separately.

In order to accommodate the integrating devices, the capacitor and the inductor, a state-variable matrix is defined. Every capacitor voltage and inductor current is a continuous time function. These are expressed as time derivatives of their respective voltages and currents:

$$I_c = C\frac{dV_c}{dt}, \tag{3}$$

and

$$V_l = L\frac{dI_l}{dt}. \tag{4}$$

The simulation performed within gck considers only specific instants of time. From this, the respective derivatives above may be approximated by a finite change over a fixed time interval:

$$\left.\frac{dV_c}{dt}\right|_{t_{n+1}} = \frac{V_{c_{n+1}} - V_{c_n}}{t_{n+1} - t_n}, \tag{5}$$

and

$$\left.\frac{dI_l}{dt}\right|_{t_{n+1}} = \frac{I_{l_{n+1}} - I_{l_n}}{t_{n+1} - t_n}. \tag{6}$$

These equations are the Backward-Euler Algorithm[11], the procedure used within gck.

Forming a circuit solution involves representing the entire circuit by a set of simultaneous first-order, linear differential equations:

$$\mathbf{M}\dot{X}(t) = \mathbf{N}_0 X(t) + \mathbf{B}_0 e(t), \tag{7}$$

where $\dot{X}(t)$ represents the change in the system of voltages and currents for the circuit. $X(t)$ forms the vector of node voltages and branch currents. $e(t)$ represents the independent sources within the system.

Any non-linear element described within gck is computed at run-time. It appears as an independent source during setup.

For each clock phase, every periodically operated switch either forms an open or short circuit. For the switch with branch number l between nodes j and k, two possible states are described:

$$i_l = 0, \qquad \text{if the switch is open, and} \tag{8}$$

$$v_j - v_k = 0, \qquad \text{if the switch is closed.} \tag{9}$$

The program solves the above equation for X by establishing the three matrices, **M**, $\mathbf{N}_0$, and $\mathbf{B}_0$. The computer's method of establishing these matrices does not involve any investigative network analysis techniques[12]. Instead, each matrix is filled with appropriate values which describe each component within the circuit.

The **M** matrix describes the network topology. This includes branch impedance's and node-to-branch connectivity. $\mathbf{N}_0$ maps the integrating components, capacitors and inductors, and how their respective values contribute to a node voltage or branch current. $\mathbf{B}_0$ contains entries for the

independent sources. Although these matrices form a hybrid, the program never re-orders the indices, since this step is transparent to the user.

The solution to the system is obtained by solving for **N**, the system of node voltages and branch currents:

$$\mathbf{N} = \mathbf{M}^{-1}\mathbf{N}_0 \tag{10}$$

Each independent source contributes to the network by solving for B:

$$\mathbf{B} = \mathbf{M}^{-1}\mathbf{B}_0 \tag{11}$$

Performing this operation is accomplished by the Gauss-Jordan[13] method. The two solutions are performed simultaneously as **M** is inverted.

After obtaining these matrices, the process is repeated for every circuit phase described. Each solution matrix is stored in memory during run-time. Determining the next time values involves performing a pair of matrix multiplications on the selected topology. This method is significantly faster than obtaining a circuit solution during every time-step. The tradeoff is significant time required to set up the system.

Prepare Circuit for Runtime

A series of lists are created which allow the circuit simulation phase to quickly step through the elements of the topology. After the linear elements, the non-linear, digital, and generic logic elements' new values must be computed separately. Prior to runtime, these are placed in individual linked lists.

By establishing these lists prior to runtime, speed of simulation is improved. No redundant type checking is required when computing the new node voltages.

Time-Domain Simulation

Once the circuit topology has been determined, the time-domain simulation involves stepping through the sequence of circuits, computing new node voltages and branch currents for the entire circuit. The backward-Euler process is used to determine these values. Although not as accurate as the classic transient analysis, the approximation is significantly faster.

After determining the new node voltages and branch currents, all non-linear elements are determined and inserted in the appropriate location. The compromise in performing the non-linear analysis this way is that all of these elements require a basic time step delay to be computed. The advantage is that the time to execute each step is fixed, and reasonably short.

Finally, the desired voltages or currents are printed.

This process is repeated until the simulation is complete. The task of switching from one circuit topology to the next is accomplished by fetching a pointer for the particular circuit topology of the current switch phase. Slight memory redundancies are imposed on the system in order for this process to occur, but the tradeoff is increased execution speed.

Additional Requirements for Simulating ΔΣ / SC Systems

Although gck provides the operator with a strong platform for simulating time-domain circuitry, a number of additional features come to mind which could minimize intermediate steps for the operator. These include minimizing setup and execution time, and allowing gck to easily converse with external tools already available.

A number of Fourier Transform algorithms are already available. However, having this feature embedded within the program would eliminate a number of intermediate steps for the operator. FFT / DFT functions, along with an array of windows, should be available.

One weak point in any batch-mode program, like gck, is poor operator feedback. Too often, the operator may key in a circuit configuration, and encounter an error which reveals nothing about the details of the error. Keying in tables, such as the multi-bit quantizer, is also error-prone. In this case, this would not be interpreted as an error to the program.

In order to make gck an easier program to use, many of these problems have been overcome. Although many features could still be added, the new program created for this thesis, "gckc V 2.0," has formed the ground work for a variety of additional features which will significantly aid the ΔΣ designer in interfacing with the program, and completing the task of thoroughly exploring a new topology.

Details on the improvements are included in the next chapter. A variety of examples are provided to illustrate the improvements. A User's Guide to gckc is provided within Appendix A.

PROGRAM DETAILS

This chapter describes the details associated with upgrading the circuit program. The major tasks were completing the conversion from Pascal to C, eliminating circuit size limitations, and creating a symbol manager, circuit compiler, and integrated FFT module.

Porting the Program

Gck was developed in Pascal. Although Pascal provides a very powerful, structured programming environment, the language standard is not very portable. The decision to convert the program from Pascal to ANSI-C[14] was made, in order to allow the program to operate on a variety of computers, both immediately, and in the future.

Fortunately, converting Pascal code to C does not involve the tedious line-by-line tasks that one might envision. A translation program was obtained to perform the bulk of the conversion automatically.

Problems Encountered

After the automated conversion was complete, the next task involved determining what may have been lost in the translation. Two major problems were encountered: Memory allocation and initialization, and the handling of character strings.

The program organizes the circuit within memory by forming a series of linked lists. Each element in the list represents a circuit element, such as a capacitor, a resistor, or an OP AMP. As new branches are encountered, memory is obtained to store these elements. Pascal performs a number of steps when a

new memory block is requested: A section of memory is set aside, initialized (all bits within the structure are set to zero), and a pointer to the base is returned. In C, the same steps are performed, but the task of initializing the record is not performed. The translator program chose to use the `malloc()` function, which does not initialize the memory.

In order to provide compatibility with the Pascal code thought process, a single memory allocation function, `c_malloc()`, was created. This function attempts to allocate the memory, initializes all bits to zero, and returns the pointer. The C function `calloc()` is used within `c_malloc()`. In the event that the allocation fails, an appropriate call to an error handling function arises, and the program terminates gracefully.

Another difference with the C programming language involves the method of dealing with character strings. Since the translator does not have the ability to interpret how strings were manipulated within the program, it should perform everything necessary to ensure that functionality is maintained after the translation is complete. This, unfortunately, did not work.

All C strings terminate with the NULL character. This concept is not understood by Pascal. All string routines within gckc were re-written using NULL terminated strings. For this, heavy use of the `string.h` library was used, speeding up execution, and minimizing the possibility of programming error.

ANSI-C Compliance was handled by enforcing prototypes, and making certain that all library functions called belong to the ANSI library. The development work was performed on the Apple Macintosh computer, using the Symantec THINK C 5.0 compiler, which allows the operator to specify ANSI-C compliance. This enforces the standards imposed by not allowing even a

warning to pass before runtime. Strong function prototypes were added to the program in order to force every aspect of the program to adhere to the structures and styles first created by gck.

Improving the Handling of Cards

The first task of the program is to extract the circuit topology from the input file. The syntax associated with describing the circuit card is straightforward. However, it was noted within the original gck that certain character strings were handled in an inappropriate or inconsistent manner. For example, adding a comment string on the end of a logic element description card would cause errors. Obtaining a particular element from each card involved performing a search task every time a string was requested.

A single card parsing routine was created which extracts every element from a single card at one time. Subsequent calls to extract an element from the card results in a quick lookup. Word counting across each line is now guaranteed, since the function parses the entire line in one pass.

Case Sensitivity

Although the UNIX™ operating system is case-sensitive, gckc is not. This decision was made because case-insensitivity should lead to fewer errors for the operator. Now, the `.print`, `.PRINT`, and `.PrInT` commands are all identical. Since interfaces to the operating system require case sensitivity, gckc stores both an exact image of each card, and a case-insensitive copy. If the decision is made to enforce case sensitivity within the program, for items such as component and node names, the change will be trivial.

Reorganizing the Data Structures

One limitation of the original program involved the size of the circuits which it could handle. A number of structures were fixed to an arbitrary size, and any circuit that was too large for this size could not be solved. Memory structures within the program allocated memory for this worst-case scenario, and manipulated only the portions required.

Frequent requests from users spurred the issue of eliminating this limit. A structure limited only by memory was developed. Two portions of the program had to be re-designed: The network solution phase, and the naming and indexing of nodes. Now, there are no limitations to the size of the circuit that gckc can simulate.

Infinite-Sized Network

Two portions of the program store a series of square arrays. These involve the storing and manipulation of the $\mathbf{M}$, $\mathbf{N_0}$, and $\mathbf{B_0}$ matrices. Since the actual size of the arrays are determined before they are loaded, a pointer can replace the arrays previously created. Prior to loading the arrays for Gaussian elimination, memory is allocated for only the number of variables required. After the Gaussian elimination step is complete, the data is passed to another structure (to hold the solution), and the intermediate memory is returned to the available pool.

Infinite Nodes -- The Node Manager

An array must be allocated to store all of the node voltages during the simulation. In the past, this array was directly accessed by the node number. A limitation of no more than 100 nodes (enforced while parsing the circuit file) guaranteed that this limitation would not be exceeded. However, this put the burden of node management on the operator, something that should be handled by the program instead.

Increasing this array size does not solve the entire problem. If the operator were to reference a node as "999," then the array would have to handle 1000 nodes, even though many nodes below 999 might not be in use. Instead of this, a node manager module was created within the program.

Every node referenced is given a name. From this, every node is assigned a new number, starting with 1 (the reference node is 0). This allows the array size to remain as small as possible, but still allows the operator to specify an arbitrary number of nodes.

The side-effect of the node manager is that nodes no longer need to be specified by numbers. Any string may represent a node name.

Since this node array needs to be filled while the circuit is being built, a structure to handle an arbitrary size needed to be developed. One alternative was to pass through the entire circuit structure twice. The first pass would determine the complete size. The second pass would build the circuit topology. Instead, an assumed size of 100 nodes is chosen as an initial guess. When the node manager exceeds this 100 node limit, the size is increased by 100. This process repeats until the circuit has been completely described.

The Symbol Manager

Many Switched-Capacitor circuit configurations rely heavily on capacitor matching. Slight mismatches between capacitor pairs in an integrator stage, for example, can cause serious degradation in circuit performance. Extensive simulations need to be performed in order to understand the behavior of a circuit when subjected to practical component tolerances.

An additional data structure was created within gckc to allow the operator to easily deal with component variability. The symbol manager holds a list of variable names and their numerical equivalent. When specifying a component, such as a capacitor, C_1, it may be described one of two ways:

```
C1 4 5 1.0P
```

or

```
C1 4 5 C1_value
.SYMBOL C1_value 1.0P
```

Each of these cards describe the device as a capacitor attached between nodes 4 and 5. However, the second method attaches the symbol `C1_value` to the component. The two methods are functionally equivalent.

With the second system, however, multiple devices could share the same symbol. This could prove useful when common devices must be scaled.

The utility of the symbol manager becomes apparent when the simulation is actually invoked. These values may be re-defined on the command-line. For example,

```
gckc foo C1_value=1.01P
```

This would invoke the circuit file foo. All instances where the symbol `C1_value` is referenced will now contain the value 1.01PF, NOT the 1.0PF originally defined.

Performing a series of tests, when adjusting a component, could be very tedious, and certainly time-consuming for most computer systems. However, this system now allows the operator to specify a series of tests to be performed, without having to create a series of input circuit files:

```
gckc foo > foo.dat0
gckc foo C1_value=1.005P > foo.dat1
gckc foo C1_value=1.010P > foo.dat2
gckc foo C1_value=1.015P > foo.dat3
```

Establishing a system to handle symbols involved defining the scope of where they were to be used. Forward referencing allows the operator to reference symbols before they are actually assigned a value. Extensive error checking must be performed in order to guarantee each symbol has been assigned a value prior to runtime.

There is no limit to the number of symbols which can be defined.

Sub-Circuit Overrides

Sub-Circuits allow the operator to describe a circuit topology, such as an integrator, then use the same circuit a number of times within the overall system, such as a multiple-stage filter. If a component within a sub-circuit has a symbol attached to it, and the operator alters this component value, then every instance of the symbol is altered. However, this may not prove useful, especially when investigating the effects associated with errors in just one component. An

additional feature to the symbol manager was created in order to accommodate this.

Consider the sub-circuit:

```
.SUBCKT filter In Out
C1    In Out C1_value=1.0P
R1    Out GND 1K
.ENDSUB
```

And the implementation:

```
X1    1 2 filter
X2    2 3 filter
```

In this case, the symbol `C1_value` is referenced within the sub-circuit `X1` and `X2`. Invoking the circuit with:

```
gckc foo C1_value=1.1P
```

would set both capacitors to 1.1PF. If the operator wanted to set only the second stage to a new component value, the following syntax may be used:

```
gckc foo X1.C1_value=1.1P
```

This allows infinite flexibility for altering component values.

The symbol manager properly handles component adjustments for resistors, capacitors, inductors, all numerical fields within source specifications, and sampling intervals.

Implementing this feature involved creating a running list of which sub-circuit was being parsed currently, such that the sub-circuit names would be

extracted properly. The most specific symbol definition will always override all other definitions for a symbol.

Library

A variety of common cards are required within every circuit file. These include the sampling rate, OP AMP or Quantizer model cards, or common sub-circuits defined by the operator. Rather than requiring the user to paste this information into the input circuit file, the concept of a library file was created.

In order to simplify the understanding of the library file for the common user, the implementation was kept simple. If a common table, sub-circuit, or A/D converter model is used within a variety of circuits, this information may be placed in another file. This file may be referenced within the main circuit file just by referencing its name. The simulation program reads the main circuit file, and branches off to any library file specified.

Since it would only make sense to have libraries reference libraries, this is allowed within the program. However, rather than allowing the redirection level to increase forever, a limit of ten levels was imposed. This wasn't meant to be a limitation on the operator, but a safety mechanism to prevent the operator from creating a circular reference.

The Circuit Compiler

Another task often performed while examining a new $\Delta\Sigma$ modulator topology is performing Signal-to-Noise measurements. This is usually accomplished by subjecting the modulator to a common input signal, but varying the amplitude of the signal across some range. This requires running rather long simulations per amplitude point.

With the exception of the Gaussian elimination stage within gckc, setting up even a modest sized circuit requires very little time. However, the time required to derive the circuit solution increases as a function of both the size of the circuit, and the number of phases within the circuit. Using order notation[15], f(t), the time required, is given by

$$f(t) = O(pn^3), \tag{12}$$

where n is the size of the array (nodes + branches), and p is the number of clock phases specified within the circuit. A circuit with 100 nodes and branches requires at least 1 Million Floating Point operations to solve the circuit topology per phase.

One unique aspect behind the repetitive nature of performing SNR calculations is that the circuit topology doesn't change; only the input signal does. In other words, once the circuit has been setup, there is no need to perform this step repeatedly.

To eliminate this redundancy, the concept of a circuit compiler was established. The "compiled" circuit topology is saved to disk during normal operation. If the operator wishes to use this saved information, a simple directive may be entered on the command line.

A date and time checking routine examines the age of the compiled circuit file, and all files which were used to create it. If any dependent file is newer than the compiled topology, then the circuit setup is performed again automatically. Because re-defining a symbolic value on the command line could alter the circuit topology, this too will cause the circuit solution to be recalculated.

Implementing the Fourier Transform

Many ΔΣ simulations require frequency domain information in order to interpret the system's results. Although a variety of FFT algorithms are already available, an integrated routine minimizes the time required before graphing. Operator error is minimized also.

An additional run-time module was created, to collect a copy of all time-domain data required after the simulation completes. A linked list of only the node voltages (or branch currents) required for the Fourier transform module is created, and data is placed into a series of records.

After the time domain portion of the simulation completes, the linked list is examined. A contiguous list of data points is formed and passed to the FFT routine. After all nodes (or branches) have completed the transform, a print routine prints the data to the console, or to a file.

The Discrete Fourier Transform is defined[16] as:

$$X[k] = \sum_{n=0}^{N-1} x[n] W_N^{nk}, \quad k = 0,1,\ldots,N-1, \tag{13}$$

where the twiddle factor is defined as

$$W_N^{nk} = e^{-j(2\pi/N)kn}. \tag{14}$$

Implementing the FFT involved determining the number of stages, computing the twiddle factors, and computing the index positions properly[17]. In order to conform with ANSI C, floating point numbers, rather than complex, had to be used. In-place computations were performed, in order to minimize memory requirements.

Although the FFT is the preferred choice for its speed, a DFT is performed whenever the length of the simulation is not a perfect power of 2. (The DFT and FFT create the same result; the FFT is an algorithm for the FFT).

The DFT takes significantly more time ($O(n^2)$) than the FFT ($O(n \log_2 n)$). By performing either algorithm, gckc eases the burden for the operator when setting up the simulation[18].

Since the output of in-place computations results in the data being placed in bit-reversed order, the data is indexed at print time. This is significantly faster than re-ordering the data.

Windowing

A variety of windowing functions were added as a convenience for the user. The default window is rectangular (no windowing).

Immediately prior to performing the FFT, the windowing function is called, and a pointer to the time-domain data is passed. A few common window functions are available: Rectangular, Bartlett (Triangular), Hann, Hamming, and Blackman[19].

Figures 4 to 8 show the magnitude of the frequency response for each of these windows, for N=128. The rectangular window has the narrowest main lobe, but the next side-lobe is only about 13 dB below the main peak. Further rejection at higher frequencies is relatively poor. For the remaining windows, the side lobes are reduced significantly. However, the main lobes are much wider.

Rectangular

$$w[n] = \begin{cases} 1, & 0 \le n \le N, \\ 0, & \text{otherwise} \end{cases} \tag{15}$$

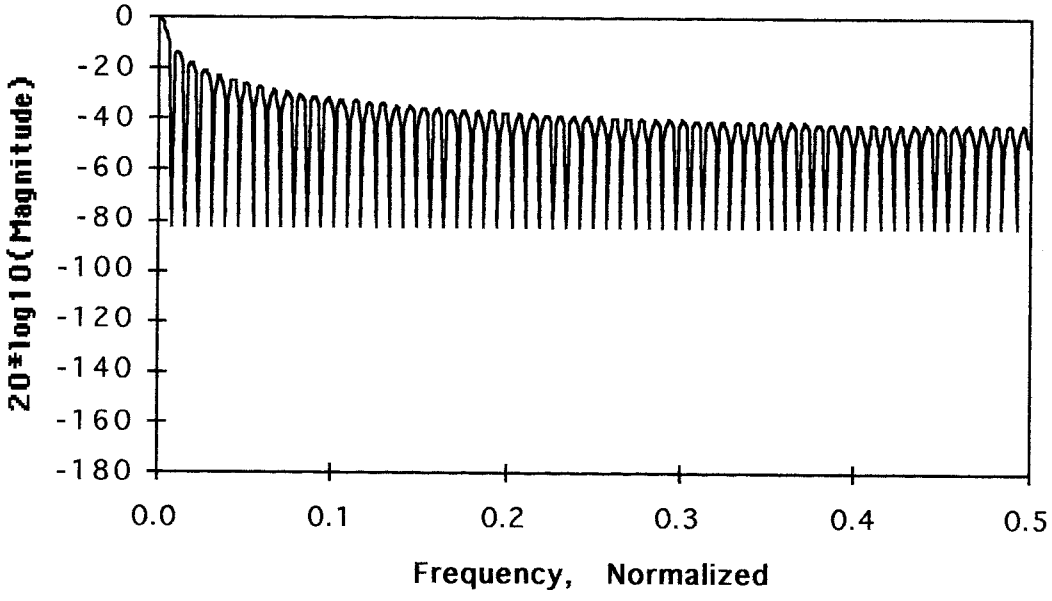

Figure 4. Fourier Transform of Rectangular Windowing

Bartlett (triangular)

$$w[n] = \begin{cases} 2n/N, & 0 \le n \le N/2, \\ 2 - 2n/N, & N/2 < n \le N, \\ 0, & \text{otherwise} \end{cases} \tag{16}$$

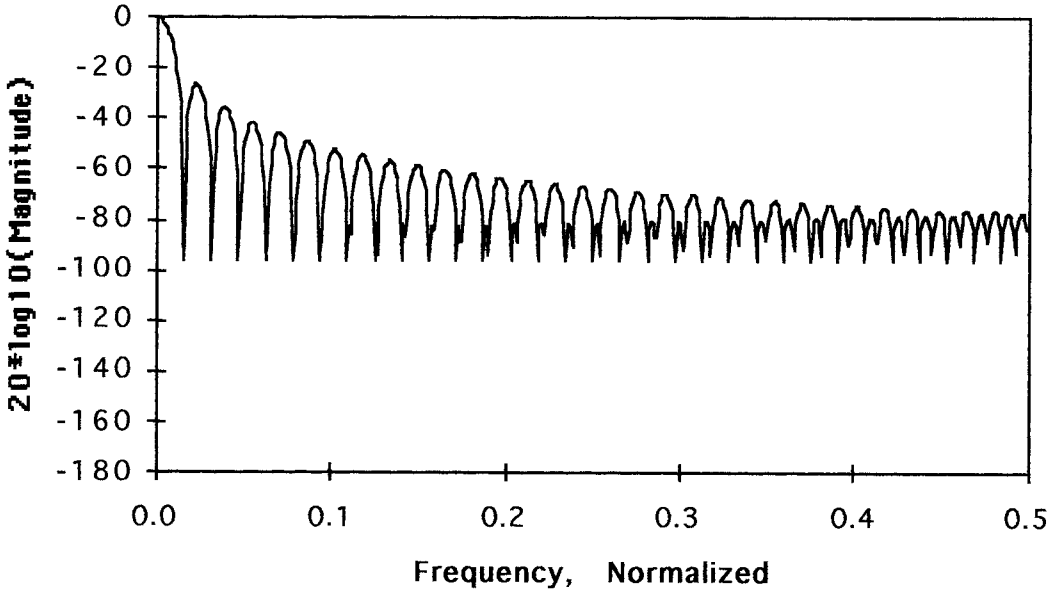

Figure 5. Fourier Transform of Bartlett Windowing

Hann

$$w[n] = \begin{cases} 0.5 - 0.5\cos(2\pi n / N), & 0 \le n \le N, \\ 0, & \text{otherwise} \end{cases} \tag{17}$$

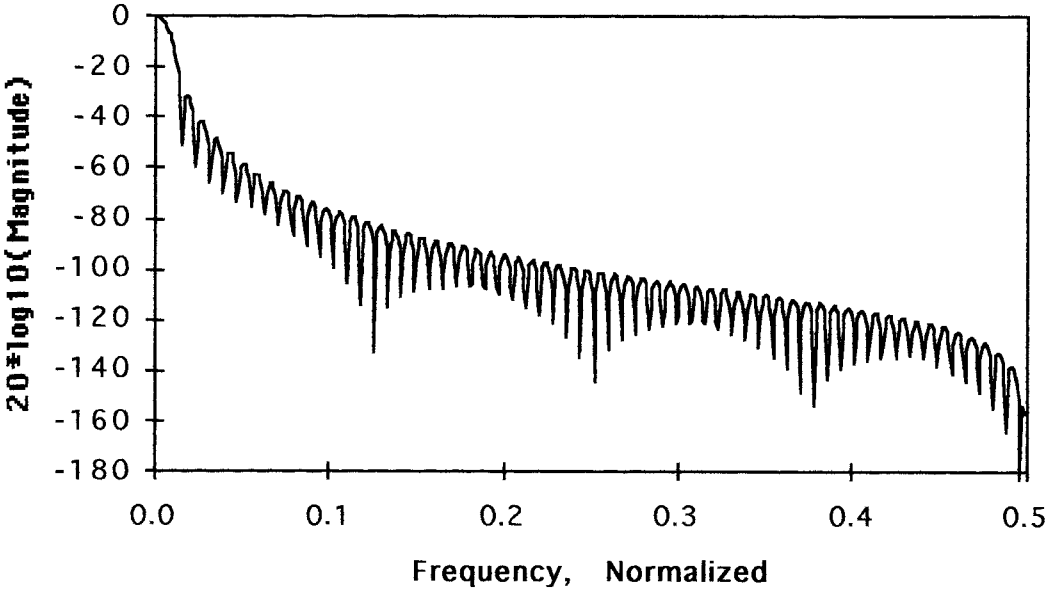

Figure 6. Fourier Transform of Hann Windowing

Hamming

$$w[n] = \begin{cases} 0.54 - 0.46\cos(2\pi n / N), & 0 \le n \le N, \\ 0, & \text{otherwise} \end{cases} \tag{18}$$

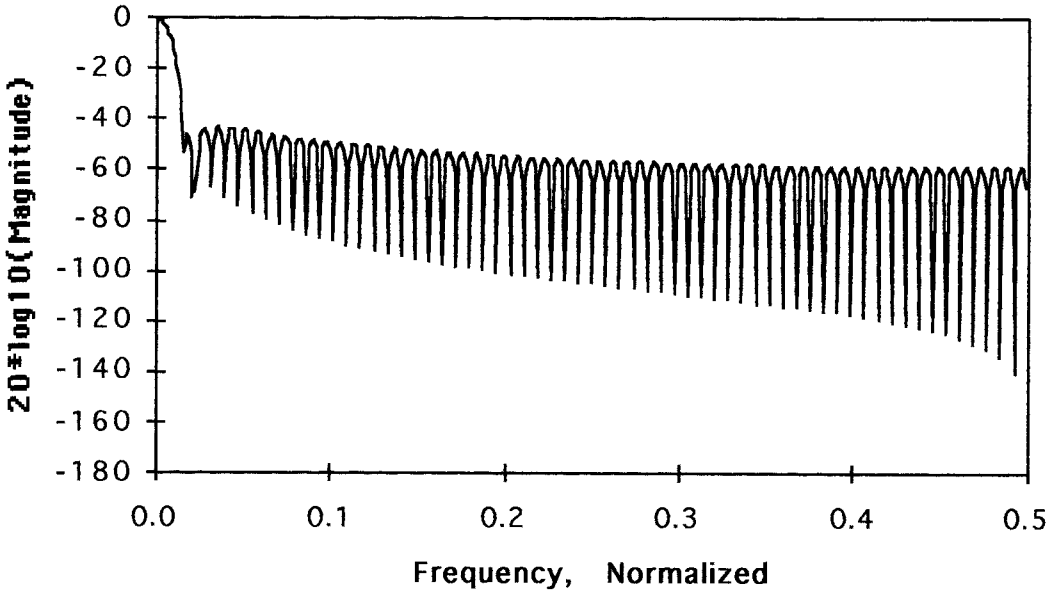

Figure 7. Fourier Transform of Hamming Windowing

Blackman

$$w[n] = \begin{cases} 0.42 - 0.5\cos(2\pi n / N) + 0.08\cos(4\pi n / N), & 0 \le n \le N, \\ 0, & \text{otherwise} \end{cases} \quad (19)$$

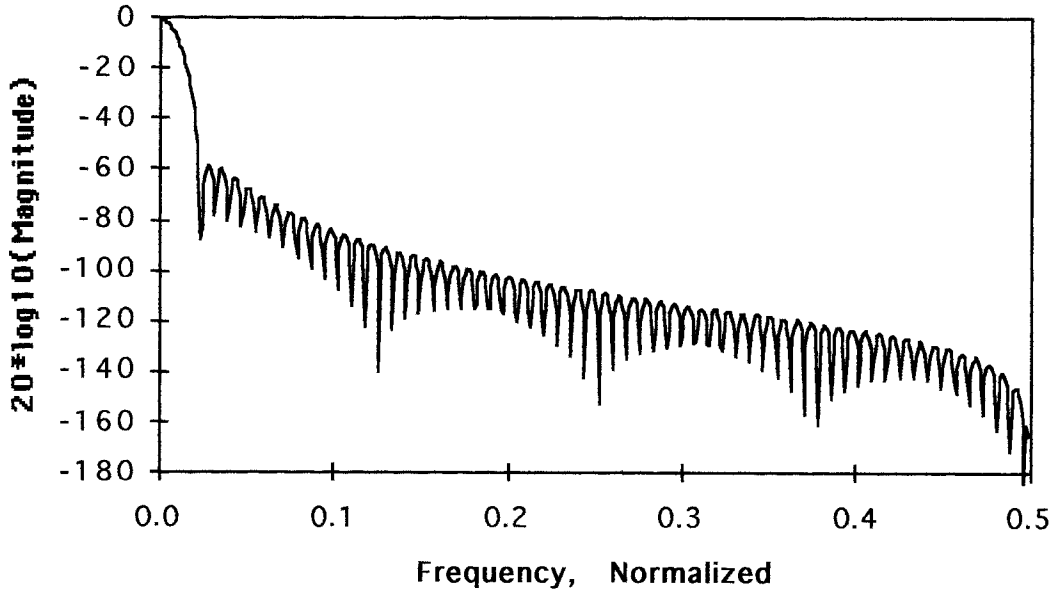

Figure 8. Fourier Transform of Blackman Windowing

EXAMPLE CIRCUITS

A number of circuits were simulated in order to demonstrate the capabilities of gckc. All simulation circuit files are included in Appendix C. Performance measurements were obtained by running the simulations on an Apollo 425t Workstation. The "DFT Time to Complete" performance measurements were measured times to perform the DFT in place of the FFT. These are for comparison purposes only.

First Order ΔΣ Modulator

The classic 1st order Low Pass ΔΣ Modulator was simulated. By attaching the input and output to the standard UNIX I/O ports, the performance of the simulator was compared against C++ models already present.

Note that the quantizer within gckc is not a delay free one. For this reason, the 1-bit quantizer and delay are lumped together in the diagram below:

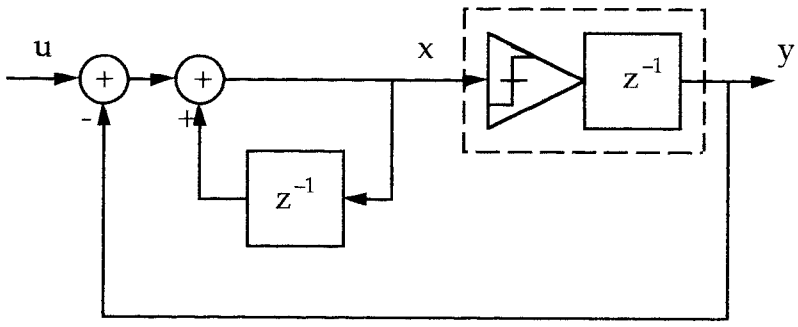

Figure 9. 1st Order Low-Pass ΔΣ Modulator

A sinusoid was placed in the first bin. An 8192 point sequence was run through the circuit. The spectral response (using a Hann window) is shown below. The response, as expected, matches the typical spectra for a low-pass ΔΣ modulator.

Performance parameters are as follows:

Parameter	Time to Perform
Parse and Setup	< 1 Second
Obtain Circuit Solution	< 1 Second
Time Domain Simulation	14 Seconds (1.7ms/Point)
Fourier Transform	< 1 Second
Simulation Length	8192 points

Table 1. Performance Measurements for 1st Order ΔΣ Simulation

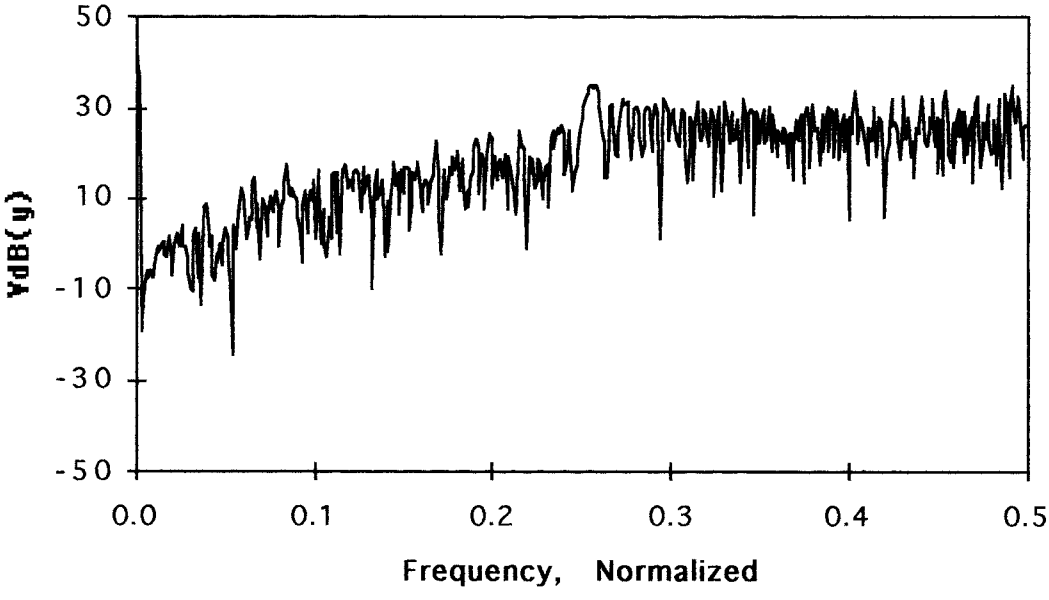

Figure 10. Spectral Response of 1st Order Low-Pass ΔΣ Modulator

Second Order ΔΣ Modulator

Similar results were found with the 2nd Order Low Pass ΔΣ Modulator. Performance measurements were nearly identical.

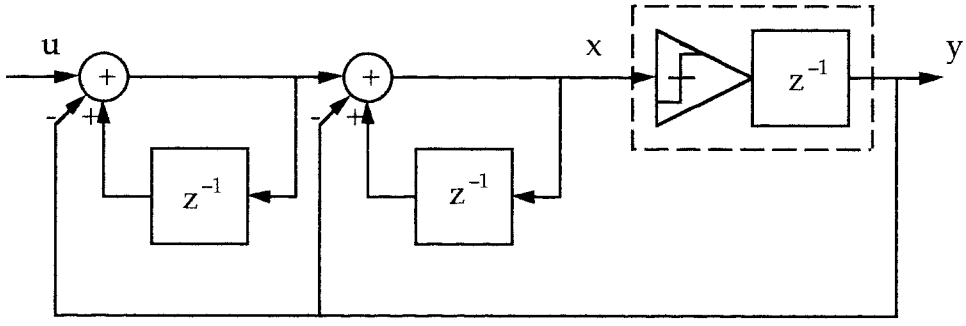

Figure 11. 2nd Order Low-Pass ΔΣ Modulator

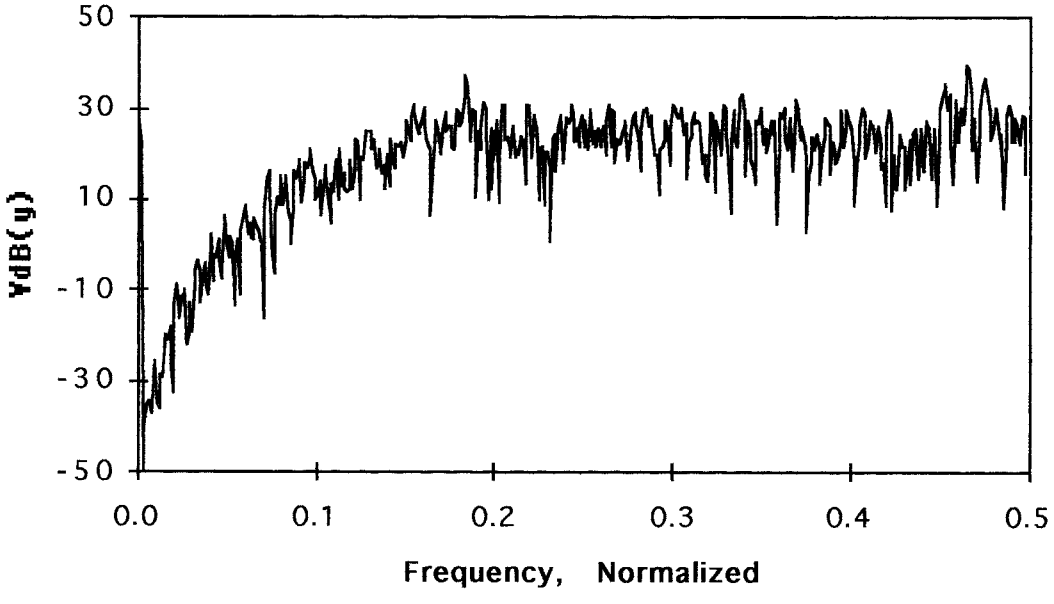

Figure 12. Frequency Response of 2nd Order Low-Pass ΔΣ Modulator

Switched Capacitor Integrator

A differential pseudo 2-path integrator was simulated. This circuit functions as a filter block with the response of

$$H(z) = \frac{1}{\left(1 + z^{-2}\right)}. \qquad (20)$$

Additional cross-switches clocked by $\phi 3$ and $\phi 5$ modulate the input signal by $f_s/4$. The signal is filtered, then modulated back up to the original carrier frequency. The result is a band-pass filter centered at $f_s/4$. For the application which this circuit was intended[5], the minimum gain required is 99dB.

Two simulations were performed. The first shows the response of the circuit due to a tone at 250 KHz. The sampling rate was 1 MHz. For the second simulation, the network was presented a broad-band noise source at -20dB.

Parameter	Time to Perform
Parse and Setup	< 1 Second
Obtain Circuit Solution	16 Seconds
Time Domain Simulation	125 Seconds (15.3ms/Point)
Fourier Transform	< 4 Seconds (4096 Point FFT)
DFT Time to Complete	136 Seconds
Simulation Length	8192 Points

Table 2. Performance for 2-Path Integrator Simulation

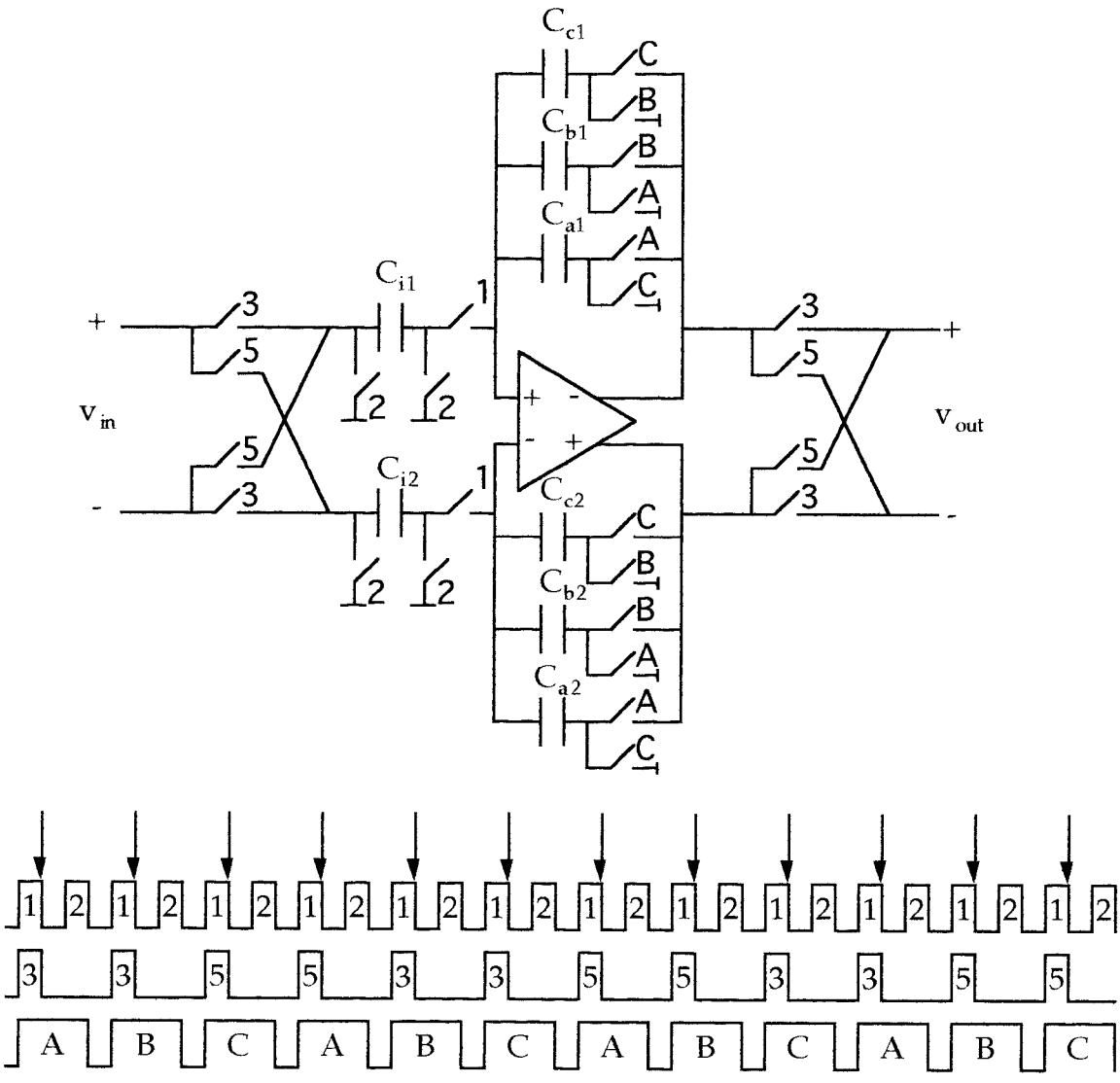

Figure 13. Modulating Integrator with Out-of-Band Noise Peaks

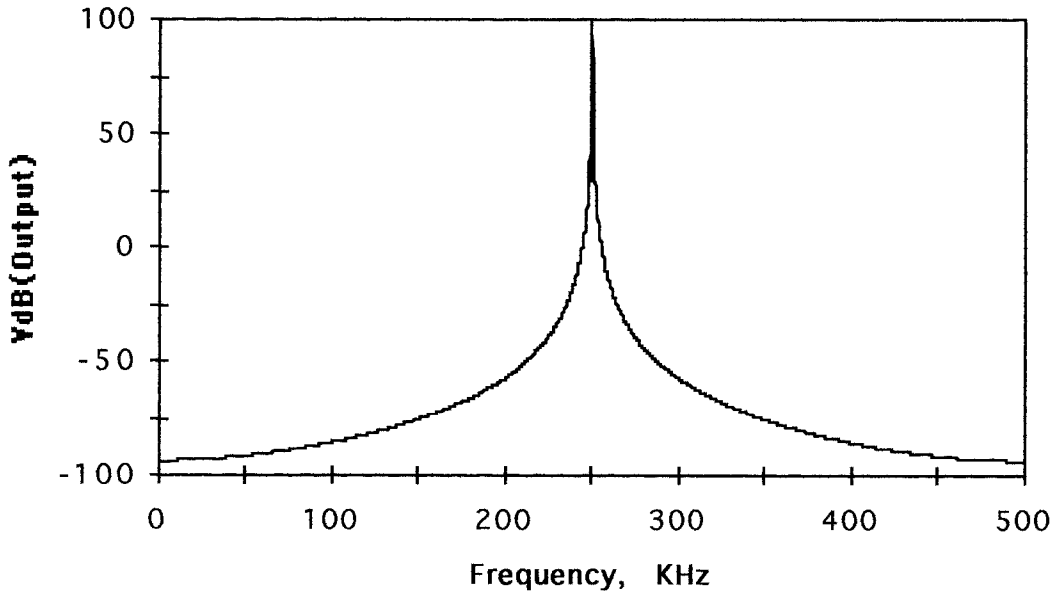

Figure 14. Spectrum of Pseudo 2-Path Integrator

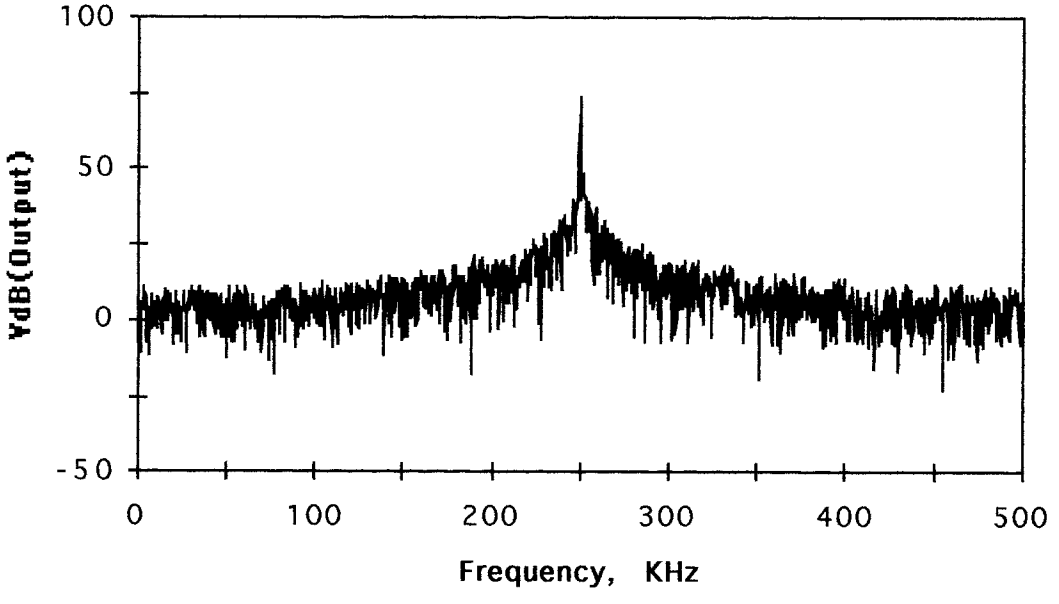

Figure 15. Response of Pseudo 2-Path Integrator to Broad-Band Noise

Three-Stage ΔΣ Modulator

The three stage ΔΣ Modulator shown by Cataltepe[4] was simulated with the new gckc. The length of the simulation changed, along with the signal source, in order to accommodate FFT output. Time-domain results match the original gck program's output.

A -20dB tone at 6835.9375 Hz was placed at V_{in}. Using a sampling interval of 1 μS, this places the tone in the 14th FFT bin of a 2048 point FFT. Figures (17) and (18) show the response of the system due to the single carrier input.

Performance measurements of gckc were taken during the simulation. In addition to the 2048 point run, another simulation with 2049 points was performed, in order to estimate the time to complete the Fourier Transform using the DFT instead. A significant time improvement is clearly evident by using the FFT instead.

Parameter	Time to Perform
Parse and Setup	< 1 Second
Obtain Circuit Solution	55 Seconds
Time Domain Simulation	315 Seconds (76.9ms/Point)
Fourier Transform	< 2 Seconds (2048 Point FFT)
DFT Time to Complete	34 Seconds
Simulation Length	4096 Points

Table 3. Performance Measurements for 3 Stage ΔΣ Modulator Simulation

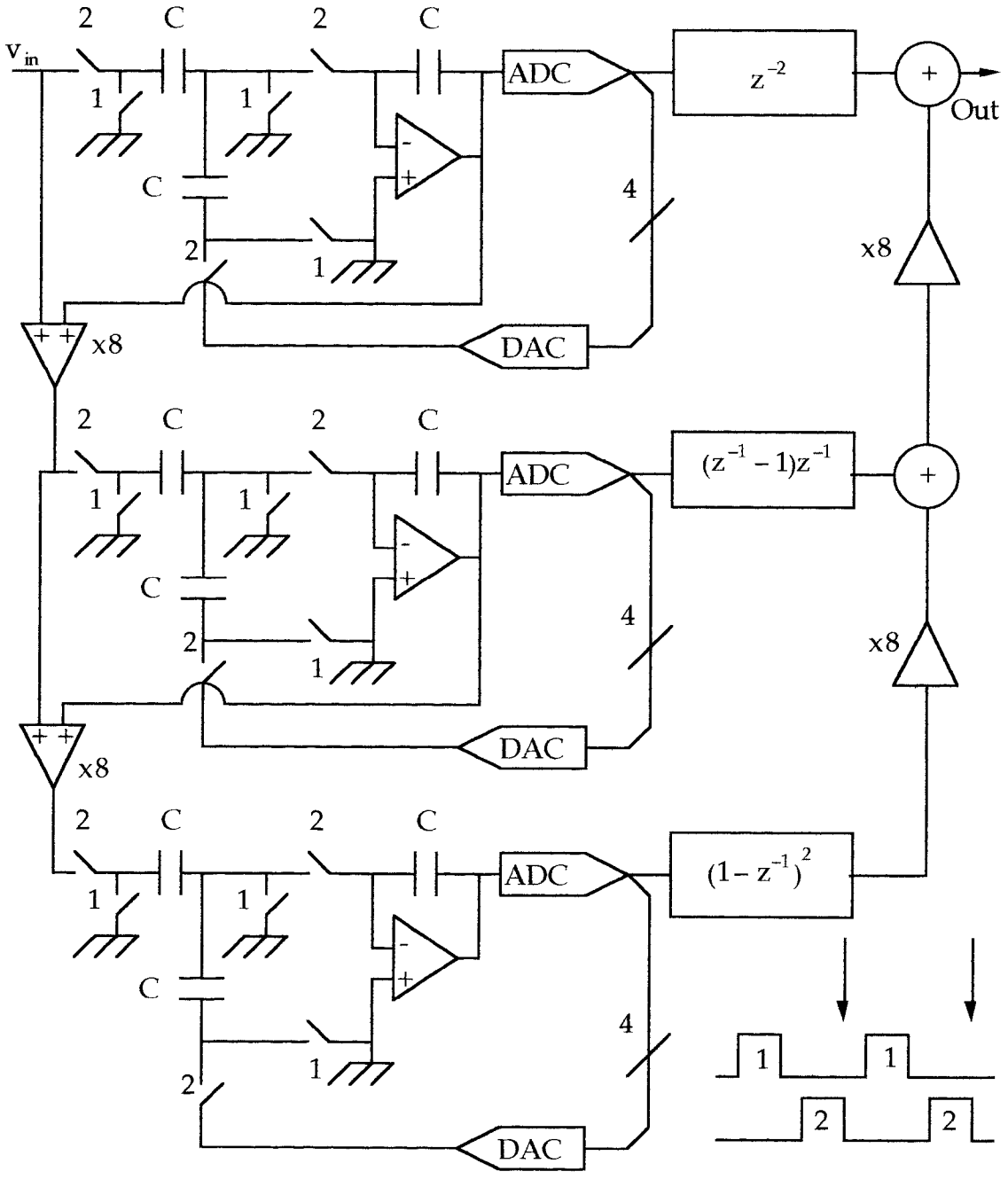

Figure 16. Three-Stage ΔΣ Modulator

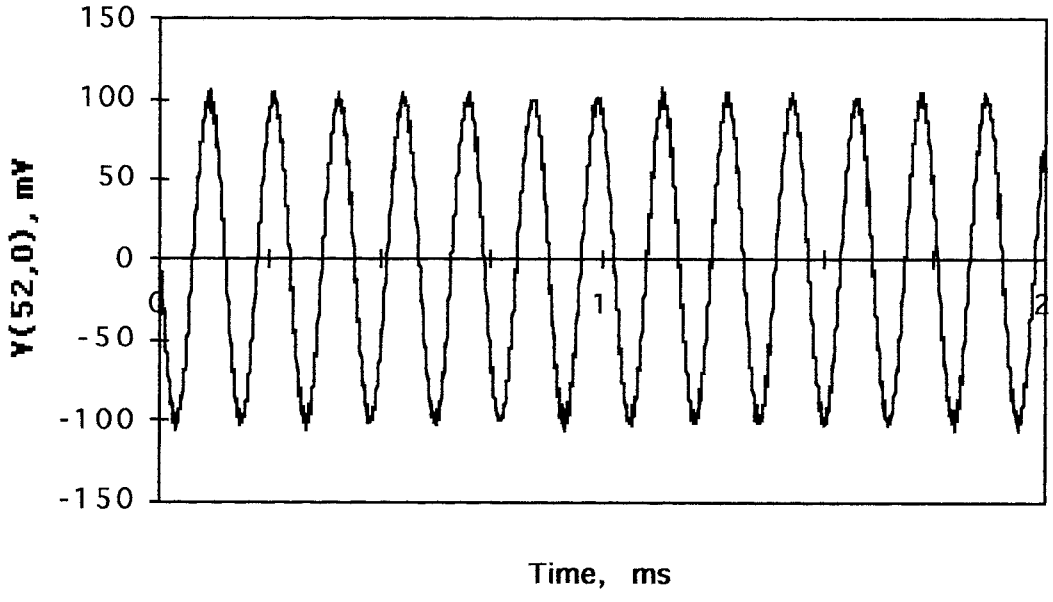

Figure 17. Time Domain Output from Three Stage Modulator

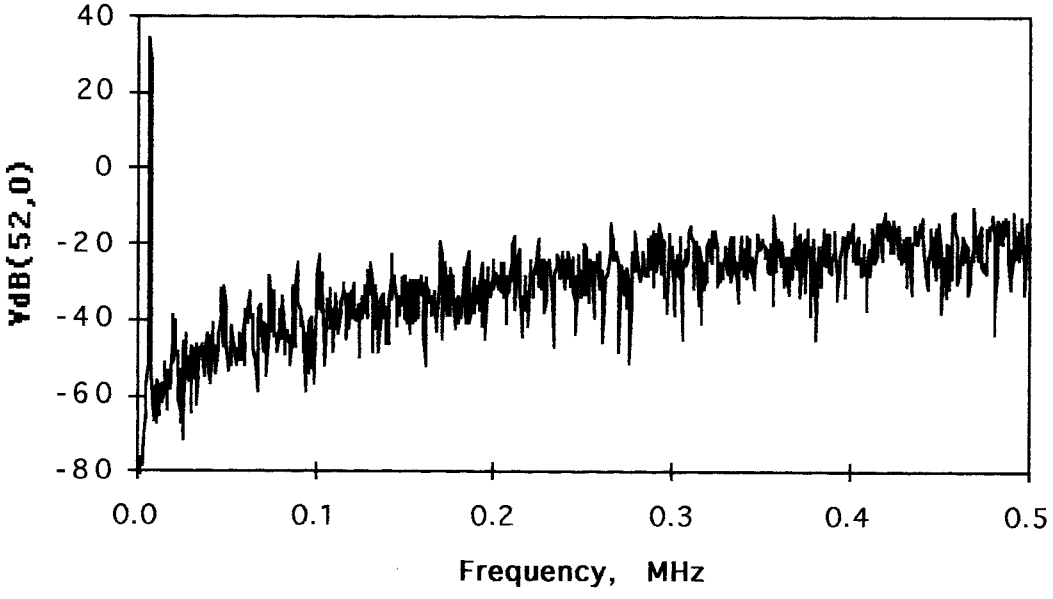

Figure 18. Spectral Response of Three Stage ΔΣ Modulator

CONCLUSION

Tasks Accomplished

The task of simulating sampled-data circuits has been improved through the work performed in this dissertation. Porting the program to C allows gckc to be run on future computer platforms with little or no maintenance. This portability will allow the department to share the tool with other facilities.

The additional features added to gckc provide the operator a cleaner and faster method for interfacing with the simulation kernel. The circuit compiler provides the ability to speed up repeated simulations. The symbolic variable manager replaces the need for repeated adjustments to the input circuit file, which will prove very useful when investigating component tolerance effects. Finally, by removing the circuit size limitations, this program will allow the operator to simulate much larger, and more sophisticated, sampled data circuit systems.

Future Tasks to Overcome

The work performed on gckc does not solve every problem associated with $\Delta\Sigma$ Modulator simulations, nor does it replace simulation tools such as SABER and SPICE. Improvements could still be made which would ease the burden of simulating circuits within this area of research.

Delay-Free Quantizer

A variety of topologies assume the quantizer / ADC block is delay-free. Currently, gckc includes an implicit time step delay through every non-linear element. This allows potential inconsistency issues to be dismissed. Altering gckc to accommodate a delay-free quantizer involves investigating methods for obtaining an iterative solution at each new sample. This would be a significant compromise in performance, however.

Improving Speed Performance

A number of computation-intensive tasks within the program could be improved if Data-Parallel techniques were implemented[15].

The Gauss-Jordan elimination step, along with the FFT, could easily be implemented on a Data-Parallel computer. For the time-domain simulation itself, dividing the circuit solution into one virtual processor per node could be explored. Since little communication between nodes is required, a significant speedup may be possible.

Since the program allows the operator to obtain any number of frequency-domain solutions from one simulation, each single FFT performed could obtain two solutions, by exploiting the fact that both input signals are real.

The User Interface

Any modern program requires a modern interface. Interacting with gckc still involves building a series of circuit files which represents the circuit. Utilizing generic schematic-capture programs would significantly ease the burden for the designer.

The improvements made to this program contribute to the active research going on in $\Delta\Sigma$ modulators. With a faster, more flexible tool, the research engineer now has the ability to explore a new circuit topology by merely describing the circuit and invoking gckc. I hope this will open the door for new techniques to be explored in the months and years to come.

BIBLIOGRAPHY

[1] R. Gregorian, G.C. Temes, Analog MOS Integrated Circuits for Signal Processing, New York, New York, Wiley & Sons, 1986.

[2] R. Schreier, Noise-Shaped Coding, Doctoral Thesis, University of Toronto, Toronto, Canada, 1991

[3] B. Johnson, T. Quarles, SPICE3 Version 3e1 User's Manual, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, April 1991.

[4] T. Cataltepe, Multibit Oversampling Data Converters, Doctoral Thesis, University of California, Los Angeles, 1989.

[5] R. Schreier, "CDADIC Progress Report", Oregon State University, December, 1991.

[6] T. C. Leslie, B. Singh, "An Improved Sigma-Delta Modulator Architecture", Proceedings of the 1990 IEEE International Symposium on Circuits and Systems, vol. 1, pp. 372-375, May 1990.

[7] K. Chan, A Gallium-Arsenide Delta Sigma Modulator for Oversampling Analog to Digital Converters, Doctoral Thesis, University of California, Los Angeles, 1991.

[8] C. M. Wolff, L. R. Carley, "Simulation of Delta Sigma Modulators Using Behavioral Models", Proceedings of the 1990 IEEE International Symposium on Circuits and Systems, vol. 1, pp. 376-379, May 1990.

[9] G. Ruan, "Modeling and Simulation of Sigma-Delta A/D Converters Using a Mixed-Mode Simulator", IEEE International Symposium on Circuits and Systems, May 1992.

[10] S. C. Fang, K. Suyama, User's Manual for SWITCAP Version 1.0, Department of Electrical Engineering and Center for Telecommunications Research, Columbia University, 1990.

[11] R. Spence, J. Burgess, Circuit Analysis By Computer, From Algorithms to Package, Englewood Cliffs, New Jersey, Prentice Hall, 1986.

[12] J. Choma, Electrical Networks, Theory and Analysis, New York, New York, Wiley & Sons, 1985.

[13] W. H. Press, Numerical recipes in C, Cambridge, Cambridge University Press, 1988.

[14] B. W. Kernighan, D. M. Ritchie, The C Programming Language, 2nd Edition, Englewood Cliffs, New Jersey, Prentice Hall, 1988.

[15] P. J. Hatcher, M. J. Quinn, Data-Parallel Programming on MIMD Computers, Cambridge, MA, MIT Press, 1991.

[16] A. V. Oppenheim, R. W. Schaffer, Discrete-Time Signal Processing, Englewood Cliffs, New Jersey, Prentice Hall, 1989.

[17] J. G. Proakis, F. G. Manolakis, Introduction to Digital Signal Processing, New York, New York, Macmillan Publishers, 1988.

[18] M. J. Quinn, Designing Efficient Algorithms for Parallel Computers, 2nd Edition, New York, New York, McGraw-Hill, 1992.

[19] R.W. Ramirez, The FFT, Fundamentals and Concepts, Englewood Cliffs, New Jersey, Prentice Hall, 1985.

APPENDICES

APPENDIX A: User's Guide to GCKC

"GCK, A circuit Simulation Program," was originally developed by Tanju Cataltepe at University of California at Los Angeles, in 1989. It was developed to help the simulation of $\Delta\Sigma$ modulators.

"Why 'GCK?'"

> "If you put the vowels in it becomes 'GICIK.' It was something like a nickname given to me by some close friends because I would disagree with them in almost all philosophical and political subjects. It may be translated as 'disagreeable'. But as 'GCK,' it is meaningless." -- Tanju Cataltepe.

The slight name change was made in order to provide the user the understanding that this is the "new" GCK, not the old. The extra "c" in the program name indicates the user is running the "C-developed" version of the simulator.

Although many improvements have been made to the program, all circuit files created to date are still compatible with this version of gckc. I hope your experience with gckc will be both pleasant and productive.

System Requirements

Although the program was written in ANSI C, source-code compatibility with all computers is not guaranteed. In general, any UNIX™ system with an ANSI C-compatible compiler will build the program without any problems.

The current release (v2.0) will execute properly on Apollo Domain (bsd4.3) systems, and the NeXT computer (mach). All examples within this document were run on the HP-Apollo 425t workstation.

In addition to UNIX, the program is available for the (higher-end) Macintosh computer. Minimum system requirements on the Macintosh are:

Macintosh II series, Powerbook 170, or Quadra
5 MB RAM Minimum
Hard Disk
Floating Point Co-Processor
System 7.0.0 or greater

Although the program has been tested on all (Macintosh) systems listed above, it is recommended that either the UNIX version be used, or the Macintosh simulations be small. Apple's poor virtual memory implementation can cause problems when many applications are loaded simultaneously, or if little physical memory is available. The "-3" directive provides the operator with dialog to show the operator what the Macintosh's memory manager is doing.

Due to size requirements, the program is not supported for DOS systems.

Setting up the Workspace on a UNIX™ System

No special installation is required in order to run the program. The user may wish to add a path to gckc in the `.cshrc` file. Add the path where the program's executable resides.

Example,

```
set path=( ~/bin ~dsptools/bin ~bin/gck )
```

Remember to "`source .cshrc`," or login again, in order to update the path.

To invoke the program, have a gckc source file ready. Example files are available in Appendix C.

Invoking the Program

The program runs only in batch mode, i.e., it is not interactive. Although a few options are available from the command line, the default parameters are probably sufficient.

To start the program, type:

```
gckc input_filename > output_file
```

This will take the file `input_filename`, and read it's circuit configuration. Dialog is provided to the user via the standard error output. In the example shown above, the output stream is redirected to the file `output_file` (although this is not necessary, you will probably want your data to be accumulated in a file, rather than on the terminal screen).

The complete listing of options available are:

```
gckc input_file [compiler_opt] [dialog_opt] [symbols]
```

`input_file`	Any input filename is allowed. Since this is a UNIX™ call, case sensitivity is enforced.
`[compiler_opt]`	Circuit compiler options.
`-C, -c`	(Default) Compile the circuit and run the simulation.
`-Q, -q`	Use the Quick-Compiler, if possible.
`-Z, -z`	Compile the circuit and halt (this causes the simulation itself to NOT run).

`[dialog_opt]`	Options related to level of dialog the program will provide to the operator.
`-0`	Provide NO dialog at all on the standard error output.
`-1`	(Default) Show the GCKC program banner on the standard error output.
`-2`	Provide "highlights" of the circuit simulation, including time-to-complete various tasks.
`-3`	Provide a "painful" amount of dialog during circuit simulation. This is used for debug purposes. For problem circuits, it may be advantageous to invoke this option in order to see what could be taking so long.
`[symbols]`	gckc allows any `component_value` to be associated with a symbol. If the user wishes to update one of these values when invoking the program, he (or she) may do so.

More than one symbol may be specified on the command line, if required.

`symbol_name=new_value.` (One word. NO spaces!)

Although the command line option ordering is ambiguous, the last occurrence of any symbol has precedence.

All of the following examples are valid:

`gckc test` — Compile the circuit `test` and run the simulation.

`gckc -3q test` — Run the simulation `test` from the previously-compiled circuit. Provide dialog messages to stderr.

```
noise 8192 -10 | gckc -0q cdadic_2a | fft -w > cdadic_2a.fft
```

Provide 8192 points of -10dB noise to the standard output. Run the simulation `cdadic_2a` from the previously-compiled circuit. Provide no message output to stderr. The simulation output will be piped to the program `fft` (with windowing). Finally, the result output is placed in the file `cdadic_2a.fft`.

```
gckc -1c test_circuit feedback_cap=1.001p
```

Run the simulation `test_circuit`. For all occurrences of the symbol `feedback_cap` within the file, set this parameter to `1.001 picofarads`.

(Note that specifying any symbol declaration from the command line causes the program to re-compile the circuit configuration automatically. A warning alerts the user of this).

Case Sensitivity

Although UNIX provides a case-sensitive environment, gckc is, in general, case-insensitive. This means that the node "`node_1`" is identical to "`NODE_1`," and "`NoDe_1`." This decision was made in order to increase the user's productivity. My apologies if this causes any confusion.

The exception to this rule is in how gckc deals with filenames. Any instance of a filename, whether it be the input file, library specification, or input signal, needs to be specified in a case-sensitive manner.

Size Limitations

In general, there are no limitations to the size of the circuit which gckc can simulate. There are a couple of details relating to memory issues which should be understood, however. Although any limitation described may appear to be restrictive, these are minor exceptions, and really shouldn't be an issue for even the most advanced user.

Nodes and Branches

Gckc specifies the "number of variables" as the number of branches, plus the number of nodes, within the entire circuit. In the past, this value was limited to 140. No limitation is imposed on the total number of variables with gckc.

Library File Depth

Any number of library files may be specified from within a circuit description.

Because the operator may wish to describe common sub-circuit topologies within a "library" file itself, library files are allowed to reference other library files (just like the #include directive in C). However, no more than 10 levels of indirection may be specified.

Sub-Circuit Depth

Because the operator may wish to describe a sub-circuit within another sub-circuit, a limitation of no more than 10 levels of indirection are enforced within the program. This limitation is enforced in order to prevent circular referencing.

Specifying Circuit Branches

Each element within a circuit, whether it be a resistor, capacitor, switch, or whatever, is described by a branch description line. For example,

```
R1 1 2 1K
```

This describes a resistor "R1," connected from node "1" to node "2," with a value of 1000Ω.

Certain components, like the OP AMP, for example, have more than two connections. The descriptions below indicate how each element's topology is described within the circuit file.

Specifying Nodes

Traditional circuit simulators of this sort require the user to specify every node with a number. Instead of this, gckc interprets every node as a string. Therefore, nodes may be numeric, or any name you choose.

The reference node, "0," may be thought of as "ground." The user may specify the term "GND" in place of "0," in order to make the circuit description easier to read.

Typical node names:

```
1
2
node_1
v
```

In general, there are no strings which cannot be used, even though many words within the simulator are "reserved words."

Examples:

```
R1    1          2          1K
R2    node_2     node_4     1K
```

Note that "2" and node_2 are NOT the same node!

Entering Numeric Values

Most circuit descriptions require the operator to input at least one numeric value, such as the capacitance of the specified capacitor. Standard engineering notation constants may be used to ease the operator in entering values.

The following three lines are all allowed as input within gckc, and are all equivalent:

```
1000.0          1.0e3          1K
```

The following strings will be recognized by gckc and the value will be interpreted accordingly:

Entry	Abbreviation	Scalar
T	Tera	$x * 10^{12}$
G	Giga	$x * 10^{9}$
MEG	Mega	$x * 10^{6}$
K	Kilo	$x * 10^{3}$
M	Milli	$x * 10^{-3}$
U	Micro	$x * 10^{-6}$
N	Nano	$x * 10^{-9}$
P	Pico	$x * 10^{-12}$
F	Fempto	$x * 10^{-15}$
dB	Decibels	$10^{(x/20)}$

Table A-1. SI Unit Conversion Table within gckc

The above terms are case-insensitive. Each term in the entry column must be entered exactly as it is shown (you cannot use "KILO" for 10^3). The "dB" term allows gains, etc., to be entered in decibel form.

Specifying Symbols

At times, the operator may wish to adjust a term, in order to witness the effects of component tolerances, etc. For example, a series of simulations may be performed in order to determine the effects of an error from "matched" capacitors mismatching between 99% and 101% of nominal. This motivated the addition of the concept of the symbol.

Symbol Names

A symbol must be one word (no spaces). The first character of the symbol must be a character, not a number. To avoid conflicts with sub-circuit specifications, the '.' character is not allowed. The '=' character is also not allowed within a symbol name.

Defining and Forward-Referencing

A symbol may be defined one of three ways:

Attached to the component:

Obviously, a symbolic reference is useless unless it is connected to at least one component within the circuit. However, the value associated with the symbol need not be immediately defined.

The following three lines are all valid cards:

```
R1 1 2 R1value
R1 1 2 R1value=1K
R1 1 2 R1value 1K
```

By a control card:

A series of constants could be defined using the `.SYMBOL` card. All three of the following lines perform exactly the same function (associating 1000.0 with the term "R1value"):

```
.SYMBOL    R1value=1K
.SYMBOL    R1value 1K
.DEFINE    R1value 1K
```

Via the Command Line:

Any symbol value specified on the command line is entered last. Note that this will over-ride any value which was specified within the circuit.

```
gckc testckt R1value=1.001K
```

Invoking the circuit with the above command line would set the value of R1value to 1.001KΩ.

Note that when specifying symbols from the command line, the entire symbol-specification must contain no space characters (and the symbol and value must be separated by the '=' character).

Symbols are case-insensitive.

Global Symbols

Any symbol defined once is considered global to the entire circuit, regardless of whether it is within a sub-circuit or not.

Local Symbol Override Within Sub-Circuits

Symbols may be defined within sub-circuits. Sub-circuits may be used a number of times within a complete circuit. The operator may wish to examine the effects of altering the same value within every sub-circuit. However, examining the effects of just one of these components being altered might need to be investigated. In order to accommodate this, a "symbolic value override" may occur locally within sub-circuits.

First, define the sub-circuit:

```
.SUBCKT RC      In    Out
R1    In    Out   R1val=1K
C1    Out   GND   C1val=1.0U
.ENDSUB RC
```

This RC stage may be cascaded to form a complete circuit:

```
V1    Stage1_input    GND        DC    10.0
X1    Stage1_input    2          RC
X2    3               4          RC
X3    4               5          RC
```

Now what if the value for R1 needed to be adjusted for only the resistor within the sub-circuit X2? This could be accomplished by specifying:

```
.SYMBOL    X2.R1val   1.001K
```

or

```
gckc testckt X2.R1val=1.001K
```

This method of adjusting component values can be very powerful in determining tolerance effects of certain circuit topologies.

If two override symbols are defined for the same value, the value used will always be the last value encountered. If a global override and a local symbol declaration are both described, the local symbol always has higher precedence than the global.

```
gckc testckt X2.R1val=1.001K R1val=1.002K
```

The above line would result in the following:

```
X1.R1val   1.002K
X2.R1val   1.001K
X3.R1val   1.002K
```

Interfacing to Existing Tools

Professor Schreier (Oregon State University, `schreier@ece.orst.edu`) has developed a series of tools which allow the designer to quickly simulate a variety of sampled-data circuits. Gckc was designed with the intent to not only work well as a stand-alone program, but interface easily with other tools. Here are a few helpful hints on how to make gckc run as a stand-alone sampled-data simulation tool.

When creating a circuit file, use the "standard input" option for the signal input:

```
V1 node_1 GND STDIN
```

This will cause the program to read the standard input stream for its data. Programs from the Schreier library, such as `noise` and `constant`, can be used for input.

The output stream from gckc is specified by the `.PRINT` control card. The `.NPRINT` card will omit the time tag for every sample output. Therefore, a single output stream can be obtained.

The program prints a variety of dialog messages when it is running. All of these status messages, along with all warnings, are directed to the standard error output (stderr). This output can either be redirected to a junk file, or shut off by invoking the `-0` option.

Putting this all together,

```
V1 node_1 GND STDIN
#circuit described here
...
.NPRINT V(Output)
```

Invoke the above `test` file with

```
noise 8192 -10 | gckc -0 test | fft -w > test.fft
```

Circuit Card Descriptions

This section of the document describes the syntax associated with each type of card within a gckc circuit configuration file.

Circuit Elements

Resistor

```
Ryyyyy <n1> <n2> <value>
```

<n1>	First Node.
<n2>	Second Node.
<value>	Resistance.

Capacitor

```
Cyyyyy <n1> <n2> <value>
```

<n1>	First Node.
<n2>	Second Node.
<value>	Capacitance.

Inductor

```
Lyyyyy <n1> <n2> <value>
```

<n1>	First Node.
<n2>	Second Node.
<value>	Inductance.

Controlled Sources

VCVS

```
Eyyyy <n1> <n2> <nc1> <nc2> <value>
```

<n1>	Positive Output Node.
<n2>	Negative Output Node.
<nc1>	Positive controlling Node.
<nc2>	Negative controlling Node.
<value>	Gain.

VCCS

```
Gyyyy <n1> <n2> <nc1> <nc2> <value>
```

<n1>	Positive Output Node.
<n2>	Negative Output Node.
<nc1>	Positive controlling Node.
<nc2>	Negative controlling Node.
<value>	Gain.

CCVS

```
Hyyyy <n1> <n2> <control_branch> <value>
```

`<n1>`	Positive Output Node.
`<n2>`	Negative Output Node.
`<control_branch>`	Name of the control branch to measure controlling current.
`<value>`	Gain.

CCCS

```
Fyyyy <n1> <n2> <control_branch> <value>
```

`<n1>`	Positive Output Node.
`<n2>`	Negative Output Node.
`<control_branch>`	Name of the control branch to measure controlling current.
`<value>`	Gain.

Switch

```
Syyyy <n1> <n2> <clock_name>
```

`<n1>`	Positive Output Node.
`<n2>`	Negative Output Node.
`<clock_name>`	Name of the controlling clock signal. During phases where the clock holds a '1' value, the switch is closed. A '0' value indicates the clock is open. See the `.CLOCK` and `.SAMPLE` cards for more details.

Independent Sources

There is no limit to the number of independent sources within the program. Each independent source may be generated by any one of the following source specifications.

The source outputs are assumed sampled at the beginning of each basic time step as specified by the .STEP card.

Voltage Source

```
Vyyyy <n1> <n2> <source_spec>
```

<n1>	Positive node.
<n2>	Negative node.
<source_spec>	One of the source options, described below.

Current Source

```
Iyyyy <n1> <n2> <source_spec>
```

<n1>	Positive node.
<n2>	Negative node.
<source_spec>	One of the source options, described below.

Note that positive current flows from <n1> to <n2>.

Source Specifications

For all source specifications, symbols may be used in place of absolute references for all numerical entries. However, these terms are global, and cannot be adjusted with sub-circuit symbol override parameters.

DC

```
DC <amplitude>
```

`<amplitude>`	DC constant value of source.

Sine Wave

```
SIN <amplitude> <frequency> [<delay>]
```

`<amplitude>`	Peak-to-Peak amplitude, a, of SIN wave.
`<frequency>`	Frequency (cycles per second) of SIN wave, f_0.
`<delay>`	The delay (in seconds) of the SIN wave, t_d. If not specified, the default is 0.

$$x(t) = a * \sin(2\pi f_0(t - t_d)) \qquad (21)$$

Comb

`COMB <amplitude> <start_freq> <delta_f> <#_of_f>`

`<amplitude>`	Peak-to-Peak amplitude, *a*, of SIN wave.
`<start_freq>`	Frequency (cycles per second) of primary SIN wave, f_0.
`<delta_f>`	The amount of change in each frequency of the comb, Δf.
`<#_of_f>`	The number of tones within the comb, N.

$$x(t) = a * \sum_{n=0}^{N-1} \sin(2\pi(f_0 + n * \Delta f)t) \qquad (22)$$

Impulse

`IMPULSE <amplitude>`

`<amplitude>`	For t = 0, this amplitude is placed on the source. For all other t, the amplitude is 0.

Pulse

```
PULSE <unpulsed_amp> <pulsed_amp>
      <period> <delay> <rise_time>
      <duration_high> <fall_time>
```

<unpulsed_amp>	Amplitude during "off" periods.
<pulsed_amp>	Amplitude during "on" periods.
<period>	Overall repetition rate of the PULSE.
<delay>	The delay before the PULSE starts to rise.
<rise_time>	The amount of time the PULSE takes to rise from <unpulsed_amp> to <pulsed_amp>.
<duration_high>	The amount of time the PULSE maintains <pulsed_amp> before beginning its fall.
<fall_time>	The amount of time the PULSE takes to fall from <pulsed_amp> to <unpulsed_amp>.

Random

```
RND <amplitude>
```

<amplitude>	Maximum amplitude of the random signal. The source specification takes on the values of [-<amplitude>,<amplitude>].

From a File

`FILE <filename>`

`<filename>`	The name of the file which contains the source signal. This file must contain one value per line. If the end of file is reached before the simulation is complete, additional values of 0.0 are appended.

From the Standard Input

`STDIN`	The source signal is read from the standard input. The standard input must contain one value per line. If the end of the standard input stream (^D) is reached before the simulation is complete, additional values of 0.0 are appended.

Non-Linear Components

Two non-linear components are available within gckc. The generic quantizer allows the operator to describe an A/D or D/A converter with custom quantization levels. The OP AMP behaves similarly to the Voltage-Controlled Voltage Source.

Quantizer (ADC/DAC)

```
Qyyyy <n1> <n2> <nc1> <nc2> <model_name>
```

<n1>	Positive Output Node.
<n2>	Negative Output Node.
<nc1>	Positive controlling Node.
<nc2>	Negative controlling Node.
<model_name>	The name of the Quantizer Model to use.

One basic clock unit delay is present within the quantizer model.

The .MODEL card for the Quantizer is of the form:

```
.MODEL 2_bit
 0.50       0.50
 0.00       0.00
-0.50      -0.50
-999e99    -1.0
.END
```

This example shows a 2-bit A/D converter. Any input greater than 0.5 receives a quantized 0.5 output. Any signal lower than -0.5 Volts receives the low output of -1.0 volts.

In general, the table is of the form:

```
.MODEL
<th[n]>    <out[n]>
...
<th[1]>    <out[1]>
.END
```

The values <th[i]> are the threshold voltages. Note that

<th[i]> > <th[i-1]> for all i > 1.

If the input value is greater than <th[i]> but less than or equal to <th[i+1]>, then the output is <out[i]>.

This table may be generated automatically by using the program gcl. Operating the program is described in Appendix B.

OP AMP

```
Oyyyy <n1> <n2> <nc1> <nc2> <model_name>
```

`<n1>`	Positive Output Node.
`<n2>`	Negative Output Node.
`<nc1>`	Positive controlling Node.
`<nc2>`	Negative controlling Node.
`<model_name>`	The name of the Op Amp Model to use.

The .MODEL for the Op Amp is of the form:

```
.MODEL <model_name>
<gain> [<slew_rate>]
.END
```

`<gain>`	Gain of Amplifier (VCVS gain).
`<slew_rate>`	Maximum Voltage Rate-of-change per unit second.

If the `<slew_rate>` is not listed, the amplifier behaves just like the VCVS.

Digital Components

All digital components begin with the '@' symbol.

Delay

```
@Dyyyy <n_out> <n_in> <delay>
```

<n_out>	Output Node.
<n_in>	Input Node.
<delay>	Number of clock ticks to delay. Note that this delay must be greater than or equal to 1.

Adder

```
@Ayyyy <n_out> <n_in_1> <n_in_2> <gain_1> <gain_2>
```

<n_out>	Output Node.
<n_in_1>	Input Node 1.
<n_in_2>	Input Node 2.
<gain_1>	Gain for Node 1.
<gain_2>	Gain for Node 2.

The output value is evaluated as:

```
<n_out> = <gain_1> * <n_in_1> + <gain_2> * <n_in_2>
```

Although this is not a true digital adder, it helps to build simple signal flow graphs of digital filters when used with the delay element.

Logic Elements

The logic element allows an arbitrary number of nodes to create an arbitrary logic output. This might be useful when simulating control circuitry within an adaptive ΔΣ modulator.

Any logic element created has one basic time step delay. Only one output is available per logic element.

Logic Element

```
$yyyyy <n_out> <n0> ... <nM> <Table> <0 | 1>
     <v_th> <v_lo> <v_hi>
```

<n_out>	Output Node.
<n0>	Input Node 1.
<nM>	Input Node M.
<Table>	Logic Table Name.
<0 | 1>	Inversion Bit.
<v_th>	Threshold Voltage.
<v_lo>	Logic '0' Output Voltage.
<v_hi>	Logic '1' Output Voltage.

The order and number of input nodes must match the table exactly. If '1' ('0') is specified after the table name then the output voltage will be <v_hi> (<v_lo>) when the input matches an entry of the table. Otherwise it will be <v_lo> (<v_hi>).

The values <v_th>, <v_lo>, and <v_hi> may be specified with symbols.

As an example, a CMOS OR and NOR Gate would use the following cards:

```
$OR1  or_gate_out  in_1 in_2 OR_GATE 1 2.5 0.0 5.0
$NOR1 nor_gate_out in_1 in_2 OR_GATE 0 2.5 0.0 5.0
```

Logic Truth Table

The logic table is of the form:

```
.TABLE OR_GATE
01
10
11
.END
```

This is a partial truth table. It should only list the values which will cause an output of <v_hi> (<v_lo>).

Sub-Circuits

Sub-circuits can have any type of components. Other sub-circuits may be specified within sub-circuits. However, all clock signals must be global to the entire circuit. With the exception of the reference node, you cannot access any branch that is outside the sub-circuit. To minimize memory usage, node voltages and branch currents within any sub-circuit cannot be monitored during the simulation.

The sub-circuits are referenced by the following card:

`Xyyyy <n1> <nM> <subckt_name>`

`<n1>`	Attachment from higher level circuit to sub-circuit's first external node.
`<nM>`	Attachment from higher level circuit to sub-circuit's Mth external node.
`<subckt_name>`	The name of the sub-circuit, as listed in the `.SUBCKT` description (below).

The definition of a sub-circuit is of the form:

```
.SUBCKT <subckt_name> <n1> ... <nM>
...
.ENDSUB <subckt_name>
```

Between these control cards, components, sources, controlled sources, etc., may be specified. Local nodes may be created, but they will not be accessible outside of the sub-circuit. The number of nodes (`M`) listed on the `.SUBCKT` card must exactly match the number listed within the `Xyyyy` card above.

Control Cards

A variety of control cards are available within gckc. These provide the program with vital information related to the simulation, the results requested, clock phases, etc.

Library Card

```
.LIB*RARY <library_name>
```

or

```
.INC*LUDE <library_name>
```

`<library_name>` The filename which gckc should read before continuing.

The '*' indicates subsequent characters are optional.

It might be convenient to develop a library of tables for future reference. Instead of having to include everything in one file, these "libraries" can be referenced by their filename. Common circuit topologies, A/D tables, etc., are likely candidates for a library file.

There is nothing unique about a library file. It is of exactly the same form as the main gckc file, and no additional "compilation" is required before using the library file(s). Note also that the gckc "quick compiler" will sense when any library file has been updated (it's time stamp is newer than a compiled circuit file's time stamp). If a file has been updated, gckc will re-compile the circuit.

Note that the four lines below all perform exactly the same operation:

```
.LIB        my_library_file
.LIBRARY    my_library_file
.INC        my_library_file
.INCLUDE    my_library_file
```

Symbol Declarations

Symbols may be defined with the value they represent. They may also be defined on the command line when invoking the program. In addition to these, symbol may be defined by the `.SYMBOL` control card:

```
.SYM*BOL   <symbol_name> [=] <value>
```

or

```
.DEF*INE   <symbol_name> [=] <value>
```

`<symbol_name>`	The variable name used to represent a component's value.
`[=]`	The 'equals' character, optional on this control card.
`<value>`	The numerical value which `<symbol_name>` inherits.
	The '`*`' indicates subsequent characters are optional.

Note that the following cards all perform exactly the same function:

```
.SYM        R1val      1K
.SYMBOL     R1val=1000.0
.DEF        R1val=  1000.0
.DEFINE     r1val   =   1e-3MEG
```

Time Step

The "basic time step" used in the simulation is defined one of two ways. The `.STEP` card defines the basic time step exactly. If the `.PERIOD` card is used instead, the value specified by `<period_value>` is divided by `<clock_phases>` to determine the basic time step.

```
.STEP      <step_value>
```

or

```
.PERIOD <period_value>
```

`<step_value>`	The time period defining a basic time step. This value may be specified by a symbol.
`<period_value>`	The basic time step is the `<period_value>` divided by `<clock_phases>`. This value may be specified by a symbol.

One of these cards MUST be defined within the circuit.

Total Simulation Time

The total simulation time must be defined. The simulation will run until the total time has been exceeded.

```
.TIME <value>
```

`<value>`	Total simulation time. This value may be specified by a symbol.

Output Sampling Instants

The output value(s) requested by the .PRINT and .NPRINT cards will occur only at phases where a sample was requested.

`.SAMPLE <10001000....>`

`<1 | 0>`	A '1' indicates that the output should be sampled during this clock phase. A '0' indicates it should not.

The total number of `<clock_phases>` (the number of '1's or '0's in the list) is limited to 30. The number of `<clock_phases>` specified in the `.SAMPLE` card must match the number of phases specified in the `.CLOCK` cards.

Clock Signals

The .CLOCK cards are used to control the switches throughout the simulation. A '1' indicates that all switches using this clock definition card will be closed during these phases. A '0' indicates the switch is open during these phases.

`.CLOCK <clock_name> <10001000....>`

`<clock_name>`	The name associated with this clocking scheme. This should match the <clock_name> entry within every associated switch.
`<1 | 0>`	A '1' indicates that the output should be sampled during this clock phase. A '0' indicates it should not.

The total number of <clock_phases> (the number of '1's or '0's in the list) is limited to 30. The number of <clock_phases> specified by all .CLOCK cards should match one another. This should also match the number of phases specified in the .SAMPLE card.

Node Specifications

The operator may specify saturation limits on any node within the system.

```
.NODE <node_name> LOW = <lower_sat>
      HIGH = <upper_sat> [SR = <slew_rate>]
```

<node_name>	The Node which this card specifies.
<lower_sat>	The lowest value the node may achieve.
<upper_sat>	The highest value the node may achieve.
[<slew_rate>]	The slew rate for this node, Volts / Second.

Symbols are not supported within any parameters on the .NODE card.

Model Specification

Currently, two types of MODELs are included within gckc. These are the Op AMP and Quantizer model descriptions. For more details on either one of these, please refer to the respective section above.

Printing

Four types of .PRINT cards are available. The number of nodes which the user may print is unlimited.

```
.PRINT     [V(<n1>[,<n2>])] | [I(<branch_name>)] |
     [> <filename>]
```

V	The voltage between two nodes.
<n1>	Positive Output Node.
<n2>	Negative Output Node. If the second node is the reference node, it need not be specified.
I	The current flowing in a branch.
<branch_name>	Name of the branch to measure current.
<filename>	The name of the file to redirect the time-domain output.

The order (voltage and current) is arbitrary.

The two PRINT card options are as follows:

.PRINT	Print the nodes specified. The first column of each line will show the time duration passed.
.NPRINT	Print the nodes specified only.

Requesting FFT/DFT Output

The FFT card within gck invokes either the FFT or DFT. The number of nodes which the user may print is unlimited.

```
.FFT [WINDOW <window_name>] | [V(<n1>[,<n2>])] |
     [I(<branch_name>)] | [> <filename>]
```

`<window_name>`	The name of the window to use when computing the FFT or DFT. (See below).
`V`	The voltage between two nodes.
`<n1>`	Positive Output Node.
`<n2>`	Negative Output Node. If the second node is the reference node, it need not be specified.
`I`	The current flowing in a branch.
`<branch_name>`	Name of the branch to measure current.
`<filename>`	The name of the file to redirect the time-domain output.

The order of these items is arbitrary.

The window types are as follows:

`RECTANGULAR` Perform no windowing. This is the default.

$$w[n] = \begin{cases} 1, & 0 \le n \le N, \\ 0, & \text{otherwise} \end{cases} \tag{23}$$

`BART*LETT` Perform Bartlett (Triangular) Windowing.

$$w[n] = \begin{cases} 2n / N, & 0 \le n \le N / 2, \\ 2 - 2n / N, & N / 2 < n \le N, \\ 0, & \text{otherwise} \end{cases} \tag{24}$$

`TRIA*NGULAR` Perform Triangular Windowing (Shown above).

`HANN` Perform Windowing using Hann's method.

$$w[n] = \begin{cases} 0.5 - 0.5\cos(2\pi n / N), & 0 \le n \le N, \\ 0, & \text{otherwise} \end{cases} \tag{25}$$

`HAMM*ING` Perform Windowing using the Hamming method.

$$w[n] = \begin{cases} 0.54 - 0.46\cos(2\pi n / N), & 0 \le n \le N, \\ 0, & \text{otherwise} \end{cases} \tag{26}$$

`BLAC*KMAN` Perform Windowing using the Blackman method.

$$w[n] = \begin{cases} 0.42 - 0.5\cos(2\pi n / N) + 0.08\cos(4\pi n / N), & 0 \le n \le N, \\ 0, & \text{otherwise} \end{cases} \tag{27}$$

Note that the FFT is performed when the number of samples is a perfect power of two. Otherwise, the DFT is performed. Windowing is unaffected by this, however.

Comment Cards

Comments may be placed arbitrarily throughout gckc. To include a comment, place the '#' or '*' characters at the beginning of a comment line. Comments may be included to the right of any line also.

All of the following examples are valid uses of comments:

```
.SAMPLE 1101          ; I hope this works
*I'm hoping this simulation will lead to a patent!
R1 Albany Corvallis Willamette  *Yes, that's valid.
```

Error Conditions

Certain conditions may cause the program gckc to terminate prematurely. Usually, termination results in a message which gives the operator an indication of why the error occurred.

Errors are categorized by the condition which cause the termination. The categories include errors encountered in parsing either the command line or the circuit file, memory errors, or operating system errors.

Each error condition is described below.

Command Line Parsing

`Command Line requires at least one parameter passed.`

Without at least a filename, invoking the program is useless. Refer to the section "Invoking the Program" for further details.

`A Symbol on the Command Line (symbol_name) is not of the correct form.`

When specifying a symbol on the command line, it must match the following form: `symbol_name=symbol_value`. No spaces are allowed within the entire string.

`The Command Line found more than one input file.`

Only one circuit file may be run each time gckc is invoked.

`Symbols cannot be specified on the command line with the -Z option. Circuit compilation not performed.`

When using the `-Z` option, you are telling the program to read the previously-compiled circuit from disk. Most symbol specifications alter the configuration of this file. Therefore, it would be redundant to specify the `-Z` option, AND specify new symbol values at the same time. Compile the circuit with the `-C` option instead.

Confusing Card Specifications

Print Card Specification is confusing. Print variables are "v" or "i."

Prototype:

```
.PRINT V(1,2) V(Output) I(R1).
```

Your Card:

```
(your card printed).
```

Something within the .PRINT card confused the parsing routine. Most likely, one of the parameters did not begin with the V, I, or > character.

FFT Card Specification is confusing. FFT variables are "v" or "i."

Prototype:

```
.FFT V(1,2) V(Output) I(R1).
```

Your Card:

```
(your card printed).
```

Something within the .FFT card confused the parsing routine. Most likely, one of the parameters did not begin with the V, I, or > character. If windowing is requested, the window name must be specified immediately after the WINDOW keyword.

Card parser unable to resolve card. Source specification for SIN card is confusing.

Prototype:

Vyyyy <node_1> <node_2> SIN <amplitude> <frequency>

[<delay>].

Your Card:

(your card printed).

The SIN card source specification was recognized, but the number of parameters read was incorrect, or of the wrong form.

Card parser unable to resolve card. Source specification for COMB card is confusing.

Prototype:

Vyyyy <node_1> <node_2> COMB <amplitude> <start_f>

<delta_f> <#_of_freq>.

Your Card:

(your card printed).

The COMB card source specification was recognized, but the number of parameters read was incorrect, or of the wrong form.

Card parser unable to resolve card. Source specification for PULSE card is confusing.

Prototype:

Vyyyy <node_1> <node_2> PULSE <unpulsed_value>

<pulsed_value> <period> <delay> <rise_time>

<duration_high> <fall_time>.

Your Card:

(your card printed).

The PULSE card source specification was recognized, but the number of parameters read was incorrect, or of the wrong form.

Card parser unable to resolve card. Digital ADDER card is confusing--check syntax please.

Prototype:

@AYYYY <n_output> <n_input1> <n_input2>

<value1> <value2>.

Your Card:

(your card printed).

The @A card was recognized, but the number of parameters read was incorrect, or of the wrong form.

```
Card parser unable to resolve card.  Digital LOGIC card is
confusing--check syntax please.

Prototype:

     $YYYY <n_output> <n_input> <node_0> ... <node_M>

     <Table> <0|1> <v_th> <v_lo> <v_hi>.

Your Card:

     (your card printed).
```

The number of nodes is arbitrary, but must match the number within the logic `.TABLE`. The parsing routine works its way backwards from the end until it finds the `0` or `1` card. From there, the correct number of nodes may be determined.

Unknown / Incorrect References

`The BRANCH (branch_reference) is unknown.`

This could happen within either a `.PRINT` or `.FFT` card. Please see that all `I(branch_reference)` items match branches previously defined.

`The MODEL (model_name) was not found.`

An OP AMP (O) or QUANTIZER (Q) was referenced, yet the model was never found.

`The CLOCK signal (clock_name) was not found.`

A Clock signal, on a Switch card (S), was reference, but the clock specification was never found (`.CLOCK`).

`The SUBCKT (subckt_name) is not defined.`

The Sub-Circuit was referenced somewhere within the circuit description. However, the circuit name, `subckt_name`, was never found.

`The MODEL (model_name) has already been defined.`

Models may only be defined once within a gckc simulation. The model may have been defined in a library file that was included in the circuit file.

`The MODEL (model_name) is defined within itself. Recursive sub-circuits, obviously, are not allowed.`

Sub-circuits may be defined within sub-circuits. Unfortunately, the model `model_name` referenced itself.

`Unknown source specification.`

The valid source specifications are `DC`, `SIN`, `COMB`, `PULSE`, `IMPULSE`, `FILE` and `STDIN`.

`The quantizer model (model_name) is not of the correct form.`

Each line within the quantizer specification must contain two numbers. No symbols are allowed.

`The Delay of (delay_element_name) must be greater than 0.`

The delay card's delay specified was either less than 1, or not found.

`Unknown digital component.`

A card parsed found a digital component (the '`@`' symbol was found), but the type is unknown.

`The TABLE (table_name) has already been defined.`

Logic tables may only be defined once within a gckc simulation. The table may have been defined in a library file that was included in the circuit file.

`The TABLE (table_name) was not defined.`

A Logic card (`$Y`) could not find the table it referenced. The table does not have to be specified before the logic card, just somewhere within the circuit file (or an included library file).

`The Window Type (type_specified) is not available.`

The current window types available are Rectangular, Triangular, Bartlett (triangular), Hann, Hamming, and Blackman. If no window is specified, the default, Rectangular, is used.

`You have specified too many levels of LIBRARY indirection. (Libraries can reference libraries, but not past 10 levels).`

Check to see that a circular reference does not exist.

`An inconsistency in the number of clock phases has been found. Make certain all .CLOCK and .SAMPLE cards all have the same number of phases.`

There is no requirement which card comes first, `.CLOCK` cards, or the `.SAMPLE` card. All of these cards, however, must specify exactly the same number of phases.

`The NODE (node_name) referenced does not exist.`

The only place a node may be referenced in this manner would be within the .PRINT or .FFT cards. Please see that all `V(node_reference)` items match nodes previously defined.

`The Maximum Sub-Circuit Depth has been exceeded.`

Since it is highly unlikely that more than a few levels of hierarchy may exist, the program has set an arbitrary limit of 10 levels deep. If this error is encountered, it may be due to a circular reference.

`The reference SYMBOL (symbol_name) was never assigned a global value. If you are referencing this SYMBOL locally, assign a dummy value to the symbol at least once.`

Self-Explanatory. If you cannot find the symbol, check all library files within the circuit specification.

File - Related Errors

`A Source Input FILE was not found.`

`(Filename sourcefile_name).`

The filename specified on an independent source card was not found.

`The Main Input FILE was not found.`

`(Filename inputfile_name).`

The filename specified when invoking the program was not found.

`The Library Input FILE was not found.`

`(Filename libraryfile_name).`

Self-Explanatory.

`The Circuit Compiler was unable to open the output file (file_name).`

The Operating System was unable to assign a pointer to the compiled circuit output file. This should never happen.

`The FFT Print Routine was unable to open the output file (file_name).`

The Operating System was unable to assign a pointer to the `.FFT` output file specified. This should never happen.

`The Print Routine was unable to open the output file (file_name).`

The Operating System was unable to assign a pointer to the `.PRINT` output file specified. This should never happen.

`The Quick Compiler couldn't find the compiled circuit file (file_name).`

Although it appears to the simulator that the circuit files are up to date, it was unable to open the compiled circuit file. Try running "`touch`" on the main circuit file, then re-run the simulator with the `-C` option.

Circuit Compilation

`Gaussian Elimination reveals a Singular system.`

`Node: node_name.`

All nodes must have at least one branch attachment during every clock phase. If this error occurs, it most likely means that a node appears to be "floating" during at least one clock cycle. The easiest thing to do is place a capacitor directly to ground at that node. Choose a value that is at least two orders of magnitude smaller than the capacitors surrounding it. This will provide a (mathematical) solution, and will have essentially no contributing effect to your overall circuit.

`Gaussian Elimination reveals a Singular system. Unable to determine node location. The trouble Node may be within a Sub-circuit.`

See the previous error. If this problem persists, take the circuit configuration within the sub-circuit, and place it in the main portion of a circuit file. Resolve all floating nodes, then place in a sub-circuit.

`The Quick Compiler found an inconsistent number of clock phases.`

When reading the previously configured file, the number of clock phases stored within the file does not match the number within the source file. Try running "`touch`" on the main circuit file, then re-run the simulator with the `-C` option.

`The Quick Compiler found an inconsistent number of Variables.`

When reading the previously configured file, the number of variables stored within the file does not match the number within the source file. Try running "`touch`" on the main circuit file, then re-run the simulator with the `-C` option.

Out of Memory

All memory errors share a common problem: The operating system failed to find enough memory to simulate the circuit. Consult your system administrator for assistance.

`Out of Memory! A Memory request when parsing a MODEL failed.`

The MODEL for either an OP AMP or QUANTIZER could not obtain sufficient memory.

`Out of Memory! A Memory request when creating a DELAY element failed.`

Each delay element requires memory be allocated to store the values propagating through it. This routine was unable to obtain sufficient memory.

`Out of Memory! A Memory request when creating additional space for NODES failed.`

An array is created just prior to runtime, such that the voltage at every node may be preserved. The memory routine was unable to allocate this list.

`Out of Memory! A Memory request when creating additional space for the NODE MANAGER failed.`

The node manager initially creates room for 100 nodes. When this is exceeded, it creates room for 100 more nodes. This process is repeated until the circuit has been completely parsed. The memory must be in a contiguous block. Therefore, the error may be due to an operating system limitation which restricts the size of a single block of memory.

`Out of Memory! A Memory request when allocating space for a SYMBOL failed.`

The symbol manager was unable to obtain sufficient memory.

`Out of Memory! A Generic Memory allocation call failed.`

An attempt to allocate memory for a branch, source specification, etc., circuit solution array, etc., failed. This is the most common memory error. Simply put, there is not enough memory on the system to handle the circuit specified.

Internal Errors

These errors should never happen. A series of traps have been added to the program such that errors found will terminate the program, rather than continuing the simulation in error. If any of these errors occur, send email to haywardr@ece.orst.edu. Send the complete circuit file, all library files, input files, etc., and specify the type of computer the program was run on.

```
Internal Circuit Compiler Error!  Gaussian Elimination
Routine is unable to resolve the BRANCH (branch_name).
```

The branch name specified could not be resolved. In other words, the name the program gave a branch was later not understood by the program.

```
Internal Circuit Compiler Error!  A request to allocate
memory of size ZERO occurred.
```

A Memory allocation routine tried to allocate memory of size zero. No calls to the memory allocation routine should perform this.

```
Internal Circuit Compiler Error!  Contact your program
support representative for assistance.  Error Code 114.
```

Send email to your support representative for gckc.

`Internal Circuit Compiler Error! The SYMBOL name requested was blank.`

The symbol manager was handed a blank name. This shouldn't happen, because the card parsing routines cannot extract a "blank" field.

`Internal Circuit Compiler Error! An internal attempt to find a SYMBOL failed.`

The symbol manager could not resolve a symbol name.

`Internal Circuit Compiler Error! The routine which creates a SYMBOL was handed a blank name.`

The symbol manager was told to create a symbol record, but the name was blank.

`Internal Circuit Compiler Error! Contact your program support representative for assistance. Error Code "DEFAULT"`

Send email to your support representative for gckc.

APPENDIX B: GCL, Generic A/D Converter Table Generator

In order to minimize error when using a generic quantizer model, a utility program was written. This program automatically generates standard ADC tables in the form which gck expects.

The program is called gcl, which stands for *generic conversion library* generator. Six parameters are passed to the program:

```
gcl <model_name> <bits> <xmin> <xmax> <ymin> <ymax>
    > <filename>
```

`<model_name>`	The name you wish to use as a model name for the quantizer within gck.
`<bits>`	The number of bits to use in the quantizer.
`<xmin>`	The minimum expected input signal value.
`<xmax>`	The maximum expected output signal value.
`<ymin>`	The minimum value represented by the quantizer.
`<ymax>`	The maximum value represented by the quantizer.
`<filename>`	The name of the library file which will contain this table.

The program divides both the x and y ranges into 2^n elements:

$$\Delta x = \frac{x_{max} - x_{min}}{2^{bits}} \tag{28}$$

$$\Delta y = \frac{y_{max} - y_{min}}{2^{bits}} \tag{29}$$

The program sweeps from (x_{max} - Δx) down to x_{min}, in steps of Δx and Δy, respectively. This forms the .MODEL table for the quantizer within gckc. Although problems associated with monotonicity and companding ADC tables cannot be modeled with this program, it does give the operator the ability to quickly generate a multi-bit table.

The extreme values are defined as:

$$y = \begin{cases} y_{max} - \dfrac{\Delta y}{2}, & x \geq x_{max} \\ y_{min} + \dfrac{\Delta y}{2}, & x \leq x_{min} \end{cases} \qquad (30)$$

A Variety of ADC tables were generated. These allow for an input range from -1.0 to 1.0 Volts, with an identical output range. These examples are usually centered about 0 Volts.

The following command creates a one-bit quantizer model, the cornerstone of ΔΣ Modulators:

```
gcl 1bitq 1 -1.0 1.0 -2.0 2.0 >quantizer.lib
```

ADC Tables are created by

```
gcl nbit n -1.0 1.0 -1.0 1.0 >nbit.lib ,
```

where n is the number of bits to represent. Offset and gain errors may be represented by altering the domain and/or range appropriately.

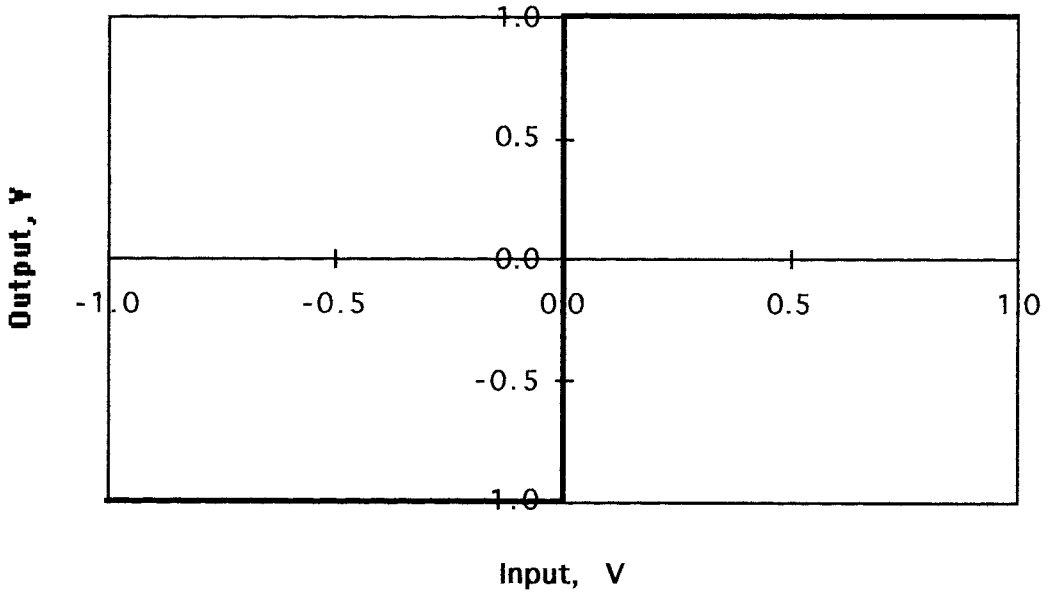

Figure B-1. 1 Bit Quantizer Translation Table

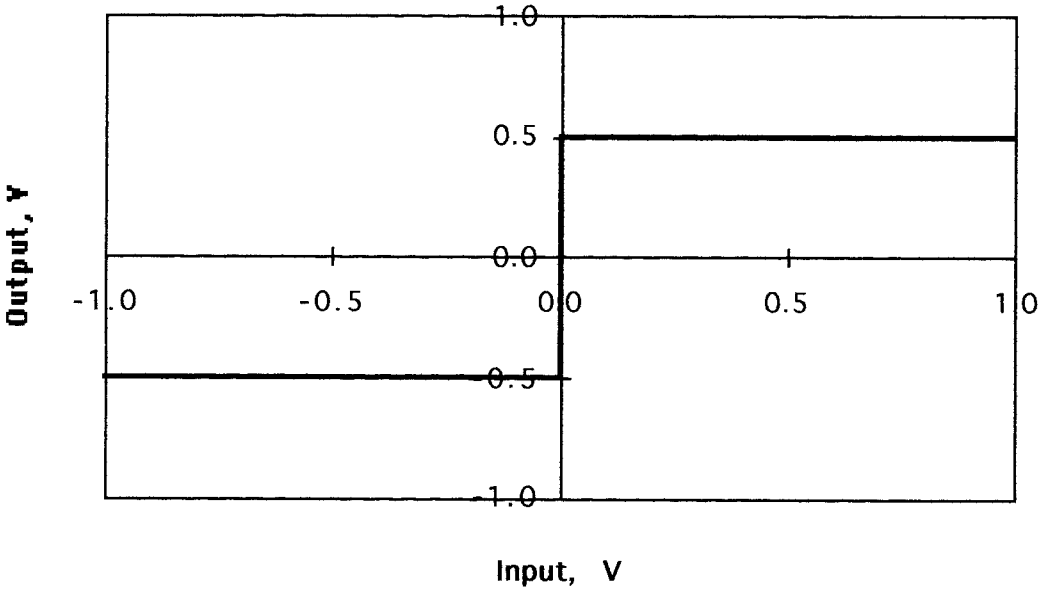

Figure B-2. 1 Bit ADC Translation Table

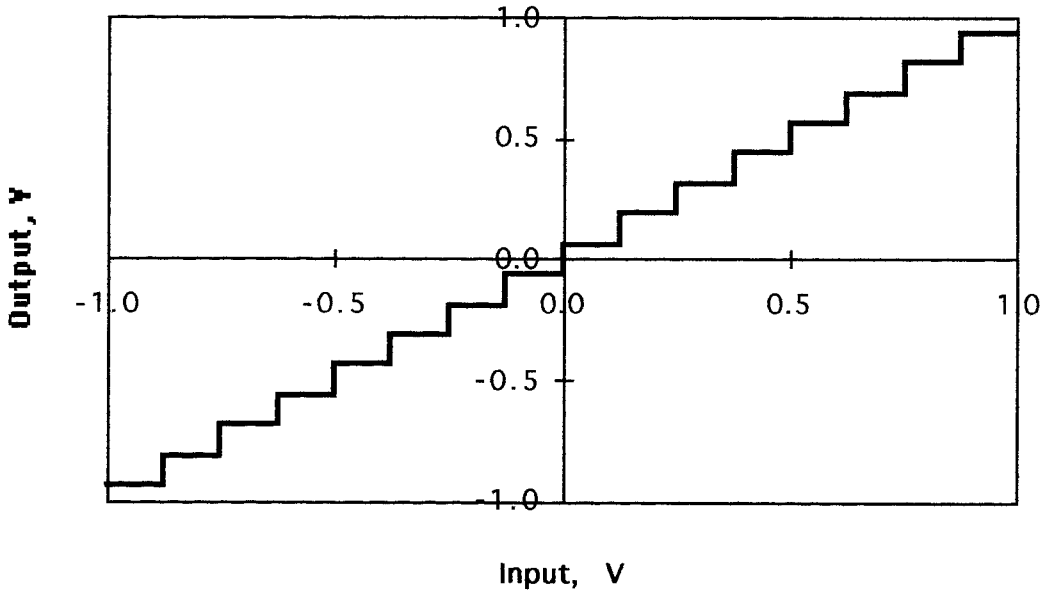

Figure B-3. 4 Bit ADC, No Offset, Gain of 1.0

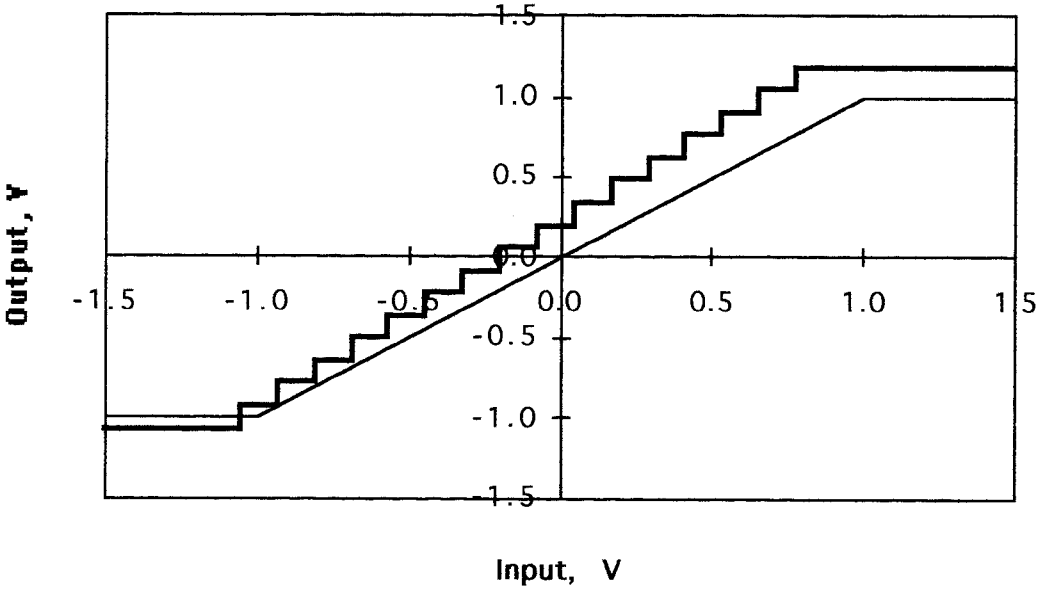

Figure B-4. 4 Bit ADC, With Offset and Gain Error

APPENDIX C: Listing of Circuit Files

This appendix contains a variety of circuit files used within the text. All simulations were run on the HP/Apollo 425t computers at Oregon State University.

1st Order Low Pass Delta Sigma Modulator

```
V1    u     0     STDIN

@a1   v     u     y     1.0   -1.0
@a2   x     v     x2    1.0   1.0
@d2   x2    x     1

q1    y     0     x     0     1b

.step       1.0
.nprint     v(y)
.fft window hann  vdb(y)      >1st.fft
.lib 1b.lib
```

2nd Order Low Pass Delta Sigma Modulator

```
V1    u     0     -20dB 976.5625u

@a1   u2    u     y     1.0   -1.0
@a2   v     u2    v2    1.0   1.0
@d1   v2    v     1

@a3   v3    v     y     1.0   -1.0
@a4   x     v3    x2    1.0   1.0
@d2   x2    x     1

q1    y     0     x     0     1b

.step       1.0
.time       1024.0
.nprint     v(y)
.fft window hann  vdb(y)      >2nd.fft

.lib 1b.lib
```

One Bit ΔΣ Quantizer Model

```
* Generated by GCL Version 2.0.Beta.2, (UNIX) March 31, 1992
*  Number of Bits :      1
*  Input, Low     :     -1.000000
*  Input, High    :      1.000000
*  Output, Low    :     -2.000000
*  Output, High   :      2.000000
.MODEL 1b
0.0          1.0
-9.9e+99    -1.0
.END
```

Differential Pseudo 2-Path Integrator

```
# 2 Path Integrator Example Circuit
vin    In     0      SIN    -20dB 1250.0
#vin   In     0      STDIN
e1     1      0      In     0      0.5
e2     2      0      In     0      -0.5
#input loop top
s0103 1      3                    phi3
s0300 3      0                    phi2
c0305 3      5                    1.0
s0500 5      0                    phi2
s0507 5      7                    phi1
s0104 1      4                    phi5
#input loop bottom
s0204 2      4                    phi3
s0400 4      0                    phi2
c0406 4      6                    1.0
s0600 6      0                    phi2
s0608 6      8                    phi1
s0203 2      3                    phi5
#feedback top
c0709 7      9                    1.0
c0711 7      11                   1.0
c0713 7      13                   1.0
s0915 9      15                   phic
s0900 9      0                    phib
s1115 11     15                   phib
s1100 11     0                    phia
s1315 13     15                   phia
s1300 13     0                    phic
```

```
#feedback bottom
c0810 8     10                1.0
c0812 8     12                1.0
c0814 8     14                1.0
s1016 10    16                phic
s1000 10    0                 phib
s1216 12    16                phib
s1200 12    0                 phia
s1416 14    16                phia
s1400 14    0                 phic
#output configuration
s1517 15    17                phi3
s1618 16    18                phi3
s1518 15    18                phi5
s1617 16    17                phi5
#and the amplifier
e3    15    16    8     7     Ampgain=99dB
# eliminates singluar system errors...
c1700 17    0                 0.00001
c1800 18    0                 0.00001
###
.step 0.5u
.time   4096.0u
.sample       10101010101010101010101010
###
.clock   phi1 10101010101010101010101010
.clock   phi2 01010101010101010101010101
.clock   phi3 10100000101000001010000000
.clock   phi5 00001010000010100000101000
.clock   phia 11000011000011000011000000
.clock   phib 00110000110000110000110000
.clock   phic 00001100001100001100001100
###
.print              v(In) v(17,18)   >2_path_integrator.prn
.fft  window black vdb(17,18)        >2_path_integrator.fft
###-------------
```

Three Stage Modulator

```
;      Three Stage Modulator from Cataltepe
Vin    1      0                  sin   0.1   6835.9375
****
xh1    1      11     72     2     tms2
e1     2      0      0      72    1e2
xq1    2      11                dlydq
****
@a1    22     1      2      8.0   8.0
****
xh2    22     12     73     3     tms2
e3     3      0      0      73    2e2
xq2    3      12                dlydq
****
@a2    23     22     3      8.0   8.0
****
xh3    23     13     74     4     tms2
e3     4      0      0      74    1e20
xq3    4      13                dlydq
****
xhA    11     31                hA
xhB    12     32                hB
xhC    13     33                hC
@aAC   51     31     33     1     0.015625
@aFIN  52     51     32     1     0.125
.nprint       v(52,0)                        >3_stage_mod.prn
.fft          window hann       vdb(52,0)    >3_stage_mod.fft
```

```
.sample           01
.clock phi1       10
.clock phi2       01
.period     period=1u
.time       time=2048u
****
.subckt hA        1     2
xin   1     10                sh2
@d1   2     10    4
.endsub hA
.subckt hB        1     5
xin   1     10                sh2
@d1   2     10    2
@a1   3     2     10    1     -1
****
@d2   4     3     2
@a2   5     3     4     0.0   1.0
.endsub hB
.subckt hC        1     5
xin   1     10                sh2
@d1   2     10    2
@d2   3     2     2
@a1   4     10    2     1     -2
@a2   5     3     4     1     1
.endsub hC
****
```

```
.subckt tms2      1     9     4     5
s12   1     2                 phi2
s20   2     0                 phi1
ca1   2     4                 1.0
s13   1     3                 phi2
s34   3     4                 phi1
ca2   3     5                 1.0
****
s56   5     6                 phi2
c2    4     6                 1.0
****
s98   9     8                 phi2
s80   8     0                 phi1
s97   9     7                 phi2
s74   7     4                 phi1
cb1   8     4                 1.0
cb2   7     5                 1.0
.endsub tms2
.subckt dlydq     1     3
xsh   1     4                 sh2
qint  2     0     4     0     4b
@dint 3     2     1
.endsub dlydq
.subckt sh2       1     3
s12   1     2                 phi2
chold 2     0                 1.0
ehold 3     0     2     0     1.0
.endsub sh2
****
.library four_bit_adc_table
```

4-Bit ADC Table

```
; This table from Cataltepe, used in 3_stage_modulator
.model 4b
0.9375      0.9375
0.8125      0.8125
0.6875      0.6875
0.5625      0.5625
0.4375      0.4375
0.3125      0.3125
0.1875      0.1875
0.0625      0.0625
-0.0625     -0.0625
-0.1875     -0.1875
-0.3125     -0.3125
-0.4375     -0.4375
-0.5625     -0.5625
-0.6875     -0.6875
-0.8125     -0.8125
-999e99     -0.9375
.end
```