Copyright © 1981, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

PERFORMANCE LIMITS OF INTEGRATED CIRCUIT SIMULATION ON A DEDICATED MINICOMPUTER SYSTEM

by

E. Cohen

Memorandum No. UCB/ERL M81/29

22 May 1981

(cover)

Performance Limits of Integrated Circuit Simulation on a Dedicated Minicomputer System

by

Ellis Cohen

Memorandum No. UCB/ERL M81/29

22 May 1981

ELECTRONICS RESEARCH LABORATORY

College of Engineering

University of California, Berkeley

94720

Performance Limits of Integrated Circuit Simulation on a Dedicated Minicomputer System

Ellis Cohen

ABSTRACT

The decreasing costs of minicomputers, coupled with recent improvements in performance, may make these machines ideal to meet the circuit simulation needs of the integrated circuit (IC) designer. An investigation has been made concerning the performance limits of circuit-level IC simulation on a dedicated microprogrammable minicomputer system.

Two types of factors limit simulation performance. First, extended precision (usually 64 bits) is customarily needed and therefore used in all floating-point calculations in circuit simulators. This precision requires twice the memory and twice the execution time of single-precision arithmetic. Second, the functional units available in the computer are not used efficiently because there is not a good match between the minicomputer instruction set and the types of operations that are needed for the simulation.

This investigation shows that circuit analysis can be performed successfully with single precision (32-bit) arithmetic through the use of a combination of numerical pivoting, sparse matrix techniques, a generalization of the indefinite admittance matrix, and voltage thresholds in the algorithms controlling convergence. The use of microcode to extend the computer instruction set greatly improves simulation speed by directly and efficiently

performing several of the most time-consuming portions of the analysis. These special instructions, coupled with the smaller wordsize of the minicomputer, reduce the memory requirements of the circuit simulator by as much as two-thirds compared to the memory needs on larger computers.

One result of this investigation has been the development of a new IC simulation program, SPUDS, which has been designed to obtain the best possible performance from a specific minicomputer system. The analysis results from SPUDS using 32-bit floating-point computations for a wide range of both analog and digital, bipolar and MOSFET circuits are almost identical with solutions based on the more common 64-bit arithmetic. Overall simulation speeds of the same order as the DEC VAX 11/780 computer are achieved using a 16-bit minicomputer. Conclusions are made concerning the suitability of several microprocessor systems for circuit-level simulation.

ACKNOWLEDGEMENTS

I would like to thank my research advisor, Professor D. O. Pederson, for the support and encouragement that he has given me throughout the course of this research.

This work was furthered by many helpful discussions with, and programming help from, G. Boyle, R. Dowell, S. P. Fan, L. W. Nagel, A. R. Newton, and A. Vladimirescu. The assistance of J. Deutsch, J. Kleckner, A. Lachner, T. Quarles, and the other members of the Integrated Circuits Group at the University of California, Berkeley, is gratefully acknowledged.

The financial support and generous equipment grants provided by Corporate Engineering at Hewlett-Packard Company, and the encouragement provided by M. Brooksby, R. Smith, and W. McCalla is gratefully acknowledged. This research has also benefited from the support of the U.S. Army Research Office under grant DAAG29-81-K-0021.

I would especially like to thank my parents, Mel and Rose, for all the guidance, love, and encouragement they have given me.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

The decreasing costs of minicomputers, coupled with recent, substantial improvements in performance, make these machines ideal candidates to perform the circuit simulations needed by each integrated circuit designer. For $25 to $30K, 16-bit wordlength minicomputer-based systems such as the Hewlett-Packard 1000 F-Series [HP78F] are available with floating-point arithmetic instruction execution times on the order of 5 microseconds (within a factor of 2 of larger computers such as the VAX 11/780 [DEC78V]). But when the mainframe version of a simulation program is implemented in a straightforward manner on the minicomputer, the execution speed is slower by factors of 10 or more. Clearly, this speed degradation cannot be attributed simply to the difference in arithmetic operation speeds.

Several minicomputer IC simulators [Youn76] [Bieh74] [Barh73] have been in the public domain for some time. These efforts to date have essentially transferred mainframe circuit simulator codes to the small computer system with minimum changes to the program structure. These simulation programs use basically the same solution algorithms as large-scale simulation programs such as SPICE2 [Nage75]; numerical problems are avoided by using extended precision (typically 64 bits) for floating-point variables. It is desirable to avoid such extended precision calculations, both because of the increased cpu time such calculations require and because of the additional memory required.

One result of investigating the performance limits of circuit-level IC simulation on a dedicated microprogrammable minicomputer system is the development of SPUDS, a new integrated circuit (IC) simulation program. In SPUDS, a better match between the basic computer hardware and the analysis algorithms and data structures is obtained with a combination of numerical pivoting, sparse matrix techniques, a straightforward generalization of the indefinite admittance matrix, and a set of specially-tailored microcoded instructions. The program successfully performs the circuit analysis with single-precision (32-bit) floating-point variables. Freret and Dutton [Frer76] reported some of these methods; SPUDS incorporates a different numerical pivoting algorithm, similar to the one in Program MICE [Cohe78] which is better suited to sparse-matrix techniques. As a result, the penalties associated with multi-word memory references (for floating-point variables) and with tracing through linked-list pointers to manipulate the linear equation coefficient matrix are essentially eliminated. Both the analysis algorithms and the 'effective' computer hardware (through microcoding) have been modified as part of the synthesis of a fast, effective IC simulation tool for the circuit designer.

Chapter 2 compares the strengths and weaknesses of different computer architectures and programming languages in terms of the problem of IC simulation. To obtain the fastest possible simulation requires the use of shorter-precision arithmetic than is used on the larger mainframe computers; Chapter 3 describes the algorithm modifications and performance trade-offs involved. Memory costs are dropping continually; however, it is still desirable, especially for a personal design system, that the memory required to perform the simulation be minimized. Chapter 4 describes the data structures used in SPUDS and the strategies adopted which reduce

main-memory needs.

Optimal performance in a simulator is obtained when the algorithms used and the hardware available are well-suited to each other. The effects of such special-purpose hardware are modelled with microcoded instructions added to a user-microprogrammable minicomputer. Chapter 5 describes the development of these instructions and the resulting improvements in simulation performance.

A summary of the conclusions reached based on this research is given in Chapter 6. A brief evaluation of the suitability of several currently-available processors is also presented, and some suggestions are made for future work in the development of dedicated IC-design systems.

Appendix 1 describes the implementation of the SPUDS program. Information concerning a complete listing of SPUDS is given in Appendix 2. Appendix 3 lists the input files describing each of the benchmark circuits referenced in this report.

CHAPTER 2

PERFORMANCE OVERVIEW

The actual execution speed of circuit simulation programs such as SPICE2 [Vlad81] is far less than is implied by the potential performance of present-day computer hardware. Although this failure to achieve potential machine performance often arises for other types of computer programs it is especially troublesome for simulation codes. These codes are characterized by the nearly ideal computational characteristics of little I/O, small working-set size, and well-defined computational kernels. A good example of this lack of optimal performance is a carefully-constructed LU factorization algorithm [Cala79] written specifically for the CRAY-1 computer [CRAY76]. Although the asymptotic achievable execution speed on the CRAY-1 is 140 million floating-point operations/second (140 MFLOPS), this algorithm can be expected to achieve a maximum execution rate of only 35.8 MFLOPS. However, for the typical sparse systems of equations which arise in integrated circuit simulation the actual LU factorization speed is roughly 2.2 MFLOPS. This large ratio between potential and actual performance applies not only to large 'super-computers' such as the CRAY-1 but also to small minicomputers. Although the floating-point arithmetic execution times for the Hewlett-Packard 1000 F-Series 16-bit minicomputer and the Control Data 6400 60-bit mainframe are within a factor of 2 or 3 of each other, the equation solution

time for the UA741 operational amplifier[1] is 8.3ms on the CDC 6400 but 252ms (in FORTRAN) on the HP 1000[2]. Clearly, machine hardware resources are not utilized effectively at either end of the computer spectrum.

The reasons for the large discrepancy between optimal and achieved performance can be divided into two categories. First, virtually none of the 'high-level' programming languages allow effective direct access to, utilization of, or control over machine hardware. Second, most computer hardware architectures are poorly suited to the existing algorithms used in circuit simulation.

Effective methods which reduce this performance gap utilize one or more of three basic ideas. First, the high-level programming language can be changed (or a new language developed) to give the programmer better access to, and explicit control over, the machine hardware. As a result, many of the difficulties in attempting to achieve 'optimal' code with a compiler can be avoided. For computers which are microprogrammed, this approach also includes the possibility of generating microcode directly with a language compiler. The current state-of-the-art is succinctly summarized [Cala79]:

> Overall it appears that for the near future only 1/4 to 1/3 of the ultimate machine speed may be routinely available through a high-level language.

A second approach changes existing algorithms to take advantage of the special properties of available machine hardware. Existing pipelined, vector, and multiprocessor computer systems are the objects of considerable research, and major algorithm speedups may be possible [Bane79] [Towl76] [Wolf78] [Sang79] [Hach81] [Lela].

---

[1] all referenced circuits are described in Appendix 3

[2] data from Chapters 3 and 5

The third approach modifies machine hardware to implement existing algorithms more effectively. For a given set of algorithms, the fastest execution speed results when a direct implementation of those algorithms is utilized. Developments in VLSI technology may make it practical to construct such a hardware implementation specifically for IC simulation needs. However, the construction and verification of systems of that size is beyond presently-available methodologies. Therefore, applications of this approach take the form of either special-purpose instructions utilized as part of the standard code-generation phase of a compiler, or special-purpose procedures invoked explicitly at the high-level language.

This chapter describes several strategies for obtaining better performance from computers. This presentation provides a perspective on the hybrid approach developed in this report. There are both potential advantages and real difficulties for each approach towards achieving the performance that is theoretically possible. Emphasis is placed, in the comparisons, on the kinds of operations and algorithms that are typical of IC circuit-level simulation programs. Section 2.1 describes some relevant characteristics of main memory and presents data on the relative importance of memory access time on overall simulation speed for a minicomputer. A description of pipelined vector computers is made in Section 2.2. Section 2.3 details the use of multiple processors to exploit parallelism. The potential use of auxiliary special-purpose array processors is described in Section 2.4. Finally, Section 2.5 presents the approach investigated in this report.

2.1. Main Memory

Both the pattern of references and speed of access to main memory are critical factors in the determination of overall simulation speed. The order

in which different words of memory are accessed is important because most computer main memories are interleaved. Interleaving utilizes multiple banks of memory, arranged so that consecutive memory addresses are in different banks; if a four-way interleaved memory is used and consecutive memory addresses are referenced, the effective memory access time is improved by a factor of four. Worst-case behavior occurs if a program consecutively accesses every fourth memory location. Clearly, the ways in which data are stored can affect execution speed. However, it is not always possible to take good advantage of interleaving. Most circuit-level simulation programs assemble a two-dimensional matrix of equation coefficients. Such a matrix can be stored such that accesses along only rows or only columns of the matrix will use different memory banks. For some kinds of row-and-column access, a skewed storage allocation technique [Kuck77] can be used. But for integrated circuits, this coefficient matrix is very sparse and is usually stored as a compressed, one-dimensional vector. In terms of memory addresses, accesses to terms in this vector are made in a very nearly random manner.

Table 2.1 shows the effects of different memory access times on overall simulation speed. The simulation code is written entirely in FORTRAN and is adapted from Version E.3 of Program SPICE2[3]. Two HP 1000 E-Series 16-bit minicomputers which differ only in memory cycle time are used. The typical cycle time of the 'standard' memory is 665ns (read or write); the 'high' speed memory requires 420ns to read and 350ns to write. The E-Series minicomputer performs all 32-bit floating-point calculations by firmware (microcode); 32-bit floating-point arithmetic execution times are $\approx 20\mu s$ for addition

[3]Reference is made frequently in this report to Version E.3 of SPICE2 because the initial programming for SPUDS is derived from SPICE2.

Numerical precision	Memory speed	CPU Model	DIFPAIR		UA741		MOSAMP2	
			DCX	TRAN	DCX	TRAN	DCOP	TRAN
32 bits	standard	E	26.6	–	–	–	–	–
32 bits	high	E	21.4	–	–	–	–	–
32 bits	high	F	18.3	–	–	–	–	–
48 bits	standard	E	30.4	44.9	202.5	250.1	94.3	718.8
48 bits	high	E	24.7	37.0	166.0	206.6	78.1	600.1
48 bits	high	F	20.1	26.8	131.4	148.0	52.8	379.3
64 bits	standard	E	103.8	197.0	747.6	1119.0	459.0	3781.0
64 bits	high	E	81.6	150.7	585.6	855.7	352.9	2876.0
64 bits	high	F	21.4	27.9	136.7	154.2	56.4	399.7

Table 2.1. Effects on Performance of Different Memory Cycle Times

and 40μs for division. According to the manufacturer [HP77E] the dependence of floating-point calculation times on memory speed is less than 3% (the time required to perform the floating-point calculations is much greater than the time spent accessing operands in memory).

The analysis times for three representative circuits are shown in Table 2.1. The 'DCX' column gives the central processing unit (CPU) time required to perform a dc transfer curve analysis, in which a dc operating-point computation is made repetitively as an independent source is swept across a range of values. The 'DCOP' and 'TRAN' columns show the CPU time needed to evaluate the dc operating point and perform a transient analysis, respectively. The CPU times in the table are in seconds; a value of '—' indicates that the particular run did not converge. When convergence is obtained, the number of iterations required depends only on the circuit and not on the numerical precision used.

The data show that the effects of memory access time on simulation speed depend strongly on the floating-point precision used. The reduction in memory access time by 33% reduces the total simulation time by 25% for four-word (64-bit) precision. The reductions for three-word (48-bit) and two-word (32-bit) precision are 18% and 15%, respectively.

Table 2.1 also shows the corresponding simulation times when an HP 1000 F-Series minicomputer with 'high' speed memory is used. The F-Series machine has hardware which directly implements floating-point arithmetic operations for 32-bit, 48-bit, and 64-bit precision operands. These CPU times show that a hardware implementation of floating-point arithmetic is most important when extended precision is used. For the 64-bit precision transient analysis of the MOSAMP2 circuit, the hardware increases simulation

speed by a factor of 7; when only 32-bit precision is used, the speedup due to hardware is only ≈15%.

2.2. Pipelined Vector Computers

Hardware speedup is obtained either 'horizontally' by replication of instruction or data streams, or 'vertically' by segmenting (or pipelining) those streams [Flyn72] [Kuck78]. Section 2.3 describes multiprocessor machines; this section concerns single-processor 'vector' computers.

Vector processors, such as the CRAY-1 or CYBER 205, have instructions which operate on vectors. This design reduces significantly the number of references to main memory for processor instructions. Also, since vector elements are frequently stored in consecutive memory locations, interleaving of memory banks can be best utilized. Of greater importance for processor speed is the capability to pipeline the arithmetic calculations. Pipeline computation decomposes a complex, time-consuming task such as floating-point multiplication into a sequence or 'pipe' of simpler and faster operations, e.g. fixed-point multiplication. Part of the increase in speed is due to the concurrent execution of these operations as the operands pass between segments in the pipeline. For an n-segment multiplier, this hardware architecture can evaluate up to n multiplications concurrently, although any particular result is not available until n clock cycles after the multiplication is started. The speed advantage of vector processors is due primarily to the potential of pipelining, although some additional performance is obtained from multiple functional units.

The extent to which vector instructions can be used effectively in a simulation program depends on the definition of 'vector' used by the manufacturer. For example, after the linearized component values for the

semiconductor device models are evaluated, those values are incrementally loaded into a matrix of equation coefficients. An indexing scheme [McCa71] is used to access these elements because the matrix is usually 85% to 95% sparse [Berr71]. As much as 22% of the total simulation time can be spent performing this matrix load operation[4]. In order to cast this load process in a form suitable for vectorization, it must be possible to define a vector indirectly; rather than using a contiguous block of memory to store vector element values, the memory block is used to store the addresses of those values. With such a capability the load operation can be expressed as

```
      DO 10 I=1,N
            MATRIX (INDEX(I)) = MATRIX (INDEX(I)) + TERMS(I)
10    CONTINUE
```

and can execute at vector operation speed. Instructions which allow vectors to be defined indirectly are not part of the repertoire of the CRAY-1 computer, although such instructions do exist for the CYBER 203 [Kasc79] [CDC80].

2.2.1. Vector Algorithms

Algorithms suitable for vector machines require both a series of identical operations that can be performed concurrently and a set of data arranged so it can be streamed into the ALU. This last arrangement usually requires that a linear indexing function be used to address the data. As a result, vector machines are especially well-suited to finite-difference algorithms for simulation of physical systems (such as weather modelling) in which banded matrices are used to solve systems of differential equations. Circuit simulation, however, raises special problems.

---

[4]Chapter 5

The algorithms used in circuit simulation are described in detail in Chapter 3. Briefly, circuit simulators predict circuit behavior by manipulating mathematical models relating branch voltages and currents for the circuit elements. For circuits containing elements with nonlinear branch relations, an iterative algorithm is used. A 'guess' is made of the actual operating point for each of the nonlinear branches in the circuit. Each branch is then linearized about that presumed operating point and the resulting linear system is then solved. If the solution does not agree sufficiently well with the 'guess', a new estimate of the actual operating point is calculated and the linearization and solution steps are repeated.

In a typical circuit simulation program such as SPICE2, the equations describing the behavior of each semiconductor device are generally expressed in terms of the most recent iteration's 'guess' at the device operating point. The particular device equations which apply to that region of operation are then evaluated. Thus, rather than

```
conductance = f(arg1,arg2,arg3)
```

the logical flow of the model evaluation code resembles

```
if (device is in saturation)
      conductance = f1 ( argument list )

else if (device is active)
      conductance = f2 ( argument list )

else          "device is off"
      conductance = f3 ( argument list )
```

The efficient implementation of this type of code on a vector computer is made difficult by the logic branches in the equation formulation. Typical vector instructions for a computer such as the CYBER 200-series perform operations on ordered scalar quantities, reading operands from consecutive

storage locations, performing some designated operation(s), and possibly storing the result back in memory. No provision is made for conditional changes in the flow of control, since in order to keep the pipeline full, n-way branching hardware would have to be available to evaluate all the possible values and extract the desired number. Automatically performing this look-ahead requires much more logic in the computer, and such a method becomes prohibitive as soon as more than one or two conditional actions are introduced into the computation [Toma67].

Solutions to this implementation problem generally compute all possible results using vector-mode instructions and then select the desired value (with another vector-mode instruction). For the sample given above of device model evaluation, all three expressions are evaluated and the results stored in temporary vectors. The final result vector of conductances is assembled by selecting each entry from the appropriate temporary vector, using a vector merge instruction. The amount of reduction in total CPU time that this approach offers depends on many factors, including the extent of pipelining, the number of equivalent functional blocks in the ALU, and the relative probability that each of the different device operating points will occur (and require evaluation).

Clearly, such a technique is worthwhile only if the potential savings in total computation time is large. Detailed timing measurements of a carefully optimized simulation program are presented in Chapter 5. The data show that only 10% to 15% of the total simulation time is spent evaluating the model equations represented by the above code fragment. This evaluation effort does not include the execution time required to load the contributions from each semiconductor device into the coefficient matrix. Since the best

possible vectorization could not eliminate totally that part of the computation, the possible savings in CPU time are no more than 5% to 10% of the total simulation time. To be effective, vector instructions must be used for the majority of the simulation code, not just for model evaluation. A high-level language is needed, however, to keep the programming effort within reason.

2.2.2. Programming Languages

The ability to express both vector operations and potential parallelism in arithmetic operations is important if the vector instructions and multiple functional units in a high-speed vector computer such as the CRAY-1 or CYBER 203 are to be used effectively. A language such as APL, which is basically vector oriented, is quite suitable. However, the most widely-available high-level language for vector processors is FORTRAN; for the most powerful computers of this type, the FORTRAN language is the only one (other than assembler) supported by the manufacturer. Many of the difficulties that arise in automatically generating effective vector-mode instructions with a FORTRAN compiler are due to limitations of the language. Some of these constraints are relaxed by the manufacturer through extensions to the FORTRAN language. For example, on the CYBER 203 the FORTRAN compiler recognizes a 'vector descriptor' which is used to specify explicitly that a variable be treated as a vector. Rather than coding a vector summation as

```
      DO 100 I=1,N
        ...
        B(I)=C(I)+D(I)
        ...
100   CONTINUE
```

in which form the FORTRAN compiler may or may not generate vector instructions, one can instead use

```
BD = descriptor of vector B(*)
CD = descriptor of vector C(*)
DD = descriptor of vector D(*)
...
BD = CD + DD
...
```

and force the FORTRAN compiler to generate vector-mode instructions.

If extensions to ANSI-standard FORTRAN are not used, the compiler must recognize situations for which vector instructions are suitable. Although the FORTRAN 'DO-loop' can be taken as a hint by the compiler, many heuristics become important when the costs of initiating vector operations are included. Algorithms which speed up FORTRAN-like programs for array machine computation have been well-developed [Kuck72] [Bane79] [Kuck78]. Nevertheless, many available compilers impose severe coding constraints on program loops in order for effective vector-mode instructions to be generated by the compiler.

An example illustrating some of these constraints is given below. Although the example is specific to the FORTRAN compiler for the CRAY-1 computer, the constraints are representative of other machines such as the CYBER 203. The FORTRAN compiler for the CRAY-1 tries to utilize vector operations whenever appropriate loop structures are detected. In particular, inner-most DO-loops are candidates for vectorization. However, vector-mode code generation is totally disabled if the loop contains any input, output, procedure call, IF, or GOTO statements. If any array subscript expressions are not linear functions of the DO-loop index variable, are parenthesized, or use a scalar temporary variable, vectorization is inhibited. Also, vector dependencies (loops in which operands are needed in a different order than generated by the vector hardware instructions) can inhibit vec-

torization. These constraints severely limit the extent to which vector-mode instructions are generated for a program module. Major restructuring of analysis algorithms may be necessary to make the program code suitable for vectorization. Some built-in vector functions are provided which can replace IF statements and permit the use of vector-mode operations in some situations. An example of such a built-in vector function for the CRAY-1 is the function

```
CVMGP ( X1, X2, X3 )
```

which takes three vector-element arguments and returns

```
X1      IF ( X3 .GE. 0.0 )
X2      IF ( X3 .LT. 0.0 )
```

However, use of such functions can incur substantial penalties in unnecessary computation. Consider the loop

```
        DO 10 I=1,N
              IF ( V(I) .GE. 0.0 )  V(I) = EXP ( V(I) )
10      CONTINUE
```

which can arise as part of the model evaluation code for a device with a *p-n* junction. This loop is not vectorized by the CRAY-1 FORTRAN compiler due to the presence of the IF statement. However, the code can be rewritten as

```
        DO 10 I=1,N
              V(I) = CVMGP ( EXP(V(I)), V(I), V(I) )
10      CONTINUE
```

which *is* vectorized.

For a large digital circuit many of the devices may be 'off', corresponding to a situation in which most of the elements in the voltage vector V are less than 0.0. For such a case the first code fragment above is faster, even

though it is not vectorized, because the vectorized code evaluates the exponential function for every element of V (even though the values are immediately thrown away). Orders-of-magnitude improvement in simulation speed have been obtained by not evaluating 'dormant' portions of large digital systems (using event-driven analysis [Szyg76] [Newt78]); vectorization of the model evaluation code alone cannot accomplish as great an improvement in speed.

2.3. Multiprocessor Computers

Multiprocessor computers obtain greater performance through hardware replication of instruction and/or data streams. If n parallel processors are applied to a problem, an ideal speedup by a factor of n is possible. At least three factors constrain the potential speedup. First, the degree of parallelism in almost any program is not uniform; therefore, all the processors cannot be kept busy. Second, if resources are shared (main memory), contention between processors degrades overall performance. Finally, transforming serial algorithms into parallel ones does not necessarily result in a theoretical speedup factor of n.

The potential improvement in simulation performance of such computers is considerable. Circuit simulation can be decomposed into the two tasks of constructing a set of simultaneous linear equations and solving those equations. The evaluation of the nonlinear device model equations for each circuit element does not depend on the state of any other circuit elements if direct equation solution techniques are used; all the evaluations theoretically can be performed in parallel. Interactions between devices occur only through the solution of the total circuit equations. For state-of-the-art simulators and circuits which require up to several hundred equations,

approximately 10% of the total CPU time is spent solving those equations. Suppose that 100 processors are applied in parallel to evaluate the equation coefficients (the remaining 90% of the total). If the algorithms are partitioned such that each processor evaluates the model for one device and if memory access contention does not degrade overall system performance, the total analysis time can be reduced by a factor of $\frac{0.1 + 0.9}{0.1 + (0.9/100)}$ or ≈9. A major problem with such an architecture is the need for inter-processor communication to ensure that the computations are performed in the correct sequence. The time required for the communication may greatly exceed the execution time needed for the desired computations.

2.3.1. Multiprocessor Algorithms

The effective use of more than one processor requires that some intermediate parts of the computation be evaluated concurrently. Such parallelism is present in virtually all programs, at least at the level of arithmetic expressions. For example, in evaluating the statement

$$T = (A \times B) + (C \times D)$$

both multiplications can be performed at the same time. Algorithms have been investigated [Kuck78] which identify the ordering of arithmetic expression operations to obtain maximum parallelism with multiple processors. The extent to which such parallelism can be exploited depends on the interconnection network of the processors.

Multiprocessor computers are potentially much faster than vector machines for code that contains many IF-THEN-ELSE decision trees. The speedup is due to the ability to evaluate possibilities in parallel. The extent to which code evaluation speed increases depends upon the relative probability that the different control paths in the decision trees are taken during

execution. The best case is for equal probabilities for all paths of control. Multiprocessor machines also have the potential to be faster than vector computers for code with nonlinear recurrences or with subscripted subscripts [Padu80] because memory addresses may be more uniformly distributed. (On the CYBER 203 vector processor, the 'subscripted subscript' problem is circumvented by 'gather' and 'scatter' instructions. These instructions take an indexing vector INDEX and directly implement

```
      DO 10 I=1,N
            B(I) = A(INDEX(I))
10    CONTINUE
```

and

```
      DO 10 I=1,N
            A(INDEX(I)) = B(I)
10    CONTINUE
```

at the rate of one data transfer every 1.25 clock cycles, or essentially at vector-operation speeds.)

2.3.2. Programming Languages

Programming multiprocessor computers revolves around the expression of parallelism in the computation. Ideally one would like to have the compiler detect possible concurrency and automatically produce an 'optimal' set of machine instructions. However, theoretical problems make such automatic code generation difficult. Even with simple programs, the problem of translating sequential programs into parallel ones of minimal execution time is NP-complete [Ullm76]. A considerable amount of research has been published on improving the parallelism of programs and on the interconnection problems of multiprocessors [Haie79] [Wen76].

An alternative to automatic recognition of parallelism is explicit program control of concurrency. Dennis and Horn [Denn66] have suggested FORK, JOIN, and QUIT as primitives to initiate and control multiple processes; other suggestions [Tsic74] [Hans77] have included the use of PARBEGIN and PAREND for bracketing of iterated control blocks which can be evaluated in parallel.

2.3.3. Dataflow Machines

Dataflow computers take the greatest advantage of all possible concurrency in programs by close examination of the data dependencies among the program statements. These machines differ from the traditional von Neumann machines which are driven by some control mechanism (system clock and program counter). Rather, dataflow machines initiate activities asynchronously based on the availability of the information needed for each activity.

The dataflow concept is currently the subject of considerable research. New high-level languages are under development [McGr80] [Comt79] and prototype computers have been constructed or are in development [Davi77] [McGr80]. It remains to be seen what ultimate speeds these machines can achieve. However, it is clear that major improvements in performance will result from the use of dataflow to reduce the complexity of inter-processor communications.

2.4. Array Processors

Array processors are special-purpose computers designed to allow vectors (contiguous blocks of numbers) to be processed efficiently. These machines are particularly well-suited for problems which have a 'setup'

phase that can be separated from the main computation (and need high-speed arithmetic). The more sophisticated array processors are user programmable, are capable of speeds in excess of 10 MFLOPS, and may run in parallel with the host computer system. There are three basic types of data formats: integer, block floating-point, and true floating-point. Integer or fixed point machines require the programmer to do any necessary normalizing or scaling of the data in order to retain accuracy. Only true floating-point processors are described in this section because of the dynamic range requirements of circuit simulation.

For 'true' floating point representations, each element has its own exponent. The Floating Point Systems AP-120B [FPS] uses a 28-bit mantissa and a 10-bit exponent, while the Computer Signal Processing Inc. MAP-300 [CSPI] uses 25 and 7 bits respectively. Potential conversion problems can arise when these processors are joined to a host computer because of differences in the formats of floating-point numbers. For example, neither of these formats are the same as the Hewlett-Packard 1000 floating-point representation, which uses a 24-bit mantissa and an 8-bit exponent. The AP-120B uses software for conversion to the HP format, while the MAP-300 provides hardware translation on the interface.

Two different design philosophies have been used to achieve comparable processing speeds on the AP-120B and MAP-300. The AP-120B uses a pipelined arithmetic logic unit (ALU); the time per stage is fixed at 167ns. Addition requires two stages; multiplication requires three stages. The MAP-300 uses a parallel architecture to achieve its processing speed; two multiplier-adder units operate in parallel with an address processor, a control processor, and one or more I/O processors. Arithmetic routines can thus be writ-

ten without regard to buffer control or I/O processing.

Several non-obvious factors affect the evaluation of the relative speeds of different array processors. Operating system overhead may limit throughput; additional memory often relieves this problem. Processor and memory options can affect computation times by a factor of four or more; for example, changing from MOS to bipolar memory can double the speed, price, and power consumption of an array processor such as the MAP-300.

2.4.1. Programming Languages

The programming of array processors requires very high-level skills because very low-level timing-dependent control must be specified. Although the manufacturers generally provide a library of utility routines, special-purpose codes must be written by the user. A FORTRAN compiler is available for the APS-120B machine, but the compiler generates rather inefficient code. In addition, the APS processor limits to 4K words the size of program code that can be downloaded from the host computer. This constraint corresponds to roughly 200 lines of FORTRAN. For comparison, the length of the subroutines in Program SPICE2 which evaluate the MOSFET device model is more than 1500 lines of FORTRAN.

2.5. Hybrid Approach

The strategy explored in this report examines the potential of several techniques mentioned in this chapter to obtain maximum performance in a dedicated, minicomputer-based circuit simulation system. Conceptually, the computer hardware is modified to perform most effectively the time-consuming parts of the circuit analysis. These modifications are modelled with instructions that are implemented in microcoded procedures. The

speedup factor for the resulting dedicated hardware and software system is as much as 20 on major portions of the calculations. This increase in performance is obtained in part by using to the fullest extent all available hardware in the computer and by overlapping arithmetic calculations and memory references asynchronously. Since the microcoded instructions directly implement higher-level operations, fewer instructions (and correspondingly fewer memory references) are used, which also increases program speed [Tann78]. Finally, the simulation algorithms are modified to make calculations using 32-bit floating-point give accurate answers, so that extended precision computations are not required.

CHAPTER 3

ALGORITHMS

The algorithms used in circuit simulation directly affect simulation speed, accuracy, and convergence. A brief summary of existing minicomputer-based circuit simulators is presented in Section 3.1. The circuit analysis methods used in Program SPICE2 [Nage75] are reviewed in Section 3.2, since those methods are representative of the ones used in presently-available simulators running on a wide variety of large-wordsize computers. Virtually all of the arithmetic calculations in these simulators are performed using 60 to 64 bits for each floating-point variable.

It is desirable to use fewer bits per variable for floating-point arithmetic, because such a change leads to a significant reduction in both computation time and memory requirements. The algorithms presented in this chapter have been found to be useful in improving simulation speed and/or accuracy in the context of 32-bit floating-point arithmetic. These algorithms are divided into two categories: methods which enhance the numerical accuracy obtained with 32-bit floating-point arithmetic, and methods found necessary for convergence of the iterative solution process with shorter wordlength (regardless of accuracy considerations). Methods for improving accuracy include numerical pivoting [Isaa66], use of an augmented Modified-Nodal-Admittance (MNA) matrix [Idle71] [Nage71] [Jenk71] [Ho75], and combining together both absolute and incremental forms of iteration [Frer76] [Cohe78]. These techniques are described in Section 3.3.

Effective convergence for a wider range of test circuits has been obtained by introducing constraints on the per-iteration change in selected circuit unknowns and by adding several threshold parameters. Section 3.4 describes in detail these parameters and their effects on convergence.

The last section in this chapter presents a comparison between the convergence and accuracy characteristics of these methods and constraints, when used with 32-bit floating-point arithmetic, and the convergence properties of 60- to 64-bit floating-point arithmetic.

3.1. Overview of Existing Minicomputer-Based Simulators

Several simulation programs exist which run on 16-bit wordsize minicomputers. Programs BIASD [Bieh74] and MSINC [Youn76] were initially written for longer wordlength machines; the minicomputer versions of these programs use extended-precision (48- or 64-bit) arithmetic to avoid numerical difficulties. Program BIASM [Barh73] ran on the IBM 1130 and IBM 1800 16-bit minicomputers and used 32-bit floating-point arithmetic. R. Barham added a convergence test which stopped the iteration when no further decrease was observed in the norm of the iteration-to-iteration change in node voltages, $||\delta V||$, (indicating that the numerical precision of the minicomputer had been reached). This modification extended the range of circuits for which the program found a solution. An experimental minicomputer version of Program SPICE1 [Nage73] was developed by P. Freret [Frer76]. This program utilized the indefinite admittance matrix, numerical pivoting, a two-stage Newton-Raphson iteration algorithm, and 32-bit arithmetic. The simulation results agree well with those of SPICE1 running with 64-bit precision on an IBM 370/168 computer. MICE [Cohe78] is another simulation program developed as a result of an investigation of ideas which

reduce memory requirements or cpu time in 16-bit minicomputers. Program MICE successfully utilizes a combination of numerical pivoting, sparse matrix techniques, the use of the indefinite admittance matrix, and both absolute and incremental voltage iteration to perform the circuit analysis with single-precision (32-bit) floating-point arithmetic.

3.2. Summary of SPICE2 Analysis Methods

Program SPICE2 uses the Modified-Nodal-Admittance (MNA) matrix formulation of the circuit equations. The MNA formulation simplifies the processing of circuit branches which are voltage-defined (voltage sources) or current-controlled (current-controlled current sources). For a circuit containing N nodes and B voltage-defined branches, the N-1 non-ground node voltages and the B voltage-defined branch currents are chosen as the unknown circuit variables.

An circuit example is shown in Figure 3.1. For this circuit, the unknown variables are the node voltages V_1, V_2, and V_3, together with the current I(VS) flowing through the voltage source VS. The circuit equations are formulated by writing Kirchhoff's Current Law (KCL) for each unknown voltage and including the branch relation for VS. For the example circuit the system of equations is

$$\begin{bmatrix} \frac{1}{R1} & \frac{-1}{R1} & 0 & 0 \\ \frac{-1}{R1} & \frac{1}{R1}+\frac{1}{R2}+\frac{1}{R3} & \frac{-1}{R3} & 0 \\ 0 & \frac{-1}{R3} & \frac{1}{R3} & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \times \begin{bmatrix} V_1 \\ V_2 \\ V_3 \\ I(VS) \end{bmatrix} = \begin{bmatrix} IS \\ 0 \\ 0 \\ VS \end{bmatrix} \qquad (3.1)$$

This system of linear equations is solved by LU factorization [Cala72] (which requires the same computational effort as Gaussian elimination

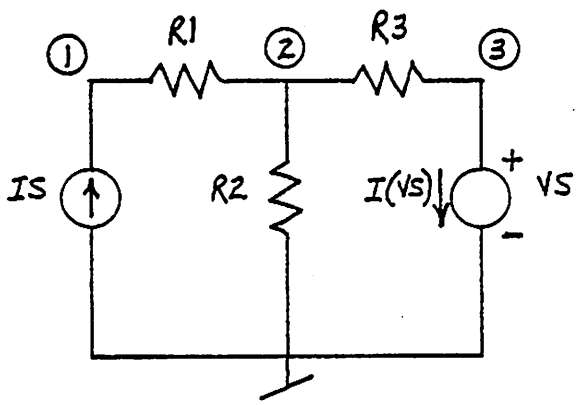


Figure 3.1. Example Circuit

[Rals65].) Note that if no reordering of the circuit equations is performed, the value of the i^{th} pivot (diagonal) entry in the coefficient matrix for the node-voltage unknowns is equal to the sum of the conductances connected to the i^{th} circuit node.

This one-step process is sufficient to find the quiescent (dc) operating point of a linear circuit. To determine the dc solution of a nonlinear circuit, the iterative algorithm shown in Figure 3.2 is used. A 'guess' is made of the actual operating point for each of the nonlinear branches in the circuit. Each branch is then linearized about that presumed operating point and the resulting linear system is then solved. If the solution does not agree sufficiently well with the 'guess', a new estimate of the actual operating point is calculated and the linearization and solution steps are repeated.

The two other frequently-used analyses, small-signal frequency-domain (AC) and large-signal time-domain (transient), build on the dc solution process. AC analysis is performed by solving a set of linear equations at each frequency point. The only difference between this analysis and the equation solution part of the dc analysis is that the matrix coefficients in general are complex-valued. Transient analysis is performed as a series of quasi-dc analyses in which energy-storage elements are modelled as timestep-dependent conductances and current sources [Cala72].

For all of these analyses the solution of a set of simultaneous linear equations is fundamental. Accuracy of the solution is critical, not only for accurate simulation results, but for stable convergence properties as well. The next section describes several methods for improving the accuracy of the equation solution.

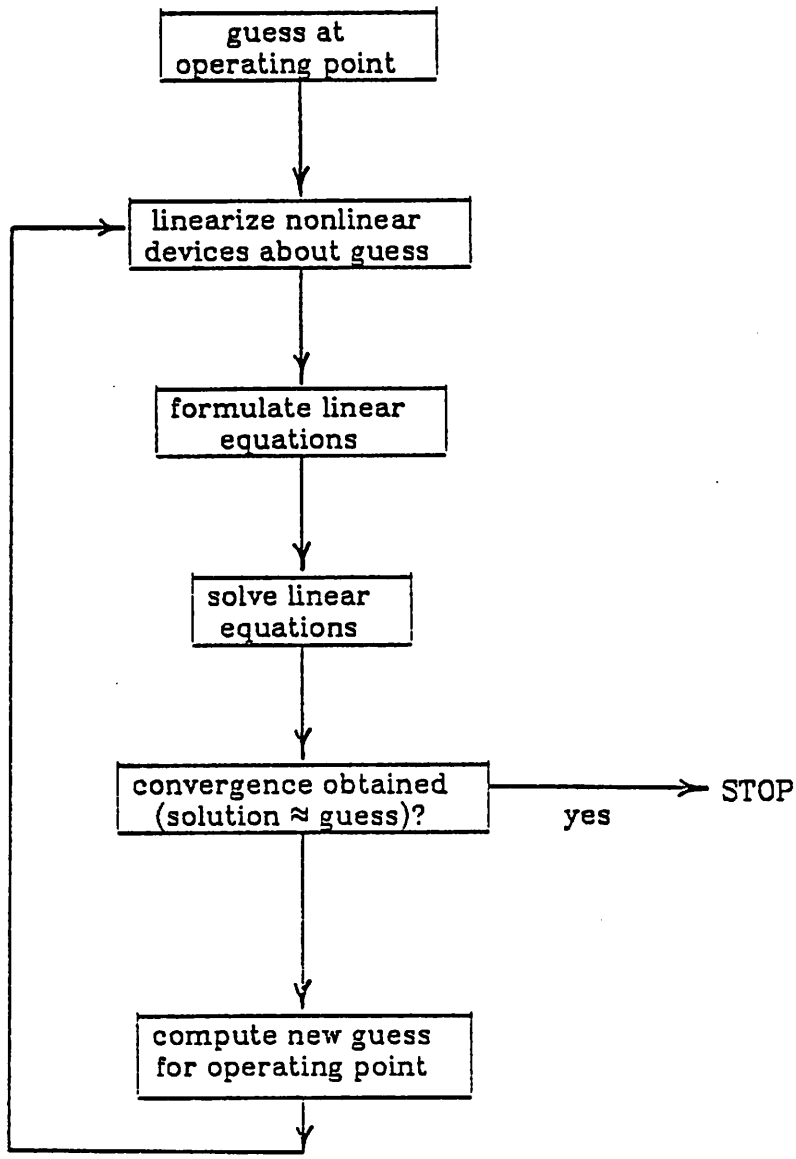


Figure 3.2. Iterative Algorithm for Solution of Nonlinear Circuit

3.3. Accuracy-Enhancing Methods

For the problem here, the algorithms used to solve the set of simultaneous linear equations must be examined carefully in the context of 32-bit floating-point arithmetic. Typically 20 to 23 bits of coefficient are available in those 32 bits [PR1ME] [HP78F] [DEC77V] which means that $\left\lfloor \frac{23}{\log_2(10)} \right\rfloor$ or 6.8 significant digits are used in the computation. Pessimistically, roundoff error in a well-conditioned set of equations can reduce the number of significant digits in the solution by a factor of $1+2\times\log_{10}(N)$, where N is the number of equations [Rals65]. If at least 3 significant digits of accuracy are desired in the output, then at least $3+(1+2\times\log_{10}(100))$ or 8 significant digits are needed for a system of 100 equations[1]. Fortunately, the roundoff error actually incurred is usually much less than this pessimistic estimate.

To obtain the maximum precision possible with a given number of significant digits requires that the circuit equations not be ill-conditioned. A polynomial p(x) is termed ill-conditioned if a small change in one of its coefficients causes a large change in the value of one or more of its zeros. If the coefficient error is due to the finite machine representation, the use of multiple-precision arithmetic can decrease the round-off error and increase the accuracy of the zeros. In an analogous manner, for the system of linear equations

$$\mathbf{A}\times\mathbf{x} = \mathbf{b}$$

the matrix A of equation coefficients is said to be 'ill-conditioned' if 'small' changes in elements of the matrix cause 'large' changes in x, the computed

---

[1]This formula implies that the 64-bit (14 significant digit) precision used on mainframe computers is sufficient for three-digit accuracy in systems of 100,000 equations. Systems of over 1000 equations are solved accurately with 64-bit precision [Nage80]; the solution of much larger systems is limited by the computer time required.

solution vector. One measure of the extent of ill-conditioning of a matrix **A** is the magnitude of $||\mathbf{A}^{-1}||$ after the elements of **A** have been normalized so that $||\mathbf{A}||$ is approximately unity [Rals65]. A large value for $||\mathbf{A}^{-1}||$ indicates an ill-conditioned system. Another measure is the 'condition number' $\mu(\mathbf{A}) \equiv ||\mathbf{A}^{-1}|| \times ||\mathbf{A}||$; it can be shown [Isaa66] that small perturbations $\delta\mathbf{A}$ in the coefficient matrix and $\delta\mathbf{b}$ in the right-hand side cause small relative changes in the solution **x** if

$$\frac{\mu}{1 - \mu \times ||\delta\mathbf{A}|| / ||\mathbf{A}||}$$

is not too large.

Several theoretical treatments of the propagation of roundoff errors in the linear equation solution process have been published [Rals65] [Isaa66] [Fors67] [Wilk64]. Unfortunately, the upper bound on the rate of growth of pivot terms in the matrix is unrealistically large. The theory does indicate, however, that the use of pivoting keeps that upper bound small. A 'complete' pivoting strategy takes as pivot the element of maximum magnitude in the submatrix of remaining equations. Partial pivoting considers only elements in the same column or row. (The use of numerical pivoting to reorder the circuit equations is described in detail in Section 3.3.1.)

Iterative schemes, such as those of Jacobi and Gauss-Seidel [Isaa66], can be used for calculating the solution of a set of linear equations or for improving the accuracy of a previously derived solution. Three factors make these algorithms unsuitable for the high-speed solution of the equations which arise in integrated circuits. First, convergence of these iterative processes is assured only when the coefficient matrix has certain rather stringent properties such as positive definiteness. Second, although the roundoff error in an iterative method does not propagate (since each iteration 'starts over'

with the same coefficient matrix), the roundoff error can be a serious problem for a ill-conditioned system of equations [Rals65]. Finally, for a linear system the use of an iterative process to improve the accuracy of a solution obtained by direct methods is not desirable simply because of the increased computational effort required. It is preferable to obtain directly a sufficiently accurate solution.

The use of an augmented MNA coefficient matrix can improve the accuracy of intermediate terms calculated during a direct solution of the system of equations. This extension requires little additional memory or computational effort and extends the accuracy in pivot values to nearly that of double precision. A detailed description of this modification is presented in Section 3.3.2.

It is possible for the overall iteration process, of which the linear equation solution is just a part, not to converge due to the presence of numerical 'noise' in the direct solution [Frer76]. Iterative refinement of the solution may be used to circumvent this difficulty. The resulting two-stage iteration process combines 'absolute' and 'delta' iterations and is described in Section 3.3.3.

Several overall observations on the accuracy-enhancing methods previously mentioned are presented in Section 3.3.4.

3.3.1. Equation Reordering

One factor in the execution time of simulation programs is that the circuit equations are usually solved using 64-bit precision in floating-point calculations. This extended precision may require 2 to 3 times more references to words in memory than does the single-precision case, although on machines such as the UNIVAC 1108 [Univ70], memory access has been

optimized for double-word memory references. The extended precision, however, always requires more time in the arithmetic logic unit (ALU) to perform the actual calculation. The speed penalty for double precision compared to single-precision arithmetic is typically a factor of 2 to 3. This speed penalty applies even when the ALU always performs the calculation in double precision regardless of the particular machine instruction, such as the UNIVAC 1108 or the CYBER 175.

Extended precision is used both for inherent accuracy requirements and because the set of circuit equations is frequently ill-conditioned. The ill-conditioning problem can be resolved in at least two different ways. If the pivot terms of the coefficient matrix are not identically zero, an accurate solution can be obtained if a sufficiently large number of significant digits are carried along in the computation. From the middle 1960s through the 1970s the principal scientific computer at the Berkeley campus of the University of California was the CDC 6400, which has a wordsize of 60 bits (corresponding to 14 significant digits). As a result, the simulation programs developed, e.g. SPICE1, SPICE2, SINC [Fan75], and SLIC [Idle71], do not need to use double precision, since single-precision arithmetic on the CDC 6400 already provides sufficient accuracy. For the shorter-wordlength minicomputer, however, the use of extended precision as a way of obtaining the desired accuracy causes a significant increase in both cpu time and memory requirements.

A second way to resolve the ill-conditioning problem is to use methods which reorder the rows and/or columns in the coefficient matrix, either before or during the equation solution process. These reordering techniques are divided into two categories according to the type of information utilized.

Topological methods rely on information regarding the specific interconnection of individual circuit elements; these methods are applied once, before any actual solution is calculated. Numerical methods depend on the actual values in the coefficient matrix; these techniques apply once values have been determined for the matrix elements and may be utilized many times as the analysis proceeds.

A circuit example for which topological reordering is useful is shown in Figure 3.3 [Nage75]. For dc analysis the MNA equations are

$$\begin{bmatrix} \frac{1}{R2} & \frac{-1}{R2} & 0 & 1 \\ \frac{-1}{R2} & \frac{1}{R2}+\frac{1}{R4} & \frac{-1}{R4} & 0 \\ 0 & \frac{-1}{R4} & \frac{1}{R4} & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} V_1 \\ V_2 \\ V_3 \\ I(VS) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ VS \end{bmatrix} \tag{3.2}$$

If these equations as shown are solved by LU decomposition, at least one of the pivot (diagonal) entries becomes identically zero and the solution process aborts. However, if the first and last rows are exchanged, the set of equations becomes well-conditioned and all of the pivot entries remain nonzero during the solution process. The resulting system of equations is shown below:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ \frac{-1}{R2} & \frac{1}{R2}+\frac{1}{R4} & \frac{-1}{R4} & 0 \\ 0 & \frac{-1}{R4} & \frac{1}{R4} & 0 \\ \frac{1}{R2} & \frac{-1}{R2} & 0 & 1 \end{bmatrix} \times \begin{bmatrix} V_1 \\ V_2 \\ V_3 \\ I(VS) \end{bmatrix} = \begin{bmatrix} VS \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{3.3}$$

The row-exchange algorithm considers each voltage-defined branch in the circuit. If the nodes to which these branches are connected are all disjoint, the algorithm simply exchanges the current equation of each branch

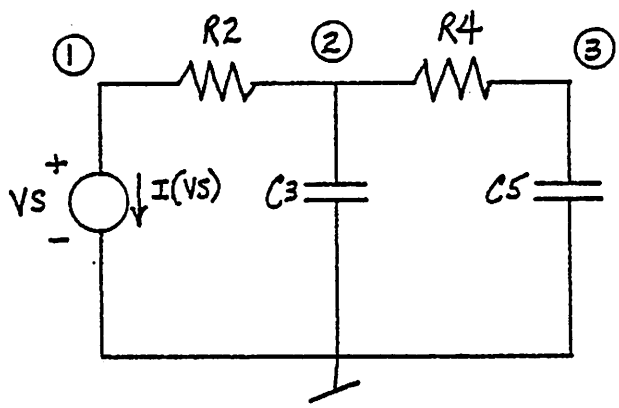


Figure 3.3. Circuit Example for Topological Reordering

with the row corresponding to the positive node of that branch. If some of the branch nodes are coincident then the algorithm orders the row-exchange process based on a mapping between the voltage-defined branches and circuit nodes which are the negative node of some voltage-defined branch. (This mapping can always be constructed unless the circuit contains a loop of voltage-defined branches [Nage75].)

Early versions (through Version F) of Program SPICE2 used only a topological reordering of the circuit equations to improve the conditioning of the system. No numerically-based reordering was used because of concern about the overhead of rebuilding the sparse-matrix pointer structure each time reordering was necessary. Also, the program was designed to run on the CDC 6400, which provided enough significant digits even with single-precision arithmetic to avoid most numerical difficulties.

It is not always straightforward to establish a valid mapping between circuit nodes and voltage-controlled branches which avoids the zero-valued pivot difficulty. The algorithm used in SPICE2 to establish that mapping is shown in Figure 3.4. The method 'walks' along any trees of voltage-defined branches and only swaps 'leaf' branches of the tree. However, the algorithm fails for the circuit in Figure 3.5. The initial set of MNA equations for this circuit are (where $Gx = \frac{1}{Rx}$):

$$\begin{bmatrix} G1 & -G1 & 0 & 0 & 0 & 0 \\ -G1 & G1 & 0 & 0 & 1 & 0 \\ 0 & 0 & G2 & -G2 & -1 & 0 \\ 0 & 0 & -G2 & G2 & 0 & 1 \\ 0 & 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} V_1 \\ V_2 \\ V_3 \\ V_4 \\ I(L1) \\ I(VS) \end{bmatrix} = \begin{bmatrix} -IS \\ 0 \\ 0 \\ 0 \\ 0 \\ VS \end{bmatrix} \qquad (3.4)$$

In the absence of any reordering, the (2,2) and (4,4) elements become identi-

```
while (there exists an unswapped voltage-defined branch (VDB)) {
  for (each unswapped VDB) {
       if (positive node of VDB != ground)
          if (number of VDBs at positive node < 2)
               if (diagonal term in equation for VDB current != 0)
                  save (+ node, VDB) for possible swap
               else
                  save (+ node, VDB) and exit 'for' loop
       if (negative node of VDB != ground)
          if (number of VDBs at negative node < 2)
               if (diagonal term in equation for VDB current != 0)
                  save (- node, VDB) for possible swap
               else
                  save (- node, VDB) and exit 'for' loop
  }
  if (any (node, branch) pair was saved)
       perform swap and mark branch 'swapped'
  else
       error  " loop of voltage-defined branches in circuit "
}
```

Figure 3.4. Mapping Algorithm for Row Swap

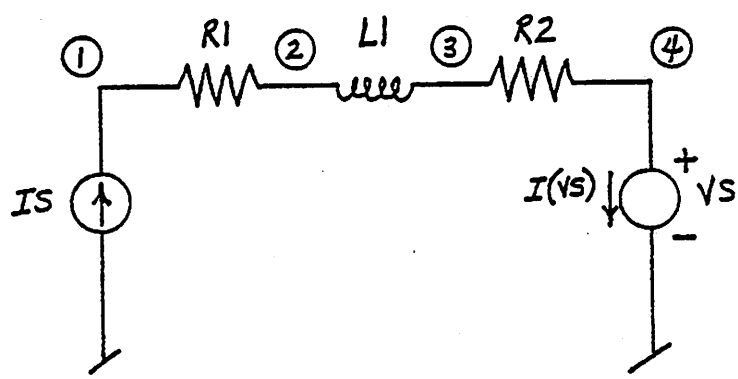


Figure 3.5. Circuit Example Showing Reordering Problem

cally zero during the LU decomposition[2]. The topologically-based reordering algorithm in SPICE2 fails for this circuit because it does not detect properly the fact that resistance R2 connects two voltage-defined branches. After applying the SPICE2 algorithm and a Markowitz reordering step to maintain sparsity, the coefficient matrix becomes:

$$\begin{vmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ G2 & 1 & 0 & 0 & 0 & -G2 \\ 0 & 0 & G1 & -G1 & 0 & 0 \\ 0 & 0 & -G1 & G1 & 1 & 0 \\ -G2 & 0 & 0 & 0 & -1 & G2 \\ 0 & 0 & 0 & 1 & 0 & -1 \end{vmatrix} \tag{3.5}$$

Even after this reordering, the (4,4) element of Equation (3.5) becomes zero during the LU decomposition. An improved algorithm has been reported recently [Hajj81] which partitions and orders the circuit variables and equations so that zero-valued pivots are always avoided and the occurrence of singular submatrices is prevented (assuming that the entire circuit matrix is nonsingular).

A reordering strategy based solely on topological considerations, however, is unable to resolve all the numerical difficulties which can arise. These difficulties are either due to the dynamically changing behavior of the circuit as the analysis proceeds, or due to topological problems which are very difficult to identify before analysis. An example of a dynamic problem is shown in the MOS sample-and-hold circuit of Figure 3.6. The voltage V_{in} is sampled and held at node (1). Unless independent row or column exchanges are made in the coefficient matrix, the magnitude of the pivot element (on the matrix diagonal) for any node in the circuit is equal to the sum of the magnitudes of the conductances connected to that node. During the sampling clock phase, the total conductance at node (1) is reasonably large.

[2]described in Chapter 5

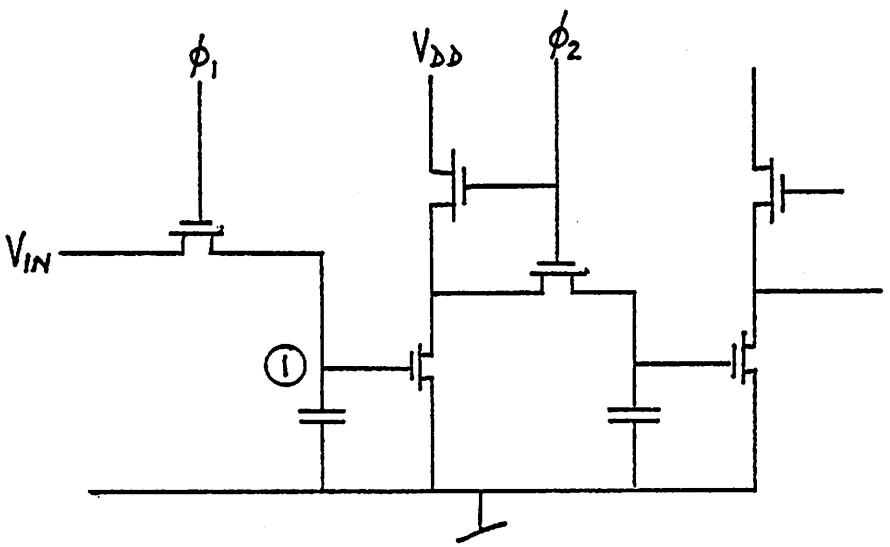


Figure 3.6. MOS Sample-and-Hold Circuit

During the hold phase, however, the conductance at the sampling node (and hence the magnitude of the pivot value) is determined by leakage terms which are nearly zero. In the absence of any numerical pivoting, a simulation program encountering such a circuit attempts to divide by an essentially zero-valued pivot and aborts. This difficulty exists regardless of the numerical precision used, but is especially aggravated for short-wordlength computations.

An example of the type of numerical problem which is topological in nature but very difficult to identify before analysis is illustrated by the cascaded gain stages shown in the circuit of Figure 3.7 [Frer76]. The submatrix for nodes 1-3 of this circuit is

$$\begin{bmatrix} \frac{1}{R}+g_\pi & 0 & g_m \\ g_m & \frac{1}{R}+g_\pi & 0 \\ 0 & g_m & \frac{1}{R}+g_\pi \end{bmatrix} \tag{3.6}$$

The nonzero off-diagonal entries in Equation (3.6) are equal to the g_m of the devices, while the diagonal entries are essentially the g_π values (which are smaller by a factor of β_F). Without pivoting, the diagonal entries are divided by powers of β_F during the LU decomposition with a significant loss of precision in the computation.

To resolve these numerical problems, SPUDS uses both a preliminary reordering of the circuit equations based solely on topological factors and then employs a form of numerical pivoting. Classical 'full' numerical pivoting [Isaa66] which always chooses that element of the remainder matrix with maximum magnitude, is not acceptable since it totally ignores any con-

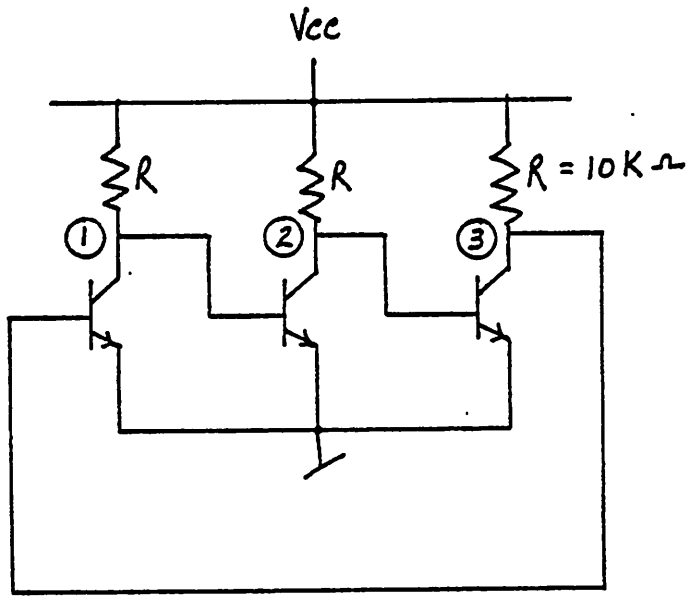


Figure 3.7. Cascaded Gain Stages

siderations of matrix sparsity[3]. Instead, SPUDS uses the strategy developed in Program MICE: the maximum value in the remainder matrix, MAXVAL, is determined and then all remaining elements with magnitude of at least PIVREL×MAXVAL are considered as potential pivot candidates. The default value of PIVREL is 10^{-3} in SPUDS (A default value of 10^{-6} is used in MICE; the larger value in SPUDS makes the system of linear equations more well-conditioned.) Diagonal matrix elements are considered first to minimize pivot search time, since those entries are usually large enough in an MNA matrix. Full pivoting (rather than partial pivoting) is performed since no increase in execution time is observed for systems of up to several hundred equations.

During the LU decomposition, elements in the matrix which are zero may become nonzero. These 'fill-in' terms can change greatly the sparsity of the matrix. The simplest strategy to minimize the number of fill-ins by reordering the elements in the matrix is due to Markowitz [Mark57]. The maximum number of fill-in terms that can be generated in a single step of the LU decomposition is equal to the product of the number of terms in the upper triangle row and lower triangle column. The Markowitz algorithm chooses the row-column pair with the minimum row-column product as the next pivot. Ties are resolved by choosing the row-column pair with the smallest number of column elements[4].

The use of this pivot strategy is always forced on the first iteration in both dc and transient analyses. Pivoting is forced at the beginning of tran-

---

[3]Freret's thesis describes the use of a full N×N matrix to store the equation coefficients. Matrix sparsity is mentioned only in relation to data structure modifications but not with respect to the effects of different pivoting strategies on matrix sparsity.

[4]The coding of the pivoting strategy in SPUDS is adapted from the implementation in Program SPICE2 by A. Vladimirescu and G. Boyle.

sient analysis because conductances for charge-storage elements, which can modify the numerical conditioning of the matrix, are not present for the dc analysis. Thereafter, pivoting is performed only if the magnitude of any value on the matrix diagonal becomes smaller than parameter PIVTOL, which has a default value of 10^{-13}, empirically chosen for 32-bit accuracy. Typically, only one pivoting step is performed during any particular analysis.

In many cases the use of just numerical pivoting to reorder the circuit equations is sufficient to avoid numerical problems. Neither Freret's simulator nor Program MICE, both of which use Nodal Analysis, use any topologically-based reordering scheme. The advantage of using topologically-based reordering is simply that most of the necessary row- and column-swaps in the modified-nodal matrix can be done before the actual equation solution is begun.

Some pivoting results are shown in Table 3.1. The two columns headed '#terms(SPUDS)' present, respectively, the number of nonzero matrix elements after all circuit elements are loaded and the number of such elements after LU decomposition and pivoting is performed for Program SPUDS. The next column to the right displays the number of nonzero matrix elements after solution for Version E.3 of Program SPICE2[5]. The last two columns show the number of matrix operations necessary to find the equation solution for the two programs. As shown in the table, the matrix sparsity obtained using both topological and numerical data (SPUDS) is very close to that obtained when only topological data is used (SPICE2). The number of matrix terms shown for SPUDS includes terms added as part of the implementation of the augmented MNA equations described in the next section. As a result, the

[5]which only performs the topologically-based reordering step. The present version of SPICE2, G.2, uses essentially the same numerical pivoting strategy as does SPUDS.

Circuit	#eqns	#terms(SPUDS)		#terms	#matrix ops	
		>load	>pivot	SPICE 2E	SPUDS	SPICE 2E
DIFPAIR	17	68	73	58	94	94
KTEST	9	36	66	41	188	80
RCA3040	33	149	187	142	304	275
UA709	44	218	301	267	643	653
UA727	62	306	400	356	746	796
UA733	25	131	156	136	290	293
UA741	52	262	342	299	640	666
RTLINV	13	42	46	35	53	53
TTLINV	29	108	137	111	211	211
TTL74	29	108	135	111	205	211
TTL74S	34	131	172	142	290	290
TTL74L	29	108	135	111	205	211
TTL9200	31	121	148	125	224	242
ECLGATE	39	153	195	160	306	312
MECLIII	51	211	294	241	521	498
SBDGATE	57	215	290	244	500	500
CCSOR	13	55	60	49	102	94
DCOSC	15	75	88	76	159	161
CFFLOP	15	68	74	61	102	102
STCRC	5	13	13	10	13	13
CHOKE	8	24	26	18	30	25
ECLINV	20	77	89	67	133	121
SCHMITT	19	77	90	73	140	140
ASTABLE	13	46	60	49	92	92
SATINV	8	27	27	21	30	28
DEPLINV	6	16	16	12	15	15
RATLOG	15	77	77	64	100	93
INVCHN	11	56	56	47	82	82
BOOTINV	10	42	43	35	52	49
MOSMEM	14	60	60	48	70	72
MOSAMP1	25	165	188	166	288	316
MOSAMP2	25	168	204	183	348	374

Table 3.1. Pivoting Effects on Matrix Sparsity

average number of terms used in SPUDS for all of the circuits is 19% greater than the number required for Version E.3 of SPICE2. However, the number of matrix operations, which relates directly to the total analysis time, is only 0.2% greater for SPUDS.

3.3.2. Augmented MNA

Numerical round-off problems can cause the coefficient matrix to be singular even when the circuit has a unique solution. Consider two conductances G1 and G2 connected as shown in Figure 3.8a. The submatrix for this circuit fragment is shown below:

$$\begin{bmatrix} G1 & -G1 & 0 \\ -G1 & G1+G2 & -G2 \\ 0 & -G2 & G2 \end{bmatrix} \tag{3.7}$$

Notice that the computed value of the (2,2) element should be equal to the sum of the two conductances. However, if $G1 \gg G2$ and the number of significant digits carried in the computation is not sufficiently large, the actual numerical value of the (2,2) element is just that of G1. (For example, if $G1=1.000\times10^4$, G2=1, and only 4 significant digits are used, then $(G1+G2)=1.000\times10^4$, not 1.0001×10^4.) Such a dynamic range of element values occurs commonly in an integrated circuit. Consider for example the resistive load with an 'off' driver transistor of Figure 3.8b. The 'off' transistor is modelled with a conductance (G2) which is very small compared with the conductance of the resistive load (G1). As a result, the submatrix effectively becomes

$$\begin{bmatrix} G1 & -G1 & 0 \\ -G1 & G1 & -G2 \\ 0 & -G2 & G2 \end{bmatrix} \tag{3.8}$$

The (2,2) pivot element becomes zero during the LU decomposition and the equation solution aborts.

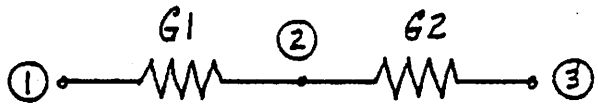


Figure 3.8a Two Conductances

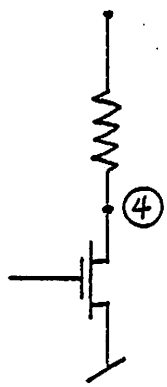


Figure 3.8b. Resistive Load

The indefinite admittance matrix (IAM) [Deso69] is obtained by adding a row and column for the ground node to the reduced admittance matrix[6]. This matrix Y has the invariant property that

$$\sum_i Y_{ij} = \sum_j Y_{ij} = 0$$

The zero row-summation property of the IAM matrix is utilized to determine with greater accuracy the true value of a pivot value by calculating

PIVOTVAL = $-\sum$(off-diagonals in the same row)

with the sum accumulated in double precision (64 bits) even though the coefficient matrix is stored in single precision (32 bits). This IAM formulation is used both in the work of Freret and in Program MICE.

Freret's dissertation presents examples in which algebraic cancellation in computing matrix elements can lead to an incorrect solution to the circuit equations. To alleviate this difficulty he uses

MAX (PIVOTVAL, pivot entry in matrix)

as the pivot value in the decomposition step. This method is also used in Program MICE. No significant difference in analysis results are found in Program SPUDS with this modification. The variable PIVOTVAL is used as the pivot value in the LU decomposition step.

For the modified-nodal-admittance matrix, the above technique must be modified slightly. The MNA matrix may be partitioned as:

$$\begin{bmatrix} Y & B \\ C & D \end{bmatrix} \times \begin{bmatrix} V_n \\ I_b \end{bmatrix} = \begin{bmatrix} J \\ E \end{bmatrix}$$

where V_n is the vector of node voltages and I_b is the vector of voltage-defined branch currents. To preserve the zero row-summation property in the MNA

[6]In practice only the column corresponding to the ground equation is added to the matrix; no additional numerical information is obtained from the addition of an extra equation (row).

matrix, terms which arithmetically cancel the nonzero entries in the B, C, and D submatrices are added to the column representing the ground node.

3.3.3. Absolute and Delta Iteration

The process of convergence to a solution can present special difficulties with short wordlength computations because the error in node voltages may be amplified by an exponential device characteristic. The symptoms of this problem are a rapid convergence to near the correct solution followed by numerical oscillation about that solution. To prevent these numerical difficulties from affecting the convergence process, a combination of 'absolute' and 'delta' (or 'incremental') iteration is implemented in SPUDS.

In 'absolute' iteration, SPUDS solves directly for the total node voltages V^{n+1} at iteration n+1 in terms of the linearized equivalent circuit determined at iteration n.

In 'delta' iteration, the system of circuit equations is modified to solve for the incremental change in node voltages between iterations. Figures 3.9a and 3.9b compare the two forms of iteration for the case of a simple diode. As shown in the figures, the value of the incremental conductance, g, loaded into the coefficient matrix is the same for both types of iteration. The equivalent current changes, however. For a given branch current I and branch voltage V, the equivalent current for 'absolute' iteration is

$$I_{eq} = I - g \times V$$

and the equivalent current for 'delta' iteration is

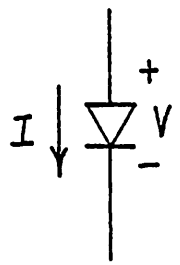
$$I_{eq} = I.$$

Once the solution δV^{n+1} at iteration n+1 has been determined, the new node voltages are calculated from

$$I = I_s \times e^{V/V_t}$$

$$= g \times V + I_{eq}$$

$$I_{eq} = I - g \times V$$



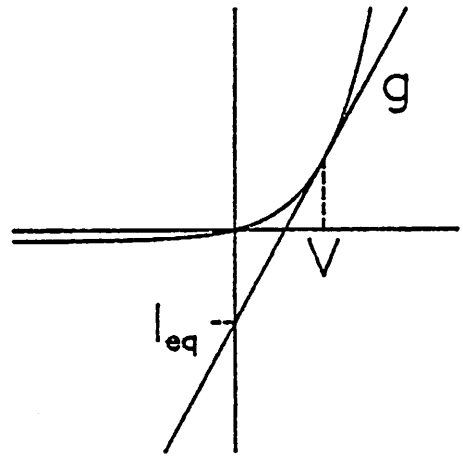


Figure 3.9a Absolute Iteration

$$V^{n+1} = V^n + \delta V^{n+1}$$

$$I = I_s \times e^{(V+\delta V)/V_t}$$

$$= g \times (V+\delta V) + I_{eq}$$

$$I_{eq} = I$$

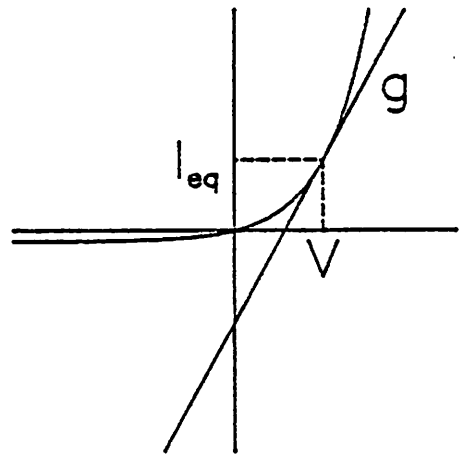


Figure 3.9b. Delta Iteration

$$V^{n+1} = V^n + \delta V^{n+1}$$

The necessity for 'delta' iteration is also supported by the data in Table 3.2, which shows three sets of node voltages found for the dc operating point of the UA741 operational amplifier test circuit[7]. The first two columns show results from Program SPUDS. Both analyses use the 'augmented' MNA matrix and numerical pivoting, but only the analysis for the second column utilizes 'delta' iteration. The third column shows results from Version E.3 of SPICE2 running on the CDC 6400 computer; the last column displays the voltage difference, in millivolts, between the second and third columns. As shown in the table, there is a marked improvement in the accuracy of the solution for some of the nodes when 'delta' iteration is used although for most of the circuit nodes the additional iteration step makes no difference.

To extract the greatest numerical advantage from the 'delta' iteration, two node voltage vectors are needed: a solution vector in single-precision and a 'reference' vector in double-precision. During 'absolute' iteration, the contents of the two vectors are the same (the single-precision vector is copied into the double-precision storage). During 'delta' iteration, all the significant digits available in the single-precision format are used to represent the incremental voltage change, which is then added to the double-precision reference voltage[8].

SPUDS always begins by using 'absolute' iteration; the change to 'delta' iteration is made when no junction-limiting methods are necessary to constrain the per-iteration change in nonlinear device operating points. A delay in switching to 'delta' iteration assures that the maximum benefit from the

[7]described in Appendix 3

[8]The implementation in SPUDS uses only one single-precision (32-bit) node voltage vector to store the linear equation solution, since the use of a double-precision reference vector made no significant difference in the circuit solution.

UA741 Operational Amplifier				
Node	SPUDS		SPICE 2E.3	error (mV)
	DVTHRS=0.0	DVTHRS=1.0		
1	-.0001	-.0001	-.0001	0.00
2	.0008	.0004	.0004	0.00
3	14.4445	14.4446	14.4446	0.00
4	-.5355	-.5356	-.5356	0.00
5	-.5352	-.5353	-.5353	0.00
6	-1.0699	-1.0700	-1.0700	0.00
7	-13.9212	-13.9211	-13.9210	0.10
8	-13.7729	-13.7728	-13.7728	0.00
9	-14.4484	-14.4483	-14.4483	0.00
10	-14.9882	-14.9881	-14.9881	0.00
11	-14.9881	-14.9881	-14.9881	0.00
12	-14.9103	-14.9103	-14.9103	0.00
13	-15.0000	-15.0000	-15.0000	0.00
14	-14.3167	-14.3166	-14.3166	0.00
15	-14.3523	-14.3523	-14.3522	0.10
17	14.3563	14.3563	14.3563	0.00
18	-14.9601	-14.9601	-14.9601	0.00
20	-.5059	-.5517	-.5493	2.40
21	.1425	.0968	.0991	-2.30
22	.7025	.6567	.6591	-2.40
23	.1030	.0573	.0596	-2.30
24	.0955	.0498	.0521	-2.30
25	.0917	.0460	.0484	-2.40
26	-15.0000	-15.0000	-15.0000	0.00
27	15.0000	15.0000	15.0000	0.00
30	0.0000	0.0000	0.0000	0.00

Table 3.2. Effects of Delta Iteration on UA741 Node Voltages

junction-limiting methods is obtained. The point at which the change to 'delta' iteration is made is also controlled by the threshold parameter DVTHRS by requiring that

$$MAX(V^n - V^{n-1}) < DVTHRS$$

for all nodes before changing to 'delta' iteration. The default value for DVTHRS is 1 volt.

3.3.4. Convergence Comparisons

The effects on convergence of the different accuracy-enhancing techniques are presented in this section. When the iterative solution process does converge, the simulation results using these methods agree well with each other and with the results obtained using Version E.3 of Program SPICE2 running on the 60-bit wordsize CDC 6400 computer. The results are identical to three significant digits and frequently identical even to four places. As a result, the emphasis of the data presented in this section is on factors associated with whether or not convergence was obtained.

The use of numerical pivoting is essential for convergence in the analysis of most integrated circuits. Table 3.3 compares the number of iterations necessary to perform dc transfer curve and dc operating point analyses for five different combinations of floating-point precision and the use of numerical pivoting, an augmented MNA matrix, and 'delta' iteration. The columns headed 'NOGO' show for each combination whether or not the analyses terminated without errors. A nonzero value marks a failure to converge in some analysis. Not all circuits require all analyses; such an omission is indicated when all five iteration counts are zero. A nonzero value of 'NOGO' when the dc operating point analysis does converge indicates that a subsequent transient analysis failed to converge.

Circuit	NOGO					dc transfer curves					dc operating point				
Case	#1	#2	#3	#4	#5	#1	#2	#3	#4	#5	#1	#2	#3	#4	#5
DIFPAIR	0	1	0	0	0	229	229	228	227	227	16	16	16	16	16
KTEST	0	0	0	0	0	0	0	0	0	0	4	4	4	4	4
RCA3040	0	1	1	0	0	239	238	238	238	238	16	16	16	16	16
UA709	0	1	0	0	0	301	342	308	297	297	26	26	26	26	26
UA727	0	1	0	0	0	223	555	375	217	217	26	0	26	26	26
UA733	0	0	0	0	0	0	0	0	0	0	8	8	8	8	8
UA741	0	1	0	0	0	303	306	351	273	272	26	0	26	26	26
RTLINV	0	1	0	0	0	240	240	235	235	235	10	10	10	10	10
TTLINV	0	1	1	0	0	269	167	541	258	258	15	0	19	15	15
TTL74	0	1	0	0	0	277	100	509	269	269	18	0	15	14	14
TTL74S	0	1	1	0	0	265	100	177	265	265	17	0	0	17	17
TTL74L	0	1	1	0	0	300	146	220	314	314	21	0	0	20	20
TTL9200	1	1	1	0	0	301	100	222	348	348	0	0	0	13	13
ECLGATE	0	1	1	0	0	227	227	227	227	227	8	8	8	8	8
MECLIII	0	1	1	0	0	243	245	240	241	241	13	13	13	13	13
SBDGATE	0	1	1	0	0	271	100	514	297	329	21	0	0	20	20
CCSOR	0	0	0	0	0	0	0	0	0	0	13	13	13	13	13
DCOSC	0	0	0	0	0	0	0	0	0	0	13	13	13	13	13
CFFLOP	0	0	0	0	0	0	0	0	0	0	7	8	8	8	8
STCRC	0	0	0	0	0	0	0	0	0	0	2	2	2	2	2
CHOKE	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
ECLINV	0	1	1	0	0	0	0	0	0	0	8	8	8	8	8
SCHMITT	0	1	0	0	0	0	0	0	0	0	8	8	7	7	7
ASTABLE	0	0	1	0	0	0	0	0	0	0	10	10	10	10	10
SATINV	0	0	0	0	0	689	606	643	643	643	51	6	30	30	30
DEPLINV	0	0	0	0	0	223	202	214	214	214	11	2	10	10	10
RATLOG	0	1	0	0	0	0	0	0	0	0	8	2	8	8	8
INVCHN	0	0	0	0	0	0	0	0	0	0	13	2	13	13	13
BOOTINV	0	0	0	0	0	0	0	0	0	0	12	4	12	12	12
MOSMEM	0	1	0	0	0	0	0	0	0	0	8	2	8	8	8
MOSAMP1	0	0	1	0	0	262	202	6	274	271	24	2	0	24	24
MOSAMP2	0	1	0	0	0	0	0	0	0	0	90	2	66	66	66

Case	precision (bits)	numerical pivoting	augmented MNA	'delta' iteration
#1	64	no	no	no
#2	32	no	no	no
#3	32	yes	no	no
#4	32	yes	yes	no
#5	32	yes	yes	yes

Table 3.3. Pivoting Convergence Comparisons

The first case in Table 3.3 shows the use of 64-bit floating-point precision without the use of numerical pivoting, the augmented MNA matrix, or 'delta' iteration. No convergence difficulties are present, with the exception of the TTL9200 circuit (for which convergence is particularly difficult even with 60-bit precision [Nage75]). More than half of the benchmark circuits (and all of the TTL ones) do not converge for the second case, which uses 32-bit floating-point precision but is otherwise the same as Case #1. The combination of 32-bit floating-point precision and numerical pivoting, shown in Case #3, is slightly more effective, especially for linear circuits. The UA741 and MOSAMP2 circuits are simulated without difficulty, but the TTL circuits still fail to converge. Case #4 adds the use of the augmented MNA matrix to Case #3; as shown in the table, all the circuits converge without difficulty. The addition of 'delta' iteration, in Case #5, does not make any noticeable improvement in convergence when compared with Case #4. The only significant change is a 10% increase in the number of iterations for the dc transfer curve analysis of the SBDGATE circuit. The analysis results for the Cases #4 and #5 differ in the fourth significant digit by at most one, and both sets of results agree to 3.5 significant digits with those of Version E.3 of SPICE2 running on the CDC 6400 with 60-bit precision.

Table 3.4 compares simulation costs (in terms of iteration counts) when 32-bit precision, numerical pivoting, and 'delta' iteration are used. The only variable in the data is the use (or non-use) of the augmented MNA matrix. The first 4 columns of each set of data show, respectively, the value of the 'NOGO' flag and the total number of iterations for the dc transfer curve, dc operating point, and transient analyses. The remaining two columns show the total number of timepoints and the number of rejected timepoints for the transient analysis.

Circuit	'standard' MNA						augmented MNA					
	N	XCI	DCI	TRI	TTP	RTP	N	XCI	DCI	TRI	TTP	RTP
DIFPAIR	0	228	16	243	107	0	0	227	16	239	107	0
KTEST	0	0	4	394	197	31	0	0	4	636	318	82
RCA3040	1	238	16	101	12	9	0	238	16	314	107	0
UA709	0	308	26	249	107	0	0	297	26	248	107	0
UA727	0	375	26	247	107	0	0	217	26	241	107	0
UA733	0	0	8	0	0	0	0	0	8	0	0	0
UA741	0	351	26	214	107	0	0	272	26	214	107	0
RTLINV	0	235	10	418	141	9	0	235	10	394	136	8
TTLINV	1	372	0	0	0	0	0	258	15	594	168	19
TTL74	0	530	15	649	187	25	0	269	14	658	186	22
TTL74S	1	177	0	0	0	0	0	265	17	523	152	10
TTL74L	1	220	0	0	0	0	0	314	20	547	164	15
TTL9200	1	222	0	0	0	0	0	348	13	623	173	20
ECLGATE	1	227	8	140	19	11	0	227	8	492	153	12
MECLIII	1	240	13	151	19	11	0	241	13	480	143	9
SBDGATE	1	1349	0	0	0	0	0	329	20	481	145	8
CCSOR	0	0	13	0	0	0	0	0	13	0	0	0
DCOSC	0	0	13	0	0	0	0	0	13	0	0	0
CFFLOP	0	0	8	0	0	0	0	0	8	0	0	0
STCRC	0	0	2	236	118	1	0	0	2	236	118	1
CHOKE	0	0	0	286	112	1	0	0	0	266	109	0
ECLINV	1	0	8	124	15	10	0	0	8	434	153	15
SCHMITT	0	0	7	358	148	9	0	0	7	366	153	10
ASTABLE	0	0	10	1000	259	38	0	0	10	909	219	28
SATINV	0	643	30	0	0	0	0	643	30	0	0	0
DEPLINV	0	214	10	0	0	0	0	214	10	0	0	0
RATLOG	0	0	8	413	180	7	0	0	8	413	180	7
INVCHN	0	0	13	0	0	0	0	0	13	0	0	0
BOOTINV	0	0	12	238	119	0	0	0	12	238	119	0
MOSMEM	0	0	8	335	164	9	0	0	8	335	164	9
MOSAMP1	1	6	0	0	0	0	0	271	24	0	0	0
MOSAMP2	0	0	66	486	178	20	0	0	66	503	187	24

Table 3.4. Augmented MNA Convergence Comparisons

The use of the augmented MNA matrix has most effect in the simulation of the TTL circuits, which do not converge, or converge very slowly, unless this extension to the matrix is used. The need for the augmented matrix can be explained readily by the fact that the TTL circuits typically have a greater range in magnitude of branch conductances than do the other circuits. The additional column in the matrix for these circuits improves the solution accuracy when such a spread in values occurs.

A comparison of iteration counts for dc transfer curve and dc operating point analyses, in which the only variable is the use of 'delta' iteration, is shown in Table 3.5. Both analyses are performed using 32-bit precision, numerical pivoting, and the augmented MNA matrix. As shown in the table, no significant differences in iteration count are observed whether or not 'delta' iteration is used.

3.4. Convergence-Enhancing Methods

The likelihood and rate of convergence to a solution can be improved by incorporating several control parameters into the iteration process. These control parameters are of even greater importance when limited numerical precision is used in the calculations. Some of these parameters constrain the per-iteration change in selected circuit unknowns; other parameters determine certain threshold levels. Each of these parameters is described in the following subsections. Of these methods, only prediction and bypass (Section 3.4.5) are used in Program SPICE2.

3.4.1. *p-n* Junction Conductance Limit

The assumed value of *p-n* junction bias can be much greater during the iteration process than the final solution value. As a result, the equivalent

Circuit	# iterations			
	DC Xfer		DCOP	
	with δ	without δ	with δ	without δ
DIFPAIR	227	227	16	16
KTEST	0	0	4	4
RCA3040	238	238	16	16
UA709	297	297	26	26
UA727	217	217	26	26
UA733	0	0	8	8
UA741	272	273	26	26
RTLINV	235	235	10	10
TTLINV	258	258	15	15
TTL74	269	269	14	14
TTL74S	265	265	17	17
TTL74L	314	314	20	20
TTL9200	348	348	13	13
ECLGATE	227	227	8	8
MECLIII	241	241	13	13
SBDGATE	329	297	20	20
CCSOR	0	0	13	13
DCOSC	0	0	13	13
CFFLOP	0	0	8	8
STCRC	0	0	2	2
CHOKE	0	0	0	0
ECLINV	0	0	8	8
SCHMITT	0	0	7	7
ASTABLE	0	0	10	10
SATINV	643	643	30	30
DEPLINV	214	214	10	10
RATLOG	0	0	8	8
INVCHN	0	0	13	13
BOOTINV	0	0	12	12
MOSMEM	0	0	8	8
MOSAMP1	271	274	24	24
MOSAMP2	0	0	66	66

Table 3.5. 'Delta' (δ) Iteration Comparison

conductance loaded into the coefficient matrix to model the junction can be much too large, accentuating roundoff errors and slowing the rate of convergence to a solution. Limiting the exponential characteristic is important for another reason as well. The internal representation for 32-bit floating-point variables typically uses 7 bits for the exponent field [HP78F] [PR1ME]; the largest value which can be represented is therefore $\approx 2^{128}$ or 3×10^{38}. This maximum value can be exceeded easily during the iteration process due to the presence of exponential functions. Without limiting several analog and digital circuits, such as the UA709 and ECLGATE benchmarks, fail to converge on the minicomputer due to numerical overflow.

In Program MICE, this difficulty is overcome by defining a maximum value for the *p-n* junction conductance, GDMAX, with a default value of 10mhos ($\frac{1}{10}$ ohms). For each device at each iteration, the junction voltage VMAX corresponding to this conductance is determined from

$$\mathrm{VMAX} = V_t \times \ln(\mathrm{GDMAX} \times \frac{V_t}{I_{sat}})$$

where V_t is the thermal voltage and I_{sat} is the junction saturation current. If the assumed junction voltage V is less than VMAX, the equivalent linear circuit which models the junction is computed using the usual exponential characteristic. If V is greater than VMAX, MICE models the junction characteristic using a linear extrapolation tangent to the exponential curve at VMAX with slope GDMAX.

In the SPUDS program a different implementation of this exponential-limiting idea is used to reduce the computational effort required. Rather than limiting the junction conductance, the assumed junction voltage is constrained to be less than or equal to the parameter PNVMAX. The default value of PNVMAX is chosen to be approximately equivalent to the default

GDMAX value of MICE. Since the equivalent conductance g is determined from

$$g = \partial I_d / \partial V_d$$
$$= \partial [I_{sat} \times (e^{V/V_t} - 1)] / \partial V_d$$
$$= I_d / V_t$$

we have

$$I_d = g \times V_t$$

Therefore, the current at which the equivalent conductance is 10mhos is approximately 260mA. If the saturation current I_{sat} is taken as 10^{-14} amps then

$$PNVMAX = 0.026 \times \ln(10 \times 0.026 / 10^{-14})$$
$$= 0.8V$$

The default value for this parameter in SPUDS is 0.9V to allow for variations in saturation current and circuit temperature.

This limiting scheme prevents premature abortion of the iteration process due to overflow during the evaluation of *p-n* junction exponentials. The use of this form of limiting does not change the number of iterations required for convergence.

3.4.2. Node-Voltage Limiting

The values of the circuit unknowns (node voltages and voltage-defined branch currents) can become very large ($\gg 10^6$) during the iteration process. For sufficiently large values of these unknowns, the intermediate computations (such as device model evaluation) can fail due to numerical overflow.

One method which avoids this numerical problem is to limit the magnitudes of the node voltages to an appropriate value, such as twice the magnitude of the greatest independent voltage source in the circuit. (Circuits

which exhibit "bootstrapping" can have transient node voltages which exceed the magnitudes of the independent power supplies.)

A comparison of the effects of node-voltage limiting is shown in Table 3.8 which displays four sets of data, corresponding to two different limiting values for node voltages and for *p-n* junctions. Each set of three columns of data shows the value of the 'NOGO' flag and the total number of iterations for the dc transfer curve and dc operating point analyses. All the analyses use the 'augmented' MNA matrix and numerical pivoting.

For the majority of the benchmark circuits the rate of convergence to a solution is not changed significantly by the use of node-voltage limiting, regardless of whether or not *p-n* junction voltages are also limited. The analyses for both the KTEST and CHOKE circuits fail to converge because the solutions for these two circuits have node voltages on the order of 200 volts.

Considerable convergence problems have been observed with the other circuits for which convergence is not obtained [Nage75]. For the TTL circuits TTL74L and TTL9200, the dc operating-point analyses do not converge when the node voltages are constrained to be less than even 50 volts, although the value of the largest independent voltage source in these circuits is +5 volts. The dc operating point analysis of the MOS amplifier circuit MOSAMP2 does not converge with 50-volt limiting, even though the largest independent voltage source magnitude is 20 volts. The limiting of circuit unknown values reduces the effectiveness of the *p-n* junction updating algorithms, leading to nonconvergence for these circuits.

3.4.3. *p-n* Junction Voltage Thresholding

One reason for difficulty in converging to a solution is the granularity of node voltage values. The minimum change in any floating-point number is

Circuit	PNVMAX=0.9						PNVMAX=100					
	|VLIM| = 20			|VLIM| = 50			|VLIM| = 20			|VLIM| = 50		
	N	XCI	DCI	N	XCI	DCI	N	XCI	DCI	N	XCI	DCI
DIFPAIR	0	228	16	0	227	16	0	228	16	0	227	16
KTEST	1	0	4	1	0	4	1	0	4	1	0	4
RCA3040	0	238	16	0	236	16	0	243	16	0	236	16
UA709	0	343	26	0	321	26	0	320	26	0	308	26
UA727	0	217	26	0	215	26	0	217	26	0	217	26
UA733	0	0	8	0	0	8	0	0	8	0	0	8
UA741	0	267	26	0	271	26	0	267	26	0	273	26
RTLINV	0	235	10	0	235	10	0	235	10	0	235	10
TTLINV	0	259	16	0	259	16	0	259	16	0	259	16
TTL74	0	274	16	0	274	19	0	274	16	0	274	19
TTL74S	0	267	18	0	265	15	0	267	18	0	265	15
TTL74L	1	305	100	1	100	0	1	305	100	1	100	0
TTL9200	1	100	0	1	100	0	1	100	0	1	100	0
ECLGATE	0	227	8	0	227	8	0	227	8	0	227	8
MECLIII	0	244	13	0	244	13	0	244	13	0	244	13
SBDGATE	0	329	20	0	329	20	0	329	20	0	329	20
CCSOR	0	0	13	0	0	13	0	0	13	0	0	13
DCOSC	0	0	13	0	0	13	0	0	13	0	0	13
CFFLOP	0	0	8	0	0	8	0	0	8	0	0	8
STCRC	0	0	2	0	0	2	0	0	2	0	0	2
CHOKE	1	0	0	1	0	0	1	0	0	1	0	0
ECLINV	0	0	8	0	0	8	0	0	8	0	0	8
SCHMITT	0	0	7	0	0	7	0	0	7	0	0	7
ASTABLE	0	0	9	0	0	9	0	0	9	0	0	9
SATINV	0	643	30	0	643	30	0	643	30	0	643	30
DEPLINV	0	214	10	0	214	10	0	214	10	0	214	10
RATLOG	0	0	8	0	0	8	0	0	8	0	0	8
INVCHN	0	0	13	0	0	9	0	0	13	0	0	9
BOOTINV	0	0	12	0	0	12	0	0	12	0	0	12
MOSMEM	0	0	8	0	0	8	0	0	8	0	0	8
MOSAMP1	0	265	25	0	351	25	0	265	25	0	269	25
MOSAMP2	1	0	100	1	0	100	1	0	100	1	0	100

Table 3.6. Convergence Comparison: Voltage Limiting

approximately one place in the least significant digit. If 64 bits of precision are utilized, more than 14 significant digits are stored and the minimum representable change is less than one part in 10^{14}. When only 32-bit precision is used in the calculations, however, approximately 6.5 significant digits are represented; for a node voltage of one volt, this granularity is approximately 0.5μV.

A junction voltage is computed as the difference between two node voltages. The numerical uncertainty in the calculation is therefore on the order of 1μV. If the simulation program attempts to make the junction voltages converge to within a tolerance less than 1μV, the iteration process frequently does not terminate. This problem is especially true for TTL circuits, which have great variation in the magnitudes of the coefficient matrix terms. This variation increases the actual change in node voltages between iterations because of increased roundoff errors.

This difficulty is overcome in Program SPUDS with the parameter DVPNJN. Changes in junction voltages less than DVPNJN in magnitude are considered as zero for purposes of convergence checking. The default value for this parameter is 0.1μV; for several of the TTL benchmark circuits, convergence is obtained only when DVPNJN is increased to 1–3μV. In all cases, the simulation results from the dc transfer curve, dc operating point, and transient analyses agree to ≈3.5 significant digits with the results obtained by running Version E.3 of Program SPICE2 on the 60-bit CDC 6400 computer.

3.4.4. 'Delta' Iteration Threshold

The decision to switch from 'absolute' iteration to 'delta' iteration is made when no junction-limiting methods are necessary to constrain the per-iteration change in nonlinear device operating points and when the maximum

per-iteration change in any node voltage is less than DVTHRS. Table 3.7 shows the effects of different values of DVTHRS. For each value, the value of the 'NOGO' flag and the number of iterations for the dc transfer curve, dc operating point, and transient analyses are given. The data in the columns headed 'DVTHRS=0.0V' are for analyses which do not use 'delta' iteration at all.

Since the switch to 'delta' iteration is not made until junction-limiting methods are not necessary, the effect of different values for the DVTHRS parameter is relatively small. As shown in the table, a too-large or too-small value can increase the number of iterations required for convergence. Empirically, the best value for DVTHRS is found to be 1V, and this number is the default in SPUDS.

3.4.5. Prediction and Bypass

Algorithms which predict the solution to the next point in a multi-point analysis and which bypass the evaluation of the nonlinear-device models can reduce the computational effort to perform an analysis considerably [Newt77]. Both of these methods also impact significantly the number of iterations required for convergence. The following paragraphs describe the convergence effects in SPUDS of each method.

For multiple-point analyses, the solution to the 'next' point can be predicted from the solution to previous points. SPUDS uses the same first-order predictor as Program SPICE2. For example in transient analysis this predictor takes the form

$$X_{n+1} = X_n + \frac{h_{n+1}}{h_n}(X_n - X_{n-1})$$

Table 3.8a shows the number of iterations and required cpu execution times

Circuit	DVTHRS=100V				DVTHRS=1.0V				DVTHRS=0.1V				DVTHRS=0.0V			
	N	XCI	DC	TRI	N	XCI	DC	TRI	N	XCI	DC	TRI	N	XCI	DC	TRI
DIFPAIR	0	227	16	238	0	227	16	238	0	227	16	238	0	227	16	239
KTEST	0	0	4	560	0	0	4	560	0	0	4	560	0	0	4	636
RCA3040	0	238	16	317	0	238	16	317	0	238	16	317	0	238	16	314
UA709	0	305	26	249	0	297	26	249	0	325	26	249	0	297	26	248
UA727	0	217	26	242	0	217	26	242	0	217	26	242	0	217	26	241
UA733	0	0	8	0	0	0	8	0	0	0	8	0	0	0	8	0
UA741	0	272	26	214	0	272	26	214	0	272	26	214	0	273	26	214
RTLINV	0	235	10	364	0	235	10	364	0	235	10	364	0	235	10	394
TTLINV	0	258	15	525	0	258	15	536	0	258	15	536	0	258	15	594
TTL74	0	269	14	627	0	269	14	627	0	269	14	627	0	269	14	658
TTL74S	0	265	17	508	0	265	17	508	0	265	17	511	0	265	17	523
TTL74L	0	314	20	584	0	314	20	584	0	314	20	584	0	314	20	547
TTL9200	0	348	13	563	0	348	13	563	0	348	13	563	0	348	13	623
ECLGATE	0	227	8	490	0	227	8	490	0	227	8	490	0	227	8	492
MECLIII	0	241	13	460	0	241	13	460	0	241	13	460	0	241	13	480
SBDGATE	0	329	20	476	0	329	20	476	0	329	20	476	0	297	20	481
CCSOR	0	0	13	0	0	0	13	0	0	0	13	0	0	0	13	0
DCOSC	0	0	13	0	0	0	13	0	0	0	13	0	0	0	13	0
CFFLOP	0	0	8	0	0	0	8	0	0	0	8	0	0	0	8	0
STCRC	0	0	2	236	0	0	2	236	0	0	2	236	0	0	2	236
CHOKE	0	0	0	261	0	0	0	261	0	0	0	261	0	0	0	266
ECLINV	0	0	8	345	0	0	8	345	0	0	8	345	0	0	8	434
SCHMITT	0	0	7	385	0	0	7	385	0	0	7	385	0	0	7	366
ASTABLE	0	0	9	886	0	0	9	886	0	0	9	886	0	0	10	909
SATINV	0	651	37	0	0	643	30	0	0	643	30	0	0	643	30	0
DEPLINV	0	214	10	0	0	214	10	0	0	214	10	0	0	214	10	0
RATLOG	0	0	8	364	0	0	8	364	0	0	8	364	0	0	8	413
INVCHN	0	0	13	0	0	0	13	0	0	0	13	0	0	0	13	0
BOOTINV	0	0	20	238	0	0	12	238	0	0	12	238	0	0	12	238
MOSMEM	0	0	8	331	0	0	8	331	0	0	8	331	0	0	8	335
MOSAMP1	0	271	24	0	0	271	24	0	0	271	24	0	0	274	24	0
MOSAMP2	1	0	100	0	0	0	66	494	0	0	66	494	0	0	66	488

Table 3.7. Convergence Comparison: 'Delta' Threshold

Circuit	No BYPASS and No PREDCT				PREDCT but No BYPASS			
	XCI	TRI	XCI	TRAN	XCI	TRI	XCI	TRAN
DIFPAIR	320	321	10.67	14.86	228	242	8.30	12.17
KTEST	0	560	0.00	15.69	0	560	0.00	15.70
RCA3040	331	126	24.32	14.12	238	312	18.62	35.02
UA709	378	313	42.02	49.85	318	248	36.74	42.10
UA727	277	322	45.20	71.03	217	243	37.97	57.61
UA733	0	0	0.00	0.00	0	0	0.00	0.00
UA741	298	314	43.21	65.45	272	214	40.90	49.24
RTLINV	284	418	6.33	11.66	235	349	5.58	10.30
TTLINV	317	556	16.12	36.37	258	544	13.83	36.08
TTL74	318	521	16.16	34.29	270	525	14.33	34.82
TTL74S	338	503	20.62	40.43	265	509	17.08	41.23
TTL74L	389	557	19.32	36.36	339	560	17.43	36.84
TTL9200	397	572	21.35	40.23	348	542	19.33	38.80
ECLGATE	264	510	19.36	47.48	227	448	17.43	42.98
MECLIII	293	524	28.75	68.96	241	446	24.82	60.65
SBDGATE	343	506	35.41	65.98	321	484	33.99	64.13
CCSOR	0	0	0.00	0.00	0	0	0.00	0.00
DCOSC	0	0	0.00	0.00	0	0	0.00	0.00
CFFLOP	0	0	0.00	0.00	0	0	0.00	0.00
STCRC	0	236	0.00	3.90	0	236	0.00	3.86
CHOKE	0	303	0.00	7.00	0	261	0.00	6.48
ECLINV	0	481	0.00	23.42	0	799	0.00	34.34
SCHMITT	0	418	0.00	20.89	0	386	0.00	19.89
ASTABLE	0	920	0.00	26.72	0	917	0.00	26.83
SATINV	726	0	21.05	0.00	644	0	19.30	0.00
DEPLINV	225	0	6.39	0.00	214	0	6.19	0.00
RATLOG	0	397	0.00	39.67	0	364	0.00	37.39
INVCHN	0	0	0.00	0.00	0	0	0.00	0.00
BOOTINV	0	247	0.00	22.41	0	238	0.00	21.85
MOSMEM	0	373	0.00	61.20	0	325	0.00	54.87
MOSAMP1	310	0	71.21	0.00	304	0	70.48	0.00
MOSAMP2	0	1504	0.00	565.66	0	480	0.00	182.14

Table 3.8a. Effects of Prediction on Convergence

for dc transfer curve and transient analyses of the benchmark circuits. The columns headed 'No BYPASS and No PREDCT' display the results for 'reference' analyses using neither prediction nor bypass methods. The remaining columns display the analysis results when the first-order prediction method (but not bypass) is employed. All the analyses use 32-bit floating-point arithmetic, numerical pivoting, and the augmented MNA matrix.

More cpu time per iteration is required if the prediction method is used. As shown in the table, for the DIFPAIR circuit the time per iteration in the dc transfer curve analysis is 33.3ms without prediction and 36.4ms with prediction. The difference, a 9% increase in the cpu time per iteration, is the time required to compute the predicted circuit variables. But the total cpu execution time for this analysis decreased by 22% because of an even greater percentage reduction in the required number of iterations. This decrease in cpu time holds true for all the benchmark circuits in the dc transfer curve analysis, although the percentage savings in cpu execution time is as little as 1% for some circuits.

For transient analysis Table 3.8a shows a similar reduction in overall cpu time except for the RCA3040 and ECLINV circuits, for which both the total cpu time and the number of Newton iterations required for the analysis increase by 50% or more. Since the data presented in [Nage75] on the use of this first-order prediction method with 60-bit floating-point arithmetic does not show this anomalous behavior for these two circuits, this increase in analysis time probably is due to numerical roundoff error in the values of the predicted circuit variables.

The bypass algorithm saves computational effort by evaluating each nonlinear branch relation and its derivative only if the difference in the argu-

ments to that branch relation is significant [Nage75]. Table 3.8b compares the analysis results when the bypass method is used with the results of the same 'reference' analysis from the previous table. For the dc transfer curve analysis, the savings in cpu execution time for all the benchmark circuits is small (at most 9% for the MOSAMP1 circuit). For the TTL74S circuit, the transfer curve analysis fails to converge. For most of the benchmark circuits, the transient analysis results are of a similar nature. The reduction in cpu time at best is small, and for some circuits the analysis time increases slightly.

The combination of both the prediction and bypass algorithms results in analysis times which are smaller, for almost all the benchmark circuits, than when either method is used alone. Most of the reduction in cpu time is due to the prediction method; the computation necessary to determine whether to bypass the evaluation of a particular nonlinear-device model is approximately offset by the savings in not having to perform those evaluations. Table 3.8c compares the analysis results when both prediction and bypass are used against the results of the 'reference' analysis of the last two tables. On the average for all the benchmark circuits, for the dc transfer curve analysis, the two methods together reduce the iteration counts and cpu times by 11.4% and 12%, respectively. The iteration counts and cpu times for transient analysis are reduced by 11% and 31% respectively. (If the very-large (and singular) reductions in transient analysis times for the MOSAMP2 circuit are removed from the comparison, the decrease in transient iterations and cpu times is only 2% and 4.5%, respectively.)

On the average, the reduction in cpu execution time with both the prediction and bypass methods is of the same order as the savings described by

Circuit	No BYPASS and No PREDCT				BYPASS but No PREDCT			
	XCI	TRI	XCI	TRAN	XCI	TRI	XCI	TRAN
DIFPAIR	320	321	10.67	14.86	320	316	10.54	14.52
KTEST	0	560	0.00	15.69	0	560	0.00	15.69
RCA3040	331	126	24.32	14.12	331	339	24.18	37.67
UA709	378	313	42.02	49.85	386	314	42.49	48.29
UA727	277	322	45.20	71.03	277	320	43.82	68.18
UA733	0	0	0.00	0.00	0	0	0.00	0.00
UA741	298	314	43.21	65.45	300	313	43.16	61.69
RTLINV	284	418	6.33	11.66	283	434	6.35	12.00
TTLINV	317	556	16.12	36.37	317	696	16.20	43.68
TTL74	318	521	16.16	34.29	319	529	16.24	34.63
TTL74S	338	503	20.62	40.43	290	0	18.29	0.00
TTL74L	389	557	19.32	36.36	364	546	18.32	35.32
TTL9200	397	572	21.35	40.23	396	610	21.39	41.90
ECLGATE	264	510	19.36	47.48	264	506	19.09	46.92
MECLIII	293	524	28.75	68.96	293	575	28.30	73.49
SBDGATE	343	506	35.41	65.98	330	511	34.32	65.53
CCSOR	0	0	0.00	0.00	0	0	0.00	0.00
DCOSC	0	0	0.00	0.00	0	0	0.00	0.00
CFFLOP	0	0	0.00	0.00	0	0	0.00	0.00
STCRC	0	236	0.00	3.90	0	236	0.00	3.87
CHOKE	0	303	0.00	7.00	0	303	0.00	7.01
ECLINV	0	481	0.00	23.42	0	355	0.00	17.99
SCHMITT	0	418	0.00	20.89	0	419	0.00	20.56
ASTABLE	0	920	0.00	26.72	0	933	0.00	27.11
SATINV	726	0	21.05	0.00	725	0	21.11	0.00
DEPLINV	225	0	6.39	0.00	225	0	6.36	0.00
RATLOG	0	397	0.00	39.67	0	397	0.00	38.86
INVCHN	0	0	0.00	0.00	0	0	0.00	0.00
BOOTINV	0	247	0.00	22.41	0	247	0.00	22.06
MOSMEM	0	373	0.00	61.20	0	371	0.00	60.21
MOSAMP1	310	0	71.21	0.00	305	0	65.01	0.00
MOSAMP2	0	1504	0.00	565.66	0	998	0.00	358.78

Table 3.8b. Effects of Bypass on Convergence

Circuit	No BYPASS and No PREDCT				BYPASS and PREDCT			
	XCI	TRI	XCI	TRAN	XCI	TRI	XCI	TRAN
DIFPAIR	320	321	10.67	14.86	227	238	8.21	11.97
KTEST	0	560	0.00	15.69	0	560	0.00	15.68
RCA3040	331	126	24.32	14.12	238	317	18.56	34.88
UA709	378	313	42.02	49.85	297	249	34.45	41.11
UA727	277	322	45.20	71.03	217	242	37.02	56.93
UA733	0	0	0.00	0.00	0	0	0.00	0.00
UA741	298	314	43.21	65.45	272	214	40.87	48.46
RTLINV	284	418	6.33	11.66	235	364	5.60	10.66
TTLINV	317	556	16.12	36.37	258	536	13.84	35.19
TTL74	318	521	16.16	34.29	269	627	14.32	40.08
TTL74S	338	503	20.62	40.43	265	508	17.10	40.46
TTL74L	389	557	19.32	36.36	314	584	16.30	37.79
TTL9200	397	572	21.35	40.23	348	563	19.35	39.48
ECLGATE	264	510	19.36	47.48	227	490	17.12	46.13
MECLIII	293	524	28.75	68.96	241	460	24.52	60.73
SBDGATE	343	506	35.41	65.98	329	476	34.55	62.18
CCSOR	0	0	0.00	0.00	0	0	0.00	0.00
DCOSC	0	0	0.00	0.00	0	0	0.00	0.00
CFFLOP	0	0	0.00	0.00	0	0	0.00	0.00
STCRC	0	236	0.00	3.90	0	236	0.00	3.87
CHOKE	0	303	0.00	7.00	0	261	0.00	6.46
ECLINV	0	481	0.00	23.42	0	345	0.00	17.88
SCHMITT	0	418	0.00	20.89	0	385	0.00	19.74
ASTABLE	0	920	0.00	26.72	0	886	0.00	26.54
SATINV	726	0	21.05	0.00	643	0	18.86	0.00
DEPLINV	225	0	6.39	0.00	214	0	6.02	0.00
RATLOG	0	397	0.00	39.67	0	364	0.00	36.49
INVCHN	0	0	0.00	0.00	0	0	0.00	0.00
BOOTINV	0	247	0.00	22.41	0	238	0.00	21.03
MOSMEM	0	373	0.00	61.20	0	331	0.00	53.47
MOSAMP1	310	0	71.21	0.00	271	0	57.80	0.00
MOSAMP2	0	1504	0.00	565.66	0	494	0.00	178.49

Table 3.8c. Effects of Prediction and Bypass on Convergence

[Nage75] for Program SPICE2 with 60-bit floating-point arithmetic. For some of the test circuits, however, the percentage reduction for SPUDS (with 32-bit floating-point arithmetic) is 2-3 times greater. For example, transient analysis cpu time reductions of 12%, 12%, and 8% are reported in [Nage75] for the UA709, UA727, and ECLINV circuits; the savings in SPUDS are 17.5%, 20%, and 24%, respectively. Moreover, for the MOSAMP2 circuit, the transient analysis time for SPUDS is reduced by 68% when both prediction and bypass methods are employed. This major reduction definitely warrants the inclusion of these methods in a minicomputer-based simulator.

3.5. Conclusions: 32-bit vs 64-bit Arithmetic

As mentioned earlier in this chapter, when the iterative solution process does converge the results of running a simulation with SPUDS agree well with the results obtained from running Version E.3 of Program SPICE2 on the 60-bit wordsize CDC 6400 computer. The node voltages are identical to three significant digits and frequently identical even to four places. Overall convergence data are presented in Table 3.9. For each of the benchmark circuits, the number of equations is shown, followed by the number of iterations for the dc transfer curve, dc operating point, and transient analyses for SPUDS and Version E.3 of Program SPICE2.

Program SPUDS converges for all of these circuits, and the number of iterations required is comparable to the number required by SPICE2. The coupling of 32-bit floating-point arithmetic with numerical pivoting, an augmented MNA matrix, incremental refinement of the iterative solution, and the threshold parameters described in Section 3.4 results in an effective circuit simulation tool.

Circuit	#eqns	SPUDS (32-bit)			SPICE 2E (60-bit)		
		XCI	DC	TRI	XCI	DC	TRI
DIFPAIR	17	227	16	238	229	16	252
KTEST	9	0	4	560	0	4	246
RCA3040	33	238	16	317	239	16	296
UA709	44	297	26	249	301	26	303
UA727	62	217	26	242	223	26	312
UA733	25	0	8	0	0	8	0
UA741	52	272	26	214	303	26	269
RTLINV	13	235	10	364	240	10	376
TTLINV	29	258	15	536	269	15	599
TTL74	29	269	14	627	277	18	534
TTL74S	34	265	17	508	265	17	558
TTL74L	29	314	20	584	300	21	651
TTL9200	31	348	13	563	340	13	558
ECLGATE	39	227	8	490	227	8	428
MECLIII	51	241	13	460	243	13	389
SBDGATE	57	329	20	476	271	21	484
CCSOR	13	0	13	0	0	13	0
DCOSC	15	0	13	0	0	13	0
CFFLOP	15	0	8	0	0	7	0
STCRC	5	0	2	236	0	2	246
CHOKE	8	0	0	261	0	0	357
ECLINV	20	0	8	345	0	8	361
SCHMITT	19	0	7	385	0	8	384
ASTABLE	13	0	9	886	0	10	1011
SATINV	8	643	30	0	689	51	0
DEPLINV	6	214	10	0	223	11	0
RATLOG	15	0	8	364	0	8	441
INVCHN	11	0	13	0	0	13	0
BOOTINV	10	0	12	238	0	18	238
MOSMEM	14	0	8	331	0	8	334
MOSAMP1	25	271	24	0	262	18	0
MOSAMP2	25	0	66	494	0	90	463

Table 3.9. Convergence Comparison: 32/64 Bit Precision

CHAPTER 4

DATA STRUCTURES

The choice of data representations and structures has a major impact on circuit simulation speed. The design of these structures is motivated towards obtaining the maximum possible simulation speed since this research is concerned, in part, with the development of a dedicated, inexpensive desktop simulation tool capable of producing analysis results for the design engineer in a few minutes. A secondary goal is to minimize the amount of high-speed memory required for the analyses, both in order to minimize the cost of the overall system and to increase the size of circuit which can be simulated within a given address space.

The coefficient matrix for the system of linearized equations that is repetitively solved during the analyses is very sparse; typically, more than 85% of its entries are zero. The internal representation for this matrix is described in detail in Section 4.1.

The data structures which represent the input circuit description and analysis 'state' information required by SPUDS are a set of linked lists and 'tables' of contiguous memory, similar to the structures used in Program SPICE2. Significant savings in memory requirements are obtained by data restructuring, by some algorithmic changes, and by taking advantage of the smaller wordsize of the minicomputer. These improvements are described in Section 4.2.

Section 4.3 presents a breakdown of the total memory requirements of SPUDS based on the functional needs of the program. A comparison is made with the total main memory required by Version E.3 of Program SPICE2.

4.1. Matrix Structures

The coefficient matrix for the system of linearized circuit equations corresponding to an average IC is very sparse [Berr71] [Nage75]; typically, 85% to 90% of the matrix entries are zero. Significant savings in both computation time and storage requirements can be achieved if advantage is taken of this sparsity.

Two basic types of operations are performed on this coefficient matrix. First, rows and/or columns of the matrix are interchanged either as part of the numerical pivoting described in Chapter 3 or in order to maintain sparsity. Second, matrix entries are modified as the system of linear equations is solved. Hence, the data structure representing the coefficient matrix must facilitate both rapid changes in matrix structure and fast access to any particular coefficient in the matrix.

In Version E.3 of Program SPICE2, two different ways of representing the matrix are used. Initially, as the structure of the coefficient matrix is established by scanning over the input circuit description, the matrix is represented in memory using an array of singly-linked lists, each of which stored the nonzero column locations of a row in the matrix (see Figure 4.1). Once the initial matrix structure is established, row- and column-reordering is performed using the Markowitz criteria [Mark57] to maintain matrix sparsity. A mock, symbolic equation solution step is then performed to identify all 'fill-in' matrix entries (terms which, though initially zero, become nonzero as a result of the equation solution process). The ease with which these

$$A \times X = b$$

$$\begin{bmatrix} 11 & 0 & 0 & 0 & 0 \\ 0 & 8 & 0 & 0 & -6 \\ 0 & -2 & 6 & -1 & -3 \\ -4 & 0 & 0 & 5 & 0 \\ 0 & 0 & -5 & 0 & 7 \end{bmatrix} \times \begin{bmatrix} X_1 \\ X_2 \\ X_3 \\ X_4 \\ X_5 \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \\ 5 \\ 2 \\ 4 \end{bmatrix}$$

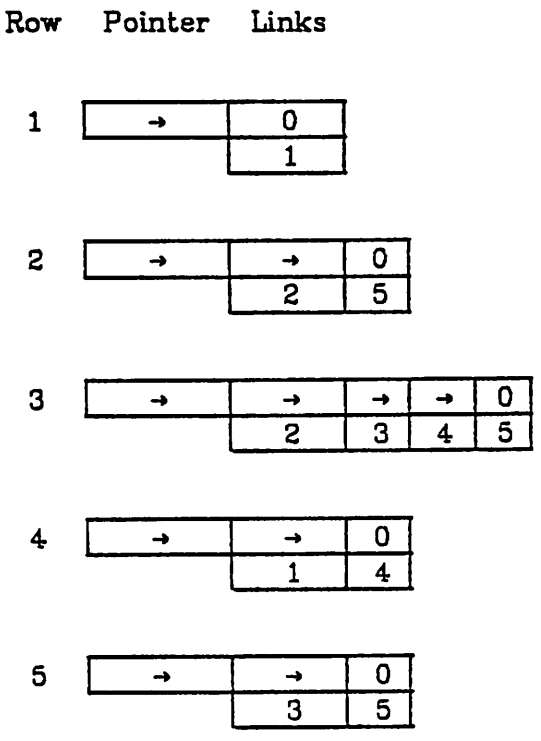


Figure 4.1. Singly-Linked List Matrix Representation

changes in the structure of the matrix can be performed using the linked-list representation motivated the use of this data structure.

Considerable overhead can be incurred, however, by always dealing with the matrix using this linked-list representation. The FORTRAN programming language is particularly ill-suited to high-speed manipulations of linked-list structures. Since numerical pivoting is not used in Version E.3 of SPICE2, the structure of the matrix is determined once and for all after the circuit description has been read and before any analysis is performed. Therefore, once the final matrix structure is determined, the linked-list representation is converted to a packed 'row-column indexing' notation which requires less memory [Nage75]. Of equal importance is the fact that the 'row-column indexing' notation allows immediate retrieval of the next nonzero row-element in the upper triangle of the matrix and of the next nonzero column-element in the lower triangle part. Both of these elements are precisely the ones required to perform efficiently the LU decomposition step of the linear equation solution. However, with the row-column notation it is much more difficult to make changes to the matrix organization. Figure 4.2, adapted from [Nage75] shows a small example matrix with the row-column indexing scheme included. Since the diagonal and right-hand-side terms are presumed to be nonzero, no sparse-matrix pointers are used to access these elements. All off-diagonal terms are stored in the A0 array, which is paired with the I0 array. These two arrays are accessed with the aid of the two index arrays IUR (Upper triangle Rows) and ILC (Lower triangle Columns). The I^{th} row matrix elements are stored in locations A0(IUR(I)) through A0(IUR(I+1)-1). If IUR(I)=IUR(I+1) there are no upper-triangle row terms in the I^{th} row. The column position is determined by the corresponding I0 array value. For example, in the second row there is one off-diagonal entry since

$$A \times X = b$$

$$\begin{bmatrix} 11 & 0 & 0 & 0 & 0 \\ 0 & 8 & 0 & 0 & -6 \\ 0 & -2 & 6 & -1 & -3 \\ -4 & 0 & 0 & 5 & 0 \\ 0 & 0 & -5 & 0 & 7 \end{bmatrix} \times \begin{bmatrix} X_1 \\ X_2 \\ X_3 \\ X_4 \\ X_5 \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \\ 5 \\ 2 \\ 4 \end{bmatrix}$$

Off-diagonal term storage					
index	IUR	ILC	I0	A0	matrix term
1	1	4	5	-6	A_{25}
2	1	5	4	-1	A_{34}
3	2	6	5	-3	A_{35}
4	4	6	4	-4	A_{41}
5	4	6	3	-2	A_{32}
6			5	-5	A_{53}

Figure 4.2. Row-Column Pointer System

IUR(3)-IUR(2)=1; the column number of that entry, 5, is contained in I0(IUR(2)). The lower-triangular terms are accessed in an identical manner, using the ILC array (instead of IUR) to index I0.

The use of numerical pivoting during analysis means that the structure of the matrix may change at any time. In Program SPUDS the matrix is therefore always stored using a linked-list representation. Chapter 5 describes the manner in which the extra overhead of manipulating the matrix due to this representation is eliminated. The data structures are patterned after the 'threaded list' technique used in the SLIC [Idle71], NICAP [Cerm71], and MICE [Cohe78] simulation programs. The example and description of the method presented here is taken from McCalla [McCa]. The formulation is illustrated in Figure 4.3. Unique storage locations are provided for each nonzero matrix term in the array VALU. The arrays IROW and JCOL record the (i,j) coordinates of each nonzero term in the system of equations. The arrays IRPT and JCPT indicate, respectively, the row, i, of the next nonzero element in a column (scanning down the matrix), and the column, j, of the next nonzero element in a row (scanning to the right across the matrix). The end of a column or row is indicated by a zero value for IRPT or JCPT. Consider column one: IRPT(1)=6 indicates that the first nonzero entry is at location 6, where IROW(6)=1 and JCOL(6)=1 indicate coefficient A_{11}. The next entry in the column is pointed to by IRPT(6)=7; at location 7, IROW(7)=4 and JCOL(7)=1 indicate coefficient A_{41}. Finally, IRPT(7)=0 indicates that A_{41} is the last nonzero term in column one. Similarly, to scan across row 3 one would begin at JCPT(3)=9, where IROW(9)=3 and JCOL(9)=2 indicate A_{32}, etc.

Note that this threaded list scheme is bi-directional. From any element, one can immediately proceed to the element below it via array IRPT and to

$$A \times X = b$$

$$\begin{bmatrix} 11 & 0 & 0 & 0 & 0 \\ 0 & 8 & 0 & 0 & -6 \\ 0 & -2 & 6 & -1 & -3 \\ -4 & 0 & 0 & 5 & 0 \\ 0 & 0 & -5 & 0 & 7 \end{bmatrix} \times \begin{bmatrix} X_1 \\ X_2 \\ X_3 \\ X_4 \\ X_5 \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \\ 5 \\ 2 \\ 4 \end{bmatrix}$$

array index	IRPT	IROW	JCOL	JCPT	VALU	matrix coefficient
1	6	–	–	6	1	b_1
2	8	–	–	8	3	b_2
3	11	–	–	9	5	b_3
4	13	–	–	7	2	b_4
5	10	–	–	12	4	b_5
6	7	1	1	0	11	A_{11}
7	0	4	1	15	-4	A_{41}
8	9	2	2	10	8	A_{22}
9	0	3	2	11	-2	A_{32}
10	14	2	5	0	-6	A_{25}
11	12	3	3	13	6	A_{33}
12	0	5	3	16	-5	A_{53}
13	15	3	4	14	-1	A_{34}
14	16	3	5	0	-3	A_{35}
15	0	4	4	0	5	A_{44}
16	0	5	5	0	7	A_{55}

Figure 4.3. Threaded-List Matrix Representation

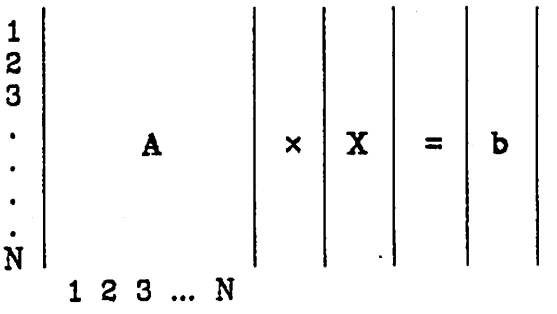
the element to the right of it via array JCPT. This capability, as noted previously, greatly facilitates the implementation of Gaussian elimination for equation solution.

The actual representation used for solving the linearized system

$$\mathbf{A} \times \mathbf{x} = \mathbf{b} \tag{4.1}$$

is shown in Figure 4.4. The right-hand-side vector **b** is appended to the coefficient matrix **A** as column N+1, so that $A_{1,N+1} = b_1$, $A_{2,N+1} = b_2$, etc. The order of the overall matrix is then N×(N+1). In SPUDS, an additional 'ground' column is also added to this system as column N+2 to implement the zero row-summation property of the matrix. This additional column is added at the far right-hand side of the matrix, rather than at the left-hand side, because the elements must be included in the steps taken as part of the Gaussian elimination to ensure that the sum of the values in any row of the augmented matrix remains zero during the solution step. This last column is otherwise ignored; in particular, it has no effect on the sparsity or reordering computations.

Each time a numerical pivoting step is performed, new matrix fill-in terms must be identified and allocated a location in the appropriate linked list. The effects of a subsequent row/column reordering can include the elimination of the need for previously-allocated fill-ins. These unnecessary terms could be eliminated only by starting over from the original matrix structure and re-establishing the positions of all required fill-ins. Fortunately, in practice, more than one numerical pivoting step is not usually required. Therefore, no search for or re-use of such fill-ins is performed by SPUDS. The matrix structure at the end of the dc analysis, including any fill-in terms which may have been added, is used as the starting structure for



Actual linear system

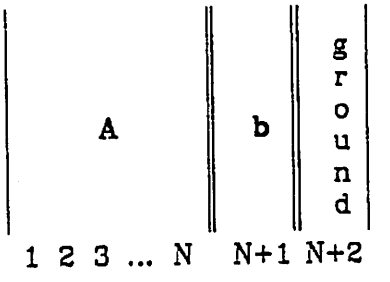


Figure 4.4. Augmented Matrix Representation

a subsequent transient analysis of the same circuit. Once an element is added to the coefficient matrix for any analysis, the term remains for the rest of the program execution.

4.2. Storage Reduction Techniques

It is desirable to reduce the high-speed memory requirements of a simulation program. Since accesses to memory require more time than do typical operations in the CPU, a reduction in data structure sizes can mean a reduction in the number of references to memory and hence an increase in program speed. Also, smaller usage of memory for a given circuit means that larger, more complex circuits may be analyzed in a given amount of memory.

The data structures used to represent the input circuit description in Program SPUDS are a set of linked lists similar to those used in SPICE2 [Cohe76]. Savings in the amount of memory required by these lists in SPUDS is due primarily to the smaller wordsize of the minicomputer, which permits input circuit variables to be stored easily in a smaller number of bits of memory than on larger-wordsize computers. Significant savings in memory requirements relative to SPICE2 have been obtained by data restructuring, by some algorithmic changes, and by taking advantage of the smaller wordsize of the minicomputer.

In Program SPICE2, intermediate analysis results and state information for energy-storage elements are stored in the 'LXi' (LX0, LX1, ...) tables. Each 'table' is a dynamically-allocated contiguous region of memory. Figure 4.5 shows the contents of the LX0 table for a MOS transistor; both dc operating-point information, e.g. drain current and transconductance, and charge-storage information (capacitor charges and currents) are stored.

LXi	+ 0:	V_{bd}	(bulk-drain voltage)
	+ 1:	V_{bs}	
	+ 2:	V_{gs}	
	+ 3:	V_{gd}	
	+ 4:	I_d	(drain current)
	+ 5:	I_{bs}	(bulk-source diode current)
	+ 6:	I_{bd}	(bulk-drain diode current)
	+ 7:	g_m	
	+ 8:	g_{ds}	(drain-source conductance)
	+ 9:	g_{mbs}	
	+10:	g_{bd}	(bulk-drain conductance)
	+11:	g_{bs}	(bulk-source conductance)
	+12:	C_{bd}: q	(charge on bulk-drain capacitance)
	+13:	C_{bd}: i	(current flowing through C_{bd})
	+14:	C_{bs}: q	
	+15:	C_{bs}: i	
	+16:	C_{gs}: q	
	+17:	C_{gs}: i	
	+18:	C_{gd}: q	
	+19:	C_{gd}: i	
	+20:	C_{gb}: q	
	+21:	C_{gb}: i	

Figure 4.5. SPICE2 LX0 Table Contents for MOS Transistors

The number of these blocks of memory is determined by the type of analysis and, for transient analysis, by the type and order of method used for numerical integration.

For the multi-point analyses such as transient analysis, the LX0 table contains the results of the current iteration while the other tables (LX1, LX2,...) contain copies of previous contents of the LX0 vector (the solutions found at previous sweep points). All of this information together is sufficient to 'back-up' to the previous sweep point if the solution at the current point is rejected (perhaps due to excessive local truncation error). The initial guess at the solution to a new sweep point is obtained by linear extrapolation from the two previous points, in the same manner as the case of dc transfer curve analysis. For example, if h_{rat} is defined as the ratio of the timesteps used at the current and previous timepoints in transient analysis, the initial guess for the solution to the new timepoint is obtained from

$$LX0 = (1+h_{rat})\times LX1 - h_{rat}\times LX2 \qquad (4.2)$$

Therefore, a complete 'back-up' requires the solutions from the two previous sweep points, which are stored in the LX1 and LX2 tables. Finally, if at least a second-order method for numerical integration is used in transient analysis, three previous values of the energy-storage element data (LX1, LX2, and LX3) are required in order to estimate the local truncation error effectively. Therefore, Program SPICE2 keeps four 'LXi' tables (LX0 through LX3) when transient analysis is performed.

In Program SPUDS, the LXi tables are split into two parts. The operating-point information is stored in a separate set of tables, LQPTi, and only the data pertaining to energy-storage elements is retained in the LXi tables. Figure 4.6 shows the revised LXi and LQPTi table contents for MOS

LQPTi	+ 0:	V_{bd}	(bulk-drain voltage)
	+ 1:	V_{bs}	
	+ 2:	V_{gs}	
	+ 3:	V_{gd}	
	+ 4:	I_d	(drain current)
	+ 5:	I_{bs}	(bulk-source diode current)
	+ 6:	I_{bd}	(bulk-drain diode current)
	+ 7:	g_m	
	+ 8:	g_{ds}	(drain-source conductance)
	+ 9:	g_{mbs}	
	+10:	g_{bd}	(bulk-drain conductance)
	+11:	g_{bs}	(bulk-source conductance)

LXi	+ 1:	C_{bd}: q	(charge on bulk-drain capacitance)
	+ 2:	C_{bd}: i	(current flowing through C_{bd})
	+ 3:	C_{bs}: q	
	+ 4:	C_{bs}: i	
	+ 5:	C_{gs}: q	
	+ 6:	C_{gs}: i	
	+ 7:	C_{gd}: q	
	+ 8:	C_{gd}: i	
	+ 9:	C_{gb}: q	
	+10:	C_{gb}: i	

Figure 4.6. SPUDS LXi and LQPTi Table Contents for MOS Transistors

transistors. Memory requirements are reduced due to two factors. First, the copy of the operating-point information that was part of the LX3 table but was never referenced is no longer stored. Conceptually, this change eliminates the need for an LQPT3 table. Second, the LQPT2 table can be omitted after careful examination of the prediction step. Just after the analysis at a particular sweep point has converged and before the analysis of the following point has begun, the LQPT0 table contains the device operating-point information for the sweep point just solved. Tables LQPT1 and LQPT2 contain the corresponding data for previous sweep points. If the new sweep point has been accepted, the contents of LQPT2 are replaced by the contents of LQPT1, the contents of LPQT1 are replaced by the contents of LPQT0, and the prediction step

$$LQPT0 = (1+h_{rat})\times LPQT1 - h_{rat}\times LPQT2 \qquad (4.3)$$

is used to compute the initial guess for the solution to the following sweep point. If the new sweep point is not accepted, the timestep is adjusted and a new prediction calculation is made.

The need for the third copy of the device operating-point information (the LQPT2 table) is eliminated by a small algorithmic change in this prediction step. If the new sweep point is accepted, the contents of the LQPT0 and LQPT1 tables are exchanged. This swap puts the information from the sweep point just solved into LQPT1 and the preceding sweep point data into LQPT0. The prediction step then becomes

$$LPQT0 = (1+h_{rat})\times LQPT1 - h_{rat}\times LQPT0 \qquad (4.4)$$

which is equivalent to the formulation using LQPT2 in Equation (4.3), since the right-hand-side of Equation (4.4) is evaluated before any assignment is made to the left-hand-side of the statement. The only difficulty with this modification arises if the newly-predicted sweep point is subsequently

rejected; since the twice-removed sweep-point data is no longer stored, the prediction step of Equation (4.4) cannot be applied. An additional flag, PREDCT, keeps track of this situation. If the newly-predicted sweep point is rejected, the initial guess is simply taken to be the value of the previous sweep point, and no linear extrapolation calculation is performed[1]. No change in analysis iteration counts is observed when this change in the prediction algorithm is used.

One aspect of designing data structures for a specific task is the trade-off between memory requirements and program performance. In one area, however, element storage requirements in SPUDS have been reduced with virtually no effects on simulation speed. Each input element contributes certain values to different locations in the equation coefficient matrix. For example, a conductance g connected between nodes a and b in a circuit adds to the matrix coefficients in the (row, column) positions (a,a), (a,b), (b,a), and (b,b). Since the matrix is sparse, only the nonzero entries are stored, in a one-dimensional vector A. In a high-level language it is a relatively time-consuming process to determine the location in the A-vector which corresponds to some (i,j) matrix entry. Therefore, in most circuit simulators, e.g. SLIC, SINC, SPICE2, MSINC, this mapping is performed exactly once, after the matrix structure has been established. Memory space is allocated for each circuit element to hold pointers to each location in the one-dimensional vector to which the element contributes. The number of such locations is a significant part of the total per-element storage; for the MOS device model in SPICE2 (and SPUDS), there are 22 such locations for each MOS transistor. In SPUDS, the special instructions developed for incremental

[1]This simple prediction method, in which the solution at the previous sweep point is used as the initial guess for the new sweep point, is the method used in Program SPICE1.

loading of the coefficient matrix evaluate this mapping function for each matrix element loaded, at a negligible overhead. As a result, this matrix pointer space is eliminated from the input element storage in SPUDS.

Finally, memory savings also arise as a byproduct of the smaller word-size of the minicomputer. Circuits for which a dedicated small computer is suitable are usually small enough that a single 16-bit integer is sufficient to represent node numbers, matrix locations, and most memory addresses. The memory in several large mainframe computers [CRAY76] [CDC79] can be accessed only as 60- or 64-bit words, which means that the storage requirements for these values are much larger than for the minicomputer. Significant savings also result, for the same reason, from the use of 32-bit floating-point arithmetic.

4.3. Comparison of Memory Requirements

It is difficult to compare memory requirements of different circuit simulation programs because differences in device model complexity and program organization can radically alter storage needs. An additional obstacle to any comparison is the difficulty of determining what the actual memory requirements are for a particular simulation program. For example, Version D.0 of Program MSINC prints its memory requirements only for the dc operating-point analysis. MSINC requires 4755 16-bit words of data space to perform a dc operating point analysis of the MOSAMP2 benchmark circuit; SPUDS requires only 4137 words for the same analysis. However, 1042 words of the total for SPUDS contain the instructions for the Linear Equation Solution Machine; if this space is not included, the data space requirements of SPUDS are 35% less than those of MSINC. In order to minimize the effects of device model and structural differences between programs, the principal

comparisons made in this section are between SPUDS and Version E.3 of SPICE2, which have similar program structures and identical device models.

A breakdown by function of the memory requirements of SPUDS is shown in Table 4.1. The table shows the number of 8-bit bytes of memory required to store the information required to perform transient analyses on several representative test circuits. The 'storage type' column identifies the purpose of each type of information. 'Input elements' is the memory required to store the input circuit description; 'Swap data' records the row- and column-swaps performed on the coefficient matrix. 'LXi tables' refers to the energy-storage element data necessary for transient analysis, and 'LQPT0+LQPT1' is the nonlinear-device operating-point information. Finally, 'MACINS' is the memory required for the generated instructions of the Linear Equation Solution Machine (LESM) described in Chapter 5. Also shown in the table is the memory required for the same circuits in Version E.3 of SPICE2 running on the CDC 6400 computer (which has a 60-bit wordsize).

Several factors should be noted in looking at this table. First, the device models in SPUDS and Version E.3 of SPICE2 are identical and require the same number of device parameters and circuit nodes. Second, the version of SPICE2 referenced uses the 'row-column indexing' scheme for storing the sparse matrix structure, which requires less memory than the linked-list approach in SPUDS. Finally, SPUDS stores all output variables to be printed and/or plotted in a disc file, while SPICE2 stores them in memory. For the test circuits, however, no more than two thousand bytes (2KB) of memory on the CDC 6400 are used for output variable storage.

On average, SPUDS requires only a third as much memory as SPICE2 for the same analyses of the same circuits. The data indicate that the 32000 16-

Storage type	DIFPAIR	UA741	MECL3	BOOTINV	MOSAMP2
Program SPUDS:					
Input elements	1208	2472	1956	1052	2584
Swap data	216	636	624	132	312
Matrix pointers	728	3160	2768	432	1840
Matrix values	364	1580	1384	216	920
LXi tables	384	2144	1184	896	4384
LQPT0+LPQT1	256	1408	800	480	2592
MACINS	578	3596	2964	340	2084
Total	4488	16064	12712	4416	15584
Program SPICE2:					
Total	13005	43125	31035	14025	42660

Table 4.1. Memory Usage (Bytes) for Transient Analysis

bit words of data space available on the minicomputer are sufficient for the analysis of 100-200 device circuits (depending on circuit complexity and type of analysis), which is enough for the analysis of building-blocks or cells in VLSI design. Since this research is aimed at providing simulation response times in at most several minutes, the amount of memory available is not a constraining factor.

CHAPTER 5

DEDICATED HARDWARE

The optimal performance from a computer-based simulator results when the algorithms used and the hardware available are well-suited to each other. More than one order-of-magnitude improvement in the speed of several sub-tasks in IC simulation can be achieved through the use of special-purpose instructions tailored for those parts of the analysis which are especially time-consuming.

For a given set of algorithms implemented in a simulation program, a performance measurement tool is necessary to determine accurately those parts of the simulation task in which the majority of the total computation time is spent. Section 5.1 describes the SPY program, which is used to obtain data on just where a program spends its time. The SPY program is the source of all measurement data in this chapter regarding the relative importance of different parts of the circuit analysis task. In order for the comparisons of CPU time to be unambiguous, all percentage breakdowns given in this chapter of CPU time during analysis are fractions of the total time spent in analysis, unless explicitly stated otherwise.

The design and construction of the dedicated computer hardware necessary to implement the special high-level operations described in this chapter are presently very time-consuming processes. In order to evaluate the effects of such hardware in a short time period, a user-microprogrammable

minicomputer is used to examine the effects of providing certain special-purpose instructions for the use of the simulation program. The use of microcode to emulate the dedicated hardware allows great flexibility and speed in changing the instruction set and data formats of the basic computer. Changes in these formats are made based on the conclusions reached in previous chapters regarding computer architectures, simulation algorithms, and data structures. The architectural aspects of the use of microcode for the minicomputer used as a test vehicle for this research are described in Section 5.2.

Analysis of the overall simulation time for code written solely in FORTRAN shows that on the minicomputer, the linearized equation solution step is the single most CPU-time-consuming part of the entire analysis. Particular attention is therefore given to this task, and special data structures and machine instructions are presented which significantly increase the overall solution speed. Section 5.3 describes this equation solution portion of the analysis problem.

Other tasks performed during analysis increase in relative importance once the equation solution time is reduced, Section 5.4 describes several of the more time-consuming portions of the problem, such as incremental loading of terms in the coefficient matrix. The improvements obtained by adding appropriate special-purpose instructions to the minicomputer are detailed.

Finally, Section 5.5 presents a summary of the improvement in simulation performance obtained by using all of the special-purpose instructions and data formats described previously in this chapter. Although the actual magnitudes of CPU times for analysis reflect the limitations of the physical hardware of the minicomputer used for this investigation, the relative

improvements are representative of the speedup possible for any comparable computer system.

The first version of SPUDS is used throughout this chapter as a reference against which speed improvements are measured. This version is written solely in FORTRAN and uses 64-bit precision for all floating-point arithmetic and data storage[1]. The starting point for this version of SPUDS is Version E.3 of Program SPICE2, with only those changes minimally required so that the code executes successfully on the minicomputer. No changes to device model equations are made in SPUDS so that simulation accuracy can be verified readily. All improvements in simulation speeds are stated relative to this original version of SPUDS.

5.1. Performance Measurement

The cost effectiveness of IC simulation depends to a large extent on the amount of computer resources required. Because the simulation task involves relatively little input or output but a large amount of computation, the amount of CPU time required is a good measure of simulation cost. Compute-bound programs usually spend the majority of their CPU time in performing the computations of relatively small parts of the program code. A program analyzer which identifies those small parts efficiently helps to focus effectively efforts to improve program performance.

Three different methods are used commonly to determine where in the code a program spends its CPU time. First, the source code may be augmented so as to record the elapsed CPU time just before and just after each section of code to be monitored; the difference in readings is used to estimate the time spent in each section. This method is effective for obtaining

[1] a 32-bit version of SPUDS is used in earlier chapters of this report

data on large sections of code (hundreds of statements) but two factors prohibit the use of this method for obtaining finer detail. First, even if the insertion of statements to record the elapsed CPU time is automated, the accuracy of the system (hardware) clock is usually insufficient to measure precisely the CPU time needed to execute individual program statements. Second, the finer the measurement interval the greater the total CPU time required, since CPU time is itself required to record the elapsed CPU time.

A second way to ascertain where a program spends its time is to trace or simulate program execution one instruction or statement at a time and to keep record of how many times each instruction/statement is executed. This simulation is performed by another program which emulates the actions of a computer executing the program of interest. However, this method is practical only for very small amounts of program code, since the typical CPU-instruction simulator is two to three orders-of-magnitude slower than the actual computer being simulated. For large circuit simulation programs this tracing of large sections of code is too expensive to be practical.

A third approach to monitoring program execution is to sample the value of the program counter (PC) of the CPU while the program is in execution. The PC contains the memory address of the current instruction. This statistical approach can be as accurate as either of the two methods described previously as long as enough samples are taken. In addition, since actual program execution is monitored, the PC samples are weighted automatically by the relative instruction execution times. This weighting is desirable since what is sought is not just how many times a given instruction is executed but rather the product of the number of executions of each instruction and the cost (CPU time) of its execution.

The SPY program[2] implements the third (statistical) monitoring technique in an efficient manner. SPY executes concurrently with the program being monitored, but at a higher scheduling priority. Every 10ms (the resolution of the hardware time-of-day clock) SPY reads the PC of the program of interest and records the data by incrementing the appropriate histogram bucket in memory. The execution of the SPY program itself does modify the CPU time available to the program that SPY monitors, but since SPY uses a negligible amount of CPU time itself and does access any peripheral equipment (such as disc), the disturbance is small. Repetitive SPY runs yield statistics which agree with each other within 1%. SPY can display interactively the histogram data on a graphics terminal or write the gathered statistics to a disc file for later output on a line printer or 4-color pen plotter.

5.2. Microcode Access

The Hewlett-Packard 1000 F-Series 16-bit minicomputer is used as a test vehicle for the investigation of special-purpose computer instructions tailored to the analysis tasks because its microprogramming structure is well documented, and the use of the machine with user-written microcode is supported by Hewlett-Packard with translation and utility programs. This section describes the characteristics of this minicomputer which are relevant to the work presented in the rest of this chapter.

All instructions on the HP 1000 are implemented through the use of microcoded subroutines. Conceptually, therefore, there is no difference between those instructions provided by the manufacturer (the 'base' instruction set) and those instructions which are developed by a user.

---

[2]developed by the author in 1978

The micro-instruction cycle time for the HP 1000 ranges between 175ns and 280ns when no reference is made to main memory. If 'high-speed' memory is used, memory references can require as much as 420ns. Typical assembly-language level instructions which reference main memory require $\approx 1\mu s$ for execution. The F-Series minicomputer has an asynchronous hardware floating-point arithmetic unit which is read and written directly by the microprocessor. All data paths are 16 bits wide.

The base instruction set supports user-written microcode through a special form of the subroutine call instruction. For example, a new instruction NEW can be developed first as a FORTRAN subroutine and can be invoked as

CALL NEW (argument list).

When the microcoded version of NEW is available, the loader can be directed to replace all occurrences of subroutine call instructions which reference the symbol NEW by the appropriate micro-subroutine call instruction.

The nominal overhead required by the base instruction set to pass control to a user-written micro-subroutine is $1.3\mu s$. A much greater additional overhead arises as a side-effect of circumventing the memory mapping hardware limitations of the HP 1000. The minicomputer has a dynamic memory-mapping system which allows up to a total of 2 million bytes (2MB) of main memory to be addressed. However, all non I/O memory references go through one of two sets of map registers called SYS and USR, each of which permits access to a 64 thousand byte (64KB) portion of the 2MB total. Access to an arbitrary location in the 2MB address space is performed by changing one of the sets of mapping registers to include the desired location within the mapped 65KB portion of memory. The standard HP operating sys-

tem[3] uses the SYS map for the memory-resident part of the operating system and the USR map for the memory needed by a user program. In order to have sufficient data space to permit simulation of circuits with 100-200 devices, the micro-subroutines developed for SPUDS which implement special-purpose instructions first save the SYS mapping registers and then change the SYS map to address a separate 65KB data area in main memory. The SYS map is restored to its original condition just before the micro-subroutine exits back to the microcode which implements the base instruction set. The time required to save, modify, and finally restore the SYS mapping registers is $\approx 83\mu s$.

A smaller number of more powerful special-purpose instructions have been designed as a result of this large overhead, rather than a larger number of simpler instructions. In all cases, an attempt is made to limit the number of times that a special-purpose instruction must be invoked. In several instances, instructions are designed to accept an arbitrary number of arguments in a single call to achieve this goal.

5.3. Equation Solution

The data obtained by running the SPY program as a monitor of the behavior of SPUDS with all-FORTRAN code and either 64- or 32-bit floating-point operations is summarized in Table 5.1. The total CPU time spent in transient analysis can be identified as 30% in solving the linearized system of circuit equations, 15% in estimating the local truncation error (LTE), and 50% in the 'device modelling' routines. These 'device modelling' routines perform several functions for each nonlinear device in the circuit. The device parameters are gathered together from different locations in memory for fast

[3]RTE-IVB

Transient analysis CPU time breakdown	
Percentage	Task
50±10	Device modelling
20	gather model parameters
15	evaluate model equations
15	incrementally load matrix
30±10	Linear equation solution
15±5	Estimate Local Truncation Error (LTE)

Table 5.1. SPY Data: All-FORTRAN SPUDS

model equation evaluation. A 'bypass' calculation is performed to determine whether device equation evaluation is in fact necessary. After model evaluation, the linearized model components are saved for possible use in a later 'bypass'. Finally, the coefficient matrix is incrementally loaded with the contributions from the device. Of the total analysis time, approximately 20% is spent gathering parameters, 15% in model evaluation, and 15% in matrix loading.

The percentage variation shown in the table is due primarily to differences in device model complexity. For example, although the MOS device model is considerably more complex than the bipolar model, an MOS circuit usually does not have significantly more nodes than does a bipolar circuit. Therefore, relatively more time is spent in the device modelling routines for MOS circuits than for bipolar ones.

This data indicates that the analysis task which requires the most CPU time and which cannot readily be decomposed into simpler operations is that of linear equation solution. This section presents the modifications to the solution code which increase solution speed through the use of special computer instructions.

Section 5.3.1 describes two preliminary test analyses which measure the extent to which the minicomputer simulation speed depends on idiosyncrasies of the software or hardware machine architecture for one particular computer. The derivation of an optimally fast implementation of the equation solution task is presented in Section 5.3.2. The resulting 'equation solution' machine is detailed in Section 5.3.3, and various measures of machine speed are presented in Section 5.3.4. The costs of using this solution machine are noted in Section 5.3.5. Finally, the dependence of solution tim-

ings on the number of circuit equations is examined in Section 5.3.6.

5.3.1. Machine Idiosyncracies

Two test runs are described in order to measure possible speed penalties inherent in the software and hardware architecture of the HP minicomputer, so as to distinguish those speed improvements which can be attributed to peculiarities of one particular minicomputer from those improvements due to more effective use of available hardware. First, Table 5.2 shows the CPU time requirements when all-FORTRAN LU decomposition and forward- and backward-substitution routines are re-written in assembly language. This modification measures the extent to which poor code generation from the FORTRAN compiler supplied by Hewlett-Packard contributes to less-than-optimal performance. The columns in the table headed 'DCX' and 'TRAN' give the CPU time, respectively, for the dc transfer curve and transient analyses. On the average, only a 10% reduction in analysis time is obtained from this recoding of the solution routines.

Part of the base instruction set supports the use of an 'Extended Memory Area' (EMA) data area distinct from regular user memory which can be used as a block of auxiliary storage by an application program. SPUDS uses a 32000-word block of EMA to hold all data needed for its circuit analyses. Access to EMA data normally requires the use of special microcoded instructions supplied by Hewlett-Packard. The costs of this access are shown in Table 5.3, which presents the results of adding special EMA-access microcoded instructions tailored to the memory referenced behavior of SPUDS. These special microcoded access routines for the assembly-language level references to EMA reduce the total analysis time by 25%.

Circuit	Equation solution with			
	FORTRAN		assembly language	
	DCX	TRAN	DCX	TRAN
DIFPAIR	21.37	27.94	19.87	26.19
KTEST	0.00	12.40	0.00	11.21
RCA304	52.52	84.99	48.02	78.62
UA709	114.64	140.33	100.23	125.68
UA727	105.23	198.68	94.23	179.74
UA733	0.00	0.00	0.00	0.00
UA741	136.71	154.22	121.53	141.81
RTLINV	14.42	25.06	13.47	23.42
TTLINV	41.28	114.33	37.14	103.19
TTL74	42.68	99.55	38.41	90.48
TTL74S	50.88	132.84	45.47	119.36
TTL74L	46.39	123.75	41.53	112.04
TTL920	54.13	0.00	47.68	0.00
ECLGATE	47.18	116.60	42.59	106.21
MECLIII	73.61	150.53	65.67	135.92
SBDGATE	84.94	193.47	75.20	174.15
CCSOR	0.00	0.00	0.00	0.00
DCOSC	0.00	0.00	0.00	0.00
CFFLOP	0.00	0.00	0.00	0.00
STCRC	0.00	7.57	0.00	7.35
CHOKE	0.00	16.86	0.00	16.28
ECLINV	0.00	47.95	0.00	44.61
SCHMITT	0.00	52.63	0.00	48.62
ASTABLE	0.00	88.73	0.00	80.38
SATINV	40.43	0.00	38.59	0.00
DEPLINV	13.15	0.00	12.70	0.00
RATLOG	0.00	95.30	0.00	91.19
INVCHN	0.00	0.00	0.00	0.00
BOOTINV	0.00	44.72	0.00	43.18
MOSMEM	0.00	115.99	0.00	111.62
MOSAMP1	150.19	0.00	139.77	0.00
MOSAMP2	0.00	399.69	0.00	379.45

Table 5.2. Solution Speedup Due to Assembly Language

Circuit	Equation solution with			
	assembly language		Special EMA microcode	
	DCX	TRAN	DCX	TRAN
DIFPAIR	19.87	26.19	18.34	24.47
KTEST	0.00	11.21	0.00	9.89
RCA304	48.02	78.62	43.00	71.87
UA709	100.23	125.68	82.36	108.29
UA727	94.23	179.74	81.40	158.37
UA733	0.00	0.00	0.00	0.00
UA741	121.53	141.81	103.39	128.27
RTLINV	13.47	23.42	12.46	21.70
TTLINV	37.14	103.19	32.48	91.66
TTL74	38.41	90.48	33.44	80.56
TTL74S	45.47	119.36	39.28	104.16
TTL74L	41.53	112.04	36.00	99.15
TTL920	47.68	0.00	40.14	0.00
ECLGATE	42.59	108.21	37.43	94.63
MECLIII	65.67	135.92	56.41	119.44
SBDGATE	75.20	174.15	64.01	151.80
CCSOR	0.00	0.00	0.00	0.00
DCOSC	0.00	0.00	0.00	0.00
CFFLOP	0.00	0.00	0.00	0.00
STCRC	0.00	7.35	0.00	7.12
CHOKE	0.00	16.28	0.00	15.28
ECLINV	0.00	44.61	0.00	41.20
SCHMITT	0.00	48.62	0.00	44.46
ASTABLE	0.00	80.38	0.00	71.29
SATINV	38.59	0.00	37.38	0.00
DEPLINV	12.70	0.00	12.53	0.00
RATLOG	0.00	91.19	0.00	89.18
INVCHN	0.00	0.00	0.00	0.00
BOOTINV	0.00	43.18	0.00	42.78
MOSMEM	0.00	111.62	0.00	111.03
MOSAMP1	139.77	0.00	134.10	0.00
MOSAMP2	0.00	379.45	0.00	367.87

Table 5.3. Time Reduction Due to EMA Microcode

5.3.2. Optimal Implementation

The algorithmic steps needed to perform the LU decomposition part of the equation solution step is shown in Figure 5.1. Because the two-dimensional matrix is stored as a non-sparse one-dimensional vector, the A_{jk} coefficients cannot be accessed directly. For each arithmetic operation in each iteration, the j^{th} row must be searched for the jk^{th} entry. (If the coefficient is in the upper-triangular part of the matrix the k^{th} column may be searched instead since the average search length is reduced.) This 'LOCATE' effort may require searching through linked lists, bit maps, or row-column indexing schemes, and this searching requires a considerable amount of CPU time. Alternatively, the locations of all the A_{jk} coefficients can be found once and stored in an auxiliary MEMO array, at the cost of additional memory.

None of these approaches, however, leads to an optimally fast equation solution because of the additional machine operations that are required. A substantial part of the computation expressed in Figure 5.1 has nothing directly to do with the desired result of obtaining the LU factorization of the coefficient matrix. Consider the normalization step

$$A_{ji} = A_{ji} / A_{ii}$$

For a typical register-architecture computer, the minimum instruction sequence to accomplish this operation is

LOAD	A_{jk} into R1 (register #1)
DIVIDE	R1 by A_{ii} (with quotient in R2)
STORE	R2 into A_{ji}

The time required by these few instructions is completely overshadowed by the time necessary to perform all the bookkeeping actions (incrementing k

```
for ( i=1 ; i<=N-1 ; i=i+1 ) {
        for ( j=i+1 ; j<=N ; j=j+1 ) {
                LOCATE ji^th and ii^th entries
                        in coefficient matrix A
                A_ji = A_ji / A_ii
                for ( k=i+1; k<=N ; k=k+1 ) {
                        LOCATE jk^th, ji^th, and ik^th entries
                                in coefficient matrix A
                        A_jk = A_jk - A_ji×A_ik
                }
        }
}
```

Figure 5.1. LU Decomposition Algorithm

and comparing with N, searching for A_{jk}, etc.) of the LU decomposition. The number of references to memory is an especially critical parameter, since central memory is much slower than the CPU speed. The bookkeeping operations require many more memory references than are needed to perform the actual LU factorization.

The fastest possible equation solution for a typical register-architecture computer is obtained when a set of machine instructions are executed which do nothing other than solve the linear system of equations. Such a set of instructions can be produced by generating FORTRAN source code in which the loops are unravelled and all array references use constant subscripts. Such code has the form

```
...
A(23) = A(23)-A(4)×A(6)
A(10) = A(10)/A(7)
...
```

where the constant subscripts (23, 4, etc.) reflect the locations of the appropriate A_{jk} terms. This generated code can then be processed by a FORTRAN compiler to obtain the desired machine instructions. Alternatively, because the structure of the arithmetic operations is quite simple and limited in scope, the desired machine instructions can be generated directly, avoiding the CPU time necessary for a compilation step. The generated code can be viewed as an extension of the MEMO array concept. Rather than just storing the location of each A_{jk} in MEMO, the arithmetic operations as well as the location of the matrix elements are stored together (as machine instructions).

This 'code generation' technique was first suggested by Gustavson, *et al* [Gust67]. The principal difficulty with this method is its potentially large requirements for memory. The special-instruction format described in the next section is designed specifically to provide a very high instruction density.

5.3.3. Solution Machine

In view of the need for high instruction density and the relatively large overhead incurred as part of invoking any micro-subroutine implementing a 'special' instruction, a 'Linear Equation Solution Machine' (LESM) is described which has its own program counter and instruction set. Once the structure of the matrix is established, the FORTRAN subroutine CODGN generates a loop-free and location-independent sequence of instructions for the LESM by performing a symbolic Gaussian elimination. Whenever the circuit equations need to be solved, the LESM is invoked by a single subroutine call and is passed the locations of the matrix pointer structures and the generated instructions. Upon completing the solution process, the LESM exits back to the microcode which simulates the nominal base set instructions of the minicomputer.

The LESM has 4 operation codes, corresponding to the four operations performed during the equation solution process. Generically, these operations are:

(0) IF (ABS(PIVOT) .LT. EPSILON)
<error: force new pivot>

(1) **A = A / B**

(2) **A = A - X × C**
(**X** = most recently computed quotient)

(3) **A = A - B × C**

Operation (0) tests whether the matrix coefficient about to be used as a pivot value is large enough to avoid numerical ill-conditioning difficulties. Operations (1) and (3) perform the indicated arithmetic operations. Operation (2) is included for the sake of efficiency; during the inner-loop iteration in the LU decomposition step

```
for ( k=i+1; k<=N ; k=k+1 ) {
   A_jk = A_jk - A_ji × A_ik
}
```

the value A_{ji} ('X') is stored in a high-speed register to avoid numerous unnecessary references to memory. A fifth operation, HALT, which tells the LESM to exit back to the nominal machine base-instruction-set microcode, is encoded as Operation (3) with an invalid operand address.

The instruction format for the LESM is shown in Figure 5.2. The four operation codes require two bits of encoding. There is a maximum of 32K 16-bit words of data space for SPUDS, due to the limitations of the dynamic mapping addressing hardware on the HP 1000 computer. Since a minimum of 32 bits (2 words) is used for floating-point arithmetic, the maximum real-valued array index, I, for any matrix coefficient is

$$I = 32768 / 2$$

$$= 16384$$

which can be represented in the 14 bits remaining in a single 16-bit word.

Operation (0): IF (ABS(PIVOT) < EPSILON)
error: force new pivot

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	< address of PIVOT >													

Operation (1): A = A / B

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	< address of A >													
<address of B >															

Operation (2): A=A−X×C

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	< address of A >													
<address of C >															

Operation (3): A=A−B×C

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	< address of A >													
<address of B >															
<address of C >															

Figure 5.2. LESM Instruction Format

Therefore, the instruction length of Operation (0) is a single 16-bit word. Operations (1) and (2) have a second operand address and therefore have instruction lengths of two words. Operation (3) is the only one which has three operand addresses; the format for this instruction requires three 16-bit words.

The number of memory references needed to fetch the LESM instructions required for a given set of equations is more than one order of magnitude less than the corresponding number for the base-set instructions of the minicomputer. This reduction is due to the higher-level primitive operations defined in the LESM and these savings contribute substantially to the increased solution speed of the LESM. (The generation of equivalent loop-free machine instructions based on the nominal instruction set of the minicomputer is not described because the instructions would not fit in the available data space.)

5.3.4. Speed Comparisons

In order to measure accurately the performance improvement in equation solution time due to the LESM, the exact sequence of instructions which solve the dc equations of the benchmark UA741 circuit, and the values in the coefficient matrix just before the solution step were extracted from SPUDS and used as input for an accurate timing comparison. For the 52 equations required by this circuit, the LESM instructions consist of 51, 179, 240, and 247 instances of Operations (0) through (3), respectively. The results of the timing comparison between a FORTRAN version of the CODEX ('CODe EXecution') subroutine which emulates the behavior of the LESM and the actual LESM are shown in Figure 5.3. As the figure presents, the solution speed is improved by a factor of approximately 18. An even greater speed-up factor

dc transfer curve analysis of UA741 (303 iterations)		
Reference	Code execution with	
	FORTRAN	microcode

do i=1,303 {
load matrix
 	
}
	do i=1,303 {
	load matrix
	CALL CODEX
	}
		do i=1,303 {
		load matrix
		CALL CODEX
		}

16.3 seconds	89.6 seconds -16.3	20.3 seconds -16.3
	73.3 seconds	4.0 seconds

Speedup factor:

microcode / FORTRAN

= 73.3 / 4.0

= 18.3

Figure 5.3. Measurement of Microcoded Solution Speedup

applies if the comparison is made between the LESM and FORTRAN code which performs the equation solution using the nested loops of Figure 5.1.

A measure of the optimality of the equation solution machine is the extent to which unnecessary references to main memory and irrelevant arithmetic operations are eliminated. Figure 5.4 shows part of the effective hardware architecture of the HP 1000 computer; note that data paths are only 16 bits wide. Consider operation (1):

A = A / B

The timing for an optimal execution of this operation is summarized in Figure 5.5 for 64-bit floating-point operands. The minimum time needed by the HP 1000 hardware to fetch the values of A and B, transfer them to the Floating Point Processor (FPP), retrieve the quotient from the FPP and store it into A is 7.14μs. The time required by the FPP to perform a 64-bit divide is approximately 7μs. The FPP hardware on the HP 1000 is asynchronous; transfers from main memory to the microprocessor can also overlap in time with transfers from the microprocessor to the FPP. Therefore, depending on the extent to which the operations can overlap, the minimum time to perform this operation is between 7μs and 14μs. The microcode which evaluates Operation (1) requires $\approx$15.21μs; 5.3μs of the 7μs divide time is overlapped with other micro-operations. This data indicates that the microcoded LESM executes within a factor of $\approx$2 of the optimal hardware speed for this particular minicomputer and that substantial concurrency of memory references and floating-point computation is possible in the linear equation solution process.

On the average, use of the LESM reduces dc transfer curve and transient analyses times for the benchmark circuits by 25-40%. SPY data show that

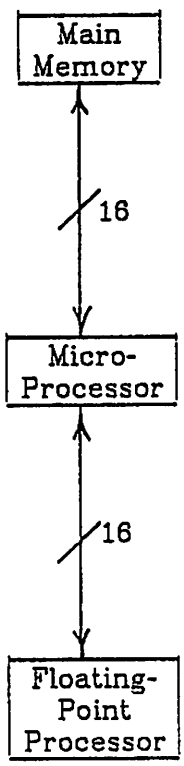


Figure 5.4. Part of Schematic of HP 1000 F-Series Hardware

Operation (1): A = A / B	
Transfer A from main memory to μp 4 16-bit words @ 420ns	1.68μs
Transfer B from main memory to μp 4 16-bit words @ 420ns	1.68μs
Transfer A from up to FPP 4 16-bit words @ 175ns	0.70μs
Transfer B from up to FPP 4 16-bit words @ 175ns	0.70μs
Perform floating-point divide (typical time for operation)	7.00μs
Transfer quotient from FPP to μp 4 16-bit words @ 175ns	0.70μs
Transfer quotient from μp to main memory 4 16-bit words @ 350ns	1.40μs
(6.86μs + 7μs divide time)	13.86μs
Actual microcoded implementation	15.21μs
Note: 5.3μs of 7μs divide time overlapped with instruction fetch and decoding	

Figure 5.5. Optimal Timing for Operation (1)

the time spent in the LESM is approximately 3% of the total analysis CPU time.

5.3.5. Code Generation Costs

The use of code generation has two costs associated with it. First, the instructions for the LESM require additional memory; second, each time a pivoting step is performed additional setup time is required to generate the LESM instructions. Each of these costs is detailed below.

The LESM commands require space in main memory in order that the instruction fetch cycle of the LESM be fast. The minimum space required for the LESM instructions is that memory needed for the MEMO array, which contains the location of each matrix coefficient in the order it is referenced in the equation solution process. The instruction set for the LESM achieves this minimum by encoding the operation code in the most-significant, unused bits in each word. Table 5.4 shows the total memory requirements and the memory needs for the LESM instructions for several representative benchmark circuits. The 'total' memory needs shown in the table are for transient analyses of the indicated benchmark circuits, with 32-bit floating-point arithmetic. The LESM instructions require an average of 18% of the total memory needs of SPUDS.

The second cost of using code generation is the CPU time required to generate the code. A symbolic Gaussian elimination is performed, and at each step the LESM operation which performs the desired arithmetic functions is added to an array of 'machine instructions' for later execution by the LESM. This generation time is relatively small, due to the simple, regular structure of the LESM instructions. Table 5.5 shows, for the benchmark circuits, the code generation time and the total CPU time required for dc

Circuit	Total Memory	LESM instructions	LESM Percentage
DIFPAIR	2364	289	12
UA741	8152	1798	22
MECLIII	6476	1482	23
BOOTINV	2328	170	7
MOSAMP2	7912	1042	13

Table 5.4. Transient Analysis LESM Memory Requirements

Circuit	'Code gen'	DCX	DCOP	TRAN
DIFPAIR	0.73	8.21	1.01	11.97
KTEST	0.78	0.00	0.68	15.68
RCA304	1.83	18.56	1.82	34.88
UA709	3.82	34.45	3.50	41.11
UA727	5.03	37.02	4.83	56.93
UA733	1.06	0.00	3.71	0.00
UA741	3.97	40.87	4.60	48.46
RTLINV	0.45	5.60	0.36	10.66
TTLINV	1.42	13.84	0.90	35.19
TTL74	1.39	14.32	0.86	40.08
TTL74S	1.80	17.10	1.19	40.46
TTL74L	1.39	16.30	1.11	37.79
TTL920	1.66	19.35	0.88	39.48
ECLGATE	1.99	17.12	0.79	46.13
MECLIII	2.99	24.52	1.45	60.73
SBDGATE	3.54	34.55	2.12	62.18
CCSOR	0.32	0.00	1.56	0.00
DCOSC	0.44	0.00	2.05	0.00
CFFLOP	0.35	0.00	1.49	0.00
STCRC	0.12	0.00	0.18	3.87
CHOKE	0.26	0.00	0.07	6.46
ECLINV	0.86	0.00	1.52	17.88
SCHMITT	0.88	0.00	1.60	19.74
ASTABLE	0.60	0.00	1.06	26.54
SATINV	0.14	18.86	1.40	0.00
DEPLINV	0.08	6.02	0.47	0.00
RATLOG	0.77	0.00	1.51	36.49
INVCHN	0.26	0.00	2.16	0.00
BOOTINV	0.41	0.00	1.47	21.03
MOSMEM	0.54	0.00	1.52	53.47
MOSAMP1	1.34	57.80	5.65	0.00
MOSAMP2	2.19	0.00	16.96	178.49

Table 5.5. Costs to Generate LESM Code

transfer curve, dc operating point, and transient analyses. In almost all cases the code generation module is invoked twice, once when the dc transfer curve analysis is begun and a second time at the start of transient analysis. The data in the table indicate that the code generation time is a negligible part of the total CPU-time requirements as long as any multi-point analysis is performed. Only in the case of a single dc operating-point analysis (for example, the UA733 benchmark circuit) is the code generation time as much as 30% of the total. If code generation is *not* used in the simulation of the UA733, the execution time for a dc operating point analysis is 3.73 seconds (compared to 3.71 seconds *with* code generation). The total simulation time is not increased by the code generation step even when only a short analysis is performed.

5.3.6. Growth Rates

The growth rate for the number of terms in the coefficient matrix and the number of arithmetic operations necessary to perform Gaussian elimination is greater than linear in terms of the number of circuit equations. A sampling of statistically generated coefficient matrices reported in [Nage75] shows that for N equations the number of nonzero matrix elements (including fill-in terms) is generally proportional to $N^{1.1}$ and the number of arithmetic operations is proportional to $N^{1.24}$. Since the remainder of the analysis effort is directly proportional to the number of circuit elements, it is clear that for a sufficiently large circuit the equation solution time can become the dominant part of the analysis.

The effects of defining the Linear Equation Solution Machine in microcode do not change this power law for the arithmetic needed to solve the circuit equations; however, the constant of proportionality is reduced

significantly. Table 5.6 presents the results of several transient analyses which measure the importance of this power law for circuit sizes of interest. In all cases, the only variable which changes among the different analyses is the number of circuit equations. This change is effected by changing the number of nonzero extrinsic device-model resistances used to model the bipolar junction or MOS transistors in the circuits. Since the device model evaluation and matrix loading code evaluates and loads these resistances into the coefficient matrix regardless of whether or not the resistances are nonzero, the only change in computational effort is directly related to the differing number of equations.

The 'DDCMP' column in the table gives the CPU time required for the very first LU decomposition of the coefficient matrix. The determination of fill-in terms in the matrix and any equation reordering (pivoting) is performed at this time; code generation time is included in the DDCMP total. As shown in the table, the matrix setup time grows approximately as N^2 primarily due to the full numerical pivoting which is performed as part of the initial determination of the matrix structure. However, once setup is finished, the time per Newton iteration ('CPU/NI') increases by only a few percent as the number of equation is increased by as much as a factor of three.

Several SPY results obtained by examining the execution of SPUDS are shown in Table 5.7. The data show the percentage of the total analysis CPU time spent in the ITER8 and CODEX subroutines during a dc transfer curve analysis of the UA741 and transient analyses of the UA741 and MOSAMP2 benchmark circuits with differing numbers of nodes (circuit equations). The ITER8 subroutine controls each Newton iteration. The only part of the itera-

Circuit	# eqns	DDCMP	TRAN	CPU/NI	% incr
UA741	52	6.08	48.45	0.198	
	74	9.92	54.11	0.207	4.5
	96	14.24	59.44	0.211	6.6
MOSAMP2	25	2.60	187.15	0.357	
	52	6.63	194.88	0.368	3.1
	79	12.27	230.31	0.379	6.2
MOS AMP 3	42	6.70	69.21	0.496	
	77	13.93	82.12	0.512	3.2
	112	23.12	98.57	0.517	4.2

Table 5.6. Effect of Number of Equations on Solution Time

Circuit	Number of equations	% of analysis time			
		DC Xfer		Tran	
		ITER8	CODEX	ITER8	CODEX
UA741	52	16	6	8	3
	74	24	10	10	4
	96	32	13	12	5
MOSAMP2	25			3	1
	52			6	3
	79			7	3

Table 5.7. Equation Solution Time with LESM

tion work performed directly by this subroutine is the node-voltage test for convergence. (The loading of the coefficient matrix is performed by other subroutines and the load time is not included in the ITER8 time.) The equation solution is determined by the CODEX subroutine (in microcode); the CPU time for solution is also not part of the ITER8 time in the table. The data show that the node-voltage convergence test accounts for as much as 32% of the total analysis time when the number of equations is large compared with the number of semiconductor devices in the circuit. This large fraction is due to the strategy adopted as part of the implementation of numerical pivoting. The effect of column-swapping in the coefficient matrix is to change the order of the circuit unknowns. Two methods can be used to keep track of this reordering. First, a 'swap vector' can be maintained which records the 'new' location of each circuit unknown, and the vector obtained from the equation solution step can be reordered based on the 'swap vector' after each iteration. In this way, if an element is connected to node number two, and the equation for that node is initially assigned as equation number two, the program can fetch the node voltage from the second location in the solution vector regardless of any pivoting. The second method to keep track of reordering is to go through the entire element list and replace equation numbers appropriately every time pivoting is performed.

The first method (use of a 'swap vector') is used in SPUDS based on the assumption that pivoting is performed frequently. Empirically, only one pivoting step is necessary per analysis; therefore, the data in the table indicate that the second method is preferable because it trades off a slight increase in setup time (to update element equation numbers) for the larger time required to reorder the solution vector at each iteration. In either case, however, the effect of the special equation solution machine (LESM) is

to make the time spent in solving the linearized circuit equations an almost-negligible part of the total, at least for circuit sizes on the order of 100 equations.

5.4. 'Gather' and Matrix Loading

After the incorporation of the special LU decomposition machine into SPUDS, several other parts of the analysis become relatively more important. To obtain the maximum performance from the existing minicomputer as an approximation to the speed possible with dedicated hardware, several special-purpose instructions are described which rapidly gather data together from scattered locations in memory and incrementally load terms into the coefficient matrix using the sparse-matrix pointer system. These instructions decrease the transient analysis time by an additional 25% over the speedup already obtained with the microcoded LESM. Each of these instructions is described in the following subsections.

5.4.1. Gather

For reasons of storage efficiency, device model parameters in SPUDS are stored in linked-list data structures as described in Chapter 4. These parameter values are retrieved from that structure and stored in local temporary variables preparatory to evaluation of the device model equations. Figure 5.6 shows part of a typical sequence of FORTRAN statements which perform this retrieval task.

The need for high accuracy in device modelling has lead to a large number of device model parameters in many circuit simulators. In Version E.3 of Program SPICE2, the bipolar junction transistor model is described with 29 parameters and the MOS transistor model has 32 possible parame-

```
BETA     = ARRAY ( LOCM+ 6 )
GAMMA    = ARRAY ( LOCM+ 7 )
PHI      = ARRAY ( LOCM+ 8 )
XLAMDA   = ARRAY ( LOCM+ 9 )
COX      = ARRAY ( LOCM+10 )
XNSUB    = ARRAY ( LOCM+11 )
XNFS     = ARRAY ( LOCM+12 )
UO       = ARRAY ( LOCM+13 )
VBP      = ARRAY ( LOCM+14 )
UEXP     = ARRAY ( LOCM+15 )
UTRA     = ARRAY ( LOCM+16 )
XD       = ARRAY ( LOCM+17 )
VMAX     = ARRAY ( LOCM+18 )
XNEFF    = ARRAY ( LOCM+19 )
           . . .

CALL GRESC ( environment, LOCM+6, BETA, GAMMA, PHI, XLAMDA, ... )
```

Figure 5.6. Calling Sequence for GRESC

ters. In addition, there are several derived model parameters which are computed by the program when the circuit description is read in order to save calculation effort during model evaluation.

As a result of this modelling effort, a considerable amount of CPU time is required just to 'gather together' these parameters into local variable storage so that the evaluation of the actual model equations is executed efficiently. SPY monitoring of SPUDS shows that ≈15% of the total analysis time is spent in gathering together these parameter values. To put this percentage in perspective, the time spent in evaluation of the model equations, e.g., the three-halves power law equation for MOSFET drain current, is only 10-20% of the total.

In order to reduce this time, the special-purpose instructions GIESC and GRESC are used. Both instructions are invoked in the same way as FORTRAN subroutines; Figure 5.6 also shows a typical parameter calling sequence. The 'environment' parameter is used to preserve the initial machine environment for restoration when the instruction completes. The only difference between the two instructions is that GIESC transfers integer values and GRESC transfers real (floating-point) values. Both instructions implement an 'indexed block transfer' capability which copies a block of memory to scattered locations in main memory under the control of a vector of destination addresses. For the HP 1000, these instructions reduced the time required to gather parameters together by a factor of 3.

5.4.2. Matrix Load

Once the linearized branch values have been determined as a result of device model evaluation, the equivalent conductances must be added incrementally to the appropriate locations in the coefficient matrix. In Program

SPICE2, this matrix load is performed with program code similar to Figure 5.7. The one-dimensional matrix storage begins at the location indicated by the variable LVN. The per-element data storage includes the one-dimensional offsets from LVN for the locations of each matrix coefficient to which the element contributes. The number of such terms depends on the complexity of the mathematical model for each element. A resistor contributes to four matrix locations, and the bipolar junction and MOS transistor device models contribute to 18 and 22 matrix locations, respectively.

The data obtained by running SPY show that the relative amount of time spent in matrix loading is of the same order of magnitude as the time spent in evaluating the device model equations. Approximately 20% of the total analysis time is spent performing the loading task. The following paragraphs describe several special-purpose instructions which reduce significantly the time needed for the matrix loading part of the analysis.

The mapping from the conceptual two-dimensional coefficient matrix to the packed, one-dimensional vector representation is performed by searching through the row- and column-linked lists as described in Chapter 4. For efficiency reasons, Program SPICE2 performs this mapping each time the structure of the coefficient matrix is changed, and pointers to the locations of each affected matrix entry are stored with each circuit element. This technique avoids the mapping costs that would otherwise be incurred with each matrix coefficient load. The first version of matrix loading microcode in SPUDS used the same load procedure and was named MATAD; the calling sequence is also shown in Figure 5.7. Consecutive locations in the IVAL array contain the offsets to the affected matrix locations for each element. Speed increases due to the much-reduced number of accesses to main memory

```
C
C  LOAD Y MATRIX FOR BIPOLAR JUNCTION TRANSISTORS
C
      LOCY=LVN+IVAL(LOC+10)
      VALUE(LOCY)=VALUE(LOCY)-GCPR

      LOCY=LVN+IVAL(LOC+11)
      VALUE(LOCY)=VALUE(LOCY)-GBPR

      LOCY=LVN+IVAL(LOC+12)
      VALUE(LOCY)=VALUE(LOCY)-GEPR

      LOCY=LVN+IVAL(LOC+13)
      VALUE(LOCY)=VALUE(LOCY)-GCPR

      LOCY=LVN+IVAL(LOC+14)
      VALUE(LOCY)=VALUE(LOCY)-GMU+GM

      LOCY=LVN+IVAL(LOC+15)
      VALUE(LOCY)=VALUE(LOCY)-GM-GO

      LOCY=LVN+IVAL(LOC+16)
      VALUE(LOCY)=VALUE(LOCY)-GBPR

      LOCY=LVN+IVAL(LOC+17)
      VALUE(LOCY)=VALUE(LOCY)-GMU

      ...
      ...

      LOCY=LVN+IVAL(LOC+28)
      VALUE(LOCY)=VALUE(LOCY)+GBPR+GPI+GMU

      LOCY=LVN+IVAL(LOC+29)
      VALUE(LOCY)=VALUE(LOCY)+GPI+GEPR+GM+GO
```

```
CALL MATAD ( environment, LVN, LOC+10, -GCPR, -GBPR, -GEPR, ... )
```

Figure 5.7. SPUDS Microcode to Load Matrix

required by the MATAD instruction.

In order to determine the possible advantages of embedding knowledge of the actual matrix pointer structure in hardware, a special microcoded instruction is used to resolve an (i,j) matrix address into its one-dimensional vector location. Careful measurement of elapsed CPU time shows that the microcoded mapping effort requires $\approx 2\mu s$ for each matrix link traversed. Since the average matrix row or column contains 2-4 nonzero off-diagonal entries [Nage75] the average total mapping time is 4-8μs (not including the 83μs overhead required to invoke the special instruction).

This very small amount of time suggests a revised matrix loading instruction, MLODA, which incorporates the matrix mapping operation as an integral part. This instruction is invoked as shown in Figure 5.8. An argument list of indefinite length is passed to the microcode for each call. The first argument is used to preserve the initial machine environment for later restoration when the matrix load is completed. The next 8 arguments pass to MLODA the addresses of the tables which contain the row- and column-swapping pivoting information, pointers to the linked-list structure used to represent the coefficient matrix, and the starting memory location of the one-dimensional vector containing the matrix coefficient values. The remaining arguments are interpreted as (Row, Column, Value) triples, with the meaning that Value is to be incrementally added to the matrix coefficient at location (Row, Column). In order to obtain the fastest possible load, a second instruction, MLODS, is also defined with meaning identical to that of MLODA except that the Value is subtracted from the matrix coefficient. The MLODA instruction increased analysis time by less than 3% relative to the time required with the MATAD instruction. The change is justified by the substan-

```
CALL MLODA ( environment, augmented_MNA_flag,
1  IRSWPR, ICSWPR, IRPT, IROWNO, JCPT, JCOLNO, LVN,
2  NODE1,NODE1, GCPR,
3  NODE2,NODE2, GBPR,
4  NODE3,NODE3, GEPR,
5  NODE4,NODE4, GMU+GO+GCPR+GCCS,
6  NODE5,NODE5, GBPR+GPI+GMU,
7  NODE6,NODE6, GPI+GEPR+GM+GO  )
```

Figure 5.8. Matrix Load with Mapping Incorporated

tial memory savings and the increase in clarity of the program code.

The right-hand-side (RHS) elements in the coefficient matrix are loaded with another special instruction. The RHS terms are stored as a contiguous one-dimensional vector and are addressed directly by equation number without the need for any mapping algorithm. The calling sequence for this LDRHS instruction is shown in Figure 5.9. The argument list for LDRHS is also of indefinite length. After several arguments defining the microprogramming machine state and the location of the start of the RHS vector, the remaining arguments are interpreted as (Number, Value) pairs, with the meaning that each Value is to be incrementally added to the RHS term for the corresponding equation Number.

5.4.3. Matrix Initialization

One further special instruction reduces part of the analysis time by a significant amount but is a result of the software and hardware design of the HP 1000. At the start of each Newton iteration, all of the matrix coefficients are initialized to zero. In SPUDS, these coefficients are stored in the 'Extended Memory Area' (EMA) memory region to allow a circuit containing over 100 devices to be simulated. However, the memory addressing architecture of the HP 1000 minicomputer makes access to EMA locations much more expensive than references to main memory. The overhead factor ranges between 5 and 20 depending on the datatype (integer, real, or double-precision). This overhead is particularly noticeable for the dc analyses. SPY data shows that matrix initialization in the dc transfer curve analysis of the UA741 requires 16% of the total CPU time. A special microcoded instruction, SRCBE, eliminates virtually all of this initialization time in SPUDS through microcoded control of EMA access.

VALUE (LVN + NODE4) = VALUE (LVN + NODE4) – CEQCS + CEQBC

VALUE (LVN + NODE5) = VALUE (LVN + NODE5) – CEQBE – CEQBC

VALUE (LVN + NODE6) = VALUE (LVN + NODE6) + CEQBE

```
      CALL LDRHS ( environment, LVN,
     1  NODE4, -CEQCS + CEQBC,
     2  NODE5, -CEQBE - CEQBC,
     3  NODE6,  CEQBE       )
```

Figure 5.9. Coding to Load Right-Hand-Side Terms

5.5. Revised CPU Times and Conclusions

It is important when evaluating speed improvements to determine the extent to which overall performance is degraded by poor code generation from language compilers. As shown in Section 5.3.1 and Table 5.2, the FORTRAN compiler for the HP 1000 produces acceptable code. Hand-written assembly language routines for equation solution reduce total analysis time by only 10%.

The use of the special-purpose instructions described in the section above affects two aspects of the simulation. First and most importantly, the total analysis time is reduced by almost 70% relative to the original, all-FORTRAN implementation of SPUDS. Second, the relative percentages change for the different tasks within the circuit analysis. Each of these aspects is described in detail in the following paragraphs.

Table 5.8 shows the CPU time requirements for dc transfer curve and transient analyses of the benchmark circuits. The simulations for the first three column pairs use 64-bit floating-point arithmetic; for the last column pair, the simulation code utilizes 32-bit arithmetic. The columns headed 'DCX' give the CPU time in seconds for the dc transfer curve analyses, and the columns headed 'TRAN' contain the times for transient analyses. The first pair of columns contain baseline data for the original, all-FORTRAN version of SPUDS. The second column pair gives the simulation times when the microcoded linear equation solution machine (LESM) is used in place of the FORTRAN code. The CPU times in the third pair of columns reflect the use of the LESM together with the GIESC, GRESC, IJTOK, MLODA, LDRHS, and SRCBE special instructions for gathering together device model parameters, mapping matrix locations, loading the matrix coefficients and the right-hand-side

Circuit	< - - - - - 64-bit arithmetic - - - - - >						32-bit math	
	all FORTRAN		LESM		Gather, *et al*		pivoting and microcode	
	DCX	TRAN	DCX	TRAN	DCX	TRAN	DCX	TRAN
DIFPAIR	21.37	27.94	16.75	22.55	11.33	16.23	8.21	11.97
KTEST	0.00	12.40	0.00	8.60	0.00	7.27	0.00	15.68
RCA3040	52.52	84.99	37.96	64.64	23.59	45.80	18.56	34.88
UA709	114.64	140.33	64.62	90.75	38.52	62.54	34.45	41.11
UA727	105.23	198.68	68.31	136.31	40.80	95.29	37.02	56.93
UA733	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
UA741	136.71	154.22	85.02	114.12	50.71	81.35	40.87	48.46
RTLINV	14.42	25.06	10.71	19.38	7.51	14.04	5.60	10.66
TTLINV	41.28	114.33	26.93	78.54	17.51	56.11	13.84	35.19
TTL74	42.68	99.55	27.67	69.27	17.98	49.77	14.32	40.08
TTL74S	50.88	132.84	32.22	87.51	20.50	61.20	17.10	40.46
TTL74L	46.39	123.75	29.62	84.64	19.07	60.37	16.30	37.79
TTL9200	54.13	0.00	31.89	0.00	20.03	0.00	19.35	39.48
ECLGATE	47.18	116.60	31.40	81.67	19.53	57.87	17.12	46.13
MECLIII	73.61	150.53	46.30	101.54	28.31	71.00	24.52	60.73
SBDGATE	84.94	193.47	51.77	133.07	32.93	92.10	34.55	62.18
CCSOR	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
DCOSC	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
CFFLOP	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
STCRC	0.00	7.57	0.00	6.16	0.00	5.27	0.00	3.87
CHOKE	0.00	16.86	0.00	13.58	0.00	11.18	0.00	6.46
ECLINV	0.00	47.95	0.00	36.53	0.00	26.79	0.00	17.88
SCHMITT	0.00	52.63	0.00	39.04	0.00	28.38	0.00	19.74
ASTABLE	0.00	88.73	0.00	60.13	0.00	43.37	0.00	26.54
SATINV	40.43	0.00	34.67	0.00	24.33	0.00	18.86	0.00
DEPLINV	13.15	0.00	11.61	0.00	8.51	0.00	6.02	0.00
RATLOG	0.00	95.30	0.00	85.01	0.00	64.13	0.00	36.49
INVCHN	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
BOOTINV	0.00	44.72	0.00	41.13	0.00	32.34	0.00	21.03
MOSMEM	0.00	115.99	0.00	108.20	0.00	81.45	0.00	53.47
MOSAMP1	150.19	0.00	127.32	0.00	81.60	0.00	57.80	0.00
MOSAMP2	0.00	399.69	0.00	351.61	0.00	267.01	0.00	178.49

Table 5.8. Performance Comparisons

terms, and initializing the matrix coefficients. The simulations for the last column pair use numerical pivoting, 32-bit arithmetic, and all the special instructions of the previous columns.

On the average, the result of all the microcoding of high-level simulation operations is a reduction in the total analysis time of 65-70%. The importance of high-level special-purpose instructions (or hardware) is shown further by the data in Table 5.9. All the FORTRAN code in SPUDS access the EMA data area on the HP 1000 by using microcode supplied by Hewlett-Packard. A more recent version of that microcode, which access EMA variables in half the time, is used for the 32-bit arithmetic SPUDS data. The revised HP microcode does not affect the special-purpose instructions included in SPUDS. As the data in the table show, the faster EMA microcode is responsible for only 5% of the 65-70% overall reduction.

Another perspective on simulation speed is shown in Table 5.10, which gives the simulation times for both the microcoded version of SPUDS and simulation runs of Version E.3 of Program SPICE2 run on the CDC 6400. (The speed of the CDC 6400 computer is comparable to that of the DEC VAX 11/780.) Without any of the changes described in this report, transient analysis with SPUDS on the HP 1000 F-Series minicomputer is 7 to 10 times slower than SPICE2 on the CDC 6400. With all the modifications, the simulation speed of SPUDS averages 3.6 times slower for dc transfer curve and 2.8 times slower for transient analyses than SPICE2 on the larger computer. The speed ratio varies for different circuits. For transient analysis of the BOOTINV circuit, SPUDS is only 1.5 times slower. Small-signal frequency-domain ac analysis times are not included in the comparison data because no special instructions are implemented in SPUDS for dealing with complex

Circuit	old EMA microcode		new EMA microcode	
	DCX	TRAN	DCX	TRAN
DIFPAIR	9.44	13.65	8.21	11.97
KTEST	0.00	18.37	0.00	15.68
RCA304	21.20	39.64	18.56	34.88
UA709	39.43	46.67	34.45	41.11
UA727	42.71	64.70	37.02	56.93
UA733	0.00	0.00	0.00	0.00
UA741	46.62	54.82	40.87	48.46
RTLINV	6.47	12.20	5.60	10.66
TTLINV	16.01	40.17	13.84	35.19
TTL74	16.53	45.76	14.32	40.08
TTL74S	19.74	46.13	17.10	40.46
TTL74L	18.82	43.05	16.30	37.79
TTL920	22.35	45.00	19.35	39.48
ECLGATE	19.80	52.67	17.12	46.13
MECLIII	28.37	69.30	24.52	60.73
SBDGATE	40.16	71.07	34.55	62.18
CCSOR	0.00	0.00	0.00	0.00
DCOSC	0.00	0.00	0.00	0.00
CFFLOP	0.00	0.00	0.00	0.00
STCRC	0.00	4.49	0.00	3.87
CHOKE	0.00	7.47	0.00	6.46
ECLINV	0.00	20.45	0.00	17.88
SCHMITT	0.00	22.46	0.00	19.74
ASTABLE	0.00	30.33	0.00	26.54
SATINV	21.16	0.00	18.86	0.00
DEPLINV	6.73	0.00	6.02	0.00
RATLOG	0.00	41.60	0.00	36.49
INVCHN	0.00	0.00	0.00	0.00
BOOTINV	0.00	23.85	0.00	21.03
MOSMEM	0.00	60.45	0.00	53.47
MOSAMP1	64.43	0.00	57.80	0.00
MOSAMP2	0.00	200.45	0.00	178.49

Table 5.9. Effects of New (Faster) EMA Access Microcode

Circuit	SPUDS on HP 1000-F		SPICE2 on CDC 6400	
	DCX	TRAN	DCX	TRAN
DIFPAIR	8.21	11.97	2.13	3.92
KTEST	0.00	15.68	0.00	1.52
RCA3040	18.56	34.88	5.02	10.74
UA709	34.45	41.11	9.00	15.02
UA727	37.02	56.93	9.12	22.68
UA733	0.00	0.00	0.00	0.00
UA741	40.87	48.46	11.87	19.22
RTLINV	5.60	10.66	1.39	3.28
TTLINV	13.84	35.19	3.65	12.94
TTL74	14.32	40.08	3.76	11.70
TTL74S	17.10	40.46	4.43	14.26
TTL74L	16.30	37.79	4.02	13.91
TTL9200	19.35	39.48	5.03	13.14
ECLGATE	17.12	46.13	4.08	13.44
MECLIII	24.52	60.73	6.17	16.68
SBDGATE	34.55	62.18	7.04	21.36
CCSOR	0.00	0.00	0.00	0.00
DCOSC	0.00	0.00	0.00	0.00
CFFLOP	0.00	0.00	0.00	0.00
STCRC	0.00	3.87	0.00	.90
CHOKE	0.00	6.46	0.00	2.10
ECLINV	0.00	17.88	0.00	5.80
SCHMITT	0.00	19.74	0.00	6.28
ASTABLE	0.00	26.54	0.00	9.36
SATINV	18.86	0.00	6.05	0.00
DEPLINV	6.02	0.00	1.87	0.00
RATLOG	0.00	36.49	0.00	14.37
INVCHN	0.00	0.00	0.00	0.00
BOOTINV	0.00	21.03	0.00	14.68
MOSMEM	0.00	53.47	0.00	18.76
MOSAMP1	57.80	0.00	20.68	0.00
MOSAMP2	0.00	178.49	0.00	68.68

Table 5.10. Performance Comparisons: microcode SPUDS and SPICE2

arithmetic.

A second effect on simulation performance of the use of microcoded special-purpose instructions is a shifting in the percentages of the total analysis time required by different parts of the simulation. Table 5.11 shows the breakdown by function for the dc transfer curve analysis of the UA741 all-bipolar amplifier benchmark circuit. As the data show, no single function takes as much as 20% of the total; most of the tasks require less than 10% of the analysis time. The table also shows the time breakdown of the dc operating-point analysis for the MOSAMP1, all-MOSFET amplifier circuit. In spite of the considerably more-complex MOS device model, the relative time percentages do not differ greatly, with the exception of the time required to fetch the device model parameters. Approximately 20% of the total analysis time is spent in fetching these parameters, even though this fetch is performed with a special-purpose microcoded instruction. Table 5.12 gives the time breakdown for transient analysis of the UA741; the single greatest amount of time, only 18%, is spent in evaluation of the device model equations. As long as the numerical algorithms and modelling techniques remain essentially the same as those in SPICE2, there is no single step left in the analysis for which special-purpose instructions (or hardware) would make any further major improvements in performance. Small savings result from minor rewriting of parts of the FORTRAN code. For example, rewriting the LTE (local truncation error) estimation code so that the divided-difference approximations are evaluated in-line rather than by iteration decreased transient analysis time by ≈5%. Performance can be improved significantly, of course, by using faster (and more expensive) hardware.

Analysis task	UA741	MOSAMP1
LOAD (init matrix & load R's)	6%	4%
ITER8	16%	4%
solve linearized equations	6%	1%
check voltages for convergence	11%	3%
'Device modelling'	72%	90%
fetch model parameters	7%	20%
predict new sweep point	8%	8%
check bypass of eqn evaluation	2%	5%
compute new branch voltages	8%	9%
limit junction voltages	4%	3%
evaluate model equations	14%	19%
check currents for convergence	6%	6%
load right-hand-side terms	3%	4%
load coefficient matrix	16%	16%

Table 5.11. Final Breakdown of dc Analysis CPU Time

Analysis task	UA741
TERR (estimate local truncation error)	14%
ITER8	8%
solve linearized equations	3%
check voltages for convergence	5%
'Device modelling'	65%
fetch model parameters	6%
predict new sweep point	5%
check bypass of eq'n evaluation	3%
compute new branch voltages	5%
evaluate model equations	18%
integrate capacitor currents	10%
check currents for convergence	5%
load right-hand-side terms	2%
load coefficient matrix	11%

Table 5.12. Final Breakdown of Transient Analysis CPU Time

CHAPTER 6

CONCLUSIONS

The increasing size and complexity of integrated circuits have placed great demands on the tools needed by the design engineer. Timing [Chaw75] [Fan77] [Boyl78], logic [Beni79] [Szyg76] [Wilc76] [Newt79], and even higher-level simulators [Barb77] [Comp74] [Hill80] [Newt80] have been developed which reduce the computer costs of simulation by orders of magnitude relative to circuit-level simulation, at the expense of 10-20% in solution accuracy. At the cell or building-block level, however, the availability of accurate circuit-level simulation is critical for the development of efficient, working circuits. The decreasing costs of minicomputers would make these machines attractive vehicles on which to run such simulations, were it not for their limited performance. The research results described in earlier chapters of this report show that speeds comparable to mainframe computers are obtained from a minicomputer when appropriate changes are made to the numerical algorithms and data structures of the program together with modifications to the machine instruction set.

The key results and conclusions of this report concerning those necessary changes in both program and computer structures are summarized in Section 6.1. The suitability of several currently-available microprocessors or minicomputers is evaluated briefly in Section 6.2. Finally, several limitations of the ideas developed in this report and areas for further research are presented in Section 6.3.

6.1. Key Results

One of the fundamental factors which determines overall simulation speed is the numerical precision used in floating-point calculations. The larger mainframe computers use ≈64 bits without speed penalties. However, the 16- or 32-bit word memories in minicomputers make those smaller machines better suited to 32-bit calculations. Even though the computational effort in IC simulation involves a considerable number of floating-point calculations, there is not a strong need for a direct hardware implementation of floating-point arithmetic when 32-bit precision gives sufficient accuracy. The data presented in Chapter 2 show that the improvement in simulation speed on the HP 1000 F-Series due to hardware floating-point arithmetic is only 15% (although for 64-bit precision the simulation is seven times faster than a microcoded implementation of floating-point arithmetic). Also, the effect of memory cycle time on total simulation speed is reduced when 32-bit precision is used. For the HP 1000, a 33% improvement in memory access speed reduces simulation time by only 15%. (For 64-bit precision, the corresponding reduction in time is 25%.)

Chapter 3 describes the numerical algorithms in SPUDS. Several methods which increase the accuracy of the circuit solution are evaluated. For the circuit sizes of interest (≈100 devices), the use of both numerical pivoting and an augmented MNA matrix is essential to obtain results with 32-bit precision that are the same as for 64-bit precision. Incremental iteration slightly improves the accuracy of the solution. A number of techniques must also be utilized to obtain rapid convergence to a solution. In particular, limiting the maximum *p-n* junction conductance values and using a threshold to reduce the sensitivity to very small changes in *p-n* junction bias are effective

at speeding the rate of convergence. The total computational effort is reduced with the use of linear prediction and model evaluation bypass algorithms; for several of the test circuits, the savings are as much as 20%.

Because of the limit on circuit size, 16 bits are sufficient to store integer quantities, e.g. node numbers. All floating-point variables (and almost all floating-point calculations) do not require more than 32 bits. Therefore, the main-memory-to-CPU data path for a CAD machine should be 32 bits wide and the memory should either have a wordsize of 16 bits and support double-word (32-bit) data transfers, or have a 32-bit wordsize and support fast halfword data access. The total memory requirements depend on the particular machine architecture and instruction set size. For Program SPUDS on the HP 1000, approximately 100KB are required to store program code (instructions); the microcode requires another 800 words of control store. The data reorganization and associated algorithm changes described in Chapter 4 reduce the data storage requirements of SPUDS by a factor of three relative to Version E.3 of SPICE2. As a result, the memory needs for data storage are quite modest; 65KB (32K 16-bit words) are sufficient for a circuit with 100-200 semiconductor devices.

The inclusion in a general-purpose minicomputer of specially-tailored machine instructions such as those described in Chapter 5 significantly improves the performance of circuit simulation. It is important when evaluating improvements in performance to determine the extent to which poor code generation from language compilers affects overall speed. For example, the FORTRAN compiler for the HP 1000 produces acceptable code (hand-written assembly language routines for equation solution reduce total analysis time by only 10%). The development in microcode of a Linear Equa-

tion Solution Machine (LESM) is detailed. This LESM executes ≈20 times faster than equivalent FORTRAN code and reduces the equation solution time to only 2% of the total. Additional speedup is obtained from special instructions which gather together device model parameters and which incrementally load the coefficient matrix. All the microcoded special instructions together reduce the total simulation time by 65-70%.

6.2. Existing Processors

A wide range of processors, from micro-processors such as the 8086 [Inte79] through larger systems such as the M68000 [Moto79] and up to 16-bit general-purpose minicomputers such as the Texas Instruments 990/12 [Texa79] [Appe79] or the HP 1000 are possible candidates on which to build a dedicated CAD machine. This section describes the features and drawbacks of these processors, from the perspective of obtaining good performance in circuit simulation.

6.2.1. 8086

The 8086 CPU is an 8- or 16-bit general-purpose microprocessor having a 16-bit external data path. Standard parts operate with a 200ns clock cycle time, and up to 1MB of central memory can be addressed. The megabyte of memory is divided into logical segments of up to 64KB each. The CPU can directly access 4 of these segments at any one time. By convention, one segment is used for code (program instructions), one for a stack, and two segments for general data storage.

Of particular interest is the 'coprocessor' concept of the 8086. A coprocessor monitors the instructions of the main CPU and can execute those operations which it recognizes as 'its own'. This capability can be used to

extend the instruction set of the main CPU. More importantly, an 8086 system can be augmented (without significant modifications to hardware or software) with a special-purpose coprocessor chip which performs certain functions especially well.

The data registers may be addressed as either 8- or 16-bit storage locations. The only 32-bit datatype recognized by the processor hardware is called 'pointer' and is used to address code or data that is outside the currently-addressable segments. The 32 bits are organized as a 16-bit segment offset and a 16-bit segment base address. The address and data lines on the chip are multiplexed together; this sharing reduces the possible overlap of instruction execution and operand fetch/store.

Some multiprocessor interlock and handshaking capabilities are provided. The LOCK instruction causes a 'lock' signal to be set high for the duration of the following instruction. The XCHG instruction (which exchanges register contents with memory) can be combined with the LOCK instruction to implement a semaphore [Dijk68] facility. The ESC (escape) instruction may be used to initiate an operation in another processor, e.g., the 'coprocessor', and simultaneously pass a 6-bit quantity. Execution may then proceed until the results of the remotely-executed operation are needed, at which time a WAIT instruction can be executed, which causes the 8086 to enter an idle state until a TEST input signal is active.

The instruction set of the 8086 is oriented exclusively around 8-bit bytes and 16-bit words. However, the 8087 Numeric Data Processor (NDP) [Inte80], a manufacturer-supplied coprocessor for the 8086, performs arithmetic and comparison operations on 32- and 64-bit floating-point operands and in addition executes several built-in transcendental functions such as tangent, log,

and exponential. The NDP implements the floating-point arithmetic standard proposed by the IEEE [Coon79]. It contains 8 80-bit registers, which may be individually accessed or may be accessed together as a register stack. The data path in the numeric execution portion of the NDP is 68 bits wide. With a 5MHz clock on the 8087, multiplication and division of 32-bit floating-point numbers require approximately 19μs and 39μs, respectively. In addition, the 'coprocessor' scheme permits the 8087 to perform computations in parallel with operations in the 8086, providing a certain amount of processor parallelism which can increase overall performance.

Overall, the combination of the 8086 and 8087 processors is adequate for the task of circuit-level simulation of small integrated circuits. The availability of multiprocessor interlocking instructions could be used to increase the total computation speed by using many of these processors together in an integrated system. The absence of 32-bit data paths, however, limits the speeds that can be achieved. If it were possible to add a second coprocessor chip (besides the 8087) implementing the special instructions detailed in Chapter 5, simulation speeds should be within a factor of 2 of those obtained with the HP 1000-F. Memory requirements for the 8086 should be essentially the same as those for the 1000.

6.2.2. TI 990/12

The Texas Instruments 990/12 is a byte-addressable 16-bit minicomputer with several high-speed caches which improve its overall performance. Instruction look-ahead is used to perform some of the memory-address mapping overhead before the actual memory access cycles. Up to 2MB of main memory can be addressed, in 64KB segments. Floating-point commands on the 990/12, implemented with microcode, support both 32-bit and 64-bit

arithmetic. Although the ALU contains a 32-bit accumulator, most data paths in the processor are only 16 bits wide.

User-written microcode can be included in the control store of the 990/12. The design of the microprocessor memory on the 990/12 can support up to 4K 64-bit words; however, only 3K words are currently supported. The first 2K are in ROM and contain the microcode for the basic instruction set and some diagnostics. The remaining 1K of control store are in RAM; 528 of these words are available to the user to implement special operations. Access to microcoded operations is through the 'XOP' instruction, which transfers control either to a specified location in main memory or to an address in the microprocessor control store depending on the setting of a processor state bit. Therefore, it is easy to develop special instructions in a high-level language and then implement the instructions with microcode for fast execution.

The 990/12 has several instructions which support inter-processor handshaking in a multiprocessor environment. The performance of circuit simulation on this machine should be similar to the speed obtained with the HP 1000-F, except for the relatively small differences due to the hardware implementation of floating-point arithmetic on the 1000.

6.2.3. M68000

The M68000 [Moto79] is a general-purpose microprocessor which supports operations on individual bits, 8-bit bytes, and 16- and 32-bit words. Both data and address registers may be 32 bits wide, but all internal data paths are only 16 bits wide. Standard parts operate with a 125ns clock cycle time. Up to 16MB of memory can be addressed directly (without segmentation). Both 16- and 32-bit arithmetic for address calculations is supported.

Overall performance is improved with the use of an instruction pre-fetch pipeline and by the fact that the address and data busses are not multiplexed. No floating-point arithmetic is presently documented, although provision for floating-point instruction opcodes has been made. (At least one-eighth of the instruction opcodes have been reserved for future instructions.) A simple 16-bit integer addition requires 9 internal (18 clock) cycles, or 2.25μs.

Multiprocessor communication is supported with the test-and-set (TAS) instruction, which performs a read, modifies the data in the ALU, and writes the data back to the same address in an indivisible hardware cycle. Bus arbitration logic is incorporated in the processor chip for a shared bus and shared memory environment.

The lack of floating-point arithmetic, the inability to customize the microcode, and the 16-bit data paths severely limit the potential analysis speed of this processor. The simulation times for the 68000 as presently available are likely to be 10-20 times longer than the times for the HP 1000-F.

6.2.4. Intel 432

As of the time this report is prepared, the Intel 432 chip set [Lati81] has just been announced. Address space is no problem, as the 432 can address 2^{32} bytes of memory. The processor supports 32-, 64-, and 80-bit floating-point arithmetic, with operation times on the order of 10–30μs. It appears that simulation speeds on the 432 should be within a factor of 2-3 of those on the HP 1000-F.

6.3. Limitations and Further Research

The research results described in the previous chapters support the idea of a desktop 'CAD machine' to meet many of the circuit-level simulation needs of the IC design engineer. If this CAD tool is to be effective, it must provide simulation results in minutes, for circuits containing 100-200 semiconductor devices. The simulation speed obtained with the HP 1000-F' minicomputer meets this criterion; therefore, the machine can be used in the design of cells (building blocks) in VLSI circuits.

The increasing size of VLSI circuits will lead to building-blocks that are too large for the methods described in this report. If circuit-level simulation is to remain a useful design tool, further research into algorithms and implementation details is needed so that the speed and power of the simulation program keeps up with the increasing size of integrated circuits.

Without a true dataflow architecture, the problems and time delays in communicating data and control information among multiple processors, e.g. a host and co-processor, make a multiprocessor system unattractive to provide a dedicated fast simulation capability for the circuit designer. A greater speed improvement can be obtained from a single-processor system in which the processor instructions are tailored to the problem area of interest.

The fastest possible execution is obtained by a direct hardware implementation of the simulation algorithms. The use of firmware (microcode) in SPUDS to implement the special-purpose instructions compromises this maximum speed in favor of increased flexibility in instruction design. Even better performance would be obtained if the microcoded tasks were performed by special-purpose hardware incorporated into a processor. The

results in this report can be used as a starting point for the development of that hardware.

APPENDIX 1

SPUDS PROGRAM STRUCTURE

The SPUDS program is written in a combination of FORTRAN-IV, assembler, and micro-assembler code. The program contains approximately 12000 FORTRAN, 400 assembler, and 1500 micro-assembler statements. SPUDS runs on a Hewlett-Packard 1000 F-Series minicomputer operating with an extended version of the manufacturer's RTE-IVa operating system[1]. The SPUDS program code is divided into a root segment, 15 first-level overlays, and an auxiliary helper program SPUDZ. The circuit to be analyzed is stored in a set of linked lists to make the maximum use of available memory. Data structures are dynamically allocated and deallocated from main memory as the analysis proceeds.

The original coding for Program SPUDS is derived from Version E.3 of Program SPICE2, and the logical flows of control in the two programs are similar to one another. The changes in the numerical algorithms and data structures of SPUDS relative to Program SPICE2 are described in the body of this report. The basic organization of SPICE2 is described in detail in [Nage75] and [Cohe76]. This appendix gives the structural changes in SPUDS which have been found necessary to make SPUDS run effectively on the Hewlett-Packard 1000 F-Series minicomputer.

[1]The operating system is extended with the Session Monitor software package developed by the Automatic Measurement Division of the Hewlett-Packard Company.

The most severe constraint imposed by the minicomputer is the very limited address space for program code. The initial (and reference) version of SPUDS uses 64-bit arithmetic. Although memory requirements for both program code and data are reduced when 32-bit floating-point arithmetic is used, substantial revisions in program structure are required to make the 64-bit version fit in the minicomputer. The 26K 16-bit words of program code space on the HP 1000 are barely adequate for the code which implements the dc operating point, small-signal frequency-domain, and large-signal time-domain (transient) analyses. The program statements which implement the other analysis capabilities of SPICE2 (small-signal distortion, element sensitivities, etc.) are not in SPUDS both due to address space limitations and because the computational effort associated with these other types of analysis is small. The code for the nonlinear controlled source, transmission line, and JFET circuit elements is also removed from SPUDS to meet the constraints imposed by the limited memory address space. The code has been removed by changing the affected statements to comments, rather than by deleting lines from the program source. Finally, the input-processing and error-checking overlays READIN and ERRCHK are each split into two smaller overlays, also to reduce memory requirements.

The largest program overlay in Program SPICE2 is DCTRAN, which controls the dc and transient analyses. Several changes to this program structure are made in SPUDS to make the program code fit in the available address space. First, all FORTRAN WRITE statements in this overlay are moved to overlay DCFMT, which is invoked immediately after the analysis completes. This change eliminates the need for approximately 2.5K words of FORTRAN library routines in the analysis overlay. Second, the size of the device modelling routines makes it impossible to have all the routines present in

memory at the same time. Therefore, in addition to the elimination of the JFET modelling noted above, the DCTRAN overlay is replaced by the two overlays DCTRB and DCTRM. The DCTRB overlay controls the dc and transient analyses for circuits whose only semiconductor devices are *p-n* junction diodes and BJTs; the DCTRM overlay performs the same function for MOSFET circuits. Circuits which contain both BJTs and MOSFETs cannot be analyzed by SPUDS.

The size of the 'LEVEL=2' MOSFET modelling routines exceeds the available memory address space even after all these modifications. In order to extend the address space so that the MOSFET device subroutines fit in main memory, the subroutines are loaded as part of a second program named SPUDZ. From the perspective of the RTE-IVa operating system, SPUDS is actually the two programs SPUDS and SPUDZ. SPUDS begins by executing an operating system call which locks the program in memory so that it cannot be swapped out to disc. Then SPUDS determines which pages of physical memory contain the values of the FORTRAN COMMON-block variables, and which physical page begins the Extended Memory Area (EMA) used for data storage. The SPUDS program then invokes Program SPUDZ. SPUDZ begins by locking itself in memory and then changes the Dynamic Mapping System (DMS) registers so that references in SPUDZ to either COMMON-block variables or EMA data reference the same physical memory locations as SPUDS. After this initialization SPUDZ suspends itself and returns control back to SPUDS. Whenever SPUDS needs to execute the MOSFET device subroutines, the SPUDZ program is invoked by a call to the operating system. Since both programs are always present in main memory, the additional execution time incurred by this strategy is a negligible part of the total analysis time.

APPENDIX 2

SPUDS PROGRAM LISTING

Program SPUDS is written for the Hewlett-Packard 1000 F-Series mini-computer using a combination of FORTRAN-IV, assembler, and micro-assembler code. Persons who wish to obtain a listing of Program SPUDS should contact the author concerning the possibility of obtaining a copy.

APPENDIX 3

LISTING OF BENCHMARK CIRCUITS

The effectiveness of the simulation algorithms described in this report can be measured by how well these algorithms work for typical integrated circuits. A mixture of both analog and digital bipolar test circuits exhibiting simulation problems such as convergence in dc analysis is described in [Nage75]. These circuits are part of the set of benchmarks used for Program SPUDS. Several analog and digital MOSFET circuits are included as well to check the effectiveness of the simulation algorithms when the circuit response is controlled by MOSFETs.

A listing of the input for the DIFFPAIR circuit is shown in Figure A3.1. Figure A3.2 shows the output listing of SPUDS for the circuit input from Figure A3.1. The input descriptions of the other test circuits are shown in Figures A3.3 through A3.34.

```
DIFFPAIR CKT - SIMPLE DIFFERENTIAL PAIR
.WIDTH IN=72 OUT=72
.OPT ACCT OPTS PIVTOL=1.0E-12 PIVREL=1.0E-4
.TF V(5) VIN
.DC VIN -0.25 0.25 0.005
.AC DEC 10 1 10GHZ
.TRAN 5NS 500NS
VIN 1 0 SIN(0 0.1 5MEG) AC 1
VCC 8 0 12
VEE 9 0 -12
Q1 4 2 6 QNL
Q2 5 3 6 QNL
RS1 1 2 1K
RS2 3 0 1K
RC1 4 8 10K
RC2 5 8 10K
Q3 6 7 9 QNL
Q4 7 7 9 QNL
RBIAS 7 8 20K
.MODEL QNL NPN(BF=80 RB=100 CCS=2PF TF=0.3NS
+   TR=6NS CJE=3PF CJC=2PF VA=50)
.PRINT DC V(4) V(5)
.PLOT DC V(5)
.PRINT AC VM(5) VP(5)
.PLOT AC VM(5) VP(5)
.PRINT TRAN V(4) V(5)
.PLOT TRAN V(5)
.END
```

Figure A3.1. SPUDS Input Listing for Differential-Pair Circuit

```
*****  03AUG80  ********  SPUDS 1A.0 (03AUG80)  ******** 21:02:57 *****
DIFFPAIR CKT - SIMPLE DIFFERENTIAL PAIR
     INPUT LISTING                    TEMPERATURE =   27.000 DEG C
**********************************************************************
.WIDTH IN=72 OUT=72
.OPT ACCT OPTS PIVTOL=1.0E-12 PIVREL=1.0E-4
.TF V(5) VIN
.DC VIN -0.25 0.25 0.005
.AC DEC 10 1 10GHZ
.TRAN 5NS 500NS
VIN 1 0 SIN(0 0.1 5MEG) AC 1
VCC 8 0 12
VEE 9 0 -12
Q1 4 2 6 QNL
Q2 5 3 6 QNL
RS1 1 2 1K
RS2 3 0 1K
RC1 4 8 10K
RC2 5 8 10K
Q3 6 7 9 QNL
Q4 7 7 9 QNL
RBIAS 7 8 20K
.MODEL QNL NPN(BF=80 RB=100 CCS=2PF TF=0.3NS TR=6NS CJE=3PF CJC=2PF
+    VA=50)
.PRINT DC V(4) V(5)
.PLOT DC V(5)
.PRINT AC VM(5) VP(5)
.PLOT AC VM(5) VP(5)
.PRINT TRAN V(4) V(5)
.PLOT TRAN V(5)
.END


*****  03AUG80  ********  SPUDS 1A.0 (03AUG80)  ********  21:02:57  *****
DIFFPAIR CKT - SIMPLE DIFFERENTIAL PAIR
     BJT MODEL PARAMETERS             TEMPERATURE =   27.000 DEG C
**********************************************************************
               QNL
TYPE           NPN
BF            80.000
BR             1.000
IS          1.00E-14
RB           100.000
VA            50.000
TF          3.00E-10
TR          6.00E-09
CCS         2.00E-12
CJE         3.00E-12
CJC         2.00E-12
```

Figure A3.2. SPUDS Output for Differential-Pair Circuit (Page 1 of 7)

```
***** 03AUG80 ******** SPUDS 1A 9 [illegible] ******** [illegible] 5* *****
DIFFPAIR CKT - SIMPLE DIFFERENTIAL PAIR
     DC TRANSFER CURVES                TEMPERATURE =   27.000 DEG C
**********************************************************************
  VIN          V(5)
                      -5.000E+00          0.000E+00          5.000E+00          1.000E+01
-2.500E-01   -7.013E-01
-2.450E-01   -6.998E-01
-2.400E-01   -6.980E-01
-2.350E-01   -6.961E-01
-2.300E-01   -6.940E-01
-2.250E-01   -6.918E-01
-2.200E-01   -6.893E-01
-2.150E-01   -6.865E-01
-2.100E-01   -6.839E-01
-2.050E-01   -6.809E-01
-2.000E-01   -6.776E-01
-1.950E-01   -6.742E-01
-1.900E-01   -6.707E-01
-1.850E-01   -6.670E-01
-1.800E-01   -6.632E-01
-1.750E-01   -6.590E-01
-1.700E-01   -6.548E-01
-1.650E-01   -6.504E-01
-1.600E-01   -6.459E-01
-1.550E-01   -6.410E-01
-1.500E-01   -6.361E-01
-1.450E-01   -6.310E-01
-1.400E-01   -6.255E-01
-1.350E-01   -6.197E-01
-1.300E-01   -6.138E-01
-1.250E-01   -6.074E-01
-1.200E-01   -6.006E-01
-1.150E-01   -5.932E-01
-1.100E-01   -5.852E-01
-1.050E-01   -5.761E-01
-1.000E-01   -5.656E-01
-9.500E-02   -5.524E-01
-9.000E-02   -5.342E-01
-8.500E-02   -5.081E-01
-8.000E-02   -4.809E-01
-7.500E-02   -1.816E-01
-7.000E-02    4.073E-02
-6.500E-02    2.869E-01
-6.000E-02    5.350E-01
-5.500E-02    8.440E-01
-5.000E-02    1.139E+00
-4.500E-02    1.493E+00
-4.000E-02    1.848E+00
-3.500E-02    2.221E+00
-3.000E-02    2.610E+00
-2.500E-02    3.014E+00
-2.000E-02    3.430E+00
-1.500E-02    3.856E+00
-1.000E-02    4.289E+00
-5.000E-03    4.727E+00
 0.000E+00    5.169E+00
 5.000E-03    5.610E+00
 1.000E-02    6.048E+00
 1.500E-02    6.481E+00
 2.000E-02    6.908E+00
 2.500E-02    7.325E+00
 3.000E-02    7.730E+00
 3.500E-02    8.121E+00
 4.000E-02    8.494E+00
 4.500E-02    8.849E+00
 5.000E-02    9.185E+00
 5.500E-02    9.500E+00
 6.000E-02    9.792E+00
 6.500E-02    1.006E+01
 7.000E-02    1.031E+01
 7.500E-02    1.053E+01
 8.000E-02    1.072E+01
 8.500E-02    1.088E+01
 9.000E-02    1.085E+01
 9.500E-02    1.090E+01
 1.000E-01    1.094E+01
 1.050E-01    1.098E+01
 1.100E-01    1.102E+01
 1.150E-01    1.105E+01
 1.200E-01    1.109E+01
 1.250E-01    1.112E+01
 1.300E-01    1.116E+01
```

Figure A3.2. SPUDS Output for Differential-Pair Circuit (Page 2 of 7)

```
1.400E-01     1.123E+01
1.450E-01     1.125E+01
1.500E-01     1.130E+01
1.550E-01     1.133E+01
1.600E-01     1.136E+01
1.650E-01     1.139E+01
1.700E-01     1.142E+01
1.750E-01     1.145E+01
1.800E-01     1.148E+01
1.850E-01     1.151E+01
1.900E-01     1.154E+01
1.950E-01     1.157E+01
2.000E-01     1.160E+01
2.050E-01     1.162E+01
2.100E-01     1.165E+01
2.150E-01     1.167E+01
2.200E-01     1.170E+01
2.250E-01     1.172E+01
2.300E-01     1.174E+01
2.350E-01     1.176E+01
2.400E-01     1.178E+01
2.450E-01     1.180E+01
2.500E-01     1.182E+01

***** 03AUG80 ******** SPUDS 1A.0 (03AUG80) ******** 21:02:57 *****
DIFFPAIR CKT - SIMPLE DIFFERENTIAL PAIR
     SMALL SIGNAL BIAS SOLUTION       TEMPERATURE =   27.000 DEG C
**********************************************************************
 NODE       VOLT     NODE       VOLT     NODE       VOLT     NODE       VOLT
(   1)     0.0000   (   2)     -.0077   (   3)     -.0077   (   4)     5.1678
(   5)     5.1683   (   6)     -.6512   (   7)   -11.3401   (   8)    12.0000
(   9)   -12.0000
    VOLTAGE SOURCE CURRENTS
    NAME         CURRENT
    VIN        -7.741E-06
    VCC        -2.533E-03
    VEE         2.549E-03
    TOTAL POWER DISSIPATION   6.10E-02  WATTS

***** 03AUG80 ******** SPUDS 1A.0 (03AUG80) ******** 21:02:57 *****
DIFFPAIR CKT - SIMPLE DIFFERENTIAL PAIR
     OPERATING POINT INFORMATION      TEMPERATURE =   27.000 DEG C
**********************************************************************

**** BIPOLAR JUNCTION TRANSISTORS


              Q1          Q2          Q3          Q4
MODEL       QNL         QNL         QNL         QNL
IB        7.74E-06    7.74E-06    1.42E-05    1.42E-05
IC        6.84E-04    6.84E-04    1.38E-03    1.14E-03
VBE           .643        .643        .660        .660
VBC         -5.176      -5.176     -10.689        .000
VCE          5.819       5.820      11.349        .660
BETA DC     88.285      88.285      97.105      80.002
GM        2.64E-02    2.64E-02    5.34E-02    4.40E-02
RPI       3.34E+03    3.34E+03    1.82E+03    1.82E+03
RO        8.97E+04    8.97E+04    4.39E+04    4.39E+04
CPI       1.20E-11    1.20E-11    1.81E-11    1.81E-11
CMU       8.05E-13    8.05E-13    5.85E-13    2.00E-12
BETA AC     88.243      88.244      97.063      79.961
FT        1.28E+08    1.28E+08    4.54E+08    1.48E+08
```

Figure A3.2. SPUDS Output for Differential-Pair Circuit (Page 3 of 7)

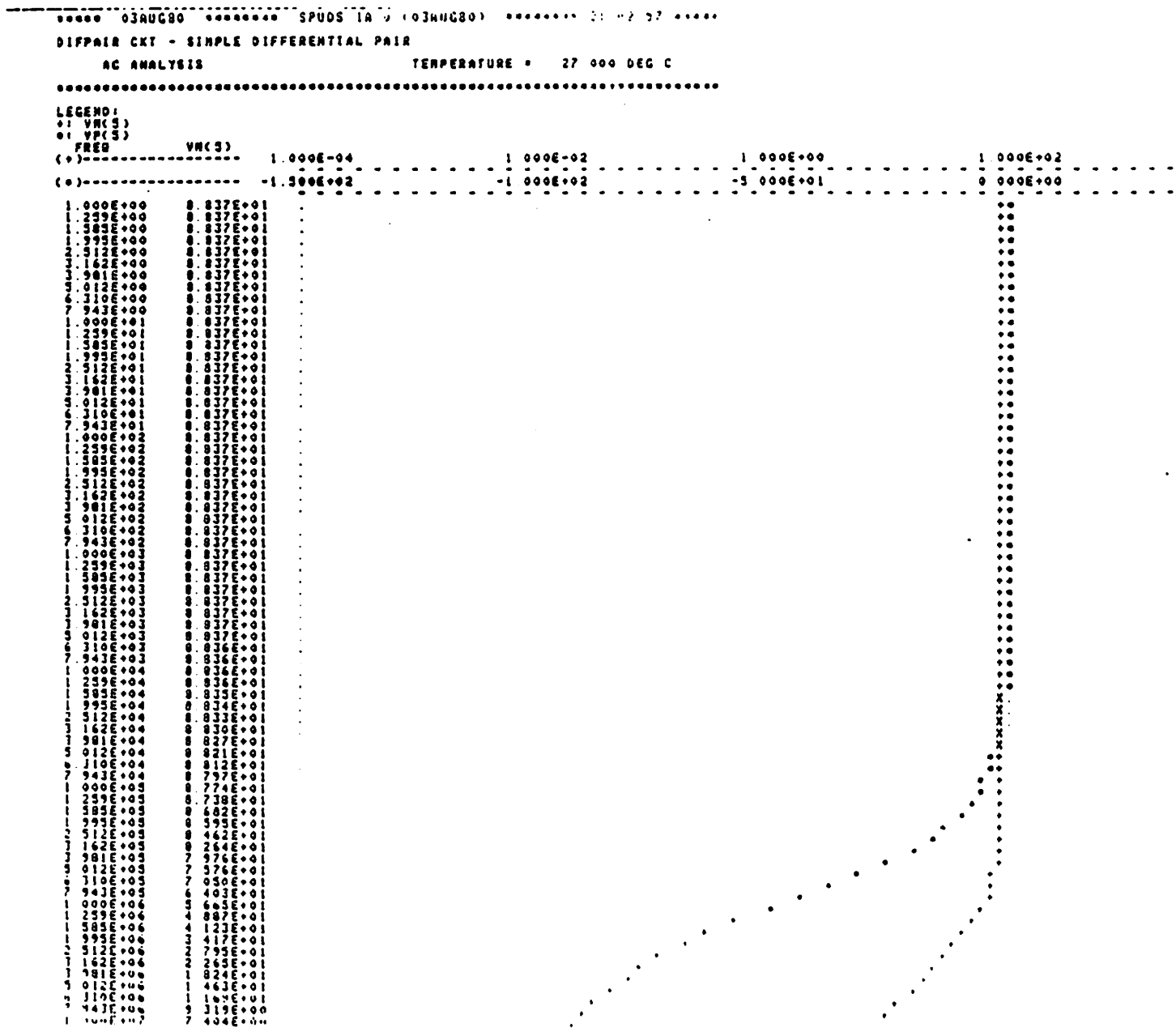
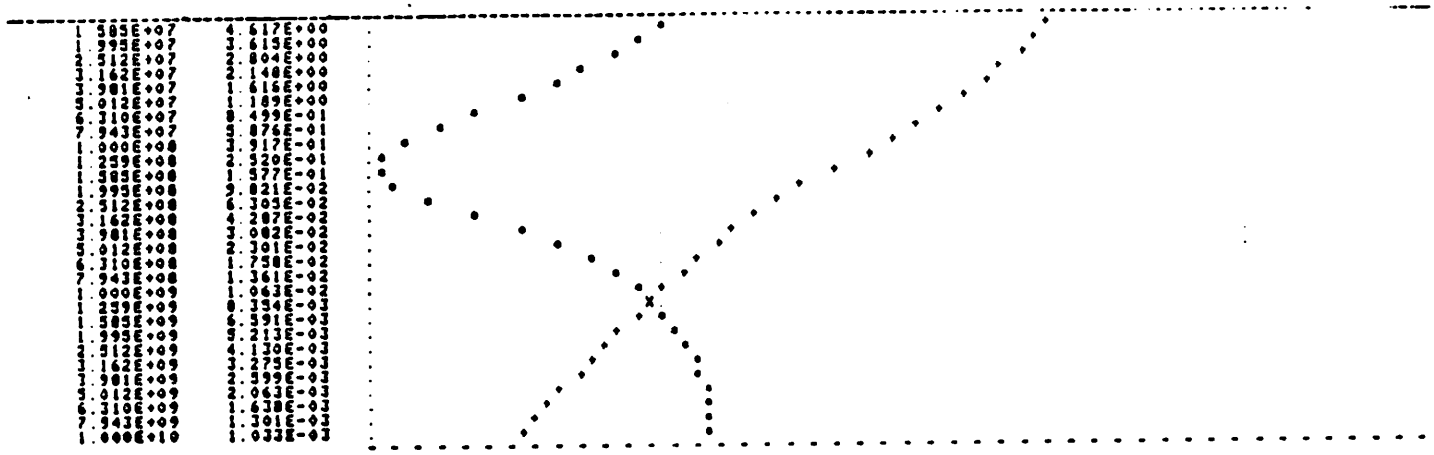


Figure A3.2. SPUDS Output for Differential-Pair Circuit (Page 4 of 7)



```
*****  03AUG80  ********  SPUDS 1A 0 (03AUG80)  ********  21:04:57  *****
DIFFPAIR CKT - SIMPLE DIFFERENTIAL PAIR
     INITIAL TRANSIENT SOLUTION       TEMPERATURE =   27.000 DEG C
********************************************************************************
 NODE      VOLT     NODE      VOLT     NODE      VOLT     NODE      VOLT
(  1)     0.0000   (  2)    -.0677   (  3)    -.0677   (  4)     5.1678
(  5)     5.1683   (  6)    -.6512   (  7)   -11.3401   (  8)    12.0000
(  9)   -12.0000
    VOLTAGE SOURCE CURRENTS
    NAME         CURRENT
    VIN        -7.741E-06
    VCC        -2.533E-03
    VEE         2.549E-03
    TOTAL POWER DISSIPATION   6.10E-02  WATTS

*****  03AUG80  ********  SPUDS 1A 0 (03AUG80)  ********  21:04:57  *****
DIFFPAIR CKT - SIMPLE DIFFERENTIAL PAIR
     OPERATING POINT INFORMATION      TEMPERATURE =   27.000 DEG C
********************************************************************************

**** BIPOLAR JUNCTION TRANSISTORS


              Q1          Q2          Q3          Q4
MODEL       QNL         QNL         QNL         QNL
IB        7.74E-06    7.74E-06    1.42E-05    1.42E-05
IC        6.84E-04    6.84E-04    1.38E-03    1.14E-03
VBE           .643        .643        .660        .660
VBC          5.176      -5.176     -10.989        .000
VCE          5.819       5.820      11.349        .660
BETA        88.285      88.285      97.105      80.602
```

Figure A3.2. SPUDS Output for Differential-Pair Circuit (Page 5 of 7)

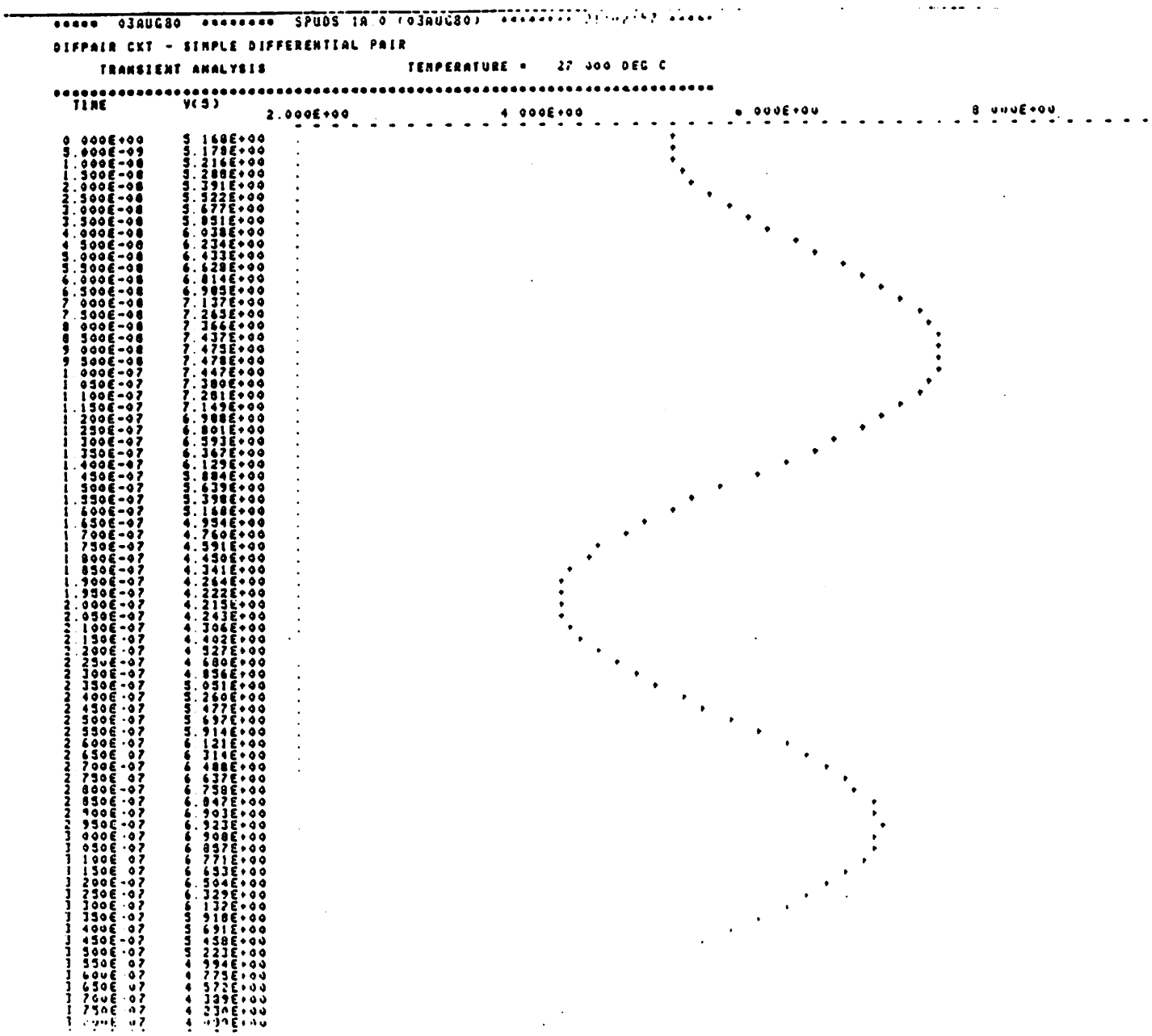
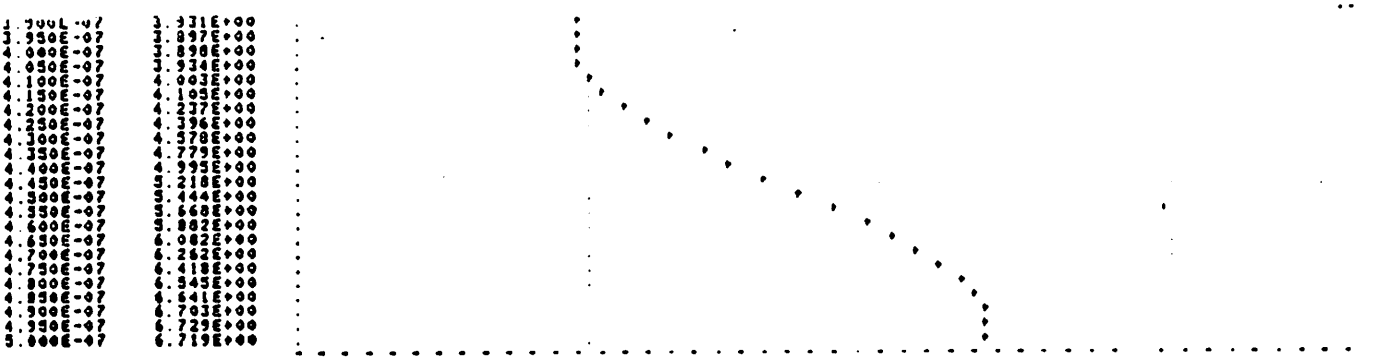


Figure A3.2. SPUDS Output for Differential-Pair Circuit (Page 6 of 7)



```
1.900E-07    3.831E+00
3.950E-07    3.897E+00
4.000E-07    3.898E+00
4.050E-07    3.934E+00
4.100E-07    4.003E+00
4.150E-07    4.105E+00
4.200E-07    4.237E+00
4.250E-07    4.396E+00
4.300E-07    4.578E+00
4.350E-07    4.779E+00
4.400E-07    4.995E+00
4.450E-07    5.218E+00
4.500E-07    5.444E+00
4.550E-07    5.668E+00
4.600E-07    5.882E+00
4.650E-07    6.082E+00
4.700E-07    6.262E+00
4.750E-07    6.418E+00
4.800E-07    6.545E+00
4.850E-07    6.641E+00
4.900E-07    6.703E+00
4.950E-07    6.729E+00
5.000E-07    6.719E+00

        JOB CONCLUDED

***** 03AUG80 ******** SPUDS (A ...03AUG80) ******** 21:03:53 *****
DIFFPAIR CKT - SIMPLE DIFFERENTIAL PAIR
     JOB STATISTICS SUMMARY              TEMPERATURE =   27.000 DEG C
**********************************************************************
  NUMNOS NCNODS NUMNOD NUMEL   DIODES   BJTS   JFETS   MFETS
      10     10     14    12        0      4       0       0
  NUMTEM ICVFLG JTRFLG JACFLG INOISE  IDIST   NOGO
       1    101    101    101       0      0       0
  MSTOP   NTTBR   NTTAR   IFILL    IOPS    PERSPA
     17.     60.     73.      5      94    77.469
  NUMTTP  NUMRTP  NUMNIT  MAXUSE   WDSHVD
     107.     0.    238.   2364  3.999E+04
          READIN                  2.410
          SETUP                    .740
          TRCURV                  8.210            227
          DCAN                    1.020             16
          ACAN                   13.440            101
          TRANAN                 11.980            238
          OUTPUT                  9.490
          DCDCMP                   .620              0
          CODGEN                   .740              2
          OVERHEAD                 .500
          TOTAL JOB TIME         47.790
```

Figure A3.2. SPUDS Output for Differential-Pair Circuit (Page 7 of 7)

```
KTEST - MUTUAL INDUCTANCE TEST
.WIDTH IN=72 OUT=72
.OPT ACCT OPTS
.AC DEC 10 1 1GHZ
.TRAN 20NS 2000NS
ISRC 1 0 SIN(0 1 1MEG) AC 1
L1 1 0 1MH
L2 2 0 1MH
L3 3 0 10MH
L4 4 0 100MH
K12 L1 L2 0.99
K13 L1 L3 0.995
K14 L1 L4 0.99
K23 L2 L3 0.995
K24 L2 L4 0.99
K34 L3 L4 0.995
R1 1 0 1K
R2 2 0 1K
R3 3 0 1K
R4 4 0 1K
.PLOT AC VM(1) VP(1) VM(2) VP(2) VM(3) VP(3) VM(4) VP(4)
.PLOT TRAN V(1) V(2) V(3) V(4)
.END
```

Figure A3.3. SPUDS Input Listing for KTEST Circuit

```
RCA3040 CKT - RCA 3040 WIDEBAND AMPLIFIER
.WIDTH IN=72 OUT=72
.OPT ACCT DVPNJN=8 0E-6 OPTS
.DC VIN -0.25 0.25 0.005
.OP
.AC DEC 10 1 10GHZ
.TRAN 0 5NS 50NS
VIN 1 0 SIN(0 0.1 50MEG) AC 1
VCC 2 0 15.0
VEE 3 0 -15.0
RS1 30 1 1K
RS2 31 0 1K
R1 6 3 4.8K
R2 5 3 4.8K
R3 9 3 0.811K
R4 8 3 2.17K
R5 8 0 0.82K
R6 2 14 1.32K
R7 2 12 4.5K
R8 2 15 1.32K
R9 16 0 5.25K
R10 17 0 5.25K
Q1 2 31 6 QNL
Q2 2 30 5 QNL
Q3 10 5 7 QNL
Q4 11 6 7 QNL
Q5 14 12 10 QNL
Q6 15 12 11 QNL
Q7 12 12 13 QNL
Q8 13 13 0 QNL
Q9 7 8 9 QNL
Q10 2 15 16 QNL
Q11 2 14 17 QNL
.MODEL QNL NPN(BF=80 RB=100 CCS=2PF TF=0.3NS TR=6NS CJE=3PF CJC=2PF
+    VA=50)
.PRINT DC V(16) V(17)
.PLOT DC V(17)
.PRINT AC VM(17) VP(17)
.PLOT AC VM(17) VP(17)
.PRINT TRAN V(16) V(17)
.PLOT TRAN V(17)
.END
```

Figure A3.4. SPUDS Input Listing for RCA304 Circuit

```
UA709 CKT - UA 709 OPERATIONAL AMPLIFIER
 WIDTH IN=72 OUT=72
 OPT ACCT OPTS PIVTOL=1 0E-12 PIVREL=1 0E-4
.DC VIN -0 25 0.25 0 005
.AC DEC 10 1 10GHZ
.TRAN 2 5US 250US
VIN 1 0 SIN(0 0 1 10KHZ) AC 1
VCC 19 0 15
VEE 20 0 -15
RS1 30 1 1K
RS2 31 0 1K
RF 30 18 100K
RCOMP 7 23 1 5K
CICOMP 23 4 5000PF
COCOMP 18 15 200PF
Q1 2 31 3 QNL
Q2 4 30 3 QNL
Q3 19 6 5 QNL
Q4 7 4 11 QNL
Q5 7 11 12 QNL
Q6 6 13 12 QNL
Q7 6 2 13 QNL
Q8 19 7 21 QNL
Q9 19 17 18 QNL
Q10 17 15 16 QNL
Q11 3 8 22 QNL
Q12 8 8 20 QNL
Q13 14 14 12 QNL
Q14 15 12 10 QPL
Q15 20 17 18 QPL
R1 5 2 25K
R2 5 4 25K
R3 22 20 2.4K
R4 8 9 18K
R5 9 12 3.6K
R6 11 14 3K
R7 19 7 10K
R8 19 6 10K
R9 9 10 10K
R10 10 18 30K
R11 19 17 20K
R12 15 16 10K
R13 16 20 75.0
R14 13 14 3K
R15 21 10 1K
.MODEL QNL NPN(BF=80 RB=100 CCS=2PF TF=0 3NS TR=6NS CJE=3PF CJC=2PF
+    VA=50)
.MODEL QPL PNP(BF=10 RB=20 TF=1NS TR=20NS CJE=6PF CJC=4PF VA=50)
.PRINT DC V(7) V(18)
.PLOT DC V(18)
.PRINT AC VM(18) VP(18)
.PLOT AC VM(18) VP(18)
 PRINT TRAN V(7) V(18)
 PLOT TRAN V(18)
 END
```

Figure A3.5. SPUDS Input Listing for UA709 Circuit

```
UA727 CKT - UA 727 AMPLIFIER
 WIDTH IN=72 OUT=72
 OPT ACCT OPTS RELTOL=.004
.DC VIN -.2 .2 .2 0 .004
 AC DEC 10 1 10GHZ
 TRAN 0 .05US 5US
VCC1 34 0 15
VCC2 35 0 15
VEE 36 0 -15
VIN 40 0 SIN(0 0.2 500KHZ) AC 1
RS1 40 1 1K
RS2 12 0 1K
IZ1 36 9 620MA
RZ1 36 9 10
IZ2 32 31 620MA
RZ2 32 31 10
R1 9 31 1K
R2 28 9 21K
R3 28 19 4.8K
R4 32 36 2.4K
R5 33 36 10
R6 26 19 2K
R7 25 36 1.5K
R8 20 36 120K
R9 11 3 60K
R10 6 8 60K
R11 34 5 3K
R12 8 9 10K
R13 22 36 15K
R14 21 36 15K
R15 23 36 15K
R16 17 9 10K
R17 34 15 3K
R18 16 17 60K
R19 11 14 60K
R21 24 36 120K
RTEMP 35 29 330K
Q3 35 29 31 QNL
Q4 35 32 33 QNL
Q5 29 33 36 QNL
Q6 29 28 27 QNL
Q7 27 27 26 QNL
Q8 19 19 25 QNL
Q9 34 1 2 QNL
Q10 2 19 20 QNL
Q11 34 34 11 QNL
Q12 3 2 4 QNL
Q13 4 19 22 QNL
Q14 6 3 5 QPL
Q15 5 6 8 QNL
Q16 34 8 10 QNL
Q17 10 19 21 QNL
Q18 34 17 18 QNL
Q19 18 19 23 QNL
Q20 15 16 17 QNL
Q21 16 14 15 QPL
Q22 14 13 4 QNL
Q23 13 19 24 QNL
Q24 34 12 13 QNL
 .MODEL QNL NPN(BF=80 RB=100 CCS=2PF TF=.3NS TR=6NS
+  CJE=3PF CJC=2PF VA=50)
 .MODEL QPL PNP(BF=10 RB=200 TF=1NS TR=70NS CJE=4PF
+  CJC=4PF VA=50)
 .PRINT DC V(10) V(18)
 .PLOT DC V(18)
 .PRINT AC VM(18) VP(18)
 .PLOT AC VM(18) VP(18)
 .PRINT TRAN V(10) V(18)
 .PLOT TRAN V(18)
 .END
```

Figure A3.6. SPUDS Input Listing for UA727 Circuit

```
UA733 CKT - UA 733 VIDEO PREAMPLIFIER
.WIDTH IN=72 OUT=72
.OPT ACCT OPTS
VCC 11 0 8
VEE 9 0 -8
Q1 3 1 4 Q1
Q2 14 2 13 Q1
Q3 17 14 16 Q1
Q4 18 3 16 Q1
Q5 11 18 19 Q1
Q6 11 17 22 Q1
Q7 6 7 8 Q1
Q8 7 7 10 Q1
Q9 16 7 15 Q1
Q10 19 7 20 Q1
Q11 22 7 21 Q1
R1 1 0 51
R2 2 0 51
R3 11 3 2.5K
R4 11 14 2.4K
R5 4 5 50
R6 13 12 50
R7 5 6 590
R8 12 6 590
R9 11 7 10K
R10 11 17 1.1K
R11 11 18 1.1K
R12 3 19 7K
R13 14 22 7K
R14 8 9 300
R15 10 9 1.4K
R16 15 9 300
R17 20 9 400
R18 21 9 400
.MODEL Q1 NPN(BF=100 BR=2 IS=0.9901E-15)
.END
```

Figure A3.7. SPUDS Input Listing for UA733 Circuit

```
UA741 CKT - UA 741 OPERATIONAL AMPLIFIER
 WIDTH IN=72 OUT=72
 OPT ACCT OPTS
 DC VIN -0 25 0 25 0 005
 AC DEC 10 1 10GHZ
 TRAN 2 5US 250US
VCC 27 0 15
VEE 26 0 -15
VIN 30 0 SIN(0 0 1 10KHZ) AC 1
RS1 2 30 1K
RS2 1 0 1K
RF 24 2 100K
R1 10 26 1K
R2 9 26 50K
R3 11 26 1K
R4 12 26 3K
R5 15 17 39K
R6 21 20 40K
R7 14 26 50K
R8 18 26 50
R9 24 25 25
R10 23 24 50
R11 13 26 50K
COMP 22 8 30PF
Q1 3 1 4 QNL
Q2 3 2 5 QNL
Q3 7 6 4 QPL
Q4 8 6 5 QPL
Q5 7 9 10 QNL
Q6 8 9 11 QNL
Q7 27 7 9 QNL
Q8 6 15 12 QNL
Q9 15 15 26 QNL
Q10 3 3 27 QPL
Q11 6 3 27 QPL
Q12 17 17 27 QPL
Q14 22 17 27 QPL
Q15 22 22 21 QNL
Q16 22 21 20 QNL
Q17 13 13 26 QNL
Q18 27 8 14 QNL
Q19 20 14 18 QNL
Q20 22 23 24 QNL
Q21 13 25 24 QPL
Q22 27 22 23 QNL
Q23 26 20 25 QPL
.MODEL QNL NPN(BF=80 RB=100 CCS=2PF TF=0 3NS TR=6NS CJE=3PF CJC=2PF
+   VA=50)
.MODEL QPL PNP(BF=10 RB=20 TF=1NS TR=20NS CJE=6PF CJC=4PF VA=50)
.PRINT DC V(8) V(24)
.PLOT DC V(24)
.PRINT AC VM(24) VP(24)
.PLOT AC VM(24) VP(24)
.PRINT TRAN V(8) V(24)
.PLOT TRAN V(24) V(8)
.END
```

Figure A3.8. SPUDS Input Listing for UA741 Circuit

```
RTLINV CKT - CASCADED RTL INVERTERS
.WIDTH IN=72 OUT=72
.OPT ACCT DVPNJN=1.0E-6 OPTS
.DC VIN 0 0 2.5 0 025
.TRAN 2NS 200NS
VCC 6 0 5
VIN 1 0 PULSE(0 5 2NS 2NS 2NS 80NS)
RB1 1 2 10K
RC1 6 3 1K
Q1 3 2 0 QND
RB2 3 4 10K
Q2 5 4 0 QND
RC2 6 5 1K
.MODEL QND NPN(BF=50 RB=70 RC=40 CCS=2PF TF=0.1NS TR=10NS CJE=0.9PF
+    CJC=1.5PF PC=0.85 VA=50)
.PRINT DC V(3) V(5)
.PLOT DC V(3)
.PRINT TRAN V(3) V(5)
.PLOT TRAN V(3) V(5) V(1)
.END
```

Figure A3.9. SPUDS Input Listing for RTLINV Circuit

```
TTLINV CKT - 74 SERIES TTL INVERTER
 WIDTH IN=72 OUT=72
.OPT ACCT DYPNJN=1.0E-6 OPTS RELTOL=0.002
.DC VIN 0 2 0.02
.TRAN 1NS 100NS
VCC 13 0 5
VIN 1 0 PULSE(0 3.5 1NS 1NS 1NS 40NS)
RS 1 2 50
Q1 4 3 2 QND
RB1 13 3 4K
Q2 5 4 6 QND
RC2 13 5 1.4K
RE2 6 0 1K
Q3 7 5 8 QND
RC3 13 7 100
D1 8 9 D1
Q4 9 6 0 QND
Q5 11 10 9 QND
RB2 13 10 4K
D2 11 12 D1
D3 12 0 D1
.MODEL D1 D(RS=40 TT=0.1NS CJO=0.9PF)
.MODEL QND NPN(BF=50 RB=70 RC=40 CCS=2PF TF=0.1NS TR=10NS CJE=0.9PF
+    CJC=1.5PF PC=0.85 VA=50)
.PRINT DC V(5) V(9)
.PLOT DC V(9)
.PRINT TRAN V(5) V(9)
.PLOT TRAN V(5) V(9) V(1)
.END
```

Figure A3.10. SPUDS Input Listing for TTLINV Circuit

```
TTL74 CKT - SERIES 74 TTL INVERTER
.WIDTH IN=72 OUT=72
*.OPT ACCT DVPNJN=1.0E-6 OPTS PIVTOL=1.0E-12 PIVREL=1.0E-4
.OPT ACCT DVPNJN=1.0E-6 OPTS
.DC VIN 0 2 0.02
.TRAN 1NS 100NS
VIN 1 0 DC 1 3 PULSE(0 3.5 1NS 1NS 1NS 40NS)
VCC 13 0 5
RS 1 2 50
Q1 4 3 2 QC
Q2 5 4 6 QA
Q3 7 5 8 QA
Q4 9 6 0 QB
Q5 11 10 9 QC
D1 8 9 DA
D2 11 12 DA
D3 12 0 DA
RB1 13 3 4K
RC2 13 5 1.4K
RE2 6 0 1K
RC3 13 7 100
RB5 13 10 4K
.MODEL QA NPN(BF=20 BR=1 RB=70 RC=40 IS=1E-14 VA=50 CJE=3PF CJC=2PF)
.MODEL QB NPN(BF=20 BR=.2 RB=20 RC=12 IS=1.6E-14 VA=50 CJE=3PF CJC=2PF)
.MODEL QC NPN(BF=20 BR=.02 RB=500 RC=40 IS=1E-14 VA=50 CJE=3PF CJC=2PF)
.MODEL DA D(RS=40 TT=0.1NS CJO=0.3PF IS=1.0E-14)
.PRINT DC V(9) V(6) V(5)
.PLOT DC V(9) V(6) V(5)
.PRINT TRAN V(9) V(6) V(5)
.PLOT TRAN V(9) V(6) V(5) V(1)
.END
```

Figure A3.11. SPUDS Input Listing for TTL74 Circuit

```
TTL74S CKT - SERIES 74S TTL INVERTER
.WIDTH IN=72 OUT=72
.OPT ACCT DVPNJN=2U OPTS
.DC VIN 0 2.5 0.025
.TRAN 1NS 100NS
VIN 1 0 DC 1.375 PULSE(0 3.5 1NS 1NS 1NS 40NS)
VCC 15 0 5
RS 1 2 50
Q1 4 3 2 QC
RB1 15 3 2.4K
RC2 15 5 800
Q2 5 4 6 QA
Q3 7 5 8 QA
Q4 7 8 9 QB
RC4 15 7 60
Q5 10 11 0 QA
RE2 6 11 250
RC5 6 10 500
Q6 9 6 0 QB
Q7 13 12 9 QC
RB7 15 12 2.4K
D1 13 14 DA
D2 14 0 DA
.MODEL QA NPN(BF=20 BR=1 RB=70 RC=40 IS=1E-14 VA=50 CJE=3PF CJC=2PF)
.MODEL QB NPN(BF=20 BR=.2 RB=20 RC=12 IS=1.6E-14 VA=50 CJE=3PF CJC=2PF)
.MODEL QC NPN(BF=20 BR=.02 RB=500 RC=40 IS=1E-14 VA=50 CJE=3PF CJC=2PF)
.MODEL DA D(RS=40 IS=1.0E-14)
.PRINT DC V(9) V(6) V(5)
.PLOT DC V(9) V(6) V(5)
.PRINT TRAN V(9) V(6) V(5)
.PLOT TRAN V(9) V(6) V(5) V(1)
.END
```

Figure A3.12. SPUDS Input Listing for TTL74S Circuit

```
TTL74L CKT - SERIES 74L TTL INVERTER
 WIDTH IN=72 OUT=72
.OPT ACCT OPTS RELTOL=0.001
.DC VIN 0.0 1.5 0.015
.TRAN 10NS 1000NS
VIN 1 0 DC 1.175 PULSE(0 3.5 10NS 10NS 10NS 400NS)
VCC 13 0 5
RS 1 2 50
Q1 4 3 2 QC
Q2 5 4 6 QA
Q3 7 5 8 QA
Q4 9 6 0 QA
Q5 11 10 9 QC
D1 8 9 DA
D2 11 12 DA
D3 12 0 DA
RB1 13 3 40K
RC2 13 5 20K
RE2 6 0 12K
RC3 13 7 500
RB2 13 10 40K
.MODEL QA NPN(BF=20 BR=1 RB=70 RC=40 IS=1E-14 VA=50 CJE=3PF CJC=2PF)
.MODEL QB NPN(BF=20 BR= 2 RB=20 RC=12 IS=1.6E-14 VA=50 CJE=3PF CJC=2PF)
.MODEL QC NPN(BF=20 BR=.02 RB=500 RC=40 IS=1E-14 VA=50 CJE=3PF CJC=2PF)
.MODEL DA D(RS=40 IS=1.0E-14)
.PRINT DC V(9) V(6) V(5)
.PLOT DC V(9) V(6) V(5)
.PRINT TRAN V(9) V(6) V(5)
.PLOT TRAN V(9) V(6) V(5) V(1)
.END
```

Figure A3.13. SPUDS Input Listing for TTL74L Circuit

```
TTL9200 CKT - SERIES 9200 TTL INVERTER
.WIDTH IN=72 OUT=72
.OPT ACCT PIVREL=1.0E-4 PIVTOL=1.0E-15 DVPNJN=1.0E-6 OPTS
.DC VIN 0.0 2.0 0.02
.TRAN 1NS 100NS
VIN 1 0 DC 1.35 PULSE(0 3 5 1NS 1NS 1NS 40NS)
VCC 14 0 5
RS 1 2 50
Q1 4 3 2 QC
Q2 5 4 6 QA
Q3 7 5 8 QA
Q4 9 8 10 QB
Q5 10 6 0 QB
Q6 12 11 10 QC
D1 12 13 DA
D2 13 0 DA
RB1 14 3 4K
RC1 14 5 1.5K
RE2 6 0 1.25K
RC2 14 7 150
RB4 8 10 4K
RC4 14 9 80
RB6 14 11 4K
.MODEL QA NPN(BF=20 BR=1 RB=70 RC=40 IS=1E-14 VA=50 CJE=3PF CJC=2PF)
.MODEL QB NPN(BF=20 BR=.2 RB=20 RC=12 IS=1.6E-14 VA=50 CJE=3PF CJC=2PF)
.MODEL QC NPN(BF=20 BR=.02 RB=500 RC=40 IS=1E-14 VA=50 CJE=3PF CJC=2PF)
.MODEL DA D(RS=40 IS=1.0E-14)
.PRINT DC V(10) V(6) V(5)
.PLOT DC V(10) V(6) V(5)
.PRINT TRAN V(10) V(6) V(5)
.PLOT TRAN V(10) V(6) V(5) V(1)
.END
```

Figure A3.14. SPUDS Input Listing for TTL920 Circuit

```
ECLGATE CKT - ECL STACKED LOGIC GATE
.WIDTH IN=72 OUT=72
.OPT ACCT OPTS DYPNJN=3U
.DC VIN -2 0 0 02
.TRAN 0.2NS 20NS
VEE 15 0 -6
VIN 16 0 PULSE(-1.8 -0 8 1NS 1NS 1NS)
VGATE 17 0 PULSE(-0.8 -1 8 5NS 1NS 1NS 5NS)
RS1 16 1 50
Q1 2 1 3 QND
Q2 0 9 3 QND
RC 0 2 100
RS2 17 4 50
Q3 0 4 5 QND
R1 5 6 60
R2 6 15 820
Q4 3 6 7 QND
RE 7 15 280
Q5 0 12 7 QND
R5 0 8 100
Q6 0 8 9 QND
R3 9 15 2K
D1 8 10 D1
R6 10 11 60
Q7 0 11 12 QND
R4 12 15 2K
D2 11 13 D1
R7 13 15 720
Q8 0 2 14 QND
RL 14 15 560
.MODEL D1 D(RS=40 TT=0.1NS CJO=0.9PF)
.MODEL QND NPN(BF=50 RB=70 RC=40 CCS=2PF TF=0.1NS TR=10NS CJE=0.9PF
+    CJC=1.5PF PC=0.85 VA=50)
.PRINT DC V(6) V(14)
.PLOT DC V(14)
.PRINT TRAN V(6) V(14)
.PLOT TRAN V(14) V(16) V(17) V(6)
.END
```

Figure A3.15. SPUDS Input Listing for ECLGATE Circuit

```
MECLIII CKT - MOTOROLA MECL III ECL GATE
.WIDTH IN=72 OUT=72
.OPT ACCT OPTS DVPNJN=40
.DC VIN -2 0 0.02
.TRAN 0.2NS 20NS
VEE 22 0 -6
VIN 1 0 PULSE(-0.8 -1.8 0.2NS 0.2NS 0.2NS 10NS)
RS 1 2 50
Q1 4 2 6 QND
Q2 4 3 6 QND
Q3 5 7 6 QND
Q4 0 8 7 QND
D1 8 9 D1
D2 9 10 D1
RP1 3 22 50K
RC1 0 4 100
RC2 0 5 112
RE 6 22 380
R1 7 22 2K
R2 0 8 350
R3 10 22 1958
Q5 0 5 11 QND
Q6 0 4 12 QND
RP2 11 22 560
RP3 12 22 560
Q7 13 12 15 QND
Q8 14 16 15 QND
RE2 15 22 380
RC3 0 13 100
RC4 0 14 112
Q9 0 17 16 QND
R4 16 20 2K
R5 0 17 350
D3 17 18 D1
D4 18 19 D1
R6 19 22 1958
Q10 0 14 20 QND
Q11 0 13 21 QND
RP4 20 22 560
RP5 21 22 560
.MODEL D1 D(RS=40 TT=0.1NS CJO=0.9PF)
.MODEL QND NPN(BF=50 RB=70 RC=40 CCS=2PF TF=0.1NS TR=10NS CJE=0.9PF
+    CJC=1.5PF PC=0.85 VA=50)
.PRINT DC V(12) V(21)
.PLOT DC V(12)
.PRINT TRAN V(12) V(21)
.PLOT TRAN V(21) V(12) V(1)
.END
```

Figure A3.16. SPUDS Input Listing for MECLIII Circuit

```
SBDGATE CKT - SCHOTTKY-BARRIER TTL INVERTER
.WIDTH IN=72 OUT=72
.OPT ACCT DVPNJN=1.0E-6 OPTS
.DC VIN 0.0 1.1 0.011
.TRAN 2NS 200NS
VCC 23 0 5
VLOAD 26 0 5
VIN 1 0 PULSE(0.5 3.6 2NS 2NS 2NS 50NS)
RS 1 2 50
RB1 23 3 15K
RB2 26 17 15K
RC1 4 5 60
RC2 6 9 30
RC3 16 15 10
RC4 18 19 60
RE1 7 8 600
RE2 20 21 600
RL1 23 10 8.75K
RL2 26 25 8K
RK 23 12 1K
RS2 24 15 50
Q1 4 3 2 QND
Q2 6 5 7 QND
Q3 16 7 0 QND
Q4 18 17 24 QND
Q5 22 19 20 QND
Q6 20 20 0 QND
QL2 25 25 22 QND
QE 12 10 13 QND
DC1 3 4 D2
DC2 5 6 D2
DC3 7 16 D2
DC4 17 18 D2
DC5 19 22 D2
DE1 8 0 D2
DE2 21 0 D2
D1 13 14 D2
D12 14 28 D2
D2 28 15 D2
DL 10 9 D2
.MODEL D2 D(RS=15 CJO=0.20PF IS=5E-10 PB=0.6)
.MODEL QND NPN(BF=50 RB=70 RC=40 CCS=2PF TF=0 1NS TR=10NS CJE=0.9PF
+    CJC=1.5PF PC=0.85 VA=50)
.PRINT DC V(15) I(VCC)
.PLOT DC V(15)
.PRINT TRAN V(15) I(VCC)
.PLOT TRAN V(15) V(1) I(VCC)
.END
```

Figure A3.17. SPUDS Input Listing for SBDGATE Circuit

```
CCSOR CKT - CONSTANT CURRENT SOURCE
.WIDTH IN=72 OUT=72
.OPT ACCT OPTS
VEE 7 0 -12
VBIAS 3 0 -6
Q1 2 3 4 Q1
Q2 2 4 5 Q1
Q3 1 6 5 Q1
Q4 1 8 6 Q1
Q5 10 1 9 Q1
Q6 10 9 8 Q1
R1 2 0 2K
R2 1 0 2K
R3 5 7 2K
R4 8 7 2K
R5 10 0 107
.MODEL Q1 NPN(BF=49.5 BR=0.5 IS=0.9802E-15)
.END
```

Figure A3.18. SPUDS Input Listing for CCSOR Circuit

```
DCOSC - DC PART OF 1KHZ OSCILLATOR
.WIDTH IN=72 OUT=72
.OPT ACCT OPTS
VCC 2 0 5.6
Q1 2 1 8 Q1
Q2 3 6 5 Q1
Q3 2 10 12 Q1
Q4 11 3 7 Q1
Q5 10 11 13 Q1
Q6 2 10 9 Q1
Q7 3 8 4 Q1
R1 2 3 12K
R2 4 5 300
R3 4 0 1.5K
R4 10 1 98.603K
R5 2 11 7.5K
R6 7 0 1K
R7 12 6 5K
R8 6 0 10K
R9 2 10 1.5K
R10 13 0 240
R11 9 0 147
.MODEL Q1 NPN(BF=60 BR=0.205 IS=1.21E-15)
.END
```

Figure A3.19. SPUDS Input Listing for DCOSC Circuit

```
CFFLOP CKT - SATURATING COMPLEMENTARY FLIP-FLOP
.WIDTH IN=72 OUT=72
.OPT ACCT OPTS
VCC1 1 0 5
VCC2 8 0 10
VEE1 10 0 -5
VEE2 9 0 -10
Q1 3 2 1 QP1
Q2 6 7 1 QP1
Q3 6 5 10 QN1
Q4 3 4 10 QN1
R1 8 2 57.2K
R2 8 7 57.2K
R3 2 6 14.3K
R4 7 3 14.3K
R5 6 4 14.3K
R6 3 5 14.3K
R7 4 9 57.2K
R8 5 9 57.2K
R9 3 0 1K
.MODEL QN1 NPN(BF=10 BR=1 IS=0 91E-14)
.MODEL QP1 PNP(BF=10 BR=1 IS=0 91E-14)
.END
```

Figure A3.20. SPUDS Input Listing for CFFLOP Circuit

```
STCRC CKT - SPLIT TIME-CONSTANT RC CIRCUIT
.WIDTH IN=72 OUT=72
.OPT ACCT OPTS
.TRAN 0.1MS 10MS
VIN 1 0 PULSE(0 1 0.1MS 0.1MS)
R1 1 2 1K
C1 2 0 1PF
R2 2 3 1K
C2 3 0 1UF
.PRINT TRAN V(1) V(2) V(3)
.PLOT TRAN V(2) V(3) V(1)
.END
```

Figure A3.21. SPUDS Input Listing for STCRC Circuit

```
CHOKE CKT - FULL WAVE CHOKE INPUT
.WIDTH IN=72 OUT=72
.OPT ACCT OPTS
.TRAN 0.2MS 20MS UIC
VIN1 1 0 SIN(0 100 50)
VIN2 2 0 SIN(0 -100 50)
D1 1 3 DIO
D2 2 3 DIO
R1 3 0 10K
L1 3 4 5.0
R2 4 0 10K
C2 4 0 2UF IC=152V
.MODEL DIO D(IS=1.0E-14 CJO=10PF BV=1E8)
.PRINT TRAN V(1) V(2) V(3) V(4)
.PLOT TRAN V(1) V(2) V(3) V(4)
.END
```

Figure A3.22. SPUDS Input Listing for CHOKE Circuit

```
ECLINV CKT - EMITTER COUPLED LOGIC INVERTER
.WIDTH IN=72 OUT=72
.OPT ACCT DVPNJN=3.0E-6 OPTS
.TRAN 0.2NS 20NS
VIN 1 0 PULSE(-1 -1.8 1NS 1NS 1NS 8NS 20NS)
VEE 8 0 -5
VREF 6 0 -1.4
Q1 3 2 4 QSTD
Q2 5 6 4 QSTD
Q3 0 5 7 QSTD
Q4 0 5 7 QSTD
RIN 1 2 50
RC1 0 3 80
RC2 0 5 135
RE 4 8 340
RTH1 7 8 125
RTH2 7 0 85
CLOAD 7 0 5PF
.MODEL QSTD NPN(IS=1.0E-16 BR=50 BR=0.1 RB=50 RC=10 TF=0.12NS TR=5NS
+    CJE=0.4PF PC=0.8 ME=0.4 CJE=0.5PF PC=0.8 MC=0.333 CCS=1PF VA=50)
.PRINT TRAN V(1) V(3) V(5) V(7) I(VIN)
.PLOT TRAN V(3) V(5) V(7) V(1) I(VIN)
.END
```

Figure A3.23. SPUDS Input Listing for ECLINV Circuit

```
SCHMITT CKT - ECL COMPATIBLE SCHMITT TRIGGER
.WIDTH IN=72 OUT=72
.OPT ACCT OPTS
.TRAN 10NS 1000NS
VIN 1 0 PULSE(-1.6 -1.2 10NS 400NS 400NS 100NS 10000NS)
VEE 8 0 -5
RIN 1 2 50
RC1 0 3 50
R1 3 5 185
R2 5 8 760
RC2 0 6 100
RE 4 8 260
RTH1 7 8 125
RTH2 7 0 85
CLOAD 7 0 5PF
Q1 3 2 4 QSTD OFF
Q2 6 5 4 QSTD
Q3 0 6 7 QSTD
Q4 0 6 7 QSTD
.MODEL QSTD NPN(IS=1.0E-16 BF=50 BR=0.1 RB=50 RC=10 TF=0.12NS TR=5NS
+    CJE=0.4PF PE=0.8 ME=0.4 CJC=0.5PF PC=0.8 MC=0.333 CCS=1PF VA=50)
.PRINT TRAN V(1) V(3) V(5) V(6)
.PLOT TRAN V(3) V(5) V(6) V(1)
.END
```

Figure A3.24. SPUDS Input Listing for SCHMITT Circuit

```
ASTABLE CKT - A SIMPLE ASTABLE MULTIVIBRATOR
.WIDTH IN=72 OUT=72
.OPT ACCT OPTS RELTOL=0.003
.TRAN 0.1US 10US
VIN 5 0 PULSE(0 5 0 1US 1US 100US 100US)
VCC 6 0 5
RC1 6 1 1K
RC2 6 2 1K
RB1 6 3 30K
RB2 5 4 30K
C1 1 4 150PF
C2 2 3 150PF
Q1 1 3 0 QSTD
Q2 2 4 0 QSTD
.MODEL QSTD NPN(IS=1.0E-16 BR=50 BR=0.1 RB=50 RC=10 TF=0.12NS TR=5NS
+    CJE=0.4PF PC=0.8 ME=0.4 CJE=0.5PF PC=0.8 MC=0.333 CCS=1PF VA=50)
.PRINT TRAN V(1) V(2) V(3) V(4)
.PLOT TRAN V(2) V(3) V(4) V(1)
.END
```

Figure A3.25. SPUDS Input Listing for ASTABLE Circuit

```
SATINV - SIMPLE SATURATED MOS INVERTER
.WIDTH IN=72 OUT=72
.OPT ACCT OPTS
.DC VS 0 10 .1
.OP
.TEMP 0 27 70
M1 1 1 2 4 MOSEN W=5U L=50U
M2 2 3 0 4 MOSEN W=5U L=5U
VBB 4 0 DC -4
VDD 1 0 DC 10
VS 3 0 PULSE(0,10)
.MODEL MOSEN NMOS(LEVEL=2 NSUB=4E15 NFS=2E11 UEXP=.15 XJ=1E-6
+ U0=700)
.PRINT DC V(2) I(VDD)
.PLOT DC V(2) I(VDD)
.END
```

Figure A3.26. SPUDS Input Listing for SATINV Circuit

```
DEPLINV - SIMPLE DEPLETION MOS INVERTER
.WIDTH IN=72 OUT=72
.OPT ACCT OPTS
.DC VS 0 2 .02
.OP
M1 1 2 2 0 MOSDN W=5U L=50U
M2 2 3 0 0 MOSEN W=5U L=5U
VS 3 0 PULSE(0,3)
VDD 1 0 DC 10
.MODEL MOSEN NMOS(LEVEL=2 NSUB=4E15 NFS=2E11 UO=700 UEXP=0.15 XJ=1E-6)
.MODEL MOSDN NMOS(LEVEL=2 NSUB=4E15 NSS=8E11 UO=700 UEXP=0.15)
.PRINT DC V(2) I(VDD)
.PLOT DC V(2) I(VDD)
.END
```

Figure A3.27. SPUDS Input Listing for DEPLINV Circuit

```
RATLOG - RATIOLESS DYNAMIC LOGIC CIRCUIT
.WIDTH IN=72 OUT=72
.OPT ACCT ABSTOL=1U OPTS
.TRAN 2.5N 250N
M1 9 11 2 10 NMOS W=20U L=7U AD=1N AS=1N
M2 9 12 4 10 NMOS W=20U L=7U AD=1N AS=1N
M3 2  1 0 10 NMOS W=20U L=7U AD=1N AS=1N
M4 4  3 0 10 NMOS W=20U L=7U AD=1N AS=1N
M5 3 12 2 10 NMOS W=20U L=7U AD=1N AS=1N
M6 1 11 5 10 NMOS W=20U L=7U AD=1N AS=1N
C1 1 0 0.05PF
C2 2 0 0.05PF
C3 3 0 0.05PF
C4 4 0 0.05PF
C5 5 0 0.05PF
VIN  5 0 PULSE(0 10 10N 5N 10N 55N 500N)
VP1 11 0 PULSE(0 12 10N 5N  5N 35N 120N)
VP2 12 0 PULSE(0 12 70N 5N  5N 35N 120N)
VDD  9 0 DC 12
VBB 10 0 DC -5
.MODEL NMOS NMOS(LEVEL=1 NSUB=8E14 TOX= 115U LD=0.1M NGATE=1E20
+    NSS=4E10 UEXP=0.36 UTRA=0.3 CBD=97U CBS=97U)
.PLOT TRAN V(4) V(5)
.PLOT TRAN V(1) V(2) V(3) V(4)
.END
```

Figure A3.28. SPUDS Input Listing for RATLOG Circuit

```
INVCHN - FIVE-STAGE SATURATED INVERTER CHAIN
.WIDTH IN=72 OUT=72
.OPT ACCT ABSTOL=50U OPTS
.OP
M1 7 7 2 8 NMOS W=10U L=8U
M2 2 1 0 8 NMOS W=70U L=8U
M3 7 7 3 8 NMOS W=10U L=8U
M4 3 2 0 8 NMOS W=70U L=8U
M5 7 7 4 8 NMOS W=10U L=8U
M6 4 3 0 8 NMOS W=70U L=8U
M7 7 7 5 8 NMOS W=10U L=8U
M8 5 4 0 8 NMOS W=70U L=8U
M9 7 7 6 8 NMOS W=10U L=8U
M10 6 5 0 8 NMOS W=70U L=8U
VIN 1 0 PULSE(5 0 0.2N 1N 1N 5N 12N)
VDD 7 0 DC 6
.MODEL NMOS NMOS(LEVEL=2 NSUB=3E15 TOX=.1375U UEXP=0.36 NSS=4E10
+    LD=0.1M XJ=1U CBD=84U CBS=84U)
.PLOT TRAN V(6) V(1)
.PLOT TRAN V(2) V(3) V(4) V(5) V(6)
.END
```

Figure A3.29. SPUDS Input Listing for INVCHN Circuit

```
BOOTINV - BOOTSTRAPPED DOUBLE INVERTER CIRCUIT
.WIDTH IN=72 OUT=72
.OPT ACCT ABSTOL=10U OPTS
.OP
.TRAN 0.2NS 20NS
M1 1 1 3 6 NMOS W=10U L=7U AD=0.02P AS=0.02P
M2 3 2 0 6 NMOS W=50U L=7U AD=2P AS=0 02P
M3 1 1 4 6 NMOS W=10U L=7U AD=0.2P AS=0.2P
M4 1 4 5 6 NMOS W=10U L=7U AD=0.02P AS=0.02P
M5 5 3 0 6 NMOS W=50U L=7U AD=2P AS=0.02P
CL5 5 0 0.1PF
CL2 3 0 0.1PF
CB4 4 5 0.1PF
VDD 1 0 DC 12
VBB 6 0 DC -4
VIN 2 0 PULSE(10 0.4 1NS 2NS 2NS 13NS 20NS)
.MODEL NMOS NMOS(LEVEL=3 NSUB=3E15 TOX= 1375U UEXP=0.15 NSS=4E10
+   LD=0.1M XJ=1U CBD=84U CBS=84U)
.PLOT TRAN V(5) V(2)
.PLOT TRAN V(2) V(3) V(4) V(5)
.END
```

Figure A3.30. SPUDS Input Listing for BOOTINV Circuit

```
MOSMEM - MOS MEMORY CELL
.WIDTH IN=72 OUT=72
.OPT ACCT ABSTOL=1U OPTS
.TRAN 20NS 2US
VDD 9 0 DC 5
VS 7 0 PULSE(2 0 520NS 20NS 20NS 500NS 2000NS)
VW 1 0 PULSE(0 2 20NS 20NS 500NS 200NS)
VW8 2 0 PULSE(2 0 20NS 20NS 20NS 2000NS 2000NS)
M1 3 1 0 0 MOD W=250U L=5U
M2 4 2 0 0 MOD W=250U L=5U
M3 9 9 3 0 MOD W=5U   L=5U
M4 9 9 4 0 MOD W=5U   L=5U
M5 5 7 3 0 MOD W=50U  L=5U
M6 6 7 4 0 MOD W=50U  L=5U
M7 5 6 0 0 MOD W=250U  L=5U
M8 6 5 0 0 MOD W=250U  L=5U
M9 9 9 5 0 MOD W=5U   L=5U
M10 9 9 6 0 MOD W=5U  L=5U
M11 8 4 0 0 MOD W=250U L=5U
M12 9 9 8 0 MOD W=5U   L=5U
.MODEL MOD NMOS(VTO=0.5 PHI=0.7 KP=1.0E-6 GAMMA=1.83 LAMBDA=0.115
+    LEVEL=1 CGS=1U CGD=1U CBD=0.5 CBS=0.5)
.PRINT DC V(5) V(6)
.PLOT DC V(6)
.PRINT TRAN V(5) V(6) V(7) V(1) V(2)
.PLOT TRAN V(6) V(5) V(7) V(1) V(2)
.END
```

Figure A3.31. SPUDS Input Listing for MOSMEM Circuit

```
MOSAMP1 - MOS AMPLIFIER - DC/AC
.WIDTH IN=72 OUT=72
.OPT ACCT ABSTOL=10N VNTOL=10N OPTS
.DC VIN -60MV +6MV 0.66MV
.OP
.AC DEC 10 100 10MEG
M1   15 15   1 32 M W=88.9U   L=25.4U
M2    1  1   2 32 M W=12.7U   L=266.7U
M3    2  2 30 32 M W=88.9U   L=25.4U
M4   15  5   4 32 M W=12.7U   L=106.7U
M5    4  4 30 32 M W=88.9U   L=12.7U
M6   15 15   5 32 M W=44.5U   L=25.4U
M7    5  0   8 32 M W=482.6U L=12.7U
M8    8  2 30 32 M W=88.9U   L=25.4U
M9   15 15   6 32 M W=44.5U   L=25.4U
M10   6 21   8 32 M W=482.6U L=12.7U
M11  15  6   7 32 M W=12.7U   L=106.7U
M12   7  4 30 32 M W=88.9U   L=12.7U
M13  15 10   9 32 M W=139.7U L=12.7U
M14   9 11 30 32 M W=139.7U L=12.7U
M15  15 15 12 32 M W=12.7U   L=207.8U
M16  12 12 11 32 M W=54.1U   L=12.7U
M17  11 11 30 32 M W=54.1U   L=12.7U
M18  15 15 10 32 M W=12.7U   L=45.2U
M19  10 12 13 32 M W=270.5U L=12.7U
M20  13  7 30 32 M W=270.5U L=12.7U
M21  15 10 14 32 M W=254U    L=12.7U
M22  14 11 30 32 M W=241.3U L=12.7U
M23  15 20 16 32 M W=19U     L=38.1U
M24  16 14 30 32 M W=406.4U L=12.7U
M25  15 15 20 32 M W=38.1U   L=42.7U
M26  20 16 30 32 M W=381U    L=25.4U
M27  20 15 66 32 M W=22.9U   L=7.6U
CC 7 9 40PF
CL 66 0 70PF
VIN 21 0 DC -30MV AC 1
VCCP 15 0 DC +15
VCCN 30 0 DC -15
VB 32 0 DC -20
.MODEL M NMOS(NSUB=2.2E15 U0=575 UCRIT=49K UEXP=0.1 TOX=0.11U XJ=2.95U
+    LEVEL=2 CGS=1.5N CGD=1.5N CBD=45U CBS=45U LD=0.83 NSS=3.2E10)
.PLOT DC V(20)
.PRINT AC VDB(20) VP(20) VDB(66) VP(66)
.PLOT  AC VDB(20) VP(20) VDB(66) VP(66)
.END
```

Figure A3.32. SPUDS Input Listing for MOSAMP1 Circuit

```
MOSAMP2 - MOS AMPLIFIER - TRANSIENT
.WIDTH IN=72 OUT=72
.OPT ACCT ABSTOL=10N VNTOL=10N GMIN=1.0E-9 OPTS
.TRAN 0.1US 10US
RKLG 8 0 150MEG
M1   15 15   1 32 M W=88.9U   L=25.4U
M2    1  1   2 32 M W=12.7U   L=266.7U
M3    2  2 30 32 M W=88.9U   L=25.4U
M4   15  5   4 32 M W=12.7U   L=106.7U
M5    4  4 30 32 M W=88.9U   L=12.7U
M6   15 15   5 32 M W=44.5U   L=25.4U
M7    5 20   8 32 M W=482.6U  L=12.7U
M8    8  2 30 32 M W=88.9U   L=25.4U
M9   15 15   6 32 M W=44.5U   L=25.4U
M10   6 21   8 32 M W=482.6U  L=12.7U
M11  15  6   7 32 M W=12.7U   L=106.7U
M12   7  4 30 32 M W=88.9U   L=12.7U
M13  15 10   9 32 M W=139.7U  L=12.7U
M14   9 11 30 32 M W=139.7U  L=12.7U
M15  15 15 12 32 M W=12.7U   L=207.8U
M16  12 12 11 32 M W=54.1U   L=12.7U
M17  11 11 30 32 M W=54.1U   L=12.7U
M18  15 15 10 32 M W=12.7U   L=45.2U
M19  10 12 13 32 M W=270.5U  L=12.7U
M20  13  7 30 32 M W=270.5U  L=12.7U
M21  15 10 14 32 M W=254U    L=12.7U
M22  14 11 30 32 M W=241.3U  L=12.7U
M23  15 20 16 32 M W=19U     L=38.1U
M24  16 14 30 32 M W=406.4U  L=12.7U
M25  15 15 20 32 M W=38.1U   L=42.7U
M26  20 16 30 32 M W=381U    L=25.4U
M27  20 15 66 32 M W=22.9U   L=7.6U
CC 7 9 40PF
CL 66 0 70PF
VIN 21 0 PULSE(0 5 1NS 1NS 1NS 5US 10US)
VCCP 15 0 DC +16
VDDN 30 0 DC -14
VB 32 0 DC -20
.MODEL M NMOS(NSUB=2.2E15 U0=575 UCRIT=49K UEXP=0.1 TOX=0.11U XJ=2.95U
+    LEVEL=2 CGS=1.5N CGD=1.5N CBD=45U CBS=45U LD=0.83 NSS=3.2E10)
.PRINT TRAN V(20) V(66)
.PLOT  TRAN V(20) V(66)
.END
```

Figure A3.33. SPUDS Input Listing for MOSAMP2 Circuit

```
MOSAMP3 - CORE AMP
M1 6 3 5 2 W W=236.0E-6 L=9.0E-6 AD=1.2E-9 AS=1.2E-9
M2 7 4 5 2 W W=236.0E-6 L=9.0E-6 AD=1.2E-9 AS=1.2E-9
M3 1 6 6 2 D W=8.0E-6 L=60.0E-6 AD=9.0E-10 AS=9.0E-10
M4 1 7 7 2 D W=8.0E-6 L=60.0E-6 AD=9.0E-10 AS=9.0E-10
M5 9 7 8 2 M W=472.0E-6 L=8.0E-6 AD=2.4E-9 AS=2.4E-9
M6 10 6 8 2 M W=472.0E-6 L=8.0E-6 AD=2.4E-9 AS=2.4E-9
M7 1 9 9 2 D W=32.0E-6 L=60.0E-6 AD=4.0E-10 AS=4.0E-10
M8 1 10 10 2 D W=32.0E-6 L=60.0E-6 AD=4.0E-10 AS=4.0E-10
M9 5 8 20 2 M W=236.0E-6 L=9.0E-6 AD=1.2E-9 AS=1.2E-9
M10 8 17 20 2 M W=944.0E-6 L=9.0E-6 AD=4.8E-9 AS=4.8E-9
M11 7 7 13 2 D W=48.0E-6 L=8.0E-6 AD=5.0E-10 AS=5.0E-10
M12 6 6 14 2 D W=48.0E-6 L=8.0E-6 AD=5.0E-10 AS=5.0E-10
C1 14 10 5.0E-12
C2 13 9 5.0E-12
C3 9 17 2.0E-12
C4 10 17 2.0E-12
.SUBCKT SAMPLE 2 72 73 74 75 76 77 81 82
M1 72 76 81 2 W W=9.0E-6 L=8.0E-6 AD=1.1E-10 AS=1.1E-10
M2 73 77 81 2 W W=9.0E-6 L=8.0E-6 AD=1.1E-10 AS=1.1E-10
M3 74 76 82 2 W W=9.0E-6 L=8.0E-6 AD=1.1E-10 AS=1.1E-10
M4 75 77 82 2 W W=9.0E-6 L=8.0E-6 AD=1.1E-10 AS=1.1E-10
C1 81 82 1.0E-12
C2 82 2 1.0E-13
**.ENDS
.ENS SAMPLE
M21 26 19 25 2 W W=236.0E-6 L=9.0E-6 AD=1.2E-9 AS=1.2E-9
M22 25 28 20 2 M W=118.0E-6 L=9.0E-6 AD=1.2E-9 AS=1.2E-9
M23 28 28 20 2 M W=118.0E-6 L=9.0E-6 AD=1.2E-9 AS=1.2E-9
M24 19 25 20 2 M W=236.0E-6 L=9.0E-6 AD=1.2E-9 AS=1.2E-9
M25 1 26 28 2 M W=118.0E-6 L=9.0E-6 AD=1.2E-9 AS=1.2E-9
M26 1 19 19 2 D W=16.0E-6 L=60.0E-6 AD=4.0E-10 AS=4.0E-10
M27 1 26 26 2 D W=8.0E-6 L=60.0E-6 AD=9.0E-10 AS=9.0E-10
X2 2 25 17 27 10 15 16 91 92 SAMPLE
X3 2 25 17 29 9 15 16 93 94 SAMPLE
X4 2 19 3 18 0 15 16 95 96 SAMPLE
X5 2 19 4 38 0 15 16 97 98 SAMPLE
C1A 10 4 5.00E-12
C1B 9 3 5.00E-12
C2A 10 0 5.00E-12
C2B 9 0 5.00E-12
C3A 3 0 5.00E-12
C3B 4 0 5.00E-12
VDD 1 0 DC -5.0 PULSE(-5.0 5.0 0 0US 25.0NS 25 0NS 1.0NS 2 0NS)
VBS 2 0 DC -5.0
VSS 20 0 DC -5.0
V15 15 0 PULSE(-5.0 5.0 0.00US 25.0NS 25 0NS 1900 0NS 4000.0NS)
V16 16 0 PULSE(-5.0 5.0 2.05US 25.0NS 25 0NS 1900 0NS 4000 0NS)
V18 18 0 PULSE( 0.0 3.5 50.0US 25.0NS 25.0NS 500 0US 1000 0US)
V38 38 0 PULSE( 0 0 -3 5 50.0US 25 0NS 25 0NS 500 0US 1000 0US)
V27 27 0 -3.5
V29 29 0 3.5
*
*ENHANCEMENT TRANSISTOR
 MODEL N NMOS(LEVEL=2 CGS=1.40E-10 CGD=1.40E-10 MSS=-2 5E0
+ RS=0 RD=0
+ TOX=0.7E-7 PB=0.86 NSUB=7.72E14 XJ=3 5E-7 LD=0.3
+ UO=784.904 UCRIT=2.537E4 UEXP=0.00828 UTRA=0.25 TPS=1
+ NGATE=1.0E20 NFS=1.00E11 CGB=1.6E-11)
*DEPLETION TRANSISTOR
 MODEL D NMOS(LEVEL=2 CGS=1.40E-10 CGD=1.40E-10 MSS=0.5E1
+ RS=0 RD=0
+ TOX=0.7E-7 PB=0.86 NSUB=7.66E14 XJ=3 5E-7 LD=0.3
+ UO=716.999 UCRIT=1.526E3 UEXP=0.108 UTRA=0.25 TPS=1
+ NGATE=1.0E20 NFS=1.00E11 CGB=1.6E-11)
*LOW THRESHOLD ENHANCEMENT
 MODEL W NMOS(LEVEL=2 CGS=1.40E-10 CGD=1.40E-10 MSS=-2 0E10
+ RS=0 RD=0
+ TOX=0.7E-7 PB=0.86 NSUB=7.72E14 XJ=3 5E-7 LD=0.3
+ UO=784.904 UCRIT=2.537E4 UEXP=0.00828 UTRA=0.25 TPS=1
+ NGATE=1.0E20 NFS=1.00E11 CGB=1.6E-11)
*LONG CHANNEL ENHANCEMENT
 MODEL M NMOS(LEVEL=2 CGS=1.40E-10 CGD=1.40E-10 MSS=-3.0E11
+ RS=0 RD=0
+ TOX=0.7E-7 PB=0.86 NSUB=8.86E14 XJ=3 5E-7 LD=0.3
+ UO=642.999 UCRIT=1.596E3 UEXP=0.188 UTRA=0.25 TPS=1
+ NGATE=1.0E20 NFS=1.00E11 CGB=1.6E-11)
*SHORT CHANNEL DEPLETION
 MODEL P NMOS(LEVEL=2 CGS=1.40E-10 CGD=1.40E-10 MSS=9.50E11
+ RS=0 RD=0
+ TOX=0.7E-7 PB=0.86 NSUB=9.274E14 XJ=3.5E-7 LD=0.3
+ UO=593.387 UCRIT=6.136E4 UEXP=0.141 UTRA=0.25 TPS=1
+ NGATE=1.0E20 NFS=1.00E11 CGB=1.6E-11)
*** TRAN required 3hr 12min on VAX
***.TRAN 100.0NS 60.0US 45.0US
 TRAN 10NS 70NS
 PLOT TRAN V(9,10) V(9) V(15) V(3,4) V(3)
 PRINT TRAN V(91) V(92) V(93) V(94) V(95) V(96) V(97) V(98)
 PRINT TRAN I(VDD) V(9,10) V(9) V(3,4) V(3) V(5) V(8) V(17)
 OPTIONS LIMPTS=0 ITL5=0 NOMOD ACCT LVLCOD=2
.END
```

Figure A3.34. SPUDS Input Listing for MOSAMP3 Circuit

REFERENCES

[Appe79] D. R. Appelt, "Making it compatible and better: designing a new high-end computer," *Electronics*, October 11, 1979, pp. 131-136.

[Bane79] U. Banerjee, S.-C. Chen, D. J. Kuck, and R. A. Towle, "Time and Parallel Processor Bounds for FORTRAN-like Loops," *IEEE Trans. Comput.*, vol. C-28, September 1979, pp. 660-670.

[Barb77] M. Barbacci, "The ISPL Language," Carnegie Mellon University, Department of Computer Science, 1977.

[Barh73] R. Barham, E. Cheung, and E. Cohen, "BIAS-M, An Experimental Circuit Simulator for the IBM 1800," Integrated Circuits Group, University of California, Berkeley, June 1973.

[Beni79] L.Bening, "Developments in computer simulation of gate level physical logic," *Proc. 16th Design Automation Conference*, San Diego, California, June 1979, pp. 561-567.

[Berr71] R.D. Berry, "An Optimum Ordering of Electronic Circuit Equations for a Sparse Matrix Solution," *IEEE Trans. Circuit Theory*, vol. CT-18, January 1971, pp. 40-50.

[Bieh74] B.L. Biehl, "BIAS-D: A Semi-Interactive Circuit Analysis Program for Desktop Calculators and Minicomputers," *Proc. Eighth Asilomar Conf. on Circ., Syst., and Computers*, December 1974,

pp. 367-372.

[Boyl78] G.R.Boyle, "Simulation of Integrated Injection Logic," ERL Memo No. ERL-M78/13, University of California, Berkeley, March 1978.

[CDC79] COMPASS Version 3 Reference Manual for the CYBER 170 Series, Publication Number 60492600, Control Data Corporation, St. Paul, Minnesota.

[CDC80] CDC CYBER 200 MODEL 203 Computer System Hardware Reference Manual (Preliminary Edition), Publication Number 60256010, Control Data Corporation, St. Paul, Minnesota, May, 1980.

[CRAY76] CRAY-1 Computer Systems Hardware Description Manual, Publication Number 2240004, CRAY Research, Incorporated, Mendota Heights, Minnesota, 1976.

[CSPI] CSPI, Billerica, Massachusetts.

[Cala72] D. A. Calahan, *Computer-Aided Network Design*, New York: McGraw-Hill, 1972.

[Cala79] D.A. Calahan and W.G. Ames, "Vector Processors: Models and Applications," *IEEE Trans. Circ. and Syst.*, vol. CAS-26, September 1979, pp. 715-726.

[Cerm71] I. A. Cermak and D. B. Kirby, "Nonlinear Circuits and Statistical Design," *Bell System Technical Journal*, vol. 50, April 1971, pp. 1173-1197.

[Chaw75] B. R. Chawla, H. K. Gummel and P. Kozak, "MOTIS - An MOS Timing Simulator," *IEEE Trans. Circ. and Syst.*, vol. CAS-22, no. 13, December 1975, pp. 901-909.

[Cohe76] E. Cohen, Program Reference for SPICE2, Memo No. ERL-M592, Electronics Research Laboratory, University of California, Berkeley, June 1976.

[Cohe78] E. Cohen, L. Jensen, A. Vladimirescu and D.O. Pederson, "MICE - A Minicomputer Integrated Circuit Emulator," *Proc. Twelfth Asilomar Conf. on Circ., Syst., & Computers*, 1978, pp. 165-168.

[Comp74] A number of references may be found in *IEEE Computer*, vol. 7, No. 12, December 1974.

[Comt79] D. Comte, N. Hifdi, LAU Multiprocessor: Microfunctional Description and Technological Choices, *First European Conference on Parallel and Distributed Processing*, IFIPS, AFCET, CNRS, 1979, pp. 8-15.

[Coon79] J. Coonen, W. Kahan, J. Palmer, T. Pittman, and D. Stevenson, "A Proposed Standard for Binary Floating Point Arithmetic," *ACM SIGNUM Newsletter*, October 1979.

[DEC77V] VAX 11/780 Architecture Handbook, Digital Equipment Corporation, Maynard, Massachusetts, 1977.

[DEC78V] VAX 11/780 Hardware Handbook, Digital Equipment Corporation, Maynard, Massachusetts, 1978.

[Davi77] A. L. Davis, "The Architecture of DDM1: A Recursively Structured Data-Driven Machine," Computer Science Department,

University of Utah, Technical Report UUCS-77-113, 1977.

[Denn66] J. B. Dennis and E. C. Van Horn, "Programming Semantics for Multiprogrammed Computations," *Comm. ACM*, March 1966, pp. 143-155.

[Deso69] C. A. Desoer and E. S. Kuh, *Basic Circuit Theory*, New York: McGraw-Hill, 1969.

[Dijk68] E. W. Dijkstra, *Cooperating Sequential Processes*, in F. Genuys (ed.), *Programming Languages*, New York: Academic Press, 1968.

[FPS] Floating Point Systems, Incorporated, Portland, Oregon.

[Fan75] S. P. Fan, "SINC-S: A Computer Program for the Steady-State Analysis of Transistor Oscillators," Ph.D. dissertation, Department of EECS, University of California, Berkeley, September, 1975.

[Fan77] S. P. Fan, M. Y. Hsueh, A. R. Newton and D. O. Pederson, "MOTIS-C: A New Circuit Simulator for MOS LSI Circuits," *Proc. IEEE Int. Symp. Circ. and Syst.*, April 1977, pp. 700-703.

[Flyn72] M. J. Flynn, "Some Computer Organizations and Their Effectiveness," *IEEE Trans. Comput.*, vol. C-21, no. 9, September 1972, pp. 948-960.

[Fors67] G. Forsythe and C.B. Moler, *Computer Solution of Linear Algebraic Systems*, New Jersey: Prentice-Hall, 1967.

[Frer76] J.P. Freret, Jr., "Overcoming Wordlength Limitations in Minicomputer Aided Circuit Analysis," Ph.D. Dissertation, Stanford University, Stanford, California, May 1976.

[Gust67] F. Gustavson, W. Liniger, and R. Willoughby, "Symbolic Generation of an Optimal Crout Algorithm for Sparse Systems of Linear Equations," IBM Report RC 1852, IBM Thomas Watson Research Center, Yorktown Heights, New York, June, 1967.

[HP77E] 21MX E-Series Computer Operating and Reference Manual, Publication No. 02109-90014, Hewlett-Packard Company, Cupertino, California, August 1977.

[HP78F] HP 1000 F-Series Computer Operating and Reference Manual, Publication No. 02111-90001, Hewlett-Packard Company, Cupertino, California, June 1978.

[Hach81] G. Hachtel and A. Sangiovanni-Vincentelli, "A Survey of Nonstandard Simulation Techniques", *IEEE Proc.*, Sept. 1981, invited paper.

[Haie79] D.A.P. Haiek, "Multiprocessors: Discussion of Some Theoretical and Practical Problems," Ph.D. Thesis, University of Illinois at Urbana-Champaign, Report No. UIUCDCS-R-79-990, November 1979.

[Hajj81] I. N. Hajj, P. Yang, and T. N. Trick, "Avoiding Zero Pivots in the Modified Nodal Approach," *IEEE Trans. Circ. and Syst.*, vol.

CAS-28, April 1981, pp. 271-279.

[Hans77] P. B. Hansen, *The Architecture of Concurrent Programs*, Englewood Cliffs, N.J.: Prentice-Hall, 1977.

[Hill80] D. D. Hill and W. M. Van Cleemput, "SABLE: Multi-Level Simulation for Hierarchical Design," *Proc. IEEE Int. Symp. on Circ. and Syst.*, Houston, Texas, April, 1980, pp.431-434.

[Ho75] C.W. Ho, A.E. Ruehli, and P.A. Brennan, "The Modified Nodal Approach to Network Analysis," *IEEE Trans. Circ. and Syst.*, vol. CAS-22, no. 6, June 1975, pp. 504-509.

[Idle71] T. Idleman, F. Jenkins, W. McCalla, D. O. Pederson, "SLIC – A Simulator for Linear Integrated Circuits," *IEEE J. Solid-State Circ.*, vol. SC-6, August 1971, pp. 188-203.

[Inte79] The 8086 Family User's Manual, Publication No. 9800722-03, Intel Corporation, Santa Clara, California, October 1979.

[Inte80] The 8086 Family User's Manual -- Numerics Supplement, Publication No. 121586-001 Rev.A, Intel Corporation, Santa Clara, California, July 1980.

[Isaa66] E. Isaacson and H.B. Keller, *Analysis of Numerical Methods*, New York: John Wiley & Sons, Inc., 1966.

[Jenk71] F. S. Jenkins and S. P. Fan, "TIME: A Nonlinear DC and Time-domain Circuit Simulation Program," *IEEE J. Solid-State Circ.*, vol. SC-6, August 1971, pp. 182-188.

[Kasc79] M. J. Kascic, Jr., *Vector Processing on the CYBER 200*, Infotech State of the Art Report "Supercomputers," Infotech International Limited, Maidenhead, UK, 1979.

[Kuck72] D. J. Kuck, Y. Muraoka, and S.-C. Chen, "On the Number of Operations Simultaneously Executable in FORTRAN-like Programs and their Resulting Speed-Up," *IEEE Trans. Comput.*, vol. C-21, December 1972, pp. 1293-1310.

[Kuck77] D. J. Kuck, "A Survey of Parallel Machine Organization and Programming," *ACM Computing Surveys*, vol. 9, no. 1, March 1977.

[Kuck78] D.J. Kuck, *The Structure of Computers and Computations*, Volume 1, Wiley: New York, 1978.

[Lati81] W. W. Lattin, J. A. Bayliss, D. L. Budde, S. R. Colley, G. W. Cox, A. L. Goodman, J. R. Rattner, W. S. Richardson, and R. C. Swanson, "A 32b VLSI Micromainframe Computer System," *Digest of Tech. Papers*, IEEE International Solid-State Circ. Conf., New York, February 18-20, 1981, pp. 110-111.

[Lela] E. Lelarasmee, A. Ruehli, and A. Sangiovanni-Vincentelli, "The Waveform Decoupling Relaxation Method", IBM Technical Disclosures Bulletin, to appear.

[Mark57] H.M. Markowitz, "The Elimination Form of the Inverse and Its Application to Linear Programming," *Management Science*, vol. 3, April 1957, pp. 255-259.

[McCa71] W. J. McCalla and D. O. Pederson, "Elements of Computer-Aided Circuit Analysis," *IEEE Trans. Circuit Theory*, vol. CT-18,

January 1971, pp. 14-26.

[McCa] W. J. McCalla, Computer-Aided Circuit Simulation Techniques, pre-publication manuscript.

[McGr80] J.R. McGraw, "The VAL Language," Lawrence Livermore Laboratory, Preprint UCRL-83251 Rev. 1, December 1980.

[Moto79] Preliminary MC68000 16-bit Microprocessor User's Manual, Publication No. MC68000UM(AD), Motorola Inc., Austin, Texas, September 1979.

[Nage71] L. W. Nagel and R. Rohrer, "Computer Analysis of Nonlinear Circuits, Excluding Radiation (CANCER)," *IEEE J. Solid-State Circ.*, vol. SC-6, August 1971, pp. 166-182.

[Nage73] L. W. Nagel and D. O. Pederson, "Simulation Program with Integrated Circuit Emphasis," *Proc. 16th Midwest Symp. Circ. Theory*, Waterloo, Canada, April 1973.

[Nage75] L.W. Nagel, "SPICE2: A Computer Program to Simulate Semiconductor Circuits," Memo No. ERL-M520, Electronics Research Laboratory, University of California, Berkeley, May 1975.

[Nage80] L. W. Nagel, "ADVICE for Circuit Simulation," presented at 1980 IEEE International Symposium on Circuits and Systems, Houston, Texas, April 28-30, 1980.

[Newt77] A.R. Newton and D.O. Pederson, "Analysis Time, Accuracy, and Memory Requirement tradeoffs in SPICE2," *Proc. Eleventh Asilomar Conf. on Circ., Syst., and Computers*, Pacific Grove, Cali-

fornia, November 1977.

[Newt78] "The Simulation of Large-Scale Integrated Circuits," Memo No. UCB/ERL-M78/52, Electronics Research Laboratory, University of California, Berkeley, July 1978.

[Newt79] A. R. Newton, "Techniques for the Simulation of Large-Scale Integrated Circuits," *IEEE Trans. Circ. and Syst.*, vol. CAS-26, no. 9, September 1979, pp. 741-749.

[Newt80] A. R. Newton, "Timing, Logic, and Mixed Mode Simulation for Large MOS Integrated Circuits," NATO Advanced Study Institute on Computer Design Aids for VLSI Circuits, Sogesta-Urbino, Italy, July 1980.

[PR1ME] The FORTRAN-IV Programmer's Guide, Publication Number 3064-001, Prime Computer, Inc., Framingham, Massachusetts.

[Padu80] D. A. Padua, D. J. Kuck, and D. H. Lawrie, "High-Speed Multiprocessors and Compilation Techniques," *IEEE Trans. Comput.*, vol. C-29, September 1980, pp. 763-776.

[Rals65] A. Ralston, *A First Course in Numerical Analysis*, New York: McGraw-Hill, 1965.

[Sang79] A. Sangiovanni-Vincentelli, "On the Decomposition of Large-Scale Systems of Linear Algebraic Equations", *Proc. of the 1979 Joint Automatic Control Conference*, Denver, June 1979, invited paper.

[Szyg76] S. A. Szygenda and E. W. Thompson, "Modeling and Digital Simulation for Design Verification and Diagnosis," *IEEE Trans.*

Comput., vol. C-25, no. 13, December 1976, pp. 1242-1253.

[Tane78] A. S. Tanenbaum, "Implications of Structured Programming for Machine Architecture," *Comm. ACM*, vol. 21, no. 3, March 1978, pp. 237-246.

[Texa79] Model 990/12 Computer Assembly Language Programmer's Guide, Part No. 2250077-9701, Texas Instruments, Inc., Austin, Texas, May, 1979.

[Toma67] R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM Journal Res. and Dev.*, vol. 11, no. 1, January 1967, pp. 25-33.

[Towl76] R. A. Towle, "Control and Data Dependence for Program Transformations," Report No. UIUCDCS-R-76-788, Department of Computer Science, University of Illinois, Urbana-Champaign, March 1976.

[Tsic74] D. C. Tsichritzis, *Operating Systems*, New York: Academic Press, 1974.

[Ullm76] J. D. Ullman, "Complexity of Sequencing Problems," pp. 139-164 in E.G. Coffman (ed.), *Computer and Job-Shop Scheduling Theory*, Wiley: New York, 1976.

[Univ70] UNIVAC 1108 Processor and Storage Programmers Reference, Publication Number UP-4053, Univac Division of Sperry Rand Corporation, St. Paul, Minnesota, 1970.

[Vlad81] A. Vladimirescu, K. Zhang, A. R. Newton, and D. O. Pederson, SPICE Version 2G.2 User's Guide, Department of EECS,

University of California, Berkeley, April 1981.

[Wen76] K. Y. Wen, "Interprocessor connections – Capabilities, exploitation and effectiveness," Ph.D. dissertation, Department of Computer Science, University of Illinois, Urbana-Champaign, Report 76-830, March 1976.

[Wilc76] P. Wilcox and A. Rombeck, "F/LOGIC - An Interactive Fault and Logic Simulator for Digital Circuits," *Proc. 13th Design Automation Conf.*, 1976, pp. 68-73.

[Wilk64] J. H. Wilkinson, *Rounding Errors in Algebraic Processes*, Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1964.

[Wolf78] M. J. Wolfe, "Techniques for Improving the Inherent Parallelism in Programs," Report No. UIUCDCS-R-78-929, Department of Computer Science, University of Illinois, Urbana-Champaign, July 1978.

[Youn76] T.K. Young and R.W. Dutton, "Mini-MSINC - A Minicomputer Simulator for MOS Circuits with Modular Built-in Models," *IEEE J. Solid-State Circ.*, vol. SC-11, no. 5, October 1976, pp. 730-732.