CIRCUIT SIMULATION IN LISP

by

K. Mayaram and D. O Pederson

Memorandum No. UCB/ERL M84/60

9 August 1984

(cover)

CIRCUIT SIMULATION IN LISP

by

K. Mayaram and D. O Pederson

Memorandum No. UCB/ERL M84/60

9 August 1984

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Circuit Simulation in LISP

ABSTRACT

The programming language LISP is no longer restricted to applications in the field of artificial intelligence. It is being used in many different areas and in particular for development of LSI design tools. This report explores the speed performance of a LISP-based circuit simulation program, BIASlisp, on three different computer systems. They are the VAX 11/780, the HP9836 desktop computer, and the Symbolics 3600 LISP machine. The results obtained using several modifications are reported.

Integrated circuit simulation requires an extensive amount of numerical computation, an application for which LISP has not been very popular. However, the results on the 3600 indicate that circuit simulation is possible in LISP without any performance penalty. Moreover, BIASlisp can be interfaced with existing LSI CAD tools in LISP thereby providing circuit simulation capabilities in the LISP environment.

ACKNOWLEDGEMENTS

The authors thank the Semiconductor Research Corporation for funding this research, Hewlett Packard for equipment support, and Analog Devices for use of their LISP machine.

Andrei Vladimirescu, Jim Kleckner, and Jeff Deutsch have contributed to this work through several useful discussions.

Allan Kuchinsky helped in porting BIASlisp to the HP9836.

The support of other colleagues of the CAD group is also appreciated, especially that of Frank Ma, Ron Gyurcsik, Jeff Burns, and Ken Keller.

1. INTRODUCTION

LISP is a list-processing language which was first developed for use in artificial intelligence research. Because of its extensive use in all branches of artificial intelligence (AI), LISP has been called an AI language [1]. However, it has not been confined to AI and has found applications in the areas of text editing [2], symbolic math systems [3], LSI design [4-6] and simulation of digital logic circuits [7]. Absence of good compilers has resulted in poor speed performance of LISP-based systems in the past, and therefore it has not been a popular language for number crunching applications.

With the advent of a special computer architecture [2], [8] and efficient compilers a significant improvement in performance has resulted. This has prompted wide acceptance of LISP in the computing community. Complex systems can now be designed using LISP without any loss of performance. Moreover, program development can be greatly aided by the advantages associated with a LISP environment.

The intent of this work has been to investigate the suitability of LISP as a language for development of circuit-simulation programs. Because of the extensive amount of number crunching involved this program is also a good tool to evaluate the speed performance of LISP-based systems involving extensive numerical computation. The higher-level data structures of LISP can be used to store input data in a way that is easy to understand thereby improving program readability. Those who have worked with SPICE 2 [9] are familiar with the amount of effort that goes into understanding how and where input data is stored. This makes it difficult to modify the program whereas using LISP one can write a circuit simulation program that readily lends itself to modifications.

A LISP-based circuit simulation program, BIASlisp, has been developed for this purpose. It is an interactive MOS circuit simulation program which belongs to the family of BIAS programs developed at UC Berkeley [10-14]. Performance evaluation has been carried out using BIASlisp on different computer systems. The machines used for this study were the VAX

11/780 under the UNIX 4.2 BSD operating system, the Symbolics 3600 LISP Machine, and the HP 9836 desktop computer. *Franz LISP* was used to develop and debug the program on the VAX 11/780. This code was then ported to *Portable Standard Lisp* (PSL) on the HP 9836 and *Zetalisp* on the Symbolics 3600 by writing appropriate macros. Experiments were carried out using several modifications and the results obtained are reported in Section 3. A comparison of speeds achieved with circuit simulators written in other languages is made in [15]. It was found that a LISP-based circuit simulator gave poor performance compared to a circuit-simulation program written in C or PASCAL on the VAX 11/780 and the HP9836.

Performance evaluation of BIASlisp on the Symbolics 3600 LISP machine indicates that LISP in itself is not a slow language. However, poor performance is observed on the VAX 11/780 and the HP9836. This is due to the lack of an architecture that efficiently supports the LISP language. Some of the significant differences between LISP and conventional programming languages and the requirement for special-purpose hardware have been discussed in Section 4.

2. PROGRAM DESCRIPTION

This section gives a description of the features of the BIASlisp program, the algorithms used, and the LISP data structures used in the implementation.

2.1. Program Features

BIASlisp is an interactive MOS circuit simulation program which has a SPICE-like input format. Nodes can be given names and need not be integers. A complete description of the commands can be found in the User's Manual given in Appendix A. The MOSFET model used in the present version is the Shichman-Hodges model [16]; a SPICE-level 2 MOS model will be implemented in the future. At present the analysis types available are DC operating point, DC transfer characteristics, and time-domain transient response. The builtin dynamic memory management of LISP has been used, and therefore the circuit size that can be simulated is limited by the physical memory of the system.

2.2. Algorithms

The algorithms used for analysis are similar to those of SPICE 2 [17]. Modified nodal analysis [18] is used to setup the system of equations. The Newton-Raphson iteration technique has been used for the DC solution of the nonlinear equations [19-21]. Trapezoidal integration with a Backward-Euler start-up is used for transient analysis. Iteration count has been used for timestep control [17]. Bi-directional threaded lists with Markowitz reordering and LU decomposition have been used in the sparse-matrix implementation [20].

2.3. Implementation

A brief description of the primitive LISP data items is given below. For a more detailed treatment the reader is referred to [22-23].

2.3.1. Basic LISP Data Objects

The fundamental objects of the LISP language are called *atoms* which correspond to variables in other languages. An atom can be either a number (integer or real) or a symbol. Groups of atoms form *lists* and lists themselves can be grouped together to form lists. Atoms and lists are called *Symbolic Expressions* (or S-expressions for short). These S-expressions are the primitive data items of LISP. Each symbol has a property list in which relevant properties of that atom can be stored. This is a unique feature of LISP.

2.3.2. Data Structures

For each element type allowed in BIASlisp there is a list which stores the names of all elements of that type in the circuit. LISP list operators are used to store and retrieve data from these lists. The various lists and their purpose are described in Appendix B. As an example: Res-list stores the names of all resistors in the circuit. If the resistor names are R1, R2, ..., RN then Res-list is (R1 R2 ..... RN).

Associated with each name in these lists is a property list that stores the relevant properties of that name. Node names/numbers, element values, and model parameters are regarded as properties in this scheme. There are built-in LISP functions which allow insertion and deletion of properties in a property list. The property list for each type of element is described in Appendix C and two examples are given in Table 1 below:

Type	Property name	Description
Resistor	n1	positive node
	n2	negative node
	value	resistance value
MOS model	mtype	model type (nmos/pmos)
	vto	threshold voltage (vbs = 0)
	kp	intrinsic transconductance parameter
	lambda	channel length modulation parameter
	gamma	bulk threshold parameter
	phi	surface potential
	js	bulk junction saturation current density

TABLE 1

Property lists for a Resistor and Mosfet

2.3.2.1. Sparse-matrix implementation

There are two implementations of the sparse-matrix version: one makes use of the property list as a sparse-matrix element and the other uses structures (similar to records of PASCAL language and structures of C language). A sparse-matrix element stores the row and column indices, the value, and pointers to the next element in its row and its column. The sparse-matrix elements are linked together by bidirectional pointers according to the method given in [24]. Pointers to the first element in a column or a row are stored in two one-dimensional arrays. Since LISP arrays can be allocated dynamically this does not place any restriction on the size of the sparse-matrix as does PASCAL [12]. The schematic of such an implementation is shown in Figure 1 and is independent of the actual data structures used.

2.3.2.2. Full-matrix implementation

The full-matrix version uses a two-dimensional LISP array on the VAX and the LISP machine. However, PSL on the HP9836 allows only one-dimensional arrays and so the implementation makes use of vectors. Since arrays can be allocated dynamically in LISP this does not place any limitation on the size of the circuit that can be simulated. Partial pivoting is used during the solution phase.

$$A = \begin{bmatrix} 5 & 3 & 2 & 1 \\ 7 & 2 & 0 & 0 \\ 0 & 1 & 3 & 0 \\ 0 & 7 & -13 & 1 \end{bmatrix}$$

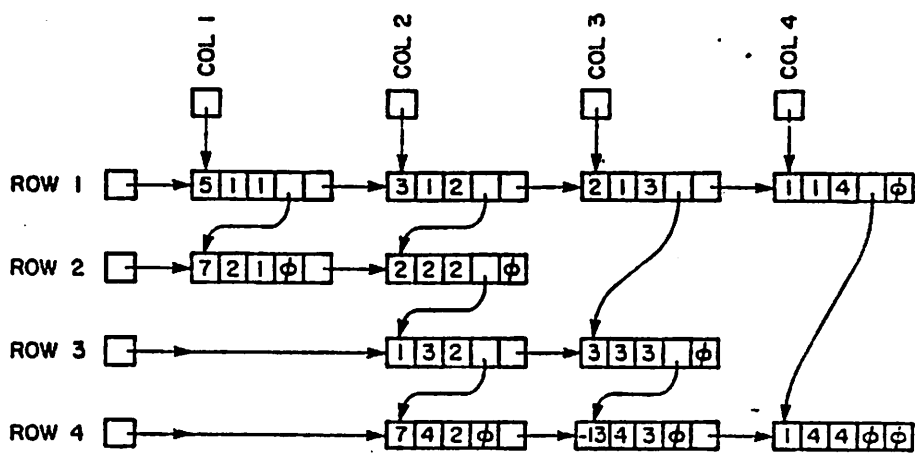

Figure 1.
The bidirectional threaded list.

3. RESULTS AND COMPARISONS

This section presents a comparison of CPU times for program execution on the three different systems. Three circuits have been used as benchmarks and are given as Figures 2 to 4. The first circuit is a ± 2.5V fast-settling CMOS opamp [25] which uses model parameters from a typical CMOS N-well process [26]. It is connected in a unity-gain configuration with the noninverting input at zero volts. Circuit 2 is a CMOS sense amplifier whereas Circuit 3 is a chain of twenty five NMOS depletion-load inverters. Both of these use model parameters from a CMOS N-well process [27]. The results are for DC operating point analysis and give a good idea about simulation speed. A comparison has also been made between the full-matrix and the sparse-matrix versions. Simulation results using BIASlisp have been given for each machine separately and some conclusions are derived thereafter. SPICE 2G.6 timings on the VAX 11/780 with a floating-point accelerator are also given for comparison purposes.

At the outset it is instructive to study the times for some basic operations in LISP. These times are given in Section 3.1.

3.1. Times for basic operations in LISP

In this section the operation times for the LISP dialects in use on the three computer systems are reported. They are *Franz LISP* on the VAX 11/780, *Portable Standard Lisp* on the HP9836, and *Zetalisp* on the Symbolics 3600. A comparison of the operation times is made with those in another programming language on the same system. FORTRAN was used on the VAX 11/780 and PASCAL on the HP9836. Operation times on the Symbolics 3600 are compared with those for FORTRAN on the VAX 11/780 using a floating-point accelerator.

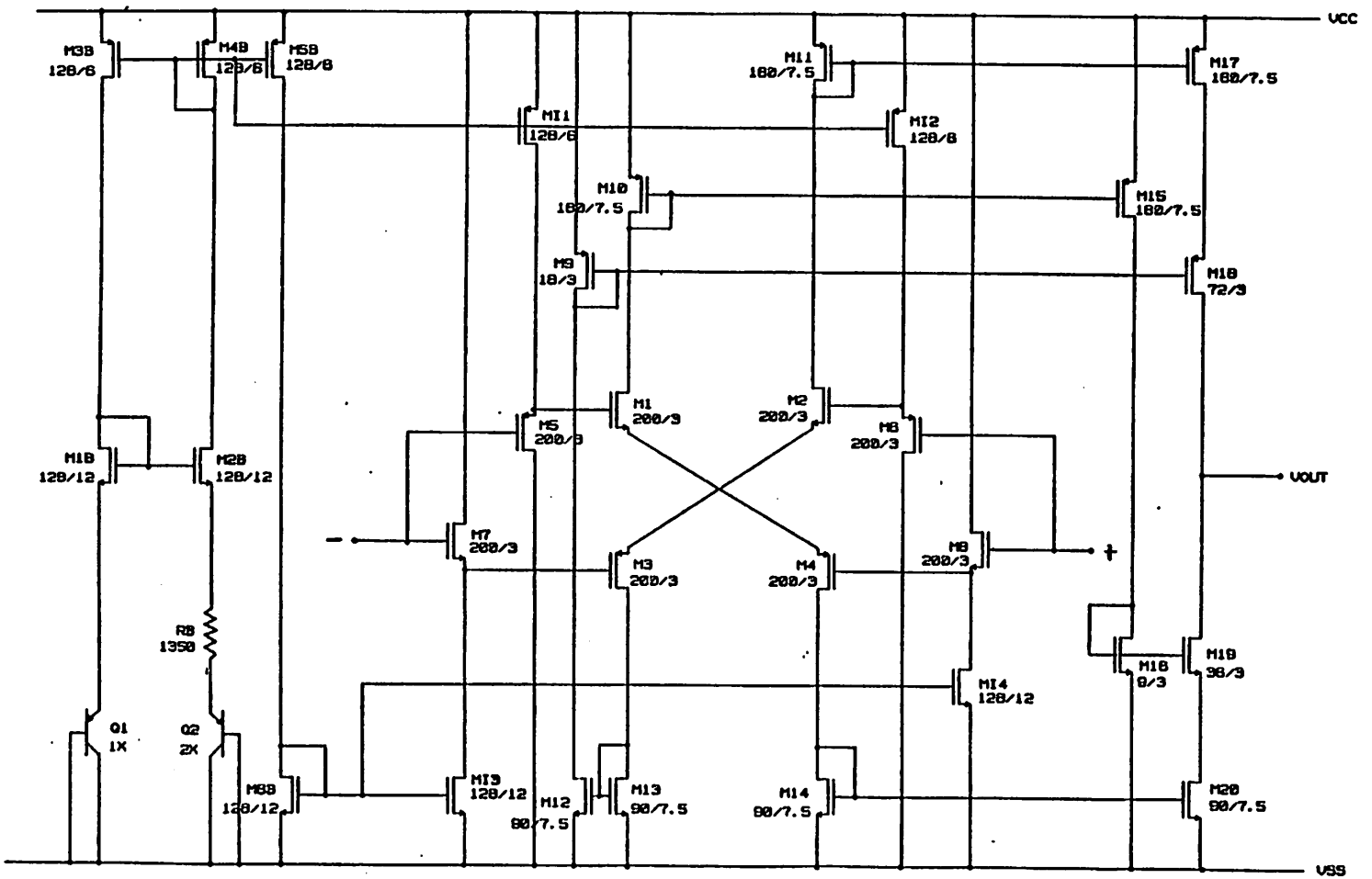
VCC
M3B 128/6
M4B 128/6
M5B 128/8
M11 180/7.5
M17 180/7.5
MI1 128/6
MI2 128/8
M10 180/7.5
M15 180/7.5
M9 18/3
M18 72/3
M1 200/3
M2 200/3
M5 200/3
M6 200/3
M1B 128/12
M2B 128/12
VOUT
M7 200/3
M3 200/3
M4 200/3
M8 200/3
RB 1350
MI4 128/12
M16 9/3
M19 36/3
Q1 1X
Q2 2X
M3B 128/12
MI3 128/12
M12 90/7.5
M13 90/7.5
M14 90/7.5
M20 90/7.5
VSS

Figure 2.

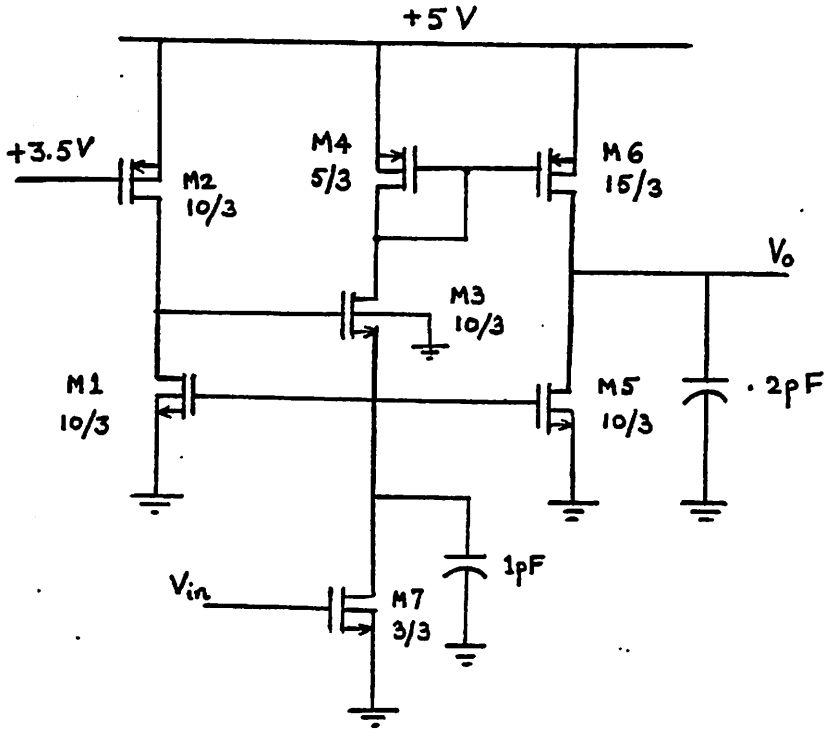

Figure 3.

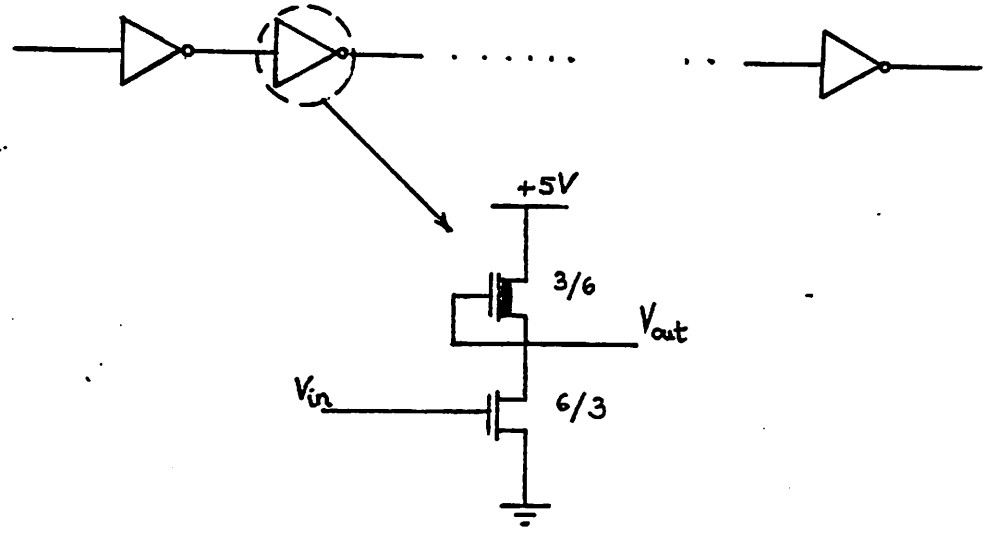

Figure 4.

3.1.1. VAX 11/780: Comparison of FORTRAN and *Franz LISP* Operation times

A comparison is made for some of the basic operation times in FORTRAN and *Franz LISP* on the VAX 11/780. Table 2 gives the time spent in doing the assignment yn(i,j) = yn(i,j), where yn is a two dimensional array, for different values of i and j. The time for garbage collection* has not been considered as that is variable. It is clear from Table 2 that arrays of floating-point-numbers (type is flonum-block) are inefficient compared to general arrays (type *t* array in *Franz LISP*). In Table 3 are presented the times for nested and single *do* loops as well as the times for executing 100,000 times the basic numeric operations +, -, *, /, and *exp*. The time taken by the functions *sqrt* and *log* varied and *Franz LISP* took approximately 20 sec more than the corresponding operation in FORTRAN. Each operation was repeated 100,000 times. It should be again mentioned that the time for garbage collection is not included as it varied from 10 sec to 150 sec. Needless to mention *Franz LISP* is very slow compared to FORTRAN even if one unrealistically ignores the time for garbage collection.

* Garbage collection is a process in which memory space occupied by objects that are no longer addressed is reclaimed. LISP has a builtin garbage collection package which recovers such sections of the memory whenever it runs out of space for new objects. This process is time consuming and the time required is therefore a parameter of interest.

	LISP (type)		FORTRAN (precision)	
#of times	t (sec)	flonum-block (sec)	double (sec)	single (sec)
1000x100	27.5	85.5	3.0	2.07
100x1000	29.3	88.0	4.9	3.9
100x100	2.68	8.6	0.45	0.36
10x1000	3.16	9.65	2.0	1.90

TABLE 2

Comparison of LISP and FORTRAN array reference timings on the VAX 11/780

operation	LISP (sec)	FORTRAN (sec)
Nested DO loops		
1000x100	6.2	0.38
100x1000	6.2	0.38
10000x10	7.8	0.43
10x10000	8.3	0.38
1x100000	15.0	0.38
100000x1	25.4	1.2
Single DO loop		
100000 times	7.6	0.35
'+'	15.0	0.35
'-'	15.0	0.35
'*'	16.0	0.55
'/'	16.0	1.03
exp	40.0	22.4

TABLE 3

Time taken for operations in LISP and FORTRAN on the VAX 11/780

3.1.2. HP 9836: Operation times in *Portable Standard Lisp*

Portable Standard Lisp (PSL) on the HP9836 uses software for floating-point calculations and, therefore, the floating-point arithmetic is very inefficient. Some functions require more computation effort depending on their arguments so it is not possible to give a representative number for the execution speed. The functions in this category are exp(), sqrt(), and log(). As an example Table 4 gives the computation time required to find log(x) for different values of x 1000 times. For comparison times for a PASCAL program are also given. It should be noted that although the values of x being used have the same number of digits the computation time may vary by a factor of two or more. Operations which gave consistent execution times are compared with execution times in PASCAL in Table 5. The garbage collection time for these operations is not included. Each operation vector† assignment v(i) = v(i), +, -, *, and / has been repeated 100,000 times.

PSL is slow compared to PASCAL in performing the simple arithmetic operations +, -, *, and /. The reason for this poor performance is due to the tag checking performed during runtime as explained in Section 4.

† PSL on the HP9836 has two types of vectors: a slow vector and a fast vector. As seen in Table 5 the fast vector is six times faster than the slow vector.

log ()	LISP (sec)	PASCAL (sec)
10000	21.8	3.6
30000	22.8	3.7
40000	29.9	3.6
44000	22.9	3.6
45000	57.2	3.6
46000	56.5	3.6
47000	56.6	3.6
50000	49.5	3.7
80000	31.9	3.6
90000	31.0	3.6
100000	23.9	3.6

TABLE 4

Computation time for log() in LISP and PASCAL on the HP9836 1000 times

operation	LISP (sec)	PASCAL (sec)
v(i) = v(i) fast vector	4.1	3.9 ‡
v(i) = v(i) slow vector	23.8	
Single DO loop 100000 times	9.4	1.29
'+'	52.6	11.25
'-'	54.6	12.25
'*'	63.6	26.75
'/'	81.6	38.75

TABLE 5

Time taken for numerical operations in PSL and PASCAL on the HP9836 100,000 times

‡ PASCAL on the HP9836 does not allow a vector of dimension 100,000. This result was obtained for an array of dimension 1000 and then scaled up by a factor of 100. The time required for the vector operation v(i) = v(i) 1000 times is 0.039 sec. Moreover, there is only one type of vector in PASCAL.

3.1.3. Symbolics 3600: Operation times in *Zetalisp*

A comparison between the times for some basic operations using single-precision and double-precision arithmetic* in *Zetalisp* on the 3600 and FORTRAN on the VAX 11/780 with a floating-point accelerator is made in Table 6. Again no definite times could be obtained for the function *sqrt(x)*. The times for LISP on the 3600 vary between 17 sec to 76 sec and that for FORTRAN on the VAX 11/780 vary between 25 sec to 64 sec for identical values of x used in the expression *sqrt(x)*. Each operation is repeated 100,000 times.

operation	LISP (sec)		FORTRAN (sec)	
Nested DO loops				
1000x100	0.38		0.38	
100x1000	0.36		0.38	
10000x10	0.44		0.43	
10x10000	0.36		0.38	
1x100000	0.36		0.38	
100000x1	1.21		1.20	
Single DO loop 100000 times	0.36		0.35	
	single-precision	double-precision	single-precision	double-precision
array reference yn(i,j)=yn(i,j) 1000x100 times	2.47	2.47	2.07	3.0
'+'	1.06	15.36	0.18	0.35
'-'	1.06	16.36	0.18	0.35
'*'	1.06	17.0	0.19	0.55
'/'	1.81	21.16	0.52	1.03
exp	18.36	56.0	22.3	22.4
log	19.03	55.1	‡	‡

TABLE 6

Time taken for operations in LISP on the Symbolics 3600 and FORTRAN on the VAX 11/780

* Both single-precision and double-precision arithmetic are available on the Symbolics 3600. *Franz LISP* and PSL only support double-precision arithmetic.

‡ FORTRAN timings on the VAX 11/780 for calculation of *log(x)* vary from 17 sec to 52 sec for the x values used. Since a consistent number was not obtained it is not reported.

Table 6 indicates that LISP on the 3600 is slower than FORTRAN on the VAX 11/780 for basic numeric operations +, -, *, and / by a factor of three to six in single-precision arithmetic and by a factor of twenty to forty for double-precision arithmetic. A comparison of the $exp(x)$ and $log(x)$ shows that *Zetalisp* performs better in single-precision arithmetic!

From Tables 2 to 6 it is seen that LISP is very slow on the VAX 11/780 and the HP9836 when compared to FORTRAN or PASCAL. However, on the Symbolics 3600 the speed is reasonable after accounting for the absence of floating-point hardware. These results give an idea of the circuit-simulation performance that can be achieved in LISP on these machines. The LISP machine is the obvious winner. A circuit-simulation program in "pure" LISP on the VAX will be very slow and has to be speeded up by some other techniques. Though the time taken for arithmetic operations in LISP on the HP9836 is quite large compared to that on the VAX 11/780, BIASlisp on the HP9836 is only slower by a factor of two compared to BIASlisp on the VAX 11/780 as shown in Section 3.5.

3.2. SPICE2 Simulation Results

Simulation results (timings and number of iterations) from SPICE 2G.6 are given for the three circuits in Table 7 below. The solution to the operating point for Circuit 1 could be obtained only after use of the '.nodeset' option on one node in SPICE [28]. To have a consistency in comparison the same option was used in BIASlisp for Circuit 1.

Description	Circuit 1	Circuit 2	Circuit 3
# iterations	20	22	38
Readin (sec)	1.88	0.85	2.42
Setup time (sec)	0.70	0.22	0.63
Analysis (sec)	**4.70**	**1.27**	**8.37**
Matrix load (sec)	*2.35*	*0.67*	*5.80*
LU decomp. (sec)	*0.85*	*0.10*	*0.45*
DC solution (sec)	*0.18*	*0.00*	*0.25*
Time/iteration (sec)	**0.24**	**0.06**	**0.22**

Table 7

SPICE Simulation Results

In Table 7 *Readin* indicates the time spent in reading and storing the circuit description. *Setup time* is the time spent in setting up the pointers for the sparse-matrix. *Analysis* denotes the total time spent in the analysis phase. This time is essentially the time spent in loading up the nodal admittance matrix with conductances, the time required to perform the LU decomposition, and the time spent in obtaining the solution by forward and back substitution. Table 7 gives these times as *Matrix load*, *LU decomp.*†, and *DC solution*, respectively. The analysis time is given in bold letters in all the tables because it indicates the total time used by the simulation engine, and is the number that is used for comparisons. Since the time per iteration

† For SPICE this time includes the time required for reordering the equations. SPICE uses a combination of the Markowitz algorithm and partial pivoting as the reordering strategy. This is done only for the first iteration to avoid substantial overhead. BIASlisp uses a pure Markowitz reordering algorithm and reorders the equations during the setup phase.

is a useful parameter when comparing different computer systems it is also given in bold letters.

3.3. BIASlisp on VAX 11/780: Full-matrix Implementation

The version of BIASlisp on the VAX was basically used to develop and debug the program. It has been written in *Franz LISP* [29] and runs under UNIX BSD 4.2. Table 8 gives a summary of results from the full-matrix version of BIASlisp. This table depicts the same type of information as Table 7 and in addition gives the time required for MOSFET related computations, denoted by *MOS calculations**, and the time required for garbage collection, indicated by *Garbage collection*. It can be seen that *MOS calculations* which is the time used for evaluation of MOS conductances and the time spent in loading these conductances in the nodal admittance matrix is a major contributor to the *Matrix load* time. The *Setup time* has significance only for the sparse-matrix version since during this phase the sparse-matrix pointers are created and reordering is performed‡. In Tables 8 to 14 this time indicates the time required to create the array/vector for the full-matrix implementation and is, therefore, a small number.

Note that there is a difference in the number of iterations required by SPICE and BIASlisp. Under such a situation the time per iteration is a useful quantity for comparisons.

A comparison with the time taken by SPICE shows that all of the circuits require a huge amount of CPU time for simulation. Most of this time is spent in the model evaluations and the solution stage. Another data of interest is the time spent in garbage collection. LISP's dynamic memory manager has it's own garbage collector which provides a safe dynamic

* SPICE run-time statistics do not give the time required for calculation of MOS conductances, therefore, this time has been not reported for SPICE.

‡ For BIASlisp the *Setup time* includes the time required for reordering. This is different from the *Setup time* of SPICE in Table 7.

Description	Circuit 1	Circuit 2	Circuit 3
# iterations	14	23	26
Readin (sec)	2.5	0.95	4.2
Setup time (sec)	0.03	0.03	0.02
Analysis (sec)	**127.4**	**34.8**	**245.6**
Matrix load (sec)	*46.25*	*19.9*	*137.4*
MOS calculations (sec)	*39.7*	*17.0*	*126.6*
DC solution (sec)	*78.7*	*13.1*	*103.6*
Garbage collection (sec)	18.1	6.21	40.68
Time/iteration (sec)	**9.1**	**1.51**	**9.45**

TABLE 8

BIASlisp Simulation Results on VAX 11/780: full-matrix Version

memory management package. However, one pays for it in performance as seen in Table 8. There is not much that can be done to reduce the time spent in garbage collection. The full-matrix DC solution stage is definitely very slow and this is quite distressing when one compares this with the time spent in SPICE to solve the equations. This then motivates the use of FORTRAN in the solution phase.

It is possible to interface functions and/or subroutines written in a language other than LISP with a *Franz LISP* program. This technique was used to write a FORTRAN equation solver for BIASlisp. The results of this experiment are presented in Table 9. There is a drastic reduction in solution time and an overall factor of 1.5 to 2 improvement in the analysis time. However, the time spent in MOSFET calculations increases which boosts-up the *Matrix load* time by approximately thirty percent. The reason for this is that data can be transferred between the FORTRAN program and the LISP code only by use of arrays of floating-point numbers (*arrays of type flonum-block*) which are inefficient in *Franz LISP* as can be seen from Table 2. After conductances have been calculated they are loaded into the admittance matrix where inefficient array referencing leads to the increase in time. The array used in obtaining the results for Table 8 was of a general type which is more efficiently handled in *Franz LISP* as indicated in Table 2.

Description	Circuit 1	Circuit 2	Circuit 3
# iterations	14	23	26
Readin (sec)	2.45	0.92	4.2
Setup time (sec)	0.03	0.02	0.02
Analysis (sec)	**66.3**	**28.1**	**194.6**
Matrix load (sec)	*61.8*	*24.75*	*186.8*
MOS calculations (sec)	*54.5*	*21.9*	*174.55*
DC solution (sec)	*1.26*	*0.38*	*1.83*
Garbage collection (sec)	14.4	6.2	41.16
Time/iteration (sec)	**4.74**	**1.22**	**7.48**

TABLE 9

BIASlisp Simulation Results on VAX 11/780: full-matrix Version with FORTRAN solve

Two distinct operations are performed in the MOSFET evaluations. One is the computation of incremental conductances and the equivalent currents, and the other is the loading of these conductances and currents into the nodal admittance matrix. The time required for the latter operation may be significant with inefficient array referencing. To speed up mosfet model evaluation, the model equations were coded in FORTRAN as well. Table 10 gives the simulation time with a FORTRAN solve subroutine and a FORTRAN subroutine for calculation of MOS conductances. No significant speed up is observed. Note that there is a change in the number of iterations.

Another thing that was tried was to load the MOS conductances in the admittance matrix in a FORTRAN coded subroutine. Table 11 gives the results with this modification: FORTRAN subroutines are used for calculation of MOS conductances, loading the MOS conductances in the nodal admittance matrix, and for the solution of the system of equations. There is a reduction in the time spent in MOSFET evaluation by a factor of two. This suggests that the time spent in computing the conductances is not significant and the array references, for loading the admittance matrix, are degrading the performance. In Table 12 this claim is verified. The results are obtained by calculating the MOS conductances in LISP and loading these conductances in the nodal admittance matrix in FORTRAN. A FORTRAN solve

Description	Circuit 1	Circuit 2	Circuit 3
# iterations	15	18	26
Readin (sec)	2.2	0.96	3.65
Setup time (sec)	0.05	0.05	0.03
Analysis (sec)	**65.8**	**20.3**	**169.5**
Matrix load (sec)	*60.2*	*17.2*	*158.9*
MOS calculations (sec)	*52.9*	*13.54*	*146.6*
DC solution (sec)	*1.33*	*0.25*	*1.83*
Garbage collection (sec)	12.1	4.13	32.47
Time/iteration (sec)	**4.39**	**1.12**	**6.52**

TABLE 10

BIASlisp Simulation Results on VAX 11/780: full-matrix version with FORTRAN solve and FORTRAN MOS conductance calculations

subroutine has also been used. At this stage it must be pointed out that though the MOSFET model evaluation takes less time, it is still not as fast as that of SPICE.

Description	Circuit 1	Circuit 2	Circuit 3
# iterations	15	18	26
Readin (sec)	2.45	0.9	3.27
Setup time (sec)	0.05	0.02	0.05
Analysis (sec)	**43.61**	**15.2**	**109.4**
Matrix load (sec)	*38.7*	*11.9*	*99.7*
MOS calculations (sec)	*30.62*	*9.15*	*7.2*
DC solution (sec)	*1.33*	*0.25*	*1.9s*
Garbage collection (sec)	9.48	4.1	21.5
Time/iteration (sec)	**2.91**	**0.84**	**4.21**

TABLE 11

BIASlisp Simulation Results on VAX 11/780: full-matrix version with FORTRAN solve, FORTRAN MOS conductance, and FORTRAN loading of MOS conductances in the nodal admittance matrix

LISP does seem to be at a big disadvantage when compared to a language such as FORTRAN. One more experiment was carried out. MOSFET model parameters and conductances were no longer stored in property lists but structures were used instead. It was expected that

Description	Circuit 1	Circuit 2	Circuit 3
# iterations	15	18	26
Readin (sec)	2.5	0.88	3.6
Setup time (sec)	0.03	0.02	0.02
Analysis (sec)	**46.5**	**15.9**	**116.2**
Matrix load (sec)	*40.7*	*12.63*	*105.5*
MOS calculations (sec)	*2.65*	*9.9*	*92.8*
DC solution (sec)	*1.27*	*0.98*	*1.88*
Garbage collection (sec)	11.8	4.68	29.0
Time/iteration (sec)	**3.1**	**0.88**	**4.47**

TABLE 12

BIASlisp Simulation Results on VAX 11/780: full-matrix version with FORTRAN solve and FORTRAN loading of MOS conductances in the nodal admittance matrix

structures, being faster than property lists, would give some speed improvement. Table 13 gives the results and upon comparing with Table 8 we find that the improvement in speed is insignificant.

Description	Circuit 1	Circuit 2	Circuit 3
# iterations	14	23	26
Readin (sec)	2.5	0.95	4.2
Setup time (sec)	0.05	0.03	0.05
Analysis (sec)	**118.6**	**35.0**	**249.5**
Matrix load (sec)	*41.7*	*19.1*	*127.5*
MOS calculations (sec)	*36.1*	*15.0*	*116.2*
DC solution (sec)	*74.3*	*14.1*	*117.2*
Garbage collection (sec)	18.8	8.22	44.1
Time/iteration (sec)	**8.47**	**1.52**	**9.6**

TABLE 13

BIASlisp Simulation Results on VAX 11/780: full-matrix version with structures used for MOSFET storage

It has been noted in [29] that hunks* are faster than arrays. However, their size imposes

* A hunk is a vector of maximum length 128.

a limitation on the maximum number of nodes in the circuit (less than or equal to 11). This is a severe limitation and hence hunks were not tried. Vectors are also available in *Franz LISP* and are intermediate in speed to a hunk and an array. Table 14 gives the simulation results obtained by using a full-matrix version implemented with vectors. These results can again be compared with Table 8. It is seen that the time per iteration does not change significantly. On the contrary it has increased slightly and there seems to be no advantage in using vectors. Note that the reduced analysis time for Circuit 3 is only due to the smaller number of iterations required for convergence. The time per iteration is in fact worse.

Description	Circuit 1	Circuit 2	Circuit 3
# iterations	14	23	23
Readin (sec)	2.4	0.98	3.53
Setup time (sec)	0.05	0.03	0.05
Analysis (sec)	**137.0**	**37.5**	**226.8**
Matrix load (sec)	*53.5*	*19.8*	*138.0*
MOS calculations (sec)	*42.3*	*16.0*	*117.9*
DC solution (sec)	*80.9*	*15.2*	*85.4*
Garbage collection (sec)	17.6	6.35	36.5
Time/iteration (sec)	**9.8**	**1.63**	**9.86**

TABLE 14

BIASlisp Simulation Results on VAX 11/780: full-matrix implementation using vectors

3.4. BIASlisp on VAX 11/780: Sparse-matrix Implementation

As stated earlier there were two different ways in which the sparse-matrix was implemented. One makes use of structures and the other makes use of property lists. Results obtained from these versions are given in Table 15 and Table 16, respectively. To compare these two implementations carefully the time required for some more operations is also given. In particular the *Setup time* has been decomposed into the time required to create the matrix pointers, fillins, and the time used in reordering. These times are indicated by *Matrix pointers*,

LU fillins, and *Reordering*, respectively. It can be seen that the implementation using structures takes less time in reordering compared to the property list implementation resulting in smaller setup times. However, the analysis times do not differ significantly: the implementation using structures requires a slightly smaller time.

Description	Circuit 1	Circuit 2	Circuit 3
# iterations	14	23	26
Readin (sec)	2.35	1.05	4.2
Setup time (sec)	7.2	1.67	9.9
Matrix pointers (sec)	*1.85*	*0.92*	*2.71*
Reordering (sec)	*4.7*	*0.57*	*6.75*
LU fillins (sec)	*0.23*	*0.02*	*.03*
Analysis (sec)	**53.35**	**23.05**	**136.2**
Matrix load (sec)	*45.1*	*17.3*	*123.6*
MOS calculations (sec)	*42.9*	*16.4*	*122.0*
LU decomp. (sec)	*1.17*	*0.3*	*0.4*
DC solution (sec)	*4.32*	*2.5*	*7.1*
Garbage collection (sec)	16.8	7.4	40.5
Time/iteration (sec)	**3.81**	**1.0**	**5.24**

TABLE 15

BIASlisp Simulation Results on VAX 11/780: sparse-matrix version using structures

There is a reduction in analysis time by use of sparse-matrix techniques but in spite of that the program has a poor speed performance. Comparing with Table 7 we see that the time for setup, LU decomposition, and DC solution is more than that required by SPICE, and even the sparse-matrix implementation does not give performance comparable to that of SPICE. Moreover, the time taken by this implementation is higher than the best time that had been achieved earlier in Table 11 by the use of some FORTRAN subroutines. Interface to a FORTRAN solver is not possible as arrays of floating-point numbers are not used to store the entries of the sparse nodal admittance matrix.

Description	Circuit 1	Circuit 2	Circuit 3
# iterations	14	23	26
Readin (sec)	2.26	0.95	4.2
Setup time (sec)	9.6	1.37	15.3
Matrix pointers (sec)	*2.2*	*0.3*	*4.1*
Reordering (sec)	*6.63*	*0.85*	*10.7*
LU fillins (sec)	*0.25*	*0.02*	*0.03*
Analysis (sec)	**57.0**	**23.52**	**143.7**
Matrix load (sec)	*47.1*	*17.8*	*129.8*
MOS calculations (sec)	*44.3*	*16.4*	*127.3*
LU decomp. (sec)	*1.78*	*0.4*	*0.43*
DC solution (sec)	*4.25*	*2.5*	*8.4*
Garbage collection (sec)	18.4	6.7	43.2
Time/iteration (sec)	**4.07**	**1.02**	**5.53**

TABLE 16

BIASlisp Simulation Results on VAX 11/780: sparse-matrix version using Property lists

3.5. BIASlisp on HP9836: full-matrix and sparse-matrix versions

The *Franz LISP* versions of BIASlisp were ported to the PSL [30] on the HP 9836 by the use of macros. Tables 17 and 18 give the simulation times for the three circuits using the full-matrix and sparse-matrix versions, respectively. The sparse-matrix version on the 9836 has been implemented only with structures since they are more efficient compared to property lists.

It is seen from Tables 17 and 18 that the *Readin* time is approximately three times that of BIASlisp on the VAX. The *Garbage collection* time is about five percent of the analysis time for large circuits. With a large physical memory this time can be reduced. The system on which these results were obtained uses 5.3 Megabytes of physical memory. Even if one ignores the time spent in garbage collection the time used for simulation is very large. A comparison of the time per iteration for each circuit on the VAX and the HP9836 is given below in Table 19. Also included is a ratio of the time/iteration for BIASlisp to the time/iteration of SPICE2

Description	Circuit 1	Circuit 2	Circuit 3
# iterations	21	23	43
Readin (sec)	10.9	5.2	14.8
Setup time (sec)	0.03	0.01	0.04
Analysis (sec)	**255.4**	**49.37**	**668.1**
Matrix load (sec)	*151.8*	*36.23*	*522.5*
MOS calculations (sec)	*143.7*	*34.8*	*510.3*
DC solution (sec)	*96.9*	*10.56*	*138.2*
Garbage collection (sec)	10.85	0.0	33.36
Time/iteration (sec)	**12.16**	**2.15**	**15.54**

TABLE 17

BIASlisp Simulation Results on the HP9836: full-matrix version

Description	Circuit 1	Circuit 2	Circuit 3
# iterations	24	23	43
Readin (sec)	10.9	5.1	14.7
Setup time (sec)	6.4	0.8	10.3
Analysis (sec)	**184.6**	**39.8**	**524.7**
Matrix load (sec)	*167.0*	*35.2*	*508.9*
MOS calculations (sec)	*162.5*	*34.6*	*506.7*
LU decomp. (sec)	*4.7*	*0.55*	*0.52*
DC solution (sec)	*6.3*	*1.2*	*7.92*
Garbage collection (sec)	11.2	0.0	33.9
Time/iteration (sec)	**7.7**	**1.73**	**12.2**

TABLE 18

BIASlisp Simulation Results on the HP9836: sparse-matrix version using structures

on the VAX 11/780. It is observed that the sparse-matrix implementation of BIASlisp is 16 to 25 times slower on the VAX 11/780 and 30 to 60 times slower on the HP9836 when compared to SPICE. The sparse-matrix implementation on the HP9836 doesn't give substantial speed improvement over the full-matrix version. This is due to the mapping of structures onto vectors instead of fast vectors*.

* PSL on the HP9836 has two types of vectors: an ordinary vector and a fast vector the latter being about six times faster.

Time/iteration Comparisons					
Description	Circuit	BIASlisp on VAX 11/780 (sec)	BIASlisp on on HP9836 (sec)	Ratio VAX/SPICE2	Ratio HP9836/SPICE2
Full-matrix version	Circuit 1	9.1	12.16	50.67	37.9
	Circuit 2	1.51	2.15	35.83	25.17
	Circuit 3	9.44	15.54	70.6	42.9
Sparse-matrix version	Circuit 1	3.81	7.7	32.1	15.88
	Circuit 2	1.0	1.73	28.8	16.7
	Circuit 3	5.24	12.2	55.5	23.82

TABLE 19

Comparison of Time/iteration for BIASlisp on the VAX11/780 and the HP9836

3.6. BIASlisp on Symbolics 3600:

Features of the Symbolics 3600 LISP machine are described in [2] where it has been pointed out that the 3600 has been built to execute large programs requiring high-speed symbolic and numeric computation. Table 20 and Table 21 give the run-times for the three circuits using the full-matrix and the sparse-matrix implementations, respectively with single-precision arithmetic. In Table 22 and Table 23 the results are given for double-precision arithmetic. The dialect of LISP on the 3600 is *Zetalisp.*

Description	Circuit 1*	Circuit 2	Circuit 3
# iterations		19	26
Readin (sec)		1.27	4.25
Setup time (sec)		0.01	0.01
Analysis (sec)		**1.69**	**11.11**
Matrix load (sec)		*0.73*	*6.78*
MOS calculations (sec)		*0.66*	*6.38*
DC solution (sec)		*0.26*	*2.51*
Garbage collection† (sec)		0.0	0.0
Time/iteration (sec)		**0.09**	**0.43**

TABLE 20

BIASlisp Simulation Results on the Symbolics 3600: full-Matrix version with single-precision arithmetic

Description	Circuit 1	Circuit 2	Circuit 3
# iterations	20	19	26
Readin (sec)	3.3	1.2	4.2
Setup time (sec)	0.45	0.07	0.68
Analysis (sec)	**4.75**	**1.89**	**8.16**
Matrix load (sec)	*2.85*	*0.65*	*6.03*
MOS calculations (sec)	*2.62*	*0.60*	*5.93*
LU decomp. (sec)	*0.14*	*0.1*	*0.01*
DC solution (sec)	*0.18*	*0.05*	*0.20*
Garbage collection† (sec)	0.0	0.0	0.0
Time/iteration (sec)	**0.23**	**0.1**	**0.31**

TABLE 21

BIASlisp Simulation Results on the Symbolics 3600: sparse-matrix version with single-precision arithmetic

* Results for Circuit 1 could not be obtained as the 3600 trapped due to single-precision underflow.

† Garbage collection was turned off on the 3600. This is an option on the 3600 and, therefore, the time has been given as 0.0 sec.

BIASlisp Simulation Results on the Symbolics 3600: full-Matrix version with double-precision arithmetic

Description	Circuit 1	Circuit 2	Circuit 3
# iterations	17	23	43
Readin (sec)	3.2	1.27	4.25
Setup time (sec)	0.01	0.01	0.01
Analysis (sec)	**28.04**	**6.57**	**87.75**
Matrix load (sec)	*14.92*	*4.24*	*63.7*
MOS calculations (sec)	*14.36*	*4.12*	*63.0*
DC solution (sec)	*10.78*	*1.33*	*17.6*
Garbage collection† (sec)	0.0	0.0	0.0
Time/iteration (sec)	**1.67**	**0.28**	**2.04**

TABLE 22

Description	Circuit 1	Circuit 2	Circuit 3
# iterations	19	23	43
Readin (sec)	3.3	1.2	4.2
Setup time (sec)	0.5	0.07	0.66
Analysis (sec)	**21.4**	**4.57**	**65.2**
Matrix load (sec)	*15.62*	*3.66*	*57.3*
MOS calculations (sec)	*14.4*	*3.51*	*54.2*
LU decomp. (sec)	*0.57*	*0.05*	*0.02*
DC solution (sec)	*1.15*	*0.22*	*2.1*
Garbage collection† (sec)	0.0	0.0	0.0
Time/iteration (sec)	**1.12**	**0.2**	**1.51**

TABLE 23

BIASlisp Simulation Results on the Symbolics 3600: sparse-matrix version with double-precision arithmetic

† Garbage collection was turned off on the 3600. This is an option on the 3600 and, therefore, the time has been given as 0.0 sec.

To get some idea about the simulation speed and to draw some conclusions the time per iteration comparision, of the sparse-matrix implementation of BIASlisp, with SPICE is given in Table 24. The numbers in parenthesis are the results obtained using single-precision arithmetic.

Circuit	SPICE 2G.6 on VAX 11/780 (sec)	BIASlisp on on 3600 (sec)	Ratio (3600/VAX)
Circuit 1	0.24	1.12 (0.23)	4.67 (0.96)
Circuit 2	0.06	0.20 (0.10)	3.30 (1.67)
Circuit 3	0.22	1.51 (0.31)	6.86 (1.41)

TABLE 24

Comparison of Time/iteration for SPICE 2G.6 on the VAX11/780 and BIASlisp on the 3600

It is seen that BIASlisp on the Symbolics 3600 gives reasonable performance. SPICE is definitely faster and this is due to the slower double-precision floating-point arithmetic on the 3600. A comparison between the times for some basic operations in LISP on the 3600 and FORTRAN on the VAX 11/780 with floating-point hardware is made in table 6. LISP on the 3600 is slower than FORTRAN on the VAX 11/780 for basic numeric operations +, -, *, and / due to the floating-point accelerator used by the VAX 11/780. The basic numeric operations +, -, *, and / are used very often in MOS related computations resulting in a poor performance of BIASlisp compared to SPICE2.

4. DISCUSSION

The performance of BIASlisp on the Symbolics 3600 LISP machine shows that LISP in itself is not a slow language. It is the lack of an appropriate machine architecture that results in its poor performance. A computer architecture which is suitable for conventional languages (FORTRAN, PASCAL, and C) does not give an optimum performance for a LISP-like language. Thus a comparison of the speed of LISP programs with programs written in other languages on traditional machine architectures is unfair. When execution speeds are compared with a LISP program on the LISP machine comparable performance results.

It is illuminating to examine the differences between a conventional programming language and LISP. The two salient features of LISP are its built-in garbage collector and the absence of type declarations for the fundamental LISP objects. These conveniences have their penalty unless something special is done about them and the advantages and disadvantages are considered below.

With a built-in garbage collection package a programmer need not be concerned with allocating and freeing memory space. This has an advantage in that bugs associated with dynamic memory management do not occur. The result is a safe and reliable storage system which is one less headache when debugging complex systems. However, garbage collection techniques can be expensive and time consuming and this inherently results in a poor over all performance. As has been seen earlier a certain amount of the total simulation time is spent in garbage collection on the VAX 11/780 and the HP9836. The 3600 provides one Gbyte of virtual memory address space and a hardware assisted system to reclaim unused sections of the memory [2]. Hardware features speed up the process and storage can be reclaimed with minimum overhead. Moreover, garbage collection on the 3600 is available as an option and can be turned on, whereby it is run when necessary in the background, or kept off.

The other unique feature of LISP is its run-time data-type checking for the primitive objects. No type declarations are required at compile time and are determined during run-time.

Different techniques can be used to store the type information.

Franz LISP uses a *bibop* memory allocator or a typed page scheme [31]. In this method each page contains only one type of data object and there is a table that keeps track of the type of object in each page. Thus data objects are allocated by pages and a whole page of objects is allocated even when only one object is required. As pointed out in [31] this scheme was chosen primarily due to the VAX architecture.

Another method, which is used in PSL and *Zetalisp*, is the typed pointer scheme. Pointers to each object have an associated tag to indicate the type of object being referenced. A certain number of bits have to be dedicated to the tag field. In PSL on the HP9836 the tag field is made up of 8-bits [32] and in *Zetalisp* on the 3600 the tag field has 6-bits [2].

Type checking in software can be very expensive as is evidenced by a comparison of the times taken for basic operations in Section 3 for the computers with conventional architecture. The time for a floating-point operation in FORTRAN is very small compared to that in LISP on the VAX and the difference can be attributed to the time required for type checking before each operation is performed. The LISP machine makes use of a tagged architecture [2], whose advantages are described in [33]. On the LISP machine run-time data checking is done in hardware. There is no loss in performance because the data-type checking is performed in parallel with the instruction.

These considerations then motivate the choice of a different hardware for efficient execution of a LISP-like language. Though the VAX and the HP9836 can run LISP their architecture cannot efficiently support LISP and it is only on the LISP machine that one gets the desired performance.

It would be interesting to compare the speed of a FORTRAN program on the LISP machine* to that on a VAX. A FORTRAN 77 environment has been provided on the Symbolics 3600 which enjoys many of the advantages associated with a LISP environment. The main

* At present a FORTRAN 77 compiler is not available on the Symbolics 3600 used for this study

advantages accompanying a LISP environment are:

a) Access to higher level data structures.

b) Interactive environment.

c) Powerful debugging facilities.

d) Philosophy of incremental compiling. Old modules may be modified and new ones can be added with immediate effect. Functions which have been satisfactorily checked can be compiled and used with new functions.

e) Dynamic linking and loading.

FORTRAN 77 on the Symbolics 3600 shares the advantages from (b) to (e). However, LISP definitely has an edge over it because of the availability of higher level data structures. Much more powerful and complex systems can be designed using LISP whereas that may not be the case with FORTRAN.

Being a very powerful language LISP is very well suited for software development. Incremental compiling along with dynamic linking and loading and the LISP debugging aids provide an extraordinary environment for program development. Consequently less time is required to debug complex programs and software development time is significantly reduced.

As far as circuit simulation is concerned BIASlisp offers a LISP environment in which new device models can be evaluated. Only the functions involving model computation need be interpretive with the rest of the code being compiled. Further, if the input language to the circuit simulator is embedded in LISP it will be very easy to define new models or make enhancements. LISP expressions would then be acceptable inputs. The user can also add his own functions and augment the capabilities of the program.

An example of the input line for a MOSFET device in BIASlisp with an embedded input format would be (*mosfet 'm1 1 2 3 3 'modn 'lc 3e-6 'wc 12e-6*). Though this format retains the flavor of a SPICE-like input, it requires parenthesis and quotes and is not very friendly from a circuit designer's point of view. The user has to be familiar with the features of LISP to be

able to interact with a circuit simulation program having such an input. It was for this reason that a SPICE-like input format was chosen for the BIASlisp program. If this program is to be used with existing LISP-based LSI design tools or expert systems there may be no controversy about the input format. The circuit simulator will derive the circuit information from the data structures set up by the other programs.

Before ending this section it must be stressed that the 3600 provides an excellent environment for software development. Compiled code can be used as compilation is fast and debuging is effective even with the compiled code. This is not true for the HP9836 where compiled LISP functions are difficult to debug [32]. The novel architecture of the Symbolics 3600 does make it an efficient system for executing LISP programs without loss of performance. With a floating-point accelerator the 3600 can achieve simulation speeds comparable to that of a VAX 11/780 with floating-point hardware.

5. CONCLUSIONS

The feasability of circuit simulation using the LISP language is investigated on three different computer systems: VAX 11/780, HP9836, and the Symbolics 3600 LISP machine. It is found that the circuit simulation program runs effectively only on the Symbolics 3600 LISP machine. Systems such as the VAX 11/780 or the HP9836 are not designed to support efficiently the LISP language.

There is no inherent shortcoming in the language itself which results in a poor performance, it is the lack of an appropriate hardware. As a language LISP is very powerful and the LISP environment significantly aids program development. This may be one of the main reasons for adopting LISP as a language for VLSI-CAD tools.

In the future, it needs to be seen how powerful a LISP-based microprocessor [34-35] or a LISP-RISC [36] system would be for running circuit simulation programs in LISP.

APPENDIX A

BIASlisp User's Guide

1. Introduction

BIASlisp is an interactive MOS circuit simulation program for DC and time domain transient analyses. Circuits may contain linear resistors, linear capacitors, independent voltage and current sources, diodes, and MOSFETs. There are built-in models for the semiconductor devices, and only the relevant model parameters have to be specified by the user. The diode is modeled by an ideal diode equation and the MOSFETs by the Shichman-Hodges MOS model.

The program has been written in FRANZ Lisp and can be ported to other dialects of LISP. Currently it can be executed on the VAX 11/780, the HP9836 desktop computer, and the Symbolics 3600 LISP machine. To begin execution the user must be in a LISP environment and the program can be invoked by typing (*biasl*).

2. Input Format

The input format is similar to that of SPICE2 with additional commands to make it interactive. A prompt '*BIASlisp ->*' is displayed after which an input can be typed. Fields on an input line can be separated by one or more blanks. No other delimiters are acceptable.

A number field has the general form x.xxxxxxe±yy. Scale factors of SPICE are not allowed. Only scientific notation is accepted.

3. Circuit Description

The circuit description can be entered in a manner similar to that of SPICE2. The default mode is the input mode. Input description can be an element specifying the circuit topology or a control command. The control commands always start with a *period*. Circuit element description lines contain the element name, the circuit nodes to which it is connected, and the values of the parameters that determine the electrical characteristics of that element. A resistor name must begin with the letter *r*, capacitor name with the letter *c*, voltage source name with a *v*, current source with letter *i*, diode with the letter *d*, and a mosfet with the letter *m*. The name string can be as long as desired.

Nodes can be integers or alphanumeric names. The datum node is numbered zero or *gnd*. The circuit should not contain a loop of voltage sources and a cutset of current sources and/or capacitors. A singular matrix results if the above condition is not satisfied. Every node must have at least two connections and an error is flagged if this condition is violated.

Comment lines begin with a '*'. Anything following the '*' is ignored and printed out as it is.

Data fields that are enclosed in '< >' in the description below are optional.

3.1. Element Descriptions

3.1.1. Resistors

General form:

rxxxx n1 n2 value

n1 and n2 are the two element nodes. *value* is the resistance value (in ohms) and may be positive or negative but not zero.

Examples:

rbias 1 2 1e3

rpullup vdd vout 100

3.1.2. Capacitors

General form:

cxxxx n+ n- value <ic initial condition>

n+ and n- are the positive and negative element nodes, respectively. *value* is the capacitance in Farads.

The optional initial condition is the time-zero value of the capacitor voltage, in volts. *Note that the initial conditions apply only if the 'uic' option is specified on the 'tran' command line.*

Examples:

ccomp 10 11 1e-6

cout vout gnd 2e-12 ic 0

3.1.3. Independent sources

General form:

vxxxx n+ n- value/function

ixxxx n+ n- value/function

n+ and n- are the positive and negative nodes, respectively. The associated reference directions are used. The voltage source value is positive when n+ is at a higher voltage than n-. Current source value is positive with current flowing from n+ to n-.

There are four independent source functions: DC, pulse, sinusoidal, and piece-wise linear. A description is given below.

DC: dcvalue

If no source value/function is given in the input data it is assumed to be a dc source of value zero.

Pulse: pulse v1 v2 td tr tf pw per

Parameters	Description	Units
v1	initial value	volts or amps
v2	pulsed value	volts or amps
td	delay time	seconds
tr	rise time	seconds
tf	fall time	seconds
pw	pulse width	seconds
per	period	seconds

Description of pulse parameters

Sinusoidal: sin vo va freq td theta

Parameter	Description	Units
vo	offset	volts or amps
va	amplitude	volts or amps
freq	frequency	Hz
td	delay	seconds
theta	damping factor	1/seconds

Description of sin parameters

Piece-Wise Linear: pwl t1 v1 t2 v2 ... tn vn

Each pair of values (ti, vi) specifies that the value of the source is vi (amps or volts) at time ti. ti's should satisfy the inequality t1 < t2 < t3 ... < tn. The value of the source at intermediate values of time is determined by using linear interpolation on the input values.

Examples:

vcc 5 0 5

isrc gnd 10 pulse -1 1 2e-9 2e-9 2e-9 50e-9 100e-9

vin vin gnd sin 0 1 100e6 1e-9 1e10

vclock clkin 0 pwl 0 -5 10e-9 -5 15e-9 5 20e-9 5

3.1.4. Junction Diodes

General form:

dxxxx n+ n- mname <area >

n+ and n- are the positive and negative nodes, respectively. mname is the model name, area is the area factor. If the area factor is omitted, a value of unity is assumed.

Example:

dclamp 2 3 dmod1 2.0

3.1.5. MOSFETs

General form:

mxxxx nd ng ns nb mname <lc val> <wc val> <ad val> <as val>

nd, ng, ns, and nb are the drain, gate, source, and bulk (substrate) nodes, respectively. mname is the model name. lc and wc are the channel length and channel width in meters. ad and as are the areas of the source and drain diffusions, in square meters. If any of lc, wc, ad, or as are not specified the default values are used. lc and wc default to 1e-6 and ad and as to unity.

Examples:

mpullup vdd vout vout gnd mdep

mpulldn vout vin gnd gnd menh

mpass 2 4 10 gnd menh lc 10e-6 wc 30e-6

3.2. Model Descriptions

The semiconductor device models are specified using the '.model' command. A set of model parameters are specified that can be used by one or more devices.

General form:

.model mname type pname1 pval1 pname2 pval2 ...

mname is the model name, and type is either diode, nmos, or pmos. Parameter values are defined by appending the parameter name, as given below for each model type, followed by one or more blanks and the parameter value. Model parameters that are not given a value are assigned the default values given below.

Examples:

.model mdep nmos vto -2.0 kp 35e-6 gamma .2

.model dmod1 diode is 1e-15

3.2.1. Diode Model

The diode model parameters are given below.

name	parameter	units	default
is	saturation current	amperes	1.0e-14

Diode Model Parameters

3.2.2. MOSFET Model

The parameters for the MOS model are given below.

name	parameter	units	default
vto	zero-bias threshold voltage	Volts	1.0
kp	intrinsic transconductance parameter	A/V**2	2e-6
gamma	bulk threshold parameter	V**0.5	0.0
phi	surface potential	volts	0.6
lambda	channel-length modulation	1/V	0.0
js	bulk junction saturation current density	A/m**2	1e-14

MOS Model Parameters

3.3. Control Commands

These commands control the program execution and provide an user friendly interface.

3.3.1. List

General form:

.ls xxx

xxx is the type of circuit information that has to be listed. It is *r* for a resistor, *c* for a capacitor, *v* for an independent voltage source, *i* for an independent current source, *mo* for device models, *op* to list analysis options, and *ckt* to list all circuit elements.

Examples:

.ls r

lists all resistors in the circuit.

.ls mo

lists all device models in the circuit.

3.3.2. Change

General form:

.ch name1 param1 value1 name2 param2 value2 ...

name is the element name or model name whose parameters have to be modified. *param* is the keyword for the parameter that has to be modified and *value* is its new value. The keywords for model parameters are the same as their names given earlier. For nodes: n1 and n2 refer to n+ and n- of a two node element and n1, n2, n3, and n4 are the drain, gate, source, and bulk nodes of a mosfet, respectively. The keyword *value* changes the value of a resistor or a capacitor.

Examples:

.ch mod1 vto 0.7

changes vto to 0.7 volts for model mod1

.ch r1 n1 3 m1 n3 0 mod4 kp 1e-6

changes *n+* to 3 for resistor r1, *ns* to 0 for mosfet m1 and *kp* to 1e-6 for mosfet model mod4.

.ch ccomp value 30e-12

changes the value of capacitor ccomp to 30e-12.

3.3.3. Delete

General form:

.del name1 name2 ...

name1, name2, ... are deleted from the circuit description.

Example:

.del r1 ccomp m1 mod2

resistor r1, capacitor ccomp, mosfet m1, and model mod2 are deleted from the circuit description.

3.3.4. Read

General form:

.read file-name

The circuit description is read in from the file file-name instead of the regular terminal input.

Example:

.read circuit1

3.3.5. Write

General form:

.write file-name

The circuit description is written into the file file-name. Analysis options and the nodes set to help convergence are also stored. This allows the user to store modified circuit descriptions.

Example:

.write circuit1

3.3.6. New

General form:

.new

This command initializes the program to read in a new circuit description.

3.3.7. Nodeset

General form:

.set node1 v1 node2 v2 ...

This command helps the program find the dc solution by specifying an initial value for voltages at node1, node2 ... Under normal circumstances this may not be required but may be necessary for convergence of circuits with strong feedback.

Example:

.set vout 0.0 12 5.0

initializes nodes *vout* and 12 to 0 and 5 volts respectively.

3.3.8. Options

General form:

.op opt1 val1 opt2 val2 ...

This command allows the user to reset program control for specific simulation purposes. Any combination of the following options may be included in any order.

Option	Description	Default
gmin	minimum conductance allowed by the program	1e-12
reltol	relative error tolerance of the program	1e-3
abstol	absolute current error tolerance of the program	1e-12 A
vntol	absolute voltage error tolerance of the program	1e-6 V
maxiter	dc iteration limit	100
dcit	dc transfer curve iteration limit	50
loitl	lower transient analysis iteration limit	4
upitl	upper transient analysis iteration limit	10

Description of Options

Example:

.op gmin 1e-8 reltol .0001 vntol 1e-4 maxiter 50

resets the values of gmin, reltol, vntol, and maxiter.

3.3.9. Bias point solution

General form:

.bias

This command is used to calculate the dc operating point of the circuit with the capacitors replaced by an open circuit. An operating point analysis is automatically performed prior to a transient analysis, to determine the transient initial conditions , if *uic* is not specified on the '.tran' command.

If no analyses are requested then a dc operating point analysis is performed.

3.3.10. DC transfer characteristics

General form:

.dc src-name VSTART VSTOP VINCR

The '.dc' command defines the dc transfer curve source and sweep limits. src-name is the name of an independent voltage or current source. VSTART, VSTOP, and VINCR are the starting, final, and incrementing values respectively.

Examples:

.dc vin 0.25 5.0 0.25

.dc vds 0 10 1

3.3.11. Transient time domain analysis

General form:

.tran TSTEP TSTOP <TSTART > <uic>

TSTEP is the suggested computing increment, TSTOP is the final time, and TSTART is the initial time. If TSTART is omitted it is assumed to be zero. The maximum step size chosen by BIASlisp is the minimum of TSTEP and TSTOP/50.0.

uic (*u*se *i*nitial *c*onditions) is an optional keyword that indicates that the user does not want the program to solve for the dc operating point before beginning the transient analysis. If this keyword is specified then the values specified using *ic* on the element cards are used for the initial transient condition.

Examples:

.tran 1e-9 100e-9 0

.tran 10e-9 1e-6 uic

3.3.12. Print Out

General form:

.print print-type type name1 type name2 ...

This command asks for a tabular listing of one to six output variables. print-type is the analysis type (bias, dc, tran) for which the results are desired. The output variables can be a node voltage or a current through a voltage source or a mosfet device. Type can be either a *v* or a *i* indicating a voltage or current variable. Names are the node names, voltage source names, or mosfet names.

Examples:

.print bias v 1 v vout

Prints the value of dc operating point voltages at nodes 1 and *vout*.

.print dc v 3 i vin i m1 i m3

Prints the value of dc transfer analysis for voltage at node 3 and currents through the voltage source *vin* and mosfets *m1* and *m3*.

3.3.13. Start analysis

General form:

.end

This command tells the program that all circuit and analysis information has been entered. The analysis is then initiated and the desired output variables are printed. Note that

no analysis will be performed until a '.end' command is entered.

Once analysis is complete control returns to the user who can edit the circuit description and perform more analyses. If no further analysis is required '.exit' command is used to get out of the program.

3.3.14. Exiting from the program

General form:

.exit

When the user wants to quit from BIASlisp he uses this command which gets him back to the LISP environment in which the program was being executed.

APPENDIX B

A description is given for all the lists that are used to store input data in BIASlisp.

List name	Description
Res-list	stores the names of all resistors
Cap-list	stores the names of all capacitors
Vsrc-list	stores the names of all voltage sources
Isrc-list	stores the names of all current sources
Diode-list	stores the names of all diode devices
Mos-list	stores the names of all MOS devices
Nodenames	stores the node names
ModDef	stores the names of Models defined using '.model' (useful for error check)

APPENDIX C

A description of the property list for each element type is given below.

Element type	Property name	Description
Resistor	n1	positive node
	n2	negative node
	value	resistance value in Ohms
Capacitor	n1	positive node
	n2	negative node
	value	capacitance value in Farads
	ic	initial condition
Voltage/ Current source	n1	positive node
	n2	negative node
	value	dc value
	function	source function (pulse, sin, pwl)
	param	function parameters
Diode device	n1	positive node
	n2	negative node
	modnam	model name
	area	area factor
MOS device	n1	drain node
	n2	gate node
	n3	source node
	n4	bulk node
	modnam	model name
	lc	channel length
	wc	channel width
	ad	drain diffusion area
	as	source diffusion area

A description of the property list for each model type is given below.

Model type	Property name	Description
Diode	is	saturation current
MOSFET	mtype	model type (nmos/pmos)
	vto	threshold voltage (vbs = 0)
	kp	intrinsic transconductance parameter
	lambda	channel length modulation parameter
	gamma	bulk threshold parameter
	phi	surface potential
	js	bulk junction saturation current density

REFERENCES

[1] W. B. Gevarter, "Expert Systems: limited but powerful", *IEEE Spectrum*, Vol. 20, No. 8, pp. 39-45, Aug. 1983.

[2] C. W. Roads, *3600 Technical Summary*, Symbolics Inc., 1983.

[3] D. R. Stoutemyer, "LISP Based Symbolic Math Systems", *Byte*, Vol. 4, No. 8, pp. 176-192, Aug. 1979.

[4] J. Batali and A. Hartheimer, "The Design Procedure Language Manual", *AI Memo. No. 598*, Artificial Intelligence Laboratory, M. I. T., Sept. 1980.

[5] B. Barrett, A. Kuchinsky, B. Rogers, M. Sorens, "Integrated Circuit Procedural Language: External Reference Specification, vs. 1.6", *HP Internal Memo*, Mar. 1984.

[6] J. T. Deutsch and A. R. Newton, "Data-Flow Based Behavioral-Level Simulation and Synthesis",*Digest of Technical Papers ICCAD-83*, Santa Clara, California, pp. 63-64, Sept. 1983.

[7] D. B. Kasle, *Private Communication.*

[8] D. Weinreb and D. Moon, *LISP Machine Manual*, Symbolics Inc., 1981.

[9] E. Cohen, "Program Reference For SPICE2", *Memo. No. ERL-M592*, Electronics Research Laboratory, University of California, Berkeley, June 1976.

[10] R. Gyurcsik, D. Cheung, F. Ma, and T. Yee, "BIAS Report", *Memo. No. UCB/ERL M82/89*, Electronics Research Laboratory, University of California, Berkeley, Dec. 1982.

[11] D. Cheung, "BIASB.36: A MOS Integrated Circuits Simulation Program for use on Desk-top Computers", *Master's Report*, University of California, Berkeley, May 1983.

[12] T. Yee, "BIASP: An Integrated Circuit Simulation Program for use on Microcomputers", *Master's Report*, University of California, Berkeley, May 1984.

[13] B. del Signore, "MOSFET Transistor Modeling for BIASB Integrated Circuits Simulation Program", *Master's Report*, University of California, Berkeley, Sept. 1983.

[14] F. Ma, "Bipolar Transistor Model for the Desktop Circuit Simulators: BIAS-B and BIAS-P", *Master's Report*, University of California, Berkeley, Dec. 1983.

[15] R. S. Gyurcsik, K. Mayaram, T. Yee, F. Ma, and D. O. Pederson, "Language Comparison for Circuit Simulation on Desktop Computers", *Proceedings ISCAS 1984*, Montreal, Canada, May 1984.

[16] H. Shichman and D. A. Hodges, "Modeling and Simulation of Insulated- Gate Field-Effect Transistor Switching Circuits", *IEEE J. Solid- State Circuits*, vol. SC-3, pp. 285-289, Sept. 1968.

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

[18] C. W. Ho., A. E. Ruehli, and P. A. Brennan, "The Modified Nodal Approach to Network Analysis", *IEEE Trans. Circuits and Systems* , vol. CAS-22, pp. 504-509, June 1975.

[19] L. O. Chua and P. M. Lin *Computer-Aided Analysis of Electronic Circuits: Algorithms and Computational Techniques*, Prentice Hall Inc., New Jersey, 1975.

[20] W. J. McCalla, *Computer-Aided Circuit Simulation Techniques*, Pre-publication Manuscript.

[21] J. Vlach and K. Singhal, *Computer Methods for Circuit Analysis and Design*, Van Nostrand Reinhold Company, 1983.

[22] P. H. Winston and B. K. P. Horn, *LISP*, Addison-Wesley Publishing Company, 1981.

[23] R. Wilensky, *LISPcraft*, Computer Science Division, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, 1983.

[24] A. L. Sangiovanni-Vincentelli, "Circuit Simulation", in *Computer Design Aids for VLSI Circuits*, P. Antognetti, D. O. Pederson, and H. De Man, Eds. Groningen, The Netherlands: Sijthoff and Noordhoff, pp. 19-113, 1981.

[25] P. R. Gray, *Class Notes EECS240*, Fall 1983.

[26] P. R. Gray and R. G. Meyer, *Analysis and Design of Analog Integrated Circuits*, John Wiley and Sons, 1984.

[27] D. A. Hodges, *Class Notes EECS241*, Spring 1984.

[28] A. Vladimirescu, K. Zhang, A. R. Newton, D. O. Pederson, and A. Sangiovanni-Vincentelli, *SPICE Version 2G User's Guide*, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, California, 1981.

[29] J. K. Foderaro and K. L. Sklower, *The Franz LISP Manual*, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, 1983.

[30] M. L. Griss, T. Larrabee, G. Osnos, A. Snyder, *HP9836 PSL Users Guide*, 1983.

[31] J. K. Foderaro, *The Franz LISP System (draft)*, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley.

[32] B. Barrett, "Prism Performance Guidelines", *HP Internal Memo.*, Feb. 1984.

[33] E. A. Feustel, "On The Advantages of Tagged Architecture", *IEEE Trans. Computers*, vol C-22, No. 7, pp. 644-656, July 1973.

[34] G. L. Steele Jr. and G. J. Sussman, "Design of a LISP-Based Microprocessor", *Comm. ACM*, vol. 23, No. 11, pp. 628-645, Nov. 1980.

[35] G. J. Sussman J. Holloway, G. L. Steele Jr., and A. Bell, "SCHEME-79 -- Lisp on a Chip", *IEEE Computer*, vol. 14, No. 7, pp. 10-21, July 1981.

[36] C. Ponder, "... but will RISC run LISP??", *Report No. UCB/CSD 83/122*, Computer Science Division, University of California, Berkeley, California, Aug. 1983.