A UNIFIED HARDWARE DESCRIPTION LANGUAGE FOR CAD PROGRAMS

by

J. D. Crawford

Memorandum No. UCB/ERL M79/64

17 August 1979

A UNIFIED HARDWARE DESCRIPTION LANGUAGE

FOR CAD PROGRAMS

by

J. D. Crawford

Memorandum No. UCB/ERL M79/64

17 August 1979

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

INTRODUCTION

The advent of LSI and VLSI circuits has generated a great deal of interest in and emphasis on the use of computer-based IC design aids. Many efforts are underway to develop large integrated design aid systems. These systems typically require the use of many programs and one of the major considerations in development is the communication between the programs.

The programs that are used in IC design aid systems include various translation programs that facilitate mobility among levels of a hierarchical circuit design. These levels range all the way from specifications for individual transistors to register diagram descriptions of large systems. Simulation programs are available at many of these levels and take as input a description of the circuit at that level. The mixed-mode simulator simulates a circuit described at several levels in the design tree. Many of these programs have been in use for some time and have developed the reliability associated with software that has been used extensively. It is desirable, therefore, to consider the use of existing programs in a new integrated design aid system.

An efficient user environment is one in which the designer can enter his circuit or system into the computer once. He can then edit the design and use available programs to verify design and to perform those functions most easily effected by a computer. One method of describing a circuit involves graphics input. Typical forms of graphic input are stick diagrams, logic

diagrams, or actual layout rectangles and polygons. Another input method uses a textual description language containing interconnectivity information, circuit elements and parameters associated with the elements. Most modern simulation programs include translators of varying sophistication for some textual circuit description. Thus, in the past, a CAD group supporting more than one program has had to require the user to learn more than one language in which to describe his or her circuits. Each circuit to be described has had to be translated by hand if more than one program is to be used.

One proposed solution to this problem is the "cross-bar switch" approach [1]. A parser-generator is used to create translators that map each input language to every other. This approach allows the user to use a language to which he is accustomed and to translate his circuit descriptions under program control for use with other simulators. Each circuit description is translated twice. One translation is from the user source to the program source. The other is from the program source into the internal circuit description. N languages and M programs require N*M translators. The scheme is detailed in Figure 1.

A second approach [2] uses at least one hi-level language and an intermediate language. A translator exists from each hi-level language to the intermediate language. Another translator is used to translate from the intermediate language into the input language of each design-aid program. Among the advantages of this scheme are that only N+M translators are

required for N languages and M programs and that the design-aid programs need not be altered for use in the scheme. However, three separate translations are required each time a program is used.

A third approach to the problem is to use a single, powerful description language coupled with information about each "target" design aid program [3]. It is this approach which is presented here. The general scheme is shown in Figure 2.

The unification of the textual input descriptions of circuits has two major advantages. The first is that the circuits described are readable not only by any of the programs used at a facility, but also by all users and are therefore appropriate for documentation. The second is that in the development of new simulation programs very little attention must be devoted to input translation. This last feature arises since the unified description language is implemented in a single stand-alone translator which generates a fully expanded intermediate language (analagous to the intermediate language generated by a typical compiler). This intermediate language is very simple to read, allowing the translator associated with each simulator to be of a very simple nature. Thus there is only one translation program to maintain. Development of new simulation tools is expedited since the program designer can focus the vast majority of his attention on the program algorithms and their implementation.

This report is divided into three parts. The first part consists of a description of the language and the goals of implementation. The second part presents a detailed description of the implementation with reference to an example use with the timing simulator MOTIS-C [4]. The third part briefly describes the performance characteristics of the program. Possible extensions and enhancements to both the language and the implementation are discussed in the conclusion.

LANGUAGE GOALS

A language capable of performing the functions herein described must be both flexible and attractive to the user. A new user should be able to learn the language quickly; yet there must be the power and flexibility required by the more experienced user. The second major requirement is the flexibility to describe circuits adequately at all levels of structure, i.e., from the register diagram level and higher down to the transistor level.

It was decided that since there has been a great deal of experience with the SPICE2 input language, the unified language would use the SPICE2 language as a basis. The large body of experience with SPICE2 has both shown the strengths of its input language and clearly defined areas in need of enhancement. Features may be added based on demonstrated need rather than speculation. Therefore, the use of the SPICE2 language as a base facilitates meeting the first requirement stated above.

Some of the enhancements that users have requested over the years are the ability to:

1) use signal-path names to describe nodes,

2) pass parameters to subcircuits,

3) access nodes internal to subcircuits for output,

4) use arithmetic expressions and functions

The parameter passing capability combined with the function evaluation provides a useful mechanism for translation among levels in the design tree. Thus the subcircuit definition facility, suitably revised, simplifies satisfaction of the second requirement.

A detailed description of the SPICE2 input language is available in [5] and is not given here. The formal specification of the unified input language appears in APPENDIX C. The language is divided into two parts. The first, a structural hardware description language, is described below. The second, an analysis and output processing control language, is covered later in brief.

The description language is based entirely on two familiar concepts. The first is the circuit element which has some number of signal paths associated with it (The terms node and signal path are used interchangeably). Each element is of a certain type and refers to a model of that type. The second is the model of a class of circuit elements. There are no inherent restrictions on the magnitude of the block being modeled and thus a model may be of a transistor, a logic gate, an ALU, or an entire computer.

Some preliminary concepts must be clarified before defining what is meant by the term model. Everything that is eventually input to a simulator must be in terms of the "low-level types" that it recognizes. A typical simulation program, i.e., the analysis portion of a simulation program, is capable of recognizing (and analyzing) certain "low-level types." An example of a low-level type for a circuit simulator might be a resistor of resistance R with two temperature coefficients TC1 and TC2. A more complex example is the MOS transistor with thirty parameters. A low-level type for a logic simulator might be an N-input NOR gate with a rise time TR and a fall time TF. N might be limited to values from N=1 (an inverter) to N=8 (an 8-input NOR gate)

Two features are common to all low-level types. First there is some inherent interconnectivity information. A resistor is usually assumed to be a two-terminal device. In many practical cases, the terminals are interchangeable. The NOR gate has a variable number of terminals, but they are not interchangeable. For example, the output node has different properties than one of the inputs. Secondly, parameters are associated with each type. These are nominally given a name and a default value. Furthermore there may be a class of values that are considered unreasonable for a particular parameter.

Circuit and hardware description languages designed for use with a particular program contain a hierarchy which is ultimately based on the low-level types recognized by the program. A rough analogy may be drawn to a high-level programming language and the

machine language of a particular computer. An approach to the resolution of the incompatability of the target programs is presented in the section on tailoring.

A circuit description language consisting only of elements can thus be envisioned. Each element would have a low-level type name, interconnectivity information, and parameter values associated with it. A description of this nature would be very lengthy and would be tailored entirely to one particular simulator. The concept of the model is introduced in order to help alleviate these two problems.

A model may be used in one of two ways. First, there may be several components of a circuit of the same low-level type. Most or all of the parameters may be the same for some subset of these components. A model is then defined for this type, specifying a model name and values for some or all of the parameters associated with the type. Elements may then refer to this model name instead of to the low-level type. Only parameter values differing for this element need be specified. The effect of this type of model is to provide a mechanism by which the user can subdivide low-level types and assign his own names to the resulting models. This is achieved by providing a set of parameter values different from the default values specified by the program. The model of this type is input as shown:

model <mdnam> <mdtyp> (<paramlist>)

where

<paramlist> ::= [<param> = <value>] | [<value>]

Example:

```
model mose nmos (vto=0.7 kp=20u phi=0.7 gamma=0.6 lambda=0.0)
```

or

```
model mose nmos (0.7 20u 0.7 0.6 0.0)
```

An example use of this type of model is a circuit using NMOS enhancement and depletion devices. Two models, MOSE and MOSD are defined which specify all process parameters. Geometrical parameters are specified with each element description. The model name on the element description immediately tells the reader whether the device was enhancement or depletion.

The structures of which circuits are composed may be built from lower-level components. In the example just given, the logic NOR gate may be built out of MOS transistors. Likewise a storage cell may be built from the NOR gates and some other parts. Several storage cells may form a register. Several registers may form a bank.

The second major role of a model is to permit the user to design an arbitrary structure from the available low-level types. The model definition consists of elements and other model definitions which are local to the model being defined. This imparts a block structure to the model definition scheme similar to that found in ALGOL-like programming languages [6]. The first line of a model definition of this type (type subcircuit) provides a model name and other information to be used within the definition. This information consists of (formal) signal path

names, (formal) parameter names, and default parameter values local to the definition. Thus on a particular invocation of the model, i.e., on an element description line, actual interconnectivity information and actual parameter values are passed.

These two general forms of model permit convenient mapping among levels of simulation as well as providing a powerful means of describing an arbitrarily complex circuit.

Once the two types of model described above are understood, the concept of the general circuit element is immediately apparent. The element description consists of an element name, interconnection information, a model name and parameter information. Parameters specified with the element description override the corresponding parameters on the model. Thus, for a particular element description, the nature of the model to which it refers is completely transparent! It may be a low-level type specific to a particular simulator. It may be either of the two forms of models described above, or it may reside on a system library. The form of the element description line is:

<elnam> <nodelist> <mdnam> (<paramlist>)

where

<nodelist> ::= [node]

<paramlist> ::= [<param> = <value>] | [<value>]

Example:

alu1 s2 s1 s0 databus<15:12> alu (tsettle=40n2)

```
m1 vout vin ground bulk mose (10u 5u)
```

A sample circuit and its description is shown in Figures 3, 4, and 5.

IMPLEMENTATION--GENERAL CONCEPTS

A large subset of the language is implemented at U. C. Berkeley in a stand-alone program called BLT (Berkeley Language Translator). A second implementation called TEKSIM exists at Tektronix Inc. [7]. Only BLT is described here. BLT takes a circuit description in the unified hardware description language as input and outputs a circuit description in intermediate form. Two program design criteria are portability and the ability to run on a minicomputer. Due to address space limitations associated with some minicomputers, disk I/O has been used extensively. Nominally, the implementation is designed to operate as follows: A circuit description is read in. It is parsed and translated into a fully expanded form consisting of models of low-level types and elements referring to these models. All parameter values are assigned. This information is written to two disk files, a model file and an element file (see Figure 6). A simulation program may then read the model file, set up the models internally, and read the element file.

One useful environment for BLT uses a main control program which is given a circuit description and the name of the program (MOTIS-C, SPLICE, etc.) to be run. The control program calls BLT to translate the circuit description using the appropriate initialization file (see the section on tailoring below). The control program then interprets analysis and output commands (using BLT routines) to control analysis and output (see Figure 7).

TAILORING

One of the fundamental considerations in the implementation of this type of unified language is the fact that nearly every design-aid program, even within a given class, recognizes different low-level types.

The differences are often transparent at higher language levels. For example, as mentioned above, one can describe a logic gate (low-level type for a logic simulator) in terms of transistors. Thus a NOR gate element may refer directly to a low-level type or to a subcircuit model which defines that type in terms of other low-level types. The model may be local to the description or may reside in a library.

The unified-language translator must be able to provide information that is immediately useful to a particular target simulator. One proposed solution [8] takes whatever information is specified by the user and places it in a data base. Each simulator then searches for and uses whatever subset of the available information it can recognize. Thus, for example, the circuit simulator above would have to provide, or somehow have available, subcircuits for logic gates built from transistors. A circuit simulator that recognizes one parameter and another which recognizes a related parameter instead have no means of communication with the user. For example, if a simulator of bipolar transistors recognizes TAUF and the user specifies FT (a parameter that is mathematically related to TAUF), the error cannot be recognized until much later. These types of problems

are better resolved at input-translation time.

Another proposed solution [3] specifies that each design-aid program be provided with exactly that subset of all possible descriptions which it can make use of. Thus, the output of the input-translator should generate a file consisting of a circuit description based on the low-level types and parameters for those types that each program recognizes. When the author of a design-aid program wishes to interface his/her product to the translator he creates a description of his program. This description lists the names of the low-level types his program handles. The names, default values, and order of the parameters associated with each type, and the number of nodes for each type are specified with the type. A special, one-time run of the tranlator generates an "initialization" file from this list for the target program (see APPENDIX B and Figure 8 for instructions and an example respectively). When a user wishes to simulate a circuit, he provides the translator with the circuit description and the name of the simulator to use. The translator then reads the appropriate initialization file and does all appropriate error-checking. The user is informed as to what information is still needed to run a particular design-aid program.

If the logic-gate/transistor example mentioned earlier is examined for a moment, the following points are observed: First, a description consisting of logic gates can be modified to run on a circuit simulator very simply. A subcircuit is provided for each general gate type built from transistors. The parameters for the logic description, typically delay times, are passed into

the subcircuit and can be used, for instance, in conjunction with the function evaluation capability to generate aspect ratios. The user has the option of maintaining a library of such subcircuits.

If the reverse situation is considered, the difficulties are more severe. Suppose a description exists consisting entirely of transistors (say extracted directly from the IC artwork). A separate program must be used to extract logic gates from the transistor description. If this gate extraction program uses the intermediate language for the circuit simulator, a logic description can be generated in the unified input language and the designer is left with the simple case described above. The use of subcircuit models in the file or on a library is all that is then needed.

The approach of tailoring the intermediate file to each "target" program addresses three important issues. First, it minimizes the effort required for a design-aid program to read a circuit description. Second, useful feedback is provided to the designer (or possibly to another program) regarding what information has not yet been provided for a complete characterization of the circuit. Third, since the tailoring concept is quite general, the types of models and parameters are not limited in any way. A model could be of a rectangle or a polygon. The parameters would then be co-ordinates. A model might be of an entire IC package and the spec-sheet information could be specified in the parameter list. The important thing to note is that the interface is not limited to use with

simulators. Example programs which could interface to the intermediate language are translators which go up the design tree using pattern matching techniques to locate higher level structures. Placement and routing programs or other utilities could be interfaced. If this concept is extrapolated somewhat, the problem of interfacing a number of programs to form an integrated IC design system could be substantially reduced by interfacing to a simple intermediate language.

There are some other features incorporated into the tailoring scheme. First, global nodes may be specified. These nodes are electrically common. Typical global nodes might have the names GROUND or VCC. They may then be referenced within a subcircuit model without being formally declared on the subcircuit model definition line.

The second feature is the ability to specify key first words for special statements for the target program. For example, MOTIS-C requires a statment of the form:

VPLUS=<value>

The inclusion of VPLUS as a keyword in the initialization file enables a small routine to be inserted which uses BLT utilities and translates the line according to a specified format.

IMPLEMENTATION--PROGRAM DETAILS AND DATA STRUCTURES

The implementation of the language is described in detail in this section. The subset of the language that is implemented is detailed in APPENDIX C. The program is written in FORTRAN IV. The program is currently running on an HP Series-1000 E under the RTE-IVA operating system at U. C. Berkeley. It is also running on the CDC 6400 at U. C. Berkeley. It is further designed to run on an HP 3000. In order to achieve this portability without maintaining several versions, the following approach is used: Standard FORTRAN-IV is used throughout. However, where machine dependent coding occurs (usually related to I/O and character manipulation), a superset of FORTRAN which requires a pre-processor called SUBST is used. SUBST is available on the U. C. Berkeley IC-CAD research machine (HP 1000). It is also on the HP 3000 contributed library. SUBST recognizes two specifications:

1) a statement of the form

   *CALL FNAME"

   results in the inclusion of the contents of file FNAME at the point of the call.

2) A statement of the form

   s/this/that/

   changes all occurrences of 'this' to 'that.'

If SUBST is not available at an installation, a small program may be written to perform these two functions. The translator uses the functions in four ways:

1) The *CALL feature is used to include files of substitution commands. These *CALL's appear at the beginning of each source file.

2) The *CALL feature is used to include COMMON blocks. Thus change to a COMMON block requires no further adjustment other than recompilation of the affected routines.

3) The substitute feature is used to substitute values for constants, for example, the number "80" for the pattern "linelength."

4) The substitute feature is used to substitute either nothing or a "C" in the first column as appropriate for statements of the form

if computer==<computertype> <fortran statement>

if computer!=<computertype> <fortran statemen>

where

<computertype> ::= 1000 | 3000 | 6400

<fortran statement> ::= <any legal FORTRAN statement>

The details of the use of SUBST and the source file structure are given in APPENDIX A.

BLT uses a memory manager that was originally written for SPICE2. This manager is currently running on several machines including a CDC 6400, an AMDAHL 470, and an HP 3000 [9]. The manager controls memory in a large array from which tables may be allocated, extended, released and cleared. BLT maintains an integer array IMEM equivalenced to REAL array RMEM and to COMPLEX array CMEM. The starting address of IMEM is passed to the manager at the start of the program on the HP machines. On the CDC machine, the address of the first word of data storage (BLANK COMMON) is passed and the limit to memory growth is determined by the total amount of memory available on the machine. The calls to the manager are described below. Further details on the memory manager are available in [10].

1) CALL SETMEM(starting address, memory size) initializes memory.

2) CALL GETM4(pointer,length) gets a block of integer memory of length 'length'. Pointer is set to offset of first word.

3) CALL GETM8(pointer,length) gets a block of double precision real memory of length 'length'. Pointer is set to offset of first word.

4) CALL CLRMEM(pointer) releases the entire block and renders 'pointer' meaningless.

5) CALL EXTMEM(pointer,amount) extends memory block starting at 'pointer' by 'amount' words (of type of block)

6) CALL RELMEM(pointer,amount) releases amount of words from block referenced by 'pointer'.

7) CALL SIZMEM(pointer,size) returns size of block referenced by 'pointer' in 'size'

The main data structures in the program are maintained in IMEM by the memory manager. All tables are of integer type with the exception of two: SYMBS is a table of type real which contains all strings used by the program. A string, as used here, is up to eight characters six on the HP 1000) left justified in a word with right blank fill. VALS is a table of type real which contains all real values used in the program. SYMBS and VALS are double precision real on the HP machines. SYMBS and VALS are accessed by integer functions FNDVR(identifier) and PTVAL(value), respectively. Each function returns an integer corresponding to the location of the identifier (or value) in SYMBS (or VALS). This location is

ascertained by a linear search for a match in SYMBS (or VALS). If no match is found, the new identifier (or value) is appended to the end of the table. If it is found that the linear search results in a serious performance degredation, a hashing scheme may be implemented that is completely transparent to the rest of the program by rewriting only these two routines.

The integer pointers returned by FNDVR (or PTVAL) are used to represent the strings (real values) everywhere else in the program and data. Thus, only one copy of a particular string (value) appears in the data. The advantages of this scheme are threefold. First, significant space savings result when using machines that have multiword real and string storage. Secondly all compares are integer operations which are faster on the above machines. Third, since no type mixing is required, and the strings and values are isolated, portability and simplicity are enhanced.

In general the following convention is used in the integer tables. Pointers to other tables are positive. The first entry in a table of pointers to strings and values is the length of the table unless the size of the table is predefined. The remaining entries are positive and negative integers indicating strings and values respectively (see Figure 9).

References in this report to strings and values will in fact be to the integer pointers. The word 'address' refers to an offset in a table. For example, in a description of the table MDPARS, 'address' means IMEM(MDPARS+address).

PROGRAM STRUCTURE

BLT is divided into three logical sections. They are (1) initialization, (2) parsing, and (3) setup and disk file creation. These three sections are treated in order.

INITIALIZATION

A number of tables are set up at initialization. FRSTWD is a list of keywords. It contains the words "MODEL", "ENDS" and any command keywords ("PLOT", "SWEEP", etc.). This table is searched to match the first word on an input line. If a match is found, the appropriate routines are called to handle the rest of the line. MDTYP, and MDPARS contain the types of models known to the simulator, and the locations of the default parameter lists, respectively. NDMIN and NDMAX contain the minimum and maximum number of nodes associated with each type of model. All four tables are of the same length. The entries in these four tables are a function of the target program for a particular run. The tables are called "parallel" in that the ith entry in each contains information about the same item (in this case a model type). The ith entries in the four tables constitute a C STRUCT [11] or PASCAL RECORD [6] containing information about a model type. The first entry in each is the number of model types (if the first entry is 'n' then n entries follow and the size of the table is n+1). n is the number of low-level model types defined in the initialization file for the target program. DEFVRS and DEFVAL are also parallel tables. Each is a table of tables. Each entry in MDPARS points to a table in both DEFVRS and DEFVAL. The first entry in a DEFVRS or DEFVAL table is the length or number of parameters 'n'. The next n entries are the parameter names (DEFVRS) and the default values (DEFVAL).

Initialization is accomplished by reading a data file into the tables described above. This file is created in a special run that is done once for a particular simulation program. The special run is used to define model types recognized by the program. The parameters, their order, and their default values are defined at this point as are additional keywords to be recognized in FRSTWD.

THE PARSER

The parser is technically a parser and semantic analyzer. It handles topological information and analysis requests. When an input line is read, the first word is scanned and table FRSTWD is searched. If a match is found the appropriate routine is called. The element line has no entry in FRSTWD and is recognized by the absence of any of the features that define the other types of input lines.

The primary translation of the topology description is detailed below and in Figures 10 and 11 The model definition structure resides in MDLST. Since a model of type SUBCKT is treated as a macro definition containing elements and models (which may be of type SUBCKT) the inherent definition structure is a multi-way tree. This tree is represented as a right-threaded binary tree [12]. Low level models have no descendants and consist of four integers:

1) RLINK: a (positive) pointer to the right brother or a (negative) pointer to the father,

2) LTYPE: a (positive) pointer to the entry in MDTYP corresponding to the model type,

3) MNAME: a pointer to the name (in SYMBS), and

4) DPARAMS: a (positive) pointer to the parameter list in PLSTS.

A model of type SUBCKT has six entries:

1) RLINK: a (positive) pointer to the right brother or a (negative) pointer to the father,

2) LTYPE/LLINK: '0' if there are no models defined within the model or a (negative) pointer to the first son,

3) MNAME: a pointer to the name (in SYMBS),

4) FPARAMS: a pointer to a formal parameter list in DEFVRS and DEFVAL,

5) FNODES: a pointer to the node list in NDLST, and

6) ELEMENTS: a pointer to the element list.

The tree just described is built as the models are read. The tree is developed with the aid of OPTSTK, NUPTR, OLDPTR and ANCPTR. OPTSTK is a stack which reflects the nesting of the models. NUPTR is the new model entry. OLDPTR is the previous model entry. ANCPTR is the parent or model within which the current model is defined. The ENDS line marks the end of the current model of type SUBCKT. This indicates that OPTSTK should

be popped and a pointer established from the last model to the parent.

The element lists are linked list structures all of which are in ELST. Each entry consists of five integers:

1) LINK: 'nil' (the last entry) or a pointer to the next entry,

2) MTYPE: the element type (to be later matched with a model name),

3) ENAME: a pointer to the element name (in SYMBS),

4) OPARAMS: nil (no parameters specified) or a pointer to a list of overriding parameters in LKEYS (the keywords or entries of '-1' depending whether keyword or positional parameter specification is used) and EPLSTS (the values of the parameters), and

5) NODES: a pointer to the nodelist in (NDLST).

The structure of MDLST and ELST is shown in Figure 11. The structure of each model and element is shown in Figure 10.

OELSTK is a stack of pointers to elements reflecting the nested structure of the model definitions. Thus, as a SUBCKT MODEL definition is entered, the position of the current element is pushed and a new list is started for the new SUBCKT MODEL.

When an ENDS input line is encountered, DELSTK is popped and the element list may be continued. DELSTK operates in parallel with OPTSTK (see above). Together they contitute a C STRUCT (PASCAL RECORD) of information to be saved on entry to a SUBCKT MODEL definition. DELPTR is the last element. NELPTR is the current element.

If no errors are discovered in the circuit description, the parsing part of the program leaves a tree of models in MDLST. An element list is associated with the root and with each SUBCKT model.

Analysis commands are handled in a simple manner. An attempt has been made to provide facilities to allow a control program to read analysis commands interpretively. At present, they are read by BLT into a table called IANAL. Each type of command has an integer associated with it which corresponds to its position in FRSTWD. When a command is read, the appropriate routine is called and an entry in IANAL is made. The entry consists of

1) LENGTH: the length (in integers) of the command entry,

2) ANTYPE: the command type (as described above), and

3) COMMAND: the specifications of the command (the format and meaning of the specifications is inherent in the value of ANTYPE).

THE SETUP PHASE

After the topology has been completely described, the 'GO' command initiates the setup phase and if no errors are found the disc file is created.

First, all non-SUBCKT MODELS, both those locally defined and the low-level type default models are written to disc. During the parse phase described above a table is maintained whose ith element points to the MDLST entry of the ith model read in. This table also reflects the order in which the models are written out. Entries for the low-level types (the entries in MDTYP) are created at this time. The setup phase requires four additional tables CKTREE, CKTEL, NDMAP, and NM2. CKTREE is built in the setup phase and reflects the calling sequence specified by the element lines. It is a multiway tree in which a path exists upward from each leaf to the root. CKTEL contains pointers to the leaves of the tree defined in CKTREE. The leaves compose the actual circuit, and the upper nodes are the subcircuits which were expanded to produce the final circuit. Each node in the tree consists of three entries:

1) PARENT: a pointer to the next level up the calling sequence,

2) ELEMENT: a pointer to the element (in ELST), and

3) MODEL: a pointer to the model (in MDLST).

A CKTREE entry is shown in Figure 12 along with the structure of CKTREE and CKTEL.

CKTREE is built as follows: The element list associated with the root is expanded. If the model type of an element is SUBCKT, the current entry is pushed on DELSTK and the element list of the SUBCKT is expanded. The actual model to which an element refers is determined by matching the element type with all model names in the current level. If no match is found the next level up is searched. If no match is found at all, MDTYP is searched. If there is still no match, an error is generated. If a match is found in MDTYP it is put in CKTREE as negative to indicate the model was not defined locally to the circuit description. If a match is found in MDLST, then CKTREE is traced upwards from the current position. If the same model is found, a recursive situation exists and an error is generated. If no recursion exists, there are two possible cases: (1) the model is a type other than SUBCKT or (2) the model is of type SUBCKT. Case one indicates that the element is an element of the circuit. The parameters and nodes are matched. An entry is created in CKTEL and the process continues. Case two requires that the current element be pushed and the element list associated with the model be expanded.

Whenever a case one element is found it is output to the element file. In order to do this, all parameters and nodes must be 'bound'. Each node name is resolved by searching the list of formal nodes associated with the current subcircuit model (the one being expanded). If the node is not there, it is local to this SUBCKT and its full pathname is:

thisnode : thissubckt : nextlevelup : etc : root.

If the node is a formal node passed into this model then the element of the next level up in CKTREE is searched for the actual name used for this formal node. The above procedure is repeated on this nodename, treating it as the formal node. This goes on until the level at which the node is local is reached. Thus the resolution of references to nodes is also block structured.

A parameter is resolved in the same way as a node with one exception. If the parameter does not appear on the SUBCKT call, the default of the SUBCKT model is returned. If a value is passed at any level that value is returned.

A model on the file is setup in the following format.

1) The low-level model type (pointer to MDTYP)

2) The number of parameters for this type

3) The values (real) of the parameters for this model.

An element on the file contains this information:

1) The file model referred to (simply 'i' for the ith model on the model file).

2) Number of nodes

3) the nodes (internal...simple integers).

4) Number of overridden parameters.

5) parameter pairs consisting of

   a. The parameter number (integer)

   b. The new value of the parameter (real)

Figure 13 shows the structure of the model and element files and sample entries.

INTERFACE OF MOTIS-C WITH BLT

MOTIS-C is a timing simulator written at U.C. Berkeley based on the MOTIS program developed at Bell Laboratories [13]. It runs on the CDC 6400 computer on campus. The program was put up on an HP 1000 computer in order to develop an interface which would permit MOTIS-C to read the intermediate file generated by BLT.

The structure shown in Figure 7 roughly describes the overlay structure of the two programs. Since the overlay structure available on the HP 1000 does not allow return from an overlay, a utility was developed locally to permit the usual tree structure overlay to one level. If a linearly linked overlay structure is all that is available, the calls to the segments may be removed from the main routine. Since the segments are called in order, the linear structure may be imposed by putting the call to the next segment at the end of the current segment. The last segment may then call the first segment to determine if more processing needs to be done. In the actual implementation of the BLT/MOTIS-C program, MAIN calls the overlay segments INIT, PARSE, PASS2, RDMOT, TRAN, and PLOT directly.

One major change was made to MOTIS-C prior to interface with BLT. The fixed array data structures originally in the program were replaced with data areas dynamically managed by the HPSPICE memory manager. It should be noted that this change is in no way related to the fact that BLT uses this manager. It was done solely to increase the power and flexibility of the MOTIS-C

program, enabling it to handle larger circuits.

The subroutine READN and its associated scanning subroutines were replaced by a set of routines that read the BLT intermediate file directly into the MOTIS-C data structures. The structures are unchanged from the original version, except for the incorporation of the dynamic memory management.

Once the circuit description is read in, analysis may proceed. During analysis, the voltages at each plot point are written to disk. The results are then read and plotted by the plot routine PLOT. Utilities are provided which enable PLOT to swap out the analysis environment, swap in the input environment, and UNMAP the internal node numbers to the full user specified node path names. These may then appear on the output as appropriate. The output from the inverter chain shown in Figures 3, 4, and 5 appears in Figure 14.

PERFORMANCE CHARACTERISTICS

BLT was found to run about twenty-five percent slower than the SPICE2 input routines on the CDC 6400 computer. A program counter frequency distribution program was run on BLT to study the dynamic behavior of the program, i.e., where BLT spends its time. BLT spends about seventy percent of its time doing three tasks.

The overhead incurred when extending a block of memory is somewhat greater than is desirable, i.e., a good deal of time is spent in routines COMPRS and COPY4. This is a function of the design of the memory manager, whose behavior is optimized for use with only a few blocks of memory. BLT uses about twenty. Changes to the memory manager which would substantially increase performance in this area could be effected transparent to BLT. Additionally, COPY4, which moves blocks of memory from one location to another, may usually be speeded up dramatically by coding it in assembler (as in SPICE2).

Character manipulation in scanning routines GETCHR, GTCRD and GETWD is time-consuming. This difficulty is circumvented in SPICE2 by means of packing eight characters into a word and using the assembly routine MOVE to extract bytes.

Appreciable time is also spent in routine FNDVR which does a linear search of the table SYMBS (where all strings reside). This search corresponds approximately to the symbol table search in a compiler. The linear search is adequate for small jobs but

a scheme such as the following should be implemented for use on larger circuits: An integer hash table and an integer table of pointers parallel to SYMBS can be allocated. These tables, in conjunction with a hash function with modulus the same size as the hash table can be used to implement an efficient hashing scheme with collision resolution by chaining [14]. This scheme approximately doubles the memory required to store the strings on the CDC machine and increases the memory by thirty-three percent and twenty-five percent for string storage on the HP 3000 and HP 1000 respectively.

CONCLUSION

The development of the translator BLT and the interface to the timing simulator MOTIS-C raised a number of interesting issues. BLT, in expanding the circuit description to the simplified intermediate form, retains a great deal of information that must be communicated to other routines for useful feedback to the user to be accomplished. For example, the internal node numbers must be unmapped for output plotting, as mentioned in the previous section.

Another issue that arises is that of responsibility for error-checking. Certain types of errors, e.g., syntax and invalid parameter names, may be detected by BLT. Those errors particular to the "target" program must be checked for by that program. For example, a general translator such as BLT has no way of knowing that MOTIS-C requires a finite capacitance at all non-voltage-source nodes.

The concepts presented in this report can be generalized. A single input language can be used in an environment wherein a number of design aid programs are available under the control of a main program. A translator such as BLT can be used by the main program to translate circuit descriptions in the unified input language for use by the other programs. The same concept could be applied to a graphical input system. Another approach might translate graphical input into the unified language for documentation purposes. A graphics-macro would map directly into a subcircuit. Libraries and user definitions would be available

to provide information about low-level types for particular programs. The general concept of using a common structural design language as a data base is described in [8]. A single translator such as BLT, coupled with initialization files for each design aid program provides a powerful tool. It takes information from various sources and expands it into a form that minimizes the complexity of the front-end of each design aid program. While a translator such as BLT has limited use on its own, it can become a most powerful utility in an integrated design aid system.

Future work on BLT may involve the addition of an arithmetic expression and function evaluation capability. Generalization of signal-path names to include busses and parts of busses would be worthwhile. BLT would be more powerful if a library is searched for resolution of model names, i.e., during the set-up phase, when no user defined model is found for an element, a library should be searched before searching the table of low-level types. A scheme of this type is used in the TEKTRONIX implementation of the Berkeley language, TEKSIM [7].

BLT, in its current implementation, provides only skeletal facilities for the incorporation of analysis commands. A powerful command language needs to be developed. An immediate need is to enhance the initialization facility for analysis commands to allow the CAD program designer to have BLT parse them accordingly. This would make it as convenient to add or alter a command as it currently is to add or alter a model type. This enhancement could make use of routines already written.

Current work at Berkeley includes interfacing the mixed-mode simulator SPLICE [15] to BLT on the CDC 6400. MOTIS-C is being interfaced to BLT on the CDC machine as well.

ACKNOWLEDGEMENTS

There are a great number of people without whose help and support this work would not have been possible. I would like to thank my research advisor, Prof. D. O. Pederson, for his constant encouragement and vision throughout the development of BLT. I would like to thank my wife Angela for her patience and understanding for the duration. Prof. A. R. Newton was instrumental in the formulation of the project and his ideas and interest were invaluable. E. Cohen provided constant help with programming problems and an excellent software environment in which to work. Prof. R. W. Dutton and his research group at Stanford University were also instrumental in the development of the project. R. I. Dowell and L. Scheffer of Hewlett Packard were extremely helpful and provided a great deal of practical expertise. Numerous discussions with G. R. Boyle of Tektronix Inc. on the issues involved in the implementation were invaluable. The financial support of Hewlett Packard and the encouragement of M. Brooksby and R. C. Smith are gratefully acknowledged. Finally I would like to thank all of my colleagues in the U. C. Berkeley IC-CAD group who provided a good working environment as well as a vast amount of expertise in all areas related to the project.

REFERENCES

[1] R. I. Gardner, "A Universal Translator for Digital Design Tools," private communication, Hughes Aircraft Co.

[2] L. A. O'Neill, etal, "Designers Workbench--Efficient and Economical Design Aids," 16th Design Automation Conference Proceedings, 1979.

[3] J. D. Crawford, A. R. Newton, D. O. Pederson, and G. R. Boyle, "A Unified Hardware Description Language for CAD Programs," 1979 International Symposium on Computer Hardware Description Languages and their Applications.

[4] S. P. Fan, M. Y. Hsueh, A. R. Newton, and D. O. Pederson, "MOTIS-C: A New Circuit Simulator for MOS LSI Circuits," in 1977 IEEE International Symposium on Circuits and Systems Proceedings.

[5] E. Cohen, A. Vladimirescu, and D. O. Pederson, "SPICE2 Users Guide," Department of Electrical Engineering and Computer Sciences, U. C. Berkeley, Version 2E.3, April 20, 1979.

[6] A. V. Aho, and J. D. Ullman, "Principles of Compiler Design," Addison-Wesley, 1978.

[7] G. R. Boyle, Tektronix Inc., private communication.

[8] W. M. vanCleemput, "A Structural Design Language for Computer Aided Design of Digital Systems," Technical Report No. 136, Stanford Electronics Laboratories, Stanford University, April, 1977.

[9] R. I. Dowell, Hewlett Packard Co., private communication

[10] E. Cohen, "Program Reference for SPICE2," ERL Memorandum, ERL-M592, Electronics Research Laboratory, U. C. Berkeley, June 14, 1976.

[11] D. M. Ritchie, "C Reference Manual," Bell Laboratories, Murray Hill, N. J., May 1, 1977.

[12] D. E. Knuth, "The Art of Computer Programming," Vol. 1, Addison-Wesley, 1973.

[13] B. R. Chawla, H. K. Gummel, and P. Kozak, "MOTIS--An MOS Timing Simulator," in IEEE Transactions on Circuits and Systems, Vol. CAS-22, No. 12, Dec. 1975.

[14] D. E. Knuth, "The Art of Computer Programming," Vol. 3, Addison-Wesley, 1973

[15] A. R. Newton, "The Simulation of Large-Scale Integrated Circuits," ERL Memorandum, UCB/ERL M78/52, Electronics Research Laboratory, U. C. Berkeley, July, 1978.

APPENDIX A--IMPLEMENTATION ON THE HP 1000.

The source exists on several files on FMGR cartridge JC. The names of all source files on the account start with the "&" character. The source files associated with the input translator start with the two character sequence "&I". The reason for this will become clear. The relocatables start with "%I". At load time, three more files are loaded with the "%I" files. They are "%MEM" which contains the dynamic memory manager, %ASLOC which contains function "IADRS" which returns the address of its argument, and "%LINK" which is used by the tree-structure overlay procedure mentioned in the section on the MOTIS-C/BLT interface.

The structure of each source file is now discussed. The first line of each of the source files is a "*CALL" to a file "SBCOMP" which can be a copy of one of any of "SB1000", "SB6400", or "SB3000" depending on which computer the compile is to take place.

Once the appropriate file is copied into "SBCOMP", the program "SUBST" may be run, taking as input a source file and creating a temporary file as output. This temporary output file may then be submitted to the FORTRAN compiler for the particular machine (FTN4 on the 1000, RUNW on the 6400, and FORTRAN on the 3000).

The following pertains to the HP1000. A compile may be effected be bringing a "&I" file into ED and issuing the command ")U)CMP". to ED.)CMP is a ED procedure file which runs "SUBST"

on the current file in ED, and generates a temporary file. This temporary file is then compiled. The relocatable file thus produced is given the same name as the source file but with the "&" replaced by a "%". For example, the following command sequence to ED will result in the compilation of "&INIT":

```
E &INIT
)U)CMP
```

At completion of ")CMP" the relocatable file named "%INIT" will contain the current relocatable version of INIT (in the absence of compile errors, of course). The program may be loaded by running ED and issuing the command ")U)NLD".)NLD creates and executes a file which, when executed, concatenates all of the relocatables and runs the system loader, LOADR, on the concatenation.

APPENDIX B--USE OF BLT WITH A DESIGN-AID PROGRAM

BLT may be interfaced with a design-aid program in a straightforward manner. The common blocks used by BLT are contained in &IBLK. The main program is called MAIN calls BLT and the analysis programs. Thus a call to the main subroutine of the new program can be included in MAIN.

A file which describes the new program must be prepared. The format of this file is:

```
XYZZY
<PROGRAM NAME>
KEYWDS
<K1> <K2> <K3> ...
MODELS
<M1> (<MINNODES>,<MAXNODES>) <NUMPARAMS> <P1>=<V1> <P2>=<V2> ...
<M2> ...
  .
  .
  .
GLOBAL
<NODE1> <NODE2> ...
END
```

XYZZY is the password that indicates that a new initialization for a program is being created. This word is wired into subroutine INIT and should be changed by the implementor. PROGRAM NAME is the name of the program for which the initialization is being generated. If an initialization for this program exists, it is destroyed and replaced by the new one. The user specifies PROGRAM NAME as the first line of the circuit description to declare which program is to be run. K1, K2, etc. are keywords which, when encountered as the first identifier on a user input-line, are treated as analysis commands. M1, M2 etc. are the names of the low-level types to be recognized by the program. For example, if the word NAND appears here, the user may define models of type NAND. MINNODES and MAXNODES are the minimum and maximum number of nodes allowed for elements of this model type. NUMPARAMS is the total number of parameters associated with this type. P1, V1 and P2, V2 are keyword/default-value pairs. An example low-level type definition line is:

```
NAND (2,9) 2 TRISE=10NS TFALL=2NS
```

NODE1, NODE2, etc. are defined as global nodes. When these are encountered any where in the user input that a node is legal, they are treated as global nodes. Thus if VDD is defined as a global node, all connections to VDD are made to the same internal node even if the reference to VDD occurs inside a subcircuit definition. The number of global nodes and their internally mapped values are written out on the analysis file described below. A sample initialization is shown in figure 8.

The initialization file which contains the information generated about each program is called INITXX. When a new file is generated, it is called NINITX. This filed must be renamed and cataloged as appropriate for the local file system under name INITXX for future use by BLT.

BLT generates three files when run. They are FELSXX, MODSXX, and ANALXX which contain the intermediate representation of the elements, the models, and the analysis commands, respectively. The files are composed of variable length binary records. The routines used to read and write them are in &IIO and are called RE and WR, respectively. RE and WR use file input/output routines local to each machine. RE reads a record of the length corresponding to the length of the record written out by WR.

ANALXX is read first. The first read returns one integer (the total number of nodes in the circuit). The next read is also of length one and returns NGLOBE (the number of global nodes). The next read is of length NGLOBE and each positive integer is the mapped internal node number for the corresponding global node. Each non-positive integer indicates that the corresponding global node was not used in the this circuit description. Each successive read is of an analysis command. The first integer is the integer number corresponding to which command it is (according to the order specified in the KEYWDS section of the initialization file). The second number is the number of parameters. The remaining entries are the parameters themselves whose characteristics are a function of the type of

command. The file is read until end-of-file.

The model file is then read. It also is read to end-of-file. Each record contains MTYPE (the model type), followed by NUMPMS (the number of parameters). The remainder of the record consists of NUMPMS real values corresponding to the parameters. The size of each parameter depends on the the number of words used to represent a REAL or DOUBLE PRECISION number on the particular machine.

The element file is last. Each element occupies a record. The first entry in the record is the model number MODNUM, i.e., MODNUM corresponds to the MODNUMth record read from the model file. The second entry is NUMNODS (number of nodes) followed by NUMNODS integer node numbers. The next entry is OPMS (number of overriden parameters) followed by OPMS integer/real pairs where the integer is the number of the parameter and the real is the new value.

Analysis may then proceed. When post-processing is done it is desirable to unmap the integer node numbers into the full signal path names. This is accomplished by the following FORTRAN sequence:

```
C       ... GET INPUT ENVIRONMENT
        CALL GETIN
C       ... UNMAP NODES
        .
        .
        .
        CALL UNMAP(INTND,BUF,LEN)
C       ... INTND IS THE INTERNAL NODE NUMBER
C       ... BUF IS A BUFFER TO STORE THE ASCII REPRESENTATION OF THE SIGNAL
C       ... PATH NAME.
```

```
C       ... LEN IS THE LENGTH OF THE SIGNAL PATH NAME (NUMBER OF ASCII STRI
C       ... OF LENGTH maxwordlength OR LESS).
C       ...
        .
        .
        .
C       ... RETRIEVE ANALYSIS ENVIRONMENT.
        CALL GETAN
```

Care must be exercised when variables are referenced between the calls to GETIN and GETAN. No calls to the memory manager should be made at this time.

APPENDIX C--FORMAL LANGUAGE DESCRIPTION

The format for the language description is taken from [6]. Non-terminals are lower case. Terminals are upper case. Literals are enclosed in quotes, e.g., "end" is a literal. Comments are enclosed in "(" and ")", e.g., (This is a comment). Features that are not implemented in BLT are indicated by asterisks (*).

```
()
cktdesc ->          progname
                    title
                    stlist
                    "end"
()
progname ->         IDENT
()
title ->            STRING
()
stlist ->           stmt
                   |stmt stlist
()
stmt ->             elmnt
                   |model
                   |subdef
()
elmnt ->            elnam spnlist mdnam ":" paramlist
()
elnam ->            IDENT
()
nodelist ->         lnlist
                   |lnlist nodelist
()
lnlist ->           IDENT
                   |IDENT "(" bindex ")"*
()
bindex ->           INTCONST*
                   |INTCONST ":" INTCONST*
                   |INTCONST "," intlist*
()
intlist ->          INTCONST
                   |INTCONST intlist
()
mdnam ->            IDENT
()
paramlist ->        param
                   |param paramlist
()
param ->            expr
                   |IDENT "=" expr
()
model ->            "model" mdnam mdtyp ":" paramlist
```

```
{}
mdnam ->            IDENT
{}
mdtyp ->            IDENT
{}
subdef ->           defline
                    stlist
                    "ends"
{}
defline ->          "model" mdnam "subckt" "!" (nodelist) paramlist
{}
expr ->             expr "+" term*
                   |term
{}
term ->             term "*" factor*
                   |factor
{}
factor ->           factor "^" exponent*
                   |IDENT
                   |INTCONST
                   |REALCONST
{}
exponent ->         INTCONST
```

APPENDIX D--MOTIS-C PROGRAM LISTING

Persons who wish to obtain the MOTIS-C program may do so from Doris R. Simpson, ERL Publications Office, 433 Cory Hall, University of California, Berkeley, CA 94720.

Legend:
SL -> Simulation language
T -> Translator
S -> Simulator

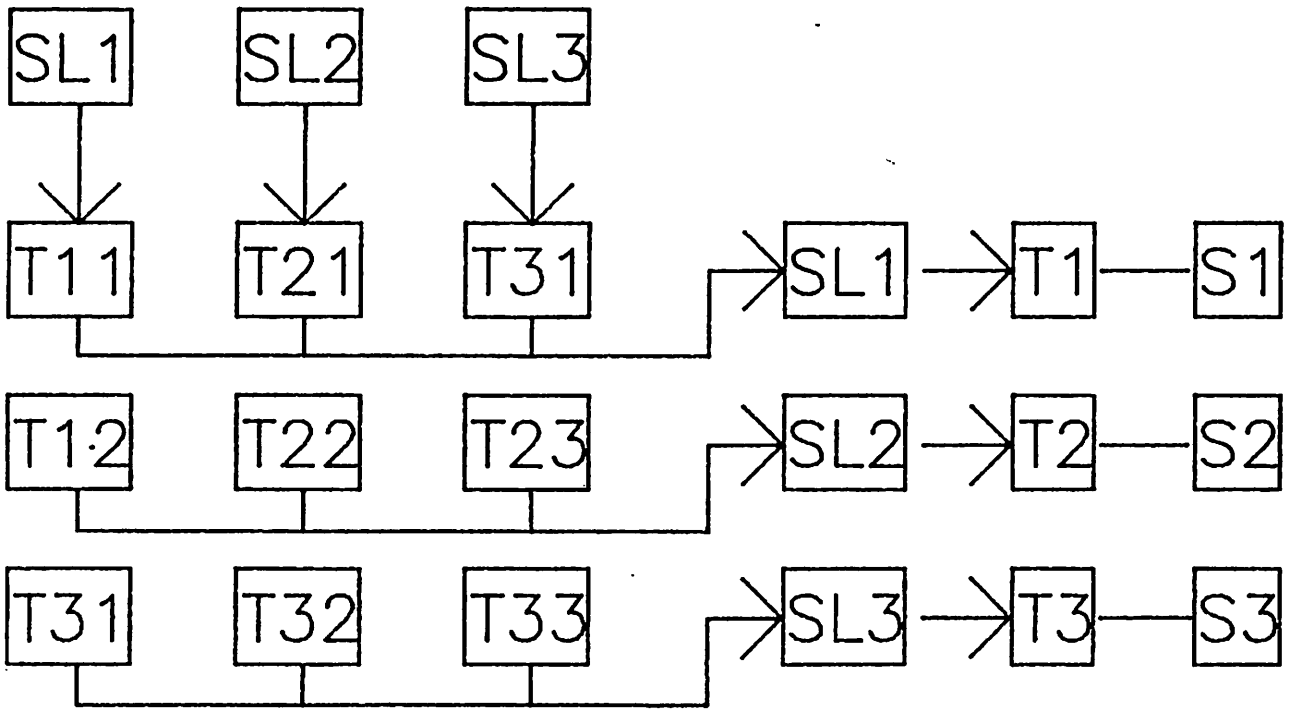


Fig. 1 Cross-bar switch approach to translation among three simulation languages

Legend:

IF -> Initialization file

ED -> Expanded circuit description

SIF -> Simple interface

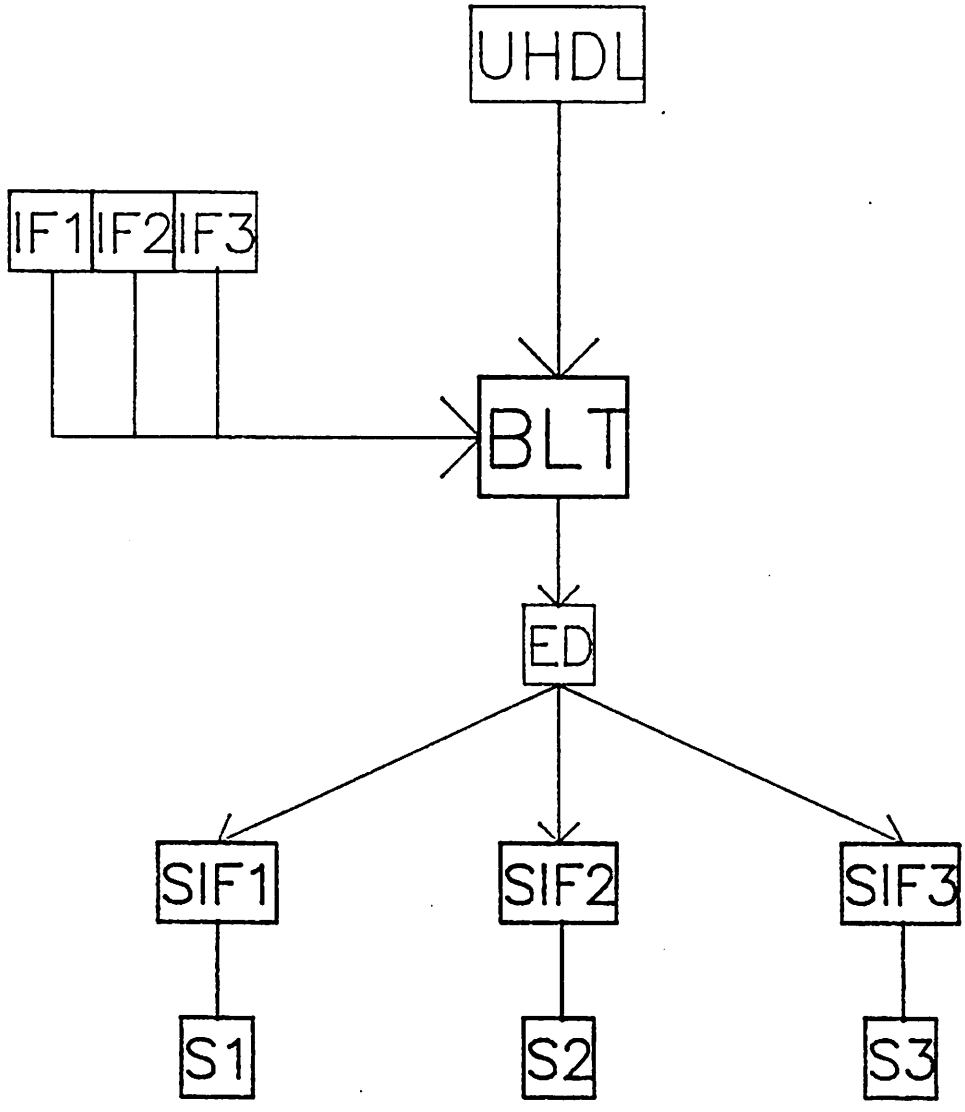


Fig. 2 Unified language approach to communication among design aid programs

MRBG

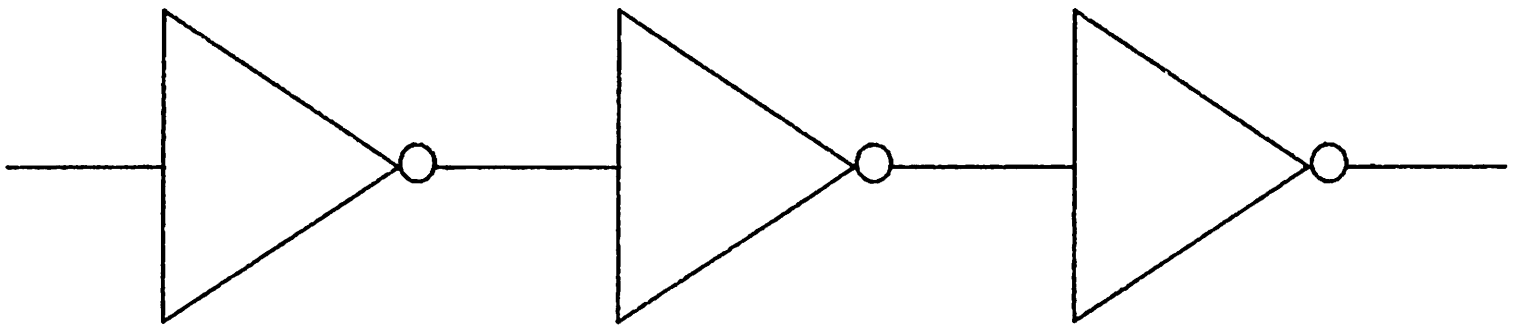
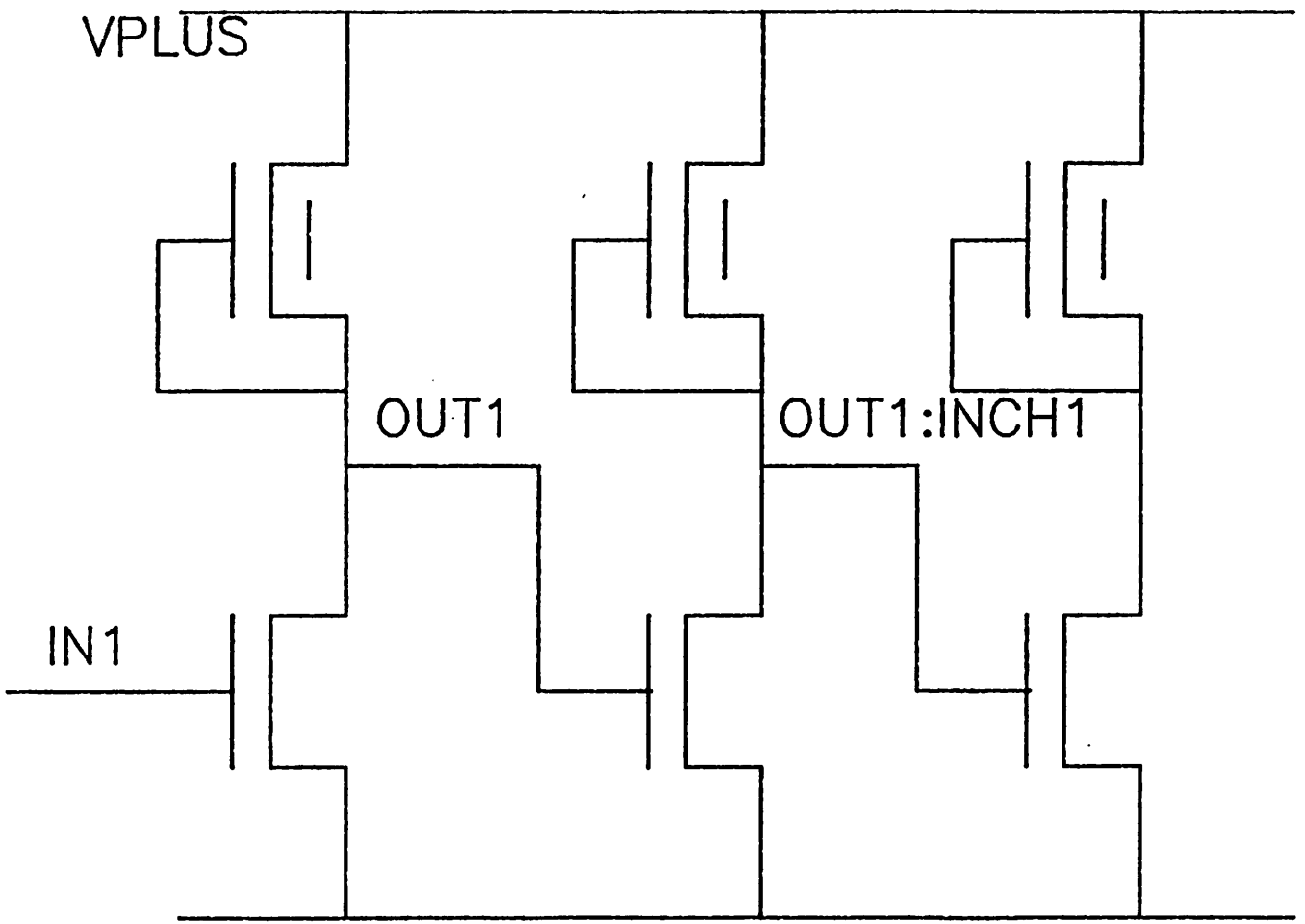


Fig. 3 Inverter chain: Gate representation

Fig. 4 Inverter Chain: Circuit Diagram



```
INVERTER CHAIN 3 STAGES
MODEL EDRV NMOS: VTO=.8 KP=20U GAMMA=.7 PHI=.6 LAMBDA=0$
  CGB=30FF
MODEL DLOD NMOS: VTO=-3 KP=20U GAMMA=.7 PHI=.6 LAMBDA=0$
   CGB=30FF
MODEL INP PULSE: 10 0 5N 2N 2N 48N 100N
;
      MODEL INV SUBCKT: (IN OUT) WD=6U LL=9U
      EDR OUT IN GND EDRV: W=WD L=6U
      DLD VPLUS OUT OUT DLOD: W=3U L=LL
      ENDS
;
       MODEL INCH SUBCKT: (IN OUT) WD=5U LL=8U
       INV1 IN OUT1 INV: WD LL
       INV2 OUT1 OUT INV: WD LL
       ENDS
;
       MODEL LINCH SUBCKT: (IN OUT) WD=12U LL=6U
       INCH1 IN OUT1 INCH: WD LL
       INCH2 OUT1 OUT INCH: WD LL
       ENDS
```

Fig. 5 Inverter Chain: Input description

```
INV1 IN1 OUT1 INV: WD=8U LL=7U
INCH1 OUT1 OUT2 INCH: 10U 6U
LINCH1 OUT2 OUT3 LINCH:
LINCH2 OUT3 OUT4 LINCH: 10U 6U
VIN IN1 GND INP:    TD=10N
;
SET OUT1 0.3  OUT2 0.3  OUT1:LINCH1 0.3  OUT3 0.3$
    OUT1:LINCH2 0.3   OUT4 0.3
;
SET OUT1:INCH1 9.7  OUT1:INCH1:LINCH1 9.7$
    OUT1:INCH2:LINCH1 9.7$
    OUT1:INCH1:LINCH2 9.7  OUT1:INCH2:LINCH2 9.7
;
SWEEP TIME FROM 0 TO 200N BY 2N
PLOT OUT1 OUT2 OUT1:LINCH1 OUT3 OUT1:LINCH2 OUT4
PLOT OUT1:INCH1 OUT1:INCH1:LINCH1 OUT1:INCH2:LINCH1$
     OUT1:INCH1:LINCH2 OUT1:INCH2:LINCH2
VPLUS=10V
;
END
```

Fig. 5 (con.) Inverter Chain: Input description

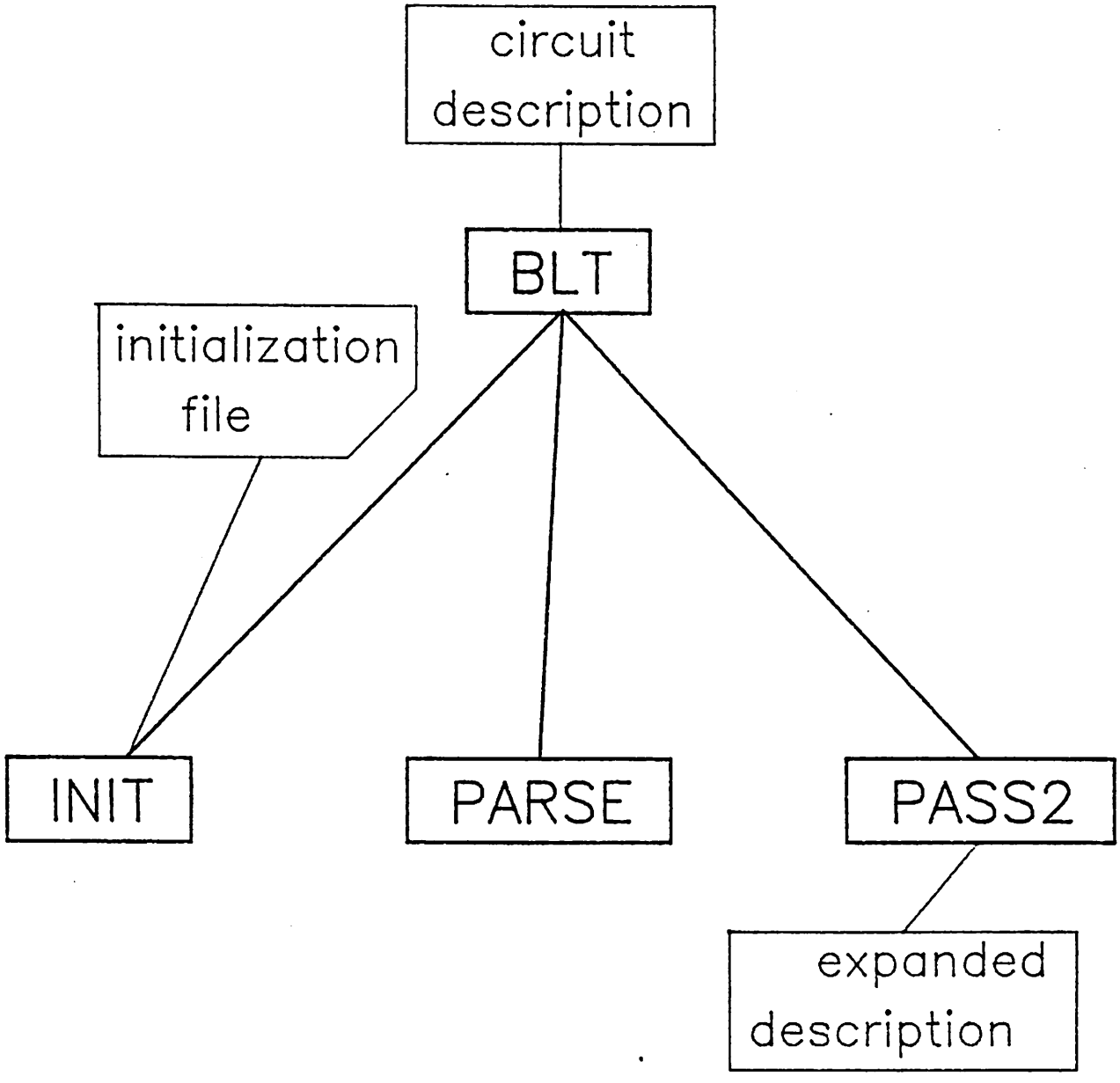
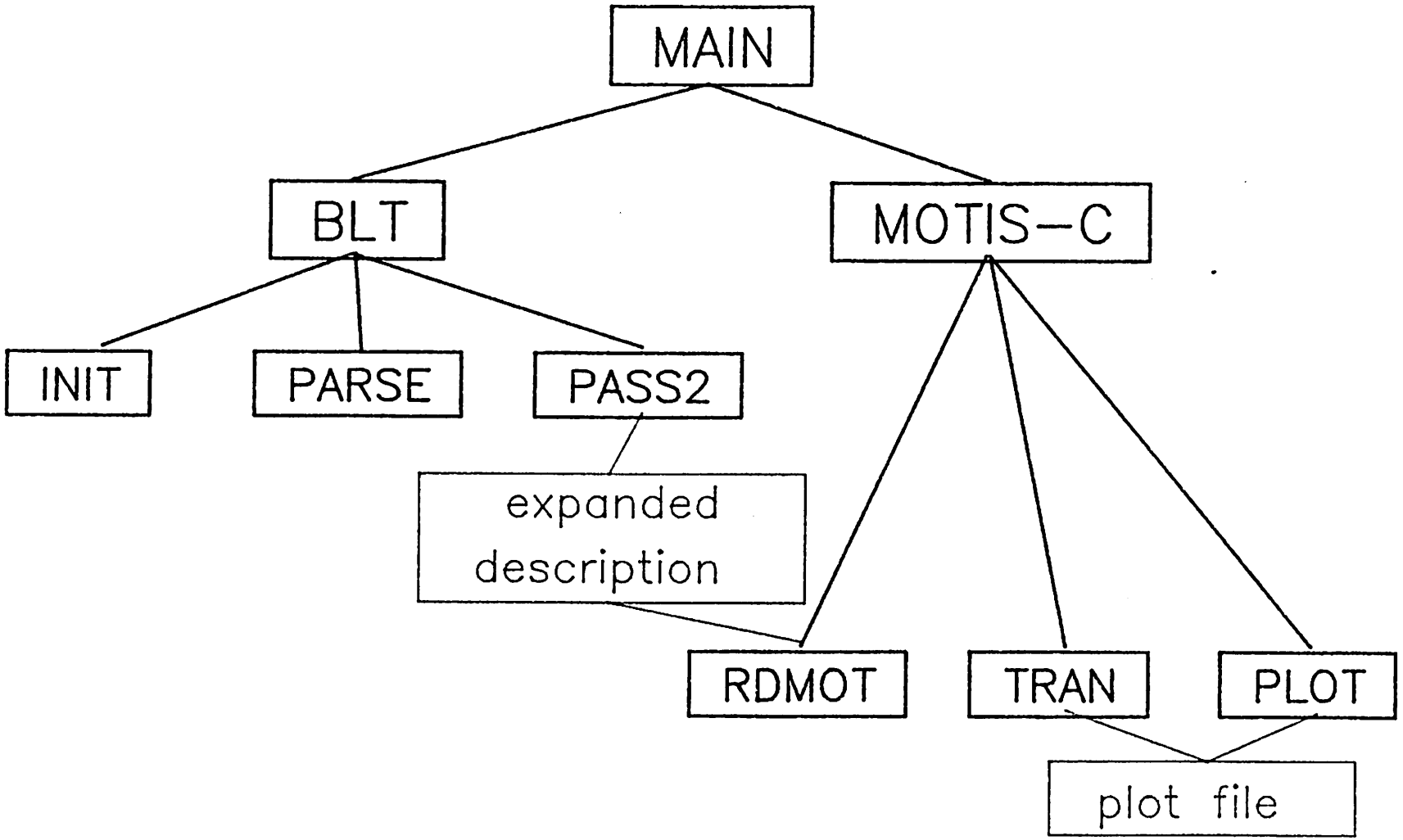


Fig. 6 BLT structure

Fig. 7 BLT and MOTIS–C: Program Structure



```
XYZZY
MOTIS
KEYWDS
TRAN  VPLUS VBG SET
MODELS
NAND2 3 4 ARA=Ø ARL=Ø CA=Ø CL=Ø
NOR2 3 4 ARO=Ø ARL=Ø CO=Ø CL=Ø
ANDOI (1,9) 8 ARA=Ø ARO=Ø ARL=Ø CA=Ø CO=Ø CL=Ø NA=Ø NO=Ø
ORANI (2,6) 8 ARO=Ø ARA=Ø ARL=Ø CO=Ø CA=Ø CL=Ø NO=Ø NA=Ø
NMOS (3,4) 3Ø W=1 L=1 AS=1 AD=1 VTO=Ø KP=24.17U $
           GAMMA=Ø PHI=.6 LAMBDA=Ø CGS=Ø CGD=Ø  CGB=Ø
PMOS (3,4) 3Ø W=1 L=1 AS=1 AD=1 VTO=Ø KP=24.17U $
           GAMMA=Ø PHI=.6 LAMBDA=Ø CGS=Ø CGD=Ø  CGB=Ø
C 2 2 C=1PF IC=Ø
PULSE 2 7 V1=Ø V2=Ø TD=Ø TR=Ø TF=Ø PW=Ø PER=Ø
DC 2 2 NH=Ø NL=Ø
GLOBAL
GND Ø VPLUS
END
```

Fig. 8 Sample initialization

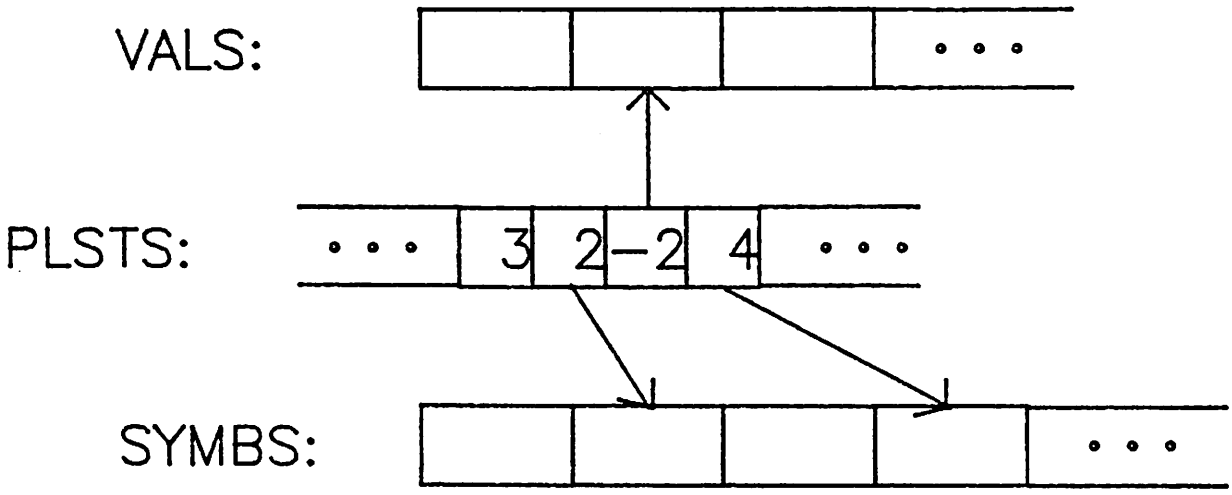


Fig. 9 Sample table

MODEL
RLINK
LTYPE
MNAME
DPARAMS

SUBCKT
RLINK
LTYPE/LLINK
MNAME
FPARAMS
FNODES
ELEMENTS

ELEMENT
LINK
MTYPE
ENAME
OPARAMS
NODES

Fig. 10 MDLST entries for model and subckt model. ELST entry.

Legend:

M -> Model entry in MDLST

S -> Subckt model entry in MDLST

E -> Element entry in ELST

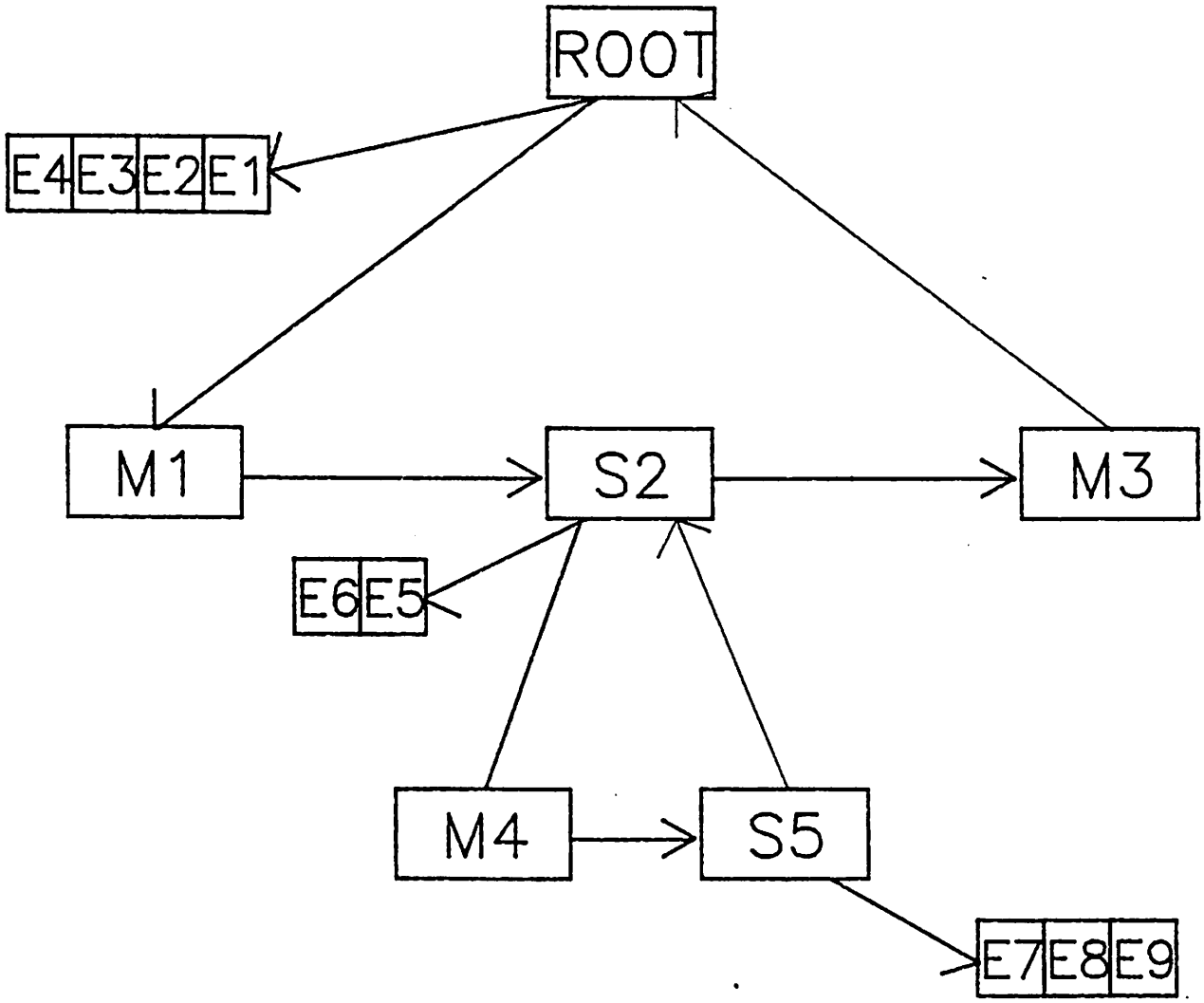


Fig. 11 MDLST and ELST

MRMDLST

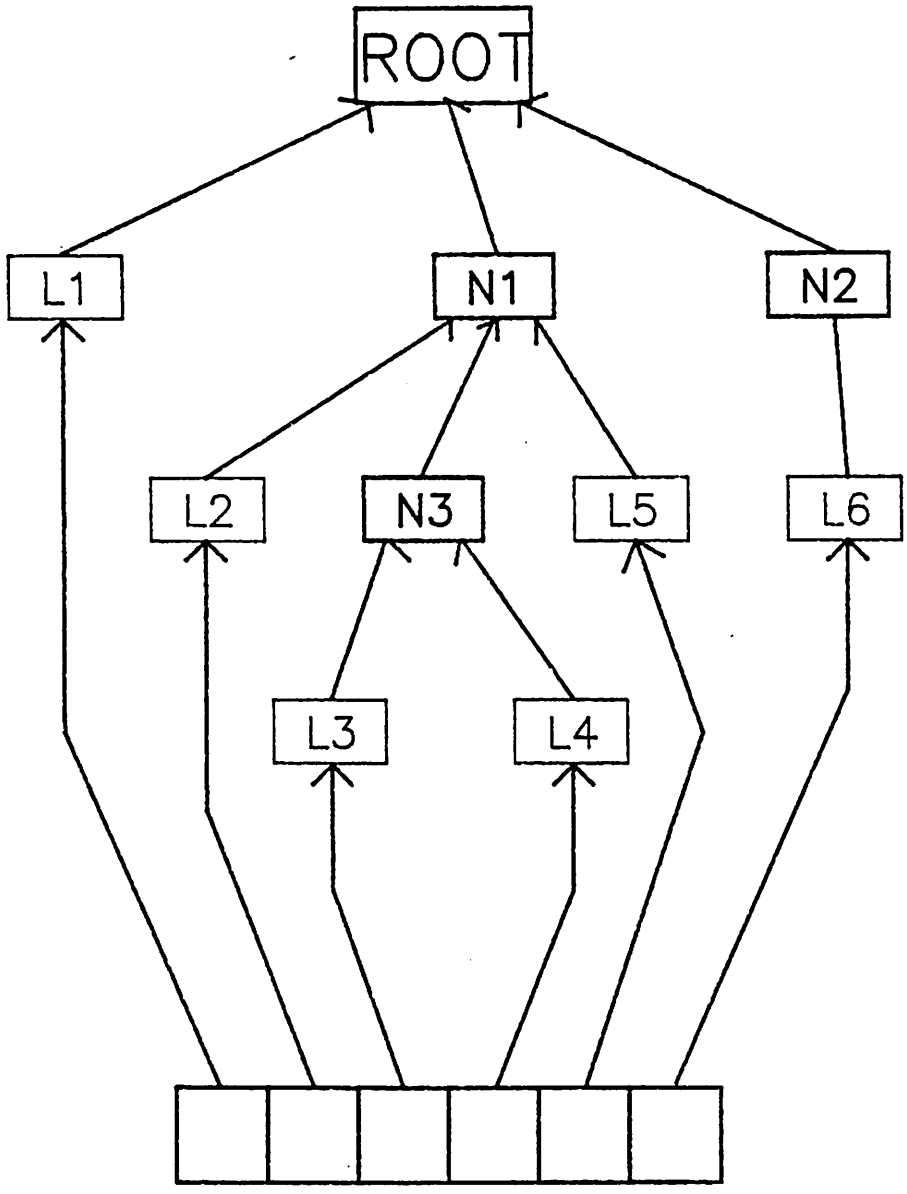


Fig. 12 Expanded—Circuit Tree Structure

MRCKTREE

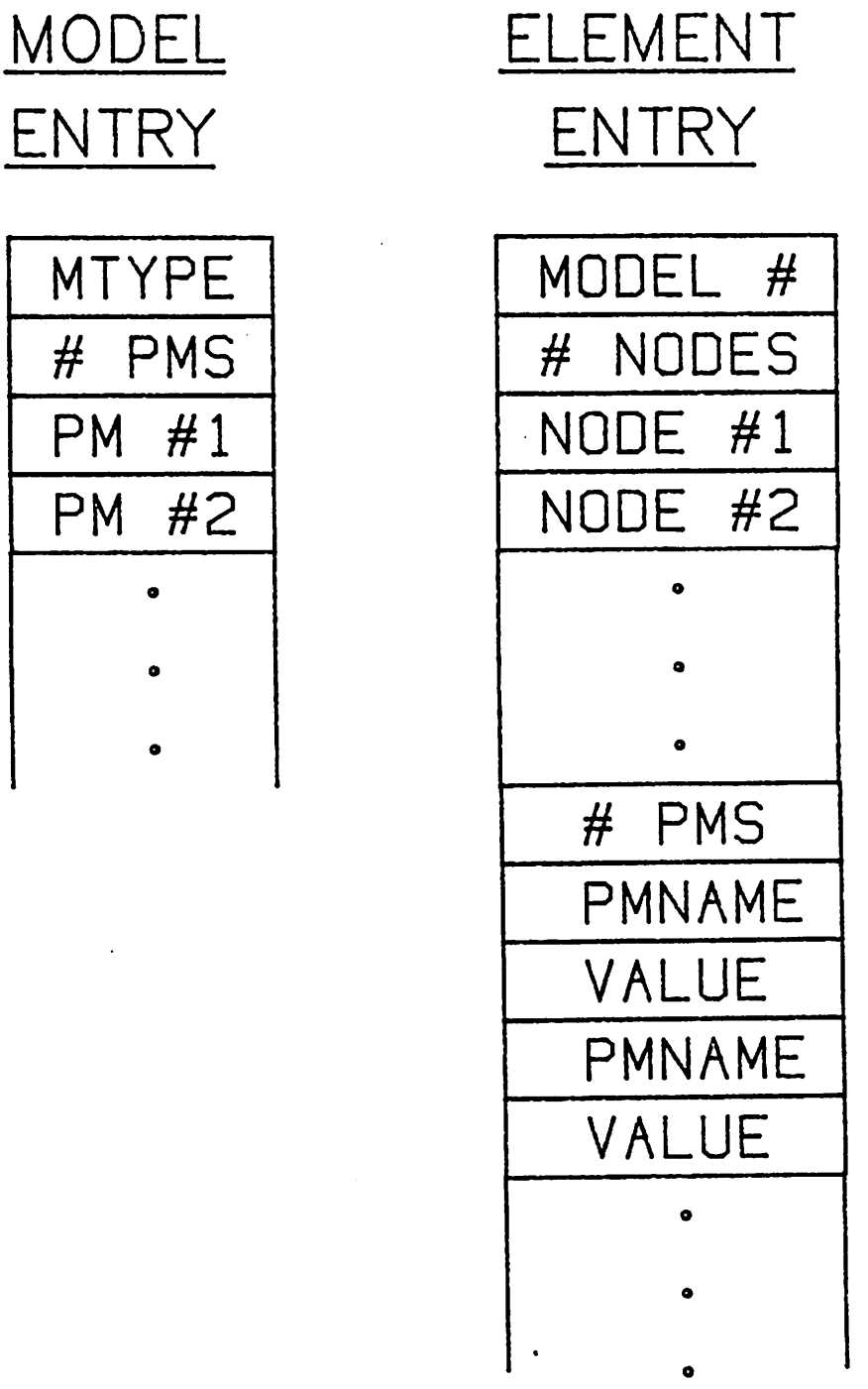


Fig. 13 File Structure

HPFSTRUCT

$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ 228:22:11:37:61 $$$$$$$$$$$$$$$$$$$$$$$$$$$$$

MOTIS-C VERSION 2.0 (1979-05-7)

INVERTER CHAINS 1

$$

LEGEND:
+ <=> OUT2 :
* <=> OUT1 :INCH1
V <=> OUT1 :
& <=> IN1 :

TIME	VOLTAGE AT OUT2 :
0.000E+00	4.840E-02
2.000E-09	1.159E+00
4.000E-09	4.899E-01
6.000E-09	3.103E-01
8.000E-09	3.466E-01
1.000E-08	7.654E-01
1.200E-08	2.282E+00
1.400E-08	3.412E+00
1.600E-08	4.530E+00
1.800E-08	5.667E+00
2.000E-08	6.199E+00
2.200E-08	7.005E+00
2.400E-08	7.427E+00
2.600E-08	8.050E+00
2.800E-08	8.463E+00
3.000E-08	8.789E+00
3.200E-08	9.134E+00
3.400E-08	7.371E+00
3.600E-08	1.123E+00
3.800E-08	2.935E-01
4.000E-08	2.137E-01
4.200E-08	2.270E-01
4.400E-08	1.778E-01
4.600E-08	9.269E-02
4.800E-08	1.509E-01
5.000E-08	2.545E-01
5.200E-08	1.623E-01
5.400E-08	1.047E-01
5.600E-08	1.576E-01
5.800E-08	1.939E-01
6.000E-08	3.325E-01
6.200E-08	1.938E+00
6.400E-08	3.196E+00
6.600E-08	4.365E+00
6.800E-08	5.440E+00
7.000E-08	6.187E+00
7.200E-08	6.728E+00
7.400E-08	7.072E+00
7.600E-08	7.680E+00
7.800E-08	8.247E+00
8.000E-08	8.600E+00
8.200E-08	8.972E+00
8.400E-08	7.237E+00
8.600E-08	1.032E+00
8.800E-08	2.797E-01
9.000E-08	1.930E-01
9.200E-08	7.427E-02
9.400E-08	2.028E-01
9.600E-08	1.305E-01
9.800E-08	1.150E-01
1.000E-07	0.000E+00