AN ABSTRACT OF THE THESIS OF

Doo-Hun Eum for the degree of Doctor of Philosophy in Electrical and Computer Engineering presented on October 19, 1990.

Title : Data-Structure Builder for VLSI/CAD Software

Redacted for Privacy

Abstract approved : ______________________
Toshimi Minoura

Relational database systems have successfully solved many business data processing problems. The primary reason of this success is that the relational data model provides a simple, yet flexible view of data as tables. In studying VLSI/CAD data, we noticed that they are often represented in formats similar to relational tuples. Therefore, they can be stored easily in relational tables. However, it is generally agreed that conventional relational database systems are inefficient for VLSI/CAD applications, since such applications often access large amounts of data repetitively.

In order to solve this problem, we designed and implemented a data mapping subsystem that converts VLSI/CAD data stored in relational tables into internal data structures so that they can be efficiently manipulated in C. By using our data mapping language, we could reduce the amount of code required by the data-structure construction parts of some real VLSI/CAD tools to about 1/10 of that required by C implementation. Our data-structure builder consumes several times more CPU cycles.

Data-Structure Builder for VLSI/CAD Software

by

Doo-Hun Eum

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Completed October 19, 1990

Commencement June 1991

APPROVED :

Redacted for Privacy

Associate Professor of Computer Engineering in charge of major

Redacted for Privacy

Head of department of Electrical and Computer Engineering

Redacted for Privacy

Dean of Graduate School

Date thesis is presented October 19, 1990

Typed by Doo-Hun Eum for Doo-Hun Eum

ACKNOWLEDGEMENTS

There are a number of people who have offered important contributions in the preparation of this thesis. Foremost among them is my major advisor, Dr. Toshimi Minoura, whose guidance has been invaluable. I would also like to thank my thesis committee, Dr. James H. Herzog, Dr. Sayfe Kiaei, Dr. Bella Bose, and Dr. Charles Drake, for their fruitful suggestions and through thesis reading.

Most of all I wish to express my gratitude to my parent, without whose constant support and encouragement this work never have been completed.

My wife, Haeng-Byong, has been an unfailing source of support and has made essential contributions to this work. I also thank my son, Sae-Jin, who has reserved his lovely and cute works and tricks until I finished this work.

TABLE OF CONTENTS

Page

I. INTRODUCTION 1

II. RELATED WORK 4
2.1 Evolving Needs for Databases in Engineering Design 5
2.2 Integrated Design Database 6
2.3 Database Support for VLSI/CAD 7

III. OVERVIEW OF DATA-STRUCTURE BUILDER 13

IV. MAPPING LANGUAGE 23
struct-statement 23
index-statement 24
simple-link-statement 24

V. ADDITIONAL EXAMPLES 30
5.1 Vic (View an Integrated Circuit Layout) 30
5.2 Preprocessing 33
5.3 Readcif 34

VI. IMPLEMENTATION OF DATA-STRUCTURE BUILDER 48
6.1 Input Scanners 48
6.1.1 Input Scanner for *.sim* Format (*Simscan*) 48
6.1.2 Input Scanner for *.ca* Format (*Cascan*) 49
6.2 Mapping Subsystem 49
6.2.1 Functions for Generating Structured View 50
init() 51
record_build() 51
index() 51
field_init() 52
link() 53
6.2.2 Translation 55

VII. PERFORMANCE MEASUREMENT 82

VIII. CONCLUSIONS AND FUTURE RESEARCH 90

BIBLIOGRAPHY 94

APPENDIX 100
A.1 BNF Grammar of Mapping Language 100
A.2 Design Formats 107
A.3 Circuit and Layout Diagrams of 2-Bit Adder 115

LIST OF FIGURES

Figure		Page
3-1	Architecture of data-structure builder	17
3-2	A circuit diagram and a *.sim* file for a CMOS inverter	18
3-3	Relational tables for CMOS inverter	19
3-4	Structured view for CMOS inverter	20
3-5	Mapping script for *presim/rnl*	21
4-1	Data structure for one-one and one-many relationships	28
4-2	Data structure for a many-many relationship	29
5-1	VLSI design tools and data formats used by them	36
5-2	Hierarchy of the *.ca* files for a 2-bit adder	37
5-3	Relational tables of a 2-bit adder for *.ca* format	38
5-4	Structured view of a 2-bit adder for *vic*	39
5-5	Mapping script for *vic*	40
5-6	The *.cif* file for a 2-bit adder	43
5-7	Relational tables of a 2-bit adder for *.cif* format	44
5-8	Structured view of a 2-bit adder for *readcif*	45
5-9	Mapping script for *readcif*	46
6-1	Algorithm *simscan*	57
6-2	Algoritm *cascan*	58
6-3	Mapping subsystem strategy	59
6-4	Header file and global definitions for *presim/rnl*	60
6-5	Generated code for function *init()*	62
6-6	Generated code for function *record_build()*	63

LIST OF FIGURES
(continued)

Figure Page

6-7 Data structure constructed by functions *record_build()* and *index()* 64

6-8 Generated code for function *index()* 65

6-9 Generated code for function *field_init()* 66

6-10 Generated code for function *link()* 68

6-11 Sort-join method for providing linkages 73

6-12 Procedure *memspace()* 74

6-13 Symbol table used by translator 75

6-14 Abstract syntax tree for link-statements 76

6-15 Algorithm for module *genInit()* 77

6-16 Algorithm for module *genRecord_build()* 78

6-17 Algorithm for module *genIndex()* 79

6-18 Algorithm for module *genField_init()* 80

6-19 Algorithm for module *genLink()* 81

7-1 Performance comparison of *presim* and *dbpresim* 84

7-2 Performance comparison of 4 modules of *dbpresim* 85

7-3 Performance comparison of *presim*, *dbpresim*, and *dbpresim2* 86

7-4 Performance comparison of *vic* and *dbvic* 87

7-5 Performance comparison of 4 modules of *dbvic* 88

7-6 Performance comparison of *vic*, *dbvic*, and *dbvic2* 89

8-1 Extended system architecture 93

LIST OF APPENDIX FIGURES

Figure | Page

A-1 Circuit diagram of 2-bit adder 116

A-2 Layout diagram of 2-bit adder 117

DATA-STRUCTURE BUILDER FOR VLSI/CAD SOFTWARE

CHAPTER I

INTRODUCTION

VLSI circuits are becoming more and more complex, and good CAD software tools are essential in their design. A database management system can play a central role in storing and integrating design data, enabling a quick development of new CAD tools.

Current VLSI/CAD systems generally use a file system provided by an operating system to store design data. Although these systems show good performance, they do not achieve a level of integration that accrues from a centralized database management system. For example, to use *rnl*, which is a timing logic simulator described in the VLSI design tools reference manual [UW-87], one needs to create a network description *.net* file or a *.cif* file. The file is then translated by a program (*netlist* for a *.net* file or *mextra* for a *.cif* file) into an intermediate circuit description *.sim* file. Finally, the *presim* program is used to convert the *.sim* file into a binary file suitable for use by *rnl*. However, if the user finds it necessary to change, for example, the value of a capacitance while *rnl* is being executed, he must exit from *rnl*, modify the *.net* file, and repeat the entire process. As seen in this example, a VLSI/CAD system built on top of a file system does not provide a good environment for the integration of design tools.

During the past several years, many researchers observed that conventional database systems are not adequate for CAD applications [EAST-80, HASK-82, BATO-85, KIM-88, HARD-87]. Some of them investigated the data modelling issues and made various proposals [BATO-84, BATO-85, KATZ-85, AFSA-85, KETA-86, KETA-87,

HASK-82, HARD-84, HOLL-84, STON-86, WIED-86]. A few researchers [KATZ-82, CHEN-88] have discussed the issues of supporting rapid development of new design tools. Despite a popular belief in *object-oriented* data models, we found that the formats of most VLSI/CAD data are relational in essence.

Relational database systems have proven very successful in business data processing applications. The primary reason of this success is that they provide a simple table-view of data and impose little preconceived structures. However, current relational database systems cannot efficiently support repetitive access of large amounts of data required by CAD application programs.

In this research, we introduce a data mapping facility, which we call a *data-structure builder*. The data-structure builder is built on top of a relational database management system, and it converts VLSI design data stored in relational tables into data structures best suited for each VLSI/CAD program. This data conversion process follows a script written in a non-procedural mapping language. The script identifies the tuples in relational tables from which the records for such entities as transistors and nodes are constructed, and then it provides linkages among those records so that the data structure can be efficiently manipulated by a conventional programming language. Besides constructing the data structure, the script can also initialize certain fields by using declarative SQL statements. We show in this thesis that the data-structure builder can significantly reduce the amount of programming required for data conversion in VLSI/CAD programs. Although our data-structure builder consumes several times more CPU cycles than a file-based C implementation, we consider this performance penalty can be tolerated because of the following two reasons.

First, a VLSI/CAD program needs to build the internal data structure only once at the beginning of its execution and the major portion of the program is usually CPU-intensive. Second, every few years, computer hardware performance improves enough to

compensate for the increased CPU cycles.

This chapter has served to introduce the data-structure builder, summarizing its objectives and significant features. Related work is reviewed in Chapter II. In Chapter III, we present an overview of the data-structure builder by using a simple example. Chapter IV presents the BNF grammar and semantics for the major constructs of the mapping language for writing data conversion script. We describe the details of the data conversion process by using additional examples in Chapter V. In Chapter VI, the implementation of the data-structure builder is discussed, and the result of performance evaluation of the data-structure builder is presented in Chapter VII.

The final chapter summarizes the significant features of the data-structure builder and the research contributions of this study. Also, remaining unsolved problems are identified, and suggestions for future research are offered.

CHAPTER II

RELATED WORK

Very-large-scale integrated (VLSI) circuits are becoming increasingly common due to their ease of manufacture, low cost, and simplified design methodologies. No longer must the designer study electronics and physics to build an integrated circuits [RUBI-87]. Digital electronic design is taught widely and is accessible to people with any scientific background.

As the complexity of these electronic circuits increase, the need to use computers for their design becomes more important. Although computer-aided design (CAD) systems have existed for quite some time, many of them are inadequate for current tasks, and a continuous flow of new tools is being developed. These tools perform more and more of the detailed and repetitive work involved in VLSI system design, thus reducing the time it takes to produce a chip.

A good CAD system should be able to understand the many interchange formats that allow it to exchange design with other CAD systems. These interchange formats not only allow free flow of design information, but also enable obscure manufacturing styles, understandable by a subset of CAD systems, to be accessible from other systems. Although there are many interchange formats targeted at mask making, there are a few standards. Caltech Intermediate Format (*.cif* format) and Calma GDS II Stream Format are well-known interchange standards, whereas Electronic Design Interchange Format (*.edif* format) is newer and less popular, but its vast extent may carry it to a wide acceptance. These interchange formats all support hierarchical description.

The interchange format for circuit-level simulators that are

integrated at the UW/NW VLSI Consortium [UW-87] is the *.sim* format and it is used to describe MOS transistor networks and their associated parameters. The UW/NW VLSI/CAD system supports the *.ca* format as well as the *.cif* format for layout description. Appendix A.2 gives the details of the *.sim*, *.ca*, and *.cif* formats.

Database management systems have generally been developed to support business applications and, as a result, have often failed to provide adequate support for the management of engineering and design data. CAD systems are powerful tools that designers can use to create, manipulate, evaluate and store design data. The management of these large quantities of data is not a trivial task and can profit from much research and new developments in the area of business database management systems (DBMS). In the following sections, we describe related work and distinguish our approach from the ongoing research in the field.

2.1 Evolving Needs for Databases in Engineering Design

Handling large amounts of data is an integral part of modern engineering practice. The traditional work area of engineers is filled with drawings, books of specifications, handbooks of standard tables and product catalogs. It is not surprising, then, that data files and databases are becoming part of the computer applications in engineering.

The need for handling extensive amounts of data arose concurrently with major applications. An early need was storing constants such as material properties for analysis programs. Such data were stored separately and sequentially read at the beginning of an application, but not modified during their use. By storing data separately, they could be used by more than one program, but still be occasionally extended or modified. When changes are made to the data, they apply to all the projects using them.

Another development was the use of files for exchanging data among different programs. Examples are textual or graphic oriented

programs that expand simple input to the proper format for an application, or that takes output from an application and reformats it into a chart or graphic review. Files provide communication among programs. This use is being applied frequently today [EAST-81].

2.2 Integrated Design Database

The preceding file operations provide interfaces with particular application programs. But as the number of programs grows, the generation and management of input data they use become a burden. Each file of input data has a distinct structure. Data preparation and the writing of interfaces for different file formats has become a major endeavor of many engineering groups.

The concept of data capture suggests that various input processes could all feed into a common data repository, from which is extracted the particular data needed for an application. Data common to a number of applications need only be entered once and used as input for all the applications, with reformatting as needed on input and output. Such a common repository has come to be called an *integrated design database* or just *design database*. This type of integrated approach can greatly reduce the cost of writing a pre-processor for yet another application or for implementing yet another set of file transfer and mapping routines. In the longer context, a design database's benefits include:

1. supporting new forms of integration, such as the automatic generation of production data, such as drawings and numerical control machine tapes, or for generation summary reports to a division or company wide management information system;
2. by keeping all design data in a common machine readable form, the possibility exists to check consistency of data, reducing or eliminating conflicts and improving control of the design product;
3. by keeping data already prepared for use, integrated design

databases support further automation;

4. by eliminating duplications of data;
5. eventually, a design database becomes an environment in which designers work directly. It supports generation as well as analysis, with the possibility of greatly improved productivity.

2.3 Database Support for VLSI/CAD

One of the current trends in database research is supporting an advanced engineering environment such as VLSI/CAD. During the past several years, researchers have realized that the conventional database systems do not support applications in the engineering domain well.

One significant characteristic of VLSI/CAD and all other design is a heavy reliance on hierarchical description. The abstractions on any given level of the hierarchy allow many details at lower levels to be temporarily ignored in understanding the abstractions on the next higher level [SMIT-77]. A *complex object*, as the name implies, is an assembly of objects, which themselves may be assemblies of other objects [HASK-82, BATO-85, KIM-87].

Complex objects are represented as collections of heterogeneous records which are often retrieved together. In VLSI design, the subassemblies are commonly referred to as cells; the use of a cell at the next level of the hierarchy is called an instance [RUBI-87]. One CAD tool may treat a subassembly as a single object, while another CAD tool may be concerned with the detailed structure of that subassembly. Since different levels of abstraction in the hierarchy are appropriate in different stages of the design process, a CAD tool needs to shift the view level in the hierarchy very frequently [KETA-88].

Another significant characteristic that is more specific to electronic design is connectivity. Circuit connectivity must be handled properly in the presence of hierarchy. When a cell instance

is placed in a circuit subassembly, its components must be able to be connected to other components within that subassembly [RUBI-87].

CAD databases are used to store design data and to integrate design tools in VLSI design systems. Efficient access of complex object is essential for computer aided design applications of database management systems. In the conventional first normal form (1NF) relational model, complex objects must usually be decomposed onto different relations. This makes the model semantically difficult to handle by the design tools to retrieve and manipulate the data and performance is guaranteed to be poor because of the large number of join operations required at execution time to construct complex objects.

Also, efficient shift of view levels and representation of circuit connectivity in the hierarchy during a single application run are not possible through traditional unstructured view of relational systems.

The main limitation of the current relational model is its inflexibility, that often prevents relational schema from modeling completely and expressively the natural relationships and constraints among entities. This observation has motivated the introduction of new *semantic data models* [ZANI-83]. The possibilities of extending the relational model to capture more meaning; as opposed to introducing a new model, have also been investigated.

One fundamental approach is to enhance the current DBMSs based on the first normal form (1NF) relational model by adding new capabilities or building another layer of software on the top of current DBMSs to support hierarchical structures. Complex objects are simply stored as ordinary relations and data clustering is not supported.

Haskin and Lorie proposed some extentions that support hierarchic structures to the relational database systems by adding several predefined attribute types and system generated keys to express hierarchical relationships and to speed up join operations [HASK-82]. Zaniolo extends the relational model to support

generalization, aggregation, null values, surrogates, and set-valued attributes [ZANI-83]. Then he extends the relational language QUEL to support these features.

Stonebraker and Rowe introduce the preliminary design of a new DBMS, called POSTGRES, which is the successor to the INGRES relational database systems [STON-86]. POSTGRES proposes to support fairly simple complex objects by supporting an extendible type system, new operators, and new access methods. To support more complex object, POSTGRES proposes to use procedures as their definition mechanism. In addition, a programming language interface mechanism called a *potal* is provided to retrieve data from the database. A potal is similar to a cursor, except that it allows random access to the data specified by the query and the program can fetch more than one record at a time.

The work outlined above only makes the model semantically easy to use by design applications, but the desired efficiency in the CAD environment may not be achieved because a large number of join operations are still required during the executions of CAD applications.

Another approach that has received considerable attention is to generalize the 1NF relational model to abandon the 1NF requirement. The key idea is to allow relations to occur as attribute values of tuples in a relation. Relations of that kind are called Non First Normal Form (NF^2) relations.

A DBMS that is based on the NF^2 relational model should provide direct storage (clustering) of complex objects. Makinouchi recognizes the need for using relational databases for nonbusiness database applications and introduces the NF^2 relations [MAKI-77]. Several researchers have reported query languages for NF^2 relational databases [SCHE-82, FISH-83, ROTH-87]. They extended relational algebra, with main emphasis on the new *nest* and *unnest* operators that transform 1NF relations to NF^2 relations and vice versa.

Dadam et al. prototype a DBMS to support NF^2 data model that

integrates flat and hierarchical relations [DADA-86]. Kemper shows that a behaviorally object-oriented system can be implemented on top of a structurally object-oriented database system that is based on a NF^2 relational model [KEMP-87].

In addition to the work outlined above considerable efforts have been made in developing specific, complex semantic data models to support VLSI/CAD applications. Batory and Kim call complex objects *molecular objects* and develops a framework for capturing the semantics of VLSI/CAD design objects [BATO-85]. A molecular object is a modeling construct which enables a database entity to be represented by two sets of heterogeneous records; one set describes the object's interface and the other describes its implementation.

Ketabchi et al. introduce an object-oriented data model called ODM, in which complex objects are defined as templates that are similar to the class in Smalltalk [KETA-86]. ODM integrates functional data model [SHIP-81] and the actor model of computation. Bancilhon et al. present a database model called FAD, which supports object identity and allows complex objects to be built out of atoms, tuples, and sets [BANC-87]. Hardwick developed an experimental database system called ROSE for CAD/CAM applications, extending relational concepts to make them more suitable to CAD/CAM [HARD-87]. The ROSE data model is based on the entity-relationship model [CHEN-76], but it allows an entity to be constructed as an AND/OR tree [McLE-83]. Such entities are stored in the files of an operating system.

Wiederhold develops a connection between object concepts in programming language and view concepts in relational database systems [WIED-86]. He proposes to combine the concepts of views and objects into a single concept: view-objects. The view-object extracts out of the base data in relational form as needed. A view tuple or set of view tuples will contain projected data corresponding to an object. The view-object generator then assembles the data into a set of objects. The object will be made available to the program by

attaching them to the predefined object prototypes. This approach does not provide the flexibility to explicitly specify the internal structure and linkages of a complex object.

Database user interface is an another area of research that is closely related to our research. We use the word user in the sense of application programs, not in the sense of terminal users. Much effort has been spent to design good programming language interfaces into databases [STON-86, WIED-86]. Current relational database systems (e.g., INFORMIX) usually require a lot of programming when delivering data items to application programs. A common method for relational systems is to execute a query that produces a flat relation (a set of records) and then use a cursor mechanism for accessing the records in the set one by one [DATE-86].

In business data processing applications, meaningful units may contain collections of heterogeneous records, this is not unusual, for example, a customer may have several orders and each order includes several items. To retrieve meaningful units into application programs for further computation, cursors must be defined over queries and repeated operations are needed to deliver data into program arrays [INFO-87]. This situation becomes even worse when relational model is applied to application areas such as information retrieval systems [SCHE-82] and CAD systems.

We believe that the relational model is a foundation, not an end in itself, and it provides a common core of functions that will be needed in all future systems, just as assembly language provides a very primitive core of functions that are needed by all software systems today [DATE-86]. We feel data clustering is difficult to achieve in specially designed semantic data models or NF^2 models when different applications want to cluster data in different ways [WIED-86, HARD-87]. We also feel that a specially designed semantic data model or NF^2 model is difficult to be extended to solve future problems [STON-86]. Rather than building a new system that is

based on a large, complex data model, we believe that we can accomplish our goals by using the small, simple 1NF relational model.

Advantages claimed for the relational model in the classical data processing domain include ease of use, data independence, increased productivity, and multi-file correlation. We believe that these advantages hold for design applications. In fact, given the complexity of design applications and the fact that complex applications evolve continuously and therefore require a lot of flexibility from the point of view of data management, the relational approach might well be the only way to go [HALL-84].

A database system for VLSI/CAD applications needs to represent both the structures and relationships of design entities. In the relational system both are represented by relations. A database system also needs to model the structure of a design entity. Structural relationships are measured using a data model. Unfortunately, the relational model is inadequate for design applications because it was invented for flat, homogeneous entities. Of course, other types of relationships can be simulated in the relational model using artificial keys. However, accessing data in a structural relationship is unnecessarily complex on both the conceptual (user) level and the physical (machine) level.

Network systems are able to represent the structural constraints of design entities more directly. Extentions to the relational model, such as the Entity Relationship model (E-R model), are also able to represent the structural constraints of a design entity directly [HARD-84].

CHAPTER III

OVERVIEW OF DATA-STRUCTURE BUILDER

From the data stored in relations, the data-structure builder constructs the internal data structure to be used by a VLSI/CAD program. A VLSI/CAD program performs this data conversion at the beginning of its execution so that the data structure can be efficiently accessed during the rest of its execution. The data structure thus constructed consists of records and explicit pointers among them. For a circuit simulation program, we provide records for such entities as transistors and nodes, and there should be pointers from the record representing a node to the records representing the transistors connected to that node. For a layout program, records are provided for the rectangles that represent regions in various (e.g., *diffusion*, *polysilicon*, and *metal*) layers. Pointers link related records and provide traversal paths. We call the internal data structure thus constructed a *structured view*.

The system architecture for the data-structure builder is shown in Fig. 3-1. The data-structure builder consists of a simple *input scanner* and a *mapping subsystem*. The input scanner, which is based on a finite state automaton, reads design data in a format such as *.cif*, *.sim*, or *.ca* format, and it then stores them in appropriate relational tables. Under this architecture all the CAD tools share the centralized relational database. Nonetheless, each CAD tool can have a different internal data structure most suitable for its own use. The mapping subsystem performs the required data conversion between these two forms of data, following a script written in a non-procedural mapping language.

The data conversion script consists of three parts: the record-definitions part, the index-statements part, and the

link-statements part. The record-definitions part specifies the records to be constructed, and the link-statements part specifies pointers to be provided among them. The index-statements part specifies an indexing mechanism for fast access to the indexed records. Field initialization statements, which may involve SQL statements, are used within each record definition part.

We now show this data conversion process by using a simple example involving only one CMOS inverter. Fig. 3-2(a) shows the circuit diagram of the CMOS inverter. The *.sim* file of the circuit is shown in Fig. 3-2(b).

The relational tables to be constructed from the *.sim* file are shown in Fig. 3-3. Information concerning nodes in the circuit is stored in table *Node*. Each node has a node name (*nname*) and a node potential (*npot*). The potential values of 0, 1, and 3 represent the logic values *low*, *intermediate or unknown*, and *high*, respectively.

Table *Cap* stores information on capacitances. Each capacitance has a capacitance number (*id*), the node name (*cnode*) of the node to which the capacitance is connected, and a capacitance value (*cval*) in pF. Only one node to which a capacitance is connected is shown because the other end of the capacitance is assumed to be grounded.

Table *Trans* stores information on transistors. Each transistor has a *gate* node, a *source* node, and a *drain* node representing its connection. Attributes *twidth* and *tlength* of table *Trans* represent in *lambda* the width and the length of the gate area of each transistor. Attribute *ttype* represents transistor type (2 for PMOS and 0 for NMOS). A resistor, which is a two-terminal device, is represented in table *Trans* with the value of attribute *gate* set to *Vdd* (a resistor in NMOS is sometimes formed in this way) and the resistance value stored in attribute *twidth*.

Fig. 3-4 shows the data structure (structured view) used by program *presim/rnl* for the logic simulation of the circuit. The current *presim/rnl* program using a file system uses about 22 pages

of programming statements to construct this structured view.

The script given in Fig. 3-5(a) creates the records for the nodes and transistors. The following convention is used in this thesis. We use identically spelled names for a relational table and for the record type for the records created from the tuples in that table. However, the names of relational tables are started with an upper-case letter (e.g., Node and Trans) and those of the record types with a lower-case letter (e.g., node and trans). As we assume that upper-case letters and lower-case letters are different, these names are actually different.

The first FOR EACH ... PROVIDE STRUCT ... statement (PROVIDE-STRUCT statement) creates a *node* record for each tuple in table *Node*. The record definition is similar to the structure definition in C language. The body of PROVIDE-STRUCT statement is applied to each tuple in a relational table.

In the PROVIDE-STRUCT statement for the *node* records, the capacitance field *ncap* is initialized with the expression involving the SQL statements. The CMOS-PW technology is assumed, and CGA (capacitance of gate area) and CPA (capacitance of polysilicon area) are electrical parameters. For each node, the gate areas (each of which can be computed as *twidth * tlength*) of the transistors to whose gates the node is connected are added and then multiplied by the capacitance per unit area (CGA - CPA). This result and other capacitances connected to the node are added to get the total capacitance for that node. The constant, LAMBDA, is the conversion factor from *lambda* to *microns*. Field *npot*, which represents the current potential of the node, is also initialized to the value of attribute *npot* in table *Node*.

The second PROVIDE-STRUCT statement creates a *trans* record for each tuple in table *Trans*. The PROVIDE ... INDEX ... statement (PROVIDE-INDEX statement) creates a hash table on the basis of string values for *nname* field of *node* records.

The scrip given in Fig. 3-5(b) creates pointers between the

node records and the *trans* records. The first LINK ... AND ... statement (LINK statement) provides pointers for the *gate* connections between the *node* records and the *trans* records. The *gate* connection relationship is one-many because there are usually many transistors whose gates are connected to a node. In the WHERE clause, the set of transistor records to be linked to a node record is specified as *Node.nname = Trans.gate*. For each *node* record, a pointer chain is created that begins at the field *ngate* of the *node* record as specified by MEMPTR (member pointer), and that threads through the fields *glinks* of the selected *trans* records as specified by SIBPTR (sibling pointer). Also, a back pointer (BCKPTR) is created in each *trans* record to its owner *node* record in the field *gate*.

The second and third LINK statements provide similar pointers for the *source* and *drain* connections, respectively, producing the structured view as shown in Fig. 3-4.

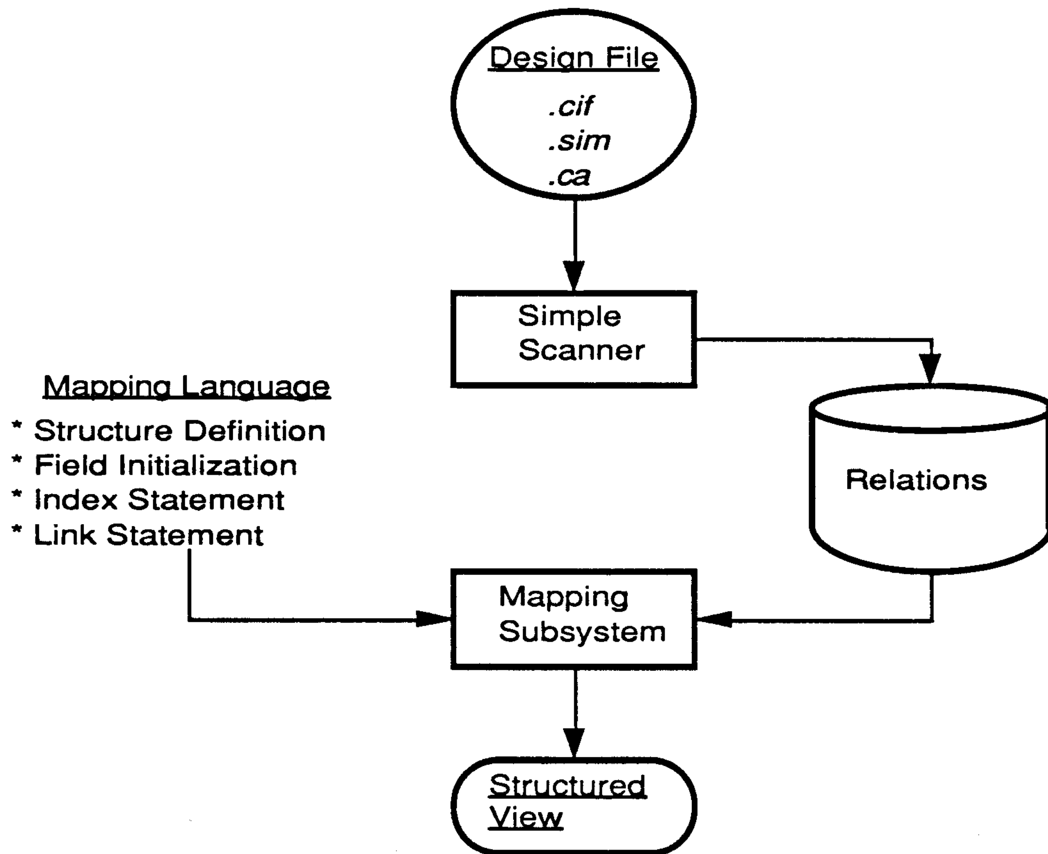
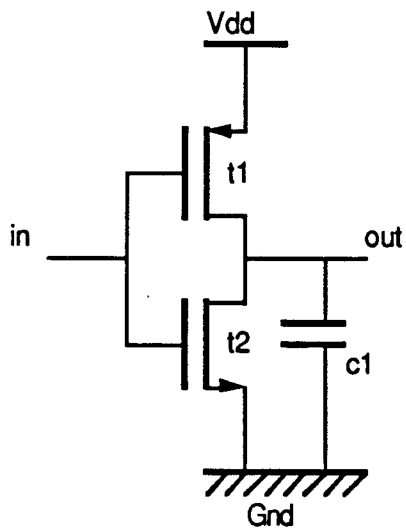


Fig. 3-1 Architecture of data-structure builder.




```
| units: 250.00 tech: cmos-pw  format: MIT
p in out Vdd 8.00 8.00 r 0 0 64.00
e in Gnd out 8.00 4.00 r 0 0 32.00
c out 0.03
```

(a) Circuit diagram. (b) The *.sim* file.

Fig. 3-2 A circuit diagram and a *.sim* file for a CMOS inverter.

Node

nname	npot
Vdd	3
Gnd	0
in	1
out	1

Cap

id	cnode	cval
c1	out	0.03

Trans

id	gate	source	drain	twidth	tlength	ttype
t1	in	out	Vdd	8.0	8.0	2
t2	in	out	Gnd	8.0	4.0	0

Fig. 3-3 Relational tables for CMOS inverter.

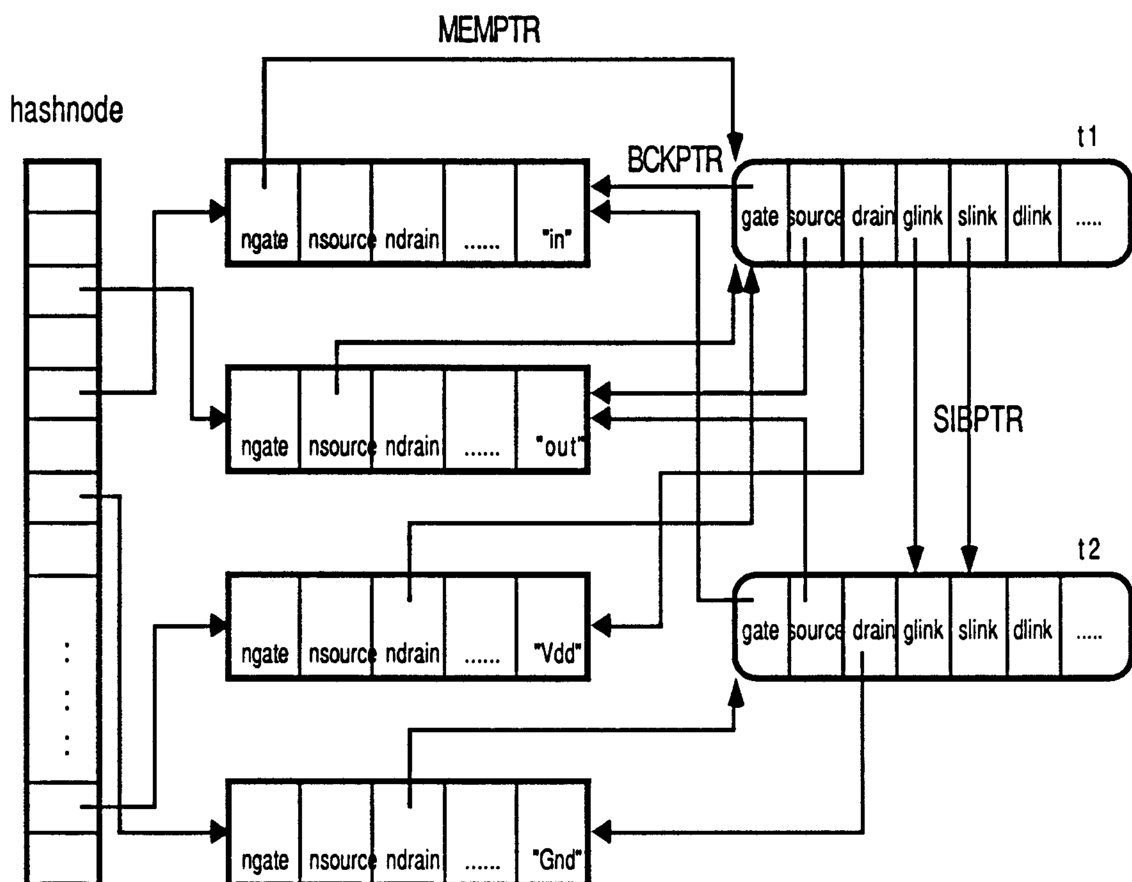


Fig. 3-4 Structured view for CMOS inverter.

```
typedef struct node *nptr;
typedef struct trans *tptr;

FOR EACH Node                          /* for each tuple in table Node */
PROVIDE STRUCT node {                  /* node record definition */
      nptr nlink;                      /* sundries list */
      tptr ngate;                      /* list of gates connected to this node */
      tptr nsource;                    /* list of sources connected to this node */
      tptr ndrain;                     /* list of drains connected to this node */
      nptr hnext;                      /* link in hash bucket */
      float ncap = (CGA-CPA) * LAMBDA * LAMBDA *
      "Select SUM(twidth * tlength) From Trans Where Trans.gate = %Node.nname" +
      "Select SUM(cval) From Cap Where Cap.cnode = %Node.nname";
                                       /* capacitance of this node */
      float vlow = LOWTHRESH;          /* low logic threshold for this node */
      float vhigh = HIGHTHRESH;        /* high logic threshold for this node */
      short tplh;                      /* low to high transition time */
      short tphl;                      /* high to low transition time */
      long ndelay;                     /* delay of last transaction */
      long ctime;                      /* time of last transaction */
      short npot = Node.npot;          /* current potential of this node */
      short nflags;                    /* flag word */
      char *nname = Node.nname;        /* name of node */
      char statestatus;                /* whether node has been set to 0 or 1 */
}

FOR EACH Trans                         /* for each tuple in table Trans */
PROVIDE STRUCT trans {                 /* trans record definition */
      nptr gate;                       /* node to which gate is connected */
      nptr source;                     /* node to which source is connected */
      nptr drain;                      /* node to which drain is connected */
      tptr glink;                      /* gate link in node connection list */
      tptr slink;                      /* source link in node connection list */
      tptr dlink;                      /* drain link in node connection list */
      float twidth = Trans.twidth * LAMBDA;
      float tlength = Trans.tlength * LAMBDA;
      float tnumber;                   /* transistor number */
      int ttype = Trans.ttype;         /* type of transistor */
}

PROVIDE STRHASH INDEX hashnode /* provide string hash index on node name */
ON node(nname)
```

(a) Record definitions.

Fig. 3-5 Mapping script for *presim/rnl*.

```
LINK node AND trans (ONE-MANY)
WHERE Node.nname = Trans.gate
WITH    MEMPTR : ngate
        BCKPTR : gate
        SIBPTR : glink

LINK node AND trans (ONE-MANY)
WHERE Node.nname = Trans.source
WITH    MEMPTR : nsource
        BCKPTR : source
        SIBPTR : slink

LINK node AND trans (ONE-MANY)
WHERE Node.nname = Trans.drain
WITH    MEMPTR : ndrain
        BCKPTR : drain
        SIBPTR : dlink
```

(b) Link statements.

Fig. 3-5 Continued.

CHAPTER IV

MAPPING LANGUAGE

In this chapter, we show the BNF grammar for the major constructs of the mapping language for writing data conversion scripts, and we then explain the semantics of those constructs. In describing the grammar, we use the following conventions.

1. [A] represents an optional (zero or one) occurrence of A.
2. A | B represents an occurrence of A or B.
3. Terminal symbols are represented in two ways: Reserved keywords appear as themselves in bold characters, and punctuation characters and operators are enclosed in single quotation marks.
4. Syntactic units appear in italic characters (e.g., *struct-statement*).

struct-statement

A *struct-statement*, which is used to construct records for the tuples in a relation, has the following syntax:

struct-statement ::= **FOR EACH** *table-name*
PROVIDE STRUCT *record-type*

record-type ::= *record-type-name* '{' *field-list* '}'

field-list ::= *field-declaration* ';' [*field-list*]

field-declaration ::= *type-specifier* *declarator* ['=' *expression*]

The type of the records to be constructed from the tuples in a table *table-name* is specified by *record-type*. The syntax of *record-type* is similar to that of a structure type in C. Selected attributes of each tuple and additional properties are grouped into

the fields specified by *field-list*, forming a record type *record-type-name*.

The declaration of a field may include an *expression* that initializes the field. An expression can be formed from the attributes of the current tuple and SQL statements enclosed in double quotation marks. An SQL statement may include parameters preceded by percent marks. This SQL statement is submitted to the database management system as a character string with the parameters replaced by the actual values for a particular record. Nested record type definitions are not allowed.

index-statement

An *index-statement*, which is used to construct an indexing mechanism for fast access to records, has the following syntax:

index-statement ::= **PROVIDE** *index-kind* **INDEX** *index-name*
ON *record-type-name* '(' *field-name* ')'

index-kind ::= **STRHASH**
| **BSTREE**
| **NUMHASH**

The name of an index is specified by *index-name*. The field to be indexed is specified as *record-type-name* '(' *field-name* ')', which indicates that the index should be provided for field *field-name* of record type *record-type-name*.

We support three kinds of indexes: STRHASH (string hashing), NUMHASH (number hashing), and BSTREE (binary search tree). Indexing mechanism STRHASH is based on hashing on character strings. Indexing mechanism NUMHASH is based on hashing on numbers. Indexing mechanism BSTREE uses a binary search tree on character strings.

simple-link-statement

A *simple-link-statement*, which is used to provide pointers between two record types, has the following syntax:

```
simple-link-statement ::=  link-clause
                           where-clause
                           with-clause

link-clause ::= LINK record-type-name AND record-type-name '(' mapping-kind ')'

record-type-name ::=   type-name [ alias ]

mapping-kind ::=   ONE-ONE
                 | ONE-MANY

where-clause ::= WHERE  predicate

predicate ::= condition  [ AND  predicate ]

condition ::=  item-name '=' item-name

item-name ::=     table-name '.' column-name
                | record-type-name '.' field-name

with-clause ::= WITH   MEMPTR ':' field-name
                       [ SIBPTR ':' field-name
                       [ BCKPTR ':' field-name ]]
```

A *link-clause* is used to provide pointers from the records of the type *R1* indicated by the first *record-type-name* to the records of the type *R2* indicated by the second *record-type-name*. An *alias* is allowed to link records of the same record type. The kind of the relationship type between the two record types should be specified by *mapping-kind*, which may be *one-one* or *one-many*.

A *where-clause* specifies the condition for establishing the linkages. The *predicate* in *where-clause* specifies that record *r1* of type *R1* and record *r2* of type *R2* are to be linked if *predicate* is true when it is evaluated using the attribute values associated with *r1* and *r2*. An attribute used in the *predicate* may be a column name of a table (*table-name* '.' *column-name*) or a field name of a *record-type* (*record-type-name* '.' *field-name*).

A *with-clause* identifies the fields where pointers are stored. We use three kinds of pointers: MEMPTR (member pointer), SIBPTR (sibling pointer), and BCKPTR (back pointer). For one-many relationship, both MEMPTR and SIBPTR must be provided, and BCKPTR is optional. For one-one relationship, MEMPTR must be

provided, and BCKPTR is optional.

We now describe the data structure used by the pointers for each kind of relationship.

1. (one-one) Suppose that we have one-one relationship from records of type *R1* to records of type *R2*. We provide, for each record *r1* of type *R1*, a direct pointer MEMPTR that points to the corresponding record *r2* of type *R2*. We then create a back pointer from *r2* to *r1* if the BCKPTR clause is provided. The resultant data structure is shown in Fig. 4-1(a).
2. (one-many) Suppose that we have one-many relationship from records of type *R1* to records of type *R2*. We create, beginning at the MEMPTR field of each record *r* of type *R1*, a pointer chain that threads through SIBPTR fields of all the type *R2* records *r1*, *r2*, ..., *rk* related to *r*. We then create back pointers from each of *r1*, *r2*, ..., *rk* to *r* if the BCKPTR clause is provided. The resultant data structure is shown in Fig. 4-1(b).

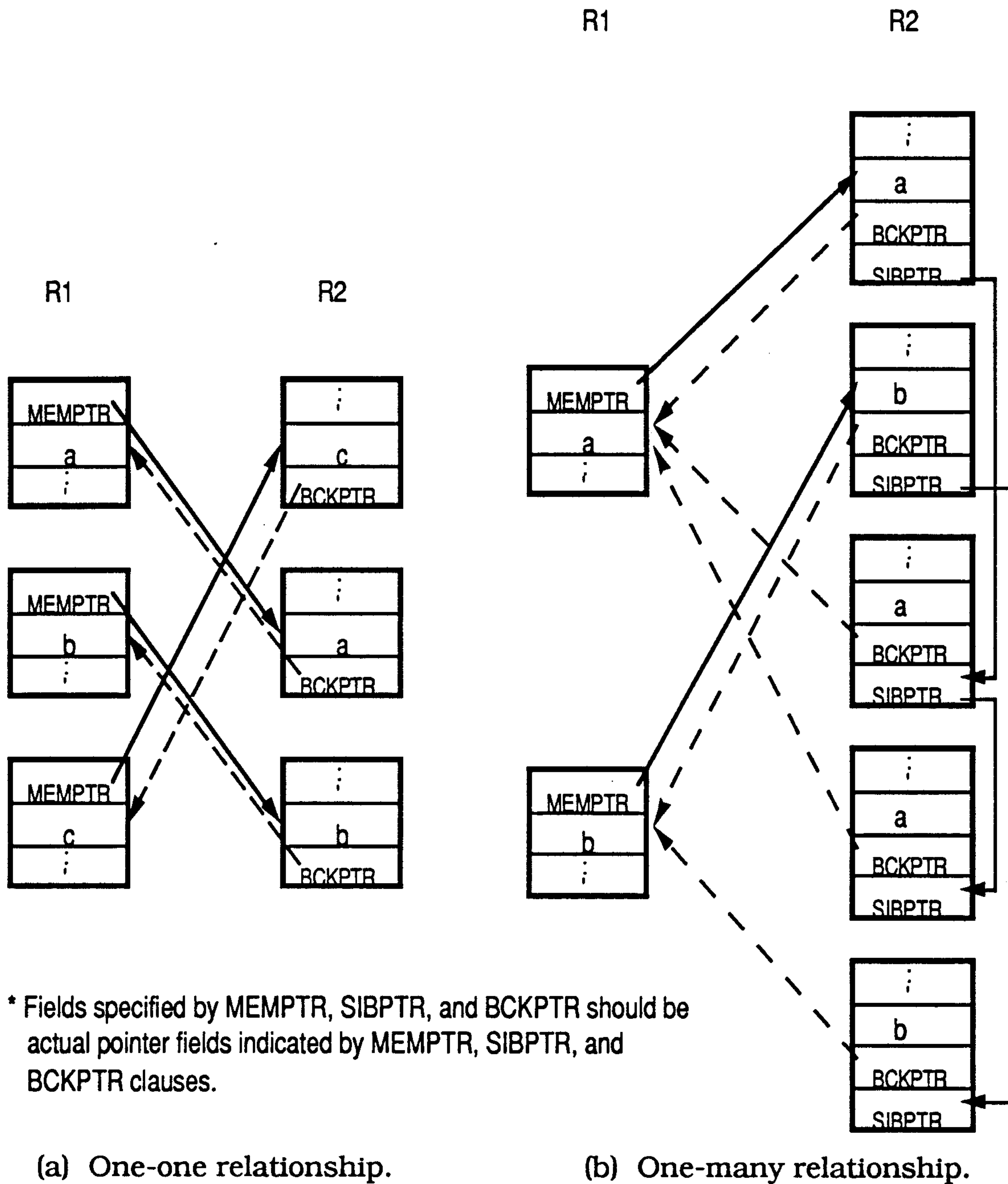
Although our language allows only one-one and one-many relationship types to be specified, we can create a many-many relationship type by using two one-many relationship types.

Suppose we want to establish a many-many relationship from records of type *R1* derived from relation *P1* to records of type *R2* derived from relation *P2*. As we handle this case by two one-many relationship types, we first create a view relation *P3* from which intermediate records of type *R3* can be derived. Assume that the many-many relationship must be established according to the values of fields *f1* of *P1* and *f2* of *P2*. Then the query to create *P3* is as follows:

```
Select  k1, k2
From  P1, P2
Where  P1.f1 = P2.f2.
```

The *k1* and *k2* denote the keys of relations *P1* and *P2*, respectively. A new record type *R3* can now be constructed from *P3* with a

struct-statement. We then provide two *simple-link-statement* statements to construct two one-many relationships: *R1* to *R3* and *R2* to *R3*. The resultant data structure, which is similar to the multilist representation of links in a network model [ULLM-83], is shown in Fig. 4-2.



(a) One-one relationship. (b) One-many relationship.

Fig. 4-1 Data structure for one-one and one-many relationships.

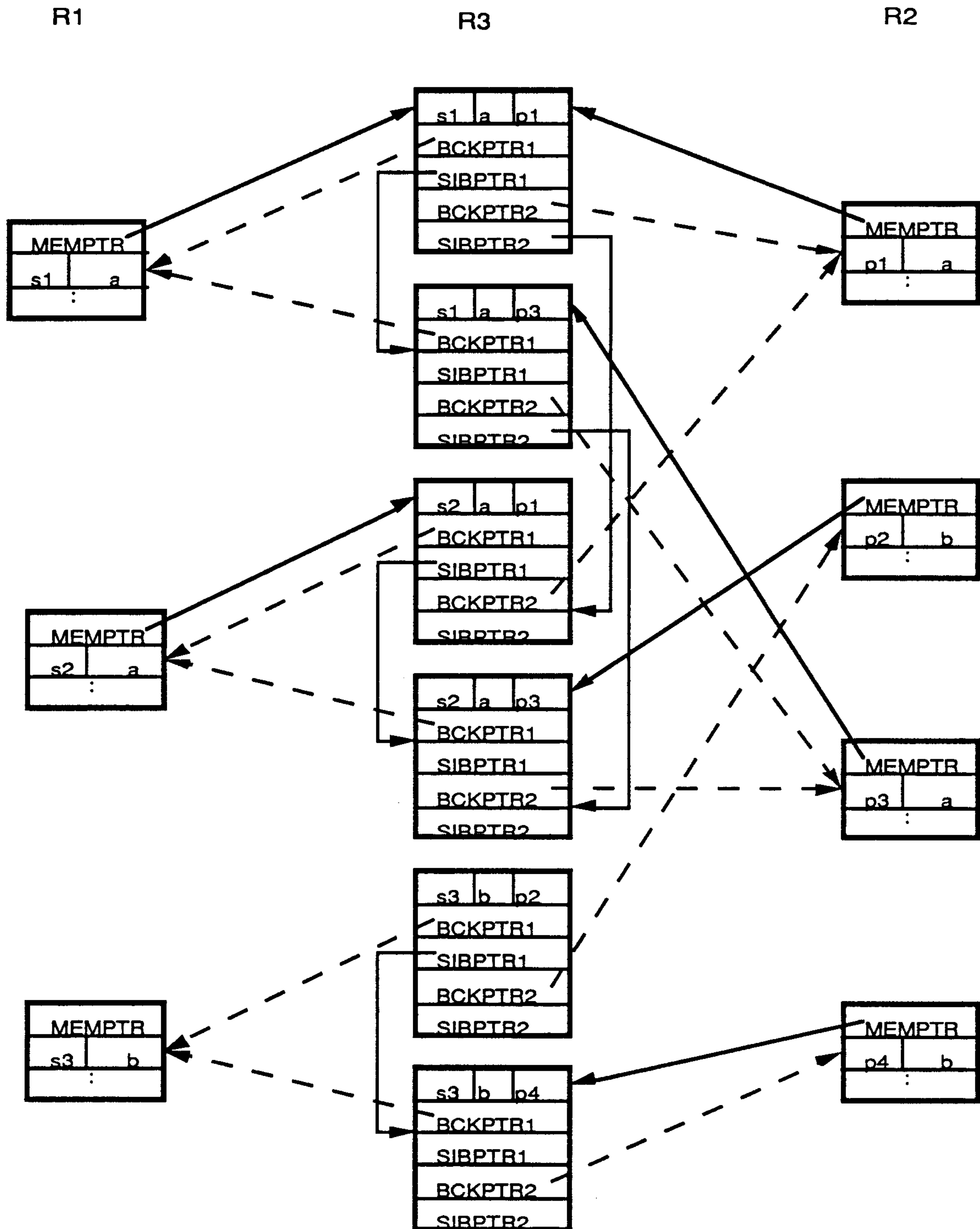


Fig. 4-2 Data structure for a many-many relationship.

CHAPTER V

ADDITIONAL EXAMPLES

Three of the most widely used formats for the representation of VLSI design data are the *.sim*, *.ca*, and *.cif* formats, and many programs accept design data in these formats. The *.sim* format is used to describe MOS transistor networks and their parameters. The *.cif* format allows a hierarchical description of VLSI geometry in a concise manner. The *.ca* format allows us to describe VLSI circuit layouts at a higher level than the *.cif* format. Fig. 5-1 shows how design data in these formats are used by some VLSI design tools.

The data conversion process required by the timing and logic simulator *presim/rnl*, which uses the *.sim* format, was discussed in Chapter III. In this chapter, we show the data conversion processes for the interactive graphics display program *vic*, which uses the *.ca* format, and the CIF library routine *readcif*, which uses the *.cif* format. As an example, we use a layout of a 2-bit adder constructed with the layout assembly program *cfl* (Coordinate Free LAP) [UW-87]. *Cfl* is a library of subroutines intended to facilitate the construction of VLSI circuit layouts. The circuit and layout diagrams of the 2-bit adder are shown in Appendix A.3. *Cfl* produces a set of *.ca* output files, each of which contains the layout of a cell in a hierarchical description of the circuit. *Caesar* can convert those *.ca* files into a single *.cif* file so that it can be used by programs like *readcif* that uses the *.cif* format.

5.1 Vic (View an Integrated Circuit Layout)

Vic, which accepts design data in the *.ca* format, is an interactive graphics display program. Fig. 5-2(a) shows the hierarchy of the *.ca* files for a 2-bit adder cell. Every cell in the hierarchy is represented by a *.ca* file. Two *.ca* files (*adder2.ca* and *fadder.ca*) are

shown in Fig. 5-2(b).

The relational tables constructed from those nine *.ca* files are shown in Fig. 5-3 (some are not complete). Some names were changed for clarity of exposition; e.g., *description* and *symbol* were changed to *cname* and *cell*, respectively.

Table *Cell* stores information on cells in the hierarchical description of the 2-bit adder. Each cell has a cell number (*id*) and a name (*cname*). Attributes *llx*, *lly*, *urx*, and *ury* of table *Cell* represent the *lower-left x-*, *lower-left y-*, *upper-right x-*, and *upper-right y-coordinates* of the cell's bounding box.

Table *Call* stores information on cell calls. A cell call creates an instance of a subcell, which may be translated, rotated, and reflected within the bounding box of the current cell. Each call has a call number (*id*), the cell name of a caller (*caller*), the cell name of a callee (*callee*), and the left six elements of a 3X3 transformation matrix for the callee (*a*, *b*, *c*, *d*, *e*, and *f*).

In circuit design, transformations are typically linear, which means that the shape can be rotated, translated, or scaled. In a two-dimensional domain, this information can be represented with a 3X3 matrix. Since homogeneous coordinate transformations are not used in circuit design, only the left six elements of this matrix are valid. However, square matrices are easier to manipulate, so they continue to be used [RUBI-87].

Information concerning labels is stored in table *Label*. Each label has a label number (*id*), a cell name to which that label belongs (*cell*), the name of the label (*lname*), and the position of the name of the label relative to its center (*position*). Attributes *llx*, *lly*, *urx*, and *ury* of table *Label* represent the *lower-left x-*, *lower-left y-*, *upper-right x-*, and *upper-right y-coordinates* of a label's rectangle. Labels are used to specify texts for the names of such components as signals and cells in a circuit. A circuit extractor often uses those labels when they produce a circuit from a layout description. Rectangles and names associated with labels are used by the graphics

editor.

Table *Box* stores information on boxes that are the basic constituents of VLSI transistors and wirings. Each box has a box number (*id*), the cell name to which a box belongs (*cell*), and a mask layer associated with the box (*layer*). The layer values of 4 and 1 represent *metal* and *polysilicon*, respectively. Attributes *llx*, *lly*, *urx*, and *ury* of table *Box* represent the *lower-left x-*, *lower-left y-*, *upper-right x-*, and *upper-right y-coordinates* of each box.

Fig. 5-4 shows the structured view used by program *vic* for graphics display. The current *vic* program uses about 24 pages of programming statements to build this structured view, which can be constructed by the script given in Fig. 5-5.

In the PROVIDE-STRUCT statement in the *cell* record type, the field *llx*, which indicates the *lower-left x-coordinate* of the bounding box of a cell, is initialized by an expression involving SQL statements. The bounding box of each subcell is transformed, within the bounding box of the cell that calls this subcell, according to the transformation matrix provided for that subcell. A cell consists of boxes that are directly contained in it and the bounding boxes of transformed subcells.

The *llx* value of a cell is computed as follows.

1. Select the minimum *llx* value among the *llx* values of the boxes that are directly contained in the cell.
2. Compute all the x-coordinates of the bounding boxes of the transformed subcells by multiplying the two diagonally located points of each subcell's bounding box, (*llx*, *lly*, 1) and (*urx*, *ury*, 1), by the first column of the 3X3 transformation matrix.
3. Compute the minimum *llx* value of the transformed subcells by selecting the minimum value among the values computed from 2.
4. Select the minimum of the two, which are the value computed from 1 and the value computed from 3, for the *lower-left x-coordinate* of the cell.

The fields *lly*, *urx*, and *ury* are initialized similarly. The functions MINI() and MAXI() compute the minimum and maximum, respectively, of their two arguments.

The last LINK statement provides pointers from the pointer array of each *cell* record to the chains of *box* records. The *box* records associated with each cell are grouped in chains; a chain of *box* records is provided for each mask layer. MAX_LAYER is the number of layers supported by a technology.

5.2 Preprocessing

The *.sim* and *.ca* formats support only such entities as transistors, nodes, rectangles, and labels. In the *.ca* format, a matrix is given as transformation parameters. Storing these entities and the transformation matrix in relational tables can be easily performed by an input scanner. No further processing is necessary for the *.sim* and *.ca* formats.

Besides the entities mentioned above, the *.cif* format supports the Manhattan polygons and wires that are converted into rectangles when an application program builds a data structure. Manhattan geometry means that the edges are parallel to the *x* or *y* axis. Therefore, our input scanner for the *.cif* format converts, in advance, the Manhattan polygons and wires into rectangles.

Although some synthesis tools generate geometry at arbitrary angles, sometimes called Boston geometry, designers rarely use such a facility. The reason is that arbitrary-angle design rules do not exist for many IC processes and, if they did, would be so difficult to check that only a computer could produce error-free layout. In this research, we only consider the Manhattan geometry.

In the *.cif* format, translation is specified as the letter T followed by an *x*, *y* offset. These offsets will be added to all coordinates in the subroutine, to translate its graphics across the mask. Rotation is specified as the letter R followed by an *x*, *y* vector endpoint that defines a line to the origin. The unrotated line has the

endpoint (1, 0), which points to the right. Mirroring is available in two forms: MX to mirror about the x axis and MY to mirror about the y axis. The geometry is flipped about the axis by negating the appropriate coordinate.

Any number of transformations can be applied to an object and their listed order is the sequence that will be used to apply them. Since transformation parameters are not given as a matrix, the input scanner for the *.cif* format must compute the transformation matrix based on the given parameters for rotation, translation, and reflection. Current programs that use the *.cif* format compute these transformation matrices at the beginning of their execution.

The *.cif* format supports arguments to the DS (definition start) statement, which are the cell number and a scaling factor. The scaling factor for a subroutine defined by DS and DF (definition finish) statements consists of a numerator followed by a denominator that will be applied to all values inside the subroutine. This scaling allows large numbers to be expressed with fewer digits and allows ease of rescaling a design. The input scanner applies the scaling factor to all values inside the subroutine and then stores the resultant values in relational tables.

5.3 Readcif

Readcif is a library routine that builds a data structure from a *.cif* file in Manhattan geometry. *Readcif* is used by many programs such as *mextra*, *cifplot*, and *mcp*. Fig. 5-6 shows the *.cif* file for the 2-bit adder, which is converted with *caesar* from the *.ca* files shown in Fig. 5-2.

The relational tables constructed from the *.cif* file after preprocessed by the input scanner are shown in Fig. 5-7 (some are not complete). Table *Cell* stores information on cells in the hierarchical description of the 2-bit adder. Each cell has a cell number (*cell_num*) and a name (*cname*).

Table *Call* stores information on cell calls. Each call has a call

number (*id*), the cell name of a caller (*caller*), the cell name of a callee (*callee*), and the left six elements of a 3X3 transformation matrix for the callee (*a*, *b*, *c*, *d*, *e*, and *f*). Three transformations can be applied in the *.cif* format: translation, rotation, and mirroring. The input scanner for the *.cif* format converts these transformation information to a 3X3 matrix.

Table *Label* stores information on labels. Each label has a label number (*id*), a cell name to which that label belongs (*cell*), the name of the label (*lname*), and label location (*x* and *y*).

Table *Box* stores information on boxes. Each box has a box number (*id*), the cell name to which a box belongs (*cell*), and a mask layer associated with the box (*layer*). Attributes *l*, *b*, *r*, and *t* of table *Box* represent the *left-*, *bottom-*, *right-*, and *top-coordinates* of each box.

Fig. 5-8 shows the part of the structured view used by *readcif*. There is one-many relationship between the *cell* record type and each of the *call*, *box*, and *label* record types. *Readcif* accepts Manhattan polygons and wires and converts them to boxes. NUMHASH, an indexing mechanism based on hashing, is supported by the mapping subsystem and is used to support fast access to *cell* records based on hashed cell numbers. The current *readcif* program uses about 21 pages of programming statements to build this structured view, which can be constructed by the script given in Fig. 5-9.

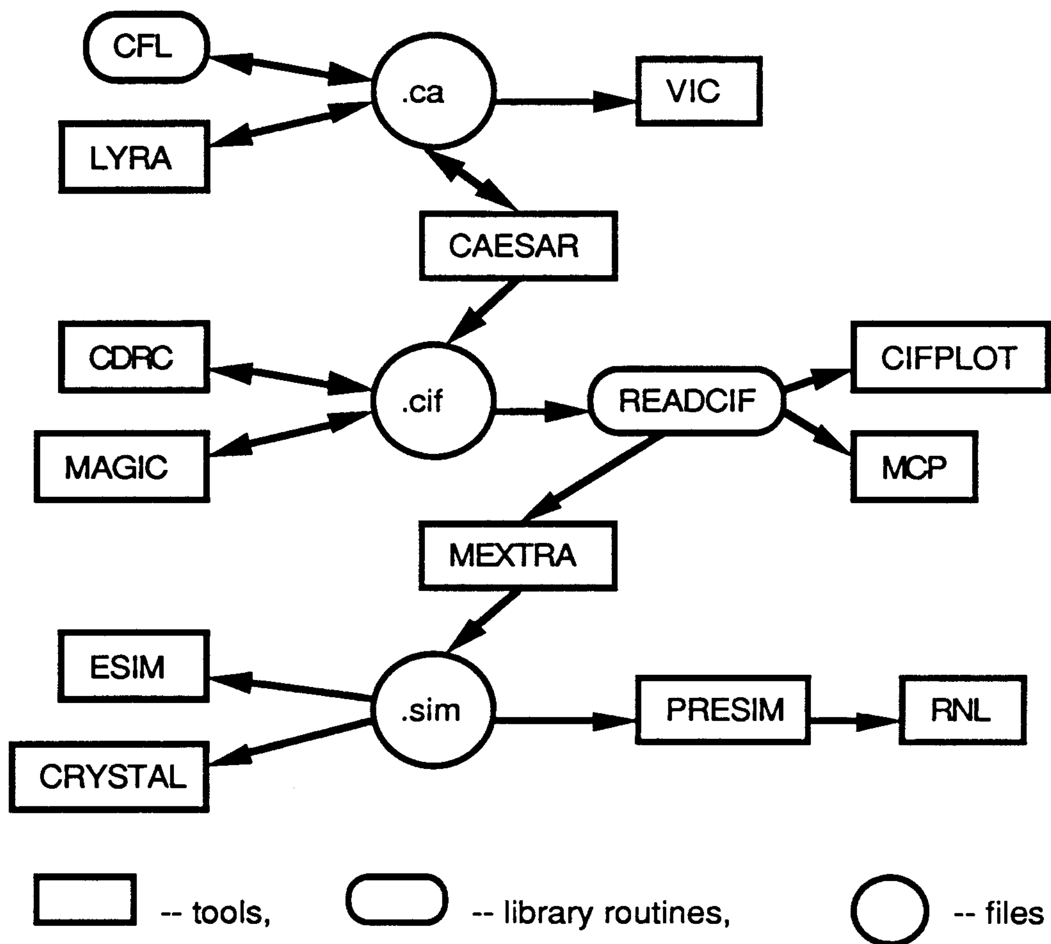
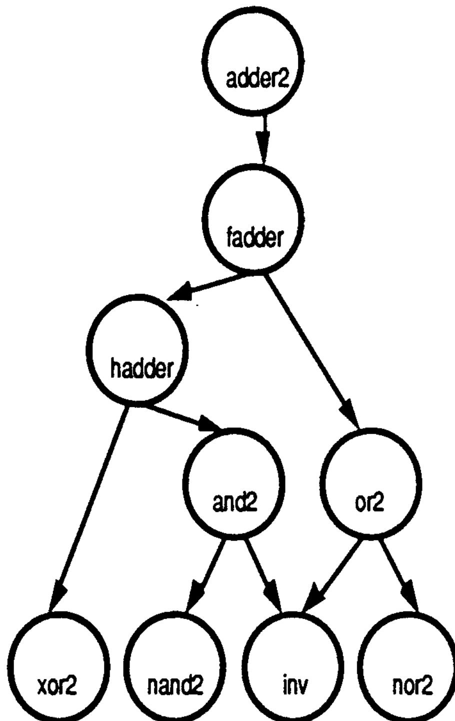


Fig. 5-1 VLSI design tools and data formats used by them.




```
tech cmos-pw
<< metal >>
rect 512 10 527 20
rect 517 10 527 20
        .
        .
<< polysilicon >>
rect 490 -40 496 177
rect 490 171 496 177
        .
        .
use fadder
transform 1 0 0 0 1 0
box 0 -40 502 374
use fadder
transform 1 0 532 0 1 0
box 0 -40 502 374
<< end >>
```

adder2.ca

```
tech cmos-pw
<< metal >>
rect 158 11 170 19
rect 162 11 170 19
        .
        .
<< polysilicon >>
rect 146 -31 152 -20
rect 146 -31 267 -25
        .
        .
use hadder
transform 1 0 0 0 1 0
box 0 0 158 354
use or2
transform 1 0 348 0 1 0
box 0 0 154 135
use hadder
transform 1 0 174 0 1 0
box 0 0 158 354
<< end >>
```

fadder.ca

(a) Hierarchy.

(b) The *.ca* files.

Fig. 5-2 Hierarchy of the *.ca* files for a 2-bit adder.

Cell

id	cname	llx	lly	urx	ury
1	adder2	NULL	NULL	NULL	NULL
2	fadder	0	-40	502	374
3	hadder	0	0	158	354
4	or2	0	0	154	135
5	and2	0	0	158	126
6	nand2	0	0	76	126
7	nor2	0	0	72	124
8	xor2	-158	-8	-40	160
9	inv	0	0	66	112

Call

id	caller	callee	a	b	c	d	e	f
1	adder2	fadder	1	0	0	0	1	0
2	adder2	fadder	1	0	532	0	1	0
3	fadder	hadder	1	0	0	0	1	0
4	fadder	or2	1	0	348	0	1	0
5	fadder	hadder	1	0	174	0	1	0
6	hadder	and2	1	0	0	0	1	0
7	hadder	xor2	1	0	158	0	1	194
8	or2	nor2	1	0	0	0	1	0
9	or2	inv	1	0	88	0	1	0
10	and2	nand2	1	0	0	0	1	0
11	and2	inv	1	0	92	0	1	0

Label

id	cell	lname	llx	lly	urx	ury	position
1	xor2	in#	28	58	28	28	3
2	nand2	GND!	4	14	4	14	3
3	nand2	out#	66	70	66	70	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Box

id	cell	layer	llx	lly	urx	ury
1	adder2	4	512	10	527	20
2	adder2	4	517	10	527	20
⋮	⋮	⋮	⋮	⋮	⋮	⋮
15	adder2	1	490	-40	496	177
16	fadder	4	158	11	170	19
⋮	⋮	⋮	⋮	⋮	⋮	⋮

Fig. 5-3 Relational tables of a 2-bit adder for *.ca* format.

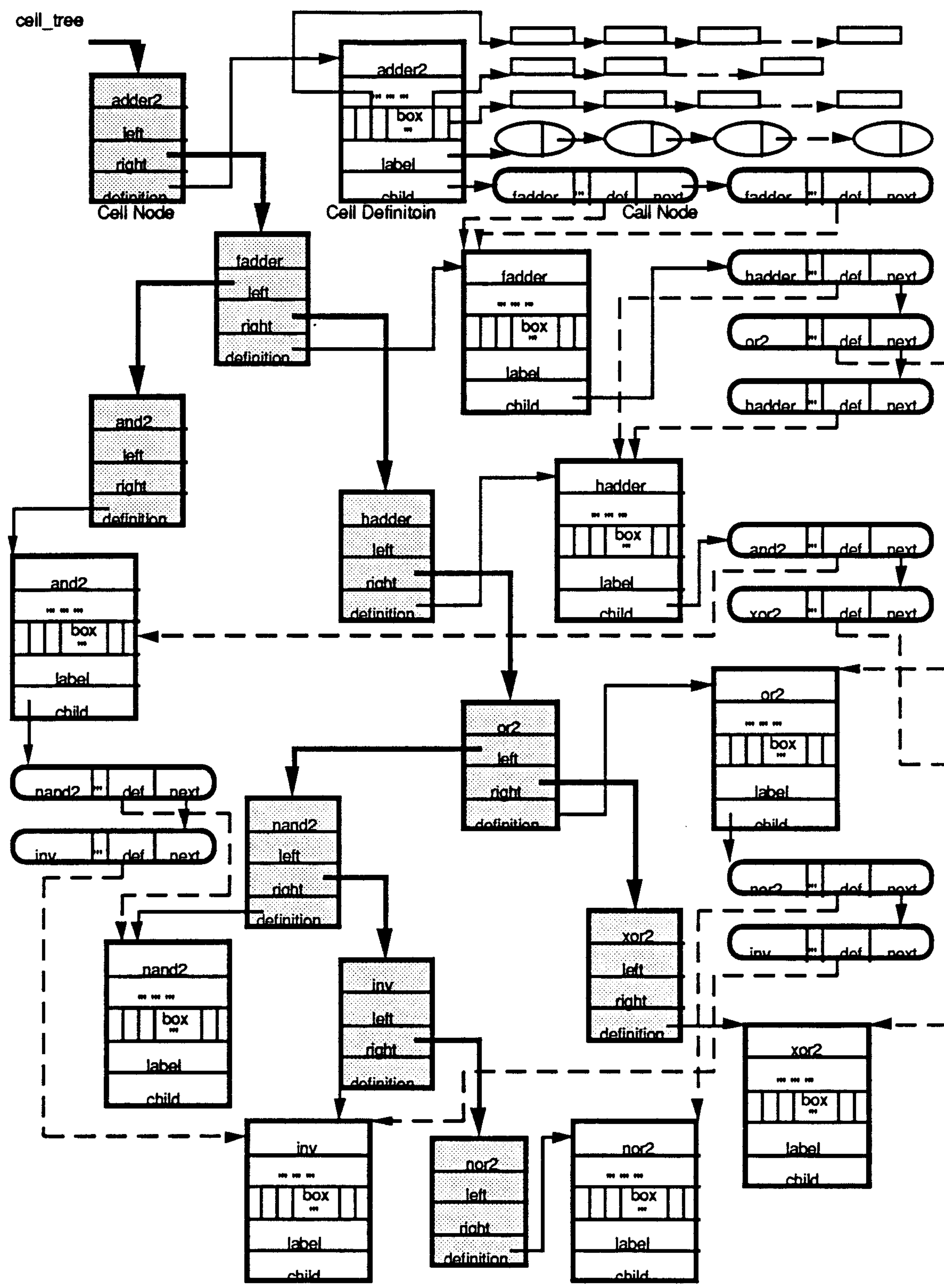


Fig. 5-4 Structured view of a 2-bit adder for *vic.*

```
FOR EACH Label                      /* for each tuple in table Label */
PROVIDE STRUCT label {              /* label record definition */
    char *name = Label.lname;       /* name of label */
    int x = Label.llx;              /* lower-left x-coordinate of label */
    int y = Label.lly;              /* lower-left y-coordinate of label */
    int layer = 32;                 /* layer index for label is 32 */
    struct label *next;             /* link in label list */
}

FOR EACH Box                        /* for each tuple in table Box */
PROVIDE STRUCT box {                /* box reocrd definition */
    int llx = Box.llx;              /* lower-left x-coordinate of box */
    int lly = Box.lly;              /* lower-left y-coordinate of box */
    int urx = Box.urx;              /* upper-right x-coordinate of box */
    int ury = Box.ury;              /* upper-right y-coordinate of box */
    struct box *next;               /* link in box list */
}

FOR EACH Call                       /* for each tuple in table Call */
PROVIDE STRUCT call {               /* call record definition */
    char *name = Call.callee;       /* cell name of callee */
    int a = Call.a;                 /* 3X3 transformation matrix */
    int b = Call.b;
    int c = Call.c;                 /*    a d 0   */
    int d = Call.d;                 /*    b e 0   */
    int e = Call.e;                 /*    c f 1   */
    int f = Call.f;
    struct call *next;              /* link in call list */
    struct cell *definition;        /* pointer to cell record of callee */
}

FOR EACH Cell                       /* for each tuple in table Cell */
PROVIDE STRUCT cell {               /* cell record definition */
    char *name = Cell.cname;        /* name of cell */
    int a = 1;                      /* multiplier, set to 1 for the .ca format */
    int b = 1;                      /* divisor, set to 1 for the .ca format */
    int llx = MINI("Select MIN(Box.llx) From Box Where Box.cell = %Cell.cname",
                 MINI("Select MIN(a*SymY.llx + b*SymY.lly + c)
                       From Cell SymX, Call, Cell SymY
                       Where SymX.cname = Call.caller
                       And Call.callee = SymY.cname
                       And SymX.cname = %Cell.cname",
                       "Select MIN(a*SymY.urx + b*SymY.ury + c)
                       From Cell SymX, Call, Cell SymY
                       Where SymX.cname = Call.caller
                       And Call.callee = SymY.cname
                       And SymX.cname = %Cell.cname"));
                                    /* lower-left x-coordinate of cell */
```

Fig. 5-5 Mapping script for *vic.*

```
    int lly = MINI("Select MIN(Box.lly) From Box Where Box.cell = %Cell.cname",
                   MINI("Select MIN(d*SymY.llx + e*SymY.lly + f)
                         From Cell SymX, Call, Cell SymY
                         Where SymX.cname = Call.caller
                         And Call.callee = SymY.cname
                         And SymX.cname = %Cell.cname",
                         "Select MIN(d*SymY.urx + e*SymY.ury + f)
                         From Cell SymX, Call, Cell SymY
                         Where SymX.cname = Call.caller
                         And Call.callee = SymY.cname
                         And SymX.cname = %Cell.cname"));
                                        /* lower-left y-coordinate of cell */
    int urx = MAXI("Select MAX(Box.urx) From Box Where Box.cell = %Cell.cname",
                   MAXI("Select MAX(a*SymY.llx + b*SymY.lly + c)
                         From Cell SymX, Call, Cell SymY
                         Where SymX.cname = Call.caller
                         And Call.callee = SymY.cname
                         And SymX.cname = %Cell.cname",
                         "Select MAX(a*SymY.urx + b*SymY.ury + c)
                         From Cell SymX, Call, Cell SymY
                         Where SymX.cname = Call.caller
                         And Call.callee = SymY.cname
                         And SymX.cname = %Cell.cname"));
                                        /* upper-right x-coordinate of cell */
    int ury=MAXI("Select MAX(Box.ury) From Box Where Box.cname=%Cell.cname",
                   MAXI("Select MAX(d*SymY.llx + e*SymY.lly + f)
                         From Cell SymX, Call, Cell SymY
                         Where SymX.cname = Call.caller
                         And Call.callee = SymY.cname
                         And SymX.cname = %Cell.cname",
                         "Select MAX(d*SymY.urx + e*SymY.ury + f)
                         From Cell SymX, Call, Cell SymY
                         Where SymX.cname = Call.caller
                         And Call.callee = SymY.cname
                         And SymX.cname = %Cell.cname"));
                                        /* upper-right y-coordinate of cell */
    struct box *box[MAX_LAYER];         /* pointer array of box for each layer */
    struct label *label;                /* list of labels connected to this cell */
    struct call *child;                 /* list of calls connected to this cell */
}

PROVIDE BSTREE INDEX cell_tree          /* provide binary search tree index */
ON cell(name)                           /* on cell name */
```

Fig. 5-5 Continued.

```
LINK cell AND label (ONE-MANY)
WHERE Cell.cname = Label.cell
WITH    MEMPTR : label
        SIBPTR : next

LINK cell AND call (ONE-MANY)
WHERE Cell.cname = Call.caller
WITH    MEMPTR : child
        SIBPTR : next

LINK call AND cell (ONE-ONE)
WHERE call.name = cell.name
WITH    MEMPTR : definition

for (i=0; i<MAX_LAYER; i++) {
    LINK cell AND box (ONE-MANY)
    WHERE Cell.cname = Box.cell AND Box.layer = $i
    WITH    MEMPTR : box[$i]
            SIBPTR : next
}
```

Fig. 5-5 Continued.

```
DS 1 200 4;
9 nor2;
L CP;           B 12 52 14 26;
                B 48 12 32 58;
                B 12 44 14 86;
                :
L CD;           B 28 28 42 30;
                B 16 28 36 58;
                :
94 in2# 72 136;
94 Vdd! 4 220;
                :
DF;

DS 2 200 4;
9 inv;
L CP;           B 44 32 98 112;
                B 12 96 114 48;
                :
94 in# 56 116;
                :
DF;
                :
                :

DS 8 200 4;
9 fadder;
L CP;           B 12 22 298 -51;
                B 242 12 413 -56;
                :
C 7 R 1 0 T 0 0;
C 6 R 1 0 T 696 0;
C 7 R 1 0 T 348 0;
DF;

DS 9 200 4;
9 adder2;
L CP;           B 12 434 986 137;
                B 12 12 986 348;
                :
94 Z1 1644 792;
                :
C 8 R 1 0 T 0 0;
C 8 R 1 0 T 1064 0;
DF;

C 9;
End
```

Fig. 5-6 The *.cif* file for a 2-bit adder.

Cell

cell_num	cname
1	nor2
2	inv
3	nand2
4	xor2
5	and2
6	or2
7	hadder
8	fadder
9	adder2

Call

id	caller	callee	a	b	c	d	e	f
1	and2	nand2	1	0	0	0	1	0
2	and2	inv	1	0	92	0	1	0
3	or2	nor2	1	0	0	0	1	0
4	or2	inv	1	0	88	0	1	0
5	hadder	xor2	1	0	158	0	1	194
6	hadder	and2	1	0	0	0	1	0
7	fadder	hadder	1	0	174	0	1	0
8	fadder	hadder	1	0	0	0	1	0
9	fadder	or2	1	0	348	0	1	0
10	adder2	fadder	1	0	0	0	1	0
11	adder2	fadder	1	0	532	0	1	0

Label

id	cell	lname	x	y	layer
1	nor2	in2#	72	136	NULL
2	nor2	Vdd!	4	220	NULL
:	:	:	:	:	:
12	inv	in#	56	116	NULL
:	:	:	:	:	:

Box

id	cell	layer	l	b	r	t
1	nor2	4	8	0	20	52
2	nor2	4	8	52	56	64
:	:	:	:	:	:	:
15	nor2	1	116	112	128	124
16	inv	4	76	96	120	128
:	:	:	:	:	:	:

Fig. 5-7 Relational tables of a 2-bit adder for *.cif* format.

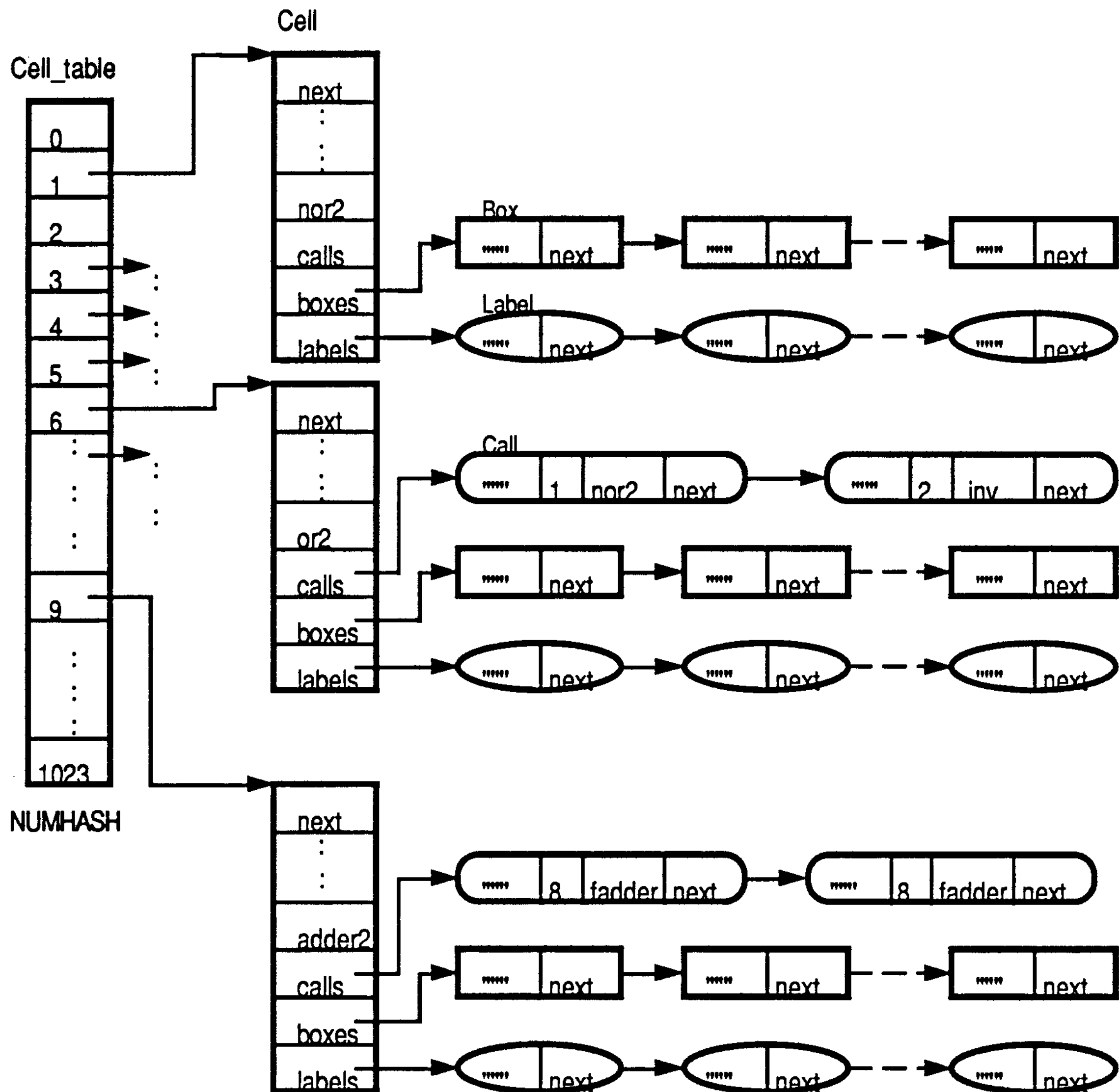


Fig. 5-8 Structured view of a 2-bit adder for *readcif.*

```
FOR EACH Box                          /* for each tuple in table Box */
PROVIDE STRUCT box {                  /* box record definition */
    int type = 15;                    /* object type */
    struct box *next;                 /* link in box list */
    int b = Box.b;                    /* bottom coordinate of box */
    int l = Box.l;                    /* left coordinate of box */
    int t = Box.t;                    /* top coordinate of box */
    int r = Box.r;                    /* right coordinate of box */
    int layer = Box.layer;            /* layer associated with box */
}

FOR EACH Call                         /* for each tuple in table Call */
PROVIDE STRUCT call {                 /* call record definition */
    int type = 14;                    /* object type */
    struct call *next;                /* link in call list */
    int b;
    int l;
    int t;
    int r;
    int layer = Box.layer;
    int symNo = "Select cell_num From Cell Where Cell.cname = %Call.callee";
                                      /* cell number of callee */
    char *callee = Call.callee;
    char *instance;
    int a = Call.a;                   /* 3X3 transformation matrix */
    int b = Call.b;
    int c = Call.c;                   /*   a d 0   */
    int d = Call.d;                   /*   b e 0   */
    int e = Call.e;                   /*   c f 1   */
    int f = Call.f;
}

FOR EACH Label                        /* for each tuple in table Label */
PROVIDE STRUCT label {                /* label record definition */
    int type = 17;                    /* object type */
    struct label *next;               /* link in label list */
    int b = Label.y;                  /* bottom-coordinate of label */
    int l = Label.x;                  /* left-coordinate of label */
    int t = Label.y;                  /* top-coordinate of label */
    int r = Label.x;                  /* right-coordinate of label */
    int layer = Label.layer;          /* layer associated with label */
    char *name = Label.lname;         /* label name */
    char *ltype;
    int used;
    int len;
}
```

Fig. 5-9 Mapping script for *readcif.*

```
FOR EACH Cell                                /* for each tuple in table Cell */
PROVIDE STRUCT cell {                        /* cell record definition */
    int type = 13;                           /* object type */
    struct cell *next;                       /* link in hash bucket */
    int b;                                   /* bottom-coordinate of cell */
    int l;                                   /* left-coordinate of cell */
    int t;                                   /* top-coordinate of cell */
    int r;                                   /* right-coordinate of cell */
    int n = Cell.cell_num;                   /* cell number */
    struct call *calls;                      /* list of calls connected to this cell */
    struct box *boxes;                       /* list of boxes connected to this cell */
    struct label *labels;                    /* list of labels connected to this cell */
    int hook;
}

PROVIDE NUMHASH INDEX Cell_table /* provide number hash index */
ON cell(n)                                   /* on cell number */

LINK cell AND label (ONE-MANY)
WHERE Cell.cname = Label.cell
WITH    MEMPTR : labels
        SIBPTR : next

LINK cell AND box (ONE-MANY)
WHERE Cell.cname = Box.cell
WITH    MEMPTR : boxes
        SIBPTR : next

LINK cell AND call (ONE-MANY)
WHERE Cell.cname = Call.caller
WITH    MEMPTR : calls
        SIBPTR : next
```

Fig. 5-9 Continued.

CHAPTER VI

IMPLEMENTATION OF DATA-STRUCTURE BUILDER

In this chapter, we discuss the implementation of our data-structure builder. The data-structure builder consists of a simple input scanner and a mapping subsystem as shown in Fig. 3-1. There should be an input scanner for each design format, and we implemented input scanners for the *.sim* and *.ca* formats. The mapping subsystem translates a mapping script into the code that reads design data from a database and constructs a structured view. We explain how this translation process is performed by the translator of the data-structure builder.

6.1 Input Scanners

A statement of each design format usually consists of a keyword or letter followed by parameters. As the first keyword or letter of a line (statement) determines a statement type, it is easy to program an input scanner based on a simple finite state automaton. An input scanner reads design data in a format such as *.sim*, *.ca*, or *.cif* format, and it then stores them in appropriate relational tables. In the following sections, we discuss the algorithms of the scanners for the *.sim* and *.ca* formats.

6.1.1 Input Scanner for *.sim* Format (*Simscan*)

Fig. 6-1 shows the algorithm *simscan*, which is the scanner for the *.sim* format. In the *main* routine, *simscan* creates three relational tables for the *.sim* format. Table *Trans* stores the information on transistors, table *Node* stores the information concerning nodes, and table *Cap* stores the information on capacitances.

The procedure *input()* scans a *.sim* file, storing record values in the appropriate relational tables. In the *.sim* format, lines beginning with the mark '@' indicate redirection of input from the named file. *Simscan* uses the recursive call for the input to revert to the current file when the end-of-file is reached.

In the procedure *newtrans()*, the hashing table over the node names is used to determine if the nodes associated with *gate*, *source*, and *drain* of a transistor are already defined. If a node is newly defined, then its name and parameters are inserted into table *Node*.

6.1.2 Input Scanner for *.ca* Format (*Cascan*)

Fig. 6-2 shows the algorithm *cascan*, which is the scanner for the *.ca* format. In the *main* routine, *cascan* creates four relational tables for the *.ca* format. Table *Cell* stores the information on cells, table *Call* stores the information concerning cell calls, table *Box* stores the information on boxes, and table *Label* stores the information on labels. *Cascan* first stores the name of a top-level cell and its parameters into table *Cell*.

The procedure *load()* scans a *.ca* file, storing record values in the appropriate relational tables. In the statements for the parameter CALL of *switch* statement, if the callee is a new cell, its name is added to the *call list*, which is the list of subcell names in the caller's cell definition. This list is used for a recursive scan in the hierarchy of the *.ca* files for a cell definition. *Cascan* also uses a binary search tree of cell nodes (*cell tree*) sorted by cell names to determine if a given cell is a new one. If there are cell calls in a cell definition, the associated files of those subcells are scanned and processed in a recursive manner.

6.2 Mapping Subsystem

The mapping subsystem converts the data stored in relational tables into internal data structures suitable for VLSI/CAD tools, following a script written in our mapping language. The mapping

subsystem translates a mapping script into INFORMIX_ESQL/C [INFO-87]. INFORMIX_ESQL/C is C with extentions for embedded SQL statements. Fig. 6-3 shows how a script is processed by the mapping subsystem of our data-structure builder.

A script file is preprocessed into two files: the header file that contains structure definitions and the function file that contains C-functions. The global variables used by the functions are defined at the beginning of the function file, and the global variables used by the application program are defined in the header file. Functions are produced in INFORMIX_ESQL/C. The header file is produced in C.

The record type definition of each *struct-statement* is translated into a corresponding C-structure definition and a *record_build()* function call that constructs the records of that type. The field initialization expression provided for a field definition is translated into a *field_init()* function call. The *index-statement* is translated into an index structure definition and an *index()* function call that constructs the indexing structure. The *link-statement* in a script is translated into a *link()* function call. The *link()* function links the constructed records by using the *sort-join* method based on the fields specified in a link statement.

The function *init()*, which calls each of the functions mentioned above, is provided at the beginning of an application program. At the beginning of a program execution, the generated code reads the design data from the database for initial C-structure building. Since the major portion of a typical VLSI/CAD program is CPU-intensive, the disk access overhead at the beginning of a program execution is not excessive.

6.2.1 Functions for Generating Structured View

In this section, we show and explain the functions that are generated from the script shown in Fig. 3-5 for *presim/rnl*. The header file *rnldb.h* containing structure definitions and global variables used by *presim/rnl* is shown in Fig. 6-4(a), and the global

variable definitions in the function file *rnldb.ec* are shown in Fig. 6-4(b).

init()

The code generated for the *init()* function is shown in Fig. 6-5. In the function *init()*, a target database is selected and then the cardinalities of tables *Node* and *Trans* are retrieved respectively into the global variables *num_node* and *num_trans* with *select* statements. The cardinality of a table is used in other functions. Function *init()* calls each of the functions described in the following for a structured view construction.

record_build()

Fig. 6-6 shows the code for the *record_build()* function. The *record_build()* function creates the dynamic array *nodeTable* whose elements are pointers to the *node* record type with *calloc()*, an UNIX system call. The size of the array is same as the cadinality of the *Node* table. The *record_build()* function then creates the *node* records, each of which is pointed by each array element. The array *transTable* and the *trans* records are created similarly. Fig. 6-7 shows the arrays *nodeTable* and *transTable*, and the corresponding *node* and *trans* records that are created by function *record_build()* for *presim/rnl*.

index()

Fig. 6-8 shows the code for the *index()* function. The *index()* function creates the hash table *hashnode* based on the *nname* field of the *node* record type as specified in the script. Each element of array *hashnode* points a *node* record. The variables *hashnode* and HASHSIZE are defined as global variables in the header file so that they can be accessed by *presim/rnl*.

The hash table *hashnode* on the basis of string values for the *nname* field of the *node* records is shown in Fig. 6-7.

field_init()

The generated code for the *field_init()* function is shown in Fig. 6-9. A field initialization expression provided in a field definition is translated into a *field_init()* function call. The expression for a field initialization may involve two kinds of special terms that need to access database tables. One, represented by *table-name* '.' *column-name*, is to get an attribute value in a table and the other, represented by an SQL statement enclosed by double quotes, is to get a value with the query. The SQL statement for a field initialization has parameters that indicate a search condition for a particular record.

In the script shown in Fig. 3-5(a), the fields *npot* and *nname* in the *node* record type definition are initialized respectively with expressions *Node.npot* and *Node.nname*. Also, the fields *twidth*, *tlength*, and *ttype* in the *trans* record type definition are initialized respectively with expressions *Trans.twidth* * LAMBDA, *Trans.tlength* * LAMBDA, and *Trans.ttype*. In these expressions, LAMBDA is 1.0. The field *ncap* of the *node* record type definition is initialized with an expression that involves two *select* (SQL) statements. Each *select* statement has a search condition: *Trans.gate = %Node.nname* for the first statement and *Cap.cnode = %Node.nname* for the second statement. For both statements, the search parameter is *Node.nname* that is preceded by a percent mark and it is same with the expression (term) for the *nname* field of the *node* record type definition.

The *field_init()* function first creates the dynamic arrays *nodenpot*, *nodenname*, *transtwidth*, *transtlength*, and *transttype*, and it then stores the attribute values for the above terms and parameter into the arrays. The *select* statements used in the *ncap* field initialization are prepared in the string forms to be used in the following *for-loop*. In the *for-loop*, these select statements are submitted to the DBMS as a character string with the parameters

replaced by a previously retrieved element of array *nodenname* for each *node* record, which will be interpreted as an SQL query and executed by the DBMS. Other terms in the expressions that need to access database tables are also replaced with corresponding array element.

link()

Fig. 6-10 shows the generated code for the *link()* function that links the constructed records by using the *sort-join* method. In the script shown in Fig. 3-5(b), there are three link statements that provide the linkages of three one-many relationships for the *gate*, *source*, and *drain* connections. In the *where-clauses*, the conditions for establishing the linkages are specified by *Node.nname = Trans.gate*, *Node.nname = Trans.source*, and *Node.nname = Trans.drain*. Therefore, we need the attribute values of the *nname* field of table *Node* and the *gate*, *source*, and *drain* fields of table *Trans* because the conditions are specified by the column names of the tables.

In the *link()* function, procedure *keytable()* retrieves these attribute values with *select* statements and stores them in the dynamic arrays *gatetrans*, *sourcetrans*, *draintrans*, and *nnamenode*. These arrays have two fields *recid* and *reckey* as declared in the header file. The *recid* field stores tuple identification numbers and the *reckey* field stores the values of an attribute used in a condition. In the *keytable()* procedure, procedure *quickSort()* sorts these dynamic arrays according to the *reckey* values and performs the *sort* part of the *sort-join* method.

The second procedure *one_many1()* in the *link()* function establishes linkages for the *gate* connection between the *node* records and the *trans* records. Procedures *one_many2()* and *one_many3()* establish linkages respectively for the *source* and *drain* connections. These procedures perform the *join* part of the *sort-join* method.

We now show how the *sort-join* method provides linkages by using a simple example. Fig. 6-11 shows the *sort-join* method for providing linkages of one-many relationship for the *gate* connection between the *node* records and the *trans* records. Fig. 6-11(a) shows arrays *nnamenode* and *gatetrans*. The *reckey* field of array *nnamenode* stores node names and that of array *gatetrans* stores the node names to which the gates of transistors are connected. Procedure *quickSort()* sorts these arrays and the resultant arrays are shown in Fig. 6-11(b).

In Fig. 6-11(c), the *indexes* of two arrays *nodeTable* and *transTable* that are created by the *record_build()* function serve as tuple identification numbers and correspond with the *recid* fields of the arrays *nnamenode* and *gatetrans*. In Fig. 6-11(b), for the first element in array *nnamenode* whose *reckey* value is "a" and *recid* value is "1", there are three elements that match with the "a" in array *gatetrans* and those three elements have *recid* values "2", "3", and "6". Therefore, the *one_many1()* function provides, beginning at the *ngate* field of the *node* record pointed by the element whose *index* value is "1" in array *nodeTable*, a pointer chain that threads through the *glink* fields of the *trans* records pointed by the elements whose *index* values are "2", "3", and "6" in array *transTable*. Procedure *one_many1()* then performs the same method for the second element in array *nnamenode*, scanning for matching elements in array *gatetrans* from the *index* value "3". The *one_many1()* procedure scans array *nnamenode*, scanning for matching elements in array *gatetrans* until either of two arrays is exhausted, providing linkages of one-many relationship. Fig. 6-11(c) shows the part of linkages established by procedure *one_many1()* for the *gate* connection.

If the condition of a link statement is specified by field names of two record types, then the records of each type themselves are sorted and scanned to establish linkages between the record types

for one-one or one-many relationships.

Procedure *memspace()* shown in Fig. 6-12, allocates a bulk memory space and returns the amount of memory that is requested by the functions described above. As there are many memory allocation requests throughout a structured view construction process, this procedure saves execution time of an application program.

6.2.2 Translation

In this section, we first discuss the symbol table and abstract syntax tree used by the translator of our mapping subsystem and then present the algorithms for the major modules that generate the functions to construct a structured view.

The symbol table in Fig. 6-13(a) describes the data structure that the translator constructs from the mapping script for *presim/rnl*. A node of a symbol table contains, among other informations, source text to make it easier to generate a target code.

In the symbol table, nodes for the record type definitions are linked in linear list and there is a head pointer *records* addressing the first node of the list. A node in the list has the field *fields*, which is a head pointer pointing the first node of the list for field definitions in a particular record type definition. If a field is initialized with an expression, then the field *init* in a field definition node points a tree that represents the expression. As shown in the script, the capacitance field *ncap* of the *node* records is initialized with an expression involving SQL statements. In Fig. 6-13(a), this expression is represented with the tree pointed by the *init* field of the *ncap* field definition node and the tree node contains the source text for each term in the expression.

If a record type is declared with the *typedef* construct, then a node containing the record name and the pointer name for that record type is added to the list pointed by the head pointer *typedefs*

as shown in Fig. 6-13(b).

Fig. 6-14 shows the abstract syntax tree that the translator constructs from the link statements of the script for *presim/rnl.* A tree node also contains source text for easy code generation. The pointer *linkages* is the head pointer addressing the first element of a linked list and the pointer *linktree* of each node in the list points a tree representing the *link-statement* construct.

Information such as *index-kind* and *index-name* on an *index-statement* construct is stored in global variables for code generation phase.

The major modules in code generation are *genInit()*, *genRecord_build()*, *genIndex()*, *genField_init()*, and *genLink()* which generate respectively the functions *init()*, *record_build()*, *index()*, *field_init()*, and *link()*. Fig. 6-15 through Fig. 6-19 show the algorithms for the major modules.

```
main () {

        Open .sim input file
        Create database tables

        input (.sim file)
        Close .sim file
}

input (file) {

        while (not end-of-file) {
            Read a line
            if (blank line)
                Skip
            if (first line and first character is '|')
                Set the unit and technology
            switch (first character of the line) {
                case '@' :    Open redirected file
                              input (redirected file)
                              Close redirected file
                case 'e' :    newtrans (n-channel)
                case 'i' :    newtrans (n-channel zero threshold)
                case 'p' :    newtrans (p-channel)
                case 'd' :    newtrans (depletion)
                case 'l' :    newtrans (low-power depletion)
                case 'c' :    Insert capacitance and its parameters into Cap table
            }
}

newtrans (type) {

        Insert the transistor and its parameters into Trans table

        if (the nodes associated with gate, source, and drain are new)
            Insert those nodes and their parameters into Node table
}
```

Fig. 6-1 Algorithm *simscan*.

```
main () {

        Set options and technology
        Create database tables

        Insert top level cell into Cell table

        load (top-level cell)
}


load (cell) {

        Open .ca file containing the cell definition
        Initialize cell definition variables

        while (not end-of-file) {
            Read a command line
            switch (command class) {
                case BOX :    Insert box and its parameters into Box table
                case CALL :   Insert cell call information and transformation matrix
                              into Call  table
                              if (there is no entry for callee in Cell table) {
                                  Add the callee and its parameters into Cell table
                                  Add callee to the call list
                /* call list keeps subcell names in the caller's cell definition */
                              }
                case END :    Restore cell definition variables
                case LABEL :  Insert label and its parameters into Label table
                case LAYER :  Set the current layer index
            }
        }

        Close .ca file
        Add the cell name to the cell tree
        /* cell tree is a binary search tree of cell nodes sorted by cell names */

        for (each callee in the call list)
            if (the callee is not found in cell tree)
                load (callee)
}
```

Fig. 6-2 Algorithm *cascan*.

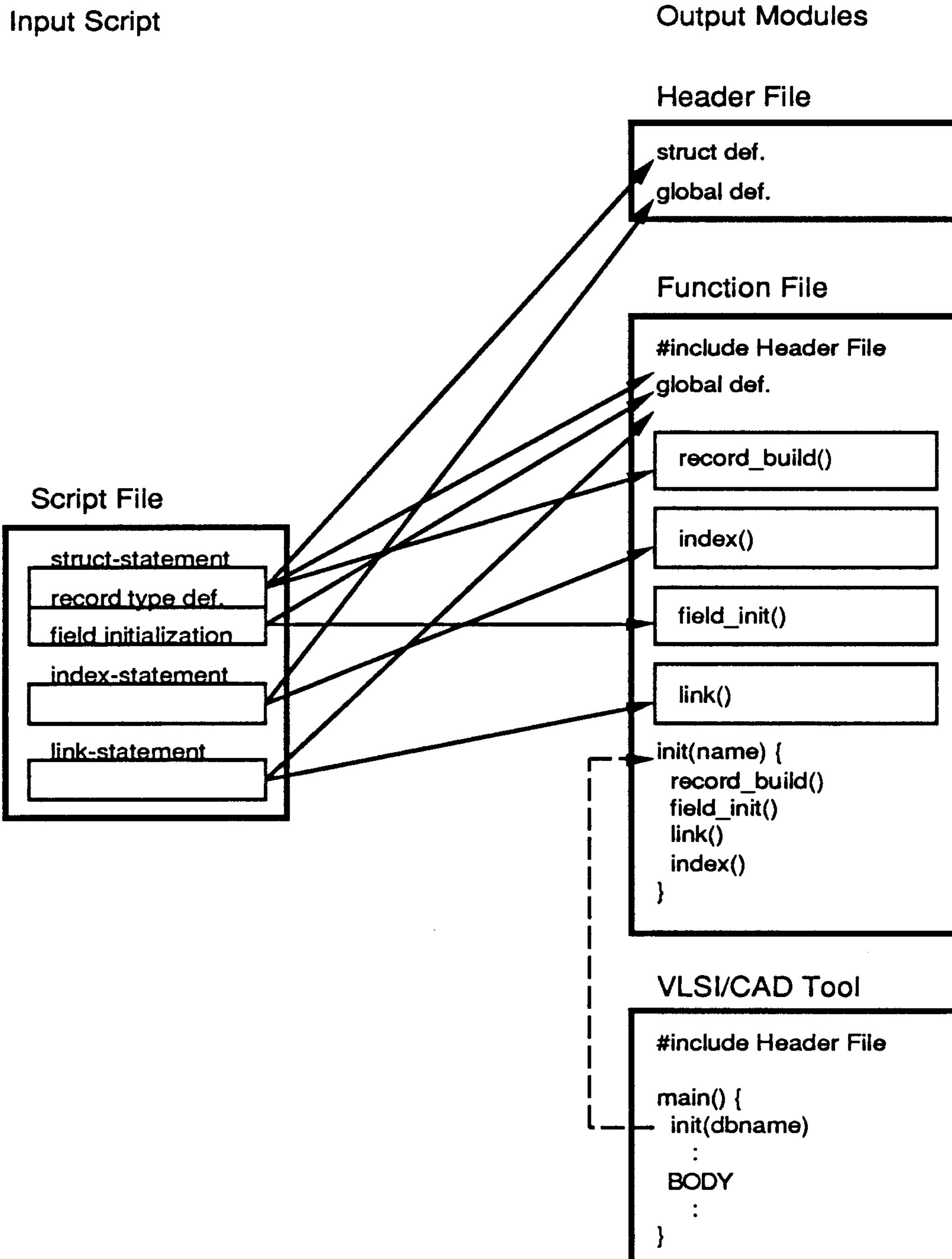


Fig. 6-3 Mapping subsystem strategy.

```
typedef struct trans *tptr;
typedef struct node *nptr;
typedef struct key *keyptr;

struct node {
        nptr nlink;
        tptr ngate;
        tptr nsource;
        tptr ndrain;
        nptr hnext;
        float ncap;
        float vlow;
        float vhigh;
        short tplh;
        short tphl;
        long ndelay;
        long ctime;
        short npot;
        short nflags;
        char * nname;
        char statestatus;
};

struct trans {
        nptr gate;
        nptr source;
        nptr drain;
        tptr glink;
        tptr slink;
        tptr dlink;
        float twidth;
        float tlength;
        float tnumber;
        int ttype;
};


struct key {
        int recid;
        char *reckey;
};

#define HASHSIZE 731
nptr hashnode[HASHSIZE];
```

(a) Header file.

Fig. 6-4 Header file and global definitions for *presim/rnl.*

```
#include <stdio.h>
#include "rnldb.h"

$include sqlca;

$int num_node;
$int num_trans;
nptr *nodeTable;
tptr *transTable;
keyptr gatetrans;
keyptr sourcetrans;
keyptr draintrans;
keyptr nnamenode;
short *nodenpot;
char * *nodenname;
float *transtwidth;
float *transtlength;
int *transttype;

char *memspace();
char *storage;
#define NFREE 4096
int nfree = 0;
```

(b) Global definitions.

Fig. 6-4 Continued.

```
init(name)
char *name;
{
$       char *dbname;

        dbname = name;
$       database $dbname;

$       select count(*)
        into $num_node
        from node;
$       select count(*)
        into $num_trans
        from trans;

        record_build();
        field_init();
        link();
        indexx();
}
```

Fig. 6-5 Generated code for function *init()*.

```
record_build()
{       register int i;

        nodeTable = (nptr *) calloc(num_node, sizeof(nptr));
        for(i=0; i<num_node; i++)
                nodeTable[i] = (nptr) memspace(sizeof(struct node));

        transTable = (tptr *) calloc(num_trans, sizeof(tptr));
        for(i=0; i<num_trans; i++)
                transTable[i] = (tptr) memspace(sizeof(struct trans));
}
```

Fig. 6-6 Generated code for function *record_build()*.

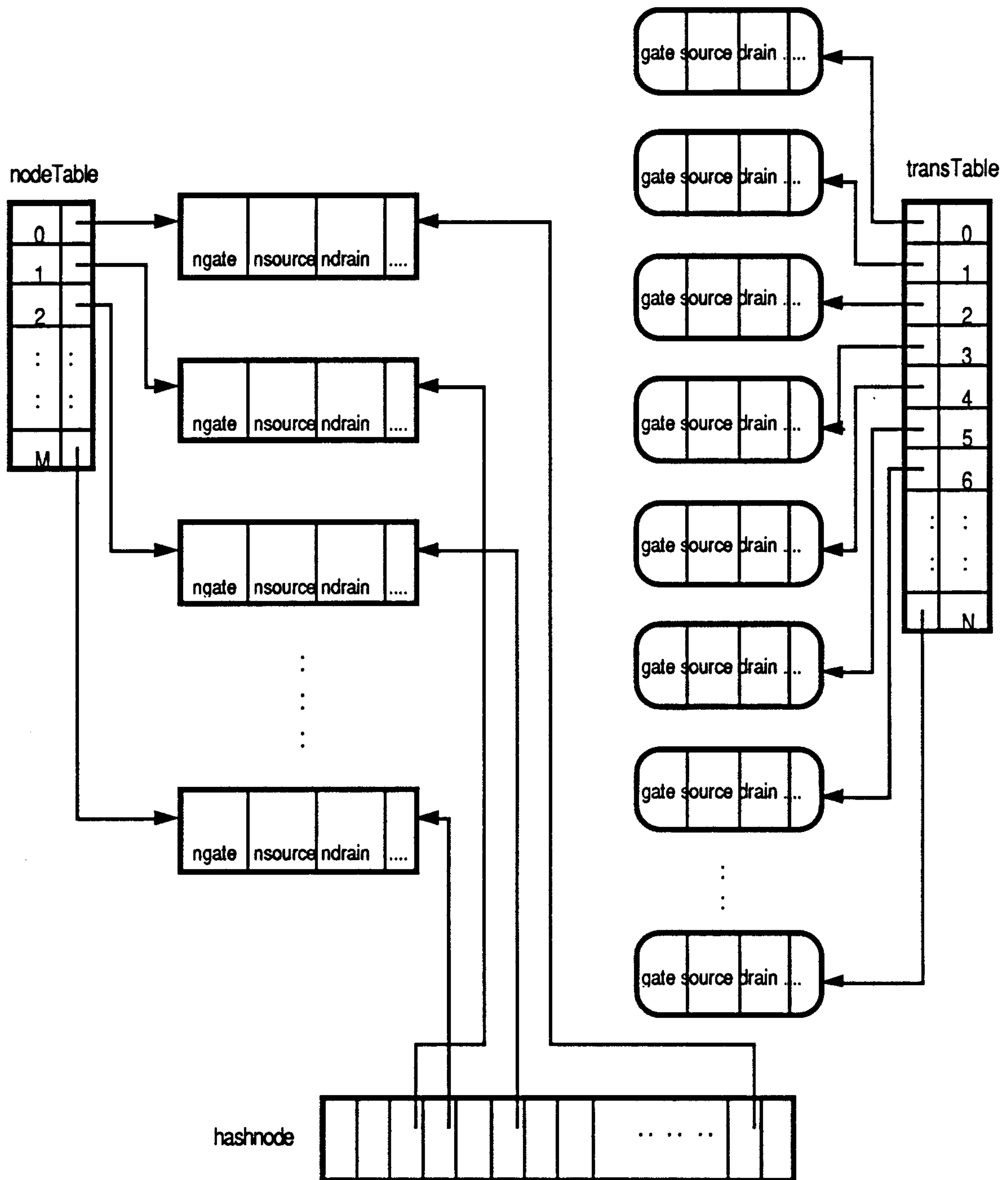


Fig. 6-7 Data structure constructed by functions *record_build()* and *index()*.

```
indexx()
{       register int i,f;

        for(i=0; i<HASHSIZE; i++)
                hashnode[i] = 0;

        for(i=0; i<num_node; i++) {
                f = hashcode(nodeTable[i]->nname);
                nodeTable[i]->hnext = hashnode[f];
                hashnode[f] = nodeTable[i];
        }
}

hashcode(name)
char *name;
{       int i=0;

        while(*name) i = (i*10 + *name++ - '0') % HASHSIZE;
        return(i<0 ? i+HASHSIZE : i);
}
```

Fig. 6-8 Generated code for function *index()*.

```
field_init()
    {       register int i;
    $       short hnpot;
    $       string hnname[20];
    $       float htwidth;
    $       float htlength;
    $       int httype;
    $       float ncap_0;
    $       char qstrncap_0[80];
    $       char * pncap_0;
    $       float ncap_1;
    $       char qstrncap_1[80];
    $       char * pncap_1;

            nodenpot = (short *) calloc(num_node, sizeof(short));
            nodenname = (char * *) calloc(num_node, sizeof(char *));
            transtwidth = (float *) calloc(num_trans, sizeof(float));
            transtlength = (float *) calloc(num_trans, sizeof(float));
            transttype = (int *) calloc(num_trans, sizeof(int));

    $       declare node_cur cursor for
            select npot, nname
            from node;

    $       declare trans_cur cursor for
            select twidth, tlength, ttype
            from trans;

    $       open node_cur;
            for(i=0; i<num_node; i++) {
    $               fetch node_cur into $hnpot, $hnname;
                    nodenpot[i] = hnpot;
                    nodenname[i] = (char *) memspace(strlen(hnname) + 1);
                    stcopy(hnname, nodenname[i]);
            }
    $       close node_cur;

    $       open trans_cur;
            for(i=0; i<num_trans; i++) {
    $               fetch trans_cur into $htwidth, $htlength, $httype;
                    transtwidth[i] = htwidth;
                    transtlength[i] = htlength;
                    transttype[i] = httype;
            }
    $       close trans_cur;
```

Fig. 6-9 Generated code for function *field_init()*.

```
        strcpy(qstrncap_0, "select sum(twidth * tlength) from Trans
                              where Trans.gate = ? ");
$       prepare qidncap_0 from $qstrncap_0;
$       declare curncap_0 cursor for qidncap_0;
        strcpy(qstrncap_1, "select sum(cval) from Cap where Cap.cnode = ? ");
$       prepare qidncap_1 from $qstrncap_1;
$       declare curncap_1 cursor for qidncap_1;

        for (i=0; i<num_node; i++) {
                pncap_0 = nodenname[i];
$               open curncap_0 using $pncap_0;
$               fetch curncap_0 into $ncap_0;
                if(sqlca.sqlwarn.sqlwarn1 == 'W')
                        ncap_0 = 0.0;
$               close curncap_0;

                pncap_1 = nodenname[i];
$               open curncap_1 using $pncap_1;
$               fetch curncap_1 into $ncap_1;
                if(sqlca.sqlwarn.sqlwarn1 == 'W')
                        ncap_1 = 0.0;
$               close curncap_1;

                nodeTable[i]->ncap =  0.000690 * ncap_0 + ncap_1;
                nodeTable[i]->vlow =  0.3;
                nodeTable[i]->vhigh =  0.8;
                nodeTable[i]->npot =  nodenpot[i];
                nodeTable[i]->nname =  nodenname[i];
        }

        for (i=0; i<num_trans; i++) {
                transTable[i]->twidth =  transtwidth[i] * 1.0;
                transTable[i]->tlength =  transtlength[i] * 1.0;
                transTable[i]->ttype =  transttype[i];
        }
}
```

Fig. 6-9 Continued.

```
link()
{
        keyTables();
        one_many1();
        one_many2();
        one_many3();
}

one_many1()
{ int fromindex=0, toindex=0;
  int caseno, memptr=1;

  while(fromindex < num_node && toindex < num_trans) {
    if(strcmp(nnamenode[fromindex].reckey, gatetrans[toindex].reckey) > 0)
      caseno = 1;
    else if(strcmp(nnamenode[fromindex].reckey, gatetrans[toindex].reckey) < 0)
      caseno = 2;
    else
      caseno = 3;

    switch(caseno) {
      case 1 : ++toindex; break;
      case 2 : ++fromindex; memptr = 1; break;
      case 3 :
        if(memptr) {
          nodeTable[nnamenode[fromindex].recid]->ngate =
transTable[gatetrans[toindex].recid];
          memptr = 0;
        }
        else
          transTable[gatetrans[toindex-1].recid]->glink = transTable[gatetrans[toindex].recid];

        transTable[gatetrans[toindex].recid]->gate = nodeTable[nnamenode[fromindex].recid];
        ++toindex; break;
      default : fprintf(stderr, "wrong case number");
    }
  }
}
```

Fig. 6-10 Generated code for function *link()*.

```
one_many2()
{ int fromindex=0, toindex=0;
 int caseno, memptr=1;

 while(fromindex < num_node && toindex < num_trans) {
  if(strcmp(nnamenode[fromindex].reckey, sourcetrans[toindex].reckey) > 0)
   caseno = 1;
  else if(strcmp(nnamenode[fromindex].reckey, sourcetrans[toindex].reckey) < 0)
   caseno = 2;
  else
   caseno = 3;

  switch(caseno) {
   case 1 : ++toindex; break;
   case 2 : ++fromindex; memptr = 1; break;
   case 3 :
    if(memptr) {
     nodeTable[nnamenode[fromindex].recid]->nsource =
                                 transTable[sourcetrans[toindex].recid];
     memptr = 0;
    }
    else
     transTable[sourcetrans[toindex-1].recid]->slink =
                                 transTable[sourcetrans[toindex].recid];

    transTable[sourcetrans[toindex].recid]->source =
                                 nodeTable[nnamenode[fromindex].recid];
    ++toindex; break;
   default : fprintf(stderr, "wrong case number");
  }
 }
}

one_many3()
{ int fromindex=0, toindex=0;
 int caseno, memptr=1;

 while(fromindex < num_node && toindex < num_trans) {
  if(strcmp(nnamenode[fromindex].reckey, draintrans[toindex].reckey) > 0)
   caseno = 1;
  else if(strcmp(nnamenode[fromindex].reckey, draintrans[toindex].reckey) < 0)
   caseno = 2;
  else
   caseno = 3;
```

Fig. 6-10 Continued.

```
    switch(caseno) {
      case 1 : ++toindex; break;
      case 2 : ++fromindex; memptr = 1; break;
      case 3 :
        if(memptr) {
          nodeTable[nnamenode[fromindex].recid]->ndrain =
                                      transTable[draintrans[toindex].recid];
          memptr = 0;
        }
        else
          transTable[draintrans[toindex-1].recid]->dlink =
                                      transTable[draintrans[toindex].recid];

        transTable[draintrans[toindex].recid]->drain = nodeTable[nnamenode[fromindex].recid];
        ++toindex; break;
      default : fprintf(stderr, "wrong case number");
    }
  }
}

keyTables()
{       register int i;
$       string kgatetrans[20];
$       string ksourcetrans[20];
$       string kdraintrans[20];
$       string knnamenode[20];

        gatetrans = (keyptr) calloc(num_trans, sizeof(struct key));
        sourcetrans = (keyptr) calloc(num_trans, sizeof(struct key));
        draintrans = (keyptr) calloc(num_trans, sizeof(struct key));
        nnamenode = (keyptr) calloc(num_node, sizeof(struct key));

$       declare ktrans_cur cursor for
        select gate, source, drain
        from trans;

$       declare knode_cur cursor for
        select nname
        from node;
```

Fig. 6-10 Continued.

```
$       open ktrans_cur;
        for(i=0; i<num_trans; i++) {
$               fetch ktrans_cur into $kgatetrans, $ksourcetrans, $kdraintrans;
                gatetrans[i].recid = i;
                gatetrans[i].reckey = (char *) memspace(strlen(kgatetrans) + 1);
                stcopy(kgatetrans, gatetrans[i].reckey);
                sourcetrans[i].recid = i;
                sourcetrans[i].reckey = (char *) memspace(strlen(ksourcetrans) + 1);
                stcopy(ksourcetrans, sourcetrans[i].reckey);
                draintrans[i].recid = i;
                draintrans[i].reckey = (char *) memspace(strlen(kdraintrans) + 1);
                stcopy(kdraintrans, draintrans[i].reckey);
        }
$       close ktrans_cur;

$       open knode_cur;
        for(i=0; i<num_node; i++) {
$               fetch knode_cur into $knnamenode;
                nnamenode[i].recid = i;
                nnamenode[i].reckey = (char *) memspace(strlen(knnamenode) + 1);
                stcopy(knnamenode, nnamenode[i].reckey);
        }
$       close knode_cur;


        quickSort(gatetrans, num_trans);
        quickSort(sourcetrans, num_trans);
        quickSort(draintrans, num_trans);
        quickSort(nnamenode, num_node);
}

quickSort(a, n)
struct key a[];
int n;
{       int k;
        char *pivot;

        if(find_pivot(a, n, &pivot) != 0) {
                k = partition(a, n, pivot);
                quickSort(a, k);
                quickSort(a+k, n-k);
        }
}
```

Fig. 6-10 Continued.

```
find_pivot(a, n, pivot_ptr)
struct key a[];
int n;
char **pivot_ptr;
{       int i;

        for(i=1; i<n; i++)
                if(strcmp(a[0].reckey, a[i].reckey) != 0) {
                        *pivot_ptr = (strcmp(a[0].reckey, a[i].reckey) > 0) ? a[0].reckey :
a[i].reckey;
                        return(1);
                }
        return(0);
}

partition(a, n, pivot)
struct key a[];
int n;
char *pivot;
{       int i=0, j=n-1;

        while(i <= j) {
                while(strcmp(a[i].reckey, pivot) < 0)
                        i++;
                while(strcmp(a[j].reckey, pivot) >= 0)
                        j--;
                if(i < j)
                        swap(&a[i++], &a[j--]);
        }
        return(i);
}

swap(p, q)
keyptr p, q;
{       struct key temp;

        temp = *p;
        *p = *q;
        *q = temp;
}
```

Fig. 6-10 Continued.

nnamenode

	recid	reckey
0	0	c
1	1	a
2	2	b
:	:	:
M	M	k

gatetrans

	recid	reckey
0	0	b
1	1	c
2	2	a
3	3	a
4	4	c
5	5	b
6	6	a
:	:	:
N	N	p

(a) Arrays *nnamenode* and *gatetrans*.

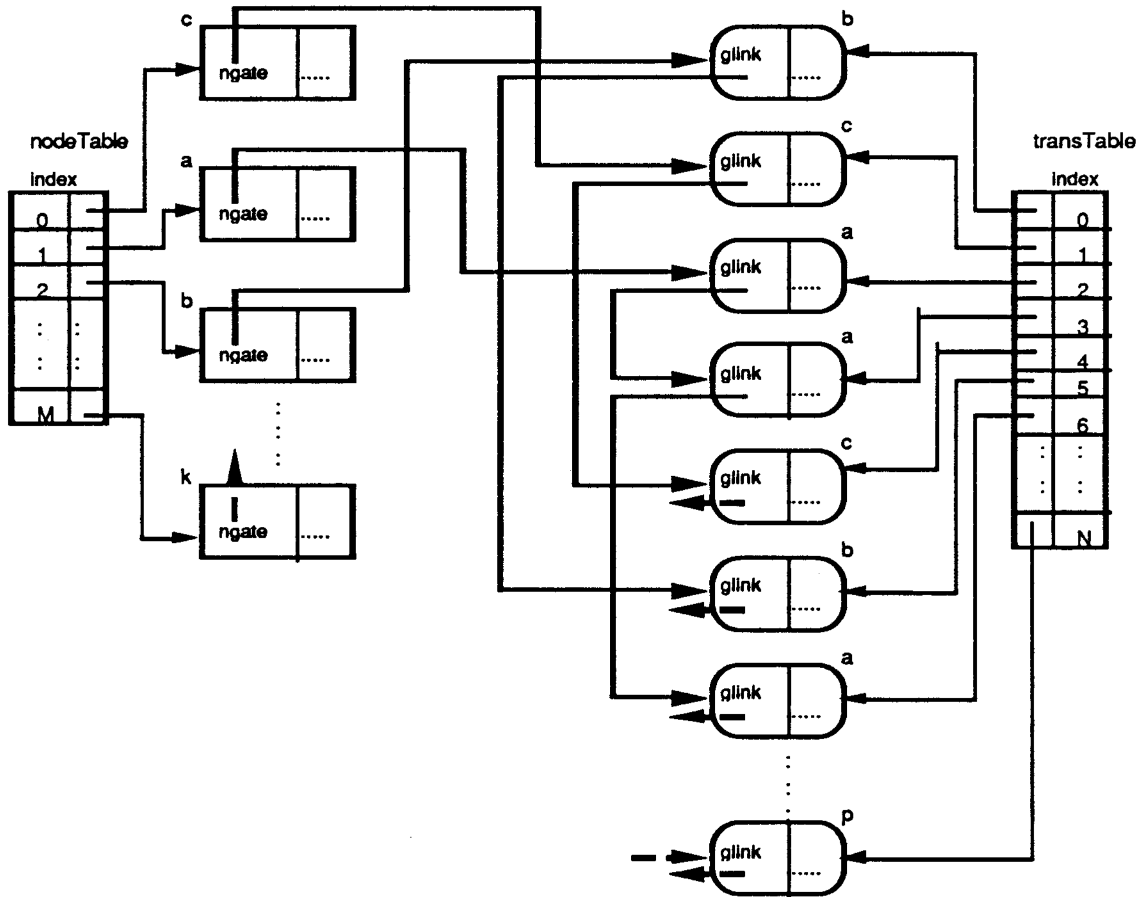
nnamenode

	recid	reckey
0	1	a
1	2	b
2	0	c
:	:	:
M	m	z

gatetrans

	recid	reckey
0	2	a
1	3	a
2	6	a
3	0	b
4	5	b
5	1	c
6	4	c
:	:	:
N	n	z

(b) Sorted arrays *nnamenode* and *gatetrans*.



(c) Part of linkages for the gate connection.

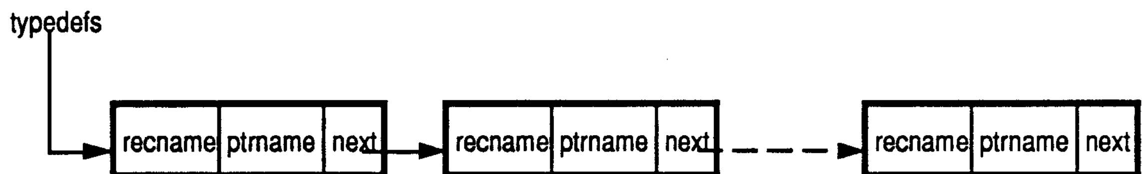
Fig. 6-11 Sort-join method for providing linkages.

```
char *memspace(n)
int n;
{               char *p;

                n = (n + 3) & ~3;
                if(n > nfree) {
                    if(n > NFREE) {
                            if((p = (char *) malloc(n)) == NULL) {
                                    fprintf(stderr, "No more room");
                                    exit(0);
                            }
                            return(p);
                    }
                    if((storage=(char *) malloc(NFREE)) == NULL) {
                            fprintf(stderr, "No more room");
                            exit(0);
                    }
                    nfree = NFREE;
                }
                nfree -= n;
                p = storage;
                storage += n;
                return(p);
}
```

Fig. 6-12 Procedure *memspace()*.

(a) Symbol table.



(b) Linked list for *typedef* construct.

Fig. 6-13 Symbol table used by translator.

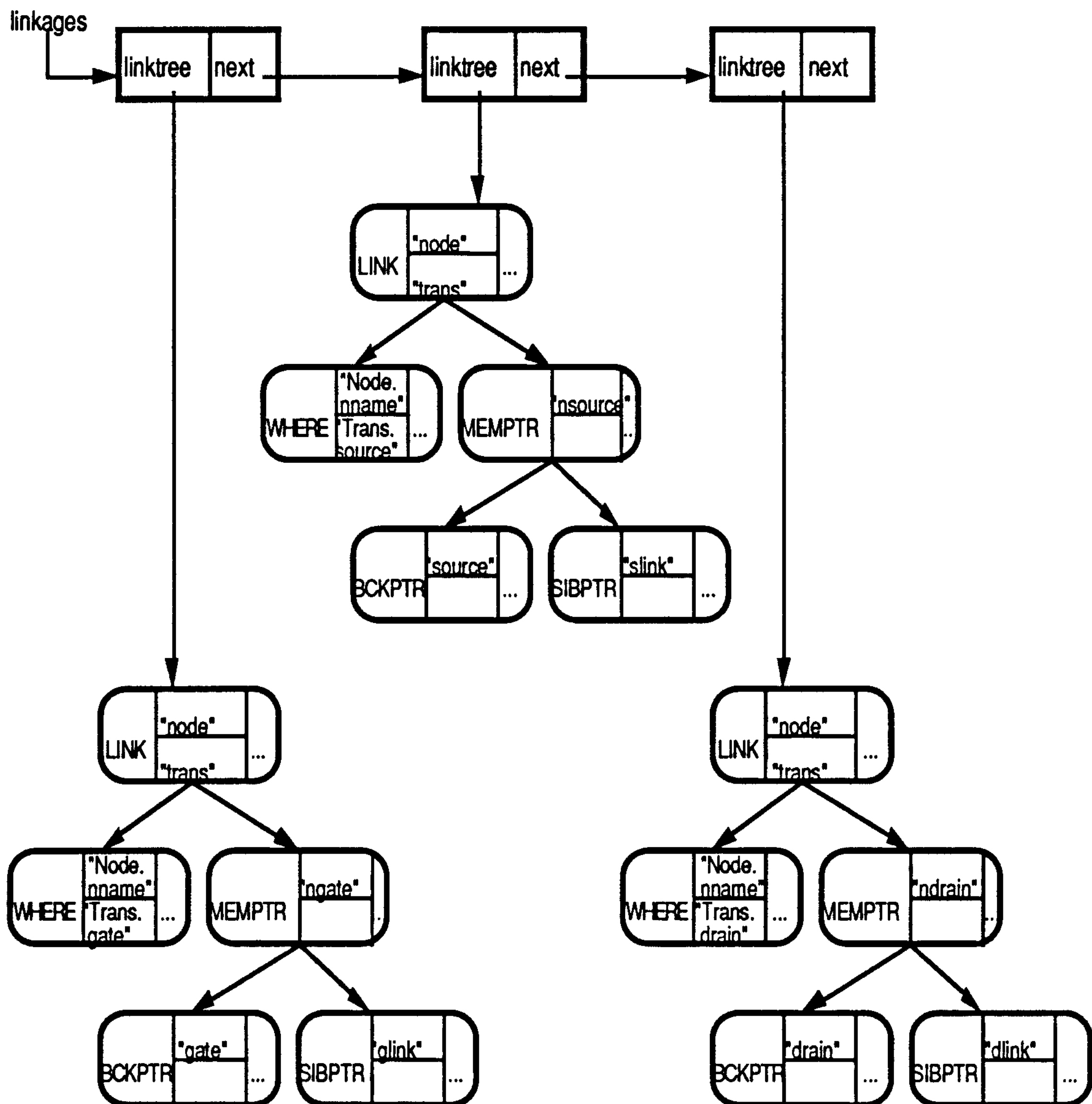


Fig. 6-14 Abstract syntax tree for link-statements.

```
genInit () {
    Declare global variables

    Generate code to select a database

    while  there are more record definition nodes in the symbol table
        if  record type is declared with struct-statement
            Get the cardinality of the table

    Generate code that calls functions for structured view construction
}
```

Fig. 6-15 Algorithm for module *genInit()*.

```
genRecord_build () {

    while there are more record definition nodes in the symbol table {
        if record type is declared with typedef construct
            Declare the record type with typedef construct in header file
        else
            Declare record type in header file
        while there are more field definition nodes for the record type
            Declare field definition in header file
    }

    Declare global variables in function file

    while there are more record definition nodes in the symbol table {
        if record definition is in struct-statement {
            Generate code that creates dynamic pointer arrays and the records
            pointed by  each of  the array element
        }
    }
}
```

Fig. 6-16 Algorithm for module *genRecord_build()*.

```
genIndex () {

    Get the data type for the indexed field from symbol table

    switch (index type) {
        case STRHASH :    Declare global variables for STRHASH in header file
                          Generate code for string hash index in function file
        case NUMHASH :    Declare global variables for NUMHASH in header file
                          Generate code for number hash index in function file
        case BSTREE :     Declare global variables for BSTREE in header file
                          Generate code for binary search tree in function file
    }
}
```

Fig. 6-17 Algorithm for module *genIndex()*.

```
genField_init () {

   while there is no entry in the term list
   /* term list keeps terms that need to access database tables in expressions */
      Declare host variables for the term

   while there is no entry in the SQL-string list
   /* SQL-string list keeps terms that are SQL statements in expressions */
      Declare necessary string variables for the term

   while there is no entry in the term list
      Generate code to create dynamic arrays
   /* these arrays will store attribute values associated with the term */

   while there is no entry in the term list {
      Declare select statement for terms in term list
      Generate code to fetch term values and store them in the dynamic arrays
   }

   while there is no entry in the SQL-string list
      Prepare and declare parameterized select statement

   while there are more record definition nodes in the symbol table {
      while there are more field definition nodes in the field list of the record node {
         if field initialization flag is set {
            if expression involves SQL-string
               Generate code to execute the SQL statement and store it in the
               variable
               GenExpress (abstract syntax tree for the expression)
            }
            else
               GenExpress (abstract syntax tree for the expression)
         }
      }
   }
}


genExpress (tree) {

   if tree is not NULL {
      genExpress (tree->leftchild)
      Print term string
      genExpress (tree->rightchild)
   }
}
```

Fig. 6-18 Algorithm for module *genField_init()*.

```
genLink () {

   while there is no entry in key list {
   /* key list keeps the conditions of where clauses in link statements */
       Declare host variables for storing attribute values of the conditions
       Create dynamic arrays to store those values
   }

   while there is no entry in key list {
       Declare select statement to retrieve attribute values of the conditions
       Generate code to fetch key values and store them in the dynamic arrays
   }

   Generate code to sort the dynamic arrays (quick sort) based on key strings

   while there is no entry in key list {
       switch (mapping kind) {
           case ONEONE :     Generate function that establish linkages for one-one
                             relationship (sort-join) using the dynamic arrays
           case ONEMANY :    Generate function that establish linkages for
                             one-many relationship (sort-join) using the dynamic
                             arrays
           case RONEONE :    Generate function that establish linkages for one-one
                             relationship (sort-join)
           case RONEMANY :   Generate function that establish linkages for
                             one-many relationship (sort-join)
       }
   }
}
```

Fig. 6-19 Algorithm for module *genLink()*.

CHAPTER VII

PERFORMANCE MEASUREMENT

We reimplemented *presim* and *vic* according to the strategy discussed in the previous chapter and analyzed their performance. We actually provided for each program (*presim* or *vic*) the following versions. The least efficient version (*dbpresim* or *dbvic*) used SQL statements for record construction, field initialization and record linking. The most efficient version (*dbpresim2* or *dbvic2*) used SQL statements only for record construction and record linking, and it performed complex field initialization whose expression involved SQL statements by user-provided C functions. Each of the other versions measured the CPU time required for record construction, field initialization, record linking, or index creation. Fig. 7-1 shows the performance measurement results for *presim* and *dbpresim*.

We used as test cases 16-bit shift register arrays of 1 to 16 stages. We plotted the CPU times required for structure building for different numbers of total records (transistor and node records) constructed for register arrays. The programs were run on the Sequent Balance 21000.

Presim is the original file-based program. *Dbpresim* is our least efficient version of *presim*. As the number of records increased, the CPU times of *presim* and *dbpresim* increased almost linearly. Since the CPU time used by *dbpresim* was about 17 times greater than that of *presim*, each function of *dbpresim* was analyzed to determine the major cause of this performance deterioration, with the result also shown in Fig. 7-2.

Each of the following programs measured the times required by record construction, field initialization, record linking and index creation. *Record_build* constructs the records of the *node* and *trans*

record types. Field initialization by the expressions provided in field definitions is performed by *field_init*. *Link* provides pointers among the constructed records. *Index* constructs the indexing structure (STRHASH).

As can be seen from the figure, we discovered that the CPU time of the *field_init* function took about 78% of that of *dbpresim*, and hence was the major source of the overhead. This is because the capacitance field *ncap* of each node record is initialized with an SQL statement. For example, in the case for 11223 records, which consisted of 4183 node records and 7040 transistor records, 4183 SQL statements were executed for this purpose.

In order to eliminate this excessive overhead, we modified *dbpresim* by replacing the SQL statement for the initialization of the *ncap* fields by a C procedure. This could be done easily because all other fields whose values were used by this procedure were already initialized by the *record_build* function. We called the modified version *dbpresim2*. This change improved the performance by a factor of about 3. Consequently, the CPU time of *dbpresim2* became only about 5 times greater than that of *presim*.

Most of the remaining overhead is in linking among constructed records by the *link* function. The CPU time required for record construction is comparable to that of the original *presim*. The overhead due to the index creation is negligible. Fig. 7-3 shows the performance comparison of *presim*, *dbpresim* and *dbpresim2*.

We also performed similar measurements for *vic*. The results are shown in Fig. 7-4, Fig. 7-5, and Fig. 7-6. The CPU time of *dbvic*, which is our least efficient version, is about 13 times greater than that of *vic*. Our most efficient version, *dbvic2*, is about 3 time slower than *vic*. The measured data for the *vic* programs were similar to those for the *presim* programs.

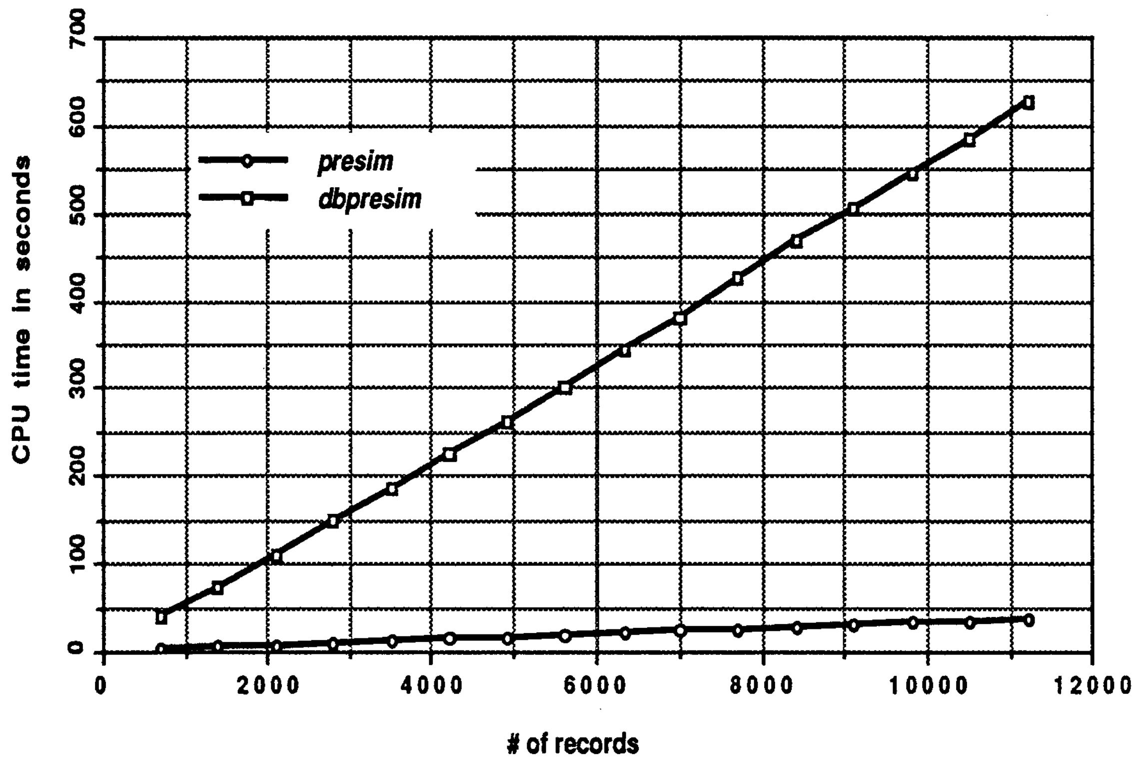


Fig. 7-1 Performance comparison of *presim* and *dbpresim*.

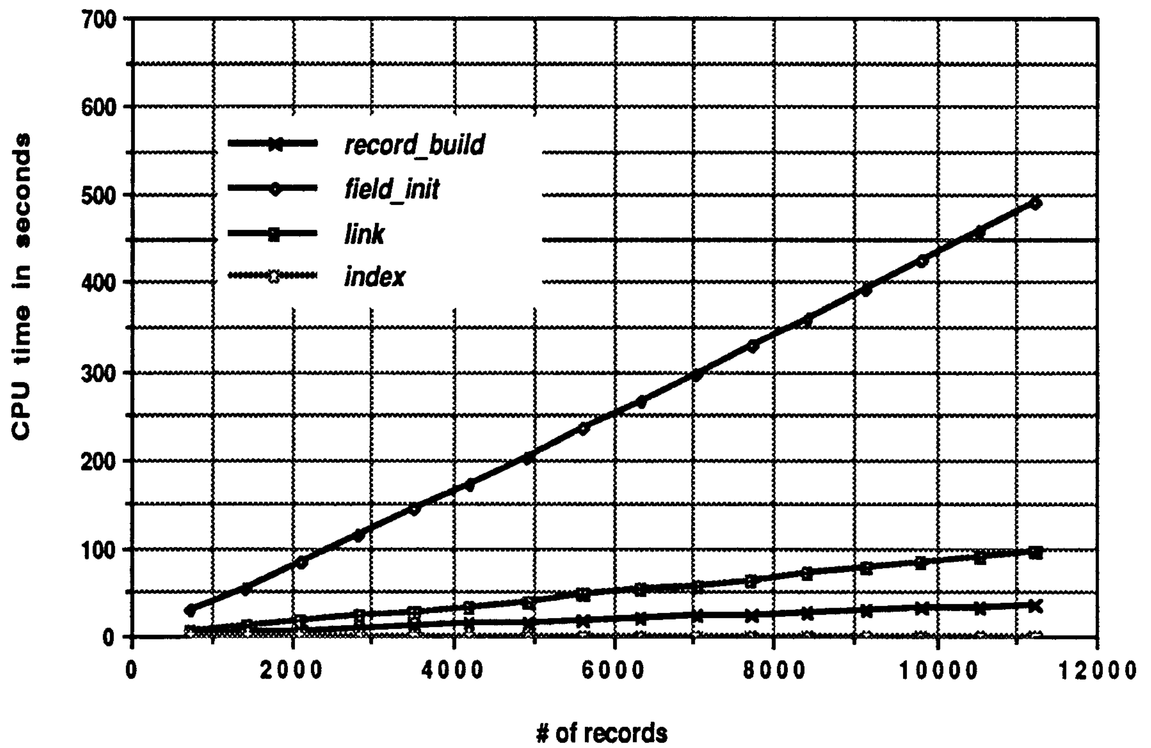


Fig. 7-2 Performance comparison of 4 modules of *dbpresim.*

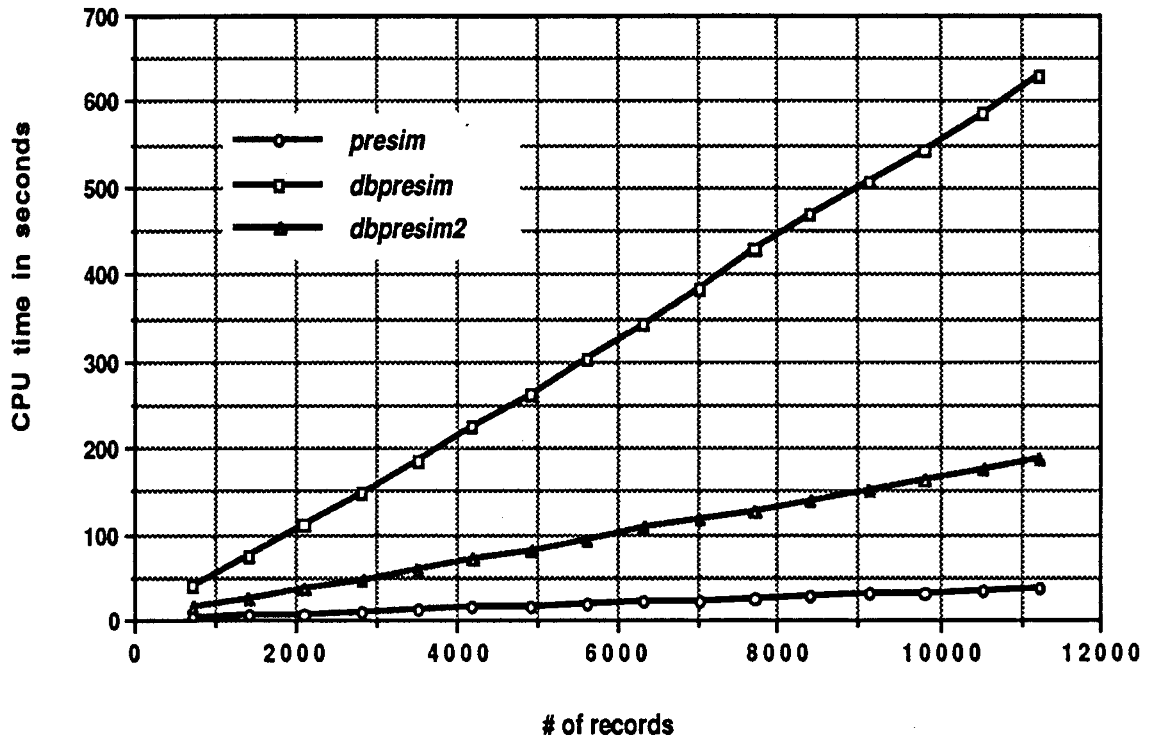


Fig. 7-3 Performance comparison of *presim*, *dbpresim*, and *dbpresim2*.

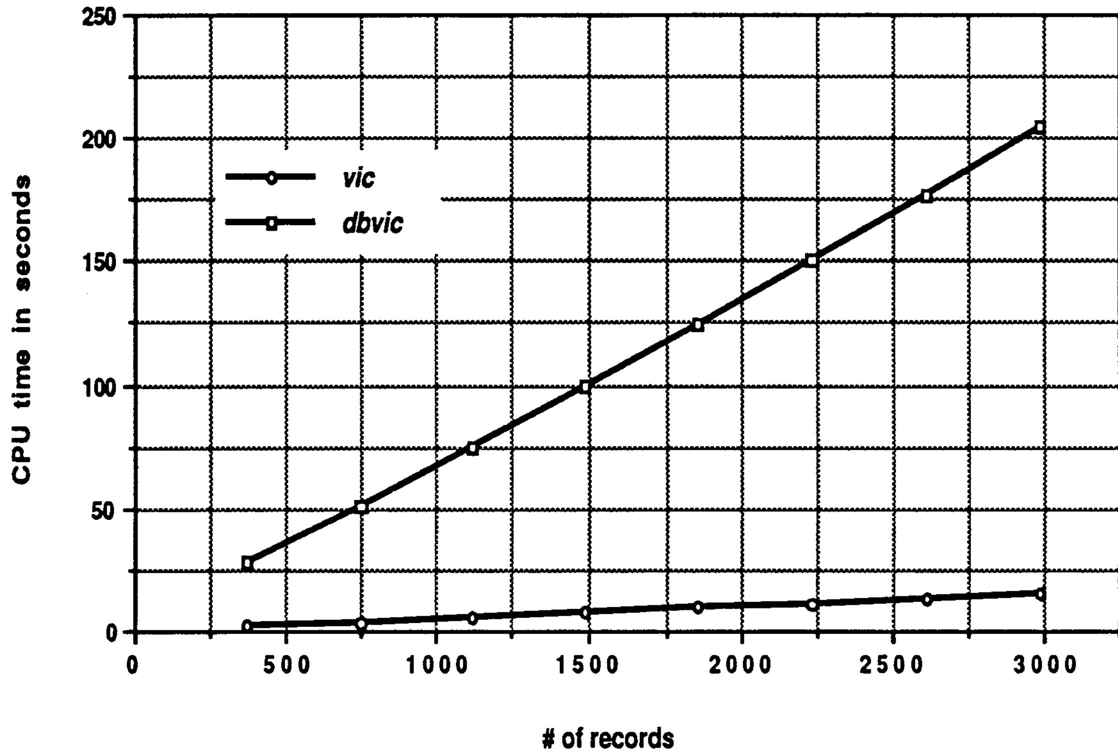


Fig. 7-4 Performance comparison of *vic* and *dbvic*.

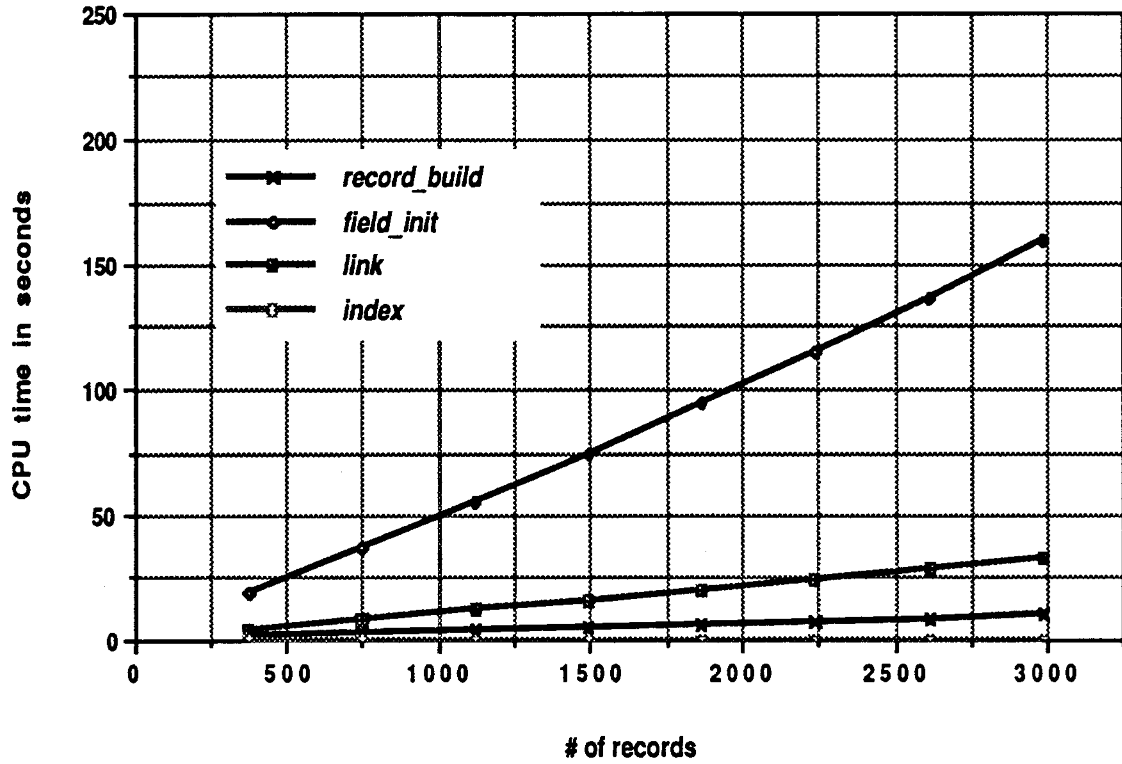


Fig. 7-5 Performance comparison of 4 modules of *dbvic.*

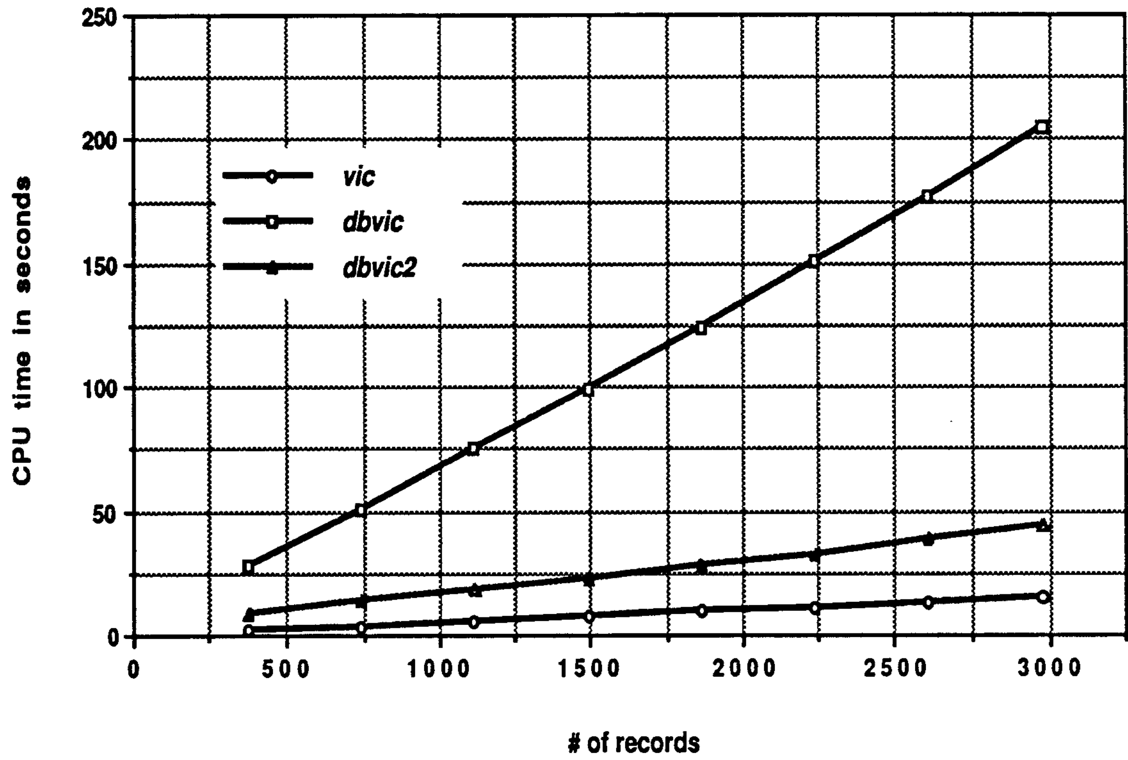


Fig. 7-6 Performance comparison of *vic*, *dbvic*, and *dbvic2*.

CHAPTER VIII

CONCLUSIONS AND FUTURE RESEARCH

In integrating various VLSI/CAD tools, it is desirable to store design data in a centralized database. However, it is generally agreed that conventional database systems are inefficient for VLSI/CAD applications, since such applications often access large amounts of data repetitively. Also, each application program needs a special data structure best suited for its own use. In order to solve these problems, we designed and implemented a data mapping subsystem that converts various VLSI/CAD design data stored in relational tables into an internal data structure that can be efficiently manipulated in C. This data conversion process follows a script written in a non-procedural mapping language. Besides constructing the data structure, the script can also initialize certain fields by using declarative SQL statements.

The results obtained from applying our technique to some real VLSI/CAD tools have shown that our approach, compared to C implementation, requires only about 1/10 of code for this data conversion in VLSI/CAD programs. Although our data-structure builder consumes several times more CPU cycles than a file-based C implementation, this performance overhead is not excessive due to the recent advancement of computer hardware.

If application programs are allowed to update data fields in a structured view, a view update mechanism that propagates updates from the structured view to the base relations may be required. The view update problem has received considerable attention in the database literature [BANC-81, DAYA-82, MASU-84, MEDE-86, KELL-86]. The view update problem can be defined as follows

[VIAN-88]: Consider a database and a view of the database, each satisfying certain integrity constraints. Suppose that an update is performed on the view that is valid with respect to the view. When and how can such a update be translated into a valid update of the underlying database?

Since the mapping from the database states to the view states is many-to-one in general, the reverse mapping from a view to a database state is not unique. Therefore, as proposed by [MASU-84] and [KELL-86], a view update translation mechanism may have to involve the users in resolving this ambiguity. The view update translation mechanism, which propagates the values that have been changed in the structured view to base relations, can be invoked at the end of the program execution.

Another area for future research is an extention of our system as shown in Fig. 8-1. *Tuple database* is simply a collection of tuples, each of which represents a fragment of design information with an arbitrary length. A meaningful unit such as *composite objects* is usually a collection of heterogeneous tuples.

A query statement returns a set of possible tuples, which may be heterogeneous. All operations on tuples are based on attribute values and the database can be projected or sliced in many different ways, allowing flexible grouping of design information without losing the simplicity of the relational model.

After retrieving a set of possible tuples (*database*) by a select statement, we can apply our mapping technique on the *database* to construct a structured view for a VLSI/CAD application program. This approach can support versioning, composite objects, layering, etc.

A simple way of supporting versions that cut across relational tables can be supported by providing attributes *version-from* and *version-to* to every tuple. Then tuples belonging to the version *ver* can be selected by applying the following condition *version-from ver*

version-to. We can support tables in a traditional relational database simply by providing enumerated-type attribute *tbl*, whose values are table names, to every tuple. The major goal of this extended system is to enhance further the flexibility of the relational model without losing its amicability for efficient query processing of design data.

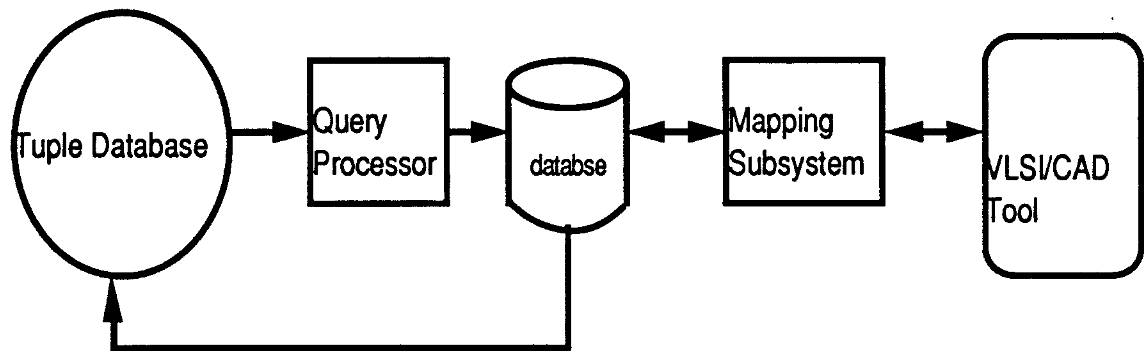


Fig. 8-1 Extended system architecture.

BIBLIOGRAPHY

[AFSA-85]
Afsarmanesh, H., and Knapp, D. An extensible object-oriented approach to data bases for VLSI/CAD. Proc. VLDB, 1985, pp.13-24.

[BANC-81]
Bancilhon, F., and Spyratos, N. Update semantics of relational views. *ACM TODS 6*, 4 (Dec. 1981), pp.557-575.

[BANC-85]
Bancilhon, F., Kim, W., and Korth, H.F. A model of CAD transactions. Proceedings of VLDB, 1985, pp.25-33.

[BATO-84]
Batory, D.S., and Buchman, A.P. Molecular objects, abstract data types, and data models: A framework. Proc. 10th Intl. Conf. on Very Large Data Bases, 1984, pp. 197-207.

[BATO-85]
Batory, D.S., and Kim, W. Modeling concepts for VLSI CAD objects. *ACM TODS 10*, 3 (Sep. 1985), 322-346.

[CHEN-76]
Chen, P.P. "The entity-relationship model - toward a unified view of data." *ACM TODS 1*, 1 (March 1976), 9-35.

[CHEN-88]
Chen, G.W., and Parng, T.M. A database management system for a VLSI design system. Proc. 25th ACM/IEEE Design Automation Conference, 1988, pp. 257-262.

[DADA-86]
Dadam, P. et al. A DBMS prototype to support extended NF^2 relations: an integrated view on flat tables and hierarchies. Proc. 1986 ACM Conf. on Management of Data, May 1986, pp. 356-367.

[DATE-86]
Date, C.J. *An Introduction to Database Systems*, 4th ed. Vol. 1, Addison-Wesley, Reading, MA: Addison-Wesley, 1986.

[DAYA-82]
Dayal, U., and Bernstein, P.A. On the correct translation of update operations on relational views. *ACM TODS 8*, 3 (Sep. 1982), pp381-416.

[EAST-80]
Eastman, C.M. System facilities for CAD databases. Proc. 17th ACM/IEEE Design Automation Conf., 1980, pp. 50-56.

[FAIR-78]
Fairbairn, D.G., and Rowson, J.A. ICARUS: An interactive integrated circuit layout program. Proc. 15th ACM/IEEE Design Automation Conference, 1978, pp.188-192.

[FRIS-86]
Frison, P., and Gautrin, E. MADMACS: A new VLSI layout macro editor. Proc. 23rd ACM/IEEE Design Automation Conference, 1986, pp.654-658.

[HARD-84]
Hardwick, M. Extending the relational database data model for design applications. Proc. 21st ACM/IEEE Design Automation Conf., 1984, pp. 110-116.

[HARD-87]
Harkwick, M. Why rose is fast: Five optimizations in the design of experimental database system for CAD/CAM applications. Proc. ACM Conf. on Management of Data, Dec. 1987, pp. 292-298.

[HASK-82]
Haskin, R.L., and Lorie, R.A. On extending the functions of a relational database system. Proc. 1982 ACM Conf. on Management of Data, Jun. 1982, pp. 207-212.

[HOLL-84]
Hollaar, L., Nelson, B., Carter, T. and Lorie, R.A. The structure and operation of a relational database system in a cell-oriented integrated circuit design system. Proc. 21st ACM/IEEE Design Automation Conference, 1984, pp. 117-125.

[HWAN-83]
Hwang, H., and Dayal, U. Using the entity-relationship model for implementing multi-model database systems, in: Chen, P.P. (ed.), *Entity-Relationship Approach to Information Modeling and Analysis*, Elsevier Science Publishers B.V. (North-Holland), 1983, pp. 235-257.

[INFO-87]
Informix, *Informix-ESQL/C*: Embedded SQL and tools for C, Programmer's Manual, Informix Software Inc., Menlo Park CA, July 1987.

[JAJO-83]
Jajodia, S., and Ng, P.A. On representation of relational structures entity-relationship diagrams, in: Davis C.G. (eds.), *Entity-Relationship Approach to Software Engineering*, Elsevier Science Publishers B.V. (North-Holland), 1983, pp. 249-263.

[KATZ-82]
Katz, R.H. A database approach for managing VLSI design data. Proc. 19th ACM/IEEE Design Automation Conference, 1982, pp.274-282.

[KATZ-83]
Katz R.H., and Goodman, N. View processing in MULTICASE, A heterogeneous database system, in: Chen P.P. (ed.), *Entity-Relationship Approach to Information Modeling and Analysis*, Elsevier Science Publishers B.V. (North-Holland), 1983, pp.257-277.

[KATZ-84]
Katz, R.H. Database support for versions and alternatives of large design files. *IEEE Trans. Software Eng., SE-10*, 2 (March 1984), 191-200.

[KATZ-85]
Katz, R.H. *Information Mamagement for Engineering Design, Survey Computer Science*, Springer-Verlag, Berlin, 1985.

[KELL-86]
Keller, A.M. Choosing a view update translator by dialog at view definition time. Proc. 12th Intl. Conf. on Very Large Data Bases, Aug. 1986, pp. 467-474.

[KEMP-87]
Kemper, A. An object-oriented database system for engineering applications. Proc. 1987 ACM Conf. on Management of Data, Dec. 1987, pp. 299-310.

[KETA-86]
Ketabchi, M.A., Berzins, V., and March, S. ODM: An object oriented model for design databases. Proc. ACM Ann. Computer Science Conf., 1986, pp. 261-269.

[KETA-87]
Ketabchi, M.A., and Berzins, V. Modeling and managing CAD databases. *IEEE Computer*, Feb. 1987, pp. 93-102.

[KETA-88]
Ketabchi, M.A., and Berzins, V. Mathematical model of composite objects and its application for organizing engineering databases. *IEEE Trans. Software Eng. 14*, 1 (Jan. 1988), 71-84.

[KIM-86]
Kim, W., and Chou, H.T. A unifying framework for version control CAD environment. Proc. 12th Intl. Conf. on Very Large Data Bases, August 1986, pp. 336-344.

[KIM-88]
Kim, W., Chou, H.T. and Banerjee, J. Operations and implementation complex objects. *IEEE Trans. Software Eng. 14*, 7 (July 1988), 985-996.

[MAKI-77]
Makinouchi, A. A consideration on mormal form of not-necessarily-normalized relation in the relational model. Proc. 3th Intl. Conf. on Very Large Data Bases, Oct. 1977. pp. 447-453.

[MASU-84]
Masunaga, Y. A relational database view update translation mechanism. Proc. 10th Intl. Conf. on Very Large Data Bases, 1984, pp. 309-320.

[MEDE-85]
Medeiros, C.B., and Tompa, F.W. Understanding the implications view update policies. Proceedings of VLDB, 1985, pp. 316-323.

[ROTH-87]
Roth, M.A., Korth, H.F., and Batory, D.S. SQL/NF: A query language NF^2 relational databases. *Inform. Systems 12*, 1 (Jan. 1987), 99-114.

[RUBI-87]
Rubin, S.M. *Computer Aids for VLSI Design.* Addison-Wesley, Reading, MA: Addison-Wesley, 1987.

[SCHE-82]
Schek, H.J., and Pistor, P. Data structures for an integrated data base management and information retrieval system. Proc. 8th Intl. Conf. on Very Large Data Bases, Sep. 1982, pp. 197-207.

[SHIP-81]
Shipman, D.W. The functional data model and the data language DAPLEX *ACM Trans. Database System 6*, 1 (March 1981), pp. 140-173.

[SHU-75]
Shu N.C., Housel, B., and Lum, V.Y. CONVERT: A high level translation definition language for data conversion. *Communications of ACM 18*, 10 (Oct. 1975), 557-567.

[SIBL-73]
Sibley, E.H., and Taylor R.W. A data definition and mapping language. *Communications of ACM 16*, 12 (Dec. 1973), 750-759.

[SMIT-77]
Smith, J.M., and Smith, D.C.P. Database Abstractions: Aggregation and generalization. *ACM TODS 2*, 2 (Jun. 1977), 105-133.

[STON-86]
Stonebraker M., and Rowe, L.A. The design of POSTGRES. Proc. 1986 ACM Conf. on Management of Data, May 1986, pp. 340-355.

[ULLM-83]
Ullman, J.D. *Priciple of Database Systems* MD: Computer Science Press, 1983.

[UW-87]
VLSI Design Tools Reference Manual, TR #87-02-01, Department Computer Science, Univ. of Washington, Seattle, Wash., 1987.

[VIAN-88]
Vianu, V. A dynamic framework for object projection views. *ACM TODS 13*, 1 (Mar. 1988), pp. 1-22.

[WIED-86]
Wiederhold, G. Views, objects, and databases. *IEEE Computer*, Dec. 1986, pp. 37-44.

[ZANI-83]
Zaniolo, C. The database language GEM. Proc. 1983 ACM Conf. on Management of Data, May 1983.

[ZANI-85]
Zaniolo, C. The representation and deductive retrieval of complex objects. Proceedings of VLDB, 1985, pp 456-469.

APPENDIX

A.1 BNF Grammar of Mapping Language

In this appendix, we describe the complete BNF grammar for our mapping language, and then explain the semantics of its various constructs. In describing the grammar, we use the following conventions.

1. [A] represents an optional (zero or one) occurrence of A.
2. A | B represents an occurrence of A or B.
3. Terminal symbols are represented in two ways: Reserved keywords appear as themselves in bold characters, and punctuation characters and operators are enclosed in single quotation marks.
4. Syntactic units appear in italic characters (e.g., *struct-statement*).

statement ::= *struct-statement*
| *index-statement*
| *link-statement*

struct-statement ::= **FOR EACH** *table-name*
PROVIDE STRUCT *record-type*

record-type ::= *record-type-name* '{' *field-list* '}'

field-list ::= *field-declaration* ';' [*field-list*]

field-declaration ::= *type-specifier* *declarator* ['=' *expression*]

type-specifier ::= *integer-type-specifier*
| *floating-point-type-specifier*
| *structure-type-reference*

integer-type-specifier ::= *signed-type-specifier*

| *unsigned-type-specifier*
| *character-type-specifier*

signed-type-specifier ::= **short** [**int**]
| **int**
| **long** [**int**]

unsigned-type-specifier ::= **unsigned short** [**int**]
| **unsigned** [**int**]
| **unsigned long** [**int**]

character-type-specifier ::= [**unsigned**] **char**

floating-type-specifier ::= **float**
| **long float**
| **double**

structure-type-reference ::= **struct** *record-type-name*

declarator ::= *simple-declarator*
| *array-declarator*
| *pointer-declarator*

simple-declarator ::= *identifier*

array-declarator ::= *declarator* '[' *constant* ']'

pointer-declarator ::= '*' *declarator*

expression ::= *term* [*arithmetic-operator expression*]

term ::= ['+' | '-'] *value*

value ::= *constant*
| *table-name* '.' *column-name*
| *SQL-string*
| '(' *expression* ')'
| *function-name* '(' *expression* ',' *expression* ')'

SQL-string ::= '"' *SQL-statement* '"'

function-name ::= **MAXI**
| **MINI**

arithmetic-operator ::= '+' | '-' | '*' | '/'

record-type-name ::= *identifier*

table-name ::= *identifier*

column-name ::= *identifier*

index-statement ::= **PROVIDE** *index-kind* **INDEX** *index-name*
ON *table-name* '(' *column-name* ')'

index-kind ::= **STRHASH**
| **BSTREE**
| **NUMHASH**

index-name ::= *identifier*

link-statement ::= *simple-link-statement*
| *for-link-statement*

simple-link-statement ::= *link-clause*
where-clause
with-clause

link-clause ::= **LINK** *record-type-name* **AND** *record-type-name* '(' *mapping-kind* ')'

record-type-name ::= *type-name* [*alias*]

mapping-kind ::= **ONE-ONE**
| **ONE-MANY**

where-clause ::= **WHERE** *predicate*

predicate ::= *condition*
| *condition* **AND** *predicate*

condition ::= *item-name* '=' *item-name*

item-name ::= *table-name* '.' *column-name*
| *record-type-name* '.' *field-name*

with-clause ::= **MEMPTR** ':' *field-name*
[**SIBPTR** ':' *field-name*
[**BCKPTR** ':' *field-name*]]

for-link-statement ::= **for** *for-expression* *simple-link-statement*

for-expression ::= '(' *loop-var* '=' *constant* ';' *loop-var* '<' *constant* ';' *loop-var* '++' ')'

loop-var ::= *identifier*

field-name ::= *identifier*

statement

A *statement* in a script can be a *struct-statement*, *index-statement*, or *link-statement*.

struct-statement

A *struct-statement* is used to construct the records for the tuples in a relational table. The type of the records to be constructed for the tuples in a table *table-name* is specified by *record-type*.

record-type

The syntax of the *record-type* is similar to that of a *structure type* in C. Selected attributes of each tuple and additional fields are specified by *field-list* to form a record type *record-type-name*. Nested record type definitions are not allowed.

field-declaration

A *field-declaration* consists of a *type-specifier* of the field and a *declarator*. The declaration of a field may include an *expression* that initializes the field value.

expression

An *expression* can be formed from constants, SQL statements enclosed in double quotation marks, and the attributes of the current tuple, each of which is specified as *table-name '.' column-name.*

function-name

A *function-name* can be MAXI or MINI, which computes the maximum or minimum, respectively, of their two arguments.

SQL-statement

A *SQL-statement* may include parameters preceded by percent marks. This SQL statement is submitted to the database management system as a character string with the parameters replaced by the actual values for a particular record.

arithmetic-operator

The binary arithmetic operators are +, -, *, and /. The binary + and - operators have the same precedence, which is lower than the precedence of * and /, which is in turn lower than unary + and -.

index-statement

An *index-statement* is used to construct an indexing mechanism for fast accesses to records. The name of an index is specified by *index-name*. The field to be indexed is specified as *record-type-name* '(' *field-name* ')', which indicates that the index should be provided for field *field-name* of record type *record-type-name*.

index-kind

We support three kinds of indexes: STRHASH (string hash), NUMHASH (number hash), and BSTREE (binary search tree). Indexing mechanism STRHASH is based on hashing on character strings. Indexing mechanism NUMHASH is based on hashing on

numbers. Indexing mechanism BSTREE uses a binary search tree on character strings.

link-statement

A *link-statement* that is used to provide pointers between two record types can be a *simple-link-statement* or *for-link-statement.*

simple-link-statement

A *simple-link-statement* consists of *link-clause, index-clause* and *link-clause.*

link-clause

A *link-clause* is used to provide pointers from the records of the type *R1* indicated by the first *record-type-name* to the records of the type *R2* indicated by the second *record-type-name.* An *alias* is allowed to link records of the same record type.

mapping-kind

The kind of the relationship type between the two record types should be specified by *mapping-kind,* which may be *one-one* or *one-many.*

where-clause

A *where-clause* specifies the condition for establishing the linkages.

predicate

The *predicate* in *where-clause* specifies that record *r1* of type *R1* and record *r2* of type *R2* are to be linked if *predicate* is true when it is evaluated using the attribute values associated with *r1* and *r2.* An attribute used in the *predicate* may be a column name of a table (*table-name* '.' *column-name*) or a field name of a record-type (*record-type-name* '.' *field-name*).

with-clause

A *with-clause* identifies the fields where pointers are stored. We use three kinds of pointers: MEMPTR (member pointer), SIBPTR (sibling pointer), and BCKPTR (back pointer). For one-many relationship, both MEMPTR and SIBPTR must be provided, and BCKPTR is optional. For one-one relationship, MEMPTR must be provided, and BCKPTR is optional.

for-link-statement

A *simple-link-statement* can be specified inside of a loop to provide pointers from each element of a pointer array field in one record type to associated records of the other record type.

A.2 Design Formats

In this section, we describe three design formats used by our data-structure builder: the *.sim*, *.ca*, and *.cif* formats.

A.2.1 Simfile - the *.sim* file format

The *.sim* files are ASCII files used by various programs to describe MOS transistor networks and their associated parameters. *Mextra* and *netlist* output *.sim* files; *sim2spice*, *presim* and others read the files.

Each line of a *.sim* file is a separate "record" whose type is determined by the first character of the line. The possible record types are described in this section.

Lines beginning with vertical bar (|) are treated as comments and ignored by programs that read *.sim* files. A only exception is if this line contains the information *units*:, *tech*:, and *format*:. *Units* specifies the conversion factor to centimicrons, *tech* declares the technology (e.g., nmos) that was used in the design and *format* declares the *.sim* file format used for the various other records. The *.sim* format uses a lambda value for transistor size and picofarad loads.

Lines beginning with @ followed by *filename* (@ *filename*) are used to redirect input from the named file. When the end-of-file is reached, input reverts to the current file at the following line. Indirect files can be nested; usually there is some system dependent limit on the number of simultaneously open files which limits the depth of nesting.

The following is the possible transistor records:

e *gate source drain length width xpos ypos [rpa] area*
i *gate source drain length width xpos ypos [rpa] area*
d *gate source drain length width xpos ypos [rpa] area*
p *gate source drain length width xpos ypos [rpa] area*
l *gate source drain length width xpos ypos [rpa] area*

The first character tells the type of the transistor:

e n-channel enhancement

i n-channel zero threshold (intrinsic) enhancement

p p-channel enhancement

d depletion

l low-power depletion

The next three parameters are the names of the *gate*, *source*, and *drain* nodes. The *length* and *width* can be either integers or floating-point giving the dimensions of the active transistor area. *Xpos* and *ypos* report the coordinates of the upper left hand conner of the transistor (*netlist* always specifies 0,0).

The next parameter is a letter specifying the shape of the transistor:

r rectangular

p rectangular, monotonically increasing width

a other shapes

Netlist always puts an "r" in this field.

Area tells the true active area; may be different from width*length if the network extractor used an approximation (in output from *netlist*, area always equals width*length).

The following record is used to specify *node* capacitance in pf.

c *node cap*

Cap can be either an integer or floating-point number. This is the type of record used by *netlist* to describe user-specified capacitors. The other node to which this capacitor is connected is assumed ground (GND).

A.2.2 Cafile - the *.ca* file format

The *.ca* format allows a hierarchical description of VLSI circuit layout at a higher level than the *.cif* format. Every cell in a hierarchy is represented by a *.ca* file, which contains the layout of the cell. The

.ca format provides only rectangles for graphics primitives, which is the most commonly used way of specifying geometry. The *.ca* format is powerful descriptive form for VLSI geometry.

There are only a few *.ca* statements and they fall into one of two categories: statement with parameters or statement without parameters. A statement without parameters is enclosed by double arrows (<< ... >>) and a statement with parameters consists of a keyword followed by parameters separated by spaces.

Statements without Parameters

The layer statement (<< *layername* >>) sets the mask layer to be used for all subsequent geometry until the next such statement. *Layername* specifies a layer-name such as *metal* or *polysilicon*. For example, the command:

<< metal >>

sets the metal layer.

The label statement (<< labels >>) specifies that all subsequent rectangles are labels to be placed in specified locations.

The final statement in a *.ca* file is the end statement (<< end >>), which terminates a *.ca* file.

Statements with Parameters

The *tech* statement declares the technology (e.g., cmos) that was used in the design. For example, the command:

tech cmos-pw

declares cmos-pw technology.

The only geometry that the *.ca* format supports is a rectangle. The *rect* statement describes a rectangle by giving its *lower-left x-*, *lower-left y-*, *upper-right x-*, and *upper-right y-coordinates*. The format is as follows:

rect *llx lly urx ury*

Labels are used to specify texts for the names of such components as signals and cells in a circuit. A circuit extractor often

uses those labels when they produce a circuit from a layout description. Rectangles and names associated with labels are used by the graphics editor. The label statement in the *.ca* format is as follows:

label *name llx lly urx ury*

Name is the name of the label. *Llx*, *lly*, *urx*, and *ury* represent *lower-left x-*, *lower-left y-*, *upper-right x-*, and *upper-right y-coordinates* of a label's rectangle.

A cell call creates an instance of a subcell, which may be translated, rotated, and reflected within the bounding box of the current cell. A cell call is specified by using the *use* statement followed by the *transform* statement.

The *use* statement invokes a subcell-file that contains the layout description of the subcell. All subcells are given names when they are defined and these names are used in the *use* statement to identify them. The names of a subcell and a subcell-file that contains the layout of the subcell are identical. The format is as follows:

use *cellname*

The *transform* statement specifies three transformations to affect the geometry inside the subcell instanced by the *use* statement. Parameters followed by the keyword *transform* represent the left six elements of a 3X3 transformation matrix. The format is as follows:

transform *a b c d e f*

The first column of the matrix is specified by the numbers *a*, *b*, and *c* and the second column is specified by the numbers *d*, *e*, and *f*.

The coordinates for the bounding box of a subcell that is instanced by the *use* statement are given with the *box* statement. The *box* statement describes the bounding box for a subcell by giving its *lower-left x-*, *lower-left y-*, *upper-right x-*, and *upper-right y-coordinates*. The format is as follows:

box *llx lly urx ury*

A.2.3 Ciffile - the *.cif* file format

The *.cif* format is a recent form for the description of integrated circuits. Created by the university community, *.cif* format has provided a common database structure for the integration of many research tools. The *.cif* format provides a limited set of graphics primitives that are useful for describing the two-dimensional shapes on the different layers of a chip. The format allows hierarchical description, which makes the representation concise.

Each statement in the *.cif* format consists of a keyword or letter followed by parameters and terminated with a semicolon. Spaces must separate the parameters but there are no restrictions on the number of statements per line or of the particular columns of any field. Comments can be inserted anywhere by enclosing them in parenthesis.

There are only a few statements in the *.cif* format and they fall onto one of two categories: geometry or control. The geometry statements are: LAYER to switch mask layers, BOX to draw a rectangle, WIRE to draw a path, POLYGON to draw an arbitrary figure, and CALL to draw a subroutine of other geometry statements. The control statements are DS to start the definition of a subroutine, DF to finish the definition of a subroutine, 0 through 9 to include additional user-specified information, and END to terminate a *.cif* file. All of these keywords are usually abbreviated to one or two letters that are unique.

Geometry

The LAYER statement (or the letter L) sets the mask layer to be used for all subsequent geometry until the next such statement. Following the LAYER keyword comes a single layer-name parameter. For example, the command:

L NC;

sets the layer to be the NMOS contact cut.

The BOX statement (or the letter B) is the most commonly used way of specifying geometry. It describes a rectangle by giving its length, width, center position, and an optional rotation. The format is as follows:

B *length width xpos ypos [rotation]*;

Without the *rotation* field, the four numbers specify a box the center of which is at (*xpos*, *ypos*) and is *length* across in *x* and *width* tall in *y*. All numbers in the *.cif* format are integers that refer to centimicrons of distance, unless subroutine scaling is specified.

The WIRE statement (or the letter W) is used to construct a path that runs between a set of points. The path can have a nonzero width and has rounded corners. After the WIRE keyword comes the width value and then an arbitrary number of coordinate pairs that describe the endpoints. For example, the command:

W 25 100 200 100 100 200 200 300 200;

specifies a wire. The endpoint and corner rounding are implicitly handled.

The POLYGON statement (or the letter P) takes a series of coordinate pairs and draws a filled polygon from them. Since filled polygons must be closed, the first and last coordinate points are implicitly connected and need not be the same. A sample polygon statement is as follows:

P 150 100 200 200 200 300 100 300 100 200;]

Hierarchy

The call statement (or the letter C) invokes a collection of other statements that have been packaged with DS and DF. All subroutines are given numbers when they are defined and these numbers are used in the CALL to identify them. If, for example, a LAYER statement and a BOX statement are packaged into subroutine 4, then the statement:

C 4;

will cause the box to be drawn on that layer.

In addition to simply invoking the subroutine, a CALL statement can include transformations to affect the geometry inside the subroutine. Three transformations can be applied to a subroutine in the *.cif* format: translation, rotation, and mirroring. Translation is specified as the letter T followed by an *x*, *y* offset. These offsets will be added to all coordinates in the subroutine, to translate its graphics across the mask. Rotation is specified as the letter R followed by an *x*, *y* vector endpoint that defines a line to the origin. Mirroring is available in two forms: MX to mirror about the *x* axis and MY to mirror about the *y* axis. The geometry is flipped about the axis by negating the appropriate coordinate.

Any number of transformation can be applied to an object and their listed order is the sequence that will be used to apply them.

A sample call statement is as follows:

C 10 **T** -50 0 **R** 0 -1 **MY MX**;

The statements to be packaged are enclosed between DS (definition start) and DF (definition finish) statements. Arguments to the DS statement are the subroutine number and a subroutine scaling factor. There are no arguments to the DF statement. The scaling factor for a subroutine consists of a numerator followed by a denominator that will be applied to all values inside the subroutine. This scaling allows large numbers to be expressed with fewer digits and allows ease of rescaling a design.

A sample subroutine is described as follows:

DS 10 2 20;

  B 10 20 5 5;

  W 1 5 5 10 15;

DF;

Note that the scale factor is 2/20, which allows the trailing zero to be dropped from all values inside the subroutine.

Control

Extensions to the *.cif* format can be done with the numeric

statements 0 through 9. Although not officially part of the *.cif* format, certain conventions have evolved for the use of these extensions. Typical user extensions to the *.cif* format are as follows:

0 x y layer N name;	Set named node on specified layer and position
0V x1 y1 x2 y2 ... xn yn;	Draw vectors
2A "msg" T x y;	Place message above specified location
2B "msg" T x y;	Place message below specified location
2C "msg" T x y;	Place message centered at specified location
2L "msg" T x y;	Place message left of specified location
2R "msg" T x y;	Place message right of specified location
9 cellname;	Declare cell name
94 label x y;	Place label in specified location

The final statement in a *.cif* file is the END statement (or the letter E). It takes no parameters and typically does not include a semicolon.

A.3 Circuit and Layout Diagrams of 2-Bit Adder

Fig. A-1 shows the circuit diagram of the 2-bit adder used in Chapter V. The layout diagram of the 2-bit adder is shown in Fig. A-2. Labels except for "X1", "Y1", and "Z1" are removed for clarity of exposition.

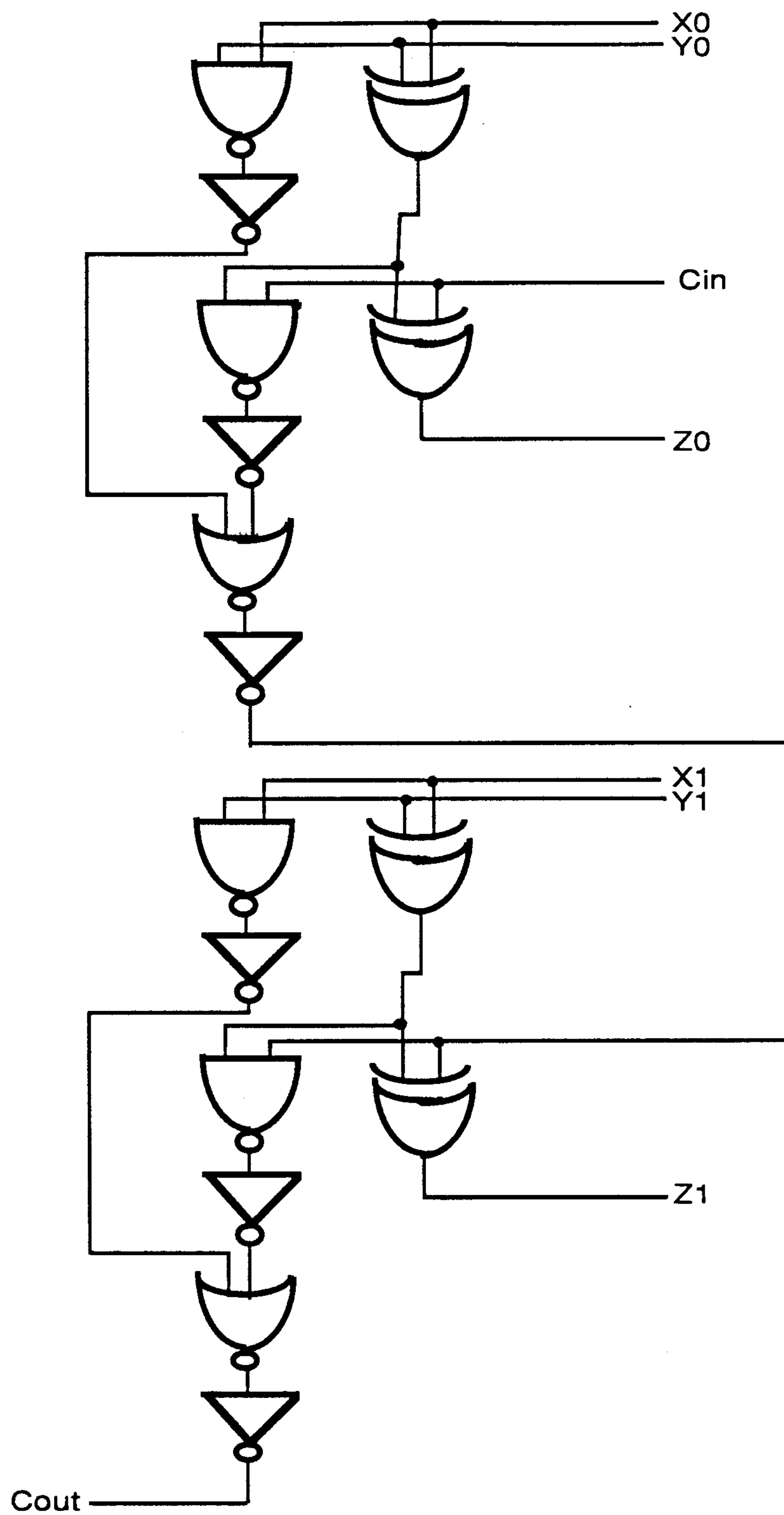


Fig. A-1 Circuit diagram of 2-bit adder.

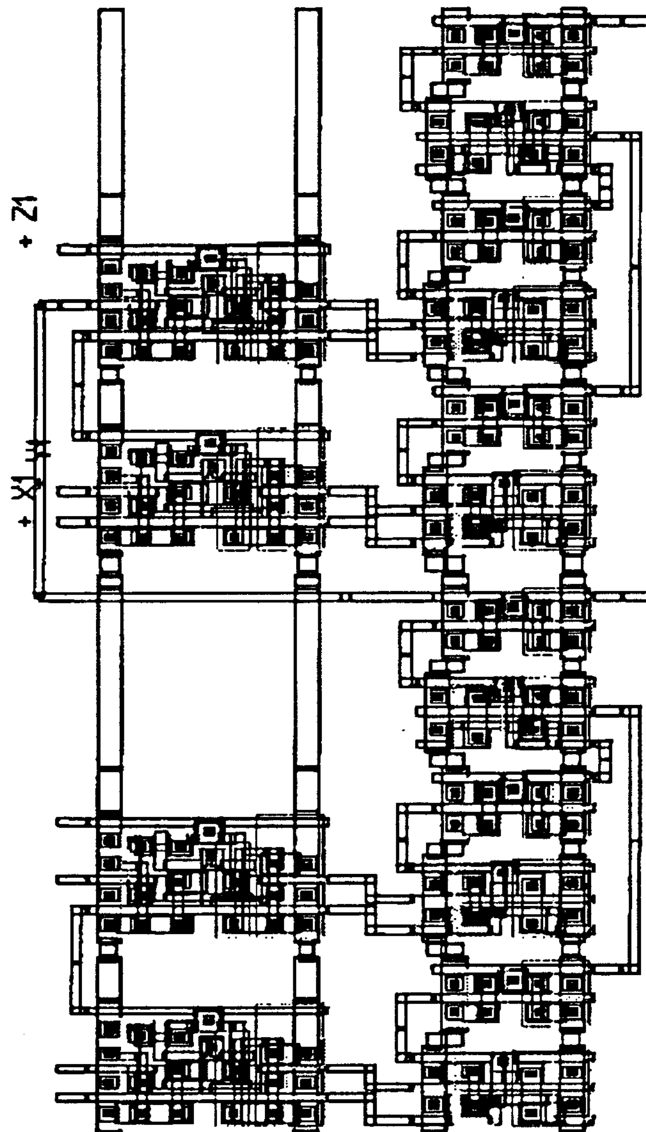


Fig. A-2 Layout diagram of 2-bit adder.