A DATABASE MANAGEMENT SYSTEM FOR COMPUTER-AIDED DIGITAL CIRCUIT DESIGN

THESIS

Sue A. Ehrhart
Captain, USAF

AFIT/GCS/ENG/88D-4

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY
Wright-Patterson Air Force Base, Ohio

AFIT/GCS/ENG/88D-4

A DATABASE MANAGEMENT SYSTEM FOR COMPUTER-AIDED DIGITAL CIRCUIT DESIGN

THESIS

Sue A. Ehrhart
Captain, USAF

AFIT/GCS/ENG/88D-4

Approved for public release; distribution unlimited

AFIT/GCS/ENG/88D-4

A DATABASE MANAGEMENT SYSTEM FOR COMPUTER-AIDED DIGITAL CIRCUIT DESIGN

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Computer Systems

Sue A. Ehrhart, B.S.

Captain, USAF

December 1988

Approved for public release; distribution unlimited

Preface

The prototype AFIT Digital Circuit Design Environment proved to be a capable design tool but it suffered from redundantly stored information and a lack of flexibility. Circuit data was stored in all the various formats used during a design session. Designs were restricted to using only those TTLs that were originally provided with the tool because, with TTL data was scattered throughout the environment, new TTLs could not be easily added. The solution was the development of a centralized database with an expandable library of TTL components. Computer listings of the DBMS modules and utility programs are not included in this document; however, they can be obtained through the Air Force Institute of Technology, School of Engineering, Wright-Patterson AFB OH 45433.

The success of this thesis effort is due in large part to my faculty advisor, Captain Bruce L. George, untiring protagonist of the AFIT Digital Circuit Design Environment. His constant understanding and encouragement made him a pleasure to work with. Thank you also to Captains Mark Roth and Brian Donlan for their guidance as members of my thesis committee, and to Captain Jorge da Silva Santos of the Brazilian Air Force, for coming to my aid during the rough spots in the development. Finally, I wish to thank my husband Ken and our daughter Becky for the extra measure of love and support they gave while this work was in progress.

Sue A. Ehrhart

Table of Contents

List of Figures

Abstract

This thesis effort documents the design and implementation of a relational database and associated database management system (DBMS) for the AFIT digital circuit design environment, a graphics oriented tool that allows circuits to be designed at a uniform, chip-level of detail, checked for proper connections, and simulated. The approach to this effort included a survey of existing methods of Computer-Aided Design (CAD) data management, analysis of the data and data manipulation requirements of the design environment, design of a data manipulation language, and implementation of a DBMS to carry out the manipulations. Implementation was done in the C programming language and based on lower-level database routines found in C Database Development by Al Stevens. Limitations encountered as a result of using these routines are discussed along with the results of testing. This effort also includes three separate database utility programs. One allows new TTLs to be added to the database but does not provide the ability to input the executable functions of those new TTLs. The second provides the capability to rebuild corrupted index files, and the third prepares floppy diskettes for data storage. A user's manual is included for the operation of the database utility programs. Recommendations for future work are also presented.

A DATABASE MANAGEMENT SYSTEM FOR COMPUTER-AIDED DIGITAL CIRCUIT DESIGN

I. Introduction

Background

Recent efforts at the Air Force Institute of Technology (AFIT) have resulted in a prototype Computer-Aided Design (CAD) environment for designing two-dimensional discrete component digital circuits on a PC AT compatible microcomputer. Figure 1 - 1 illustrates how the environment combines three software programs, the graphic interface program, the InterConnect Expert (ICE), and the Logic Simulator (LOGSIM), into one complete circuit design package for use by students in digital logic courses at AFIT and, eventually, at other learning institutions.

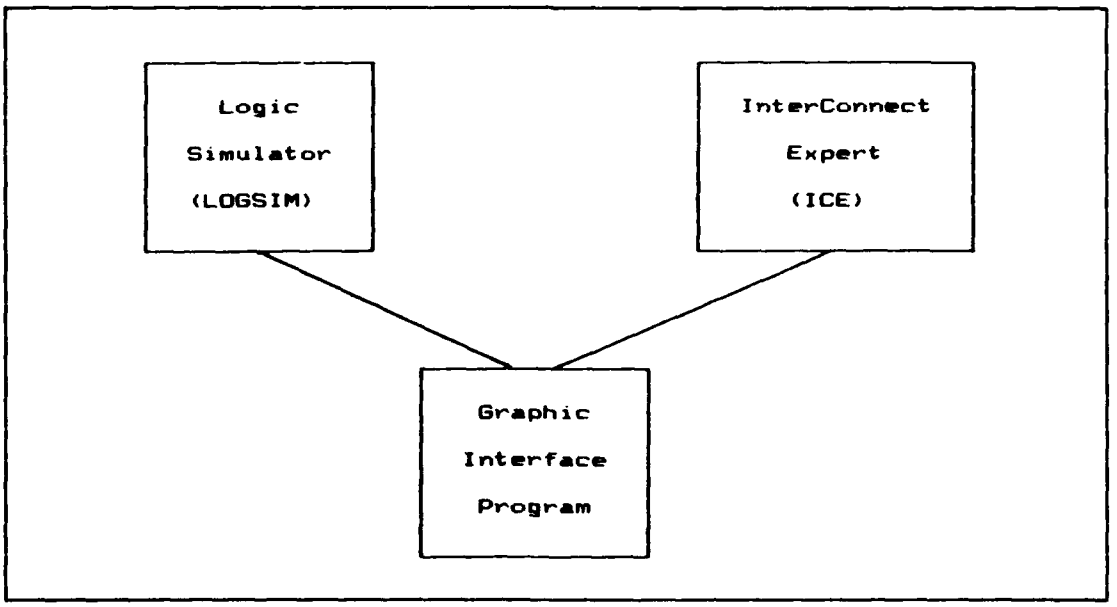

Fig. 1 - 1. Organization of AFIT's Digital Circuit Design Environment

The user interface allows a designer to construct a small, single screen circuit by selecting components from a library of 34 pre-defined integrated circuit chips (TTLs) and then wiring the connections between those components on the screen (1). ICE can be used to check the circuit for possible wiring errors, and LOGSIM, to simulate the circuit's operation (13; 5). Designing circuits on the environment could, in some cases, replace hardware designing but its main purpose is as a pre-hardware design tool to save the designer time and frustration wondering whether a problem is being caused by a design flaw or hardware problem.

While the environment currently does an adequate job of storing and manipulating all of the data it requires to perform its purpose, there are some problems both with what is being stored and how it is being stored.

The inability to obtain a listing of available circuits is a problem caused by what is or, more accurately, is not being stored. To retrieve a circuit, the designer must remember the filename he assigned to it. There is no information currently stored to remind the designer of the function of a circuit or what stage of development it may be in.

The problems caused by how data is stored are more serious and can be divided into two areas, the TTL library and the circuit data.

Information about the 34 TTLS in the library is found in each of the three programs in the environment. The interface maintains information for graphic representation, ICE maintains pin assignments, and LOGSIM maintains functions to transform input signals into output signals. To add a new TTL to the library would require software

modification and recompilation, essentially restricting the designer to using only those components currently available.

Information pertaining to each circuit design is stored in as many as ten files on any disk drive the designer specifies (1). Each file contains the set of information required for a particular activity and in those cases where multiple activities require the same piece of information, that information is stored in multiple files. Storage of redundant information is a concern when working with microcomputers because of the limited fixed disk space they have available.

Each circuit file contains what would be the result of a database query if the environment had a central database and a database management system (DBMS) that would allow such queries. A commercial DBMS was not considered for use with the environment however, primarily so it could be distributed without licencing restrictions.

Problem Statement

The primary goal of this effort is to implement a central database and associated database management system (DBMS) specifically tailored to the AFIT digital circuit design environment. The secondary goal is to develop a utility program for inserting new TTLs into this database.

The addition of a central, easily expandable database will increase the environment's usefulness. The database, in conjunction with its DBMS, should reduce memory usage due to redundant data and increase flexibility by allowing each of the environment's component programs to access needed information for itself, without relying on the interface program to provide the right pieces in the proper format.

Scope

This effort addresses the design, implementation, and testing of a database and an associated DBMS to store and manipulate two-dimensional discrete component digital circuit CAD data.

A limited Data Manipulation Language (DML), based on the C programming language, is defined to provide the required access to the database.

A DBMS consisting of the minimum functions required for current capabilities and foreseen enhancements of the environment was implemented on a Zenith 248 microcomputer. Because the environment is stand-alone and single-user, no concurrency control measures were necessary. Crash recovery consists entirely of user-initiated back-ups. The DBMS does not perform integrity checks, checking input data against valid values specified in a data dictionary, because the data is created and accessed only by the software programs, not directly by any designer. It is left up to the using program to ensure database queries are properly formatted and that fields contain valid values. Data security for circuit designs is accomplished by forcing design data to the floppy disk drive and having each designer maintain his circuits on his own floppy diskette. The TTL database is protected by allowing the casual user no access. A separate utility program is provided to allow a data manager to add new TTLs.

This effort does not include adding the capability to input the executable function of a TTL. This means that LOGSIM will not be able to simulate the operation of any new TTLs added via the utility program.

Finally, while not actually part of the DBMS, this effort includes the conversion routines necessary to recreate the file structures used by the graphic interface program.

Assumptions

It is assumed that the reader has a basic knowledge of database theory, particularly in the area of relational databases. It is also assumed that the reader has a basic knowledge of the workings of TTLs and the design of digital circuits.

General Approach

This effort was accomplished in four stages. The first stage included a literature review, or survey, of the data storage techniques used in similar CAD applications to determine the most appropriate database model. Meetings were held during this time with Capt Santos and Capt Matuszek, who were involved in concurrent enhancements to the environment, to reach agreement on what information is needed and how it should be displayed and manipulated (11; 8). The second stage involved analyzing the environment's three component programs to determine the precise format of all data entities to be stored in the database. The logical organization of the data finalized during this stage, and the data manipulation language defined. The detailed design and implementation of the database and the DBMS took place during the third phase. The DBMS was written in the C programming language to conform with the existing programs. The fourth and final stage was the testing and evaluation of the DBMS.

Sequence of Presentation

Chapter 2 provides the results of a survey covering data models used by similar CAD applications and identifies the data model selected for use. Chapter 3 identifies the required data elements and justifies their logical organization. Chapter 4 provides the index structure and the detailed design of the database and DBMS program. Chapter 5 discusses the operation of each module of the DBMS as it was implemented. Chapter 6 contains an evaluation of the test results and, finally, Chapter 7 contains the concluding remarks and specific recommendations for areas of improvement and further research. A user's guide for the database utility programs is included at Appendix A.

II. Literature Review

This phase of the thesis effort involved a survey of Computer-Aided Design (CAD) applications similar in nature to the discrete component digital circuit CAD environment developed at the Air Force Institute of Technology (AFIT). The goal was to find the various ways CAD data is being represented and select the method best suited to AFIT's CAD environment.

Commercial Package versus Custom Design

While existing database management systems are adept at handling traditional business applications, usually working with numbers or strings of characters, the characteristics of CAD data make it difficult to store and manipulate. Graphic images in general, and computer-aided design data especially, are highly complex in nature. Data associated with graphic images can be oriented in space, for example, and viewed from different angles and from different levels of detail. In addition, some designs, such as the digital circuits AFIT is concerned with, can be simulated on the computer. This simulation requires the storage of time-sensitive input parameters and their associated time-sensitive responses.

There has been an increasing number of CAD applications in recent years, and the interest in CAD data management has risen accordingly. Some commercial CAD software packages, such as TEO/Electronics, are available (9). In some cases, however, this option is being turned down in favor of custom designed software and the reason for doing so is easy to see. Each CAD application requires some features peculiar

to it and any commercial package that can meet those highly specific requirements is probably too specialized to be profitable and remain on the market. If the requirements don't allow for flexibility, a custom designed CAD package and database is the answer.

Database Models

Chang and Kunii conducted their own survey of methods for CAD data representation and said:

> The three main models for a traditional alphanumeric data base are the relational, the network, and the hierarchical. The varying requirements of the pictorial data base have led to modifications and extensions of these traditional models; the relational and hierarchical models have attracted the most attention [4:14].

Hierarchical. A hierarchical database is one in which records, or sets of attributes, are connected in a Parent-Child relationship. While one parent record may be linked to many children, a child record can be linked to only one parent, resulting in the hierarchical model's characteristic "family tree" structure (7:143). In hierarchical design databases, each level of the tree corresponds to a particular level of detail in the design (6:318; 10:40). In that sense, the hierarchical model conforms well to the concept of building a complex design in progressive stages, basic components making up larger components and so on up the tree until the design is complete.

Some CAD database designers choose the hierarchical structure because it offers a convenient way to divide a design into segments, either to distribute the work among several people, or to fit a large design within the confines of a computer screen. In all but the most

trivial cases, for example, a complete circuit design at the lowest level of detail would probably not fit on one screen, so groups of circuit components would be represented by a "black box." Individual "black boxes" could then be selected for viewing in greater detail.

This representation can be particularly useful for mechanical designs, where the "black box" would actually be the outside surface of a complex component whose internal workings are normally hidden at that level. It is impossible to both show all of the pieces of a three-dimensional object simultaneously and show the pieces in their proper relationships to each other. The "exploded view" does not accurately represent what the actual object looks like.

There are drawbacks to using the hierarchical model for applications such as circuit design. It would be difficult to envision the physical layout and interconnections of the circuit, for example, without printing out a copy of each component design and pasting them together. It further complicates the process of simulation, too. Although TEO/Electronics can simulate the operation of a multi-level circuit, LOGSIM, the simulator used by AFIT's CAD environment, does not have this ability (9:131; 5).

Relational. Data in the relational model is organized as a collection of tables (7:45). The data in these tables can, when necessary, be combined, or related, by matching key attribute fields from two or more tables. A problem with the relational model is that each table entry can contain only one value for each field in that table. Because each design, or circuit in this case, is unique and can be made up of any number of chips, the only way to represent a circuit

as a single relation or table entry is to set an upper limit on the number of chips per circuit and then leave space for each circuit to have the maximum number. This approach would waste a great deal of space and place an unnecessary restriction on circuit size.

Another possible solution to this problem is to represent a circuit not by one relation, but by a set of relations. Chang and Kunii discuss a CAD system that stores individually unique images by the features they have in common (4:14). For example, while all circuit designs are unique, chips and wires are features common to every circuit. All circuit designs, regardless of the number of chips or other features it has, can be represented by a relatively small number of relations if each feature that could possibly vary in number from one design to the next was organized as a separate relation.

The database Capt Adams designed for AFIT's digital circuit design environment is relational and uses essentially this same method, organizing data by features. Each circuit designed on AFIT's system results in zero or more entries into each of 10 relations. Those relations correspond to five categories of information that pertain to a circuit: interface information, graphical display, electrical continuity, expert system interface, and simulator interface (1:3-2).

Where the hierarchical model was best for viewing designs from many different levels of detail, the relational model is best suited for two-dimensional designs, like digital circuits, where the entire design can be represented at the same level of detail. Although it might be possible to represent a hierarchy of detail levels with the relational model, it is not desirable to use "black boxes" because, as stated

earlier, envisioning the physical layout would be difficult. Different levels of detail could be represented as "white boxes", where chunks of the design were thought of as a single component but where all of the internal workings remained visible. To do so, however, would require adding another field to the relations for each level maintained. Then, like limiting the number of chips allowed in each design, the maximum number of levels of detail would have to be set and used for all, again wasting space and placing artificial restrictions on the design process.

An Object Oriented View of CAD Data

Batory and Kim describe an excellent example of a relational model for the storage of circuit design data (2). The difference between their model and the relational model described earlier is that they view designs as objects, not simply as collections of common features, and all objects consist of two parts, an interface, or external characteristics, and an implementation, or internal characteristics. They say, "The interface of a circuit specifies the function of the circuit and lists its inputs and outputs" (2:324). The interface would not be made up of chips or wires. It would be a "black box" that could, given some inputs, perform a function and produce some outputs. The chips and wires inside the "black box" would make up the implementation portion of the circuit.

The same data that was present in the earlier relational model is still there, organized as sets of relations based on common features, as before. The only difference is that some of the relations together

make up the interface and some other relations together make up the implementation. This separation of interface and implementation allows a form of version control.

Different designers can use different internal components to building a circuit, or make connections between the internal components in a different manner, and yet still create circuits that perform the same functions. These circuits would be different versions of the same circuit type. It could be useful to be able to identify all of the circuits of the same type.

According to Batory and Kim, "all versions of a design object share the same interface description and differ only in their implementations descriptions" (2:323). All the circuits of the same type could easily be found, then, if all circuit implementations were associated with both a type identifier and version identifier.

Summary

The complex structure of CAD data requires careful consideration when selecting the most appropriate data model for a particular application. It appears from the surveyed applications, when either the physical structure of the design object is three-dimensional, or it is logically viewed at varying levels of detail, hierarchical is the most appropriate data model to use. If both the physical and logical structure of the design object is two-dimensional, however, as in this application, the relational data model is best.

III. Requirements Analysis and Design

A requirement of this thesis effort was that the central database and its associated database management system support the system as it originally existed. Requirements analysis therefore consisted of a thorough examination of the existing system. First, the individual pieces of data being used by each of the three programs in the design environment were studied to determine the required contents of the central database. Then, a study was made of how these data elements were used, most importantly how they were combined, to determine the operations and join strategies required in the DBMS.

The remainder of this chapter is organized into two main sections, the first section discusses the content and organization of the original system and the new central database. The second section covers the DBMS, the operations required to support the original system, and the data manipulation language developed to support those operations.

Database Content and Organization

Throughout this section, please note that the names given to the data fields in each figure may not match those assigned to them in the source referenced. Changes were necessary for consistency across the three programs of the design environment as well as with the new central database. Equivalent pieces of information are assigned the same name.

Please note also that in the figures used in this section, when an ellipsis (i.e. ...) appears between two fields on the same line that

means some fields have been omitted from the figure. When an ellipsis appears at the beginning of the second line of a figure it indicates continuation.

Original System. Data elements to be included in the new, centralized database were found in two areas. Information pertaining to TTLs was found hardcoded into the ICE and LOGSIM programs, and in two separate files used by the graphics interface program. Information pertaining to circuits was found stored as sets of files. The organization of the data elements from both of those areas is described below.

TTL Data. Those attributes required to graphically represent a TTL were found in two files used by the graphic interface program, "ttl.des" and "ttl.dat" (1:3-4; 1:3-3). Figures 3 - 1 and 3 - 2 show the data elements that made up those files.

TTL	TTL_DESC

Fig. 3 - 1. Elements of "ttl.des" File

TTL	NO_OF_PINS

Fig. 3 - 2. Elements of "ttl.dat" File

The ICE program used two hardcoded tables, the Engineering Workstation Chip Library and the Database Library, to store the information it needed to make sure TTLs are properly connected (13:56;

13:58). Figures 3 - 3 and 3 - 4 show the data elements contained in those tables.

TTL	TTL_DESC

Fig. 3 - 3. Elements of Engineering Workstation Chip Library

TTL	PIN	GATE	VALUE	TTL_DESC

Fig. 3 - 4. Elements of Database Library

TTL information in the LOGSIM program is organized as a CASE statement associating each TTL name with its executable chip configuration routine (5:A-31). Because this thesis effort does not include storing the executable function of the TTLs in the central database, this information is not considered any further in this section.

Circuit Data. The information required to represent a particular circuit design was found to be stored by the graphic interface program into anywhere from two to ten files, depending on the stage of development or analysis that the design was in. The remainder of this section describes the data elements found in each of those files.

Note that there is no field in any of these files to identify the design they belong to. This is because the design identifier was actually part of the file name. When a design was retrieved to be

worked on further, copies were made of any files beginning with the desired design identifier and renamed "temp." All changes made during the design session are made to these "temp" files.

The icons referred to in these files are the entities in the design that are connected to each other. They are uniquely identified by type and instance number. There are five types of icons: TTL, power, ground, clock, and input port. All icons except TTLs are considered to have only one pin, or connection point.

One of the first files created, "temp.loc", contained information about the space occupied by each icon including the dimensions of the graphic image and a location reference point (1:4-21,22). Figure 3 - 5 shows the data elements contained in "temp.loc."

LOC_X	LOC_Y	HIGH	WIDE	ICON

Fig. 3 - 5. Elements of "temp.loc" File

Another of the first files to be created, the "temp.icn" file contained further information used in the display of icons, such as the color and the title (1:4-22,23). The title could be either the number of the TTL or a two character input port identifier. Neither the user-specified position or the system-assigned position are the same as the location reference point contained in "temp.loc." Some redundancy was present, however, because the system-assigned Y-coordinate was always the same as the user-specified Y-coordinate. Figure 3 - 6 shows the data elements contained in "temp.icn."

ICON	TITLE	USER_X	USER_Y	SYSTEM_X	USER_Y	I_COLOR

Fig. 3 - 6. Elements of "temp.icn" File

The "temp.grf" file contained information for displaying connections as a series of twelve points, or ten line segments, together with the link identifier and color (1:4-23). This format not only limited the number of turns in any link to ten, but in those cases where fewer than ten turns were taken, ten turns were "fabricated" by filling the remaining fields with line segments of null length, starting and ending at the same point. Figure 3 - 7 shows the data elements contained in "temp.grf."

X_1	Y_1	...	X_12	Y_12	L_COLOR	LINK

Fig. 3 - 7. Elements of "temp.grf" File

The "temp.ckt" file contained further information to identify links as pin to pin connections (1:4-23,25). The comment fields (O_COMMENT, I_COMMENT) were found to be "for future use" and filled with strings of *'s. Figure 3 - 8 shows the data elements contained in "temp.ckt."

O_ICON	O_PIN	O_TITLE	O_COMMENT	LINK
...	I_ICON	I_PIN	I_TITLE	I_COMMENT

Fig. 3 - 8. Elements of "temp.ckt" File

The "temp.iot" file contained the information produced by the ICE program (1:4-25). All problems with a circuit design were categorized by ICE as being either a "questionable link" or a "missing connection" and this file could contain one or more of either one, both, or none of these problems. If there was some problem that prevented analysis however, the "temp.iot" file could also contain a short message indicating the cause of the problem. The error message would not be stored in the new database. Figure 3 - 9 shows the data elements contained in "temp.iot."

PROB_DESC	ERR_CODE

Fig. 3 - 9. Elements of "temp.iot" File

The "temp.in" file contained the input data stream assigned to each input port for use by the LOGSIM program (1:4-25,26). The only icon type found in this file is "input port" and because has only one connection point, no pin information is needed. Figure 3 - 10 shows the data elements contained in "temp.in."

ICON	INPUT

Fig. 3 - 10. Elements of "temp.in" File

The "temp.ind" file contained the same data elements as found in the "temp.in" file with the addition of the title of the input port

(1:4-26). This file was used to display LOGSIM input data in human-readable format. Figure 3 - 11 shows the data elements contained in "temp.ind."

ICON	TITLE	INPUT

Fig. 3 - 11. Elements of "temp.ind" File

The "temp.dis" file contained the TTL pins designated as monitoring points for use by the LOGSIM program (1:4-26,27). As was the case with the "temp.ckt" file, the comment field was designated as "future use" and always contained a string of *'s. Figure 3 - 12 shows the data elements contained in "temp.dis."

ICON	TITLE	PIN	COMMENT

Fig. 3 -12. Elements of "temp.dis" File

The "temp.out" file generally contained the output produced by the LOGSIM program for each of the monitored points specified in the "temp.dis" file (1:4-27). The output would be in binary, "1" and "0" format. If there was some error that prevented simulation however, the "temp.out" file could also contain a short message indicating the reason for simulation failure. The error message would not be stored in the new database. Figure 3 - 13 shows the data elements contained in "temp.out" when the simulation is successful.

ICON	TITLE	PIN	OUTPUT

Fig. 3 - 13. Elements of "temp.out" File

The tenth file, "temp.wav", contained a combination of the information contained in "temp.ind" and "temp.out" (1:4-27). The only difference is that the input and output fields are formatted for graphical waveform display instead of binary.

An eleventh file, "temp.txt" was found to be used during the design session to pass information to ICE (1:4-25). Because the ICE program was large and overlaid the graphic interface program, the presence of the "temp.txt" file was also used as a flag to indicate to the interface program whether it was just beginning execution or if it was, instead, returning from ICE. "Temp.txt" was never saved when a session was ended however, and is not considered here further.

Central Database. The original method of storing circuit and TTL information, as outlined in the preceding section, resulted in redundant data and wasted memory. The eight relations, whose organization is described in this section, were designed to store all of the data elements identified to this point, as well as the additional information required to provide a "circuit directory," as compactly as possible. The key fields in each relation are shown underlined in the figures. A complete description of the central database can be found in the source code file "circuit.sch."

TTL Data. The two relations defined to contain TTL data, TTLS and PINS, are described in the following section.

The TTLS relation contains data pertaining to TTLS as whole entities. It records the identifying number, the number of pins associated with it, its description and degree of fanout. FANOUT was not mentioned as a data element in the existing documentation, but was found to exist in the actual program. The dimensions of the TTL are not stored because they are calculated by the graphics interface program based on the NO_OF_PINS attribute. Figure 3 - 14 shows the data elements contained in the TTLS relation.

TTL	NO_OF PINS	TTL_DESC	FANOUT

Fig. 3 - 14. Elements of TTLS Relation

The PINS relation contains an entry for each pin of a TTL. Figure 3 - 15 shows the data elements contained in the PINS relation.

TTL	PIN	GATE	VALUE

Fig. 3 - 15. Elements of PINS Relation

Circuit Data. The six relations used to store information related to circuit designed are described in the following section.

The CKTS relation contains the new information required to support a circuit directory. Figure 3 - 16 shows the data elements contained in this relation. Some elements may not be used in the design environment currently, but are included for possible future use.

CKT	CKT_DESC	STATUS	OWNER	MAX_HIGH	MAX_WIDE

Fig. 3 - 16. Elements of CKTS Relation

In the original system, users specified an eight character filename. Here, it is reduced to six alpha-numeric characters but remains user-specified. This was considered preferable to using a system generated identifier, not only to simplify programming but also for flexibility. There are more definitions to the term "version" than just the one described by Batory and Kim (2) and the designer is free to use the identifier and description fields and devise his own system for keeping track of his designs.

The ICONS relation contains information pertaining to the icons in the circuit. Figure 3 - 17 shows the elements in this relation.

CKTS	ICON	TITLE	USER_X	USER_Y	SYS_X
...	LOC_X	LOC_Y	HIGH	WIDE	I_COLOR

Fig. 3 - 17. Elements of ICONS Relation

The dimensions of each icon is stored here and not in the TTLS relation because there are other types of icons involved, and the dimensions of each of them are calculated by the graphic interface program based on different criteria.

The LINKS relation contains information pertaining to the connections. Figure 3 - 18 shows the data elements in this relation.

CKT	LINK	O_ICON	O_PIN	O_COMMENT	I_ICON	I_PIN
...	I_COMMENT	X_1	Y_1	...	X_12	Y_12	L_COLOR

Fig. 3 - 18. Elements of LINKS Relation

Please note that, according to relational database theory, because there are a variable number of line segments associated with each link, this relation is not normalized and should be broken into two relations, one for links and one for line segments. To simplify storage and retrieval however, the decision was made to continue storing each link with a fixed number of line segments. The small amount of redundant data stored in the form of null length line segments is offset by the amount of overhead that would be required to maintain the second relation.

The IOUT relation contains the problem statements issued by the ICE program. Figure 3 - 19 shows the elements in this relation.

CKT	PROB_DESC	ERR_CODE

Fig. 3 - 19. Elements of IOUT Relation

The LIN relation contains the input data streams required by LOGSIM. Figure 3 - 20 shows the data elements included in this relation.

CKT	ICON	INPUT

Fig. 3 - 20. Elements of LIN Relation

And finally, the LOUT relation contains the output of each point monitored by LOGSIM. Figure 3 - 21 shows the data elements included in this relation.

CKT	ICON	PIN	OUTPUT	COMMENT

Fig. 3 - 21. Elements of LOUT Relation

Database Management System

Operations. As mentioned at the beginning of this chapter, the DBMS must perform all of the operations necessary to support the original system's functions. The following is a discussion of each of these functions, first those involving circuit data and then those involving TTL data, and the DBMS operations required to support them.

On Circuit Data. As could be expected, because the purpose of the system is circuit design, the majority of functions performed by the system relate to circuit data. A discussion of each of these functions can be found below.

Insert. When the original system "saved" a circuit design, copies of the TEMP files were made, renamed by the designer, and stored to whatever disk drive and directory he specified. Using the new central database, the insert operation reads information from a known place, the TEMP files, and stores it to a known place, the appropriate relations. For reasons discussed in detail in Chapter 4, the relations themselves are stored on the "A" drive. Only the information stored in the CKTS relation, needs to be obtained from the designer for input.

Delete. Deletion of circuit designs was not actually a function of the original system. The operating system command "Del <filename>.*" served the purpose. Using the new central database, however, this would not be possible because circuit designs are no longer stored separately. This function had to be added to the original system. The DBMS delete operation removes information from a known set of relations, again, stored on a known drive. Only the circuit identifier needs to be obtained from the designer and input to this operation.

Retrieve. When the original system "retrieved" a circuit design, working copies, or TEMP files, were made for each file bearing the designer-specified filename. To support the function of the original system, the DBMS must be able to perform the retrievals necessary to reconstruct these TEMP files. If this were the only retrieval function, the retrieve operation would need only know the circuit identifier, because the information is being copied from a known place to a known place, but with the central database other retrievals become necessary. For example, obtaining a directory of circuit designs was originally handled by the operating system. The command "Dir *.LOC" served this purpose. Again, as with the delete operation, this does not work with the new central database because each circuit design is not stored separately. This information must be retrieved. Still more retrievals of TTL data will be discussed in the next section.

Given that the results of the retrieval will be put in a known place, the information required to perform a general retrieval can be clearly pictured in the form of an SQL query. SQL is a well known,

commercially available relational database query language whose queries typically consist of three parts, or clauses (7:71):

Select : the data elements

From : the relations the elements should be taken from

Where : the conditions to be tested for

Analysis of the retrievals performed by the system showed that "select" always contains multiple data elements. Only in the retrieval required to reconstruct TEMP.CKT are tuple variables required to distinguish between multiple elements with the same name. Figure 3 - 22 shows that again with the exception of the retrieval to reconstruct TEMP.CKT, "From" always contains one or two relations. TEMP.CKT requires tuple variables to distinguish between multiple copies of the same relation. Finally, "where" was found to always contain either one or two entries. The first entry is either the TTL or circuit identifier, or a wildcard symbol to indicate a directory of all entries. The second entry is used in requesting TTL data pertaining to a specific PIN. Database entries are tested for equality with the values contained in the "where" clause.

Join Strategies. Further examination of Figure 3 - 22 shows that in all cases where data from more than one relation is required, the ICONS relation is one of those included. The joins are performed using the ICON identifier field(s) present in each of the relations being joined.

Modify. Throughout the design session, all changes made to a circuit are recorded only in the TEMP files not to the original, or "database" files. This is consistent with the trial-and-error nature of

TEMP FILE	REQUIRED RELATIONS
.LOC	ICONS
.ICN	ICONS
.CKT	LINKS, 2 X ICONS
.GRF	LINKS
.IOT	IOUT
.IN	LIN
.IND	LIN, ICONS
.DIS	LOUT, ICONS
.OUT	LOUT, ICONS

Fig. 3 - 22. Required Joins for Reconstructing TEMP Files

the design process. Only when the designer is satisfied with the design is it "saved" or inserted into the database. "Modify" is not required.

On TTL Data. Because the TTL database of the original system is static, the only function performed on TTL data was retrieve, either to get information regarding a single TTL, a single pin, or a TTL directory. For the protection of the TTL database, this remains the only function allowed within the context of the design environment. To increase the usefulness of the system, a separate utility program allows TTL data to be inserted, deleted, and modified. These operations are not part of the DBMS designed to support the functions of the original circuit design environment, however, and will not be considered here.

Data Manipulation Language. The data manipulation language (DML) is the means by which the users, in this case the three programs that make up AFIT's circuit design environment, invoke the operations supplied by the database management system. Because the three using programs are all written in the C programming language, it was most convenient to design a C-based DML, one that could be embedded directly

in the code of the using program and compiled normally, without the assistance of a preprocessor. The format settled upon, that of a parameterized subroutine call, is based on examples given in C Database Development by Al Stevens (12:135).

The DML consists of a total of three subroutine calls, one for each of the operations supported by the DBMS. The remainder of this chapter is dedicated to a detailed discussion of each of them.

Insertion. The format of the insertion subroutine call is:

Insert(Address)

The single parameter, Address, is the address of a data structure containing information to identify the circuit being inserted. With the exception of the last two fields, MAX_HIGH and MAX_WIDE, the information in this structure is obtained from the designer. In the example shown in Figure 3 - 23, MAX_HIGH and MAX_WIDE are left blank because they are not currently being used by the system. All other information relating to the circuit is assumed by the DBMS to be in the TEMP files.

```
#include "circuit.c1"
main()
{
static struct ckts a_circuit = {"CKT_ID",
                                "The description can be 65 char",
                                "Status ",
                                "Designer field is 25 char",
                                "    ",
                                "    "};
insert(&this_circuit);
}
```

Fig. 3 - 23. Example of Insert Operation

Delete. The format of the deletion subroutine call is:

Delete(CKT_ID)

The single parameter, CKT_ID, is the six character alphanumeric identifier of the circuit to be deleted. Figure 3 - 24 shows an example of the delete operation.

```
main()
{
static char this_circuit[] = "CKT_ID";

delete(this_circuit);
}
```

Fig. 3 - 24. Example of Delete Operation

Retrieve. The format of the retrieval subroutine call is:

Query(Select,From,Where)

The three parameters, Select, From, and Where, are each addresses of lists, or arrays. The lists contain data elements, relation names, and test conditions, respectively. The last entry in each list, as shown in the example at Figure 3 - 25, is a "terminator" value.

Summary

The data elements used by the original system were identified and organized into a central database consisting of eight relations: TTLS, PINS, CKTS, ICONS, LINKS, IOUT, LIN, and LOUT. The operations necessary to support the original system's functions were found to be insertion, deletion, and retrieval. A simple data manipulation language consisting

of three commands, one for each of the operations supported, was constructed. Based on the C programming language, the commands are formatted as parameterized subroutine calls.

```
#include "circuit.cl"
main()
{
static int  select[] = {ICON,TITLE,INPUT,0};
static int  from[]   = {LIN,ICON,-1};
static char *where[] = {"CKT_ID","0");

query(select,from,where);
}
```

Fig. 3 - 25. Example of Retrieve Operation

IV. Detailed Design

The database management system constructed for AFIT's circuit design environment uses the record-manipulating functions of Cdata as the basis for implementing the operations defined in the previous chapter. Cdata, "a library of C functions that do for you what a DBMS would do," is described in detail in C Database Development (12:53). The primary advantages of using Cdata over a commercial DBMS are given as: "the efficiency of the resulting system, the absence of licencing costs, the control you have over the DBMS and its destiny, and the open architecture of the database files and indexes" (12:53). The database files and indexes used by the Cdata functions, and ultimately by this circuit design DBMS, are described in this chapter, along with a discussion of where the various files are stored and the overhead required.

Storage Structures

According to Korth and Silberschatz, storing each relation in a separate file is an approach "well suited to database systems designed for personal computers" (7:248). It is this approach that Cdata takes.

Each relation is stored in its own file identified by <relation name>.DAT. The file consists of a header record and a series of fixed length data records. When the database is first initialized, naturally, only the 10 character long header record is present (12:119).

The header file consists of three fields: a pointer to the first record in the file, a pointer to the next available position, and the

length of the data records. The pointers are of type LONG and are capable of referencing up to 2^{32} records (12:118). This number is more than sufficient for the purposes of this DBMS.

The length of the data records is determined by the lengths of the individual data elements that make up the relation. "Cdata stores data values in files as null-terminated ASCII strings, regardless of the data types" (12:119). Because every data element is stored as a character string, even those defined in the schema as being of numeric type, the length of the record can be calculated by summing the individual element lengths and adding one for the null character that terminates each element in the relation.

The high level Cdata functions for maintaining the database, those used in the implementation of the circuit design DBMS operations, can be found in the source code file "database.c" (12:134). Lower level functions for maintaining the data files are grouped together in the source file "datafile.c" (12:151).

Access Structures

Cdata provides only a single access mechanism, a balanced tree structure, or B-tree index of key values (12:119). Given the nature of the operations performed on the circuit design database, this is not a particularly good access mechanism, but it was readily available and time constraints precluded the development of a more efficient mechanism.

The key-value index performs efficiently only in those cases where directories of TTLs or circuits are requested, or when the information

regarding a single TTL or a single pin of a TTL is requested. In the majority of cases, however, we aren't looking for a particular record from a relation, but rather all of the records pertaining to a particular circuit identifier. Because we cannot supply the whole key value we are forced to do a sequential reads of the relations, testing for a match to the circuit identifier.

The index file associated with each relation is identified by <relation name>.X01. Each file consists of one header record and a variable number of node records. The number of node records present in the index file depends on the number of data records present in the corresponding data file. The number of data records each node can support is given by the equation:

$$m = ((NODE - ((sizeof(int)*2) + (ADR * 4))) / (len + ADR)) \quad (9:161)$$

In this equation, len, the length of the data record being indexed, is the only true variable. The number of data records supported by each node varies inversely with the size of the data record.

Header records and node records are each 512 characters long. Again, just as with the data files, when the database is first initialized only the header record is present.

The header record contains the following information: the pointer to the root node, the length of the key, the maximum number of keys per node, a pointer to the last node that was deleted, a pointer to the next available node that can be added, a "lock" status flag, a pointer to the leftmost node, and a pointer to the rightmost node (12:159).

The node record contains the following information: a "nonleaf" flag, a pointer to the parent node, a pointer to the left sibling, a pointer to the right sibling, a count of the keys pointed to, a pointer to the node containing the preceding key values, the key values and their associated pointers, and finally, a spill area for insertions (12:159).

The definition of the B-tree structure and the functions for maintaining B-trees are grouped together in the source file "btree.c" (12:158).

File Location

The two entities involved in circuit design, TTLs and circuits, have characteristics requiring that they be located apart from one another. The TTL database, on one hand, will normally be much smaller than the circuit database. It is required to be present for every user, and, not subject to frequent change, it is protected to the extent that it cannot be modified without the separate utility program. The circuit database, on the other hand, can conceivably become quite large. And, unlike the TTL database, there are no security measures in place that would protect the circuit database and still allow access to it. It is desirable therefore, for each user to maintain individual control over his designs.

The solution dictated by both memory constraints and access considerations is to have the TTL database, both data and index files, reside in internal memory, on the C drive, while the circuit database, again both data and index files, be maintained on one or more floppy

diskettes. One drawback to this solution, however, is that it effectively limits the size of circuits that can be designed on the system to one that will fit on a floppy diskette.

The TTLS and PINS files, populated with the same 34 TTLs used in the original system, are present on the C drive at startup. The DBMS expects the circuit files to be present on the A drive before any operations will occur. If no circuit files exist, another database utility program, Makedb, must be run to create the empty files on the floppy diskette. A user's guide describing the operation of all of the database utility programs is included as an appendix to this thesis effort.

Summary

Using the data and index file structures discussed in this chapter, the TTL database, populated with the same 34 TTLs present in the original system, occupies 23,714 characters or bytes of internal memory. The overhead required to maintain the circuit database on the floppy starts at 3132 bytes empty, and continues to grow as more records are added. The size of circuit that can be inserted or "saved" using this DBMS is limited by the amount of memory on the floppy diskette. And, while the effectiveness of the DBMS is unaffected, its efficiency is hampered by the access mechanism being used.

V. Implementation

General Description

Figure 5 - 1 illustrates the overall implementation of the new central database and its associated DBMS and utility programs.

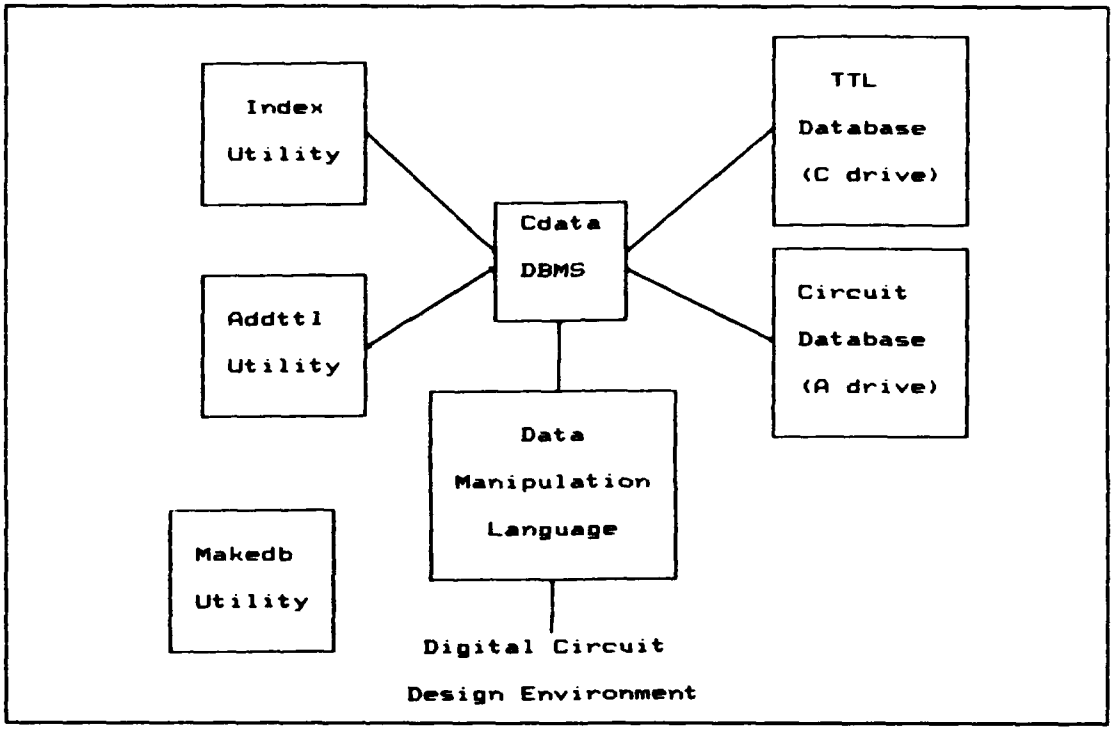

Fig. 5 - 1. Overall Implementation of the New DBMS

On the right-hand side of the figure, the TTL and circuit portions of the database are shown to be stored in physically separate locations. The data contained in both portions of the database is accessed directly, on a record-by-record basis, by the Cdata functions. The Cdata functions, in turn, are invoked by the Data Manipulation Language (DML) module. The DML module in this figure consists of the executable

functions invoked by the DML commands described in Chapter III. It is the DML module that allows the digital circuit design environment to think that it is dealing with an undivided database of TTLs and circuits instead of collections of records in physically separate locations. The functions in the DML module invoke Cdata functions to accomplish their work. Appendix B contains a short description of each of the Cdata functions invoked directly by functions in the DML module. On the left-hand side of the figure are the database utility programs, executed from outside the design environment. Both the Index and Addttl utilities access the databases through Cdata functions. The Makedb utility does not require any access to the data.

The remainder of this chapter is a more detailed look at the DML module functions or DBMS operations, and the database utility programs.

DBMS Operations

Insert. The DBMS insert operation refers only to the insertion of circuit designs into the database. TTLS are inserted only by using the Addttl utility. Figure 5 - 2 illustrates the general flow of the insert operation. Insert begins by opening the CKTS relation. Once open, the CKTS relation is checked to see if there is a match to the circuit identifier passed as input. A match indicates that the circuit design had been previously saved and was modified during the current design session. If a match is found, the CKTS relation is closed and the delete operation is invoked to clear the database of the old version of the design. When the delete operation is completed, the CKTS relation is reopened. If the circuit does not already exist in

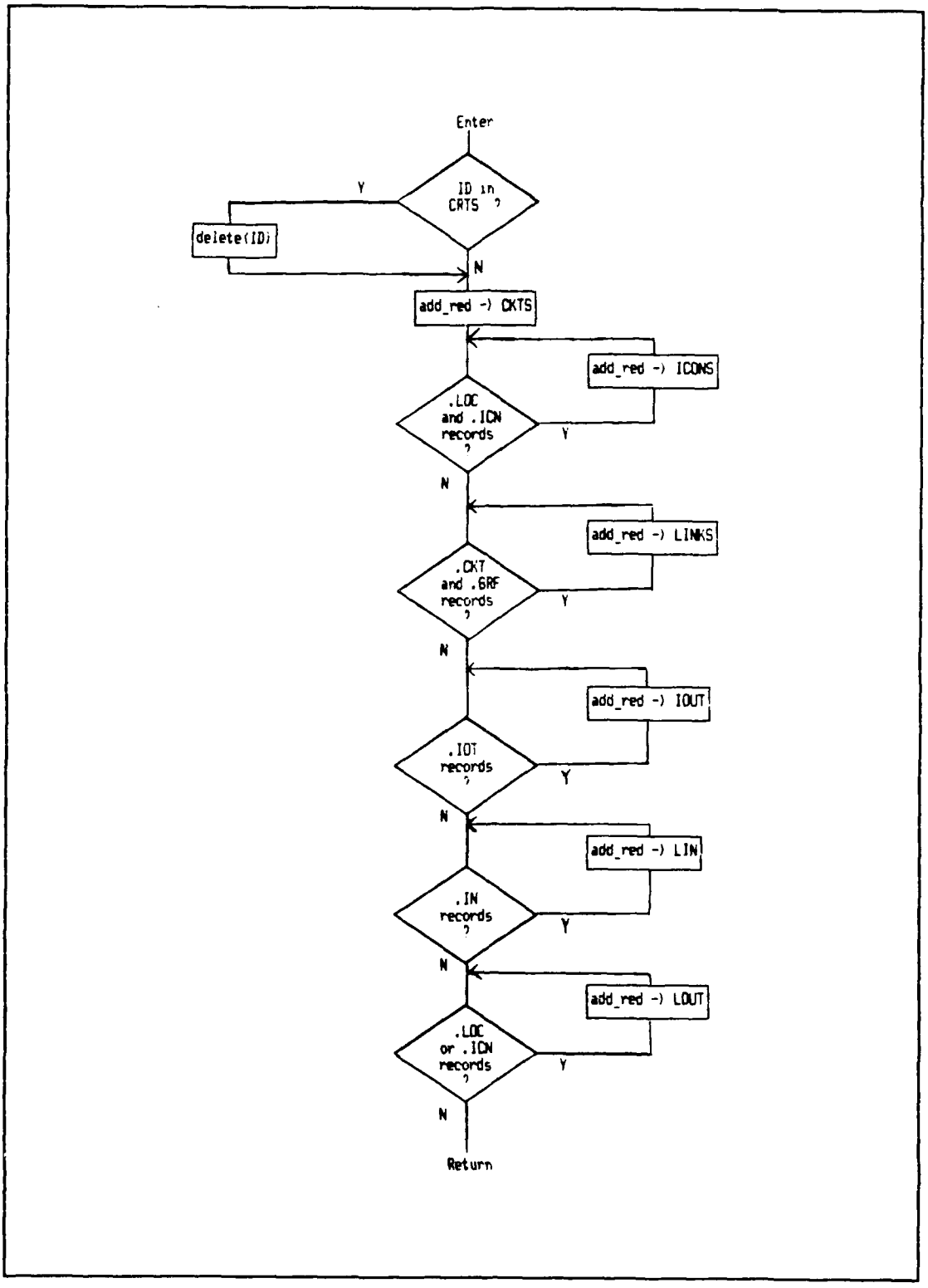

Fig. 5 - 2. Implementation of the Insert Operation

the database, the CKTS relation would remain open. From this point on processing is the same for both cases.

Insertion of circuit data begins with an entry into the CKTS relation. Data required for the CKTS relation must be provided, in the format of that relation, at the address given as an input parameter. This data must be reinserted each time a circuit is saved even if none of the values have changed. The values are transferred directly from the using program's area into the CKTS file. The CKTS relation is then closed.

Insertions into the ICONS relation begin by opening the ICONS relation along with TEMP.LOC and TEMP.ICN. These two TEMP files are the minimum required input data. If either of the files is not present, the CKTS record already entered is removed and the operation terminated. Assuming they are present, values are read from the two TEMP files into a holding structure in the database area. Before writing the new record to the ICON file, a check is made to ensure that the records being read from the two TEMP files are joined correctly. This, and all other instances of "proper join" checking mentioned in this section are included only to safeguard the integrity of the database, the graphic interface program creates and manipulates the TEMP files in such a way that the records should always be in the same order. When end of file is reached, the ICONS relation and the TEMP files involved are closed.

In the same manner, if both TEMP.GRF and TEMP.CKT are present, insertions are made into the LINKS relation by combining values read from those files.

If TEMP.IOT is present, it along with the IOUT relation are then opened. If the first thing read from TEMP.IOT is "No output from ICE", nothing will be written to the database. Otherwise, values are read from TEMP.IOT and transferred to the holding area. The values for the ERR_CODE field are determined by the insertion procedure. The first character of the PROB_DESC field is forced to "A" or "Z" depending on this ERR_CODE, to ensure that all "questionable link" errors will appear before "missing link" errors during subsequent retrievals. Records are written to the IOUT file as they are completed. When end of file is reached, the IOUT relation and the TEMP.IOT file are closed.

Next, insertion into the LIN relation begins. If the TEMP.IN file exists, it is opened along with the LIN relation. Data is read in to the holding area and written out as each record fills. When the end of file is reached, the LIN relation and the TEMP.IN file are closed.

If TEMP.OUT exists, it is opened and the first few lines are checked. If the file contains the statement "No output", TEMP.OUT is closed and a check is made for TEMP.DIS. If TEMP.OUT does contain data, it is assumed that TEMP.DIS is also present, as there must be monitoring points specified before output signals will be recorded. If either TEMP.DIS or TEMP.OUT contain data, the LOUT relation is opened. Values are read from the open TEMP files into the holding area. If only TEMP.DIS is present, a flag will be placed in the OUTPUT field to indicate "No output." If both TEMP files contain data, a "proper join" check is made before writing the new record to the LOUT file. Records are written to the LOUT relation as they are formed. When the end of file is reached, all open files and relations are closed.

At this point the insert operation returns control to the using program with an indication of successful completion. It is left to the using program to dispose of the TEMP files.

Error Handling. Errors that may occur during the insertion of a circuit design are: corrupted index, inability to open database files, invalid data received either in the form of missing files or improper joins, and finally, insufficient space. Each of these errors causes the operation to terminate and return a specific error code to the using program. In the case of an "insufficient space" error, all of the insertions made to that point are undone by calling the deletion procedure before returning to the using program.

Recovery Considerations. If a power failure should occur during the insertion procedure, only that part of the circuit successfully inserted prior to the failure would be saved. By inserting into the database relations in the proper order, that being CKTS, ICONS, LINKS, IOUT, LIN, and LOUT, the integrity of the database is nonetheless ensured no matter where it may be interrupted. Nothing will exist without a circuit to relate it to, no links will be between icons that don't exist, and no LOGSIM output will be present without the input it is based on. In no case would data exist that the designer was not aware of. For example, even if only the CKTS record was inserted the circuit would still appear in the circuit directory even though the design would not be available.

To recover, the designer would have to run the index utility and then, reentering the design environment, begin again by retrieving a backup copy of the circuit. Because the first thing the insert

operation does is delete all traces of a circuit design with the same identifier, that backup must either have been stored on the same floppy with a different circuit identifier or on a separate floppy.

Flexibility vs Ease of Use. A drawback to this implementation of the insert operation is that it is highly dependent on the structure of the TEMP files. A more flexible alternative, considered early on, would have placed the burden of converting TEMP file format into database format on the program responsible for the TEMP files. The insert command would simply place the data supplied into the relation specified. While flexible, this would require the using program to have extremely detailed knowledge of the database structure when it should only be concerned with inserting a circuit. The implementation described in this section was decided upon because, although not as flexible, it frees the using program from unnecessary detail and shifts the burden of maintaining database integrity back to the database management system where it belongs.

Delete. The DBMS delete operation, like the insert operation, refers only to the deletion of circuit designs from the database. The TTL database is modified only by using the Addttl utility program. Figure 5 - 3 illustrates the general flow of the delete operation.

Delete begins by opening all of the database files pertaining to circuit data. It is assumed that the using program would not be asking for the deletion of a non-existent circuit, and therefore no preliminary check of the CKTS relation is done to confirm its existence. Sequential searches are made through LOUT, LIN, IOUT, LINKS, and ICONS in that order. To make the search as efficient as

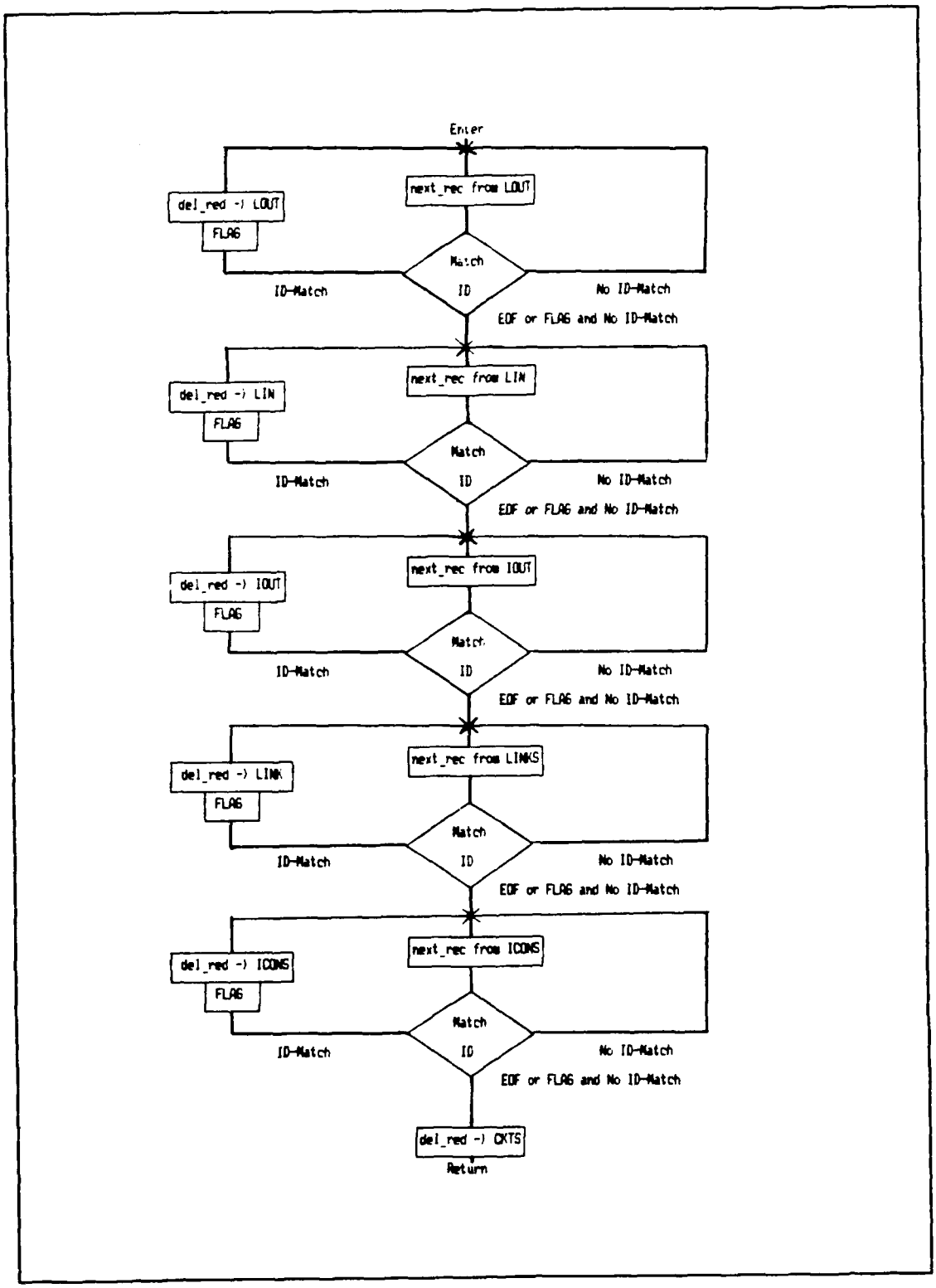

Fig. 5 - 3. Implementation of the Delete Operation

possible, a flag is set to indicate when a match is found in each relation. Once the flag is set, the search can be discontinued as soon as the next non-matching record is found because all records pertaining to the same circuit would be located together by key value. Once the sequential searches are finished, the CKTS relation is checked. When this last record has been deleted, the database files are closed.

At this point the delete operation returns control to the using program with an indication of successful completion.

Error Handling. The only errors possible are corrupted indexes or the inability to open the database files. Should either occur, the operation would terminate and a specific error code would be returned to the using program.

Recovery Considerations. Records are deleted from the database in the opposite order of their insertion. This, as was discussed in the previous section, will ensure the integrity of the database. If a failure should occur in the middle of a deletion, a circuit would still be listed in the circuit directory until the last record pertaining to it was gone.

Retrieve. The DBMS retrieve operation, called query, is applicable to both circuit and TTL data. Figure 5 - 4 illustrates the general flow of the retrieve operation or query.

Query begins by opening all of the database files contained in the list pointed to by the "from" parameter. The standard DBMS output file, "Db_reply", is prepared to accept the output by opening it in the write mode. This automatically erases the previous contents. A check is then made to see if the requirement is for a directory. A directory

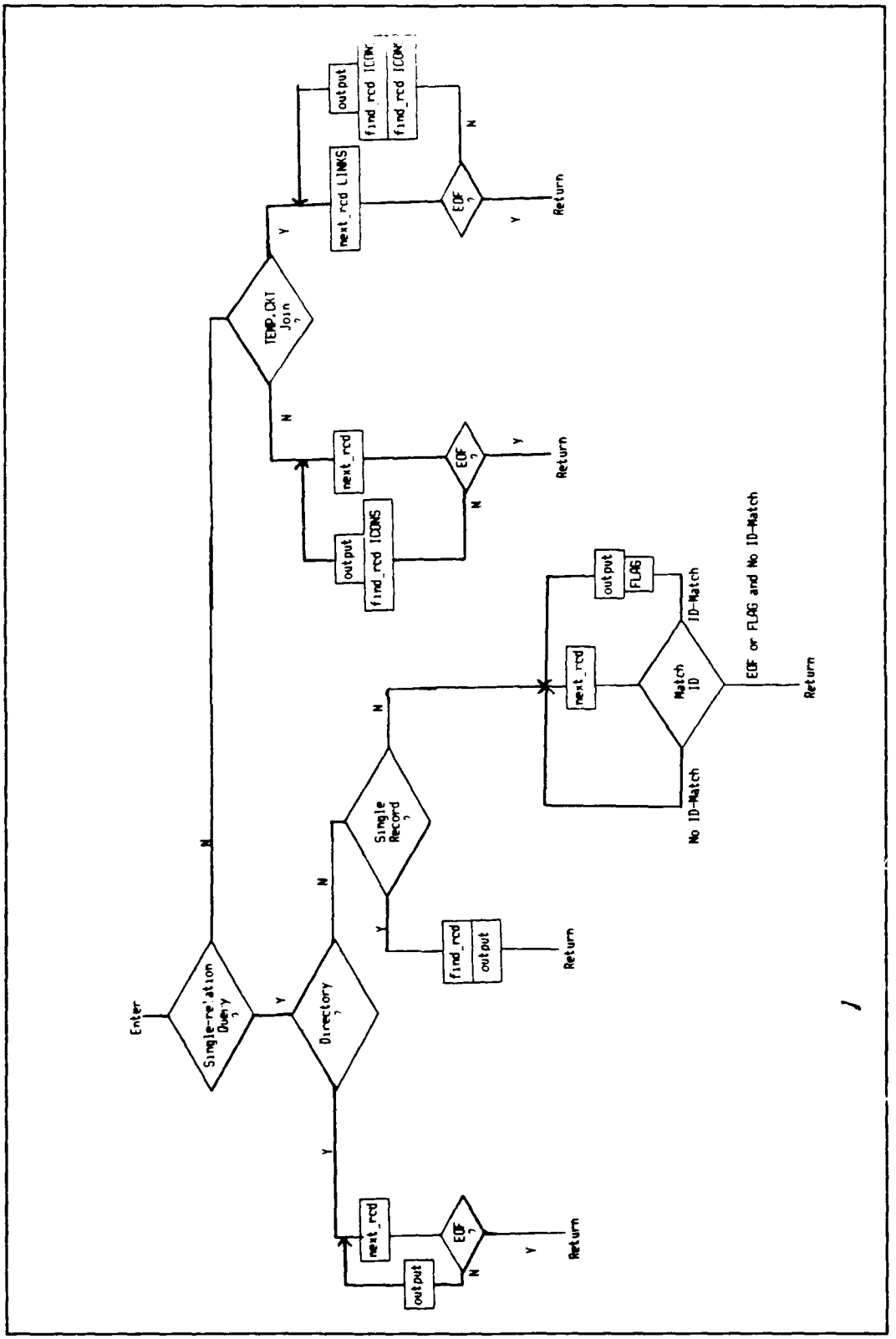

Fig. 5 - 4. Implementation of the Retrieve Operation

request is signaled by a "*" in the first element in the "where" list and causes one record to be output for each entry in the single relation specified. The implementation does not recognize joins in this case. While requests for only TTL and circuit directories are expected, it works equally well on all relations.

If the request involves a single relation but is not a directory request, a check is made to see if the relation involved is either TTLS or CKTS. If that is true, or if the "where" clause contains the second value needed to complete the concatenated key of all but the LOUT relation, direct access of a single record is carried out. Otherwise, the operation performs a sequential search the specified relation, checking the first field against the circuit or TTL identifier supplied in the "where" list. If a match is found the selected data elements are transferred to an output buffer.

If a request involves two relations, the second relation must be ICONS. If the first relation specified is LINKS, a special routine to retrieve TEMP.CKT data is triggered. Exactly why a special routine is required is discussed in the "problem area" section that follows. What the special routine involves is described here. A sequential search through the LINKS relation is made, checking for the circuit identifier specified in the "where" clause. If a match is found, all of the fields required, with the exception of the TITLE fields in the second and seventh positions, are transferred to the output buffer. The first title is found by concatenating the first and third fields of the LINKS relation to form the unique key necessary to directly access the first ICONS record. The third field of the first ICONS record is transferred

to the second field of the output buffer. The second title is found by concatenating the first and thirtieth fields of the LINKS relation to form the unique key necessary to directly access the second ICONS record. The third field of the second ICONS record is transferred to the seventh field of the output buffer. Please note this special routine is highly dependent on the format of LINKS, ICONS, and TEMP.CKT and must be modified if any of them change.

If it is a two relation join but the first relation specified is not LINKS, a sequential search will be made through the first relation checking for a match with the circuit identifier specified in the "where" clause. If a match is found, the first and second fields of that record are concatenated together to form the unique key needed to directly access the joining ICONS record. The specified fields are then transferred to the output buffer from the records of both relations. Note that the only relations this join will work with are LIN and LOUT, the only ones which have the ICON field in the second position of the relation.

No matter what type of retrieval request was made, the last step of the operation, before returning control to the using program, is to close the database files and "db_rely." At this point the retrieve operation returns control to the using program with an indication of successful completion.

Error Handling. Errors that could occur during the course of the retrieve operation are: corrupted indexes, the inability to open the database files, inability to open the output file, and, though not technically an error, the failure to find the requested data. The

operation will terminate at the point any of these conditions are detected and the specific error code for the problem will be returned.

Recovery Considerations. Because the retrieve operation does not modify the database, no recovery considerations are necessary. If a power failure were to occur during a retrieval, the index utility would have to be run and then, upon reentering the design environment, the retrieval could simply be reissued.

Problem Area. As mentioned in Chapter III, the retrieval required to reconstruct TEMP.CKT is the most complex, the only one requiring the use of tuple variables to distinguish between multiple data elements with the same name and multiple copies of the same relation. Cdata, unfortunately, does not support the use of tuple variables and does not allow duplicate names to be present in either the "select" or "from" lists. Duplicate names in the "from" list cause problems because Cdata tries to open the same relation file twice. This was handled by listing ICONS only once in the "from" list. Duplicate element names cause problems for Cdata's generic routines for transferring fields of data. The transfers are made based on the positions of the data element names in the "select" list, and are confused by duplicate field names, so the only solution was to create a special routine that bypassed Cdata's generic routines and move the data "by hand" in this one case. Admittedly, this is not an ideal situation, but if the format of the TEMP files remains stable, it should not cause any problems.

Associated Format Conversion Routines. The requirement of this DBMS, to support the system as it originally existed, means

reconstructing the TEMP files it is used to working with. While the retrieve operation alone is capable of supplying all of the data, it is not in the business of formatting and punctuating. Every TEMP file, with the exception of TEMP.LOC, contains some "non-data" characters, usually for purposes of improving the display. To support the system as it existed, a set of conversion routines were written to format the data supplied by the retrieve operation in "db_reply" back into TEMP files. While these are not technically part of the data manipulation language, they can be found, along with the code implementing the insert, delete, and retrieve operations, in the source code file "dml.c."

Utility Programs

The database utility programs, Index, Addttl, and Makedb, are not part of the DML used by the circuit design environment and are executed only when outside the environment. A user's manual covering all three programs can be found in Appendix A.

Index. The Index utility is a modified version of Index function provided by Cdata (12:214). It is modified only to the extent that it will index circuit data on the floppy drive and TTL data on the internal drive. The Index utility is necessary for those times when power is interrupted during a database manipulation. Any database files open at the time of a power failure will be "locked" until they are rebuilt using this utility.

Addttl. Addttl is a modified version of QD, the Data Entry and Query program provided as part of the Cdata toolset (12:200). It is

modified to the extent that is allows access only to the TTLS and PINS relations. It is intended only for the purpose of adding TTLs to the database, but it is recognized that when people input data, there is always the possibility of errors. For this reason, the ability to modify and delete are also included. This program relies on the user to maintain the accuracy and integrity of the database and should therefore be used only by a responsible individual who can then copy the updated files (TTLS.DAT, TTLS.X01, PINS.DAT, PINS.X01) to the student design stations. This program should not be left on student machines.

The program allows the insertion of all the information needed to describe a new TTL with the exception of its executable function. This means that circuits can be designed using the new TTL and the design can be checked by the InterConnect Expert, but it cannot be simulated by LOGSIM. Suggestions for how this capability might be added are discussed in Chapter VII.

Makedb. Makedb is a batch file that copies the empty data files (CKTS.DAT, ICONS.DAT, LINKS.DAT, IOUT.DAT, LOUT.DAT, LIN.DAT) and their associated index files (CKTS.X01, ICONS.X01, LINKS.X01, IOUT.X01, LOUT.X01, LIN.X01) from internal memory, where they were created on system startup, to the floppy diskette residing on the A drive. It is included for the purpose of preparing an empty floppy disk to accept circuit design information. If any of the circuit database files existed on the floppy prior to running this batch file, the information contained in them will be wiped out.

Summary

The chapter reviewed in detail the implementation of each DBMS operation and utility program. The implementation of the insert operation traded flexibility for ease-of-use and is highly sensitive to changes in the format of the TEMP files or the database relations. Another area of format sensitivity is in the retrieve operation where the lack of support for tuple variables required one retrieval to be "hard-coded".

The majority of the development work was done using a C-86 compiler before finally changing over to Microsoft C. In converting, it was discovered that the two compilers are not completely compatible in the way they read and write to files. The source code has been tailored to work with the Microsoft compiler.

VI. Testing and Results

Testing

The tests used to evaluate the centralized CAD database management system (DBMS) fall into two categories, those intended to test functional capability, and those intended to measure its efficiency. A brief description of each of the specific tests conducted can be found in the following sections.

Functionality Tests. A separate test set was designed for each design function involving the database. Descriptions of those test sets follow.

Inserting Circuit Data. The test set for this function consisted of saving circuits designs in all stages of development and testing. They included: designs that contained only icons, complete designs containing icons and links, designs with and without design errors, with and without LOGSIM inputs, designs that had been analyzed by ICE, and/or simulated by LOGSIM, or neither. It included designs that had been previously saved and modified, and new, previously unsaved, designs. The test set also included attempting to save to a non-existent floppy, a floppy with insufficient space, a floppy with missing database files, and interrupting the power in the middle of the "Save" procedure.

Deleting Circuit Data. The test set for this function consisted of deleting all the different types of circuit designs that had been used to test the "Save" capability and included interrupting the power in the middle of the "Delete" procedure.

Retrieving Circuit Data. The test set for this function consisted of retrieving all the different types of circuit designs that had been used to test the "Save" capability. It also included attempting to retrieve a design that did not exit and attempting to retrieve when there was no floppy diskette in the A drive or when the floppy present did not contain one or more of the database files.

Retrieving TTL Data. The test set for this function consisted of retrieving information for a directory of all TTLs in the database, and retrieving information about an individual chip.

Modifying the TTL Database. The test set for this function consisted of using the Addttl utility program to enter data for a new TTL, modify that data, and then delete it, repeating the "Retrieving TTL Data" test set after each change to confirm the presence of the new information.

Preparing Diskettes for Database Use. The test for this function consisted of using the Makedb batch file to prepare a floppy diskette for database use. It included: attempting to prepare a non-existent diskette, an unformatted diskette, one with insufficient space for even empty database files, one that already contained database files, and one that did not.

Efficiency Tests. The tests used to determine the efficiency of the DBMS again fell into two categories: space requirements and response times. Descriptions of the specific tests conducted in each category follow.

Space Requirements. The amount of space occupied by the various components of the database and DBMS was analyzed from the

standpoint of location, looking first at the internal memory required and then at the external.

Internal Memory. The DBMS components taking up internal memory are: the DBMS object modules, the TTL database, and the utility programs. While the DBMS object modules will not necessarily be maintained in internal memory once they have been linked with the circuit design environment, they are the only objective measure of DBMS size. Obtaining the percentage of executable code size directly attributable to the DBMS would be difficult because any changes made to the circuit design environment to interface it with the new database would also have some effect on its executable size.

External Memory. External memory is occupied by circuit data files and their associated index files. A comparison was made of the amount of memory required to store varying numbers of a single circuit design using the original system and the new central database. For the original system, this was the sum of the space occupied by each file of each circuit design. For the central database system, this was the sum of the space occupied by the circuit database files (CKTS.DAT, ICONS.DAT, LINKS.DAT, LIN.DAT, LOUT.DAT, IOUT.DAT) and their respective index files. The comparison began with empty databases and continually accumulated data.

Response Times. The response time of each DBMS operation was measured as the time spent executing the applicable procedure, from the call to the return.

Inserting a Circuit. The test set consisted of circuit designs in varying sizes, as measured by the total number of database

records they generate. Each circuit was stored first to an empty database and then again to a partially loaded database.

Deleting a Circuit. The test set consisted of selected CKT identifiers, such that the search time for the particular circuit's records would vary from first found, to last found and therefore requiring a search through the length of each relation. Circuit size remained constant throughout the test. The test was run twice, first with the database half full, and then again with it completely full.

Retrieving a Circuit. The test set and procedure in this case is the same as that described for "Deleting a circuit." Only the operation being performed was changed.

Retrieving a Single TTL Record. The test set consisted of selected TTL identifiers, such that the identifiers ranged from the highest to the lowest value stored. Each TTL was retrieved from the same database of 34 TTLs.

Retrieving a Single Pin Record. The test set consisted of the first pin of each of TTL identifiers used in "Retrieving a Single TTL Record." Each pin record was retrieved from the same database of 34 TTLs.

Retrieving TTL Directory Information. The test consisted of a single request for a directory TTLs. The number of TTLs in the database for the test was 34.

Modifying the TTL Database. This was not measured because the speed of the Addttl utility program is operator dependent.

Preparing Floppy for Database Use. The test set was a single run of the Makedb utility program.

Results

Functionality. The TEMP files used for processing by the graphics interface program were able to be stored in the database, retrieved, reformatted, and used again just as if they were original TEMP files. Only in the cases where TEMP.OUT and TEMP.IOT originally contained notifications of "No output" are those TEMP files not reconstructed upon retrieval. It would appear to the design environment as if LOGSIM and ICE had never been run against the circuit being retrieved. The format of the files returned by the central database vary from those originally stored in one minor way, the order of the records within the files is changed because the database returns records in ascending order of their key values. This variation has not been found to have any effect on the functionality of the circuit design environment.

Efficiency.

Space Requirements.

Internal. The DBMS object modules, "ts.lib", "dml.obj", and "dbdefine.obj", take up a total of 48 Kbytes of memory. Keep in mind, however, this amount does not translate directly into the space required for the executable code. The TTL data and index files, loaded with 34 TTLs, take up 24 KBytes of memory. The empty circuit data and index files maintained in internal memory for use by the Makedb utility program take up 3 KBytes. And, finally, the utility programs themselves, Index.exe, Addttl.exe, and Makedb.bat, together occupy 51 KBytes. Internal memory requirements then total 126 KBytes.

External. Figure 6 - 1 compares the amount of memory used to store designs on both the original and the new system.

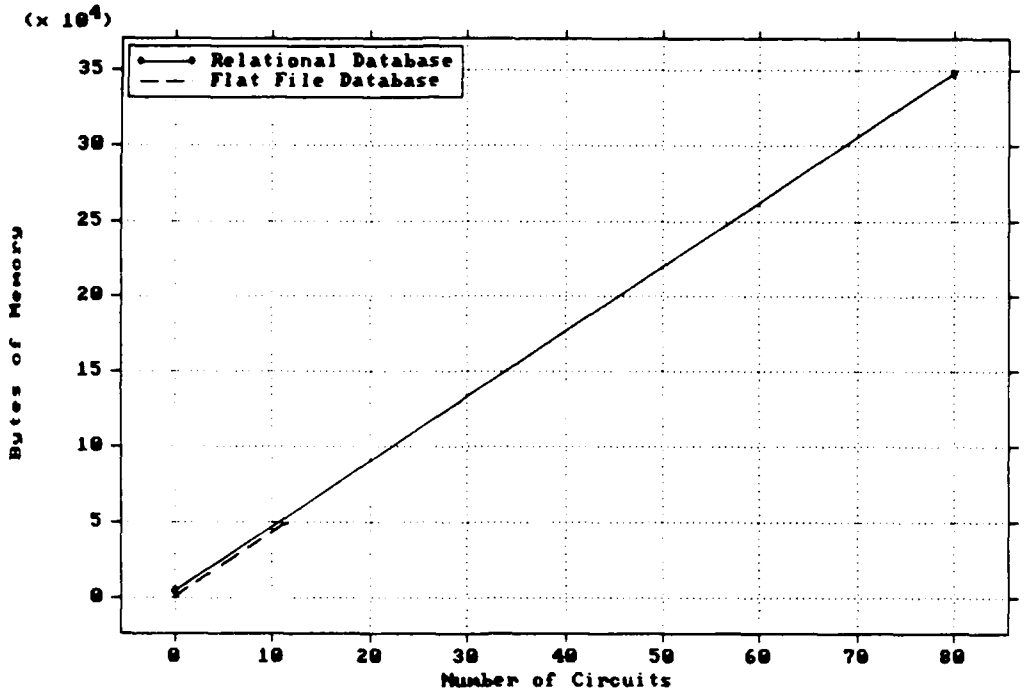

Fig. 6 - 1. Comparison of Circuit Design Memory Requirements

The central database system is at a slight disadvantage, for the first 11 circuits, because of the index files it must maintain. Beyond that point, the central database is able to make much more efficient use of the space because the data is organized into fewer and larger files. In one directory on the floppy diskette, there are only spaces to maintain a maximum of 112 files, and with 10 files per circuit for the original system, 11 circuits fill the directory and prevent any more from being added, even though a large percentage of the memory space remains unused.

Response Times. The response times for the insertion, deletion, and retrieval of circuit designs are shown in Figures 6 - 2, 6 - 3, and 6 - 4, respectively. Please note the time recorded a retrieval is actually the time required for six retrievals, one for

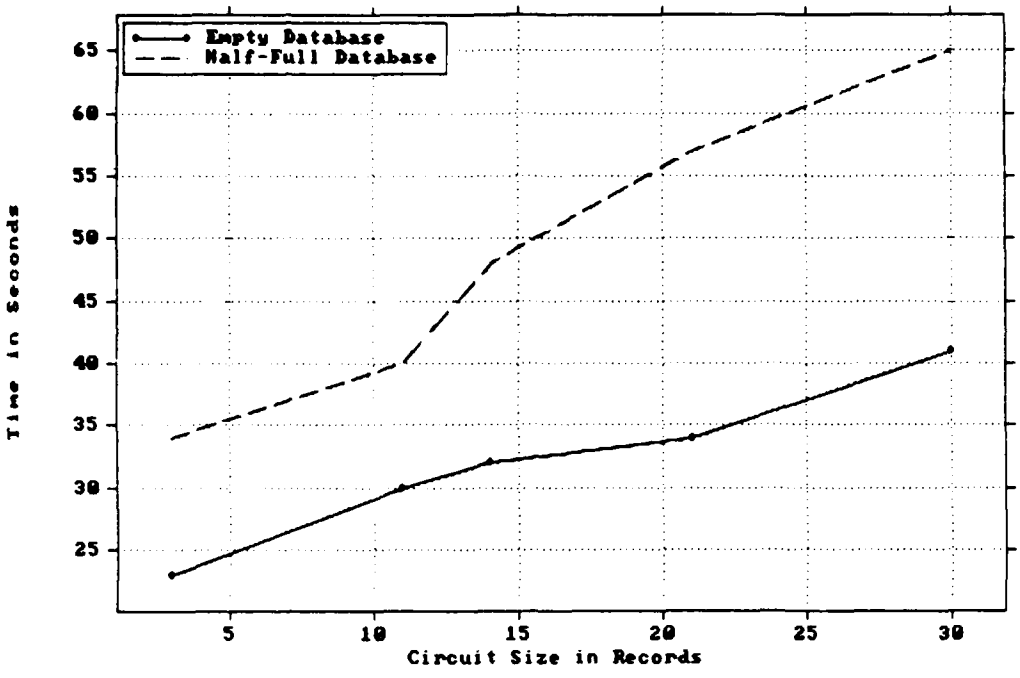

Fig. 6 - 2. Response Times of Circuit Insertion

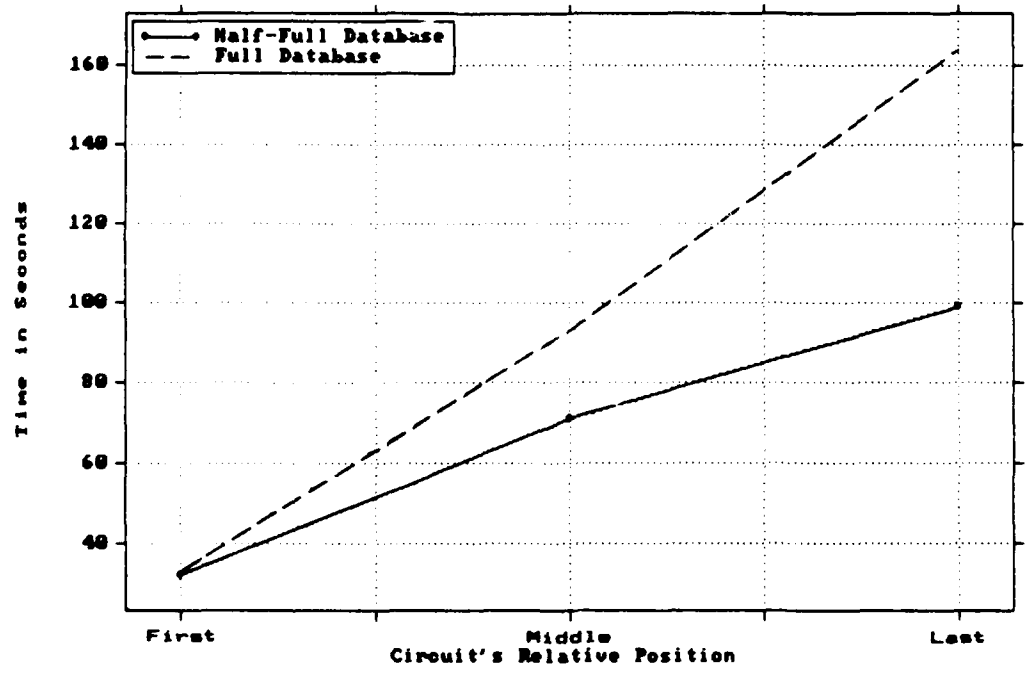

Fig. 6 - 3. Response Times of Circuit Deletion

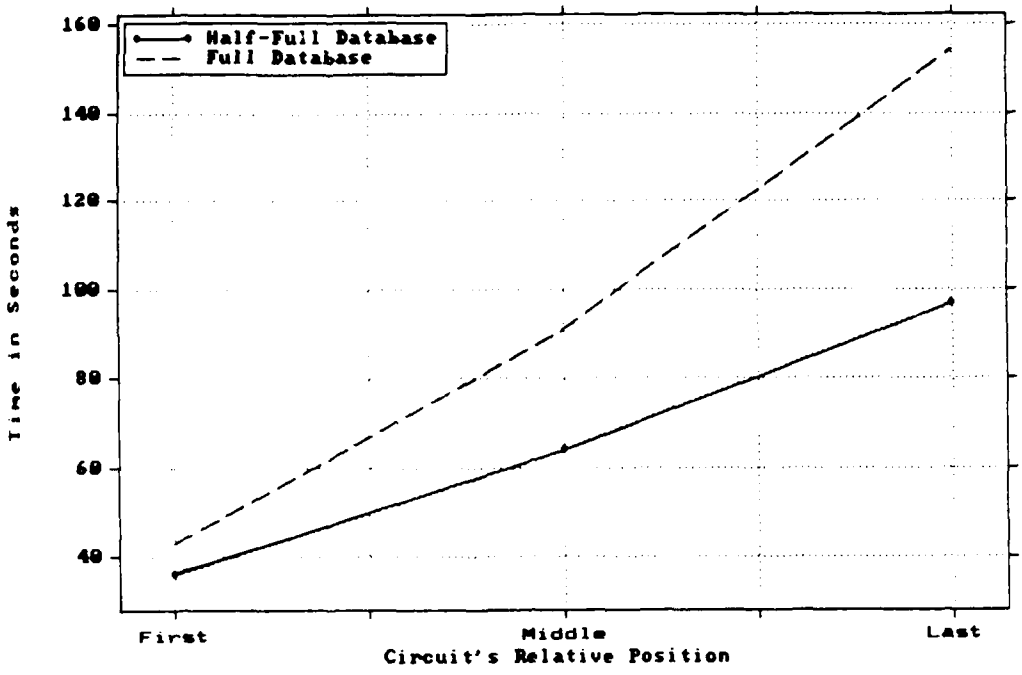

Fig. 6 - 4. Response Times of Circuit Retrieval

each circuit relation, and six conversions to TEMP file format. For all three of three operations, a more crowded database resulted in a longer response time. For deletion and retrieval, the average time should be slower when the database is larger because of the sequential searches being done. The slower response times for insertions into a crowded database may be due to the more complex index structure that must be manipulated. Also note that variations in the time versus size ratio for insertions may be due to individual characteristics of the different designs stored, for example it would take less time to store ten records from one file than it would tc store one record from each of ten files. Finally, keep in mind the circuit database is maintained on the floppy drive which is inherently slower than internal memory. The slow accesses, though, occur only at the beginning or end of the

design session, when a circuit design is saved, deleted, or retrieved. The circuit database is never accessed during the design process itself.

The TTL database is what would be accessed during the design process and it has the advantage of being stored on the faster internal hard disk. A directory of the 34 TTL values, for example, takes only two seconds to produce.

Another advantage the TTL retrievals have over circuit retrievals is that TTL retrievals are generally for a single TTLS or PINS record. Because the retrievals are for a specific record, that record can be accessed directly by its key value and not have to resort to the sequential searches used in most circuit retrievals. The response time for a specific record retrieval would not vary by the relative position of the identifier in the database. All of the retrievals done for this test took approximately the same amount of time to complete, that time being under one second.

The final response time measured, the time it takes to prepare a floppy with empty database files, was found to be 17 seconds.

Summary

The database and DBMS developed for this effort were able to support all of the functions required of them by the digital circuit design environment. Only the TEMP files containing a "No Output" indication were not saved and returned when a circuit design was retrieved. The response times for circuit design manipulations, to and from the floppy drive, can be fairly long but they occur only at the

beginning and end of a design session and not during the design process itself. The speed of hard disk storage was traded for the data security of the floppy drive. The response times for TTL retrievals is quite good. As for memory utilization, while the new circuit database starts off at a slight disadvantage because of index overhead, it is able to use all the memory of a floppy disk and generally store more circuit designs than the original system because the data is stored in fewer, larger files. The original, up to ten files per design, method tends to use up the available directory slots before it uses up the memory.

VII. Conclusion and Recommendations

Conclusion

The aim of this thesis effort was twofold. The primary goal was to design and develop a centralized database and a custom tailored CAD Database Management System (DBMS) to work with AFIT's existing digital circuit design environment on a Zenith Z-248 workstation. The secondary goal was to develop a utility program that would allow new TTLs to be easily inserted into this new database. The goals were intended to solve certain problems caused by the data storage practices of the original system. One of those problems was the flat file method for storing circuit design information. It was adequate but resulted in duplication of data and an inability to obtain a listing of available designs. TTL information was another problem. The original system maintained portions of the TTL database in each of the three programs that make up the design environment, again resulting in duplication of data and making it very difficult to add new TTLs.

The goals stated above were met. A database, DBMS, and utility program were developed for this thesis effort and evaluated using the test sets described in Chapter VI. They were found to support all of the functions of the original design environment, and correct those problems just discussed, with one exception. As it stands, new TTLs added to the database are only for use by the ICE and graphics interface programs. Until the utility program is expanded to input the executable function of a TTL, the new TTLs cannot be simulated by the LOGSIM program. The DBMS conserves the limited space of the Z-248, and encourages data security by storing all circuit design data to the

designer's own floppy diskette. It significantly increases the number of small circuit designs that can be stored in a single directory of a floppy diskette. Of course, the DBMS that provides these advantages does take up some memory itself, and the increased data manipulation does take up some time, but the advantage of a more flexible and efficient digital circuit design environment is worth the cost.

The approach used in reaching the goals of this effort included researching existing methods of CAD data management, researching the data and data manipulation requirements of the existing environment, and then designing and implementing a DBMS to meet those specific requirements. The basic relational database routines used in the implementation were obtained from C Database Development by Al Stevens (12).

Recommendations

The database, DBMS, and utility programs designed and implemented for this thesis accomplished the goals set for them, but the possibility for improvement exists in nearly all things and this is no exception. The following areas are recommended for future studies:

First, design and implement a more efficient means of accessing the data. The access method provided in C Database Development is a balanced tree (B-tree) index to each unique key value, or each individual record. This works fine in those cases where directories, or all records in a relation, are requested or when a single specific record is requested, but in the vast majority of cases in this application, the search is not for one or all, but for a group of

records associated with a specific TTL or circuit. The new index should be a B-tree of nodes for each TTL or circuit identifier. Each node would then point to list of records for that identifier, as well as to the next and previous nodes. There would be no need to maintain the records in each list in any particular order.

Second, modify the graphics interface program to display and work with the six character TTL identifier allowed by the database. This would allow the five digit TTL number to be followed by either a blank or a specific package designator (i.e., J or N).

Third, modify the graphics interface program to calculate the "bounding box" or the maximum X and Y coordinates occupied by a design at the time it is saved. This information would be stored in fields already defined in the CKTS relation and could be used to determine if a whole circuit design would fit if pulled into another design at any given location. To use a circuit design as a component in another circuit would require additional coding changes in the graphics interface program to perform the following functions: change the ICON identifiers in the component circuit into identifiers that do not conflict with the identifiers already in use in the other circuit, offset all coordinates in the component circuit by the new position of its bottom left corner, allow for the possibility of having more than one of each of the power, clock, or ground icons in a single design, and finally, define a new icon of type OUTPUT PORT and manipulate it according to Capt Adams' recommendation. (1:7-4,5) Note that defining a new type of icon would require some minor changes be made to the DBMS code also.

Fourth, modify the LOGSIM program to make use of the central database. Primarily, this would involve separating the executable functions of each TTL from the main body of LOGSIM code so that all of LOGSIM would not have to be recompiled each time a new TTL is added. The method Batory suggested for maintaining executable functions in a database was to compile the TTL functions into a single module to be indexed into by the TTL identifier (3). It is my feeling that maintaining each TTL's function in its own module, <TTL-identifier>.EXE, would be even more convenient.

Fifth, expand the database utility program to allow easy input of the executable function of new TTLs. As it stands now, while they can be used in designs and analyzed by ICE, no TTLs added by means of the database utility program can be simulated by LOGSIM. Ideally, a designer should be able to extract information directly from a TTL description/characteristics book, enter it and compile it via the utility program, and have it produce as its result the individual executable module described in the immediately preceding recommendation.

Sixth, modify the graphics interface program to prompt the designer for which pieces of information he would like to have saved (i.e., design only, design and LOGSIM inputs, design and ICE results, or ALL) instead of automatically saving all. Currently, once a design has been analyzed, simulated, or had inputs or monitoring points designated, the resultant TEMP files will remain with the design forever. This modification would save some space and time in storing and retrieving unnecessary information.

Seventh, modify the graphics interface program to allow designers to view the pin assignment information for any TTL currently in the database. This would eliminate the need for keeping a TTL IC description/characteristic book handy.

Eighth, modify the Cdata "db_open" and "index" functions to provide automatic reindexing of all files found to be corrupted. The circuit designer should not have to be bothered with database problems.

Finally, modify the basic database routines to allow the use of tuple variables. Because these routines currently do not recognize tuple variables, they will not allow multiple copies of the same relation to be specified in the "from" list or two or more data elements with the same name to be specified in the "select" list. The lack of this capability forced the undesirable situation of creating code for a specific query in one case.

Appendix A:

User's Manual for the Database Utility Programs

Written by: Capt Sue A. Ehrhart

Date: November 1988

Table of Contents

Introduction

General. This manual describes the basic procedures required to setup the AFIT digital circuit design environment database and operate its associated utility programs, Makedb, Addttl, and Index.

System Requirements. The database and utility programs require a Zenith Z-248 workstation (or compatable) with the MS-DOS operating system. Because the TTL database is maintained on the internal C drive, a hard drive is required. To compile the system using Startup.bat, a Microsoft C compiler is required.

Software Installation. All database source files, "*.c", must be loaded into the same directory on the hard drive along with the batch files, Startup.bat and Makedb.bat, and data files, ttl.dat, ttl.des, and pinfile. The Startup batch file expects the Microsoft compiler to be located in the c:\ms-c directory with compiler commands in \ms-c\bin, compiler include files in \ms-c\include, and an empty work directory called \ms-c\test. With the files in place, the database can be initialized by executing the Startup batch file. When the batch file completes processing, the database will be loaded with an initial set of 34 TTLs and be ready to go.

> **NOTE**
>
> Throughout this manual, the symbols <CR>,<ESC>,<HOME>,<END>,<Page Up>, and <Page Down> indicate to keys to be pressed, not text to be typed in. <CR> is normally marked on the keyboard as Return or Enter, <ESC> is normally marked Esc, <HOME> is 7 on the number pad, <END> is 1, <Page Up> is 9, and <Page Down> is 3. The "arrow keys" are 2,4,6,and 8 on the number keypad.

Makedb Utility

The Makedb utility is a batch file that copies empty circuit database files from the C drive to the A drive to prepare floppy diskettes for circuit design storage.

> **CAUTION**
>
> If any of the circuit database files exist on the floppy before running this utility they will be overwritten, and any data they contained will be lost. The circuit database files are listed in Figure A - 1.

Use. In the directory containing the Makedb.bat file, place a formatted floppy diskette in the A drive, and enter:

Makedb <CR>

The batch file will print the following message on the screen as it copies files to the A drive:

```
rem Preparing floppy in drive A with empty database files
ECHO OFF
```

The system prompt will return when the operation is complete. A directory of the floppy in drive A should contain all of the files listed in Figure A - 1. Any other files that were on the floppy to start with will also be present.

```
CKTS      DAT        10   12-02-88   12:53p
CKTS      XO1       512   12-02-88   12:53p
ICONS     DAT        10   12-02-88   12:53p
ICONS     XO1       512   12-02-88   12:53p
LINKS     DAT        10   12-02-88   12:53p
LINKS     XO1       512   12-02-88   12:53p
LIN       DAT        10   12-02-88   12:53p
LIN       XO1       512   12-02-88   12:53p
LOUT      DAT        10   12-02-88   12:53p
LOUT      XO1       512   12-02-88   12:53p
TOUT      DAT        10   12-02-88   12:53p
TOUT      XO1       512   12-02-88   12:53p
```

Fig. A - 1. Directory of Circuit Database Files

Error Conditions

1. Door Open.

Indication: Receipt of operating system error message

```
Error reading drive A
abort,retry or ignore?
```

Correction: Close the door of the drive and enter "r" for retry.

2. Diskette Not Formatted.

Indication: Receipt of operating system error message

```
General Failure error reading drive A
abort,retry, or ignore?
```

Correction: Format the diskette in accordance with MS-DOS operating instructions and reexecute the Makedb utility.

3. Insufficient Space.

Indication: Receipt of operating system error message

```
Insufficient Space
0 file(s) copied.
```

Correction: There was not enough room to store even empty files on the diskette in drive A. Delete some non-database files to make room, or better yet, select another diskette for use. Then reexecute the Makedb utility.

Addttl Utility

The Addttl utility is an executable program intended for placing additional TTLs into the design database. Because of the possibility of errors, however, the capability to modify, delete, and view the data have also been included. To enter data for one or more TTLs, the Addttl utility must be executed twice, the first time to fill in the data required by the TTLS relation, and then again to fill in the data required by the PINS relation. This utility is currently limited in that it cannot be used to input the executable function of any new TTL.

> CAUTION
>
> The operator of this utility is responsible for maintaining the accuracy and integrity of the TTL database. A data manager should make all modifications to the database and then copy TTL.DAT, TTL.X01, PINS.DAT, and PINS.X01 from his machine to the design workstations. This utility should not be left on student workstations.

> NOTE
>
> When entering data on the screen, the cursor will move to the next field automatically when the current field is full or when a <CR> is entered. The <CR> can be entered at any time. There is no need to blank fill any unused space in any data fields.

TTLS Data. While in the directory containing the "Addttl.exe" file, enter:

Addttl ttls <CR>

The screen will be displayed as shown in Figure A - 2. The cursor will automatically be positioned at the beginning of the TTL field.

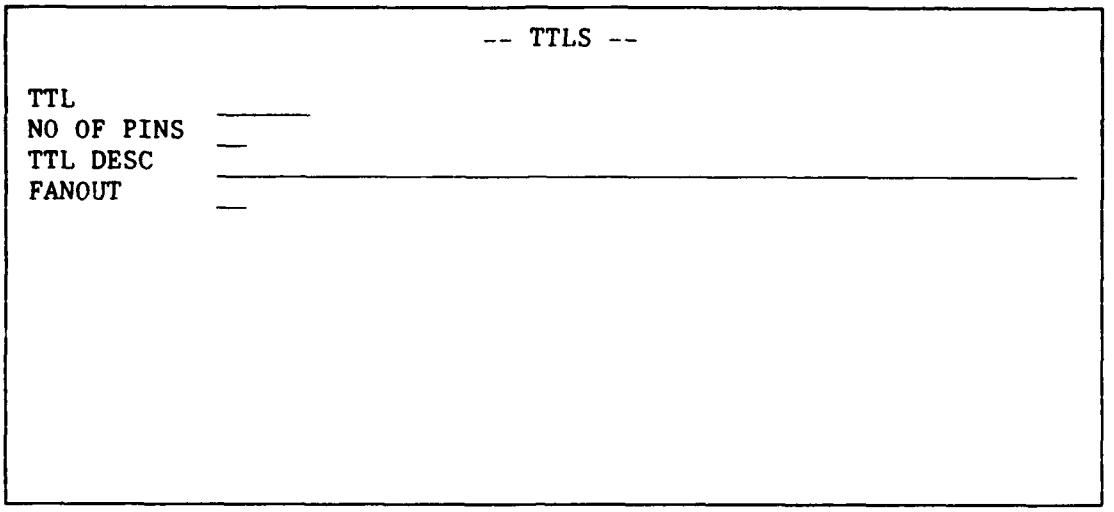

Fig. A - 2. TTLS Relation Template

Add. Use the steps listed in this section to add a new TTL to the database.

Step 1: Enter the identifier of the TTL being added. The format of the identifier should be left justified, digits only, with no leading zeros. While the database allows up to 6 alphanumeric characters, the design environment currently expects only 4 or 5 digit identifiers. Use the backspace key and make any corrections necessary before pressing <CR> to advance the cursor to the NO OF PINS field. Once <CR> has been pressed, the notice "New record" should appear in the bottom left corner of the screen. If that doesn't occur and, instead, information automatically appears in the remaining fields, the TTL you are attempting to enter is already in the database. Press <ESC> to clear the template and start over. Press <ESC> a second time to end the program.

Step 2: Enter the two digit number corresponding to the number of pins for the TTL just entered. The cursor will automatically advance to the TTL DESC field.

Step 3: Enter the description of the TTL function. As a reminder to subsequent users that there is no executable function for this TTL, be sure to include the phrase "NO SIMULATION". Enter <CR>, if necessary, to advance to the FANOUT field.

Step 4: Enter the two digit fanout associated with the TTL being entered. This value is generally 10.

Step 5: If there is an error in the TTL identifier field that was not caught during Step 1, press <ESC> to clear the template and start over. If there are errors in any fields other than TTL identifier, use the arrow keys to position the cursor and make corrections.

Step 6: When the data is correct, enter <F1> to store the record in the database. The message "New record added" will appear at the bottom left corner of the screen. Press <ESC> once to clear the template and begin another operation on the TTLS data. Press twice to end the program. Remember to now add the pin data for all the new TTLs.

Modify. If a TTLS record has already been stored in the database before errors have been detected, it is still possible to correct them. If the error is in the TTL identifier field, the record

will have to be deleted and then re-added. See the instructions for those sections. If the errors are in any other fields, use the instructions in this section.

Step 1: Enter the TTL identifier just as it was originally entered. If you are not sure of the identifier, see the instructions for "View" in this section. Once <CR> has been pressed, the data previously entered should appear in the remaining fields. If they do not and, instead, the message "New record" appears in the bottom left corner of the screen, you have not entered the TTL identifier correctly. Press <ESC> to clear the template and then try again.

Step 2: Use the arrow keys to position the cursor to make the necessary corrections.

Step 3: When the data is correct, enter <F1> to store the changed record back in the database. The message "Returning record" will appear at the bottom left corner of the screen. Press <ESC> once to clear the template and begin another operation on the TTLS data. Press <ESC> a second time to end the program.

Delete. If there was an error in the TTL identifier field, or some other reason to want the TTL out of the database, follow the steps in this section. To preserve database integrity, all pin data of the ttls concerned should be removed first.

Step 1: Enter the TTL identifier just as it was originally entered. If you are not sure of the identifier, see the instructions for "View" in this section. Once <CR> has been pressed, the data previously entered should appear in the remaining fields. If they do not and, instead, the message "New record" appears in the bottom left corner of the screen, you have not entered the TTL identifier correctly. Press <ESC> to clear the template and then try again.

Step 2: Once the correct record is displayed on the screen, press <F7> to delete. The message "Verify w/F7" will appear at the bottom left corner of the screen.

Step 3: If you wish to carry out the deletion, press <F7> to confirm and the data will disappear from the screen. Any key other than <F7> will abort the delete request.

Step 4: If the template is not already clear, press <ESC> once to clear it and begin another operation on the TTLS data. Press <ESC> with the template empty to end the program.

View. If you are unsure of the TTL identifier you are looking for, the records of each TTL in the database can be viewed by using the following steps.

Step 1: Press <Home> to bring up the first record in the database or <end> to bring up the last record in the database. Records are stored

in "dictionary" order from smallest to largest, with 44151 coming before 7420.

Step 2: Press <Page Down> or <Page Up> to view any records in between. The message "Beginning of file" or "At end of file" will appear at the bottom left corner of the screen when the extreme values contained in the database are displayed.

Step 3: Press <ESC> once to clear the template and begin another operation on the TTLS data. Press <ESC> a second time to end the program.

PINS Data. While in the directory containing the "Addttl.exe" file, enter:

Addttl pins <CR>

The screen will be displayed as shown in Figure A - 3. The cursor will automatically be positioned at the beginning of the TTL field.

Add. Use the steps listed in this section to add the pins of a new TTL to the database. Please note that the TTLS record MUST be entered before the PINS records will be accepted into the database.

Step 1: Enter the identifier of the TTL exactly as it was stored in the TTLS relation. Use the backspace key and make any corrections necessary before pressing <CR> to advance the cursor to the PIN field.

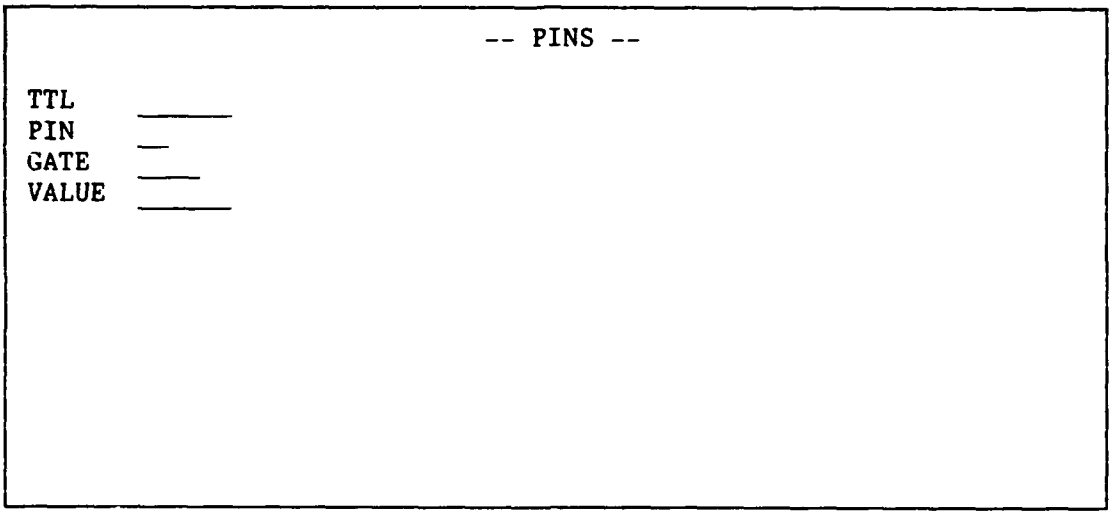

Fig. A - 3. PINS Relation Template

Step 2: Using a leading zero if necessary, enter the two digit number of the pin being added. The cursor will automatically advance to the GATE field and the notice "New record" should appear in the bottom left corner of the screen. If that doesn't occur and, instead, information automatically appears in the remaining fields, the pin you are attempting to enter is already in the database. Press <ESC> to clear the template and start over. Press <ESC> a second time to end the program.

Step 3: Enter the GATE the pin is part of. This field must be left justified and all capital letters. If the function of the pin is as CLOCK, POWER, or GROUND input, the GATE field should read "CHIP." If the pin is not connected, the GATE field should read "NC." Otherwise, the design environment is expecting a single letter. Enter <CR>, if necessary, to advance to the VALUE field.

Step 4: Enter the 'ALUE, or function, associated with the pin being entered. Legal values are POWER, CLOCK, GROUND, INPUT, OUTPUT, and NC for "not connected." This field must be left justified and all capital letters.

Step 5: If there is an error in either the TTL identifier field or the PIN field that was not caught during Step 1, press <ESC> to clear the template and start over. If there are errors in the GATE or VALUE fields, use the arrow keys to position the cursor and make corrections.

Step 6: When the data is correct, enter <F1> to store the record in the database. The message "New record added" should appear at the bottom left corner of the screen. If, instead, the message says "Record not found", you tried to enter a pin record for a TTL that is not in the database. Refer, in that case, to the section on "Error Conditions." Otherwise, press <ESC> once to clear the template and begin another operation on the PINS data. Press twice to end the program.

Modify. If a PINS record has already been stored in the database before errors have been detected, it is still possible to correct them. If the error is in either the TTL identifier field or the PIN field, the record will have to be deleted and then re-added. See the instructions for those sections. If the errors are in either the GATE or VALUE fields, use the instructions in this section.

Step 1: Enter the TTL identifier just as it was originally entered in the TTLS relation. If you are not sure of the identifier, see the instructions for "View" in the TTLS Data section. Press <CR> to advance the cursor to the PIN field.

Step 2: Enter the two digit PIN number just as it was entered originally. The cursor will automatically advance to the next field. Once the cursor advances to the next field, the data previously entered should appear in the remaining fields. If it does not and, instead, the message "New record" appears in the bottom left corner of the screen, you have not entered the PIN number correctly. If you are not sure of the PIN number, see the instructions for "View" in this section. Press <ESC> to clear the template and then try again. When the data does appear, use the arrow keys to position the cursor to make the necessary corrections.

Step 3: When the data is correct, enter <F1> to store the changed record back in the database. The message "Returning record" will appear at the bottom left corner of the screen. Press <ESC> once to clear the template and begin another operation on the PINS data. Press <ESC> a second time to end the program.

Delete. If there was an error in either the TTL identifier field or the PIN field, or some other reason to want the PIN record out of the database, follow the steps in this section.

Step 1: Enter the TTL identifier just as it was originally entered in the TTLS relation. If you are not sure of the identifier, see the instructions for "View" in the TTLS Data section.

Step 2: Enter the PIN number just as it was entered originally. Press <CR>, if necessary, to advance the cursor to the next field. Once the cursor advances to the next field, the data previously entered should appear in the remaining fields. If it does not and, instead, the message "New record" appears in the bottom left corner of the screen, you have not entered the PIN number correctly. If you are not sure of the PIN number, see the instructions for "View" in this section. Press <ESC> to clear the template and then try again. When the data does appear, use the arrow keys to position the cursor to make the necessary corrections.

Step 3: Once the correct record is displayed on the screen, press <F7> to delete. The message "Verify w/F7" will appear at the bottom left corner of the screen.

Step 4: If you wish to carry out the deletion, press <F7> to confirm and the data will disappear from the screen. Any key other than <F7> will abort the delete request.

Step 5: If the template is not already clear, press <ESC> once to clear it and begin another operation on the PINS data. Press <ESC> with the template empty to end the program.

View. If you know the TTL identifier, but are unsure of the PIN number you are looking for, there are two possibilities:

1. If you can remember the exact way ANY PIN number was entered for the particular TTL concerned, enter it. Once the record for that pin is displayed, use <Page Up> and <Page Down>, as described in the TTLS Data "View", to find all of the records for that TTL. The records will be grouped together by TTL number.

2. If you cannot remember ANY of the PINs entered for the TTL, it will be necessary to search the entire PINS relation from the top, or bottom, as described in the TTLS Data "View."

Error Conditions.

1. No Relation Name Specified on the Command Line.

Indication: The program returns only the system prompt, no data entry template is displayed.

Correction: the program must know which relation you wish to modify. Reexecute, specifying TTLS or PINS

2. Illegal Relation Name Specified on the Command Line.

Indication: The error message "Only the PINS and TTLS relations can be modified using this program" is displayed and the system prompt returns.

Correction: To protect the integrity of the database to the greatest

extent possible, only TTLS and PINS may be modified using the Addttl Utility. The program is not case sensitive, but double check that the relation names are spelled correctly and reexecute.

3. Attempted Addition of a Pin Record Without an Associated TTL Record.

Indication: When a pin record is entered, <F1> is pressed, the error message "Record not found" appears in the bottom left corner of the screen.

Correction: For database integrity, the TTL information must be present before the pin data can be entered. Press <ESC> once to clear the template and a second time to exit the Addttl program. Reexecute the program, entering the TTL data first.

Index Utility

The Index utility is an executable program that rebuilds the database index files to reflect the contents of its associated data file. This utility should be used in the event a power failure occurs during data manipulation. In the circuit design process, data manipulation takes place at the following times: retrieving a circuit design, deleting a circuit design, saving a circuit design, and executing the InterConnect Expert (ICE) program. Outside the design environment, data manipulation takes place during the Addttl process. The Addttl utility and the ICE program manipulate only TTL data and the other activities, as their names imply, manipulate only circuit data. If you are performing any of these activities when a power failure occurs, you will need to run the Index utility on whichever half of the database was involved. It is best to run this utility as soon as possible after power is restored. If you delay, and later inadvertently try to use the affected data during the circuit design process, the design environment should remind you which half of the database needs to be reindexed. This reminder may come at an inopportune moment, for example when you have just finished work on a brand new circuit design and find you cannot save it because the circuit database needs to be reindexed. This would not be a problem if, as is advisable, you had a second floppy prepared and available to save it to, but is still best to take care of the situation before reentering the circuit design environment.

Indexttl. If the circuit design environment informs you that the TTL database needs reindexing, or you wish to recover from a power failure that occurred during ICE or Addttl processing, follow the instructions given in this section.

In the directory containing the "Indexttl.bat" file, enter:

Indexttl <CR>

This batch file will invoke the Index utility once for each of the relations containing TTL data. Reindexing can take some time is the data file is large. Please be patient. The Index utility will print the following messages on the screen:

```
ECHO OFF
Indexing TTLS
Indexing PINS
```

The system prompt will return when the operation is complete.

Indexckt. If the circuit design environment informs you that the circuit database needs reindexing, or you wish to recover from a power failure that occurred while saving, deleting, or retrieving a circuit design, follow the instructions given in this section.

In the directory containing the "Indexckt.bat" file, place the floppy containing the affected data in the A drive. Check to make sure the circuit database files are on the floppy you are using (Check for the

files listed in Figure A - 1) and, if they are, enter:

Indexckt <CR>

This batch file will invoke the Index utility once for each of the relations containing circuit data. Reindexing can take some time is the data file is large. Please be patient. The Index utility will print the following messages on the screen:

```
ECHO OFF
Indexing CKTS
Indexing ICONS
Indexing LINKS
Indexing IOUT
Indexing LIN
Indexing LOUT
```

The system prompt will return when the operation is complete. The response will be the same even if the circuit files were not on the floppy.

Error Conditions. There are no error conditions likely to occur when reindexing the TTL database. The following error may occur when reindexing the circuit database:

DOOR OPEN

Indication: Receipt of operating system error message

```
Error reading drive A
abort,retry or ignore?
```

Correction: Close the door of the drive and enter "r" for retry.

Appendix B:

Cdata Functions Used by the DML Module

add_rcd: "Use this function to add a record to a file" (12:125). Returns an integer status flag.

db_cls: This function closes all data base files, flushes the buffers, and closes the index files (12:130).

db_open : "Its purpose is to initialize the data base files and their index files for access by your programs" (12:124). Modified to ignore the path parameter and look only on the A drive for circuit data and only on the C drive for TTL data. Modified to return an integer status flag.

del_rcd: "Use this function to delete a record that was previously retrieved" (12:129). Returns an integer status flag.

epos: "If the data element integer passed as the first parameter is not is the list (or is zero), then the function returns the length of a buffer required to hold the data elements in the list" (12:132).

find_rcd: This function retrieves a specific record from a file and stores a copy in a local buffer. Returns an integer status flag (12:126).

init_rcd: "Use when you want to initialize a data base file's record buffer to null values. It sets each data element field in the buffer to a null-terminated string of spaces" (12:133)"

next_rcd: If no prior access has been made to the file, this function retrieves record corresponding to the lowest index value and places a copy in a local buffer. "Successive calls to next_rcd will deliver the records in the ascending sequence of the index" (12:128). Returns an integer status flag.

rcd_fill: Moves selected elements from one data structure to another. "Data elements that are in the source record and not in the destination are not moved, and data items that are in the destination record but not in the source are not disturbed" (12:131).

verify_rcd: "Similar to find_rcd except that it does not retrieve a record" (12:126). Returns an integer status flag.

Bibliography

1. Adams, Capt Charles A., Jr. A Digital Circuit Design Environment. MS thesis, AFIT/GCS/ENG/87D-1. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1987 (AD-A188831).

2. Batory, D.S. and Won Kim. "Modeling Concepts for VLSI CAD Objects," ACM Transactions on Database Systems, 10: 322-346 (September 1985).

3. Batory, D.S., Assistant Professor, Personal Correspondence, Department of Computer Sciences, University of Texas at Austin, 21 June 1988.

4. Chang, Shi-Kuo and T. L. Kunii. "Pictorial Data-Base Systems," Computer, 14: 13-21 (November 1981).

5. Deloria, Capt Wayne C. A Digital Logic Simulator with Concurrent Programming Considerations. MS thesis, AFIT/GCS/ENG/87D-10. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1987 (AD-A188823).

6. Duke, Kieth A. and K. Maling. "ALEX: A Conversational, Hierarchical Logic Design System," 17th Design Automation Conference Proceedings. 318-327. Minneapolis: ACM/Sigda and IEEE Computer Society DATC, 1980.

7. Korth, Henry F. and A. Silberschatz. Database Systems Concepts. New York: McGraw-Hill Book Company, 1986.

8. Matuszek, Michael L., MS Student, Class GCE-88D. Personal interviews. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, 1 March through 1 December 1988.

9. McCaskey, John. "Object-Oriented Database Keeps the House in Order," Electronic Design, 35: 129-134 (March 1987).

10. McKeown, D. M., Jr. and D. J. Reddy. "A Hierarchical Symbolic Representation for Image Databases," Proceedings of the IEEE Workshop on Picture Data Description and Management : 40-44. April 1977.

11. Santos, Jorge da Silva, MS Student, Class GCS-88D. Personal interviews. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, 1 March through 1 December 1988.

12. Stevens, Al. C Database Development. Portland, Oregon: Management Information Source, Inc., 1987.

13. Wagner, 1Lt Steven M. An Expert System for Discrete Component Digital Circuit Design. MS thesis, AFIT/GCS/ENG/87D-28. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1987 (AD-A189680).

VITA

Captain Sue A. Ehrhart [redacted] She graduated from the University of Arizona at Tucson in 1978, with a Bachelor of Science Degree in Chemistry. She received her commission through the Officer Training School (OTS) in April 1980. After graduation from OTS, she was stationed at Keesler Air Force Base (AFB), Mississippi, where she attended the Communications-Electronics Officer School. Upon graduation in April 1981, she was assigned as Automated Communications Computer Programmer, Strategic Air Command Total Information Network Applications Section, Data Systems Division, 1st Aerospace Communications Group (AFCC), Offutt AFB, Nebraska. In June 1984, she was assigned to Headquarters Air Force Operational Test and Evaluation Center, Kirtland AFB, New Mexico, where she became Chief, Information Systems Customer Support Branch. In May 1987, Captain Ehrhart entered the School of Engineering, Air Force Institute of Technology.

[redacted]

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) AFIT/GCS/ENG/88D-4	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION School of Engineering 6b. OFFICE SYMBOL (If applicable) AFIT/ENA	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State, and ZIP Code) Air Force Institute of Technology Wright-Patterson AFB, Ohio 45433	7b. ADDRESS (City, State, and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION None 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO / WORK UNIT ACCESSION NO.
11. TITLE (Include Security Classification) A Database Management System for Computer-Aided Digital Circuit Design UNCLASSIFIED	
12. PERSONAL AUTHOR(S) Sue A. Ehrhart, Captain, USAF	
13a. TYPE OF REPORT MS Thesis 13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (Year, Month, Day) 1988 December 15. PAGE COUNT 101
16. SUPPLEMENTARY NOTATION	

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP
12	05	

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Data Bases,
Computer Aided Design, Circuits

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Thesis Advisor: Bruce L. George, Captain, USAF
Assistant Professor of Electrical Engineering
and Computer Science

Abstract:
This thesis effort documents the design and implementation of a relational database and associated database management system (DBMS) for the AFIT digital circuit design environment, a graphics oriented tool that allows circuits to be designed at a uniform, chip-level of detail, checked for proper connections, and simulated.

cont on reverse

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Bruce L. George, Captain, USAF	22b. TELEPHONE (Include Area Code) 513-255-3576 22c. OFFICE SYMBOL AFIT/ENG

DD Form 1473, JUN 86 *Previous editions are obsolete.*

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

UNCLASSIFIED

Box 19 continued:

The approach to this effort included a survey of existing methods of Computer-Aided Design (CAD) data management, analysis of the data and data manipulation requirements of the design environment, design of a data manipulation language, and implementation of a DBMS to carry out the manipulations. Implementation was done in the C programming language and based on lower-level database routines found in C Database Development by Al Stevens. Limitations encountered as a result of using these routines are discussed along with the results of testing. This effort also includes three separate database utility programs. One allows new TTLs to be added to the database but does not provide the ability to input the executable functions of those new TTLs. The second provides the capability to rebuild corrupted index files, and the third prepares floppy diskettes for data storage. A user's manual is included for the operation of the database utility programs. Recommendations for future work are also presented.