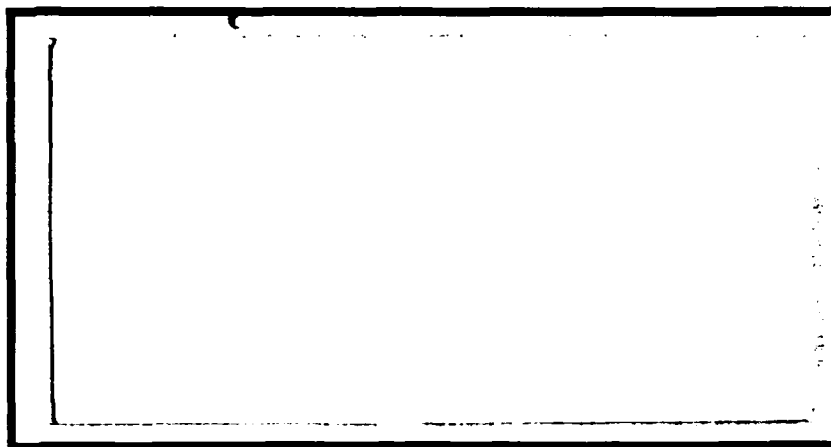


DTIC FILE COPY

AD-A202 619



DTIC
ELECTE
18 JAN 1989
S D
E

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

This document has been approved
for public release and sales its
distribution is unlimited.

89

1

17

118

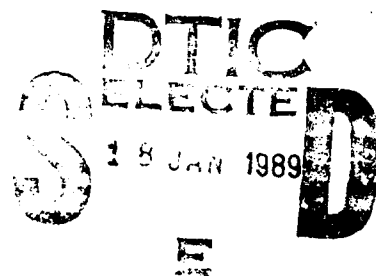
AFIT/GE/ENG/88D-25

THE DESIGN AND IMPLEMENTATION
OF A GRAPHICAL VHDL USER INTERFACE

THESIS

Stephen M. Matechik
Captain, USAF

AFIT/GE/ENG/88D-25



Approved for public release; distribution unlimited

AFIT/GE/ENG/88D-25

THE DESIGN AND IMPLEMENTATION
OF A GRAPHICAL VHDL USER INTERFACE

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 Electrical Engineering

Stephen M. Matechik, B.S.
Captain, USAF

December 1988

Approved for public release; distribution unlimited

Acknowledgements

To Stephen, Matthew, and Thomas - my children. All the times that I could not be with you may be found here in this work. Befittingly so, I dedicate this work to you because it was generated out of time that belonged to you.

To Susie - my wife. Your thoughtfulness and personal sacrifices never went unnoticed or unappreciated. Your efforts are as much a part of this thesis as are my own. That is why this work belongs to you as much as it does to me. I thank God for the strength and perseverance needed to complete this work and for a most supportive and loving wife.

To Captain Bruce George - my faculty advisor. Your confidence in my abilities often exceeded my own. Your high expectations and continual "push" played an instrumental role in the successful completion and delivery of this product.

To Major James Howatt and Dr. Matthew Kabrisky - members of my thesis committee. Thank you for your time, assistance, and willingness to be a part of this endeavor.

Accession For	
NTIS GRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	



Table of Contents

	Page
Acknowledgements	ii
List of Figures	vii
List of Tables	viii
Abstract	ix
1. Introduction	1
1.1. VHDL Background	1
1.1.1. Development	1
1.1.2. Characteristics	1
1.1.3. Support Environment	2
1.2. GVUI Thesis Effort	3
1.2.1. Goals	3
1.2.2. Scope and Assumptions	4
1.2.3. Approach	4
2. SYSTEMS REVIEW	6
2.1. VHDL Environments	6
2.1.1. IBM VHDL Design System	6
2.1.2. Viewlogic VHDL Translator	6
2.1.3. University of Pittsburgh Sced	7
2.1.4. General Electric IVW	7
2.1.5. Vista Technologies VDW	7
2.1.6. AFIT VHDL Environment	7
2.2. Operational Systems	8
2.2.1. VHDL Graphic Environments	8
2.2.1.1. Observations	9
2.2.1.2. Conclusion	10

2.2.2. PC-Based Schematic Editors	10
2.3. Software Development	12
2.3.1. Top-Down	13
2.3.2. Radical Top-Down	14
2.3.3. Conservative Top-Down	15
2.3.4. Zigzag	16
2.3.5. Bottom-Up	16
3. System Design	18
3.1. Overview	18
3.2. Hardware Requirements and Constraints	18
3.2.1. Host Platform	18
3.2.2. Video Adapters	18
3.2.3. Memory and Storage	20
3.3. Software	20
3.3.1. Approach	20
3.3.2. Reliability and Maintainability	21
3.3.3. Human Factors	21
3.4. Capabilities	22
3.4.1. Environment Manager (1.0)	23
3.4.2. Librarian (2.0)	23
3.4.3. Entity Interface Editor (3.0)	23
3.4.4. Architectural Body Editor (4.0)	25
3.4.5. Generate Graphics (5.0)	25
3.4.6. Generate Code (6.0)	26
4. DETAILED DESIGN	27
4.1. Screen Layout	27
4.1.1. Menu Bar	28
4.1.2. Feedback Area	28

4.1.3. Viewport Locator Area	29
4.1.4. Graphic Viewport	29
4.2. Environment Manager	30
4.2.1. Input Interpreter	31
4.2.1.1. Menu Environment	31
4.2.1.2. Menu Hierarchy	32
4.2.1.3. Data Structure	34
4.2.1.4. Control Flow	35
4.2.2. Coordinate Manager	36
4.2.2.1. World and Screen Coordinate Systems	36
4.2.2.2. Viewport Coordinate System	38
4.2.2.3. Cursor Movement	40
4.3. Entity Interface Editor	42
4.3.1. Symbol Design	45
4.3.2. Control	47
4.3.3. Data Structure	50
4.4. Architectural Body Editor (ABE)	53
4.4.1. Control	54
4.4.2. Data Structure	61
4.4.2.1. Component Instantiations	62
4.4.2.2. Architectural Body Ports	65
4.4.2.3. Connections	68
5. TESTING	71
5.1. Dimensions	71
5.2. Levels	71
5.3. Approach	72
5.4. Functional System Tests	72
5.4.1. Main Menu	73

5.4.1.1. Editor Selection Menu	74
5.4.2. Entity Interface Editor Menu	75
5.4.2.1. Entity Symbol Menu	77
5.4.3. Architectural Body Editor Menu	81
6. CONCLUSION	89
6.1. Future Recommendations	89
6.1.1. Video Adapter Compatibility	89
6.1.2. Mouse Input	90
6.1.3. Data File Compaction	90
6.1.4. Additional Feature	91
6.1.4.1. MOVE Option	91
6.1.4.2. Bus Routing	91
6.2. GVUI Phase I Summary	92
Appendix A: AFIT GVUI User's Guide	94
Bibliography	BIB-1
Vita	VITA-1

List of Figures

	<i>Page</i>
Figure 1. VHDL Support Environment (11:43)	3
Figure 2. IVW & VDW Autorouting.	11
Figure 3. A Typical Program Structure	14
Figure 4. GVUI Top-Level SADT Diagram	24
Figure 5. Environment Manager SADT Diagram	25
Figure 6. GVUI Functional Areas	27
Figure 7. GVUI Menu Hierarchy	33
Figure 8. Input Interpreter Flowchart	37
Figure 9. World Coordinates and Viewport Scroll Increments	39
Figure 10. Scrolling and SZS Coordinates	41
Figure 11. Dependent Scrolling	43
Figure 12. Independent Scrolling	44
Figure 13. Entity Interface Editor Flowchart	48
Figure 14. Entity Interface Editor Flowchart	49
Figure 15. Entity Interface Data Structure	53
Figure 16. Architectural Body Editor Flowchart	55
Figure 17. Architectural Body Editor Flowchart	56
Figure 18. RMV CONNECT Verification	59
Figure 19. Architectural Body Data Structure	62
Figure 20. Instantiation Data Structure	66
Figure 21. Architectural Body Ports Data Structure	67
Figure 22. Connection Data Structure	69

List of Tables

	Page
Table I. Entity Symbol Types	51
Table II. Entity Interface Symbols	57
Table III. Component Symbols	64

Abstract

This thesis effort outlines the design and implementation of a Graphical VHDL User Interface (GVUI). Though the GVUI is designed as an integral component of the UNIX-based Air Force Institute of Technology (AFIT) VHDL Environment (AVE), it is a stand alone tool that operates under MS-DOS on an IBM PC-XT personal computer.

The goal of the GVUI is to automatically generate VHDL source code from a graphic schematic diagram. To meet this goal, two phases of development were identified. Phase I primarily addresses the schematic generation and editing capabilities of the system. Phase II shall focus on the system's automatic code generation capabilities. This thesis specifically addresses and completes the first of these two phases.

This research paper is organized in the following manner. A brief introduction to VHDL and current VHDL programming environments is presented as background information. The GVUI development is then tracked from its requirements analysis and system design, through detailed design, and on through functional testing. In conclusion, recommendations for future enhancements are provided. A comprehensive user's guide is included as an appendix.

THE DESIGN AND IMPLEMENTATION OF A GRAPHICAL VHDL USER INTERFACE

1. Introduction

1.1. VHDL Background

In support of the development of Very High Speed Integrated Circuit (VHSIC) technology, Congress authorized funding of the VHSIC Program in 1979. It was headed by the Office of the Undersecretary of Defense for Research and Engineering (17:45). The goal of the DoD VHSIC Program was to insert VHSIC technology into military systems and to extend integrated circuit capability in the United States by one to two orders of magnitude in density and throughput (29:81).

1.1.1. Development. To meet its goals, the VHSIC Program required a standard medium of communication for transmitting complex VHSIC design data from one party to another. English is too imprecise to convey a common understanding among electronic designers and design automation tools, and traditional descriptive methods will become inadequate as the complexity of electronic systems grows (7:12). As a result, in 1983 the VHSIC Program Office at Wright-Patterson Air Force Base, Ohio, initiated the development of the VHSIC Hardware Description Language (VHDL). Gadiant and Dewey from the VHSIC Program Office appropriately summarized the motivation behind VHDL as follows:

Computer-aided engineering is a nightmare of incompatible formats and a Babel of different languages. No standard hardware description language exists, and therefore, each manufacturer's workstation and software forms an integrated package; translators must be devised (7:13).

VHDL addresses the stringent documentation requirements required for advanced military electronic systems, while simultaneously providing a standard hardware description language (1:27).

Although other hardware description languages exist, none match the total capabilities of VHDL when working with VLSI and VHSIC circuits (27). Integrated design environments which cover the circuit development spectrum from layout to simulation through the generation and analysis of VHDL code are only now emerging.

1.1.2. Characteristics. VHDL centers around the design entity, which is a description of a physical hardware component. An entity consists of two parts: an entity interface and an entity body. The entity interface describes the hardware component's ports, or connections, to the outside world. The entity body describes the functional design of the component. The body may be either a structural or behavioral description. A structural description is based upon the flow of data through the entity, as could be conceptualized in a schematic diagram. A behavioral description relies on the algorithmic control flow that transforms data within the component and may often be simplified by a mathematical equation.

Examples of design entities include digital logic gates, integrated circuits, and even entire microprocessor systems. VHDL's ability to describe entities such as those just listed is credited to its hierarchical descriptive capabilities; that is, any design entity may include in its body other previously defined design entities.

1.1.3. Support Environment. Once a design entity has been created, it may be simulated using VHDL's support environment. The VHDL support environment consists minimally of an analyzer, a design library, a design library manager, and a simulator (11:43), as shown in Figure 1. The VHDL analyzer accepts VHDL hardware descriptions and checks for port usage and syntax errors. It then translates the VHDL source code description to an intermediate form which is stored in the design library. The design library is a repository of VHDL design units which are shared by all users of the support environment (11:43-44). All support environment components communicate with the design library through the design library manager. The simulator predicts how the VHDL-described hardware would behave if it were actually implemented in hardware.

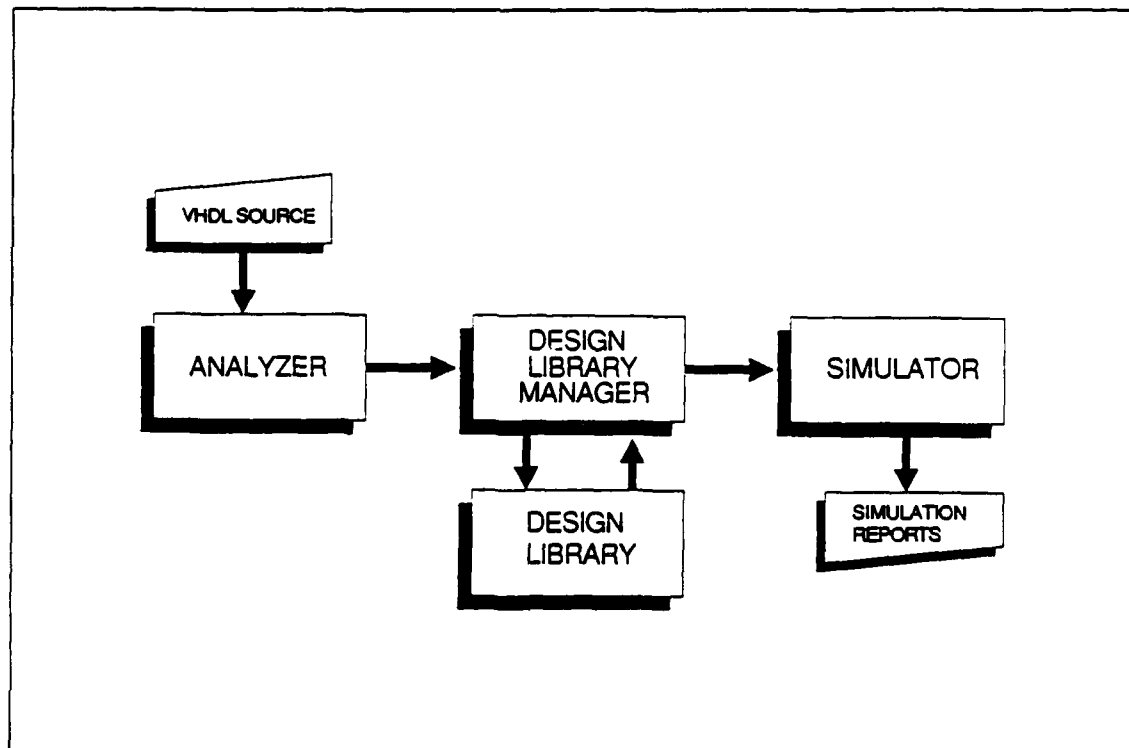


Figure 1. VHDL Support Environment (11:43)

1.2. GVUI Thesis Effort

AFIT seeks to broaden VHDL's user base through increased exposure. In support of this goal AFIT is offering a "full spectrum" VHDL design suite, the AFIT VHDL Environment (AVE), to colleges and universities free of charge. To increase the AVE user's productivity and simplify the AVE's user interface the Graphical VHDL User Interface (GVUI) is being developed under this thesis effort.

1.2.1. Goals. The GVUI is projected to increase the AVE's productivity in two ways: first, by decreasing the time required to train an engineer VHDL; second, by automating the VHDL code generation process, thereby reducing the likelihood of errors.

The "keep it simple" philosophy is a hallmark of the GVUI, not necessarily in terms of code complexity, but rather with regards to human factors and the human-computer interface. For instance, though the GVUI appears to its users as a typical schematic editor,

closer inspection reveals the GVUI is not so typical. Logic devices displayed on the screen are not labeled with device and pin numbers, but instead with entity names and port labels. This simple labeling transformation repetitively exposes the engineer who may be unfamiliar with VHDL to conventional and familiar layout symbology encompassing VHDL terminology and concepts. The GVUI maps what is new - VHDL, to what is already known - schematic layout.

1.2.2. Scope and Assumptions. The GVUI shall provide the UNIX-based AVE with a graphical front-end which can be executed on an IBM PC-XT class personal computer running MS-DOS. To simplify integration to the AVE and to other VHDL analyzers as well, the GVUI shall be capable of producing VHDL source code as standard ASCII text files just as if the user had entered the code via a keyboard.

Since educational institutions are the intended AVE recipients with students being the actual users, the hardware to host the GVUI was selected with the student in mind. From a systems design standpoint a platform with a high-resolution display is desired. From a student's standpoint, affordability is paramount.

This thesis effort addresses the design and coding of the GVUI's schematic editing capability. It also includes the code necessary to extract and link the graphic information which will enable the GVUI to self-generate VHDL source code. All coding is accomplished in the C language using Borland's Turbo C Compiler and Integrated Design Environment.

Although the GVUI user needs to be only minimally familiar with VHDL in order to properly and effectively operate the GVUI, it is assumed that the user is knowledgeable in digital circuit design and also knows how to operate an IBM PC-XT class computer.

1.2.3. Approach. Not unlike most complex projects that one endeavors, this thesis effort commenced with a thorough review of current literature. The literature review focused primarily on the following topics: C computer language, CAD tools, computer graphics, graphic data structures, human factors, MS-DOS, computer assisted schematic layout, software development techniques, and VHDL.

Some of the most valuable research that was obtained resulted from hands-on reviews of two prototype VHDL environments with integrated graphic editors, and two commercially available PC-based schematic editors. Exercising these tools provided a first-hand insight to the capabilities that exist today, and the features which generally enhance performance versus those which do not.

With a fairly clear understanding of how similar tasks were implemented, the functional requirements for a graphical VHDL user interface were analyzed and the system's baseline capabilities were established. Once these capabilities had been defined, a detailed design followed. The system shell was designed, coded, and tested before any other functions were implemented. Therefore, at any given time a workable subset of the system was available for critique.

All coding was done in the C language using Borland's Turbo C Integrated Development Environment.

2. SYSTEMS REVIEW

2.1. VHDL Environments

Review of current literature revealed the following six VHDL programming environments which are of direct interest to the development of the GVUI. They are the IBM VHDL Design System, Viewlogic VHDL Translator, University of Pittsburgh Sced. General Electric Interactive VHDL Workstation (IVW), Vista Technologies VHDL Design Workbench, and AFIT VHDL Environment (AVE).

2.1.1. IBM VHDL Design System. In late 1983, IBM embarked on an extensive review of its computer aided engineering tools and strategies for hardware design. The review resulted in a decision in mid-1984 to select VHDL as the hardware description language to be used by their new in-house computer aided engineering (CAE) system which became known as the IBM VHDL Design System. IBM's selection of VHDL as their in-house hardware description language is evidence that VHDL is reaching beyond the DoD and into the commercial sector.

In his article "The IBM VHDL Design System", Larry Saunders noted some interesting observations concerning the use and acceptance of the system. Based upon its first year of use, Saunders observed that IBM engineers resisted defining hardware using a keyboard entry "programming language" (21:489). Saunders noted uncertainty as to whether graphical entry might be more productive to describe logic devices.

The most common complaint received was "VHDL is hard to remember" (21:489). This can be at least partially attributable to VHDL's strong typing. However, in spite of these complaints Saunders emphasized that the IBM system met or exceeded all of IBM's goals and expectations (21:489).

2.1.2. Viewlogic VHDL Translator. Last year Viewlogic (Marlboro, MA) was striving to become the first PC-based commercial implementation of VHDL for electronic design automation (12:21). Their VHDL Translator would generate only behavioral VHDL

descriptions of design entities, but would be hosted on an IBM personal computer, thereby introducing VHDL to the engineer's desktop.

Last July (1988), Viewlogic sponsored a meeting for the VHDL Design Exchange Group to define the needs of current CAE tools. At that meeting, Moe Shadad, considered by many to be the architect of VHDL, stressed that care must be taken in designing subsets of VHDL so that interoperability and transportability between CAE software could be maintained. (28). One such subset of VHDL is that used by Viewlogic's own system.

2.1.3. University of Pittsburgh Sced. Built around a Sun Workstation and the Sunview Environment, Sced is an interactive design tool for creating schematic layouts of electrical circuits. From the graphical layout, Sced can also extract an intermediate VHDL format, (9:1) capable of directly being simulated on a VHDL simulator.

2.1.4. General Electric IVW. The IVW, being developed by General Electric under an Air Force contract, is hosted on a Symbolics (LISP) workstation. The IVW incorporates several editors from which the user may enter a circuit description: a VHDL Language Editor, Structure Editor, State Machine Editor, and Decision Table Editor. The VHDL Language Editor provides program source text manipulation with automatic incremental syntactic and semantic analysis (6:1). The Structure Editor, State Machine Editor, and Decision Table Editor provide graphical means of entering designs which can automatically be transformed to VHDL source code via the IVW code generator (6:1).

2.1.5. Vista Technologies VDW. The VDW, being developed under an Army contract, is hosted on a SUN (UNIX) workstation. Like the IVW, the VDW is capable of generating VHDL source code from a schematic diagram that is graphically entered by the user. The VDW consists of an Interface Editor to create and edit graphical entity interface descriptions, a Behavior Editor to specify behavioral architectural bodies, and a Structural Body Editor to specify the internal, structural description of an entity (26).

2.1.6. AFIT VHDL Environment. The AFIT VHDL Environment is an advanced prototype UNIX-based VHDL programming environment that was initiated in 1986 (5:325). Captain

Frauenfelder developed a prototype analyzer and Major Lynch developed the core of a prototype simulator. These initial prototypes were expanded in 1987 to a version 7.2 subset system. The evolution of the AVE continues with current ongoing enhancements including conformity to VHDL IEEE Standard 1076-1987. This thesis effort, the phase I development of the GVUI, shall serve as the foundation for yet another AVE capability.

Limited to an analyzer and simulator, the development of the AVE was not intended to compete with or replace commercial VHDL programming environments (5:328). Its primary purpose is to promote the use of VHDL and advanced VHDL research (5:324). To serve as a catalyst in accomplishing this goal, the AVE will be made available to colleges and universities at no charge, with releasability controlled by the VHSIC Program Office.

2.2. Operational Systems

In order to effectively promote VHDL, the AVE is to be as capable and flexible as present VHDL Environments while simultaneously extending its usability to the personal computer (PC). To accomodate these seemingly conflicting goals the AVE analyzer and simulator shall continue to run under UNIX, but the graphical front end shall be designed to operate on a PC.

Two groups of systems were therefore reviewed to assist in molding the GVUI design. They include VHDL graphic environments and PC-based schematic editors.

2.2.1. VHDL Graphic Environments. AFIT has access to two systems which generate VHDL descriptions based upon graphic inputs: the General Electric Interactive VHDL Workstation (IVW), and the Vista Technologies VHDL Design Workbench (VDW).

The IVW is hosted on a Symbolics workstation . It incorporates three different editors to input the user's design: a Decision Table Editor, a graphic State Machine Editor, and a graphic Structure Editor. (The limited scope of the GVUI thesis effort shall constrain the review of the IVW to only the graphic structure editor.)

The IVW graphic Structure Editor portrays circuit entities as rectangular boxes. This portrayal occurs whether the entity is a single logic gate or a complex network of gates. The IVW is not capable of displaying design data in conventional schematic format.

The VDW is hosted on a SUN workstation. In addition to rectangular boxes, the VDW can display buffers, inverters and, nand, or, nor, xor, and xnor gates.

2.2.1.1. Observations. Both tools function quite similarly. Each tool requires formal ports to be defined for an entity interface before a structural body may be created. Once an entity is defined, each tool permits its user to save the entity in a library for later recall. When defining the body of an entity other previously defined entities in the library may be included and instantiated as components. Actual ports of instantiated components may be interconnected by wires. The wires designate VHDL signals which must also each be defined. Defining each signal interconnection between instantiated components enables these tools to generate structural VHDL code which consists of actual port declarations for each instantiated component.

In support of the automatic code generation function, the IVW and VDW both provide very similar graphic capabilities; they also each lack similar capabilities. Neither tool incorporates a real-time object drag operation. Move operations provided are nonresponsive, unpredictable and inflexible.

On the VDW if a close proximity move is performed on an object where the final object position overlaps the original position, the overlapping area of the object would be lost. A manual refresh of the screen display would have to be directed before the object would reappear in its entirety. When moving interface ports, the VDW sets up a gravity field that disallows ports from being positioned within an entity perimeter. However, the gravity field does not allow ports from being incorrectly placed outside the entity perimeter at a meaningless location. Moves are not permitted at all for connected components on the VDW. Yet, this system allows components to be rotated. It even allows components that are already connected to be rotated. However, if a component with connections is rotated its connections

are graphically severed, but its VHDL port mapping remains unchanged. The VDW user must therefore use caution when invoking the ROTATE command.

On the IVW, results of a move cannot be seen until the move is complete. Component interconnections are maintained throughout object moves. However, IVW users also deserve a word of caution when moving objects: the IVW reserves the right to rearrange previously positioned objects to accommodate a move.

Each tool incorporates an auto-route function for placing component interconnections. As presently implemented, the auto-route function is more of a hindrance to the design process than it is an aid. Connections which are auto-routed are certain to overlap the perimeters and/or ports of the components which are to be connected. (See Figure 2.) The alternative to auto-routing is to manually route the connections, placing bend points at desired locations along the connection path.

The automatic placement of object labels routinely writes over objects and often hides critical port connections. This problem is more or less pronounced on the IVW depending upon the selected magnification of zoom. The VDW does not incorporate a zoom function.

All graphic operations offered on each tool are strictly object-oriented; the user cannot perform operations such as copy, move, or delete on user defined areas within his graphical description. As a last observation, each tool appeared to have sacrificed human factor considerations to simplify its own design.

2.2.1.2. Conclusion. The ability to generate hierarchical code of actual port declarations and component instantiations is perceived to be the most valuable feature of the IVW and VDW. This feature guarantees correct syntax and consistent use of variables throughout a designer's VHDL description.

2.2.2. PC-Based Schematic Editors. Two popular and successful low-cost (under \$500) professional schematic editors were reviewed for this thesis effort. They were reviewed to determine the schematic editing capabilities that are currently available to design engineers using MS-DOS personal computers. They are OrCad by OrCad Systems Corporation (23) and

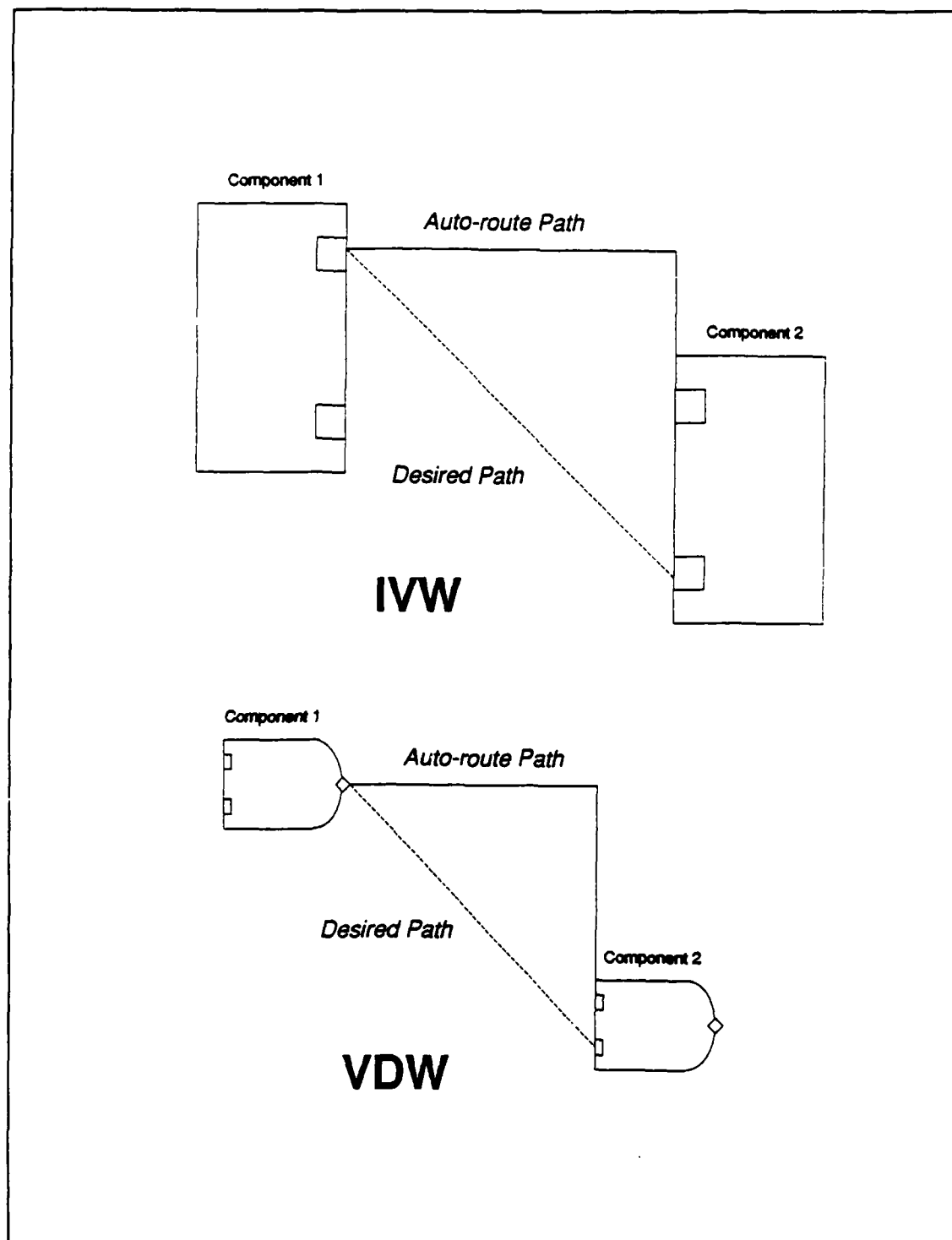


Figure 2. IVW & VDW Autorouting

Schema II by Omaton (22). Neither of these editors generate any kind of HDL source code. They are, however, each capable of generating net lists which structurally relates them to the GVUI. In effect, the GVUI also generates a net list from which it extracts the necessary data to construct a VHDL source code template.

OrCad and Schema both provide all of the graphic functions offered by the IVW and VDW. In addition, OrCad and Schema offer area commands which permit the user to perform graphic operations (ie. copy, delete, move) on entire displayed areas of the current design, versus on a single identified object.

Human factors considerations were given a high priority in the design of each tool. The noteworthy error-handling and backup capabilities of OrCad and Schema permit new users to confidently explore each tools' capabilities without being concerned about losing valuable data and time as a result of mistakes made by the user. Proper error-checking through the design of a system is vital to increasing the acceptance of a system.

Not only were human factor considerations noted in the area of error-checking and recovery, but also throughout their available array of graphic operations. For instance, when moving components the user receives continual accurate feedback as to the exact location of the component undergoing the move. To further increase response time and decrease screen clutter when moving components, OrCad redraws only an outline of the component's area rather than redrawing the entire component from pixel to pixel. Once the system no longer detects further move commands from the user interface, only then is the component fully redrawn and labeled.

2.3. Software Development

"Designing software requires both patience and bravery: Patience is needed to keep from rushing toward a solution that may, due to haste, be incomplete, and bravery is required because many discoveries will be made as we proceed from problem to solution. (25:329)"

Many different design approaches exist to aid the software engineer in laying out a design which is complete yet flexible enough to accommodate unforeseen problems, and structured in a manner which emphasizes reliability through comprehensive testing strategies. Five different approaches to software coding and testing, as presented by Yourdan and Constantine (27:376-394), were considered for this thesis effort; they are: top-down, radical top-down, conservative top-down, zigzag, and bottom-up.

2.3.1. Top-Down. The top-down approach proceeds, as the name implies, from upper level to lower level details. Consider the program structure shown in Figure 2-2. The top-down approach requires that the TOP module be coded and tested before the coding and testing of modules A, B, or C. During the coding and testing of TOP, only the existence and interfaces of A, B, and C need to be specified. After TOP is thoroughly tested, modules A, B, and C are implemented. When coding module A, only the interfaces to modules A1 and A2 need to be specified. A similar requirement exists for the implementation of modules B and C. Finally, after modules A, B, and C have been successfully tested, coding continues on modules A1 through C4.

Several advantages of the top-down approach include an emphasis on user feedback early on in the design process, simplified debugging, and the ability to enable the designer to deal with requirements which may not be defined below a certain level of detail.

The top-down approach encourages user feedback by allowing the designer to present the user with an operating preliminary version of the system while the system is still under development. Test stubs are used in place of low level modules for preliminary demonstrations and actual testing of the upper level modules.

Characteristic of incremental implementations in general, top-down design inherently simplifies debugging by making the overall debugging procedure more logically organized. That is, new modules which potentially contain bugs are added to the maturing product one at a time in a stepwise fashion.

Instances exist when a designer, by default, must implement his or her design using a top-down approach. In the absence of a complete predefined structural representation of a

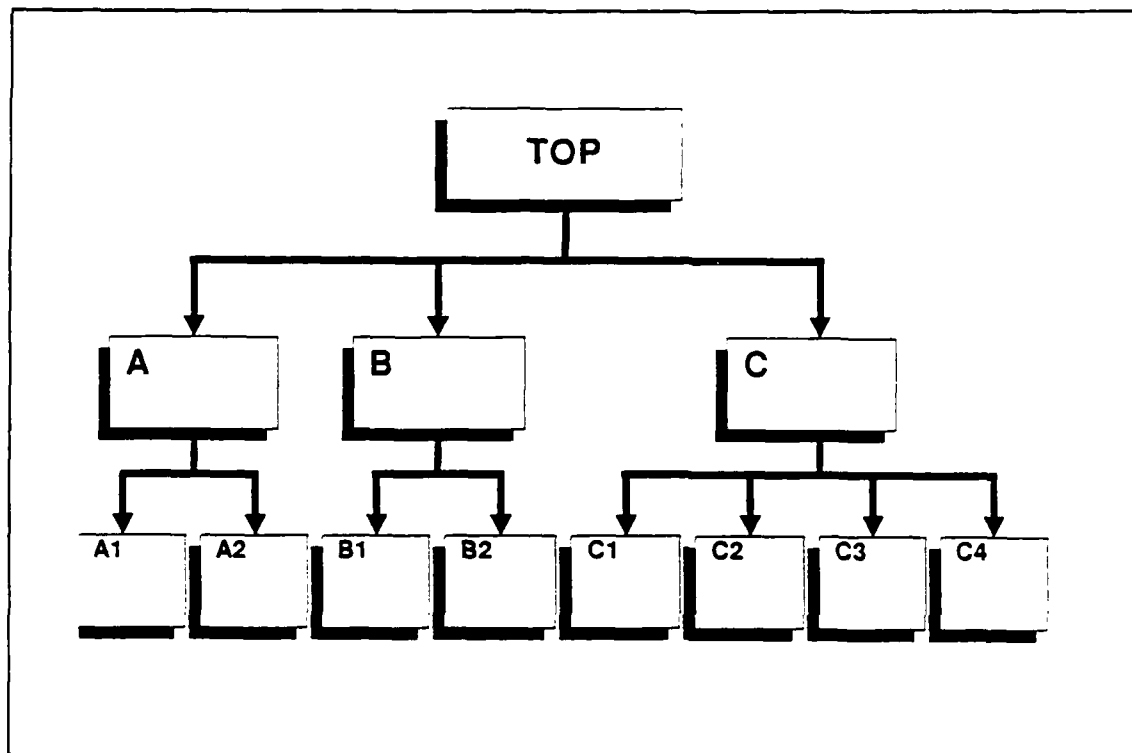


Figure 3. A Typical Program Structure

design, some detailed processes which must be performed by the system may not have been formalized by either the user or the designer. In these instances, the designer focuses his or her initial efforts on the top-level structural decisions and then, based upon those decisions directs his or her attention to more detailed aspects of the design.

2.3.2. Radical Top-Down. One extreme of a top-down approach is to design, code, and test one level, and only one level, of a system hierarchy at a time. This approach is referred to as the radical top-down or the "design as you go" approach. Its proponents argue that it provides realizable evidence of progress to the user. Yourdan and Constantine, however, warn that a strict literal interpretation of the top-down approach is impractical. Upper level modules cannot be effectively tested under all conditions if all subordinate modules are test stubs (27:391). Furthermore, unless some low-level modules are implemented and integrated early on in a design real world input and output cannot be introduced to the evolving system.

Another disadvantage of the radical top-down approach is largely a result of human nature. It is the reluctance to change what already works or what is perceived to work. When developing software this reluctance often results in the failure on the part of the programmer to combine similar low-level modules into more efficient common modules. Though the similar low-level modules may already have been proven to work, their duplication of code could negatively impact the efficiency and maintainability of the system.

2.3.3. Conservative Top-Down. Opposite the extreme to the radical approach of the top-down spectrum is the conservative top-down approach. This approach requires that a structural design be completed in its entirety and documented before any coding and testing of modules begins. Once coding does begin it progresses in a top-down fashion. Unlike other top-down approaches, the programmer always knows what lies ahead since the entire design has been meticulously mapped in advance.

An advantage of this approach is the comparative ease with which the design can be altered while in the precoding phase. The clarity and preciseness conveyed by a complete structural design simplifies the ability to detect areas in the design which could be or may need to be changed, enhanced, or deleted to improve the product's performance. Though a detected design fault may propagate throughout the entire system, its emergence during the structural design phase of the conservative approach precludes the necessity to recode, recompile, and retest modules.

The conservative top-down approach guarantees maximum module cohesion and minimum intermodule coupling in a design. As module cohesion is increased and intermodule coupling decreased, the ease of system maintenance and enhancement dramatically increases. Therefore, a properly implemented conservative top-down approach infers the most maintainable product possible.

Engineering is a science of solving real world problems under the constraints of real world boundaries. Time is the real world boundary when a designer chooses to apply the conservative top-down approach to his or her design. A considerable amount of time is required to conceive and iteratively refine a complete detailed structural design for a system

of any significant complexity. During the time the structural detailed design is being worked, no code is written. There exists no physical evidence of progress from the viewpoint of many users.

To worsen the situation, Yourdan and Constantine note that many users do not really know what they want from a system, or do not understand the consequences of what they have specified (27:393). Cosgrove points out that the programmer delivers satisfaction of a user need rather than any tangible product. Both the actual need and the user's perception of that need will change as the program is built, tested, and used (4). Under the conservative top-down approach, the designer accepts the costly risk that his or her time-consuming system analysis and design are, for one reason or another, unacceptable to the user. Again, the designer is left with little to show for his or her effort expended.

2.3.4. Zigzag. A top-down approach which focuses on completing the detailed design for some processes in the system before other processes is termed by Yourdan as the zigzag approach (27:394). Using this approach legs of the structured system hierarchy which are associated with the processes of interests are mapped out and refined to their lowest levels of detail. Once satisfied with the limited design, the programmer may begin coding the newly designed processes.

Four reasons have been formulated for applying the zigzag approach. First, a requirement may be levied on the designer to develop a system using real world input and output. Second, user requirements may dictate that certain system processing capabilities be completed early in the development schedule. Third, it provides a satisfying sense of progress to the user. The user is able to witness portions of his or her fledgling product operating as per the specifications. Fourth, in the multiprogrammer environment the zigzag approach may be unavoidable since all programmers do not produce code at the same rate.

2.3.5. Bottom-Up. Bottom-up development is characterized by the bottom-to-top sequence in which system modules are designed and/or tested. Once again, consider the program structure shown in Figure 2-2. The bottom-up approach requires that modules A1 and A2

be coded and tested separately before being packaged together under module A. Only after modules B and C have been separately coded and tested in a similar manner would modules A, B, and C be combined under the TOP module. To exercise each module under test, specialized test drivers must be written to simulate the calling module.

Yourdan and Constantine (27:389-390) imply that a primary reason for choosing a bottom-up approach is quite often a result of the politics of program management rather than any technical merits of the approach.

Consider the scenario where a number of programmers are simultaneously assigned to work on a typical large project.¹ The program manager, in fear of losing a temporarily underworked staff, keeps the programmers busy and assigns a low level module to each programmer. The programmers work in parallel, devoting their attention and efforts to the tactical details of their respective assignments without regard to the strategic objective of the overall project.

Even if the critical level of intercommunication between team members is maintained, a bottom-up approach can only complicate system integration tests. Consider modules C1 through C4 in Figure 3, page 14. Though each module may independently function as desired, where does one begin to look when the four modules fail to operate as planned when they are integrated as a single package under module C?

¹ A typical project is defined as a system which has few top-level modules and many low-level modules.

3. *System Design*

3.1. *Overview*

This chapter reviews the constraints which have been levied on the GVUI design. It also formulates the GVUI system requirements based upon the capabilities which are offered by current existing systems, and in agreement with the aforementioned constraints. Selection of the software development approach employed in the implementation of the GVUI is also discussed.

3.2. *Hardware Requirements and Constraints*

It is critical that insertion of VHDL into the academic curriculum begins immediately to insure that engineers and circuit designers of the next decade are proficient in system specification and design using Hardware Description Languages (5:324).

Providing the AVE to colleges and universities is intended to prompt the introduction of VHDL to the academic community. To further ease and assist in the transition to VHDL, the GVUI is being developed and was conceived to extend the VHDL environment to the student's or engineer's desk, and even into their homes.

3.2.1. Host Platform. Proficiency comes with practice, no matter what the subject may be. Practice is performed with less hesitance when the required tools are made readily available. In an effort to maximize VHDL's usage, thereby increasing the user's proficiency in the language, the decision was made to host the GVUI on a class of computers which is most prevalent throughout the technical community. That computer is the MS-DOS-based IBM PC-XT.

3.2.2. Video Adapters. To be most useful the GVUI should convey as much information as possible about the displayed circuit description and, at the same time, avoid a cluttered appearance. The tool would be of little benefit if the user could only see one or two

components displayed on the video monitor at any time, or if relevant information associated with each component could not be displayed due to a lack of available screen area.

Therefore, to maximize the utility of the GVUI a high resolution display capability is desired, as well as a design which optimizes the display area. Affordability shall be a key criteria in the selection of the host platform's display capabilities. Three video display cards were considered for the initial development of the GVUI: the IBM Color/Graphics Adapter (CGA), the IBM Enhanced Graphics Adapter (EGA), and the Hercules Monochrome Graphics board.¹

The CGA video card was the first color and graphics capable video board provided by IBM for use on their PC computer, the predecessor to the IBM PC-XT. It provided an affordable system in that it could drive a low-cost composite monitor, and an upgradable system in that it could also drive a medium resolution RGB monitor. However, its low resolution capabilities of 320 vertical lines by 200 horizontal lines disqualified it from further consideration.

IBM followed their introduction of the IBM PC-XT with their EGA video card which provides 640 vertical by 350 horizontal lines of resolution on a color RGB monitor. Though colleges and universities could most likely absorb the additional costs for the EGA board and RGB monitor, many students may not be able to do so.

The introduction of integrated programs which were text intensive but which also required graphics display capabilities prompted Hercules to introduce their Monochrome Graphics Board. The Hercules graphics board provides 720 x 348 lines of resolution and is compatible with low cost monochrome composite monitors. It has become a defacto standard for business graphics in the IBM PC arena (25:37). Its high resolution capabilities and low system cost resulted in it being selected for the initial development of the GVUI.

¹ The PS/2 Display Adapter which would enable the addition of IBM's VGA adapter on existing IBM PC-XT computers was not considered due to its late availability (July 1988) and high cost (\$595) (18:43).

3.2.3. Memory and Storage. The ability of the GVUI to create a library of entity interfaces and architectural body descriptions implies that at least a small capacity hard disk drive be required. This is not necessarily true. If necessary, the GVUI could be configured to operate on a dual drive or even single drive IBM PC-XT computer. On the other hand, large capacity hard drive systems result in the ability to build large data libraries which directly support the philosophy of reusable VHDL modules of code.

For large circuit descriptions, it is desirable to dynamically allocate memory to the description as the architectural body is created and grows in size. Then after saving the description and before starting a new description, the GVUI should free the previously allocated memory so that it may be reused by the new description.

3.3. Software

The primary objective of the GVUI in its completed form is to automatically generate VHDL source code from a graphically entered circuit description. Though the scope of this thesis effort is somewhat narrower in that it would culminate in the incorporation of interactive schematic editing capabilities, the phase I design is required to directly support the system's primary objective - automatic code generation.

3.3.1. Approach. Based upon the assumptions that the GVUI phase I effort can operate as a stand-alone package, a natural strategy unfolds as to how to approach its overall development. A generic schematic editor would require the capability to: edit graphical circuit descriptions, save those descriptions, and manage the support environment. The phase I GVUI would generate a data structure that is analogous to a schematic net list. The phase II effort would derive the necessary component instantiation port mapping from the data structure to generate VHDL code.

Unlike the PC-based schematic editors discussed in Chapter 2 which are constrained to call only those components defined in their libraries, the GVUI shall be required to create design entities which do not yet exist. Therefore, the GVUI phase I effort additionally

requires a capability to design, edit, save, and recall entity interfaces.

Before components may be instantiated in a circuit description, entity interfaces must be previously defined and stored, which implies a storage and recall capability must also exist. The module development dependency which is evolving, together with the expectation that an operational, tested subset of the GVUI be implemented, readily maps the GVUI software development and testing approach to the zigzag approach (discussed previously in section 2.3.5.).

3.3.2. Reliability and Maintainability. Pressman defines software maintainability as the ease with which software can be understood, corrected, adapted, and/or enhanced (19:532). The fact that the complete GVUI is a two-phased effort suggests that software maintainability is critical to the successful completion of the overall project. There may exist a number of capabilities or functions which will not have been implemented in the phase I development effort. The design shall therefore be maintainable such that any incomplete functions may be coded and integrated into the GVUI in a straightforward fashion.

A combination of two factors shall insure a high degree of software maintainability within the GVUI phase I development. They are the design of the support environment, and the software development and testing approach employed. The zigzag approach supports the philosophy of implementing one capability of the system in its entirety before starting another. In this manner, new system capabilities may be included at a later date by adding another leg at the appropriate level to the system's hierarchical structure. The support environment shall be designed so that any newly introduced functions need only be introduced to the environment manager at the desired location in the menu hierarchy so that subsequent calls to the new function are properly interpreted and executed.

3.3.3. Human Factors. Ease of use is an absolute requirement for educational tools. If the GVUI is to shorten the VHDL learning curve then it must not inhibit the learning process. For this reason, the GVUI shall not be designed around a command line environment which would require the user to learn a "mini language" just to operate the tool. Instead, it shall be

menu oriented. Menus relieve the user of the burden of remembering input options (14:334). The user may therefore concentrate not on learning how to use the GVUI tool, but rather on learning VHDL, assisted by the GVUI.

Also inherent to menu-oriented systems is error prevention. The available menu options disallow the user from selecting invalid options. Input which cannot be handled via the menu, such as a response to a system-generated request, shall be immediately checked for correctness and accepted only if valid.

The GVUI shall also include a feedback and status area where an interactive dialogue between the user and the system may occur. Within this area the user shall be prompted for input, provided with status about his or her requests and warned of any errors which may have been made. If an error should occur an audible tone shall direct the user's attention to the status and feedback area where the specific error shall be displayed.

3.4. Capabilities

This section presents an overview of the top-level functions which must occur to meet the system objectives and requirements. The functions which are responsible for schematic editing were derived through a hands-on analysis of two commercial schematic editors designed for MS-DOS personal computers. Though the reviewed schematic editors did not address or support VHDL, they provided bountiful insight into the structure and graphic abilities of two schematic editors which operate under MS-DOS and which fulfill similar needs to those which drive the GVUI development. (The reviewed editors are discussed in more detail in section 2.2.2.)

A hands-on review of two prototype VHDL environments was also conducted. The two environments supported the graphical entry of a user's design, but neither was capable of running on a personal computer. Attention was focused throughout the reviews on each tool's automatic code generation capabilities and how to integrate these GVUI phase II code capabilities into the phase I design. (The reviewed environments are discussed in more detail in section 2.2.1.)

In search of a logical solution to the GVUI development task, SofTech Corporation's Structured Analysis and Design Technique (20) was iteratively applied. The resultant model depicts the interrelationships between the available data and the functional transformations which must occur. It consists of the following six modules (shown in Figure 4): the Environment Manager, the Librarian, the Entity Interface Editor, the Architectural Body Editor, the Graphics Generator, and the Code Generator.

3.4.1. Environment Manager (1.0). The Environment Manager, module 1.0, shall be responsible for the overall control of the GVUI and is the primary link between the system and the user. Two submodules, the Input Interpreter and the Coordinate Manager, comprise the decomposition of the Environment Manager. (See Figure 5). The Input Interpreter would translate the user's input, and output internal directives to the remainder of the system or user responses to the system prompts. Before transmitting user textual input to other parts of the system the Input Interpreter shall confirm its validity. The Coordinate Manager shall track and update world and screen coordinates. Both sets of coordinates are necessary to construct and display each component in a circuit description.

3.4.2. Librarian (2.0). The Librarian, module 2.0 in Figure 4, shall permit design files to be permanently stored and retrieved at the user's request on designated disk drives. Design files shall support both entity interface and architectural body descriptions. They shall include all necessary parameters to completely reconstruct an interface or architectural body that had been previously saved.

3.4.3. Entity Interface Editor (3.0). The Interface Editor, module 3.0 in Figure 4, shall provide the user with the capability to create an entity interface. This shall include the assignment of an entity name, its formal port declarations, and an associated graphics symbol. The Entity Interface Editor shall output a .int file. A .int file shall include parameters to reconstruct the interface when editing or when incorporating it into an architectural body.

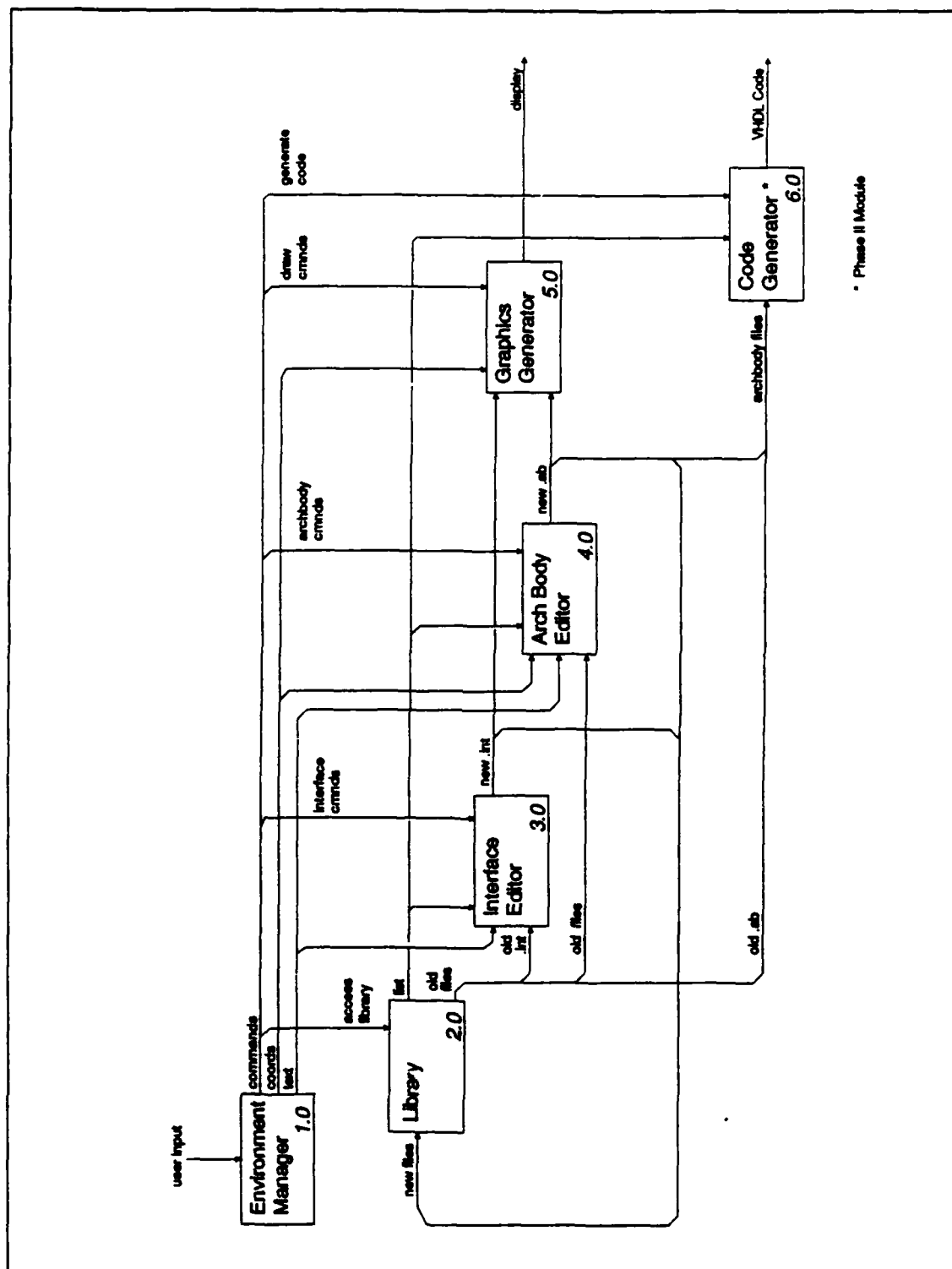


Figure 4. GVUI Top-Level SADT Diagram

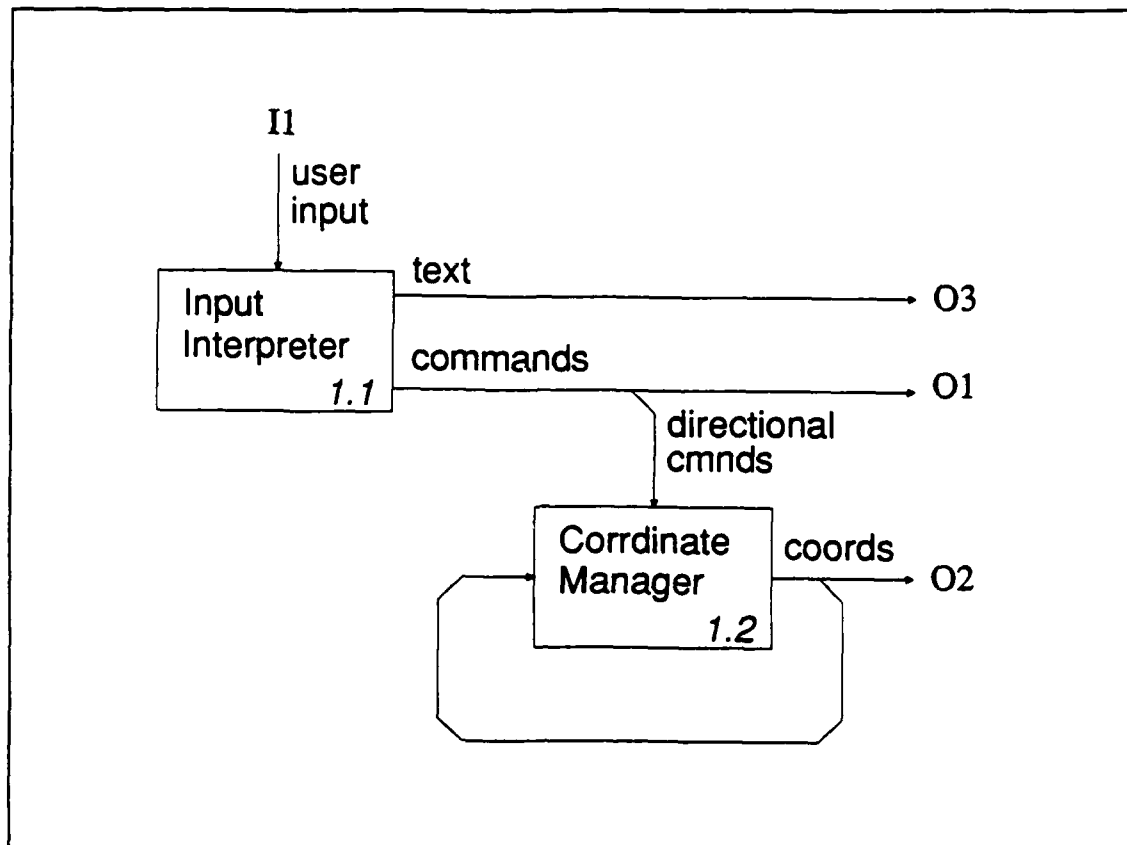


Figure 5. Environment Manager SADT Diagram

3.4.4. *Architectural Body Editor (4.0)*. The Architectural Body Editor (ABE), module 4.0 in Figure 4, shall provide the user with the ability to create or edit an architectural body of a predefined entity interface. The ABE shall also permit the user to instantiate and interconnect components. Based upon the component instantiations and interconnections, the ABE shall build and output a .ab file. The .ab file shall include parameters to reconstruct each instantiation, as well as all associated interconnections. In phase I the .ab file's utility would be in redrawing architectural bodies for editing or for visual inspection. In phase II the .ab file would additionally be required to automatically generate VHDL code.

3.4.5. *Generate Graphics (5.0)*. The Generate Graphics module shall accept .int files from the Interface Editor and .ab files from the ABE. The Generate Graphics module shall

perform the necessary transformations to enable selected entity interfaces or architectural bodies to be displayed on the display monitor. The display of architectural bodies from *.ab* files also necessitates knowing where in the world domain the current viewport is positioned so that components or connections residing on the viewport boundaries may be properly clipped.

3.4.6. Generate Code (6.0). The Generate Code module shall accept a *.ab* file as input and from that file, extract the port and signal mapping which are necessary to generate VHDL source code. To increase its flexibility and compatibility with other VHDL analyzers, Generate Code shall output VHDL source code in the form of ASCII text. Generate Code shall be formally implemented under the GVUI phase II development effort.

4. DETAILED DESIGN

4.1. Screen Layout

The GVUI screen is partitioned into four functional areas. They are the menu bar, the graphic viewport, the feedback area, and the viewport locator area (VLA), shown in Figure 6. The design and integration of each functional area sought simplicity and efficiency.

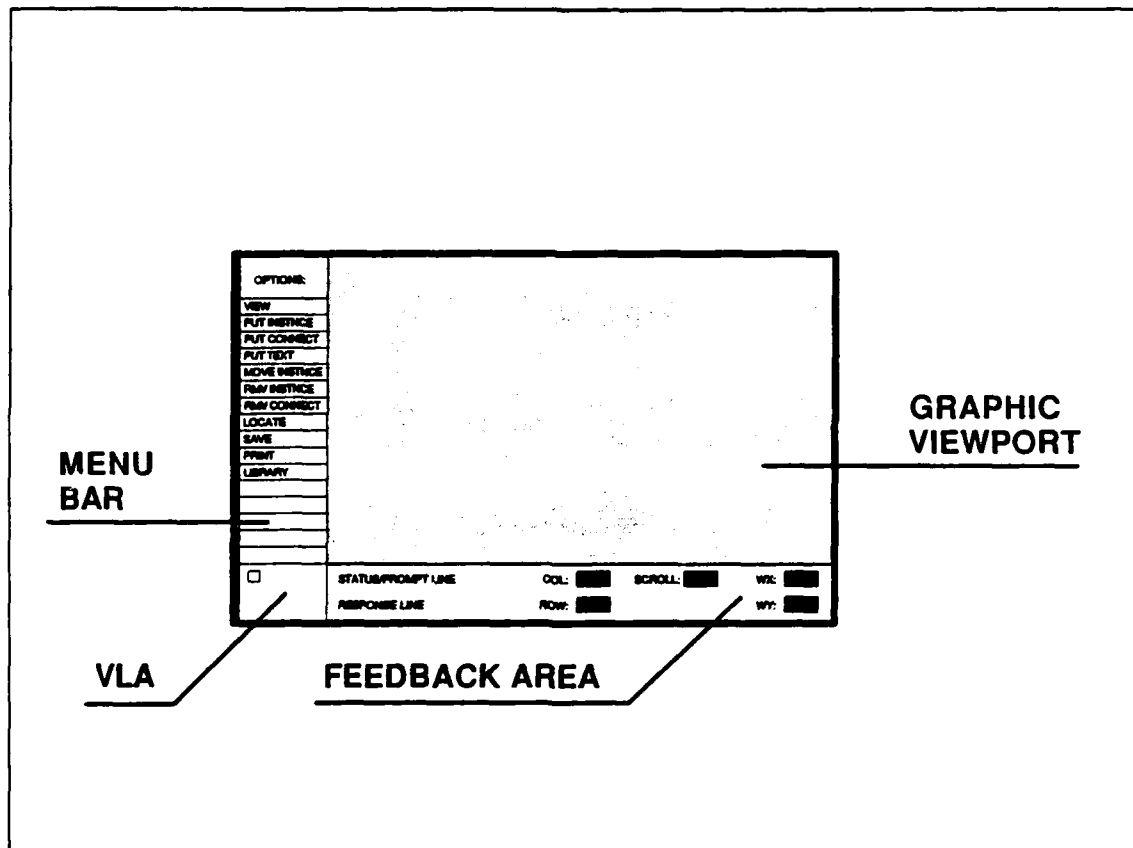


Figure 6. GVUI Functional Areas

As an educational tool the GVUI is designed to be simple to operate. It boasts self-explanatory commands which are concise and yield consistent, predictable results. Under the constraint of limited resources, the interface of the GVUI is designed to be efficient, optimizing the available screen real-estate to maximize the transferral of information to the user.

4.1.1. Menu Bar. The menu bar is the means by which a user directs and controls the execution of the GVUI. It appears on the left side of the GVUI screen and has the capability to display as many as 16 options simultaneously. Though no single menu in the current implementation requires this total capability, it has been provided as a maintainability feature to accommodate future enhancements.

Each menu option is allotted a single compartment within which the option is textually displayed. A single rectangle, referred to as the option selector, always encompasses one of the current menu options. The option selector directs the user's attention to the menu option which shall be executed upon pressing the ENTER key on the keyboard. The user may increment the option selector up or down with each press of the respective cursor direction key.

The option selector also incorporates a wrap-around capability; that is, if the option selector is commanded by the user to increment up but is already situated around the top option of the menu bar, it shall "wrap around" the list of options to the bottom menu bar option. Similarly, if the option selector is situated around the bottom menu bar option, when commanded to increment down, it shall move to the top option.

4.1.2. Feedback Area. The feedback area is located on the bottom of the GVUI screen, as shown in Figure 6. It serves six purposes: first, it displays system prompts; second, it echoes user responses to the system prompts; third, it displays error messages, including those to assist in system development and debugging; fourth, it continuously displays the current SCROLL MODE; fifth, it continuously displays the viewport locator position; and sixth, it provides a continuous readout of the current X and Y world coordinate values. The feedback

area also provides a continuous readout of the current X and Y screen coordinate values. These coordinates, those present in the prototype GVUI, are intended for development purposes only and are not required for the final GVUI production system.

System status and error messages were designed to be clear and concise so that all the information which must be exchanged between the system and the user can occur within the feedback area's two lines of text, thereby maximizing the size of the remaining graphic work area.

4.1.3. Viewport Locator Area

The VLA is located in the lower left corner of the GVUI screen below the menu bar and to the left of the feedback area. The VLA is a downscaled representation of the entire GVUI world coordinate system. The world coordinate system is presently 7038 horizontal by 3476 vertical pixels. Thus, 11.5 by 11.0 viewport areas are required to view the entire world coordinate system.

The VLA serves two purposes. First, the VLA conveys to the user the location of the graphic viewport, the users relative to the bounded world coordinate system via the viewport box. The viewport box is a small rectangular box within the VLA. It represents the current location of the graphic viewport. The viewport box is designed to move synchronously with the up, down, left, and right scrolling of the graphic viewport.

The second function of the VLA is to support the LOCATE function which enables the user to identify the location of an instantiated component within his or her current architectural description. A solid box, the component locator, highlights the relative location of a component instance that has been identified via the LOCATE function.

4.1.4. Graphic Viewport. The graphic viewport, located to the right of the menu bar and above the feedback area, serves as the window through which the user views the GVUI world coordinates. The viewport size remains constant, 612 horizontal by 316 vertical pixels. Its relative location within the world coordinate system is graphically represented by the viewport box in the VLA.

To move the viewport window to other areas within the world coordinate boundaries the user may shift, or scroll, the viewport in increments that are one-half of the displayed viewport area. That is, two consecutive shifts of the viewport in the same direction are required to view an area that had previously been situated just outside the original viewport window.

For ease of enhancement the dimensions of the GVUI world coordinate system may be varied, as well as the rate with which the viewport increments when scrolled. For the most part these changes would involve simply changing parameter definitions in the source code header file *coords.h*.

To expedite the redrawing of graphic images in the viewport only objects whose coordinates are either totally inside the viewport or outside the viewport but within a distance equal to or less than one-half the viewport area are considered for clipping. This frees the processor from calculating and drawing images which do not appear within the graphic viewport. Once the determination has been made as to which objects lie within or on the current viewport boundaries, clipping is accomplished using procedures from Borland's Turbo C Graphic Library.

4.2. Environment Manager

The Environment Manager directs the overall control of the GVUI. In response to the system design requirements analysis (see section 3.4.1.), the EM has been functionally decomposed into two submodules: the Input Interpreter and the Coordinate Manager. In addition to serving as the primary link between the system and the user, the Input Interpreter also provides AFIT with a Turbo C compatible user-interactive menu bar environment which may be incorporated into systems other than the GVUI. The Coordinate Manager continuously tracks and maintains the GVUI system coordinates which are used extensively by the ABE and the system drawing routines.

4.2.1. Input Interpreter. All user input falls under one of two categories: commands or responses. In accordance with section 3.4.1., the Input Interpreter translates user inputs as belonging to one of these two categories. If the input is a command, the Input Interpreter controls the subsequent execution of the GVUI based upon that specific user command. If the input is a response to a system prompt, the validity of the response is confirmed. If the response proves to be valid, the Input Interpreter transmits the response to the intended recipient submodule. If the Input Interpreter determines the response to be invalid, an audible tone precedes a warning message which temporarily appears in the feedback area, or the response is ignored. The Input Interpreter's reaction to an invalid response depends on the severity of the detected error.

For instance, suppose the system prompts the user to enter a name for a disk file that is to be created. If the user enters no name but simply presses the RETURN key, the Input Interpreter ignores the RETURN and continues to wait for a valid response. On the other hand, if the user unknowingly enters a filename that already exists, the system would audibly and visibly warn the user that the named file already exists. It would then prompt the user for further directives.

As a result of the GVUI being a menu-oriented versus a command-line system, the command translation performed by the Input Interpreter is more of an administrative management function rather than an interpretive function. By tracking the current menu level and location of the menu option selector, the Input Interpreter is only required to retrieve a selected command from its list and linkages of predefined options.

4.2.1.1. Menu Environment. To insure that a workable subset of the phase I GVUI could be implemented and demonstrated, coding and testing were incrementally accomplished using the zigzag development approach. The zigzag approach inherently supports the philosophy of developing one function at a time in its entirety before starting another. This implies that the menu environment be functionally complete to a level of abstraction that is at least as detailed as the functions that are to be implemented.

Therefore, the menu environment was designed, coded, and tested in its entirety before any low level modules were incorporated. This does not infer that the menu environment, once coded, could not be altered. On the contrary, the menu environment is designed independent of a given menu hierarchy. In general, its flexible design permits any number of levels to be implemented, each level with up to sixteen selectable options.

Links between any single option in any level may, upon selection, direct control to any other predefined menu level or to the execution of the selected function. Control links within the environment are not constrained to be top-down in nature, but may also be bottom-up; that is, each level in the menu hierarchy may be assigned an escape link which permits the user to return to a previously accessed level within the menu hierarchy.

A benefit of the menu environment design is that a generic Turbo-C compatible menu environment is now available for use by AFIT and the public domain. By accessing Borland's graphic drivers, the environment is capable of operating with not only a Hercules monochrome graphics card, but with CGA, EGA, and VGA boards as well.

4.2.1.2. Menu Hierarchy. A list of baseline functions for the GVUI system were derived from the GVUI requirements analysis. The list was structured into a hierarchical arrangement, each level in the hierarchy being more detailed than the previous level. The final menu hierarchy evolved to that shown in Figure 7.

Four of the six top-level system requirements (as defined in section 3.4.) may be mapped directly to the GVUI top level menu options. (The top level menu is hereafter referred to as the Main Menu). They are the Library, the Interface Editor, the Architectural Body Editor, and the Code Generator. The zigzag approach enables these top-level functions to be singly developed one at a time before pursuing the implementation of the others.

Remaining are system requirements for an Input Interpreter and a Graphics Generator. Employment of the zigzag approach results in the Input Interpreter being, in effect, the main routine which directs executable control to one of the four modules defined above. The Graphics Generator, which cannot be directly mapped into the menu hierarchy, provides a

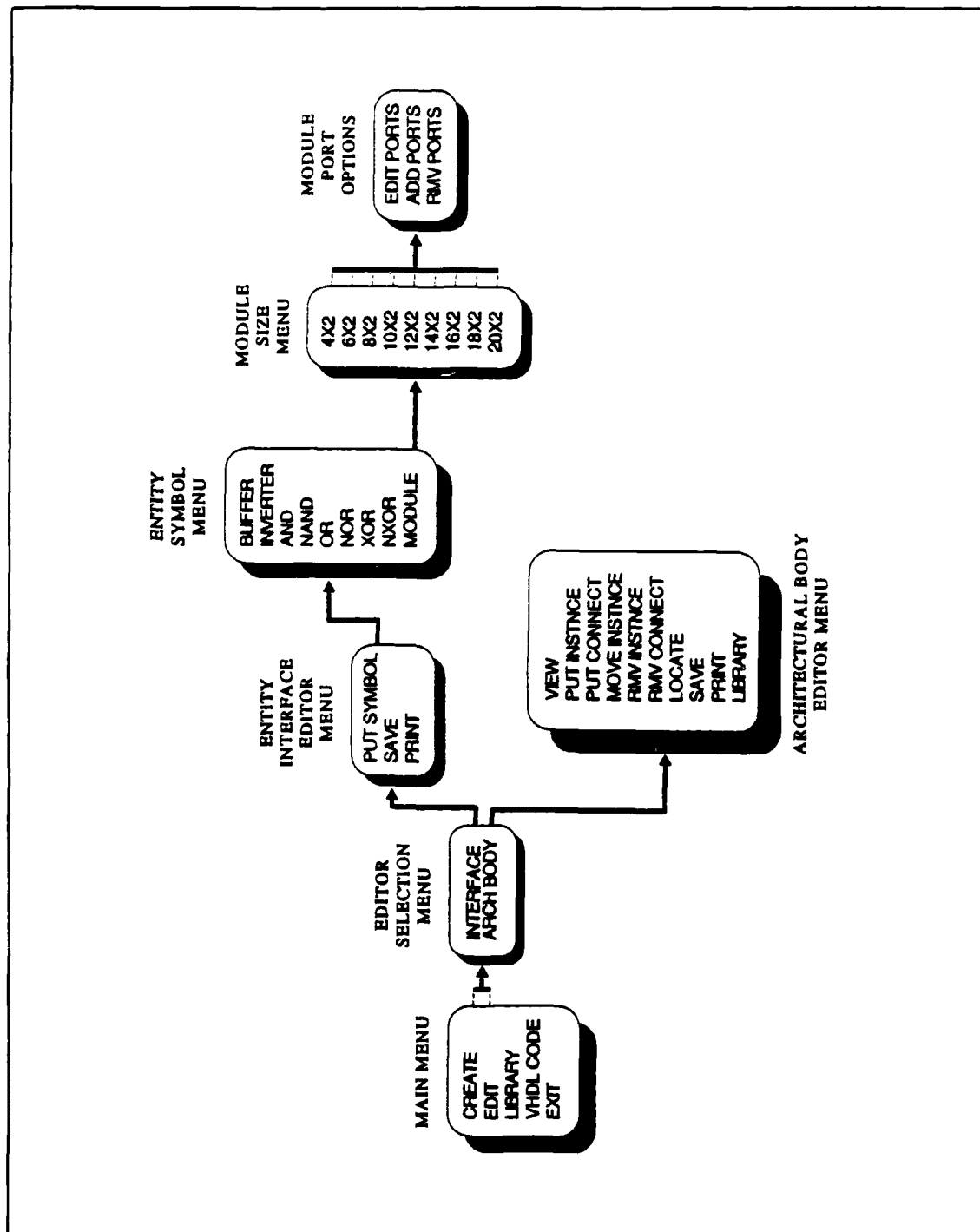


Figure 7. GVUI Menu Hierarchy

set of routines for generating and displaying graphic objects. The routines are shared extensively by the Entity Interface Editor and the Architectural Body Editor.

4.2.1.3. *Data Structure.* Two types of data forms, referred to as structures in the C language, are required to establish a menu hierarchy. In the GVUI source code, these structures are named *lvl* (for level) and *map*.

An array of *lvl* structures is declared for each level, or list of menu options, in the system menu hierarchy. Similarly, an array of *map* structures is declared for the system menu hierarchy. The length of the array corresponds to the number of separate menu levels in the system menu.

The *lvl* and *map* structures are defined in the source code header file *menudef.h* as follows:

```
struct lvl
{
    char          cmd[12];
    struct map    *next;
    int           lindx;
};

struct map
{
    struct lvl    *first;
    int           length;
    struct map    *last;
    int           mindx;
};
```

The *length* of the array of *lvl* structures for each menu level is equivalent to the number of selectable options in the current menu. Elements of each *lvl* structure include an option, or command name, which is an array of characters, *cmd[12]*; a pointer to a *map* structure, **next*; and an integer, *lindx*. The command name, *cmd*, is the name assigned to the n^{th} option of the current menu, where $n = \text{lindx}$. The pointer to the next *map* structure, **next*, serves as the link to the next lower-level menu to be displayed upon selection of the current option in the current menu.

Members of each *map* structure include a pointer, **first*, to the *lvl* structure associated with the first selectable option in the current menu (which is indexed in the *map* structure

array by *mindx*); an integer, *length*, which is the number of selectable options in the current menu; a pointer to a *map* structure, **last*, which serves as the link to the menu which is to be displayed upon the user indicating the desire to escape from the current menu to a previous upper-level menu; and an integer, *mindx*, which is the associated index for the current *map* structure in the *map* structure array.

Referring back to Figure 7, it may be seen that the forward links which take the user from upper-level to lower-level menus are the **next* pointers which are resident within each *lvl* structure. If the **next* pointer is a null pointer no lower-level menu is associated with the current option. This signifies to the Input Interpreter that the user has selected an executable option.

When the **next* pointer is not assigned a null value and a lower-level menu is subsequently displayed to the user, the **first* pointer in each *map* structure serves the purpose of providing a navigational bearing to the menu environment so that the system knows its current location in the menu hierarchy.

Reverse links return the user from lower-level to upper-level menus. They are the **last* pointers which are resident in each *map* structure. The only occurrence of a null **last* pointer is associated with the upper-most main menu. A null value for **last* provides additional navigational input to the system, indicating that the main menu is currently displayed.

4.2.1.4. Control Flow. The Input Interpreter was designed in accordance with the zigzag development approach to enable incomplete functions and future enhancements to be easily integrated into the overall system. As new functions are conceived and coded their identifying name may be included in the desired menu level list.

When new functions are added, the *length* of the associated *map* structure must be incremented to accommodate the new function. A function call to the newly coded module may be placed in the Input Interpreter routine in a location which is associated with the menu level in which the function's identifying name is placed. Under normal operation when the

newly inserted option is selected, its *map* structure array index, *mindx*, and its menu level position index, *lindx*, shall direct the Input Interpreter to execute the new function.

The flow of control through the Input Interpreter is pictorially represented in the simplified flowchart shown in Figure 8.

4.2.2. Coordinate Manager. Coordinate management is critical to both phase I and phase II GVUI development efforts. Phase I requires accurate coordinate information in order to generate, recall, and display graphic files and images. Phase II will extract coordinate information from the generated graphic files to reconstruct the component port interconnections of architectural bodies. The reconstruction of interconnections will enable the GVUI to determine the signal-port associations within the architectural bodies. These associations are critical to the GVUI code generation capabilities.

Another important coordinate management function is cursor control. Movement of the cursor is the stimulus which triggers coordinate updating to occur. Therefore, the responsibility of cursor movement is also assigned to the Coordinate Manager.

The Coordinate Manager is responsible for the initialization, maintenance, and availability of all coordinate data used throughout the GVUI system. Three coordinate systems are continuously maintained by the Coordinate Manager. They are world, screen, and viewport coordinates. Transformations are applied to these coordinates when needed to obtain equivalent scrolled zoomed screen (SZS) coordinate values. All coordinate systems specify their top-left point as the origin with increasing X values progressing to the right and increasing Y values progressing downward.

4.2.2.1. World and Screen Coordinate Systems. The world coordinate system represents the entire work area that is available to the user to manipulate his or her graphic description. The world coordinate system is not constrained to a particular output device. The world coordinate dimensions of the GVUI phase I prototype are 7038 horizontal by 3476 vertical pixels. For ease of future enhancement these values may be altered by changing the XWRLD and YWRLD definitions in the source code header file *coords.h*. Since every point in the

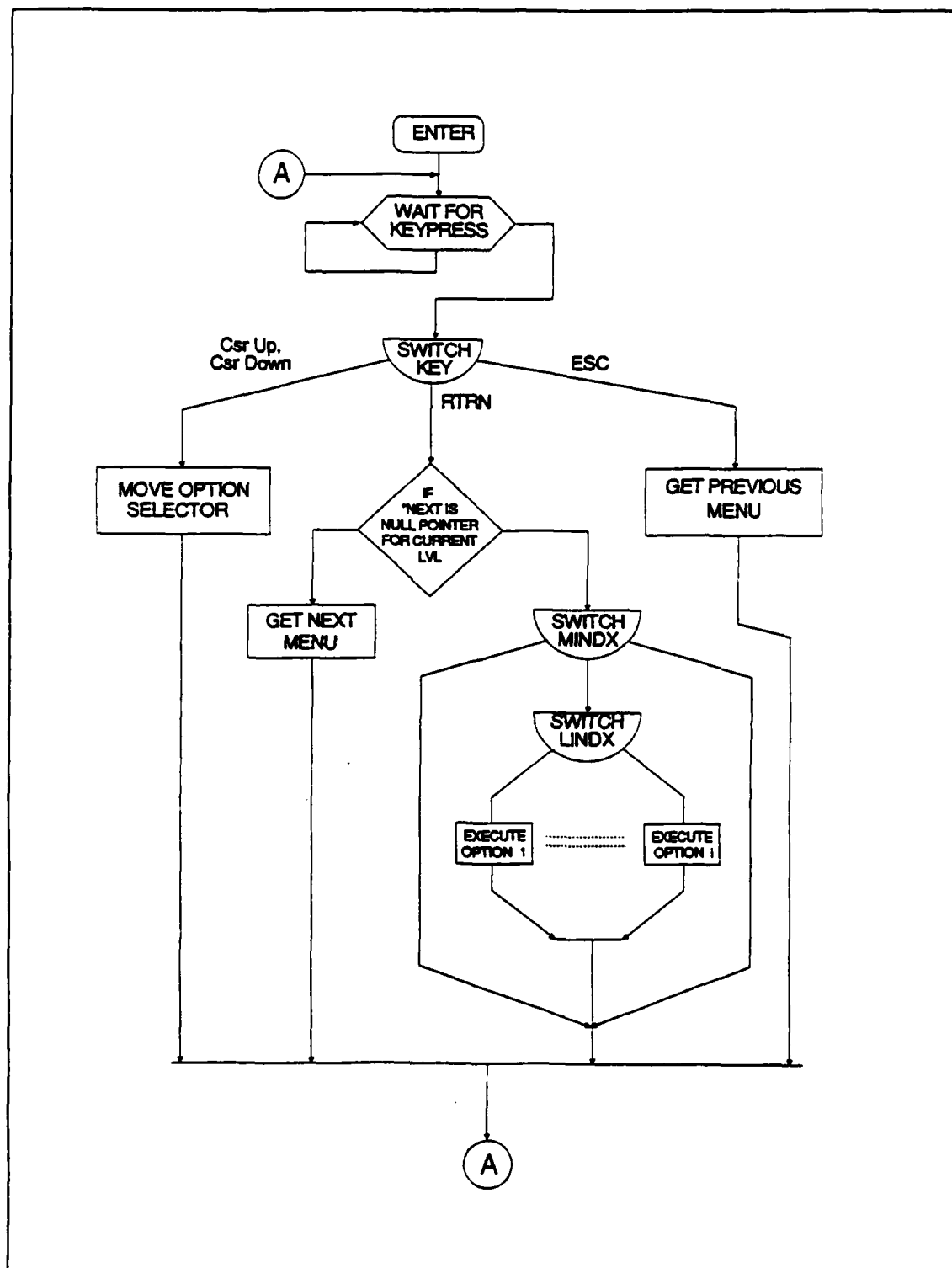


Figure 8. Input Interpreter Flowchart

world coordinate system is addressable by one and only one set of X and Y coordinate values, component objects within an architectural body are positioned, referenced, and saved in terms of world coordinates. To assist the user in positioning objects, current X and Y world coordinate values are continuously displayed in the GVUI feedback area.

The graphics generation routines which are responsible for the final drawing of graphic objects on the display screen require a coordinate system that is specific to the display monitor and video adapter. The coordinate system that satisfies this requirement is called the screen coordinate system. Its dimensions are dictated by the resolution of the video adapter. In the case of the prototype GVUI, the dimensions of the screen coordinate system are 720 horizontal by 348 vertical pixels. These values, which are due to the Hercules monochrome graphics card, are defined as XHERC and YHERC respectively in the *coords.h* source code header file.

Since the screen coordinate system encompasses the entire output display, including the menu bar, feedback area, VLA, and borders around the graphic viewport, zoomed screen coordinates are required to display and clip objects within the graphic viewport. Zoomed screen coordinates effectively narrow the viewport of the display monitor from the entire screen area to that of the GVUI graphic viewport. This prevents objects being drawn from exceeding the boundaries of the GVUI graphic viewport.

4.2.2.2. Viewport Coordinate System. Viewport coordinates are simply the integer *row* and *col* values of the viewport locator within the VLA. Each increment of the viewport coordinates coincide with movement of the graphic viewport over the world coordinate system. Each time the viewport shifts as a result of a scroll directive it moves a distance equal to one-half of the corresponding dimension of the viewport. Recall that the dimensions of the GVUI viewport are 612 horizontal by 316 vertical pixels. If the user directs the viewport to scroll right or left it would do so by 306 pixels, one-half of the viewport's horizontal dimension. Likewise, a user directive to scroll down or up would result in the viewport shifting by 158 pixels, one-half of the viewport's vertical dimension. Therefore,

the world coordinate system (7038 horizontal by 3476 vertical pixels) is equivalent to 23 by 22 viewport scroll increments. (See Figure 9).

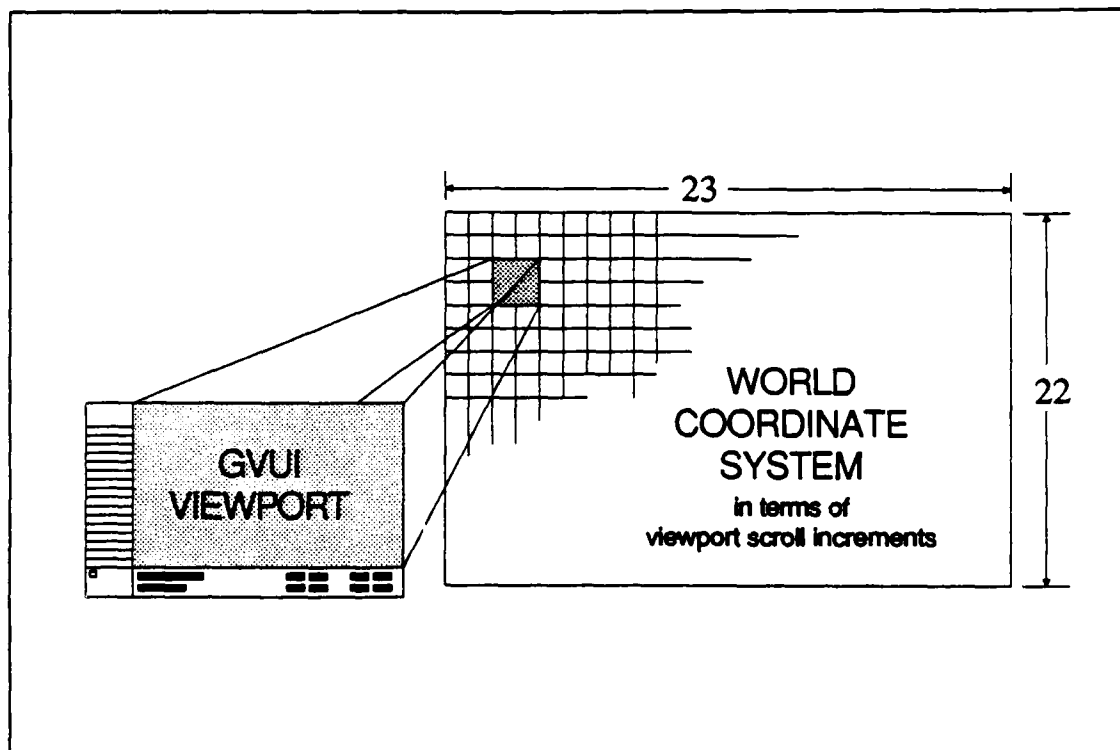


Figure 9. World Coordinates and Viewport Scroll Increments

Viewport coordinates are maintained for four reasons. First, they are used to position the viewport locator in the VLA. The viewport locator moves synchronously with viewport scrolling so that the user is aware of his or her location within the world coordinate system. Second, viewport coordinates are associated with each graphic component to assist in determining which components need to be redrawn or clipped with each refresh or scroll of the graphic viewport area. This enables the GVUI to perform screen refreshes in an efficient and rapid manner.

Third, the viewport coordinates are directly used in SZS coordinate transformations. SZS transformations perform the graphic object translations which are necessitated by

viewport scrolling. For instance, consider the case where a component is located at world coordinates (X,Y) and is displayed to the right of center on the GVUI display. Suppose the X and Y world coordinate values translate to zoomed screen coordinates (Xs, Ys) before any scrolling occurs. (See Figure 10). Upon scrolling right one increment, the world coordinates (X,Y) associated with the component remain unchanged. However, the zoomed screen coordinates, (Xs, Ys) undergo the SZS transformation to compensate for the viewport shifting right by one-half of its horizontal dimension. Therefore, after scrolling the new zoomed screen coordinates are (Xs-x, Ys) where x is one-half the horizontal dimension of the GVUI viewport.

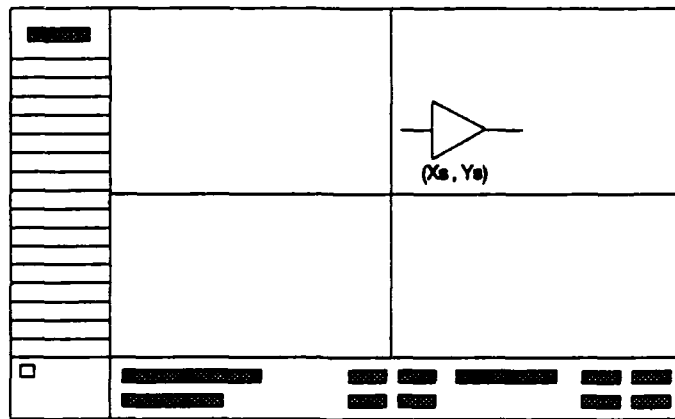
Lastly, viewport coordinates support the LOCATE function which is used to identify the location of a component in the world coordinate system. The viewport coordinates of the component to be identified are used to position a solid viewport box, the component locator, within the VLA. The component locator highlights the location of the requested component relative to the current viewport location.

4.2.2.3. Cursor Movement. A crosshair cursor becomes visible in the GVUI viewport upon entering the Architectural Body Editor. Using the right, left, down, and up arrow keys, the user may move the cursor across the graphic viewport. To increase the rate of advancement, the user may hold down one of the SHIFT keys in conjunction with one of the four arrow keys. This increases the rate of cursor advancement by a factor of 10. The advancement rate is defined in terms of number of pixels and may be found in the cursor.c source code file as follow:

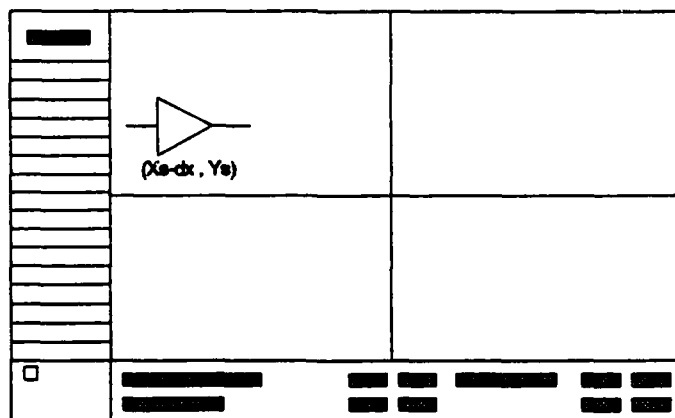
```
#define UDFAST 60 /* fast up/down cursor speed */
#define UDSLOW 6 /* slow up/down cursor speed */
#define LRFAST 80 /* fast left/right cursor speed */
#define LRSLOW 8 /* slow left/right cursor speed */
```

The differences between the horizontal and vertical rates of movement have the partial effect of visually normalizing the nonsquare aspect ratio of the host platform video adapter.

Before the Coordinate Manager executes a cursor move directive it insures that the cursor will not move beyond the boundaries of the GVUI viewport. If the user desires to



BEFORE RIGHT SCROLL



AFTER RIGHT SCROLL

Figure 10. Scrolling and SZS Coordinates

work beyond the current viewport boundaries, he or she must shift the viewport in the desired direction by pressing the ALT key in conjunction with one of the arrow keys.

The GVUI provides the user with two modes of viewport scroll. The modes may be toggled by pressing the F9 key. The default mode is referred to as cursor dependent scroll. When in this mode, "Mode: DEP" is displayed in the feedback area. When the viewport is scrolled in the cursor dependent mode, the cursor remains at the same world coordinates throughout the scroll. This mode of scroll is advantageous when placing components or connections near the current viewport boundary because all relative distances between the cursor and other objects in the viewport are maintained throughout the scroll. However, dependent scroll constrains the user to scrolling to areas which adjoin the screen quadrant in which the cursor lies. (See Figure 11 for an example of cursor dependent scrolling).

The alternative scroll mode is cursor independent scroll. When in this mode, "Mode: INDEP" is displayed in the feedback area. When the viewport is scrolled while in the INDEP mode the Coordinate Manager attempts to maintain the same relative screen coordinates throughout the scroll. Since execution of this mode is not constrained by the cursor world coordinates before the scroll, this scroll mode permits the user to rapidly move the viewport across the world coordinate system. (See Figure 12 for an example of cursor independent scrolling.)

4.3. Entity Interface Editor

Before a detailed architectural description of an entity may be defined the entity interface must be known. The Entity Interface Editor provides the user the capability to create interfaces for previously undefined circuit entities, thereby permitting their inclusion in architectural bodies of other circuit entities.

A named entity interface minimally requires that its ports have a specified mode and a formal name. Provided with this information the Entity Interface Editor is able to build a .int file. The .int file contains all the information that is required to not only reconstruct the entity's graphical representation, but also to label and identify all of its ports. The

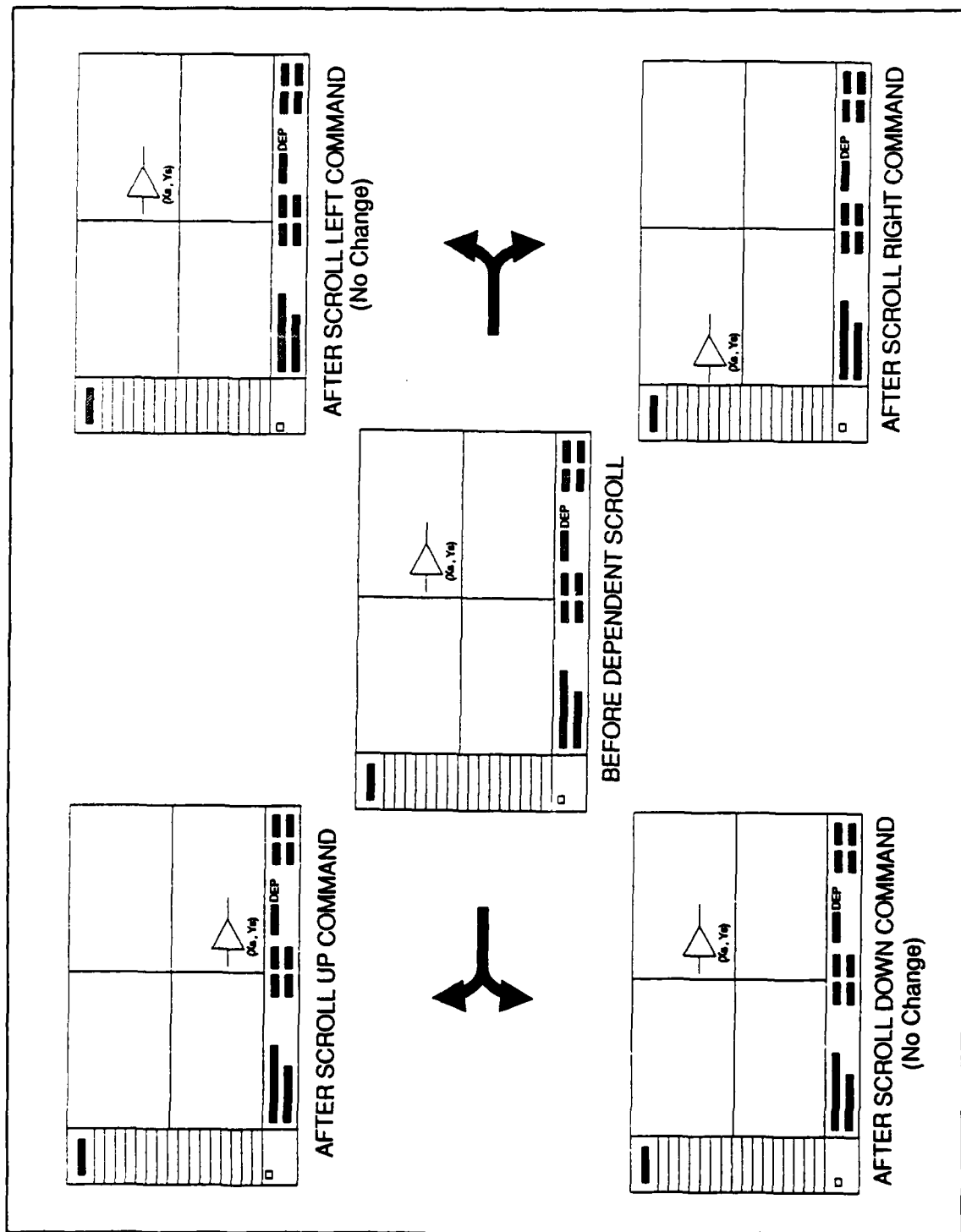


Figure 11. Dependent Scrolling

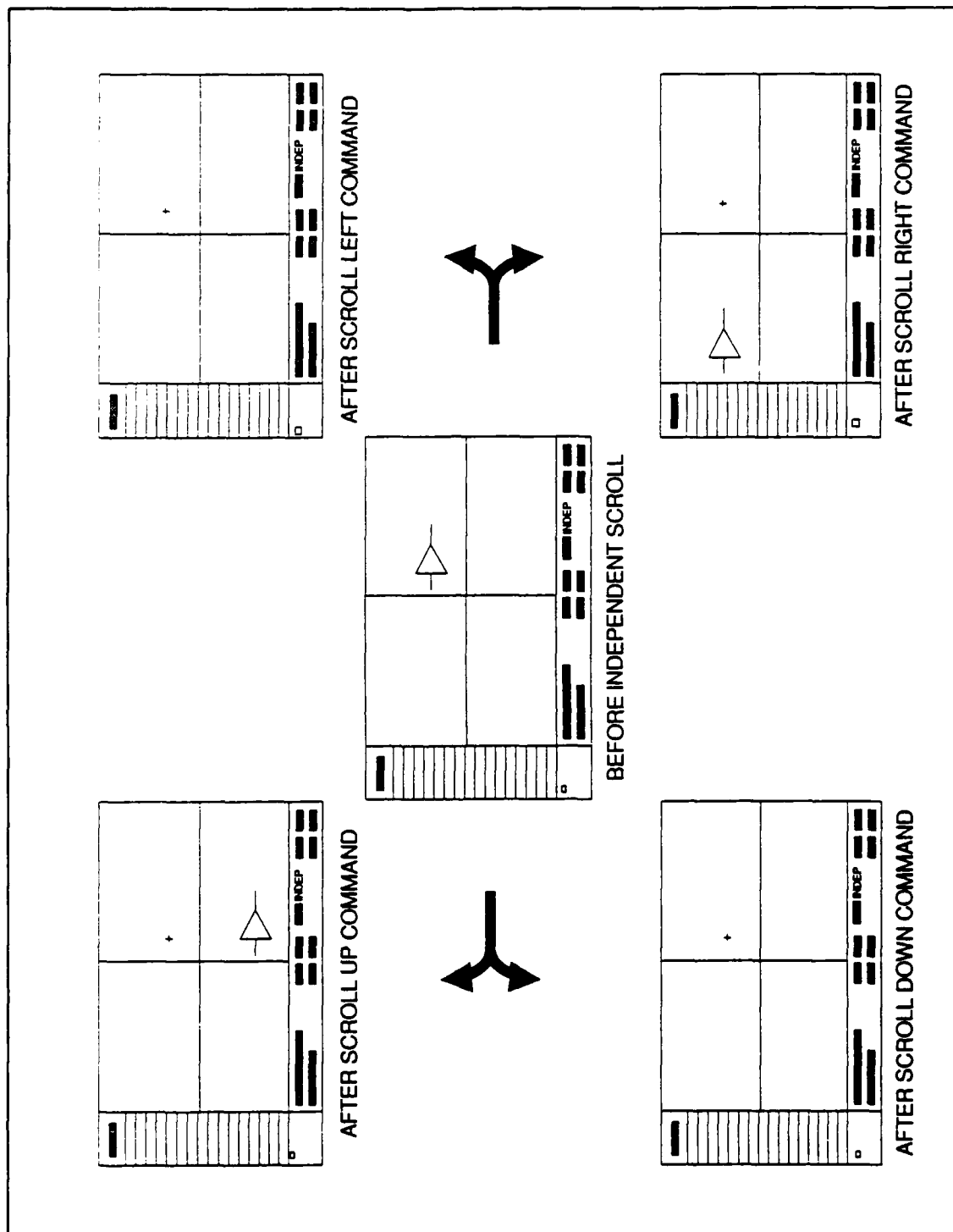


Figure 12. Independent Scrolling

sections that follow will describe the operation of and data structures for the Entity Interface Editor in further detail.

4.3.1. Symbol Design. Much effort was expended in the design and layout of the GVUI screen display and symbol design to enable a maximum amount of information to be transferred to the user via a minimal amount of screen real estate. As mentioned earlier, the Hercules monochrome graphics card was selected as the prototype GVUI's video interface adapter. Though other video boards may be accommodated by altering coordinate values in the *coords.h* header source file, the symbols used to graphically represent entities have been designed based upon the horizontal to vertical aspect ratio of the Hercules video card.

Operation on systems using video adapters other than the Hercules graphics card would require a scaling transformation be added to functions in the *drawsymbols.c* source file. The scaling transformation would maintain the same horizontal and vertical dimension proportions which appear on the prototype GVUI.

Once the user has accessed the Entity Interface Editor the GVUI presents a list of nine possible symbols from which one may be selected to graphically represent the entity being designed. The first eight symbols may be recognized as the following digital logic gates: *buffer*, *inverter*, *and*, *nand*, *or*, *nor*, *xor*, and *nxor*. Each of these symbols may be assigned to have 2, 3, 4, 8, or 16 input ports. The ninth symbol option is referred to as a module. A module is represented graphically by a rectangular box which may have as many as 40 ports.

All symbols have enough area within their perimeter and between their ports to display the formal port labels. The entity name is displayed below the symbol. The symbols are drawn based upon vector formulas which are functions of symbol type and port number. The *buffer* and *inverter*; *and* and *nand*; *or*, *nor*, *xor*, and *nxor* symbols share the same basic formulas to draw their perimeters.

Since the ability to rotate components in architectural bodies would increase the flexibility of the GVUI, symbols were designed with the future integration of rotational capabilities in mind. Rotational considerations also played a significant role in determining

the distance associated with each horizontal and vertical incremental movement of the cursor. These nonvisible incremental distances are herein referred to as gridpoints.

The Hercules graphics card has an aspect ratio of 4:3; that is, for every four screen pixels in the horizontal direction there exist only three pixels in the vertical direction. Consider drawing a square on such a system. If it is desired that the square appear to be 12 pixels on a side using the horizontal dimension as the reference, the system would have to draw a rectangle with dimensions of 12 pixels horizontally by 9 pixels vertically.

Now consider drawing a square on the same system so that it appears to be 10 pixels on a side. This time the system would calculate a rectangle with dimensions of 10 pixels horizontally by 7.5 pixels vertically. The system would then be forced to draw a rectangle that was either 10 by 7 pixels or 10 by 8 pixels. The exact equivalency cannot be projected on the video display as a result of the video aspect ratio.

Once more, consider what happens if a 12 by 9 pixel rectangle that appears as a square on a display with a 4:3 aspect ratio is rotated 90 degrees. With no scale factor to normalize the size of the object before the rotation, after the rotation the square would appear to have a vertical dimension twice that of its horizontal dimension. The rotation would have significantly compressed the horizontal and expanded the vertical dimension.

To further complicate the situation the GVUI cursor is only permitted to move to specific grid points throughout the world coordinate system. If an object were rotated 90 degrees, ports which previously aligned with the grid points in one dimension would not align in the other. Though a rescaling factor could be used to resize the object to its original dimensions, if the objects are not carefully thought out, the rescaling factor would result in noninteger dimensional results. This would not be acceptable since the signal-port associations, which are required for the phase II code generation capability, would be severed.

In anticipation of this occurrence all symbols were designed such that a combination of the permissible overall horizontal and vertical dimensions of each symbol and the rescaling factor used would provide integer dimensions upon rotation.

The nonvisible grid points within the GVUI world coordinate system occur every 8 pixels horizontally and every 6 pixels vertically. These values enable a symbol that has been rotated 90 degrees to maintain its original appearance and also enables its port locations to realign with the world grid points after rotating.

4.3.2. Control. (Throughout this section, the reader may refer to Figures 13 and 14, Entity Interface Editor Flowchart). Upon entering the Entity Interface Editor the user has the option of creating a new entity interface or editing an existing interface. If the user decides to create a new interface the system would prompt the user to assign a graphic symbol to the entity. The symbol options would be presented on the menu bar and would include the following: BUFFER, INVERTER, AND, NAND, OR, NOR, XOR, NXOR, and MODULE. Once a symbol has been selected the system prompts the user to enter the number of interface ports. Knowing the total number of ports associated with the entity being designed, the size of the entity interface data structure is calculated and memory is dynamically allocated to accommodate it. During the memory allocation procedure, blocks of memory are linked via pointers to establish the shell of the entity interface data structure. Relative locations between ports and default port labels are derived from the drawing formulas which are resident in the *drawsymbols.c* source file and are assigned to the proper elements of the entity interface data structure.

Port mode types are also stored in the data structure but they are embedded as part of the formal port labels. The first character of the formal port label identifies the mode of the port. The five possible port modes and their identifying character are: in (I), out (O), inout (Z), buffer (B), and linkage (L).

Rather than create a new entity interface, if the user decides to EDIT a previously defined interface, the system would prompt the user for the name of the *.int* file to be edited. Once the system verifies that the file exists, it determines the symbol type and number of ports from the first two entries in the *.int* file. With this information memory is allocated and linked to establish the shell of the entity interface data structure. Now that the size of the

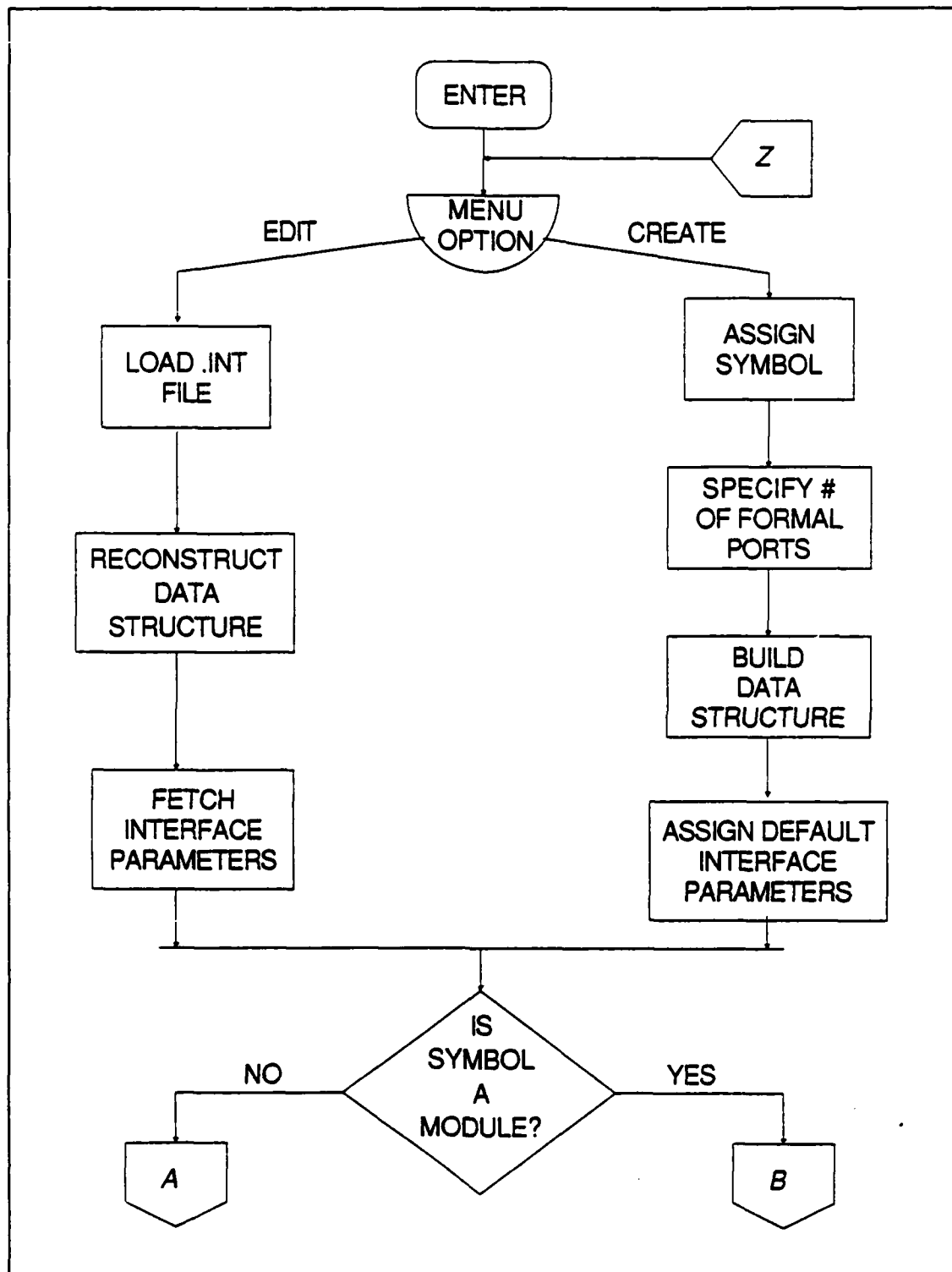


Figure 13. Entity Interface Editor Flowchart

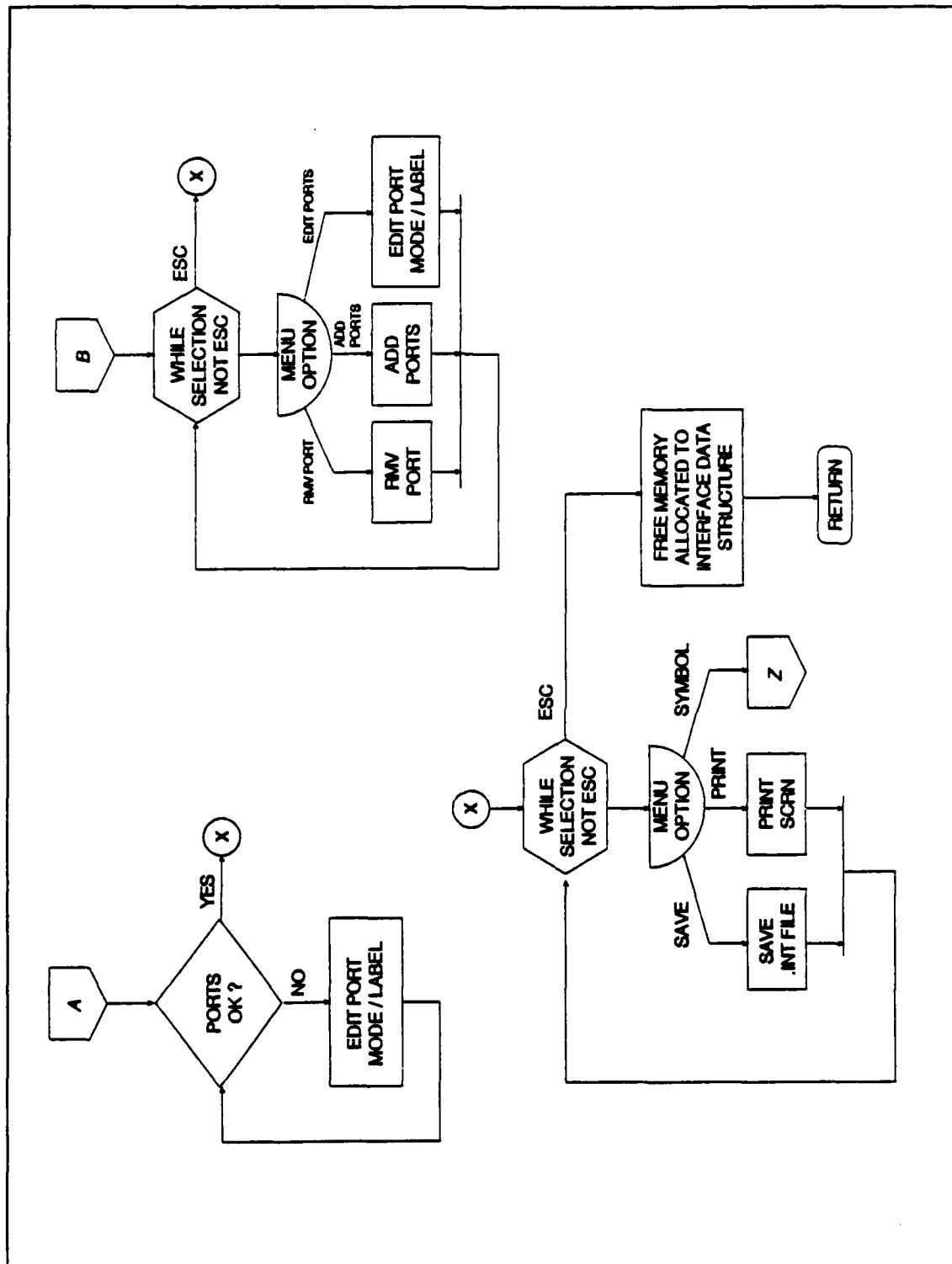


Figure 14. Entity Interface Editor Flowchart

empty data structure correlates with that of the *.int* file, the file is read and the members of the entity interface data structure are filled.

At this point, it does not matter whether the entity interface was loaded from a *.int* file or created interactively; formal modes and labels may now be edited at the user's option. Once the user edits the port modes and labels to his or her satisfaction, the system provides the user the option to either save the entity interface as a *.int* file, print the screen for archival purposes, immediately free the currently allocated memory and begin designing a new entity interface, or escape to the main menu. Escaping to the main menu also causes the memory that is allocated to the current entity interface to be freed for reuse.

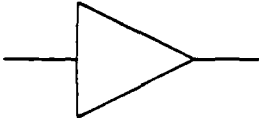
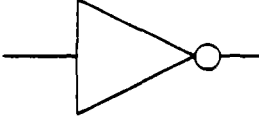
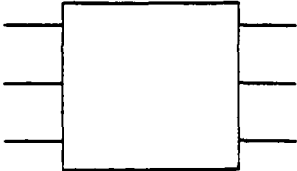
4.3.3. Data Structure. Two types of data forms are required to create the entity interface data structure. These C structures are named *interface* and *portinfo*. They are defined as follows in the *port.h* source code header file:

```
struct interface
{
    struct    portinfo *first;
    int       symbol;
    int       num;
    char      ename[13];
    int       xname;
    int       yname;
};

struct portinfo
{
    int       id;
    char      pname[8];
    int       xcon;
    int       ycon;
    int       xlabel;
    int       ylabel;
    struct    portinfo *next;
    struct    portinfo *prev;
};
```

An *interface* structure is declared for each entity interface that is to be created or edited in the Entity Interface Editor. The *interface* structure contains information that is required to construct the data structure shell. *Symbol* is an integer value from 1 to 9 inclusive. It specifies the graphic symbol representation for the entity interface being edited. Table I associates a symbol type to each permissible integer value.

Table I. Entity Symbol Types

<i>SYMBOL</i> (INTEGER VALUE)	MENU OPTION	<i>SYMBOL</i> (GRAPHIC)
1	BUFFER	
2	INVERTER	
3	AND	
4	NAND	
5	OR	
6	NOR	
7	XOR	
8	NXOR	
9	MODULE	

Num is also an integer. It specifies the number of ports in the entity interface. Associated with each formal port of the entity interface is a *portinfo* structure. *Num* is therefore critical to the dynamic memory allocation routine since it determines the amount of memory that is to be reserved to accommodate all necessary *portinfo* structures.

Ename[13] is an array of characters which contains the entity name. Its X and Y coordinate values, relative to the symbols hook point, are specified by the integers *xname* and *yname*. The relative coordinate values are used to position the text label on the graphic screen.

The *interface* structure contains general information about the entity interface being edited. Information about each formal port is contained in a *portinfo* structure that is associated with each formal port. When *num* is used to allocate memory for each *portinfo* structure, links are established between the blocks of memory as they are allocated, thus creating a linked list of *portinfo* structures. The **first* pointer in the *interface* structure points to the memory block that has been allocated for the first *portinfo* structure. *Id*, the first element in the *portinfo* structure, identifies each port in the interface. For instance, if an entity interface has eight ports, *id* for the first port would have the integer value 1; *id* for the second port would have the value 2; and so on. The last port would have an *id* value of 8.

Pname[8] is an array of characters which contains the formal port name. Its X and Y relative coordinate values are given by the integers *xlbl* and *ylbl* respectively. A connection lead is drawn to each port to assist the user when interconnecting components in an architectural body. Relative X and Y coordinates are also provided as inputs for the connection drawing routines. These integer values are given by *xcon* and *ycon*, respectively.

The linked list of *portinfo* structures that is created is a doubly-linked list; that is, links are maintained which point from one structure to the succeeding structure as well as the preceding structure. The two pointers in each *portinfo* structure which link structures together are **next* and **prev*. **Next* points to the next *portinfo* structure in the list and **prev* points to the previous *portinfo* structure in the list. **Prev* is a null pointer for the first

portinfo structure in the list. Likewise, **next* is a null pointer for the last *portinfo* structure in the list. (The entity interface data structure is shown in Figure 15).

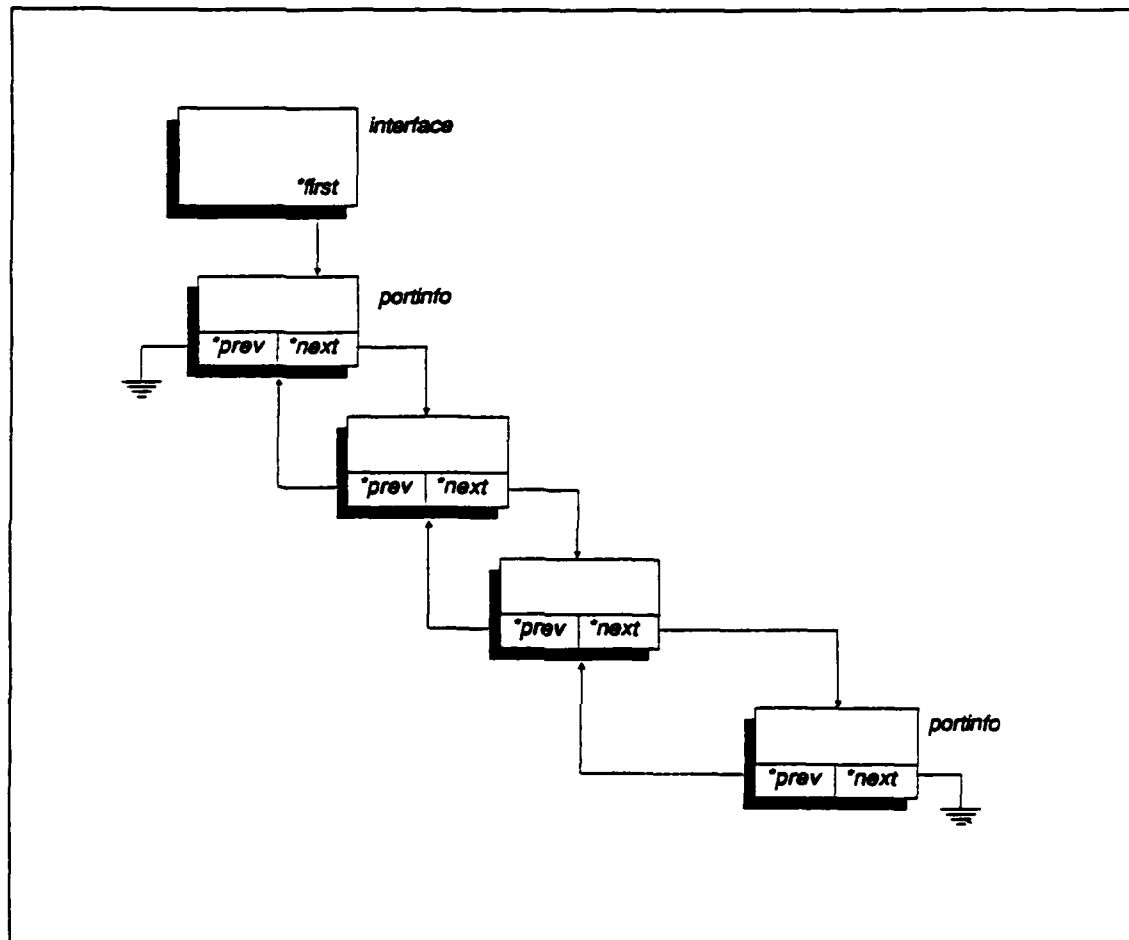


Figure 15. Entity Interface Data Structure

4.4. Architectural Body Editor (ABE)

The GVUI ABE is analogous to a variant of a schematic editor. The ABE permits the user to instantiate components for entities whose interfaces have already been defined using the Entity Interface Editor. The prototype GVUI is presently configured to permit the user to instantiate a maximum of 250 components in his or her architectural description.

However, the GVUI may be configured at setup time to be capable of instantiating more than the 250 component limit by changing the CMAX value in the *port.h* header file.

The ABE incorporates each component instantiation, which is constructed from a *.int* file, into the current ABE data structure which in turn can be transformed into a *.ab* file. A *.ab* file may be saved for later review or editing. It contains all the information that is required to reconstruct the entire graphical architectural description.

The sections which follow will describe the operation of and data structures for the ABE in further detail.

4.4.1. Control. (Throughout this section, the reader may reference Figures 16 and 17, Architectural Body Editor Flowchart). Upon entering the ABE, the user has the option of creating a new architectural description for an existing entity interface or editing an existing architectural description. If the user decides to create a new description the system would prompt the user to enter the file name assigned to the entity interface that he or she desires to describe.

If the interface exists the system displays each of the interface ports down the left side of the graphic viewport. In addition to the first character of each port label, the port symbol has different features to identify the port's mode. (See Table II). Though the prototype GVUI cannot as of yet move port symbols from their default world coordinate locations, functional provisions have been included to accomplish this task.

If the user decides to edit a previously entered architectural description, the system prompts the user to enter the file name assigned to the architectural body that he or she desires to edit. If the architectural body exists, the system searches the *.ab* file for component instantiations and connections. As instantiations and interconnections are located, the system dynamically allocates memory and reconstructs the architectural body data structure

Once the data structure's reconstruction is completed, the ports of the associated entity interface are drawn, followed by the display of component instantiations and interconnections which fall within the location of the graphic viewport. At this point, no matter whether the architectural description was loaded from a *.ab* file or is to be created interactively, control

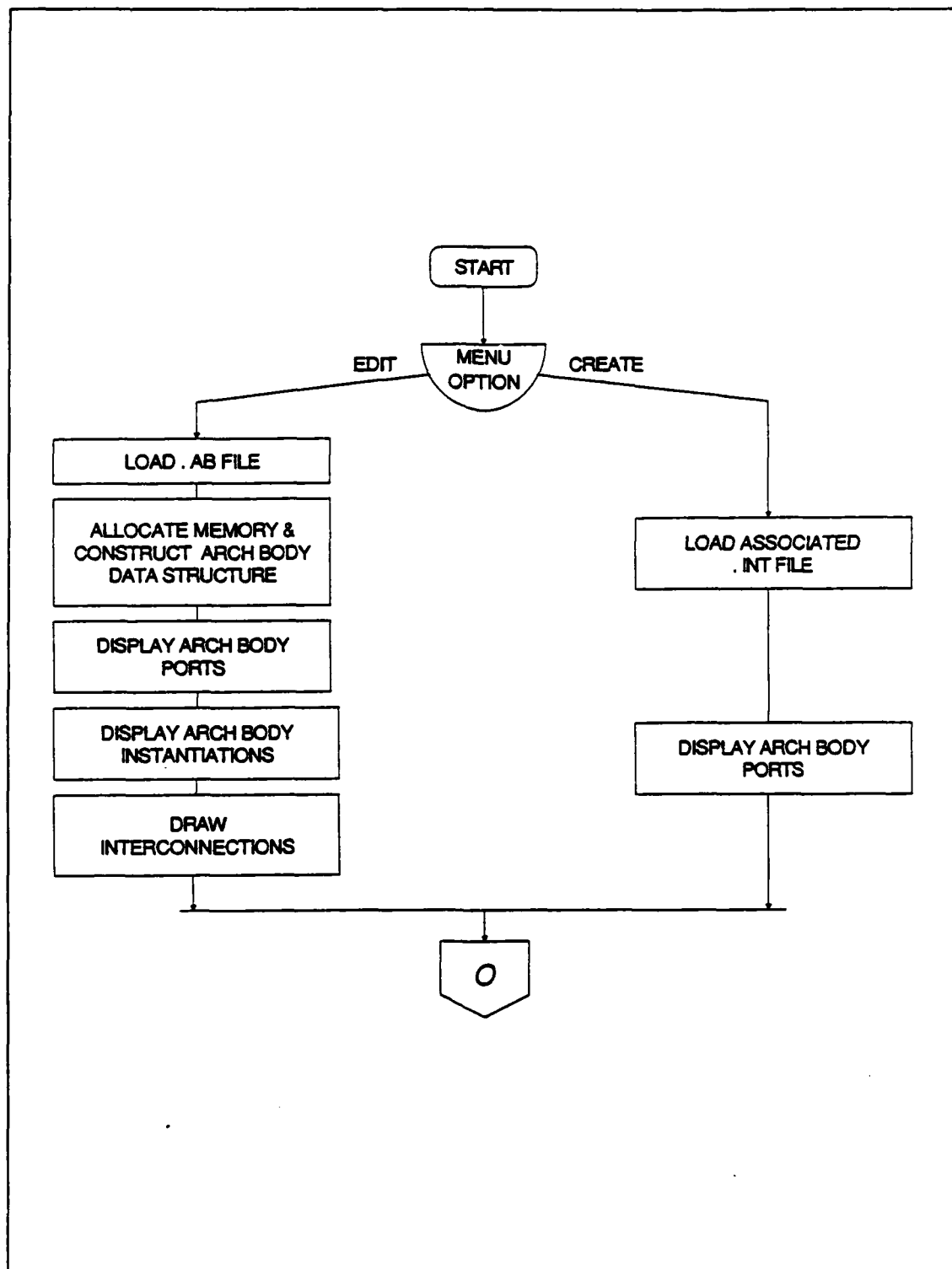


Figure 16. Architectural Body Editor Flowchart

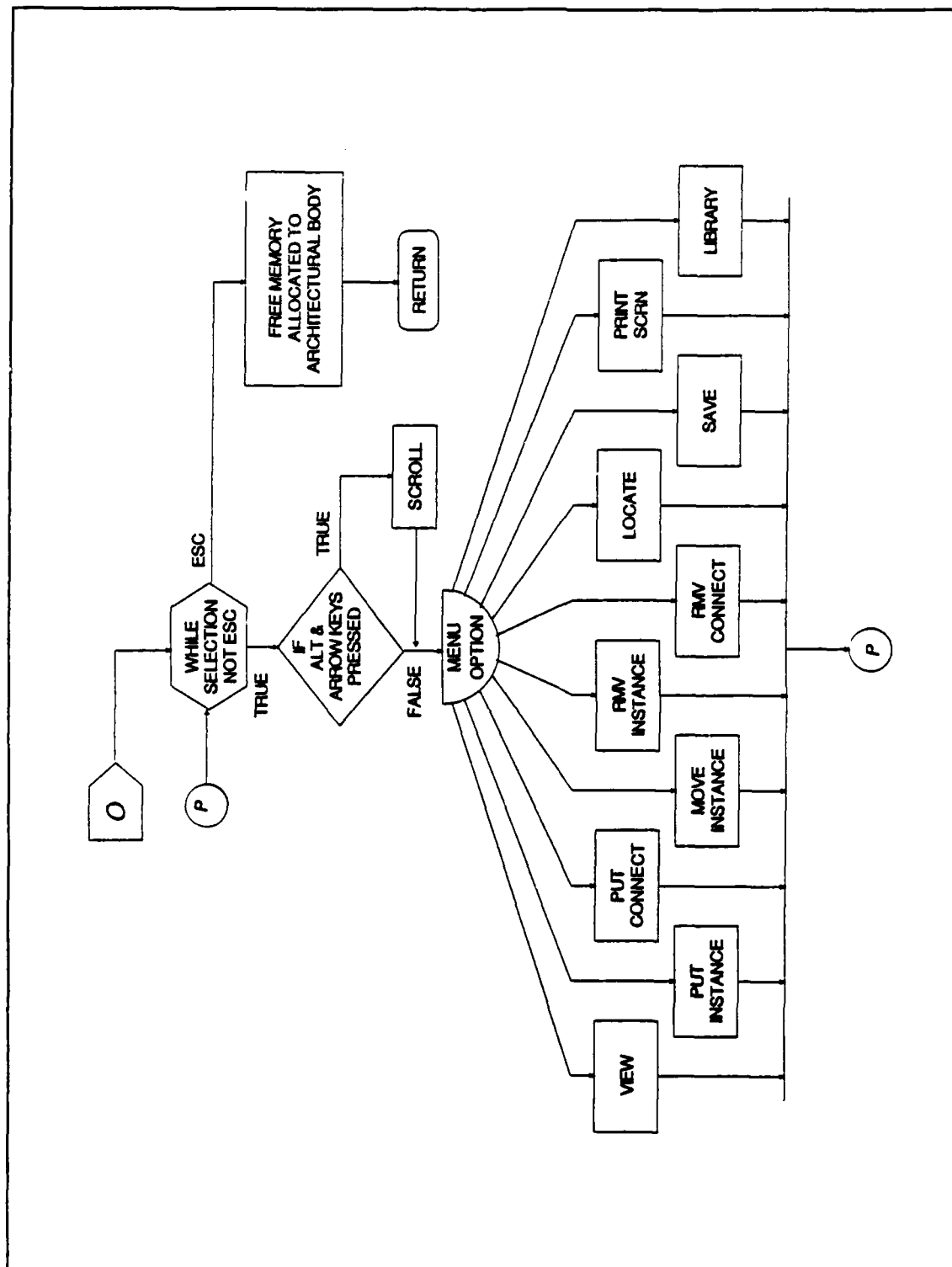
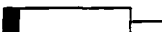
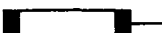


Figure 17. Architectural Body Editor Flowchart

Table II. Entity Interface Symbols

Port Mode	Port Symbol
IN	
OUT	
INOUT	
BUFFER	
LINKAGE	

returns to the same point; a menu of options for editing the architectural body appears.

VIEW prompts the user to enter a pair of viewport locator row and column coordinates. The system immediately moves the viewport to the specified area and displays that area on the screen. The advantage of VIEW over scrolling is its ability to immediately position and display the contents of the viewport area in one screen refresh cycle. Scrolling would require a screen refresh for each scroll increment of the viewport.

PUT INSTANCE permits the user to position and instantiate components within the current architectural body. In order to instantiate a component, its interface must already exist in the form of a *.ini* file. Once the system verifies its existence, a rectangle is drawn within the graphic viewport area at the present world coordinate cursor location.

The rectangle encompasses the area required by the component that is to be instantiated. Drawing a rectangle around the component's perimeter rather than drawing and labeling the component increases the speed with which the viewport can be refreshed while the system executes component moves and before the component is positioned.

Viewport scrolling is automatically enabled during component moves. If the user decides to place a component outside of the graphic viewport area he or she must scroll the viewport to the desired destination using one of the two scroll modes provided. Once the component perimeter is situated at the desired location, pressing RTRN initiates the instantiation sequence.

The instantiation sequence includes the following: first, the user is prompted to assign a name to the instantiated component; second, the component instance is added to the architectural body data structure by allocating and linking system memory for the new instance; and third, the component is drawn at the indicated location using vector drawing formulas in the *drawsymbols.c* source file

PUT CONNECT is the option which the user selects to route signal interconnections between component instances. A connection, the physical translation of a signal, is routed in a segmented fashion. The user would identify the connection's starting location, its intermediate "bend points" where the connection changes directions, and its terminal location.

To maximize the interactive response time of the system when routing connections, the ABE concerns itself only with the world coordinate locations of each of the connection segment endpoints. Any automated searching that is required to identify whether a segment endpoint is positioned over an actual port or intersects another connection should be performed within the Code Generator. By implementing these searches in the Code Generator they may be accomplished in parallel with other code generation search tasks.

The RMV CONNECT option permits the user to delete connections that have already been placed. To identify the connection that is to be removed the user positions the cursor over a junction that lies on the connection. The system then highlights the connection for verification as shown in Figure 18. In cases where the junction consists of three or more intersecting paths, the intersecting connections are highlighted one at a time under control of the user.

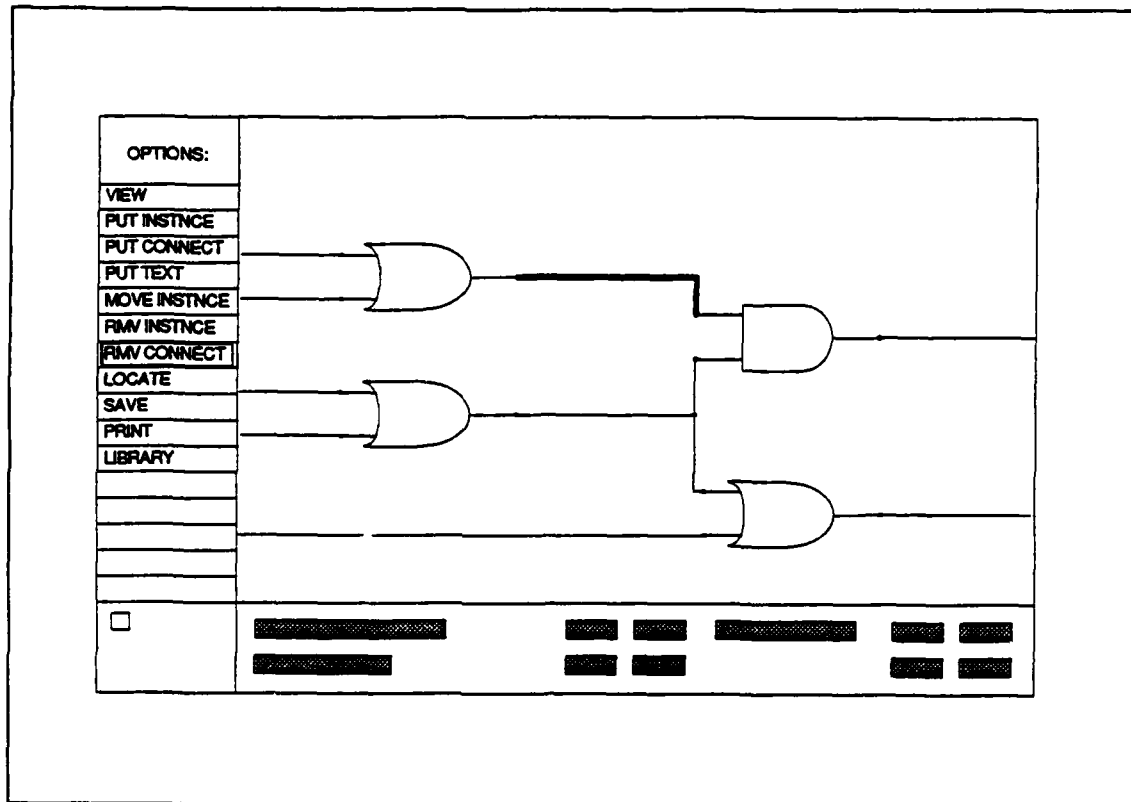


Figure 18. RMV CONNECT Verification

Once the system receives confirmation from the user that the connection has been correctly identified, it is removed from the connection data structure, memory that was allocated to the connection is freed, and the viewport is redrawn to reflect the removal.

The MOVE option, which is not yet integrated into the ABE, would provide the user a quick and efficient means of moving components and connections. An alternative approach to moving objects, whether they be component instances or connections, would be to delete the object with the appropriate RMV option and then reposition and reinstantiate the object using the appropriate PUT option.

Though MOVE could be implemented by simply automating the above RMV-PUT sequence, a more efficient and elegant implementation would be to update the X and Y hookpoints of the object with each redraw of the object. The X and Y hook points of an

object locate the object's position in the world coordinate system. The hookpoint coordinates coincide with the cursor crosshair location which is visible during object perimeter redraws. By updating the object hookpoint coordinates with each move of the cursor, a MOVE could be performed without the memory deallocation and reallocation which occurs using the RMV-PUT sequence.

RMV INST permits the user to delete the occurrence of an instantiated component in an architectural body. If the user selects this option the system prompts him or her to enter the name of the component to be deleted. If the component instance is found to exist and is visible within the current viewport, a box is drawn around the instance to highlight its location. The system then requests that the user reaffirm removal of the instance before it is removed from the data structure. Once removed, the component is erased from the viewport and memory that was allocated to the instance is freed for reuse.

If the instance is found to exist but lies outside the current viewport, the system notifies the user and requests that he or she reaffirm the request. Once reaffirmed, the system *erases the component instance from the world coordinate system and frees the memory that was previously allocated to it.*

The LOCATE option permits the user to identify the relative location of a component in the current architectural body. This feature alleviates the need of the user to scroll across the world coordinate system in search of a component that he or she believes may have been instantiated. The LOCATE option performs two tasks. First, it searches the data structure to determine if the component does in fact exist and notifies the user of the search results. Second, if the component search is successful the system positions and highlights the component locator to reflect the relative position of the requested component.

The SAVE option allows the user to save the architectural body as a .ab file. The system prompts the user to assign a name to the architectural body. If the file does not already exist, the file is saved with a .ab extension. If a file already exists with the same name, the system asks the user if he or she desires to overwrite the existing file.

PRINT SCRIN provides the user the option of outputting the current display to a printer. It is anticipated under the phase II development effort that this option would provide the user with the capability to print either the current display or the entire architectural body.

The LIBRARY selection is identical to that listed in the GVUI main menu except, upon return from the directory listing of the GVUI library, the previous graphic display of the current architectural body is restored and control is returned to the ABE.

After execution of any of the ABE menu options is completed the user may select another menu option or may exit to the GVUI main menu by pressing ESC. If the user presses ESC and confirms the validity of his or her intention, the architectural body data structure is deleted, all memory that was allocated to the data structure is freed, and the viewport is cleared.

4.4.2. Data Structure. The architectural body data structure is really an integration of three more narrowly defined data structures which are linked together under a C structure, *a_body*, of type *arch_body*. (See Figure 19.) An *arch_body* structure is defined in the header file *port.h* as follows:

```

struct arch_body
{
    char    ab_name[13];
    char    ent_name[13];
    int     io_num;
    struct  instantiation    *instance[CMAX];
    struct  ab_io            *first_io;
    struct  connection      *first_con;
};

```

Ab_name[13] is an array of characters which contains the name of the description. The filename under which a description is saved is created by concatenating a *.ab* extension to *ab_name*. *Ent_name[13]* is an array of characters which contains the name of the entity for which the given architectural body is associated. *Io_num* is the number of interface ports associated with the entity being described.

The remaining elements of *arch_body* are pointers which provide links to the three data substructures which together contain all the information required to reconstruct an

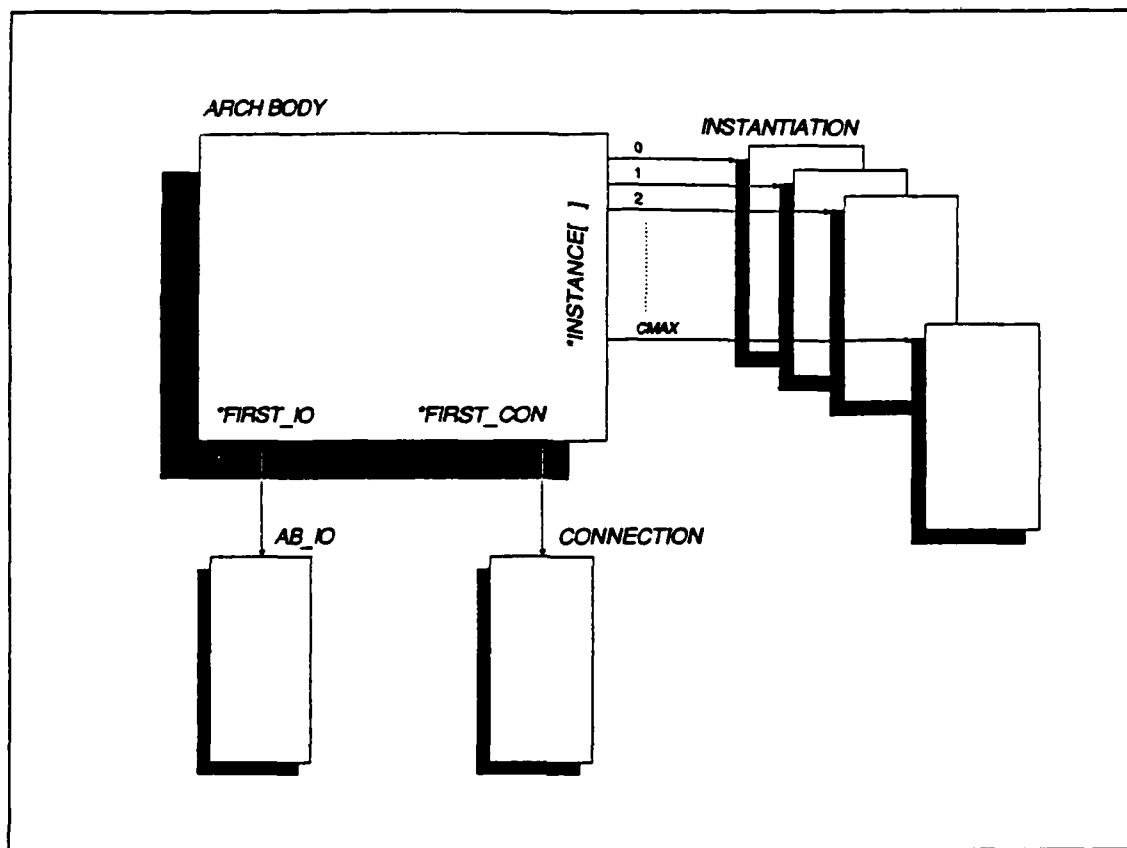


Figure 19. Architectural Body Data Structure

architectural description of an entity. **Instance[C MAX]* is an array of C MAX pointers to component *instantiation* structures. These pointers are initially set to null. They are assigned to point to *instantiation* structures as each component is instantiated. C MAX is the upper limit to the number of component instantiations that may occur in a single description. **First_io* is a pointer to the first port in the architectural body ports data structure. Similarly, **first_con* is a pointer to the first connection in the connection data structure.¹

4.4.2.1. *Component Instantiations.* An *arch_body* structure may point to as many *instantiation* structures as are bounded by C MAX. An *instantiation* structure is very similar

¹ C MAX = 250 for the phase I prototype.

to an *interface* structure (discussed in section 4.3.3.) *Instantiation* structures are defined in the *port.h* header file as follows:

```

struct instantiation
{
    int      symtype;
    int      symports;
    char     ilabel[9];
    int      row;
    int      col;
    int      xhook;
    int      yhook;
    int      xlabel;
    int      ylabel;
    int      orient;
    struct   portinfo *pfirst;
};

```

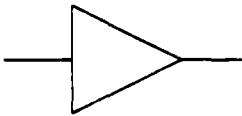
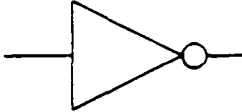
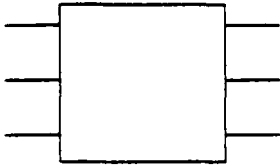
Symtype is an integer value from one to nine inclusive. It specifies the graphic symbol representation for the component being instantiated. Table III associates a symbol type, *symtype*, to each permissible integer value.

Symports is also an integer value. It specifies the number of ports in the associated entity interface and the number of *ab_io* structures present in the architectural body ports data structure. *Ilabel[9]* is an array of characters which contains the component instantiation label. This label is assigned to the component by the user at the time the component is instantiated. Its X and Y world coordinate values are specified but the integers *xlabel* and *ylabel*. *Symtype*, *symports*, *xlabel*, and *ylabel* are derived from their counterparts, *symbol*, *num*, *xname*, and *yname*, respectively, in the *interface* structure which is associated with the component being instantiated.

Row and *col* are provided as inputs to a filtered clipping algorithm. The filtering uses the *row* and *col* coordinates to quickly determine if the component lies entirely outside of the current graphic viewport, in which case the component would not be considered for clipping.

Xhook and *yhook* are world coordinate integer values which represent the handle point, or hook point, for the component. The component's hook point is used by the drawing routines as the origin of reference from which symbol segment lengths are calculated.

Table III. Component Symbols

<i>SYMTYPE</i> (INTEGER VALUE)	MENU OPTION	<i>SYMBOL</i> (GRAPHIC)
1	BUFFER	
2	INVERTER	
3	AND	
4	NAND	
5	OR	
6	NOR	
7	XOR	
8	NXOR	
9	MODULE	

Orient is an integer value that has been included for future enhancement. It is anticipated that the GVUI shall be upgraded to permit the user to rotate components within an architectural body. Once rotational capabilities are added *orient* shall identify the graphic orientation of each instantiated component. For instance, a component shall have one of the following orientations: left-to-right (default), right-to-left, top-to-bottom, or bottom-to-top. Since this capability is not yet implemented, *orient* is set to zero.

**Pfirst* is a pointer to the first *portinfo* structure in a list of structures that contains information about each actual port for a given component. Recall (from section 4.3.3.) that each entity interface data structure consists of an *interface* structure and a linked list of *portinfo* structures. Each *portinfo* structure in the list contains information about a formal port for the current entity. Just as each entity interface *portinfo* structure linked list contains information regarding the entity's formal ports, each component instantiation requires a similar list to represent a component's actual ports. However, instead of relative world coordinate values, absolute world coordinate values are maintained in the list of actual ports. **Pfirst* points to the head of the component instantiation *portinfo* structure list. A single component *instantiation* structure is shown in Figure 20.

4.4.2.2. *Architectural Body Ports*. Ports of the entity being described in the ABE shall herein be referred to as architectural body ports. The data structure that is associated with the architectural body ports is similar to the entity interface *portinfo* structure. Each port of an entity to be described has associated with it an *ab_io* structure. Each *ab_io* structure is defined in the *port.h* header file as follows:

```

struct ab_io
{
    int      id;
    char     pname[8];
    int      row;
    int      col;
    int      xhook;
    int      yhook;
    int      xconnect;
    int      yconnect;
    struct   ab_io *next;
};

```

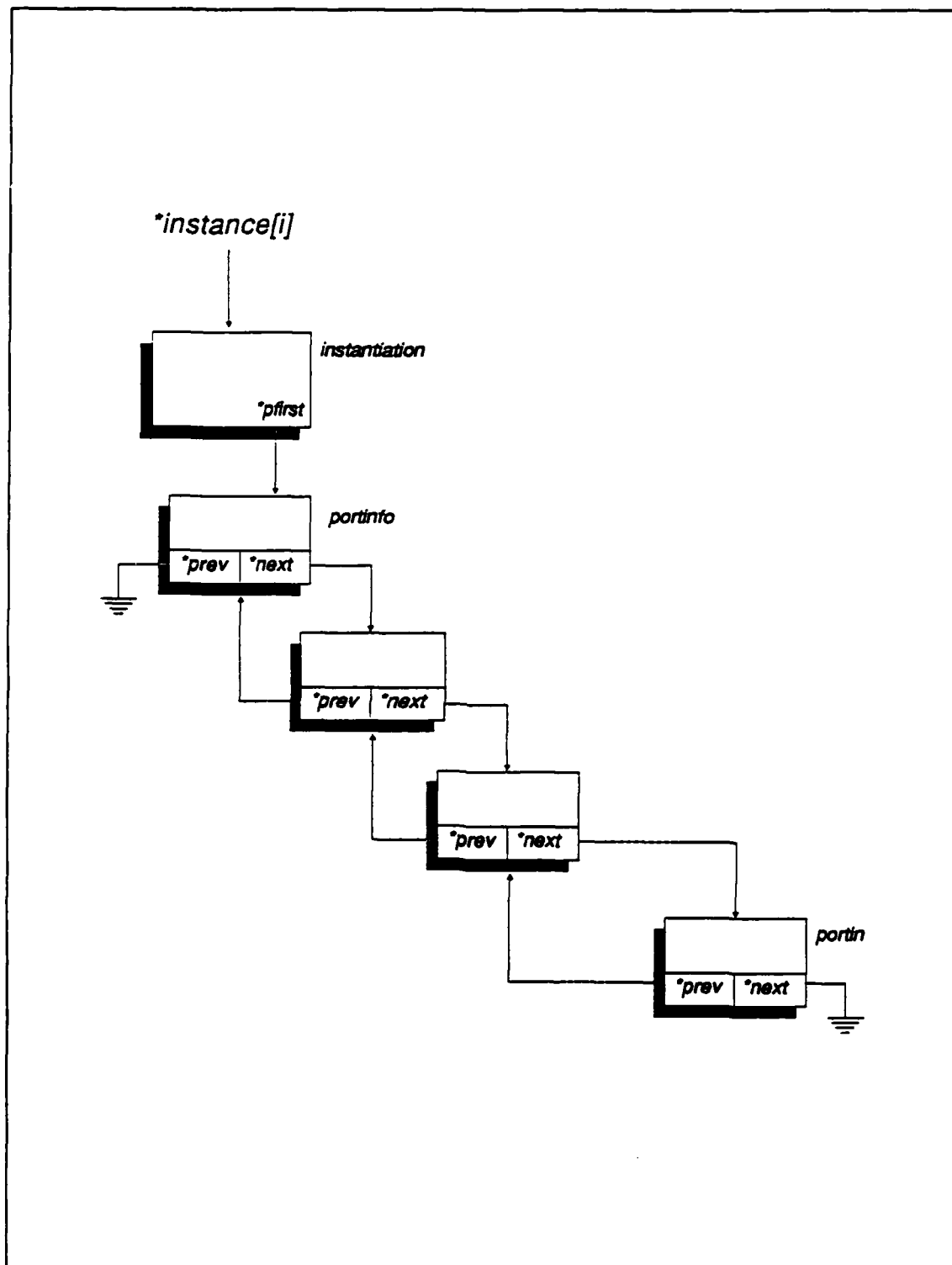


Figure 20. Instantiation Data Structure

Upon entering the ABE, the system creates a linked list of *ab_io* structures. One structure is allocated for each interface port. **Pfirst* in *a_body*, the parent *arch_body* structure, points to the *ab_io* structure that is associated with the first entity port. **Next*, a pointer to a structure of type *ab_io*, points to the next structure in the linked list (as shown in Figure 21). In the structure associated with the last port of the entity, **next* is set to null.

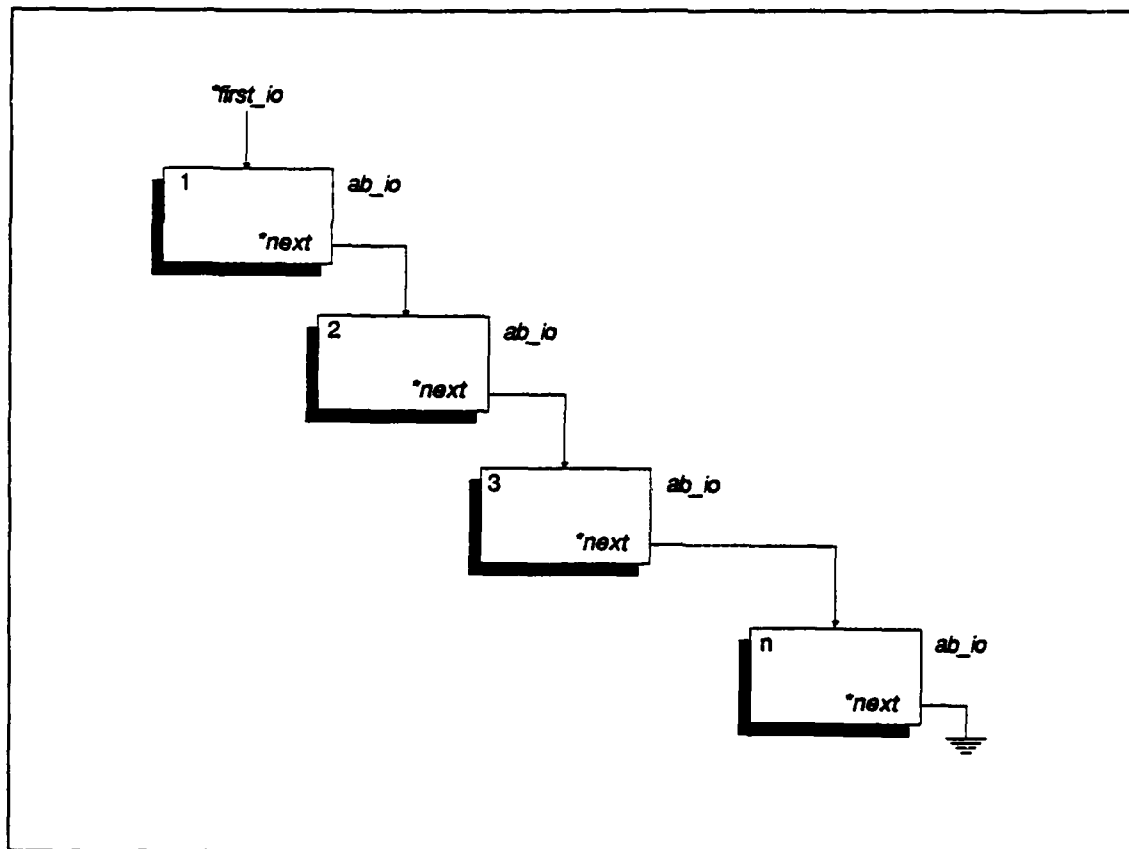


Figure 21. Architectural Body Ports Data Structure

Id, the first element within each *ab_io* structure, is an integer that identifies each port in the architectural body port structure. For instance, if the entity being described has eight ports, *id* for the first port would have the integer value one; *id* for the second port would have the value two; and so on. The last port would have an *id* value of eight.

Pname[8] is an array of characters which contains the formal port name. *Row* and *col* identify the port's position in viewport coordinates. These values serve the same purpose as they do in the *instantiation* structure; they are inputs to the clipping algorithm. *Xhook* and *yhook* are world coordinate integer values which represent the graphic port hook point. *Xconnct* and *yconnct* are world coordinate integer values which identify the permissible junction point to which a connection may be made.

4.4.2.3. *Connections*. The connection data structure is designed to accommodate an unknown number of connections in the description of an architectural body. As component interconnections are laid by the user, the system dynamically allocates and appends memory to the connection data structure.

The connection data structure is a doubly-linked list of *connection* structures. (See Figure 22). Each *connection* structure is defined in the *port.h* header file as follows:

```
struct connection
{
    struct connection    *cprev;
    struct connection    *cnext;
    struct point         *first;
};
```

The first *connection* structure in the list is pointed to by **first_con* in *a_body*, a structure of type *arch_body*. **Cnext* and **cprev* are pointers to other *connection* structures in the connection data structure. **Cprev* is set to null for the first structure in the connection list. Likewise, **cnext* is set to null for the last structure in the list.

An individual connection consists of a series of adjoining line segments. Each line segment has two endpoints. The first endpoint of the first segment of a connection is called the initial endpoint of the connection. The last endpoint of the last segment of a connection is called the terminal endpoint of the connection. The endpoints of each segment which comprise a connection are given by *point* structures.

Point structures are defined in the *port.h* header file as follows:

```
struct point
{
    int      wx;
    int      wy;
    int      jct;
    struct    point *next;
    struct    point *prev;
};
```

Each *point* structure consists of three integer values: *wx*, *wy*, and *jct*. *Wx* and *wy* identify the respective X and Y world coordinates of the particular segment endpoint. *Jct* is a boolean value of type integer that designates whether or not the given point is an initial or terminal connection endpoint. *Jct* is also used by the drawing routines to determine if a junction symbol is to be placed at the specified location. **Next* and **prev*, pointers to other structures, link the *point* structures of a connection together to form a doubly-linked list of *point* structures, much like **cnext* and **cprev* form a doubly-linked list of connection structures.

As shown in Figure 22, the initial endpoint of a connection corresponds to the first *point* structure in the list of *point* structures and it is pointed to by **first*. **First* is a pointer to a *point* structure and is an element of each connection structure.

5. TESTING

"The purpose of testing is to make quality visible. (15:8)"

5.1. Dimensions

Three dimensions exist in the software quality space. They are functionality, engineering, and adaptability. (15:8) The development of this thesis effort sought to maximize quality by addressing each of these dimensions. The GVUI's exterior qualities, its functionality, are reflected by its reliability and usability. The GVUI's interior qualities, its engineering, are emphasized in its structure and efficiency. The systems future qualities, its adaptability, are pronounced through its flexibility and maintainability.

5.2. Levels

Before a product is transitioned to its user, three levels of testing typically occur. They are module, integration, and system testing. Module testing, also called "testing in the small", independently tests logical units of code. Module testing serves a two-fold purpose. First, it insures that the logic of the module under test functions properly. Second, it insures that all the logic of the module under test is in fact present.

Integration testing, also called "testing in the large," is a progressive series of tests in which software modules which are designed to interact with one another are combined and tested. Integration testing assumes that each module to be integrated has successfully passed module testing. Integration testing occurs until the entire system has been integrated.

System testing starts when integration testing has been completed (15:115). System testing insures that all functions specified in the system specification have been incorporated. It also demonstrates that the system's performance and quality will meet the operational requirements of the system (15:118).

Each level of testing may be either functional or structural in nature. Functional testing is formulated from an external knowledge of what the system is supposed to do. This is why functional testing is often referred to as black box testing. Structural testing, on the other hand, is performed independent of a program's functionality and is derived from the internal logic and structure of the program. This testing approach, which is opposite that of black box testing, is often referred to as white box testing. Pressman states that white box testing is necessary and far more likely to uncover the "bugs that lurk in corners and congregate at boundaries. (19:472)"

5.3. Approach

Recall that the zigzag development approach required that the GVUI Environment Manager be operational before any functions in the menu hierarchy could be implemented. Once the Environment Manager was complete functions were then developed one at a time. One function would be coded, tested in the small, integrated into the environment, and undergo integration testing before coding would commence on any other function. Thus the order of integration was accomplished in a zigzag fashion, function by function. Focusing on each individual function, however, would reveal that a top-down testing technique was employed for each of the functions submodules.

5.4. Functional System Tests

In an effort to systematically provide the reader some insight to specific test cases which were used to qualify the GVUI, the following sections list the functional system tests which were applied to each menu option within the menu hierarchy.

NOTE: Unless otherwise noted, all messages and prompts should be displayed in the feedback area.

5.4.1. *Main Menu.* The Main Menu provides the user with the following options: CREATE, EDIT, LIBRARY, VHDL CODE, EXIT and ESC. A brief description of the expected test results based upon the given test cases for these options is as follows:

NOTE: From hereafter, all references made to a positive response may be interpreted as the user pressing Y for "yes." Likewise, all references made to a negative response may be interpreted as the user pressing N for "no."

- | | |
|-------------|--|
| CREATE - | A1. Selection should result in replacement of the Main Menu by the Editor Selection Menu. |
| EDIT - | B1. Selection should result in replacement of the Main Menu by the Editor Selection Menu. |
| LIBRARY - | C1. Selection should result in replacement of the Main Menu by a directory listing of the GVUI Library.

C2. The library listing should accurately display up to and including the first 115 files in the directory.

C3.1. If less than 115 files exist in the library, pressing RTRN should result in the replacement of the GVUI Library by the Main Menu.

C3.2. If greater than 115 files exist in the library, pressing RTRN should result in the next 115 files in the directory to be displayed. This process should be repeated until all files in the library have been displayed.

C3.4. Once all files have been displayed, pressing RTRN should result in the replacement of the GVUI Library by the Main Menu. |
| VHDL CODE - | D1. Selection should be ignored by the system. (VHDL code generation is to be implemented in the phase II development effort.) |
| EXIT GVUI - | E1. Selection should result in a message that prompts the user to confirm his or her intentions of exiting the GVUI. |

E2.1. If the user provides a positive response by pressing Y for "yes," the system exits to the operating system.

E2.2. If the user provides a negative response, the prompt should be erased and the user should be able to select another Main Menu option.

ESC - F1. An attempt to escape from this menu level to a (non-existent) higher-level menu should be ignored.

5.4.1.1. Editor Selection Menu. The Editor Selection Menu, which is obtained by selecting the CREATE or EDIT option in the Main Menu, provides the user with the following three options: INTERFACE, ARCH BODY, and ESC. A brief description of the expected test results based upon the given test cases for these options is as follows:

INTERFACE - A1.1. If the user arrives at this selection through the CREATE option in the Main Menu, the Editor Selection Menu should be replaced by the Entity Interface Editor Menu.

A1.2. If the user arrives at this selection through the EDIT option in the Main Menu, a message should prompt the user to enter the name of the *.int* file to be edited.

A1.2.1. If the filename does not exist in the GVUI Library, the system should generate a warning tone and a message that notifies the user of the file's nonexistence. The system should then return the user to the Main Menu.

A1.2.2. If the filename exists as a *.int* file, the Editor Selection Menu should be replaced by the Entity Interface Editor Menu.

ARCH BODY - B1. If the user arrives at this selection through the CREATE option in the Main Menu, a message should prompt the user to enter the name of the *.int* file for which an architectural body is to be created.

B1.2. If the filename does not exist in the GVUI Library, the system should generate a warning tone and should notify the user of the file's nonexistence. The system should then return the user to the Main Menu.

B1.3. If the filename exists as a *.int* file, the Editor Selection Menu should be replaced by the ABE Menu. Also, the architectural body parts of the associated entity should be drawn in the viewport area.

B2. If the user arrives at this selection through the EDIT option in the Main Menu, a message should prompt the user to enter the name of the *.ab* file that is to be edited.

B2.1. If the filename does not exist in the GVUI Library, the system should generate a warning tone and should notify the user of the file's nonexistence. The system should then return the user to the Main Menu.

B2.2. If the filename does exist as a *.ab* file, the Editor Selection Menu should be replaced by the ABE Menu. Also, a portion of the architectural body to be edited should be visible in the viewport area.

ESC - C1. An attempt to escape from this menu level to a (non-existent) higher-level menu should be ignored.

5.4.2. *Entity Interface Editor Menu.* The Entity Interface Editor Menu, which is obtained by selecting the INTERFACE option in the Editor Selection Menu, provides the user with the following options: PUT SYMBOL, SAVE, PRINT SCRIN, and ESC. A description of the expected test results based upon the given test cases for these options is as follows:

SYMBOL - A1. Selection should result in the replacement of the Entity Interface Menu by the Entity Symbol Menu.

SAVE - B1. Selection should result in a message that prompts the user to enter a filename under which the entity interface would be saved.

B2.1. If the filename entered by the user already exists, the system should generate a warning tone and should ask the user if the existing file should be overwritten.

B2.1.1. If the user provides a negative response, the system should display the message, "No SAVE occurred," and should permit the user to select another option from the Entity Interface Editor Menu.

B2.1.2. If the user provides a positive response, the system should display the following confirmation request: "Proceed?"

B2.1.2.1. If the user provides a positive response to the confirmation request, the system should save the entity interface under the user-entered filename with a *.int* extension. The file should be saved to the directory specified by OUTDIR in the header file *sysconfig.h*.

B2.1.2.2. If the user provides a negative response to the confirmation request, the system should display the message, "No SAVE occurred," and should permit the user to select another option from the Entity Interface Editor Menu.

B2.2. If the filename entered by the user does not already exist, the system should display the following confirmation request: "Proceed?"

B2.2.1. If the user provides a positive response to the confirmation request, the system should save the entity interface under the user-entered filename with a *.int* extension. The file should be saved to the directory specified by OUTDIR in the header file *sysconfig.h*.

B2.2.2. If the user provides a negative response to the confirmation request, the system should display the message, "No SAVE occurred," and should permit the user to select another option from the Entity Interface Editor Menu.

PRINT SCRN - C1. Selection should be ignored by the system. (PRINT SCRN has not been coded under the phase I development effort.)

ESC - D1. Pressing ESC while in the Entity Interface Editor Menu should cause the system to warn the user that the current entity interface will be lost if it is not saved before leaving the current menu. A message should also request confirmation from the user to proceed.

D1.1. If the user provides a positive response, the entity interface should be erased from the viewport and the user should be returned to the Main Menu.

D1.2. If the user provides a negative response, the feedback area should be cleared and the user should be able to select another option from the Entity Interface Menu.

5.4.2.1. *Entity Symbol Menu.* The Entity Symbol Menu, which is obtained by selecting the SYMBOL option in the Entity Interface Editor Menu, provides the user with the following options: BUFFER, INVERTER, AND, NAND, OR, NOR, XOR, NXOR, MODULE, and ESC. A brief description of the expected test results based upon the given test cases for these options is as follows:

BUFFER,
INVERTER - A1. Selecting either of these options should result in the viewport being cleared, the selected symbol being displayed in the viewport, and a message that prompts the user to assign an entity name to the entity being edited.

A2.1. If the user enters an entity name that is not already identified by a .int file in the GVUI Library, the entity name and formal port names should appear in their appropriate locations about the selected entity symbol. Also, a message should appear and ask if the formal port names/modes are acceptable.

A2.2.1. If the user provides a positive response, the entity image should remain in the viewport, but the user should be returned to the Entity Interface Editor Menu.

A2.2.2. If the user provides a negative response, a message should prompt the user to enter the name of the formal port to be edit.

A2.2.2.1. If the user enters a valid formal port name to edit, the user should then be prompted to enter a new port name/mode.

A2.2.2.1.1. If the newly entered port name/mode already exists, the system should generate a warning tone and prompt the user to reenter the new formal port name/mode.

A2.2.2.1.2. If the newly entered port name/mode does not already exist, it should be displayed in the viewport at the proper entity port location. The user should then be asked if the formal port names/modes are acceptable. (Continue following step 2.1.)

A2.2.2.2. If the user enters the name of a formal port to edit but the port does not exist for the displayed entity, the system should generate a warning tone. Also, a message should warn the user that an invalid entry has been made and the name of the port to be edited should be reentered. (Continue following step A2.2.2.)

AND,
NAND,
OR,
NOR,
XOR,
NXOR -

B1. Selection of any of these options should result in the current viewport being cleared and a message that prompts the user to enter the number of inputs to the entity.

B2.1. If the user enters a number other than those allowed (2, 3, 4, 8, 16), the system should ignore the input.

B2.2. If the user enters an allowable number (2, 3, 4, 8, 16), the selected symbol should be drawn in the viewport and the user should be prompted to assign a name to the entity.

B2.3. After a name is assigned to the entity, the symbol should be labeled with the entity and port names and a new message should appear. It should ask the user if the formal port modes/labels are acceptable.

B2.3.1. If the user provides a positive response, the entity image should remain in the viewport and the user should be returned to the Entity Interface Editor Menu.

B2.3.2. If the user provides a negative response, a message should prompt the user to enter the name of the formal port that is to be edited.

B2.3.2.1. If the user enters a valid port name to edit, the user should be prompted to enter a new port mode/label.

B2.3.2.1.1. If the newly entered port mode/label already exists, the system should produce a warning tone and prompt the user to reenter the new formal port mode/label.

B2.3.2.1.2. If the newly entered port mode/label does not already exist, it should be displayed in the viewport at the proper entity port location. The user should then be asked via a message if the formal port modes/labels are acceptable. (Continue following step B2.3.)

B2.3.2.2. If the user enters the name of a port to edit but the port does not exist, the system should produce a warning tone and warn the user that an invalid entry has occurred and the name of the port to be edited should be reentered. (Continue following step B2.3.2.)

MODULE -

C1. Selection should result in the current viewport being cleared and the Module Size Menu being displayed.

C2. Selection of any of the module size options should result in the module perimeter being drawn in the viewport and a message that prompts the user to assign a name to the entity being edited.

C3. Once a name is assigned to the entity, the module symbol should be labeled with its entity and port names and the Module Size Menu should be replaced by the following options: EDIT PORT, ADD PORTS, and RMV PORT.

C3.1. If the user presses ESC, he or she should be returned to the Entity Interface Menu.

C3.2. If the user selects the EDIT PORT option, a message should prompt the user to enter the name of the entity port to be edited.

C3.2.1. If the user enters a valid port name to edit, the user should be prompted to enter a new port mode/label.

C3.2.1.1. If the newly entered port mode/label already exists, the system should produce a warning tone and prompt the user to reenter the new formal port mode/label.

C3.2.1.2. If the newly entered port mode/label does not already exist, it should be displayed in the viewport at the proper entity port location. The user should then be able to make another selection from the list of module port editing options. (Continue following step C3.)

C3.2.2. If the user enters the name of a formal port to edit but the port does not exist, the system should produce a warning tone and should warn the user that he or she has made an invalid entry and should reenter the name of the port to be edited. (Continue following step C3.2.)

C3.3. If the user selects ADD PORTS, the entity interface should be redrawn with four new uniquely identifiable ports. The user should be able to make another selection from the list of module port edit options. (Continue following step C3.)

C3.4. If the user selects RMV PORT, a message should ask the user for confirmation to proceed.

C3.4.1. If the user provides a positive response, a message should prompt the user to enter the name of the port to be removed.

C3.4.1.1. If the user enters a valid port name to remove, the port should be erased from the entity displayed in the viewport. The user should be able

to make another selection from the list of module port edit options.
(Continue following step C3.)

C3.4.1.2. If the user enters the name of a formal port to remove but the port does not exist, the system should produce a warning tone and a message that warns the user that he or she has made an invalid entry and should reenter the name of the port to be removed. (Continue following step C3.4.1.)

5.4.3. *Architectural Body Editor Menu.* The Architectural Body Editor (ABE) Menu, which is obtained by selecting the ARCH BODY option in the Editor Selection Menu, provides the user with the following options: VIEW, PUT INSTNCE, PUT CONNECT, MOVE INSTNCE, RMV INSTNCE, RMV CONNECT, LOCATE, SAVE, PRINT, LIBRARY, and ESC. A description of the expected test results based upon the given test cases for these options is as follows:

- VIEW -
- A1. Selection should result in a message prompting the user to enter the column number of the viewport locator for the area desired to be viewed.
 - A2. The user should be prompted to enter the row number of the viewport locator for the area desired to be viewed.
 - A3. The viewport should display the area identified by the two previous entries.
 - A4. The viewport locator in the VLA should graphically reflect the change in the location of the viewport.
 - A5. The viewport locator coordinates and world coordinates should have been updated.
 - A6. The user should be able to select another option in the ABE Menu.
- PUT INSTNCE -
- B1. Selection should result in a message that prompts the user to enter the name of the entity to be instantiated.

B2.1. If the entity name entered does not exist as a *.int* file in the GVUI Library, the system should generate a warning tone and notify the user of the file's nonexistence. The user should be able to make another selection from the ABE Menu.

B2.2. If the entity name entered exists as a *.int* file, a rectangle representing the entity's perimeter should be displayed in the viewport. A message should prompt the user to press RTRN once the entity perimeter is positioned, or to press ESC to exit to the ABE Menu.

B2.2.1. Once the entity perimeter is positioned and the user presses RTRN, a message should prompt the user to enter an instantiation label for the newly positioned component.

B2.2.1.1. If the user assigns a label that is already assigned to another component, a warning tone should be generated and a message should appear in the feedback area. The message should notify the user of the attempted duplication and should prompt the user to reenter the label. (Continue following step B2.2.1.)

B2.2.1.2. If the user assigns a label that has not yet been assigned to any other component, a message should appear asking if the port labels are acceptable.

B2.2.1.2.1. If the user provides a positive response, the system should complete the instantiation process and the user should be able to make another selection from the ABE Menu.

B2.2.1.2.2. If the user provides a negative response, a message should prompt the user to enter the name of the actual port of the most recently positioned component that is to be edited.

B2.2.1.2.2.1. If the user enters a valid actual port name, a message should prompt the user to assign a new label to the port.

B2.2.1.2.2.1.1. If the new port label already exists on the component, the system should generate a warning tone and the user should be prompted

to reenter the new port label. (Continue following step B2.2.1.2.2.1.)

B2.2.1.2.2.1.2. If the new port label does not exist on the component, it should be displayed in the viewport at the proper component port location. The user should be asked if the actual port labels on the most recently positioned component are acceptable. (Continue following step B2.2.1.2.)

B2.2.1.2.2.2. If the user enters a port name that does not correspond to any of the ports of the most recently positioned component, the system should generate a warning tone. Also, a message should warn the user that an invalid entry was made and the name of the port to be edited should be reentered. (Continue following step B2.2.1.2.2.)

PUT CONNECT - C1. Selection should result in a message that prompts the user to identify the connection's initial point and to press SPACE once the point has been identified.

C2. Once the user identifies the initial point by pressing SPACE another message should appear. It should prompt the user to identify the segment endpoint and to press SPACE if it is an intermediate bend point or RTRN if the point terminates the connection.

C2.1. If the user identifies an intermediate bend point by pressing SPACE, a line segment should be drawn from the previous endpoint to the new segment endpoint. A message should prompt the user to identify the segment endpoint and to press SPACE if it is an intermediate bend point or RTRN if the point terminates the connection. (Continue from step C2.)

C2.2. If the user identifies a connection's terminal point by pressing RTRN, the system should complete the connection by drawing a line segment from the last segment endpoint to the connection's terminal point. A junction should also be drawn at both the connection's initial and terminal points.

C3. The user should be able to select another option from the ABE Menu.

MOVE INSTNCE - D1. Selection should be ignored by the system. (This function is not yet implemented under the phase I development effort.)

RMV INSTNCE - E1. Selection should result in a message that prompts the user to enter the name of the component instance to be removed.

E2.1. If the name entered does not correspond to any instantiated component in the current architectural body, a warning tone should be generated and a message should notify the user that the particular instance was not located.

E2.2. If the name entered corresponds to an instantiated component that lies outside of the current viewport, the system should generate a warning tone and a message. The message should notify the user of the location of the instance and should prompt the user for confirmation to proceed.

E2.2.1. If the user provides a negative response, no removal should occur.

E2.2.2. If the user provides a positive response, the system should notify the user that the instance has been removed.

E2.3. If the user enters a name that corresponds to an instance that is visible in the viewport, the system should draw a rectangle around the instance and request confirmation before removing the instance.

E2.3.1. If the user provides a negative response, the rectangle around the instance should be removed, but the instance should remain.

E2.3.2. If the user provides a positive response, the instance should be removed from the current architectural body.

E3. The user should be able to make another selection from the ABE Menu.

RMV INSTNCE - F1. Selection should result in the user being prompted to identify a junction on the connection that is to be removed and to press RTRN.

F2.1. If the user fails to position the cursor over a junction and presses RTRN, the system should generate a warning tone and should inform the user that it has failed to locate a connection endpoint at the cursor location.

F2.2. If the user positions the cursor over a junction and presses RTRN, the system should highlight a connection that intersects the junction. Also, "Correct?" should appear in the feedback area.

F2.2.1. If the user provides a positive response, the viewport should be redrawn, reflecting the removal of the connection.

F2.2.2. If the user provides a negative response and other connections (that have not yet been highlighted) intersect the identified junction, the system should redraw the viewport and highlight another connection. "Correct?" should appear in the feedback area. (Continue following step 2.2.)

F2.2.3. If the user provides a negative response and no other connections (that have not yet been highlighted) intersect the junction, the system should generate a warning tone and notify the user that it was unable to locate another connection that intersects the junction.

F2.3. If the user presses ESC, the connection removal process should be cancelled and the cursor should disappear from the viewport.

F3. The user should be able to select another option from the ABE Menu.

LOCATE -

G1. Selection should result in a message that prompts the user to enter the name of the instance to be located.

G2.1. If the user enters the name of a component that is not instantiated in the current architectural body, the system should generate a warning tone and should inform the user of its inability to locate the desired component

G2.2. If the user enters the name of a component that is instantiated in the current architectural body, the system should display the viewport locator

coordinates for the area within which the component is positioned. Also, the component locator should appear in the VLA.

G3. The user should be able to select another option from the ABE Menu.

SAVE -

H1. Selection should result in a message that prompts the user to enter a filename under which the architectural body is to be saved.

H2.1. If the filename entered by the user already exists, the system should generate a warning tone and should ask the user if the existing file should be overwritten.

H2.1.1. If the user provides a negative response, the system should display the message, "No SAVE occurred," and the user should be able to make another selection from the ABE Menu.

H2.1.2. If the user provides a positive response, the system should request confirmation to proceed.

H2.1.2.1. If the user provides a positive response to the confirmation request, the system should save the current architectural body under the user-entered filename with a *.ab* extension. The file should be saved to the directory specified by OUTDIR in the header file *sysconfig.h*.

H2.1.2.2. If the user provides a negative response to the confirmation request, the system should display the message, "No SAVE occurred," and should permit the user to select another option from the Entity Interface Editor Menu.

H2.2. If the filename entered by the user does not already exist, the system should display the following confirmation request: "Proceed?"

H2.2.1. If the user provides a positive response, the system should save the architectural body under the user-entered filename with a *.ab* extension.

The file should be saved to the directory specified by OUTDIR in the header file *sysconfig.h*.

H2.2.2. If the user provides a negative response, the system should display the message, "No SAVE occurred," and should permit the user to select another option from the ABE Menu.

PRINT -

I1. Selection should be ignored by the system. (PRINT has not been coded under the phase I development effort.)

LIBRARY -

J1. Selection should result in replacement of the ABE Menu by a directory listing of the GVUI Library.

J2. The library listing should accurately display up to and including the first 115 files in the directory.

J3.1. If less than 115 files exist in the library, pressing RTRN should result in the replacement of the GVUI Library by the ABE Menu.

J3.2. If greater than 115 files exist in the library, pressing RTRN should result in the next 115 files in the directory to be displayed. This process should be repeated until all files in the library have been displayed.

J4. Once all files have been displayed, pressing RTRN should result in the replacement of the GVUI Library by the ABE Menu.

ESC -

K1. Pressing ESC should display a message that warns the user that the current architectural body will be lost if it is not saved before leaving the current menu. The system should also request confirmation to proceed.

K1.1. If the user provides a positive response, the architectural body should be erased from the viewport, messages should provide the status of the component and connection deallocation processes, the viewport locator should move to its origin, and the user should be returned to the Main Menu.

K1.2. If the user provides a negative response, the user should be able to select another option from the ABE Menu.

6. CONCLUSION

6.1. Future Recommendations

In addition to automatic code generation, which shall be the primary focus of the phase II development effort, areas have been identified from each of the three dimensions of the software quality space to maximize the overall quality of the AFIT GVUI. If implemented, the GVUI would be at least as capable and powerful, and in some respects more capable, than either of the UNIX-based graphic editors reviewed for this thesis. Furthermore, it would compete head-on with the PC-based VHDL environments which are only now beginning to emerge in the commercial sector.

6.1.1. Video Adapter Compatibility. Though the Hercules Graphics Card was selected as the phase I baseline graphics adapter, the system's usability and flexibility would dramatically increase if the GVUI were executable under a wider selection of video graphics adapters. Fortunately, this upgrade was anticipated from the earliest stages of development. Addressable system coordinates therefore employ parameters that are not constants, but rather are defined at system initialization via a header file.

Borland's Turbo C Graphics Library would further simplify the upgrade. In addition to the Hercules Card, Borland provides drivers for the following graphics adapters:¹

- Color Graphics Adapter (CGA)
- Multicolor Graphics Adapter (MCGA)
- Enhanced Graphics Adapter (EGA)
- Video Graphics Array (VGA)
- AT&T 400-line Graphic Adapter
- 3270 PC Graphic Adapter
- IBM 8514 Graphics Adapter

¹ These drivers are available in Turbo C version 1.5

6.1.2. *Mouse Input.* A decision was made in the early stages of the GVUI development to minimize the hardware requirements of the system in order to broaden its user base. Partially based upon this decision and with a logical system evolution in mind, graphic entry and menu selection was designed to be accomplished using the arrow keys on the host platform keyboard. There is presently no capability to select menu options or position design components via a mouse input device. An in-house evaluation of the system's user interface revealed, however, that a mouse input device is preferred to a keyboard for entering graphic data and should be integrated into the system to further increase its flexibility and user productivity.

6.1.3. *Data File Compaction.* As the functional capabilities of the system were implemented in the zigzag fashion, the run-time data structures expanded to accommodate them. Likewise, the structures of the associated I/O data files were concurrently evolving. As a result, the design of the I/O data structures emphasized simplicity over efficiency. All elements within each I/O file structure were kept independent of one another. This made the I/O file structures more adaptable to the unfolding functional run-time requirements, but did not guarantee minimal sizes.

Now that the contents of all data file storage requirements are firm, it is recommended that the *.ab* and *.int* data file storage requirements be lessened.

Consider the *portinfo* structure which is defined in the *port.h* header file as follows:

```
struct portinfo
{
    int      id;
    char     pname[8];
    int      xcon;
    int      ycon;
    int      xlbl;
    int      ylbl;
    struct   portinfo *next;
    struct   portinfo *prev;
};
```

All five of the integer parameters in the *portinfo* structure shall always be less than 512 and therefore require only one byte of memory. However, MS-DOS allots two bytes of

memory to every integer. Therefore, to store these values on a permanent storage would consume twice the amount of memory space than is required. Five extra bytes of overhead is not significant in itself, but recall that a *portinfo* structure is assigned to every port of every instantiated component in an architectural body. Given 250 components with 16 ports per component, this would equate to an overhead of 20,000 bytes.

Manipulating these five integer values as short integers would make no difference in the MS-DOS environment since MS-DOS considers both short and normal integers as 16-bit values. If a simple compaction scheme could be implemented, however, to combine *xcon* and *ycon* into one integer and *xlbl* and *ylbl* into another integer, then the 20,000 byte overhead in the above example could be reduced by 80%. Applying similar schemes throughout all the I/O data structures could significantly decrease the amount of storage space required to store entity interfaces and architectural bodies.

6.1.4. Additional Feature. The implementation of three graphic features are required of the GVUI to increase its capabilities to those presently found in PC-based schematic editors and to those expected of commercial VHDL environments. These features include the ability to move instantiated components, print entire circuit descriptions, and route data buses.

6.1.4.1. MOVE Option. Moving components in the prototype system requires the user to first remove the component using the RMV INSTNCE option and then to reinstantiate the component at the new location using the PUT INSTNCE option. A more efficient method would permit the user to drag components from one position to another using a single MOVE INSTNCE command.

The implementation of MOVE INSTNCE should not demand a great deal of effort on behalf of the developer. The MOVE INSTNCE option is already included in the ABE Menu and the graphic procedures which it would require already exist and are used by the RMV INSTNCE and PUT INSTNCE options.

6.1.4.2. Bus Routing. A third limitation of the phase I system is its inability to route composite type signals, such as data buses and address buses. To incorporate this capability

would require the same graphic procedures as those required for the PUT CONNECT option. The only difference would be the width of the line that is drawn to represent a signal bus.

The PUT CONNECT option presently lays a 1 pixel wide line to represent a bit value signal. Minimal effort would be required to enhance the routines to draw either 1 or 3 pixel wide line, assuming a 3 pixel wide line would represent the composite signal. Considering the impact on the associated connection structure, an argument could be added to reflect line width, thereby reflecting signal type. Also, error-checking would naturally be desired to prevent such occurrences as an attempt to connect a multielement bus to a bit value port. Incorporating bus routing would not be difficult and the benefit would be great.

6.2. *GVUI Phase I Summary*

Hardware design engineers currently enter VHDL source code as text which they input through a keyboard. Consequently, the circuit designer must learn the intricacies of a new computer language to describe his or her design. If some of the pressures associated with learning and transistioning to VHDL could be relieved, perhaps the language would become more widely accepted and its productivity would increase. In support of this philosophy, the AFIT VLSI Group sought to develop the Graphical VHDL User Interface as an integral component of the AVE.

The GVUI was envisioned to be capable of generating VHDL source code from a schematic diagram. It would run under MS-DOS, but be compatible with the UNIX-based AVE. Its development would occur in two phases. Phase I would primarily emphasize the schematic editing capabilities of the system and phase II would primarily address the automatic code generation feature.

This thesis effort completed the first of the two development phases. In compliance with the original objectives of the system, the phase I GVUI is a graphics-oriented, menu-driven schematic editor that runs on an IBM PC-XT personal computer. The executable code, developed in C, requires less than 256K to operate. Up to 1 megabyte of memory beyond the code requirements is addressable to accommodate large architectural descriptions.

The system is capable of supporting both top-down and bottom-up design methodologies. The number of hierarchical decompositions for a given circuit description is limited only by the amount of permanent storage capability on the host platform.

The phase I GVUI serves as a capable foundation upon which the phase II development can be built.

Appendix A:
AFIT GVUI User's Guide

USER'S GUIDE

OPTIONS:	
CREATE	AFIT GRAPHICAL VHDL USER INTERFACE By: Steve Matechik
EDIT	
LIBRARY	
VHDL CODE	
EXIT	
☐	

Table of Contents

	Page
List of Figures	A-iv
List of Tables	A-iv
1. Introduction	A-1
1.1. Purpose	A-1
1.2. Development	A-1
1.3. Requirements	A-1
1.4. Installation	A-2
2. Getting Started	A-4
2.1. Functional Overview (Phase I)	A-4
2.2. Loading	A-4
2.3. Selecting Menu Options	A-5
2.4. Returning to a Previous Menu	A-6
3. Main Menu	A-8
3.1. CREATE	A-8
3.2. EDIT	A-8
3.3. LIBRARY	A-8
3.4. VHDL CODE	A-9
3.5. EXIT GVUI	A-9
4. Editor Selection Menu	A-10
4.1. INTERFACE	A-10

4.2. ARCH BODY	A-10
5. Entity Interface Editor Menu	A-12
5.1. SYMBOL	A-12
5.2. SAVE	A-12
5.3. PRINT SCRIN	A-12
6. Entity Interface Editor Submenus	A-13
6.1. Entity Symbol Menu	A-13
6.1.1. BUFFER, INVERTER	A-13
6.1.2. AND, NAND, OR, NOR, XOR, NXOR	A-15
6.1.3. MODULE	A-16
6.2. Module Size Menu & Module Port Options	A-17
6.2.1. EDIT PORT	A-18
6.2.2. ADD PORTS	A-18
6.2.3. RMV PORT	A-20
7. Architectural Body Editor Menu	A-21
7.1. VIEW	A-21
7.2. PUT INSTNCE	A-22
7.3. PUT CONNECT	A-23
7.4. MOVE INSTNCE	A-24
7.5. RMV INSTNCE	A-24
7.6. RMV CONNECT	A-25
7.7. LOCATE	A-26
7.8. SAVE	A-27
7.9. PRINT	A-27
7.10. LIBRARY	A-27

List of Figures

	Page
Figure 1. GVUI Directory Structure	A-2
Figure 2. Functional Overview	A-4
Figure 3. GVUI Menu Hierarchy	A-7
Figure 4. 6x2 MODULE Entity	A-17
Figure 5. Adding Ports to a MODULE Entity	A-19

List of Tables

	Page
Table I. Formal Port Symbols	A-11
Table II. Entity Symbols	A-13
Table III. Entity Formal Port Modes	A-15

GVUI USER'S GUIDE

1. Introduction

1.1. Purpose

The Graphical VHDL User Interface (GVUI) provides VHDL support environments with a PC-based graphical front-end. In its completed form, the GVUI shall be capable of generating structural VHDL source code from a schematic diagram entered by the user. To simplify the integration of the GVUI to the VHDL support environment, the generated source code shall be output as standard ASCII text files. The text files may then be uploaded to a VHDL analyzer and simulator for analysis and simulation.

1.2. Development

Development of the GVUI consists of two phases. Phase I focuses primarily on the schematic editing capabilities of the system. The automatic generation of VHDL source code shall be accomplished under the phase II effort.

This user's guide addresses only that portion of the GVUI that has been completed under the phase I development effort.

1.3. Requirements

The GVUI runs on IBM PC computers, including the PC-XT, the PC-AT, and true compatibles. The system must be equipped with a Hercules Monochrome Graphics Card and must be running MS-DOS 2.0 or higher. The executable code minimally requires 128K of RAM and at least an additional 64K must be available for data. These are the

minimal memory requirements for the system to be operable. More realistic requirements are at least 512K of RAM. Any additional memory, up to 1024K, will be automatically allocated for data manipulation.

A hard disk drive is also recommended as a permanent storage medium, though a single floppy drive system may be used for small circuit descriptions or demonstrations.

1.4. Installation

The same procedure is used to install the GVUI on a hard drive or a floppy disk system. The user must first create a subdirectory off of the root directory and name it *CODE*. The *gvui.exe* and *herc.bgi* files should be copied into this directory. Another subdirectory must also be created off of the root directory. It should be named *LIBRARY*. All library files created by the GVUI will be written to this directory. The directory structure is shown in Figure 1.

If the user desires to alter the default subdirectory names, the appropriate definitions may be altered in the *sysconfig.h* header file and all source code files may then be recompiled and relinked using Borland's Integrated Development Environment. The default subdirectory names are defined in the *sysconfig.h* header file as follows:

```
#define ROOTDIR    "\\"
#define TURBODIR  "\\CODE"
#define OUTDIR    "\\LIBRARY"
#define GRAPHDIR  "\CODE"
```

If the subdirectory names are altered, pay close attention to the occurrences of the single and double backslashes in the *sysconfig.h* definitions. The *ROOTDIR*, *TURBODIR*, and *OUTDIR* directory definitions **must** be preceded by double backslashes, whereas the *GRAPHDIR* definition follows the standard convention of being preceded by only a single backslash.

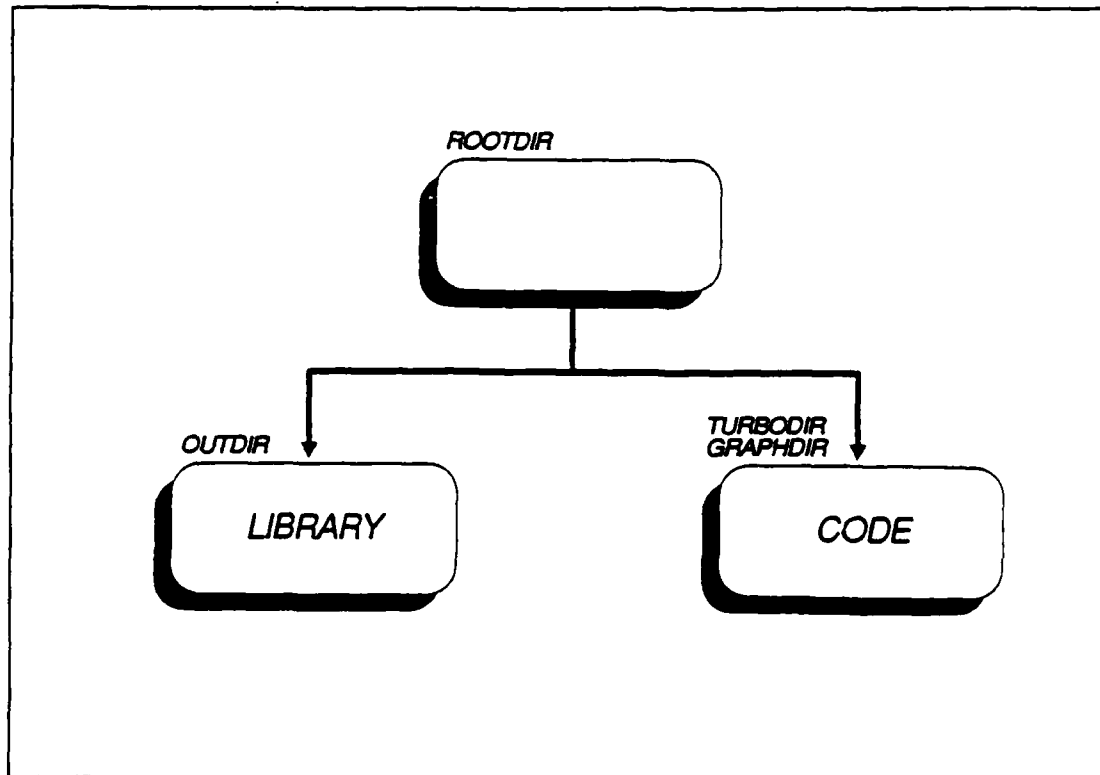


Figure 1. GVUI Directory Structure

2. Getting Started

2.1. Functional Overview (Phase I)

Before an entity's structural architectural body may be described, the interface of the entity must be defined; that is, a *.int* file must exist for the entity in the system library. Using the Entity Interface Editor, an entity interface may be created "from scratch," or may be derived from an interface that already exists. Once an entity interface is defined it may be saved in the system library as a *.int* file.

Provided that an entity's interface has been defined and saved in the library as a *.int* file, its architectural body may be described using the GVUI's Architectural Body Editor. The Architectural Body Editor permits the user to describe an entity by instantiating and interconnecting other entities whose interfaces have already been defined. The architectural body being described may be created "from scratch" as an entirely new design, or may be derived from an architectural body that already exists in the library.

Once the phase II development effort is complete, the system shall generate structural VHDL source code as an ASCII text file from the *.ab* files that are generated using the Architectural Body Editor. The relationship between the Entity Interface Editor, the Architectural Body Editor, the Library, and the phase II Code Generator are shown in Figure 2.

2.2. Loading

1. To initiate execution of the program, enter the *\CODE* subdirectory (or the subdirectory that has been defined for the *TURBODIR* in the *sysconfig.h* header source file).
2. Type: GVUI <RTRN>
3. The executable file will load into memory and the GVUI logo will momentarily appear in the graphic viewport area.

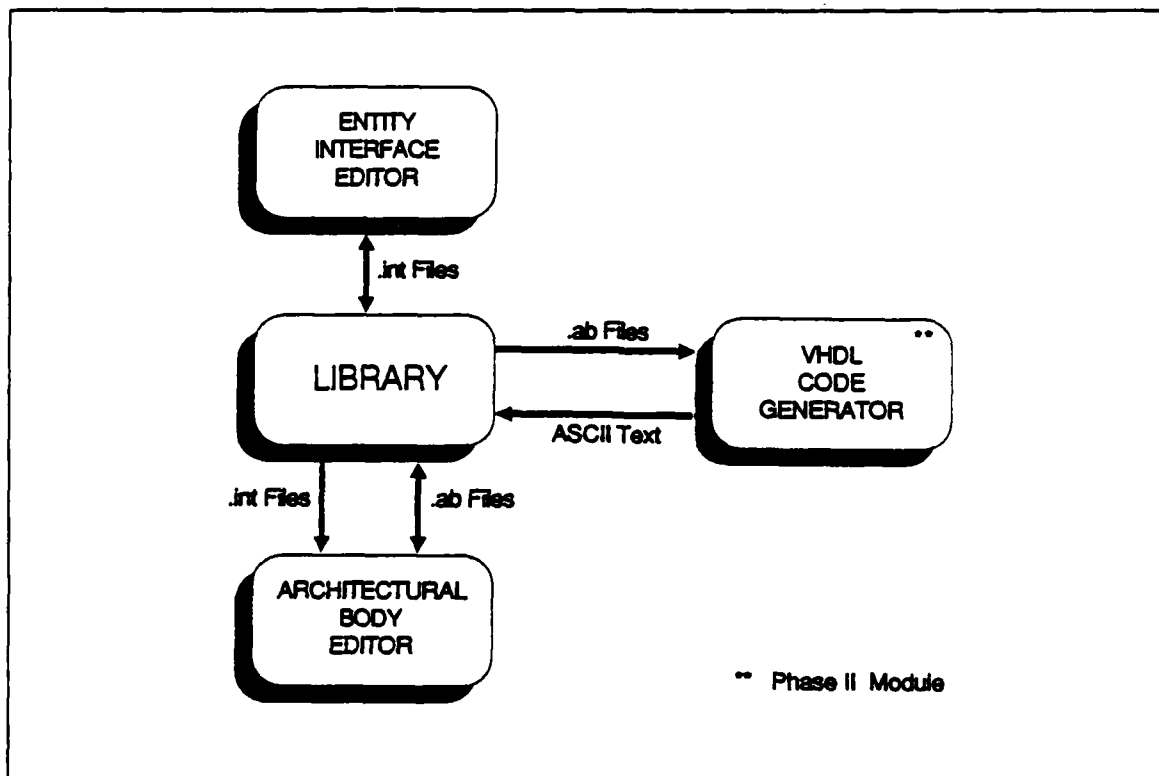


Figure 2. Functional Overview

4. Once the logo disappears, the Main Menu options will be displayed in the menu bar and the user may begin working.

2.3. Selecting Menu Options

1. To move the menu selector up, press the UP arrow key. To move the menu selector down, press the DOWN arrow key.
2. Pressing RTRN will cause the selected menu option to be executed.
3. Figure 3 details the complete GVUI menu hierarchy and may be referenced throughout the remainder of this document.

2.4. Returning to a Previous Menu

1. Complete any selected operations which may be in progress.
2. Once the system enables the menu selector to be moved, the user may press the ESC key to return to a previous menu, assuming the Main Menu is not already displayed.

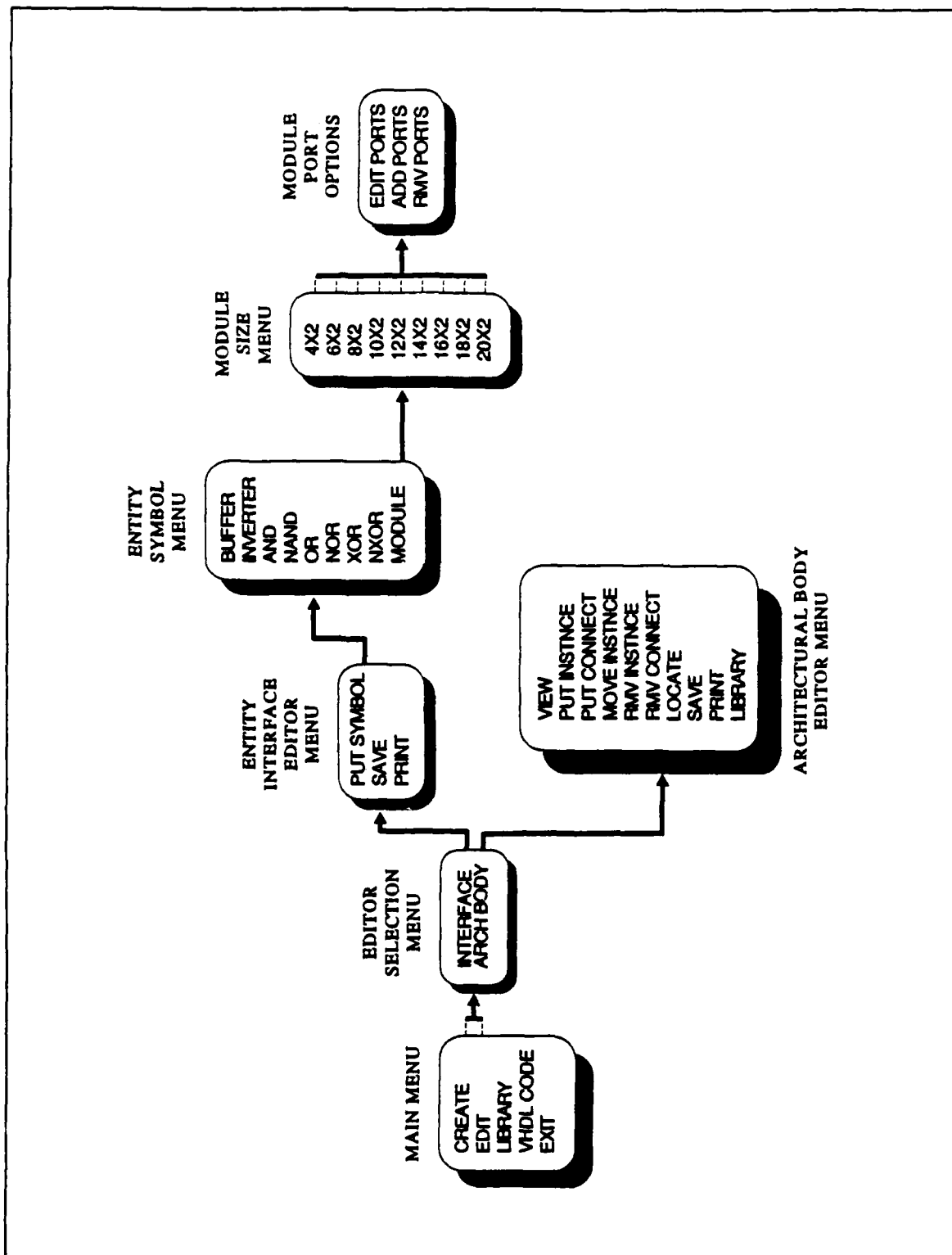


Figure 3. GVUI Menu Hierarchy

3. Main Menu

The Main Menu provides the user with the following options: CREATE, EDIT, LIBRARY, VHDL CODE, and EXIT GVUI. (See Figure 3, page 7.) Each option is briefly described in the following sections.

NOTE: From hereafter, all references made to a positive response may be interpreted as the user pressing Y for "yes." Likewise, all references made to a negative response may be interpreted as the user pressing N for "no."

3.1. CREATE

This option should be selected if the user intends to CREATE a new entity interface or architectural body.

1. Selection results in replacement of the Main Menu by the Editor Selection Menu. (See Chapter 4.)

3.2. EDIT

This option permits the user to edit an existing entity interface or architectural body.

1. Selection results in replacement of the Main Menu by the Editor Selection Menu. (See Chapter 4.)

3.3. LIBRARY

This option permits the user to view the system library.

1. Selection results in replacement of the Main Menu by a directory listing of the GVUI Library.

2. The library listing displays up to and including the first 115 files in the directory.
 - 3.1. If less than 115 files exist in the library, pressing RTRN results in the replacement of the GVUI Library by the Main Menu.
 - 3.2. If greater than 115 files exist in the library, pressing RTRN results in the next 115 files in the directory being displayed. This process repeats itself until all files in the library have been displayed.
 - 3.3. Once all files have been displayed, pressing RTRN returns the user to the Main Menu.

3.4. VHDL CODE

Selection is ignored by the system. (VHDL code generation is to be implemented in the phase II development effort.)

3.5. EXIT GVUI

This option permits the user to exit the GVUI and return to DOS.

1. Selection results in a message that prompts the user to confirm his or her intentions of exiting the GVUI.

4. Editor Selection Menu

The Editor Selection Menu, which is obtained by selecting the CREATE or EDIT option in the Main Menu, provides the user with the following options: INTERFACE and ARCH BODY. (See Figure 3, page 7). These options are briefly described in the sections that follow.

4.1. INTERFACE

This option directs the user to the Entity Interface Editor.

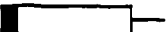
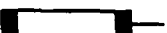
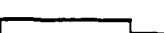
1. If the user arrives at this selection through the CREATE option in the Main Menu, the Editor Selection Menu is replaced by the Entity Interface Editor Menu. (See Chapter 5 for information on options found in the Entity Interface Editor Menu.)
2. If the user arrives at this selection through the EDIT option in the Main Menu, a message prompts the user to enter the name of the *.int* file to be edited. Do not include the *.int* extension to the filename. (See Chapter 5 for information on options found in the Entity Interface Editor Menu.)

4.2. ARCH BODY

This option directs the user to the Architectural Body Editor.

1. If the user arrives at this selection through the CREATE option in the Main Menu, a message prompts the user to enter the name of the *.int* file for which an architectural body is to be created. Do not include the *.int* extension to the filename.
 - 1.1. If the filename does not exist in the GVUI Library, the system returns the user to the Main Menu.
 - 1.2. If the filename exists as a *.int* file, the Editor Selection Menu is replaced by the ABE Menu. Also, the architectural body ports of the associated entity are drawn in the viewport area. The port symbols differ according to the mode of the port it represents. Table I lists each port mode and its respective symbol. (See Chapter 7 for further information on options found in the Architectural Body Editor Menu.)

Table I. Formal Port Symbols

Port Mode	Port Symbol
IN	
OUT	
INOUT	
BUFFER	
LINKAGE	

2. If the user arrives at this selection through the EDIT option in the Main Menu, a message prompts the user to enter the name of the .ab file that is to be edited. Do not include the .ab extension to the filename.
 - 2.1. If the filename does not exist in the GVUI Library, the system returns the user to the Main Menu.
 - 2.2. If the filename does exist as a .ab file, the Editor Selection Menu is replaced by the ABE Menu. Also, a portion of the architectural body to be edited should be visible in the viewport area. (See Chapter 7 for further information on options found in the Architectural Body Editor Menu.)

5. Entity Interface Editor Menu

The Entity Interface Editor Menu, which is obtained by selecting the INTERFACE option in the Editor Selection Menu, provides the user with the following options: PUT SYMBOL, SAVE and PRINT SCRN. (See Figure 3, page 7.) Each option is briefly described in the sections that follow.

5.1. SYMBOL

This option initiates the creation of a new entity interface.

1. Selection results in the replacement of the Entity Interface Menu by the Entity Symbol Menu, where a symbol for the new entity will be selected. (See Chapter 6 for information on options found in the Entity Symbol Menu.)

5.2. SAVE

This option permits the user to save the current entity interface.

1. Selection results in a message that prompts the user to enter a filename under which the entity interface would be saved. Do not attempt to include an extension to the filename.
2. A *.int* extension will automatically be added to the entered filename and the entity will be saved in the GVUI Library.

5.3. PRINT SCRN

Selection of this option is presently ignored by the system. (PRINT SCRN has not been coded under the phase I development effort.)

NOTE: Pressing ESC will cause the system to escape to the Main Menu, but in the process, the current entity interface will be destroyed unless it is saved before ESC is pressed.

6. Entity Interface Editor Submenus

6.1. Entity Symbol Menu

The Entity Symbol Menu, which is obtained by selecting the SYMBOL option in the Entity Interface Editor Menu, permits the user to assign one of the following symbols to the entity to be created: BUFFER, INVERTER, AND, NAND, OR, NOR, XOR, NXOR, or MODULE. The location of the Entity Symbol Menu in the GVUI menu hierarchy may be seen in Figure 3, page 7. The actual symbol that is associated with each symbol option is shown in Table II. Each option is briefly described as follows:

6.1.1. *BUFFER, INVERTER.* These options assign the selected symbol to the entity being edited.

1. Selecting either of these options results in the viewport being cleared, the selected symbol being displayed in the viewport, and a message that prompts the user to assign an entity name to the entity being edited. (The name the user assigns to the entity may be different than the filename under which the entity interface will be saved.)
2. Once an entity name has been assigned, it will appear along with the formal port names in the appropriate locations about the selected entity symbol. Also, a message will appear and ask if the formal port names/modes are acceptable.

NOTE: For BUFFER and INVERTER symbols, a single character jointly serves as the port label and mode identifier. Therefore, only the five characters listed in Table III may be considered as formal port labels for BUFFERS and INVERTERS.

- 2.1. If the user provides a positive response, indicating the port names/labels are acceptable, the user will be returned to the Entity Interface Editor Menu.

Table II. Entity Symbols

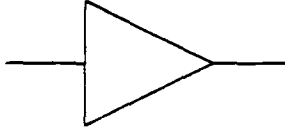
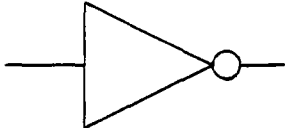
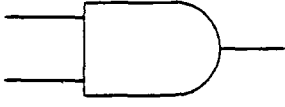
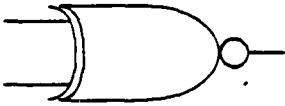
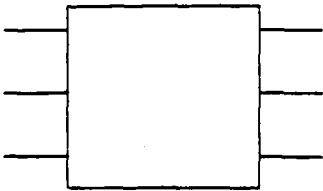
MENU OPTION	SYMBOL
BUFFER	
INVERTER	
AND	
NAND	
OR	
NOR	
XOR	
NXOR	
MODULE	

Table III. Entity Formal Port Modes

PORT MODE	CHARACTER IDENTIFIER
IN	I
OUT	O
INOUT	Z
BUFFER	B
LINKAGE	L

- 2.2. If the user provides a negative response, indicating the port names/modes are not acceptable, a message prompts the user to enter the name of the formal port to be edited.
- 2.2.1. The newly entered port name/mode will be displayed in the viewport at the proper entity port location.
- 2.2.2. The user will again be asked if the formal port names/modes are acceptable. (Continue following step 2.)

6.1.2. *AND, NAND, OR, NOR, XOR, NXOR*. These options assign the selected symbol to the entity being edited.

- 1. Selection of any of these options results in the current viewport being cleared and a message that prompts the user to enter the number of inputs to the entity. The system will accept only the following number of inputs: 2, 3, 4, 8, or 16.

2. Once a valid number of inputs has been entered, the entity symbol is drawn in the graphic viewport area and the user is prompted to assign a name to the entity. (The entity name assigned may be different than the name under which the entity interface is/will be saved.)
3. Once a name is assigned to the entity, the symbol is labeled with the entity and port names and a message appears in the feedback area. It asks the user if the formal port modes/labels are acceptable.

NOTE: The first character of each formal port name identifies its mode. There are five different port modes to choose from. They are listed with their identifying characters in Table III, page 15.

- 3.1. If the user provides a positive response, indicating the port modes/labels are acceptable, the user will be returned to the Entity Interface Editor Menu.
- 3.2. If the user provides a negative response, indicating the port modes/labels are not acceptable, a message will prompt the user to enter the name of the formal port that is to be edited.
 - 3.2.1. Once a valid port name has been entered to be edited, the user will be prompted to enter a new port mode/label. The first character of the new port label designates the port mode. It is therefore constrained to be one of the following: I, O, Z, B, or L.
 - 3.2.2. The newly entered port mode/label will be displayed in the viewport at the proper entity port location. The user will again be asked if the formal port modes/labels are acceptable. (Continue following step 3.)

6.1.3. *MODULE*. This option permits the user to assign the MODULE symbol to the entity being created. A MODULE is represented by a rectangular box whose dimensions are dependent upon the number of ports that are affiliated with the given entity.

1. Selection of this option results in the current viewport being cleared and the Module Size Menu being displayed. (See Section 6.2.)

6.2. Module Size Menu & Module Port Options

The size of a MODULE is given in terms of its number of ports. For instance, if the user selects 6x2 from the Module Size Menu, an entity with 12 ports will be displayed in the viewport. By default, six ports will be designated as input ports and six ports will be designated as output ports. (See Figure 4 for an example of a 6x2 MODULE.)

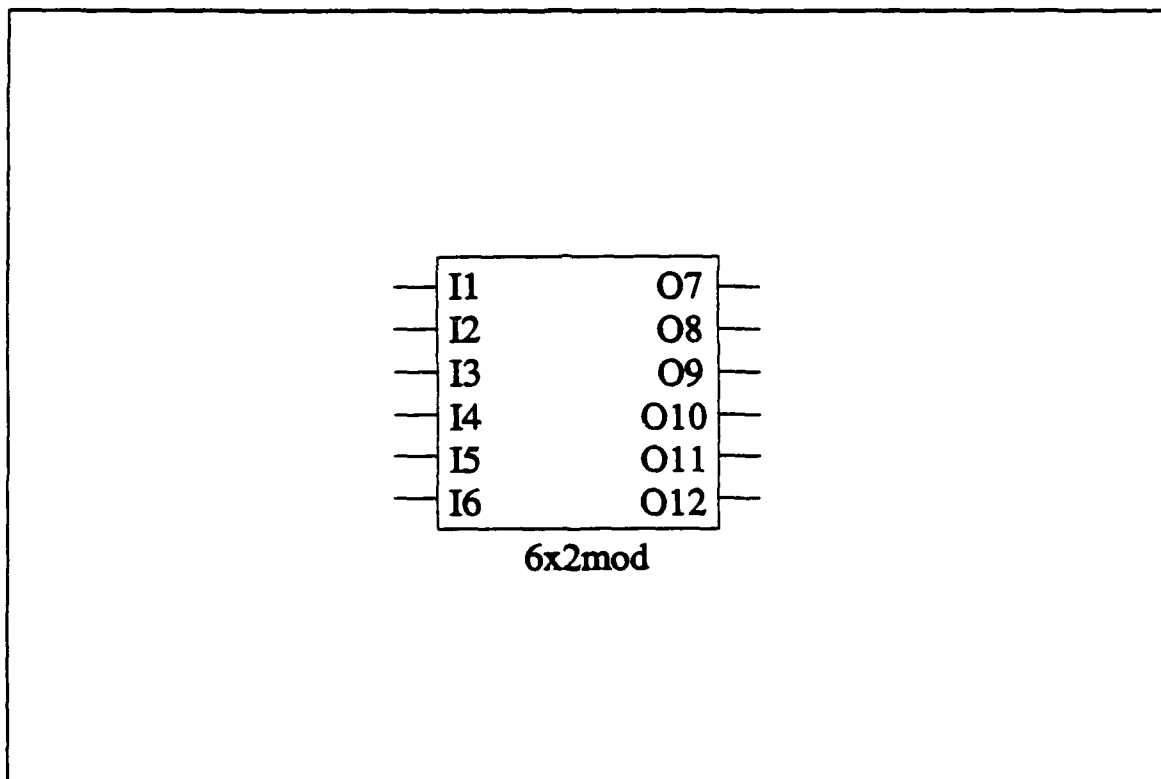


Figure 4. 6x2 MODULE Entity

1. The Module Size Menu permits the user to select the size of the rectangular MODULE symbol which will represent the entity being edited.
2. Selection of any of the module size options results in the module perimeter being drawn in the viewport and a message that prompts the user to assign

a name to the entity being edited. (The name the user assigns to the entity may be different than the filename under which the entity is/will be saved.)

NOTE: The first character of each formal port name identifies its mode. There are five different port modes to choose from. They are listed with their identifying characters in Table III, page 15.

3. Once a name is assigned to the entity, the module symbol is labeled with its entity and port names and the Module Size Menu is replaced by the following Module Port Options: EDIT PORT, ADD PORTS, and RMV PORT. (See Sections 6.2.1. - 6.2.3.)

6.2.1. EDIT PORT. This option permits the user to edit the formal port mode/label for any port on the displayed MODULE.

1. If the user selects the EDIT PORT option, a message will prompt the user to enter the name of the entity port to be edited.
 - 1.1. Once the name of a valid port has been entered to edit, the user will be prompted to enter a new port mode/label. The first character of the new port label designates the port mode. It is therefore constrained to be one of the following: I, O, Z, B, or L. Also note that the Entity Interface Editor will not allow two ports to be assigned the same port mode/label for a given entity.
 - 1.2. The newly entered port mode/label will be displayed in the viewport at the proper entity port location. The user will again be able to make another selection from the list of module port editing options, or may press ESC to return to the Entity Interface Editor Menu.

6.2.2. ADD PORTS. This option permits the user to add ports to a displayed entity MODULE. Ports are symmetrically added four at a time. By default, each port will be

assigned a unique mode/label identifier which may be edited using the EDIT PORT option discussed in section 6.2.1.

1. If the user selects this option, four new uniquely identifiable ports will be added to the entity and displayed in the viewport. (See Figure 5.)

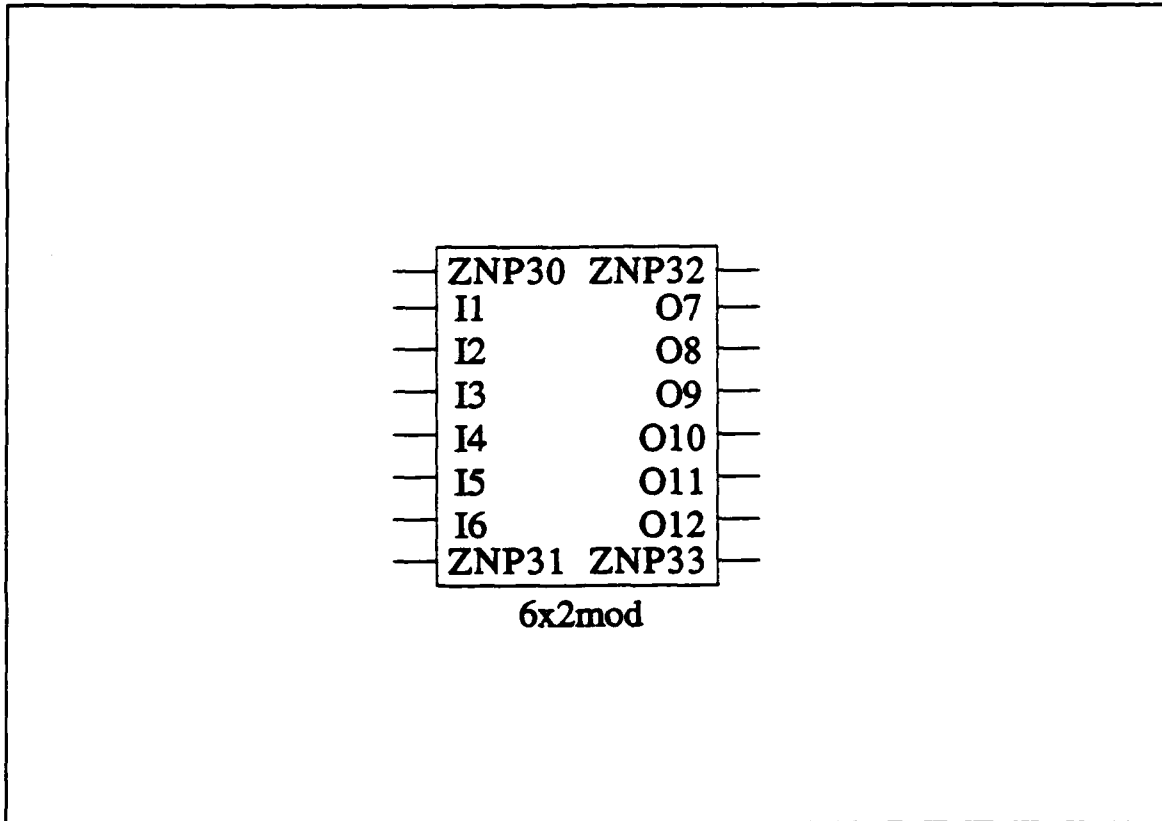


Figure 5. Adding Ports to a MODULE Entity

2. Most likely, the default modes/labels assigned to the newly added ports will not be acceptable to the user. If this is the case, the user may edit the new ports by selecting the EDIT PORT option (discussed in Section 6.2.1.)
3. If four additional ports are too many, the user may remove some of the new ports by selecting the RMV PORT option (discussed in Section 6.2.3.)

NOTE: As a reminder, pressing ESC will return the user to a higher level menu, in this case, the Entity Interface Editor Menu.

6.2.3. *RMV PORT*. This option permits the user to permanently remove formal ports from the displayed entity interface.

1. If the user selects RMV PORT, a message will ask the user for confirmation to proceed.
2. If the user provides a positive response, indicating his or her desire to permanently remove ports from the displayed entity interface, a message will prompt the user to enter the name of the port to be removed. Once a valid port name is entered, the port will be permanently removed from the entity interface.
- 2.2. If the user provides a negative response, indicating the RMV PORT selection was made by error, the selection is disregarded and the user may either make another selection from the list of menu options or may press ESC to return to the Entity Interface Editor Menu.

7. *Architectural Body Editor Menu*

The Architectural Body Editor is analogous to a schematic editor. It permits the user to instantiate components for entities whose interfaces have already been defined through the Entity Interface Editor. The prototype GVUI is presently configured to permit the user to instantiate a maximum of 250 components in his or her architectural description. (The GVUI may be configured at setup time to be capable of instantiating more than 250 components by changing the *CMAX* value in the *port.h* source code header file.)

The Architectural Body Editor (ABE) Menu, which is obtained by selecting the ARCH BODY option in the Editor Selection Menu (see Figure 3, page 7), provides the user with the following options: VIEW, PUT INSTNCE, PUT CONNECT, MOVE INSTNCE, RMV INSTNCE, RMV CONNECT, LOCATE, SAVE, PRINT, and LIBRARY. Each option is briefly described in the sections that follow.

7.1. *VIEW*

This option permits the user to move the viewport to an area specified by a pair of viewport row and column coordinates. The advantage of VIEW over scrolling is its ability to immediately position and display the contents of the viewport area in one screen refresh cycle. Scrolling would require a screen refresh for each scroll increment of the viewport.

1. Selection of this option results in a message prompting the user to enter the column number of the viewport locator for the area desired to be viewed. Viewport column coordinate values range from 0 to 22 inclusive.
2. The user will be prompted to enter the row number of the viewport locator for the area desired to be viewed. Viewport row coordinate values range from 0 to 21 inclusive.
3. The viewport will display the area identified by the two previous viewport coordinate entries.

4. The viewport locator will have repositioned itself in the VLA in accordance with the viewport's new position.
5. The viewport locator and world coordinates will have been updated in the feedback area.
6. The user may select another option in the ABE Menu.

7.2. PUT INSTNCE

This option permits the user to position and instantiate components within the current architectural body. In order to instantiate a component, its interface must already exist in the form of a *.int* file. Once the system verifies its existence, a rectangle is drawn within the graphic viewport area at the present world coordinate cursor location. The rectangle encompasses the area required by the component that is to be instantiated.

Viewport scrolling is automatically enabled throughout the placement process. If the user decides to place a component outside of the graphic viewport area, he or she may *scroll the viewport to the desired destination* using one of the two provided scroll modes.

1. Selection results in a message that prompts the user to enter the name of the entity to be instantiated.
2. If a valid entity name has been entered, a rectangle representing the entity's perimeter will be displayed in the viewport. A message will prompt the user to press RTRN once the entity perimeter is positioned, or to press ESC to exit to the ABE Menu.
3. Once the entity perimeter is positioned and the user presses RTRN, a message will prompt the user to enter an instantiation label for the newly positioned component.

NOTE: The ABE will not permit two components of an architectural body to share the same instantiation label.

4. If a unique instantiation label is assigned to the newly positioned component, a message will ask if the port labels are acceptable.

NOTE: Actual port labels may only be edited at the time the component is instantiated. Once the user indicates that the actual port labels are acceptable, they can no longer be altered.

- 4.1. If the user provides a positive response, indicating the actual port labels are acceptable, the system will complete the instantiation process and the user may select another option from the ABE Menu.
- 4.2. If the user provides a negative response, indicating that the actual port labels are not acceptable, a message will prompt the user to enter the name of the actual port that is to be edited.
 - 4.2.1. If the user enters a valid actual port name, a message will prompt the user to assign a new label to the port.
 - 4.2.2. The new port label will be displayed in the viewport at the proper component port location. The user will again be asked if the actual port labels on the newly positioned component are acceptable. (Continue following step 4)

7.3. PUT CONNECT

This option permits the user to route signal interconnections between component instances. A connection is routed in a segmented fashion. The user must identify the connection's starting location, its intermediate "bend points" when the connection changes directions, and its terminal location. Like PUT INSTANCE, viewport scrolling is automatically enabled when routing connections.

1. Selection of this option results in a message that prompts the user to identify the connection's initial point and to press SPACE once the point has been identified.

2. Once the user identifies the initial point by pressing SPACE another message will appear. It prompts the user to identify the segment endpoint and to press SPACE if it is an intermediate bend point or RTRN if the point terminates the connection.
- 2.1. If the user identifies an intermediate bend point by pressing SPACE, a line segment will be drawn from the previous endpoint to the new segment endpoint. A message will prompt the user to identify the segment endpoint and to press SPACE if it is an intermediate bend point or RTRN if the point terminates the connection. (Continue from step 2.)
- 2.2. If the user identifies a connection's terminal point by pressing RTRN, the system will complete the connection by drawing a line segment from the last segment endpoint to the connection's terminal point. A junction will also be drawn at both the connection's initial and terminal points.
3. Upon completing the connection, the user will be able to select another option from the ABE Menu.

7.4. MOVE INSTNCE

Selection of this option will be ignored by the system. (This function is not yet implemented under the phase I development effort.)

7.5. RMV INSTNCE

This option permits the user to delete the occurrence of an instantiated component in the current architectural body. A component that is to be removed is identified by its instantiation label and is not constrained to lie within the current viewport area.

WARNING: No UNDO option is available. The removal of an instantiated component via the RMV INSTNCE option is permanent.

1. Selection of this option results in a message that prompts the user to enter the name of the component instance to be removed.

2. If the name entered corresponds to an instantiated component that lies outside of the current viewport, the system will notify the user of the location of the instance and will prompt the user for confirmation to proceed.
 - 2.1. If the user provides a negative response to the confirmation prompt, no removal shall occur.
 - 2.2. If the user provides a positive response to the confirmation prompt, the system will indicate that the instance has been removed from the current architectural body.
3. If the user enters a name that corresponds to an instance that is visible in the viewport, the system will draw a rectangle around the instance and request confirmation before removing the instance.
 - 3.1. If the user provides a negative response to the confirmation prompt, the rectangle around the instance will be removed, but the instance shall remain intact.
 - 3.2. If the user provides a positive response to the confirmation prompt, the instance shall be removed from the current architectural body.
4. The user may select another option from the ABE Menu.

7.6. RMV CONNECT

This option permits the deletion of connections that have already been placed in an architectural body. To identify the connection to be removed, the user must position the cursor over a junction that lies on an endpoint of the connection. The system will then highlight the connection for removal verification. In cases where the identified junction consists of three or more intersecting paths, the intersecting connections are highlighted one at a time until one is positively identified by the user for removal.

1. Selection of this option results in the user being prompted to identify a junction on the connection that is to be removed and to press RTRN.
2. If the user positions the cursor over a junction and presses RTRN, the system will highlight a connection that intersects the junction. Also, "Correct?" will appear in the feedback area.

- 3.1. If the user provides a positive response, indicating the correct connection has been highlighted, the highlighted connection will be removed.
- 3.2. If the user provides a negative response and other connections (that have not yet been highlighted) intersect the identified junction, the system will highlight another connection. "Correct?" will again appear in the feedback area. (Continue following step 2.)
- 3.3. If the user provides a negative response and no other connections (that have not yet been highlighted) intersect the junction, the system will notify the user that it was unable to locate another connection that intersects the junction.

NOTE: The user may press ESC to cancel the connection removal process, but the cancellation may only occur before a removal has actually occurred.

4. Once the connection has been removed, the user may select another option from the ABE Menu.

7.7. LOCATE

This option permits the user to identify the relative location of a component with respect to the location of the current viewport. This option alleviates the need to scroll aimlessly across the world coordinate system in search of a component that he or she believes to exist.

Upon locating a requested component, the system will display the component's viewport row and column coordinates in the feedback area and the component locator will appear in the VLA. The component locator appears as a small solid box in the VLA and portrays the relative location of the requested component with respect to the present viewport location.

1. Selection of this option results in a message that prompts the user to enter the name of the instance to be located.

- 2.1. If the user enters the name of a component that has not been instantiated in the current architectural body, the system will inform the user of its inability to locate the desired component.
- 2.2. If the user enters the name of a component that has been instantiated in the current architectural body, the system will display the viewport locator coordinates for the area within which the component is positioned. Also, the component's relative location will be highlighted by the component locator in the VLA.
3. The user may select another option from the ABE Menu.

7.8. SAVE

This option permits the user to save the current architectural body as a *.ab* file in the GVUI Library.

1. Selection results in a message that prompts the user to enter a filename under which the architectural body is to be saved. Do not attempt to include an extension to the filename.
2. A *.ab* extension will automatically be added to the filename and the current architectural body will be saved under this filename in the GVUI Library.

7.9. PRINT

Selection of this option will be ignored by the system. (PRINT has not been coded under the phase I development effort.)

7.10. LIBRARY

This option is identical to the LIBRARY option listed in the GVUI Main Menu, except upon returning from the Library, the graphic display of the current architectural body will be restored and control will be returned to the ABE.

1. Selection results in replacement of the ABE Menu by a directory listing of the GVUI Library.
2. The library listing displays up to and including the first 115 files in the directory.
- 3.1. If less than 115 files exist in the library, pressing RTRN results in the replacement of the GVUI Library by the ABE Menu.
- 3.2. If greater than 115 files exist in the library, pressing RTRN results in the next 115 files in the directory being displayed. This process repeats itself until all files in the library have been displayed.
- 3.4. Once all files have been displayed, pressing RTRN returns the user to the ABE Menu.

NOTE: Pressing ESC in lieu of any other ABE option will cause the system to warn the user that the current architectural body will be lost if it is not saved before leaving the ABE Menu. The user must save the current architectural body before escaping to the Main Menu if he or she intends to use or edit it later on.

Bibliography

1. Aylor, J.H., R. Waxman and C. Scarratt. "VHDL - Feature Description and Analysis," *IEEE Design & Test of Computers*, 3:17-27 (April 1986).
2. Collett, Ronald E. "VHDL and EDIF Standardize CAE/CAD," *ESD: The Electronic System Design Magazine*, 17:28,30 (December 1987).
3. Computer Science Branch. *Interactive VHDL Workstation: Program Status Review Report*, 8 October 1987. Air Force Contract F33615-85-C-1862. Information Systems Laboratory, Corporate Research & Development, General Electric Company.
4. Cosgrove, J. "Needed: A New Planning Framework," *Datamation*, 17:37-39 (December 1971).
5. DeGroat, Joseph and others. "The AFIT VHDL Environment," *Proceedings of the IEEE 1988 Frontiers in Education Conference*, 324-329. New York: IEEE Press, 1988.
6. Department of the Air Force, Avionics Laboratory. *Interactive VHDL Workstation (Prototype Version 4.2) Software User's Manual*. Air Force Contract F33615-85-C-1862. CDRL Sequence No. 10. General Electric Company, Schenectady, NY, 15 January 1988.
7. Dewey, Allen and Anthony Gadiant. "VHDL Motivation," *IEEE Design & Test of Computers*, 3:12-16 (April 1986).
8. Dudley, Robert S. "Making Warplanes Lean and Mean," *Air Force Magazine*, 71:38-42,45 (January 1988).
9. Elliot, John P. and Steven P. Levitan. *Sced: An Icon Based Schematic Editor*. TR-CE-88-001. Department of Electrical Engineering, University of Pittsburgh, Pittsburgh PA, 1988.
10. Freeman, P. "Toward Improved Review of Software Designs," *Proceedings of the 1975 National Computer Conference*, 44:329-334, 1975.
11. Gilman, Alfred S. "VHDL - The Designer Environment," *IEEE Design & Test of Computers*, 3:42-47 (April 1986).
12. Goering, Richard. "VHDL Implentation Allows Behavioral Descriptions," *Computer Design*, 27:21 (January 15, 1988).
13. Gregory, B.L. "Solid State Technology in the 21st Century," *Radio-Electronics*, 58:97-98 (May 1987).
14. Hearn, Donald and M. Pauline Baker. *Computer Graphics*. Englewood Cliffs: Prentice-Hall, 1986.
15. Hetzel, William. *The Complete Guide to Software Testing*. Wellesley MA: QED Information Sciences Inc, 1984.
16. Heilmeier, George. "The Future of Artificial Intelligence," *Radio-Electronics*, 58:85-90 (May 1987).

17. Myrvaagnes, Rodney. "VHSIC Program Moves on in Phase 2," *Electronic Products*, 28:45-51 (October 1, 1985).
18. Petzold, Charles. "IBM Goes Analog: New Video Standards Show Off Color," *PC Magazine*, 6:10 (May 26, 1987).
19. Pressman, Roger S. *Software Engineering*. New York: McGraw-Hill, 1982.
20. Ross, D.T. and J.W. Brackett, "An Approach to Structured Analysis," *Computer Decisions*, 8:40-44 (September 1976).
21. Saunders, Larry F. "The IBM VHDL Design System," *Proceedings of the 24th ACM/IEEE Design Automation Conference*. 484-490. New York: IEEE Computer Society Press, 1986.
22. *Schema II Demonstration Disk*. Software. Omaton, Inc., Richardson TX, 1987.
23. *Schematic Design Tools Demonstration Disk, Version 1.20*. Software. OrCAD Systems Corporation, Hillsboro OR, 1987.
24. Shadad, Moe. "An Overview of VHDL Language and Technology," *Proceedings of the 23rd ACM/IEEE Design Automation Conference*. 320-326. New York: IEEE Computer Society Press, 1986.
25. Veit, Stan, "What You Need to Know," *Computer Shopper's PC Clones*, 1:1 (1987).
26. *VHDL Design Workbench User's Manual*. Vista Technologies, Inc., Rolling Meadows IL, December 1987.
27. Waxman, Ron. "The VHSIC Hardware Description Language - A Glimpse of the Future," *IEEE Design & Test of Computers*, 3:10-16 (April 1986).
28. Weiss, Ray. "VHDL Subsets for CAE," *Electronic Engineering Times*, Issue 497:51-62 (August 1, 1988).
29. "What's Happening at ASD," *Air Force Magazine*, 71:71-87 (January 1988).
30. Yourdan, Edward. *Techniques of Program Structure and Design*. Englewood Cliffs: Prentice-Hall, 1975.
31. Yourdan, Edward and Larry L. Constantine. *Structured Design*. Englewood Cliffs: Prentice-Hall, 1979.

VITA

Captain Stephen M. Matechik [REDACTED] He studied electrical engineering at the Pennsylvania State University under a four-year academic AFROTC scholarship and received the degree of Bachelor of Science in Electrical Engineering in August 1983, at which time he also received his commission. He was assigned to the Aeronautical Systems Division, Wright-Patterson AFB, Ohio, where he became the lead avionics systems engineer for advanced development communications and identification systems for the F-16 fighter aircraft. He served at the F-16 Systems Program Office until entering the School of Engineering, Air Force Institute of Technology, in May 1987.

[REDACTED] [REDACTED]
[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 / GE / ENG / 88D-25			5. MONITORING ORGANIZATION REPORT NUMBER(S)			
6a. NAME OF PERFORMING ORGANIZATION School of Engineering		6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION			
6c. ADDRESS (City, State, and ZIP Code) Air Force Institute of Technology Wright-Patterson AFB OH 45433-6583			7b. ADDRESS (City, State, and ZIP Code) <i>12 Jan 1989</i>			
8a. NAME OF FUNDING / SPONSORING ORGANIZATION AF Wright Aeronautical Labs		8b. OFFICE SYMBOL (If applicable) AFWAL / AADE	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER			
8c. ADDRESS (City, State, and ZIP Code) Wright-Patterson AFB OH 45433			10. SOURCE OF FUNDING NUMBERS			
			PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.
11. TITLE (Include Security Classification) THE DESIGN AND IMPLEMENTATION OF A GRAPHICAL VHDL USER INTERFACE						
12. PERSONAL AUTHOR(S) Stephen M. Matechik, Capt, USAF						
13a. TYPE OF REPORT MS Thesis		13b. TIME COVERED FROM _____ TO _____		14. DATE OF REPORT (Year, Month, Day) December 1988		15. PAGE COUNT 140
16. SUPPLEMENTARY NOTATION						
17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)			
FIELD	GROUP	SUB-GROUP				
12	05					
			Computer Aided Design (CAD)			
			Hardware Description Language (HDL)			
			Schematic Diagram			
			Schematic Entry			
			User Interface			
			VHDL			
19. ABSTRACT (Continue on reverse if necessary and identify by block number) Thesis Advisor: Bruce George, Captain, USAF This thesis outlines the design and implementation of a Graphical VHDL User Interface (GVUI). Though the GVUI is designed as an integral component of the UNIX-based Air Force Institute of Technology (AFIT) VHDL Environment (AVE), it is a stand alone tool that operates under MS-DOS on an IBM PC-XT personal computer. The goal of the GVUI is to automatically generate VHDL source code from a graphic schematic diagram. To meet this goal, two phases of development were identified. Phase I primarily addresses the schematic generation and editing capabilities of the system. Phase II shall focus on the system's automatic code generation capabilities. This thesis specifically addresses and completes the first of these two phases. A user's guide is included as an appendix to this thesis.						
20. DISTRIBUTION / AVAILABILITY OF ABSTRACT ☐ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS				21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED		
22a. NAME OF RESPONSIBLE INDIVIDUAL Bruce George, Captain, USAF				22b. TELEPHONE (Include Area Code) (513) 255-3576		22c. OFFICE SYMBOL AFIT / ENG