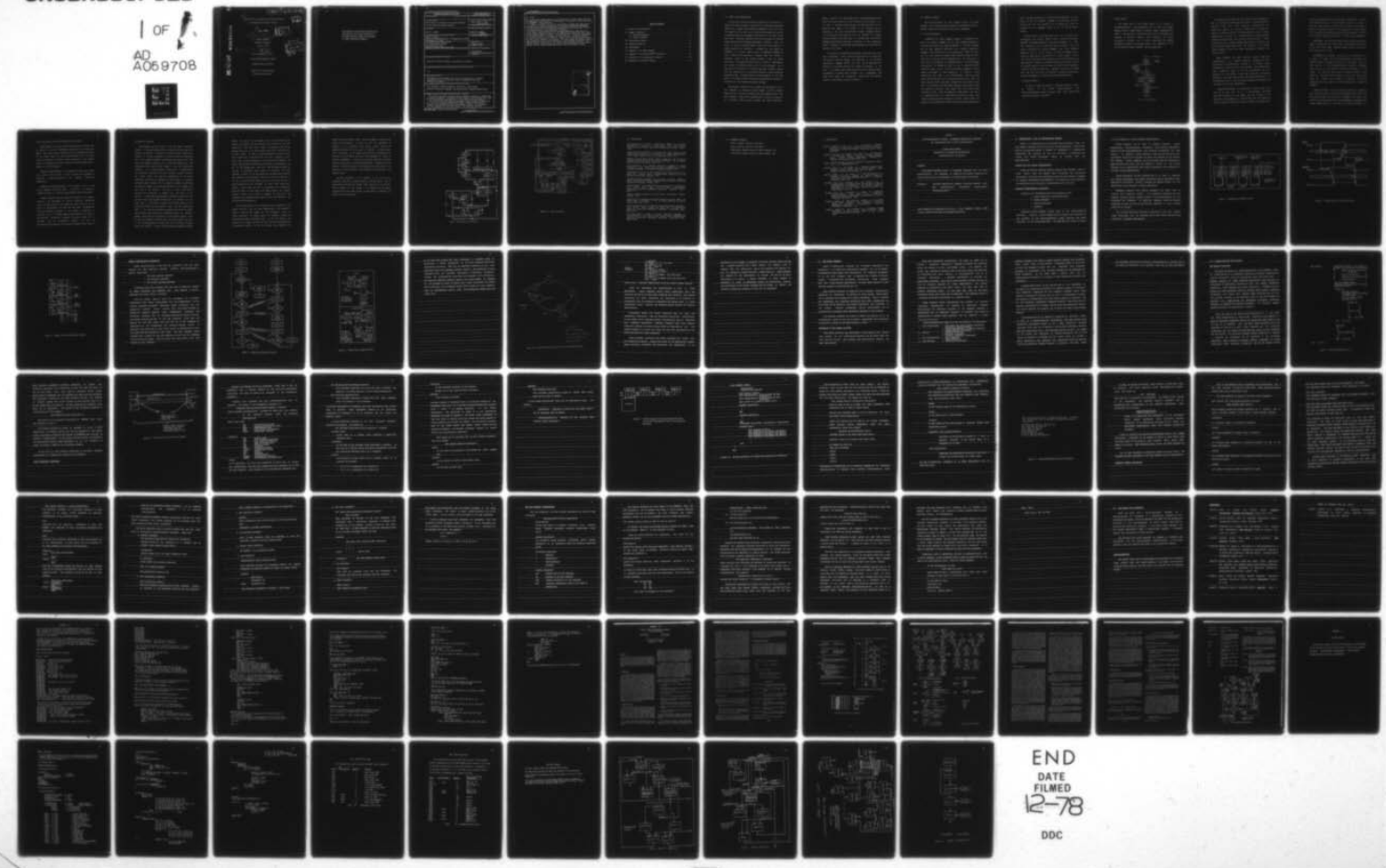
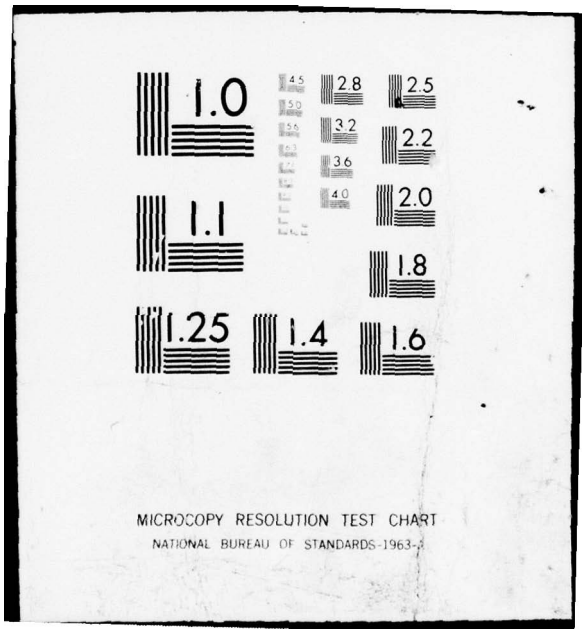
AD-A059 708 CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF ELECTRICA--ETC F/G 9/2
SPECIFICATION, SIMULATION AND AUTOMATED DESIGN OF INTERFACES AN--ETC(U)
JUL 78 A C PARKER DAAG29-76-G-0224

UNCLASSIFIED ARO-13670.10-EL NL

1 OF 1
AD A059708



END
DATE FILMED
12-78
DDC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

(18) ARO (19) 13670.10-EC

(6) Specification, Simulation and Automated Design of Interfaces and Digital Circuits.

12 SC

(9) FINAL REPORT, 1 May 76 - 30 May 78

LEVEL I

(10) Alice C. Parker

Assistant Professor

July 31, 1978

(11) 31 Jul 78 (12) 92 p.

U.S. Army Research Office

(15) Grant Number DAAG29-76-G-0224

Carnegie-Mellon University

Approved for Public Release;

Distribution Unlimited.

403 445

78 10 06 030

AB

THE FINDINGS IN THIS REPORT ARE NOT TO BE CONSTRUED AS AN OFFICIAL DEPARTMENT OF THE ARMY POSITION, UNLESS SO DESIGNATED BY OTHER AUTHORIZED DOCUMENTS.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Specification, Simulation and Automated Design of Interfaces and Digital Circuits		5. TYPE OF REPORT & PERIOD COVERED Final: May 1, 1976- May 30, 1978 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Alice C. Parker		8. CONTRACT OR GRANT NUMBER(s) DAAG29-76-G ~~0024~~ 0224
9. PERFORMING ORGANIZATION NAME AND ADDRESS Alice C. Parker Electrical Engineering Dept., Carnegie-Mellon Univ. Pittsburgh, PA 15213		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS U.S. Army Research Office Post Office Box 12211 Research Triangle Park, NC 27709		12. REPORT DATE July 31, 1978 13. NUMBER OF PAGES
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

NA

18. SUPPLEMENTARY NOTES

The findings of this report are not to be construed as an official Department of the Army position, unless so designated by other authorized documents.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

I/O, Interface, Design Automation, Simulation, Logic Design, Hardware Descriptive Language, Bus Specification, Computer-Aided Design.

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

This report describes research done in hardware description, simulation, and design automation. Although the basic thrust of the work has been aimed at I/O and interface problems, most of the results are more general.

The efforts in formal hardware description have produced a language for bus, I/O and interface specification, GLIDE. GLIDE is supported by a compiler which performs syntactic and semantic checks. A translator to the ISPL language has also been written. The resulting code and → next page

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. (cont.)

non-translatable GLIDE semantics forced abandonment of ISPL either directly or indirectly from GLIDE for I/O description. An example of GLIDE describing the UNIBUS[tm] is included here.

In order to simulate I/O and bus transactions, changes were made to the ISP simulator to allow specification of timing and independent PROCESSES. A further experiment in simulation was done with ISP descriptions of micro-code execution for a dozen commercial processors.

The major research reported here is design automation work, part of a larger design automation project at Carnegie-Mellon University. This grant has supported synthesis research - the mapping from a functional description of the system (ISP) to be designed to the structure. Automated design of the control circuitry is in the early stages, but a working program designs the data paths, registers and memories. An early design the program produced came within 25% of the cost of the design a human designer produced. The PDP-8/E design, included here, has a chip count within 50% of that commercial design, but the design program produced an entirely different design.

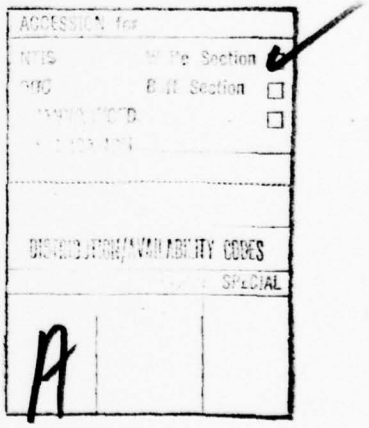


TABLE OF CONTENTS

1.0 SCOPE OF THE INVESTIGATION

The motivation behind the research described in this report is to enhance the digital designer's capabilities by producing more powerful design tools. Digital logic design has progressed to the point that the operation of the logic can be functionally expressed by a variety of hardware descriptive languages, ISP being one of the more widely used ones. Functional simulators exist and are useful for verifying system operation and performance measurements (Barb77a). Thus, the state of the art in digital design is such that the next addition to design aids should be synthesis, a program that can design the STRUCTURE of a digital system, given its FUNCTION or BEHAVIOR as input. Along with the synthesis of hardware comes the problem of producing optimal or near optimal designs to meet the design constraints. The research reported on here is aimed at understanding the design or synthesis process so that it can be automated. One of the goals of this project is to produce logic-level hardware designs from ISP descriptions in a non-optimal fashion to better understand automated design. (A parallel goal of related research by the same group is to develop discrete optimization algorithms and technique to be applied in a more complex and powerful package.

Unfortunately, the area of I/O interface and bus design is not as well organized or developed as digital design. In order to produce design aids for this kind of problem, much more background effort has been necessary. First of all, specification of bus and I/O operation is a different, more difficult problem than logic description.

Second, simulation is more complex due to timing dependencies which affect the logical operation of the interfaces to I/O and buses. So, in order to automate interface design, the remaining effort on this grant has been aimed at the problem of I/O interface and bus description. The goal here has been to produce a language suitable for I/O description and simulation, with the automation of complex interface design a more distant goal. At the same time, the synthesis programs described above have been constructed so that reasonable, simple interfaces to the hardware being designed can be specified and included in the design.

In the course of pursuing the above goals, some other areas have been investigated. These include the specification of a module set data base for interface designs, the comparison of the interface specification language (GLIDE) with ISPL, and the description and simulation of microcode execution for a number of processors. In the area of I/O design, some further specifications of a general-purpose programmable I/O processor were produced, and a programmable FIFO buffer chip design was investigated. Publications and technical reports in these areas are listed in Section 3.

2.0 SUMMARY OF RESULTS

We are presenting here two main research results - the GLIDE language, and a working synthesis program, the data-memory allocator. Related conclusions and results are also briefly enumerated.

2.1 The GLIDE Language

A summary of the GLIDE language progress is presented here. Since the complete language has not been published elsewhere, a pre-publication report is attached as Appendix I. The GLIDE language has now been completely specified, and a compiler supports the language. Early effort went into the comparison of GLIDE and ISPL, and this work produced a compiler which translated GLIDE into ISPL. The result of this was a better understanding of the limitations of ISPL, and the introduction of the PROCESS concept in the ISPS language and simulator. By PROCESS we mean the set of register-transfer operations which exist in a control environment independent from the control environment of other operations. In addition, timing capabilities were added to the ISPS simulator. Some of the GLIDE control structures could not be translated accurately into ISPL, and some primitive GLIDE operations expanded into large blocks of ISPL code. In particular, the GLIDE memory constructs include FIFO queues and associative memories, which expand into long routines when translated to ISPL. Also, the semantics of GLIDE contain the notion of synchronous data I/O, which cannot be described in ISPL. Other Primitive operations which translate to routines include parity bit generation and checking, data formatting, and packing and unpacking of

words. The major conclusion to be drawn from the comparison is that GLIDE and ISP are different languages for describing different entities, and that the problems with I/O description force the existance of both languages - GLIDE for I/O and ISP for digital systems.

The maor output of the GLIDE effort so far is two partial bus descriptions - the military computer GYK/12 I/O bus and the PDP-11 UNIBUS™. The UNIBUS description is attached as Appendix II. Three main conclusions can be drawn from these two examples. First, the control structures for nesting PROCESSES have some undefined semantics, and it is not obvious the effect the PROCESS priority structure should have on the execution of a GLIDE program. Second, the control structures inside processes are not block structured, and hence unwieldy. However, the descriptions seem to accurately reflect the logical operation of the two buses, and therefore, the language is viable for bus and I/O description. (Attempts to describe the UNIBUS with ISPL and ISPS have not resulted in complete descriptions). Efforts are underway to validate the GLIDE UNIBUS description.

2.2 Design Automation

In order to discuss the results of the design automation effort, an overview of the RT-CAD (Register-Transfer level Computer-Aided-Design) system is presnted here. This overview was originally published in (Snow78a).

RT-CAD OVERVIEW

The ulmate goal of the RT-CAD project is to provide a technology-relative, structured-design aid to help the hardware designer explore a larger number of possible design implementations. Inputs to the system are a behavioral description of the system to be designed, an objective function which specifies the user's optimization criteria, and a library specifying the hardware components available to the design system. The components of the RT-CAD system are shown in Figure 1 and discussed below.

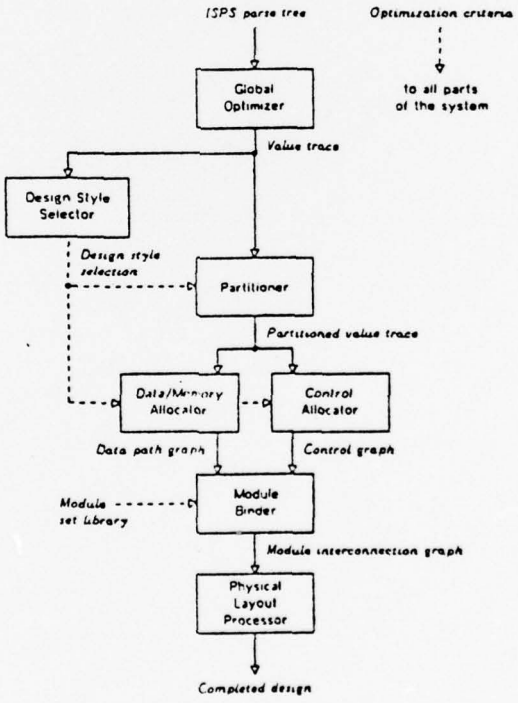


Figure 1: RT-CAD System Overview

The RT-CAD system differs from other design automation systems in that it operates from a behavioral specification. Such specification provides a model that, while accurately characterizing the input-output behavior of a piece of hardware, does not necessarily reflect its internal structure. The design process is one of binding implementation decisions in a top-down manner as a design proceeds through the RT-CAD system. More and more structural detail is frozen at each level until a complete hardware specification is obtained, the most influential design search space. The functions of the design system components which bind these implementation decisions are described below.

GLOBAL OPTIMIZER. The global optimizer applies high-level transformations to a design's behavioral representation after translating it from ISPS notation (Barb77b) to an abstract design representation called the value trace (Snow78b). The transformations have a significant impact on the cost, performance, and other parameters of the designs to which they are applied. The research described in this paper centers around the design representation, the transformations upon it, and the strategy guiding their application in the search for an optimal implementaion.

DESIGN STYLE SELECTOR. By considering the various module sets that can be used (e.g., TTL vs. a microprocessor), the design constraints imposed (e.g., cost, speed), and the structure of the algorithm to be designed (e.g., pipeline data flow), the design style selector decides on the specific style of design to be employed (e.g.,

bit:slice microprocessor, MOS microprocessor, SSI/MSI logic). Earlier work (Thom77b) shows this to be an influential decision in terms of cost and speed tradeoffs. When the style is selected, the design is passed to an allocator specific to the design style. Initial research into the design style selection process has been completed (Thom 77a) and an automatic design style selector is currently being programmed.

PARTITIONER. The partitioner groups operations from the abstract design representation into control steps. This effectively binds the control flow for the design. Tradeoffs between the data and control parts are made at this level.

DATA/MEMORY (DM) ALLOCATOR. The function of the DM allocators is to decide the number and type of data operators, muliplexors, and registers needed to implement the data part of the design. They are style specific in that they embody analytic and heuristic knowledge about a style (e.g., the trade-offs involved in the design of a TTL system), but they do not have access to the specific details of each module set. The output of the allocator is a data path graph whose nodes are elements such as adders or registers. An initial implementation of an allocator for the TTL design style is reported in (Hafe78).

CONTROL ALLOCATOR. The control allocator generates a sequential state machine to control the data paths produced by the DM allocator. The control allocator has the option of designing the control unit around control philosophies such as microprogramming, programmed logic arrays, random logic, etc. The output of the control allocator is a

control path graph whose nodes represent control states.

MODULE BINDER. The module binder selects physical modules from the module set library to implement a design's data and control path graphs. The library contains descriptions of the components available to the design system and may be freely updated so that it is kept current with respect to advances in module technology. This dynamic aspect of the module set library provides for the technology-relative aspects of the RT-CAD system.

PHYSICAL LAYOUT PROCESSOR. This component partitions the system into printed circuit boards or chips, decides the placement of components, routesinterconnections, and prepares engineering documentation.

Research is currently underway into the design of all of the system components described above. In addition, the problem of integrating them into a coherent design system is being investigated.

Research supported under this grant has focused on the synthesis routines - the data-memory and control allocators. Although the control allocation effort is just beginning, some ideas as to the nature of the problems to be solved have been posed. The generation of control hardware is analogous to the problem of generation of microcode, with its inherent computational complexity, but there is one difference. Generation of hardware introduces another set of variables into the optimization routines. Not only are microinstructions generated, but the control hardware itself must be

designed and optimized.

More progress has been made on the data memory allocation problem. A non-optimizing allocator has been written and reported in (Park78) and (Hafe78). In order not to duplicate these publications, (Hafe78) is attached as Appendix III, and the results are summarized here. This allocator produces a distributed logic design of the data paths and storage locations for a given ISPL description). (The program uses ISPL instead of ISPS because of the ISPS development timetable. It is being modified to accept ISPS). It performs some error checking to indicate to the user potential resource conflicts and design errors, and functions independently of the actual integrated circuits used to implement the logic diagram it produces. Preliminary checks indicate that the designs are capable of performing the functions present in the original description. Two designs have been done by the allocator. The first is part of an elevator controller and is described in Appendix III. The second is the PDP-8/E. A non-optimal hand mapping of integrated circuits onto the allocator output logic diagram has been done, and estimate of chip count made. It is difficult to compare the automated design with the original DEC design for three reasons. First, the ISPL description input to the allocator declares as registers some values the PDP-8E uses but never stores explicitly in registers, such as the effective address. These show up as registers in the allocator's design. Also, the allocator designs distributed logic, and the DEC design was done in the central-accumulator design style (For a discussion of design styles, see (Thom77)). Finally, the DEC design has assumed a boundary

between the control and data-memory parts of the design, but the boundary is different from that imposed on the allocator by the ISPL description. Thus some tests, flags, and registers which must be declared explicitly in the ISPL description are part of the control in the DEC design. In spite of these differences, estimates of chip count indicate that the allocator uses 50% more integrated circuit chips than the human designers for the data paths and registers. Of course, these estimates were made using the same 1970 technology chip set the DEC designers had to deal with. The 50% excess hardware can be found in muliplexers which connect the registers, the extra registers declared in the ISPL description, and duplicated operators like increment, add, and compare. Much of this can be attributed to the way in which the ISPL description had to be written, and some of these constraints will not be present in future ISPS descriptions. However, other chips can only be eliminated when optimization algorithms operate at some stage of the design process. The complete allocator output can be found in Appendix IV, along with the implementation information used to make the chip count estimates, and the PDP-8/E ISPL description.

One interesting point to be illustrated is the differences in the design seen even from the block diagram level. This is shown in Figure 2. There are two reasons for the differences. First, as stated previously, the design styles are different. Second, the multiplexing is used in different ways. In the DEC version, the operators are shared, and are even used to provide no-op paths from one register to another. In the CMU version, only registers are

shared and use multiplexed inputs. The ISPL language is partially the source of this disparity. In ISPL, the user can repeatedly use register A as a destination from various sources. However, the expressions A+B and C+D do not imply (or discount) a single adder. Other differences in the design include the use of multiplexers for shifting in the DEC design, and use of true/complement 0/1 chips for creating complements. "Oring" of the MQ and AC registers in the DEC version is done within the multiplexing hardware. Constants are often created in one place and gated over already existant data paths to the registers. In the CMU version, these constants are multiplexed at the register inputs.

One final difference is the treatment of the Link FF and Accumulator register as a single register in the CMU version. This is done because of the way the PDP-8/E ISPL description was written. Further analysis of this design is in progress and includes an implmentation of the control by hand. Comparisons of the DEC and CMU speeds will then be possible.

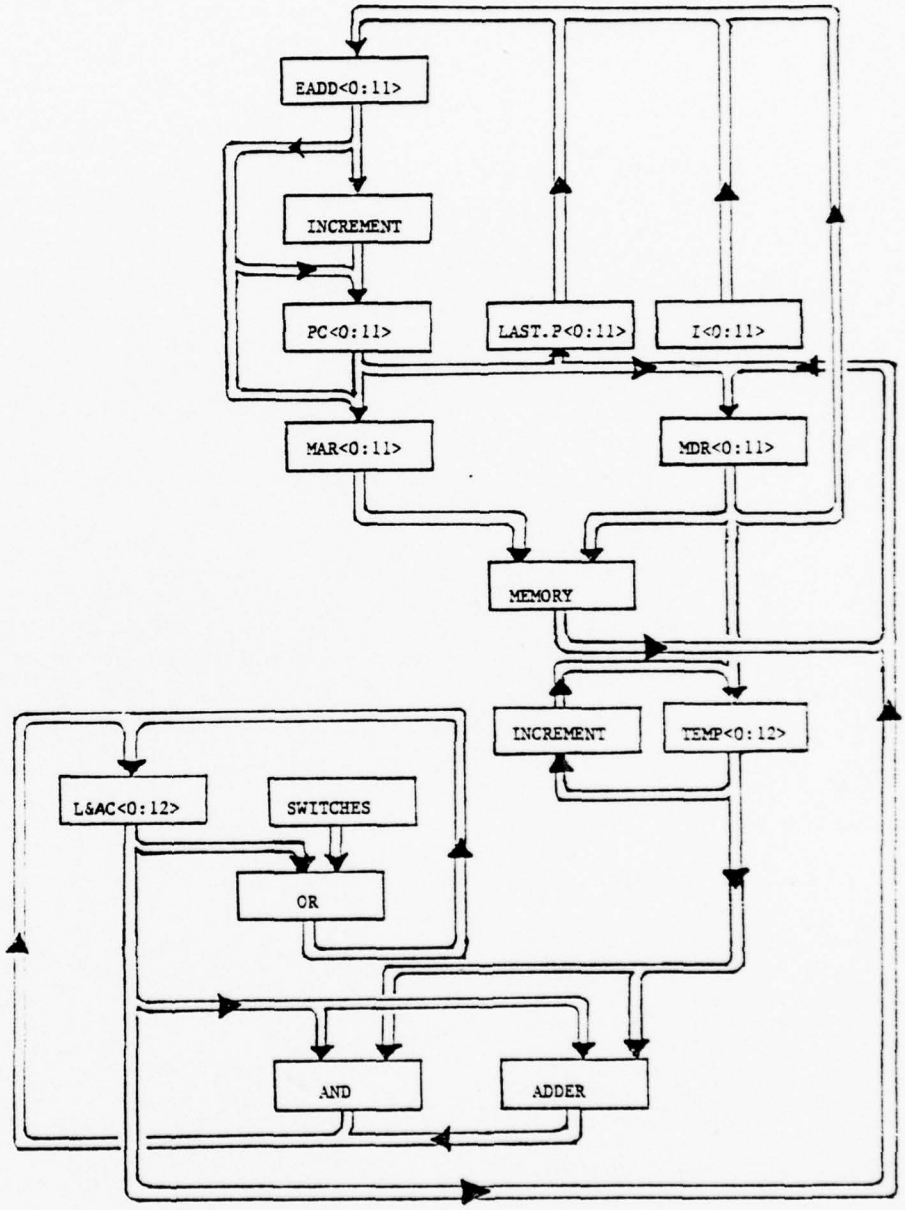


Figure 2a. Block Diagram of CMU PDP-8 Design.

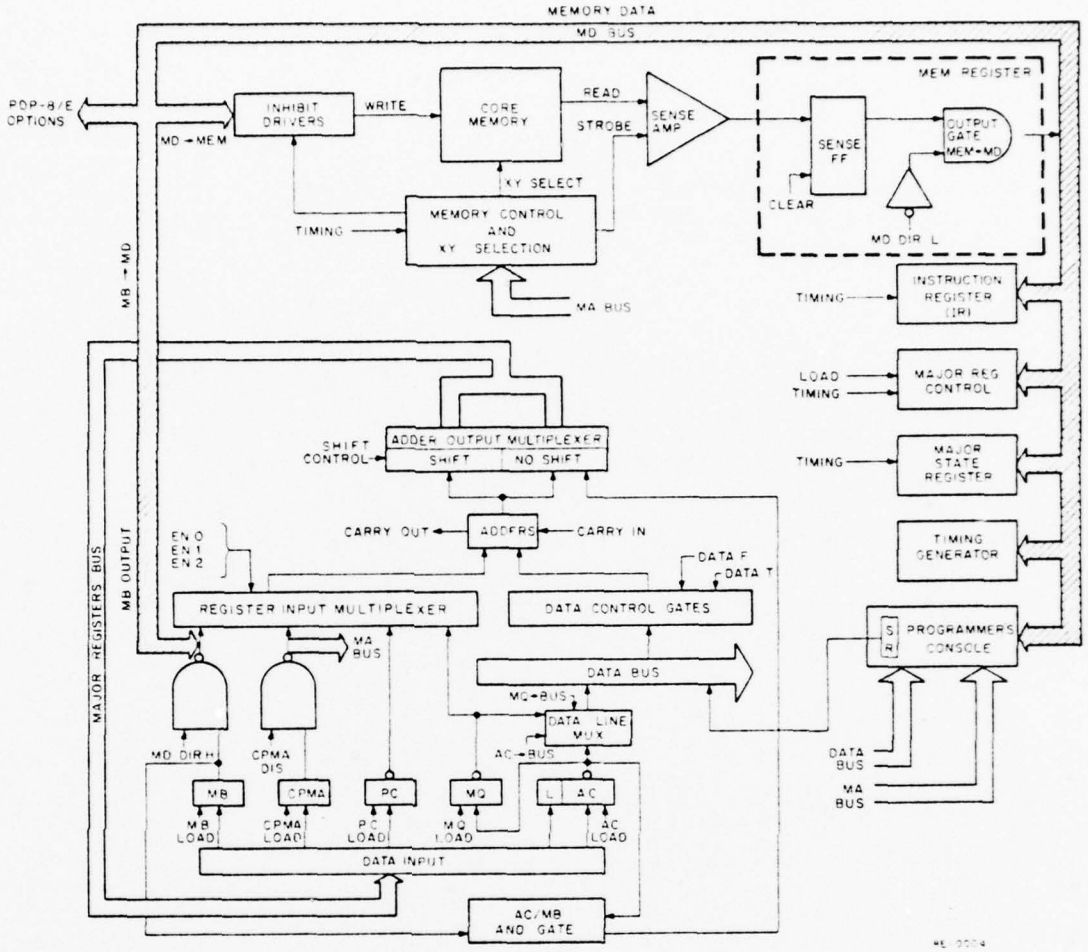


Figure 2b. Basic Data Paths.

3.0 PUBLICATIONS

"An Investigation of Glide - A Generalized Language for Interface Description and Evaluation," Andrew Nagle, M.S. Project Report, Carnegie-Mellon Universty, Electrical Engineering Department, August, 1976.

"Hardware/Software Tradeoffs in a Variable Word Width, Variable Queue Length Buffer Memory," A.C. Parker with A.W. Nagle, Proceedings of the 4th Annual Computer Architecture Symposium, March, 1977.

"Register Transfer Level Digital Design Automation: The Allocation Process," Louis Hafer and Alice Parker, Proceedings of the 15th Annual Design Automation Conference, June, 1978.

"The Application of a Hardware Descriptive Language for Design Automation," Alice Parker and Louis Hafer, Proceedings of the Third Jerusalem Conference on Information Technology, August, 1978.

"Data-Memory Allocation in the Distributed Logic Design Style," Louis Hafer, M.S. Project Report, Carnegie-Mellon Universty, Electrical Engineering Department, December, 1977.

"Automatic Design of Sequencers for the Control of Digital Hardware," Andrew Nagle, Thesis Proposal, Carnegie-Mellon University, Electrical Engineering Department, January, 1978.

"The Development of a Hardware Descriptive Language for Interfacing," Alice Parker, Andrew Nagle, and Bill Lyden, Carnegie-Mellon University, Electrical Engineering Department Technical Report, August, 1977.

"Digital Interface Description," Alice Parker, Proceedings, COMPCON, February, 1978.

"Description and Simulation of Microcode Execution," Alice Parker and Andrew Nagle, Proceedings of the 5th Annual Symposium on Computer Architecture, April, 1978.

"Structure and Function of a General Purpose Input/Output Processor," Alice Parker, Andrew Nagle, and James Gault, Carnegie-Mellon University, Electrical Engineering Department, unpublished paper, August, 1977.

"The Development of GLIDE: A Hardware Descriptive Language for Interfacing and I/O Port Specification," Alice Parker, Carnegie-Mellon University, Electrical Engineering Department, unpublished paper, August, 1978.

4.0 PERSONNEL SUPPORTED

Alice C. Parker, Principal Investigator

Daniel Siewiorek, Associate Investigator

Andrew Nagle, Research Assistant, MSEE, December, 1976

Louis Hafer, Research Assistant, MSEE, December, 1976

5.0 BIBLIOGRAPHY

(Barb77a) Barbacci, M.R., et. al, "Architecture Research Facility: ISP Descriptions, Simulation and Data Collection," Proceedings, 1977 National Computer Conference, Dallas, Texas, June 1977.

(Barb77b) Barbacci, M.R., Barnes, G.E., Cattell, R.G. and Siewiorek, D.P., "The ISPS Computer Description Language," technical report, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1977.

(DEC72) Digital Equipment Corporation, "PDP-8/E Maintenance Manual, vol. 1, no. DEC-8E-HR1B-D, 1972.

(Hafe78) Hafer, L.J. and Parker, A.C., "Register-Transfer Level Automatic Digital Design: The Allocation Process," Design Automation Conference Proceedings, vol. 15, 1978.

(Park78) Parker, A.C. and Hafer, L.J., "The Application of a Hardware Descriptive Language for Design Automation," Proceedings of the Third Jerusalem Conference on Information Technology, August 1978.

(Snow78a) Snow, E.A., Siewiorek, D.P. and Thomas, D.E., "A Technology-Relative Computer-Aided Design System: Abstract Representations, Transformations and Design Tradeoffs," Proceedings of the 15th Design Automation Conference, Las Vegas, Nevada, June 1978.

(Snow78b) Snow, E.A., "Automation of Module Set Independent Register-Transfer Level Design," Ph.D. dissertation, Electrical Engineering Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1978.

(Thom77a) Thomas, D.E., "The Design and Analysis of an Automated Design Style Selector," Ph.D. dissertation, Electrical Engineering Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1977.

(Thom77b) Thomas, D.E. and Siewiorek, D.P., "Measuring Designer Performance to Verify Design Automated Systems," Design Automation Conference Proceedings, vol. 14, pp. 411-418, 1977.

APPENDIX I

The Development of GLIDE: a Hardware Descriptive Language for Interfacing and I/O Port Specification

Alice Cline Parker
Department of Electrical Engineering
Carnegie-Mellon University

ABSTRACT

This paper presents GLIDE, a language designed for I/O Port specification. The language is based on the process construct and allows description of interfaces, I/O controllers, and buses.

Keywords: hardware descriptive language, register-transfer level bus specification, interfacing, input/output, process-oriented

This research is supported by the U.S. Army Research Office under grants #DAAG29-76-G-0224 and #DAAG29-78-G-0070.

I. INTRODUCTION = THE I/O SPECIFIATION PROBLEM

Buses, I/O transactions and interfaces can be grouped under the more general category of ports and port interconnections. Ports often have to be specified along a number of dimensions - reliability, performance and function as well as structure. In addition, the overall port design philosophy should be evident from the specification.

Methods for Bus Design Specification

There are several possible ways to specify port designs. Some of these verify that the design meets structural and functional constraints, and some also demonstrate that the port itself is capable of meeting performance requirements. Of course it is more difficult to demonstrate that reliability requirements have been met.

Standard Specification Techniques

Virtually all implemented ports are specified with:

- Block Diagrams of Interconnections
- Timing Diagrams
- Verbal Discussion
- Examples

Certainly, block diagrams convey some of the interconnection structure. Ideally, a block diagram should indicate the direction of the transfer on the interconnections, daisy chaining and basic functions of the interconnections. The IEEE-488 bus, shown in Figure

1, is an example of a block diagram specification.

Timing diagrams can be used to indicate protocol, timing dependencies, synchronization mechanisms, data transfer mechanisms, control of the interconnections, and priority allocation (to some extent). In addition, timing diagrams illustrate the potential port performance assuming the modules the ports are attached to can sustain the speeds. Timing diagrams should be drawn from the viewpoint of both communicating ports like the example in Figure 2. The timing diagrams can be supplemented with verbal discussion, and indication of critical timing dependencies.

Verbal discussion is most valuable when it is used to describe the design philosophy, and overall operation, and to supplement other descriptive techniques. Figure 3 contains an example portion of a bus description which presents a design philosophy.

Examples, complete with timing diagrams, are often used to clarify more difficult points. It is extremely important that the examples describe points already covered in the specification and not introduce new concepts. In addition, examples should be clearly specified as such and any bus functioning peculiar to that example should be so stated.

All of these techniques provide a viewpoint on the port design. These techniques must be combined with a more formal specification method for a complete description.

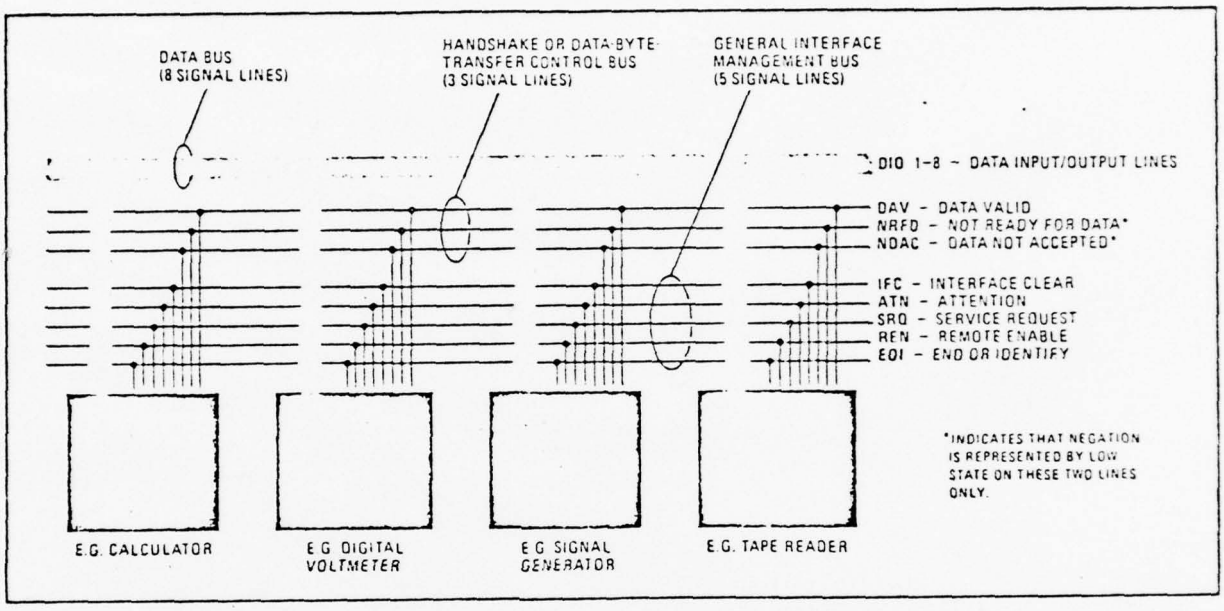


Figure 1. Example Block Diagram (Ricc74).

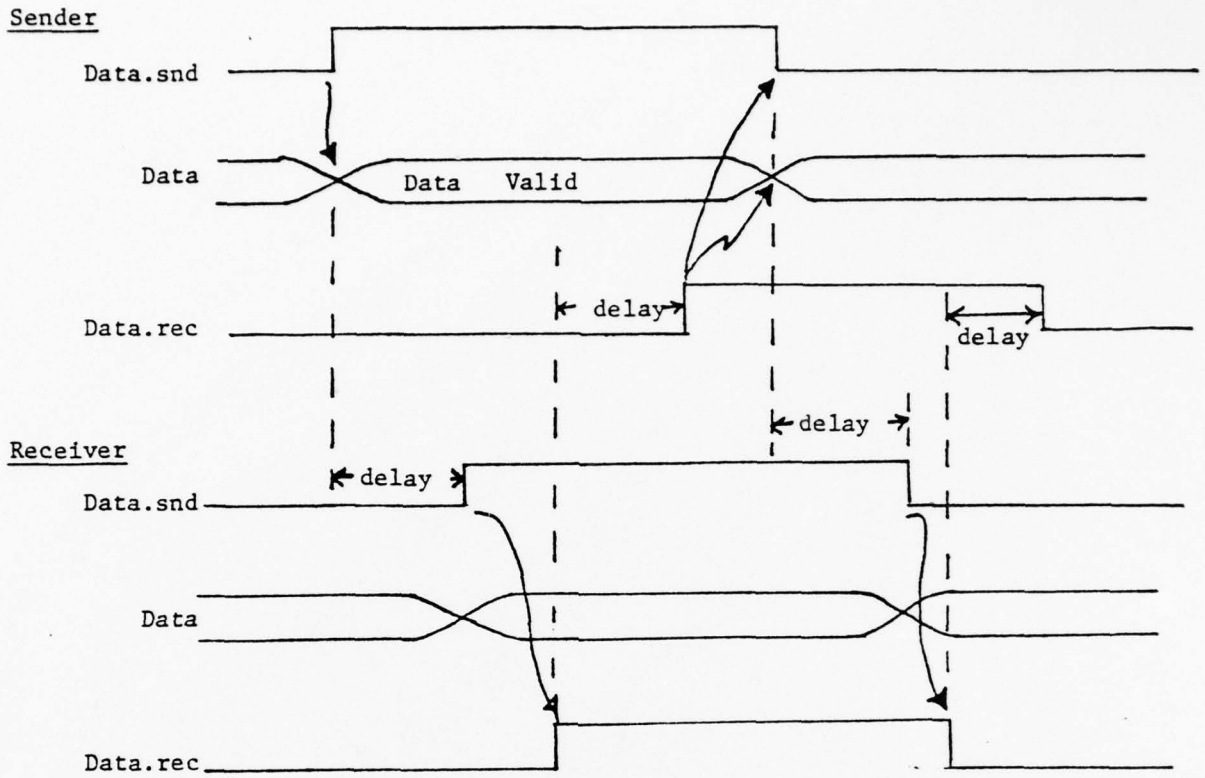


Figure 2. Timing Diagram for Two-Wire Protocol.

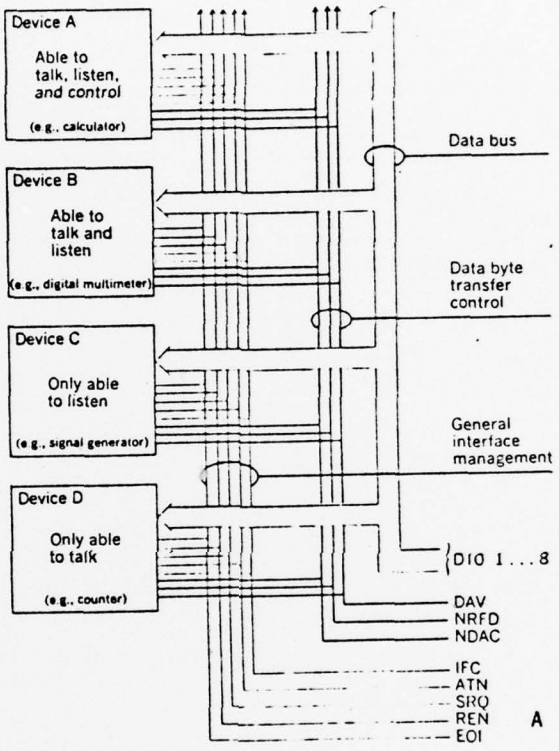


Figure 3. Example Interface Discussion (Loug74).

Formal Specification Techniques

Formal specifications of port and bus operation have not been applied for many practical designs. However, some techniques do exist. These are:

- The state diagram approach
- The flowchart approach
- The formal language approach

Flowcharts and state diagrams have been used to describe a number of buses - including the IEEE-488 bus. (See Figures 4 and 5). Languages have been slower to develop.

Bell and Newell (Bell71) laid the groundwork for interface description with their requirements for port description. In the interim, interface standards committees began struggling with the description problem. Curtis, working with the Purdue Workshop on Industrial Computer Systems, Data Transmission Interface and Committee, proposed IDS, an Interface Description System (Curt75). The system involved the use of the PMS notation, (from Bell and Newell) at the top level, and the use of a new language for description at the programming and register-transfer levels. In addition, the system was to cover other levels of description as well, but these were not defined at the time. The new language Curtis proposed was a version of ISP with features of AHPL and with necessary timing constructs added. (Most of which were being added to the ISPL version of ISP) (Siew74).

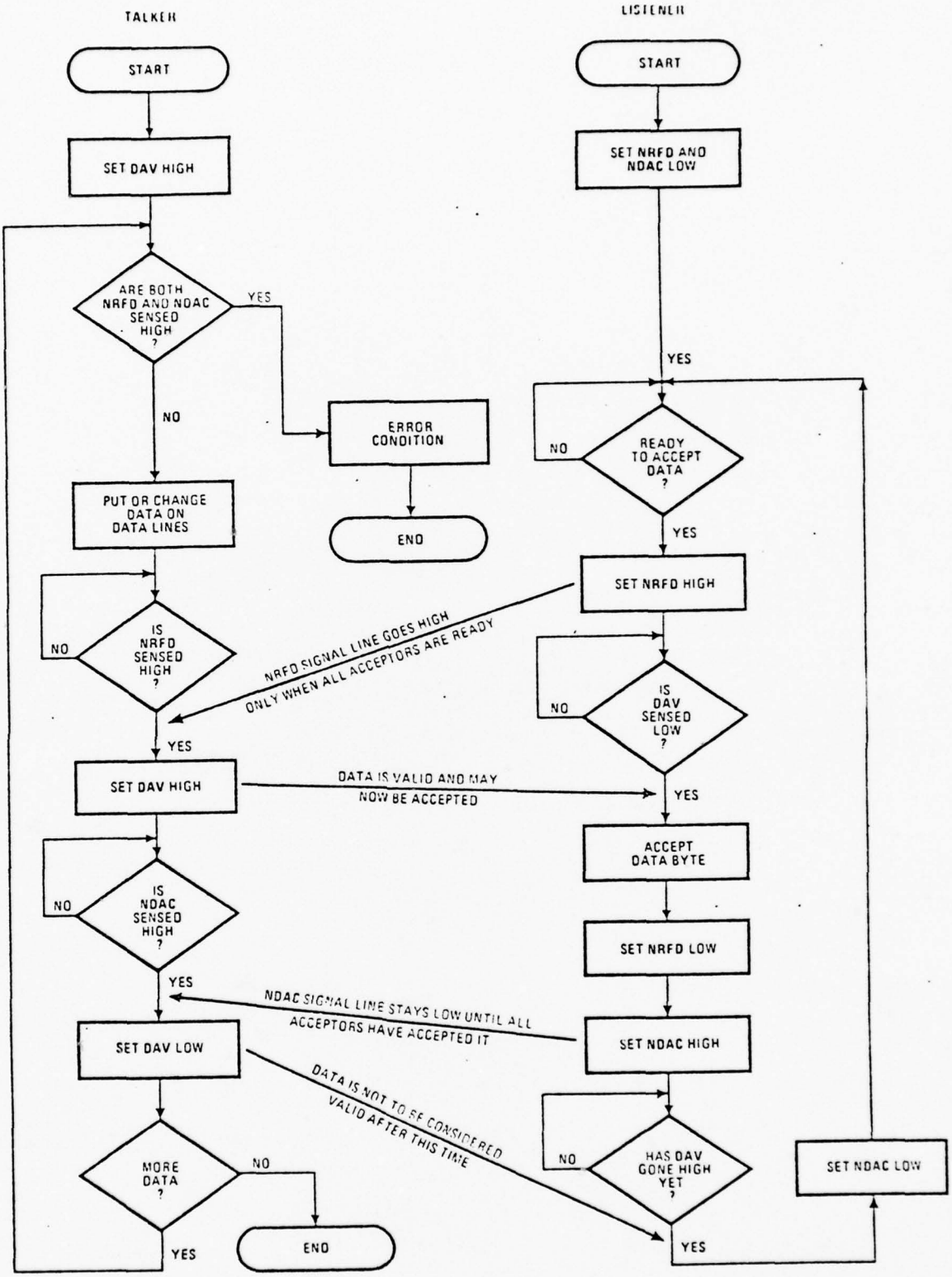


Figure 4. Example Bus Flowchart (Ricc74).

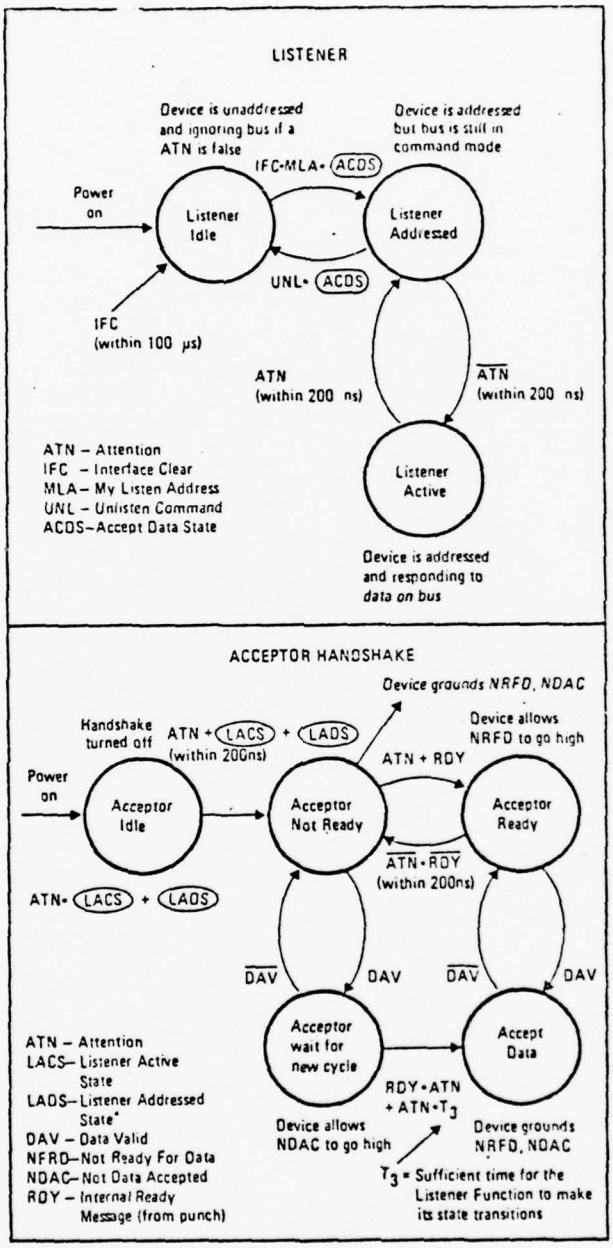
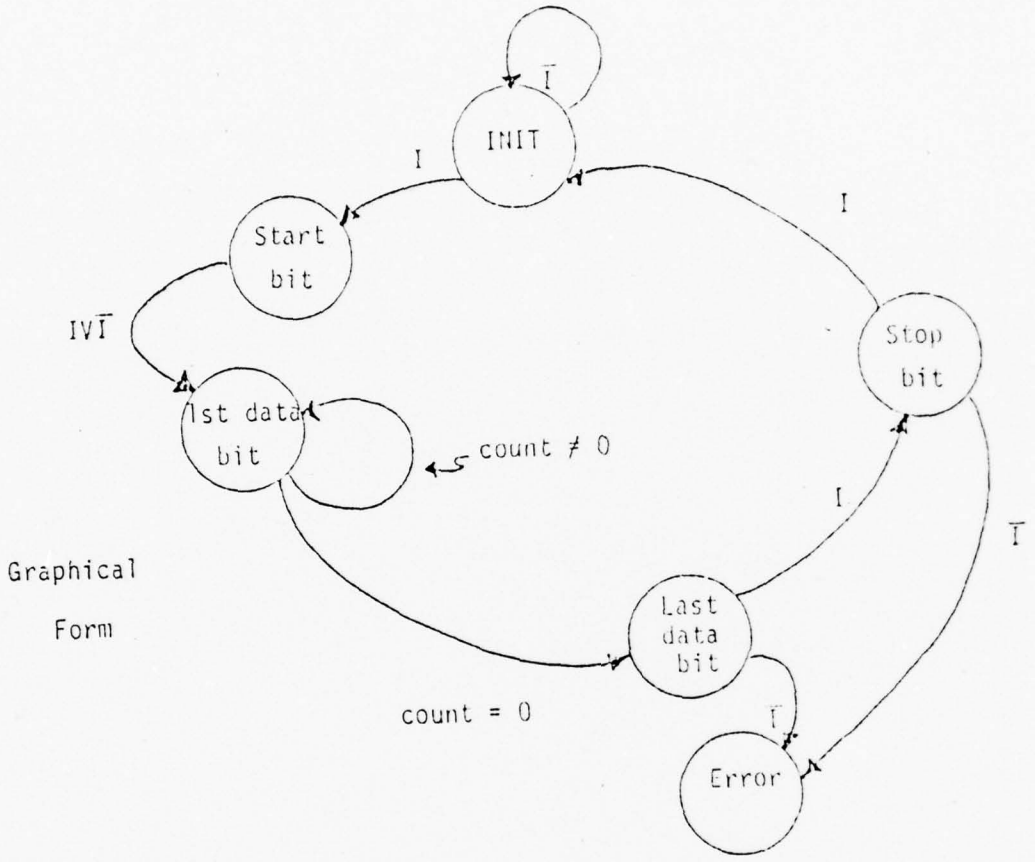


Figure 5. Example State Diagram (Ricc74).

At the same time Vissers had been developing a language which is essentially a formal description of the state diagrams describing interfacing functions, SDLC and the IEEE-488 interface bus have been described using this approach (Viss76, Knob75). The formalism of this language allows one important application - functional simulation. Vissers has intentionally restricted the coverage of the language to the gate and register transfer levels, with timing added. An example of the language is shown in Figure 6(b), which illustrates the states and transitions of an interface device which inputs an 8 bit character from an aysnchronous serial line. The transitions depend upon the input bits.



I: = Input bit

INIT: = Initial State

REG: = Input 1 bit register

T1: = Time between bits

COUNT:= Bit count

Figure 6(a). Interface Description Using the State Diagram Approach

Formal Language

```
V SERINPUT
[1] INIT: →  (I = 0) INIT
[2] COUNT ← = 8
[3] P1:  DELAY T1
[4] REG ←  I
[5] COUNT ←  COUNT + 1
[6] → (COUNT ≠ 0)/P1
[7] DELAY T1
[8] → (I = 0)/ERROR  (No stop bits)
[9] DELAY T1
[10] ⟶ (I = 0)/ERROR (Only one stop bit)
[11] ⟶ INIT
```

Figure 6(b). Interface Description Using the State Diagram Approach

There are advantages and disadvantages of each of these approaches. State diagrams convey state transitions well, but functions performed at each state sometimes are obscured. Even though conditions for state transition are specified, it is difficult to extrapolate from the diagrams to determine the actual order in which transitions occur. Finally, the diagrams become unwieldy for complex descriptions.

Flowcharts convey the actual functions well but have one fundamental limitation: they are inherently sequential. Parallelisms on the global level (separate control environments) must be described with separate flowcharts. Parallel branches must have special constructs created, and some things cannot be described at all. The classic example is a reset line which can stop port functioning at any point and reset all state variables.

State diagrams, flowcharts and formal languages all suffer from the forest-tree problem. Unless the writer of the description chooses labels carefully, documents, and structures his description, it is

difficult for the reader to ascertain the basic function being carried out. A paging algorithm for a cache memory, for example, must be deduced from the description and if the variables are labeled "A", "B", "C" instead of "Higher.bits.PC", "Lower.bits.PC", "Next.address" the meaning of the description is unclear. For any of the above techniques to be useful, a global flowchart or state diagram is necessary in order to understand overall I/O functioning. Finally, the primitives of the formal language must be chosen to reflect the primitive operations inherent in port and I/O operation.

II. THE GLIDE LANGUAGE

GLIDE - a Generalized Language for Interface Description and Evaluation - is a relatively undeveloped language. Yet, it is capable of addressing some issues left untouched by the existing languages. It is the only language that is designed for a subset of digital systems - interfaces and peripheral controllers; other languages have had a more general application. As such, GLIDE focuses on some problems germane to interfacing and I/O.

Some specific features of GLIDE are the following: Values can be specified as transitions in addition to level values; FIFO buffers can be declared and accessed with single statements; Parity checking and generation are primitive operations for GLIDE; Synchronous I/O and clocks can be declared, and timeout statements are available to the writer. Finally, independent control environments can be synchronously initiated under conditions declared in the program.

This section presents the origin of GLIDE, and section III is an introductory guide for the use of GLIDE. Discussions and conclusions in section IV focus on the applications of GLIDE.

Background - The Origin of GLIDE

When GLIDE (Park77a) was introduced, it was based on the premise that digital I/O and interfacing functions can be broken down into four primitive groups: data storage, data manipulation, control, and data input/output.

Given this conceptual partitioning, the goal of GLIDE is to provide a convenient descriptive language to specify the functions within the modules in a port, and between ports. For example, the writer can declare an external line or group of lines, and bind them to a direction like send, receive, or bi-directional, and to an interface technology. A separate declaration is provided for synchronous transfers. Data can be moved directly through the port or into an internal register for later manipulation. The control responds to conditions on the input lines, and sits idle when no startup conditions exist. In the same way, each of the other functional boxes shown above can be described. The combined result is a complete functional description of a specific port.

Since normally GLIDE descriptions are written to provide functional descriptions, as opposed to gate-level hardware descriptions, the operators provided in the language comprise a specialized set of operators designed to maintain this level of abstraction.* A typical GLIDE statement may not describe a single

---

*These operators are based on a set of primitive I/O functions:

1.	Data Input/Output	. Data paths and data transfer mechanisms . Electrical characteristics . Timing dependencies (including timeouts) . Synchronization mechanisms
2.	Control	. Control of the bus . Priority allocation . Protocol (including interrupts)
3.	Data Manipulation	. Error checking . Data formatting
4.	Data Storage	. Buffering and storage

register transfer, but rather a single digital function such as buffer store, parity test, or data encode. A conscious effort is also made to limit the language to ports and I/O. Therefore many functions peculiar to processors have not been included in the repertoire of GLIDE operators. On the other hand, a fairly rich set of special-purpose operators is provided to make description writing more convenient.

A second major result of the earlier work is the embodiment of the same set of primitive operators in a general purpose I/O processor (Pio) (Park77b). Specified at the register transfer level, this machine is designed to emulate any Pio, and it is microcoded in GLIDE. The machine is modularized, with each module capable of executing one of the primitive operations referred to above. Thus there are I/O modules, synchronous I/O modules, parity check and generation modules, and control modules, for example, all of which are under GLIDE program control.

The generalized Pio is capable of replacing an arbitrary number of Pios in a single-computer environment. This one machine could dynamically reconfigure itself to be a disk controller, teletype interface (more than one if desired), line printer interface, magnetic tape unit controller, etc. Its modularity allows its size to depend on the number and complexity of desired operations, and it could easily be extended to encompass any new device smply by writing a GLIDE description and entering that description into the machine, inserting additional hardware modules, if required. The total number

of interfaces which could be emulted simultaneously is limited only by the speed and bandwidth of the machine, which has not been determined.

III. A GUIDE FOR THE USE OF GLIDE

The Process Structure

The basic structure of a GLIDE description is the PROCESS, which is a description of an autonomous operating environment (asynchronous control environment) along with operations which take place there. A PROCESS consists of GLIDE STRUCTURE statements, which describe the environment of operations, followed by DATA and CONTROL SEQUENCING statements, which describe the actual operations. A PROCESS may also consist of local processes, which are started under certain specified conditions, and remain idle whenever those conditions are not met. The outermost process, or the outermost block, is the MAIN PROCESS and contains all subprocesses and subprocess initiation conditions necessary to describe the complete interface. An artifical example PROCESS structure is shown below in Figure 7.

Here, the name of the entire GLIDE description is A, and there are three subprocesses AA, AAAA and BB. Inside the MAIN PROCESS A, a pair of initiation staements sets up the conditions under which AA and BB are to be started. Any time after these statements have been executed, as long as A is still being executed and the initiation conditions for AA and/or BB have been met, either or both processes can be started. If a priority ordering between processes is desired, a priority for initiation can be specified in the initiation statement. Once a process is started, however, processes of higher priorities whose initiation conditions are met can suspend certain

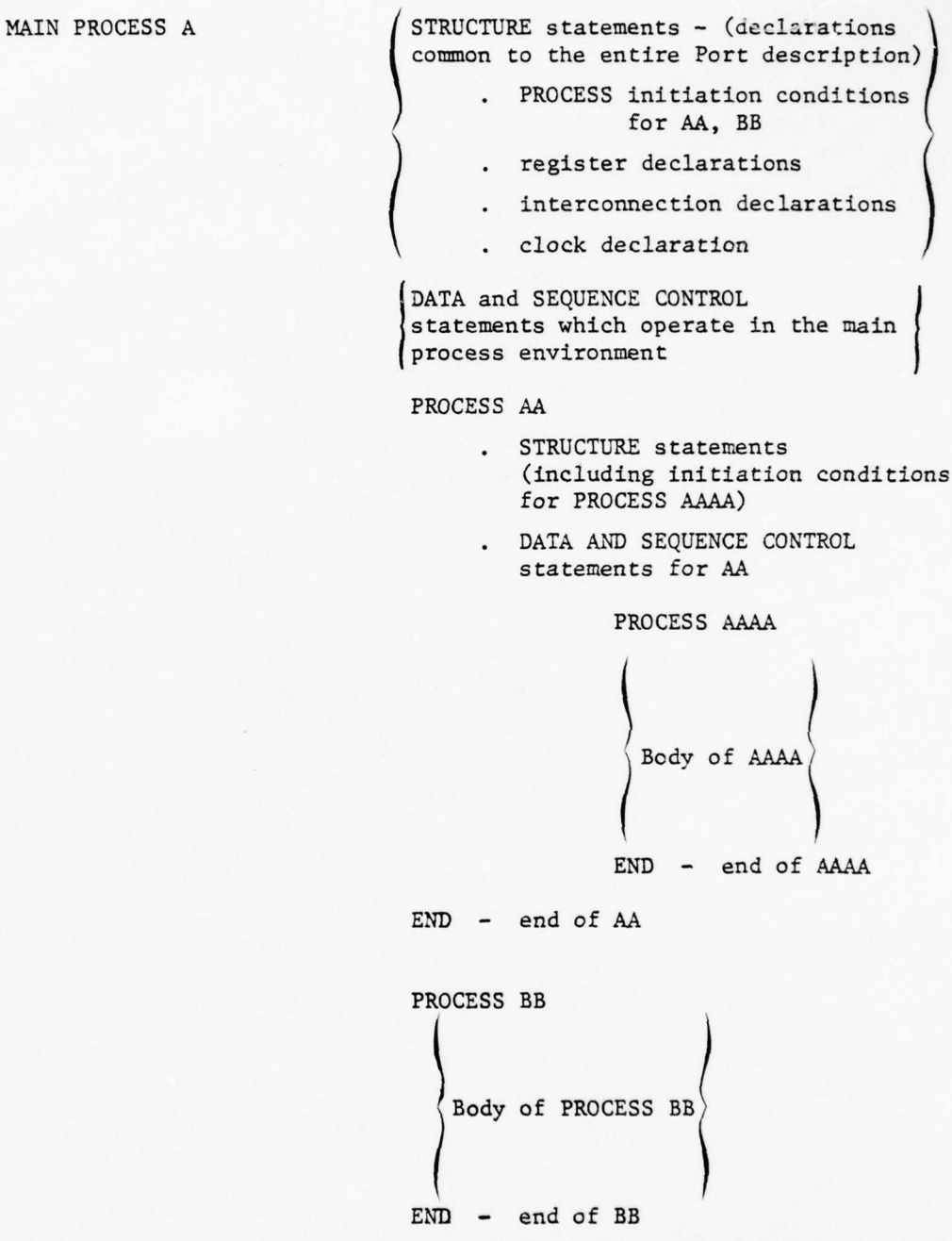


Figure 7. The GLIDE PROCESS Structure.

lower priority processes currently executing. Of course, the initiation conditions can be specified so that this does not occur, as will be shown later. Also, high priority processes nested inside lower priority processes will not suspend the lower priority processes when initiated. Process AA, if higher priority than BB, can suspend it. However, process AAAA cannot suspend ay other process since it is at the lowest level of nesting. (In fact, AAAA can be initiated only while AA is executing). The syntax of the initiation statement is illustrated with this example:

```
INIT AAAA:3,(ba AND bc AND bd) = 1
```

This sets up the initiation conditions for PROCESS AAAA, which has priority of level 3.

The process structure of GLIDE is designed to allow a GLIDE description to indicate overall port and bus operation to the reader. Proper initiation conditions and nesting of PROCESSES can provide an accurate representation of the actual hardware control environments. The relationship between GLIDE PROCESSES and the I/O structure is shown in Figure 8, based on our artificial example.

We now turn to a more complete discussion of the GLIDE language, using parts of a UNIBUS-like structure as an example.

GLIDE STRUCTURE STATEMENTS

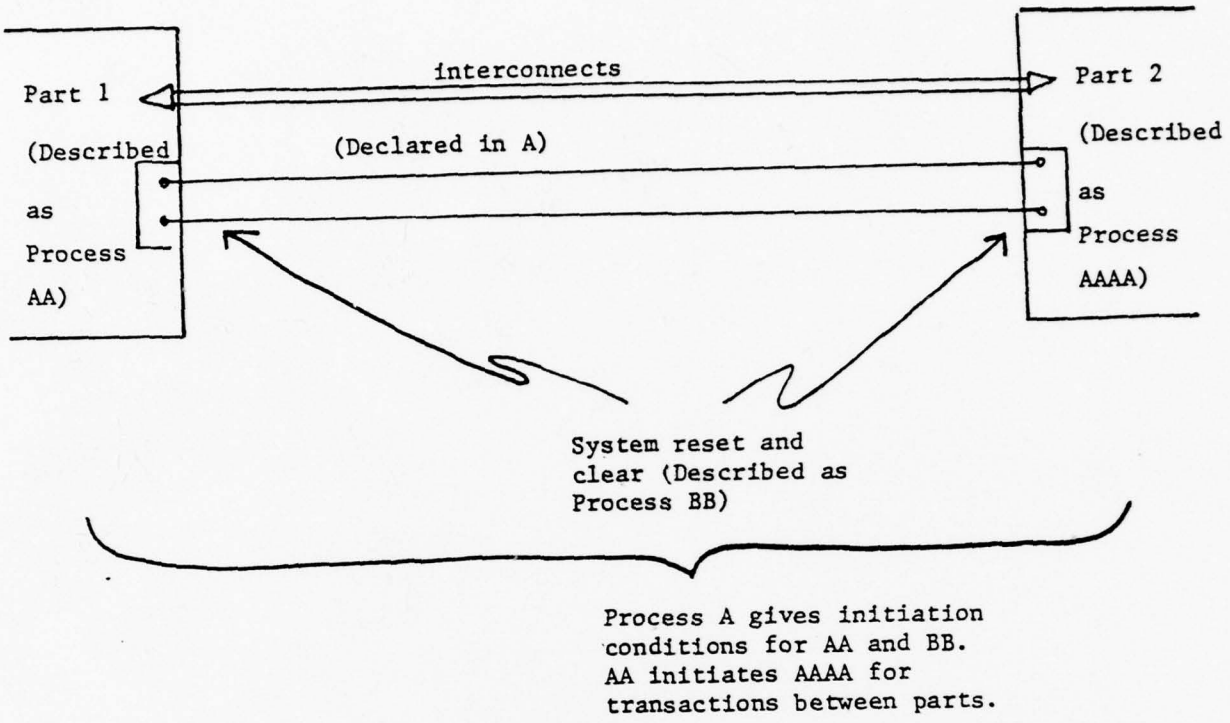


Figure 8. I/O Structure of the Artificial Example.

Besides the PROCESS initiation statement, there are a set of statements used to declare aspects of the port and interconnect structures. The first of these to be discussed is the Declaration Statement.

Internal port registers and port interconnections must be declared. The syntax of the declaration statement is:

<hardware specification>\<name>,<name>,...,<name>;

where <hardware specification> defines the logic type and function of the declared hardware using the following abbreviations:

Basic logic type:

ECL	emitter-coupled logic
TTL	transistor-transistor logic
CMOS	Complementary MOS
MOS	Metal oxide semiconductor
TRANS	transformer coupled

Function:

CRL	Current Loop
HIR	High impedance receivers
HLT	High level transmitters
OC	Open collector gates
SCH	Schmitt trigger
TRI	Tri-state
INT	Internal register
DIFSND	Differential sender
SND	Sender
REC	Receiver
BI	Bi-directional
DIFREC	Differential receiver
DIFBI	Differential bi-directional

<name>

is an identifier, the first four characters of which must be unique. The declarations may also have length and width attached, as in ISPL and ISPS (Barb78). Some examples of the declaration statement are:

TTL OC\data.bus<7:0>,address.bus<7:0>;

This statement describes two 8-line TTL open collector bus segments, one named data.bus, and one named address.bus.

ECL INT\count.register<7:0>;

This statement describes a single 8-bit ECL logic register which is internal to the interface.

A function designation is mandatory for every declaration, but a logic type is optional. Every assignment bearing an INT functional designation is considered to be a register, and all others are external lines.

A second STRUCTURE statement is the INIT (initiate) Statement previously discussed. Its syntax is:

INIT <process name>:<priority>,<variable> = <value>;

<process name>

is the name of a process which performs a particular interface task.

<variable>

is the name of the variable which initiates a process. It may also be a Boolean and/or arithmetic expression involving any previously declared lines, or a compound.

<value>

is the binary or octal value of the variable which is to initiate the process.

/ is a 0 to 1 transition of a single bit

\ is a 1 to 0 transition of a single bit

<priority>

is the interrupt priority of the process.

Default is to the lowest priority declared.

EXAMPLE:

INIT teletype;1,status=/

Process teletype is initiated when a line called "status" by the programmer goes from 0 to 1 and the teletype process has priority level 1 (level 0 is highest priority). In the UNIBUS description, the structure is based on a bus arbitration mechanism, as shown in Figure 9. The daisy chain processes in the first level of nesting allow bus request and grant signals to travel properly up and down the UNIBUS. The arbitration process, also in the first level, has nested within itself the bus granting processes. The process structure (Figure 10) should clarify the configuration described.

FIFO queues can be declared with the BFR (buffer statement), with this syntax:

BFR <name>[length]"<"<width>">";

<name>

is the name to be assigned to the buffer for later program use.

[length]

is the number of words in the buffer queue.

<width>

is the bits in each word.

EXAMPLE:

BFR DISKQUEUE[4096]<32>

This statement establishes a buffer of length 4096 words, each word 32 bits in length.

In the UNIBUS description, there are two declaration types. For example:

OC\MSYN<0>; describes a single bit line named "MSYN." The OC indicates open collector.

INT\adatareg<15:0>; defines a 16 bit internal data register named "adatareg."

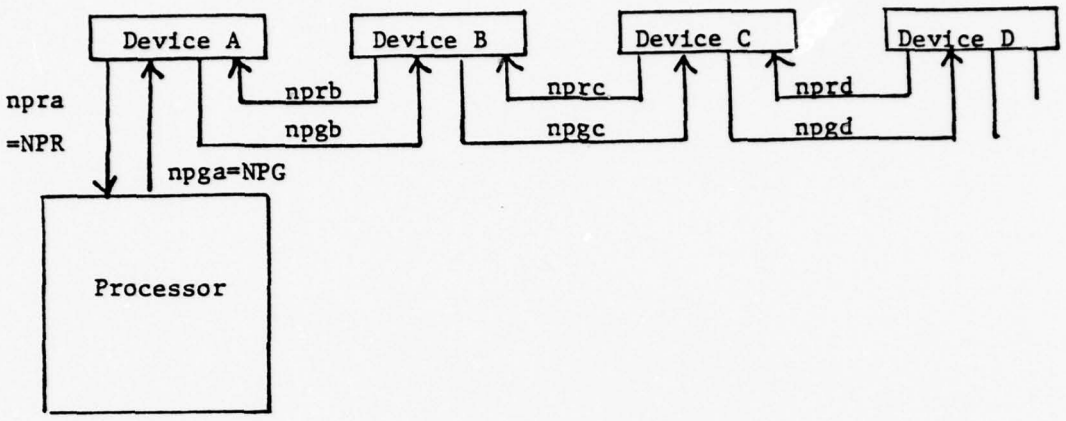


Figure 9. A Block Diagram illustrating Naming Convertions for the Bus Grant/Request Lines of the Arbitration Mechanism.

```
MAIN PROCESS UNIBUS

    Declarations
    Compound Declarations

  INIT arbit:0, SACK=\;
  INIT ndaisychain:0, (npra OR nprb OR nprc
                       OR nprd)=1
  INIT 6daisychain:0, (br6a OR br6b OR br6c
                       OR br6d)=1
  {also INITS for priority levels 4,5,7}

  PROCESS ndaisychain
    .
    .
    .
  END

  PROCESS 6daisychain
    .
    .
    .
  END
  {PROCESSES 5daisychain, 4daisychain, 7daisychain}

  PROCESS arbit

        INIT nprgrant:1,NPR=1;
        INIT BR4grant:5,(bg.enable AND BR4)=1;
        INIT BR5grant:4,(bg.enable AND BR5)=1;
        INIT BR6grant:3,(bg.enable AND BR6)=1;
        INIT BR7grant:2,(bg.enable AND BR7)=1;
          .
          .
          .
  END
    .
    .
    .
END
```

Figure 10. Process structure of UNIBUS Bus Arbitration Mechanism.

Code conversion is often done by table lookup. (Of course, hardwired logic is also used for this purpose and can be described in GLIDE with DATA CONTROL statements to be introduced later). GLIDE has special primitives for table lookup, after the table has been declared with the table instruction. The syntax for this is:

TABL <name>[length]"<"<width1>"><"<width2>">"

which is used to set up a table for code conversion when conversion must be done by table lookup.

<width1> and <width2> refer to the bit widths of the input and output words respectively.

<name> and <length> are the same as the Buffer statement. Table entries follow immediately after the table declaration, using this format:

<encoded value>=><decoded value>

<encoded value> is an input code table entry.

<decoded value> is an output code table entry.

An example of this is:

```
TABL grey [4]<2><2>
00=>00
01=>01
10=>11
11=>10;
```

Description of synchronous I/O is difficult because of the different implementations of hardware which perform synchronization. GLIDE

allows only a limited description of synchronous I/O. Synchronous lines are declared with the synchronize statement, with syntax:

SYNC <name>"<"<length>">"@<time>

This describes the synchronization mechanism for inputting and outputting synchronous data from magnetic tape, magnetic disk, and synchronous data links.

<name>
is the variable name of the synchronous line(s).

<time>
is the cycle time in clock periods.

<length>
is the length of the word shifted in parallel during each synchronous cycle.

EXAMPLES: SYNC tape1<8:0>@10000

describes the synchronous transfer of 9 bits in parallel onto/off a bus called tape1 at a frequency of 10 usec.

SYNC disk<>@9000

describes the synchronous shifting of data onto a single line called disk, at a 900ns rate.

The use of synchronous variables in a GLIDE description will be described later.

```
!The following declarations are compound expressions.  This means
 that there exists logic to continually evaluate the following
 expressions!

dvaddress:=a<0> AND a<1> AND (NOT a<2>);
dati:=(NOT C0) AND (NOT C1));
pdati:=(NOT C1) AND C0;
dato:=(NOT C0) AND C1;
bdato:=C0 AND C1;
p.error:=(NOT PA) AND PB;
no.error:=(NOT PA) AND (NOT PB);
```

Figure 11. Compound Expressions From the Bus Example.

In order to declare snchronous lines, however, a clock must also be declared. The clock statement allows description of a simple clock. The syntax is:

CLK <period>;

where <period> is expressed in nanoseconds. The default clock period is one nanosecond. One extremely powerful statement used in conjunction with Declarations is the Compound Declaration. This instruction has the form:

<name>:=<expression>,
<name>:=<expression>, ... ;

<name> is an identifier and <expression> is an arithmetic and/or logical expression using previously declared variables. It is continuously evaluated and is useful for describing combinational logic like address recognition hardware.

Figure 11 contains the Compound Declarations found in the UNIBUS description, dvaaddress is the address of device A, dati, pdati, dato and bdato refer to the modes of data transfer (dati, datip, dato, datob) on the bus. PABB indicates parity error, and PABB indicates no error.

All of these declaration statements precede the Data Control and SEQUENCE CONTROL statements in the MAIN PROCESS and the subprocesses.

SEQUENCE CONTROL Statements

Many of the Sequence Control Statements are conventional, but a few have features not found in other HDLs. These less-conventional instructions will be described first.

The most powerful of these is the Delay Until Statement:

DLAY <range> UNTL <variable>=[<variable> <value>]
THEN <branch> ELSE <error>;

This statement delays the process execution for a specific time or until a variable reaches a given value or equals another variable.

<variable>

is a variable named in a structure statement.

<value>

is a value expressed in binary, octal or decimal.

<branch>

is a program label branched to if equality condition is met in the given time period.

<error>

is a program label branched to if equality condition is not met in the given time period.

<range>

is a range of values of time the delay is to last.

Here are some simple uses of the DLAY statement: DLAY 4000;
This statement delays the interface 4 usec (assuming a 1ns clock).

DLAY UNTL check=001;
This statement delays the interface until the variable check=001, and then continues program sequence.

DLAY 5000 UNTL instr=code1 THEN read ELSE write;
This statement delays the interface 5usec or until the variables instr and code1 are equal, whichever comes first. It branches to "read" for equality and "write" if time runs out.

This statement is useful for describing skew of lines and timeout conditions. The range of delay times is useful when specifying a potential bus design where variable length delays or timeouts need to be specified. The Delay Statement is used widely in the example bus description. For example, inside the 6daisychain PROCESS, DLAY UNTL BG6=/, implies a suspension of PROCESS execution until BG6 has an upward transition (symbolized by the forward slash "/"). Inside the NPR grant mechanism DLAY75; is a 75 nonosecond delay to allow for signal skew. Then, DLAY 7500+-2500 UNTL SACK=1; indicates that the PROCESS suspends waiting for SACK to have a positive transition but only for 5-10 microseconds, depending on the bus implementation.

Another useful statement is the Parallel Branch Statement, used to allow execution of multiple instructions at one time. This statement is paired with the MRG (merge) statements at the end of each paralel branch.

PBR <label>,<label>,...,<label>:[FRUN/SYNC];

This statement performs the following sections of code referred to by labels, either statement by statement synchronously or on a free-run basis.

FRUN

signifies free run execution. Statements in each code segment are executed as soon as previous operations are completed.

SYNC

indicates that statement executions in each code segment are to be synchronized. In other words, the nth statements in all code segments are executed simultaneously.

Examples:

```
          PBR input,convert:FRUN;
input     . . .
          . . .
          MRG;
convert   . . .
          . . .
          MRG;
```

This set of statements causes the section of code labeled input to be executed simultaneously with the section of code labeled convert. MRG indicates a merge at the end of each segment.

```
          PBR output,check:SYNC;
output    statement1;
          statement2;
          MRG;
check     statement1;
          statement2;
          MRG;
```

This set of statements causes statements 1 to be executed simultaneously and statements 2 to be executed simultaneously.

Two other interesting sequence control structures, WAIT and an error branch associated with buffer accesses will be discussed later with their associated Data Control Statements.

The other statements used to control program flow are like those found in conventional microprogramming languages. These are:

- Compare statement

  CMP <source1>:<source2>^=> <branch1>,=> <branch2>;

  Compares values of two variables or a variable and an integer value (second source field).

  ^=><branch1>

  implies branch for a not equal condition, while

  => <branch2>

  means branch for an equal condition.

  Here is a simple example:

  CMP check1:23^=> error,=> ok;

- Call subroutine statement

  CALL <subroutine label>;

  The next statement executed will be that labeled. Control is returned to the statement following the call statement

when a RETURN command is encountered in the subroutine.

- End subroutine statment

RETURN;

Ends a subroutine and returns control to the calling site.

END;

Terminates a process description.

- No Operation Statement

NOP; is used primarily when two segments of code are executed in parallel and are synchronized.

- Serial Branch Statement

BR <label>; is an absolute branch.

- Loop Statement

LOOP<statement label>,<repetitions>;

This statement executes all statements through the labeled statement the specific number of times, an integer value.

Example:

```
          LOOP done,3;
          <statement 1>;
done:     <statement n>;
```

This executes statements 1 through n three times.

- Run Until Statement

RUN <label>UNTL<variable>=(<variable>/<value>)
THEN <branch>;

This statement is similar to the loop statement but terminates when a particular condition is reached, then branches out of the program. Testing is done at the start of each loop. If THEN <branch> is omitted, execution picks up at the first statement after the loop.

Example:

```
          RUN proc1 UNTL var3=var4 THEN interrupt;
          .  .  .
          .  .  .
Proc1:    .  .  .     End of loop
          .  .  .
interrupt:            New code segment begins here.
```

- Set Statement

Set <label>;

This sets an interrupt line from the interface. The interrupt line may be any declared external variable.

- Reset Statement

RSET <label>;

This resets an interrupt line.

The example bus description uses the compare statement in the daisy chain processes. CMP br6a:1 => done; causes control to go to the "done" label. If not, control flow continues to the next statement.

Control normally flows from statement to statement, with the delimeter between statements being a semicolon. If two statements are to be executed in the same clock period, however, the ; delimeter can be replaced with ALSO. For example,

```
A← 1 ALSO B← 1;
C← 0;
```

causes A and B to be set to 1 then C to be set to 0.

THE DATA CONTROL INSTRUCTIONS

With one exception, the DATA CONTROL statements all have the same structure:

<destination><-<source expression>

<destination>
is any name bound to a hardware structure (i.e., external lines, internal registers, buffers, synchronous lines, etc.).

<source expression>
is a variable, value, compound, arithmetic and/or logical expression or an expression using the operators described below.

Arithmetic operators:

+	addition
-	subtraction
*	multiplication
/	division

Logical operators:

AND	logical and of both operands
OR	logical or of both operands
XOR	logical exlusive or of two operands
NOT	logically complements source and places in destination

The logical operators are used widely in the COMPOUND, DLAY and INIT statements. For an example see Figure 11 which contains compound expressions from the bus example. The INIT statement uses logical operators widely, as in this statement from the UNIBUS description:

```
INIT 6daisy chain:0,(br6a OR br6b OR br6c OR br6d)=1;
```

Of course, the simple source-destination transfer is widely used in the example. BR6←1; is one instance of this.

There are seven primitive I/O operations. The first two are Encode and Decode.

ENC[<table>];
Encode from binary using the table specified. This operator returns 0 if the value canot be encoded. Choices of tables are UNARY, GREY, HAMMING and DISTANCE 3.

DCO [<table>];
Decode into binary using the table specified, returning 0 if not decodable.

It should be noted here that other encoding/decoding problems have to be declared explicitly usng the TABL declaration. Here is an example of both methods:

```
TABL grey[4]<2><2>
          00   00
          01   01
          10   11
          11   10;
```

This table is accessed by the statement:

<destination> <table name><source>;

or for this instance:

(1) A<1:0><-grey'10;

and the value in A = 11 at the end of execution.

(2) A<1:0><-ENC[grey]'10;

is an equivalent statement. The inverse of this statement is:

A<1:0><-DEC[grey]'11;

and the value returned is 10.

Besides the encoding data function, formatting, packing and error checking, are important functions performed by ports and interfaces. Formatting can be as simple as concatenation or as complex as the intermixing of two operands in a random pattern. The Format Statement (FMT) provides a general mechanism for this.

FMT <source1> [<pattern>] <source2>;

where <source1> and <source2> are operands or values and <pattern> is a pattern of * and /. An * indicates a bit taken from source1 and a / indicates a bit from source2. The absence of a pattern implies concatenation. For example:

a<7:0><-FMT '0000[*/*/*/*/]'1111;

returns the value '01010101. (' indicates a binary value).

Packing and unpacking bit slices from words is less complex, and the Pack (PAC) and Unpack (UNPAC) statements provides for this. PAC(<variable>) packs a bit slice into the position in the word

specified by the <variable>. UNPAC(<variable>) returns the slice from the word. For instance:

A<3:0>← PAC(1)B<1:0>;

places the B bits into the lowest order (1 and 0) bits of A.

A<3:0>← UNPAC(2)B<15:0>;

places values from B<7:4> into A.

Parity but generation and checking is done with a set of statements, the Test and Parity satements.

PARE <source> generates an even parity bit and PARO <source> generates an odd parity bit. TSTE <source> and TSTO <source> perform a parity check and return a 1 if parity is valid, a 0 if there is a parity error.

The last two statements to be presented concern synchronous data transfer and buffer accesses. Since the variables permitted in GLDE statements may be one of several different types, some comments concerning the use of each type should make their intent clearer.

Use of a variable declared in a BFR statement carrier with it an implicit buffer memory access. The total number of items stored in the buffer may not exceed the declared size. If it does, an error signal will be generated, and the most recent store will not be completed. The error will be detected if a statement label is attached to the buffer name in the execution statement with a ":". For example, in the statement FIFO:bfrful←A<7:0>; if FIFO is a declared buffer memory and execution of this statement causes it to

overflow, the next statement to be executed will be "bfrful." The default error branch also applies to buffer accesses when an empty condition occurs. A←FIFO;Bfrepty;

Use of a variable declared in a SYNC statement carries with it an implicit synchronous transfer of the data. For a parallel tranfer, the word length of both source and destination must match the synchronous variable length. For serial transfer, the synchronous variable is always of length 1, so the other variable may be of any length greater than or equal to 1. For the parallel case, the meaning of the transfer is clear; for the serial case the transfer means that a shift takes place at the frequency specified (with no extra delay when the shift register is loaded with a new word or emptied). If a delay is desired, it may be achieved by using the DLAY statement.

Immediately after a synchronous variable is accessed/stored into a shift register, a WAIT statement must be used to allow time for the data to be synchronously input. Here is a brief example.

In the declarations we find:

```
SYNC disk <0> @ 200;
```

which means we have a synchronous line caled disk which sustains a data rate of 1 bit every 200 s.

In the body we find:

```
LOOP done, 32;
A<31> ← disk;
A ← A/2;   !shift right A
```

```
done: WAIT;
```

which fills A off the disk.

IV. CONCLUSIONS AND DISCUSSION

GLIDE has grown from a microprogramming language for a general-purpose I/O processor to a descriptive language for I/O, interfaces and buses. While no apologies are made for the syntactical artifacts of the micro-language, we should stress that GLIDE is evolving and continually changing. It has proved itself adequate for the UNIBUS descriptive task, a non-trivial problem. A compiler exists, and a simulator is being written.

The simulator will allow research to proceed in studying bus structures, I/O, and multiprocessor communications. It will also provide a tool for teaching the above in an interactive fashion.

ACKNOWLEDGEMENTS

The author would like to acknowledge the assistance of two people whose efforts made this paper possible: Bill Lyden, who wrote the original GLIDE compiler, and Andy Nagle, who labored over the language itself.

REFERENCES

(Bell71) Bell, G. Gordon and Newell, Allan, Computer Structures: Readings and Examples, McGraw Hill, 1971.

(Curt71) Curtis, Richard, "IDS, An Interface Description System," unpublished document, ALCOA, November, 1975.

(Knob75) Knoblock, D.E., Loughry, D.D. and Vissers, C.A., "Insight Into Interfacing," IEEE Spectrum, Volume 12, Number 5, May, 1975, pp. 50-57.

(Loug74) Loughry, Donald, "What Makes a Good Interface," IEEE Spectrum, November, 1974.

(Park77a) Parker, A., Nagle, A. and Lyden, W., "The Development of a Hardware Descriptive Language for Interfacing," Department of Electrical Engineering Technical Report, Carnegie-Mellon University, August, 1977.

(Park77b) Parker, Alice, Nagle, Andrew and Gault, James, "Structure and Function of a General Purpose Input/Output Processor," unpublised paper, Department of Electrical Engineering, Carnegie-Mellon Universty, August, 1977.

(Ricc74) Ricci, David and Nelson, Gerald, "Standard Instrument Interface Simplifies System Desgn," Electronics, November 14, 1974.

(Siew74) Siewiorek, Daniel, "Introducing ISP," Computer, Volume 7,

Number 12, December 1974, pp. 39-41.

(Viss76) Vissers, C.A., "Interface, A Dispersed Architecture," Proceedings of Third Annual Symposium on Computer Architecture, 1976, pp. 98-104.

```
!This is a partial description of the UNIBUS operation.  In order to
 make the description complete, the daisy chain bus request and
 grant signals are included, but only specified for four devices for
 each level of priority, devices A,B,C, and D, where A is electrically
 closest to the processor.

 A GLIDE description contains a main PROCESS and as many nested sub-
processes  as possible to describe a set of interfaces connected together.
 In each PROCESS, including the main one, there are declarations
 which define the structure of the interfaces, and sequence statements
 which describe the operations.!

MAIN PROCESS UNIBUS

!Here we have declared the main process!

!Signal lines!

OC\a<17:0>;    !address lines,open collector!
OC\D<15:0>;    !Data lines, open collector!
OC\C0<0>;    !control bit!
OC\C1<1>;    !control bit!
OC\MSYN<0>;    !master sync line!
OC\SSYN<0>;    !slave sync line!
OC\PA<0>;    !parity line!
OC\PB<0>;    !parity line!
OC\INTR<0>;    !interrupt line!
OC\BR4<0>;    !bus request line, priority level 4!
OC\BR5<0>;    !bus request line, priority level 5!
OC\BR6<0>;
OC\BR7<0>;
OC\BG4<0>;    !bus.grant line, priority level 4!
OC\BG5<0>;
OC\BG6<0>;
OC\BG7<0>;
OC\NPR<0>;    !non processor request line!
OC\NPG<0>;    !non processor grant line!
OC\SACK<0>;    !selection acknowledge!
OC\BBSY<0>;    !bus busy!
OC\bg.enable<0>;  !bus grant enable (set/reset by processor)!
INT\interrupt.vector<15:0>;  !interrupt vector internal to device!
INT\slave.address<15:0>;     !slave address used by device B for NPR!

!The following lines make up the daisy chain bus request and grant
 mechanisms.  Each part of the chain is specified as a separate
 line for ease of description!
OC\br6a<0>;  !line from the devA to the grant mechanism!
OC\br6b<0>;  !line from the devB to devA!
OC\br6c<0>;  !line from devC to devB!
OC\br6d<0>;  !line from devD to devC!
OC\bg6a<0>;    !grant line from grant mechanism to devA!
OC\bg6b<0>;    !grant line from devA to devB!
OC\bg6c<0>;
OC\bg6d<0>;
OC\npra<0>;  !The same for nonprocessor request and grant lines!
```

```
OC\nprb<0>;
OC\nprc<0>;
OC\nprd<0>;
OC\npga<0>;
OC\npgb<0>;
OC\npgc<0>;
OC\npgd<0>;
INT\adatareg<15:0>;  !data register in device A!
INT\bdatareg<15:0>;  !data register in device B!

!The following declarations are compound expressions.  This means
 that there exists logic to continually evaluate the following
 expressions!

dvaddress:=a<0> AND a<1> AND (NOT a<2>);
dati:=(NOT C0) AND (NOT C1));
pdati:=(NOT C1) AND C0;
dato:=(NOT C0) AND C1;
bdato:=C0 AND C1;
p.error:=(NOT PA) AND PB;
no.error:=(NOT PA) AND (NOT PB);

!Now come a series of initiation conditions for various
 "processes".  These are declared now, but the processes are
 initiated whenever the conditions become true, and the process
 has higher priority than any other process which also has met
 its initiation conditions!

INIT arbit:0,SACK=\;

!the above statement initiates the arbit process with 0 priority,
 whenever the SACK line has a negative transition!

INIT dvaslave:0,(BBSY=/ AND DVADDRESS) = 1;

!Device A is initiated as a slave when bus busy is asserted high,
 and the device address has been decoded!

INIT 6daisychain:0,(br6a OR br6b OR br6c OR br6d)=1;

!6daisy chain is the daisy chain mechanism for priority level 6!

INIT ndaisychain:0,(npra OR nprb OR nprc OR nprd)=1;

!this is the daisy chain mechanism for the NPR requests!
!There are similar processes for the other priority levels!

        PROCESS 6daisychain
        BR6←/;  !Send the bus request through!
        DLAY UNTL BG6=/;  !Wait until the grant signal comes back!
        bg6a←/;  !device A, closest to the processor, gets the grant!
        CMP br6a:1 => done;
        !compare bus request from dev. A to 1. If equal, then you are
         done.  Otherwise, pass it on!
        bg6b←/;  .
```

```
        CMP br6b:1 => done;
        bg6c←/;
        CMP br6c:1 => done;
        bg6d←/;
  done: end;  !end 6daisy chain!

        PROCESS ndaisychain
        NPR ← /;
        DLAY UNTL NPG = /;
        npga ← /;
        CMP npra:1 => quit;
        npgb ← /;
        CMP nprb:1 => quit;
        npgc ← /;
        CMP nprc:1 => quit;
        npgd ← /;
  quit: end;  !end daisy chain for NPR!
        PROCESS arbit
        INIT npr.grant:1,NPR=/;
        INIT BR4.grant:5,(bg.enable AND BR4)=1;
        INIT BR5.grant:4,(bg.enable AND BR5)=1;
        INIT BR6.grant:3,(bg.enable AND BR6)=1;
        INIT BR7.grant:2,(bg.enable AND BR7)=1;
!These grant mechanisms are initiated in a priority order!
!For simplicity,only the priority level 4 and NPR levels will
 be shown!
DLAY UNTL (NPR OR ( (BR4 OR BR5 OR BR6 OR BR7)AND bg.enable))=1
!As soon as one of the lines is raised, the PROCESS ends,to
 prevent any other PROCESS from now being started.  Otherwise,
 a higher priority PROCESS could be started!

        END;  !End of process arbit!

        PROCESS npr.grant
        DLAY 75;
        NPG ← /;
        DLAY 7500+-2500 UNTL SACK = /;
        NPG ← \;
        END;

        PROCESS br4.grant
        DLAY 75;
        BG4 ← /;
        DLAY 7500+-2500 UNTL SACK = /;
        BG4 ← \;
        END;

PROCESS dvamaster
!This process describes the sequence device A goes through to
 interrupt the processor!
!This device is capable of bus mastership and has priority level
 6.   It is wired closest to the processor of all priority level 6
 devices!

br6a ← /;
```

```
!The device requests bus mastership by raising its bus request line!

!This causes the daisy chain for priority level 6 to be initiated
 and subsequently causes the arbitration and granting mechanisms to
 be initiated!

DLAY UNTL bg6a = /;

!Wait for the bus grant!

SACK ← /;
!Acknowledge the bus grant!

PBR one,two: FRUN

!This construct, the parallel branch(PBR), forces execution of
 code segments labeled "one" and "two" to be done at the same time,
 but not in lockstep(FRUN signifies this)!

one: DLAY UNTL BBSY = \;
     BBSY ← /;

!Wait until the bus is no longer busy, then make it busy!

     D<15:0> ← interrupt.vector
     DLAY UNTL SSYN = 0;
     INTR ← /;
     DLAY UNTL SSYN = /;
     D<15:0> ← 0;
     INTR ← \;
     !SSYN and INTR are handshake lines!
     BBSY ← \;
     MRG;  !wait for "two" to finish!
!End "one" code segment!

two: DLAY UNTL bg6a = 0;
     SACK ← \;
     MRG;  !wait for "one" to finish!
     !After the gran line goes away, the SACK line should too!
END;

!This is the end of dvamaster!

PROCESS dvbmaster

!This is a device which does an NPR (non-processor request)
 and is electrically second closest to the processor!

!For this process, it does the DATI operation!

nprb ← /;

!Request bus mastership - start the daisy chain!
```

```
DLAY UNTL npgb = /;

!Wait for the bus grant!

SACK ← /;
nprb ← \;

DLAY UNTL BBSY = \;
BBSY ← /;
!Wait until the bus is not busy then get it!

A<17:0> ← slave.address;
C0 ← 0 ALSO
C1 ← 0;  !This is the code for DATI!

!DATIP is similar except the C0 and C1 code is different!

DLAY 150;
DLAY UNTL SSYN = 0;
MSYN ← \;
DLAY 10000 UNTL SSYN ← /;
bdatareg ← d<15:0>;
MSYN ← \;
DLAY 75;
A<17:0> ← 0;
C0 ← 0;
C1 ← 0;
BBSY ← \;
END;
!End of device B DATI,(PROCESS devbmaster)!

!For DATIP, BBSY would have stayed asserted, then the output
 cycle would have begun just as in DATO or DATOB!

PROCESS dvaslave

!This describes how device A responds as a slave device to DATI,
 DATIP, DATO, and DATOB!

DLAY UNTL MSYN = /;
CMP DATI:1 => in;
!If DATI =1, the master wants to input data, go to "in"!

CMP PDATI:1 => in;
!This is the same as DATI from the point of view of the slave!

CMP DAT0:1 => outword;
!Master wants to output data to slave!
outbyte: CMP A<0>:0 =>higher;
         lower: adatareg<7:0>←D<7:0>; !get lower order byte!
                SSYN ← /;
                DLAY UNTL MSYN = \;
                BR finish;
                !Go to the finish!
        higher: adatareg<15:8>←d<15:8>; !Input higher order byte!
```

```
!NOTE - in reality the adataregister is only 8 bits long when
 bytes are transferred.  It was described this way, so that both
 word and byte I/O could be covered!

                SSYN ← /;
                DLAY UNTL MSYN = \;
                BR finish;
outword:  adatareg ← D<15:0>;
          SSYN ← /;
          DLAY UNTL MSYN = \;
          BR finish;
in:       D<15:0> ← adatareg;
          SSYN ← /;
          DLAY UNTL MSYN = \;
          D<15:0> ← 0;
finish:   SSYN ← \;
          END;
END

!This ends the UNIBUS description as far as it is implemented!
```

APPENDIX III

Register-Transfer Level Digital Design Automation: The Allocation Process

Louis J. Hafer
Research Assistant

Alice C. Parker
Assistant Professor

Department of Electrical Engineering
Carnegie-Mellon University
Pittsburgh, Pa. 15213

ABSTRACT

This paper presents a portion of the register-transfer level computer aided design (RT-CAD) research at Carnegie-Mellon University. This part of the research involves the design and construction of an allocator, consisting of a set of algorithms and data structures which synthesize hardware at the logical level from a behavioral description. Preliminary results indicate the allocators performance compares favorably with a human designer.

I Introduction

The research described in this paper represents a portion of the overall Register-Transfer Level Computer Aided Design (RT-CAD) effort at Carnegie-Mellon University. This is a continuing research project which grew out of an initial design automation system reported in [Barbacci 75]. The present system differs from conventional CAD systems in that the input to the system is a behavioral description of the hardware to be designed. Modules in the system include a compiler for the input hardware descriptive language, ISPL, and a simulator. Currently under construction is a module which selects the design style for the logic to be constructed [Thomas 77]. Manipulation of a data structure derived from the input description in order to optimize price/performance constraints has been investigated by [Snow 78]. The allocator routines, which occur after this manipulation, output a logical design which is transformed by the data-base module into integrated circuit chips and interconnections. All of the modules, with the exception of the data-base, either function independently of or relative to the contents of the data base module sets. One of the allocation routines, the distributed logic allocator, is the subject of this paper. A more detailed overview of the system can be found in [Snow 78].

II Allocator Overview

After the algorithmic description (ISPL) of the system to be designed has been manipulated by the higher level design routines to achieve the price/performance objectives provided by the user, it is used as input to the allocation routines. The allocators fall into two categories, data-memory and control. The data-memory allocators perform a mapping function from the algorithmic description to the data part of the hardware implementation. The data part is considered to consist of the data storage elements, data operators, and data paths (including multiplexers and demultiplexers) necessary to implement the operations specified in the algorithmic description. It should be noted that due to the characteristics of the ISPL language, this mapping may be one-to-many or many-to-one, rather than a simple one-to-one translation. The control allocators perform a slightly different function,mapping the timing, sequencing, and branching information implicit in the ISPL description onto control states, control signals, and conditional branching signals to control the data part. Again, the mapping is not a simple one.

The allocator described here is a data-memory allocator for the distributed logic design style. As we pointed out earlier, the allocator itself is technology independent, and the mapping of data operations onto specific integrated circuit packages is performed by the data base module. The data-base module can be updated as new packages are added to the module sets. It should be understood that the process referred to as allocation throughout the remainder of this paper is a logical allocation in terms of a generic set of data storage elements, operator primitives defined by the ISPL language, data paths, and the abstract switching functions multiplex and demultiplex.

The first version of the allocator is experimental, and it performs only minor optimizations on the designed hardware. Rather, it has been designed to illustrate :

* The feasibility of hardware synthesis from an ISPL description
* The independence of the allocator from specific integrated circuit module set information
* Information necessary to the design process which cannot be expressed in ISPL
* The types of data structures needed for the allocation
* Bounds on the size of the ISPL input description that can be processed by the system
* Exceptional constructs possible in ISPL which may be difficult or impossible to design or implement in hardware
* Types of error checking that can be performed by the allocator
* Areas where optimizations are possible in future, more sophisticated allocators

In addition, the allocator has been designed as a possible skeletal structure for future allocators in order to ease the programming overhead and standardize input/output formats and data structures.

In the following description of allocator structure and function, we attempt to extrapolate details learned through implementation into a basic philosphy of allocator design. The

The research described in this paper was supported in part by the U.S. Army Research Office under grant # DAAG29-76-G-0224.

procedure used by the allocator might be compared to a two pass compilation. The first pass may be considered as a syntax and feasibility check. The allocator inputs a parsed ISPL description, constructs data structures analogous in function to symbol tables, and enforces various constraints necessary to insure that the data storage locations, logical mappings, and input/output interface characteristics specified in the description can be implemented in hardware. If no errors are encountered, it proceeds to allocate the basic data storage structures called for in the description, and any additional data paths, storage, and operators necessary to implement variable accessing schemes described by the logical mapping facility of ISPL. The second pass may be considered as the semantic phase with the activity of code generation replaced by the allocation of data paths, operators, and additional storage as needed to implement the functions specified in the ISPL description. Parallelism analysis is performed at several levels to warn the user of error conditions (array acess conflicts; variable value ambiguity due to parallel assignment/use) and determine constraints relating to optimization of the hardware. The allocation is then completed by the addition of multiplexing where necessary.

Allocation differs from compilation, however, in that in a compilation one is concerned with implementing the specified data operations on a fixed data part whose capabilities are known a *priori*. In allocation, the allocator must be able to recall and utilize the capabilities of a data part which is being dynamically created. The allocator thus works from the inside out, first creating the data storage and access structures, and then adding the necessary data paths and operators to perform the desired data operations. In addition, the output of the allocator is, in the general case, a non planar directed graph, rather than a linear list of compiled instructions.

The central concept used in the allocation process is the operation path. In the most general case, this consists of two sources, an operator, and the data paths from each source to the operator. This concept was chosen as the base because the operation path is the minimal unit which can be considered when analyzing for possible parallelism conflicts during optimization to reduce the number of allocated operators. Throughout the following allocation description, the term path will mean operation path, while the physical data path between a source and a destination will be referred to as a link.

The detailed description of the allocator will focus on the allocator inputs, data structures, algorithms, and outputs.

III Allocator Inputs

The primary input is the compiled ISPL description, in the form of a symbol table and a statement table. This input is augmented with information supplied by the user in the form of a "technical file". This file contains two types of information :

- Global information about the description which cannot be specified in ISPL.
- Interface information to be passed to the module data base for selection of specific IC's. This information describes the desired logical and electrical characteristics of the inputs and outputs of the device being allocated.

A small portion of the ISPL description of an elevator controller is shown in fig. 1, along with relevant portions of the compiler symbol table and statement table. This description will be used as a running example in describing the allocator. The technical file for the controller is shown in fig. 2. The PROCESS specification indicates that there are two separate asynchronous control environments present in the complete controller description. The input and output lines and their characteristics are specified under the headings INPUT and OUTPUT.

IV Allocator Data Structures

As shown in table 1, the allocator builds seven major data structures during the allocation of an ISPL description. The first three, along with the compiler symbol table, comprise the portion of the data structures which function as symbol tables for the allocator. The path graph (fig. 3) contains the allocated data part of the device and is the primary output from the allocator. The path table and path parallelism table are the major working data structures. Although the internal details of these structures are not important, certain overall characteristics should be discussed.

```
! technical file for elevator control

PROCESS :        MAIN , LOOK.FOR.CALLS ;

INPUT :

! inputs for LOOK.FOR.CALLS

        up.call = level down / elec ttl ,
        down.call = level down / elec ttl ,
        button = level down / elec ttl ;
```

Fig.2 Technical File for LOOK.FOR.CALLS

Data Structures

1) Process Table: one entry per process, records variables and defined procedures used by each process

2) Called Procedure Table: one entry per defined procedure, records procedure usage information

3) Allocator Symbol Table: one entry per allocated variable (includes inputs and outputs). Contains physical characteristics of the variable

4) Operator Table: performs symbol table function for all allocated operators

5) Path Graph: graph representation of the data part. Contains all allocated variables, operators, and links

6) Path Table: per process record of paths used by the ISPL description

7) Path Parallelism Table: per process record of parallelism described in the ISPL description

8) Compiler Symbol Table: } inputs from ISPL compiler

9) RTM Statement Table: }

Table 1

```
        ! Declarations for Simple.elevator.control !

        car.floor<2:0> ;        !floor car is on
        car.call[15:0]<> ;      !8 floors, 1 bit each u/d
    macro top.floor :=    7#


        ! Declarations for Look.for.calls !

        scan.floor<2:0> ;       !scan inputs from 8 floors
        up.call<> ;             !input used with scan
        down.call<> ;           !input used with scan
        button<> ;              !input used with scan

        ! end declarations for Look.for.calls !

Look.for.calls := (

          !floorscan controls multiplexors whose inputs
          !are the call buttons.  The outputs of the multiplexors
          !are upcall, downcall, and button

  scan.floor ← 0 next
  Next.floor := (
        !upcall at scanned floor ?
        (IF up.call => car.call[1@scan.floor] ← 1) next
        !down call at scanned floor ?
        (IF down.call => car.call[0@scan.floor] ← 1) next
        !button in car pushed for floor ?
        (IF button => (
                !decide wether up or down call
                DECODE car.floor GTR scan.floor =>
                car.call[1@scan.floor] ← 1;
                car.call[0@scan.floor] ← 1
                )) next
        scan.floor ← scan.floor + 1 next
        (IF scan.floor LEQ top.floor => next.floor))
        !leave if all floors scanned; otherwise look at next floor
        next look.for.calls
        );
```

Fig.1(a) ISPL Source Code for LOOK.FOR.CALLS

```
INDEX  LABEL    FLAG   OPCODE  DEST    SOURCE1 SOURCE2 MERGE   PATHS
    5              0   'CLEAR ''SCAN.F'
                                (    33)
    6  'NEXT.F'    1   'SMERGE'
       (    30)
    7              0   'ISP   '                                  4
   10              0   'IF    '        'UP.CAL'                 13  13,11
                                       (    37)
   11              0   'CONC  ''%TRAAC'       1'SCAN.F'
                                (    52)(    42)(    33)
   12              0   'WRITE ''CAR.CA''%TRAAC'       1
                                (     5)(    52)(    42)
   13              0   'SMERGE'
   14              0   'IF    '        'DOWN.C'                 17  17,15
                                       (    15)
   15              0   'CONC  ''%TRAAC'       0'SCAN.F'
                                (    52)(    41)(    33)
   16              0   'WRITE ''CAR.CA''%TRAAC'       1
                                (     5)(    52)(    42)
   17              0   'SMERGE'
   20              0   'IF    '        'BUTTON'                 31  31,21
                                       (     3)
   21              0   'GTR   ''%TFAAA''CAR.FL''SCAN.F'
                                (    47)(     6)(    33)
   22              0   'BRANCH'        '%TFAAA'                 30  23,26
                                       (    47)
   23              0   'CONC  ''%TRAAC'       1'SCAN.F'
                                (    52)(    42)(    33)
   24              0   'WRITE ''CAR.CA''%TRAAC'       1
                                (     5)(    52)(    42)
   25              0   'JOIN  '                                  30
   26              0   'CONC  ''%TRAAC'       0'SCAN.F'
                                (    52)(    41)(    33)
   27              0   'WRITE ''CAR.CA''%TRAAC'       1
                                (     5)(    52)(    42)
   30              0   'SMERGE'
   31              0   'SMERGE'
   32              0   'INCR  ''SCAN.F''SCAN.F'
                                (    33)(    33)
   33              0   'LEQ   ''%TFAAA''SCAN.F'       7
                                (    47)(    33)(    44)
   34              0   'IF    '        '%TFAAA'                 36  36,35
                                       (    47)
   35              4   'JOIN  '        'NEXT.F'                  6
                                       (    30)
   36              0   'SMERGE'
   37              0   'NOOP  '        'NEXT.F'                  6
                                       (    30)
```

Fig.1(c) ISPL Compiler Statement Table for LOOK.FOR.CALLS

```
INDEX TYPE    FLAGS  DEF  BLK  LBL BCNT      WCNT PNAME   WORDS:BITS:NAME(POSITION)
    3    2 10000000    0    0    0    1         1 'BUTTON'
    5    1 10000000    0    0    0    1        20 'CAR.CA'(0(17):17(0))
    6    2 10000000    0    0    0    3         1 'CAR.FL'<0(2):2(0)>
   15    2 10000000    0    0    0    1         1 'DOWN.C'
   22    4 10001100    0    0    3    0         0 'LOOK.F'
   30    4 10001100    0    0    6    0         0 'NEXT.F'
   33    2 10000000    0    0    0    3         1 'SCAN.F'<0(2):2(0)>
   37    2 10000000    0    0    0    1         1 'UP.CAL'
   41    3 10000001    0    0    0    1         0       0
   42    3 10000001    0    0    0    1         0       1
   44    3 10000001    0    0    0    3         0       7
   47   10 10000001    0    0    0    1         0 '%TFAAA'
   52    7 10000001    0    0    0    4         0 '%TRAAC'
```

Fig.1(b) ISPL Compiler Symbol Table for LOOK.FOR.CALLS

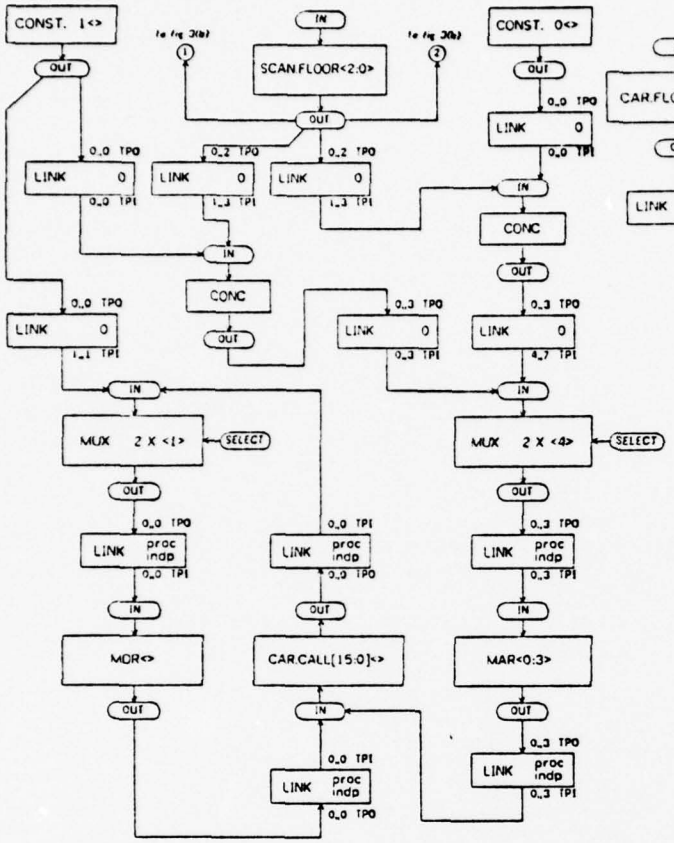


FIG. 3(a) PATH GRAPH FOR LOOK.FOR.CALLS

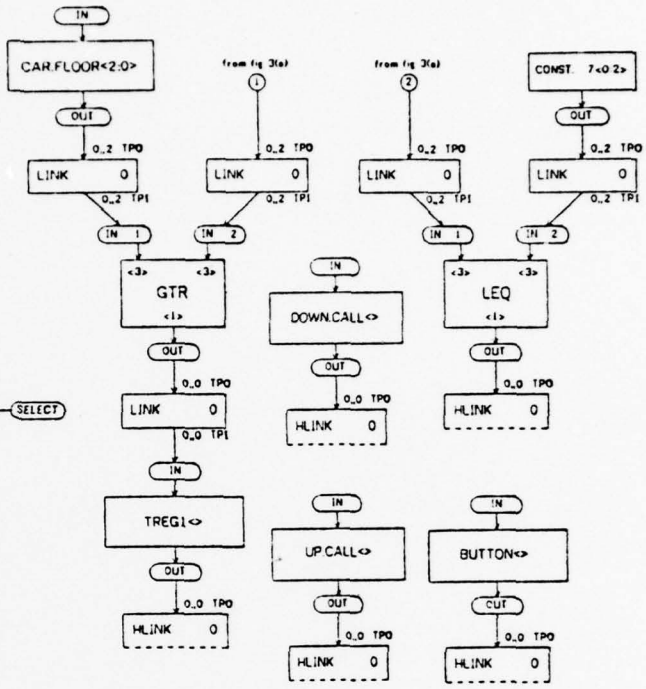


FIG. 3(b) PATH GRAPH FOR LOOK.FOR.CALLS

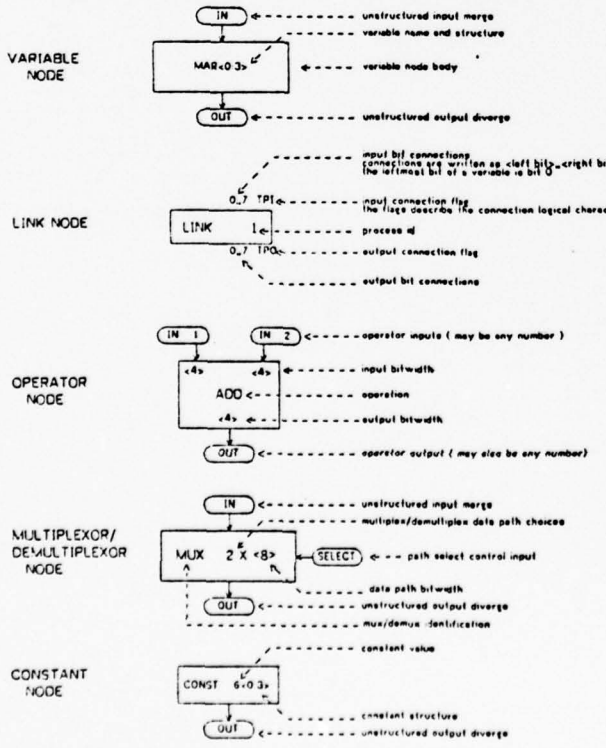


FIG. 3(c) BASIC PATH GRAPH GROUPS

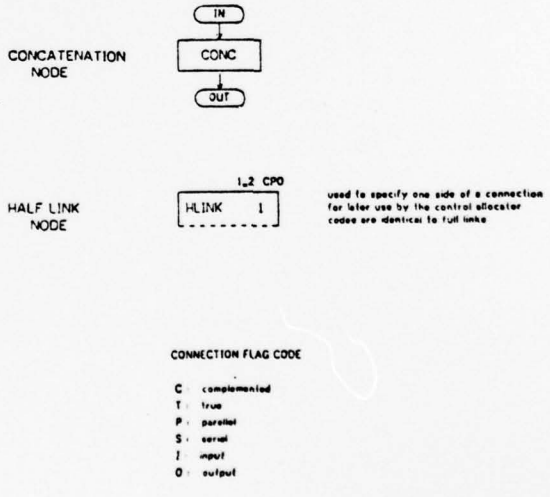


FIG 3(d) BASIC PATH GRAPH GROUPS

Due to the diversity and quantity of information required by the allocator, and the need to access this information in a variety of orderings with a variety of keys, the data structures are heavily cross referenced and contain varying amounts of redundant information. In particular, it may seem that the path table and path graph are highly redundant. However, the graph structure of the path graph makes path location in the graph a combinatorial problem. The path table provides an alternative organization which reduces the location of a path to a linear search and facilitates optimization by keying on the operation and its sources. In general, the diversity of structures enables the allocator to efficiently locate information and relate it to both the original ISPL description and the allocated data part.

A rough measure of the complexity of the data structures can be obtained in terms of the example ISPL description. The process LOOK.FOR.CALLS contains 16 RTM operations in the compiled ISPL code requiring path allocation, and 6 variables. Although the complexity varies due to the dynamic creation and deletion of some structures and is also highly dependent on the degree of parallelism and the number of repetive operations present in the description, approximately 55 words of storage are used per line of ISPL RTM code, and 45 words of storage per variable.

V Allocator Algorithms

The allocator begins by examining the variables and logical mappings declared by the user to see if they can be implemented in hardware. A complete description of the constraints imposed on mappings, and the implementation of array mapping access structures is included in appendix A. If no constraint violations are detected, the allocator proceeds to break the ISPL description into processes and construct the three additional symbol tables used in the allocation. The importance of this breakdown is derived from the fact that each process represents a separate asynchronous control environment. If there are no shared resources, this presents no problem, as each process can be treated in essence as a separate device description and allocated independently. The existence of shared variables or procedures, however, introduces several problems. In the case of procedures, the allocator is capable of two actions. It can absorb the procedure into each process by creating a separate hardware incarnation of the procedure internal to each process, or it can treat the procedure as if it were itself a separate process and allocate an independent hardware incarnation. For the second action, access to the procedure would then be controlled by an arbitration structure subsequently created by the control allocator. In the case of variables, the access arbitration structure would again be created by the control allocator. However, within the control context of a given process, the variable must be regarded as subject to random change due to accesses by other processes. This will have a strong influence on temporary storage allocation.

Upon completing the process breakdown, the allocator initializes the path graph by performing the allocation of the base variable storage elements. Since ISPL does not require explicit memory address or data registers, these are also assigned and allocated. If these registers have indeed been explicitly specified in the ISPL, this can be readily ascertained at the end of the allocation and the allocator MAR and MDR deleted. After the basic storage elements have been allocated, the hardware necessary to implement any array mappings is allocated (again, see appendix A). Pass one processing is completed by the creation of standardized access pointers for each entry in the compiler symbol table indicating where and how the entry may be accessed in the path graph. This relates variables in the ISPL RTM code to the allocated storage locations in the logical design.

The allocator now begins pass two processing. This is performed by process, with hardware allocation and optional optimization (in future versions) performed independently for each process. The allocator obtains the starting point of a process in the RTM statement table and performs a statement by statement allocation of the links, operators, and (if necessary) temporary storage needed to implement the operation. Allocation of an RTM statement begins with an examination of the access pointers of the source variables. A check is made for array memory accessing conflicts if both sources are stored in array storage and, if needed, a temporary is allocated to hold one source while the other is accessed. After the necessary operations to make the sources accessible have been determined and any required links and storage have been entered in the path graph, the path table is accessed to determine if the path has been previously allocated. If it has, it is reused, and the asociated operator is expanded in size if necessary. If the path has not been allocated, the necessary links and operator are entered in the path graph and the path is recorded in the path table. An analysis is now made to determine the appropriate destination, based on the destination specified by the statement table entry and the characteristics of the operation sources. Temporary destinations generated by the ISPL compiler are removed at this point and, depending on the stability of the operation sources, the operation result is either left at the operator output for direct cascading to the next path (as in a combinatorial logic network), or gated to an allocator generated temporary.

Temporaries generated by the ISPL compiler are useless to the allocator due to a generation and usage criterion which, though adequate for simulation, is unsuitable for hardware implementation. The allocator generates temporaries when either source could change before the operation result is used. Cases where this might occur are :

* Either source is shared between two or more processes and thus is subject to random change.

* Either source is an array memory access. This is necessary due to the fact that it is impossible in the general case to know if the value in the MDR will change without extensive processing. A worst case can plausibly be developed here which requires n statement look-ahead, a value trace of the MAR, and is still undecidable without violating the definition of the ISPL parallel action construct.

* Either source is a temporary storage location allocated in a previous instruction. The justification for this condition is a complex combination of problems in the ISPL language and problems internal to the allocator, relating again mainly to problems in determining the stability of the temporary over parallel operations.

When the destination has been chosen, the necessary link is created in the path graph and any further operations necessary to store the destination in array storage are determined and performed if they are required.

Several special cases are worth mentioning. Subfield accesses of variables and logical negation, which are operations in the ISPL RTM code, do not require the full processing just described, but are handled by manipulation of the access pointers. The ISPL control operations using data values as conditionals require only the source processing, so that a pointer can be constructed to allow the control allocator to access the data value. Finally,

operations commonly available as register functions (inc/dec,clear,shifts) are recognized and the required function is added to the register characteristics.

After each statement is processed, the path created is checked against all other paths in use concurrently. Parallel operations are recorded in the path parallelism table, and the user is warned of any array memory or register access conflicts or ambiguities that might occur. No more than a warning can be generated, however, as the allocator cannot detect synchronization mechanisms specified by the user in the ISPL description which could resolve the ambiguity or conflict.

After a process has been allocated as described above, the path graph contains a worst case allocation of the process in terms of operators and temporary registers. Optimization can be performed here using the parallelism information and path allocation records formed during the allocation of the process. As an example optimization, compression of temporary registers is performed using the criterion "combine the registers if they are compatible in size and no parallel or sequential usage conflicts exist". After the optimization routines, the path parallelism table is cleared and the path table selectively purged so that these structures do not become unmanageably large.

After all processes have been allocated, multiplexers are allocated where needed and the allocation is complete.

VI Allocator Outputs

The primary output of the allocator is the path graph, which is used by the data base module in conjunction with the allocator symbol table to map IC's onto the logical allocation. The allocator also produces a dump of the major internal tables for the user.

VII Allocator Performance

The allocator described here is currently undergoing testing, so the results presented here are preliminary. Also, the actual hardware is to be allocated by the module data base, which is currently in the design stage. However, by performing a hand allocation of IC chips onto the path graph, we have obtained preliminary results. The ISPL description used was part of the design style experiment described in [Thomas 77], which provides a controlled measure of how well the allocator performs in relationship to human designers. The results are shown in table 2. The cost figures for the allocator design were derived using Thomas' cost estimate :

Cost for data part of process LOOK.FOR.CALLS as implemented for Thomas' design experiment:

Designer 10: $32.49 Designer 5: $71.45

Cost for data part of process LOOK.FOR.CALLS as implemented from allocator path graph*:

$47.63

*The same subset of the TTL chip family used in the design experiment was used to implement the allocator path graph.

Table 2

Cost = (total chip cost) + ($3 overhead/chip)

One can see that the cost of the automated design falls within the same order of magnitude as the cost of designs produced for the design experiment, with no significant optimization on the part of the allocator.

VIII Conclusions and Future Research

The encouraging results described here have led us to the following conclusions :

* the basic experimental allocator described here will function successfully as the base for an expanded allocator with optimization capabilities
* specific module set information is not needed to produce a non-optimal allocation
* the size of the ISPL description which can be handled depends only on the amount of core available to the allocator
* the allocator can detect user constructs in the ISPL description which will produce complex hardware or unreliable operation

In addition, we have concluded that a large portion of the complexity of the allocator is due to :

* The ability to allocate multi-process ISPL descriptions.
* The lack of a one-to-one correspondence between the compiled RTM statement table operations and the actions necessary to perform these operations in a hardware implementation.
* The availability of the logical mapping facility in ISPL.

Future results will include a more extensive evaluation of the allocator performance using Thomas' design style experiment as a yardstick, and comparison with other large, multi-process designs currently under construction at C-MU.

Appendix A Mappings

The ISPL language contains a generalized logical mapping facility allowing the user to declare a logical variable in terms of a previously declared physical or logical variable. The only restriction imposed by the ISPL compiler is that the physical size (total bits) of each side of the declaration be equal. For mappings where the mapped variable is declared as a register type variable, this restriction is sufficient. When the mapped variable is declared as an array structure, however, additional restrictions must be imposed to insure that a reasonable hardware implementation of the mapping can be created. Consider the following example mapping, which will be used to define the terms used in the equation which defines the mapping constraints. The first declaration is the physical (base) declaration, and the second is the mapping declaration.

```
A1[0:1,8:9,14:15]<0:7> ;
AA2[5,9,12]<0:15> := A1[0:1,8:9,14:15]<0:7> ;
```

Define the following terms:

main declaration	::=	declaration of A1
mapping primary	::=	left half of mapping declaration (AA2)
mapping secondary	::=	right half of mapping declaration (A1) (may in general be all of the main declaration or a subset of the addressing space defined by the main declaration)
$bitcnt_p$	::=	bit size of primary word (16 bits in this example)
$bitcnt_s$	::=	bit size of secondary word (8 bits in this example)
d	::=	$lg_2(bitcnt_p/bitcnt_s)$ (1 in this example)
$b0_p$	::=	lowest address of primary (5 in this example)
$b0_s$	::=	lowest address of secondary (0 in this example)
adr_p	::=	address from primary address space
adr_s	::=	address from secondary address space

With these definitions, any mapping satisfying the equation

$(adr_p + \lambda)\uparrow d = adr_s$ where $\lambda = (b0_s\uparrow(-d)) - b0_p$ = constant and $\uparrow$ is the logical shift operation (left shift is positive direction)

can be implemented with at most an addition and wired shift in the address path, a multiplexor/demultiplexor pair to provide the necessary gating into and out of the MDR of the main definition variable, and, for the case where $bitcnt_p > bitcnt_s$, a MDR for the primary declaration to assemble the primary word while the necessary number of memory accesses are made in the main memory.

The path graph representation of the example mapping as allocated by the data memory allocator is attached as fig. A-1.

References

[Barb 75] Barbacci, M.R., Sieworek, D.P., "The CMU RT-CAD System: An Innovative Approach to Computer Aided Design",AFIPS Conference Proceedings, vol. 45 pp 643-655, 1976.

[Thom 77] Thomas, D.E., Sieworek, D.P., "Measuring Designer Performance to Verify Design Automation Systems", Design Automation Conference Proceedings, vol. 14 pp 411-418, 1977.

[Snow 78] Snow,E., Sieworek, D.P., Thomas, D.T., "A Technology Relative Computer Aided Design System: Abstract Representations, Transforms, and Design Tradeoffs", Design Automation Conference Proceedings, vol. 15, 1978.

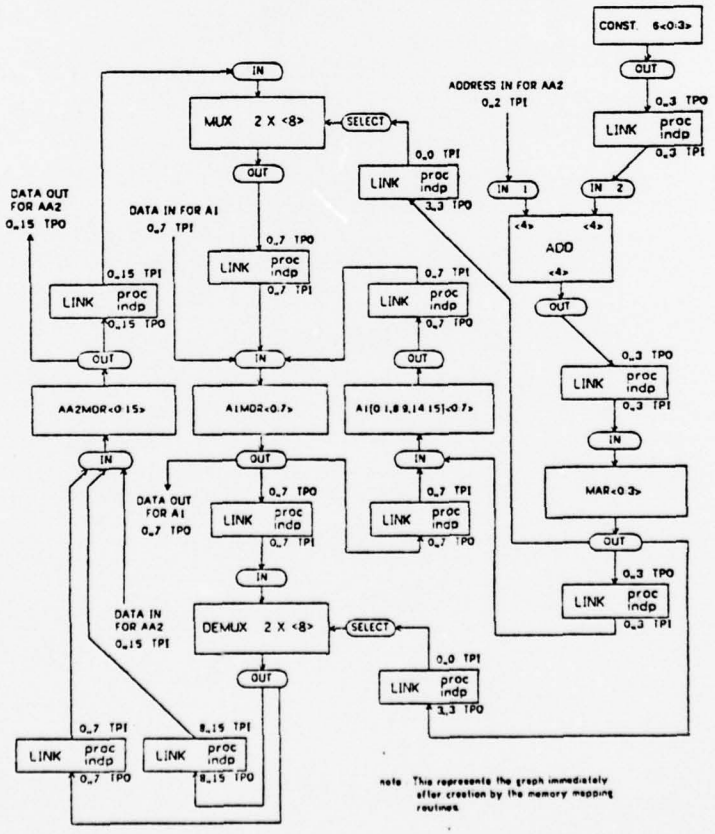


FIG. A1 PATH GRAPH FOR MEMORY MAPPING EXAMPLE

APPENDIX IV

The PDP-8 Design

This section contians the PDP-8 ISPL description, a block diagram of the allocator output and the Chip Count for the DEC and automated CMU PDP-8. THE CMU DESIGN IS PRELIMINARY - IT HAS NOT BEEN VERIFIED AND THE CHIP COUNT IS APPROXIMATE.

```
PDP8 :=(declare

! The basic PDP-8 instruction set, not including the extended arithmetic
! element (EAE) option. I/O instructions are limited to those dealing
! with the interrupt mechanism

!** Memory.State **

M\Memory[0:4095]<0:11>;

!** Processor.State **

lac<0:12>;
       L\Link<>                 := lac<0>;
       AC\Accumulator<0:11>     := lac<1:12>;
PC\Program.Counter<0:11>;
RUN<>;
I.STATE<>;
I.REQUEST<>;
SWITCHES<0:11>;

!** Instruction.Format **

i\instruction<0:11>;

       op\operation.code<0:2>  := i<0:2>;
       ib\indirect.bit<>       := i<3>;
       pb\page.0.bit<>         := i<4>;
       pa\page.address<0:6>    := i<5:11>;

       IO.SELECT<0:5>          := i<3:8>;      ! device select
       io.control<0:2>         := i<9:11>;     ! device operation
              IO.P1<>            := io.control<0>;
              IO.P2<>            := io.control<1>;
              IO.P4<>            := io.control<2>;

       sma<>   := i<5>;              ! skip on minus AC
       spa<>   := i<5>;              ! skip on positive AC
       sza<>   := i<6>;              ! skip on zero AC
       sna<>   := i<6>;              ! skip on AC not zero
       snl<>   := i<7>;              ! skip on L not zero
       szl<>   := i<7>;              ! skip on L zero
       is<>    := i<8>;              ! invert skip sense
       group<> := i<3>;              ! microinstruction group
       cla<>   := i<4>;              ! clear AC
       cll<>   := i<5>;              ! clear L
       cma<>   := i<6>;              ! complement AC
       cml<>   := i<7>;              ! complement L
       rar<>   := i<8>;              ! rotate right
       ral<>   := i<9>;              ! rotate left
       rt<>    := i<10>;             ! rotate twice
       iac<>   := i<11>;             ! increment AC
       osr<>   := i<9>;              ! logical or AC with SWITCHES
       hlt<>   := i<10>;             ! halt the processor
```

```
!☆☆ Effective.Address ☆☆

skip<>;
last.pc<0:11>;
eadd\effective.address<0:11>;

efadd :=(
      (Decode pb =>
                eadd ←'00000 @ pa;
                eadd ← last.pc<0:4> @ pa)
                Next
       (If  ib =>
      ( If eadd<0:8> Eql #001 =>( M[eadd] ← M[eadd] + 1)) Next
       eadd ← M[eadd]));

input.output :=(
(if i<3:11> eql '000000001 =>
                        (I.STATE ← 1
                         ));
(if i<3:11> eql '000000010 =>
                        (I.STATE ←0))
);


skip.group :=(
       skip ← 0 Next
       (Decode is =>
                        ((If snl And (L Eql 1) => skip ← 1);
                        (If sza And (AC Eql 0) => skip ← 1);
                        (If sma And (AC Lss 0) => skip ← 1));
                        (( IF( szl and sna and spa) Eql 0 => skip ← 1);
                       ( If szl And (L Eql 0) => skip ← 1);
                        (If sna And (AC Neq 0) => skip ← 1);
                        (If spa And (AC Geq 0) => skip ← 1)) )
                Next
      ( If skip => PC ← PC + 1)                    ! Skip

               );
operate :=
       ((Decode group =>
                        ((If cla => AC ← 0);
                        (If cll => L ← 0) Next
                        (If cma => AC ← Not AC);
                        (If cml => L ← Not L) Next
                        (If iac => lac ← lac + 1) Next
                        (Decode rt =>
                                        ((If ral => lac ← lac ↑rl 1);
                                        (If rar => lac ← lac ↑rr 1));
                                        ((If ral => lac ← lac ↑rl 2);
                                        (If rar => lac ← lac ↑rr 2))
));
                        (Decode i<11> =>
                                        ((If hlt => RUN ← 0);
                                        skip.group Next
```

```
                                        (If cla => AC ← 0) Next
                                        (If osr => AC ← AC Or SWITCHES));
                                        (If cla => AC ← 0)          ! eae group
)
)
);
execute :=(
        EFADD NEXT
        (Decode op =>
                AC ← AC And M[eadd];
                lac ← lac + M[eadd];
        (
                        M[eadd] ← M[eadd] + 1 Next
                        (If M[eadd] Eql 0 => PC ← PC + 1));
        (
                        M[eadd] ← AC Next
                        AC ←  0);
                        (M[eadd] ← PC Next
                        PC ← EADD + 1);
                PC ← eadd;
                input.output;
operate
)       )


eralced
!** Instruction.Interpretation **

start :=(
                i ← M[PC]; last.pc ← PC Next
                PC ← PC + 1 Next
                execute Next
                (If I.STATE And I.REQUEST =>
                        ( M[0] ← PC Next
                        PC ← 1))
                        )

NEXT START
)
```

DEC - PDP-8/E Chip Count

The following chip count was taken from M8300 "major registers."

Part #	(TI TTL equiv.)	Quantity	Function
7400		3	Quad 2 input NAND
7402		1	Quad 2 input NOR
74H04		2	Hex inverter
7420		1	Dual 4 input NAND
7430		1	8 input NAND
74H87		3	4 bit true/complement
7483		3	4 bit binary full adders
84151		12	1 of 8 MUX
74153		6	dual 4 to 1 MUX
8271	74194	15	4 bit universal shift reg.
8266	74157	8	Quad 2 to 1 MUX
8235	74H87	3	4 bit true/complement
8881	7401	6	Quad 2 input NAND
	Total	64	integrated circuit chips

CMU - PDP-8 Chip Count

The following chip count was taken from the part of the automated design corresponding to the DEC PDP-8 M8300 "major resistors." The chips we allocated by hand using a "worst case" allocation. For example, if the allocator specified a 7 to 1 13 bit MUX, one was provided, even if the MUX could be decomposed into a simpler structure.

Part #	(TI TTL Equiv.)	Quantity	Part of Path Graph Implemented
74151		13	MUX 7 x <13>
8271	74194	3	AC<0:11>
7474 (½)		1	L<0>, TREG3 <0>
7402		3	OR<12>
7404		2	
74153		7	MUX 3 x <13>
8271	74194	3	TREG3 <1:12>
8271	74194	3	LAST.P<0:11>
7483		3	
7400		3	INCR<13>
7404		3	
7420		1	EQL<12>
7430		1	
7483		4	ADD<14>
7400		3	AND<12>
7404		2	
74153		6	MUX 4 x <12>
8271	74194	3	TREG1 <0:11>
74153		6	MUX 3 x <12>
8271	74194	3	PC<0:11>
74153		6	MUX 3 x <12>
8271	74194	3	TREG2 <0:11>
7483		3	INCR<13>
8271	74194	3	EADD<0:11>
74153		6	MUX 3 x <12>
	Total	94	integrated circuit chips

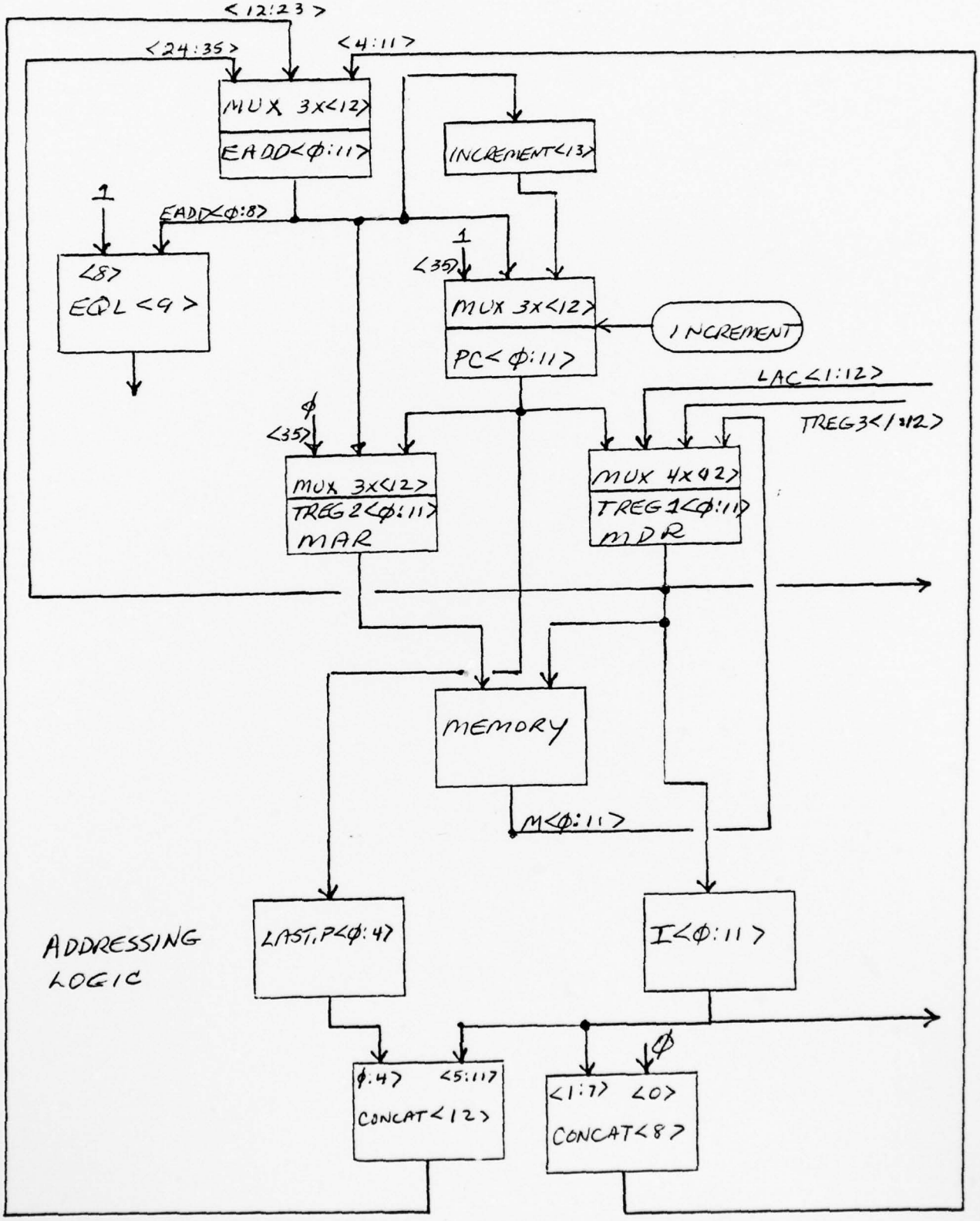
CMU PDP-8 Design

The next 4 pages contain the automated PDP-8 design.

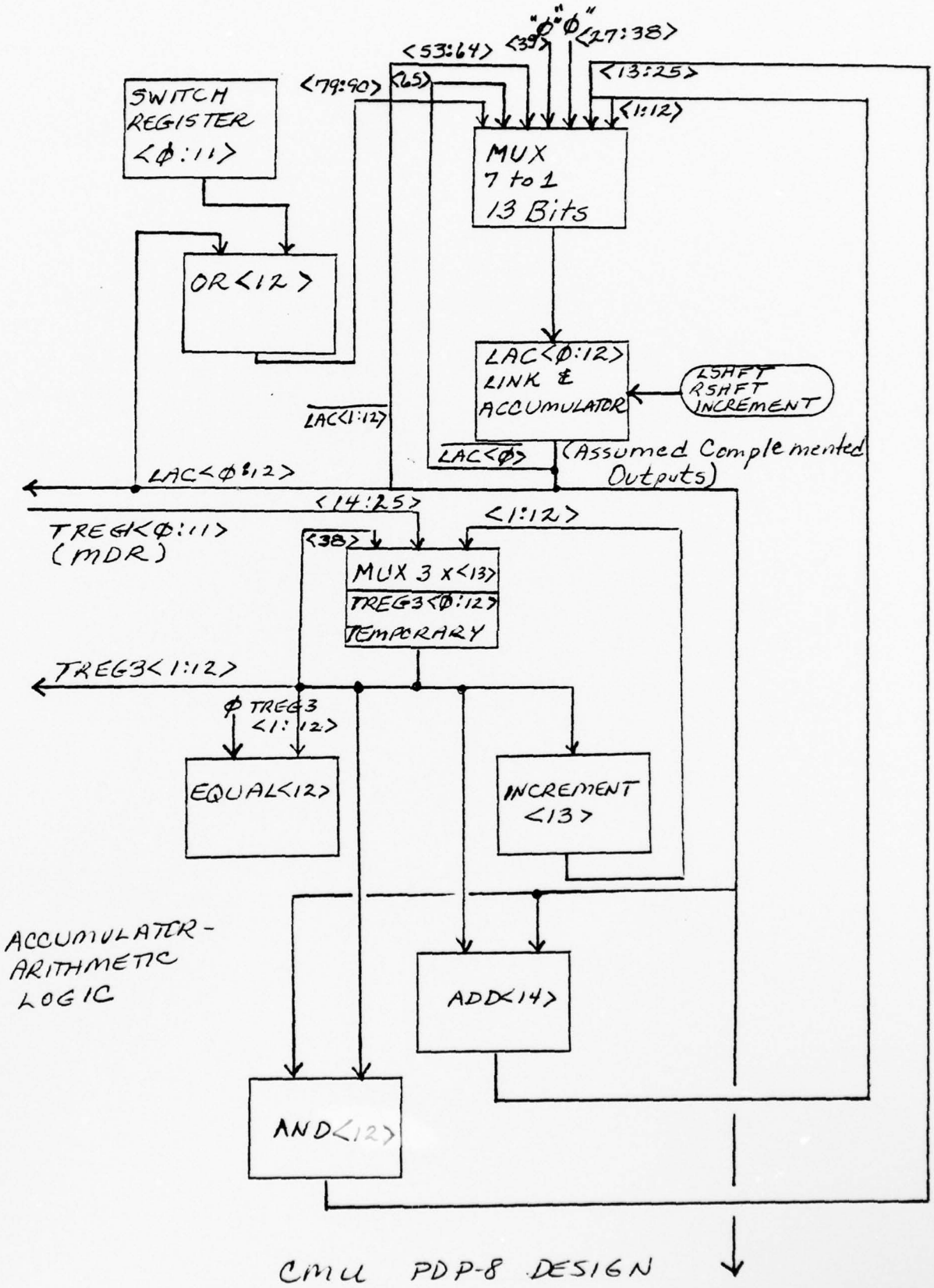
All connections contain the number and position of bits specified.

Output wires not attached are used by the control to test for certain conditions.

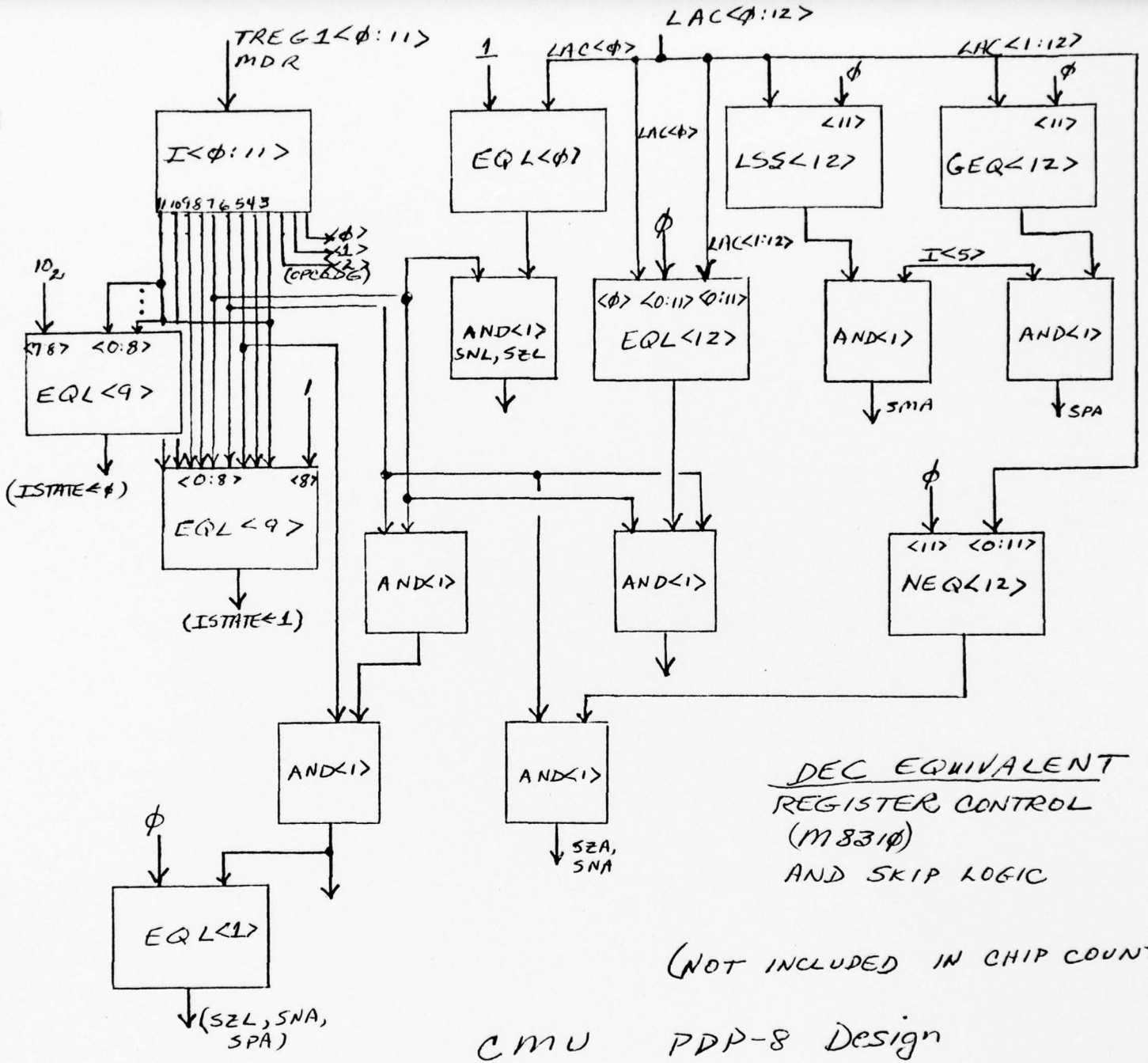
Some MUX connections not specified(and operator input connections) are padded with zeros; a few are padded with ones or not used. Details are found in the actual allocator output files.



CMU PDP-8 DESIGN



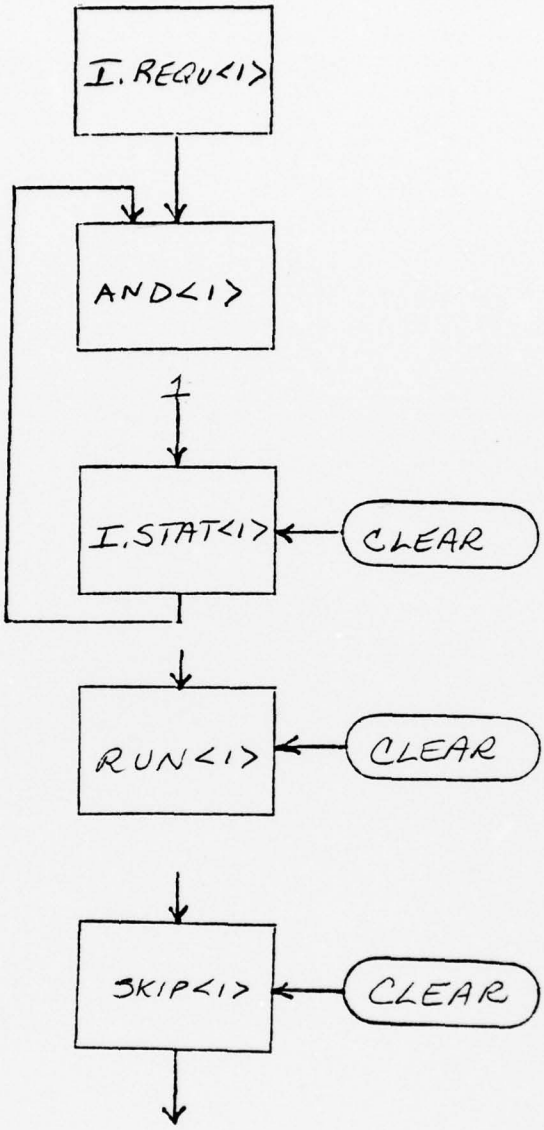
CMU PDP-8 DESIGN



DEC EQUIVALENT
REGISTER CONTROL
(M8310)
AND SKIP LOGIC

(NOT INCLUDED IN CHIP COUNT)

CMU PDP-8 Design



MAJOR CONTROL

CMU PDP 8 DESIGN