
Doctoral Dissertations

Student Theses and Dissertations

1970

IDDAP – Interactive computer assistance for creative digital design

Richard Franklin Crall

Follow this and additional works at: https://scholarsmine.mst.edu/doctoral_dissertations



Part of the [Electrical and Computer Engineering Commons](#)

Department: **Electrical and Computer Engineering**

Recommended Citation

Crall, Richard Franklin, "IDDAP – Interactive computer assistance for creative digital design" (1970).
Doctoral Dissertations. 2040.

https://scholarsmine.mst.edu/doctoral_dissertations/2040

This thesis is brought to you by Scholars' Mine, a service of the Missouri S&T Library and Learning Resources. This work is protected by U. S. Copyright Law. Unauthorized use including reproduction for redistribution requires the permission of the copyright holder. For more information, please contact scholarsmine@mst.edu.

IDDAP -- INTERACTIVE COMPUTER ASSISTANCE

FOR CREATIVE DIGITAL DESIGN

by

RICHARD FRANKLIN CRALL, 1943-

A DISSERTATION

Presented to the Faculty of the Graduate School of

UNIVERSITY OF MISSOURI - ROLLA

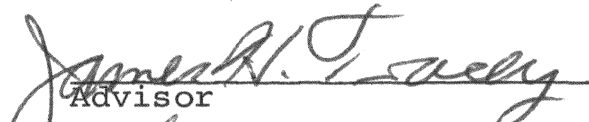
in partial fulfillment of the requirements for the degree


DOCTOR OF PHILOSOPHY


in

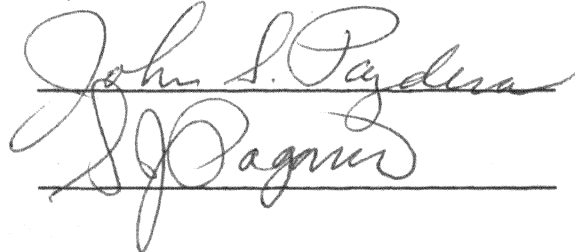
ELECTRICAL ENGINEERING

1970



Advisor






ABSTRACT

A new computer-aided design program to assist in the initial phases of logical design is described. The program, intended for use via an on-line remote terminal, will allow the designer to study and experiment with design alternatives during the initial creative design phases. An ALGOL-like language is used for specifying the system being designed. In addition to simulating the design, the program allows the user to perform on-line design changes, reorganize data and generate timing diagram information.

ACKNOWLEDGEMENTS

I would like to express my appreciation to Dr. Tracey for his helpful suggestions, stimulating questions, and for his prompt careful review of this dissertation.

I wish to acknowledge Mr. George Rhine and Mr. Wayne Omohundro for their patience and helpful comments while using IDDAP during its development.

The Assembler Language program for communicating with the remote terminal is the work of Mr. John Wood. Professor Ralph Lee and the University of Missouri - Rolla Computer Center staff also deserve thanks for their help and cooperation.

Most of all, I wish to thank Barb for her understanding and sacrifice freely given throughout this work.

TABLE OF CONTENTS

	page
ABSTRACT	ii
ACKNOWLEDGEMENT	iii
LIST OF ILLUSTRATIONS	vi
I. INTRODUCTION	1
II. REVIEW OF THE LITERATURE	4
A. Iverson Notation	4
B. Programming Language/One	6
C. Digital Design Language	7
D. Register Transfer Language	8
E. Computer Design Language	8
F. Design Oriented Language	9
G. Other Automated Digital Design Efforts	10
H. Summary and Comparison	11
III. SYSTEM DESCRIPTION OF IDDAP	13
IV. COMMUNICATING WITH IDDAP	17
A. Description Statements	17
1. Type-One Statements	18
2. Type-Two Statements	22
B. Control Commands	26
C. Simulator Output	33
V. USING IDDAP	34
A. Fundamental Concepts	34
B. Clocked Simulation	39
C. Additional Features	44

TABLE OF CONTENTS (Continued)

VI.	THE SOFTWARE FOR IDDAP	48
VII.	CONCLUSION	58
	REFERENCES	61
	VITA	64

LIST OF ILLUSTRATIONS

Figures		page
1	IDDAP System Block Diagram	13
2	Logic Diagram of the Circuit Described in Example 3	40
3	Block Diagram of Supervisor	49
4	Block Diagram of Description Translator (IDD_TRAN)	52
5	Block Diagram of Simulator (IDD_SIM)	53

I. INTRODUCTION

The basic concepts and some of the details of an interactive computer-aided design program to assist logical designers in the initial phases of design are described. It differs from most other design aid programs in that it is interactive and usable by the designer in the initial creative and study phases of the design process. The system will also be useful for instruction and experimentation in digital system design.

The Interactive Digital Design Assistance Package (IDDAP) is intended to be used via a remote terminal. After entering a description of the contemplated design, the user can request any of several forms of assistance. Based on the resulting information displayed by the computer, the designer may make modifications to the original description and then again call upon the computer for assistance. The designer thus interacts with the computer to obtain a workable solution to his design problem.

Other computer programs have been developed for a variety of design tasks. The design language used to communicate with these programs may also serve as a vehicle for creative thought, and the resulting description can serve as a useful form of documentation. The design language selected for use in IDDAP has the familiar

ALGOL-like structure. It is essentially the language proposed by Chu to formally describe digital systems.

Although descriptions at all levels of complexity are possible, interactive use is generally inefficient for very large design problems. One large class of design problems at an intermediate level is the design of interface units. Other problems appropriate for interactive design include the design of sub-units, such as a floating point arithmetic unit, and special purpose computers. The present version IDDAP is not particularly well suited to the study of a large computer nor for such detailed studies as critical race or hazard analysis.

Computer-assisted design has been used to perform simulation and optimization (minimization), conduct race and fault analysis, synthesize logic equations and state tables, and to provide documentation such as logic diagrams and wiring lists. Only a few of these functions are relevant to the initial phases of the design process. However, there are several forms of assistance peculiar to this phase of design which have received little attention in the past. Besides simulating the described digital system, IDDAP also provides, upon request, a variety of other presentations such as cross reference lists and description reorganizations. The interactive approach is itself an important form of assistance; the user can make on-line design changes and can exercise more control over

the simulation process than would be possible in a batch-mode environment.

II. REVIEW OF THE LITERATURE

Prior work in the area of computer-aided design of digital systems may be conveniently discussed by considering each of the languages developed for expressing digital designs. Not all of the proposed forms of describing digital systems have actually been used for communication with computer-aided design programs. Each form to be discussed, however, shows at least some potential as an input language. Many of the languages in their original form make use of graphic symbols which are difficult or impossible for common computer input/output equipment. For use as an input language, straightforward substitutions are or could be made.

A. Iverson Notation

Also known as A Programming Language (APL)^[1], Iverson Notation was originally developed as a general purpose problem solving language. Its ability to handle vectors, arrays, and other complex structures in a concise, elegant fashion makes it a powerful language. A description in Iverson Notation consists of three parts: a main program, "system programs", and "defined operations". System programs describe activities occurring concurrently with the main program such as input/output and interrupts. Conceptually, the defined operations are subroutines which may be used by

the main program or other defined operations. Statements in the main program and in the defined operations are executed in sequence except when altered by a transfer statement. Thus, parallel operations are as difficult to express as in other general-purpose languages such as FORTRAN.

Use of Iverson Notation has proceeded along three paths. In its original form, Iverson Notation has been used to describe IBM's System/360^[2] and as a notational tool in discussing computer architecture^[3]. A conversational remote terminal version, consisting of the arithmetic and mathematical functions, has been developed^[4,5]. The ALERT^[6,7] system makes use of another subset of Iverson Notation to obtain the Boolean equations from a high level description depicting the architecture of a proposed digital system. An important aspect of ALERT is that it automatically generates any intermediate registers or control logic which may have been implied by the high-level description.

The unusual symbols of Iverson Notation, when converted for the purposes of ALERT, make the language appear much less strange. With the added conventions for defining names of logic signals and units, the language begins to resemble other ALGOL-like languages. Reducing a formal Iverson description such as that of the IBM System/360 into a form suitable for ALERT involves, among

other things, replacing the semi-graphical representation of conditional branching with the familiar IF *expression* THEN *action* form.

The ALERT system was written for batch-mode operation on the IBM 7094. Its primary purpose is to produce the final logic design which it presents as gate-level logic diagrams or Boolean equations. ALERT is but one part of a series of automated design programs used at IBM. One of the programs uses the output of ALERT to simulate the system being designed. If design errors are found as a result of using the simulator, then presumably the designer has the choice of modifying the Boolean equations produced by ALERT or correcting the original Iverson description and using ALERT again.

The remote-terminal version of APL, while not supporting some of the bit-string operations, can be (has been) used to simulate digital designs^[8].

B. Programming Language/One

Like Iverson Notation, Programming Language/One (PL/1) is intended to be a general problem-solving language. Due to its ability to handle bit (and character) strings in a straightforward fashion, it also is suitable for describing digital systems. No computer-aided design programs have been specifically written to use PL/1 as the input language. However, the PL/1 compiler itself may be used to process the description of a digital system and thus to obtain a

simulation of the system being designed. The central processing unit of the ILLIAC IV^[9] and the SCC 650, a small general purpose computer^[10], have been described in PL/1. Use of general purpose languages points up the fact that the design need not be reduced to gate-level specification before testing the design through simulation.

The Conversational Programming System (CPS) is essentially a version of PL/1 for use on a remote terminal. Interactive simulation of a digital description is thus possible, again without any special software effort oriented toward digital design.

C. Digital Design Language

Digital Design Language (DDL), proposed by Duley and Dietmeyer^[11], is a high level language suitable for describing the organization and operation of large digital systems. DDL is comparable to CDL in its generality, conciseness and preciseness. Although its flexibility is impaired by its rigid modular organization, such an approach may be desirable, especially for complex systems.

System descriptions in DDL are arranged in a strict hierarchical fashion. Each module is described in terms of other modules of successively lower levels. Modules at the lowest level specify detailed operations at the functional or logic-gate level. Only the statements at the lowest level resemble ALGOL-like statements. Otherwise, a system

description resembles an outline with each "outline heading" naming a module to be described by what follows.

Duley and Dietmeyer^[12] have defined the operations required for a computer-aided design program to convert descriptions in DDL into Boolean and next-state equations. The work on the software system was reported to be in progress and may now be completed.

D. Register Transfer Language

Appearing in various forms^[13,14,15] Register Transfer Language (RTL) is fairly close to hardware and deals with modules at the register and gate levels. It is not well suited to the description of large, complex systems, but its ALGOL-like structure and clear concise means of defining control and timing make it attractive for designs of intermediate complexity. Although some efforts were made^[14] to provide automated analysis and synthesis available in this language, no apparent progress has been made within the last six years.

E. Computer Design Language

Y. Chu has suggested a formal means of describing digital systems through Computer Design Language (CDL)^[16]. His language resembles Register Transfer Language but is more concise and precise. As in RTL, parallel operations, timing, and control are all easily expressible. The language is to a great extent independent of hardware

technology and can be used to express either synchronous or asynchronous designs. The designer is free to organize his description in such a manner as to emphasize either the control, the sequencing, or the modular aspects of his design.

A description in CDL consists of two parts: statements giving names to hardware units and ALGOL-like statements describing the interaction of hardware units. Modules may be defined in terms of these ALGOL-like statements. Reference to such modules in other statements permits the designer to indicate hierarchical structures.

A description of the SCC 650 has been formulated^[17] in CDL. McCurdy and Chu have developed a translator which converts a CDL description into a set of Boolean equations^[18]. Their translator is written in MAD, and in order to simplify the programming effort, the CDL description input "was transliterated into one with a fixed format and written also in MAD language".

F. Design Oriented Language

Rouse describes the Design Oriented Language (DOL)^[19], a semi-graphical language intended to clarify the sequencing and control of complex digital systems. A form similar to CDL is used for the description of blocks at the lowest level.

The same design philosophy which underlies DOL was used by Rouse as the basis for the organization of a

comprehensive simulation program^[20]. That program is capable of performing in-depth simulation at the module and/or gate level. It includes capabilities for taking into consideration unequal time delays, indeterminate logic signals, the effect of faults and the possibility of races. The input description consists of fixed format lists of each module or gate type with its fan in and fan out, together with appropriate control cards. Efficient use of this simulator would appear to require that the design already be complete and, at least in principle, correct.

G. Other Automated Digital Design Efforts

In addition to the above, computer-aided design efforts at the computer-system level and at the gate level have gone on.

A number of programs are available to assist in the design and evaluation of computer systems. Computer Description Language (CDL1)^[21] is oriented toward specifying overall attributes such as speed, cost, compatibility, etc., in a formal manner. SODAS^[22] (System Oriented Design and Simulation) is a simulation language which includes the means of specifying the behavior of both the hardware and software of a system.

At the opposite extreme, a number of programs using more or less fixed-format input describing state tables or logic gate connections exist for a variety of purposes.

These include simulation, simplification, synthesis, hazard and race analysis, fault analysis, and diagnostic test generation^[23-27].

H. Summary and Comparison

Rouse^[19] makes a comprehensive comparison of APL, CDL, DDL, and RTL, including a single example expressed in each language. Using a different example, Pottinger^[17] has expressed it in each of the languages APL, CDL, and RTL.

Disregarding the use of a left arrow, right arrow, or equals sign, and other symbol conventions, statements at the lowest level in each design language resemble ALGOL-like statements. Ignoring the differences in symbolism, the available languages are nevertheless distinct in their abilities to express features such as parallel operation, timing and control, and hierarchical structure. Both CDL and DDL have these features. The main difference is that DDL places the hierarchical structure prominently while in CDL the control is usually, but not necessarily, more prominent. The general problem solving languages (APL and PL/1) do not provide satisfactory methods for specifying parallel operations or for specifying timing or control logic.

With the exception of the general problem solving languages, none of the reported efforts have included simulation at an early stage in the design process, and

none are used interactively to provide design assistance during the initial design phases. Direct simulation of designs in DDL, CDL, RTL, or DOL have not been possible heretofore. Simulation has involved manual translation into fixed format or the use of simulators operating on the output produced by synthesis programs available for some of these languages. Thus, despite the large number of efforts to provide designers with languages to formulate designs and efforts to reduce such descriptions to the level of Boolean expressions, the designer is still left with no adequate form of assistance to create a correct description in the first place.

The familiarity of ALGOL-like structures and the ability to clearly express control and parallelism has led to the choice of CDL as the description language for this work. Further justification for this choice is based on the ease with which others^[28,29,30] have learned to use IDDAP. A few additions to CDL have been made based on experience gained using CDL and on features found in other languages.

III. SYSTEM DESCRIPTION OF IDDAP

The block-diagram in Figure 1 describes the overall structure of IDDAP. The block labeled MESSAGE HANDLER is a small sub-program which is responsible for all communication between IDDAP and the user. During the development of IDDAP, the Message Handler used the system card-reader and line printer for input and output. By replacing this single program with another version, IDDAP could be made to communicate with any connected input/output devices. Specifically, an Assembler Language program was used to communicate with an IBM 2741 terminal via IBM's Basic Telecommunication Access Method (BTAM) software. The block labeled USER therefore refers to whatever input/output device is currently in use for communication with IDDAP.

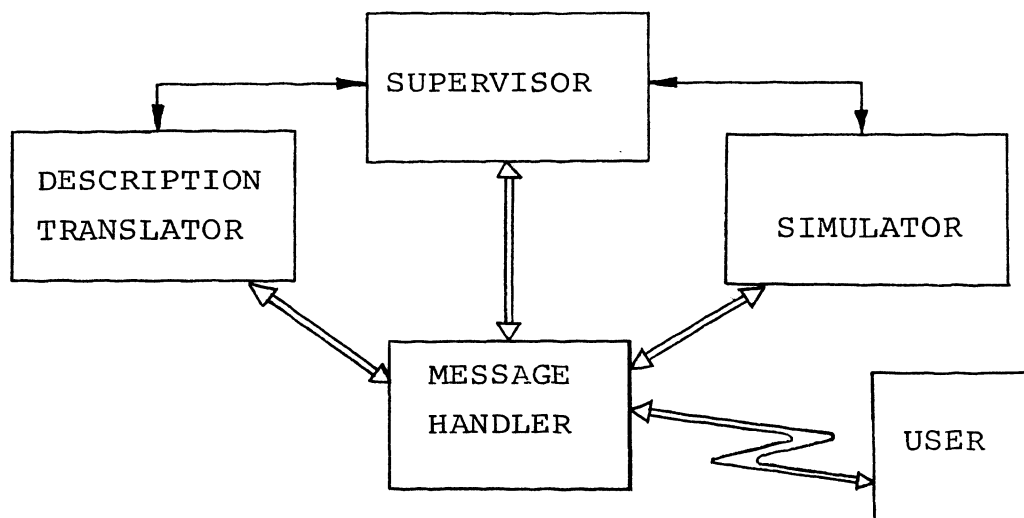


Figure 1. IDDAP System Block Diagram

The Supervisor's function is to invoke the Description Translator or the Simulator and perform other requested operations. Upon the initial entry, the Supervisor automatically invokes the Description Translator. Thereafter, the Description Translator may be invoked for the purpose of changing or updating the description or to start a new description from the beginning. Once invoked, the Description Translator remains in control until the supervisory mode is explicitly requested by the user.

Upon the user's request, the Supervisor invokes the Simulator. At or before requesting simulation, the user specifies details of how the simulation is to be performed. Once the Simulator is entered, the user has only limited control over IDDAP. At regular intervals, the length of which may be set by the user or by default, the Simulator will ask whether or not it should continue with the simulation. A negative response returns control to the Supervisor at which time the user is again in full control.

Operation normally consists of the following steps.

1. The user enters statements describing his design. After each statement IDDAP will type the next line number if the previous statement was free of errors. If the statement contained an error, IDDAP responds with a diagnostic message and then retypes the previous line number. As each statement is entered, entries are made in appropriate internal tables. This data will be used later to perform the tasks which the user requests.

2. Upon completion of the description, the supervisory mode is entered. Although the user may now call upon IDDAP to perform any of the available functions, the usual sequence would be to issue commands necessary to prepare for simulating his design. These could include initializing selected logic signals, indicating the expected number of iterations, and stating whether or not waveforms are to be plotted.

3. After specifying the constraints for the simulation, the user may instruct IDDAP to begin simulating. The simulation will be carried out based on the data in some of the tables which were formed as the user entered his description. Certain errors which were undetectable earlier may occur while simulating the design. Such errors result in a return to the supervisory mode accompanied by a diagnostic message.

4. Following a period of simulation, IDDAP will again be in the supervisory mode. Based on the results of the simulation, the user may request that the Description Translator be reentered for the purpose of making changes to his description. In order to help isolate problems, the user may instead request cross-reference information. In order to facilitate the study of his description, the user may request that his description be reorganized according to specified control variables. Eventually the user will probably wish to

return to steps two or three. If utterly confounded, he can return to step one and start from the beginning.

5. At the end of a design session with IDDAP, the user may wish to save some of his work or produce hard-copy versions. These options are also included in the Supervisor.

As has already been suggested, communication with IDDAP can be divided into two categories, description statements and control commands. These two categories are further sub-divided in the next section which describes the means of communicating with IDDAP in detail.

IV. COMMUNICATING WITH IDDAP

In the discussion to follow, square brackets [] are used to denote optional items. The notation []... means that the item(s) contained inside the brackets may be repeated zero or more times. Italics will be used to represent categorical items which are to be replaced by specific items in actual statements.

The terms "name" and "constant" are used in the discussion of both description statements and control commands. A name is a string of alphanumeric characters with no intervening blanks. The first character must be alphabetic and only the first eight characters are significant. A constant may be specified as a decimal, binary, octal, or hexadecimal integer as indicated by the absence of a suffix, the suffix character B, the suffix character K, or the suffix character X, respectively. The first digit of a constant must be numeric. The letters A through F are used for the hexadecimal digits 10 through 15.

Comments and/or blanks may be inserted anywhere that a non-alphanumeric symbol is allowed. A comment must be enclosed in dollar signs, e.g., \$THIS IS A COMMENT\$.

A. Description Statements

The description of a digital system for IDDAP consists of two types of statements. The first type is defining

statements used to assign names to flip-flops, registers, combinational logic outputs, and so forth. The second type of statements resemble executable ALGOL-like statements and serves to specify how the various logic components are to interact.

1. Type-One Statements

Defining statements have the following general form
.type. name[qualifications] [,name [qualifications]]...
 The *type* is enclosed in periods and may be abbreviated by its first three letters. It refers to the kind of hardware being named. Valid types are REGISTER, SUBREGISTER, MEMORY, INPUT, CLOCK, DECODER, TERMINAL, and NETWORK. If a valid type was in effect on the preceding line, then *.type.* can be omitted if the current line defines items of the same type.

Two additional type-one statements, *.LABEL.* and *.END.*, do not define logic components. A label may be defined for the convenience of the user in referring to points within the description. The *.END.* statement is used to mark the end of the description and also causes a return to the supervisory mode.

In the following discussion the *length/position* specification referred to has the form $([n_1 -] n_2)$. If n_1 is omitted, then n_2 is the length in bits and one-origin indexing is assumed. That is, the bits are assumed to be numbered in left-to-right order from 1 to n_2 . When n_1 is specified the bits are numbered n_1 to n_2 in left to right

order and the length is $n_2 - n_1 + 1$. The current implementation restricts the length to a maximum of 32 bits.

Negative-origin indexing is not allowed. The maximum value of n_2 is 999, and n_1 must be less than or equal to n_2 .

A register, input, and clock are all defined in the same manner. The main difference between them is a conceptual one. A register is usually thought of as a memory device consisting of one or more bits. An input is usually regarded as a signal whose source is external to the system and whose value cannot be changed by the system itself. One may think of a clock as a device internal to the system but which cannot be affected by operations within the system.

The current implementation makes no distinction between these three types, but an appropriate selection is an aid to documentation. In many cases it is convenient to have some statements which assign values to inputs or control clocks in order to make the description self-contained and operable in a continuous or iterative fashion. Statements assigning values to inputs may be viewed as representing an abbreviated description of the external device supplying those inputs.

The defining statement for these three types has the following form

```
.REGISTER.  
.INPUT.  name[length/position] [,name[length/position] ]...  
.CLOCK.
```

The *name* and *length/position* are as described above.

Omission of the *length/position* specification is equivalent to specifying *name(1)*.

The form used to specify a subregister is the same as for a decoder but they denote quite different items.

Their general form is

```
.SUBREGISTER.  name = base-name[length/specification] [, ]...
.DECODER.
```

The *length/position* specifies the portion of *base-name* which is to be associated with the name being defined. If a subregister is being defined, reference to *name* will in effect be a reference to the indicated bits of the base. A decoder is a combinational logic network having n inputs and 2^n outputs. All 2^n outputs are 0 (or False) except the one corresponding to the binary encoding of the inputs. A reference to a decoder output requires a numeric suffix to specify one of the 0 through 2^n-1 decoder outputs. Thus, for a decoder, *length/position* defines which n bits of the base are being decoded by *name*. In both cases, if the *length/position* specification is omitted, then the entirety of *base-name* is assumed to be specified.

A memory definition has the form

```
.MEMORY.  name( $n_1, n_2$ ) [, name( $n_1, n_2$ )]...
```

The number of bits per word is given by n_1 , and n_2 specifies the total number of words. Both must be simple integer constants. Zero-origin indexing is assumed for memory words. Individual bits cannot be referenced directly.

For the current implementation the total memory defined cannot exceed 128 words. Memories declared under a single `.MEMORY.` heading will be assigned sequential locations in real memory. Subsequent use of `.MEMORY.` will cause the simulated memories to overlay memories previously defined.

Thus for the definitions

```
.MEMORY.  A(ni, 16), B(nj, 32)
.MEMORY.  Z(nk, 32)
```

references to `A(17)`, `B(1)`, or `Z(17)` in simulated memory all refer to the same position in real memory.

The declaration of a terminal includes a logic expression specifying the function to be performed. It has the form

```
.TERMINAL.  name = expression
```

The rules for expressions are discussed in the next sub-section.

The heading `.NETWORK. name[length/position]` is used to specify a more complex logic module having any number of outputs. Following the line on which the network is named, as many type-two statements as are needed may appear. All the rules given below for type-two statements apply to the body of a network's description. The group of statements describing a network are terminated by the appearance of a statement beginning with a `.type.`

2. Type-Two Statements

Type-two statements have the general form

$[[condition\ prefix]:] unit_1 [,unit_2]\cdots [@]$

The *condition prefix* controls the operations in the remainder of the statement as well as the operations in all subsequent statements up to the next statement containing a condition prefix. The condition prefix consists of one-bit logic signals connected by asterisks (*) and optionally preceded by not-signs. All of the items in the condition prefix must be true before the operations controlled by it can take place. A colon alone may be used to introduce unconditional operations. When the operations under control of a condition prefix do take place, they all occur simultaneously. The *.type.* used for defining names and the condition prefix have one important property in common; each continues in effect for subsequent lines of the description until the appearance of another *.type.* or *condition prefix*.

The *units* of the statement are ALGOL-like structures of one of the following types:

1. An assignment statement of the form

identifier = expression

2. An IF statement of the form

IF *expression* THEN *action*

where *action* may consist of one or more *units* of any type other than an IF statement. All units following

the THEN up to the end of the statement are regarded as a group whose operations are under control of the IF clause.

3. A transfer statement of the form GO TO *label-name*. This statement does not represent any logic but may be used to assist the designer in formulating control and to cause iteration.

4. A statement of the form DO *network-name* which is conceptually similar to the CALL *subroutine-name* statement in most algorithmic languages.

The optional at-sign (@) at the end of the statement may be used to indicate that the statement is to be continued on the next line. There is no specific limit to the number of such continuation lines.

Expressions in IDDAP are identical in form to expressions in other algorithmic languages. The symbols * and + are used to denote the operations of logical AND and inclusive OR, respectively. For multi-bit operands, these logical operations are applied bit for bit. For operands of unequal length, the shorter is extended on the left with zeros, and the result has the length of the longer. The prefix not-sign (→) may be used in an expression in the same way that a prefix minus sign is used in other algorithmic languages. The relational operators >, <, and = all have the usual meaning. The result of any of these comparison operations is a one-bit intermediate logic signal.

All other available operations are specified with keywords enclosed in periods. Those operations include ADD, SUBTRACT, MULTIPLY, DIVIDE, REMAINDER, LSHIFT, RSHIFT, LCIRCULATE, RCIRCULATE, CONCATENATE, COUNT, XOR (exclusive-or), and DISPLAY and may all be abbreviated by their first three letters.

The ADD, SUBTRACT, MULTIPLY, DIVIDE, and REMAINDER operators enable the user to perform integer arithmetic. For addition and subtraction, the 2's-complement form of negative numbers is assumed. The built-in operands .CRO. and .OVF. may be referenced to ascertain if there was a carry into the position to the left of the left-most bit or if overflow occurred. The operands of MULTIPLICATION, DIVISION, and REMAINDER are assumed to be positive integers. The .REMAINDER. operator gives the value of its left argument modulo its right argument.

For the shift operations, LSHIFT, RSHIFT, LCIRCULATE, and RCIRCULATE, the first letter (L or R) denotes left or right. The operand or expression following the shift operator specifies the number of bit positions that the left operand is to be shifted.

Use of the .XOR. operator yields the exclusive-or of its operands.

The .CONCATENATE. operator provides the means of specifying a composite register whose length is the sum of its operand's lengths.

The `.COUNT.` operator is similar to the `.ADD.` operator except that `.CRO.` and `.OVF.` are unaffected.

The `.DISPLAY.` is not a logical operator but is a means of causing the simulator to display intermediate results during the course of simulation. Syntactically, it functions like the equals sign in an assignment statement. The left operand is an integer in the range one through sixteen which specifies the arithmetic base in which the operand or expression on the right is to be displayed. Specifying one for the base results in the display of either the word "TRUE" or "FALSE". As an illustration of the use of `.DISPLAY.`, the statement `8 .DISPLAY. ACCUM` in Example 2 of Chapter V causes the value of the register ACCUM to be printed in base-8 (Octal).

The term "identifier" used in the preceding discussion refers to either a simple name or a qualified name. Names of multi-bit structures other than decoders may be qualified using parenthesized subscript notation consisting of a specification of the form " $(constant_1 - constant_2)$ " or of the form " $(position-expression)$ ". The first form can be used to reference a multi-bit subsection while the second form is useful for referencing a varying bit position. Memory references are always qualified using the second form to specify a memory word. Names may also be qualified with a numeric suffix attached without intervening blanks. For a decoder reference, the suffix is mandatory and refers to one of the 0 through $2^n - 1$ output lines.

For other multi-bit structures, the suffix refers to a one-bit signal according to the bit position numbering defined for that name.

In addition to the statements already discussed, the description translator also recognizes five one-character commands. The first of these is the break (`_`) which causes an immediate return to the supervisory mode without formally ending the description. The other four are editing characters which can be used to control the current line number and to make corrections during the entry of a description.

Entering a slash (`/`) will decrement the line count by one without affecting any previously entered statements. Typing an equals sign (`=`) reestablishes the statement stored at the current line number and advances the line count by one. If the statement at the current line is uncertain, a question mark (`?`) may be typed to have that line displayed without reestablishing it or changing the line count. After displaying the line, the user may enter a new statement to replace that line or type an appropriate edit character. An ampersand (`&`) may be typed to indicate that the next statement to be entered is to be inserted ahead of the statements beginning at the current line.

B. Control Commands

An additional set of statements is used for communicating with the supervisor. These control commands are

relatively simple and are of fixed format requiring little punctuation other than intervening blanks. With the few exceptions noted, the command words may be shortened to their three (or more) leading letters. The control commands listed below are arranged approximately in the order that one might normally use them.

DESCRIPTION

The description translator is invoked for the purpose of entering a new description.

SET [*name = constant*]...

The logic unit "name" will be set to the value specified by the constant prior to simulation.

LOOP *number*

This command allows the user to specify the maximum number of iterations which will take place before the simulator will ask whether or not to continue with the simulation.

WAVES [*n name*]...

This command indicates that during simulation information necessary to produce waveform timing diagrams is to be gathered. The optional list specifies that the logic signal *name* is to have its waveform plotted as the n^{th} waveform on the page. Default is to plot the first fifteen names defined in the order of their definition. A slash (/) may be entered in place of *name* to specify that the n^{th}

through 15th lines are not being used. The numbers *n* need not appear in order, and any lines not specified will remain as they were.

NOWAVES

This command indicates that waveforms will not be needed (abbreviated NOWAV).

CLOCKED *name*

The simulation is to be run in "clocked mode" controlled by the logic signal *name*.

NOCLOCK

This command specifies that no signal is to be used as a "clock", and is the default condition (abbreviated NOCLO).

DEFINE *number*

The current values of all logic signals will be stored for later use. The three such sets of values which may be stored are identified by the *number* (1, 2, or 3) specified. The values of the logic signals are left unchanged.

ESTABLISH *number*

The set of values stored by use of a DEFINE command are used to re-initialize the logic signals. The *number* (1, 2, or 3) indicates which set is to be used. The set of values stored is left unchanged.

In the discussion of the following three commands, *point* refers to a position in the description. It may be

either a line number or the name of a label, terminal, or network. For these commands the following defaults apply:

1. If no *points* are specified, the first line and last line are assumed for $point_1$ and $point_2$ respectively.
2. If only $point_2$ is specified, a comma must be used in place of $point_1$ and $point_1$ is assumed to be line one.
3. If only $point_1$ is specified and is a terminal or network, then $point_2$ is assumed to be the next statement after the end of the terminal or network definition. Otherwise, $point_2$ is taken to be the last line.

SIMULATE [*point*₁] [*point*₂]

Simulation will begin at $point_1$ and continue up to but not including $point_2$. If $point_1$ does not refer to a terminal, network, or type-two statement, then simulation begins at the next type-two statement which is not a part of a terminal or network definition. If $point_2$ is never reached, a return to the supervisory mode will occur if the .END. statement is reached, or when the user gives a negative response to the simulator's question, "Shall I Continue?".

LIST [*point*₁] [*point*₂]

Description statements between the limits indicated will be listed.

CHANGE *point*

The description translator is invoked for the purpose of making changes. The current line number will be as specified by *point*.

DISPLAY WAVES

The information accumulated for waveforms will be displayed. Note that if the buffers for this display become full during the course of simulation, waveforms will automatically be displayed, and simulation resumed.

DISPLAY [*base*] *name* [*name*]*...*

The values of the named items will be displayed in the base specified. The base may be any integer from one through sixteen. If *base* is omitted, decimal is assumed.

DISPLAY FLOAT *mantissa exponent*

The value of the integer *mantissa* will be multiplied by two raised to the two's complement integer *exponent*, and the result will be displayed as a floating point decimal number with decimal exponent.

UPDATE

This command causes any projected values of logic signals to become the current values.

REFERENCE *text*

A list of statement numbers will be printed in which *text* appears. *Text* cannot be longer than 20 characters.

REORGANIZE/ [*text* /]...

The description will be reorganized as follows:

1. All logic-defining statements in their original order will appear first.
2. All groups of statements for which the first (or next) *text* is contained in the controlling condition prefix (provided these statements have not already been included).
3. Repeat number 2 until the list of *texts* is exhausted.
4. All remaining statements.

Any labels or comments are moved along with the statement which they precede. Note that if no *texts* are listed the effect is to merely collect all defining statements together at the beginning of the description.

ARRANGE [*line-number*]...

The description will be arranged in the order specified in the line-number list. Any lines not included in the list will follow the specified lines.

FINISHED

Used to sign off. The terminal will become inactive.

In addition to the above listed commands, the following software debugging and installation-dependent hard-copy options are included. These commands cannot be abbreviated.

TRACE *number*

Number may be an integer from zero through five. Zero specifies that no program check-out information is to be printed and is the normal default value. Numbers one through five specify successively more print-outs. The information is printed on the System Line Printer.

PUNCH

A copy of the description will be punched on cards.

DUMP

Causes a listing of all IDDAP's internal tables to be printed on the system line-printer. This action also occurs prior to an abnormal termination.

COPY / NOCOPY

If COPY was last invoked, then the user's entries are copied on the system line-printer. If NOCOPY was last invoked, then no such copying occurs.

SYSREAD

This command allows the user to read in cards which were submitted in batch-mode. If a card contains a control command, the command will be displayed on the user's terminal before being acted upon. If the user enters an equals sign, the command will be processed. Any other entry causes IDDAP to discontinue reading cards and to request a control command from the terminal.

C. Simulator Output

Communication with the simulator itself is limited. The simulator will display values as directed by the .DISPLAY. statements imbedded in the description and may display waveforms if the buffers for accumulating waveform information become full. The only input from the user is his response to the question "Shall I Continue?" to which he answers "Y" or "N". A negative answer ("N") results in a return to the supervisory mode.

V. USING IDDAP

The approach taken in this chapter is to present specific examples illustrating important features and general rules of IDDAP. In each sample dialogue, the computer's response is the first thing on each line. If a user's entry is required, an underbar (_) is the last character typed by the computer, and the user's response follows.

A. Fundamental Concepts

The first example represents a complete, although short, design session with IDDAP. Future examples will not always represent complete design sessions, but will consist of plausible segments of a longer session.

For the first example, the design objective is to use a control signal Q to control the transfer of one register into another. Specifically, if the control signal is one (or true) the ACCUM register is to be copied into the INDEX register. If Q is zero, the opposite transfer is to occur.

In the dialogue for Example 1, the first line defines the two 12-bit registers ACCUM and INDEX. Since line two contains neither a *.type.* nor a condition prefix, the *.REGISTER.* from line one continues in effect. Thus Q is defined to be a register and, since no length is explicitly

Example 1:

```

1_.REGISTER.  ACCUM(12), INDEX(12)
2_           Q
3_.LABEL.    B1
4_   Q:      INDEX = ACCUM
5_  ~Q:      ACCUM = INDEX
6_.END.
**_SET      ACCUM = 0      INDEX = 5      Q = 0
**_SIMULATE B1
...
**_DISPLAY  ACCUM
ACCUM      = 0005
**_DISPLAY  2 ACCUM INDEX Q
ACCUM      = 000000000101
INDEX      = 000000000101
Q          = 0
**_FINISHED

```

specified, it is a one-bit register. It would also have been correct to have placed Q in the first statement.

The label defined in line 3 is not essential, but provides a convenient reference point for use later in the example.

The condition prefix in line 4 is Q. Therefore, the transfer of ACCUM into INDEX will occur only if Q is one. Since the condition on line 5 is ~Q, the indicated transfer occurs only if Q itself is zero.

The .END. in line 6 denotes the end of the description. IDDAP enters the supervisory mode and indicates that it is ready for a control command by typing "***_". In response, the user's first command is to initialize each of the registers.

The next control command instructs IDDAP to begin simulating the design at the statement labeled B1. Since no end-point is specified, simulation will proceed up to the end statement. In this example the same effect could be achieved by specifying simply SIMULATE. Specifying either SIMULATE B1 6 or SIMULATE 4 6 would also produce the same effect.

After the simulation, ACCUM is displayed and seen to contain the correct result. Since no base was specified, decimal was assumed. The second DISPLAY command illustrates the capability to present results in other bases, in this case Binary. The command FINISHED was typed to sign-off.

The second example is similar to the first. A six-bit register has been added and if that register contains the value three, then when Q is zero the two registers are to be interchanged. This situation calls for simultaneous (or parallel) transfers of data between the two registers.

The first line of Example 2 is the control command requesting that the description translator be entered. This command is needed except for the first design description of the session.

The description follows the pattern of Example 1. In this example all four registers are defined in a single statement. A misspelling in line 2 resulted in the error message. On the second attempt, line 2 was entered successfully.

Example 2:

```

**_DESCRIPTION
1_.REGISTER. ACCUM(12), INDEX(12), Q , B(6)
2_  Q:      INDES = ACCUM
ERROR 115, IN TEXT INDES= -- Identifier is undefined
2_  Q:      INDEX = ACCUM
3_  ^Q:     ACCUM = INDEX          $DO THIS ANYWAY!$
4_      IF B=3 THEN INDEX = ACCUM
5_.END.
**_SET ACCUM=15K INDEX=23K Q=1 B=0
**_SIMULATE
...
**_DISPLAY 8 ACCUM INDEX
ACCUM   = 0015
INDEX   = 0015
**_CHANGE 5
5_&
5_      : 8 .DISPLAY. INDE *** IGNORED ***
        : 8 .DISPLAY. ACCUM, 8 .DISPLAY. INDEX
6_
**_LIST
1_.REGISTER. ACCUM(12), INDEX(12), Q , B(6)
2_  Q:      INDEX = ACCUM
3_  ^Q:     ACCUM = INDEX          $DO THIS ANYWAY!$
4_      IF B=3 THEN INDEX = ACCUM
5_      : 8 .DISPLAY. ACCUM, 8 .DISPLAY. INDEX
6_ .END.
**_SET ACCUM=134K INDEX=35K Q=0 B=3
**_SIMULATE $IT SHOULD EXCHANGE THE REGISTERS$
ACCUM   = 0035
INDEX   = 0134
...
**_

```

When Q is zero, the transfer of Index into ACCUM occurs regardless of the value of B. Note the comment to this effect enclosed in dollar-signs.

Since line 4 contains no colon, it is under control of the condition prefix of the prior line, line 3. Since all operations under a single condition prefix occur simultaneously, if at all, a favorable comparison between

B and the constant 3 will cause the other half of the transfer required to exchange the contents of ACCUM and INDEX.

The end statement ends the description and causes a return to the supervisor which in turn requests a control command. The end statement also serves to end the last condition-prefix-block of the description. In general, a condition-prefix-block is ended by either another condition prefix or the .type. of a defining statement.

To test the design just described, the first control command following the description initializes the registers preparatory to issuing the simulate command.

Upon completing the simulation (indicated by "...") the user directed IDDAP to display the value of the registers, which are correct for the initial values specified. Probably realizing that further testing was needed to verify the design completely, the user chose to make the displaying of the results automatic.

The next sequence of entries resulted in the change necessary to automatically display the results. After the description translator was reentered using the CHANGE command, the entry of the ampersand (&) specifies that the following line is to be inserted ahead of what was formerly in line 5. If the colon had been omitted on the new line then the results displayed would be those values prior to the performance of the other operations in the

containing condition-prefix-block. The underbar (_) entry returns control to the supervisor.

One of the features included in the message handler allows the user to delete the line being typed. He does this by pressing the "ATTENTION" key. When the attention key is depressed, the message handler replies with "*** IGNORED ***" and then allows the user to reenter the statement on the next line.

After listing the revised description, the user proceeded to again initialize the registers and to simulate the design. This time the results were automatically printed.

B. Clocked Simulation

For the two examples discussed above, there was no signal acting as a clock to synchronize transfers. Upon completion of each condition-prefix-block all signals which had been specified to change acquired their new values simultaneously. If a clocked simulation is specified, then instead of signals acquiring their new values at the end of each block, the signal designated as the clock is examined. Only if the clock-signal has been specified to change will the signals acquire their new values. If the clock-signal is not about to change then any other changes remain pending until the clock finally changes.

Descriptions of clocked mode systems will usually contain a statement like line 2 of Example 3. Example 3 may be thought of as the description of a clock which changes state every time the simulation of the description is iterated, provided A is one. Figure 2 shows one possible hardware configuration which could correspond to the description in Example 3. The input signal A in Example 3 and Figure 2 may be thought of as an ON/OFF switch for the clock. The fact that B was defined as a CLOCK is important only for documentation purposes. Both A and/or B could just as well have been defined as registers.

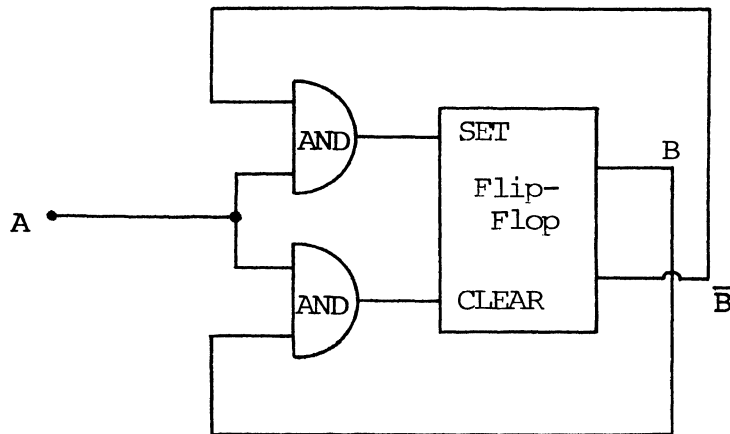


Figure 2. Logic Diagram of the Circuit Described in Example 3

Example 3:

```

**_DESCRIPTION
 1_.INPUT.  A
 2_.CLOCK.  B
 3_.LABEL.  FIRST
 4_   A: B =  $\bar{B}$ 
 5_   : GO TO FIRST
 6_.END.
**_CLOCKED  B
**_WAVES  1 A  2 B  3/
**_LOOP   5
**_SET    A=1
**_SIMULATE
SHALL I CONTINUE? N
...
**_SET    A=0
**_SIMULATE
SHALL I CONTINUE? N
...
**_DISPLAY WAVES

A          _____
B          | | | | | | |
          **_

```

The preparation for the simulation of this description illustrates several of the control commands not used before in this chapter. That this is to be a clocked simulation, clocked by B, is specified in the first control command following the description. The next command states that a waveform timing diagram is to be presented. The list following the command WAVES states that A and B are to be presented on lines 1 and 2 respectively, and that lines 3 through 15 are not being used.

The maximum number of iterations is set by the LOOP command. Thus, after the simulate command, the simulation will be iterated five times before the message "Shall I Continue?" is printed. The negative reply "N" returns control to the supervisor in order to change the value of A.

After the second simulation, the command to display the waveforms was given. If the waveforms' buffers had become full during the simulation, they would have automatically been displayed and the simulation would continue.

The fact that the waveforms reflect only the first simulation consisting of five half-cycles is a result of specifying clocked mode. Since the clock was turned off in the second simulation, nothing happened and no additions to the waveforms were made.

Example 4 illustrates a somewhat more sophisticated example of clocked mode simulation. In it, an up/down counter described in lines 7 and 8 is controlled by the statements in lines 9 and 10. If UP is initially one and the counter is initially zero, it will count up until statement 9 causes it to reverse and count down. When the count reaches zero going downward, statement 10 will not cause further iteration.

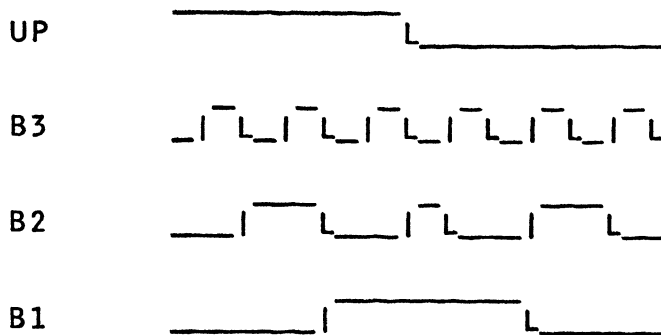
Since the LOOP command was not issued, the loop size is ten by default. Thus the user responds Y (for yes) to the simulator's first "Shall I Continue?" message.

Example 4:

```

**_DESCRIPTION
1_.REGISTER.   COUNTER(3)
2_.INPUT.      UP
3_.SUBREGISTER. B3=COUNTER(3), B2=COUNTER(2)
4_.           B1=COUNTER(1)
5_.TERMINAL.   DOWN = ~UP
6_.LABEL      RPT
WARNING 142, IN TEXT LABEL -- TERMINATING "." INSERTED
7_   UP:      COUNTER = COUNTER .ADD. 1
8_   DOWN:    COUNTER = COUNTER .SUB. 1
9_   :        IF COUNTER=6 THEN UP = 0
10_  :        IF ~(DOWN*(COUNTER=0)) THEN GO TO RPT
11_.END.
**_WAVES 1 UP 2 B3 3 B2 4 B1 5/
SAVE WAVES IN BUFFER? N
**_SET UP=1 COUNTER=0
**_SIMULATE
SHALL I CONTINUE? Y
...
**_DISPLAY WAVES

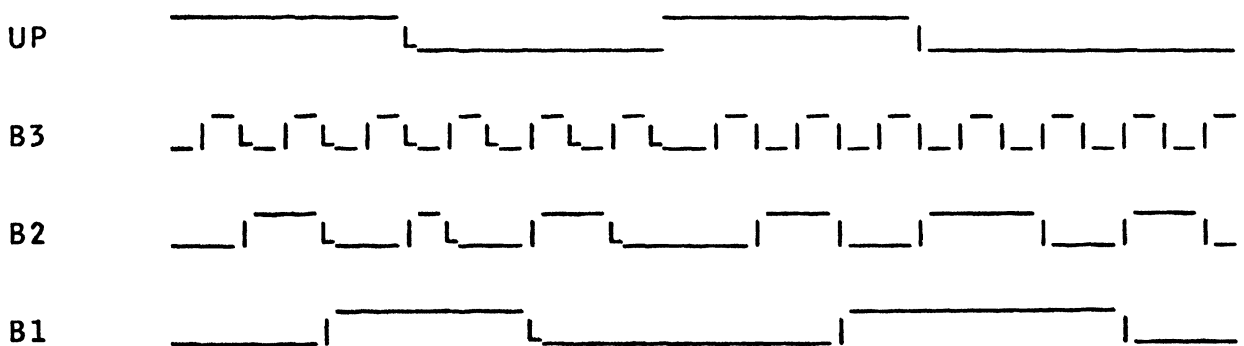
```



```

**_CLOCKED COUNTER
**_SIMULATE *** IGNORED ***
   SET UP=1 COUNTER=0
**_SIMULATE
SHALL I CONTINUE? Y
...
**_DISPLAY WAVES

```



The first simulation, and hence the first part of the waveforms, are in non-clocked mode. The second simulation was performed in clocked mode, clocked by changes in the counter's value. The discontinuity in the waveform for UP is a result of changing its value externally to the simulator by use of the SET command.

C. Additional Features

Although an exhaustive treatment of each type of statement and control command discussed in Chapter III is impractical, a few deserve illustration. The ability to specify rather complex operations as terminals is illustrated in Example 5.

Threshold gates may be indicated by making use of the relational operators greater than ($>$) and less than ($<$). Example 5 illustrates such use in what may be regarded as a crude description of a neuron. The output A responds immediately if the neuron's input, IN, is greater than two. It responds only once for every two time-increments during which its input remains one.

The 24-bit register SEQUENCE is used to provide a series of inputs by shifting it two places to the right after each iteration.

The use of networks apart from the use of the DO statement is possible, but together they may be used to represent a digital system in a modular fashion. Example 6

Example 6:

```

**_DESCRIPTION
1_.REGisters. INSTRUCTION(8), DATA(8)
2_.SUBregister. OPCODE=INSTRUCTION(1-4)
3_          ADDRESS=INSTRUCTION(5-8)
4_.REGister.  MEMBUF(16), LOCATION(3), SAME
5_.MEMory.    MEMORY(16,8)
6_.REGister.  READ,  TIMMER(3)
7_.TERminal.  WRITE = ~READ
8_.DECoder.   T = TIMMER
9_          $ The description of the memory module follows.$
10_.NETwork.  ACCESS
11_ T2:  IF LOCATION=ADDRESS(1-3) THEN SAME=1
12_ T1:  SAME=0
13_ ~SAME*T3: MEMORY(LOCATION) = MEMBUF $Rewrite old $
14_          $      location $
15_ ~SAME*T4: LOCATION=ADDRESS(1-3)
16_ ~SAME*T5: MEMBUF=MEMORY(LOCATION) $Get new words$
17_ WRITE*T7: IF ADDRESS(4)=1 THEN MEMBUF(9-16)=DATA
18_          IF ADDRESS(4)=0 THEN MEMBUF(1-8)=DATA
19_ READ*T7:  IF ADDRESS(4)=1 THEN DATA=MEMBUF(9-16)
20_          IF ADDRESS(4)=0 THEN DATA=MEMBUF(1-8)
21_          $ End of memory module description $
22_.TERminals. LDA=(INSTRUCTION=6)
23_          LDX=(INSTRUCTION=2)
24_          STA=(INSTRUCTION=3)
25_          STX= INSTRUCTION=1
26_.LABel. CPU $This is the computer module$
27_          : DO ACCESS
28_ T1:      READ=1
29_          IF STA THEN DATA=ACCUMULATOR
ERROR 115, IN TEXT LATOR; -- Identifier is undefined
29_.REGister. ACCUMULATOR(8), INDEX(8)
30_ T1:      IF STA THEN DATA=ACCUMULATOR
31_          IF STX THEN DATA=INDEX
32_ T2:      IF STA+STX THEN READ=0
33_ T7:      IF LDA THEN ACCUMULATOR=DATA
34_          IF LDX THEN INDEX=DATA
35_ T0:      $Get next instruction. To be designed later. $
36_          :      TIMMER = TIMMER .COUNT. 1
37_          IF ~(TIMMER=0) THEN GO TO CPU
38_.END.

```

having to know in advance the details of the memory actually in use.

For the purposes of this example, a CPU is postulated which has a word size of eight bits and which has only

four instructions. Those instructions are Load Accumulator (LDA), Load Index (LDX), Store Accumulator (STA) and Store Index (STX). The memory module in this example has 16-bit words and is to be used in such a way as to avoid unnecessary stores and fetches. The use of comments in this example should make it self-explanatory.

VI. THE SOFTWARE FOR IDDAP^[31]

The overall structure of IDDAP was discussed in Section III and illustrated in Figure 1. The purpose of this section is to provide further details on the programs which make up the Interactive Digital Design Assistance Package. Except for the use of Assembler Language for the Message Handler, all programs were written in PL/1 and were run on a Model 360/50 IBM Computer.

There are three main programs corresponding to the blocks in Figure 1 labeled SUPERVISOR, DESCRIPTION TRANSLATOR, and SIMULATOR. In addition, a number of short subprograms are used by all three main procedures to perform frequently needed operations.

The supervisor is the main procedure and is shown diagrammatically in Figure 3. Its function is to interpret and act upon control commands.

The initializing which the supervisor performs on its first entry includes assigning values to the various arrays. It also includes initialization of the remote terminal. The first communication must be a message from the user which is a part of the initialization process. That message is not processed.

Having completed the initialization, the supervisor invokes the description translator which responds by typing "1_" -- the number of the first line.

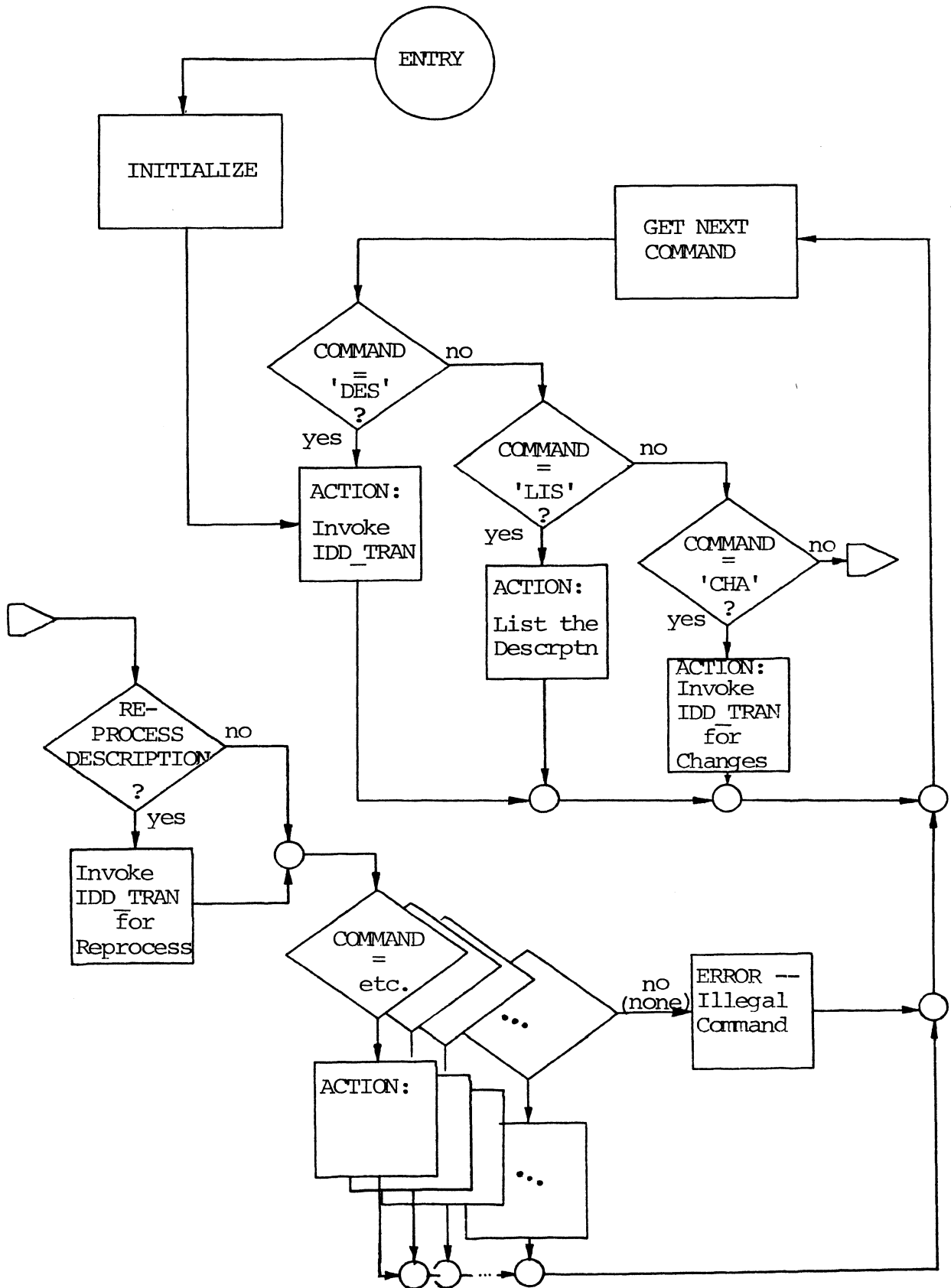


Figure 3. Block Diagram of Supervisor

When control is returned to the supervisor, the user may issue any of the available control commands. The first alphanumeric string in the control command is obtained using the external function subprogram NEXT_ID. The supervisor then proceeds to "sift" through a number of IF statements until it finds the one corresponding to the command given. This process is shown in an abbreviated fashion in Figure 3. This method of determining which control command was given makes it a simple matter to add new control commands to the repertoire. After the command has been identified and the associated operations performed, a new control command is requested.

Due to use of the CHANGE command, it may be necessary to invoke the description translator in order to reprocess the description. If this reprocessing is necessary, it occurs without explicit knowledge of the user other than a pause of less than 0.1 seconds per line of description. It is possible that a change to the description may result in an error in some part other than that which was changed. If such an error occurs, the user will be informed and invited to correct the error. If reprocessing is necessary, it must be carried out prior to any commands which may depend on the description being correct, consistent and complete. Thus, if the command is not the DESCRIPTION, CHANGE, or LIST command, a check is made to determine if reprocessing is necessary.

The description translator, Figure 4, is invoked by the supervisor. The organization may be thought of as having three main parts along with statements to determine the type of structure being processed. The three main parts are represented by the blocks labeled PROCESS TYPE-ONE STATEMENT, PROCESS CONDITION PREFIX, and PROCESS TYPE-TWO STATEMENT.

The block which translates type-two statements is the most complex portion of IDDAP. It applies the precedence rules, analyzes syntax and context of the statement, and generates entries in the "M-table". The M-table represents the operations specified in the description in a form close to machine language and is the primary table used during simulation.

The simulator, Figure 5, is actually two nested procedures. The external, outer procedure is invoked by the Supervisor. After a small amount of initialization, the inner procedure is invoked. In order to perform each operation, the current value(s) of the operand (or operands) are first obtained. If an operand is a terminal or network then the inner procedure recursively invokes itself. When the inner procedure is recursively invoked to evaluate a terminal or network, the activity previously in progress is temporarily suspended. Upon return from a recursive entry, the suspended activities are resumed as though they had never been interrupted.

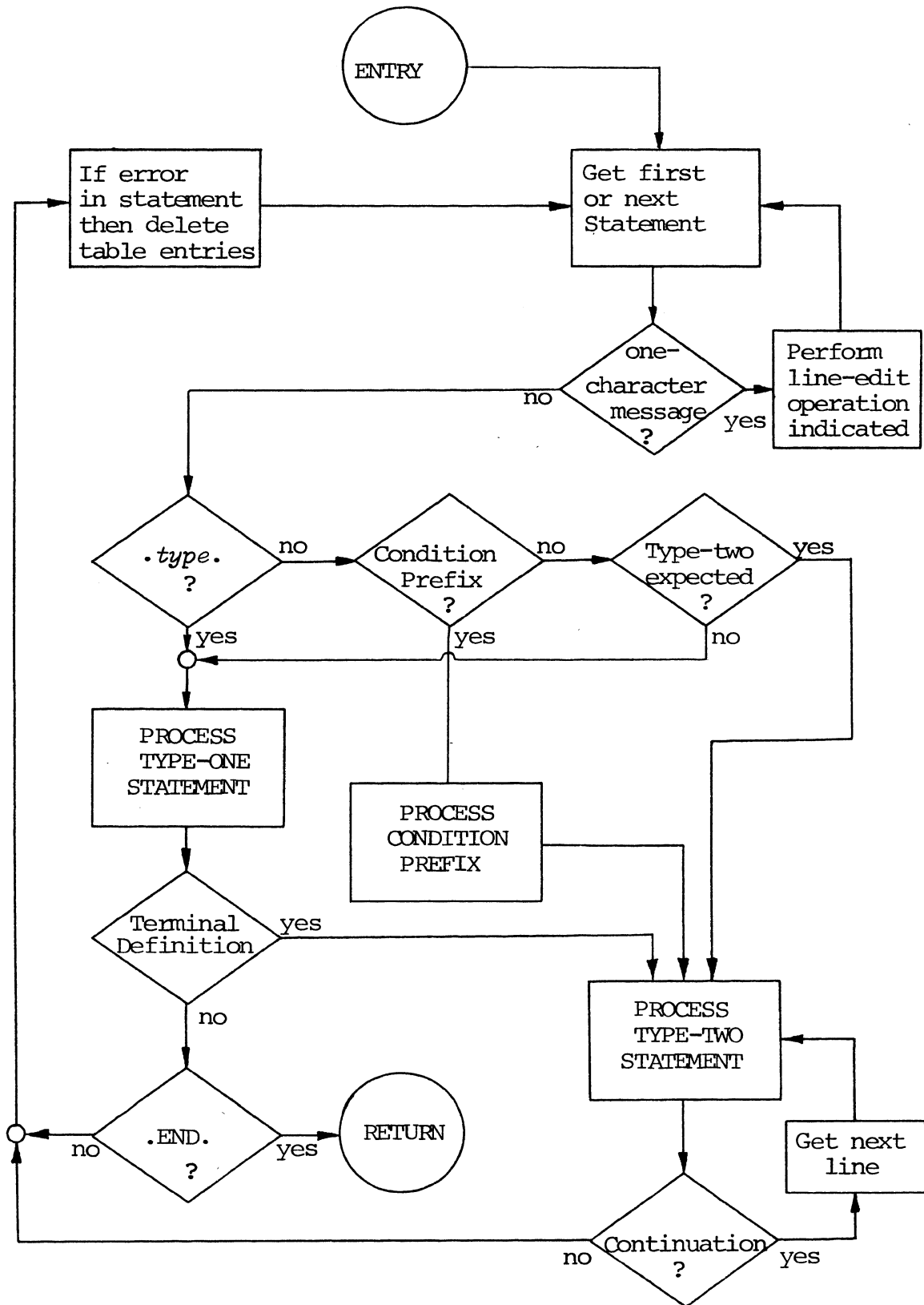


Figure 4. Block Diagram of Description Translator (IDD_TRAN)

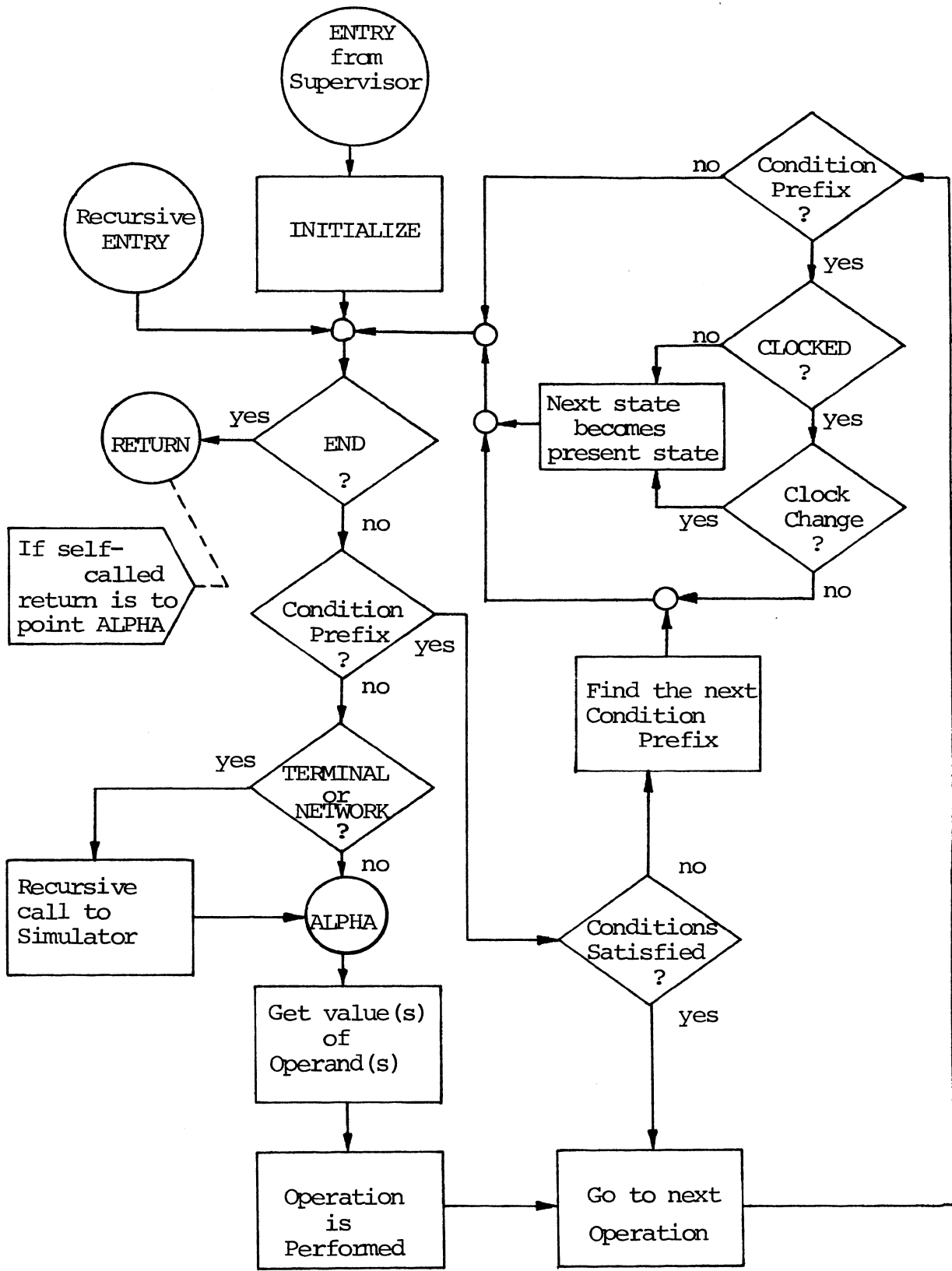


Figure 5. Block Diagram of Simulator (IDD_SIM)

Having obtained values for the operands (perhaps through recursive calls) the simulator proceeds to find the block of code corresponding to the current operator and to perform the indicated operation.

Following the simulation of a group of statements headed by a condition prefix, the simulator checks to see if the projected values of logic signals should become the current values. If the simulator is not operating in clocked mode it performs this "updating". If clocked mode was specified, then it compares the current and projected values of the signal designated as the clock. If the clock's value is to be changed, then updating takes place. The simulator then proceeds to search for the next satisfied condition prefix.

Besides the normal return to the supervisor at the user-specified point, three abnormal returns are possible. If the .end. statement is reached unexpectedly or if a serious error occurs, the immediate return is accompanied by a diagnostic message.

Whenever a GO TO statement is encountered, the simulator checks to see if it transfers control to an earlier part of the description. For each such backward transfer of control, the loop counter is incremented. If the number of loops exceeds the maximum number of loops specified, the simulator asks the question "Shall I Continue?". A negative response to this question is the

third way in which an immediate return to the supervisor may occur. An affirmative reply allows simulation to continue normally and resets the number of loops to zero.

A number of relatively small separately-compiled subprograms are used by the three main programs. The function of each is discussed in the following paragraphs.

The three subroutines NEXT_ID, NEXTNUM, and NUMBER are contained in the procedure called NXTITEM. When an identifier is expected, the function subprogram NEXT_ID is used to extract that identifier from the statement being processed. The returned value is the alphanumeric character string starting at the next position after the position to which the scan pointer (CI) is currently pointing. On exit, the scan pointer is left pointing to the last alphanumeric character of the identifier. Another function subprogram, NEXTNUM, is called when a constant is expected. Its function is similar to NEXT_ID except that it returns an integer constant instead of a character string. An additional entry point, NUMBER, may be referenced if the constant begins at, instead of after, the position indicated by the scan pointer.

The function IDCODE# is used to identify an alphanumeric string. The argument of this function is a character-string whose position in the name-table is to be found. If the character-string argument is found, the returned value is a positive integer indicating the

identifier's location in the name-table. Otherwise the value -1 is returned.

All diagnostic messages are handled by a subprogram having three entry points. The ERO_RET entry is used to pass a label establishing a common return point for the ERROR entry. If the subprogram is entered at the ERROR entry, it sets E_FLAG, an external logical variable, to one. Entry at the WARN entry point does not affect E_FLAG. For entry at either ERROR or WARN, an integer argument is passed denoting the code number of the diagnostic message to be printed. From either the ERROR or WARN entry points, the following takes place: The diagnostic message is printed according to the number passed as the argument. Then, if E_FLAG is one it exits to the label specified by the prior invocation of ERO_RET. Otherwise, it makes a normal return to the next statement following the invoking statement.

A fairly simple subprogram, DIS_WAV, exists to display the waveform data accumulated during simulation. This program may be called automatically by the simulator if the buffers become full, or it may be called at the request of the user by the supervisor.

Most, but not all, communication passes through a subprogram called GASOB. In addition to requesting and subsequently receiving an entry from the user, it also performs some preliminary processing of the user's response. This processing involves removing comments and

unnecessary blanks, and placing a terminator symbol, a semicolon, at the end of the user's text. Both the original entry and the processed version are made available to IDDAP in the form of external character-string variables. GASOB utilizes the MESSAGE HANDLER to perform the actual communication between IDDAP and the user.

Finally, a subroutine exists to convert the internal form of a signal's value to a character string. This function is invoked with three parameters giving the signal's current value, the user-defined length in bits, and the desired arithmetic base in which the value is ultimately to be displayed. The returned value is a character string representing the value in the specified base. The length of the resulting character string is just sufficient to contain the largest possible value expressible with the indicated number of bits when converted to the requested base.

VII. CONCLUSION

Fast execution and small memory storage requirements were not the main goal of IDDAP. The convenience to the user was considered more important. However, the speed of execution and the memory requirements do have an effect on its usefulness and efforts were made to write efficient programs. Both the memory requirements and speed of IDDAP compare favorably with other similar design-aid programs.

The package presented here runs in a 130 K byte-partition of memory and does not require any external bulk storage nor does it involve the use of overlay techniques. The approximate storage requirements may be broken down as follows:

IDDAP object code	59 K bytes
PL/1 library routines	22 K bytes
Static array storage	28 K bytes
Allowance for dynamically allocated storage	20 K bytes
	<hr/>
Total	129 K bytes

By elimination of program debugging statements and reduction of the maximum number of description statements to 100, the core requirements could be reduced to less than 100 K bytes.

The execution speed is of course highly problem dependent. For "average" description statements the

description translator requires approximately 0.1 seconds per statement. Rouse^[20] gives an example of the simulation of an oscillating NAND gate with a fan-out of 20 for which he gives the simulation times for 95 oscillations. Those times ranged from about five seconds to fifteen seconds depending on the "mode" of simulation. A similar example using IDDAP required twelve seconds for 95 oscillations in non-clocked mode, and ten seconds in clocked mode.

In simple descriptions of fewer than 20 statements, the user would scarcely be aware of delays due to processing. For descriptions ranging from 100 to 200 statements, noticeable waiting periods of five to thirty seconds would not be surprising.

This version of IDDAP should not be regarded as the ultimate form of assistance to creative design. The addition of a synthesizer to reduce the formal description to the hardware level is just one additional option which could be added.

Presently, IDDAP is unable to accommodate more than a single user at a time. Thus, another improvement would be to enable multiple users to be using IDDAP concurrently. This feature along with several of the other features found in general-purpose interactive systems would result in more economical and efficient use.

The largest amount of time devoted to this project was spent writing and debugging the programs. The most difficult part, however, was in deciding what features

should be made available. For example, the decision to represent most of the operators as keywords enclosed in periods was a compromise between several criteria which were felt to be important. The use of special symbols would have made the language more concise and easier to process, but more difficult to learn and to use. The decision to enclose the keywords in periods avoids the necessity of having reserved words, and allowing three-letter abbreviations provides some conciseness.

It is felt that the system described here offers concrete, convenient, useful help to designers in formulating, studying, and testing digital designs. By enabling the designer to interact with the computer in a suitable language IDDAP offers this assistance during the critical, creative, initial phases of the design process.

The formal description of a digital system along with the simulation results, especially waveform-timing-diagrams, also provides valuable documentation for the system being designed.

REFERENCES

1. K. E. Iverson, A Programming Language, New York, John Wiley and Sons, 1962.
2. K. E. Iverson, A. D. Falkoff, and E. H. Sussenguth, "A Formal Description of System/360," IBM Systems Journal, Vol. 3, No. 3, pp. 198-262, 1964.
3. H. Hellerman, Digital Computer System Principles, New York, McGraw-Hill, 1967.
4. "APL/360 User's Manual," IBM Program Produce GH20-0683-1, 1969.
5. "APL/360 Primer," IBM Program Product GH20-0689-1, 1969.
6. T. D. Friedman and S. C. Yang, "Quality of Designs from an Automatic Logic Generator," IBM Research Report RC 2068, April 25, 1968.
7. T. D. Friedman and S. C. Yang, "Methods Used in an Automatic Logic Design Generator (ALERT)," IEEE Transactions on Electronic Computers, Vol. C-18, pp. 593-614, July 1969.
8. "The Use of APL in Teaching," IBM Manual G320-0996-0, p. 27, 1969.
9. P. L. Koo and D. E. Atkins, "Arithmetic Unit of ILLIAC III - Simulation and Logical Design," Abstract Number 6975, IEEE Transactions on Electronic Computers, Vol. C-18, p. 868, September 1969.
10. R. F. Crall, "Description and Simulation of the SCC 650 Computer in PL/1," unpublished, June 1969.
11. J. R. Duley and D. L. Dietmeyer, "A Digital System Design Language (DDL)," IEEE Transactions on Electronic Computers, Vol. C-17, pp. 850-861, September 1968.
12. J. R. Duley and D. L. Dietmeyer, "Translation of a DDL Digital System Specification to Boolean Equations," IEEE Transactions on Electronic Computers, Vol. C-18, pp. 305-313, April 1969.

13. T. C. Bartee, I. L. Lebow, and I. S. Reed, Theory and Design of Digital Machines, New York, McGraw-Hill, 324 p., 1962.
14. H. Schorr, "Computer-Aided Digital System Design and Analysis Using a Register Transfer Language," IEEE Transactions on Electronic Computers, Vol. EC-13, pp. 730-737, December 1964.
15. Scientific Control Corporation, "Machine Instructions 650-2," Dwg. No. A 10547C, Dallas, Texas, 78 p., 1967.
16. Y. Chu, "An ALGOL-like Computer Design Language," Communications of the ACM, Vol. 8, pp. 607-615, October 1965.
17. H. J. Pottinger, "A Formal Description of the SCC 650 Digital Computer," Master's Thesis, University of Missouri - Rolla, Rolla, Missouri, 44 p., 1968.
18. B. D. McCurdy and Y. Chu, "Boolean Translation of a Macro Logic Design," Digest of the First Annual IEEE Computer Conference, Chicago, pp. 124-127, September 1967.
- ✓ 19. D. M. Rouse, "A Design Oriented Digital Design Language," Master's Thesis, University of Missouri - Rolla, Rolla, Missouri, 64 p., 1969.
20. D. M. Rouse, "A Simulation and Diagnosis System Incorporating Various Time Delay Models and Functional Elements," Ph.D. Dissertation, University of Missouri - Rolla, Rolla, Missouri, 136 p., 1970.
21. C. V. Srinivasan, "CDL1, A Computer Description Language," Scientific Report No. 3, AFCRL-69-0322, Clearinghouse, Department of Commerce, 26 p., July 1969.
22. D. L. Parnas, "More on Simulation Language and Design Methodology for Computer Systems," Proceedings of the Spring Joint Computer Conference, pp. 739-743, 1969.
23. R. J. Smith II, "Synthesis Heuristics for Large Synchronous Sequential Circuits," Ph.D. Dissertation, University of Missouri - Rolla, Rolla, Missouri, 79 p., 1970.

24. R. J. Smith II, J. H. Tracey, W. L. Schoeffel, and G. K. Maki, "Automation in the Design of Asynchronous Sequential Circuits," Proceedings of the Spring Joint Computer Conference, Vol. 32, Washington, D. C., Thompson, pp. 53-60, April 1968.
25. R. E. Marsh, "Logic Simulation Made Simple with LOGISIM," Abstract, IEEE Computer Group News, Vol. 3, No. 3, p. 70, March 1970.
26. C. G. Hays, "Computer-Aided Design: Simulation of Digital Design Logic," IEEE Transactions on Electronic Computers, Vol. C-18, pp. 1-10, January 1969.
27. S. A. Szygenda, "TEGAS - A Diagnostic Test Generation and Simulation System for Digital Computers," Proceedings of the Third Hawaii International Conference on System Sciences, January 1970.
28. E. L. Huelsman, "Design of an Asynchronous Tic Tac Toe Machine," Undergraduate Seminar Paper, University of Missouri - Rolla, Rolla, Missouri, 18 p., May 1970.
29. G. I. Rhine, "Design of an Interface Between the SCC 650 Computer and an ARDS Graphics Terminal," Master's Thesis, University of Missouri - Rolla, Rolla, Missouri, expected completion date - December 1970.
30. W. E. Omohundro, "Design of an Interface Between the SCC 650 Computer and a Communications Data Set," Master's Thesis, University of Missouri - Rolla, Rolla, Missouri, expected completion date - January 1971.
31. R. F. Crall, IDDAP -- Program Maintenance Notes, Technical Report CRL 69.1, Computer Research Laboratory, Electrical Engineering Department, University of Missouri - Rolla, Rolla, Missouri, August 1970.

VITA

Richard Franklin Crall was born on December 7, 1943, in Lafayette, Indiana. He received his primary and secondary education in Flint, Michigan. He then returned to Lafayette, Indiana, to study Electrical Engineering at Purdue University, receiving the degree of Bachelor of Science in Electrical Engineering in August, 1965.

He has been enrolled in graduate school at the University of Missouri - Rolla since September, 1965. During that time he married Barbara Miller, who also received her Bachelor's degree from Purdue University.

He was a member of the staff of the Electrical Engineering Department at the University of Missouri - Rolla at the rank of graduate assistant from February, 1966, to June, 1967, and at the rank of instructor (one-half time) from September, 1967, to June, 1970. He received the degree of Master of Science in Electrical Engineering there in June, 1967.

Upon completion of his degree he will be an assistant professor of Electrical Engineering at Sacramento State College, Sacramento, California.