AFCRL-67-0565

AN INTRODUCTION TO CDL1, A COMPUTER DESCRIPTION LANGUAGE

CHITOOR V. SRINIVASAN

RADIO CORPORATION OF AMERICA
RCA LABORATORIES
PRINCETON, NEW JERSEY

CONTRACT NO. AF19(628)4787
PROJECT NO. 5632 TASK NO. 563202
WORK UNIT NO. 56320201

SCIENTIFIC REPORT NO. 1

SEPTEMBER 1967

CONTRACT MONITOR: ROCCO H. URBANO
DATA SCIENCES LABORATORY

PREPARED FOR

AIR FORCE CAMBRIDGE RESEARCH LABORATORIES
OFFICE OF AEROSPACE RESEARCH
UNITED STATES AIR FORCE
BEDFORD, MASSACHUSETTS

AFCRL-67-0565

AN INTRODUCTION TO CDL1, A COMPUTER DESCRIPTION LANGUAGE

CHITOOR V. SRINIVASAN

RADIO CORPORATION OF AMERICA
RCA LABORATORIES
PRINCETON, NEW JERSEY

CONTRACT NO. AF19(628)4789
PROJECT NO. 5632 TASK NO. 563202
WORK UNIT NO. 56320201

SCIENTIFIC REPORT NO. 1

SEPTEMBER 1967

CONTRACT MONITOR: ROCCO H. URBANO
DATA SCIENCES LABORATORY

PREPARED FOR

AIR FORCE CAMBRIDGE RESEARCH LABORATORIES
OFFICE OF AEROSPACE RESEARCH
UNITED STATES AIR FORCE
BEDFORD, MASSACHUSETTS

ABSTRACT

The objective of this report is to develop a formal language to describe hardware and software computing systems. The language is to provide a linguistic basis to consider machine-aided solutions of a variety of design problems; i.e., problems concerning design documentation, data retrieval systems, system simulation, diagnosis, analysis and synthesis.

This report discusses in some detail the considerations that went into the design of the computer description language, called CDL1; it points out the need for developing such a language and briefly discusses the kinds of applications such a language may have.

The report points out the various kinds of system descriptions one may encounter in a design process and relates them to the language features necessary to express them; the language itself is described informally. Examples are presented to illustrate the use of the language, the concepts associated with descriptions of systems at various stages of design, and the consequent hierarchical structure such descriptions acquire.

TABLE OF CONTENTS

ILLUSTRATIONS

I. INTRODUCTION

CDL1 (Computer Description Language No. 1) is a formal language for describing computing systems in various stages of design. Such computing systems could be hardware or software systems, or combinations thereof. At the beginning of a system design one may describe in CDL1 its functional specifications; when design is complete one may describe its structural and operational characteristics in terms of its logical nets, controls, and sequencing. Between these two extremes, during the course of design, there may exist a variety of descriptions of parts of a system in CDL1, specifying to various degrees of detail their functional, structural, and operational characteristics.

As a formal language to describe and talk about systems of this kind at various levels of design, CDL1 provides a linguistic basis to consider machine-aided solutions of a variety of system design problems: problems concerning design documentation and their attendant data retrieval schemes, and system simulation at different stages of design, system analysis and synthesis. In fact, we plan to use CDL1 as a basic description language in a class of interactive Design Aid Systems (DAS). At present we visualize DAS's with the following capabilities: They should have facilities to check descriptions in CDL1 for validity and file them appropriately; retrieve stored data as necessary; provide editing and updating facilities for documenting design changes and system modifications; interpret a body of description for simulation, design analysis or synthesis; accept commands from users to perform desired design tasks; gather abstracts of a body of description in

manners specified a priori; build a design library of systems used earlier; allow several designers to access and use its files simultaneously; and finally even tutor users on the proper means of using the design systems. In this environment we wish to automate as many of the design tasks as possible.

To create such DAS's a formal, system description language is essential. The language should not only be capable of describing a system at various stages of design, but also provide implicitly a filing scheme for the descriptive data. We believe CDL1 has the necessary expressive power and logical structure to build design aid superstructures of the above kinds.

Presently we do not know how one may go about specifying in detail DAS's of the size and complexity suggested above and estimating the computational facilities they should have. Systems like these should grow with usage; they cannot be specified a priori. Our first task in this endeavor has been the specification of an adequate description language. In its present form CDL1 may already be used as the basis for developing a first DAS for design documentation and product specification. Later, in Section VII, we shall discuss our plans and the tasks we have set for ourselves in the immediate future.

Several investigators have attempted to develop description languages for hardware and software systems description [2-6, 8, 10-13]. In my opinion, none of these have adequately considered all the design requirements. We hope the reason for this conclusion will become clear in this report.

The general structure of CDL1 is described in this report, with examples illustrating its use. We begin in Section II with a discussion of what we wish to describe, introducing the principal concepts in the organization of the language. In Section III we have attempted to justify the need for a new language like CDL1, by contrasting it with existing programming and description language. Sections II and III together also present indirectly the considerations that have led us to create in CDL1 the features it has. The features themselves are discussed in Section IV. Sections V and VI present a rather detailed discussion of two examples, 'The 70/15 Core Memory System', and 'A Parallel Processing Procedure', respectively. Though the examples get burdened with details at times, they are illustrative of the mass of data that may be systematically described, filed, and later referred to in other parts of the description. The report concludes with a discussion of our future plans.

In the forthcoming report (A Formal Definition of CDL1) under this contract, the language will be described formally.

II. WHAT WE WISH TO DESCRIBE

Computing machines are command-obeying machines. The basic dichotomy of a computing system into hardware and software arises directly because of this. It is, therefore, appropriate to describe computing machines in terms of their commands, what the commands mean, and how they are implemented. Indeed, this is just the way they are described in existing machine manuals.

In CDL1 we address ourselves to the problem of describing them formally* in this manner, from three points of view: functional, structural, and operational. In doing so we shall find it convenient to recognize the following four levels of design:

(1) specifications,

(2) data-flow sequences,

(3) data & control flow sequences, and

and (4) logical nets.

Let me discuss these concepts in greater detail and illustrate them through a simple example.

A. The Three Kinds of Descriptions

The workings of a computer involve movements of data from one place to another within the machine, and data modifications performed by logical nets. Such movements and modifications are always under the control of the commands to the machine. Within a machine, data are represented in terms of signal level configurations (states) that parts of the machine may assume. So also, the controls initiating data movements and modifications

*By a 'formal description' we mean one that is machine-interpretable.

are represented in terms of signal levels and pulses that its control lines may carry. A description of a machine in terms of its parts, the signals they carry and the actions they initiate is referred to in CDL1 as the operational description of the machine. A designer may view such a description as a detailed specification of the implementation of a set of commands to a machine, in a given hardware complement.

An operational description is usually the result of a design process which had its beginnings in an understanding of what a command is to do, and what the available hardware facilities are. The function of a command may be described as an algorithm in an algorithmic language, and the hardware facilities may be described in terms of the component items, their input, output and control lines, their interconnections, and finally, their own functional and/or operational characteristics.

We shall call the description of the function of a command or a hardware item as an algorithm, its functional description. A functional description could be in the form of a table of input/output relationships, or it could be a function already defined in the algorithmic language used, or it could be a program in the language. It could be completely independent of the details of implementation of the hardware item or the command.

The description of a hardware item in terms of its input, output and control lines, and a hardware system in terms of the interconnection structure of its components, we shall call a structural description.

The examples in Section II-B and Section IV will illustrate the differences of the above three kinds of descriptions. Before considering the example, let me point out the essential qualitative differences existing

among the three kinds of descriptions; these differences will in turn manifest themselves in the syntax of statements used to describe them.

The distinction between a structure specification and the other two kinds of descriptions should be obvious. Let us direct our attention to the differences between an operational and a functional description, which are more subtle.

The execution of a program (an algorithm) will usually imply the existence of a control mechanism, a scheduler, to read the statements in the program in the proper order and to activate them at the right times. One may imagine a pointer (or pointers) tr versing a program, selecting the commands to be executed at different times; a command (statement) will become alive only when it is pointed to, otherwise it will remain dormant. We shall call such statements in a language, the dormant statements. The rules regulating the operation of the scheduler may either be implicit in the 'sequential' or 'data-availability' structure of a program, or else may be given explicitly through 'transfer control' statements in the program itself. We shall use the phrase 'functional description' to denote exclusively a description containing dormant statements. A program so described obtains its full meaning only in the context of its scheduler.

On the other hand, an operational description should be totally self-contained. It should be in terms of the signals, signal lines, gates, etc., within a machine, all of which should have been explicitly defined. No part of a machine sequencing of its actions may be left unspecified; they should all be explicitly indicated in terms of the controls that initiate them. Every statement in an operational description should be self-activating on the basis of its event-control signals, which themselves should be part

of the description. We shall call such self-activating statements, (like the 'ON'-statement in PL1 [15]) autonomous statements.

The autonomous statements are always alive, just as a hardware item within a system might be. An operational description of a system will consist of only autonomous statements; it may actually resemble the internal operation of the system itself. The meaning of such a description will depend only on the structure of the system being described; whereas data in a functional description might assume a structure as a result of conventions in the language, in an operational description data might assume only the structure imposed on it by the hardware. Thus, an operational description cannot ever be independent of the details of implementation of a hardware or a command.

B. The Four Levels of Design: An Example

Suppose ADD(A,B,C) is a command to a machine yet to be designed. One might describe functionally what one wants of ADD(A,B,C) in some language, which might look as follows:

D1: "THERE EXISTS A SCALAR ARRAY OF SIZE N, CALLED MEM. FOR $0 < (A,B,C) =< N$, ADD(A,B,C) MEANS: PUT IN LOCATION C OF MEM THE SUM OF CONTENTS OF LOCATIONS A AND B OF MEM."

If the description language had conventions for array declarations, for accessing elements of an array through indexing and for interpreting '+', then the description might look like:

D2: "DECLARE MEM, SCALAR ARRAY, SIZE N. IF $0 < (A,B,C) =< N$ THEN ADD(A,B,C) MEANS MEM[C] ← MEM[A] + MEM[B]."

Both D1 and D2 are meaningful only if there are conventions in the language to interpret 'scalars', 'sum', '+', etc.; the semantics of the language should be well defined in some sense. D1 and D2 may not yet describe ADD(A,B,C) truly, since they do not specify scalar representation and the add algorithm used in the object machine. As given above 'scalar', 'sum', '+', etc., have their meanings given to them in the language, and not the meanings they are to have in the object system being designed. One may now declare the desired scalar representation in some form, define the add algorithm to be used, and invoke these while interpreting D1 or D2. Thus, one may obtain a more precise functional description of ADD(A,B,C).

A collection of such functional descriptions, one for each command of a machine, might form part of a specification for a new machine. To complete the specification one may have to say how the instructions are cycled within the machine, what hardware structure (block-diagram) the machine may have, and speed, cost and myriads of other restrictions it may have to satisfy.

Suppose that the command, ADD(A,B,C), is to be implemented in the system whose block diagram is shown in Fig. 1. We may not yet know the details of I/O gating of the registers and memory in the block diagram. Still, the very presentation of the structure in Fig. 1 enables one to go to the next level of specification of ADD(A,B,C), the data-flow sequence level.

Let us assume that the block diagram was declared in some form to the description files, as part of a structural description of the machine, and also it was indicated that 'SUM' or '+' is to be done using the adder network shown. Further, suppose that the operation of the adder network was specified in terms of the I/O relationship it maintains; this description

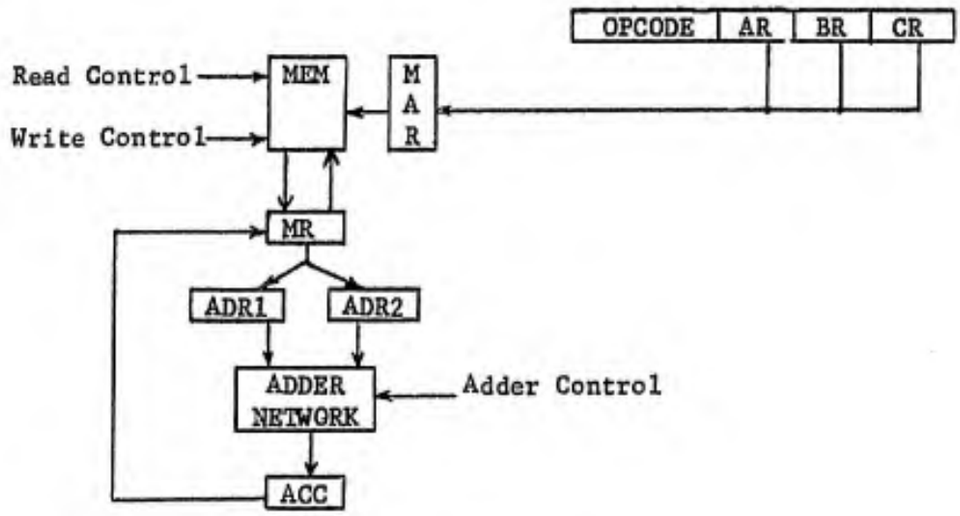

FIG. 1 Block Diagram of a Hypothetical Machine

could itself be functional, or operational in terms of component parts of the network. Assume also that the scalar representation was specified in terms of the bit patterns within MEM.

One may now obtain the data-flow description D3 below, by following the only paths available from memory to adder network and back. In fact, one can visualize an automatic synthesis algorithm which could produce D3 from D2 and all the auxiliary declarations made on the side.

D3: Data-flow sequence for ADD(A,B,C)

(1) MR ← MEM[A],

(2) ADR1 ← MR,

(3) MR ← MEM[B],

(4) ADR2 ← MR,

(5) TRIGGER adder control

(6) MR ← ACC, and

(7) MEM[C] ← MR.

To interpret D3 formally one should have some conventions concerning data movement, adder over-flow, and interpretation of registers and MEM as data items. These conventions could be as follows:

1. Data transfer between registers is indicated by '←', and is admissible only if the registers are of the same dimension. We may assume that the dimensions of the registers have been declared elsewhere.

2. The over-flow bit might have been stored in a flip-flop within the adder/network and this could have been described in the adder network operation.

3. TRIGGER is to be interpreted as a call to the description of the operation triggered by the control signal.

4. Registers are interpreted as vectors and memory as an array of vectors. The vectors, in this case, consist of 0's and 1's only.

Description D3 is still functional, but it speaks more about how the ADD command is to be implemented in the given structure. In D3 we do not talk about data movements in terms of I/O gatings of the registers themselves, but in terms of what the data names mean within the language. Also, we have not specified timing, and how MEM as a hardware array accesses its elements. The only place where we could introduce an operational description in D3 is in the adder network call. If the adder had been described operationally then one could invoke it in response to 'TRIGGER adder control'. Thus D3 could be partly functional and partly operational in nature. Let me now take this one step further to illustrate how memory operation could be introduced in the description. We need some more declarations.

Assume that the number of input and output lines of MEM have been fixed, and MEM has two control lines, as shown in Fig. 1. We shall first describe what happens in response to signals on these control lines. We shall assume that control lines assume values 1 or 0 (TRUE or FALSE). The description of memory operation might look as follows:

D4. Functional Description of Memory

1. WHEN READ = 1 DO

   MR ← MEM[The binary number corresponding to the contents of MAR],

   READ ← 0.

2. WHEN WRITE = 1 DO

   MEM [The binary number corresponding to the contents of MAR] ← MR,

   WRITE ← 0.

3. READ and WRITE shall never be simultaneously equal to 1.

Now assume that the instruction format (the way instruction appears within a machine) of ADD(A,B,C) was specified elsewhere as the contents of the instruction register, IR, as shown in Fig. 1, where the binary representations of numbers A,B, and C are stored in registers AR, BR and CR, respectively. One may now rewrite D3 as follows:

D5: Data & Control Flow Sequence

1. MAR ← AR,
2. TRIGGER read,
3. ADR1 ← MR, MAR ← BR,
4. TRIGGER read,
5. ADR2 ← MR,

6. TRIGGER adder control,

7. MAR ← CR, MR ← ACC, and

8. TRIGGER write.

The trigger commands in D5 constitute a partial control flow sequence, which may be interpreted as calls to appropriate descriptions declared elsewhere. Depending upon the nature of descriptions so-called D5 may be partly functional and partly operational.

To obtain a completely operational description of ADD(A,B,C) one should declare I/O gatings of the registers (by associating controls with them), and specify timing and control signals (pulses and levels). Also, the actions initiated by the hardware items in response to control commands (like in D4) should be described. One may then give a sequence of control lines to be triggered, with the necessary timing pulses. Such a control sequence may then be interpreted in a variety of ways. One may either invoke a totally operational description of actions initiated by the controls or a totally functional description, or any combination thereof. For the sake of simplicity and brevity of this discussion we shall not presently enter into the details of logical net designs appropriate for the current example. The reader may imagine how it might be done.

A design process may thus be captured: At each level of design one may obtain a formal description, which is not only a true documentation of items designed, but also is capable of being interpreted by a machine for simulation, design analysis, or automatic synthesis. Also, one may design a complex system in terms of its parts and plug in descriptions of parts so designed appropriately in a procedure describing the operation of the total system. A hierarchical structure of descriptions is so obtained.

Design changes may now be viewed as moving up and down this hierarchy. Also, at the end of a design one may compare the operation of a system with its functional specification through simulation, and thus develop product assurance and system diagnosis techniques. Also, the functional specification of a system may be used to debug software written for the system. The variety of design tasks for which automatic design aids could be developed in this environment is quite numerous. Let us now take a brief look at what was involved in the creation of the above sequence of descriptions, to understand the kinds of facilities a description language should have.

C. Discussion of the Example

In the above example we were able to capture the design process by having a main scheme of what was to be designed [D2 of ADD(A,B,C)], and progressively modifying this scheme in response to declarations of other items created in the design process. Each level of transition in design required new objects to be declared. Since the items so declared could themselves be cast into a similar description methodology we get a hierarchical structure. During the transition from D2 to D5 we had occasion to assume declarations of following kinds of items:

1. Data-structures: Like registers, memory array, and scalars.
2. Definition of operators: Like '+'.
3. Formats: Like number representation format and the instruction format.
4. Hardware items & structure: Like registers, memory and the block diagram.

5. Procedures: Like the functional and/or operational descriptions of the hardware items.

6. Signals: We did not discuss these in detail in the example.

One may also need declarations of

7. Codes of symbols and strings.

8. Functions on software or hardware items.

9. Tables and tabular functions: To define these one may have to specify how tables are organized, accessed and kept updated, and, also, what functions are defined on them.

10. Complex data structures like lists, trees, etc.

The last three will be particularly useful in software descriptions, as we shall later see. In a system design process the number of declarations of the above kind are likely to be very large. Also, they are likely to be made by different people, at different stages of a design process. There should be a scheme to make these declarations in some standard format and file them appropriately, so that they may be accessed and used by all the designers. To do so it is very desirable to have a uniform set of conventions.

D. Description of Software Systems

The principal constituents of software systems are aggregates of data and algorithms operating on the data. Every data aggregate is associated with a storage format and an addressing structure, which determine the way it is accessed, treated and kept updated. In the description of a software system we shall choose to separate the data definitions from the specification of the algorithms [1]. Such a separation might not only

make the relevant algorithms easy to understand, but also provide a facility to experiment with alternate data organization schemes suitable for a given algorithmic task. Data accessing, updating and creation procedures may be declared separately as subroutines, or functions, or macros and called into a procedure as and when necessary, through appropriate naming schemes. The facilities to invoke the macros, functions and subroutines and link them in various ways will then give such a body of description, a hierarchical structure. With this in mind in CDL1 we have provided some primitive facilities for data-structure definitions. We shall have more to say about this in another report.

E. A Summary

This then is what we wish to describe: We wish to describe the design process in terms of descriptions of the items designed at various stages of design (the items could be software or hardware systems); at each level of description we wish the documentation to be complete, in the sense that the descriptions themselves could be directly simulated by a machine if necessary. We wish to talk about a large system in terms of its parts, design the parts separately and link them together as necessary, taking care of the boundary conditions. We want to describe procedures as well as hardware structures. We wish to describe the parallelism within an algorithm or a hardware item as it may actually appear; and to talk about data as symbolic items or as machine states, and to describe the relation between the two. We wish to be able to specify design alternatives and design changes and to assign hardware facilities to various tasks in a procedure, either dynamically or statically. Finally, we wish to organize all such descriptions in a manner which would be easily accessible and modifiable.

III. THE NEED FOR A NEW LANGUAGE

I shall present the case for a new language from three points of view:

1. Objects spoken about, statements and operations.
2. Program or description structure.
3. Uses.

Many of the language features discussed below are available individually or in groups in one or the other of the existing languages. But no existing language has them all together. The need for three kinds of descriptions, the implied documentation facility and the hierarchical structure of descriptions make it desirable to develop a standard set of uniform conventions for the entire spectrum of design applications that the language may have. Many design problems may then be studied with reference to a single common base. This is the principal reason for developing a new language.

Even if one chose to modify an existing language to suit the new requirements the extent of modification and addition will be so great that one would essentially arrive at a new language; the existing processors for the parent language could not be profitably used. Also, the kinds of uses a body of design description might be put to are far different from the uses one gets from a program written in a conventional language. Processors for design documentation, data retrieval, design analysis and synthesis will have to be created from the start; so also, processors for simulation will have to be created anew. The existing simulation languages do not have a hierarchical structure, and facilities for different kinds of descriptions that may arise in a design environment. Thus, they are not suitable to

describe systems in the process of design. The description languages proposed so far [2-5, 8, 13] also suffer from the same kind of inadequacy.

The discussion below will make clear the essential differences.

A. Objects Spoken About, Statements and Operations

In a conventional language the objects spoken about are variables and constants, which are usually of fixed types, a characteristic of the language itself. Also, one speaks of operations on these objects. Again, the kinds of operations and their interpretation are fixed characteristics of the language. One may have facilities to create complex data structures and specify schemes for addressing them.

An algorithm in a conventional language is always a finished product, not one in the process of design. All details of data and sequencing are specified either explicitly or implicitly through the language conventions. Usually one has only limited facilities or none at all for specifying parallel or autonomous processes.

In a description language the variety of objects spoken about could be much larger. An object could be a constant, a variable, or a data structure, or it could be a format, a hardware item, its structure, an algorithm, or a definition of an operation. Standard formats for declaring these various kinds of objects are necessary.

A given operand like scalar may be used at times as a string, at times as a vector or at other times in terms of its bit string representation inside a machine. To do so one needs a dynamic declarative facility indicating how an operand is to be used in a given instance. A large variety of operand types is desirable.

Also, operands and operators may not always have the same standard set of possible interpretations; at times one may have to invoke the interpretation they ought to have in the object system being designed (like '+' and 'scalar' in description D2 in Section II). Operand representations may change from one system to another; accordingly, the operator interpretation also should be changed. There should be facility to define operands and operators. In a software description it is desirable to have a facility to declare data structures associated with data types.

A description in a description language may not be that of a finished product. A hierarchical descriptive structure is necessary to describe systems only partially designed. In many cases in the initial stages of design of a system the precise nature of hardware items, or data structures may not be known. A user should have facilities to declare partially defined items and later specify them in greater detail.

A description language should also contain a greater variety of of expressions than what the conventional languages usually have. Besides arithmetic, boolean, and string expressions, one would also like to have reduction, selection, and permutation expressions; these were first pointed out by Iverson [6]. Also, the need for three kinds of descriptions (functional, operational and structural) point out the need for three kinds of statements: dormant, autonomous, and connection statements, the last one being used to indicate hardware connections. Since machine operations take place almost always in parallel there should be facilities to describe parallel processes.

In the description of a task to be executed by a system one might need a facility to assign hardware items or algorithms to perform the

different subtasks that might arise in the course of executing a task. Such assignments could be conditional; the very execution of a subtask could be conditional on the satisfaction of some error criteria or hardware availability criteria. These criteria might not have been known at the time the descriptions were written. It will be convenient if one could attach such conditions to subtasks, or to variables and hardware items entering a task, as and when they become evident, without having to modify the already existing descriptions. A designer may profitably use such a facility to keep a running commentary on error checks that he might introduce within a process.

None of the existing languages can provide all these facilities without undergoing some modifications, and additions. The description language proposed by Iverson [6] is very skeletal and elementary: It is more a notational schema than a language; it does not have declarations, or facilities for naming objects, or facilities for function, macro and subroutine calls. The design languages proposed by Gorman, Chu, Dennis and others [5, 3, 2, 8] are in the nature of experimental systems, which did not consider all aspects of design requirements. They have made a valuable contribution to our understanding of description languages.

B. Program Structure

Programs in languages like ALGOL and PL1 may have a block and/or group structure, which is used to control dynamically the creation and elimination of variables, locally within a program. Also, a block introduces an additional level of labelling of statements; two statements in different blocks may have the same label, and may also use the same variable name to denote different local variables. This facility is useful for dynamic

storage allocation and efficient use of memory space. Block and group structures of a program are part of a programming language, in the sense that they affect the way the variables are to be interpreted within a program.

In addition to these, in a time-sharing system programs may also have a file structure in terms of segments and pages. These are used to address programs within a file in convenient unit sizes, for execution or editing. Otherwise, they do not have any relevance within a program.

A description file should be easily accessible and modifiable. Some form of classification of a large body of description in terms of smaller parts of it is very desirable. Such classification should not only provide a filing scheme, just as segments and pages do, but if it is to be useful, it should also reflect naturally the subdivisions of a system into its parts. Thus, file classification schemes should be part of a description language; the interpretation given to a statement in the language might even depend on the classification of the body of data in which it appears.

In CDL1 such a classification is obtained by the use of what are called module-types. A module is a body of data with a title. The title consists of two parts: a module-name* and a module-type. Each module is, in a sense, a self-contained entity, and usually pertains to a description of a particular object or kind of object within a system. In addition to this modular structure a procedure (or a description) may also have a block and a group structure.

C. Uses

A program written in a programming language has only one purpose: It is to be executed by a machine. A body of description might serve

*This could be vacuous.

several purposes in a design environment. It may be just a documentation device for which simple retrieval facilities may be built; or one may perform some analysis of a body of description and answer some involved questions that a designer may ask; or it may be executed in the usual sense of a program to cause a simulation; or finally, it may be used as a specification to go to the next level of design which may itself be performed automatically. The kinds of processors that one needs for a design implementation of a description language are quite different from a compiler.

We leave it to the reader to judge for himself whether the differences pointed out above justify the development of a new language. This author is of the opinion that only through the creation of a new language and associated operating facilities may one introduce a uniform and consistent set of conventions adequate for the entire range of discourse.

IV. FEATURES OF CDL1

A. Operands

CDL1 has a variety of operands. They fall into two basic categories: software and hardware operands. Within each category the operands are classified according to their data types and attributes. The type of an operand governs the way it is stored, accessed and used in expressions; within a given data type, the attributes of an operand may introduce variations on these.

The software data types are:

1. Scalar (S)
2. Vector (V)
3. Matrix (M)
4. Array (A)
5. STring (ST)
6. Bit STring (BST)
7. STring Vector (STV)
8. STring Array (STA)
9. Bit STring Vector (BSTV)
10. Bit STring Array (BSTA)
11. Index Vector (IV)
12. Index Vector Array (IVA)
13. Pulse (P)

A string vector is a vector whose elements are all strings. So also a BSTV will have bit strings as its elements. A BST is always reckoned in terms of its bits, where as a ST is reckoned in terms of its characters (which may appear in their encoded form).

The hardware types are basically of two kinds: (1) One-of-a-kind type or (2) an array-of-a-kind type. In the latter case the addressing mechanism for accessing one or more items of an array should be part of the hardware. All items in a hardware array should be identical to each other, with some minor exceptions. The hardware data types recognized in CDL1 are:

1. Flip-Flop (FF)
2. Register (R)
3. Gate (G)
4. Switch (SW)
5. Network (NW)
6. Bus (B)
7. System (SYS)
8. Delay Line (DL)

and 9. I/O Unit (I/OU)

These may occur either as one-of-a-kind items, or else as part of an array of identical units. In the case of arrays the suffix 'A' is attached to the abbreviations given above, denoting unit types. Elements of an array may be addressed through suitable indexing conventions which themselves may be declared.

Each operand when first declared should contain a flag denoting its data type. The abbreviations indicated above are used as flags, which are affixed as suffixes to the respective operand names, with a blank in between, like ADDEND S (a Scalar), ACC R (Register), MEMORY RA (Register Array), etc. These flags may be omitted in later use, unless one wishes to indicate explicitly a change in the operand type, in a specific instance of reference to it. If so a new flag may be inserted. The operand will then be given the interpretation corresponding to its new type, if possible, for that particular occurrence only. Thus, one obtains a dynamic declarative facility.

Each hardware operand is given a software equivalent, which will govern its interpretation in expressions. Thus, a Register will be interpreted as a Vector in expressions, a Flip-flop as a Scalar, the inputs of a network as a Vector or an Array, as the case may be, etc. The data-type of a hardware operand may not be changed ever. However, its software equivalent may undergo type changes. Thus, one may treat the contents of

a register as a vector, or a bit string, or a string, or even a scalar. The admissible software type changes are restricted to changes between any two among the following data-types: Scalar, Vector, String and Bit String.

DATATYPE has the status of a retrieval function in CDL1. One may ask for the DATATYPE of an operand, through the functional notation: 'DATATYPE (Operand name)'. Every data type may have attribute-types associated with it. Thus, a Scalar may have four attribute-types: BASE, TYPE, MODE, PRECISION, where BASE could be any integer, TYPE could be FIXED or FLOAT, MODE could be REAL or IMAG, and PRECISION, a pair of integers. One may ask for 'ATTRIBUTES (data type)' to get the attribute-types associated with a given data type. One may also ask for 'ATTRIBUTES (operand name)' to get the attributes associated with a given operand. Thus, for JIM S, ATTRIBUTES (JIM) could be '10, FIXED, REAL, (8,2)', by a previous declaration.

Every attribute-type is also the name of a retrieval function in CDL1. Thus, for JIM above, MODE (JIM) is REAL. A user may define new attribute-types for a data type and declare them while declaring the data. Operations on data, and their storage and accessing mechanisms may be dependent on their attributes. Thus, the operation '+', has different interpretations for a scalar, depending upon, whether the scalar is of type FIXED or FLOAT.

In CDL1 the standard data types have standard attribute-types associated with them. Those of a scalar were discussed above. Within the language system there are conventions, default conditions (like in PL1), which take care of undeclared data attributes. Some of the standard attribute-types associated with the CDL1 data types are listed below.

1. Software Operands

1. Data types: V, STV, BSTV, IV.

   Attribute-type: DIMN. This stands for the 'dimension' of the vectors.

2. Data types: A, STA, BSTA, IVA, IV

   Attribute-type: SIZE. The size of an array is a vector whose i^{th} element is the size of the i^{th} dimension of the array. The size of an IV is also a vector, whose i^{th} element is the DIMN of the i^{th} element of the IV. (An index vector is a vector of vectors.)

3. Data types: ST, BST

   Attribute-type: LENGTH.

4. Data type: P (Pulse).

   Attribute-types: WIDTH, AMP, RISE, FALL, DIRECTION.

   'AMP' stands for amplitude. The direction of a pulse is positive (if it goes from 0 to 1) or negative (1 to 0). All pulses are treated as logical pulses. One may invoke the amplitude of a pulse in an expression only by calling for AMP(pulse name).

2. Hardware Operands

The standard attribute-types are:

1. ISIZE : Input size
2. OSIZE : Output size
3. SSIZE : Storage size
4. CSIZE : Control size

Inputs, outputs, controls, and storage may be structured as arrays of arbitrary dimensions for indexing purposes. Thus, a hardware item with 25 inputs may have them indexed linearly from 1 to 25 or as a 5x5 array.

5. SIZE : This applies only to hardware arrays. An array could have an arbitrary number of dimensions.

6. STATES : This refers to the number of states a storage item can remain in. If unspecified it is assumed to be 2.

7. ISTATES
   OSTATES
   CSTATES : These refer to the number of states the I,O and C lines may assume. If unspecified they are assumed to be 2.

8. DELAY : This applies to buses, delay lines, gates, switches, etc.

A user may define new attribute-types for any of the above data types, and declare modes of interpretation on the basis of such attributes. In the next section we shall see the kinds of facilities available in CDL1 for defining data types and other objects. A user should use these facilities only if he wishes to create new data types or other objects which have not been already defined in the language. One may skip the next section in the first reading, and go directly to Section IV-C to get a quick look at the language.

B. Definitions

Six kinds of definitions may be made in CDL1:

(1) Data definition.

(2) Hardware definition.

(3) Table and tabular function definition.

(4) Function definition.

(5) Macro definition.

(6) Command definition.

In each of these one may either define an individual item with a given name, or a class of items characterized by a type, and possibly, also attributes. Individual members of such a class may later be named and declared to a description file. For example, one may define a class of data characterized by the type name, TREE, specify its data structure and also define operations on it. Later in a description one may declare trees with given names, and use them in manners specified in the TREE definition. Similarly, one may define a new hardware-type called, say CPU (Central Processing Unit), and specify its function, structure, and operation. In a system description, one may declare as many of these as necessary, naming them each separately, or naming them as members of an array of CPU's. Let me discuss in some detail the convention for data definition through examples, since it is of some interest.

1. Data Definition

In order to define a software data-type or a datum one should declare one or more of the following items:

(i) Attribute definition,
(ii) Declaration format,
(iii) Declarations,
(iv) Data structure,
(v) Patterns,
(vi) Print format,
(vii) Interpretation, and
(viii) Type changes.

Let me explain what these items are.

(i) Attribute Definition This may be necessary in case one wishes to define new attributes to a standard data type or new data types with attributes. The attribute-types one wishes to use are enumerated and defined in this section. Thus, for a scalar its attribute definition might look as follows:

```
'1. BASE      ::= INTEGER;
 2. TYPE      ::= 'FIXED' | 'FLOAT';
 3. MODE      ::= 'REAL' | 'IMAG';
 4. PRECISION ::= (INTEGER, INTEGER).',
```

where the items on the left of '::=' are the attribute-types, and those on the right, their possible values. The value of an attribute-type in CDL1 could be a string constant (like 'FLOAT', 'REAL', etc.) or a pattern (like INTEGER), which is already defined in the language, or could be any data of a specified data-type.

(ii) Declaration Format This format is to be used to declare token members of a given data type to a description file. A declaration format should be specified only when one is defining a new data type or a standard data type with new attributes. For a scalar, the declaration format might look like:

'NAME S[BASE,TYPE, MODE,PRECISION]'

indicating the order in which the attributes are to be declared. 'NAME' here refers to a pattern, already defined in CDL1, as a 'letter' followed by an arbitrary number of letters and/or digits.

(iii) Declaration This may be used to declare directly to a description file a datum that is being defined. The datum should have a distinguished name (different from other names already declared to the files). It may or may not have a data type associated with it. Depending upon the data type and attribut s associated with it the following cases arise.

Case a A standard data type with no distinguishing attributes:

In this case the datum will obtain the standard interpretation in all contexts of its use in CDL1.

Case b. A standard data type with a distinguishing attribute:

In this case, on the basis of the distinguishing attribute, one may define a new data structure to be used exclusively for the datum only. If no new data structure is defined then the datum will have the structure associated with its standard data type.

So also one may define new interpretations for some or all of the operators operating on the datum, or functions using the datum. For operators and functions not so defined the interpretation will depend only on the data type of the datum.

A typical use of this kind of definition could be the following. It is desired to create a scalar array, with the usual indexing facility, and the usual operations within expressions. However, since most of the array elements are going to be zeros, the data structure for the array is to be different; only the non-zero elements of the array are to be stored. A new data structure for such an array may be defined on the basis of a distinguished attribute associated with it. This data structure will determine the manner in which elements of the array are to be accessed, modified, or created.

Case c No data type or a nonstandard data type:

In this case it is mandatory that a new data structure be defined, and also that every possible interpretation of the datum in expressions be specified. The datum may be used only in contexts so defined.

Whenever a datum is defined under Case b , or Case c with a nonstandard data type then the data types, and the attributes will them-

selves assume the status of having been defined. That is, one may later declare other data having the said data types and attributes; these will be given exactly the same interpretation as the originally defined datum. Considerations of Cases b and c thus apply also for data type definition.

(iv) Data Structure There are several ways of specifying a data structure. In the case of elementary items, like scalars, it may be specified in terms of the storage format of the scalar: The storage format of a base 10, FIXED, REAL, PRECISION (m,n) scalar could be:

'⟨DIGIT ·/m+n,m+n/· DIGIT⟩'

where DIGIT is a pattern defined in CDL1, 'digit ·/m+n,m+n/· digit' is a string expression denoting exactly m+n occurrences of digit, and the angular brackets, '⟨...⟩', called the value brackets, denote the value of the string expression as being the desired format, in contrast to the expression itself. Each digit is to be stored in terms of its machine representation, its code, which itself may have to be declared.

One may also describe a data structure in terms of algorithms for address calculation. For example, the address of the $(i,j)^{th}$ element of an array may depend on the address of its $(1,1)^{th}$ element as follows:

ADDRESS (ARRAY[I,J]) = ADDRESS(ARRAY[1,1]) + 100 * (I-1) +
5 * (J-1).

where 100 and 5 are the characteristics of a particular array organization.

To describe more complex data structures consisting of aggregates of other data structures, it is desirable to have the following additional facilities [14]: Facilities to

a) declare the addressing structure in terms of linkages among the elements of the data, and

b) declare functional relations among the various elements in a data aggregate: Like NEIGHBORS of a node in a graph, or DESCENDENTS of a node in a tree, or the TAIL of a list, etc.

We shall illustrate these facilities through an example.

EXAMPLE: Definition of a generalized list: By a generalized list, we mean one whose elements could be of arbitrary data types. One of the elements of the list could be, say, a scalar, another an array and the third, even another list. We shall define first the HEAD of a list and then define the notion of the NEXT element in a list. We shall then define an indexing convention to refer to elements in a list by their position within the list. Also, we shall define the TAIL of a list as another LIST consisting of all the NEXT elements in the list. Such a description might appear in English as follows:

D6. 1. The POINTER to a LIST points to either NULL, or the HEAD of a LIST which could be of any data type, and either the NEXT of the LIST which could also be of any data type, or NULL.

2. The HEAD of a LIST points to ITSELF.

3. The NEXT of a LIST points to ITSELF, and to either NULL, or NEXT of NEXT, which again could be of any data type.

4. NEXT of LIST is the same as NEXT of HEAD of LIST.

Schematically D6 implies the pointer structure shown in Fig. 2.

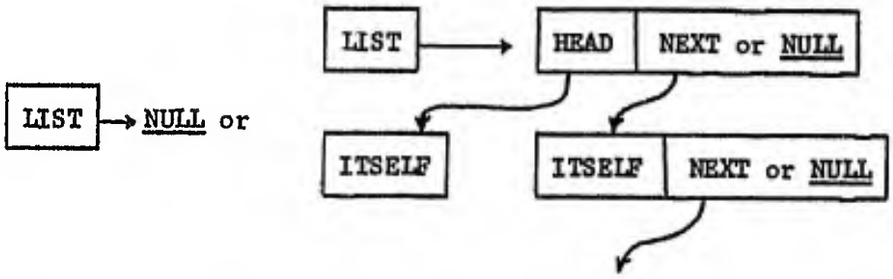

Fig. 2 Schematic of List Structure Described in D6

One may write this formally as follows:

D7. LIST → {HEAD:ANY | {NEXT:ANY,NULL},NULL};

HEAD → ITSELF;

NEXT → ITSELF | {NEXT:ANY,NULL};.

Here the chain brackets denote the selection of one of the items enclosed, separated by commas. The words ITSELF and ANY have been used as reserved words; 'ANY' denotes 'any data type'. One may now describe the indexing scheme and functions on a LIST as follows:

D8. Indexing Scheme for a LIST:

LIST[1] = HEAD(LIST);

NEXT(HEAD) = NEXT(LIST);

FOR 2 =< I =< SIZE(LIST)

LIST[I] = NEXT(LIST[I-1]);.

The SIZE(LIST) could be the total number of elements in the list.

D9. Functions on LIST

1. DECLARE

TAIL(LIST), Data-type LIST;

TAIL(LIST) = LIST[2 TO SIZE(LIST)].

(v) Patterns For a string operand, with or without attributes, one may define a pattern for the values which the operand may have. When the operand is later used in a description or a procedure its value should always satisfy the pattern defined for it. An example of pattern definition occurs in the 'Parallel Processing Procedure' in Section V. So we shall not discuss it any further here.

(vi) Print format This is used to declare the way a datum is to appear at the input and output. Specifications may have the usual kinds of format statements.

(vii) Interpretation In this section one may define algorithms describing the operations performed by operators (standard operator symbols used in the language) on the various data types.

(viii) Type changes In this section one may specify the conventions for changing the type of a data type from one to another.

2. Hardware Definition

A hardware definition may consist of one or more of the following items:

(i) Attribute definition,
(ii) Declaration format,
(iii) Declarations,
(iv) Structure,
(v) Signal characteristics,
(vi) Operations,
(vii) Compatibility,
(viii) Properties.

Under 'compatibility' one may declare the hardware types, with or without attributes, which should be mutually input/output compatible. All signal lines are assumed to have logical values, which could be multi-valued. Under 'signal characteristics' one may associate voltages, currents or other physical quantities with the logical values of a line. The notion of

compatibility will depend on the signal characteristics so defined. Under 'properties' one may list any property of interest to be associated with an item. Each such property must be titled. The value of a property could be any string.

All hardware items will have the standard set of attributes, which were enumerated in Section IV-A. One may define additional attributes and describe items on the basis of such attributes. Examples of hardware definitions appear in Section IV.

3. Table and Tabular Function Definition

One may declare names to the rows and columns of a table and specify table accessing and updating algorithms. Also one may specify table updating rules, which may depend on the input strings to the document files. Every 'row name (col.name)', or 'col.name(row name)' have the status of a table look-up function.

In a procedure one may assign values to the functions, col. name (row name) or row name (col. name). If a table look-up function is called before it is defined then a 'SKIP' condition will be set on the statement in the procedure issuing such a call. One may name such SKIP conditions, as part of the restrictions on a statement, as we shall see, and later use them in a procedure in various ways. Once it is used its value will be reset to zero.

4. Functions and Macro-Definitions

The definition of these is rather straightforward, and we shall not discuss them any further.

5. Command Definition

In a command definition module one may declare one or more of the following items:

(i) Command format,

(ii) Functional descriptions at various levels of design. The levels are distinguished by labelling: Functional Description 1, Functional Description 2, etc. Some of these could be data-flow sequences.

(iii) Operational descriptions at various levels of design.

In the command format one may specify the command name, and its pattern within a machine.

The syntax and format for all these definitions appear in the 'formal definition of CDL1'.

C. Operators and Expressions

Operators are classified as unary, binary and string operators. The unary operators are

$$\neg, \quad \text{MOD}, \quad \text{CLG}, \quad \text{FLR}, \quad +, \quad -,$$

where '$\neg$' is logical NOT, 'MOD' is modulus, 'CLG' is the ceiling function, $x \leq \text{CLG}(x) < x+1$, 'FLR' is the floor function $x-1 < \text{FLR}(x) \leq x$, and '+' and '−' are arithmetic signs. The binary operators are:

$$+, \quad -, \quad *, \quad \%, \quad -., \quad **, \quad \text{MOD}, \quad \wedge, \quad \vee,$$

$$(+), \quad \neg\equiv, \quad \equiv, \quad \neg=, \quad =, \quad =<, \quad <, \quad >, \quad >=,$$

where '%' is the divide symbol, '−.' is proper subtraction, (+) exclusive-or, 'x MOD y' is x modulo y, and '$\equiv$' is the symbolic identity operator. The rest of the symbols have their usual significance. ('x' $\equiv$ 'x') has value

1 (TRUE), whereas ('x' ≡ 'y') has value 0 (FALSE). (x ≡ y), without quotes stand for (value of x ≡ value of y) where x and y may themselves be string expressions or string names. We may also have the combination ('x' ≡ y) or (x ≡ 'y') with the corresponding meanings. '=' is numerical or logical value equality operator. (x = y) always implies value of x = value of y. 'x' = 'y' has no meaning.

All binary and unary operators are initially defined on scalars, and then extended to vectors, matrices and arrays component-wise. A binary operator on nonscalar operands is well defined if and only if the operands have the same dimensionality. Thus,

x[1], x[2], x[3] + y[1], y[2], y[3] =
x[1] + y[1], x[2] + y[2], x[3] + y[3], and
x[1], x[2], x[3] + y[1], y[2] is undefined.

A special case arises when one of the operands is a scalar. Then the scalar is distributed throughout the second operand for each element of the non-scalar. Thus,

x[1], x[2], x[3] + y[1] = x[1] + y[1], x[2] + y[1], x[3] + y[1].

The above conventions apply also to all relational operators, namely ≡, ≠, ≤, <, ≥, >, ¬= and ¬≡. '≡' and '¬≡' will operate component-wise in the case of string vectors and string arrays.

The string operators are

‖, ‖., ‖:, ..., ·/m·n/·,

where ‖ is the string concatenation operator, ‖. is the row concatenation operator and ‖:, the column concatenation operator. '...' is a string generator symbol. (x...x) stands for an arbitrary number of occurrences of x, including zero occurrences (the NULL string). In general, in

(string 1...string 2) string 2 should always be a prefix of string 1. String 2 identifies the beginning of the string which is to be repeated, the termination of the string to be repeated being determined by the first '.' in '...'. If string 2 is not a prefix of string 1, then the expression has NULL value.

(x ·/n·m/· x) denotes at least n and at most m occurrences of x. If m < n then it has NULL value. If m or n is vacuous then the corresponding limit is undefined.

In CDL1 expressions are classified according to the data type of the result they would produce. Thus we have

Sexp, Vexp, Mexp, Aexp, IVexp, STexp, STAexp, STVexp, IVAexp, BSTexp, BSTVexp, and BSTAexp.

Arithmetic, boolean and string operators may occur intermixed. String operators have precedence over others. The syntax of the various expressions are given in the 'Formal Definition of CDL1'. Also, expression interpretation under data-type changes of operands are explained. Let me briefly explain here the reduction and selection expressions:

For any binary operator B, (B/Vector) is a reduction expression. For example +/(3,8,7,2) = (((3+8)+7)+2) = 20. In the case of arrays (B/[i_1,i_2,...,i_n] Array) stands for a reduction of the array first along dimension i_1, then i_2, and so on. The dimension of the resultant array will be n less than its original dimension. Reduction operations have been extended to strings with appropriate conventions.

'{...,...,...,...}' is a selection expression, which stands for 'any one of' the items enclosed, separated by commas. In the case of string vectors '!' is used as the separator of string elements, instead of commas. In CDL1, '!' is not a character; it is used exclusively as a delimiter.

'(BitString/String or Vector)' is also a selection expression, when the lengths of the items on either side of the slash match. The elements in string or vector corresponding to the 1's in the bit string are selected.

One may also have a binary selection expression: (/Vector; bit string; Vector/) denotes the selection of the elements of the first vector corresponding to 1's in the bit string, and those of the second vector corresponding to 0's. Similar expressions are available also for strings, bit strings, string vectors and BSTV.

One may also have cyclic and noncyclic shift operators on vectors, strings and BST. In all these we have essentially adopted the operations first proposed by Iverson [6].

There are also expressions denoting number representations and number valuations.

(k)N/(vector or string) denotes base k value of the vector or string (if such an interpretation is possible for the string). Similarly,

(I(J)R/Scalar) denotes base I, J digit representation of the scalar.

Let me conclude this section with a note about notations. We have freely used symbols as we saw fit. In the internal representation of a description each one of these symbols may have a code. In any implementation of CDL1 it is possible to introduce a 'notation converter' at the I/O interface to which a user may declare the notations he wishes to use. The user's character set may be less than that proposed in CDL1. In such a case the user may have to define character strings in his alphabet to correspond uniquely in a one-one fashion, with the character set of the

language. Thereafter, one may use one's own notational conventions for the descriptions; the 'notation converter' will translate such inputs to the standard internal storage codes. In the output process the internal codes may be retranslated to the notations of the user concerned. In this manner different users utilizing a design aid system may each obtain symbol and notational conventions suited to his own I/O equipment characteristics and personal preferences.

The notational schema used should not be considered as the essential part of a language. One may if one chooses redefine a language with different notations, maintaining in some well-defined fashion a one-one correspondence with its original version. The features of a language lies in the kinds of operands, expressions, statements, naming schemes and declarations it provides, and the kinds of facilities it may have for program control within an algorithm.

The variety of expressions in CDL1 is quite large. Most of these, if not all, will find application in the description of a machine.

D. Functions

CDL1 has in all about 24 system functions. A list of these is given below without much explanation. The list includes the functions first proposed by Iverson [6].

1. MAX (I,...I)
2. MIN (I,...I)
3. LENGTH (string)
4. SIZE (ARRAY) : Also ISIZE, OSIZE, CSIZE, and other attribute functions
5. DIMN (Vector)

6. STATES (signal line or a memory unit)

7. COUNTN (string!...string! string!).

   Nonoverlapping counts of occurrences of strings 1,2,...etc. in the last string. The result is a vector.

8. COUNTO (string!...string! string!)

   Overlapping count.

9. COUNT{O,N} ({string!...string} string!)

   Overlapping or nonoverlapping count of occurrences of any one of the strings within {...} in the last string.

10. FULL(I) a vector of I "1's".

11. ZERO(I) a vector of I "0's".

12. CH(I,J) characteristic vector of dimension I with a 1 in position J.

13. PX(I,J) Prefix vector of dimension I. The k^{th} element is 1 if $k < J$.

14. SX(I,J) Suffix vecto of dimension I. The k^{th} element is 1 if $k \geq (I-J)$.

15. BITS(I) denotes an arbitrary vector of 0's and 1's of dimension I.

16. BITS(I,J) Vectors of 0's and 1's with exactly J 1's, and of dimension I.

17. DITS(I) An arbitrary vector of decimal digits of dimension I.

18. DITS(I,J) The sum of elements is J.

19. INTRL(I,J) Equals (I, I+1,...,I+J-1) or (I,I-1,...,I-J+1).

20. BLANK(K) K blanks.

21. INSERT (string; I; string) Insert string 1 at the I^{th} position of string 2.

22. REPLACE ((string! string!); string) Replace every occurrence of string 1 by string 2 in string 3. Only nonoverlapping occurrences of string 1 are to be reckoned, in a left-to-right scan.

23. FIRST (Vector) The result is a vector with a 1 in the first non-zero element of the argument.

24. LAST (Vector) The result has a 1 in the last non-zero element of the argument. The result could be all 0's in both 23 and 24.

E. Statements

1. Declarative Statements

There are six kinds of declarations possible in CDL1.

a. Declaration through a 'DEFINITION' or a 'DECLARATION' module.

b. Through a LET-statement within a module.

c. Implicitly through usage on the left side of an assignment statement.

d. Through a dynamic declaration of the data type of an item.

e. RELEASE statement.

f. Comments.

Let me now briefly explain the nature of these declarations.

a. The kinds of items that may be declared through definition modules were discussed in Section IV-B. The scope of such declarations is limited to the system in whose description they appear.

b. Through a LET-statement one may make the following kinds of declarations:

Conditional or unconditional

(i) symbolic equality,

(ii) replacement rule, or

(iii) value equality declarations; and/or

(iv) a data type which has been already defined,

(v) event names and their initial values,

(vi) USE-commands,

(vii) DELAY and WAIT declarations,

(viii) SAVE-declarations,

(ix) SCOPE-declarations.

A LET-statement may appear anywhere within a procedure (or a description), in any type of module. The statement begins with the reserved word 'LET'; this word may then be followed by as many of the above nine kinds of declarations as necessary; it is finally terminated by the '!' symbol. The word 'LET' introduces a new level of labelling. The declarations within a LET-statement may be labelled arbitrarily with mutually distinct labels. In CDL1 a group of statements with a new level of labelling, but with no block structure is called a group.

Let me briefly explain the nature of the above nine kinds of declarations.

(i) Symbolic Equality Declaration

1. JIM ≡. JOE;
2. JIM ≡. JOE IN label IN label IN PARALLEL PROCESSING PROCEDURE, GLOBAL
3. JOE ≡. JOHN /// SAVE;
4. '(+)' ≡. '⊕';

are examples of symbolic equality statements. Their general forms and interpretation are as follows:

1. "Label." Name"btype" ≡. Name"btype""IN generalized label" "/// SAVE";
2. "Label"'string' ≡. 'string';

The above forms are to be interpreted as follows: The label is optional; this is indicated by the double quotes enclosing it. The label is to be followed by a period '.'. "b<u>type</u>", where b stands for a blank, the "<u>generalized label</u>" and "/// SAVE" are also optional. A generalized label has the following form:

⟨label IN...label⟩⟨IN module title...IN module title⟩

"⟨[GLOBAL, ⟨↑...↑⟩]⟩"

The generalized label in example 2 above is an instance of this form. It points to the block or module, in which the named item had been declared.

There are two kinds of generalized labels: one is with the tag GLOBAL, and the other without the tag, but possibly with an arbitrary number of up-arrows of the form ⟨↑...↑⟩. The latter is called a LOCAL label. The global label points to an item with reference to the root of a search tree, and the local label points to an item with reference to its own position in the tree. Every '↑' denotes a jump from a node in the tree to its <u>parent</u> node. The label decoding schemes are described in Section IV-G.

The symbolic equality statements have the following meaning: In its first form it identifies two names as being synonymous; both names denote the same data. In its second form it identifies two strings as being symbolically equivalent. Thus, in example 4 the string '(+)' is being identified with the exclusive-or symbol ⊕. Both identifications are valid only within the scope of the declarations.

The name on the left of form 1 always refers to an operand within the scope of the declaration: The operand may or may not have been declared earlier; in the latter case its appearance in the statement will be construed as a declaration of the operand. The type of a name may be omitted only

if it had been declared earlier. If the types of the two operands are different, then the statement is valid only if a change of type convention for the two types exist. The '/// SAVE' is a restriction on the statement having the following significance:

Without the restriction the data denoted by the names will have two copies of it stored under the two names. Both copies will be kept simultaneously updated. With the 'SAVE' restriction, only the name on the left side will be kept updated, and the one on the right will maintain the old value it had.

In the case of form 2 the two strings will be treated as being symbolically equivalent to each other in all contexts of their appearance within the scope of the statement, with the following restriction:

The strings should be bounded on either side by a string constant, in every instance of their appearance within the scope of the statement, unless the strings are so bounded in the statement itself. A string constant is any one of the following set of symbols:

blank, any operator symbol, (,), [,], {, }, ⟨, ⟩, ⌐, ≡., =.,
::=, /, ↑, →, ←, ,, ;, :, ., !, ‘, ’, “, ”.

This form of the statement may be used for taking note of a notation conversion locally within a description.

Both symbolic equality statements may appear as conditional statements, with IF - THEN - ELSE clauses. The condition should always be a boolean condition, which may include relational operators. In that case the statements will assume significance only on the satisfaction of the conditions. The variables used in the boolean conditions should themselves have been declared within the scope of the statement.

The symbolic equality statement will be very useful in a design environment where different designers might have used different names for the same object, which might have to be later identified as being synonymous. Also, it is useful to invoke in a procedure or a description a variable declared elsewhere.

(ii) Replacement Rule

A replacement rule has the following form: Let 'stid' denote a string identifier, which could be a string constant, or a string under quotes, as '...', or a string name. A replacement rule is,

"Label." stid ::= ⟨{stid, string expression}⟩ "/// restrictions";

The restrictions can be any relational or boolean expression on the two stid's and/or the string expression. The rule will be executed within its scope only if the restrictions are satisfied, at every instance of its application. The rule has the following meaning.

It is a mandatory replacement rule. Every instance of appearance of the stid on the left side of the rule is to be replaced by the right side within the scope of application of the rule, provided that the following restrictions are satisfied:

1. The restrictions on the rule itself.
2. Every appearance of the left side stid is bounded by string constants, or the stid itself is so bounded in the rule.

If 'SAVE' is one of the restrictions then the execution of the rule will be stayed; in this case the rule will be executed dynamically during interpretation of the module or block in which the declaration appears. The old form of every statement within the scope of the rule will be maintained.

If the boolean and/or relational conditions in the restrictions are not satisfied then a flag will be set, and the name of the flag will be the label of the statement.

This rule may be used to edit a text within the scope of a declaration. The rule may appear also in a conditional form with IF - THEN - ELSE clauses. In such a case the rule will assume significance only on the satisfaction of the IF-condition. No flags will be set if the IF-condition is not satisfied.

(iii) Value Equality Statements

"Label." ⟨data declaration format⟩ =. value.

This may be used to assign initial values to data names being declared. The statement is especially useful to assign initial values to various event names while declaring them. The value could be expressed in terms of other values already declared, or soon to be declared within the block or module in which the statement appears. Thus, at the time of declaration the value may not be well defined. The value equality statement also has a conditional form with IF - THEN - ELSE clauses.

(iv) Data Declaration

Any data of a specified recognizable type may be declared according to its declaration format, with or without label, thus:

"Label." Declaration format.

(v) Event Declarations

These are prefixed with the reserved word EVENT. The declaration will appear as follows:

"Label." EVENT⟨event name =. value,...

event name =. value⟩.

The events have significance only within the scope of the declaration.

(vi) USE Commands

These have the form:

"Label." USE ⟨name FOR ⟨{name or operator}⟩,

... name FOR ⟨{name or operator}⟩⟩;

The first 'name' in each 'name FOR etc...'phrase should be the name of a procedure, or a body of description. The second name could be that of a function or a macro. The operation could be any one of the standard operators in the language. This declaration causes the object denoted by the first name to be used in the computation of the process denoted by the second name or operator, for each 'name FOR etc.' phrase, within the scope of the declaration.

(vii) DELAY & WAIT Declaration

This has the form:

"Label." DELAY time unit.

The time unit can be in seconds, MICROS, NANOS, MINUTES, etc. It is interpreted as specifying the time to be taken to execute the body of procedure within the scope of the statement. One may also specify

"Label." WAIT UNTIL event expression.

This indicates that the body of the procedure within the scope of the statement should be executed only on the satisfaction of the given event expression condition.

(viii) SAVE Declaration

This has the form:

"Label." SAVE(({data-name, declaration format}),...

({data-name, declaration format}));

The values of the named data are to be saved after each execution of the procedure within the scope of the declaration.

(ix) Scope Delcaration

The scope of a LET-statement, if not declared explicitly, is the smallest program unit, a block or a module as the case may be, in which it appears. The declaration may appear anywhere within its scope.

If necessary one may declare the scope explicitly as follows: SCOPE (generalized label,...generalized label); or SCOPE WITHIN generalized label The SCOPE WITHIN declaration invokes all the variables declared in another block or module into the block or module of current interest. One may also declare SCOPE GLOBAL to declare globally within a given system.

A given scope declaration in a LET-statement applies to all the declarations made prior to it within the LET-statement itself, up to the previous SCOPE declaration in the statement. The LET-statement may thus have the following form:

'LET

declarations of items described above, as many as necessary!'

It is terminated by '!'.

The SAVE declaration may appear by itself outside of a LET-statement, anywhere within a procedure or description. The symbolic equality, replacement rule, and value equality statements may appear only within a LET-statement. So also, the EVENT names and their initial values. USE and DELAY commands may appear as restrictions on an assignment statement, as we shall later see.

c. Implicit declarations of data may appear through usage of the data name on the left side of an assignment statement, within a program or a description. The declaration will have the scope of the smallest program unit, a block or a module, in which it appears.

In both b. and c. the scopes of declarations will permeate all the submodules, blocks, and sub-blocks of the module or block to which they apply.

d. We have earlier, in Section IV-A, discussed the significance, use, and form of a dynamic type change declaration.

e. Save & Release Declaration

This has the form

"Label." RELEASE ⟨⟨{name, label}⟩,...

⟨{name, label}⟩⟩ "/// restrictions";

This releases the scope of a variable or a labelled declarative statement before the termination of its natural scope. The SAVE declaration was discussed in b. (viii).

f. Comments

Comments may appear anywhere and may contain any string. These are not formally interpretable. ':-' is the 'comment begin' symbol and ';' is its termination; a comment termination is indicated by the first free ';' that appears in it. A ';', under quotes, is not free.

2. Executable Statements

There are four broad classes of these:

a. Dormant Statements,

b. Autonomous Statements,

c. Connection Statements,

and d. Program Control Statements.

Within each one may have

(a) either conditional or unconditional statements,

and (b) serial or parallel iteration statements.

Every statement may also be either

(i) deterministic or

(ii) nondeterministic.

a. Dormant Statements

(1) Assignment Statement

"Label." Operand ← ⟨{exp, IFexp}⟩ "/// restrictions";

The type of operand should be the same as that of the expression, or there should be a conversion rule between the two types. An IFexp has the form:

'IF ⟨{Bexp, Rexp}⟩ THEN (⟨{exp, IFexp}⟩ "ELSE ⟨{exp, IFexp}⟩")'

where Bexp stands for a boolean expression, and Rexp for a relational expression using the operators <, =<, =, >= , >, ¬=, ≡, ¬≡.

The SIZE and DIMN of the two sides of the statement should match, otherwise, during execution, an interrupt will be caused. '←' is the assignment symbol. Let me explain, at this point, the kinds of restrictions that a statement may have:

(i) Rexp,

(ii) Bexp,

(iii) An IF-Rexp of the following form:

'IF ⟨{Rexp, Bexp}⟩ THEN (⟨restrictions,...restrictions⟩ "ELSE

⟨restrictions,...restrictions⟩");'

Notice that the bracketing of the THEN-clause to include the corresponding, optional ELSE-clause will make the expression unambiguous.

(iv) A series of indexed restrictions of the form:

'FOR I = lower bound TO upper bound CHECK

⟨restrictions,...restrictions⟩!'

(v) USE-commands of the form discussed earlier. In each 'name FOR ⟨{name, operator}⟩' phrase of the USE-command the name or operator occurring on the right side should be the ones appearing in the statement to which the restriction applies. The connection symbols, '←*' and '*←*' may be used as operators in a USE-command. So also, the assignment symbol '←'.

(vi) SAVE restriction of the form:

'SAVE ⟨name,...name⟩;'

The names should be the ones appearing on the left side of the assignment statement(s).

This will cause the old values of the names to be saved in a pushdown stack.

(vii) DELAY and WAIT restriction of the form:

'DELAY time unit' or 'WAIT UNTIL event expression'.

The time unit specifies the time to be taken for the execution of the statement. The WAIT UNTIL-clause will stay the execution of the statement until the given condition is satisfied.

viii) SKIP-restrictions:

In case the restrictions are not satisfied, instead of causing an interrupt, one may cause a flag to be set. This flag is called the skip condition. The flag is labelled in the restriction as follows:

'SKIP ‖ INTEGER'

where 'INTEGER' refers to 'positive integer'. Thus SKIP1, SKIP10, or SKIP483 could be the labels of the flags. A statement may have more than one restriction. Individual restrictions are separated by commas. The restriction field itself is terminated by the first free semicolon.

(2) Conditional Assignment Statement

Henceforth, we shall denote by 'statement' any statement in CDL1, including also a block of statements of the form:

BEGIN⟨statement...statement⟩ END.

Let IFCOND be ⟨{Bexp, Rexp}⟩,

THEN-clause be ⟨statement...statement⟩

and ELSE-clause be the same as the then-clause. The form of the statement is:

'IF IFCOND THEN (THEN-clause "ELSE ELSE-clause")"/// restrictions";'

The bracketting of the THEN-clause and the ELSE-clause makes the statement unambiguous.

(3) Nondeterministic Forms

The above forms of the assignment statements are deterministic, and are thus executable. Their nondeterministic forms have the status of a declarative, indicating the possible alternatives, but not specifying which one to choose. Nondeterminism may appear either on the left or on the right, or on both sides, of an assignment statement, as follows:

Form i) {operand,...operand} ← ⟨{exp, IFexp}⟩ "/// restrictions";

Form ii) operand ← {⟨{exp, IFexp}⟩,...⟨{exp, IFexp}⟩} "/// restrictions";

or Form iii) a combination of forms i) and ii).

A procedure containing nondeterministic statements is not executable. One may set up a system to request a rule of decision everytime such a statement is encountered during an execution. Such rules of decision may be named and later appended to a nondeterministic statement in the form of a USE-restriction of the following form: USE(decision rule name). These decision rules are called 'scheduling algorithms'.

A nondeterministic statement may be made deterministic also by appending to each selection expression, '{...,...,...}', an index in the following manner:

'{...,...,...}[I]'

where 1 =< I =< the # of alternatives.

Form iv) A statement may be nondeterministic also as a result of a nondeterministic USE-restriction on it. Such a USE-restriction may have one or more phrases of the following forms:

i) {⟨name,...name⟩} FOR ⟨{name, operator}⟩,

or ii) name FOR {⟨⟨{name, operator}⟩,...⟨{name, operator}⟩⟩}

or iii) a combination of forms (i) and (ii).

These forms may be useful to specify dynamic resource allocation, through scheduling algorithms, which may be specified separately, at any stage of a design process.

There is a special form of a conditional statement which is used as a short form for a series of conditional statements. This uses the nondeterministic statements of the forms i), ii) and iii) in a special sense, as follows:

Let 'ND-statement' denote a nondeterministic statement of forms i), ii) or iii), and let 'D-statement' denote a deterministic statement. Also, let NDTHEN-clause be:

'"BEGIN"⟨⟨{ND-statement, D-statement}⟩ ...

⟨{ND-statement, D-statement}⟩⟩ "END"'.

The short form for a series of conditional statements may then look like:

'"Label." IF {IFCOND,...IFCOND} THEN

(NDTHEN-clause) "/// restrictions";

The alternate IFCOND's should be mutually exclusive, and every ND statement in the NDTHEN-clause should have exactly the same number of alternatives as '{IFCOND,...IFCOND}'. The statement has the following meaning:

If the i^{th} IFCOND is satisfied then in each nondeterministic statement in the NDTHEN-clause choose the i^{th} alternative for execution. For every IFCOND satisfied execute all D-statements in the NDTHEN-clause. If no IFCOND is satisfied then skip to the next statement in order.

Notice that this special form is not a ND-statement by itself, even though it uses ND-statements within. The entire statement might, however, become nondeterministic as a result of a nondeterministic USE-restriction on it.

(4) Iteration Statements

(a) Simple Iteration

"Label." FOR I = lower bound TO upper bound "BY Number"

DO ⟨statement...statement⟩

"CONTINUE ("I," label)" "/// restrictions";!

In the above form one may use any index variable instead of I. The index variables in CDL1 are {I,J,K,L,M,N,Index variable ‖ #}. The lower and upper

bounds could be any scalar expressions. If "BY number" is absent then increment (or decrement) by 1 is assumed. If "CONTINUE(...)" is absent then the iteration will be performed beginning with the first statement following 'DO' everytime. Otherwise, through a CONTINUE statement one may declare iterations to be performed from the statement specified by the label.

For example, one could have a FOR-loop of the form:

```
"Label."  FOR J = 1 TO 28 + xy DO
                JIM ← 128;
                JOE ← MARY + 3;
              1. statement
                 ⟨statement...statement⟩
                 CONTINUE (1);
```

In this case the iterations will be performed from statement 1 everytime. In each FOR-loop one may begin an additional level of labelling. A statement inside a FOR-loop may be referred to from outside by using the generalized labelling schema. In the above loop, JIM and JOE will be set to their initial values in the first pass.

(b) Multiple Iteration

One may iterate simultaneously on several index variables. An example of such a statement is given below:

```
2A1.  FOR I1 = 1 TO 100,
          I2 = 22 TO -10 BY -2,
          I4 = x + 3 TO 81 DO
      statement
    1. statement
    2. statement
```

CONTINUE (I1,I2,1)!!

⟨statement...statement⟩

CONTINUE (I3,2)!

The above statement has the following meaning: Iteration is to be performed on indices I1 and I2 simultaneously; they are both to be incremented or decremented, as the case may be, simultaneously and the iteration is to be repeated beginning from statement 1 until both of them reach their 'upper bound'. If one of them reaches the 'upper bound' before the other then it will maintain its index value in all subsequent iterations until the other one also reaches its upper bound.

Thereafter, the execution is to continue past the 'CONTINUE (I1,I2,1)' statement, and the iterations on I3 are to begin, beginning from statement 2 everytime. During these iterations I1 and I2 are to maintain their previous values. When I3 also reaches its upper bound the execution of the FOR-loop is complete.

The FOR-loop should contain as many free '!' marks as there are index variables in the FOR-loop. The general form of such a statement is given below.

Let 'Inva' denote 'Index variable', 'Lb', lower bound and 'Up', upper bound.

"Label." FOR ⟨Inva = Lb TO Ub BY Number,...

Inva = Lb TO Ub BY Number⟩ DO

⟨⟨statement...statement⟩ CONTINUE (⟨Inva,...Inva⟩, label)

"/// restrictions"; ⟨!...!⟩ ...

⟨statement...statement⟩ CONTINUE ⟨⟨Inva,...Inva⟩, label)

"/// restrictions"; ⟨!...!⟩⟩

The reader may supply the following restrictions on the above form: All Inva's are to be distinct and the count of free '!' should equal the count of Inva's in the loop.

One may also have nested FOR-loops. In the case of multiple iterations the CONTINUE-statements are mandatory. A nondeterministic FOR-loop will contain nondeterministic statements. Each loop within a multiple FOR-loop may have associated with it its own set of restrictions.

(c) Parallel Iteration

A simple example of parallel iteration is the following:

"Label." FOR I = 1 TO 25 DO PARALLEL ⟨statement...statement⟩!

This has the following meaning:

All the 25 versions of the body of the loop consisting of the statements, one for each index value within the range, are to be executed in parallel, simultaneously. A parallel iteration may contain multiple indices, with the following form:

"Label." FOR ⟨Inva = Lb TO Ub "BY Number,...

Inva = Lb TO Ub BY Number⟩ DO PARALLEL

⟨statement...statement⟩ "/// restrictions";!

In this case all the sets of statements, one for each possible combination of the index values, are to be executed in parallel.

b. Autonomous Statements

(1) Serial Autonomous statement

This has the form:

"Label." WHEN Bexp DO

⟨statement...statement⟩ "/// restrictions";!

On the satisfaction of the event-control boolean expression the series of statements following 'DO' are to be executed serially (or according to the program control specified within). The WHEN-statement is subject to the restrictions specified for it. The statement is considered to be 'ON' only as long as the boolean expression itself is true (its truth value could be a pulse), unless the following happens:

The Bexp is turned off within the body of the statement, by turning off (setting to 0) one or more of its variables. In that case, a local 'lock' will be set up for the statement indicating that it has been turned on, and it cannot be reinitiated until the task has been completed. When the task is done the lock will be released.

If the Bexp is not a pulse, then it should be always turned off explicitly by setting the appropriate variables to 0 or 1, as the case may be, at the very beginning of the body of the statement; that is, a lock should be always created. The Bexp should contain only event variables which have been declared earlier, or will soon be declared.

None of the statements contained within can be a GO TO statement, to a label outside of the WHEN-statement.* Within the WHEN statement one may begin a new level of labelling, but its body does not become a block, unless otherwise made so explicitly.

In CDL1 a collection of statements with a new level of labelling, but without a block structure, is called a group. (The statements in a FOR-loop also form a group.)

---

*This restriction was pointed out by Gorman.

A branch to another part of a procedure can be made through a BRANCH TO-statement. One can branch to only another event, not a label. When a branch is encountered in a WHEN-group its lock will be released.

(2) Parallel Autonomous statements

This has the form:

"Label.""Bexp":: ⟨statement...statement⟩ ::

⟨[ANY, ALL, Bexp]⟩ "/// restrictions";

The first Bexp turns the statement on;* all the statements contained within are then to be executed in parallel. None of the statements may be a 'block', or a GO-TO-statement. One can have BRANCH-TO statements within. A new level of labelling is effective for the statements within. Thus, these statements form a group.

The second Bexp gives a termination condition in terms of the labels of the statements in the group. The labels here are used as pseudo-event variables. The entire statement is to be considered terminated when the Bexp of the labels becomes true; a label is considered true if the statement denoted by it had been executed. 'ANY' or 'ALL', instead of Bexp, indicates that the termination occurs when any or all of the statements in the group have been executed. When no termination condition is given, 'ALL' is implied.

An autonomous statement is nondeterministic if it contains non-deterministic statements.

c. Connection Statement

(1) Unconditional Form

Within a procedure a connection statement has the status of an executable statement. It should then contain only hardware items which have

*When it is vacuous the statement is to be turned on unconditionally.

been already declared, and it denotes a direct connection between two items. Inside a structure definition, they have the status of declaratives, which specify the available (or admissible) connections in a structure. In its simplest form it may look like,

"Label." name ← * name;

where the names refer to hardware items. The statement specifies a connection between the outputs of the item on the right to the inputs of the one on the left. In a procedure such a connection is to be made only if the lines are compatible, in their signal characteristics, or otherwise, have been declared as being compatible in their definition. One may impose restrictions on a connection statement as follows:

"Label." name ←* name /// restrictions;

The restrictions can be WAIT UNTIL event exp., or bounds that may appear on the indices of input/output lines. An example, of such a statement is the one appearing in the 70/15 memory description:

```
STACK--XDRIVES[I, 8*(J-1) + K,L] ←*
TRANSFORMER--OUTPUT[I,1,J,K,L] ///
1 =< (I,L) =< 2, 1 =< (J,K) =< 8;
```

This specifies the lines with the corresponding indices, given as functions of I,J,K and L, to be connected. Thus, for I = 1, J = 1, K = 2 and L = 1 STACK--XDRIVES[1,2,1] will be connected to TRANSFORMER--OUTPUT[1,1,1,2,1]. The indexing structure for these lines should have been declared earlier. In the case of WAIT UNTIL restriction the connection is to be delayed until the given boolean event expression is not true.

Explicit references to input (I), output (O) or control (C) lines of an item are made by the use of compound names:

name--INPUT, name--OUTPUT, name--CONTROL.

While declaring a hardware item one may assign names to subsets of its I,O or C lines. Thus, STACK--XDRIVES may refer to a subset of input lines of the stack, called XDRIVES, with a well specified index structure. We shall see examples of these in the 70/15 description.

In the case of hardware array the suffixes INPUT, OUTPUT and CONTROL will always refer to the inputs, outputs and control lines of the array itself. Thus, STACK--INPUT[1,2,3,5] will refer to the input line [1,2,3,5] of the stack, and not to the element [1,2,3,5] of the stack, even though STACK had been declared as a hardware array. The inputs, outputs and control lines of an element of a hardware array are referred to by the compound names:

name--ELEINPUT, name--ELEOUTPUT, name--ELECONTROL.

Individual lines may be referred to again through indexing on these compound names.

Nondeterministic forms of a connection statement always have the status of a declarative. The meaning of such a statement may be obtained by reading them thus:

name ↚ {name,...name} is to be read as:

The output of any of {name,...name} may be connected to the inputs of 'name' (on the left). The chain bracketed expression is read as 'any of'.

Other forms of nondeterministic connection statements are:

{name,...name} ↚ name; and {name,...name} ↚ {name,...name}.

Every such nondeterministic statement implies the existence of a switching network at the connection interface to execute instances of the admissible connections.

Of course, the nondeterministic statements may also appear with restrictions.

No connection declaration may contradict a hardware compatibility declaration. A connection statement is not valid if the hardware items involved have been declared as being not compatible. If no hardware compatiblity had been declared then a connection statement will impose compatibility on the items.

(2) Conditional Form

A connection statement may appear with IF-THEN-ELSE clauses. The IF-condition should be a boolean or relational expression, on variables already declared. As before, the THEN-clause is bracketed thus, (...), to include the ELSE-clause. Connection statements may also appear as part of autonomous statements.

d. Program Control Statements:

There are in all 10 program control statements in CDL1:

1. "Label." GO TO LOCAL label "/// restrictions";

One may GO TO only labelled statements within the same module; one may jump between blocks within a module.

2. "Label." INITAIATE generalized label "/// restrictions";

3. "Label." INITIATE ⟨event name,...event name⟩ "/// restrictions";

The initiate command initiates the labelled statements or the said events in parallel with the currently running process.

4. "Label." WAIT time unit.

The time unit can be in seconds, minutes, MACROS, NANOS, etc. The time unit may be specified by a scalar expression.

5. "Label." TERMINATE ⟨event name,...event name⟩ "/// restrictions";

This causes the said events to be turned off instantaneously.

6. "Label." RETURN.

Within a block or a module it causes the system to return to the point in a program next to the one from which it entered the block or module. The entry itself might have occurred as a result of a GO TO statement, a function or a subroutine call. If there is no place to return to then the system is to return to the console.

7. "Label." CONTINUE

Outside a FOR-loop this has the status of a blank statement.

8. "Label." CALL⟨{procedure name, generalized label}⟩;

9. "Label." BRANCH TO event name "/// restrictions";

This statement may occur only within autonomous statements. One may branch to only the statements within the same module.

10. TRIGGER control line name ", pulse name";

This causes the control line to be triggered with the given pulse, or else set to 1.

A program control statement with parameters is nondeterministic if there is nondeterminism in the parameter values. Thus,

GO TO {label 1, label 2,...}

is nondeterministic. These have the status of a declarative.

A program containing nondeterministic statements, itself becomes nondeterministic, and thus has the status of a declaration.

A generalized label has the form:

⟨label IN ... label⟩ ⟨IN module title ... IN module title⟩"⟨{GLOBAL, ⟨↑..↑⟩}⟩"

and it is decoded from right to left. The rightmost module title must be uniquely identifiable, and every other reference in the generalized label should denote a unique item in the context of the items selected to its right.

This completes our discussion of statements in CDL1. Let me now briefly discuss the indexing conventions in the language, and then describe the modular structure of descriptions in the language.

F. Indexing Conventions

1. Vectors

The elements of a vector are numbered 1 through DIMN of the vector. One may visualize the elements arranged in the left to right order.

(i) Name $\underline{V}$ [I] denotes the I^{th} element of the named vector for $1 \leq I \leq$ DIMN (Name). (The suffix '$\underline{V}$' is not necessary.)

(ii) Name $\underline{V}$ $[(i_1, i_2, \ldots i_r)]$ denotes a new vector of dimension 'r' consisting of the elements $i_1, i_2, \ldots, i_r$ of the named vector.

(iii) Name $\underline{V}$ $[i_1 : i_2]$ denotes

Name $\underline{V}$ $[i_1]$ if $i_1 = i_2$,

Name $\underline{V}$ $[(i_1, i_1+1, \ldots, i_2)]$ if $1 \leq i_1 < i_2 \leq$ DIMN (Name),

Name $\underline{V}$ $[(i_1, i_1-1, \ldots, i_2)]$ if $1 \leq i_2 < i_1 \leq$ DIMN (Name).

The elements of a string vector are strings and those of a bit string vector are bit strings. The name of a vector without indexing refers to the entire vector.

(iv) The individual characters (bits) of a string (bit string) in a string (bit string) vector may be denoted through double indexing as follows:

Name [i][j],

which refers to the j^{th} character (bit) in the i^{th} element of the vector.

Name STV[$(i_1,\ldots,i_r)$][j]

refers to a new string vector consisting of the j^{th} characters of the elements $(i_1,\ldots,i_r)$ of the old string vector.

Name STV[i][$(j_1,\ldots,j_r)$],

Name STV[$(i_1,\ldots,i_r)$][$(j_k,\ldots,j_k)$]

Name STV[i_1:i_2][j_1:j_2]

have all the corresponding obvious interpretations.

2. Arrays

Arrays are multi-dimensional entities, whereas a vector is always of dimension 1.

i) Name A denotes an entire array

ii) Name [$i_1,i_2,\ldots,i_k$] denotes the [$i_1,i_2,\ldots,i_k$]th element of an array if $1 =< [i_1,i_2,\ldots,i_k] =<$ SIZE (name).

iii) Name [$(i_1,i_2,\ldots,i_r)$, $(j_1:j_2)$, $*,\ldots,\ell$] denotes a subarray consisting of elements of the array denoted by the set cross-product of all the following indices:

$$(i_1,i_2,\ldots i_r)\times(j_1,j_1\pm1,\ldots,j_2)\times(1,2,\ldots,m)\times\ldots\times\ell,$$

where m is the size of the 3^{rd} dimension of the array. A '*' as an index denotes all the indices in a particular dimension of an array.

Thus,

JIM[1:2,(8,6,6), 8:7]

denotes an array of elements of JIM with the following indices:

(1,8,8), (1,8,7), (1,6,8), (1,6,7), (1,6,8), (1,6,7)

(2,8,8), (2,8,7), (2,6,8), (2,6,7), (2,6,8), (2,6,7)

Notice that in the new array some elements may repeat.

If the elements of an array are themselves strings, bit strings or vectors then one may use double indexing, as in the case of STV and BSTV, to denote individual components of the strings, bit strings or vectors.

3. Special Multiple Indexing Schemes

It may be sometimes convenient to consider an array as being made of two or more parts from each of which, everytime one or more elements may be selected, in anyone reference to the array. Thus, the drive lines of a memory may be structured as:

MEM--DRIVES[128;128]

indicating two sets of lines, 128 each (the X and Y lines) from which everytime one may select two lines, one belonging to each set. Notice that in this case the total number of line elements in the array is only 128+128 = 256, and not 128×128. Thus, if the size of an array is

$[i_1,i_2;\ j_1,j_2,j_3;\ k_1,k_2]$

then the total number of elements in the array is $(i_1 \times i_2 + j_1 \times j_2 \times j_3 + k_1 \times k_2)$. Such multiple indexing schemes may be useful to structure input/output/control lines of a hardware item, as we shall see in the 70/15 memory description.

G. The Modular Structure of Descriptions and the Generalized Labelling Schema

Descriptions in CDL1 are organized in titled units called modules. Each module title consists of two parts: a name* and a module-type. The module-types recognized in CDL1 are:

*name is optional.

1. SYNTAX,

2. FORMAT,

3. Various kinds of DECLARATIONS,

4. Various kinds of DEFINITIONS,

5. PROCEDURE,

6. DESCRIPTION.

'MEMORY SYSTEM, DESCRIPTION' is a typical module title, whose module-type is DESCRIPTION. '70/15, HARDWARE DECLARATION' is another module title, whose type is 'Hardware declaration'.

Every module-type has an associated format: The format specifies the subtitles of items which a type of module may contain. Each such subtitle defines a submodule of the parent module-type. Thus a 'Data Definition' module will have the eight subtitles discussed in Section IV-B-1. The syntax and format of data that a subtitle may have are also specified. In an interactive system one may use the module formats to indicate to a user the kinds of objects to be described in a given module-type, and the forms they should have.

A module may thus have submodules, which are contained within. Also, a module of a given type may contain other modules of different types within its scope. Thus, a system description module may contain every other type of module within its scope; a type 'Procedure' module may contain a Data Definition module as a submodule. The module containment relationship in a description will have, in general, a tree structure. It is this tree structure that makes a generalized labelling scheme possible.

Every part of a system description should belong to some module. Thus, module titles provide a naming scheme for various parts of a body of

description. A designer may create arbitrarily any tree structure of inclusion relationships, that is best suited for his purposes, using the various module-types provided in the language.

Every module title is delimited by a colon ':', and the module itself is terminated by an 'END module title' statement or simply by a triple vertical bar, '|||'.

We have already discussed the formats of some of the definition modules. The example in Section V will give an idea of the format of a type 'Procedure' module. The use of type SYNTAX and FORMAT modules will become clear in the definition of CDL1 itself.

Let me now describe the decoding scheme for a generalized label. Let us consider, for example the tree of inclusion relationships shown in Fig. 3. Let each node denote a module, or a group, or a block, or a label of an item in a group, block, or module. Thus, the Root R contains, say three modules, with titles N1, N2 and N3, each of which contain, say two submodules, and so on the tree of modules, blocks and groups grow. Finally, on the top we have two lists of labelled items (11 to 13) and (11 to 17).

From any one of the nodes in the tree a reference to the generalized label

'3 IN 2 IN L6 IN N3, GLOBAL'

will denote the search path indicated in Fig. 3 with double lines. The object denoted is the item at the top end of the path.

A reference to the local label

'13 IN L1 IN N1 IN ↑↑↑'

within the boxed node 16 will have associated with it the search path denoted by the arrows in Fig. 3. The object denoted is the node 13 at the end of

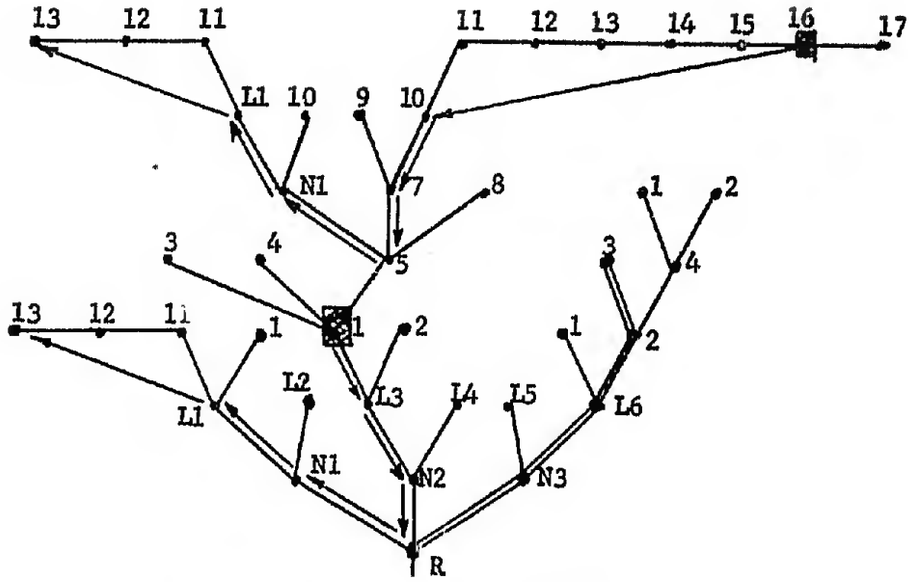

Fig. 3 A Tree of Inclusion Relationships

the last arrow. The same label, if referred to in the boxed node 1, will have associated with it the new path shown by the arrows beginning at node 1 in Fig. 3.

In both global and local labels the decoding is done right to left. Every up arrow, '↑', in a local label denotes a jump in the family tree from one node to its parent node. These arrows may occur only on the rightmost end of the label. In the global label above, 'IN R' is implied at the rightmost end of the label.

H. Block and Group Structure of Programs

The BEGIN-END blocks have the usual significance, already discussed in earlier chapters. In CDL1 there is another kind of block, called an Interpretation Block. This is used to describe the functions associated with the various named items in the language while defining the items. In a function definition module the function interpretation may appear as follows:

"Label." MAX S (Sexp, Sexp) ->

IF Sexp(*1) >= Sexp(*2) THEN

(MAX ← Sexp(*1) ELSE MAX ← Sexp(*2); <-

Here 'Sexp' stands for a scalar expression. One may read '->' as 'does' or 'means', whichever is suitable in a given context. The companion symbol '<-' indicates the end of interpretation of the item to the left of '->'. It signifies in the language a 'RETURN' statement. The program following '->' may refer to the items occurring in the string to the left of '->'. Thus Sexp(*1) refers to the first 'Sexp' from left in 'MAX(Sexp,Sexp)'. The result of the function is assigned to the name of the function. Thus a function name without arguments will always refer to its latest value.

The arguments (Sexp, Sexp) specify the type of arguments the function may have. The general form of an interpretation block is as follows:

(i) "Label." "name"btype" =." ⟨{expression, function name}⟩

"/// restrictions;" -> program <-

(ii) "Label." string "/// restrictions;" -> program <-

In form 2 the string should be the name of some item in the description.

In its first form the program might describe the algorithm to evaluate the expression or function, and assign its value to the fictitious name given. This name is local to the block and will not have any significance outside the block, unless otherwise declared in a procedure. The manner in which an interpretation block is called depends upon the kind of item it defines. Thus a function interpretation may be called by the function name, the actions initiated by a control signal of a hardware item may be called by triggering the control line, the interpretation of an expression may be called by recognition of the expression within a statement, etc. All such

expressions may contain only one operator symbol. The module-type of the module in which the interpretation block occurs controls the manner of its call. We shall see examples of this in the 70/15 memory description in Section V.

I. The Character Set

I have deliberately relegated this to the last part in this section, because I think it does not capture any essential features of a language, other than the incidental conveniences pertaining to a specific implementation of a language. No doubt, it might considerably influence the 'human engineering' aspects of a language. It is, therefore, all the more important that the character set be not fixed a priori. The language has been defined using a basic character set of 68 elements:

A,B,...,Z,0,1,...,9,

.,,,:,;,',?,",",

|,/,*,+,−,%,(,),[,],{,},⟨,⟩,

←,↑,≡,=,∧,∨,¬,<,>,b.

In an implementation one may choose another character set with fewer elements, if need be, and define some of the characters above as strings in the new character set. It is possible to devise schemes for consistency checks in such a notation conversion, and also create an I/O software interface which might translate the internal form of the language to its external forms, and vice versa.

The exclamation mark '!' is used purely as a string delimiter in the language. A string in CDL1 may not contain '!' as one of its characters.

V. 70/15 CORE MEMORY SYSTEM

We shall describe the memory system functionally and operationally to a certain depth of detail. The various parts of the description may well correspond to the various design stages of the memory itself. The data presented in this description have been obtained from '70/15 logic diagrams', 'correspondence programs', and 'status flows'. All the data given here do not appear at one place in any of the above manuals.

The description is not complete in the sense that the logic diagrams and other details of implementation have not been given. It captures the essential structure and timing details of the system. However, the description at various levels is complete in the sense that one may simulate the memory, if need be, by interpreting the description given here.

The kinds of details dealt with in this example give a good indication of how one may use CDL1 to describe complex structures in parts, at various levels of detail. The description is compact, precise and rather detailed in some parts.

It is probably not necessary to describe a memory system in as much detail as given here, in a design process. We chose this description to illustrate the complexity of details, which may be handled in CDL1. Unless the reader is genuinely interested in knowing about the 70/15 core memory system, it is not necessary to read through the entire example in detail. The functional description at the beginning is easy to read and gives a precise description of what the memory does, including the options on the two sizes of the memory. The structural and operational description is rather loaded with details, which are not easy to follow. The reader should probably read a few of the subsystem declarations and descriptions,

to get a feel for the style of description and declarations. The description of TP, the time-pulse generator, (the 3rd functional description) is probably representative of the brevity of expression and the details captured. An interesting example of the uses of parallel autonomous and FOR-loop statements occurs in the decoder description (item 8 in functional description). The structure declaration describes the entire memory system structure in terms of line to line connections of all its subsystems; yet it consists of only 16 statements. Through the use of signal line (input, output and control lines) indexing and naming schemes one may declare thousands of connections, if need be, through just a single statement. The reader may find it interesting to follow the details of one or two statements in the structure declaration.

The entire set of functional descriptions and the structure declaration are embodied in the operational description, presented at the end, which gives the sequence of controls initiated in the memory LOAD and GET functions. The description is not entirely operational, since the MR and MAR register loadings are not described in terms of their associated control signals. This operational description assumes significance by virtue of all the functional and structural descriptions declared earlier.

Let me first describe the declaration format for the hardware items. A hardware item can be a Flipflop (F), Register (R), Network (NW), Delay line (DL), Switch (SW), Gate (G), Bus (B) or an array of any one of the above items, denoted by

FA, RA, NWA, DLA, SWA, GA and BA, respectively.

The type of an item is identified by using the above abbreviations as tags, which are suffixed to the hardware names while declaring them. Thus, MAR R

denotes a register called, MAR and MEM RA denotes a register array called, MEM. During declaration one may specify one or more of the following attributes associated with hardware items in CDL1.

1. Array size: SIZE
2. Array Input size: ISIZE
3. Array Output size: OSIZE
4. Array Control size: CSIZE

These specify the number of input, output or control lines in an array, and the way they are structured for indexing. The SIZE of an array similarly specifies the number of elements in the array and their indexing structure. Thus, if the SIZE is declared as (64,64,2) it means that the array has 64x64x2 elements, any one of which is pointed to by a triple indexing scheme: If the name of the array is MEM, then MEM[I,J,K] will denote the $(I,J,K)^{th}$ element of MEM for $1 \leq I \leq 64$, $1 \leq J \leq 64$ and $1 \leq K \leq 2$.

5. Array Element size if applicable: ELESIZE

This specifies the number of storage units in an element of an array. In the case of MEM, the element size will be the size of the individual registers in MEM.

6. Array Element ISIZE: ELEISIZE
7. Array Element OSIZE: ELEOSIZE
8. Array Element CSIZE: ELECSIZE

These specify the number of I,O and C lines of each element in the array, and their indexing structure. Thus, an ELEISIZE = (5,5) will mean that the element has 5x5 inputs, any one of which is denoted by a double index (I,J) for $1 \leq (I,J) \leq 5$.

Unless otherwise specified all input, output, and control lines, and the storage units are assumed to be binary. A nonbinary signal line or storage unit may be declared by specifying the 'STATES' of the line or the storage unit. 'STATES' gives the number of states in which an item may remain.

After specifying the various sizes of I,O,C lines and storage, one may, if necessary, name subsets of these lines or storage units, for easy reference to them later in a description. Thus, if CSIZE = 2 for MEM one may name the individual control lines as, say

READ[1] and WRITE[2];

The first control line then assumes the name MEM--READ, and the second, MEM--WRITE.

In any one declaration it is not necessary to specify any or all of the above attributes. However, the type of a hardware item should always be indicated with the appropriate suffixes.

With every declaration one may associate restrictions. These restrictions may impose constraints on the values, which the various parts of an item may assume. For example, the declaration of the MR R of size 9, with the parity constraint may appear as follows:

1. MR R[9] /// MR[9] $=\neg$ ($\oplus$/MR[1:8]).

The triple slashes separate the constraints from the main body of the declaration. The above constraint specifies that the 9^{th} bit of MR has to be equal to the complement of the mod 2 sum of the bits 1 through 8 of MR. This declaration has the following interpretation: In every use of MR within a system its contents should always satisfy the given restriction. If at any time it does not, then it would cause an interrupt in the system operation.

In the case of arrays of hardware items, whose elements themselves are arrays, the nested indexing scheme will be used to denote an element of an element of an array. Thus, if MEM RA consists of 4096 registers, each of SIZE 9, then MEM[I][J] will denote th J^{th} bit of the I^{th} array element for $1 \leq I \leq 4096$ and $1 \leq J \leq 9$. Also, MEM[I] will denote the entire I^{th} element of MEM, and MEM[*][J] will denote the set of all the 4096 J^{th} bits of every register in MEM.

We may now proceed with the description of the core memory.

HARDWARE DEFINITION

DECLARATION:

1. MAR R [SIZE = 13];
2. MR R [SIZE = 9] /// MR[9] = $\neg\oplus$/MR[1:8];
3. MEM RA

   [SIZE = ⟨{4096,8192}⟩;

   ISIZE = 9; OSIZE = 9; CSIZE = 2;

   ELESIZE = 9;

   CONTROL LINE NAMING: READ[1], WRITE[2]]

   /// MEM[*][9] = $\neg\oplus$/MEM[*][1:8],

   MEM--READ $\wedge$ MEM-WRITE = 0;

:- This has two restrictions: The ninth bit of every register in memory should be equal to the complement of the mod. 2 sum of its 1 through 8^{th} bits. And, the MEM-READ and MEM-WRITE control signals should not both ever have value 1. The size of MEM could be either 4096 or 8192. (The 70/15 memory comes in two sizes);

:- It should be mentioned that it is not necessary to write every time the name of an attribute before assigning a value to it. One may order the

attributes in a certain fashion and choose to list their declaration according to the order. We have here chosen to write them down explicitly since there are quite a few of them, and declaring them in an order without names may not be very readable. If an interactive system is available, then one may make the system ask for the various attributes in a certain order. The user need not remember them all.;

END(DECLARATION)

FUNCTIONAL DESCRIPTION:

1. MEM--READ ->

WHEN MEM-READ DO

IF SIZE(MEM) = 4096 THEN

(MR ← MEM[N/MAR[1:12]+1]; ELSE

MR ← MEM[N/MAR[1:13]+1];)

/// DELAY 2 MICROS;! <-

:- N/MAR[1:12] denotes the binary number represented by bits 1 through 12 of MAR. The '+1' occurs because of our convention of numbering the memory locations from 1 through SIZE(MEM). (The '->' sign may be read as 'DOES'.) If the SIZE of MEM in an installation is 4096, then only bits 1 through 12 of MAR are used to compute the address. If the size is not 4096 then it is 8192, and in this case all the 13 bits of MAR are used for the address. The entire memory read operation takes 2 microseconds. For a discussion of the WHEN-statement the reader may see Section IV-E, on autonomous statements. The above interpretation block describes the function initiated by the MEM-READ control signal. This functional description is to be invoked, whenever the MEM-READ control is triggered in a body of description.;

2. MEM--WRITE ->

WHEN MEM--WRITE DO

IF SIZE (MEM) = 4096 THEN

(MEM[N/MAR[1:12]+1] ← MR; ELSE

MEM[N/MAR+1] ← MR;) ///

DELAY 2 MICROS;! <-

:- Notice that the above two interpretation blocks describe precisely the memory read and write functions in 70/15. The indicated assignments to the MEM and MR are to be made only if the parity conditions, specified in the hardware declarations, are satisfied. If the parity is not satisfied, an interrupt is to be caused.;

END(FUNCTIONAL DESCRIPTION)

END(HARDWARE DEFINITION)

:- We shall now proceed to describe the memory in greater detail. We shall follow the following scheme of description:

We shall identify each memory subsystem in terms of its input, output and control lines, and describe the operations initiated by the control lines for each subsystem. Then we shall specify the detailed interconnection structure of the memory system in terms of input, output, and control line connections of each subsystem. After this, we shall list the control sequences, with timing restrictions. Each control line triggered in a sequence is to cause its corresponding action description to be called in. A control line may also be triggered indirectly: A pulse or level signal applied by one subsystem at one end of a signal line, is to be traced to its other end, along paths specified in the structure declaration. If at the other end, the line is a control line of another subsystem, then its

corresponding action description is to get called in. We may thus capture the precise details of memory operation, with its component subsystems described either functionally or operationally. Also, in the case of a system design, as design progresses one may update subsystem descriptions, without having to modify other parts of the description (unless, of course, there are design changes). Since, the descriptions will be complete one may also simulate the system (or subsystems) in the course of their design.

The characterization of each subsystem in terms of its input, output, and control lines, and the specification of their indexing schema are, as we shall see, very important steps in the subsequent description of the entire system.;

HARDWARE DEFINITION:

:- This is to reopen the definition module which was closed earlier. All hardware definitions are to be filed in the same file, within a description module.;

DECLARATION:

:- The submodule, declaration, within the hardware definition module is now to be reopened. One may now request the filing system to issue the next item label in order.

Item 4 of the declaration is the memory stack, whose block-diagram is shown in Fig. 4. The reader should probably see the diagram first.

If a display console is available for I/O then one may declare these hardware items in terms of their schematic diagrams as shown in the figures. The filing system may construct from such a declaration the various attribute functions associated with the hardware item.

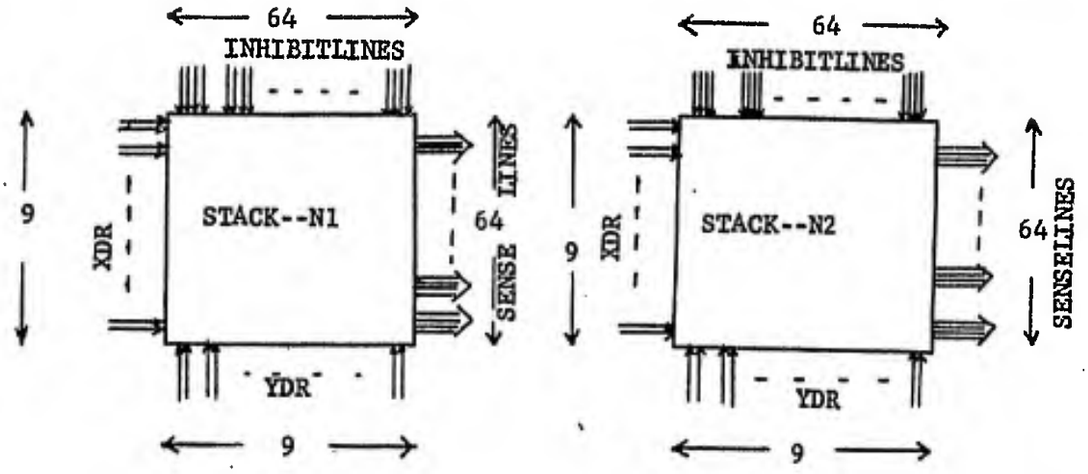

Fig. 4 Stack Configuration in 70/15

The declaration of STACK in CDL1 may appear as follows:

```
4.  STACK RA
      [SIZE = (2,64,64);
      ISIZE = (2,4,9);
      OSIZE = (2,4,9);
      CSIZE = (2,64,2; 2,64,2);
      ELESIZE = 9;
    STACK NAMING:  N1[1,1:64,1:64],
                   N2[2,1:64,1:64];
    INPUT NAMING:  INHLINES;
    OUTPUT NAMING:  SENSELINES;
    CONTROL NAMING: XDR[1:2,1:64,1:2;];
                    YDR[;1:2,1:64,1:2];];
```

:- This is a rather complex, and interesting hardware array declaration. The array, STACK itself has 2x64x64 registers, of element size 9. The registers with indices [1,1 THROUGH 64,1 THROUGH 64] in the STACK are to be

called STACK--N1, and those with indices [2,1 THRO 64, 1 THRO 2] are to be called STACK--N2. Thus STACK consists of two substacks. (This is precisely the case in 70/15 memory system.) The substacks were named in the above declaration in 'STACK NAMING'.

STACK has 2x4x9 input lines called, INHLINES. In the physical stack these are grouped as follows: For each plane in each substack there are 4 INHLINES terminals and each substack has 9 planes. If necessary, we could have named, N1--INHLINES as [1,1:4,1:9] and N2--INHLINES as [2,1:4,1:9], and PLANE1-INHLINES, as[1:2, 1:4,1], PLANE2--INHLINES, as [1:2,1:4,2], etc. Since, such naming is not necessary we have chosen not to do so.

In 70/15 memory, each memory plane has two independent inhibit line loops. Thus, there are 4 inhibit line terminals per plane.

Similarly, it has 2x4x9 output lines called, SENSELINES, with the same kind of distribution as the INHLINES. The control lines are 2x64x2 + 2x64x2 in number; the subset of lines denoted by indices [1 THRO 2, 1 THRO 64, 1 THRO 2;] are called XDR (X-drives) and the rest YDR(Y-drives). These are distributed in the physical stack as follows: each substack has 64 X-drives and 64 Y-drives, and each X and Y drive has 2 terminals in each substack. The schematic diagram of the stacks, reflected by the declaration, is shown in Fig. 4. This describes truely the structure of the stacks in 70/15.

The next item is the array of XFORMERS whose schematic diagram is shown in Fig. 5 The declaration of this schematic appears as follows. (The reader should see the schematic first.);

5. XFORMERS NWA

[SIZE = (2,2,8,8);

ISIZE = (2,2,8,8,2);

OSIZE = (2,2,8,8,2);

CSIZE = (2,2,8,8);

ELEISIZE = 2; ELEOSIZE = 2, ELECSIZE = 1;

XFORMERS NAMING:

STK1X[1,1,1:8,1:8], STK1Y[1,2,1:8,1:8],

STK2X[2,1,1:8,1:8], STK2Y[2,2,1:8,1:8];

INPUT NAMING: RDINPUT[1:2,1:2,1:8,1:8,1],

WRINPUT[1:2,1:2,1:8,1:8,2];

OUTPUT NAMING: CUROUT[1:2,1:2,1:8,1:8,1],

CURRETURN[1:2,1:2,1:8,1:8,2];

CONTROL NAMING: CNTRTAP;]; :-

:- The 'voltage switches' have the schematic diagram shown in Fig. 6. Their declaration is as follows:;

6. VOLSWITCHES NWA

[SIZE = (2,2,8);

ISIZE = (2,2,8);

OSIZE = (2,2,8);

VOLSWITCHES NAMING: STK1X[1,1,1:8],

STK1Y[1,2,1:8],

STK2X[2,1,1:8],

STK2Y[2,2,1:8];

INPUT NAMING: DECODEROUTPUTS;

OUTPUT NAMING: TOCNTRTAP;];

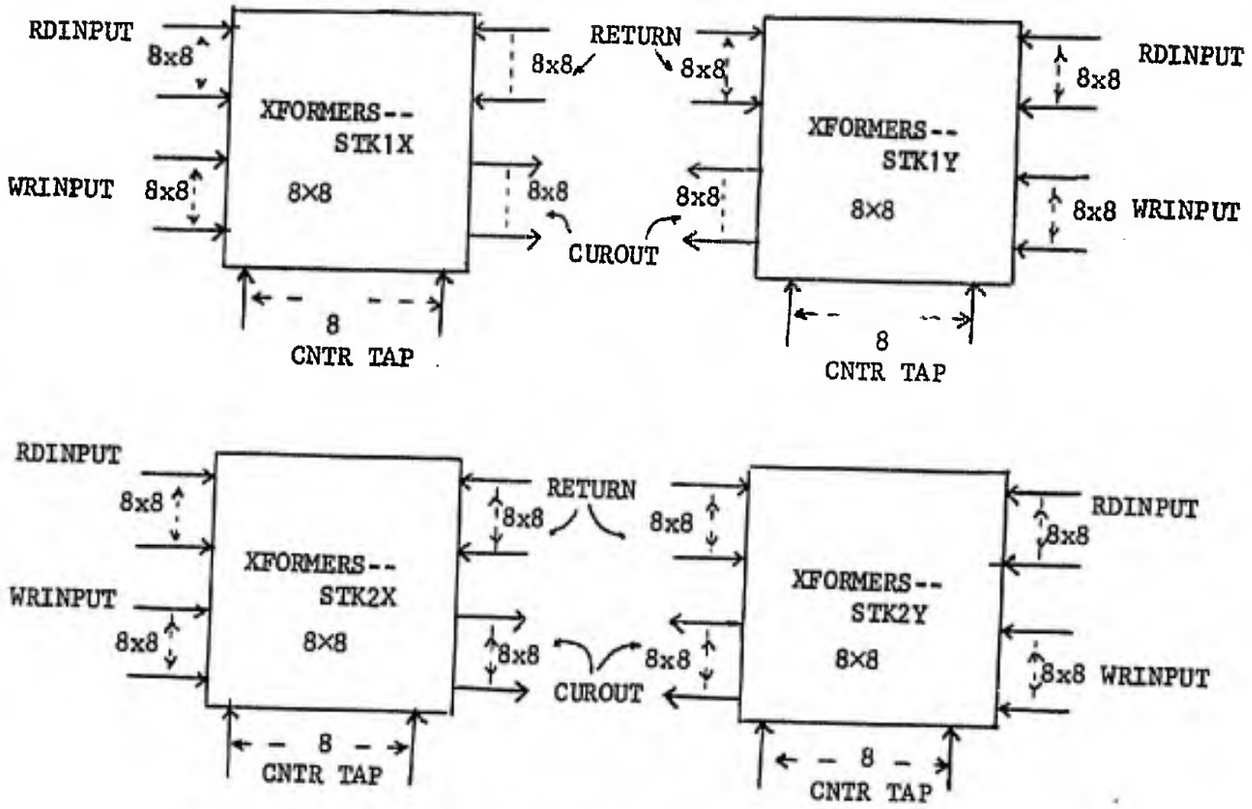

Fig. 5 Schematic of the XFORMERS Network Array

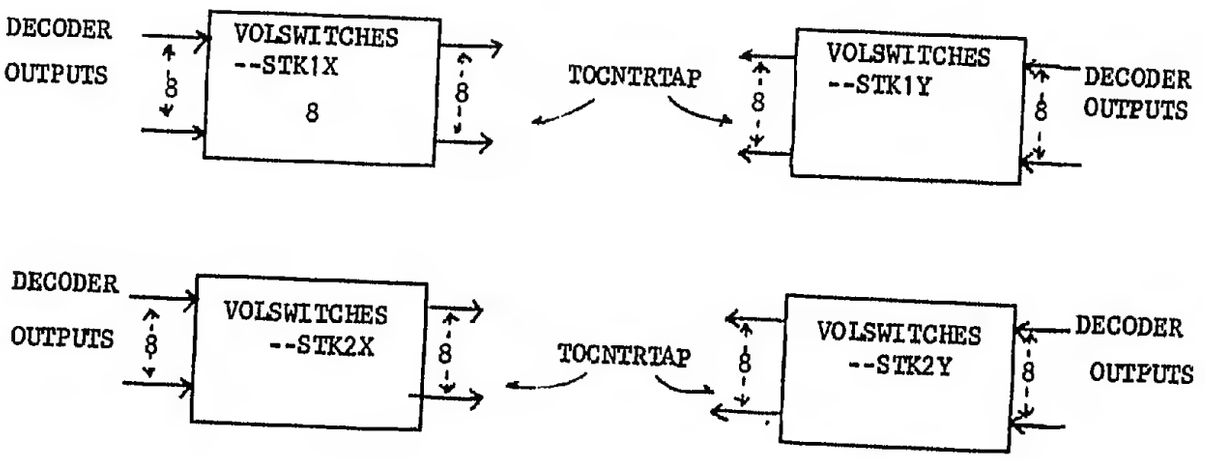

Fig. 6 Schematic of VOLSWITCHES, a Network Array of 32 Switches

:- Current switches, network array:

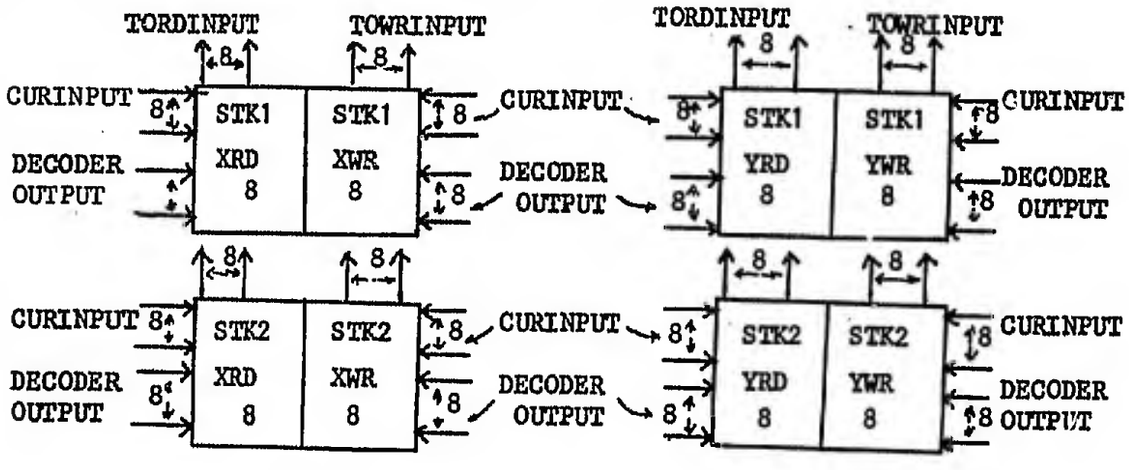

Fig. 7 Schematic of Current Switches, an Array of 64 Switches

7. CURSWITCHES NWA

[SIZE = (2,2,2,8);

ISIZE = (2,2,2,8,2)

OSIZE = (2,2,2,8);

CURSWITCHES NAMING:

STK1XRD[1,1,1,1:8], STK1XWR[1,1,2,1:8],

STK2XRD[2,1,1,1:8], STK2XWR[2,1,2,1:8],

STK1YRD[1,2,1,1:8], STK1YWR[1,2,2,1:8],

STK2YRD[2,2,1,1:8], STK2YWR[2,2,2,1:8];

INPUT NAMING: CURINPUT[1:2,1:2,1:2,1:8,1],

DECODEROUTPUT[1:2,1:2,1:2,1:8,2];

OUPTUT NAMING: TORDINPUT[1:2,1:2,1,1:8],

TOWRINPUT[1:2,1:2,2,1:8];];

:- Pulse generator network array:

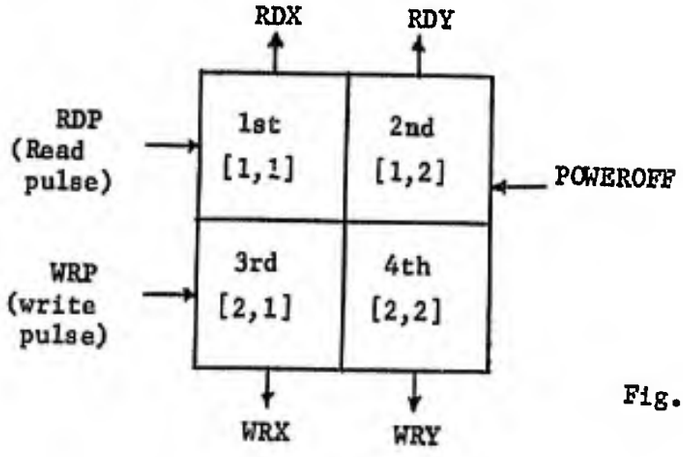

Fig. 8 Schematic of the Pulse Generator, an Array of Four Networks

8. PULSEGEN NWA

[SIZE = (2,2);

ISIZE = 2;

OSIZE = (2,2);

CSIZE = 1;

INPUT NAMING: RDP[1], WRP[2];

OUTPUT NAMING: RDX[1,1], RDY[1,2], WRX[2,1], WRY[2,2];

CONTROL NAMING: POWEROFF;];

:- Memory timing network:

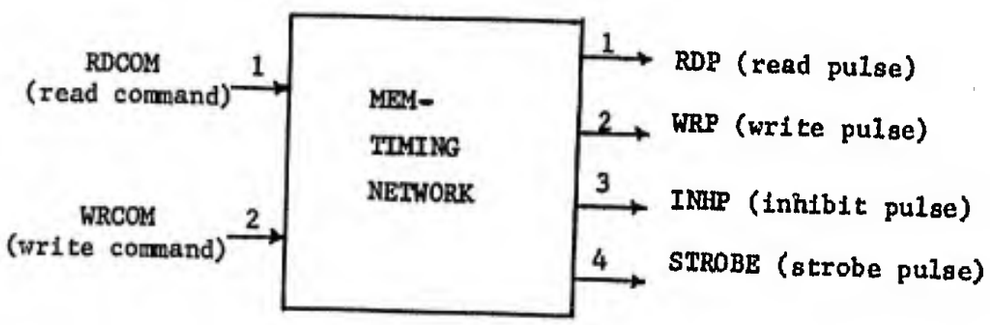

Fig. 9 Schematic of Memory Timing Network

9. MEMTIMING NW

[ISIZE = 2;

OSIZE = 4;

INPUT NAMING: RDCOM[1], WRCOM[2];

OUTPUT NAMING: RDP[1], WRP[2], INHP[3], STROBE[4];];

:- Notice that this is not any array.;

:- Decoder Network:

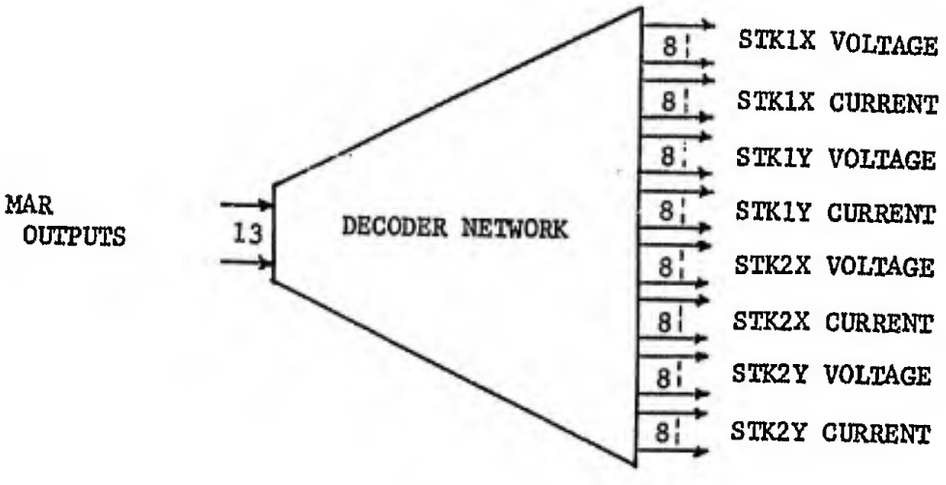

Fig. 10 Schematic of Decoder Network, with 13 Inputs, and 64 Outputs

10. DECODER NW

[ISIZE = 13;

OSIZE = (2,2,2,1:8);

INPUT NAMING: MAROUTPUTS;];

:- I have omitted the output naming scheme from the declaration.;

11. INHDRIVERS N/A

[SIZE = (2,2,9);

ISIZE = 9;

OSIZE = (2,4,9);

CSIZE = 3;

INHDRIVERS NAMING: STK1PATH1[1,1,1:9],

STK1PATH2[1,2,1:9],

STK2PATH1[2,1,1:9],

STK2PATH2[2,2,1:9];

INPUT NAMING: MROUTPUTS;

OUTPUT NAMING: S1P1[1,1:2,1:9], S1P2[1,3:4,1:9],

S2P1[2,1:2,1:9], S2P2[2,3:4,1:9];

CONTROL NAMING: MAR13[1], MAR6[2], INHP[3];];

:= For each stack and each path there are 2x9 output lines: a pair for each one of the nine planes, one for the current output and the other for the return. The schematic is shown below:

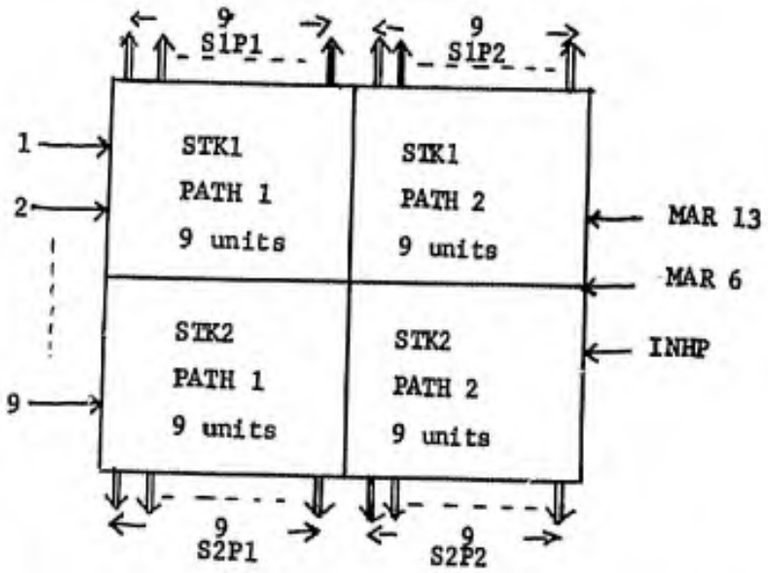

Fig. 11 Schematic of the Inhibit Driver Network Array

12. SENSEAMP NWA

[SIZE = 9;

ISIZE = (2,9,4);

OSIZE = 9;

CSIZE = 2;

CONTROL NAMING: STROBE[1], BIAS[2];

OUTPUT NAMING: MEMOUT;];

:- The schematic diagram is shown below:

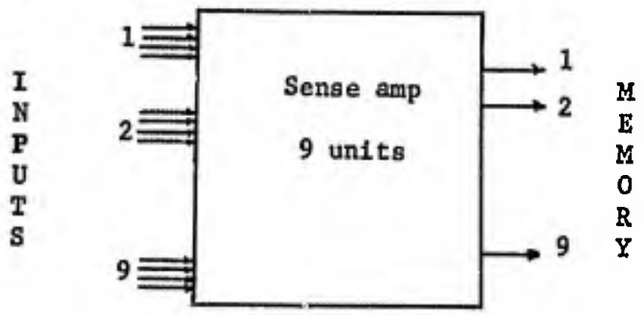

Fig. 12 Schematic of Sense Amplifier Array

13. PCHECKER NW

[ISIZE = 9;

OSIZE = 1;

CSIZE = 1;

CONTROL NAMING: WRCOM;];

END (DECLARATION)

FUNCTIONAL DESCRIPTION:

:- This is to reopen the functional description module which was earlier closed. To describe the operation of the above hardware items we shall need

to define several kinds of pulses. We shall first declare these pulses. With each pulse one may associate the following attributes:

1. AMP amplitude,
2. RTIME: rise time,
3. FTIME: fall time,
4. WIDTH: pulse width,
5. DIRECTION: + or −.

One may declare the first four of these with tolerances. Thus, amplitude could be (100, +5, -5) volts.;

DATA, DECLARATION:

1. RDCOM P [100 NANOS];
2. WRCOM P [100 NANOS];
3. RDP P [500 NANOS];
4. WRP P [500 NANOS];
5. INH P [600 NANOS];
6. STROBE P [200 NANOS];
7. SENSE P [300 NANOS];
8. TIME P [300 NANOS];
9. RDDRIVE P [500 NANOS];
10. WRDRIVE P [500 NANOS];
11. INHDRIVE P [500 NANOS];

:- For none of these pulses were we able to find the amplitude, and rise and fall time data from the available documentation;

12. RDCUR P [400 NANOS; AMP = 320 MA];
13. WRCUR P [400 NANOS; AMP = -320 MA];

:- 'MA' stands for milliamperes. We have omitted writing the attribute 'WIDTH' everytime. Notice that all these pulses have the status of logical pulses within the language. When DIRECTION is not declared it is assumed positive. Notice that direction has nothing to do with the sign of the amplitude of the pulse. The positive direction indicates that all pulses go from logical value 0 to 1.;

END (DATA, DECLARATION)

END (FUNCTIONAL DESCRIPTION)

:- I forgot to declare earlier the hardware item, time pulse generator. Let me declare it now;

DECLARATION:

14. TP NW [ISIZE = 1; OSIZE = 10;

INPUT NAMING: START;]; |||

:- The triple vertical bar denotes, the end of the declaration module;

FUNCTIONAL DESCRIPTION:

3. TP--START ->

WHEN TP--START DO

1. FOR I = 1 TO 10 DO

TP--OUTPUT[I] ← TIME P;

WAIT (200 NANOS);! GO TO 1; ! <-

:- Notice that the pulse, 'TIME' was earlier declared as a 300-nanosecond pulse. In the above statement the first '!' denotes the end of the FOR-loop, and the second, the end of the WHEN-statement. The FOR-loop consists of applying the TIME pulse successively to the 10 output lines of the generator, with 200-nanosecond delay in between. After applying the pulse to the last

line (the 10^{th}) the entire process is repeated, because of 'GO TO 1'. This operation will continue as long as TP--START has logical value 1.

Thus, once the start switch is turned on the time pulse generator will begin producing the 300-nanosecond pulses on its output lines. The pattern of these pulses is shown below:

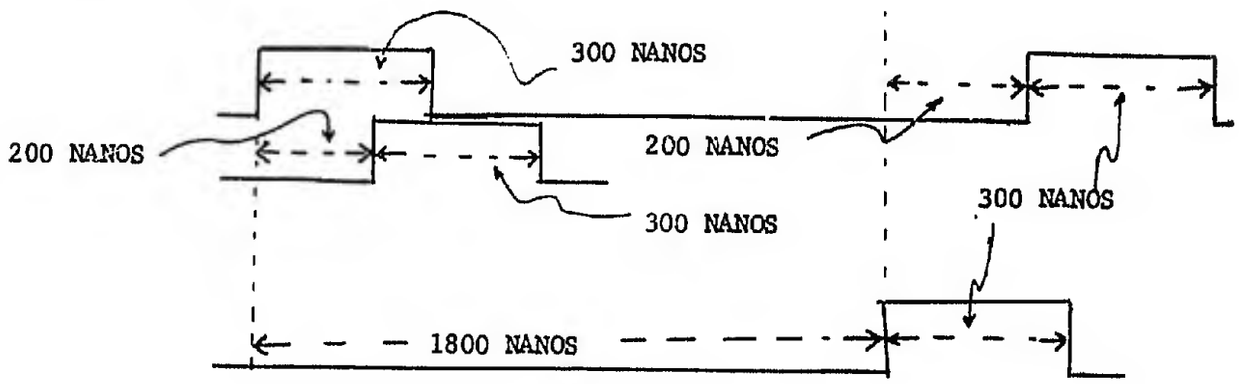

FIG. 13 Time Pulses

```
4.  MEMTIMING--RDCOM  ->
    WHEN MEMTIMING--RDCOM DO
    WAIT (100 NANOS);
    MEMTIMING--RDPOUTPUT ← RDP P;
    WAIT (450 NANOS);
    MEMTIMING--STROBEOUTPUT ← STROBE P; ! <-

5.  MEMTIMING--WRCOM ->
    WHEN MEMTIMING--WRCOM DO
    MEMTIMING--INHPOUTPUT ← INH P;
    WAIT (50 NANOS);
    MEMTIMING--WRPOUTPUT ← WRP P; ! <-
```

```
6.  PULSEGEN--RDP ->

    WHEN PULSEGEN--RDP ∧¬ PULSEGEN--POWEROFF DO

    PULSEGEN--RDXOUTPUT, PULSEGEN--RDYOUTPUT ← RDDRIVE P;!

:- The RDDRIVE pulse is to be applied to both the outputs; <-

7.  PULSEGEN--WRP ->

    WHEN PULSEGEN--WRP ∧ ¬ PULSEGEN--POWEROFF DO

    PULSEGEN--WRXOUTPUT, PULSEGEN--WRYOUTPUT ← WRDRIVE P;! <-

8.  DECODER ->

:- We shall describe the I/O relationship maintained by the decoder. When
a structure containing the decoder is invoked in a procedure, the I/O pro-
cedure described here is to be invoked to maintain the decoder I/O lines
at the proper values, every time its input value changes.;

IF SIZE (MEM) = 4096 ∨ ¬ MAR [13]
THEN
    (FOR I = 1 TO 8 DO PARALLEL

        1.  TRUE :: DECODER-OUTPUT[1,1,1,I] ←
                    (I = N/DECODER--INPUT[1:3] +1);
DECODER--OUTPUT[1,1,2,I] ← (I = N/DECODER--INPUT[4:6] +1);
DECODER--OUTPUT[1,2,1,I] ← (I = N/DECODER--INPUT [7:9] +1);
DECODER--OUTPUT[1,2,2,I] ← (I = N/DECODER--INPUT [10:12] +1);

ELSE FOR I = 1 TO 8 DO PARALLEL

        2.  TRUE :: DECODER--OUTPUT[2,1,1,I] ←
                    (I = N/DECODER-INPUT[1:3] +1);
DECODER--OUTPUT[2,1,2,I] ← (I = N/DECODER--INPUT[4:6] +1);
DECODER--OUTPUT[2,2,1,I] ← (I = N/DECODER--INPUT[7:9] +1);
```

```
DECODER--OUTPUT[2,2,2,I] ← (I = N/DECODER--INPUT[10:12] +1); ::;!)
/// DELAY (DECODER-DELAY); <←
```

:- The IF-THEN-ELSE statement above is a complex parallel statement. In each FOR-loop all the statements obtained for I = 1,2,...,8, are to be executed in parallel. For every value of I one gets four statements, which are contained within the parallel autonomous statement (see Section 4.5 B2-2) beginning with 'TRUE ::'. This autonomous statement is to be executed unconditionally; all the 4 statements contained within are to be executed in parallel. Thus, each parallel FOR-loop specifies 32 parallel statements. This entire set of 32 statements is to be done in a time period equal to the DECODERDELAY. Since, we could not get the data on this delay we have left it undefined.;

```
9.  INHDRIVERS ->
      WHEN INHDRIVERS--INHP DO
      IF ¬( INHDRIVERS-MAR13) ∨ (SIZE(MEM) = 4096)
    THEN (IF INHDRIVERS--MAR6
          THEN (FOR I = 1 TO 9 DO PARALLEL
                  INHDRIVERS--OUTPUT[1,1,I] ←
                  MR[I] ∧ INHDRIVE P;!
          ELSE FOR I = 1 TO 9 DO PARALLEL
                  INHDRIVERS--OUTPUT[1,3,I] ← MR[I] ∧ INHDRIVE P;!);
```

:- This puts the INHDRIVE pulse in path 1 or path 2 depending upon the value of INHDRIVERS--MAR6 control lines. All the 9 planes in a stack are excited simultaneously.;

```
ELSE IF INHDRIVERS--MAR6
    THEN FOR I = 1 TO 9 DO PARALLEL INHDRIVERS-OUTPUT[2,1,I ←
        MR[I] ∧ INHDRIVE P;!
    ELSE FOR I = 1 TO 9 DO PARALLEL
        INHDRIVERS--OUTPUT[2,3,I] ←
            MR[I] ∧ INHDRIVE P;!););! ←
```

:- The second stack inhibit drives are excited only if the size of MEM is 8192 and MAR[13] = 1. Notice that the entire WHEN-statement will be active only for the duration of the pulse on the INHDRIVERS--INHP, control line.;

```
10. CURSWITCHES ->
    FOR
      I = 1 TO 2,
      J = 1 TO 2,
      K = 1 TO 2,
      L = 1 TO 8, DO PARALLEL
    WHEN
      CURSWITCHES--DECODEROUTPUT[I,J,K,L] DO
      CURSWITCHES--CURINPUT[I,J,K,L] ←*
      CURSWITCHES--OUTPUT[I,J,K,L];!! ←
```

:- Notice that this establishes a connection between selected I/O lines of the current switch on the satisfaction of the control condition;

```
11. VOLSWITCHES ->
    FOR I = 1 TO 2,
        J = 1 TO 2,
        K = 1 TO 8, DO PARALLEL
```

```
        WHEN
            VOLSWITCHES--DECODEROUTPUTS[I,J,K] DO
            VOLSWITCHES--TOCNTRTAP[I,J,K] ← 1;!! <-
12.     XFORMERS ->
        FOR I = 1 TO 2,
            J = 1 TO 2,
            K = 1 TO 8,
            L = 1 TO 8, DO PARALLEL
        WHEN
            XFORMERS--RDINPUT[I,J,K,*] ∧ XFORMERS--CNTRTAP[I,J,*,L] DO
            XFORMERS--CUROUT[I,J,K,L] ← RDCUR P;!
        WHEN
            XFORMERS--WRINPUT[I,J,K,*] ∧
            XFORMERS--CNTRTAP[I,J,*,L] DO
            XFORMERS--CUROUT[I,J,K,L] ← WRCUR P;!! <-
13.     STACK ->
        FOR I = 1 TO 2,
            J = 1 TO 64
            K = 1 TO 64, DO PARALLEL
        WHEN STACK--CONTROL[I,J,1; I,K,1] DO
        IF (AMP(STACK--XDR[I,J,1])  = 320 MA) ∧
           (AMP(STACK--YDR[I,K,1]) = 320 MA)
        THEN
            (IF ((J,K) =< 32 ∨ (J,K) > 32)
            THEN (STACK--SENSELINES[I,1,*] ←
                  SENSE P ∧ STACK[I,J,K];
```

```
        ELSE STACK--SENSELINES[I,3,*] ←
                SENSE P ∧ STACK[I,J,K];););
    IF (AMP(STACK--XDR[I,J,1]) = -320 MA) ∧
       (AMP(STACK--YDR[I,K,1]) = -320 MA)
    THEN
       (STACK[I,J,K] ← IF J =< 32 THEN
         (¬( STACK--INHLINES[I,1,*])
    ELSE ¬( STACK--INHLINES[I,3,*])););!! <-
```

:- Both reading and writing of STACK has been described above in terms of the amplitudes of the current pulses on the X and Y drive line. The AMP (of a line) is that of the pulse on the line.;

```
14. SENSEAMP ->
    WHEN
        SENSEAMP--STROBE DO
    FOR I = 1 TO 2, K = 1 TO 9 DO PARALLEL
    IF (AMP(SENSEAMP--INPUT[I,K,1]) >= AMP(SENSEAMP--BIAS)
       ∨ (AMP(SENSEAMP--INPUT[I,K,3]) >= AMP(SENSEAMP--BIAS)
    THEN (MR[K] ← 1;
    ELSE MR[K] ← 0;);!! <-

15. PCHECKER ->
    WHEN TP--OUTPUT[5] ∧ PCHECKER--WRCOM DO
        MR[9] ← ¬⊕/MR[1:8];!
    WHEN TP--OUTPUT[7] DO
        ERROR FF ← ¬⊕/MR[1:9];! <-
```

END (FUNCTIONAL DESCRIPTION)

STRUCTURE:

:- In this module we shall describe the detailed interconnection structure of the subsystems of the memory. A schematic of the block diagram is shown below in Fig. 14.

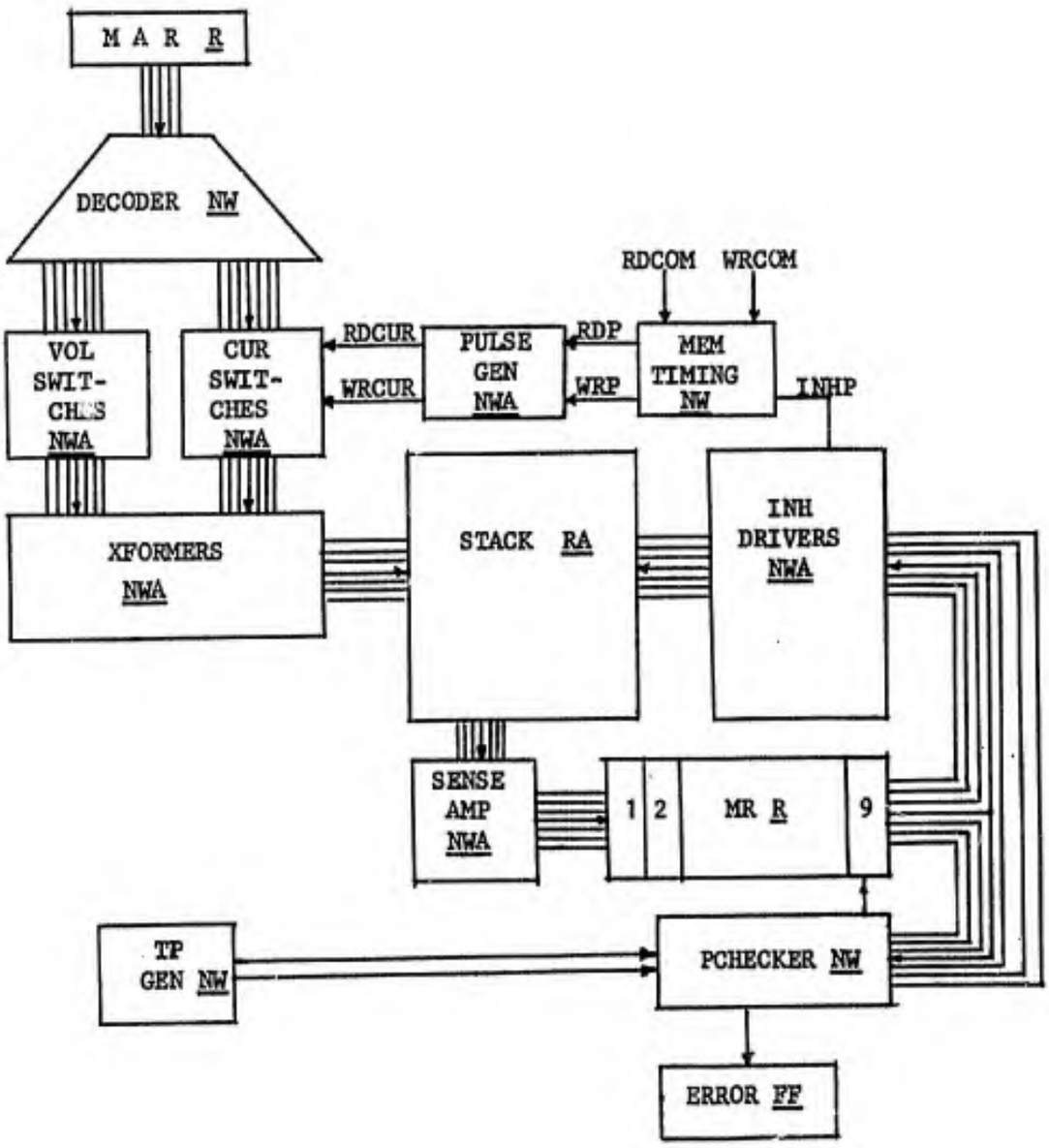

Fig. 14. Block Diagram of Memory System

```
1.  STACK--XDR[I,8*(J-1) +K,L] ← *
        XFORMERS--OUTPUT[I,1,J,K,L] ///
        1 =< (I,L) =< 2, 1 =< (J,K) =< 8;
```

:- Within the bounds of the indices all the corresponding connections declared in statement 1 are admissible. Notice that all statements in structure declaration are declaratives.;

```
2.  STACK-YDR[I,8*(J-1) +K,L] ←*
        XFORMERS--OUTPUT[I,2,J,K,L] /// (///1.);
```

:- (/// 1.) denotes 'the same restrictions as in 1'.;

```
3.  XFORMERS--CNTRTAP[I,J,*,K] ←*
        VOLSWITCHES--OUTPUT[I,J,K] /// 1 =< (I,J) =< 2, 1 =< K =< 8;
4.  XFORMERS--RDINPUT[I,J,K,*] ←*
        CURSWITCHES--TORDINPUT[I,J,1,K] /// 1 =< (I,J) =< 2, 1 =< K =< 8;
5.  XFORMERS--WRINPUT[I,J,K,*] ←*
        CURSWITCHES--TOWRINPUT[I,J,2,K] /// (/// 4.);
6.  VOLSWITCHES--DECODEROUTPUTS[I,J,K] ←*
        DECODER--OUTPUT[I,J,1,K] /// (/// 4.);
7.  CURSWITCHES--DECODEROUTPUTS[I,J,*,K] ←*
        DECODER--OUTPUT[I,J,2,K] (/// 4.);
8.  CURSWITCHES--CURINPUT[*,I,J,*] ←*
        PULSEGEN[I,J] /// 1 =< (I,J) =< 2;
9.  PULSEGEN--INPUT[I] ←* MEMTIMING[I] /// 1 =< I =< 2;
10. STACK--INHLINES[I,J,K] ←*
        INHDRIVERS--OUTPUT[I,,K] ///1 =< I =< 2, 1 =< J =< 4, 1 =< K =< 9;
11. INHDRIVERS--CONTROL[1:3] ←* MAR[13], MAR[6], MEMTIMING--INHP;
```

12. SENSEAMP[I,J,K] ←* STACK--SENSELINES[I,K,J,] ///

1 =< I =< 2, 1 =< J =< 4, 1 =< K =< 9;

13. SENSEAMP--STROBE ←* MEMTIMING--STROBE;

14. PCHECKER ←* MR;

15. MR[9], ERROR ←* PCHECKER;

16. MR ←* SENSEAMP.

END (STRUCTURE)

OPERATIONAL DESCRIPTION:

:- We shall define here two functions on MEM; one will be called LOAD and the other GET. We shall describe the execution of these functions operationally.;

1. LOAD(MEM,NAME R, NAME R) ->

:- NAME R here refers to the fact that two of the arguments are to be registers. The size of these registers will get defined in the procedure below;

TRIGGER (TP--start); :- This sets the TP-start control signal to 1.;

LET

USE STRUCTURE IN HARDWARE DEFINITION, GLOBAL;

'TP--OUTPUT' ≡. 'TP';!

1. TP[1] ∧¬ TP[2] :: MR, MAR ← 0; ::;

2. TP[1] :: MAR ← NAME (*1);

READ FF ← 1; ::;

:- We have here implicitly declared a FF, called READ;

3. TP[2] :: WRITE FF ← 1; ::;

```
4.  WHEN READ DO
    TRIGGER (MEMTIMING--RDCOM,RDCOM P);!
5.          TP[3] :: READ ← 0; ::;
6.          TP[5] :: MR[1:8] ← NAME (*2);
                     MR[9] ← PCHECKER ::;
7.          TP[7] :: TRIGGER (MEMTIMING--WRCOM,WRCOM P); ::;
8.          TP[8] :: WRITE ← 0; ::; <-
```

:- Within the structure of the memory system the trigger commands initiated above will cause a whole series of pulses to be generated and applied to the various signal lines within the system. As these pulses travel along the signal paths specified in the structure, they in turn will initiate other actions. Finally, by the time the 8th time pulse arrives, the memory would have been loaded. Notice that MAR ← NAME (*1), defines the size of NAME(*1), by virtue of the size of MAR. So also, MR ← NAME(*2) defines the size of NAME(*2).;

```
2.  GET (MEM, NAME R) ->
    TRIGGER (TP-START);
    LET
        USE STRUCTURE IN HARDWARE DEFINITION,  GLOBAL;

        'TP--OUTPUT' ≡. 'TP'; !
1.  TP[1] ∧¬ TP[2] :: MAR, MR ← 0;
2.          TP[1] :: MAR ← NAME;
                     READ FF ← 1; ::;
3.          TP[2] :: WRITE FF ← 1; ::;
4.  WHEN READ DO
TRIGGER (MEMTIMING--RDCOM, RDCOM P);!
```

```
5.          TP[3] :: READ ← 0; ::;
6.          TP[6] :: TRIGGER (MEMTIMING--WRCOM, WRCOM P); ::;
7.          TP[8] :: WRITE ← 0; ::;
8.          TP[4] :: MR ← SENSEAMP--OUTPUT; ::; <-
END (OPERATIONAL DESCRIPTION)
END (HARDWARE DEFINITION)
END (70/15 CORE MEMORY SYSTEM, DESCRIPTION)
```

VI. A PARALLEL PROCESSOR

In this section we shall describe a version of the parallel processor scheme proposed by Saul Levy [7]. The description is in the form of a procedure. The description is not that of a parallel processing machine, but is that of a scheme, which such a machine might follow. It is entirely functional, in terms of the string variables, which the procedure handles. In a sense, this description offers a contrast to the one presented in the previous section; the interest is purely in the processes involved, and not in how they are done.

In the example we make use of the 'PATTERN' definition facility which was alluded to in Section IV-B-1. The facility is the following:

While defining string variables, one may define the patterns of the string values which the variable may assume. Consider for example the definition:

'INS ST ≡. OPCODE || OP1 || OP2 || OP3 || BOUNDS;'.

This defines the string INS (instruction) as consisting of 5 parts. The successive parts are shown separated by the concatenation operator. The patterns of OPCODE, OP1, OP2, OP3 and BOUNDS were themselves, say defined a priori. The above statement in a PATTERN module indicates to the system the following properties of INS:

1. The string INS should always have the said pattern.

2. The pattern consists of five subpatterns concatenated as shown. each subpattern having been named as shown.

3. OPCODE(INS), OP1(INS),...,BOUND(INS) are to be given the status of pattern functions: In a procedure one may now ask for OPCODE(INS) to

obtain the opcode of the string called, INS. Thus, pattern definition gives a naming scheme for substrings of a string. The concatenation operator separates the individual pattern names, for which pattern functions are to be formed, for a given string variable. The pattern functions will exist for a string only if the concatenation operator appears in the pattern definition of the string. Thus, BOUNDS V defined by,

'BOUNDS V ≡. (INTEGER, INTEGER);'

will not have any pattern functions associated with it.

The name of a pattern (like OPCODE, OP1, etc.) cannot be used also as the name of a variable in a procedure. All pattern names appearing on the right side of the '≡.' symbol in a pattern definition statement, should have been defined a priori. The symbols,

'string' (a string under quotes),

b, operator, (,), [,], ~, ≡., =., ::=, /, ↑, →, ←, ,, ., :,

;, and !

are treated as pattern constants. In

'OP1 ≡. NAME || [INDEX];'

OP1 will acquire two pattern functions:

NAME (OP1) and INDEX (OP1). Notice that the second pattern function is 'INDEX' and not '[INDEX]'; the pattern constants appearing in a pattern are to be stripped off, while naming a pattern.

New patterns may be created through the use of the nondeterministic replacement rule. Thus,

'OPCODE' ::= {'ADD', 'SUB', 'DIV', 'MUL'};

defines the pattern of OPCODE as consisting of anyone of the four strings shown on the right. Only after defining OPCODE thus, one may use it as a pattern name on the right side of a '≡.' statement.

A pattern module may not contain any other kinds of statements.

We may now proceed to the parallel processor description. We leave it to the reader to understand the procedure from the description below:

PARALLEL PROCESSOR, PROCEDURE:

DATA-DEFINITION:

PATTERN:

1. 'IX ST' ::= {'I', INTEGER, (IX), IX⟨{+,−,*}⟩ IX};

:- This statement defines the pattern of a string called, IX(index). IX can have as its value 'I', or an integer (INTEGER is a pattern already defined in the language), or IX under parenthesis, (IX), or IX + IX, IX - IX or IX * IX. The definition involves recursion in the usual Backus Normal form sense. The use of value brackets in ⟨{+,-,*}⟩, denotes the value of the selection expression, {+,-,*} as part of the pattern, rather then the expression itself.;

2. BOUNDS V ≡. (INTEGER, INTEGER) ///

1 =< INTEGER (*1) =< INTEGER (*2);

:- 'BOUNDS' is defined as a vector with two elements, which are both integers satisfying the indicated restriction. In a procedure one may now ask for BOUNDS[1] and BOUNDS[2] to call for the first or second element of the variable.;

3. OP1 ≡. OP2 ≡. OP3 ≡. NAME "‖ [IX]";

:- This is a short form for three separate statements: OP1, OP2 and OP3 all have the same pattern with possibly two pattern functions. One is 'NAME' and the other IX. The pattern '[IX]' is optional, as indicated by the double quotes. Thus, the value of the pattern function IX(OP2) could be sometimes vacuous. Since IX has now been used as the name of a pattern function, it cannot be used as the name of a variable in the procedure.;

4. 'OPCODE' ::= {'ADD', 'SUB', 'MUL', 'DIV'};

5. INS ST ≡. REG ST ≡. OPCODE ‖ OP1 ‖ OP2 ‖ OP3 ‖ BOUNDS;

:- Now, 'BOUNDS' has become a pattern function. So, BOUNDS[1] or BOUNDS[2] will refer to the latest values of the pattern function, 'BOUNDS',if such values exist, and BOUNDS(INS)[1] and BOUNDS(INS)[2] have their corresponding connotations.;

6. AREG[I] ≡. MREG[I] ≡. DREG[I] ≡. OPCODE ‖ OP1 ‖ OP2 ‖ OP3

/// I >= 1;

:- The indexed variables AREG, MREG and DREG have the indicated pattern. The index should be always >= 1. These are the local registers for the various adders, multipliers and dividers in the procedure.;

7. ADDQ STV ≡. MULQ STV ≡. DIVQ STV ≡.

⟨OPCODE ‖ OP1 ‖ OP2 ‖ OP3 ! ... OPCODE ‖ OP1 ‖ OP2 ‖ OP3⟩;

:- The pattern of the above string vectors has been defined as the value of the given string expression. The functions

OPCODE (ADDQ[I]), OP1(MULQ[J]), etc.

are now meaningful in the procedure for 1 =< (I,J) =< DIMN (the STV).;

8. ACAM STV ≡. MCAM STV ≡. DCAM STV ≡.

⟨NAME"[INTEGER]"! ./I.I/.

NAME"[INTEGER]"⟩ /// I >= 1;

:- There should be at least one NAME"[INTEGER]" in the above string vectors; they cannot be vacuous. CAM stands for Content Addressed Memory. ACAM is the content addressed memory associated with the adders. Similarly, the prefixes M and D denote multipliers and dividers, respectively. The various adder, multiplier and divider units store their results in the ACAM, MCAM and DCAM respectively.;

9. STACK STV ≡. ⟨INS!...INS!⟩; END(PATTERN).

:- In the above pattern definition all the variable names (those, which have not been used as pattern function names) are to acquire the status of declared variables within the ensuing procedure. In the following module, 'INPUT DATA', the necessary external inputs to the procedure are listed. The procedure can be executed only if all the inputs listed in this module have been specified.;

INPUT DATA:

1. STACK STV [DIMN VARIABLE];

:- The value of the string vector STACK is to be declared. Notice that its value is restricted to be a series of instructions of the form of INS (pattern definition 9). The dimension of the STV is a variable; it may change within the procedure.;

2. ACAM STV [DIMN VARIABLE];

3. MCAM STV [DIMN VARIABLE];

4. DCAM STV [DIMN VARIABLE];

5. DIMN (AR V);

:- AR is a vector declared within the procedure. The dimension of this vector is the number of adder units to be used in the parallel processor. The elements of AR are bits, AR[I] = 1 if the I^{th} adder unit is ready to accept its next job, otherwise AR[I] = 0. The vectors MR and DR are used, similarly, with the multiplier and divider units, respectively.;

6. DIMN (MR V),

7. DIMN (DR V);

8. AT S;

:- This is the add time.;

9. MT S;

10. DT S;

11. ST S;

:- subtract time;

END (INPUT DATA)

PROCEDURE:

1. LET

    1. ADDQ[DIMN VARIABLE];

    2. MULQ[DIMN VARIABLE];

    3. DIVQ[DIMN VARIABLE];

    4. EVENT:- We shall now declare a series of event variables;

        S1 =. 1, :- The start event. Initial value 1.;

        S2 =. 0; :- Fetch instruction from STACK;

(A1,M1,D1) =. 0, :- Check ACAM, MCAM and DCAM, respectively, to see whether the operands in the 'present' instruction are available in the CAM's.;

(F1,F2,F3) =. 0, :- The above checks have been completed;

(AQ,MQ,DQ) =. 0, :- The 'present' instruction is to be put in the add, mul., or div. queue, respectively.;

F =. 0, :- Instruction has been attached to the appropriate queue.;

AC V [DIMN = DIMN(AR) +1] =. (1, ⟨0,...0⟩),

:- AC[i] for i > 1 indicates that ACAM is available for the $(i-1)^{st}$ adder unit, to store its results. AC[1] indicates that ACAM is available for questioning under event A1, above. The following events MC and DC have similar significance for MCAM and DCAM, respectively.;

```
MC V [DIMN = DIMN(MR) +1] =. (1, ⟨0,...0⟩),
DC V [DIMN = DIMN(DR) +1] =. (1, ⟨0,...0⟩)
AR V =. 1, MR V =. 1, DR V =. 1.
(AQR, MQR, DQR) =. 1,
:- These indicate that the ADDQ, MULQ and DIVQ, respectively, are ready for
questioning.;
FA V [DIMN(AR)] =. 0,
:- FA[i] = 1 if the i^th adder unit is ready to store its results in the
ACAM, otherwise, FA[i] = 0. Similarly, FM and FD are used with the
multiplier and divider units.;
FM V [DIMN(MR)] =. 0,
FD V [DIMN(DR)] =. 0,;!
:- The ';' terminates the event declaration, and '!' the LET-statement.;
WHEN S1 ∧ STACK ¬≡ NULL DO
     S1 ← 0; REG ← STACK[1];
     STACK ← SHIFTN(-1; STACK); S2 ← 1; !
:- On the satisfaction of the condition S1 ∧ STACK ¬≡ NULL the tasks are
executed in the given order within the ON statement. '!' terminates
the ON-statement. SHIFTN is a non-cyclic shift function. Since STACK
dimension was declared as a variable, and no 'fill-ins' has been given to
put into the shifted positions of STACK, the stack dimension will be reduced
after the shift. Thus, STACK acts as a 'pop-up' stack.;
WHEN S2 DO
     S2 ← 0; IF BOUNDS(REG)[1] = BOUNDS(REG)[2] THEN
                (BRANCH TO(S1); ELSE I ← BOUNDS(REG)[1];);
```

```
OPE 1 ← '⟨NAME(OP1(REG))⟩"[⟨⟨IX(OP1(REG))⟩⟩]"';
OPE 2 ← '⟨NAME(OP2(REG))⟩"[⟨⟨IX(OP2(REG))⟩⟩]"';
OPE 3 ← '⟨NAME(OP3(REG))⟩"[⟨⟨IX(OP3(REG))⟩⟩]"';
A1, M1, D1 ← 1;!
```

:- Within a statement '⟨⟩' are always treated as meta value brackets. The values of items enclosed are to be substituted before executing the statement. Thus, in the third statement the value of NAME(OP1(REG)) is to be substituted first. NAME(OP1(REG)) calls for the pattern NAME in the pattern OP1 of REG. Similarly ⟨IX(OP1(REG))⟩ calls for the pattern IX in OP1 of REG, and ⟨⟨IX(OP1(REG))⟩⟩ calls for the value of ⟨IX(OP1(REG))⟩, which is obtained by evaluating the arithmetic expression. (Notice that IX could be an arithmetic expression.);

```
WHEN A1 ∧ AC[1] DO
    A1, AC[1] ← 0;
    T1,T2 ← (OPE1 ≡ {ACAM}), (OPE 2 ≡ {ACAM});
    AC2 ← 1; F1 ← 1;!
WHEN ¬ A1 ∧ AC[1] DO
    AC[1] ← 0; AC[2] ← 1;!
```

:- AC[1] indicates that ACAM is available for questioning by the program analyzer to find out whether OPE1 and OPE2 are in ACAM. OPE 1 ≡ {ACAM} stands for (OPE1 ≡ any one of the elements in ACAM). The results of tests are stored in T1 and T2.;

```
WHEN M1 ∧ MC[1] DO
    M1, MC[1] ← 0;
    T3, T4 ← OPE1 ≡ {MCAM}, OPE2 ≡ {MCAM};
    MC[2] ← 1; F2 ← 1;!
```

```
WHEN ¬M1 ∧ MC[1] DO
     MC[1] ← ∅; MC[2] ← 1;!
WHEN D1 ∧ DC[1] DO
     D1, DC[1] ← 0;
     T5, T6 ← OPE1 ≡ {DCAM}, OPE2 ≡ {DCAM};
     DC[2] ← 1; F3 ← 1;!
WHEN ¬D1 ∧ DC[1] DO
     DC[1] ← 0; DC[2] ← 1;!
WHEN F1 ∧ F2 ∧ F3 DO
     F1, F2, F3 ← 0;
     IF (T1 ∨ T3 ∨ T5) ∧ (T2 ∨ T4 ∨ T6) THEN
     ((T1,T2,T3,T4,T5,T6) ← 0;
     IF {OPCODE(REG) ≡ 'ADD', OPCODE(REG) ≡ 'SUB'.
     OPCODE(REG) ≡ 'MUL', OPCODE(REG) ≡ 'DIV'} THEN
     ({AQ,AQ,MQ,DQ} ← 1;); ELSE
     STACK ← STACK!REG; BRANCH TO(S1););!
```

:- (T1 ∨ T3 ∨ T5) ∧ (T2 ∨ T4 ∨ T6) checks for operand availability. If the operands are available then the instruction in REG is ready to be put in the appropriate queues, and AQ, MQ,DQ are accordingly set to 1. The complex IF statement with {.,.,.,} in the IF condition is an abbreviated form of 4 different IF statements. If OPCODE(REG) ≡ 'ADD' then only AQ will be set to 1, and similarly for the other conditions within {,,,}. Notice that the 'ELSE' is to be associated with the first IF and not the second one.;

```
WHEN AQ ∧ AQR DO
     AQ, AQR ← 0;
```

```
        ADDQ ← ADDQ!OPCODE(REG) || OPE1 || OPE2 || OPE3;
        AQR ← 1; F ← 1;!
    WHEN MQ ∧ MAR DO
        MQ, MQR ← 0;
        MULQ ← MULQ! OPCODE(REG) || OPE1 || OPE2 || OPE3;
        MQR ← 1; F ← 1;'.
    WHEN DQ ∧ DQR DO
        DQ, DQR ← 0;
        DIVQ ← DIVQ!OPCODE(REG) || OPE1 || OPE2 || OPE3;
        DQR ← 1; F ← 1;!
    WHEN (AQ ∧¬ AQR) ∨ (MQ ∧ ¬ MQR) ∨ (DQ ∧¬ DQR) DO
        WAIT;!
    WHEN F DO
        F ← 0; BOUNDS(REG)[1] ← BOUNDS(REG)[1] +1;
        BRANCH TO (S2);!
    WHEN ADDQ ¬≡ NULL DO
    WHEN AR[1] ∧ AQR ∧¬ AQ DO
        AQR ← 0; AREG[1] ← AQDQ[1];
        ADDQ ← SHIFTN(-1; ADDQ); AQR ← 1; AR[1] ← 0;
        IF OPCODE(AREG[1]) ≡ 'ADD' THEN
        (WAIT(AT); ELSE WAIT(ST););
        BRANCH TO (FA[1]);!
        FOR I = 2 TO DIMN(AR) DO PARALLEL :- PARALLEL indicates that in the
range of index I all the iterations obtained for the values of I are to be
executed in parallel.;
```

```
      WHEN ¬( ∧/AR[1:I-1]) ∧ AR[I] ∧ AQR ∧¬AQ DO
      AQR ← 0; AREG[I] ← ADDQ[1];
      ADDQ ← SHIFTN (-1; ADDQ); AQR ← 1;
      AR[I] ← 0; IF OPCODE(AREG[I]) ≡ 'ADD'
      THEN WAIT (AT) ELSE WAIT (ST););
      BRANCH TO (FA[I]);!!
FOR I = 1 TO DIMN(AR) DO PARALLEL
WHEN FA[I] ∧ AC[I+1] DO
      FA[I] ← 0;
      ACAM ← ACAM!OP3(AREG[I]);
      AC[I+1] ← 0; AR[I] ← 1;
      IF I+1 = DIMN(AR)+1 THEN
      (AC[1] ← 1; ELSE AC[I+2] ← 1;);!
WHEN FA[I] ∧¬ AC[I+1] DO WAIT;!
WHEN ¬ FA[I] ∧ AC[I+1] DO AC[I+1] ← 0;
      IF I+1 = DIMN(AR) +1 THEN
      (AC[1] ← 1; ELSE AC[I+2] ← 1;);!!!
WHEN MULQ ¬≡ NULL DO
WHEN MR[1] ∧ MQR ∧¬ MQ DO
      MQ ← 0; MREG[1] ← MULQ[1];
      MULQ ← SHIFTN (-1; MULQ); MQR ← 1; MR[1] ← 0;
      WAIT(MT); BRANCH TO (FM[1]);!
FOR I = 2 TO DIMN(MR) DO PARALLEL
WHEN ¬(∧/MR[1:I-1]) ∧ MR[I] ∧ MQR ∧¬MQ DO
      MQR ← 0; MREG[I] ← MULQ[1];
      MULQ ← SHIFTN(-1; MULQ); MQR ← 1; MR[I] ← 0;
      WAIT(MT); BRANCH TO (FM[I]);!!
```

```
FOR I = 1 TO DIMN(MR) DO PARALLEL
WHEN FM[I] ∧ MC[I+1] DO
     FM[I] ← 0; MCAM ← MCAM!OP3(MREG[I]);
     MC[I+1] ← 0; MR[I] ← 1;
     IF I+1 = DIMN(MR)+1 THEN
     (MC[1] ← 1; ELSE MC[I+2] ← 1;);!
WHEN ¬ FM[I] ∧ MC[I+1] DO
     MC[I+1] ← 0;
     IF I+1 = DIMN(MR)+1 THEN
     (MC[1] ← 1; ELSE MC[I+2] ← 1;);!
WHEN FM[I] ∧¬ MC[I+1] DO WAIT;!!!
WHEN DIVQ ¬≡ NULL DO
WHEN DR[1] ∧ DQR ∧¬ DQ DO
     DQR ← 0; DREG[1] ← DIVQ[1];
     DIVQ ← SHIFTN (-1; DIVQ); DQR ← 1; DR[1] ← 0;
     WAIT(DT); BRANCH TO (FD[1]);!
FOR I = 2 TO DIMN(DR) DO PARALLEL
WHEN ¬ (∧/DR[1:I-1]) ∧ DR[I] ∧ DQR ∧¬ DQ DO
     DQR ← 0; DREG[I] ← DIVQ[I];
     DIVQ ← SHIFTN (-1; DIVQ); DQR ← 1; DR[I] ← 0;
     WAIT(DT); BRANCH TO (FD[I]);!!
FOR I = 1 TO DIMN(DR) DO PARALLEL
WHEN FD[I] ∧ DC[I+1] DO
     FD[I] ← 0; DCAM ← DCAM!OP3(DREG[I]);
     DC[I+1] ← 0;  DR[I] ← 1;
     IF I+1 = DIMN(DR) +1 THEN
     (DC[1] ← 1; ELSE DC[I+2] ← 1;);!
```

```
WHEN ¬ FD[I] ∧ DC[I+1] DO
     DC[I+1] ← 0; IF I+1 = DIMN(DR) +1
THEN (DC[1] ← 1; ELSE DC[I+2] ← 1;);!
WHEN FD[I] ∧ ¬ DC[I+1] DO WAIT;!!!  |||

END(PARALLEL PROCESSOR 1 PROCEDURE)
```

VII. CONCLUDING REMARKS

Our aim was to develop a formal language with adequate expressive power and logical structure to describe and document various aspects of a computing system both during its design phase and after the design has been completed. We have in this report discussed in fair amount of detail the considerations that went into the design of such a language, and the language features that the considerations gave rise to. Also, we have pointed out the need for a new language, both from the point of view of the expressive power necessary and the range of applications envisioned.

The examples presented in Sections V and VI point out the variety of objects that one may describe in the language, and the range of kinds of descriptions possible. The example in Section IV explains the four levels of design, and shows how the resulting objects in various levels of design may be precisely described.

Even though the language has a complex structure, a novice can easily write descriptions in it using only its more elementary facilities: the standard operands, and simple assignment and program control statements. A more sophisticated user will find more sophisticated and powerful facilities in the language to meet the needs of his applications: data structure, table and other definitions, pattern declarations, dynamic memory allocation facilities, autonomous statements, hardware allocation facilities, facilities to specify error control strategies etc. To use these powerful facilities appropriately the user should be well versed in the language, and in the implications the use of these various features may have on the task at hand. Generally, it is true that the language forces its users to have a thorough understanding of the objects they may wish to

describe, a desirable feature by itself. Also, it imposes on the users a disciplined way though at an elementary level - of organizing the descriptive data. The process is quite similar to the one of writing a program in a programming language. In the context of a design aid system the benefits for going through such a discipline, and going through a period of training are quite numerous.

One obtains a systematic, precise and complete (in the sense of being simulatable) documentation of a system under design at various stages of its design, from product specification, to its final hardware structure, operation and software assists. During design one may easily keep track of design alternatives, design changes and changes in product specification. One may even create automatic processors to trace the consequences of every given change throughout an entire system. In its present form the language may already be used for design documentation on paper.

Since descriptive data will be presented in a codified form, and filed according to a well-defined filing structure it will be possible to design a great variety of automatic design aids, based on a common data base: One may design an automatic retrieval system to answer queries, designers may ask. CDL1 provides a logical structure to consider the problems arising in such a retrieval system. Such retrieval may be direct retrieval of portions of descriptive data in file, or it may call for an analysis of a body of data to obtain the necessary answers. For example, a designer may want to know the commands which use a particular register or network in a system; or the number of times a given bus is used in an instruction; or the nature of traffic pattern across a given bus, etc. A large number of such simple analysis tasks may be formulated and solved in the context of CDL1.

Since descriptions are supposed to be complete, at each level of design one may call for a simulator, to simulate the system at that level of design. Design alternatives may be evaluated through simulation, or a design so verified. An entire simulation system may be created to work directly from the descriptive data base. One need not write special purpose simulators to test each stage of a design process, either hardware or software. The formal system in CDL1, we believe, is adequate to create such simulators.

While describing a system in CDL1, one can set up system error control criteria and develop error control strategies on the basis of such criteria. Also, we believe, it would be possible to develop techniques for generating automatic system diagnostic aids. In fact, this has been our primary motivation for developing the language itself.

Finally, one may attempt to automate synthesis tasks in the context of CDL1. The synthesis may be viewed as a translation of a description at one level to the one at its next more detailed level, satisfying the design contraints imposed by the auxiliary declarations of objects and processes made on the side. Though the feasibility of design through such translation has been studied [3,5,8], the problems involved in the process have not been well understood.

We hope to study these processes, as also those of system error control, diagnostic aids and simulation, during the course of our future work. In studies on system organization CDL1 will prove to be useful to describe new systems, and the associated design aid systems may be used to simulate and evaluate them. With our present work we have thus laid the foundation, not only for the development of numerous design aids of

immediate practical interest, but also for the creation of a powerful tool for research on system organization, error control, and analysis and synthesis techniques.

Our immediate next goal is the creation of a documentation facility: an automatic filing system. We anticipate this system to check its inputs for proper form and syntax and to make certain necessary abstracts of input data. The system is also to have an elementary data retrieval capability, to retrieve data called by name. Recently, the problems involved in the creation of such a system are being studied from the point of view of file flexibility, the features of CDL1 to be implemented and the techniques of generating syntax checkers automatically from the formal description of the language. This study is being conducted jointly with the Product Planning Group* at Cherry Hill, N. J.

CDL1 has been described formally in CDL1 itself. The language description is itself in modules, which decompose the language into parts. Care has been taken in the definition of the language to make it deterministic (parsable in a single left to right scan). We cannot, however, guarantee determinism at this time: we hope it is deterministic. The sheer size of the language calls for innovations in the organization of its syntax and format checker. We believe, the decomposition of the language into parts will greatly aid its implementation.

*Donald Gorman, McAllister, and Mary Dempsy of RCA's EDP Product Planning Group are contributing toward this effort.

VIII. ACKNOWLEDGMENTS

The work reported here was started by this author during the first quarter of 1966, when a formal description of 70/15 was written. Our ideas on the description language grew directly from this experience.

Donald Gorman and Justin Kodner joined me in this effort in October 1966. The work is the result of joint effort by all three. This author assumes responsibility for the general organization of the language: Its module structure, module classification schema, module format and syntax; the operand classification and indexing structure (the multi level nested indexing was suggested by Donald Gorman); the various expressions syntax; classification of statement types, their form and syntax (the use of IF-expression was suggested by Donald Gorman); and concepts associated with the various kinds of system descriptions, and various levels of design.

I wish to thank Gorman and Kodner for the many discussions we had and Gorman, especially, for proofreading the several versions of the syntax, that were written for the language.

IX. REFERENCES

1. Balzer, R., "Dataless Programming," unpublished interim report.

2. Burnett, G. J., "A Design Language for Digital Systems," Master's thesis, M.I.T.; August 1965.

3. Chu, Y., "An ALGOL-like Computer Design Language," Comm. of the ACM, Vol. 8, No. 10, pp. 607-614; Oct. 1965.

4. Falkoff, A. D., et al., "A Formal Description of SYSTEM/360," IBM Systems Journal, Vol. 3, No. 3, pp. 198-262, 1964.

5. Gorman, D. F. and Anderson, J. P., "A Logic Design Translator," 1962 Proc. of Fall Joint Computer Conf., pp. 251-261.

6. Iverson, K. E., A Programming Language, John Wiley and Sons, Inc., New York, 1962.

7. Levy, S., "Automatic Sequencing in a Parallel Processor," private communication.

8. Metze, G. and Seshu, S., "A Proposal for a Computer Compiler," 1966 Proc. of Spring Joint Computer Conf., pp. 253-263.

9. Naur, P., et al., "Revised Report on the Algorithmic Language ALGOL 60," Comm. of the ACM, Vol. 6, No. 1, pp. 1-17; Jan. 1963.

10. Proctor, R. M., "A Logic Design Translator Experiment Demonstrating Relationships of Language to Systems and Logic Design," IEEE Trans. Elect. Computers, Vol. EC-13, pp. 422-430, Aug. 1964.

11. Reed, I. S., "Symbolic Design Techniques Applied to a Generalized Computer," M.I.T. Lincoln Labs., Lexington, Mass., TR. No. 141; Jan. 3, 1957.

12. Schlaeppi, H. P., "A Formal Language for Describing Machine Logic, Timing and Sequencing (LOTIS)," IEEE Trans. Elect. Computers, Aug. 1964, pp. 439-448.

13. Schorr, H., "A Register Transfer Language to Describe Digital Systems," Tech. Rep. No. 30, Dept. of Elec. Eng. Digital Systems Lab., Princeton University, Sept. 1962.

14. Standish, T. A., "A Data Definition Facility for Programming Language," PhD. thesis, Carnegie Inst. of Tech., 1967.

15. PL/1: Language Specifications, IBM Operating System/360. IBM Systems Reference Library File No. S360-29, Form C28-6571-2, Jan. 1966.

UNCLASSIFIED
Security Classification

DOCUMENT CONTROL DATA - R&D
(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1 ORIGINATING ACTIVITY (Corporate author)
Radio Corporation of America
RCA Laboratories
Princeton, New Jersey 08540

2a. REPORT SECURITY CLASSIFICATION
Unclassified

2b GROUP
N/A

3 REPORT TITLE

AN INTRODUCTION TO CDL1, A COMPUTER DESCRIPTION LANGUAGE

4 DESCRIPTIVE NOTES (Type of report and inclusive dates)
Scientific. Interim.

5 AUTHOR(S) (Last name, first name, initial)

Srinivasan, Chitoor V.

6 REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
September 1967	124	15

8a. CONTRACT OR GRANT NO.	9a. ORIGINATOR'S REPORT NUMBER(S)
AF19(628)4789 b. Project, Task, Work Unit Nos. 5632-02-01 c DoD Element 61445014 d. DoD Subelement 681305	Scientific Report No. 1 9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report) AFCRL-67-0565

10. AVAILABILITY/LIMITATION NOTICES
Qualified requestors may obtain additional copies from the Defense Documentation Center. All others should apply to the Clearinghouse for Federal Scientific and Technical Information.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
TECH, OTHER	Air Force Cambridge Research Laboratories (CRB) L. G. Hanscom Field Bedford, Massachusetts 01730

13. ABSTRACT

The objective of this report is to develop a formal language to describe hardware and software computing systems. The language is to provide a linguistic basis to consider machine-aided solutions of a variety of design problems; i.e., problems concerning design documentation, data retrieval systems, system simulation, diagnosis, analysis and synthesis. This report discusses in some detail the considerations that went into the design of the computer description language, called CDL1; it points out the need for developing such a language and briefly discusses the kinds of applications such a language may have. The report points out the various kinds of system descriptions one may encounter in a design process and relates them to the language features necessary to express them; the language itself is described informally. Examples are presented to illustrate the use of the language, the concepts associated with descriptions of systems at various stages of design, and the consequent hierarchical structure such descriptions acquire.

DD FORM 1 JAN 64 1473

UNCLASSIFIED
Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Design aid systems Computer description Documentation Simulation Automatic synthesis						

INSTRUCTIONS

1. ORIGINATING ACTIVITY: Enter the name and address of the contractor, subcontractor, grantee, Department of Defense activity or other organization *(corporate author)* issuing the report.

2a. REPORT SECURITY CLASSIFICATION: Enter the overall security classification of the report. Indicate whether "Restricted Data" is included. Marking is to be in accordance with appropriate security regulations.

2b. GROUP: Automatic downgrading is specified in DoD Directive 5200.10 and Armed Forces Industrial Manual. Enter the group number. Also, when applicable, show that optional markings have been used for Group 3 and Group 4 as authorized.

3. REPORT TITLE: Enter the complete report title in all capital letters. Titles in all cases should be unclassified. If a meaningful title cannot be selected without classification, show title classification in all capitals in parenthesis immediately following the title.

4. DESCRIPTIVE NOTES: If appropriate, enter the type of report, e.g., interim, progress, summary, annual, or final. Give the inclusive dates when a specific reporting period is covered.

5. AUTHOR(S): Enter the name(s) of author(s) as shown on or in the report. Enter last name, first name, middle initial. If military, show rank and branch of service. The name of the principal author is an absolute minimum requirement.

6. REPORT DATE: Enter the date of the report as day, month, year; or month, year. If more than one date appears on the report, use date of publication.

7a. TOTAL NUMBER OF PAGES: The total page count should follow normal pagination procedures, i.e., enter the number of pages containing information.

7b. NUMBER OF REFERENCES: Enter the total number of references cited in the report.

8a. CONTRACT OR GRANT NUMBER: If appropriate, enter the applicable number of the contract or grant under which the report was written.

8b, 8c, & 8d. PROJECT NUMBER: Enter the appropriate military department identification, such as project number, subproject number, system numbers, task number, etc.

9a. ORIGINATOR'S REPORT NUMBER(S): Enter the official report number by which the document will be identified and controlled by the originating activity. This number must be unique to this report.

9b. OTHER REPORT NUMBER(S): If the report has been assigned any other report numbers *(either by the originator or by the sponsor)*, also enter this number(s).

10. AVAILABILITY/LIMITATION NOTICES: Enter any limitations on further dissemination of the report, other than those imposed by security classification, using standard statements such as:

(1) "Qualified requesters may obtain copies of this report from DDC."

(2) "Foreign announcement and dissemination of this report by DDC is not authorized."

(3) "U. S. Government agencies may obtain copies of this report directly from DDC. Other qualified DDC users shall request through

_____."

(4) "U. S. military agencies may obtain copies of this report directly from DDC. Other qualified users shall request through

_____."

(5) "All distribution of this report is controlled. Qualified DDC users shall request through

_____."

If the report has been furnished to the Office of Technical Services, Department of Commerce, for sale to the public, indicate this fact and enter the price, if known.

11. SUPPLEMENTARY NOTES: Use for additional explanatory notes.

12. SPONSORING MILITARY ACTIVITY: Enter the name of the departmental project office or laboratory sponsoring *(paying for)* the research and development. Include address.

13. ABSTRACT: Enter an abstract giving a brief and factual summary of the document indicative of the report, even though it may also appear elsewhere in the body of the technical report. If additional space is required, a continuation sheet shall be attached.

It is highly desirable that the abstract of classified reports be unclassified. Each paragraph of the abstract shall end with an indication of the military security classification of the information in the paragraph, represented as *(TS)*, *(S)*, *(C)*, or *(U)*.

There is no limitation on the length of the abstract. However, the suggested length is from 150 to 225 words.

14. KEY WORDS: Key words are technically meaningful terms or short phrases that characterize a report and may be used as index entries for cataloging the report. Key words must be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location, may be used as key words but will be followed by an indication of technical context. The assignment of links, rules, and weights is optional.