CALSIM: A COMPUTER HARDWARE DESCRIPTION LANGUAGE FOR COMPUTER SCIENCE EDUCATION

by

WILLIAM ARGLE SKELTON, JR.

Presented to the Faculty of the Graduate School of
The University of Texas at Arlington in Partial Fulfillment
of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT ARLINGTON

September 1982

CALSIM: A COMPUTER HARDWARE DESCRIPTION LANGUAGE FOR COMPUTER SCIENCE EDUCATION

APPROVED:

[illegible]
(Supervising Professor)

Roger S. Walker

[illegible]

Bill D. Carroll

[illegible]

Bob F. Perkins
(Dean of The Graduate School)

DEDICATED TO

Tillie, Andy, and Sandi

ACKNOWLEDGEMENTS

I wish to express my appreciation to each of the professors who have helped me during this effort, Dr. Walker who suggested the project to me, for the three years he served as my advisor and for his unwavering confidence that I would some day finish the effort. I am deeply indebted to Dr. Underwood for assuming responsibility for supervision of the project in mid-stream. Without his interest and leadership in the final phases of my research and thesis preparation, the work would never have been completed.

The idea of using formal grammar and the LALR system to develop cohesive grammars for the Description Language and the simulator driver, I owe to to Dr. Schember. The course work under Dr. Sparr gave me the understanding needed to organize the data handled in the system. The comments of Drs. Carroll and Sparr during the review of my work have been particularly helpful. My thanks to Drs. Buckles and Elizandro for their interest in the project.

My family gave me great support during this effort, particularly Sandi for typing, proofreading and suggestions. My greatest debt is to my wife, Tillie, who has exhibited uncommon patience, understanding, and unfailing support over these many years.

September 10, 1982

ABSTRACT

CALSIM: A COMPUTER HARDWARE DESCRIPTION LANGUAGE FOR COMPUTER SCIENCE EDUCATION

Publication No._________

W. A. Skelton, Ph.D.

The University of Texas at Arlington, 1982

Supervising Professor: Stephen Underwood

A computer hardware description language (CHDL) and its compiler/simulator system, designed for student use at graduate and undergraduate levels are described. The system is usable above the switching circuit level and incorporates features to investigate designs using microprogrammable components including bit-sliced chips such as the the AMD-2900.

The order of execution in the simulator is controlled by an event table using each time/component as a separate event. The handling of event timing for items copied from

the library of components is unique in that the individual times may be changed as they are copied into the active file. The system also allows the user to make several copies with a single statement and adjust the timing of each copied item separately.

The LALR formal grammar, presented in the Appendix, was develeoped to make the English-like language follow as closely as possible the hierarchic structure of the system being described. This supports a hierarchic design process through the sytem, programming and register transfer levels. Use of the language below the bit level has not been investigated.

The contents of the main memory, micromemory and up to three (3) auxiliary memories are developed separately and read from files into the program memory prior to simulation. The simulation driver incorporates breakpoints, trace, display and other tools needed to follow the simulation which may be carried out in either step-wise or run-to-break fashion. The interactive system is written in Cobol and resides on the DEC-20.

The system has been used one semester for a graduate course in computer organization. The User's Manual (Appendix 6) contains examples from flip-flops to microprogramming and show examples, diagrams, explanations, and coded descriptions of the logical device.

TABLE OF CONTENTS

CHAPTER I

INTRODUCTION

Background

Although microprogramming dates from the early fifties (107), and Computer Hardware Description Languages (CHDL's) date from the early sixties (24), effort to adapt the CHDL's for use with bit-sliced microprogrammable computer architecture design and development has not been entirely successful (92). To a large extent, each area has developed separately serving the technology of the time in it's own way. A Computer Hardware Description Language and software support system is described here which adapts the technology of the CHDL's for use in design studies using bit-sliced components. Since the system is usable as low as the bit level, it can be used for Computer Science education courses from Computer Organization to Microprogramming.

An increase in the use of bit-sliced components for application hardware was predicted for the eighties (75). The families of bipolar, bit-sliced components offers the developer more design freedom and a better organized approach than conventional computer architectures (14).

These are available from sources such as the Advanced Micro Devices AMD-2900 or Texas Instrument SN-74S481 (2).

However, as Alexandrididis (5) states, "The advantages of bit-sliced microprocessors are not free -- they require a serious investment in system support software". Currently, application designs using bit-sliced hardware are developed using custom software, simulators, and prototypes supported by extensive hardware monitoring tools (29). This high initial cost of support hardware and the prospect of preparing custom software causes many designers to avoid approaches using bit-sliced technology (5) and precludes extensive use in Computer Science Education where funds may be limited.

It is evident that the CHDL's have not met the need for development software since more expensive methods are still widely used (1), both in industry and Computer Science education. Several reports, however, have been published on the use of CHDL's in Computer Science education (19), (23), (40), (99) and others have addressed the use of CHDL'S for applications with bit-sliced hardware (43), (50), (72).

Premises on Which the System is Designed

The extensions and differences of Calsim (Computer Architecture Language for Simulation) and its software system from existing CHDL's were developed based on three premises. First, a change will occur in the eighties in the fundamental way logical devices are designed due to the increased size of the chips available. Bit-sliced chips are predicted to be among those which will have increased use (2). Second, increased emphasis should be placed on the use of these components in Computer Science Education. Third, the best way to teach this subject is to use a CHDL for all Computer Science courses where a CHDL is applicable.

Classification of CHDL'S

The primary method of classifying CHDL'S is by the level of abstraction of the language. This has varied from four to seven levels by different workers in the field. Throughout this treatise, five levels will be used as reported by Barbacci (14) -- PMS (roughly equivalent to system) programming, register transfer, switching circuit (roughly equivalent to gate), and circuit level.

CHDL'S may be intended primarily for either simulation at a particular level of design or the generation of design details below the level described. Each of these may also be further divided depending on the level of abstraction.

Hardware language also vary depending on the intended user -- academic, industry, with further divisions under each. Obviously the more detail that is incorporated into a model, the more expensive it will be. Cost is reflected in all aspects of the system: software cost, operation and maintenance cost, user learning time, and the time required for the user to develop the hardware description.

Many of the CHDL'S also address specific problems or portions of computer technology -- input/output, graphics, asynchronous operation, timing problems. Lastly the various CHDL'S reflect the technology of the time in which they were developed, changing their emphasis as technology changes.

Each CHDL will be usable over only a portion of this broad area; usually there are secondary uses on the fringe of the primary uses. There is no universal CHDL, however.

Targeted Use for the Language and the System

The system presented here is designed for the register transfer, programming, and system levels and specifically addresses microprogramming capabilities, use of multiple and bit-sliced components. The language reflects the current technology using the LSI as the basic building block and is intended for simulation studies in Computer Science from micro-programmable system designs through flip-flops in an interactive system.

The system description is hierarchic and supports a hierarchic design process, thus allowing successive examination at system, programming and register transfer levels. Hierarchic design is discussed by vanCleemput (100).

Comparison of Calsim to Six Popular CHDL'S

Appendix one (1) compares Calsim with six currently popular languages. These were chosen to represent various types available and to more clearly define the position of Calsim in the broad area of CHDL uses. Calsim is shown in the middle of the table with "**" at items where the difference is significant. A brief discussion of the differences follow.

The time event table uses both a time event, as commonly used in simulation languages such as SIMSCRIPT combined with the part component number. Since a time 'RESET' is also available, branching can be accomplished easily. The need for "major labels" and branching was pointed out in (49).

Memory handling in Simcal requires that the memory content be read from a file and further provides for both micromemory and up to three (3) other memories to also be read into the simulator's main memory.

Special syntax is used by Calsim to describe the type of data being sent or received so that conversion at the simulators terminal will be in readable characters. The port (terminal) descriptions in Simcal allow the user to specify the type of data expected from the port or is being sent to the port. The simulator converts the data to or from Ascii so that it is readable if outputted or can be inputted as Ascii and sent to the computer in its correct format. Heath, Carroll, and Cwik discuss the need for data conversion of simulation output in terms of a postprocessor (49); this idea has been adapted to an interactive system.

Time Resetting System for Copied Items

The nature of the targeted system demanded that the library utility facilitate copying groups of components with minimum difficulty. Since no method of handling the problem could be found in other systems, the multiple copy command was developed. It is available in addition to the command which allows the user to suffix a component name. As each component copied may have several time events within its description, the copy command was then modified to allow both multiple times within the component description to be either replaced or modified as well as allowing several replicates to be made by a single statement. The details of the library and time resetting systems are discussed in sections 3.10 AND 3.11 of the Users' Manual (Appendix 6).

Other Extensions for Computer Science Education

1. Calsim offers the user freedom to represent the machine being studied without being bound to preconceived patterns of computer architecture, such as the fetch/execute cycle. See (44) and (100).

2. Calsim faithfully represents the physical organization down to the bit level and may be used at the gate level. Most register transfer CHDL'S give this aspects get little attention. This principle is discussed at by Borrione in (18) and Bressy in (19).

3. Calsim resembles English to a much greater extent than most other CHDL's. Although Chu stated in the early sixties that CHDL's should resemble a natural language (28), most of the CHDL'S are more cryptic than a natural language.

4. The data conversion available in the I-O chip has already been discussed. Calsim also allows logical action within the chip and allows the port to be "tied" to a file so that data can be transmitted to and from a file directly. Important aspects of input-output are discussed by Parker and Wallace in (77) and (102).

5. Since Calsim must operate at the bit level to fulfill its intended purpose of manipulating the bit patterns throughout a microprogrammable machine, data types are not used. The user, of course is always free to "build" data types into his hardware design.

The Software support System -- User's Viewpoint

The software support system used with Calsim is called SIMCAL (Simulator for Computer Architectural Language). It contains four primary sections in an interactive environment which follow the pattern found in most software support systems for CHDL's. The compiler verifies the syntax of the description, converts the English-like language of the hardware description into tables and checks for errors or omissions. The second section, a simulator driver, interfaces the user to the simulator, providing extensive user support for controling execution and showing status of the simulated logical device. The third component, the simulator itself, carries out the actual simulations commanded by the user in the simulator driver, stopping at break points as instructed. The last part embraces the support programs - Documantation, Library, Help, Table Preparation, and Memory Read-in.

CHAPTER II

HARDWARE DESCRIPTION LANGUAGES

What are Hardware Description Languages?

In every specialized field, one finds unique ways of expressing thoughts, designs and ideas. So in a broad sense "Development of Computer Hardware Description Language's (CHDL's) began at the same time as the birth of the computer" (72). In 1964, Schorr (88) stated "No adequate way of describing a digital system in terms of sequential circuit theory, nor to present this information to a computer, is known". He then suggested a register transfer language as a way of coping with the "Heuristic methods of design". The same year a formal language called LOTIS (Logical Language for Timing and Sequencing) was proposed by Schlaeppi (87). He suggested that software could be prepared to use the semantic output from syntactical analysis of the language to both simulate performance and to synthesize circuits.

In 1965, Chu (26) proposed another language in his report on CDL (Computer Design Language). He used a level of abstraction, just above the electronic component level, now commonly referred to as "Register Transfer Level" to accomplish his stated purpose "To bridge the gap between hardware and software design".

Since then, the proposed uses of CHDL's have greatly expanded and one finds with this expansion, invention of new names for the term CHDL -- ADL (Architectural Description Language), SDL (System Design Language), HDL (Hardware Description Language) among others. Whatever the generic name and specialized or extended uses, they are generally considered a CHDL by the Computer Scientists working in the field. In addition to communication among the designers studying a system, the CHDL's are expected to communicate to a computer sufficient data to create a simulator for the described hardware and automatically generate the circuits below the level described as proposed by Schlaeppi (87). Needless to say, these requirements greatly affect form and content of the proposed languages. See Shiva (89) for a tutorial on CHDL'S.

The languages have been classified as either procedural or non-procedural depending upon the syntactic organization and the emphasis on various aspects of the hardware. Vogel proposes a third approach in (103) consisting of a language based on "Mathematical modeling of real-time automation encompassing the concept of time". Lipovski, however takes a simpler view and calls the CHDL's "A variation of a programming language tuned to the overall needs of describing hardware" (71). Baer (11) states "We can justify CHDL's only because they bring more clarity and impose less complexity in the data and control structure of the specific application". However the difference between CHDL's and programming languages is much more fundamental than indicated by the superficial appearance of the two families of languages.

The difference in purpose between the CHDL's and programming languages is sometimes clouded by the similar constructions of the two. The programmng languages are vitally concerned with algorithmic processes; the CHDL's algorithms are limited. The programming languages use complex data structures; the CHDL's data structures are usually limited. The CHDL's must address timing constructively; the programming languages either ignore timing or treat it as a peripheral issue. The CHDL's must be concerned with hardware organization; programming languages do not even contain such a concept and if addressed at all, it must be done by the language user.

The two types of languages use many of the same basic elements to build their structure, just as a car and a truck use the same basic components. The end use, however, is so different that the actual language design must be, or should be affected in numerous important areas.

A complete CHDL addresses both the physical and functional aspects of the hardware. The physical aspects include component names, connectors, number and name of pins, wiring, physical organization, registers, and clocks. The functional aspects include a description of how the logic state within the hardware changes, how the state is stored, and timing of the logic flow. Among the many languages in use, one finds a wide variation in the levels of abstraction, the physical/operational emphasis, the formal syntax and grammar. For the purpose of this report, we will use the following definition of a CHDL.

> A Computer Hardware Description Language is a language defined by a formal set of syntactic and semantic rules which can be used to precisely describe significant portions of both the physical and functional aspects of a logical design and will support a simulator, hardware generator or both.

Proliferation of Hardware Description Languages

CHDL's have been investigated and reported since the early sixties (95), (87). Their value in the areas of design description, organization of design information, teaching of computer science, documentation, automatic circuit generation and simulation have been recognized and described (24), (59), (60), (76). Su has reported on twenty-one (21) CHDL's in the U.S. (96). More than a dozen languages have been reported by Vaucher and others outside of the U.S., (102). Since publication of those reports in 1974, many additional languages have been reported. Chin has described a language to be used graphically called FLOWWARE (23). Stewart has described a language called LOGAL (Logic Algorithmic Package) and a software system called LADS (Logic Algorithmic Design) (94). Analuf has introduced a language for logic and timing (7), Parker and Wallace have proposed a special input/outout Hardware Description Language (77), (104). Tomek has reported a simulation language called HARD (Hardware Simulation in Education) to be used exclusively for teaching Computer Science. (99). These and many others have proposed new approaches to established uses and extensions of the functions the languages perform. See Appendix 2 for a list of the published languages.

Lipovski describes the proliferation of HDL's, called by him the "Tower of Babel", and suggests industry cooperation as a solution to the problem (71). In spite of the obvious problem with multiple languages, they continue to proliferate. Smith recognized this and stated in the beginning of a paper presenting a new language "... the rationale for creating yet another Computer Hardware Description Language in a field already stocked with languages...'(93). (His stated purpose is to reduce dependance on labels, increase readability, and efficiency). Heath, Carroll, and Cwik in (48) stated "CDL was found to be quite cumbersome to use, ..." and then described modification of several CDL constructions. Su states in (98) "The process of transferring a high level computer hardware description language specification of a system into a logic diagram is still in its infancy."

A committee(Conlan), chaired by R. Piloty, was formed to study ways to bring more order and uniformity to CHDL'S, but apparently has been unable to reach an agreement since no publications from the committee were found.

Since the introduction of CHDL'S two decades ago, over 50 additional languages have been proposed and over 200 papers have been published on the topic. Each of the proposed languages extends existing languages to include a new function or addresses an old function in a different way. In many cases the languages are devised to overcome problems reported in the technical journals by workers in the field. Appendix 1 summarizes 50 of the languages discussed in the references and some of the important characteristics of each language. There is no indication that proliferation has abated.

An Overview of CHDL's

The listing in Appendix 2 illustrates the great variety of Computer Hardware Description Languages which have been written and implemented. The brief discussion of each is intended to give the reader an overview of progress in this field and provide some of the distinguishing characteristics of each language.

The name of each language, usually an acronym, is given first followed by the meaning of the acronym, the authors name and the date of the original article. The references are listed in the order of importance in providing material for the entry. They have been used freely in obtaining direct quotations and paraphrasing for the paragraphs on "Purpose" and "Discussion" of the language.

Each new language developed, presumably, solves a current problem or performs new functions not being performed by existing languages. In some cases the author of the reference material has stated explicitly the purpose of the new language, but in many others the reason for the new language must be inferred from the text of the article. In either case the reason presented in the Appendix is derived from the material in the text of the article to the greatest extent possible.

The "Discussion" paragraph centers around the primary characteristics and uniqueness of the language and where information is available, implementation. It would be desirable to discuss each language using the same format and exploring the same aspects of the language such as grammar, concurrency, level of abstraction, etc. However, this is not possible as the authors have not discussed the same characteristics in their respective papers. The entry therefore reflects the emphasis used by the author of the reference.

Anatomy of a Computer Hardware Description Language

The question as to what a CHDL is has already been addressed, but no details were provided as to the component parts of the CHDL. We now examine the parts commonly found in a CHDL, restricting the discussion, to the language itself, not it's means of implementation.

Registers, Subregisters, and Cascade Registers: Each CHDL must have a way to declare the existence of certain registers and to specify their sizes. In some languages the user is allowed two dimensions, thus is able to create a series of registers with a common name. The languages use both left to right and right to left numbering of the bit positions, some allowing one, some the other, and a few allowing the user to choose. A single flip-flop is usually regarded as a one-bit register. The usual form of register description is to use a key word, usually "REGISTER", followed by the dimension(s) in parenthesis. Calsim follows this form, using "REGISTER" followed by the register name with the dimensions in parenthesis. Calsim will accept two dimensions, the first is the bit length; the second is optional and may be used to establish a group of registers with same name to be accessed by subscript.

It is usual to include a means to describe parts of a register as a subregister just as the term is ordinarily used with hardware. A means is also usually present to cascade two or more registers to form a longer register with a new name. Calsim supports both sub and cascade registers using the keywords "SUBREGISTER" and "CASCADE".

Special Registers: Some CHDL's have given special treatment to the registers common to conventional computers such as SP (stack pointer), MAR (Memory Address Register), PC (Program Counter), IR (Instruction Register), MDB (Memory Data Buffer). Since each of the special registers are treated in a special way within the respective software support systems, the user must be aware of the restrictions on the system in use and its implication during simulation. Since Calsim, like several other languages, makes no assumptions as to architecture, it uses no special registers.

Hierarchical Groupings: With respect to this property, the CHDL's seem to be divided into two groups, one emphasizing the CPU portion of a computer with other devices as appendages to the CPU and the second is more system oriented. Many of the first were developed to synthesize the internal circuits in the CPU. The second group of languages perform these same functions and also allow grouping of the logical components at several levels using some type of connection or communication between the components. Calsim is the latter type allowing up to 99 levels of hardware descriptions, thus directly supporting a hierarchical design process as described by vancleemput (100). However, Calsim will not support design descriptions below the register transfer level.

Connectors: In many register transfer languages, the values can be transferred and tests made without regard to actual hardware connections; they are assumed to exist. Languages like Calsim, which accept hierarchical descriptions, also perform in the same manner within each component description. Between components, however the connectors must be declared, and a means of communication established between the chips. Calsim does this by describing the connectors in a "CONNECTION" statement and connecting the components to each other with a "WIRING" statement.

Signals: The signals in most CHDL's are only 0 and 1, but in few languages the signal may also be defined as "unknown", "high impedance", or "open"; Calsim permits four states, combining "high impedance and "unknown". Values of zero and one have the usual meaning as used in hardware descriptions. Wires not connected to any component are "open".

Data Types: All programming languages allow the user to specify by some means the data type of variables and stored characters. Most languages have integers, character, and floating point formats and may also include double precision and some type of packed decimal or BCD. In those CHDL's which are closely related to programming languages, these same data types are usually present. In other languages such as CDL and Calsim, where such types would mask the primary objective of the language, data types are not used. All data is simply strings of zeroes and ones. Calsim, however, provides for data types associated with each port (terminal) so that conversion to/from ASCII format at the simulated terminal is accomplished.

I-O and Terminals: If the terminal in the "designed" machine is an ASCII terminal, then some action in the machine or program being simulated will have produced a conversion to the ASCII format. Ports or terminals which are used for input or output of other formats such as BCD must be converted if they are to be displayed properly at the ASCII terminal interfacing the DEC-20. It is not desirable to require the user to make the translation himself. If a design, for example, includes a series of ports reading BCD, the user should be able to introduce at the proper point in simulation a series of numbers values which would be converted to BCD by the simulator.

Timing: A wide variety of approaches to timing have been used by CHDL's. The most simple approach is to use each statement (in a procedural CHDL) as a "tick" of the clock and advance the timing one cycle. The simulation then progresses through the instructions just as execution of a program progresses through the instructions. Such languages usually allow looping, and "GO TO" statements of various types which is equivalent to resetting the clock. Many variations of this basic idea have been used.

Calsim uses an event clock which the user must set for each executable component. The active components may have simulated concurrent operations, and each component may use several events. Resetting may be called for in the hardware description language or it may be dynamically reset during simulation as the result of a test. The Calsim system allows the user to "HOLD" by the looping just described.

Concurrency: If a CHDL supports concurrency, additional checking must be performed during simulation. This is handled in a variety of ways, but they all involve setting flags of some type to assure that all actions have been taken before proceeding to the next step. Closely connected with this, is the handling of asynchronous operations. In some cases the language must provide special construction to escape its fetch/execute cycle so interrupts can be serviced in an asynchronous manner.

The Calsim/Simcal system uses an event table in which any number of active components can share one or more events. Those operations are then carried out in the order entered, but no data is placed on the connectors to/from the component until all of the components sharing a time event have completed execution. If there is data being passed between two or more components as they execute concurrently, then smaller time slices are used to further divide the time increments. If a long series of operations must occur in one or more components, then these can be listed under a single timed event, thus precluding the transfer of the signals to the connectors until both components have completed the computations.

Storage/Retrieval of Prechecked Descriptions: Several languages contain "MACROS" which are closely akin to the same item found in assembly languages. Calsim does not use Macros in that sense, but uses a "COPY FROM LIBRARY" concept as found in high level language compilers. In Calsim, a library of components may be written, prechecked, and stored; then copied from the library into the hardware description.

The copy clause has been extended to allow the user to make multiple copies of the item, a situation found in most designs, but particularly in those supporting bit-sliced components. As the items are copied two digits are affixed to the end of the name giving each a unique identity. An alternate method allows the user to add a suffix to the component identifier.

Calsim also allows the user to adjust the time found within the items copied. This may be done even when there are several times within the item and when several items are copied with a single statement.

Elements of Software support Systems for CHDL's

If a CHDL is to do more than provide communication among personnel, a software support system must be prepared which will read the language and perform functions in response to the description. It is usually prepared following the language design rather than concurrently with it and must perform one or more of several functions. The software may be designed for batch use or for interactive support; it may be a single program or a group of programs; may provide little or significant support. A description of these functions with special attention to the support software for Calsim follows.

Compilation: Compilation is widely recognized as consisting of three tasks (6), namely:

(1) Lexical Scanning.
(2) Syntactical Analysis.
(3) Generation of Semantic Output.

The hardware description language compilers follow the same form and like the programming languages must output both data structure and executable code. In the seventies, the grammar of programming languages came to be understood well enough that syntax directed compilers could be written which would be driven by tables from grammar analyzers (6). The compiler writer still must prepare the lexical scanner, a relatively simple task, copy the syntax algorithm and provide the semantic output statements. Although the semantic output, by far the largest of the tasks, still must be prepared, the task is now highly structured into small manageable pieces. The Calsim compiler was developed using the LALR grammar analyzer by LaLonde of the University of Toronto (63) and residing on the UTA IBM-4341.

Interpretation: Interpretive languages, such as APL, must perform the same syntactic checks as are performed in compilation, but tables are not created from the description. The incoming code is executed directly. In such cases data items have been previously declared, with storage already established.

Simulation: Even though several functions other than simulation may be performed by software for a CHDL, the software may be referred to as a simulator. The discussion here will be centered on the simulation itself, not the other functions. The simulator actually performs two (2) separate and distinct functions --

(1) Interfacing with the User.

(2) Carrying Out the Execution.

The first of these, called "Simulator Driver" interfaces with the user, accepting the commands given, analyzing for correct syntax, setting various flags and finally passing control to the simulator. The simulator itself carries out the logical operations which represent the machine logic, testing the flags previously established for needed displays and breaks and finally returning control to the simulator driver. In a batch environment, the first function is usually managed by a deck of formatted control cards. In an interactive environment the service performed by the driver for the user is usually broader, but varies widely up to that represented by the following functions. The reader should note the similarity between this list of functions and those performed using instruments to bench check a hardware prototype.

1. Preventing loss of user control of the simulator.
2. Wide capability to set break points.
3. Step-wise execution.
4. Execution through a given number of steps.
5. Use of "MENUS" to set-up display patterns.
6. Means to change values in the simulated machine.
7. Display of values in registers, wires, memories.
8. A TRACE command to follow the execution path.
9. A record (history) of where execution has occurred.

The actual simulation in languages oriented around an instruction set, follows the pattern of the instruction set. They basically follow a tree like structure to arrive at the particular instruction. When the necessary data moves or calculations are completed, the simulator is ready for the next instruction. When action is taking place concurrently in several units, as it is in most real devices, the action is more complex. Many CHDL's simply execute the logic in the order submitted using "GO TO" to provide repetition when required, while others, such as Calsim, use a table of events and provide a method of resetting the time. If concurrency is addressed, then the software system must have a way to assure that simulated concurrent actions fully represents the action without producing erroneous results.

Calsim takes the incoming logic from the description, reduces it to compressed symbology, performs a slight amount of rearranging and stores it for use by the simulator. The grammar of the stored logic is also LALR and the simulation is driven by LALR tables. At the start of execution, the "current" time points to the location in the event table where the first instruction is found. The process then proceeds through the table incrementing time and executing the logic for each component in turn. At the end the process loops back to zero time or to the pre-set time values unless the order has been changed by a RESET entry.

Synthesis of Circuits: The emphasis in the early years was to relieve the designer of the tedium of producing electronic designs at the gate level and below which contained hundreds or even thousands of repetitions of the same circuits. Sometimes such software would also generate graphics representing the logic design. System and computer design now start with medium scale integrated circuits (MSI's), large scale integrated circuits (LSI's), and very large scale integrated circuits (VLSI's), chips containing up to 100,000 gates. The original problem no longer exists in circuit design of application hardware composed of these large chips. The problem, however, is intensified in the design of the chips themselves, making this a primary function in that area. The system presented here is intended for the study of application hardware and systems using these large chips. Synthesis of circuits is not addressed in this work. One of the other languages are needed to address hardware at the switch circuit level and below.

Storage/Retrieval of Multiuse Code: While copying of stored code is rather common in programming languages, there is little mention of it in CHDL references. The use of MACROs has already been discussed. There appears to be a much greater need to apply a "copying" technique to the use of CHDL's since most system designs contain only a few components used several times. In a given facility, activity is usually limited to two or three families of chips. Each family may have 10 to 30 members depending on the type of chip. The bit-sliced families tend to be nearer the upper value.

Miscellaneous Other Functions: The support software must perform several other functions such as :

(1) Provide user assistance as requested.
(2) Print tables of read-in data.
(3) Provide documentation.
(4) Accept and store memory content.
(5) Provide status of certain actions.

Special Problems with a CHDL for Bit-slice Hardware

Microprogramming itself presents several special problems in designing a CHDL and the support software for the compiler and the simulator. If architecture is bit-sliced also, using the type of components found in the AMD-2900 series, TI SN-74S481 and SBP-0400A/401A or Intel 3000 series, then additional problems of handling arise. These are discussed below.

Memories: A structure for microprogramming will always require at least one additional memory, and in many cases other ROM's (Read Only Memorys) will also be required. The contents of these memories, being essential to even minimum testing, requires that means be provided to conveniently enter the data through a file structure. The Calsim system addresses this by providing read-in capability for main, micro, and three auxiliary memories. The data for these memories can be prepared, edited by any convenient means then read into the Simcal system just prior to simulation. The values in these memories can be changed during simulation as can all data values within the system.

Fetch/Execute Cycle: Many, but not all, CHDL's have the von Neumon cycle built into the language in such a way that no practical way exists to avoid it. While most application designs still perform fetch/execute within the design structure, many others perform the fetch/execute at more than one level and still others do not perform fetch/ execute at all, in the conventional sense. In any case, experimental architecture experiments can hardly be performed if the architecture accepted by the description language has already been partially determined. The system presented here makes no assumptions as to architecture; where the fetch/execute is used, the user simply provides it in the hardware logic of the design.

Using 2 and 4 Bit Slices: A more significant problem is the mundane handling of multiple identical cascaded chips while avoiding repetitive and burdensome bookkeeping on the part of the system user. The syntax of Calsim partially overcomes these problems. First the language represents the logic of the machine above the gate level. Second the copy statement allows copies to be made from descriptions while avoiding the burdensome task of repeating the description. The timing of these duplicates are easily modified when the copy statement is prepared, and the wiring statement allows the connections to be accomplished with little difficulty.

Pipelining and Concurrency: The problem of concurrency in structures with a CPU, memory, and a few terminals is fairly simple when addressed at the register level. When this same capability is addressed using a microprogrammable structure, the problem of concurrency becomes one of the primary problems and if pipelining is added, the problem becomes even more complex. The method used in this system has already been discussed and is in fact fairly simple. Each time event may be used by two or more components. While the clock is pointing to a particular value, the operation is completed on each component but no values are placed on the connectors to the component until all items for that time event have completed execution. At the end of the time increment, if the outputs to a common wire do not agree, then a warning message will be given by the simulator. It is interesting to note that if a hardware prototype had been under test, no such error message would have been issued by the test equipment monitoring the test.

In pipelining, in which data is pushed into a register on one side while the previous signals already in the register are being executed from the opposite side, special effort may be needed. The user may have to set up a latch, register or other member which may not actually exist in the hardware itself or may actually exist but is not shown in the manufacturers' published descriptions.

CHAPTER III

MICROPROGRAMMING & MICROPROGRAMMING SUPPORT SYSTEMS

Historical Review

Since the introduction of the microprogramming concept by Wilkes (107) machine instructions have been controlled by a series of smaller steps which in turn are controlled by a read-only memory, usually called a micromemory. The development of LSI technology with chips containing thousands of gates, has led to the design and production of the central processing unit in elements of two or four bit widths, which can then be assembled to any width desired. These processing elements (CPEs'), called bit-slices, are being used more extensively in both application and computer hardware as the number of gates per chip increases (20). In fact the better organization possible and the higher gate density of currently produced bipolar chips promise a change in computer organization itself (2).

A review of recent publications, concerned with bit-slice and other microprogrammable designs, reveals that most of the writers discuss simulation in terms of either specific software simulators or the use of emulators of various types (5), (22), (58), (62). The requirements for a system to check microprogrammable hardware designs are discussed by Fuller and others in (42) and also by Gordon and Stallard in (46). The most popular way of checking a design seems to be through the use of a hardware prototype in conjuction with a logic analyzer (29), (8), (32), (65). It is proposed to substitute a software support system using the Calsim language as a basis, to perform the verification function now performed by the hardware support systems.

Uses of the Microprogrammable Concept

The case for the use of microprogramming as a far more organized approach in the design of computers was well established in the fifties (107) (108) (109), however it was not until the development of the family of IBM 360's in the early sixties that its use came to be realized in a commercial sense. Through its use, IBM was able to develop a family of machines, all of which used the same set of assembly instructions, but varied widely in speed, capacity and price. In effect, the lower priced machines were emulating the more expensive machines.

It had long been realized that sales of newer machines could be severely delayed because of the user's huge investment in software already intact and working.

Microprogramming offered a way to overcome this problem by allowing the newer machine not only to offer all of the newer features but to also completely support the older software. This is more fully discussed by Burris in (21) and Rosin, Frieder and Eckhouse in (86).

Hewlett-Packard, among others, added to two of their models (21 and 1000) features which allowed the users to prepare and add their own instructions to those supplied with the machine. The memory was called a "writable Control Store" and could be used to prepare an instruction which could speed up execution where a time constraint problem existed. However intriguing the idea may appear on the surface, there is no indication that this has been a widely used approach to problem solution.

MOS vs Bipolar Technology

By the mid 1970's it was apparent that MOS technology would produce a flood of microprocessor driven application devices by the early eighties. This has certainly come to pass. Texas Instruments, Intel, and Advanced Micro Devices also saw the need to organize the fundamental building blocks of the microprocessor architecture around something other than either the complete microprocessor chip, which limited design freedom or Medium Scale Integrated (MSI) chips with 10 to 50 gates. The MSI chips included such devices as registers, latches, adders, decoders, and multiplexers (2). All three companies chose to invest in Bipolar technology which was at that time at least ten times as fast as the MOS technology and used larger chips with a different organization. To make the application hardware versatile, bit-slice chips were introduced which allowed the designer to vary the size of the hardware under construction. The units other than the Central Processing Element (CPE) included special chips for selecting the next microinstruction, priority handlers, direct memory access, front panel control, program instruction counter and input-output control.

The problem of testing and proving a microprogrammed system is more complex than a conventional machine since an additional level is involved. This means that one must test the hardware system to assure that it will perform as intended, but this must be done with a firmware sequence as yet untested. The hardware emulator and prototype model resolve this by allowing the design group to carry out tests to assure the machine will perform as intended. Meantime if the firmware which will actually drive the device is to be completed, a simulator must be constructed so that written and assembled firmware can be tested. The system presented here allows the designer to provide a system description at the programming level while retaining hardware identity. The microprogrammer can then use a copy of the description to simulate the firmware execution. As the design develops and the hardware designer provides additional details, the hardware description is modified and the firmware rechecked to assure that the defined hardware and the firmware are still compatable.

A hierarchical design process (100) may be used by providing only functional details in the first pass, thus making the description quite close to the design specification. This can be used as a test bed for firmware (microprogram) and while it is under preparation, the hardware design can be augmented with further details in a second version of the hardware. When the firmware is complete and proven on the functional design, it can then be used to check the detailed design of the hardware.

Assembly of Microcode

The problem of developing an assembler for microcode will not be addressed here. Each design must have special treatment for the special fields it contains and hence must be tailored for the specific design. These problems are discussed at length in (57). Once the microcode has been assembled however, it is ready for the Simcal system to use.

CHAPTER IV

CALSIM: COMPUTER ARCHITECTURE LANGUAGE FOR SIMULATION

Project Background

In 1977 while studying bit-sliced microprocessors, Dr. Walker suggested that there should be a way, not only to "assemble" a set of microinstructions, as was being done by the XMAS-CHROMIS system, but to also verify that the program was correct. The bit-sliced instructions, unlike machines using fixed length instructions, might be any length and have any number of different configurations. It was from this need to verify the microinstructions that this project developed.

The implementation of the project started in the fall of 1978 with a literature search, followed in the first half of 1979 with design of the language. The top level of the software support system was written in the fall of 1979 and routes the user to the various environments -- the compiler, simulator, library (to search/ edit), micromemory read-in, main-memory read-in, table, and documentation requests. The lexical scanner and the syntactical analyzer portions of the compiler and several of the minor modules were completed in the spring of 1980. Work on the semantic code generator was completed in the summer of 1980, allowing the compiler to be used on a trial basis during the fall semester. Revisions, based on input from the students, were completed the next semester. Work on the simulator was completed during the summer of 1981. Preparation of the dissertation extended from August 1981 through September 1982. The grammar was rewritten in August, 1982 and the final modifications of the software was made during June though September 1982.

The Requirements for the Grammar

The requirements for Calsim were largely developed before the Calsim language was written the first time using the characteristics found in other CHDL's. These were adopted, extended, and modified as appropriate. Each of the appropriate articles were also searched for descriptions of weaknesses in other languages and suggested extensions. These were collected and used in preparing this list of requirements. The last significant changes to the grammar were made as new aspects of CHDL's became apparent during the preparation of this thesis.

1. The grammar of the language must be LALR (Look Ahead Left Right) so that support for verification and syntactical analysis of the grammar can be performed on an available grammar analyzer. This assures a cohe-

sive, more easily maintained language and a compiler/simulator with fewer errors than use of ad hoc methods. The superiority of compilers written using formal methods is now well established in the area of programming languages (6). The same superiority is true of CHDL's.

2. The language must be free form to save the user the bother of rigid formatting. The text must extend from column 1 through column 80 using space or spaces as delimiters for the words of the language, just as natural languages use spaces.

3. The language must be capable of describing a hierarchical structure so that representation will be similar to the hardware. This representation is to be centered around a "system" but must also accept collections of systems. The description should closely parallel real hardware and be capable of describing logic and storage from the bit level upward.

4. The grammar must not contain features which presume a pre-determined architecture.

5. The language must support comment insertion into

the middle of text as well as full lines. This must be accomplished in an easy and natural way.

6. The grammar must allow descriptions of registers, subregisters, and cascaded registers of any practical length and subscripted to one level if desired.

7. The language must be statement oriented making the task of preparing the description easier for the user.

8. The grammar must provide capability to describe parallel operations so simulation is realistic.

9. The grammar must provide timing ability so that the asynchronous operations and interrupts can be simulated to the level of user interest. The timing must allow the logic to repeat operations until the specified conditions are fulfilled thus duplicating a "HOLD" condition in the hardware.

10. There must be provision to use micromemory, main memory, and at least three auxiliary memories copied from external files.

11. The system must provide capability to develop data

to fill the memory files separately and provisions to "read" the data into the system when the file is complete either before or during simulation.

12. The language must provide special features for ports (terminals) so that data passing through the terminal will be converted to ASCII as received at the terminal and from ASCII as sent from the terminal.

13. The grammar must support entries to and retrieval of descriptions from a library. The copy ability must include provisions to store complete descriptions during compilation and to retrieve the description during that compilation or a subsequent compilation. The library must be accessable for listings and searches of the contents from an area of the software other than the compiler itself. The time events stored in the component description must also be adjusted at the same time the item is copied, using the content of the copy command to determine the new event times.

14. The grammar must provide for entry of multiple components of the same type with minimum difficulty. The use of multiple components in bit-sliced and conventional microprocessors require that this be done

to reduce the time used for multiple entries.

15. The grammar must provide for logic description in a way most likely to be in the language of the user, such as found in Fortran, Algol, or PL/I.

16. The language must provide for connectors and sets of connectors between the components so that output from one component may be either held or transferred to another component in a manner similar to hardware.

17. The language must provide for a start condition. This will consist of a way to set wires, registers, pins, and memories to specified values so the device can begin execution.

18. Language construction which allows the compiler to perform work which in fact must be done in hardware must be avoided or be made apparent to the user.

The Grammar -- Development

The grammar for the Calsim language is LALR (look ahead left right) and was developed using a grammar analyzer by LaLonde (63) which resides on the UTA IBM-4341. The

LaLonde analyzer verifies the grammar as meeting the requirements of an LALR (K=1) grammar and produces tables which control the execution of the syntax analyzer.

The constuction of the grammar itself, is somtimes better understood by looking at the language examples. These may be found in abundance in Appendix 3 of the User's Manual (Appendix 6). The Backus Normal Form productions (BNF) are shown in their entirety in Appendix 3.

The special output (grammar and tables only) from the LaLonde analyzer were transferred to the DEC-20 using a link provided by the UTA Computing services and "cleaned" with a special piece of software called "TRAPLINK" which is part of the UTA Computer Center Utilities. Several formatting steps were performed on the output file from the analyzer to make it compatable with the Simcal software system as written. These steps include elimination of "page" lines, comments, leading spaces, a 'START-STATE LITERALLY ...' statement and change of the null character in the vocabulary list to a space. A line with "END OF BNF" starting in column 1 was added between the grammar listing and the driver tables. The lines in the tables which declare table size were moved up, keeping the same order, to a point just above the vocabulary listing enabling all of the table sizes to be read by the Simcal system at one time. The LaLonde analyzer numbers the files from zero, but Cobol requires files to begin at one, hence the table limit and the index value is increased by one in the support software.

The "END OF BNF" marker separates the productions (which may be requested by the system user) and the tables which actually drive the syntax analyzer. By transferring a single file from the grammar analyzer with only the modifications listed above, added assurance is provided that the displayed list of productions will not be a different version from the tables which actually perform the compilation. The transferred tables perform the following functions:

1. Control the execution of the syntax analyzer.
2. Provide the structure for the semantic generator.
3. Provide a listing of the BNF productions.
4. Support a set of software tools built into the Simcal system for program development and debugging. The routines have been left in place but cannot be "Turned on" unless a keyword is entered.

The Grammar -- Structure

The grammar requires a system name at the beginning of the hardware description, followed by a series of statements. At the end of the system description, an end statement is used and may be followed by other system descriptions. The last card is "END OF DESCRIPTIONS" card. Each system described will start with the items which are to be stored in the library, followed by the timing and hierarchy statements beginning with the top level. The descriptions of the connectors, memories, input-output devices and other components are then entered. The last statements to be entered should be the wiring statement, which provides the connections between the components, and the start statement which provides the conditions for initialization of the machine action. The reader is referred to chapter 3 of the Calsim/Simcal User's Manual (Appendix 6) which contains detailed instructions for each of the Calsim statements with examples of syntax.

CHAPTER V

SIMCAL: THE SOFTWARE COMPILER/SIMULATOR SYSTEM

The Requirements for the Support System

Most publications which discuss CHDL's say little concerning the software support system which actually does the work of compilation and simulation. In the fifty or so papers in the list of references which discuss CHDL's, there is little material on the construction of the software compiler or simulator. Hemming and Hemphill (51) develop a case for a specialized simulator as opposed to the more general simulators such as GPSS, SIMSCRIPT or GASP. The requirements for the support system were developed not only from discussions in those publications, but also from experience with interactive systems and from checking prototypes using instrumentation of various types. Personal use of the DEC-20, IBM-4341, PDP-11, INTEL 8080 and Zilog Z-80 development systems also contributed to the development of the requirements list.

The objective of the support system is to provide an easy to use package which supports the user in a friendly and natural way. It is probable, with the wide spread use of small home computers, and the ease with which they can be operated, future user acceptance will be seriously limited for software not meeting high standards. The requirements list for the software system follows.

1. Provide a series of environments which match the various functions the software will provide.

2. Provide "?" in each environment so that the user can immediately determine the commands available.

3. Provide "HELP" in each environment so the user can find immediately the function of the particular module.

4. Make sure all responses provided by the software are friendly, useful, and clear.

5. Eliminate all responses which show "Do not understand" type answer when it is possible to make a reasonable assumption and continue the processing.

6. Provide an entry level message to assist those not familiar with the system.

7. Provide documentation of the system from within the system itself. Provide printing on a line printer for the User's Manual.

8. Build the compiler to print local messages (at point of error) which are clear, concise, and complete.

9. Build the compiler to output tables to drive the simulator and also provide the user with a tabular description of the object machine.

10. Format the tabular description for easy reading and to assist in discovering discrepencies.

11. Provide for at least three (3) data files which can be read into the executing machine or will receive data from the executing machine.

12. Provide library support to interact during compilation to accept and retrieve descriptions.

13. Provide the means to "COPY" multiple copies and at the same time change each timing for each copy as required.

14. Prepare a simulation driver which allows the user to perform a wide variety of functions including:

   * Set values of pins, registers, memories, wires.
   * Display logical values of each hardware item.
   * Set locations of break points in a table.
   * Provide means to build menus of display items.
   * Set a time loop for execution repetition.
   * Execute in a "Run-to-break" fashion.
   * Execute a specified number of steps.
   * Execute one step at a time.
   * Set a trace which shows execution sequence.
   * Save trace on a history file.
   * Allow the user to "Tie" a port to a file so that incoming data comes from a file or output is collected onto a file.

15. Construct the simulation driver using LALR (k=1) grammar and the LALR syntax analyzer.

16. Construct a simulator which directly executes the user logic using a timing system to control the sequence of executions to support interrupts, concurrent operations and asynchonous communication.

17. Provide support to search the library and print an index, a description, or all of the descriptions and also copy from the library to the compiler.

Construction of the Software System

The software was developed on the DEC-20 using Cobol as the source language. The system is well modularized and contains about 8500 lines of code including the embedded documentation. The code follows the standard practice of most Cobol users - meaningful names for data items and paragraphs; modularization; careful attention to alignment of IF/ELSE, PIC, level numbers and other items; use of indentations and skipped lines; restricted use of "GO TO"; verification of numerics at entry time; and comments where there may be doubt as to the function being performed or the method being used to achieve that function. The construction of the program follows that advocated by Armstrong (9) - decomposition of the program into successively smaller modules through the use of the perform statement.

The user enters the system at the "ENTRY" level and is then able to go to any of the twelve major areas of the program. Each of these areas comprise a major module and contain commands of their own and explanations of the functions performed in the area. The reader is referred to the CALSIM/SIMCAL USER'S MANUAL found in Appendix 6 for a full explanation of the software system from the user's viewpoint. The discussion here will be restricted to the more interesting features of construction in several of the major modules.

Entry Loop: The entry loop provides an entry message which explains the system to those unfamiliar with Simcal. It's primary function is to route the user to the various areas of the software support system as requested.

Compiler: The compiler contains five (5) significant sections -- (1) the pre-compiler, (2) lexical scanner, (3) syntax analyzer, (4) semantic code generator and (5) post-compiler. By far the most significant in terms of effort is the semantic code generator. The pre-compiler reads the grammar tables for storage and querries the user for certain information. The lexical scanner selects the next word or character from the incoming hardware description, determines its position in the LALR vocabulary, then passes both position and word to the syntax analyzer. The syntax analyzer follows the standard algorithm for LALR grammars by checking the correctness of the syntax, and if correct either requesting the next token from the scanner or calling the semantic code generator to write the output for the correct production. If not correct, an error message is written and a recovery routine executed. The post-compiler performs several cleanup functions on the collected files.

Library Construction: The library is constructed using a single sequential file and two working files. When the the compiler submits an item for entry, the table is accessed, and if the item is not present, the description is stored on working file 1. If the entry fails, it is so noted in a key and processing is stopped. If the entry is satisfactory, the item is merged with the master file. Copying is accomplished by locating the item in the master file and making one or more copies modifying the names as required. The file is closed and re-opened and a key causes the temporary file to be read instead of the incoming hard-ware file.

The library is also accessed from a "LIB" environment here the user requests an index, a particular listing, a full listing, or a deletion. In these cases the file is opened and the desired item or items are printed or deleted if found otherwise a "Not found" message is printed.

Simulator Driver: On entering the simulator driver, the software reads the syntax tables for both the simulator driver and the simulator and provides the user with a prompt for a simulation command. The User's Manual contains complete instructions for use of the simulator and a listing of the key words which can be used as commands. The driver accepts a command string on a carriage return entry, analyzes the string for correct syntax and sets the proper flags or takes other action as required. If the command is an execution command, control is passed to the simulator otherwise the driver takes action.

Simulator: The simulator, like the compiler and simulator driver also uses LALR grammar. The grammar was copied directly from the statements of CALSIM grammar containing logical operations, however compressed character representation is used for the simulator grammar. When control is passed to the simulator, it sets a counter at one and starts a count each time a set of execution statements is accessed. This is compared to the upper limit which was set in the simulator driver to no more than 500 steps. The operations are then carried out according to the event table starting with the current time.

Programmer Support: Programmer support is provided throughout the program by comments and working modules which have been left in place. These are inactive unless the programmers key is set at a particular place in the program by entry of the proper password. Explanations to the programmer for the use of these displays are contained in the program listing. Typical of these are the following:

1. A printout of the BNF production each time a reduction occurs.

2. A display of a set of variables connected with syntax analysis each time a token is sent from the scanner to the syntax analyzer.

3. Display of a trace through the four major sections of the syntax analyzer including a display of the token being processed.

The software support system - implementation

The DEC-20 at UTA was chosen over the IBM-4341 because of its greater interactive capability. The only languages available on the DEC-20 were Fortran, Cobol, Basic, and an unsupported version of Pascal. Pascal was eliminated because of the lack of adequate documentation and lack of portability. Basic seemed to lack adequate strength for the task. Finally a careful comparison of Fortran vs. Cobol was made for the task at hand. Although Cobol is considered a "Business" language and Fortran a "Scientific" language, the features of the two are similar in many respects e.g. computations look the same in both languages except for the word "Compute" in Cobol and will be carried out the same way given that the same data type is used.

An examination showed there would be many routines to support the interaction between the computer and the user, a significant amount of character manipulation in the three scanners and syntactical analyzers, a multitude of tables to be read, constructed and displayed, and fifteen files to be processed. The only area where time would become critical was in the simulator where routines would be repetitive. Since "computations" would involve bit level manipulation, it was necessary that data movement be as fast as possible. The comparison of the two languages follow.

	FORTRAN	COBOL
Functions and passing data between modules	excellent	poor
Manipulating bytes	poor	excellent
Data structuring	fair	excellent
User interface commands	good	excellent
Searches	fair	excellent
Data editing, checking	poor	excellent
Report preparation	good	excellent

Cobol seemed to be the stronger of the two languages for this application and was chosen over Fortran.

CHAPTER VI

A PROPOSAL FOR USE IN TEACHING COMPUTER SCIENCE

The Proposal

By the time a Computer Science student is introduced to a CHDL, usually in a first course in Computer Organization, he is already familiar with some of the aspects of computer components and architecture. He is aware of the function of registers, CPU, memories, the ALU, and control portions of the CPU. He may also have extensive information beyond this from personal study, other courses or use of home computers.

One of the problems at this stage is to give the student a means to communicate so that he can be sure of both understanding and being understood. Chu introduces CDL in (25) in chapter 1 and uses it throughout the text to describe the components and suggests its use by the student to work the exercises. His emphasis is on using CDL as a means of communication and as a symbology useful in study of logical design as does Posch in (82).

The use of a CHDL to bring uniformity and understanding to computer design is quite similar to the use of a high level language, such as Fortran, to teach algorithmic processes. Without the programming language, a symbology of some kind must be used, and it is likely each person will use their own variation of the specified symbology. Fortran brings order - the compiler requires that every description meet requirements, unfailingly detecting all errors. It is well recognized that programming greatly increases ones ability to understand algorithmic processes, primarily, some might say, because one is forced to "Do it over and over and over again until he gets it right".

In the study of computer organization, simple hardware design exercises may be checked in a laboratory in which observed results are recorded and later placed in a report. More likely, the exercises are not verified through the use of hardware at all, but handed in as homework or quizzes after deskchecking. The instructor then reviews and grades this work and the student must wait for the feedback. I propose to use Calsim and its simulator in exactly the same way that the Fortran and it's compiler are used to allow the student to detect and correct the errors, make the repairs and try again. This rapid feedback, certainty of error detection, and freedom to experiment using a tireless machine, all contribute to the learning process. It also frees the teacher from significant amounts of grading without sacrificing his ability to verify that the concept has been grasped or needs attention.

Many of the CHDL's described in Appendix 2, were written in universities and were at least partly intended for student use. Some of the languages have been included in textbooks (25). APL, which has been proposed as a language to describe hardware and software with equal ability, is widely known and available as are Chu's CDL and Hill's CHDL. At some universities the CHDL's have been used extensively for several years. A course at Stanford is described by vanCleemput in (101). Yet a search of the current journals shows a limited interest in the use of CHDL's as a tool in teaching Computer Science (7), (60), (82), (92). Tomek (99) states as the reason for this:

(1) The CHDL's are professional designer oriented.

(2) Inexpensive portable models are not available.

(3) There are no text books covering the material.

Calsim has been designed to overcome these problems. First the language itself is easy to learn because of it's close relationship to English, its statement orientation, and its meaningful error messages. Second, the support system (SIMCAL) is a friendly, interactive system with "guideposts" at each "corner" to keep one from getting lost or discouraged, leaving the student free to concentrate on work content, not the mechanics of the system. Third, the User's Manual is complete and easy for the student to use. It is produced in it's entirety in Appendix 6 and contains examples over all levels of hardware study. Fourth the system is written in a widely available, easily modified, high level language (Cobol) allowing easy movement from one facility to another.

In the use of the system, the student is free to use designs completely his own or the examples from the User's Manual. Each of the examples include a diagram of the hardware, a Calsim listing, and an explanation of the operation of the unit. Machine copies are available to the student, precluding the use of valuable time reentering the descriptions. The system will support a wide variety of designs over several levels. At the beginning of a Computer Organization course, flip-flop descriptions are available for use by the instructor with the student simulating the operation, saving a printout of the action; or the student may be required to modify the descriptions and show the simulation. The description of adders, shift registers, multiplexers, and parts of the ALU are available in subsequent examples.

At a more advanced level, a complete machine using microprogrammable component parts may be prepared by the student. An alternate approach is to use completed designs and require the student to make modifications. The student then uses the simulator to observe the changing values to prove his design. At a still more advanced level, the student can be required to design a complete hardware/firmware system using microprogrammable bit-sliced components. Using the system, the student is able to see graphically how the microprogramming works and prove both the hardware and firmware designs. Trade-off studies between firmware and hardware can be performed also.

Introduction to Calsim in Computer Organization

Use of the system would start with the first course in computer hardware organization in which the student uses the simulator on a series of simple circuits -- flip-flops, adders, shift registers, latches, multiplexers, etc. These laboratory assignments would require the student to simulate these devices which have already been written and are available from the system. The exercises would include simulation using several different inputs, modifications, and extensions of the design. The results of the exercise could be printed directly at the terminal ready to be turned in. The examples shown in the Appendix of the Calsim User's Manual (Appendix 6) could be used as assignments.

After several exercises, the student would be somewhat familiar with both hardware and Calsim and could be introduced to the more complex structures of the language. The examples found in the Appendix of the User's manual contains descriptions of program control units, input-output devices, and ALU circuits. Each example contains a hardware description, the Calsim code, and a block diagram of the unit. In most cases the design presented is already available on the machine saving the user the bother of reentering. By the end of the course the student will be able to describe computer hardware, compile, simulate and check out system designs. The graduate students could be assigned more complex projects.

Use in The Study of CPU's and Microprocessors

In a microprocessor course, several different chips are usually introduced and the student is expected to do programming in at least one of the assembly languages of the chips. This may be done batch wise or in some cases using a development system from the chip manufacturer. In the last few years development systems have become available which will do any chip set if the ROM for that set is purchased. The Calsim/ Simcal system meets this requirement in the same way. The instructor would need to prepare a description of the target chip and a "terminal" description in Calsim. The hardware description would include the CPU chip, the I-O chip, a bus and a "designers" terminal to make communication with the simulator easier. This would be available to the students who would then prepare program code using an available assembler. The code would then be read into the Simcal system along with the machine description and the chip would be ready to interactively simulate execution.

Use in the Study of Microprogramming

Examples five (5) through eight (8) in the Appendix of the User's Manual provide approaches to introduce the student to microprogramming. The examples include a description of the AMD-2901 CPE, the AMD-2909 sequencer (2), and the AMD-2900 learning kit (3). The learning kit was designed by Advanced Micro Devices to introduce those in the field not familiar with microprogramming to some of the principles of controlling execution through firmware rather than hardware using AMD-2900 components. The example does not attempt to replace the kit manual, which must be studied and used along with the example.

A second approach to introduction of microprogramming is presented in example seven (7). This approach uses a small microprogrammed computer representing no machine in particular, called MIC (Microprogammable Instruction computer). The machine is meant to perform a useful function and generally works at a higher level than that in the learning kit, which is primarily at bit level.

Use in Application Program Design Studies

The typical course in design studies includes one or more projects in which the student will solve a functional problem by selection and integration of several components. The student may have the opportunity to try the design in hardware, but more likely it will be only a paper design. It is well recognized that production of a prototype, however desirable, cannot usually be done because of time, cost of the components and lack of skill to assemble the prototype. The student, of course, never has the opportunity to see the defects in his design or the problems which would be uncovered if the prototype was built. The work many times is at a level which blurrs the details of how the interfaces are accomplished. The use of Calsim is a natural step toward the same goal and accomplishes well over half the benefit of a prototype with far less investment by either the student or the university.

For this purpose, the instructor would use a library of chip descriptions. The student would use the library to build his system much as he would if actual hardware chips were used for a prototype. In some cases the instructor might have the student add items to the library.

Class Experience with Calsim

The language and the compiler/simulator were used and evaluated by a group of seventeen (17) graduate students in an introductory computer organization course at UTA in the fall semester of 1980. The students, all of whom had undergraduate degrees in fields other than Computer Science, had completed six to twelve hours of graduate Computer Science prior to taking the course. By mid-semester, the students had covered operations, number systems, Boolean algebra, gate networks, and elementary logical designs.

For the mid-semester (take-home) examination, students received a microcomputer design with some fifteen explicitly ..ired components along with a detailed explanation of the action of each component. The students were required to design several of the units using gates, e.g. a unit was shown which could select the next micromemory address from a ROM, the pipeline register, or the microprogram counter. The students were required to design a selector switch which would place the correct value on the output pins of the unit based on the input from three control lines.

The term project, assigned during the second week of the semester, required the students to choose a microcomputer and to develop a block diagram using the components. They were then to describe in the CALSIM language the hard-..are represented by the block diagram and to process the description through the compiler. Finally, critiques of the CALSIM language, the compiler/simulator and their use as teaching tools were required. To encourage the search for defects in the Calsim/Simcal system, extra points were given for each Software Trouble Report.

Several 30 minute lectures on the CALSIM grammar, language and compiler/simulator were given during the third quarter of the semester. The students were permitted to work individually or in groups; a total of eleven reports were received. Student program lengths varied from 95 to over 1000 lines of code. The components numbered from 12 to 20. The response to the language and grammar indicated that both were easy to learn and use. This was confirmed by the relatively long error-free programs.

CHAPTER VII

CONCLUSIONS AND POSSIBLE EXTENSIONS

Possible extensions to the work seem to lie in three areas -

(1) Testing of the system to discover omissions, errors and areas requiring improvement.

(2) Expansion of the software system to provide new and improved capability.

(3) Application of the methodology to other Specialized Description Languages (SDL).

System Testing

A system has been presented which proposes to go beyond the capability of the current CHDL's. Potentially it offers not only a language but also the necessary support to provide a viable learning environment over the full span of Computer Science education. Significant testing, however remains to be done in the following areas.

(1) An independent party needs to compare all of the requirements, as detailed in chapters IV and V, to to the actual delivered product. This should be

done by preparing a test plan that checks each item for inclusion and performance.

(2) The system needs to be used at the beginning level (Computer Organization) for at least two semesters using feedback from the student to correct system deficiencies.

(3) The system needs to be used by a group of students studying microprocessors the first time to determine changes needed for chips such as Zilog 8000, Intel 8085 or similar microprocessor chips.

(4) A group of advanced graduate students in a special seminar course could develop the descriptions of many of the components needed in the other courses and could also provide valuable critiques.

(5) Use by a group of students for microprogramming only, the instructor providing a proven "hardware" system.

Software System Expansion

Several software modules could be prepared which would complement and extend the system. These are listed below.

(1) An interactive module to accept hardware descriptions from the user. This would be of assistance to the beginner as it would prompt him for input.

(2) A general purpose microassembler configured with special input for one of the bit-sliced chips (of various configuration), such as those manufactured by Advanced Micro Devices and Texas Instruments.

(3) A graphic output module which plots the item under study. The current availability of pinprinters ..ill make this feasable in the near future.

(4) A module to produce a parts list.

(5) A program to read an execution module from an assembler package and prepare it for Simcal entry.

(6) An expansion of the software allowing the user to save the status from one session to the next.

Applications of the Methodology to Other Areas

In addition to presenting a CHDL and its support system, the technique used to build the system has also been described. The method consists of several major steps, each subdivided into smaller steps. The method could also be applicable toward developing other Specialized Description Languages (SDL) where either programming or general purpose simulation languages are now used. The Encyclopedia of Computer Science, page 1265, (85) estimates that 75 % of all simulation is done in Fortran and most of the remainder is in GPSS, Simscript or Simula although eleven other general simulation languages are discussed.

Simulation is widely used to study various systems such as job shop manufacturing, weather, economics, and materials handling. The value of a specialized description language to study an operation has been clearly demonstrated in the case of the CHDL's. The CHDL's perform their specialized function better than either general purpose simulation or programming languages. The advantages of the SDL as a "tool" over the general purpose "tool" include:

(1) Closer Representation of the Model.

(2) Less Time to Prepare the Description.

(3) Broader Use During Simulation.

These advantages are offset by the investment in building the system and the time the user must spend in learning to use the system. We now examine the steps taken in the methodology, attempting to state the steps in a general language which would fit any attempt to build a specialized description language and it's software support system.

(1) Identify the basic units of the system under study.
(2) Develop the BNF statements to fit these units.
(3) Develop the BNF subunits' clauses.
(4) Prove the language using a grammar analyzer.
(5) Prepare and prove the syntax analyzer.
(6) Construct the semantic portion of the compiler.
(7) Prepare a tabular output from the compiler.
(8) Write a user interface language for the simulator.
(9) Write the table driven simulator.
(10) Prove the system and make changes as required.
(11) Prepare user documentation.
(12) "Sell" the system to a group of users.

We now examine the problems of using such concept to develop a specialized language. A manufacturing simulation system is chosen for an example because it has received significant attention for at least ten years and continues to receive attentions as interest in CAD/CAM (Computer Aided Design/ Computer Aided Manufacturing) continues. The ideas presented here are strictly superficial and in no way represent a study of the languages used for simulation of manufacturing processes. It is presented to illustrate that the methodology used to develop the Calsim/Simcal system could be used for other types of simulation and to draw the analogy between the CHDL, Calsim and an SDL in another area.

Let us consider a manufacturing system to be any set of facilities, capital, and personnel which can receive raw material, parts, and/or assemblies (input) and produce one or more end items (output). We will assume several systems may interact with each other or the system under description may be divided into lower level items until reaching areas such as "detail shop", "major assembly", "foundry", "Plant A" or other organizational segment. These may then be carried to lower and lower levels as desired. These descriptions are obviously analgous to the hierarchy statement used in Calsim and represent the organization of the manufacturing operation.

In a manufactuing system, the details, assemblies, supplies, and raw material must be moved from one location to another using one or more of numerous types of transportation systems. Such systems are functionally the same as connectors in an electical circuit. The manufacturing system, however, moves more than two types of articles through the connectors, whereas a CHDL has only "ones" and "zeros" to move; this hints at the greater complexity of a manufacturing system as compared to a computer system.

We must now provide a grammar to describe the transportation system. This might be done with a series of clauses including such items as "TYPE" (conveyor, flat-bed in-plant truck, special trailer, and dozens of others). We would probably wish to have other characteristics identified such as capacity, speed, loading and unloading time. When all of these significant parameters which affect the operation are identified, then appropriate clauses can be made a part of the grammar.

If our generalization is to hold, we should be able to find something in the manufacturing world, which is similar to a register, and indeed we do. It is the backlog of work awaiting processing and the completed work awaiting shipment to the next step. More correctly the work-in-process is analgous to the "data", the storage area is analgous to a register or latch. So at the unit level, we must have some arrangement for storage. Like a register its size is most important. But in manufacturing, storage bins are more complex due to the variety of things to store.

One of the most important factors in the manufacturing system, as in a computer system is the operations to be performed. The operation may be milling, drilling, routing, assembling, painting, inspecting, etc. As with computers, only certain operations are performed on certain materials and the grammar/language must recognize this. Other items of interest are the special tooling needed, the skills required, personnel, and the supplies which are necessary.

Manufactuing will usually be carried out to prepared work instructions. This will include manufacturing planning ..hich calls out the operations needed to make the end item, usually specifying the special tools needed, the material needed, the sequence of steps to follow, and the location where the operations are to be done. It is easy to see that the manufacturing work instructions are analgous to a computer program to be run in a computer system.

The analogy to a microprogram is found within the manufacturing unit itself -- the detail procedures for specific operations. The specific operator instructions of various types including those which the operator has memorized, machine instructions, and detailed methods of performing skills such as soldering, welding, or plating.

The analogy will not be carried any farther. It can be seen there are many analgous situations and also that the manufacturing system is actually far more complex than a computer system. In closing the comparison, some of the possible items of interest in a manufacturing simulation system are shown. In each of these cases the study may be applied to future systems, a major change in a present system, or problem solving in a current system.

(1) Optimization of work-in-process inventory.

(2) Trade off studies of various combinations of machines, manning, work-in-process.

(3) Various approaches to materials transportation movement, and storage.

(4) Effect of various rejection rates on schedule, cost, and required work-in-process.

(5) Optimization of inspection effort.

(6) Determination of critical points affected by an increased schedule rate.

(7) Determination of the proper time to reduce manning level given a reduced schedule.

(8) Effect of worn equipment and increased maintenance on cost, schedule, and rejection rate.

(9) Optimization of maintainance expenditures.

(10) The effect bargaining unit rules on productivity.

(11) Determination of return on investment (ROI) under various conditions.

(12) Effect of change in interest rate on planned facilitization.

The similarity of the trial manufacturing grammar in appendix 5 to that of Calsim in appendix 3 is easily seen. The system name, the descriptive hierarchy, the several types of statements and the clauses which further describe the portion being studied all have similar structure.

Summary

A hardware description language has been presented specifically designed for use by students in the first course in hardware organization and in several other course including those using microprogramming. The grammar is sufficiently English like to make the language easy to learn and use. The system lends itself to the study of micro-programming and to hierarchic design studies.

In addition to presenting "Yet another" Computer Hardware Description Language, a methodology for developing the language and simulator has been presented. Many of the ideas, grammar, techniques, methods used in the language it-self have been used by others. The best from several of the languages have been combined, reorganized and rearranged to give a unique language. The methodology used, however, as far as can be determined has not been reported before. The application of the methodolgy to other areas of simulation may be of greater significance than the language itself.

APPENDIX

APPENDIX 1

CHARACTERISTICS OF SIX CHDL'S VERSUS CALSIM

LANGUAGE:	CDL(COMPUTER DESIGN LANGUAGE)	DDL(DIGITAL DESIGN LANGUAGE)	SDL(SYSTEM DESIGN LANGUAGE)	CALSIM(COMPUT. ARCHITECTURE LANGUAGE FOR SIMULATION)	ISPS(INSTRUCTION SET PROCESSOR SYSTEM)	PMS(PROCESSORS MEMORIES, SWITCHES)	AHPL (A HARDWARE PROGRAMMING LANGUAGE)
REFERENCES	(24),(12),(79) (27),(26),(33)	(36),(35),(31) (26)	(89)	---	(90),(96),(13) (77),(26)	(91),(96),(26) (34)	(52),(54),(51)
CHARACTERISTICS:							
WRITTEN BY/DATE	Chu 1965	Duley 1967	vanCleemput 1977	Skelton 1982	Bell and Newall 1971	Bell and Newall 1971	Peterson and Hill 1973
PRIMARY PURPOSE	Funcional algorithms and sequential processes	Alleviate problems in design documentation.	Hierarchical use over multi-levels.	Microprogramming in educational environment.	Describe the programming level of a computer.	Describe the physical level of a computer.	To translate design into hardware via a compiler using APL.
PROCEEDURAL?	NO	NO	NO	NO	YES	NO	YES
NET CONNECTIONS	NO	YES	YES	YES	YES	YES	YES
LNG. LEVELS(1)							
PMS	---	---	secondary	secondary	secondary	primary	---
PROGRAMMING	secondary	secondary	secondary	primary	primary	secondary	secondary
REG. TRANSFER	primary	primary	primary	secondary	secondary	---	secondary
SWITCH CIRCUIT	secondary	primary	primary	---	---	---	primary
CIRCUIT LEVEL	secondary	tertiary	primary	---	---	---	---
REGISTERS:							
DIMENSIONS	2	multiple	2	2	n/a	n/a	multiple
SUB-REGISTERS?	YES	YES	YES	YES	YES	n/a	NO
CASCADED?	YES	YES	YES	YES	YES	YES	NO
DATA TYPES (2)	BINARY	BINARY	BINARY	BINARY	GENERAL	n/a	BINARY
MEMORYLESS CPRS	NO	YES	YES	NO	--	--	--
SPECIAL REGS(3)	YES	YES	NO	NO	YES	n/a	NO
COMPONENT STMNT:							
USE OF PINS(4)	NO	NO	YES	YES	NO	NO	YES
TIMING (5)	clock	clock	clock	**event	clock	n/a	clock
EQUIVALENCE(6)	NO	NO	YES	NO	NO	NO	NO
MEMORY HANDLING:							
PRIMARY	as registers	as registers	as registers	**read from file	as registers	n/a	as registers
MICRO-MEMORY	no provision	no provision	no provision	**read from file	no provision	n/a	no provision
OTHER-MEMORY	no provision	no provision	no provision	**read from file	no provision	n/a	no provision
SPECIAL SYNTAX	NO	NO	NO	**YES	NO	NO	NO
PERIPHERAL TRTMNT:							
DATA CONVERSION	NONE	NONE	NONE	**YES	NONE	NONE	NONE
I-O DATA FILES	NO	NO	NO	**YES	NO	NO	NO
SPECIAL SYNTAX	NO	NO	NO	**YES	NO	NO	NO

CHARACTERISTICS OF SIX CHDL'S VERSUS CALSIM (CONTINUED)

	CDL	DDL	SDL	CALSIM	ISPS	PMS	AHPL
NUMR OPERARTORS:(7)							
ARITHMETIC	2	4	Accurate	4	4	n/a	4
LOGICAL	3	3	information	5	5	n/a	5
RELATIONALL	2	5	not	6	6	n/a	6
MONADIC	2	10	available.	4	2	n/a	2
OTHER	4	9		0	3	n/a	2
SEQUENCING: (8)	entered order, call, jump	entered order, call, jump reset timing	entered order call, jump reset timng	**by assigned timing, reset entered order	enterd order	n/a n/a n/a	branching, reset timing
MACRO/LIBRARY:							
TIME ADJUSTED?	NONE	NO	NO	**YES(9)	NO	n/a	NO
TIMING:							
CONCURRENCY	NO	YES	YES	YES	YES	n/a	YES
INTERUPTS	NO	YES	YES	**YES(10)	NO	n/a	YES
TIME DELAY	NO	YES	YES	YES	NO	n/a	YES

(1) Levels are based on the five levels used by Barbacci in (14) -- PMS (processor, memory, switch), programming, register transfer, switch circuit, and circuit level. The degree to which the language is usable at various levels has been evaluated, and values of primary, secondary and tertiary have been given those rated fully usable, partly usable, and barely usable.

(2) Data types, if used, are the same as usually found in programming languages, otherwise only ones and zeroes are used. The bits may also have values of "open", "high impedance", and "unknown" in the CALSIM language.

(3) Where special registers such as SP, IR, MAR, and ACC are used, then part of the machine architecture is implied.

(4) As used here, PINS refers to the use of the manufacturers pin number and names as part of the hardware description and is used to make connections to the outside world.

(5) Clock refers to lapsed time as found in CDL, event refers to a table of events.

(6) EQUIVALENCE refers to the syntax necessary to specify that connections or groups of connections are interchangeable.

(7) The arithmetic operators are +, -, /, and *.
The logical operators are AND, OR, NOT and in some cases NAND, NOR.
The relational operators are <, >, =, ~=, ~<, and ~>.
The monadic operators are SHIFT LEFT & RIGHT, ROTATE LEFT AND RIGHT, COMPLEMENT, and NEGATE. Various rules are used in filling the bit vacated in the shift operation.
OTHER includes COUNTUP, COUNTDOWN, Concatenation.

(8) CALSIM also permits time event bracketing entered through the simulator.

(9) Calsim allows times within the library description to be adjusted as they are copied into the active hardware description.

(10) The interrupt in Calsim is handled through the simulator driver.

** Distinctive differences between CALSIM and other languages are shown with a double asterik.

APPENDIX 2

SUMMARY OF COMPUTER HARDWARE DESCRIPTION LANGUAGES

EXPLANATION OF APPENDIX. The brief explanations of the CHDL'S presented here attempt to emphasize the same aspects which the author of the articles referenced emphasized. In most cases there is a "PURPOSE", which states the reason the originator had in creating another CHDL. If this is omitted it is because there was no clear indication of the purpose of the language over those already available. When the "DISCUSSION" is omitted it is because there was insufficient information. See page 15, "An overview of CHDL's" for a discussion of these entries.

ADLIB (A Design Language for Indicating Behavior) by Dwight D. Hill, 1979, (52).

PURPOSE: To describe the timing and behavior of several interconnected computer components, which are then combined and "connected" by a simulator called SABLE (Structure and Behavior Linking Environment) and used in conjunction with SDL (Structure Design Language).

DISCUSSION: A behavior language using a superset of PASCAL with six special types of statements added to the set. Used in combination with SDL, SABLE, and SUDS2 (A graphical structure editor). The special types are:

```
(1) ASSIGN <EXPR> TO <NET NAME> <TIMING CLAUSE>
(2) WAITFOR <BOOLEAN EXPR> <CONTROL CLAUSE>
(3) SENSITIZE; DESENSITIZE; DETACH
(4) UPON <BOOLEAN EXPR> <CHECK LIST> DO <START>
(5) TRANSMIT <EXPR> TO <NET> <TIMING CLAUSE>
(6) INHIBIT and PERMIT
```

ADL (An Architectural Description Language) by C. K. C. Leung, 1979, (67).

PURPOSE: To complement many existing CHDL's in the area of packet switching communication system descriptions.

DISCUSSION: Designed for use in documentation, design, and language interface in a design automation system. "A packet communication system consists of hardware modules which communicate only by sending information packets to each other. ADL provides a language to describe these interconnections". Examples of ADL follow.

```
TYPE ALP = MODULE
  INLET OPN-IN: OPN-PKT;
  OUTLET RES-OUT: RESULT-PKT;
  PAREM N-OF-ALU: INTEGER;
  SUBMOD
  K: CONTROLLER(N-OF-ALU)
  INLET OPN-IN: OPN-PKT;
     ALU-IN[1...N-OF-ALU] ALU-RES;
  OUTLET RES-OUT: RESULT PKT;
     ALU-OUT[...M-OF-ALU]: ALU-OPN;
END
```

AHPL I, II, III (A Hardware Programming Language), by F. J. Hill and G. R. Peterson, 1973, (53), (55), (54), (26).

PURPOSE: Extends the syntax of APL to include parallel and asynchronous operations. Specifically recognizes the control section and register section of the CPU.

DISCUSSION: Developed at the University of Arizona primarily to assist in the synthesis of electrical circuits from hardware design. The language follows the same philosophy as APL with additional features of the language allowing a rapid register transfer description to be accomplished. This is one of the most widely used CHDL's.

APDL (Algorithmic Processor Design Language) by John A. Darringer, 1968, (31), (96).

PURPOSE: To describe the behavior of digital, synchronous systems and to "Enable the designer to conveniently describe any part of a processor as an algorithm".

DISCUSSION: An extension of Algol which includes register data types, register computations, time blocks for delay, parallel operations. It can be used at several levels down to the gate level. "APDL has been used for several semesters at Carnegie-Mellon University for both instruction and course projects and proved to be convenient notation for describing existing hardware". Examples of the language follow.

```
BINARY REGISTER ACC<0:3>
BINARY REGISTER ARRAY MEMORY [1:NOPAGES,1:128] <1:31>;
BINARY REGISTER PROCEDURE SHIFTON (A) <1:31>;
VALUE A; INTEGER A;
3TIME BEGIN
 AM <- AM + XM
 OV <- OVERFLOW
 END
IF EVER MB = CORE (N) THEN 3TIME BEGIN
END
```

APL (A Programming Language) K. E. Iverson, 1962, (96), (45).

PURPOSE: To provide a programming language which will Address operators in a uniform way and be capable of describing the hardware and software logic equally well.

DISCUSSION: The language is interactive and uses an interpreter to process one line at a time. Processing is from right to left. There is an extensive set of operators, virtually all of those used in mathematics, but many are unsuitable for hardware descriptions. A special keyboard and printer is used which redefines the upper case letters; the lower case letters are redefined as capitals. The ability to reach the bit level and to manipulate values allows the full set of functions in a CPU to be compressed into a few hundred lines. In simulation, a single line of code will usually simulate a single machine instruction. This language is broadly used and widely available as a time sharing system. It has not been found entirely suitable for hardware descriptions due to its deficiencies in timing and sequencing control.

APL*DS (A Programming Language for Design and Simulation), W. R. Franta and W. K. Giloi, 1975, (41).

PURPOSE: To provide an interactive, top down, stepwise, refinement system which can address real-world problems, "not just the limited problems for which CHDL's are usually suitable".

DISCUSSION: A procedural language and system somewhat like APL. The statements include: transfer, declaration, branch, process directive, assignment, and indexing. The language uses, as do many CHDL's, the convention that each program line is a time segment. The system uses a preprocessor which translates the description into APL which then simulates the machine action.

ASM (Algorithmic State Language) by C. R. Clare, 1972, (33).

DISCUSSION: Developed and used by Hewlett-Packard in Great Britian. An ASM description is a sequential machine realizing a particular algorithm. The operations of the target system are expanded on a flow chart and the description is realized as a sequential machine.

CALSIM (Computer Architectural Language for Simulation), by W. A. Skelton, 1981, (92).

PURPOSE: To directly address the problem of description and simulation of microprogrammable bit-sliced hardware in an English like interactive CHDL suitable for Computer Science education.

DISCUSSION: A hierarchical, non-procedural language implemented on the DEC-20 suitable for register-transfer, programming, and system level studies. The LSI chips and buses are considered the elementary units and are used to make up the larger aggegate of components. The language and simulator address timing and concurrency by use of an eight level event clock. See the Appendix of the User's Manual for examples of the language.

CASD, (Computer Aided System Design), by E.D. Crocket, et.al., Reported under development in 1969, (30), (96).

PURPOSE: A complete set of software packages to assist computer designers.

DISCUSSION: Developed at IBM facilities at Los Gatos, Ca., and based on PL/I with several additions and deletions to the language. The user specifies the register, sub-register and memories. The flow of control is implicitly defined by the order of instructions. The user has a choice of five major facilities: (1) the CASD language compiler, (2) the simulator, (3) detail logic generator, (4) on-line changes in conversational mode, and (5) documentation. "The basic unit for describing what is to be done to the data items is the expression, defined as in PL/I". "The basic statement types for describing action on data items are ASSIGNMENT, WAIT, CALL, GO TO, IF, DO, and RETURN".

CASL (Computer Architecture Specification Language), by G. F. Maxey, 1979, (74).

PURPOSE: To assist design architects in experimenting with new designs "As easily as programming language designers now experiment with language functions" by automating system design.

DISCUSSION: A register transfer language for system architects allowing decomposition of a machine into co-operating asynchronous modules, each of which contains "Abstractions", "Structural" and "Procedural" sections. The abstraction section may be used to define the data representation and primitive operators. The structural sections describe the components and the connections between them. An example follows.

```
MODULE TYPE (BLACKJACK).
ABSTRACTION:
     DATA REPRESENTATION:
          USERBIN: ARITHMETIC(BASE(2), UNSIGNED).
          STANDARD ARITHMETIC: USBIN.
     SYMBOL DEFINITION:
          WORD 5 BITS.
          MINUS-TERM: "22" USBIN.
     STRUCTURE:
          .
     PROCEDURE:
          .
     TRANSFER-VALUE:
```

CASS (Computer Aided Schematic System) by H. M. Bayegan, under development in 1979, (16), (15).

PURPOSE: To provide a design automation system which will operate at different levels.

DISCUSSION: Cass was developed at the Central Institute for Research at Oslo, Norway as part of a complete design automation system. The language provides a set of primitives which include capability to perform transfer, indicate timing, set up "IF" and "WHEN" perform register operations etc. These can be used to combine elements into complete working systems. The output not only includes simulation but also circuit generation. An example in (16) uses a UART to demonstrate a topdown structured approach.

CASSANDRE (Computer Aided Design and Simulation of Logical Systems) by F. Lustman and Jean Mermet, 1968 (19), (15), (97).

PURPOSE: Precise hardware equivalent at the semantic level, provides better parallelism, synchronous and asynchronous simulation and delays.

DISCUSSION: Based on the works of Chu, Schlaeppi, Iverson, and the language EPICURE. The language uses a basic feature called a "unit" which represents either a piece of hardware or an arbitrarily defined group of components. "A Cassandre description is a set of trees of units. Each unit with the set of units it contains is a complete Cassandre description. It can be compiled, run, . . . in synchronous or asynchronous mode".

CDL (Computer Design Language), Yaohan Chu, 1964, (24), (12), (80), (27), (26), (33) (49).

PURPOSE: One of the first CHDL's. Originally proposed as a means of communication and as a means to precisely define the design description with respect to functional organization, algorithms, and sequential operation.

DISCUSSION: A non-procedural language containing several statements to describe registers, sub-registers, arrays, cache-registers memories and other components. Start conditions are provided. Bara and Born (12) describe a compiler/simulator developed in 1967 for CDL. Many other CHDL's have used CDL as the starting point. An example of the language follows.

```
REGISTER R(0-23), F(0-5), A(0-23)
SUBREGISTER R(OP) = R(05), R(1) = R(6)
MEMORY  M(C) = M(0 - 32767, 0-23)
DECODER K (0-9) = F
SWITCH POWER (ON)
TERMINAL ADD = K(0)
         SUBTRACT = K(1)
IF G=0 THEN F( <- 10)
```

CSL (Computer Structure Language) by David R. Smith, 1975, (93).

PURPOSE: To overcome the shortcomings of other register transfer language (e.g. CDL) such as "Labels, (unreadability, error proneness, unnecessary wordiness), inadequate sub-routine features, insufficient input/output features and compiler speeds.

DISCUSSION: A compromise between nonprocedural and procedural language is proposed using a "Task format". The language was developed at SUNY at Stony Brook for use in education. This extension of CDL supports parallelism more readily than CDL itself. Reference (12) describes an extensive set-up at Michigan Technical University. The system there includes handling of encoders, buses, partitions, and I/O flags. Includes six logical operators but does not handle parallel functions. Examples follow.

```
FLIP FLOP : A, B, C; REGISTER H(-1:7), I(15:0);
SUBREGISTER : M <=> H(1), N(1:3) <=> (L14:12);
TERMINAL : <-> A&B; SEARCH : X, MASK := Y
   MATCHES = Z/;
```

DCDS (Digital Control Design System) by H. Potash, 1969, (96), (83).

DISCUSSION: A language for the simulation of digital structure developed at UCLA to assist the designer of computer systems.

DDL (Digital Design Language) by J. R. Duley, 1967, (36), (35), (31), (26).

PURPOSE: To assist the designer of logical computers in the area of system design, logic design, Boolean equations, and documentation of these activities. This is accomplished by alleviating "Time-consuming and error-prone problems encountered in design documentation both as successive stages are completed and at the interface between groups.".

DISCUSSION: A block orientated language containing statements for clocks, comments, wires, elements, delays, memory and registers, The existence of logic to handle complex data types is implied by the operations performed without representing how this logic is implemented. Bit numbering may ascend or descend. A subset has been implemented at Carlton University (Canada) by Professor Bowen and others. Darringer states "The language reflects the viewpoint that a digital computer is a set of automata controlling the flow of data among registers, memories, and interfaces. Unfortunately the designer must show this viewpoint and is required to translate his algorithm into state diagram form. He then specifies the operations that occur in each state".

DIDL (DIGITAL INTEGRATED DESIGN LANGUAGE), by A. M. Despain, 1975, (34).

DISCUSSION: DIDL has "Combined the best features of Algol, DDL, CDL, APL, ISP and other programming and design languages". The language was used by Despain in combination with PMS in the design of a Fourier transform system (93).

DIGITEST II(Digital Testing) by F.J. Ramig, 1975 (84).

PURPOSE: To overcome four "Well known" deficiencies in current CHDL's - structural description, asynchronism and parallelism, "Oversimplified data structure", and widely varying syntax and language from the languages ancestors.

DISCUSSION: Based on DIGITEST, to describe structure, PL/I to describe behavior and Petri-nets to describe complex control structure. The language uses three data types and PL/I declarative statements. The principle of Petri-nets is used as the theoretical approach to solve parallelism. Petri-nets consists of a set of places containing a set of tokens, a set of transitions with firing rules which move tokens and directed edges. The presence or absence of the token provides permission to "Fire" or withhold firing. An example of the language follows.

```
DCL(A,B,C) BIT (-16)"1","(4)FFF",("(1)1(4)D")"1");
DECLARE NUMBER FIXED (4,UN)DEFINED (B POSITION(2));
DCL 1 INPUT(5) 2 (START1, START2) BIT(1) INIT(1);
ON CASE (D: (A,C),E:(B,C);
B: ON(A); CALL C
ON(A): IF (P) THEN B:
```

DSDL (?) BY J. L. Houle, 1974 (96) (56).
DISCUSSION: Adapted from DDL; written in XPL for the IBM-360.

ERES ("ERLANGER RECHNER ENTWURFS SPRACHE" = CHDL OF ERLANGER) by P. P. Spies, M. Becker, and R. Klar, 1972, (43), (80), (61).
PURPOSE: Extends CDL by providing a unique way of describing functional and temporal behavior of storage elements and extends timing concepts to include execution time of micro-operations.
DISCUSSION. A non-procedural, register-transfer language using statements to describe structure of the architecture functional behavior, and timing. The registers and memories are regarded as primitives, the timing is synchronous with local embedded asynchronous micro-operations. The language uses 0, 1, and "U" for values in the circuits with the clock having values of zero or one. The language supports both "active" and "passive" carriers. According to Piloty, ERES has been used successfully as a teaching tool to describe microprogramming methods.

EPICURE -- See Casandre.

FLOWWARE by Shingfat S. Chin, 1977, (23).
PURPOSE: To overcome the tediousness of converting a pictorial description of registers into a written description.
DISCUSSION: The language was developed at the University of Missouri-Rolla for student use. A Techtronix graphics terminal is used to describe the proposed design followed by the invocation of Flowware which converts the graphic description into character form. The compilation and simulation is also carried out at the graphics terminal. The language is be used in conjuction with other support software.

FST (Functional Simulator and Translator) by E. A. Frank, 1967, (96).

DISCUSSION: Written in Fortran for PhD thesis for use on IBM-360 at the Case Western Reserve University (originally called CADSS). The system consists of three programs - a translator, a simulator and a design generator. "The source language allows specification of logical sequences of operation without explicit specification of the control logic. ... in either sequential or concurrent blocks, ... the system has been used for instruction in logic design course at Texas A & I, as well as for several hardware designs".

G (for Directed Graph and Data Graph), by D. Bain and P. J. O'Callaghan, 1976-1978, (99), (39).
PURPOSE: To address parallelism more realistically through syntax based on Petri-net concepts.
DISCUSSION: The three basic concepts of LOGOS are retained, namely, separation of the control function, the use of directed graph notation concepts, and a hierarchical description. In using G language, the designer draws the necessary data and control graphs, then converts them to G for simulation.

GLIDE (Generalized Language for Interactive Description), by Alice Parker, 1975, (77).
PURPOSE: To provide a language which addresses the problem of interface between components.
DISCUSSION: The language is built around a group of primitives which form stand alone "processes". Each process is enclosed by "BEGIN" and "END" between which the process and its hardware are described. The primitives include CONTROL, SYNCHRONIZATION, PRIORITY ALLOCATION, STATISTICS, TECHNICAL PROBLEM, BUFFERING, ERROR CHECKING, and FORMATTING.

HARD (Hardware Simulation in Education), Ivan Tomek, 1981, (99).
PURPOSE: Developed to replace simple laboratory experiments usually performed in the first course in computer organization. It was found that existing languages were either not suitably orientated toward the student level or were unavailable due to either cost or portability.
DISCUSSION: The Language was developed at Acadia University, Wolfville, Nova Scotia to assist instruction in a first course on computer organization. Thus it is specifically adapted toward showing circuits such as gates, flip-flops, binary adders and other devices introduced at that

level. The following example shows several lines of language used to describe one of the circuits in the course.

```
CIRCUIT: ADDER 2
ADDER: A1, A2 :MODULE OF TYPE ADDER CALLED A1, A2
PARTS: INPUT A(2), B(2)
       OUTPUT SUM(2), COUT
END
CONNECT: 0 TO: A1, CIN etc
```

HDL(Hardware Description Language) by W. A. Johnson, Jane Crowley and J. D. Ray, under development in 1980, (59).

PURPOSE: To provide a language for mixed level simulation suitable for assistance in design of VLSI components.

DISCUSSION: This language and support system were reported as under development at Texas Instrument (Dallas) in 1980 and is intended for use in VLSI designs. The language is used in conjunction with INTSIM, a mixed level simulator, SIMCL, a simulator control language and CSL, a circuit selection program.

HILO (for HIGH LEVEL, LOW LEVEL), by P. L. Flake and G. Musgrove, 1974, (38), (33).

PURPOSE: To provide a language which is capable of describing the two levels usually used by the engineer -- high levels which ignores propagation delays, wave forms, etc and includes the full description of the hardware; and low level which takes these things into consideration.

DISCUSSION: The system uses two languages, one for control using a procedural type and a second for data description, register and IC packages. An example follows.

```
PROC  CONTROL=(REFROC ENABLEAB, CLEARAB, LOADAB,
      ADD, ENABLEC1, READY,
      SIGNAL COUNT15, LSBB, START)
BEGIN A; READY IF START THEN CLEARAB, GO TO B
         ELSE GO TO A;
```

ISP and ISPS (INSTRUCTION SET PROCESSOR) BY C. G. Bell and A. Newell, 1974, (90), (96), (13), (78), (26).

PURPOSE: According to Siewiorek, the language was developed to "precisely describe the programming level of a computer."

DISCUSSION: A Register-transfer, procedural language written in Bliss, adapted from Algol, and implemented on the PDP-10. The Language has been used in teaching, to compare machines via benchmarks, and as the descriptive language for a design automation package. The language is primarily a series of conditional/assignment statements, but does provide for concurrent operation. A sample of the language follows:

```
PC STATE
R[0:15] <0:31>          GENERAL REGISTER
RO := R[0]              REGISTER 0
PC PANEL
SS<1:4>                 SENSE SWITCH
DATA
COMPUTE                 PANEL SWITCH
INSTRUCTION FORMAT
I<0:31>                 INSTRUCTION
IB := I<0>              IDENTIFIER BIT
DATA TYPES
BYTE := 8 BITSS
FUNCTIONS
BA := (70 <= OP <= 75)  BYTE ADDRESS
```

LALSD (A Language for Automated Logic and System Design), by S. Y. H. Su and M. B. Baray, 1971, (97).

PURPOSE: "Suitable for describing documentation, simulation and synthesizing digital systems".

DISCUSSION: Implemented on the IBM-360 using PL/I as the host language. The language provides a clear separation of system behavior and system structure and language has been found suitable for study of operation system problems such as deadlocks and determinancy, without exhaustive simulation. It provides the means to decompose the description to the level desired by the use of parallel control operations. Synchronous and asynchronous operations may be mixed.

LASCAR (A Language for Simulation of Complex Architecture) by Dominique Borrione, 1975 (18).

PURPOSE: To extend Cassandre so that architecture using a hundred or more microprocessors running in parallel can be studied.

DISCUSSION: A procedural, register transfer language developed at the Compaignie Internationale pour l'informatiques and Ecole Nationale Superieure d'Informatique et de Mathematiques Appliquees de Grenoble. The Lascar extension includes two new types of variables - "integer" and "counter", two new conversion operators to go between Boolean and Integer, and an assignment symbol for integer and subroutine calls.

LCD (A Language for Computer Design) by C. J. Evangelisti, G. Goertzel and H. Ofek, 1977, (4), (37).

PURPOSE: To help an engineer develop the control for a clocked digital machine given a data flow and a behavior specification of the machine.

DISCUSSION: Specifically planned to assist in design of logic of the control section of the CPU. "Both the data flow and the specification are described in LCD". The language works as part of a larger system which includes a specification description and simulator.

LDT (Logic Design Translator), by D. F. Gorman and J. P. Anderson, 1962, (96).

DISCUSSION: A register transfer language implemented on the Burroughs machine using Algol58 as the host language.

LOGAL (Logic Algorithmic Language), by John Lund, 1973, (94), (96).

PURPOSE: To reduce the required development time for a new computer design by use of "Rapid top-down design iterations" and providing "Feedback to the designer early in the process."

DISCUSSION: A register-transfer language adapted from RTL, used with LADS (Logical algorithmic design system). Both a simulator and a hardware generator have been written in Fortran for use on the Univac 1108. An example follows.

```
>CMT    ***** 4 BIT BI-DIRECTIONAL JOHNSON COUNTER
>PLC    JOHNSON 00-03; COUNT DIRECTION TO CRD1;
>CMT    TOGGLE COUNT DIRECTION WHEN COUNTERS VALUE 0.
>CMD 1  NOT COUNT DIRECTION TO COUNT DIRECTION IF JOHNSON
        00-03 = 0#4
>IFB    COUNT DIRECTION
>CMD 2  JOHNSON 00-03 >PRT2> JOHNSON 4 BIT DN COUNTER
>CMD 2  (NOT JOHNSON 03) & JOHNSON 00-02 TO JOHNSON 00-03
>IFB    NOT COUNT DIRECTION
>CMD 2  JOHNSON 00-03 >PRT2> JOHNSON 4 BIT UP COUNTER
>CMD 2  JOHNSON 01-03 & (NOT JOHNSON 00) TO JOHNSON 00-03
```

LOGOS (?) by E. L. Glasser, 1969, (39), (33).

PURPOSE: To provide better treatment of parallelism in the simulated model.

DISCUSSION: A hierarchical, directed graph, language originated at Case Western University loosely related to Petri-nets. The rationale of Petri-nets is used to accomplish parallelism. Patrick Foulk at Heriot-Watt University, Edinburgh has developed a translator for the language.

LOTIS (Language for Describing Logic, Timing, and Sequencing) by Schaeppi, 1964 (87), (96), (31).

PURPOSE: "Intended for formally describing the logical structure, the sequencing, and the timing of digital machines".

DISCUSSION: A hierarchical, register-transfer language dating to the early sixties in which every linguistic constant corresponds to a unique machine element. Timing can be specified as synchonous or asynchonous or any combination of the two. Concurrency, time sharing and interlocks can be described. The language allows an arbitrary number of levels to be described. The buses are handled as "transients". Schlaeppi proposes syntactic checkers, and circuit synthesizers but does not discuss implementation.

MANO's RTL (Register Transfer Language), H. Morris Mano, 1976, (68), See RTL also.

PURPOSE: To improve Chu's CDL for use in education.

DISCUSSION: The paper by Lewis describes the effort at the University of Santa Clara to develop a compiler based on Manos's RTL and the effort in progress in 1979 to develop a simulator.

MDL (Modular Design Language) by Jack Lipovski, 1973, (96), (70).

PURPOSE: To provide a standard way to describe memory and link variables.

DISCUSSION: The language is used to describe IC's or a microprocessor using an event as the time from one fetch to the next. APL has been used as the host language.

MODAL (?) BY G. R. Hellestrand and J. W. Makepeace, 1976, (50).

PURPOSE: The system, including the MODAL language was developed so that a better teaching tool would be available. The language and system places emphasis on being able to define the system without pre-determined constraints and upon a friendly interactive environment.

DISCUSSION: A hierarchical, register-transfer, discrete event, concurrent, block-structured CHDL. It has been used for teaching for several years at the University of New South Wales and is part of "A large and complex multi-faceted project incorporating a CHDL, simulator and monitor". The monitor working environment is described as "APL like" including the means to store descriptions, work in an interactive environment, step through simulation etc. The system has been used to describe parts of the AMD-2900 bit-slice hardware components.

MODEL/LINDA (MODEL describes the hardware, LINDA describes the circuit behavior) by Irwin Lewis and Arnold Peskin, 1975, (69).

PURPOSE: Designed toward overcoming shortcoming of other design automatic systems, namely -- "usage complexity, internal compatability among tasks, suitability for new technology, simulator fidelity, extensibility, portability".

DISCUSSION: The language is statement oriented and relatively close to a restricted form of a natural language (English), "Where the operating symbols are evokative of the designers jargon for those operations". All operations are carried at the gate level (macros are available) since "Register transfer may gloss over subtle design problems". The statement types include DEVICE, CONNECTOR, MONITORING and ACTION. The system uses an event based table driven algorithm. Some of the statement forms are shown below.

```
DEVICE DEFINITION STATEMENT
     NAME(I,J) /TYPE/INPUTS/ OUTPUTS/DELAY
     type may be JK, RS, AND, OR, INV, NAND, NOR or ROM
CONNECTION STATEMENT
     FROMTO/ NAME, OUTPUTNO/ NAME INPUTIO
     BUS/ NAME(I,J) OUTPUT NO/ NAME (M, N)
MONITORING STATEMENT  F /N= GA
SCOPE /OUTPUT LIST
SNAP /OUTPUT LIST / TIME
CONTROL STATEMENT
    END /FUNCTION
    END /MACRO
ACTION STATEMENT
    START
STOP
```

OSM (an acronym for "JEZYK OPISU STRUKTUR MICROPROGRAMOWANYCH", Polish for "A Language for Describing Microprogrammed Structure"), by P. W. Marczyzinski, 1974, (72).

PURPOSE: Designed for describing a microprogrammed computer at the register level.

DISCUSSION: OSM is a "quasi-hierarchical language in which the most extended description contains a clock, non-sequential, and sequential structures". The sequential structure contains multicoders, decoders, function units, and a control graph which defines precedence of activity. The non-procedural part contains "global decoders" which are activated when distinguished registers change. This stops other actions until the global actions are completed. OSM has been implemented through a software support system called 3SM (Structure Simulation System for Microprogramming).

PHPL (Parallel Hardware Processing Language) by H. Analuff and P. Funk, 1979, (7).

PURPOSE: To provide greater realism in simulating real-time behavior by more closely addressing the handling of signal changes.

DISCUSSION: A Hierarchical, procedural, register-transfer language using several unique features for handling timing, both synchronous and asynchronous. The clock can describe complete architecture as well as details down to flip-flops. The clock generator has three parameters -- phase, pulse width, and period. The language allows change to occur either as a rising or falling edge. There are six types of operators. Example of the language follow.

```
REG R(0:9,1:0), T(1:0)
ACCESS REG(ADR) =
    FOR ADR FROM 0 BY 1 TO 9 : R(ADR)
/ M / P <- REG(F)
TIME TSETUP = 17 NS
```

PMS (Processor, Memory, Switches) by C. G. Bell and A. Newall, 1971 (91), (96), (26), (34).

PURPOSE: To provide a notation to describe various parts of a computer and computer network so that a uniform nomenclature can be used.

DISCUSSION: The system uses seven basic components - memory, link, switch, transducer, data-operation, and processor. The components can be connected to make a computer with varying levels of detail. Components are themselves decomposable. The PMS system can be used to develop an understanding of a computer system. For example several systems can be combined into a larger system using the PMS language.

RTL (Register Transfer Language) by C. G. Reed, et.al. 1952 and modified by H. Schorr, 1964, (88), (31), (96). See also Mano's RTL.

PURPOSE: Circuit Synthesis using register transfer statements.

DISCUSSION: The language "Describes a synchronous digital computer as a set of conditional transfers of data among registers and control variables". It has been implemented on the CDC-1604 using Algol.

RTS I, II, AND III (Registertransferprache) by R. Piloty, 1968, (80), (81).

PURPOSE: Developed at Technische Hochschule Darmstadt for use in teaching Computer Science.

DISCUSSION: A register transfer, event oriented language in three versions - I, II, and III. It is "used to describe behavior and structure of digital systems in courses on switching circuits and computer organization". The software is written in Fortran and the language itself is "Algol-like."

SDL (System Description Language) by W. M. vancleemput 1977, (100).

PURPOSE: "A language is needed that can be used by the designer at all levels of the design process and that allows him to record accurately all the information pertinent to his design.".

DISCUSSION: The language includes the concepts of "accurate representation of structural information, usefulness over all levels", applicable to different purposes of the designer. The language is able to map higher-level primitives into lower level ones. Language examples follow.

```
NAME: SN7491;
PURPOSE: LOGSIM, CRTANALYSIS
LOGIC: GATE
TYPES NAMP, INV, RS;
EXT:  DATA,. ENABLE,. CLOCK
OUTPUTS: Q
NAND: G1
INV: G1, G2;
RS: FF1, FF2, FF4, FF5, FF6, FF7, FF8
NET1 = FROM (.DATA) TO (G1.IN1)
NET2 = FROM (.ENABLE) TO (G1, IN2)
.
.
```

```
END
```

SDL I & II (System Design Language), E. P. Stabler, 1970. See ADLIB for discussion.

SFD-ALGOL (System Function Description) by D. L. Parnas, 1966, (31).

PURPOSE: Intended to accept any algorithm as a behavioral description of a digital computer.

DISCUSSION: The system accepts the algorithmic description and converts it to a state table. The difficulty is that any real computer has far too many states to be contained in finite space. "e.g. a 32 bit word has four (4) billion states".

SIMBOL and SIMBOL2 (?) by J. W. Williams and R. W. McGuffin, 1979, (105), (33).

PURPOSE: To improve the ability to verify that final design is equivalent to the original design concept.

DISCUSSION: Developed at the International Computer Ltd., Manchester as part of overall system to assist in the development of mainframe computers. It is used in conjuction with an existing design automation system. Each SIMBOL2 description specifies how to simulate the element as well as a description of the element. Each description includes an identifier, input/output, memories, delay and logic. An example follows.

```
'SPEC'
   'INPUTS' (W, H(4), H(32), H(32));
   'OUTPUTS' (W, H(32));
   'MEMORIES' (W);
   'ELTYPE' "MILL"
'ESR'
   'DELAY' D = (2, 5);
   'IF' 'BOOLVAL' 'THEN'
         'CASE' ++ 'ABS' 'INPUT' 2 'IN'
          'C' 0000 1 'BECOMES' WO 'AFTER' D;
```

SL-1 (Structural Modelling Language) by R. Gardner, 1975, (44).

PURPOSE: To provide a language which focuses attention on the Structure of a system, leaving behavior description for other techniques.

DISCUSSION: The language uses three standard parts to define the system -- modules, sockets, and interconnections. The connection descriptions permit directional, bidirectional and undirectional descriptions. The module represents one item in the system and the socket is the interface from the module to the system. The language accepts a hierarchical description.

SLIDE (Structural Language for Input/output), by Alice Parker and J. J. Wallace, 1979 (104), (77).

PURPOSE: To better describe input/output devices and interconnections.

DISCUSSION: A non-procedural language with parallelism and delay. The language is built around a "process" which is an independent executing environment, i.e. a piece of hardware which has timing, registers etc. The language is built upon GLIDE which is also listed in this table.

SSM (Simulation Language for Switching Circuits, using Multivariant Edges), by W. Goerke and H. J. Hoffman, 1974, (47).

PURPOSE: An extension of CDL providing gate delays, definition of new type components, a check for stable states, and speed up of execution on the Burrough 6700.

DISCUSSION: A comparison with CDL shows that SSM (on B-6700) is about four times as fast, and SSM is more convenient because of abbreviated expression. However memory and subregisters are difficult to define in SSM. The SSM simulator processes delays and asynchronous transactions much better than CDL and provides capability for detection of race and hazard conditions. Simulation and printing are both better controlled in SSM, but SSM requires more memory space.

VDL (Vienna Definition Language) by P. Wegner, 1972, (66), (79), (106).

APPENDIX 3

THE FORMAL GRAMMAR OF CALSIM

CALSIM GRAMMAR VERSION 8-01-82.

```
 1   <GOAL> ::= <SYSTEMS> END OF DESCRIPTIONS .
 2   <SYSTEMS> ::= <SYSTEM>
 3               | <SYSTEMS> <SYSTEM>
 4   <SYSTEM> ::= <SYSTEM NAME><STATEMENTS><END OF STMNT>.
 5   <SYSTEM NAME> ::= SYSTEM NAME <ARE> <IDENTIFIER> .
 6   <ARE> ::= ARE
 7           | IS
 8           |
 9   <END OF STMNT> ::= END OF <IDENTIFIER>
10   <STATEMENTS> ::= <STATEMENTS> <STATEMENT>
11                  | <STATEMENT>
12   <STATEMENT> ::= <STORE STMNT> .
13                 | <COPY STMNT> .
14                 | <HIERARCHY STMNT> .
15                 | <CONNECTOR STMNT> .
16                 | <TIME STATEMENT> .
17                 | <COMPONENT STMNT> .
18                 | <MEMORY STMNT> .
19                 | <PERIPHERAL STMN> .
20                 | <WIRING STMNT> .
21                 | <START STMNT> .
22                 | .
23   <STORE STMNT>:=<STORE HEAD><STATEMENTS><END OF STMNT>
24   <STORE HEAD> ::= <STORE AS> <IDENTIFIER>
25   <STORE AS> ::= STORE AS
26   <COPY STMNT> ::= <COPY HEAD>
27                  | <COPY HEAD> <,> <TIME CLAUSE>
28   <COPY HEAD> ::= COPY <IDENTIFIER>
29                 | COPY <NUMBER> <EACH> <IDENTIFIER>
30                 | COPY <IDENTIFIER> <RENAME>
31                 | COPY <IDENTIFIER> <SUFFIXID>
32   <EACH> ::= EA
33            | EACH
34   <RENAME> ::= RENAME <IDENTIFIER>
35   <SUFFIXID> ::= SUFFIX <IDENTIFIER>
36   <,> ::= ,
37         | <EMPTY>
```

```
38   <TIME CLAUSES> ::= <TIME CLAUSE>
39                    | <TIME CLAUSES> <;> <TIME CLAUSE>
40   <TIME CLAUSE> :: = <TIME=> <TIME> <,> <TIME2-3>
41   <TIME=> ::= TIME =
42   <TIME> ::= <NMBORSTAR>
43            | <TIME> : <NMBORSTAR>
44   <NMBORSTAR> ::= <NUMBER>
45                 | *
46   <TIME2-3> ::= <TIME> <TIME INCR>
47               | <EMPTY>
48   <TIME INCR> ::= <,> <TIME>
49   <HIERARCHY STMNT> ::= <LEVEL WORD> <MAJ. ASSY LIST>
50   <LEVEL WORD> ::= LEVEL <NUMBER> :
51   <MAJ.ASSY LIST> ::= <HIERARCHY HEAD> <MAJ.ASSEMBLIES>
52                     | <MAJ. ASSY LIST> <;>
                         <HIERARCHY HEAD><MAJ.ASSEMBLIES>
53   <HIERARCHY HEAD> ::= <IDENTIFIER> CONTAINS
54   <MAJ. ASSEMBLIES> ::= <HARDWARE UNITS>
55                       | <MAJ. ASSEMBLIES> <,>
                           <HARDWARE UNITS>
56   <HARDWARE UNITS> ::= <IDENTIFIER>
57                      | <NUMBER> <EACH> <IDENTIFIER>
58   <;> ::= ;
59   <CONNECTOR STMNT> ::= <CONNECTOR HEAD> <ARE NUMBERED>
                           <ARE NAMED>
60   <CONNECTOR HEAD> ::= BACKPLANE : <IDENTIFIER>
61                      | CORD : <IDENTIFIER>
62                      | BUS : <IDENTIFIER>
63   <ARE NUMBERED> ::= <ARE> NUMBERED <NUMBER LIST>
64                    |
65   <NUMBER LIST> ::= <INTEGERS>
66                   | <NUMBER LIST> <,> <INTEGERS>
67   <INTEGERS> ::= <RANGE1> <RANGE2>
68                | <NUMBER>
69   <RANGE1> ::= <(> <NUMBER>
70   <RANGE2> ::= - <NUMBER> )
71   <(> ::= (
72   <ARE NAMED> ::= <ANDNAMED> <NAME LIST>
73                 |
74   <ANDNAMED> ::= AND NAMED
75                | NAMED
76   <NAME LIST> ::= <NAME LIST> <,> <IDENTIFIER>
77                 | <IDENTIFIER>
78   <TIME STATEMENT> ::= <LIMIT TIME> <TIME>
79   <LIMIT TIME> ::= LIMIT <TIME=>
80   <COMPONENT STMNT> ::= <COMPONENT HEAD><CMPNT CLAUSES>
81   <COMPONENT HEAD> ::= COMPONENT : <IDENTIFIER> <,>
82                      | CHIP : <IDENTIFIER> <,>
```

```
 83   <CMPNT CLAUSES> ::= <CMPNT CLAUSE>
 84                     | <CMPNT CLAUSES><;><CMPNT CLAUSE>
 85   <CMPNT CLAUSE> ::= <COMMON CLAUSES>
 86                    | <CMPNT IF CLAUSE>
 87                    | <ON PULSE> <CMPNT LOGIC>
 88   <COMMON CLAUSES> ::= <PIN CLAUSE>
 89                      | <TIMING CLAUSE>
 90                      | <REGISTER HEAD> <REGISTER LIST>
 91                      | <SUBREG HEAD> <SUB REGISTERS>
 92                      | <CASCADE REG>
 93   <PIN CLAUSE> ::= <PIN> <ARE NUMBERED> <ARE NAMED>
 94   <PIN> ::= PIN
 95           | PINS
 96           | WIRES
 97   <TIMING CLAUSE> ::= <SET TIME> <TIME>
 98   <SET TIME> ::= SET <TIME=>
 99   <REGISTER HEAD> ::= REGISTERS <ARE>
100                     | REGISTER <ARE>
101   <REGISTER LIST> ::= <REGISTER DSCR>
102                     | <REGISTER LIST><,><REGISTER DSCR>
103   <REGISTER DSCR> ::= <IDENTIFIER> <DIMENSION>
104   <DIMENSION> ::= <(> <SUBSCRIPT> )
105                 | <(> , <SUBSCRIPT> )
106                 | <FIRST DIM> <,> <SECOND DIM>
107   <SUBSCRIPT> ::= <INTEGERS>
108                 | <IDENTIFIER>
109                 | <PIN POSITIONS>
110   <PIN POSITIONS> ::= <PIN> <{> <NAME LIST> <}>
111                     | <PIN> <{> <NUMBER LIST> <}>
112                     | <PIN> <IDENTIFIER>
113                     | <PIN> <INTEGERS>
114   <{> ::= {
115         | [
116   <}> ::= }
117         | ]
118   <FIRST DIM> ::= <(> <SUBSCRIPT>
119   <SECOND DIM> ::= <SUBSCRIPT> )
120   <SUBREG HEAD> ::= SUBREGISTERS OF <IDENTIFIER> <ARE>
126   <SUB REGISTERS> ::= <REGISTER BITS>
122                     | <SUB REGISTERS><,><REGISTER BITS>
123   <REGISTER BITS> ::= <IDENTIFIER> <INTEGERS>
124   <CASCADE REG> ::= CASCADE <STRINGOFCADES> INTO
                        <IDENTIFIER>
125   <STRINGOFCADES> ::= <IDENTIFIER>
126                     | <STRINGOFCADES> <,> <IDENTIFIER>
127   <CMPNT IF CLAUSE> ::= <IFSTRINGTHEN> <CMPNT LOGIC>
128                       | <IFSTRINGTHEN> <CMPNT LOGIC>
                            ELSE <CMPNT LOGIC>
129   <IFSTRINGTHEN> ::= <IF> <STRINGOFBOOLS> <THEN>
```

```
130   <IF> ::= IF
131   <STRINGOFBOOLS> ::=<STRINGOFBOOLS><NDOR><BOOLEAN EXP>
132                     | <BOOLEAN EXP>
133   <NDOR> ::= AND
134            | OR
135   <BOOLEAN EXP> ::= <LOGICAL REPLMNT> <RELATIONAL>
                        <LOGICAL REPLMNT>
136                   | <(> <STRINGOFBOOLS> )
137   <LOGICAL REPLMNT> ::= <SYMBOLIC VALUE>
138                       | <VALUE>
139   <SYMBOLIC VALUE> ::= <REGISTER DSCR>
140                      | <IDENTIFIER>
141                      | <PIN POSITIONS>
142   <VALUE> ::= <NUMBER> <BASE INDICATOR>
143   <BASE INDICATOR> ::= O
144                      | Q
145                      | H
146                      | B
147                      | D
148                      |
149   <RELATIONAL> ::= NOT EQ
150                  | EQ
151                  | <
152                  | >
153                  | NOT <
154                  | NOT >
155   <THEN> ::= THEN
156   <CMPNT LOGIC> ::= <CMPNT REP EXP>
157                   | <CMPNT LOGIC> <,> <CMPNT REP EXP>
158   <CMPNT REP EXP>::=<SYMBOLIC VALUE><=><RIGHTHND LOGIC>
159                   | <RESET TIME> <TIME>
160   <=> ::= =
161   <RIGHT HND LOGIC> ::= <OPERAND>
                        | <RIGHT HND LOGIC><OPERATOR><OPERAND>
163   <OPERAND> ::= <LOGICAL REPLMNT>
164               | <MONADIC OPRATR> <LOGICAL REPLMNT>
165               | <(> <RIGHT HND LOGIC> )
166               | NOT <(> <RIGHT HND LOGIC> )
167   <MONADIC OPRATR> ::= RR
168                      | SR
169                      | RL
170                      | SL
171                      | CMP
172                      | NEG
173   <OPERATOR> ::= + | - | / | * | ! | &
179                | XOR
180                | NOR
181                | NAND
182   <RESET TIME> ::= RESET <TIME=>
```

```
183   <ELSE> ::= <,> ELSE
184   <ON PULSE> ::= ON PULSE <,>
185                | ON <,>
186   <MEMORY STMNT> ::= <MEMORYHEAD> <MEMORY CLAUSES>
187   <MEMORYHEAD> ::= MEMORY : <IDENTIFIER>
188                  | MICROMEMORY : <IDENTIFIER>
189                  | AUXMEMORY : <IDENTIFIER> <NUMBER>
190   <MEMORY CLAUSES> ::= <MEMORY CLAUSE>
191                      |<MEMORY CLAUSES><;><MEMORY CLAUSE>
192   <MEMORY CLAUSE> ::= <COMMON CLAUSES>
193                     | <SIZE CLAUSE>
194                     | <MEMIF CLAUSE>
195                     | <ON PULSE> <MEM REP EXPRS>
196   <SIZE CLAUSE> ::= <FIRST SIZE NO> <SECOND SIZE NO>
197   <FIRST SIZE NO> ::= SIZE = <NUMBER>
198   <SECOND SIZE NO> ::= * <NUMBER>
199   <MEMIF CLAUSE> ::= <MEMIFTHEN> <MEM REP EXPRS>
200                    | <MEMIFTHEN> <MEM REP EXPRS> <ELSE>
                         <MEM REP EXPRS>
201   <MEMIFTHEN> ::= <IF> <STRNGMEMBOOLS> <THEN>
202   <STRNGMEMBOOLS> ::= <STRNGMEMBOOLS> <NDOR> <MEMBOOL>
203                     | <MEMBOOL>
204   <MEMBOOL> ::=<MEM REPLCMNT><RELATIONAL><MEM REPLCMNT>
205   <MEM REPLCMNT> ::= <LOGICAL REPLMNT>
206                    | <MEMORY VALUE>
207   <MEMORY VALUE> ::= <MEMORY> <DIMENSION>
208   <MEMORY> ::= MEMORY
209   <MEM REP EXPRS> ::= <MEM REP EXPRS> <,> <MEM REP EXP>
210                     | <MEM REP EXP>
211   <MEM REP EXP> ::= <MEM REPL LEFT> <=> <MEM REP RIGHT>
212                   | <RESET TIME> <TIME>
213   <MEM REPL LEFT> ::= <MEMORY VALUE>
214                     | <SYMBOLIC VALUE>
215   <MEM REP RIGHT> ::= <MEMORY VALUE>
216                     | <RIGHT HND LOGIC>
217   <PERIPHERAL STMN> ::= <PERIPHERAL ID><PERIPH CLAUSES>
218   <PERIPHERAL ID> ::= I-O : <IDENTIFIER>
219                     | PORT : <IDENTIFIER>
220                     | TERMINAL : <IDENTIFIER>
221                     | PRINTER : <IDENTIFIER>
222   <PERIPH CLAUSES> ::= <PERIPH CLAUSE>
223                      |<PERIPH CLAUSES><;><PERIPH CLAUSE>
224   <PERIPH CLAUSE> ::= <COMMON CLAUSES>
225                     | <FORMAT CLAUSE>
226                     | <PERIPHIF CLAUSE>
227                     | <ON PULSE> <IN-OUT EXPRNS>
228   <FORMAT CLAUSE> ::= <FORMAT IS> <FORMAT>
229   <FORMAT IS> ::= FORMAT <ARE>
```

```
230   <FORMAT> ::= HEX
231              | HEXADECIMAL
232              | OCTAL
233              | BINARY
234              | ASCII
235              | BCD
236              | EBCDIC
237              | GREY
238              | PCKD-DEC
239              | SGND-BIN
240              | FLPOINT
241   <PERIPHIF CLAUSE> ::= <IFSTRINGTHEN> <IN-OUT EXPRNS>
                          | <IFSTRINGTHEN> <IN-OUT EXPRNS>
                            LSE> <IN-OUT EXPRNS>
243   <IN-OUT EXPRNS> ::=<IN-OUT EXPRNS><,><IN-OUT REP EXP>
244                    | <IN-OUT REP EXP>
245   <IN-OUT REP EXP> ::= <OUTPUT EXP> <=> <INPUT EXP>
246                      | DISPLAY <"> <">
247                      | <RESET TIME> <TIME>
248   <"> ::= "
249   <OUTPUT EXP> ::= <OUTPUT>
250                  | <LOGICAL REPLMNT>
251   <OUTPUT> ::= OUTPUT
252   <INPUT EXP> ::= <INPUT>
253                 | <LOGICAL REPLMNT>
254   <INPUT> ::= INPUT
255   <WIRING STMNT> ::= <WIRING HEAD> <TERMINAL PNTS>
256   <WIRING HEAD> ::= CONNECT <IDENTIFIER>
257   <TERMINAL PNTS> ::= <TO TERMINAL> <END ONE> <END TWO>
258                     | <TERMINAL PNTS> ; <TO TERMINAL>
                          <END ONE> <END TWO>
259   <TO TERMINAL> ::= TO <IDENTIFIER>
260   <END ONE> ::= <{> <NUMBER LIST> <}>
261   <END TWO> ::= <{> <NUMBER LIST> <}>
262   <START STMNT> ::= <STRT STMNT HEAD> <START CLAUSES>
263   <STRT STMNT HEAD> ::= START :
264   <START CLAUSES> ::= <START CLAUSES> ; <START CLAUSE>
265                     | <START CLAUSE>
266   <START CLAUSE> ::= <MEM REPL LEFT> <=> <VALUE>
267                    | <MEM REPL LEFT> <OF-IN>
                         <IDENTIFIER> <=> <VALUE>
268                    | <TIMING CLAUSE>
269   <OF-IN> ::= OF
270             | IN
```

APPENDIX 4

INDEX TO RESERVED WORDS USED IN CALSIM GRAMMAR

SYMBOL	USED IN PRODUCTION	SYMBOL	USED IN PRODUCTION
.	1,4,5,12-22	RL	169
<	151,153	RR	167
(	71	SL	170
+	173	SR	168
&	178	TO	259
!	177	AND	74,133
*	45,176,198	ARE	6
)	70,104,105,119,	BCD	235
	136,165,166	BUS	62
;	58,258,264	CMP	171
-	70,174	END	1,9
/	175	HEX	230
,	36,105	I-O	218
>	152,154	NEG	172
:	43,50,60-62,81,82,	NOR	180
	187-189,218,219	NOT	149,153,154,166
	220,221,263	PIN	94
=	40,41,160,197	SET	98
"	248	XOR	179
{	114	CHIP	82
}	116	COPY	28-31
[	115	CORD	61
]	117	EACH	33
B	146	ELSE	183
D	147	GREY	237
H	145	INTO	124
O	143	NAME	5
Q	144	NAND	181
AS	25	PINS	95
EA	32	PORT	219
EQ	149,150	SIZE	197
IF	130	THEN	155
IN	270	TIME	40
IS	7	ASCII	234
OF	1,9,120,269	INPUT	354
ON	184,185	LEVEL	50
OR	134	LIMIT	79

OUTPUT	251
RENAME	34
SUFFIX	35
SYSTEM	5
<EMPTY>	8,37,48,64,73,148
CASCADE	124
CONNECT	256
DISPLAY	246
FLPOINT	240
PRINTER	221
<NUMBER>	29,44,50,57,68,69,70,142,189,197,198
CONTAINS	53
NUMBERED	63
PCKD-DEC	238
REGISTER	100
SGND-BIN	239
TERMINAL	220
AUXMEMORY	189
BACKPLANE	60
COMPONENT	81
REGISTERS	99
HEXADECIMAL	231
MICROMEMORY	188
<IDENTIFIER>	5,9,24,28,29,30,31,34,35,53,56,57,60,61, 62,76,77,81,82,103,108,112,120,123,124, 125,126,140,187,188,189,218,219,220,221 256,259,267
DESCRIPTIONS	1
SUBREGISTERS	120
<(>	69,104,105,118,136,165,166
<;>	52,84,191,223
<,>	27,39,47,55,66,76,81,82,102,106,122,126, 157,183,184,185,209,243
<=>	158,211,245,266,267
<">	246
<{>	110,111,260,261
<}>	110,111,260,261
<IF>	129,201
<ARE>	5,63,99,100,120,229
<PIN>	93,110,111,112,113
<EACH>	29,57
<ELSE>	128,200,242
<GOAL>	NOT USED IN THE RHS OF ANY PRODUCTION
<NDOR>	131,202
<THEN>	129,201
<TIME>	38,39,43,46,47,78,97,159,212,247
<INPUT>	252
<OF-IN>	267
<TIME=>	38,39,79,98,182

<VALUE>	138,266,267
<FORMAT>	228
<MEMORY>	207
<OUTPUT>	249
<RANGE1>	67
<RANGE2>	67
<RENAME>	30
<SYSTEM>	2,3
<END ONE>	257,258
<END TWO>	257,258
<MEMBOOL>	202,203
<OPERAND>	161,162
<SYSTEMS>	1,3
<TIME2-3>	39
<ANDNAMED>	72
<INTEGERS>	65,66,107,113,123
<ON PULSE>	87,195,227
<OPERATOR>	162
<SET TIME>	97
<STORE AS>	24
<SUFFIXID>	31
<ARE NAMED>	59,93
<COPY HEAD>	26,27
<DIMENSION>	103,207
<FIRST DIM>	106
<FORMAT IS>	228
<INPUT EXP>	245
<MEMIFTHEN>	199,200
<NAME LIST>	72,76,110
<NMBORSTAR>	42,43
<STATEMENT>	10,11
<SUBSCRIPT>	104,105,118,119
<TIME INCR>	46
<COPY STMNT>	13
<LEVEL WORD>	49
<LIMIT TIME>	78
<MEMORYHEAD>	186
<OUTPUT EXP>	245
<PIN CLAUSE>	88
<RELATIONAL>	135,204
<RESET TIME>	159,212,247
<SECOND DIM>	106
<STATEMENTS>	4,10,23
<STORE HEAD>	23
<BOOLEAN EXP>	131,132
<CASCADE REG>	92
<CMPNT LOGIC>	87,127,128,157
<MEM REP EXP>	209,210
<NUMBER LIST>	63,66,111,260,261

<SIZE CLAUSE>	193
<START STMNT>	21
<STORE STMNT>	12
<SUBREG HEAD>	91
<SYSTEM NAME>	4
<TIME CLAUSE>	27
<TO TERMINAL>	257,258
<WIRING HEAD>	255
<ARE NUMBERED>	59,93
<CMPNT CLAUSE>	83,84
<END OF STMNT>	4,23
<IFSTRINGTHEN>	127,128,241,242
<MEM REPLCMNT>	204
<MEMIF CLAUSE>	194
<MEMORY STMNT>	18
<MEMORY VALUE>	206,213,215
<START CLAUSE>	264,265
<WIRING STMNT>	20
<CMPNT CLAUSES>	80,84
<CMPNT REP EXP>	156,157
<FIRST SIZE NO>	196
<FORMAT CLAUSE>	225
<IN-OUT EXPRNS>	227,241,242,243
<MEM REP EXPRS>	195,199,200,209
<MEM REP RIGHT>	211
<MEM REPL LEFT>	211,266,267
<MEMORY CLAUSE>	190,191
<PERIPH CLAUSE>	222,223
<PERIPHERAL ID>	217
<PIN POSITIONS>	109,141
<REGISTER BITS>	126,122
<REGISTER DSCR>	101,102,139
<REGISTER HEAD>	90
<REGISTER LIST>	90,102
<START CLAUSES>	262,264
<STRINGOFBOOLS>	129,131,136
<STRINGOFCADES>	124,126
<STRNGMEMBOOLS>	201,202
<SUB REGISTERS>	91,122
<TERMINAL PNTS>	255,258
<TIMING CLAUSE>	89,268
<BASE INDICATOR>	142
<COMMON CLAUSES>	85,192,224
<COMPONENT HEAD>	80
<CONNECTOR HEAD>	59
<HARDWARE UNITS>	54,55
<HIERARCHY HEAD>	51,52
<IN-OUT REP EXP>	243,244
<MAJ. ASSY LIST>	49,52

<MEMORY CLAUSES>	186,191
<MONADIC OPRATR>	164
<PERIPH CLAUSES>	217,223
<SECOND SIZE NO>	196
<SYMBOLIC VALUE>	137,158,214
<TIME STATEMENT>	16
<CMPNT IF CLAUSE>	86
<COMPONENT STMNT>	17
<CONNECTOR STMNT>	15
<HIERARCHY STMNT>	14
<LOGICAL REPLMNT>	135,163,164,205,250,253
<MAJ. ASSEMBLIES>	51,52,55
<PERIPHERAL STMN>	19
<PERIPHIF CLAUSE>	226
<RIGHT HND LOGIC>	158,162,165,166,216
<STRT STMNT HEAD>	262

APPENDIX 5

TRIAL GRAMMAR OF MANUFACTURING DESCRIPTION LANGUAGE

<GOAL>	<SYSTEMS> END OF DESCRIPTIONS .
<SYSTEMS>	<SYSTEM>
	<SYSTEMS> <SYSTEM>
<SYSTEM>	<SYSTEM NAME> <STATEMENTS> <END OF STMNT>.
<SYSTEM NAME>	SYSTEM NAME <IDENTIFIER> .
<END OF STMNT>	END OF <IDENTIFIER>
<STATEMENTS>	<STATEMENTS> <STATEMENT>
	<STATEMENT>
<STATEMENT>	<SAVE STMNT> .
	<COPY STMNT> .
	<ORGNZTION STMNT> .
	<CONTNERS STMNT> .
	<XPORTUNITS STMNT> .
	<SKILLSREQ STMNT> .
	<FACILTIES STMNT> .
	<RECEIVING STMN> .
	<WAREHOUSE STMNT> .
	<MFG-UNIT STMNT> .
	<INSPECTION STMNT> .
	<MFGPLANNG STMNT> .
	<ROUTING STMNT> .
	<SHIPPING STMNT> .
	<START STMNT> .
<SAVE STMNT>	<SAVE HEAD> <STATEMENTS> <END OF STMNT>
<SAVE HEAD>	<SAVE AS> <IDENTIFIER>
<SAVE AS>	SAVE AS
<COPY STMNT>	<COPY LEADER> <STATEMENT> <END OF STMNT>
<COPY LEADER>	<COPY HEAD> <USING TIME> <TIME CLAUSES> ;
<COPY HEAD>	COPY <IDENTIFIER> <RENAME>
<RENAME>	RENAME <IDENTIFIER>
<SUFFIXID>	SUFFIX <IDENTIFIER>
<ORGNZTION STMNT>	<LEVEL WORD> <UPR ORGAN LIST>
<LEVEL WORD>	LEVEL <NUMBER> :
<UPR ORGAN LIST>	<ORGNZTION HEAD> <MAJ. WORK AREAS>
	<UPR ORGAN LIST> ; <ORGNZTION HEAD>
	<MAJ. WORK AREAS>
<ORGNZTION HEAD>	<IDENTIFIER> CONTAINS
<MAJ. WORK AREAS>	<HARDWARE UNITS>
	<MAJ. WORK AREAS> <,> <HARDWARE UNITS>

```
<HARDWARE UNITS>   <IDENTIFIER>
<,>                ,
                   <EMPTY>
<CONTNERS STMNT>   <CONTNERS HEAD> <CONTNER CLAUSES>
<CONTNERS HEAD>    <CNTNER: <IDENTIFIER>  <NUMBER> <EACH>
<CONTNER CLAUSES>  <CONTNER CLAUSE>
                   <CONTNER CLAUSES> ; <CONTNER CLAUSE>
<CONTNER CLAUSE>   CAPACITY = <NUMBER> POUNDS
                   <SIZE=> <WIDTH> X <LENGTH> X <HEIGHT>
                   TOOL # = <IDENTIFIER>
                   <ETC ETC TO DEVELOP DESCRIPTIONS>
<SIZE=>            SIZE =
<WIDTH>            <NUMBER>
<LENGTH>           <NUMBER>
<HEIGHT>           <NUMBER>
<EACH>             EA
                   EACH
<MFG-UNIT STMNT>   <MFG-UNIT HEAD> <MFG UNIT CLAUSES>
<MFG-UNIT HEAD>    MFG-UNIT : <IDENTIFIER>
<MFGUNITCLAUSES>   <MFGUNITCLAUSES> ; <MFGUNIT CLAUSE>
                   <MFGUNIT CLAUSE>
<MFGUNIT CLAUSE>   EQUIPMENT IS <FAC. LIST>
                   MANNING IS <BILLET LIST>
                   AREA IS <NUMBER>
                   IN STORAGE IS <STORAGE LIST>
                   OUT STORAGE IS <STORAGE LIST>
                   COM STORAGE IS <STORAGE LIST>
                   HANDTOOLS ARE <TOOL LIST>
<XPORTUNITS>       <XPORT HEAD>  <XPORT CLAUSES>
<XPORT HEAD>       MOVER: <IDENTIFIER>
<XPORT CLAUSES>    <XPORT CLAUSES> <,>  <XPORT CLAUSE>
                   <XPORT CLAUSE>
<XPORT CLAUSE>     <CAPACITY HEAD> <CAPACITY LIST>
<CAPACITY HEAD>    CAPACITY =
<CAPACITY LIST>    <NUMBER> <EACH> <TYPE>
                   <CAPACITY LIST> <OR> <NUMBER <EACH> <TYPE>
<FACILTIES STMNT>  <FACILITY HEAD> <FAC. CLAUSES>
<FACILITY HAED>    FACILITY : <IDENTIFIER>
<FAC. CLAUSES>     <FAC. CLAUSES> ; <FAC. CLAUSE>
                   <FAC. CLAUSE>
<FAC. CLAUSE>      COST = <NUMBER>
                   PURCHASE DATE = <NUMBER>
                   MFG = <IDENTIFIER>
                   SPACE NEEDED = <NUMBER>
                   POWER REQ = <NUMBER>
                   EXPECTED LIFE = <NUMBER>
                   ID = <IDENTIFIER>
<SKILLSREQ STMNT>  <SKILL HEAD> <SKILL CLAUSES>
<SKILL HEAD>       SKILL: <SKILL NMBR>
```

```
<SKILL NMBR>        <NUMBER>
<SKILL CLAUSES>     <SKILL CLAUSES> <;> <SKILL CLAUSE>
<SKILL HEAD>        <SKILL : > <IDENTIFER>
<SKILL CLAUSES>     <SKILL CLAUSES> ; <SKILL CLAUSE>
                    <SKILL CLAUSE>
<SKILL CLAUSE>      DEGREE = <IDENTIFIER>
                    GRADE  = <NUMBER>
                    RATE   = <NUMBER>
                    OVHD RATE = <NUMBER>
                    TRADE TNG = <NUMBER>
                    GEN EXP   = <NUMBER>
                    SPECIF EXP = <NUMBER>
                    WT LIMIT = <NUMBER>
                    SEEING = <NUMBER>
                    PHYS COND = <NUMBER>
```

APPENDIX 6

USERS' MANUAL FOR CALSIM/SIMCAL

COMPUTER ARCHITECTURE LANGUAGE FOR SIMULATION

AND

SIMULATOR FOR COMPUTER ARCHITECTURE LANGUAGE

A COMPUTER HARDWARE DESCRIPTION LANGUAGE AND SIMULATOR

FOR

COMPUTER ORGANIZATION AND DESIGN STUDIES IN

COMPUTER SCIENCE EDUCATION

W. A. Skelton

THE UNIVERSITY OF TEXAS AT ARLINGTON

September 1982

PREFACE

The language and software support system described here may be used to assist the student in understanding computer organization and architecture above the switch circuit level. The system is equally usable for both beginning and advanced studies at the architectural, programming, and register transfer levels.

A complete description of the language is presented in chapter three (3), a design approach in chapter five (5), and detail instructions for using the simulator in chapter six (6). Like programming languages, the method of learning Calsim must be a spiral one of learning a small part of the syntax, followed by use to "fix" that portion studied.

At the beginning level, previously prepared code, available through the system, may be used to simulate flip-flops, half-adders, and other elementary devices. The student is also free to make changes to the configuration in his copy so that various actions can be studied.

At an intermediate level, the system can be used to build simulators for eight bit and sixteen bit microprocessors. Simulation can be carried out allowing the user to debug programs prepared for the devices. Operating hardware may also be designed and tested using either microprocessors or more elementary components.

At an advanced level, the system will support simulation and design activity using microprogramming including bit-sliced hardware. Other advanced projects may include investigation into parallel operations and the use of hardware to study communication protocol problems.

W. A. Skelton
September 10, 1982

T A B L E   O F   C O N T E N T S

1.0 INTRODUCTION

1.1 THE NATURE OF HARDWARE DESCRIPTION LANGUAGES

Languages have been developed in the last two decades to define computer hardware designs in precise formal terms. These Languages have been known by a variety of names some of which are: Hardware Description Language (HDL), Computer Hardware Description Language (CHDL), Computer Design Language (CDL), System Design Language (SDL) and Architecture Design Language (ADL). These languages have many functions but they all attempt to convey more preciseness in the functional and physical design description without introducing more complexity. In addition, CHDL's are expected to be machine processable, i.e., it is expected that the language will allow the input to be checked for completeness and accuracy and that extensions can be directly prepared which will provide simulation and/or circuit generation.

1.2 PURPOSES OF THE LANGUAGES.

A means of describing design intent unambiguously.
To assist simulation of the proposed design.
To assist in automatic generation of circuits.
To provide a test bed for microprogramming studies.
To evaluate alternate hardware designs.
To aid in Computer Science education.

The languages have been developed at several levels, depending on the intended use, and proposals have been made to create a universal language in which one could simply use the portion of the language for the level desired. Unfortunately, this has not yet come to pass.

At the gate (switch circuit) level, the actual arrangement of the gates are a part of the hardware under consideration, whereas the register transfer level is more more abstract, not addressing anything below a single bit. As the LSI's (Large Scale Integrated Circuits) became available in the seventies, emphasis shifted to still higher levels of computer organization. Calsim addresses the register transfer, programming, and system (architectural) levels.

The system works well for investigations using microprogrammable designs including those using bit-sliced components.

1.3 THE CALSIM LANGUAGE.

The language called CALSIM (Computer Architecture Language for SIMulation) consists of a series of English-like statements which describe the hardware system under study. The user starts the description by providing a system name and a name for each of the modules comprising the system, including connectors such as cords, buses and backplanes.

The assemblies are next decomposed into smaller groups until the basic chips and connectors are reached. The description of these elementary items - chips, memories, and connectors, are then entered. At the lowest hierarchical level, the connectors will be given numbers and may be given names of up to six (6) characters. The logical component descriptions are usually taken directly from the manufacturer's data sheet, a complex chip requiring up to several hundred lines, with the user modifying the timing as required. The elementary item description will include:

1. Registers:	Names, type, length (and width).
2. Pins Description:	Names and numbers of pins.
3. Timing functions:	Sets times the chip is active.
4. Logical functions:	Describes the logic of the chip.

Both memory and peripheral device descriptions use the above functions and have additional language features to augment their descriptions. Examples of this are the memory size clause and a format clause for passing data in and out of the peripheral. This allows data passing to the user's terminal to be converted to a readable ASCII format and the data from the terminal to be converted from ASCII to the format required for storage.

When all of the components, including the peripherals and memory have been entered, the components are "wired" using the wiring statement. The starting conditions for the trial machine may also be set using the start statement.

1.4 DESIGNING A SYSTEM.

The user prepares the design, writes the Calsim description, and processes it through the compiler. Errors are corrected and the description is reprocessed until it is free of syntax and logical errors at which time it is ready to simulate. The user first provides the contents of main memory, micromemory, memory, other memories, and input data. He then starts the simulation. Appendix 3 contains a variety of examples which may be used for familiarization.

The user will find that the block diagram and logic description of the system under study are sufficient to represent the object machine providing the block diagram includes a clear representation of the connections between the components. In many cases the user may wish to use an abstraction of the actual design, particularly in the early stages of the project or where the details are not significant to the work being performed.

It is usually beneficial to develop the descriptions of the chips to be used prior to system design so that the "chips" are ready for use when needed. Such descriptions may be checked for syntax errors and proper operation in the simulator prior to storing in the user's library. In some cases, a proven description of the chip may already be available from a library of chip descriptions.

The complete software support system consists of:

* A LANGUAGE -
  CALSIM (Computer Architecture Language for SIMulation)

* A SOFTWARE SUPPORT SYSTEM -
  SIMCAL (SIMulator for Computer Architectural Language)

* THE DEC-20 SYSTEM -
  used to create/edit support files and execute Simcal
  (All DEC control characters are active inside Simcal.)

1.5 SYSTEM OVERVIEW

Support is provided for study of logical designs, from single component checkout to microprogrammable machine designs, for the following functions:

A language to specify a design including the logic.

A compiler to build tables to drive the simulator.

Software to print the tables in readable format.

Support for a user's component library.

Read-in of memory, microcode, and other memory code.

A simulator driven by the hardware description tables.

A friendly interface between the user and simulator.

Several proven designs which may be used as exercises.

2.0 GETTING STARTED WITH THE CALSIM/SIMCAL SYSTEM

2.1 GETTING ON THE DEC-20.

The user should first acquire skill with the "TOPS" and "EDIT" commands, using the "HELP" and "?" as needed or the user may prefer to use Wybur on the IBM-4341 to build files. These can be transferred to the DEC-20 with the TRANSFER command and "cleaned up" on the DEC-20 by using "TRAPLINK".

2.2 GETTING THE SIMCAL MASTER FILE AND STARTING EXECUTION.

The user will need several files, some of which are empty. The list of files needed may be obtained by entering "COPY <CS.Bnnn-acntname>SIMCAL.CMD." The user then prints the contents of that file and creates in his own workspace all of the files listed (they may be empty) which do not have "<CS.Bnnn-acntname>" at their beginning. The remaining files will be obtained from from CS.Bnnn at execution time. The user is then ready to execute the command file:

```
TAK<esc>SIMCAL.CMD
```

The machine will start execution of the command file by first obtaining SIMCAL.REL from CS.Bnnn storage, binding the support files to SIMCAL.REL, and starting execution. The entry message displays the version number of Simcal. The user should become familiar with the software by using the commands in the software support system including "HELP" and "?". "HELP" will provide an explanation and "?" will list the commands availble in each particular environment.

2.3 ENTERING SIMCAL AND COMPILING A DESIGN DESCRIPTION.

The user prepares a description of a hardware design by using one of the examples shown in Appendix 3 or preparing a description from an original design. The file must be left unnumbered and renamed "SIMHDW.DAT". Execution is started by entering "TAKE SIMCAL.CMD". Once in Simcal, entry of "HDW" at the terminal will place the user in the compiler environment where compilation is started by use of "C,P" (COMPILE, PRINT). Error correction and recompilation is repeated until the description is error free. The user then enters the 'TAB' section of Simcal, prints the tables, and compares them to the original design intent.

2.4 MEMORY CODE NEEDED.

If program code is needed to test the design, it is prepared in a work file and renamed "SIMPGM.DAT" when ready for use. When complete, the memory file is read into the simulator memory, after compilation of the hardware description, but before simulation. If the design is microprogrammable, the user must also prepare microcode. The file containing the microcode will be renamed SIMMIC.DAT when complete. Other memory is entered in one or more files named 'SIMAXn.DAT', where 'n' is either 1, 2, or 3. These will be read-in through the 'AUX' section of Simcal. The format, uses, and details of reading in the code is found in section three.

2.5 INPUT AND OUTPUT DATA FILES

Finally if an input data file is needed by the executing module during simulation, then up to three files can be used which are labeled SIMWK1.DAT, SIMWK2.DAT, and SIMWK3.DAT. If an output file is needed, any of these files may be used as output from the program being executed by creating an empty file. When a file is used as input or output, the file is automatically opened on entry to simulation and automatically closed on exit from simulation. Reading or writing may be accomplished a byte at a time or other increments as defined by the user's hardware description or program. Input and output files from the executing program will always start at the beginning of the file. Output files will write over any data present. Restart of input or output files may be accomplished by either leaving and returning to the simulator or by execution of the "START" command as explained in section 6.9 of this manual.

2.6 SIMULATION.

When all of the steps above are complete, the user is ready to simulate the hardware description. The steps that have been necessary to reach this point are summarized below.

1. Design (or COPY) device descriptions to be simulated.
2. Rename the CHDL description file as SIMHDW.DAT.
3. Enter Simcal and compile the description.
4. Correct the errors and recompile until error free.
5. Print tables of the compilation and compare to intent.
6. Correct hardware descriptions as required.
7. Recompile as required.
8. Write the program code and place in SIMPGM.DAT file.

9. Write the microcode and place in SIMMIC.DAT file.
10. Write other memory code as required.
11. Read all memory code into Simcal.
12. Write input data into SIMWKn.DAT as required.
13. Write a test plan.
14. Simulate the hardware by entering "SIM".
15. Determine errors and repeat as required.

3.0 THE CALSIM LANGUAGE

3.1 STARTING A HARDWARE DESCRIPTION

Calsim is free form and accepts all entries from column one through column eighty without using continuation characters. In the examples shown here, the indentation is made so the reader can more clearly distinguish the description from the actual text. Calsim language is organized by statements, each ending in a period, and most are divided into clauses separated by semicolons. A Backus Normal Form (BNF) listing of the grammar is available through Simcal by entering "BNF" in the "DOC" environment.

The compiler will terminate the analysis of a statement if an error is discovered in the syntax, skipping to the next statement to continue compilation. The user must then correct the error in order for the syntactical analysis of that statement to continue. Calsim uses approximately one hundred reserved words and symbols (see Appendix 1) which may not be used for component names (identifiers).

The description of the user's design will be included between the two statements in the example shown below using a system name of up to 30 characters. Two or more hardware descriptions during one compilation may be accomplished by following the end statement in the example with a second system name statement and description. Note that the statements end with a period, a part of the required syntax.

```
SYSTEM NAME  myveryowndesign.
description of the system
END OF myveryowndesign.
SYSTEM NAME  second-description.
second system goes here.
END OF  second-description.
END OF DESCRIPTIONS. (Last entry in listing)
```

3.2 THE HIERARCHICAL STATEMENT

The design under study is sub-divided into major consoles, boxes, boards, etc. These in turn are sub-divided into lower levels down to the component level. The LSI is ordinarily the lowest level, but breakdown may be carried to lower levels when required. Each of the components are given a name of no more than 12 letters, hyphens, or numbers, starting with a letter. The words of the language are separated by spaces and where a list is used, commas may be used also to separate the members. An example follows:

```
LEVEL 1: MYVERYOWNDESIGN CONTAINS FRONTPANEL, BOX1,
         CORD1, TERMINAL1.
```

The level number may be any integer from 1 through 99 and as each lower level is described, a larger number is used. In the example BOX1 might then be further decomposed as:

```
LEVEL 2: BOX1 CONTAINS BGBKPLANE BOARD1 BOARD2 BOARD3.
```

The assemblies are then divided into their component parts. If a single type component is used several times, syntax is available to preclude the necessity of repetitive entries of the same item as shown below.

```
LEVEL 3: BOARD1 CONTAINS 4 EACH AM931, BRDA, BRDB,
        2 EA AX972.
```

The system will assign a two digit number to the end of any description preceded by the "EACH" expression. The statement above calls for only one "BRDA" and "BRDB", but four (4) "AM931" chips and two "AX972" chips. The AM931's will have "01", "02", "03", and "04" added to the identifier giving the names "AM93101", "AM93102", "AM93103", and "AM93104" to the actual description saved in the tables during compilation. If the length of the name is already twelve (12) characters, the last two characters will be replaced. Very likely there will be several "CONTAINS" clauses.

```
LEVEL: BOX1 CONTAINS PCB1, PCB2, PCB3, BPLANE1;
       BOX2 CONTAINS PCB4, PCB5, PCB6, BPLANE2.
```

Note the semi-colon at the end of the second line. The "." is only used at the end of the statement. The user, however, could have written two statements for the description as shown below:

```
LEVEL: BOX1 CONTAINS PCB1, PCB2, PCB3, BPLANE1.
LEVEL: BOX2 CONTAINS PCB4, PCB5, PCB6, BPLANE2.
```

At the completion of the hierarchy statement descriptions, the elementary items still remain to be described as either peripherals, memory, components, or conductors. Each of these elementary items must be fully described as shown in the syntax examples in sections 3.5 through 3.7.

Once the system design is complete, the heirarchy, including the connectors, should be entered and checked against the output table from Simcal, the software support system. Some users may prefer to check the hierarchy of the design prior to entering the detailed descriptions of the chips and buses. This is done by compiling the design and requesting the hierarchy tables.

3.3 THE COMMENT STATEMENT

Comments may be placed anywhere within the CALSIM code by entering "/*". The comment may be terminated at any point by "*/". The compiler will recognize the symbol "/*" as the beginning of comment and will continue to ignore text until the comment termination symbol "*/" is scanned. Thus several lines of comments require only a single beginning and a single ending symbol as shown below.

```
LEVEL 3: TOP-CONSOLE CONTAINS /* AN INSERTED COMMENT
         IS IN THE MIDDLE OF THIS STATEMENT */ AA, BB.
```

3.4 THE TIMING STATEMENT

The timing statement places upper bounds upon each of the timing segments in the simulated machine. The clock may have up to eight segments, each of which may have any value from 0 through 9. In the following example, the first six segments, of a possible eight (8), are used and the values within each segment may have any value not greater than the digit shown. The total time events available to the designer in this example, is 4 X 3 X 10 X 1 X 5 X 2 = 1200.

```
      CLOCK LIMIT IS 3:2:9:0:4:1.
```

By setting the upper limit on the clock, the user causes the compiler to check each of the time clauses submitted, as part of an elementary item description, to assure that no segment value is greater than that allowable. During simulation, the sequencing of the component execution will be controlled by the time clauses provided in the elementary item descriptions which allows the user to repeat cycles and perform concurrent operation. See section 3.14 for a discussion of timing clauses used in the item description.

3.5 THE CONNECTOR DESCRIPTION STATEMENT

The connector description statements generally, but not necessarily follow the heirarchial statements and provides the system with the wire names and numbers within the connector. The wire numbers may be as high as 999 and names may have up to 6 characters. The connector statement does not concern itself with the external connections of the bus, cord, or cable, only the wires in the connector and their related names. The statement must start with either "Cord:" "Bus:" or Backplane:" which are actually equivalent to the compiler and are provided for user convenience only. An example follows:

```
CORD: CORD1 IS NUMBERED (1-18).
BACKPLANE: BIG-PLANE NUMBERED (1-256).
```

The verb "IS" is optional and the value "(1-18)" represents a range. The names of the connectors were not included in the example above, but could have been added to the description as shown here. The "AND" and the spaces following the comma are optional. If fewer names are entered than numbers, the system provides a warning, but the numbers and names provided are compiled.

```
CORD: CORD1 NUMBERED (1-18) AND NAMED PRW, D1, D2,
D3, D4, INT, GWD.
```

It is necessary that all components - memories, peripherals, and LSI's be completely connected using this statement. Calsim does not permit testing (of a register content or value on a pin) in one chip to result in an action in another chip. Such action must be done by passing the signals through a connector and using logic in a second item to complete the action. Actions within an elementary component are carried out, however, without the details of the wiring within the chip being described. While it is usually expected that a complex chip (such as the AMD-2901) will comprise the elementary level, such chips may be considered a group item, i.e. they may be broken into smaller parts. In the latter case, of course, the component parts must be connected to each other with a bus.

3.6 THE MEMORY STATEMENTS

The system provides up to five memories which can be used as read only or random access and may be copied from an existing file. The memory size clause is shown first in the examples and the remaining clauses used in the memory statement -- PIN, REGISTER, TIME, IF, ON PULSE, will be discussed in section 3.15 since their form is common with several other statements. In the examples which follow, the clauses are placed in proper order. Note that memory may be addressed directly or the address may be given on the pins.

```
MEMORY: MMEM SIZE = 2048 X 8;
     PINS ARE NUMBERED (1-42) AND NAMED D1,D2,D3,D4,
     D5,D6,D7,D8,ADR1,ADR2,ADR3,ADR4,ADR5,ADR6,ADR7,
     ADR8,ADR9,ADR10;
     REGISTERS ARE ABC (16), DEF (24), GHI (8, 16);
     TIME = 2:1:3:4:2;
     IF PINS (1-3) EQ 131Q THEN MEMORY (PINS(16-24))
        = PINS (4-6);
     TIME = 2:1:3:4:3;
     ON PULSE MMEM (672) = PINS (25-32).

MICROMEM: MICROM SIZE = 12 * 64; PINS NUMBERED (1-80);
   AND NAMED D1,D2,D3,D4,D5,D6,D7,D8,D9,D10,D11,D12,
   D13,D14,D15,D16,D17,D18,D19,D20,D21,D22,D23,D24,
   D25,D26,D27,D28,D29,D30,D31,D32,D33,D34,D35,D36,
   D37,D38,D39,D40,D41,D42,D43,D44,D45,D46,D47,D48,
   D49,D50,D51,D52 D53 D54 D55 D56 D57 D58 D59 D60,
   D61,D62,D63,D64,A1,A2,A3,A4,A5,A6,A7,A8,A9,A10,
   A11,A12,RD1,CNT1,PLUS,GRND;
   REGISTER MAR (1-16), SPCREG (1-10);
   TIME = 4:2:0:4;
   IF PIN 2 EQ 1 AND PIN 22 EQ 1 THEN PINS (D1, D2, D3,
      D4, D5, D6, D7, D8) =  MEMORY (PINS (9-16));
   ELSE MEMORY (PINS (9 - 16)) = PINS (1 - 8).
```

3.7 THE PERIPHERAL STATEMENT

The peripheral statement describes a port, terminal, or other input/output device. The statement contains the format clause which is unique to the peripheral statement and uses all of the clauses which are common to logical device descriptions, explained in section 15. At the user's option, the data passing in and out of a peripheral may be taken from or sent to a file. This is done in the simulator, not in the hardware description, by "tieing" the peripheral to a work file as explained in section 6.9.12.

```
I-O: TERMINAL1 PINS NUMBERED (1-22);
FORMAT BINARY;
```

```
REGISTERS ARE KEY-CD (10), KEY-CD2 (19);
```

The peripheral statement has a special form for the replacement expression which is part of both the "IF" and the "ON PULSE" clauses. In the peripheral statements, the reserved words INPUT and OUTPUT are substituted for the appropriate side of the expression. Note that 'OUTPUT' displays the value at the terminal, with allocation of bits starting on the right. When "INPUT" is encountered during simulation, the system will wait for an appropriate input from the terminal after displaying a message requesting input.

```
TIME = 6:3:1:0:4:1;
IF PIN (7) EQ 1 AND PIN (8) EQ 1
THEN OUTPUT = PINS (11-18).
IF PINS (7-8) EQ 01B THEN PINS (11-18) = INPUT.
```

Since data must be in ASCII format to be displayed on the user terminal, and ASCII format is received from the user terminal, conversion to/from ASCII is required if the data is to be legible to the user. In the user's object machine the data may have a wide variety of formats. Simcal provides conversions at input/output time (to the user's terminal) for several of these. The data format determines the number of bits used to form a character at the terminal during an OUTPUT operation and the number of stored bits resulting from an INPUT operation. Binary will use one bit per character; octal will use three bits; BCD and HEX will use four bits per character; ASCII and EBCDIC will require eight bits per character. On input, a data item described as binary will convert each input character (must be a 0 or 1) to a single bit for storage. On output each bit is converted and displayed as a zero or one.

If the format is ASCII, then each eight bits will be used to form a character on the screen on output; fragments will be disregarded in this case. On input, the system will be expecting an input character for each eight bits or part of eight bits. If the input is binary then the user should enter a 0 or 1 for each bit position followed by a carriage return. If the pins or register length is not an even multiple of the format requirement, allocation to the character will start on the right (usually the low order value) and proceed to the left. The right side is considered to be the higher numbers in the pin expression. e.g. in the expression "PINS (4-10)", pin 10 would be considered on the left side. The available formats follow.

BCD	On entry, a character will be converted to 4 bits. on display, 4 bits are used to form 1 character.
GRAY	Converted to and from the gray scale.
OCTAL	Converted to/from octal code, 3 bits/character.
ASCII	These values will be sent/received as they are.
EBCDIC	Converted to and from the Extended Binary Coded Decimal format, 8 bits per character.
BINARY	The data will be broken into a series of zeroes and ones when displayed. On terminal entry, sufficient ones or zeros are expected to fill the request. If less are received, the high order (left side) will be padded with zeroes.
PCKED-DEC	Converted to/from the packed decimal format.
HEXADECIMAL	Data will be displayed as its hexadecimal representation, and when received from the terminal the hexadecimal will be converted using 4 bits for each character.

3.8 THE COMPONENT STATEMENT

All items at the elementary level not described as conductors, peripherals or memories must be described as components. The memory and peripheral statements are really special forms of the component statement using all of its syntax plus an additional clause in each case. The component statement is made up of the common clauses which are described in paragraph 3.15. An example follows, listing these clauses in correct order and followed by an actual example.

```
CHIP: CHIPNAME
      PIN DESCRIPTION CLAUSE;
      REGISTER DESCRIPTION CLAUSES;
      TIMING CLAUSE; LOGICAL ACTION CLAUSES;
      TIMING CLAUSE; LOGICAL ACTION CLAUSES;
      TIMING CLAUSE; LOGICAL ACTION CLAUSES.

CHIP: FD14 PINS ARE NUMBERED (1-42);
      REGISTERS ARE A(8), B(8), C(8), MRR (16, 16);
      SUBREGISTERS OF F ARE SIGN, ZERO, F3(3), F2, F1;
```

```
CASCADE H, L, INTO HL; CASCADE B, C, INTO BC;
TIME = 2:1:7:3:4:0;
ON PULSE PINS (12-14) = 0.
IF PINS (1-8) EQ 10010001B THEN A = A + B,
ELSE PINS (1-8) = 4BH;
TIME = 2:1:8:0:1:4;
IF PINS (1-8) <451Q THEN A = A - B;
TIME = 3:4:2:1;
IF A EQ B THEN PIN 4 = 0 ELSE PIN 4 = 1.
```

3.9 THE WIRING STATEMENT

The wiring statement provides the logic path between various components. BUS1, in the example below, is connected to several terminals, each connection being treated individually with a clause, or a series of individual statements. In the case of connectors, the number refers to the wire number and in the case of the component the pin number.

Note that as BUS1 is connected to each item that only those items of concern are listed. The order of the wire and pin numbers is of no consequence, only the mapping of one onto the other. Each list is enclosed in "<...>", and a range is enclosed in "(nnn)". Several connections can be made using a single or series of statements can be used as shown below:

```
CONNECT BUS1 TO BOARD1{(1-120)}{(5-95) 3,1,2,(96-120)};
             TO BOARD2 {(1-95)} {(1-95)};
             TO BUS2 {6, 19 (22-25)} {(1-6)}.
CONNECT BUS1 TO BOARD4 {(21-120)} {(1-100)}.
CONNECT BUS1 TO BOARD5 {(25-50),57 62 (81-89)}{(1-37)}.
```

3.10 THE STORE STATEMENT

The store statement allows the user to store syntactically correct statements in a personal library on the DEC-20, thus providing assistance for those statements which involve many lines of code and significant amounts of preparation time. The user must first choose a suitable library name for each item to be stored, which may be the same as the chip name or entirely different. The library names may have up to 30 characters, beyond which they are truncated, while component names are limited to twelve characters. The first and last lines of the store sequence are shown in the example.

The description, including timing, is placed in the library during compilation prior to use in the copy statement, although timing may be modified when the item is copied. If an error occurs during the process of storing the statement, the store operation is halted and the partial description is deleted. Since an entry with an already used name will not cause deletion of the old description. The user must first delete the old name from inside the library module. The library is primarily intended for storing (and using) descriptions of chips, but may also be used for assemblies. An example follows.

```
     STORE AS AMD-9301-BIT-SLICE-CPE.
      ...
     The description of the component goes here.
      ...
     END OF AMD-9301-BIT-SLICE-CPE.
```

The user may not wish to bother with timing when the unit is placed in storage since the timing will probably be different at the time of use than would be entered in the description. To avoid this the user can enter "TIME = *;" for the time. When the items are withdrawn, the correct time must be substituted in the description.

3.11 THE COPY STATEMENT

The copy statement is used to retrieve one or more copies of item descriptions from the user's library, making timing corrections for insertion into the code being compiled. Where a single copy is needed, the syntax is:

```
  COPY AMD-9301-BIT-SLICE-CPE,
  TIME = 1:3:2, 1:3:8, *:*:2.
```

In this case the item stored as "AMD-9301-BIT-SLICE-CPE" will be retrieved and will be compiled as if the code were in the incoming hardware description. If either the requested name is not found in the library, or the modified name is not found in the hierarchical description an error message will result. Since the syntax of the stored item was checked when placed in the library, there should be no syntactical errors. The four time slices shown in the example (between 1:3:2 and 1:3:8) will be substituted for the times found in the item description in the library in the order shown. If the description contains fewer or more than four time increments, an error message will result.

When there are several of the items required in the design of the object machine, it will be necessary to add suffix numbers to the identifiers to make these names match the names originally entered in the hierarchy statement. The copies required will be entered and the compiler will then provide the suffix numbers. The entry is as follows:

```
   COPY 2 EACH AMD-9301-BIT-SLICE-CPE,
   TIME = 3:6:0, 3:6:6, *:*:2; TIME = 3:7:0, 3:7:3.
```

The above statement will cause the compiler to copy the description of the chip from "AMD-9301-BIT-SLICE-CPE", attach "01" and "02" to the identifying number, (unless these have already been used), and apply the two time replacements to the copied items. The times on the first item copied will be 2:6:0, 2:6:2, 2:6:4, 2:6:6. The times on the second item will be 3:7:0, 3:7:1, 3:7:2, and 3:7:3. The user should be sure the revised names exactly match those already entered in the hierarchy statements.

In addition to the method described above, the user may also suffix his own 2 letter (or number) designation to the embedded name or replace the name with a new one. This example shows an increment of 3 units in the time setting. The compiled name will be the identifier found within the library description with the "AB" attached and in the second case the new name, "ABC" will be substituted.

```
      COPY 1 EACH AMD-9301-BIT-SLICE-CPE SUFFIX AB,
         TIME = 2:3:6:0, 2:3:7:2, *:*:*:3.

      COPY 1 EACH AMD-9301-BIT-SLICE-CPE REPLACE WITH ABC,
         TIME = 2:3:6:0, 2:3:7:2, *:*:*:3.
```

In the use of STORE and COPY the user should carefully differentiate between the library name, the name stored within the library description and the name following compilation. e.g., the name above, AMD-9301-BIT-SLICE-CPE, is the library name not the name of the component. If the stored name were "AMD-9301", then "AMD-9301AB" would be the compiled name. The addition of suffixes must be carefully coordinated with the descriptions previously provided in the hierarchy statements so that spelling of the names are the same.

In some cases, the user may wish to add the code from the library to the hardware description file so that the description file, while longer, is self-contained. This may be done in one of two ways. The first method uses the TOPS facility to copy the material from the library file into the hardware description file. The DEC manual should be consulted for a full description of this method.

The second method uses the Simcal system and is implemented as follows. Include the COPY statement in the hardware description at the proper location. When turned on, the SIMCAL.PRT file will contain the original description and code as it is copied, including error messages if any. The user may then enter SIMPRT.DAT after leaving the executing Simcal and "clean up" the file renaming it SIMHDW.DAT.

Copy statements may not appear in the items stored in the library. The user may work around this, however, by including the copy statement as a comment. After the first level is copied into the save file, the comment delimiters are removed and the the description is compiled again. This can be repeated to any depth desired.

3.12 THE NUMBERING SYSTEMS AVAILABLE FOR USE

The values to be stored on pins or in registers may be expressed in binary, octal, decimal, or hexadecimal. Only decimals may be used for other application such as pin, wire, or register number. The following symbols will be recognized by the software support system for values actually on the connectors and in register and memory of the object machine. Only decimals may be used for other purposes such as index values.

```
00010101B  (binary)
4927O (<--OH, not ZERO)  or 4927Q  (octal)
8245 or 8254D (no symbol will be treated as a decimal)
0ACE348H (hexadecimal, the value must start with a digit)
```

3.13 THE START STATEMENT

The start statement is used to set starting values in registers, memories, pins and/or buses so that the machine will have proper values stored at actual simulation beginning. In the example, zeroes are placed in the recorded values of the named components and the clock is set to zero. The user should place this statement last to assure that the named components have already been compiled. Once simulation is performed, the values are destroyed, but may be easily reset by entering "START" in the simulation environment. Any legal number, octal, hexadecimal, decimal, or binary is permissable to specify the values.

```
START:  ADRBUS=0; ABCD = 1010B;  REGC = 0AFH;
        CNTL6 = 1; RD =1; TIME = 0:0:0:0:0:0:0:0.
```

3.14 CONTROL OF TIMING

Simcal requires that a time slice be assigned to all logical actions which take place in either the memory, peripheral or component statements. The clock used is broken into eight (8) levels, each of which may be any value from 0 through 9, with the larger segments on the left and the smaller segments proceeding to the right. A colon separates each segment. It is not necessary to use all eight (8) segments; use only sufficient number to meet the need for the system being described.

Before a logical action is described within a component statement, a time for the action must be assigned. While compiling, this time will remain constant, until changed by another time clause entry but each component must have a time clause before the first logical clause. Actions in different components, which have the same time assigned are presumed to be running in parallel. In actual simulation they will be executed in the order entered.

When items are being sent to the library and the time of use is unknown, or partly unknown, an asterick may be substituted for one or more digits, e.g. "*", "5:6:2:*:*", and "*:2:1" are all acceptable for library storage. The user resets the segments when copied from the library. An asterisk is used in the areas to be copied from the file and numbers where values are to be inserted. In most cases the user will show the same number of segments in the copy statement as are in the original description. If they differ, however, the system wiil accept the value but provide a warning message.

CONCURRENCY: To assist the user in simulating concurrency, Simcal will not transfer internal values onto the connectors unless both of the following conditions are met:

A TIME CHANGE OCCURS.
SIMULATION IS PASSED TO THE NEXT COMPONENT.

When both of these conditions are met, the simulator then moves the value from the pins to the conductor and resets all pins in all components connected to the conductor. If conflicting values have been provided during a single time period, the simulator will "break" and warn the user of the conflict. Thus, if components A and B have changed pin values to wire N during period X, one being high and the other low, the result would be unpredictable in the hardware, hence the simulator will issue a warning. The user may supply a value for the wire during simulation but he should also correct the design problem.

3.15 COMMON CLAUSES AND EXPRESSIONS

The following clauses are used to describe logical components, memories, and peripherals.

3.15.1 PIN CLAUSES

The PIN clause provides the compiler with the number of pins on the component and allows the user to also enter names for the pins. Its form is similar to the connector statement as shown in the following example. The pin numbering must consist of a range starting with the number one and not exceeding 999. The names are optional, may have up to six letters or numbers, however a warning is displayed if partial name lists are supplied.

```
PINS ARE NUMBERED (1-64) AND NAMED A1, A2, A3, A4;
```

3.15.2 REGISTER CLAUSES

The grammar provides capability to define a single or a group of registers, to cascade registers into longer registers and to define portions of registers as subregisters. The first value within the parentheses following the register name always represents the length in bits, the second, if present, shows the number of registers in the group. The upper limit on bit length is 99 and the maximum number of registers is only limited by storage in the software support system. Examples follow of register definition, cascading, and creation of subregisters.

```
REGISTERS ARE SP (16), GENPP (8, 8), H(8), L(8), F(8);
CASCADE H, L INTO HL;
SUBREGISTERS OF F ARE SIGN, ZERO, F5(3), F2, F1, F0;
CASCADE HL,F1 INTO HLR;
```

3.15.3 TIME CLAUSE

The time clause assigns a specific time slice for a series of events to occur within a component and therefore must preceed the the description of the logical action to take place within the chip. When the order of events are of no consequence within the chip, a single time clause is adequate, and the events will be carried out in the order entered. However the user should realize that during simulation, breaks can only occur at the completion of the operations for a given time slice or for a given component. e.g. the time "2:3:4:5:6:0:0:0" may be set aside for a given component with several operations. The user may further divide the last three areas into a group of two, further divided into a group of five and then a group of two. All of the possible segments may not be used. The description might be as follows.

```
CHIP: EXAMPLE
 . . .
TIME = 2:3:4:5:6:0:0:0;
Logical descriptions A goes here.
TIME = 2:3:4:5:6:0:0:1;
Logical descriptions B goes here.
TIME = 2:3:4:5:6:0:1:0;
Logical desriptions C goes here.
TIME = 2:3:4:5:6:1:3:1;
Logical descriptions D goes here.
```

The user is now able during simulation to "break" following any of the four sets of operations and examine the value of pins or registers. If another item had also been given the same time, then the breaks could have been made following either component.

3.15.4 LOGICAL ACTIONS - THE CONDITIONAL EXPRESSION

In the simulator, logical action may be executed as the clock reaches a certain time, explained in the "ON PULSE" expression, or at a certain time events depending on the outcome of a conditional expression. The conditional expression, starts with "IF", may be followed with an ELSE clause. The IF-ELSE clauses however, may not be nested. The values on the pins or registers may be checked using the following series of relational operators:

```
EQ    NOT EQ    <    NOT <    >    NOT >
```

Note that the "=" sign is reserved for the assignment function and the 'EQ' must be used as a conditional. The values on either side of the relational operator may be either a numeric value (in binary, octal, hexadecimal or decimal), a set of pin values, a register value or a memory value. If pins are to be checked, then the user enters "PIN" or "PINS" followed by either a single number (decimal only) or a range enclosed in parentheses. The decimal values which follow the register names or the reserved words "PIN" and "PINS" will cause the simulator to determine the values on the pins and test them against the values on the other side of the relational operator. Several forms of the expression is shown in the example.

```
IF 12Q EQ HL THEN ...           (where HL is a register)
IF PINS (42-49) EQ OFFH THEN...(the value on pins 42-49
    must all have a value of 1 for the test to be true)
IF HL EQ PINS (1-16) THEN ...
IF PIN 21 NOT EQ 0 THEN...

IF PIN 21 EQ  1 THEN ...
IF PINS (30-34) EQ HL... (no good, not the same length)
IF PINS (4-44) EQ 1... (numerics padded with 0 on left)
IF H(4-8) EQ 12H THEN ...     (A portion of H is tested)
```

3.15.5 LOGICAL ACTIONS - BOOLEAN EXPRESSION

Several of the conditional expressions may be combined to form a Boolean expression. Simcal allows the use of parentheses and operator precedence is that usually found in programming languages. The ascending order is: NOT, AND, OR. Examples of their use follow.

```
IF PIN (5-8) EQ 1010B OR PINS (5-8) EQ 1011B THEN ...
IF NOT (PIN 4 EQ 1 AND PIN 33 EQ 1) THEN ...
IF (HL EQ 0ABH OR PIN 14 EQ 1) AND PIN 16 EQ 1 THEN ...
    (pin 16 must be true for statement to be true)
IF PIN 5 EQ 0 AND PINS (20-27) EQ 0BCH THEN ...
```

3.15.6 LOGICAL ACTIONS - THE ON PULSE CLAUSE

The system allows the user to execute an action each time a time setting for a logical device is reached by using the words "ON PULSE" rather than checking the condition on a pin or in a register. The reserved words "ON PULSE" are substituted for the relational expression from "IF" through "THEN". An example follows.

```
ON PULSE ... replacement expression;
```

3.15.7 LOGICAL ACTIONS - REPLACEMENT EXPRESSION

One or more replacement expressions, separated by commas, may follow "THEN" or "ELSE" of the conditional expression or the reserved words, "ON PULSE". The memory statement and the peripheral statements allow operation on the data as well as the replacement of data. The replacement expression uses the "=" sign to assign a value to the register or the pins on the left of the equal sign. If the length of the fields on the right of the equal sign are not of the same length an error will result. However numeric values used in the expression need not be padded with leading zeroes. The replacement expression can be used to move values and to logically manipulate the data as illustrated below.

```
    ... PINS (1-8) = REGA, PINS (31-46) = SP;
    ... A = A + B, PINS (1-8) = A; PINS (9-10) = A - B;
```

There are fifteen operators available, nine using two operands and the remaining six using a single operand. The operators using two operands follow.

```
        +   -   *   /   AND   OR   XOR   NOR   NAND
```

The operators using a single operand are:

```
        SL   SR   RL   RR   CMP   NEG
```

All of the operators presume that the data format is binary, that is the values are simply strings of ones and zeros. Formats such as twos complement, or floating point must be "built" by the user. The operator, "CMP" when used will simply take a value and complement it, "Negate" will convert the number to it's twos complement value. The dyadic operators have the usual meaning, the monadic operators are shift left, shift right, rotate left and rotate right, complement, and negate. The vacated positions are left as is. The operations are performed in order from left to right, without operator precedence but the order may be modified by the use of parenetheses.

4.0 THE CALSIM GRAMMAR

The formal grammar for the CALSIM language may be obtained by entering the table section of Simcal and entering "BNF". The serious student will request a copy and use it as a guide in preparing his descriptions. The grammar was developed using an LALR analyzer by LaLonde from the University of Toronto. The grammar provides starting and ending statements of the hardware description and uses a series of statements between these to specify the hardware components, connections and logical actions. Several of the productions for statements are listed below, but the reader is referred to the formal grammar for details of the subsidiary clauses.

```
<SYSTEM NAME> ::= SYSTEM NAME <IDENTIFIER>.
<END ID> ::= END OF <IDENTIFIER>.
<STATEMENT> ::= <STORE STMNT>.
              | <COPY STMNT>.
              | <HIERARCHY STMNT>.
              | <CONNECTOR STMNT>.
              | <CLOCK STMNT>.
              | <MEMORY STMNT>.
              | <PERIPHERAL STMNT>.
              | <COMPONENT STMNT>.
              | <WIRING STMNT>.
              | <START STMNT>.
```

Clauses are provided with the hierarchy statement for a series of "LEVEL" expressions. The "CONNECTOR" statement provides clauses for both numbers and names of the individual wires within a connector. The components at the lowest level can contain logical action and include the MEMORY, PERIPHERAL and COMPONENT statements each of which have several clauses. These include the PIN, REGISTER, IF, ON PULSE, and TIME clauses. The PERIPHERAL statement also uses a FORMAT clause.

Since changes in grammar may occur, it is advisable to note the version number of both the software and the grammar when entering Simcal. While Simcal may be changed when there is no change in the grammar, a change in the grammar will almost certainly result in a change in the software. The current version of the grammar is printed on entry to the system.

5.0 PREPARING THE DESIGN AND HARDWARE DESCRIPTION

5.1 THE VARIOUS USES FOR THE CALSIM/SIMCAL SYSTEM

The Calsim/Simcal system may be used in a variety of ways from elementary circuit studies to various aspects of bit slice design studies. Suggested uses are shown below and examples of these uses will be found in the Appendix 3.

* Study of basic component operation, such as flip-flops, adders, multiplexers, latches and registers.

* More complex items such as shift registers, decoders, UARTS, and programmable interface chips.

* Study of major portions of a circuit such as the ALU.

* Representation and simulation of CPU chips such as the 8080, Z80, DEC-11, Z8000, or 6800.

* Study of the operation of a set of chips to perform microprogrammable operations such as the AMD-2900 series.

* System development using microprocessors.

* Verification and proving a hardware design.

* Assisting development of microprogrammable instructions.

5.2 DISCUSSION OF EXAMPLES

The series of examples progress from a simple flip-flop to complex logical devices. Each of these may be tried in the machine and are available to the user through the TOPS environment by requesting <CS.Bnnn>SIMHDW.DAT. The unwanted items can then be deleted from the file. The student should start with the more simple circuits. Each example in the Appendix includes:

1. A description of the machine including a diagram.

2. The Calsim program listing produced at compilation.

Each example can be used to increase the students understanding of computer organization while learning to use the Calsim/Simcal system.

5.3 PREPARING THE DESIGN AND BLOCK DIAGRAMS.

The first step is to prepare a block diagram of the machine, being careful that all of the connectors between components are detailed. Be sure that a clear definition of the pins on each component and the registers in each chip are defined.

In most cases the chip descriptions will not contain a definition of any working registers not available to the user, however this should not deter the user from creating such registers if it makes the job of logic execution easier. CPU chips commonly contain working registers not shown in the manufacturers literature since they are of no interest to the user. It may also be advantageous to group the components prior to beginning the actual entry.

5.4 PREPARING DESCRIPTIONS FOR PERSONAL LIBRARY.

If the design contains more than one of a single chip, it is probably better to write the description of these chips first and place such items in the library. This is done by entering only the chips to be stored immediately following the system name statement. Caution -- the Simcal system must have the system name statement at the beginning of the description or it will not operate properly. When the chips to be placed in the library have been written and debugged, they can either be entered in a run prior to the point of use or can be entered as a separate run. If part of the same run, it is best to use the "store" statement just after the system name.

5.5 PREPARING THE CALSIM DESCRIPTION.

When the system under study is fully designed, the description should be written in the order shown here. If another order is used, the description may be satisfactory even though in most cases, error messages will result. These messages may be true at the point written, but corrected in subsequent descriptions, however the system does not check for this. The recommended order is:

SYSTEM NAME STATEMENT.(Required to be the first statement)
STORE STATEMENTS.

```
HIERARCHY STATEMENTS STARTING WITH LEVEL 1.
LEVEL 2 HIERARCHY STATEMENTS, LEVEL3, ...
CONNECTOR STATEMENTS.
TIME STATEMENT.

MEMORY, MICROMEMORY, AUXILIARY MEMORY STATEMENTS.
   SIZE CLAUSE;
   PIN CLAUSES;
   REGISTER CLAUSES;
   TIMING/LOGICAL ACTION CLAUSES;

PERIPHERAL STATEMENTS.
   FORMAT CLAUSES;
   PIN CLAUSES;
   REGISTER CLAUSES;
   TIMING/LOGICAL ACTION CLAUSES;

COMPONENT STATEMENTS.
   PIN CLAUSES;
   REGISTER CLAUSES;
   TIMING/LOGICAL ACTION CLAUSES;

WIRING STATEMENTS.
START STATEMENTS.
SYSTEM END STATEMENT.
END OF DESCRIPTIONS STATEMENT.
```

5.6 COMPILATION AND TABLE EXAMINATION.

Be sure the file containing the description has been named SIMHDW.DAT and on leaving the file, the file is closed with "EU", leaving the file unnumbered. Compilation will not be performed on a numbered file. If compilation is performed at a terminal, copies of the output may be retained by use of "PHOTO" on the DEC system or by use of SIMPRT.DAT which is part of the Simcal system. When compilation is error free, the tables should be requested and examined to assure the organization is that expected. Entry of "ALL" in the "TAB" environment will list all of the tables of interest. Read-in of memory content is performed next followed by simulation.

5.7 PREPARING PROGRAM, MICRO AND OTHER MEMORY CODE.

The program code must be prepared from whatever resources the user has available. It is suggested that the output of the assembler be saved and the code be manipulated into the pattern described here. The format for Simcal is as follows:

ADDRESS IN HEX   SPACE   64 CHARACTERS (32 BYTES)

The Simcal system will accept each line independently of the others, allowing the address to start at any point. If the address of the line is exactly 32 bytes greater than the address of the preceeding line, spaces can be used in the address location and the system will calculate the address. The addresses outside of those established during compilation will result in an error. The memory code is entered following compilation and should specify the same memory size as that given in the hardware description. Simcal reads the memory from the file until either reaching the end of the file or satisfying the requested amount of storage. If the requested amount of storage and the file length are different an error message will result.

The micro and auxiliary codes are entered as a string of binary characters starting at zero position. The entire micromemory need not be filled. but the entry must be no more than 99 columns.

5.8 OBTAINING THE STATUS IN THE SOFTWARE.

At the entry level to Simcal, determine that all is ready to simulate by entry of "STATUS". This will show the status of memory file entries and compilation and will remind the user of items which may have been forgotten. An example follows.

```
> STATUS <CR>
SYSTEM STATUS:
  COMPILATION COMPLETE WITH NO ERRORS, 5 WARNINGS.
  MAIN MEMORY READ IN.
  NO MICROMEMORY READ.
  NO AUXILIARY MEMORY READ.
>
```

5.9 SIMULATION.

Prior to the use of the simulator, section 6 of this manual should be carefully studied followed by simulation of one of the simple devices found in the examples. Without a basic knowledge of "MENU", "DISPLAY", "BREAK", "STEP", "RUN" and other terms, the simulation will be meaningless. When familiar with these terms and their use, develop a test plan. This usually is step-wise detailed listing of the checks which are to performed starting with a top-level verification and working downward.

The student is cautioned against using an ad hoc plan of testing in which he starts simulation and "tries it out". This approach to testing provides little assurance beyond that of knowing that the system will actually run, but more frequently the system will not run at all and this produces discouragement.

On entry to the simulator, the user will receive the "SIM>" prompt showing that the simulator driver is awaiting a command. The user should select the "MENUS", "BREAKS", and "DISPLAYS" with care. The maximum interval should be set fairly short and a step-wise execution tried. The following is suggested as an orderly approach to carrying out simulation:

1. Carefully plan the action to take so that the testing is top down. The first action will be to determine if the machine can "stay alive". Tests of more detail are then planned, taking one section of the hardware at a time.

2. Select the values you will wish to display, and place the items in groups using DEFINE MENU.

3. Select the component/time when these groups of items will be displayed while simulation is proceeding. Use the DEFINE DISPLAY to record the selection.

4. Select the component/times when breaks will occur and the menus which will be displayed. Use the DEFINE BREAK command to record the selection.

5. Set the maximum time increment to activate the automatic break.

6. Print the listing of menus, displays, and breaks and modify as required.

7. Try to execute a few steps at the highest level in the system.

8. Follow the written test plan using a terminal with a printer (or using PHOTO) so a written copy will be available.

9. When errors are discovered be sure that they are carefully recorded. Try to duplicate the error.

10. Develop the corrective action carefully in writing.

11. Carry out the corrective action detailing each action taken and recording all anomalies.

6.0 USING THE SIMCAL SOFTWARE SUPPORT SYSTEM

6.1. SYSTEM ENTRY

The user begins a session by entering from the terminal:

```
TAKE SIMCAL.CMD  <CR>.
```

The command file then binds the support files to a copy of the SIMCAL object file (SIMCAL.REL) which is copied from <CS.Bnnn-mm> and starts system execution. On entry, the version number of Simcal is displayed and the user is given an opportunity to display and read a brief set of instructions. At the entry level, as at all levels, the user has both "HELP" and "?", unique for each environment, available to obtain either an explanation of the environment or a list of commands acceptable in the environment. The commands available at the entry level of Simcal are:

HDW	(HARDWARE compiles the user's design into tables).
TAB	(TABLES is used to print tabulations of hardware).
PGM	(PROGRAM is used to read the user's program code).
MIC	(MICROCODE is used to read the user's microcode).
AUX	(AUXILIARY is used to read other memories).
LIB	(Provides access to saved hardware descriptions).
DOC	(DOCUMENTATION provides access to documentation).
STA	(STATUS lists the status of the system).
SIM	(SIMULATE is used to enter the simulation driver).
HELP	(HELP at top level an overview of SIMCAL).
QUIT	(QUIT returns the user to TOPS level of DEC-20).
?	(Prints commands available in entry area).

6.2. ACTIONS IN SIMCAL PROVIDED FROM THE DEC-20 SYSTEM

The rubout and all of the control commands available at the TOPS level and inside the DEC editor are also available inside of Simcal. Several control commands available are:

CNTL C	Interrupts program action, returns to TOPS.
CNTL O	Stops the display, program execution continues. Also restarts the output to the terminal.
CNTL S	Stops the program (so screen can be read).
CNTL Q	Restarts the program (used with CNTL S, above).
CNTL U	Deletes the entire line being entered.
CNTL W	Deletes the last word entered.

CNTL R Rewrites the line being entered.
CNTL T Provides status of the executing user program.

The DEC-20 utility PHOTO may be used to record the output to a user's terminal onto a user named file. This is done at the TOPS level by entering "PHOTO" and providing a file name. After the session through Simcal, the user again enters "PHOTO", ending the recording session and making the user named file ready for use. The data can then be listed offline or read at the terminal at a more leisurely pace.

6.3 USING THE COMPILER

The compiler environment is entered from Simcal by entry of "HDW" The user wishing to compile the hardware file enters the following: "C,P <CR>". Following this the machine will take two actions:

(1) Locates a file of the CALSIM grammar in <CS.Bnnn> library. This file is named SIMLLR.DAT and is read by Simcal into the user program memory. This will be done once each time the user enters Simcal. The grammar version will be displayed as it is read.

(2) Locates SIMHDW.DAT in the user's workspace, reads and compiles the description.

CAUTION: SIMHDW.DAT must be UNNUMBERED.

The user may choose, in response to questions, to display the listing and messages or to record the data on a print file, or do both. If the run contains errors, these must be corrected in the original text and compilation repeated.

6.4 READING IN MEMORY AND DATA FILES

The format and the method of reading into Simcal the three types of memory files has already been discussed in section five (5). The environments in Simcal used to read the memory content are:

> PGM
> MIC
> AUX n (n = 1, 2, or 3)

This same command is used in all three environments. The format for starting the read process is as follows:

```
    PGM>  READ
    MIC>  READ L X W (L = length of file, W = width of file)
    AUX>  READ  L X W
```

6.5 USING THE LIBRARY OF COMPONENTS

6.5.1 LIBRARY ACTIONS WITHIN THE COMPILER

When the user has determined the components which will occur in his design, he may start a file of their descriptions. These should be prepared from the manufacturers data sheet and checked for correct format through the compiler. The user may also wish to simulate the operation of the component prior to use. When the descriptions are complete, with no errors, the STORE and "END OF" lines should be added to each and the descriptions placed in the user's library as shown below.

```
     SYSTEM NAME STORE-SOME-CHIPS.
     STORE AS AM2914ENCDRE /* This is an AMD-2914 encoder */
     CHIP: AM2914
           PINS ARE (1-40) AND NAMED P3, M3, GAREC, ...
           ...
           ...
           ...
     END OF AM2914ENCDRE.
     END OF STORE-SOME-CHIPS.
     END OF DESCRIPTIONS.
```

When copies of the encoder are used, the copy statement is placed in the Calsim description of the hardware at the correct location. The compiler opens the library file retrieves the item, replacing the time as instructed in the hardware description. If more than one unit is requested, copies are made for each one requested.

6.5.2 LIBRARY ACTION OUTSIDE OF THE COMPILER

The user may also enter the library environment and check on the contents of the file. He may ask for a specific item, a listing of all items, or a complete listing of the whole file.

```
     LIB> LIST <CR> (supplies a list of the items stored.)
     LIB> FIND <identifier> <CR> (finds the item if present)
               (The user can then either LIST or DELETE.)
     LIB> ALL  Lists the names and text of all entries.
```

```
LIB> QUIT <CR> (returns to entry level)
LIB> ? <CR> (lists these commands)
LIB HELP  <CR> (Explains the use of the library)
```

6.6. PRINTING TABLES PRODUCED FROM YOUR DESCRIPTION

Compilation produces several tables which are used to drive the simulator. These tables are formatted and displayed using the following commands in the "TAB" environment.

HIER	A listing of the components in the description with pointers showing their relationship.
WIRE	A listing of the wire numbers and names. The hierarchy table will have pointers to this table.
REGI	A listing of registers in the system.
TIME	A table showing the components arranged by time.
SIZE	Will show the size of the tables.
LOG	Will provide a listing of the logic in the system arranged by time.
ALL	Will print all of the tables listed above.
PROG	A listing of the program code read in.
MIFL	Will list the contents of the microprogram file.
AUX n	Allows the user to list of an auxiliary file.
HARD	Prints the listing of the hardware description.
STAT	Status, also available at entry level.

6.7 OBTAINING DOCUMENTATION AND HELP

Documentation may be obtained by entering the "DOC" environment and executing a request. The documentation may be printed at your terminal or at the main printer. The commands available follow.

```
DOC>   MANUAL     (This manual)
DOC>   BNF        (Compiler grammar)
DOC>   HELP       (Explains documentation available)
DOC>  ?          (These commands)
DOC>  QUIT       (returns user to entry level)
```

6.8 GETTING THE STATUS

The status of the user's actions may be obtained by entering "STATUS". This tells the user whether he has acceptable compilation and memory files in place. The user is returned to the Simcal level after the information is displayed. An example follows.

```
> STATUS <CR>
  COMPILATION COMPLETE WITH NO ERRORS, 5 WARNINGS.
  MAIN MEMORY READ IN.
  NO MICROMEMORY READ.
  NO AUXILIARY MEMORY READ.
```

6.9. USING THE SIMULATOR

6.9.1 GENERAL DESCRIPTION OF THE SIMULATOR

Before attempting simulation, the user should be sure that the following actions have been completed by entering "STATUS" in either "SIM" environment or the entry level of Simcal.

A good compilation of the design has been achieved.
The program file has been read into Simcal.
The micromemory file has been read in (if needed).
All required PROMS have been read in as AUXMEMORY.
No conflict exists in memory sizes.
Data files, if required, have been prepared.
A plan for simulation has been developed.

The user enters the driver for the simulator by executing "SIM" at the entry level. In addition to the simulation commands, the driver will also respond to 'QUIT', '?', and 'HELP'. The latter will show examples of the commands available within the module if "HELP" is followed by one of the Simulator Driver command words.

The simulation commands may be divided into three kinds - those which preset controls prior to simulation, those concerned with setting and displaying values on connectors in components, and those causing actual execution to begin. Following the discussion of protection in the next paragraph, each type will be discussed in turn. All of the items in the following discussion are lost at the end of each session, therefore the user may wish to keep copies of the items he has established.

6.9.2 PROTECTION FROM AN ENDLESS LOOP (SET INTERVAL)

When control is passed from the user interface to the simulator, a counter is reset to zero and is incremented each time an elementary timing signal is processed. If more than one component has the same time, then each component will be counted as one of the time increments in the counter. When the value of a preset interval is reached, control is returned to the user, i.e. the simulation system "breaks". Without this protection, the user would not be able to regain control of the system except by control C, which would place the program back in the TOPS environment of the DEC-20. The default value for the interval is 250, but may be reset by the user to any value from 1 through 999, as shown below:

```
     SIM> SET INTERVAL = 100 SHOW n, m, ... <CR>
     SIM> SET INTERVAL = 500 SHOW 5, 10, 11 <CR>
```

Usually the value of the interval is set below 250 as a safety measure to return control to the user if the intended break points are missed. When the automatic break occurs, all menus referenced by the numbers following the reserved word "SHOW" will be displayed. The menu and break are discussed in the sections which follow.

6.9.3 DEFINING THE MENU, DISPLAY AND BREAK TABLES

The system provides storage capability for three lists with twenty items in each list. These lists are entered by the user in an interactive mode. The lists are (1) groups of identifiers which will be displayed at the terminal at one time (called menus), (2) times and/or active component where the execution (simulation) is to "break" and (3) items and/or times where values are to be displayed. The following command shows the syntactical form to start the collection of data in any one of the three tables.

```
     SIM> DEFINE <TYPE> n <CR>  where TYPE may be MENU,
                   DISPLAY, or BREAK and n is 1 through 20.
     SIM> DEFINE MENU 12 <CR>   (one entry only)
     SIM> DEFINE BREAK 4 <CR>   (one entry only)
     SIM> DEFINE DISPLAY <CR>   (system uses next number and
SIM> DEFINE MENU   <CR>   continues requesting entries)
SIM> DEFINE BREAK 6,7 <CR> (request two entries)
```

The value of "n" may be from 1 through 20 to designate the number the item will use as an identifier. If the type is MENU, the collection mode will accept up to five identifiers per entry one at a time. The identifiers may be any item with a specific value such as pins, registers, memory location or a connector. The DISPLAY and BREAK modes are entered in the same way, but a single component, specific time or both specifies the component/time which is to cause either a display or break. Up to five menu numbers may be associated with the break or display items. These definitions are lost when the user leaves Simcal.

6.9.4 LISTING MENUS, BREAKS, AND DISPLAYS

The user will need to know what has been placed in the three tables so that corrections can be made and proper use of the tables can be realized. The following commands show how all or part of the tables may be displayed.

```
LIST <TYPE> m <CR> where m may be a number, a range
                       such as n - n, or "ALL".
LIST BREAK 1 - 10 <CR>
LIST DISPLAY 12 <CR>
LIST MENUS ALL <CR>
```

6.9.5 DELETING(UNDELETING) ITEMS IN THE TABLES

The user may clear the items in the tables, making the table more legible, and also making the storage available for entry of other items by use of the DELETE command. The form is that of the previous command.

```
DELETE <TYPE> m <CR>(The values are the same as above).
```

The DELETE command does not actually destroy the memory image but sets a key allowing the item to be written over. During the same session, the user can reverse the process by entering "UNDELETE" followed by the type and number.

6.9.6 ACTIVATE/DEACTIVATE THE BREAK AND DISPLAY ITEMS.

The break and display items may need to remain active or be set to a dormant state until needed, depending on the actual work in progress. This is accomplished as shown below.

```
ACTIVATE  <TYPE> m, (m =number or range, TYPE is BREAK
                     (or DISPLAY.)
DEACTIVATE (TYPE) m - n (DEACTIVATE m through n)
```

6.9.7 SETTING THE FORMAT FOR INPUT and OUTPUT.

The data transfers during simulation have been set when the description of the hardware was written and those will be followed when the simulator is in control. The user should keep in mind that printing and acceptance of data under program control, although at the same actual terminal being used to control simulation, is controlled from the hardware description, whereas the simulation is under the immediate control of the user. At every break, the Sim-driver is in control and the user is free to set the format which is to be used to receive and to submit data and change values in storage. The format for this command is:

```
SET FORMAT TO BIN <CR> (HEX OCT DEC ASCII also avaialble)
```

6.9.8 DISPLAYING VALUES IN THE SIMULATOR.

Any stored value in the machine can be displayed using any format desired. The default is hexadecimal, but can be changed by use of the SET command just discussed. The command to display stored values in registers, pins, wires or memory is shown in the following example:

```
SIM> SHOW mmm <CR> where mmmm may be any register.
SIM> SHOW mmm (nn) <CR> where nn is the number of a
     dimensioned register or memory.
SIM> SHOW PINS (34-38) OF <identifier> <CR>
SIM> SHOW TIME (shows the next time increment which is
               (to be simulated.)
```

6.9.9 SETTING VALUES IN THE SIMULATOR.

The values in the registers, on the pins, and on wires may be set using the command as shown in the following examples.

```
SIM> SET <identifier> = value <CR>
SIM> SET <identifier> (nnn) <CR>
SIM> SET TIME = n:n (n:n represents a time increment)
SIM> SET  <identifier> (nnn)   <CR>
SIM> SET PINS (26-32) OF <identifier> <CR>
SIM> SET WIRES (29 - 64) OF <identifier> <CR>
```

The "value", shown above is a numeric value in binay, octal, decimal, or hexadecimal. The hexadecimal value must start with a numeral. The base is indicated by a suffix of "B", "O" (or "Q"), "D" (or suffix omitted), or "H". The value is checked to assure there is room in the component. If the numeric value is smaller than the target components, the simulator will assume the higher numeric value in a group of pins or wires is the low order. e.g., If "SET PINS (30-37) = 3", then pins 30-15 will be set on one and pins 32 - 37 will each be set to zero. All values in the machine (except memories) may be set to either 1 or 0 by entering:

```
     SIM> SET ALL = 0 <CR>
     SIM> SET ALL = 1 <CR>
```

Memory may be set to 1 or 0 by:

```
     SIM> SET MAIN  = 1 <CR>
     SIM> SET MICRO = 0 <CR>
     SIM> SET AUXn  = 1 <CR> (here n is 1, 2, or 3.)
```

All of the values of a dimensioned component may be set by omitting the value in parenthesis which follows the name.

6.9.10 EXECUTION COMMANDS

Execution of the simulator may be started by one of four (4) commands, "START", "RUN", "STEP", or "NEXT".

START

The start command places the values on the components which were entered in the description, sets the clock to the value specified, and starts execution. This is done each time the user requests the action without regard to whether simulation has been occurring previously. This will not ordinarily be used until the design has been at least partly verified. The user is cautioned that different results may be obtained from different start ups, since the values not set on start will likely not be the same each time.

RUN

The RUN command is executed by entry of "RUN". The BREAKS and DISPLAYS having already been set, the simulator will then run until a break position is executed or until reaching the interval limit which has been set.

STEP

The STEP command is also available to allow the user to step through a program. Unlike the RUN command which uses the tables to preset the display and break positions, the step command requires that the menus which are to be displayed be entered with the command. After the step is started, the requested display will be made at each time increment (if concurrent items, then after both items) and break from the operation will be after each time interval (after each component when concurrent operations using a single time event is executing). Only a <CR> is needed to go to the next step. The user can turn off the step mode by entry of "NOSTEP" or by entry of any legitimate simulation command. Examples of the three execute commands so far discussed follow.

```
START <CR>
RUN   <CR>
STEP show mmm, mmm, mmm <CR> (mmm are menu numbers)
```

NEXT

The NEXT command will execute the number of steps indicated by the value following the word "NEXT" and will display as required by the display actions included in the command. This is of particular use during the early stages of test and checkout. An example follows.

```
NEXT 55, SHOW 19 AT TIME=8:1:0, SHOW 12 AT TIME=0 <CR>
```

6.9.11 USING THE TRACE COMMAND

The TRACE command is available to determine the path a machine is following. As an execution of a specific item is started, a message is displayed which shows the item and the time. Data may also be displayed in conjunction with a trace by activating a display from a referenced menu. Once execution passes an item which has turned the trace off, it will no longer be active until the execution again passes a trace on command.

```
SIM>  TRACE ON AT <component>    <CR>
SIM>  TRACE ON AT <component>    <CR>
SIM>  TRACE ON AT (time>         <CR>
SIM>  TRACE ON                   <CR>
SIM>  TRACE OFF AT <component>   <CR>
SIM>  TRACE OFF ALL              <CR>
SIM>  TRACE OFF <component>      <CR>
```

This feature allows the user to repeat certain areas of execution while omitting other areas. By setting the limits over which the TRACE will range, the user can control the output.

6.9.12 TIMING AND LOOP CONTROL

The timer uses up to eight levels, each value may have a value from 0 through 9. Thus the minimum is 0:0:0:0:0:0:0:0 and the maximum value is 9:9:9:9:9:9:9:9. The use of SET to set the current time was explained in section 6.9.8.

During compilation a table is constructed of times used (all do not have to be used). A component is associated with each time increment used, but a component may use several time increments or share these times with another component. When simulation begins, execution will start at the item activated by the timer setting and then move to the next item. If the time is not set by the user, the value will be 0:0:0:0:0:0:0:0. The user may also set a sequence of events with a range of times. In the example below, if the first value is less than the second the time will proceed from the first to the second and repeat else an error will be displayed.

```
SET LOOP FROM n:n:n TO n:n:n;
SET LOOP FROM 0:1:4 TO 0:2:3;
SET LOOP FROM 2:4:5:1 TO 2:4:5:9;
```

6.9.13 HISTORY

By entering "HISTORY", the user can set a flag which causes an internal record of the execution path to be saved. Up to 100 component/times are placed in a table which may then be displayed to the user in reverse order. These can be displayed a few at a time by following the history display command with a number. An example of the commands follow.

```
HISTORY (sets flag)
NO HISTORY (resets flag)
SHO HIST (lists 100 items starting with most recent)
SHO HIST nnn (lists nnn at a time most recent first)
             (A CR will restart execution.)
SHO HIST FORWARD (lists items in forward order)
SHO HIST FORWARD FROM mmmm (lists history from time)
```

6.9.14 TIEING A PORT TO AN EXTERNAL FILE

Three data files are available to the user to either store data for input during execution or to collect data from the object machine during execution. These may be connected to one or more ports at the user's discretion. Input data may be reused by use of the reset command. Examples of the commands follow.

```
TIE n TO identifier IN (reads n as execution requires.)
TIE n TO identifier of identifier OUT (qualifies the
         identifier and ties to receive data)
RESET FILE n (sets pointer for n back to zero for read-
              ing RE-WRITING from the first.)
```

APPENDIX AND EXAMPLES TO THE CALSIM/SIMCAL USERS' MANUAL

APPENDIX 1

CALSIM RESERVED SYMBOLS AND WORDS

SYMBOLS AND LETTERS

.	End of statement marker.
<	Less than.
{	or [Opening bracket in wiring statement.
}	or] Closing bracket in wiring statement.
(	Parenthesis.
+	Arithmetic plus operator
&	Logical AND operator.
!	Logical OR operator.
*	Arithmetic multiply operator.
)	Closing parenthesis.
;	Separator symbol between clauses.
-	Arithmetic minus operator, also range separator and connector in identifiers.
/	Arithmetic divide operator.
,	Used as separators in several lists
>	Greater than.
:	Required punctuation in several locations.
=	Replacement symbol.

B	Used at end of 0/1 string to indicate a binary number.
D	At end of number string to indicates a decimal number.
H	At the end of string starting with a number, indicates A hexadeciaml number.
O	At end of number string to indicate an octal number.
Q	At end of number string to indicate an octal number.

WORDS

AND	Used in connector statement.
ARE	Used as part of several statements.
AS	Used as part of store statement.
ASCII	Indicates that format of storage is ASCII.
AUXMEMORY	Auxiliary memory description follows.
BACKPLANE	Indicates a conductor description follows.
BCD	Designates type of I-O expected at terminal.
BINARY	Describes format to and from I-O device.
BUS	The beginning of a connector statement.
CASCADE	Cascades registers into a single register.

CHIP	Start of component statement.
CMP	Monadic operator meaning complement.
CONNECT	Key word starting the wiring statement.
CONTAINS	Used in hierarchy statement.
COMPONENT	Key word in component statement.
COPY	Start of the copy statement.
CORD	Key start word in the connector statement.
DESCRIPTIONS	Used in last statement in input stream.
DISPLAY	Used during I O to give a message to the person simulating the system.
EA	Alternate spelling for each.
EACH	Specifies several LSI's is in description.
EBCDIC	One of the formats used in the format clause.
ELSE	Start of alternate in conditional statement.
END	Indicates the end of a series of statements.
EQ	The conditional equal.
FLPOINT	One of the formats for I-O.
FORMAT	Key word in format clause.
GREY	Used to designate GREY code.
HEX	Used for Hex code designation.
I-O	Key word in I-O description.
IF	Beginning of conditional expression.
IN	Equivalent to "OF" in start statement.
INPUT	Part of I-O statement.
INTO	Part of Cascade clause.
IS	Used in several statements, same as "ARE".
LEVEL	Key word in Hierarchy statement.
LIMIT	Used in time limit statement.
MEMORY	Key word to start memory statement.
MICROMEMORY	Key word to start micromemory statement.
NAME	Used in system identifier statement.
NAMED	Used in connector statement.
NAND	Boolean operator.
NEG, NEGATE	Monadic operator.
NOR	Boolean operator.
NOT	Boolean operator.
NUMBERED	Used in connector statement.
Octal	Sets data type.
OF	Part of syntax of several statements.
ON	Part of "ON PULSE" clause.
OR	Boolean operator.
OUTPUT	Used in the I-O statement.
PCKD-DEC	Format in I-O statement.
PIN	Key word in pin clause.
PINS	Same as PIN.
PORT	Peripheral keyword euqivalent to I-O.
PRINTER	Equivalent to I-O.

PULSE	Part of ON PULSE clause.
REGISTER	Part of register description clause.
REGISTERS	Same as register.
RENAME	Used to rename a copied item description.
REPLACING	Used in Copy statement.
RESET	Used to reset time.
RL	Rotate left.
RR	Rotate right.
SET	Used to set time in timing clause.
SGND-BIN	Signed binary in format clause.
SIZE	Shows size in memory descriptions.
SL	Shift left.
SR	Shift right.
START	Key word in start statement.
STORE	Key word in store statement.
SUBREGISTERS	Key word in subregister clause.
SUFFIX	Specifies suffix in copy statement.
SYSTEM	Appears in the system name description.
TERMINAL	Equivalent to I-O.
THEN	Used in the IF statement.
TIME, TIMES	Used in time statement and time clauses.
TO	Used in several statements and clauses.
WIRES	Used with connector description.
XOR	Boolean operator.

APPENDIX 2

SIMULATOR INTERFACE RESERVED WORDS

SYMBOLS: ; ( ) - : =

RESERVED WORDS:

ACTIVATE, ACT	Used to activate BREAKs.
ALL	Used in several commands.
ASC (ASCII)	Designates data type.
AT	Connector in step command.
AUX	Used to designate an auxiliary file.
BIN	Designates data type.
BREAK	Key word to start BREAK collection.
CRNT	Used with TIME for current.
DEACTIVATE, DEACT	Used to deactivate BREAKs, DISPLAYs.
DEC	Data type.
DEFINE, DEF	Key word to start list collection.
DELETE, DEL	Used to delete items from the lists.
FILE	Names a file to connect/disconnect.
FORMAT	Sets the user format in simulation.
FORWARD, FORW	Reverses order of history display.
FROM	A connector in the history display.
HEX	Designates data type.
HISTORY, HIST	Used in SHOW HISTORY command.
INPUT, IN	Used in TIE command.
INTERVAL, INT	Used to set maximum interval.
LIST, L	Displays the items in the LISTs.
LOOP	Key word in the LOOP command.
MAIN	Refers to main memory.
MENU, MEN	Key word to collect menus.
MICRO	Reference to micro file.
NEXT, N	Key word in NEXT command.
NO	Used to turn off HISTORY collection.
OCT	Designates OCTAL.
OF, IN	Connector to qualify identifier.
OFF, ON	Used in TRACE command.
OUTPUT, OUT	Used in the TIE command.
PINS PIN, P	Used in several commands.
RESET	Used to set a file pointer to start.
RUN	Starts operation.
SET	Used to set various values.
SHOW, SHO, S	Used to display identifier values.

START	Starts the simulation process.
STEP	Key word to start the STEP process.
TIE	Ties system to a specified file.
TIME	Used to reference the clock.
TO	Connector in SET and TIE commands.
TRACE, TR	Starts or stops TRACE action.
WIRES, W	Designates WIRE in several commands.

APPENDIX 3

EXAMPLES IN THE CALSIM/SIMCAL SYSTEM

This series of examples begin at the lowest level (the flip-flop) and increase in complexity through the bit-sliced microprogrammable components. Each example allows the reader to build on that already learned and to further extend his use of the language. Each example includes:

A. A description of the device.
B. The Calsim program listing of the device.
C. A block diagram of the device.

The examples include:

APPENDIX 3.1 FLIP-FLOPS

A simple circuit is used to introduce the Calsim language and support system. Only the most basic language structures are used in this example; other structures will be introduced in subsequent examples. The flip-flop is the basic bi-stable device used in computer circuits to capture and store the binary value. The description in Calsim includes the NOR gate as a one bit register as shown in figure 1. The clock circuits, usually shown are omitted since our Computer Hardware Description Language will provide the timing without the need to "wire" the component with a timing circuit.

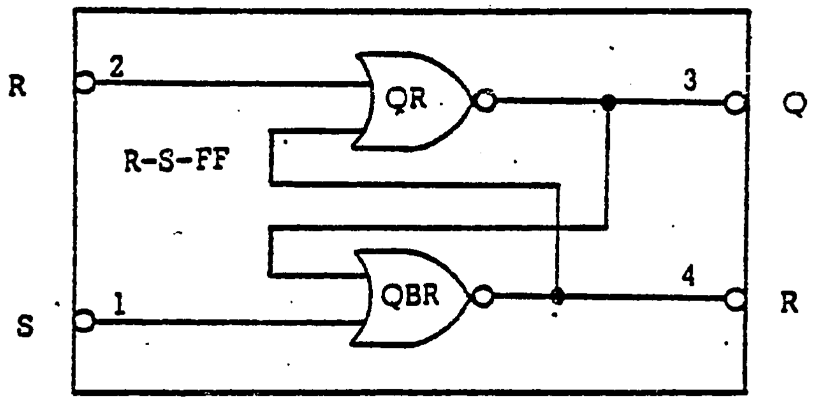


FIGURE 1 AN R-S FLIP-FLOP.

CALSIM LISTING:

SYSTEM NAME IS R-S-FLIP-FLOP.

LEVEL 1: R-S-FLIP-FLOP CONTAINS R-S-FF.

COMPONENT: R-S-FF

```
    PINS ARE NUMBERED (1-4) NAMED R, S, Q, QB;
    REGISTERS ARE QR(1), QBR(1);
    SET TIME = 1:0;
```

```
   /* THE PIN ON THE LEFT - # 1 - IS CONSIDERED THE LOW
      ORDER PIN. THAT IS IF THE VALUE IS 01 THEN PIN 1 IS
      1 AND PIN 2 HAS A VALUE OF 0. */

   IF PINS (1-2) EQ 01 THEN PIN QB = 0, PIN Q = 1 D,
      QR(1) = 1, QBR(1) = 0A H;
   IF PINS (1-2) EQ 10B THEN QB=1, Q = 0, QR = 0, QBR = 1;
   IF PINS [R S] EQ 3D THEN QB = X, Q = X, QR = X, QBR = X.

   /* NOTE: IF PINS(1-2) = 00 THEN NO CHANGE OCCURS */

END OF R-S-FLIP-FLOP.

END OF DESCRIPTIONS.
```

APPENDIX 3.2 REGISTERS AND SHIFT REGISTERS

FIGURE 2 shows a 4-bit parallel to serial shift register. The shift register includes a "clear" capability which will place all zeros on the output lines. Note that four time periods are needed to get all four bits into the shift register and that the bit first received ripples through each position. The terminal included in the description is a software version of the test instruments used with hardware when checking hardware operation.

Two new Calsim statements are introduced in the descriptions in this example. The first of these is the START statement in which PIN, REGISTER, MEMORY, and TIME values can be established. The second is "COPY". Note that copying is anticipated in the LEVEL statement which preceeds the actual copying. The user should also note that the description is stored under one name (up to thirty characters) but the name which actually appears in the description is the name embedded in the description.

CALSIM LISTING:

```
SYSTEM NAME FOUR-BIT-SHIFT..

  /* WE WILL STORE THE CHIP AND THEN COPY IT FROM THE
     LIBRARY. */

 STORE AS D-FLIP-FLOP.
 CHIP: DFF PINS ARE NUMBERED (1-3) AND NAMED SGIN, OUT, CLR;
       REGISTER IS RDFF(1);
   SET TIME = *:0; /* DIGIT 1 TO BE SUPPLIED AT COPY TIME.*/
    /* THE CHIP IS FIRST ZEROED  IF THE CLR LINE IS HIGH. */
       IF CLR EQ 1 THEN RDFF = 0, OUT = 0;
       SET TIME = *:1;
       IF SGIN EQ 1 THEN RDFF = 1, OUT = 1
       ELSE RDFF = 0 OUT = 0.
END OF D-FLIP-FLOP.

LEVEL 1: FOUR-BIT-SHIFT CONTAINS 4 EACH DFF,
         USER-TERMINAL, SHIFT-BUS.

 /* IN THE FOLLOWING TIME STATEMENT THE VALUES 1 THROUGH 4
     ARE PLACED IN THE FIRST POSITION OF THE CLOCK. */

COPY 4 EACH D-FLIP-FLOP TIME = 1:*,4:*,1:*.
```

```
   /* NOTE THAT HIGH ORDER  (DFF01) IS SHIFTED FIRST SO THAT
      THE  INCOMING  BIT WILL RIPPLE TO  THE HIGH END OF THE
      SERIES OF FLIP-FLOPS. */

CORD: SHIFT-CORD  NAMED INPT, CLR, Q1, Q2,
      Q3, Q4.

TERMINAL: USER-TERMINAL PINS ARE NUMBERED (1-6) AND NAMED
     OUT, CLR, Q1, Q2, Q3, Q4;
     FORMAT IS HEX;
     SET TIME = 0:1;
     ON PULSE  DISPLAY "ENTER VALUE FOR PINS 1-2",
               PIN (1-2) = INPUT;
     SET TIME = 5:0;
     ON PULSE DISPLAY "PINS-3-6-ARE "
              OUTPUT = PINS (3-6).

CONNECT SHIFT-BUS TO USER-TERMINAL  [(1-6)] [(1-6)];
                  TO DFF01          [5 6 2] [(1-3)];
                  TO DFF02          [4 5 2] [1 2 3];
                  TO DFF03          [3 4 2] [1 2 3];
                  TO DFF04          [1 3 2] [(1-3)].
START: SET TIME = 0:0; WIRE (2) OF SHIFT-BUS = 1B.

END OF FOUR-BIT-SHIFT. END OF DESCRIPTIONS.
```

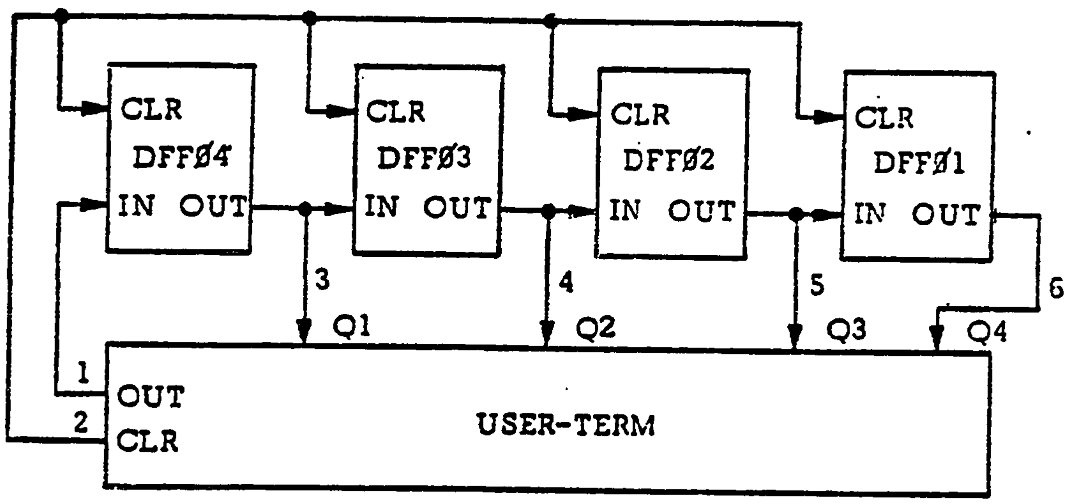


FIGURE 2   A SHIFT REGISTER WITH A "USERS' TERMINAL".

APPENDIX 3.3 COUNTERS, ENCODERS, AND DECODERS

A counter, terminal and counter is presented in this example and the use of multiple I-O devices is introduced with the 20 light emitting diodes (LED) for output. The output of each of these will be printed (if "ON") on the user terminal along with the device number.

CALSIM DESCRIPTION:

```
SYSTEM NAME IS COUNTER-DECODER.

STORE AS LIGHT-EMITTING-DIODE.
PORT: LED PIN IS NUMBERED 1; FORMAT IS BINARY;
      SET TIME = *;
      IF PIN 1 > 0 AND PIN 1 < 2 THEN OUTPUT = 1.

  /* NOTE THAT THERE IS OUTPUT ONLY IF THE LIGHT IS ON. */

END OF LIGHT-EMITTING-DIODE.

LEVEL  1: COUNTER-DECODER CONTAINS COUNTER, DECODER, 20 EACH
          LED, INPUT-TERMINAL, Q-BUS.

COPY 4 EACH LIGHT-EMITTING-DIODE TIME = 2:0.

COPY 16 EACH LIGHT-EMITTING-DIODE TIME = 4:0,5:5.

  /* THE FIRST 4 WILL HAVE THE SAME TIMES, THE LAST 16 WILL
     HAVE TIMES SEPARATED BY 0:1 UNITS. */

COMPONENT: COUNTER WIRES ARE NUMBERED (1-6) AND NAMED CLR
           CNT Q1, Q2, Q3; /* HERE THE LAST NAME IS OMITTED
                                TO SHOW CALSIM RESPONSE. */
           REGISTER IS BIN-CNT (4);
           SET TIME= 1:1;
           IF CLR EQ 1 THEN BIN-CNT=0 Q0=0 Q1=0, Q2=0, Q3=0;
           IF CNT EQ 1 THEN BIN-CNT = (BIN-CNT + 1);
           ON Q0 = BIN-CNT (1), Q1 = BIN-CNT (2),
              Q3 = BIN-CNT (3), Q4 = BIN-CNT (4).

     /* NOTE THAT Q0 AND BIT 1 IN BIN-CNT ARE LOW ORDER */

CHIP: DECODER PINS ARE NUMBERED (1-20) AND NAMED IN1, IN2,
      IN3, IN4, ZERO, ONE, TWO, THRE, FOUR,FIVE, SIX, SEV,
        EIGH, NINE, TEN, ELEV, TWEL, THIR, FRTN, FIFT;
        SET TIME = 3:0; ON PULSE PINS(5-20) = 0;/*RESET PINS*/

        SET TIME = 3:1;
        IF PINS (1-4) EQ 0 THEN PIN 5 =  1;
        IF PINS (1 - 4) EQ 1Q THEN PIN 6 =  1;
        IF PINS (1 - 4) EQ 2  THEN PIN 7 =  1;
        IF PINS (1 - 4) EQ 3  THEN PIN 8 =  1;
        IF PINS (1 - 4) EQ 4  THEN PIN 9 =  1;
        IF PINS (1 - 4) EQ 5  THEN PIN 10 = 1;
        IF PINS (1 - 4) EQ 6  THEN PIN 11 = 1;
        IF PINS (1 - 4) EQ 7O THEN PIN 12 = 1;
        IF PINS (1 - 4) EQ 8  THEN PIN 13 = 1;
        IF PINS (1 - 4) EQ 9  THEN PIN 14 = 1;
        IF PINS (1 - 4) EQ 10 THEN PIN 15 = 1;
        IF PINS (1 - 4) EQ 11 THEN PIN 16 = 1;
        IF PINS (1 - 4) EQ 12 THEN PIN 17 = 1;
        IF PINS (1 - 4) EQ 13 THEN PIN 18 = 1;
        IF PINS (1 - 4) EQ 14 THEN PIN 19 = 1;
        IF PINS (1 - 4) EQ 15 THEN PIN 20 = 1.

BUS: Q-BUS IS NUMBERED (1-22) AND NAMED CLR SETT Q0
     Q1, Q2, Q3, ZERO, ONE,TWO THRE FOUR FIVE SIX SEVE EIGH,
     NINE, TEN, ELE, TWEL, THIR, FRTN, FFTN.

CONNECT Q-BUS TO TERM    [(1-2)]  [(1-2)];
              TO BINCNT  [(1-6)]  [(1-6)];
              TO DECODER [(3-20)] [(1-20)];
              TO LED01   [7]      [1];
              TO LED02   [8]      [1];
              TO LED03   [9]      [1];
              TO LED04   [10]     [1];
              TO LED05   [11]     [1];
              TO LED06   [12]     [1];
              TO LED07   [13]     [1];
              TO LED08   [14]     [1];
              TO LED09   [15]     [1];
              TO LED10   [16]     [1];
              TO LED11   [17]     [1];
              TO LED12   [18]     [1];
              TO LED13   [19]     [1];
              TO LED14   [20]     [1];
              TO LED15   [21]     [1];
              TO LED16   [22]     [1];
              TO LED17   [23]     [1];
              TO LED18   [24]     [1];
              TO LED19   [25]     [1];
```

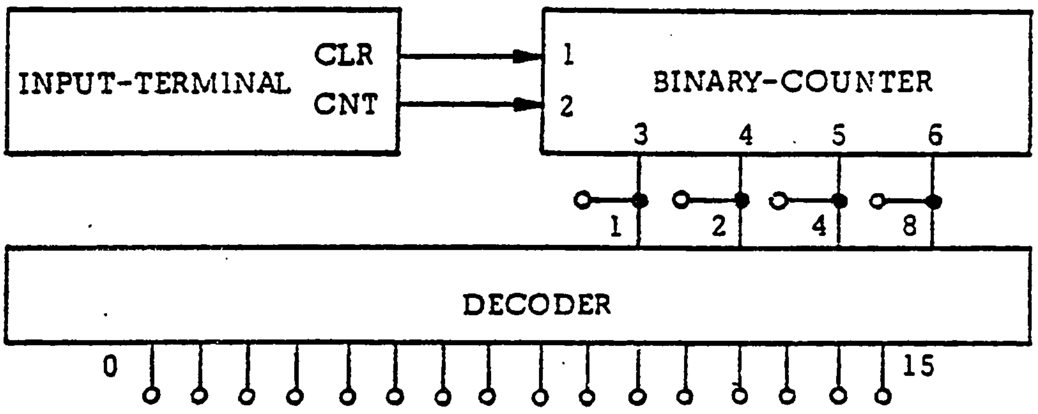
```
               TO LED20   [26]     [1].

I-O: INPUT-TERMINAL PINS ARE NUMBERED (1-2) AND NAMED CLR
     CNT; /* NOTE THAT ONLY 01 & 10 ARE LEGAL ENTRIES. */

     FORMAT IS BINARY; SET TIME = 0:5;
     ON PULSE DISPLAY "ENTER 10 TO CLEAR, 01 TO COUNT";
     ON PULSE PIN (1-2) = INPUT;
     SET TIME =9:0; ON DISPLAY " LIGHTS ARE ON.".

  END OF COUNTER-DECODER.

  END OF DESCRIPTIONS.
```



COUNTER-DECODER

FIGURE 3   COUNTERS, ENCODERS, DECODERS

APPENDIX 3.4 HALF ADDERS, FULL ADDERS

The full adder shown in figure 4 uses the truth table shown in table 4. Note that the description uses the full adder as the elementary component. The internal wiring shown in the figure is for user information only. Calsim need not make these connections.

An additional Calsim construction is introduced here -- the use of the qualifier to show which identifier is being referenced. In this case the callout is not needed since there is only one set of pins. The EACH expression is also used here in a more complex structure than when used with the LEDs.

TABLE 4 TRUTH TABLE FOR THE SUM OF A, B, CRY

A	B	CRY	SUM	CRY OUT
0	0	0	0	0
0	0	1	1	0
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

CALSIM DESCRIPTION:

```
  /*  A FOUR BIT FULL ADDER */

SYSTEM NAME IS FOUR-PLACE-FULL-ADDER-CHIP.

LEVEL 1: FOUR-PLACE-FULL-ADDER-CHIP CONTAINS FOUR-PL-ADDER.

LEVEL 2: FOUR-PL-ADDER CONTAINS 4 EA FULL-ADDER ADDER-BUS.

CHIP: FULL-ADDER01, PINS ARE NUMBERED (1-5) AND NAMED
      CI, A, B, CRY, SUM;  /* LOW ORDER */
      SET TIME = 0:5;
      IF PINS (1-3) EQ 000B THEN SUM = 0, CRY = 0;
      IF PINS (1-3) EQ 001B THEN SUM = 1, CRY = 0;
      IF PINS (1-3) EQ 010B THEN SUM = 1, CRY = 0;
      IF PINS (1-3) EQ 011B THEN SUM = 0, CRY = 1;
```

```
        IF PINS (1-3) EQ 100B THEN SUM = 1, CRY = 0;
        IF PINS (1-3) EQ 101B THEN SUM = 0, CRY = 1;
        IF PINS (1-3) EQ 110B THEN SUM = 0, CRY = 1;
        IF PINS (1-3) EQ 111B THEN SUM = 1, CRY = 1.

CHIP:   FULL-ADDER02 PINS ARE NUMBERED (1-5) AND NAMED
        CI, A, B, CRY, SUM;
        SET TIME = 0:6;
        IF PINS (1-3) EQ 0 THEN SUM = 0, CRY = 0;
        IF PINS (1-3) EQ 1 THEN SUM = 1, CRY = 0;
        IF PINS (1-3) EQ 2 THEN SUM = 1, CRY = 0;
        IF PINS (1-3) EQ 3 THEN SUM = 0, CRY = 1;
        IF PINS (1-3) EQ 4 THEN SUM = 1, CRY = 0;
        IF PINS (1-3) EQ 5 THEN SUM = 0, CRY = 1;
        IF PINS (1-3) EQ 6 THEN SUM = 0, CRY = 1;
        IF PINS (1-3) EQ 7 THEN SUM = 1, CRY = 1.

CHIP:   FULL-ADDER03 PINS ARE NUMBERED (1-5) AND NAMED
        CI, A, B, CRY, SUM;
        SET TIME = 0:7;
        IF PINS (1-3) EQ 0 THEN SUM = 0, CRY = 0;
        IF PINS (1-3) EQ 1 THEN SUM = 1, CRY = 0;
        IF PINS (1-3) EQ 2 THEN SUM = 1, CRY = 0;
        IF PINS (1-3) EQ 3 THEN SUM = 0, CRY = 1;
        IF PINS (1-3) EQ 4 THEN SUM = 1, CRY = 0;
        IF PINS (1-3) EQ 5 THEN SUM = 0, CRY = 1;
        IF PINS (1-3) EQ 6 THEN SUM = 0, CRY = 1;
        IF PINS (1-3) EQ 7 THEN SUM = 1, CRY = 1.

CHIP:   FULL-ADDER04 PINS ARE NUMBERED (1-5) AND NAMED
        CI, A, B, CRY, SUM;
        SET TIME = 0:8;
        IF PINS (1-3) EQ 0 THEN SUM = 0, CRY = 0;
        IF PINS (1-3) EQ 1 THEN SUM = 1, CRY = 0;
        IF PINS (1-3) EQ 2 THEN SUM = 1, CRY = 0;
        IF PINS (1-3) EQ 3 THEN SUM = 0, CRY = 1;
        IF PINS (1-3) EQ 4 THEN SUM = 1, CRY = 0;
        IF PINS (1-3) EQ 5 THEN SUM = 0, CRY = 1;
        IF PINS (1-3) EQ 6 THEN SUM = 0, CRY = 1;
        IF PINS (1-3) EQ 7 THEN SUM = 1, CRY = 1.

START: SET TIME = 0:5; PINS (1-5) IN FULL-ADDER01 = 0;
                       PINS (1-5) OF FULL-ADDER02 = 0;
                       PINS (1-5) OF FULL-ADDER03 = 0;
                       PINS (1-5) OF FULL-ADDER04 = 0.

BUS: ADDER-BUS IS NUMBERED (1-17) AND NAMED CI,A1,B1,X12,S1,
     A2,B2,X23,S2,A3,B3,X34,S3,A4,B4,CO,S4.

CONNECT ADDER-BUS TO FULL-ADDER01 [(1-5)][(1-5)];
                  TO FULL-ADDER02 [4,(6-9)][(1-5)];
                  TO FULL-ADDER03 [8, (10-13)][(1-5)];
                  TO FULL-ADDER04 [12, (14-17)][(1-5)].

END OF FULL-ADDER-CHIP.

END OF DESCRIPTIONS.
```

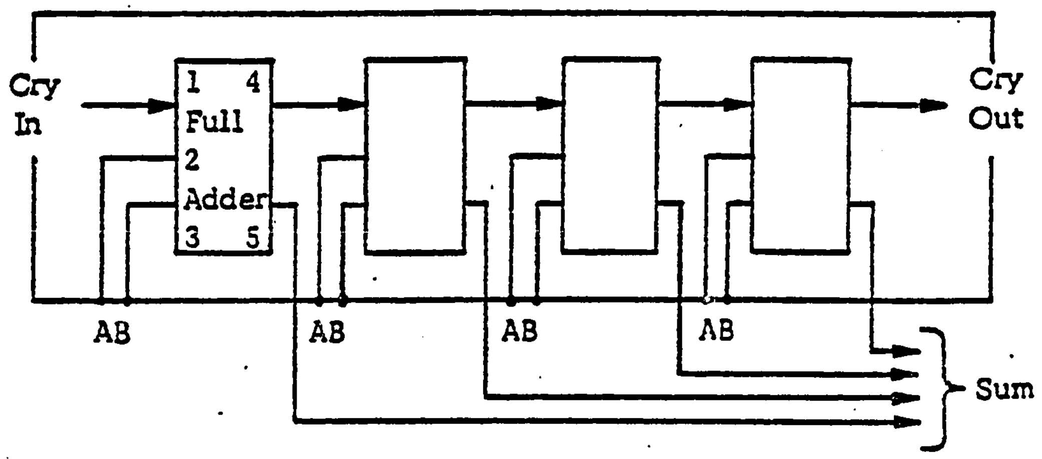


FIGURE 4   A FULL ADDER

APPENDIX 3.5 THE ARITHMETIC LOGIC UNIT (ALU)

The arithmetic logic unit performs operations on either one or two operands usually selected from those available in the immediate registers or held in a latch. In some systems, the operand may be fetched from main memory. The resulting value is then stored or made available for other action by placing the value on the data bus. The AMD-2901 four bit-slice central processing element (CPE) shown in figure 5.A is chosen for this example as it demonstrates several of the functions found in the ALU. The logic of the system is shown in Tables 5.1, 5.2, and 5.3.

The AMD-2901 is a four (4) bit slice which may be cascaded into a width any multiple of four. The item contains 16 general registers, and a special register (Q). These may be loaded with data which has been rotated left or right or unrotated. The ALU instruction itself is divided into three parts of three bits each as shown in tables 5.1, 5.2 and 5.3. These are used for the source of data, operations on data, and disposition of data.

When the chips are cascaded, the carry in is connected to the adjoining carry out so that the whole series has a single carry-in and a single carry-out. The same is true of the RAM0, RAM3, Q0 and Q3. The reader is referred to the AMD-2901 data sheet for a further explanation of the chip. The designations here follow that of Advanced Micro Devices designation for the 40 pin DIP (Dual Inline Pins).

This example introduces the use of "working" registers which are analgous to working registers found in hardware. Such registers are available to the hardware (in this case SIMCAL), but not to the assembly programmer.

TABLE 5.1 ALU SOURCE SELECTION OPERANDS

	OCTAL VALUE			ALU SOURCE OPERANDS	
	I2	I1	I0	R	S
0	0	0	0	A	0
1	0	0	1	A	B
2	0	1	0	0	Q
3	0	1	1	0	B
4	1	0	0	0	A
5	1	0	1	D	A
6	1	1	0	D	Q
7	1	1	1	D	0

TABLE 5.2 ALU FUNCTION CONTROL

OCTAL CODE (I6, I5, I4	MNEUMONIC CODE	FUNCTION
0	ADD	R + S
1	SUBTRACT	S - R
2	SUBTRACT	R - S
3	OR	R OR S
4	AND	R AND S
5	NOTRS	NOT R AND S
6	EXOR	EXCL R OR S
7	EXNOR	NOT (R NOR S)

TABLE 5.3 DESTINATION CONTROL

OCTAL	RAM FNCTION		QREG FNCTN		Y	RAM		Q-REG	
I8,7,6	SHIFT	LOAD	SHIFT	LOAD	OUT	RM0	RM3	Q0	Q3
0	X	NONE	NONE	F->Q	F	X	X	X	X
1	X	NONE	X	NONE	F	X	X	X	X
2	NONE	F->RAM	X	NONE	A	X	X	X	X
3	NONE	F->RAM	X	NONE	F	X	X	X	X
4	DOWN	F/2->RAM	DOWN	Q/2->Q	F	F0	NC	Q0	NC
5	DOWN	F/2->RAM	X	NONE	F	F0	X	Q0	X
6	UP	2F->RAM	UP	2Q->Q	F	NC	F3	NCC	Q3
7	UP	2F->RAM	X	NONE	F	NC	F3	X	X

CALSIM DESCRIPTION:

```
SYSTEM NAME IS AM2901A-CPE.

LEVEL 1: AM2901A-CPE CONTAINS AM2901A.
COMPONENT: AM2901A PINS ARE NUMBERED (1-40) AND NAMED

   /* THIS IS FOR DIP ONLY, FLAT HAS OTHER VALUES */
    A3,A2,A1,A0,I6,I8,I7,RAM3,RAM0,VCC,F0,I0,I1,
    I2,CP,Q3,B0,B1,B2,B3,Q0,D3,D2,D1,D0,I3,I5,I4,
    CN,GND,F3,GBAR,CN4,OVR,PBAR,Y0,Y1,Y2,Y3,OEBAR;

   REGISTERS ARE Q-REG(4) RAM(4,16) RAM-SHIFT(4) Q-SHIFT(4);
   /* WE WILL SET UP SOME WORKING REGISTERS FOR OUR USE TO
    MAKE THE MANIPULATION A LITTLE EASIER TO HANDLE. */

   REGISTERS ARE A(4), B(4), Q-LOOP(4), R(4), S(4), F(4);
   /* WE WILL WORK THE FIRST THREE VALUES OF THE OP CODE. */
```

```
SET TIME = 2:0;
ON PULSE Q-LOOP = Q-REG, A = RAM(, PINS[A0 A1 A2 A3]),
D = PINS[D0 D1 D2 D3], B = RAM(, PINS[B0 B1 B2 B3]);
    IF PINS(1-3) EQ 0 THEN R = A, S = Q-LOOP;
    IF PINS(1-3) EQ 1 THEN R = A, S = B;
    IF PINS(1-3) EQ 2 THEN R = 0, S = Q-LOOP;
    IF PINS(1-3) EQ 3 THEN R = 0, S = B;
    IF PINS(1-3) EQ 4 THEN R = 0, S = A;
    IF PINS(1-3) EQ 5 THEN R = D, S = A;
    IF PINS(1-3) EQ 6 THEN R = D, S = Q-LOOP;
    IF PINS(1-3) EQ 7 THEN R = D, S = 0;

 /* LETS RESET THE TIME AND WORK I5,I4,I3 */
 SET TIME = 3:0;
    IF PINS(4-6) EQ 0 THEN F = R + S;
    IF PINS(4-6) EQ 1 THEN F = S - R;
    IF PINS(4-6) EQ 2 THEN F = R - S;
    IF PINS(4-6) EQ 3 THEN F = R ! S;
    IF PINS(4-6) EQ 4 THEN F = R & S;
    IF PINS(4-6) EQ 5 THEN F = R & S;
    IF PINS(4-6) EQ 6 THEN F = R XOR S;
    IF PINS(4-6) EQ 7 THEN F =  NOT (R XOR S);

/* LETS RESET TIME AND WORK I6, I7, I8 */
SET TIME = 4:0;
   IF PINS (7-9) EQ 0 THEN Y = F, Q-REG = F;
   IF PINS (7-9) EQ 1 THEN Y = F;
   IF PINS (7-9) EQ 2 THEN Y=A, RAM(, PINS (17-20)) = F;
   IF PINS (7-9) EQ 3 THEN Y=F, RAM(,PINS (17-20))=F;
   IF PINS (7-9) EQ 4 THEN Y=F RAM(,PINS (17-20))=F / 2,
      Q-LOOP = Q-LOOP / 2, PIN R0=F(1), PIN Q0=Q-LOOP(1);
   IF PINS (7-9) EQ 5 THEN Y=F, RAM(,PINS (17-20))=F / 2,
                          PIN R0 = F(1), PIN Q0 = Q-LOOP(1);
   IF PINS (7-9) EQ 6 THEN Y=F, RAM(, PINS(17-20)) = F*2,
      Q-REG = SL Q-REG, PIN R3 = F(4), PIN Q3=Q-LOOP(4);
   IF PINS (7-9) EQ 7 THEN Y=F, RAM(,PINS (17-20)) = 2*F,
                          PIN R3 = F(4), PIN Q3 = Q-LOOP(4).
END OF AM2901A-CPE.

END OF DESCRIPTIONS.
```

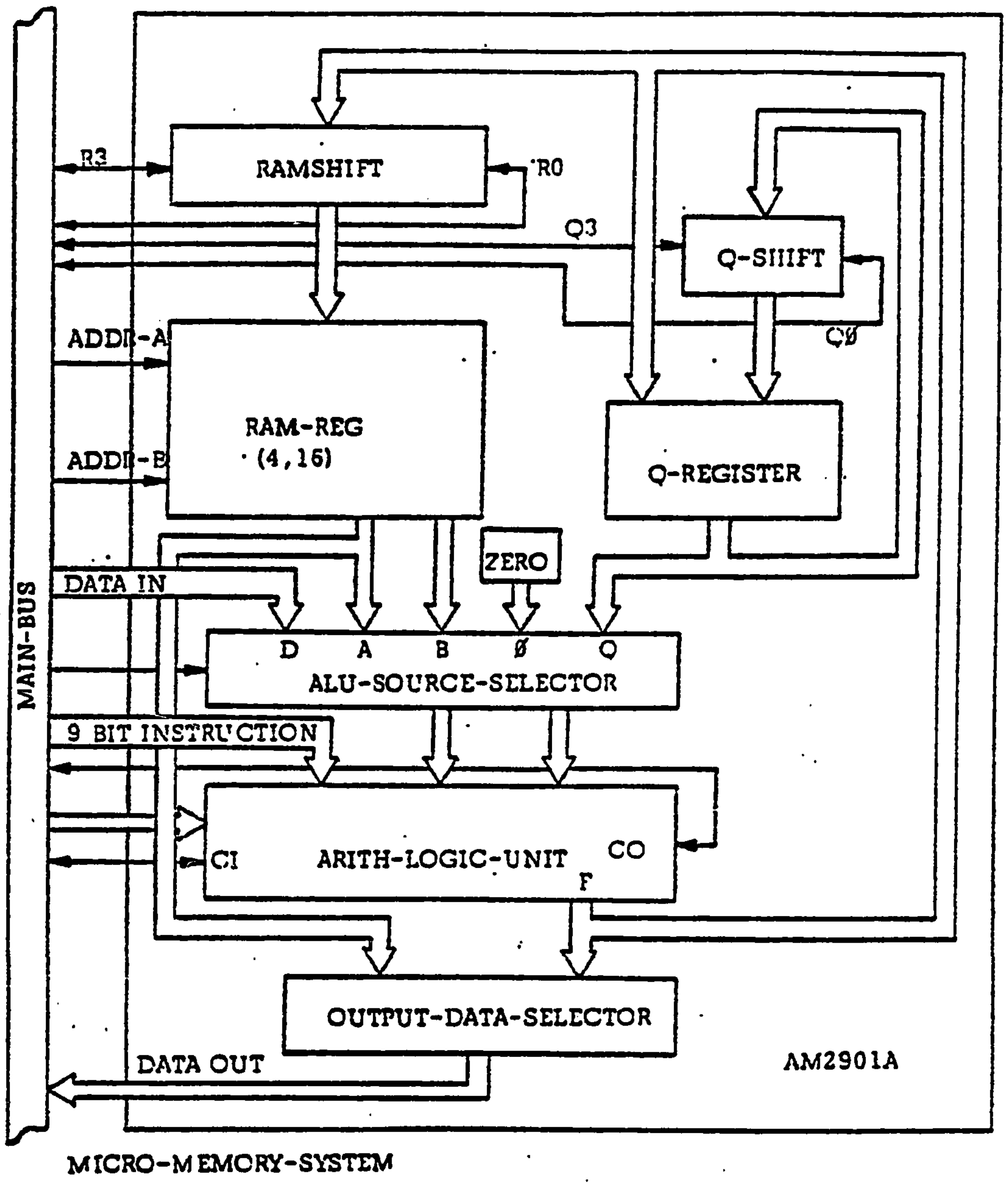


FIGURE 5 THE AMD-2901 ARITHMETIC LOGIC CHIP.

page 63

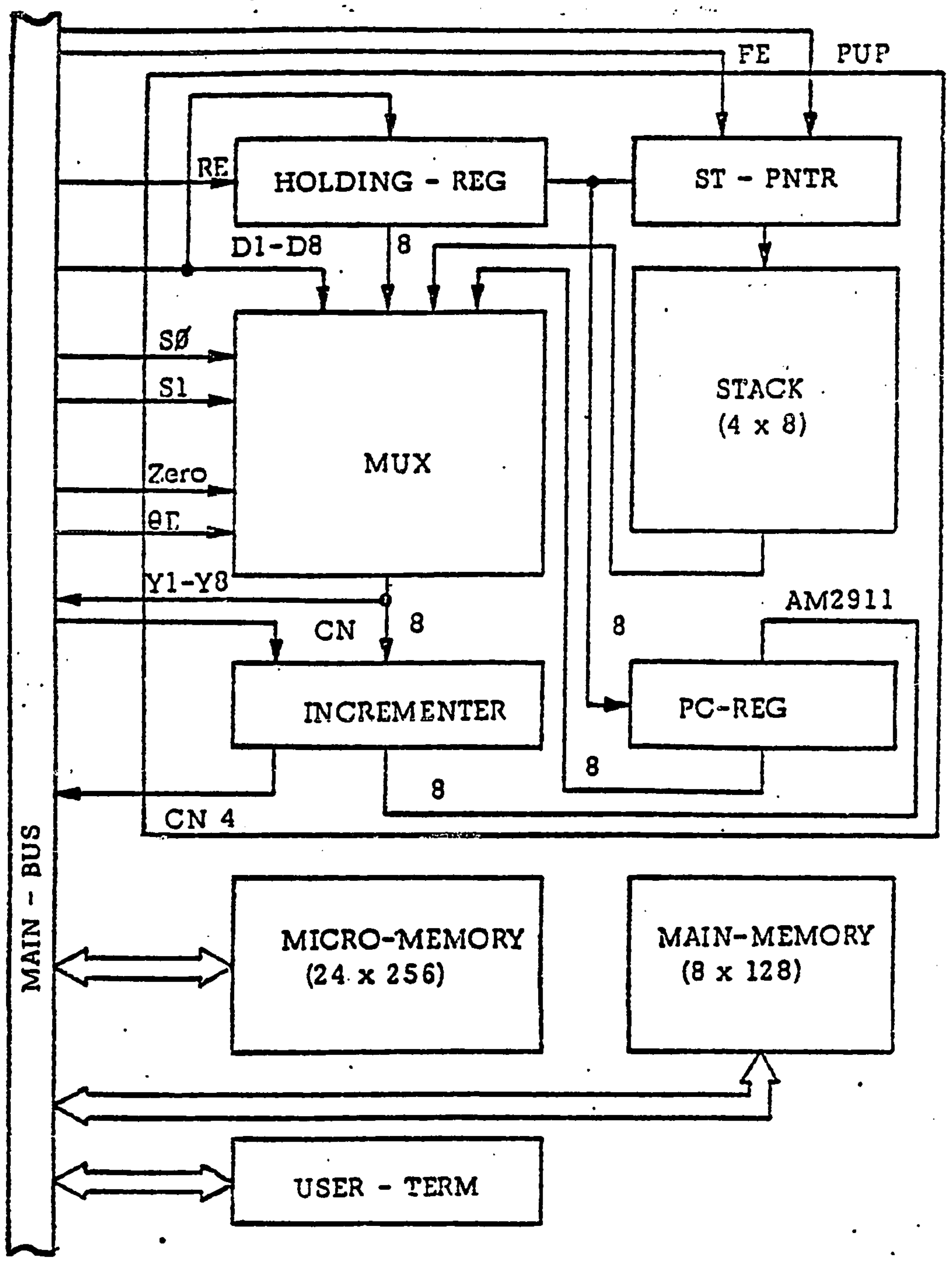


FIGURE 6 AM 2911 SEQUENCE SYSTEM

APPENDIX 3.7 THE MIC COMPUTER

The MIC (Microprogrammable Instructional Computer) is presented in this example to acquaint the reader with some of the aspects of microprogramming without introducing unnecessary complexity.

The cycle of this machine always ends with microinstruction number zero, which then requests the next program instruction. The next program instruction is provided by main memory and converted to a microinstruction address by the ROM. To carry out the intended operation one to ten microinstructions are processed with the last always returning control to the instruction at location zero. A brief description of the process follows.

A. The microinstruction is taken from the micromemory and placed in the pipeline register. In this example, that instuction will be used as the next executed instruction although in actual practice the previous instruction would be executing while the register is being loaded with the next instruction.

B. It is necessary at this point to specify the sequence in which the instructions will be executed. Table 8.1 shows the correct order so we will start with the action on pins 1-2 which selects the data to be placed on the ADDR-BUS. A value of 0 will place PGM-CNTL on the address bus and a value of 1 will place the NXT-ADDR on the bus. If no value is needed then, the value must be 1 since the use of the PGM-CNTL causes it to increment.

C. Positions 3-4 specifies the I-O device as follows:

00	No device action
01	Terminal number 1.
10	Terminal number 2.
11	Main Memory.

D. The third field (bit 5) determines whether the I-O is READ or WRITE. READ = 0. WRITE = 1. If the command is READ then the action will take place prior to the ALU action; If the command is WRITE, then the action will take place later after the ALU has completed the action. This is reflected in the timing in the description.

E. Bits 6-7 determine the action of the latch on the data appearing on the data bus.

00	No Action
01	Load into A.
10	Load into B.
11	Load zeroes into both A and B.

F. Bits 8-11 provide sixteen(16) ALU actions which are possible using A and/or B with the result always placed in the accumulator. These actions are:

DECIMAL VALUE OF BITS 8-11	ACTION
00	NO ACTION, NOP
01	ACC = A + B
02	ACC = A AND B
03	ACC = A OR B
04	ACC = A
05	ACC = A - B
06	ACC = A XOR B
07	ACC = B
08	NEGATE ACC
09	SHIFT LEFT ACC
10	SHIFT RIGHT ACC
11	SHIFT LEFT THRU LINK ACC
12	SHIFT RIGHT THRU LINK ACC
13	COMPLEMENT ACC
14	ROTATE LEFT THRU LINK
15	ROTATE RIGHT THRU LINK

G. Bits 12-13 determine the disposition of the accumulated data as shown below.

00	No Action
01	To low value in NXT-ADDR
10	To high Value in NXT-ADDR
11	To data bus

H. If the data is to be deposited then the address has already been deposited and the device chosen. This completes the ALU action.

I. Pins 14-15 allow the user to select which flag may be used to control either the microprogram or the main program sequence. The setting merely selects the bit which is to be placed on the wire that can then be used to decide on a jump around. The code follows.

00	flag Z
01	flag C
10	flag V
11	flag S

J. The next instruction to be executed may come from either the address given from the PROM, the address shown in the MM-PC or the address shown in the JUMP position. The latter may be conditional on the value from the chosen flag.

00	Get address from the PROM
01	Get address from the MM-PC
10	Jump to address given in locations 18 -27.
11	If flag line high jump to address from locations 18-27, else to MM-PC.

K. Bits 18-27 is the 10 digit next address.

L. Bit 28 is set on 1 to restart the master clock, The master clock is reset to 0 on each fetch of a program instruction.

```
SYSTEM NAME IS MICRO-PGMBL-INST-COMPUTER.

LEVEL 1:
MICR-PGMBL-INST-COMPUTER CONTAINS MAIN-MEMORY, TERM1,
    TERM2, ADDR-SEL-ROM, ADDR-SEL-MM, MMPC, MICRO-MEM,
    PIPELINE, ALU, ADDR-BUS, DATA-BUS, CNTL-BUS.

BUS: ADDR-BUS IS NUMBERED (1-16).
BUS: DATA-BUS IS NUMBERED (1-8).
BACKPLANE: CNTL-BACKPLANE IS NUMBERED (1-95).

I-O: TERM1 PINS ARE NUMBERED (1-11);
   /* PINS 9,10,11 FROM BUS(93, 94, 95) PIPELINE (30-32) ARE
      USED AS FOLLOWS:
            9 & 10       ACTION
             00          NO READ OR WRITE
             01          TERMINAL 1
             10          TERMINAL 2
             11          MAIN MEMORY

      PIN 11   0 =  READ;  1  =  WRITE.  */

    FORMAT IS GREY;
    SET TIME = 0:5;
```

```
IF PINS(9-10) EQ 01 AND PIN 11 EQ 0 THEN OUTPUT=PINS (1-8);
IF PINS(9-10) EQ 01 AND PIN 11 EQ 1 THEN PINS (1-8) = INPUT.

I-O: TERM2 PINS ARE NUMBERED (1-11);
     FORMAT IS  ASCII;
     SET TIME = 9:0;
IF PINS(9-10) EQ 01 AND PIN 11 EQ 0 THEN OUTPUT=PINS (1-8);
IF PINS(9-10) EQ 01 AND PIN 11 EQ 1 THEN PINS (1-8) = INPUT.

CHIP: ADDR-SEL-ROM PINS ARE NUMBERED (1-18);
      REGISTER IS ROM-REG (10,256);
      SET TIME = 1:0;
      ON PULSE PINS (9-18) = ROM-REG(,PINS(1-8)).

CHIP: ADDR-SEL-MM PINS ARE NUMBERED (1-43) AND NAMED
      PC1, PC2, PC3, PC4, PC5, PC6, PC7, PC8, PC9, PC10,
      RO1, RO2, RO3, RO4, RO5, RO6, RO7, RO8, RO9, RO10,
      JM1, JM2, JM3, JM4, JM5, JM6, JM7, JM8, JM9, JM10,
      OT1, OT2, OY3, OT4, OT5, OT6, OT7, OT8, OT9, OT10,
      FLG, CNT1, CNT2;
      SET TIME = 2:0;
  /* PIN 41 CARRIES 1 (FLAG SET) OR 1(FLAG NOT SET) FROM THE
        ALU FLAGS. PINS [41, 42] ARE USED AS FOLLOWS:
               SETTING          ACTION
                  00            USE THE PC ADDRESS
                  01            UNCONDITIONAL JUMP
                  10            USE VALUE FROM ROM
                  11            JUMP IF FLAG SET */

IF PINS [CNT1 CNT2] EQ 00B THEN PINS (11-20) = PINS (01-10);
IF PINS [CNT1 CNT2] EQ 01B THEN PINS (11-20) = PINS (31-40);
IF PINS [CNT1 CNT2] EQ 10B THEN PINS (11-20) = PINS (21-30);
IF PIN FLG EQ 1 AND PINS [CNT1 CNT2] EQ 11B THEN
     PINS (11-20)=PINS (31-40)
ELSE PINS (11-20)=PINS (1-10).

MEMORY: MAIN-MEM PINS ARE NUMBERED (1-27) AND NAMED
     D1, D2, D3, D4, D5, D6, D7, D8, AD1, AD2, AD3, AD4,
     AD6, AD7, AD8, AD9, AD10, AD11, AD12, AD13, AD14, AD15,
     AD16, CNT1, CNT2, CNT3;
     SIZE = 8 * 256;
     SET TIME = 0:5;
   IF PINS (25-27) EQ 70 THEN PINS [D1 D2 D3 D4 D5 D6 D7 D8]
       = MAIN-MEM (,(9-24)).

CHIP: MM-PC PINS ARE NUMBERED (1-20) AND NAMED IN1, IN2;
      REGISTERS ARE PC-REG(10);
```

```
        SET TIME = 3:0;
        ON PULSE PC-REG = PINS (1-10) + 1 D + 0D;
        SET TIME = 3:1;
        ON PULSE PINS (1-10) = PC-REG.

MICROMEMORY: MICROMEM SIZE = 40 * 1024;
        PINS ARE NUMBERED (1-50);
        SET TIME = 4:0;
        ON PULSE  PINS (11-50) = MEMORY (, PINS(1-10)).

CHIP: PIPELINE PINS ARE NUMBERED (1-64);
        REGISTERS ARE PL-REG(32);
        SET TIME = 5:0;
        ON PULSE  PINS (33-64) = PL-REG;
        SET TIME = 5:1;
        ON PULSE  PL-REG = PINS (1-32).

CHIP: ALU PINS ARE NUMBERED (1-40) AND NAMED D1, D2, D3, D4,
        D5, D6, D7, D8, AD1, AD2, AD3, AD4, AD5, AD6, AD7, AD8
        AD9,AD10,AD11, AD12, AD13, AD14, AD15, AD16, FLG, RCL1
        RCL2, FLC1, FLC2, AL1, AL2, AL3, AL4, AL5, AL6, AL7,
        AL8, AL9, NXT1, NXT2;

        REGISTERS ARE REC-LATCH (8), REG-A(8), REG-B(8),
        PGM-CNTR(16),NXT-ADDR(16),ACCUM(8), LINK(1),FLAGS (4);

        SUBREGISTERS OF FLAGS ARE ZERO 1 OVF 1, CRY 1, SIGN 1;
        CASCADE LINK ACC INTO LACC;CASCADE ACC LINK INTO ACCL;

        SET TIME = 6:0;
        IF PINS (26-27) NOT EQ 0 THEN REC-LATCH=PINS (11-18);
        IF PINS (26-27) EQ 01B THEN REG-A = REC-LATCH;
        IF PINS (26-27) EQ 10B THEN REG-A = REC-LATCH;
        IF PINS (26-27) EQ 11B THEN REG-A = 0, REG-B = 0;

        SET TIME = 6:1;
        IF PINS (30-38) EQ 1 THEN ACC = A + B;
        IF PINS (30-38) EQ 5 THEN ACC = A - B;
        IF PINS (30-38) EQ 2 THEN ACC = A & B;
        IF PINS (30-38) EQ 3 THEN ACC = A ! B;
        IF PINS (30-38) EQ 4 THEN ACC = A;
        IF PINS (30-38) EQ 6 THEN ACC = A XOR B;
        IF PINS (30-38) EQ 7 THEN ACC = B;
        IF PINS (30-38) EQ 8 THEN ACC = NEG ACC;
        IF PINS (30-38) EQ 9  THEN ACC = SL ACC;
        IF PINS (30-38) EQ 10 THEN ACC = SR ACC;
        IF PINS (30-38) EQ 11 THEN LACC= SL LACC;
        IF PINS (30-38) EQ 12 THEN ACCL= SR ACCL;
        IF PINS (30-38) EQ 13 THEN ACC = RL  ACC;
        IF PINS (30-38) EQ 14 THEN ACC = RL  ACC;
        IF PINS (30-38) EQ 15 THEN ACC = RR  ACC;
        IF ACC EQ 0 THEN ZERO = 1;
        IF ACC(8) EQ  1 THEN SIGN = 1 ELSE SIGN = 0;
        IF LINK EQ 1 THEN OVFL = 1;
        IF LINK EQ 1 THEN CRY = 1;

        SET TIME = 6:4;
        IF PINS (7-8) EQ 1 THEN NXT-ADR((1-8)) = ACC;
        IF PINS (7-8) EQ 2 THEN NXT-ADDR((9-16)) = ACC;
        IF PINS (7-8) EQ 3 THEN PINS(9-16) = ACC.

     /* NOW CONNECT ALL OF THE COMPONENTS TOGETHER USING
        THE CONNECT STATEMENT.  */

     CONNECT DATA-BUS TO MAIN-MEM     [(1-8)][(1-8)];
                      TO TERM1        [(1-8)][(1-8)];
                      TO TERM2        [(1-8)][(1-8)];
                      TO ADDR-SEL-ROM [(1-8)][(1-8)];
                      TO ALU          [(1-8)][(1-8)].

     CONNECT ADDRESS-BUS TO MAIN-MEM [(1-16)][(9-24)];
                         TO ALU      [(1-16)][(9-24)].

   CONNECT CNTL-BUS TO MICRO-MEM    [(21-30) (44-75)][(1-42)];
                    TO MM-PC        [(1-20)][(1-20)];
                    TO ADDR-SEL-MM  [(1-43)][(1-43)];
                    TO ADDR-SEL-ROM [(21-30)][(9-18)];
                    TO MAIN-MEM     [(93-95)][(25-27)];
                    TO TERM1        [(93-95)][(9-11)];
                    TO TERM2        [(93-95)][(9-11)];
                    TO ALU          [41 (76-92)][(25-42)];
                    TO PIPELINE     [(31-40)(42-95)]
                              [(3-12) 1 2 (33-64)(13-32)].

      START: MEMORY(,1) = 012H; MEMORY(,2) = 13H.
  END OF MICRO-PGMBL-INST-COMPUTER.

  END OF DESCRIPTIONS.
```

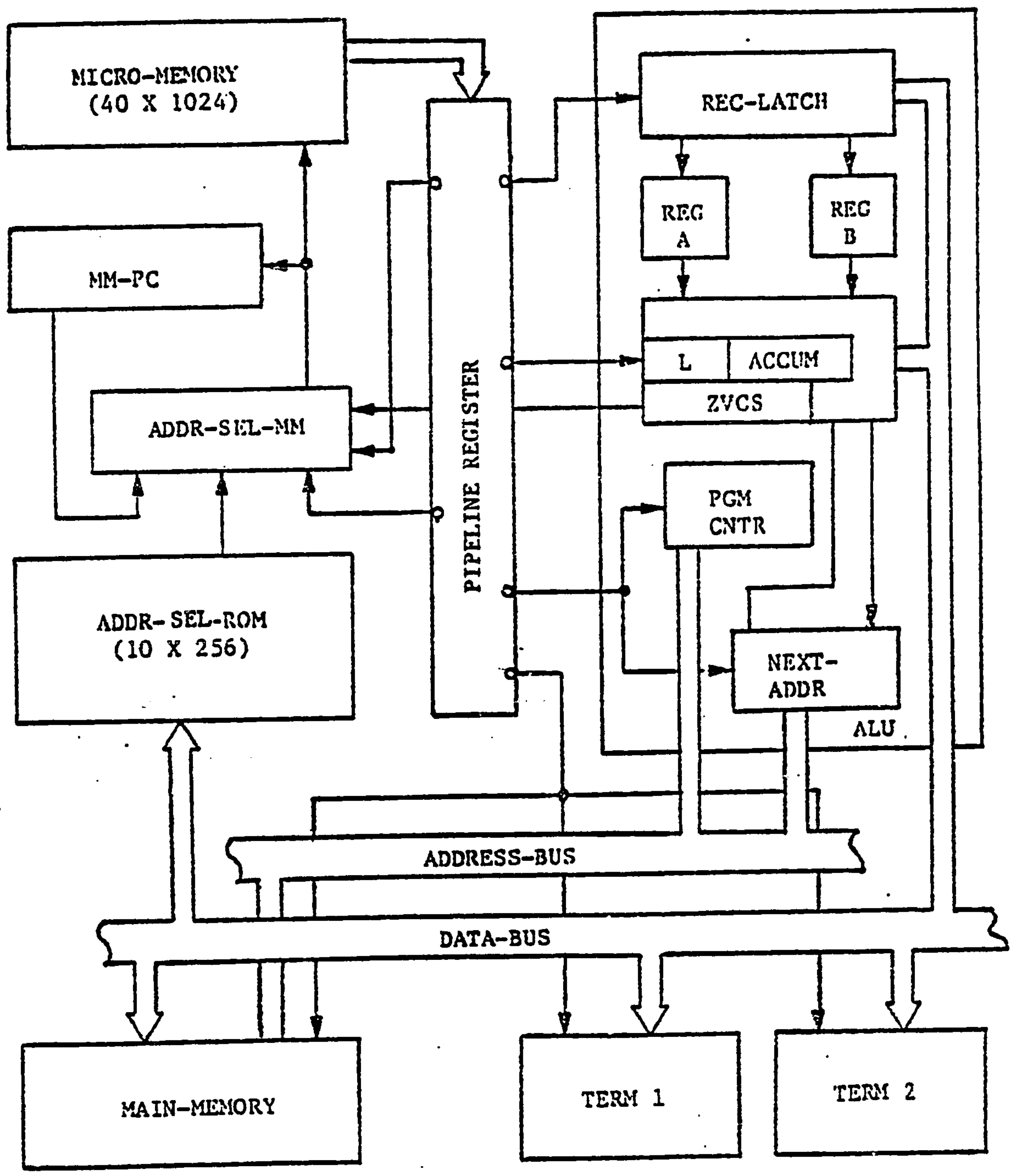


FIGURE 7 THE MIC COMPUTER

APPENDIX 3.8 THE AMD LEARNING KIT

The AMD-2900 learning kit has been designed by Advanced Micro Devices to assist the engineer not familiar with microprogramming techniques, to grasp the principles involved in microprogramming without excessive effort. This exercise must be worked in conjunction with the learning kit manual and should be studied concurrently with the board.

The introduction to the manual states "The purpose of the kit is twofold. First it is intended to be an instructional tool for the engineer faced with his first microprogramming job. The kit consists of one 2901 Bipolar microprocessor, one AMD-2909 bipolar microprogrammer sequencer, and several memories, registers, and multiplexers, organized in a typical CPU structure. It should be said at the outset that the purpose of this kit is to introduce the design engineer to the AM2900 family devices and provide a microprogram learning tool. This kit is not a four-bit computer".

A block diagram and picture of the board is shown for your convenience in figures 8.A and 8.B. These can be used to study the operation of the board. The Calsim description presented follows the plan of the board itself except for I-O. The I-O of the board is entirely binary using toggle switches to input zeros and ones; this has been converted to hexadecimal input-output for user convenience. The three sets of lights are displayed not as 12 LED's but as three hexadecimal numbers. The switches are set by the use of hexadecimal numbers.

```
SYSTEM NAME IS AMD2900LEARNING-BOARD.

LEVEL 1: AMD2900LEARNING-BOARD CONTAINS AM2901, AM2907,
         CCMUX, MUX1234, ST-REG, PIPELINE-REG, PROM,
         ADDR-SW, MIC-MEM, AM2909, DUMMY, DSPL-BOARD.

  /* WE WILL NOW DESCRIBE THE PIPELINE REGISTER */

COMPONENT: PIPELINE-REG PINS ARE NUMBERED (1-64) AND NAMED
    D0,D1,D2,D3, B0,B1,B2,B3, A0,A1,A2,A3, I3,I4,I5,CN,
    I0,I1,I2,MX0, I6,I7,I8,MX1, P0,P1,P2,P3, BR0,BR1,BR2,BR3
    D0OUT;
    REGISTER IS PL-REG(32);
    SET TIME = 2:0; ON PULSE PINS (33-64) = PL-REG;
    SET TIME = 2:1; ON PULSE PL-REG = PINS (1-32).
```

```
COMPONENT: AM2901 PINS ARE NUMBERED (1-40) AND NAMED

          /* THIS IS FOR DIP ONLY, FLAT HAS OTHER VALUES */
           A3,A2,A1,A0,I6,I8,I7,RAM3,RAM0,VCC,F0,I0,I1,
           I2,CP,Q3,B0,B1,B2,B3,Q0,D3,D2,D1,D0,I3,I5,I4,
           CN,GND,F3,GBAR,CN4,OVR,PBAR,Y0,Y1,Y2,Y3,OEBAR;

REGISTERS ARE Q-REG(4), RAM(4,16), RAM-SHIFT(4), Q-SHIFT(4);
  /* WE WILL SET UP SOME WORKING REGISTERS FOR OUR USE TO
     MAKE THE MANIPULATION A LITTLE EASIER TO HANDLE. */

REGISTERS ARE A(4), B(4), Q-LOOP(4), R(4), S(4), F(4);
SET TIME =  3:0;

  /* WE WILL WORK THE FIRST THREE VALUES OF THE OP CODE. */

 IF PINS [I0,I1,I2] EQ 0 THEN R = RAM( ,PINS[A0 A1 A2 A3]),
                             S = 0;
 IF PINS [I0,I1,I2] EQ 1 THEN R = RAM( ,PINS[A0,A1,A2,A3]),
                             S = RAM( ,PINS[B0,B1,B2,B3]);
 IF PINS [I0,I1,I2] EQ 2 THEN R = 0, S = Q-REG;
 IF PINS [I0,I1,I2] EQ 3 THEN S = RAM( ,PINS[B0,B1,B2,B3]),
                             R = 0;
 IF PINS [I0,I1,I2] EQ 4 THEN S = RAM( ,PINS[A0,A1,A2,A3]),
                             R = 0;
 IF PINS [I0,I1,I2] EQ 5 THEN S = RAM( ,PINS[A0,A1,A2,A3]),
                             R = PINS[D0,D1,D2,D3];
 IF PINS [I0,I1,I2] EQ 6 THEN R = PINS[D0,D1,D2,D3],
                             S = Q-REG;
 IF PINS [I0,I1,I2] EQ 7 THEN R = PINS[D0,D1,D2,D3], S = 0;

   /* LETS RESET THE TIME AND WORK I5,I4,I3 */

    SET TIME = 3:2;
    IF PINS[I3,I4,I5] EQ 0 THEN F = R + S;
    IF PINS[I3,I4,I5] EQ 1 THEN F = S - R;
    IF PINS[I3,I4,I5] EQ 2 THEN F = R - S;
    IF PINS[I3,I4,I5] EQ 3 THEN F = R ! S;
    IF PINS[I3,I4,I5] EQ 4 THEN F = R & S;
    IF PINS[I3,I4,I5] EQ 5 THEN F = R & S;
    IF PINS[I3,I4,I5] EQ 6 THEN F = R XOR S;
    IF PINS[I3,I4,I5] EQ 0 THEN F = NOT (R XOR S);

   /* LETS RESET TIME AND WORK I6, I7, I8 */

  SET TIME = 3:3;
  IF PINS [I6,I7,I8] EQ 0 THEN Y = F, Q = F;
```

```
    IF PINS [I6,I7,I8] EQ 1 THEN Y = F;
    IF PINS [I6,I7,I8] EQ 2 THEN Y = A RAM(,PINS (17-20)) = F;
    IF PINS[I6,I7,I8] EQ 3 THEN Y=F, RAM(,PINS (17-20)) = F/2;
    IF PINS[I6,I7,I8] EQ 4 THEN Y=F RAM(,PINS (17-20))  = F/2,
                Q = Q / 2, PIN R0 = F(1), PIN Q0 = Q-LOOP(1);
    IF PINS[I6,I7,I8] EQ 5 THEN Y=F, RAM(,PINS (17-20)) = F/2,
                          PIN R0 = F(1), PIN Q0 = Q-LOOP(1);
    IF PINS [I6,I7,I8] EQ 6 THEN Y = F, B= 2 * F, Q = Q*2,
                           PIN R3 = F(4), PIN Q3 = Q-LOOP(4);
    IF PINS[I6,I7,I8] EQ 7 THEN Y=F, RAM(,PINS (17-20)) = 2*F,
                           PIN R3 = F(4), PIN Q3 = Q-LOOP(4).

    /* IN THIS SYSTEM WE AVOID A BUS ALTOGETHER BY CONNECTING
       COMPONENTS TO EACH OTHER. SO THAT THERE IS ALWAYS SOME-
       THING TO CONNECT TO WE WILL ONLY CONNECT EACH COMPONENT
       AS WE GO TO THOSE ALREADY DESCRIBED. */

  COMPONENT: DUMMY PINS ARE NUMBERED (1-1).
    /* DUMMY CONNECTS WITH THE LOOSE WIRES SO THERE WILL BE
       NO LISTED ERRORS FOR UNCONNECTED WIRES. */

  CONNECT AM2901 TO PIPELINE-REGISTER
          [(1-7)(12-14)(17-20)(22-29)]
          [12, 11, 10, 9, 21, 23, 22, 17, 18, 19, (5-8)
           4, 3, 2, 1, (13-16)];
          TO DUMMY [15, 32, 35, 40] [1 1 1 1].

  CHIP: MUX1234 PINS ARE NUMBERED (1-8) AND NAMED CNTL1, CNTL2
        RAM0, RAM3, Q0, Q3, F0, ZERO;
       /* THE MUX LOGIC IS GIVEN ON PAGE 3-9 OF THE AMD-2900
          MANUAL. IT CONSISTS OF THE FOLLOWING:
          MUX CODE           UP                DOWN
            0 0        SHIFT UP 0->RAM0      SHIFT DOWN RAM0,-0
            0 1        ROTATE                ROTATE
            1 0        DOUBLE ROTATE THROUGH RAM AND Q
                       Q3 -> RAM0            RAM0 -> Q3
                       RAM3 -> Q0            Q0 -> RAM0
            1 1        DOUBLE SHIFT          DOUBLE SHIFT
                       0 -> Q0 Q3->RQM0      F3->RAM3 RAM0->Q3 */
    /* NOTE THAT THE FOUR MUX ARE COMBINED INTO ONE. */

    /* WE SET VALUES ON PINS PRIOR TO CPE ACTION. */

  SET TIME = 3:4;
  IF PINS(1-2) EQ 00 THEN PINS (3-6) = 0;

       /* PIN 8 USED HERE AS A WORKING REGISTER. */
```

```
IF PINS(1-2) EQ 01 THEN PIN 8 = PIN 3 PIN 3 = PIN 4, PIN4 =
   PIN 8  PIN 8 = PIN 5, PIN 5 = PIN 6, PIN 6 = PIN 8;
IF PINS (1-2) EQ 10 THEN PIN 8 = PIN 3, PIN 3 = PIN 6, PIN 6
   = PIN 8, PIN 8 = PIN 5, PIN 5 = PIN 4, PIN 4 = PIN 8.


CONNECT MUX1234 TO PIPELINE-REGISTER [1 2] [20 24];
                TO AM2901 [(3 - 7)] [9 8 21 16 31].

CHIP: AM2909 PINS ARE NUMBERED (1-28) AND NAMED /* JUST
   AS THEY ARE NAMED IN THE AMD DIP. */ REB, R3, R2, R1, R0,
   OR3, D3, OR2, D2, OR1, D1, OR0, D0, GND, ZERO, S0, S1,
   Y0, Y1, Y2, Y3, OEB, CN, CN4, FEB, PUP, CP, VCC;
   REGISTERS ARE HOLDING-REG (8), ST-PNTR (2), STACK(8,4),
   PC-REG(8), INCREMENTER(8), I(1);
   ON, PC-REG(I) = 10 + 0Q + 0B;

   SET TIME = 8:0;

   /* THIS LOGIC IS BASED ON THE TABLE BELOW WHICH WAS TAKEN
      FROM FIGURE 5 PAGE 2-78 AMD-2900 FAMILY DATA BOOK.
                           OUTPUT CONTROL
          ZERO      OE        YI
           X        H          Z
           L        L          L
           H        L        SOURCE SELETED BY S0 S1   */

   IF OE EQ 1 THEN Y = ZERO, RESET TIME = 5:0;
   IF (ZERO EQ 0 AND OE EQ 0) THEN Y = 0, RESET TIME = 5:0;
   SET TIME = 8:1;

   /* NOW CONTROL IS FROM S0 S1 AS PER TABLE BELOW
       OCTAL    S1    S0    SOURCE FOR Y OUTPUTS     SYMBOL

           0       L     L     MICROPROGRAM COUNTER      MPC
           1       L     H     REGISTER                  REG
           2       H     L     PUSH-POP STACK            STK0
           3       H     H     DIRECT INPUTS             DI   */

  IF PINS[S0 S1] EQ S0 THEN PINS [YS0 Y1 Y2 Y3 Y4 Y5 Y6 Y7] =
       PC-REG, RESET TIME = 5:0;
  IF S1 EQ 0 AND S0 EQ 1 THEN PIN[Y0 Y1 Y2 Y3 Y4 Y5 Y6 Y7] =
                   HOLDING-REG, RESET TIME = 5:0;
  IF S1 EQ 1 AND S0 EQ 1 THEN PIN [Y0 Y1 Y2 Y3 Y4 Y5 Y6 Y7] =
             PINS [D1 D2 D3 D4 D5 D6 D7 D8],RESET TIME = 5:0;
    SET TIME = 8:3;
   /* WE NOW HAVE PUSH-POP CASE WHICH IS BASED ON FOLLOWING:
        FE      PUP      PUSH-POP STACK CHANGE
        H        X       NO CHANGE
        L        H       INCREMENT STACK POINTER THEN PUSH
                         CURRENT PC ONTO STK0
        L        L       POP STACK (DECREMENT SP)    */
   IF FE EQ 1 OR FE NOT EQ 0 THEN PINS[(12-15), (21-24) ] =
        PINS[13 11 9 7 (25-28)],  RESET TIME = 5:0;
   IF FE EQ 0 AND PUP EQ 0 THEN PINS[(12-15)(21-24)] =
                              STACK (4), SP = SP - 1;
   IF SP EQ 0 THEN SP = 4;
   IF FE EQ 0 AND PUP EQ 0 THEN RESET TIME = 5:0;

    /* INCREMENT AND PUSH REMAINS. */
   ON SP = SP + 1; IF SP EQ 5 THEN SP = 1;
   ON PULSE STACK(4) = PC-REG;
   ON PULSE PINS[(12-15)(21-24)] = PINS[13 1 9 7 (25-28)].

CONNECT AM2909 TO PIPELINE-REGISTER [(2-5)][(29-32)];
         TO DUMMY [6, 15, 22, 23, 24, 27, 1][1 1 1 1 1 1 1].

CHIP: ST-REG /* LS08 */ PINS ARE NUMBERED (1-15) AND NAMED
      07, Q0, X, D0, D1, X, Q1, X, CP, Q2, X, D2, D3, X, Q3.
     TIME = 4:0;
     ON PULSE PINS (2 7 10 15) = PINS (4 5 12 13).

CONNECT ST-REG TO AM2901 [4 5 12 13][11 31 34 33];
               TO DUMMY [3 6 8 9 11 14][1 1 1 1 1 1].

CHIP: CCMUX /*AMD-9309 */ PINS ARE NUMBERED (1-14) AND NAMED
      CCE, X, S1, I0B, I1B, I2B, I3B, X, I3A, I2A, I1A, I0A,
      S0, W21B.
      TIME=5:0;
      IF PIN 4 EQ 1 OR PIN 5 EQ 1 OR PIN 6 EQ 1 OR PIN 7 EQ
         1 THEN PIN 1 = 1 ELSE PIN 1 = 0.

CONNECT CCMUX TO ST-REG [(4-7)][2 7 10 15];
 TO PIPELINE-REGISTER [3 13 14 (9-12)][25 24 22 22 22 22 22]

CHIP: AM2907 PINS ARE NUMBERED (1 - 20) AND NAMED RLEB, R0,
      A0, BUS0, GND1, BUS1, A1, R1, BEB, OFB, OEB, R2, A2,
      BUS2, GND2, BUS3, A3, R3, DRCP, VCC.
      SET TIME= 3:8;
      ON PULSE PINS (2 8 12 18) = PINS (3 7 13 17);
      ON PULSE PINS (4 6 14 16) = PINS (3 7 13 17).

CONNECT AM2907 TO AM2901 [3, 7, 13, 17][(36-39)];
```

```
                    TO DUMMY [1, (9-11)][1, 1, 1, 1].

MICROMEMORY: MIC-MEM PINS ARE NUMBERED (1-36),
    TIME = 1:0
    ON PULSE PINS (1-32) = MEMORY (PINS(33-36)).

CONNECT MIC-MEM TO PIPELINE-REGISTER[(1-32)][(33-64)];
                TO AM2909 [(33-36)][(18-21)].


CHIP: PROM PINS ARE NUMBERED (1-14) AND NAMED Q0 Q1 Q2 Q3 Q4
      Q5 Q6 Q7 X A0 A1 A2 A3 A4.
      TIME = 6:0;

      REGISTER SLCT(5); ON PULSE SLCT = PINS (10-14);
      REGISTER OUTPUT(8);
      IF SLCT=0                THEN OUTPUT = 8AH;
      IF SLCT=1                THEN OUTPUT = 81H;
      IF SLCT=2 OR SLCT=3      THEN OUTPUT = 0AH;
      IF SLCT=4 OR SLCT=5      THEN OUTPUT = 02H;
      IF SLCT=6 OR SLCT=7      THEN OUTPUT = 0EH;
      IF SLCT=8                THEN OUTPUT = 89H;
      IF SLCT=9                THEN OUTPUT = 82H;
      IF SLCT=10 OR SLCT=11 THEN OUTPUT = 09H;
      IF SLCT=12 OR SLCT=13 THEN OUTPUT = 04H;
      IF SLCT=14 OR SLCT=15 THEN OUTPUT = 06H;
      IF SLCT=16               THEN OUTPUT = 86H;
      IF SLCT=17               THEN OUTPUT = 80H;
      IF SLCT=18 OR SLCT=19 THEN OUTPUT = 01H;
      IF SLCT=20 OR SLCT=21 THEN OUTPUT = 00H;
      IF SLCT=22               THEN OUTPUT = 86H;
      IF SLCT=23               THEN OUTPUT = 80H;
      IF SLCT=24 OR SLCT=26 THEN OUTPUT = 82H;
      IF SLCT=25 OR SLCT=27 THEN OUTPUT = 8AH;
      IF SLCT=28 OR SLCT=30 THEN OUTPUT = 82H;
      IF SLCT=29 OR SLCT=31 THEN OUTPUT = 8AH;
      SET TIME = 6:9; ON PULSE PINS (1-7) = OUTPUT.

CONNECT PROM TO PIPELINE REGISTER [(11-14)][(25-28)];
             TO AMD2909 [(5-7) 1][12 10 8 26];
             TO ST-REG [8][1];
             TO DUMMY [9][1];
             TO CC-MUX[10][1].

COMPONENT: DSPL-BOARD IS NUMBERED (1-32) AND NAMED Y3, Y2,
           Y1, Y0, MP3, MP2, MP1, MP0, CN4, OVF, F3, F0, PAR
           CCE, PBAR, GBAR, ST3, ST2, ST1, ST0, Q0, Q3, RAM0
```

```
                RAM3, BUS3, BUS2, BUS1, BUS0, R3, R2, R1, R0;
                REGISTER: DSP (4,8);
                TIME = 1:9; ON PULSE DSP = PINS (1-32);
                TIME = 2:9; ON PULSE DSP = PINS (1-32);
                TIME = 3:9; ON PULSE DSP = PINS (1-32);
                TIME = 4:9; ON PULSE DSP = PINS (1-32);
                TIME = 5:9; ON PULSE DSP = PINS (1-32);
                TIME = 6:9; ON PULSE DSP = PINS (1-32);
                TIME = 7:9; ON PULSE DSP = PINS (1-32);
                TIME = 8:9; ON PULSE DSP = PINS (1-32).

CONNECT DSPL-BOARD TO AMD2909[(1-4)][21 20 19 18];
                   TO AMD2901 [(5-12) (21-24) 18 19]
                              [39 38 37 36 21 16 9 8 35 32];
                   TO AMD2907 [16 (25-32)]
                              [10 16 14 6 4 18 12 8 2];
                   TO ST-REG  [(20-23) 17][2 7 10 15 1].

CHIP: ADDR-SW  PINS ARE NUMBERED (1-14) AND NAMED S, 1A 1B
      1Y 2A 2B 2Y X 3Y 3B 3A 4Y 4B 4A.
      TIME = 7:0;
      ON PULSE PIN 9 = PIN 14, PIN 7 = PIN 5, PIN 4 = PIN 2.

 CONNECT ADDR-SW TO PROM [2 5 11][(2-4);
                 TO AMD2909 [4 7 9][25 17 16];
                 TO DUMMY[3 6 10 13 14 1 12][1 1 1 1 1 1 1].

END OF AMD2900LEARNING-BOARD.

END OF DESCRIPTIONS.
```

TABLE 8.I MICROPRORGAM FIELD DESCRIPTIONS

RAM & MUX SEL	ROM LOCATION	BIT NUMBER	BIT DEFINTION	FIELD DEFINITION
0	U2	0	D0	"D" DATA
		1	D1	
		2	D2	
		3	D3	
1	U3	4	B0	"B" ADDR
		5	B1	
		6	B2	
		7	B3	
2	U4	8	A0	"A" ADDR
		9	A1	
		10	A2	
		11	A3	
3	U5	12	I3	ALU SEE TABLES IN AM2901 DATA SPECIFICATION.
		13	I4	
		14	I5	
		15	CN	
4	U6	16	I0	ALU SEE TABLES
		17	I1	
		18	I2	
		19	MUX0	SEE MUX TABLE
5	U7	20	I6	ALU SEE TABLES
		21	I7	
		22	I8	
		23	MUX1	SEE MUX TABLE
6	U8	24	P0	SEE TABLES FOR NEXT INSTRUCTION
		25	P1	
		26	P2	
		27	P3	
7	U9	28	BRO	BRANCH ADDRESS
		29	BR1	
		30	BR2	
		31	BR3	

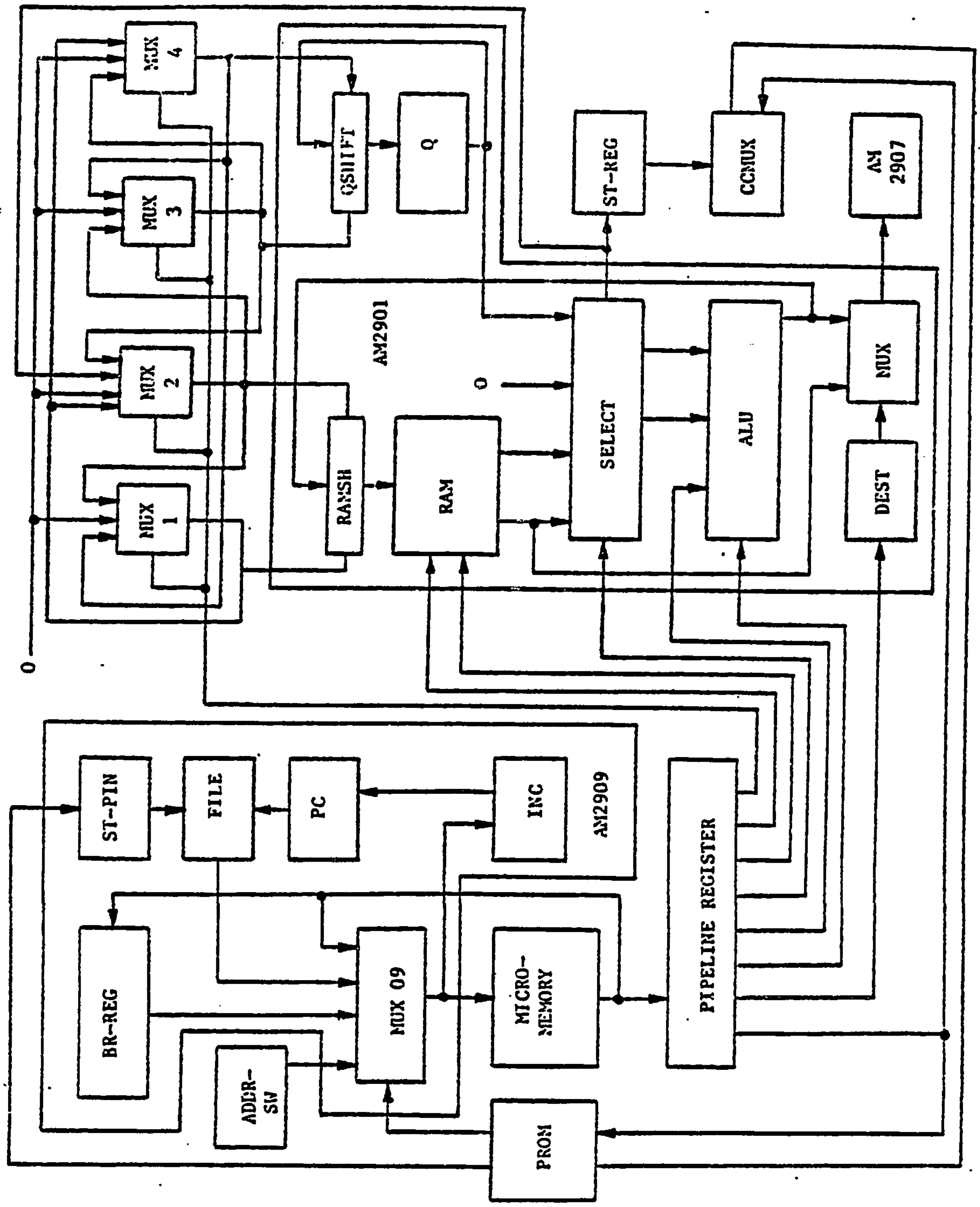


FIGURE 8A AMD LEARNING KIT BOARD BLOCK DIAGRAM

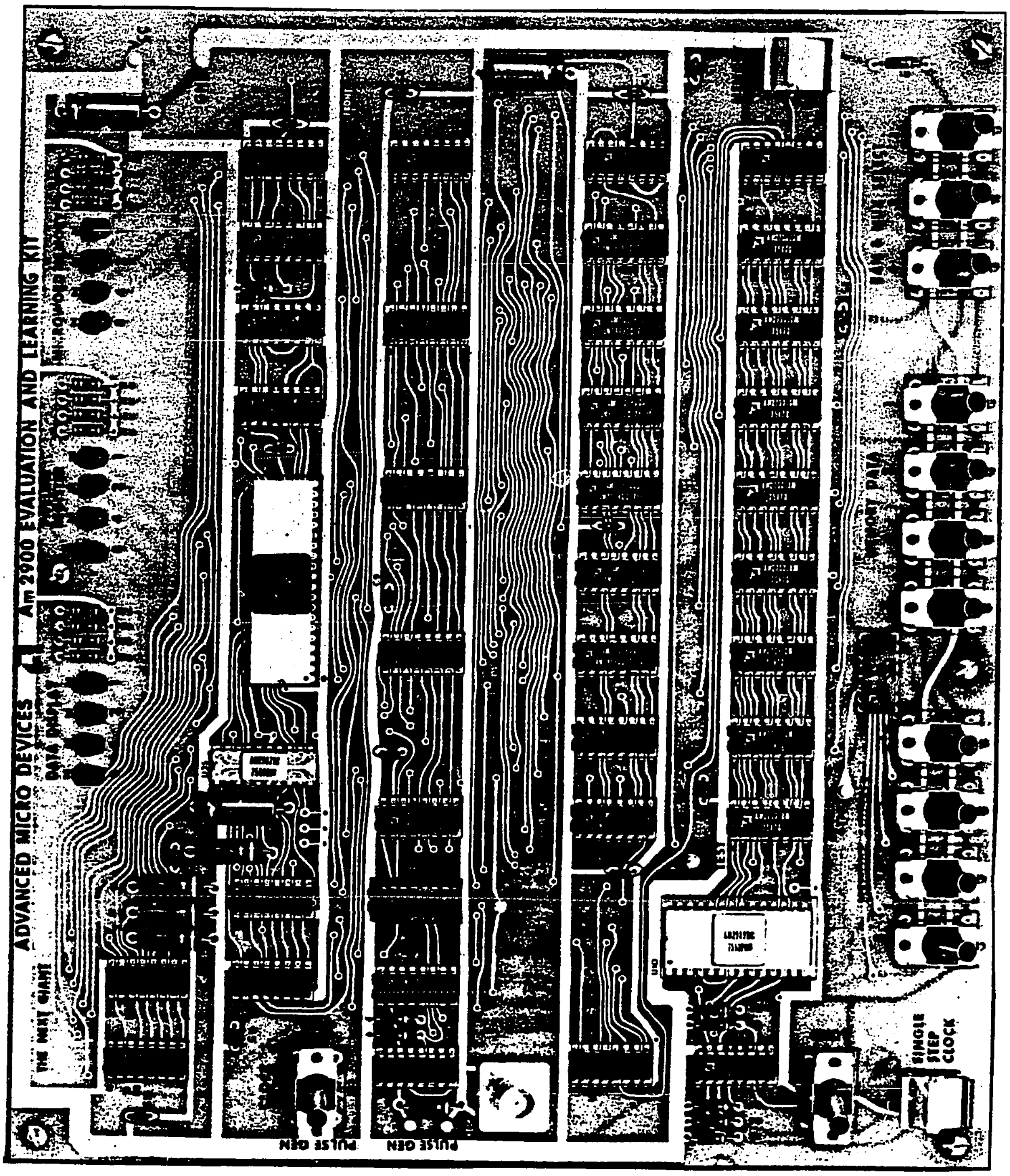


FIGURE 8B AMD LEARNING KIT BOARD

WORKS CITED

(1) Advanced Micro Devices, "Build A Microcomputer", 901 Thompson Place, Sunnyvale, Cal, 1979.

(2) Advanced Micro Devices, "The AM2900 Family Databook", 901 Thompson Place, Sunnyval, Ca, 94086.

(3) Advanced Micro Devices, "AM2900 Learning and Evaluation Kit and User's Manual", 901 Thompson Place, Sunnyvale, Ca, 1976.

(4) B. Agule, G. Goertzel, and H. Ofek, "LCD - Language for Computer Design", Proceedings of 1975 International Symposium on Computer Hardware Description Languages and Their Applications, p 180.

(5) N. A. Alexandridis "Bit-sliced Microprocessor Architecture", Computer, June 1978, pp 56-80.

(6) Alfred V. Aho and J.D. Ullman, "Principles of Compiler Construction", Addison-Wesley, 1977.

(7) P. F. Analuf and P. Meinen, "PHPL - A New Hardware Description Language for Modular Descriptions of Logic and Timing", Proceedings of the 4th International Symposium on Computer Hardware Description Languages, 1979, pp 124-130.

(8) J. R. Armstrong and G. W. Woodruff, "Chip level Simulation of Microprocessors", Computer, January 1980, pp 94-100.

(9) Russel M. Armstrong, "Modular Programming in Cobol", John Wiley & sons, 1973.

(11) Jean-Loup Baer, "Computer Systems Architecture", Computer Science Press, 1980.

(12) Jonathan Bara and R. Born, " A CDL Compiler for Designing and Simulating Digital Systems at the Register Level", Proceedings of 1975 International Symposium on Computer Hardware Description Languages and Their Applications, pp 96-102.

(13) M. R. Barbacci and D. P Siewiorek, "Applications of an ISP Compiler in a Design Automation Laboratory", Proceedings of 1975 International Symposium on Computer Hardware Description Languages and Their Applications, pp 69-75.

(14) M. R. Barbacci, "A Comparison of Register Transfer Languages for Describing Computers and Digital Systems", IEEE Transactions on Computers, vol C-24, No 2, February 1975, pp 137-150.

(15) H. M. Bayegan, "Computer Aided Schematic Systems", Proceedings of the 14'th Annual Design Automation Conference, June 1977, pp 396-404.

(16) H. M. Bayegan, O. Baadsvik and O. Kirkaune, "An Interactive Graphic High Level Language for Hardware Design", Proceedings of the 4'th International Symposium on Computer Hardware Description Languages, 1979, pp 184-190.

(17) Maurice Blackman, "The Design of Real Time Applications", John Wiley, 1978.

(18) D. Borrione, "LASCAR: A Language for Simulation of Computer Architecture" Proceedings of the 1975 International Symposium on Computer Hardware Description Languages and Their Applications, pp 143-152.

(19) Y. Bressy, Y. David, J. Fantino, , J. Mermet, "A Hardware Compiler for Interactive Realization of the logical Systems Described in Cassandre", Proceedings of the 1975 International Symposium on Computer Hardware Description Languages and Their Applications, pp 62-68.

(20) John Brick and John Mick, "Microprogramming Ups Your Options", EDN, January 20, 1978, pp 105-110 and EDN February 5, 1978, pp 53-56.

(21) H. R. Burris, "Instrumented Architectural Level Emulation Technology", National Computer Conference, 1977, pp 937-946.

(22) W. C. Carter, W. H. Joyner, and D. Bland, "Microprogram Verification Considered Necessary", Proceedings of 1978 National Computer Conference, AFIPS, pp 657-664.

(23) S. S. Ching and J.H. Tracey, "An Interactive Computer Graphics Language for Design and Simulation of Digital Systems", Computer, June 1977, pp 35-41.

(24) Yaohan Chu, "An lgol-like Computer Design Language", Communications of ACM, October 1965, pp 607-615.

(25) Yaohan Chu, "Computer Organization and Microprocessors", Prentice-Hall, 1972.

(26) Yaohan Chu, O. N. Garcia, M. A. Brever, J. P. Hayes, Nowle, R. Hartenstein, P.C. Barr, F.J. Hill, G. R. Peterson, J. Livoski, "Why Do We Need Computer Languages?", Computer, December 1974, pp 18-22.

(27) Yaohan Chu, "Introducing CDL", Computer, December, 1974, pp 31-33.

(28) Yaohan Chu, "Foreward and Introduction" (Special issue on Microcomputing) IEEE Transactions on Computers, October, 1976, p 961.

(29) Computer Science Corp., "AMDASM", (A product description), 650 North Sepulveda Blvd., El Segundo, Ca, 90245.

(30) E. D. Crocket, et.al. "Computer Aided System Design", Proceedings of 1970 Fall Joint Computer Conference, pp 287-296.

(31) J. A. Darrington, "A Language for the Description of Digiital Computer Processes", Proceedings of the Design Automation Workshop, July 15-18, 1968, pp 15-1 to 15-8.

(32) Scott Davidson and Bruce D. Shriver, " An Overview of Firmware Engineering", Computer, May 1978, pp 21-23.

(33) Bulent Dervisglu, "Hardware Description Languages in Great Britian", Computer, December, 1974, pp 64-66.

(34) Alvin M. Despain, "The Use of Two CHDL Systems, PMS and DIDL in the Design of a Fourier Transform Processor", Proceedings of 1975 International Symposium on Computer Hardware Description Languages and Their Applications, pp 76-84.

(35) D. L. Dietmeyer, "Introducing DDL", Computer, December, 1974, PP 34-38.

(36) J. R. Duley, "DDL - A Digital System Design Language" PhD Dissertation, University of Wisconsin, Madison, 1967.

(37) C. J. Evangelisti, "Designing with LCD Language for Computer Design", Proceedings of IEEE 14'th annual Design Automation Conference, June, 1977, pp 369-376.

(38) P. L. Flake, G. Musgrove, and M. Shorland, "The HILO Logic Simulation Language", Proceedings of 1975 International Symposium on Computer Hardware Description Languages and Their Applications", pp 134-142.

(39) Patrick. W. Foulk, "The Formal Design of Parallel Hardware", Proceedings of the 4th International Symposium on Computer Hardware Description Languages, 1979, pp 162-168.

(40) E. A. Franke, "Automated Functional Design of Digital Systems", PhD Thesis, Case Western Reserve University November, 1967.

(41) W. R. Franta and W. K. Giloia, "APL*DS: A Hardware Description Language for Design and Simulation", Proceeding of the 1975 International Symposium on Computer Hardware Description Languages and Their Applications, pp 45-52.

(42) S. H. Fuller, V. R. Lesser, C. G. Bell, and C. H. Kaman, "The Effects of Emerging Technology and Emulation Requirements on Microprogramming", IEEE Transactions on Computers, vol c-25, no. 10, October, 1976, pp 1000-1009.

(43) Rudolph Gardill and Ranier Klar, "Storage Declarations in the Hardware Description Language ERES", ACM - Sigda, vol. 9, no. 2, June, 1979, pp 21-33.

(44) G. E. Gardner, G. Estrin, and H. Potash, "A Structural Modelling Language for Architecture of Computer Systems", Proceedings of 1975 International Symposium on Computer Hardware Description Languages and Their Applications, pp 161-171.

(45) Leonard Gilman and Allan J. Rose, "APL an Interactive Approach", John Wiley & Sons, 1971.

(46) P. Gordon and S. Stallard, "Microprogrammed CPU Architecture Offers User-alterable Minicomputer Performance", Computer Design, June, 1978, pp 91-99.

(47) W. Goerke and W. J. Hoffman "Simulation of Switching Circuits by a New Hardware Description Language", Proceedings of 1975 International Symposium Computer Hardware Description Languages and Their Applications pp 125-133.

(48) J. R. Heath, B. D. Carroll, and T. T. Cwik, "CDL - A Tool for Concurrent Hardware and Software development", Proceedings of Design Automation Conference, June 20-22, New Orleans, Louisiana, IEEE catalog 77, CH1216-1C, pp 445-449.

(49) J. R. Heath, B. D. Carroll, and T. T. Cwik, "Capabilities and Limitations of CDL as a system Hardware and Software Descign Aid", Journal of Design Automation and Fault Tolerant Computing, vol. 2 May, 1979, pp 230-240.

(50) G. R. Hellestrand, "MODAL: A System for Digital Hardware Description and Simulation", Proceedings of the 4th International Symposium on Computer Hardware Description Languages, 1979, pp 131-137.

(51) C. W. Hemming Jr. and J. W. Hemphill, "Digital Logic Simulation Models and Evolving Technology", Proceedings of the 12'th Automation Design Conference, June 1975, pp 205-214.

(52) D. D. Hill, "ADLIB: A Modular, Strongly-typed Computer Design Language", Proceedings of the 4th International Symposium on Computer Hardware Description Languages, 1979, pp 75-81.

(53) F. J. Hill, "Introducing AHPL", Computer, December, 1974, pp 28-30.

(54) F. J. Hill, "Updating AHPL", Proceedings of the 1975 International Symposium on Computer Hardware Description Languages and Their Applications, 1975, pp 22-29.

(55) F. J. Hill and Z. Navabi, "Extending the Second Generation AHPL Software to Accomodate AHPL III", Proceedings of the 4th International Symposium on Computer Hardware Description Languages, 1979, pp 47-53.

(56) J. L. Houle, "A Formal Language for the Description Model and Realization of Digital Systems", PhD Thesis, Computer Science Department, University of Waterloo, 1974.

(57) Intel Corp., "Intel Series 3000 Programming Manual" 3065 Bowers ave, Santa Clara, Cal.

(58) M. S. Jayakumar and T. M. McCalla Jr., "Simulation of Microprocessor Emulation Using GASP-PL/I", Computer, April 1977, pp 20-26.

(59) W. A. Johnson, J.J. Crowley, and J.D. Ray, "Mixed Level Simulation from a Hierarchical CHDL", ACM-SIGDA Newsletter, vol 10, number 1, January 1980, pp 2-10.

(60) J. M. Kerridge and N. Willis, "A Simulator for Teaching Computer Architecture", ACM-SIGCSE Bulletin, vol. 12, number 2, July 1980, pp 65-71.

(61) Rainer Klar, "The Status of the Implementation of ERES", Proceedings of 1975 International Symposium on Computer Hardware Description Languages and Their Applications, p 187.

(62) E. E. Klingman, "Comparisons and Trends in Microprocessor Architecture", Computer Design, September, 1977, pp 83-92.

(63) W. R. LaLonde, An LALR Grammar Analyzer running on the IBM-4341 prepared by the Computer System Research Group, University of Toronto.

(64) F. W. Lancaster and E. G. Fayen, "Information Retrieval On Line", Melville Pub. Co.

(65) Stephen Lau, "Bit-slice Microprogramming Saves Software Compatability", EDN March 5, 1978, pp 41-46 and March 20, 1978, pp 67-74.

(66) J. A. N. Lee, "VDL - A Definitional System for all Levels", Proceedings of the first Annual Symposium on Computer Architecture, Computer Architecture News, December, 1973, vol 2, no. 4, pp 41-48.

(67) C. K. C. Leung, "ADL an Architecture Description Language", Proceedings of the 4th International Symposium on Computer Hardware Description Languages, 1979, pp 6-13.

(68) D. W. Lewis, "A Hardware Compiler for Mano's RTL", Proceedings of the 4th International Symposium on Computer Hardware Description Languages, 1979, pp 145-150.

(69) I. Lewis and A. M. Peskin, "The MODEL/LINDA Design Automation System", Proceedings of the 1975 International Symposium on Computer Hardware Description Languages", pp 53-61.

(70) G. Jack Lipovski, "On Gray Box Descriptions of Microprocessors", Proceedings of 1975 International Symposium on Computer Hardware Description Languages and Their Applications", pp 184-186.

(71) G. J. Lipovski, "Hardware Description Languages, Voices from the Tower of Babel", Computer, June 1977, pp 14-17.

(72) R. W. Marczynski, W. T. Pulczyn and J. M. Sochacki, "OSM: Microprogrammed Hardware Structure Description Language", Proceedings of 1975 International Symposium on Computer Hardware Description Languages and Their Applications, pp 172-178.

(73) R. W. Marczynski and P. Bakowski, "What Do The Hardware Description Languages Describe?" Proceedings of the 4th International Symposium on Computer Hardware Description Languages, 1979, pp 178-183.

(74) G. F. Maxey and E. T. Organick, "CASL - A Language for Automating the Implementation of Computer Architectures", Proceedings of the 4th International Symposium on Computer Hardware Description Languages, 1979, pp 102-107.

(75) J. R. Mick, "AM2900 Bipolar Microprocessor Family", Proceedings of Eighth Annual Workshop on Microprogramming, September 1975, pp 56-63.

(76) G. J. Nutt, " A Case Study of Simulation As a Computer System Design Tool", Computer, October 1978, pp 31-36.

(77) A. C. Parker and J. W. Gault, "A Language for the Specification of Digital Interfacing Problems", Proceedings of 1975 International Symposium on Computer Hardware Description Languages and Their Applications, pp 85-90.

(78) A. C. Parker, D. E. Thomas, Stephen Crocker, and R.G. Cattell, "ISPS: A Retrospective View", Proceedings of the 4th International Symposium on Computer Hardware Description Languages, 1979, pp 28-32.

(79) D. A. Patterson, "STRUM: Structured Development System Correct Firmware", IEEE Transactions on Computers, October 1976.

(80) R. Piloty, "Hardware Description Languages in the Federal Republic of Germany", Computer, December, 1974, pp 57-58.

(81) R. Piloty, "Segmentation Constructs for RTS III, a Computer Hardware Description Language Based on CDL", Proceedings of the 1975 International Symposium on CHDL and Their Applications, pp 115-124.

(82) Reinhardt Posch, "Modelling a Hardware Structure for Computer Science Education", ACM-SIGCSE Bulletin, volume 11, Number 2, June 1979, pp 60-68.

(83) H. Potash, et.al., "DCDS Simulating System", Proceedings of 1970 JCC, pp 707-720.

(84) F. J. Ramig, "Digitest II: An Integrated Structural and Behavioral language", Proceedings of the 1975 International Symposium on Computer Hardware Description Languages and Their Applications pp 38-44.

(85) Justin Rattner, Jeane-Claude Cornet and M. E. Hoff, Jr., "Bipolar LSI Computing Elements Usher in New Era of Digital Design", Electronics, September 5, 1974, pp 89-96.

(86) R. F. Rosin, G. Frieder, and R.H. Eckhouse, Jr., "An Environment For Research in Microprogramming And Emulation", Communications of ACM, Volume 15, Number 8, August 1972, pp 748-760.

(87) H. P. Schlaeppi, "A Formal Language for Describing Machine Logic, Timing and Sequencing (LOTIS)", IEEE Transactions on Electronic Computers, August 1964, pp 439-448.

(88) H. Schorr, "Computer Aided Digital Design and Analysis Using a Register-Transfer Language", IEFE Transactions on Electronic Computers, vol EC-13, December 1964, pp 730-737.

(89) Sajjan G. Shiva, "Computer Hardware Description Languages -- A Tutorial", Proceedings of IEEE, vol. 67, no. 12, December 1979, pp 1605-1615.

(90) Dan Sieworek, "Introducing ISP", Computer, December, December, 1974, pp 42-44.

(91) Dan Sieworek, "Introducing PMS", Computer, December, 1974, pp 42-44.

(92) W.A. Skelton and R.S. Walker, "Simulation of Computer Architecture Using a Hierarchical Computer Hardware Description Language", Proceedings of the National Educational Computer Conference 1981, pp 115-120.

(93) David R. Smith, "Computer Structure Language", Proceedings of the 1975 International Symposium on CHDL'S and Their Applications, PP 153-160.

(94) J. H. Stewart, "LOGAL: A CHDL for Logic Design and Synthesis of Computers", Computer, June 1977, pp 18-26.

(95) L. R. Stine and F.J. Mowle, "A Position Paper on Extensions to the Computer Design Language", Proceedings of the 1975 International Symposium on Hardware Description Languages, 1975, pp 103-114.

(96) Y. H. Su, "A Survey of Computer Hardware Description Languages in the U.S.A.", Computer, December 1974, pp 45-51.

(97) S.Y. Su and M.B. Baray, "LALSD - A language for Automated Logic and System Design", Proceedings of the 1975 International Symposium on Computer Hardware Description Languages and Their Applications pp, 30-31.

(98) Y. H. Su, "Hardware Description Language Application An Introduction and Prognosis", Computer, June 1977, pp 10-13.

(99) Ivan Tomek, "HARD -- Hardware Simulation in Education", ACM-SIGCSE Bulletin, vol 13 number 1, February 1981, pp 268-270.

(100) W. M. vanCleemput, "An Hierarchical Language for the Structural Description of Digital Systems", Proceedings of 14'th Annual Design Automation Conference, June 1977, pp 377-385.

(101) W. M. VanCleemput, "A Digital Design Automation Course For Logic Designers", ACM-SIGDA Newsletter, Volume 8, Number 1, March 1978, pp 13-15.

(102) J. Vaucher, "Hardware Description Languages in Canada", Computer, December 1974, pp 53-66.

(103) E. W. Vogel, "A Model Approach to the Description of Hardware Systems", Proceedings of 1975 International Symposium on Computer Hardware Description Languages and Their Applications, pp 32-37.

(104) J. J. Wallace and A. C. Parker, "SLIDE: An I/O Hardware Descriptive Language", Proceedings of the 4th International Symposium on Computer Hardware Description Languages, 1979, pp 82-88.

(105) M. J. R. Williams and R. W. McGuffin, "A High Level Logic Design System", Proceedings of the 4th International Symposium on Computer Hardware Description Languages, 1979, pp 40-46.

(106) P. Wegner, "The Vienna Definition Language", Computer Surveys, vol no. 1, March 1972, pp 5-63.

(107) M. V. Wilkes, "The Best Way to Design an Automatic Calculating Machine", Manchester University Computer Inaugral Conference, 1951, pp 16-21.

(108) M. V. Wilkes and J. B. Stringer, "Microprogramming and the Design of the Control Circuits in an Electonic Digiital Computer", Cambridge Philosophical Society, vol. 49, (April 1953), pp 230-238.

(109) M. V. Wilkes, "The Growth of Interest in Microprogramming: A literature Survey", Computer Surveys, vol 1, no. 3, September 1969, pp 139-145.

OTHER REFERENCES

(110) Advanced Micro Devices, "A Microprogrammed 16 Bit Computer", 1976, 901 Thompson Place, Sunnyval, Cal.

(111) Tilak Agerwala, "Microprogramming Optimization: A Survey", IEEE Transactions on Computers, vol c-25, no 10, October, 1976, pp 962-973.

(112) P . Azemo, M. Diaz, and J. E. Doucet, "Multilevel Descriptions Using Petri-Nets", Proceedings of 1975 International Symposium on Computer Hardware Description Languages and Their Applications", pp 188-190.

(113) J.W. Bowra, and H.C. Torng, "The Modeling and Design of Multiple Unit Processors", IEEE Transactions on Computers, vol C-25 no. 3, March, 1976, pp 210-221.

(114) Melvin Breuer, "Digital System Design Automation: Languages, Simulation and Data Base", Computer Science Press, 1975.

(115) H. Brineen, "Bit-slice Design Approaches", Computer Design, April 1980, pp 184-192.

(116) S-J Chang, "Some Applications of Hardware Description Languages in a Real-time Digital System Deelopment", Proceedings of 1975 International Symposium on Computer Hardware Description Languages and Their Applications, pp 181-182.

(117) L. A. Cox Jr., "Performance Prediction from a Computer Hardware Description", Proceedings of the 4'th International Symposium on Computer Hardware Description Languages, 1979, pp 116-123.

(118) Subrata Dasgupta, "Some Aspects of High-level Microprogramming", Computing Surveys, Volume 12, no. 3, September 1980, pp 296-323.

(119) "Design Automation Courses Offered at The City College of the City University of New York", ACM-SIGDA, vol 4 number 4 December, 1974, p 48.

(120) C. J. Evangelisti, G. Goertzel, and H. Ofek, "Using The Dataflow Analyzer on LCD Descriptions of Machines to Generate Control", Proceedings of the 4th International Symposium on Computer Hardware Description Languages, 1979, pp 109-115.

(121) E. R. Fiala, "The Maxc Systems", IEEE Computer, May, 1978 pp 57-67.

(122) M. J. Flynne and M. A. MacLaren, "Microprocessing Revisited", Proceeding of ACM National Meeting, 1967, pp 457-464.

(123) R. E. Frankel and S. W. Smaliar, "Beyond Register Transfer: An Algebraic Approach for Architectural Description", Proceedings of 4th International Symposium on Computer Hardware Description Languages, 1979, pp 1-5.

(124) F. J. Hill and B. Huey, "A Design Language and Approach to Test Sequence Generation", Computer, June 1977, pp 28-33.

(125) B. M. Huey and F. J. Hill, "Fault Test Generation Using a Design Language", Proceedings of the 1975 International Symposium on CHDL's and Their Applications, pp 91-95.

(126) Intel Corp., "Intel Series 3000 Reference Manual", 3065 Bowers Ave, Santa Clara, Ca.

(127) H. F. Jorden and B. J. Smith, "The Assignment statement in Hardware Description Languages", Computer, 1977, pp 43-49.

(128) M. J. Knudsen, "PMSL, An Interactive Language for System-level Description and Analysis of Computer Structures", PhD Thesis, Department of Computer Science, Carnegie-Mellon, Pittsburgh, Pa, April, 1973.

(129) G. R. Lloyd and A. Van Dam, "Design Considerations for Microprogramming Languages", National Computer Conference, 1974, 537-543.

(130) G. J. Lipovski, "Naming Convention for Modular Design Language", First Workshop on Computer Hardware Description Languages, Rutgers University, New Brunswick, N.J. Sept 6-7, 1973.

(131) W. G. Magnuson, "CHDL Workshop Held in Germany", Computer, December, 1974, p 67.

(132) E. G. Mallach, "Simulating an Aerospace Multiprocessor", Proceeedings of the Ninth Annual Simulation Symposium, 1976, pp 241-252.

(133) F. Marcoz and O. Tedone, "Hardware Description Languages in Italy", Computer, December, 1974, pp 60-61.

(134) T. M. McWilliams, S. H. Fuller and W. H. Sherwood, "Using LSI Processor Bit-slices to Build a PDP-11 -- A Case Study in Microcomputer Design", Proceeding of The National Computer Conference, 1977, pp 243-253.

(135) Jean Mermet, "Defintion du Language CASSANDRE", Theses Docteur-ingenieur, Grenoble, March 1970.

(136) Jean Mermet, "Hardware Description Languages in France", Computer, December 1974, pp 55-56.

(137) John R. Mick, "Microprocessor Handbook", Advanced Micro Devices, 901 Thompson Place, Sunnyvale, Cal.

(138) David A. Patterson, "An Approach to Firmware Engineering", Proceedings of 1978 National Computer Conference, AFIPS vol 47, pp 643-647.

(139) N. C. Pearson, "Programmable Controllers", Design News, April 17, 1978, pp 76-81.

(140) F. J. Ramig, "The Implementation of the Computer Hardware Description Language CAP and its Application Languages and Their Applications, pp 34-37. tions", Proceedings of the 4th International Sympos ium on Computer Hardware Description Languages", 1979, pp 138-144

(141) T. G. Rauscher, "A Unified Approach to Microcomputer Software Development", IEEE Computer June 1978, p 44.

(142) B. D. Shriver, "Firmware, The Lessons of Software Engineering", IEEE Computer, May 1978, pp 19-20.

(143) W. A. Skelton, "A Study of the Problems in Simulating Intel 3000 Microcode", a paper written for a computer Science course at UTA, 1975.

(144) Texas Instruments, The Bipolar Microcomputer Data Book for Design Engineers, December 1977, TI, PO 5012 Dallas, Texas, 75222.

(145) M. Tokoro, T. Watanabe, K. Kawakami, J. Sugano, and K. Noda, "PM/II -- Multiprocessor Oriented Byte Sliced LSI Processor Modules", Proceedings of the 1977 National Computer Conference, pp 217-225.

(146) S. G. Tucker, "Microprogrammed Control for System/ 360", IBM System Journal, vol. 6, no. 4, 1967, pp 222-240.

(147) W. M. vanCleemput, "A Bibliography of Theses on Digital Design Automation", Proceedings of 1975 International Symposium on Computer Hardware Descrip-

(148) H. Watanambi and K. Fujino, "Hardware Description Languages in Japan", Computer December, 1974, p 62.