A POST-PROCESSING SYSTEM
FOR AN AHPL SIMULATOR

by

PUNDI SREENIVASAN MADHAVAN

B.S., Madras Institute of Technology, 1976
M.S., University of Madras, 1980

A MASTER'S THESIS

submitted in partial fulfillment

of the requirements for the degree

MASTER OF SCIENCE

DEPARTMENT OF ELECTRICAL ENGINEERING

KANSAS STATE UNIVERSITY

Manhattan, Kansas

1982

Approved by:

[illegible]
Major Professor

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

Chapter I

INTRODUCTION

A. Motivation for HDL's (Hardware Description Languages)

A digital computer is an extremely complex device and, likewise, a complete description of such a device is also complicated [1]. It takes a lot of words to describe even a few operations of a very simple computer. Certainly something better than the English language is required in order to describe the design and functioning of a computer efficiently. Several languages such as AHPL (A Hardware Programming Language), CDL (Computer Design Language), DDL (Digital Design Language), ISP (Instruction-Set Processor) and PMS (Processor Memory Switch) have been proposed to describe the computer structures and hardware algorithms.

There are different levels of digital system design and they range from circuit design, packaging design, logic design, structure design, behavior design to system design. Each level has a specific purpose and requires a description. Computer hardware description languages can provide a common denominator for this broad spectrum.

Different types of hardware description languages reflect different levels of abstraction of computer hardware. Their use offers the following advantages [10].

1. They serve as a means of communication among computer engineers.
2. They permit a precise and concise description.
3. They provide a convenient documentation.
4. They are amenable to simulation on a computer.
5. They aid greatly in an integrated and total design automation system - ranging from the

> design of the computer structure to the wiring list or even to the patterns for making large-scale integrated semiconductor devices.

In just the same way as software design engineers use high level languages for expressing algorithms, hardware design engineers use hardware description languages to describe the design of digital systems [7]. Other important applications of CHDL's (computer hardware description languages) include teaching logic design, inputting to an automatic design system, generating user manuals, etc.

B. The Different Levels of Abstraction

A digital system can be described at different levels of abstraction [6] such as (a) the circuit level, (b) the switching circuit level, (c) the register transfer level, (d) the program level, (e) the PMS (Processor Memory Switch) and (f) the ISP (Instruction-Set Processor) level. Also, the complexity of computer systems is better understood when such systems are organized into different levels. Each of the above-mentioned levels of abstraction is briefly discussed below.

1. The Circuit Level

The components at the circuit level of abstraction are resistors, inductors, capacitors, diodes, transistors, voltage sources and other non-linear devices. The behavior of the system is measured in terms of current, voltage, resistance, magnectic flux and so on. The description of the system is given by algebraic equations and also in terms of a graphical picture. Various theorems like Ohm's law and Kirchoff's law are used to develop the equations describing the system.

2. The Switching Circuit Level

At the switching circuit level, the system is viewed as consisting of logic gates and flip-flops. The behavior of the system is described by discrete variables which assume only one of two values, 0 or 1. (True or false, high or low). The components involved in the system perform logic functions such as AND, OR, NOR, NOT and EX-OR. The behavior of the system can be described by a set of Boolean equations. The important point to note here is that the designer does not concern himself with the behavior of the basic components such as resistors, capacitors, inductors, diodes and transistors. Also the description of the circuits, which constitutes the system, is given in terms of a set of interconnected gates and flip-flops. The description at the switching circuit level suppresses the lower level details such as resistors, capacitors, inductors, diodes and transistors.

3. The Register Transfer Level (RT Level)

The description of the system at the RT level consists of register transfers under various conditions. The system can be visualized as consisting of two parts: (1) the data section, and (2) the control section. The data section is composed of registers and data paths. The control section provides the time dependent signals which are responsible for the various register transfers in the data section. Combinational logic networks are described differently. Usually a combinational network performing a specific function is described in terms of the specific combinational logic function it performs. At the register transfer level, the designer does not concern himself with the interconnections of logic gates. The designer at this level would be more interested in the operation of the whole system as such with reference to the control signal sequence and the various data transfers that take place among the registers. Various languages at the register transfer level have been proposed, one example of which is AHPL (A Hardware Programming Language) [1].

4. The Program Level

The program level is not just a unique level of description for digital systems but is uniquely associated with computers. The program level deals with the specific algorithms implemented in the hardware for the solution of a problem, and there is a central component that interprets a programming language. Considering the examples of digital controls and instrumentation, it is easy to visualize that in such systems there is no interpreting device and, hence, no program level of description but only a logic level of description.

The program level of description consists of the following components: (1) a set of memories and (2) a set of operations. These memories hold data structures and the operations, accept the various data structures as inputs, and create more data structures which also reside in the memory. The unique feature here is the representation of the various operations on various data structures in a specific sequence. Usually the operations continue in sequence unless a branch instruction is encountered.

Computer structures described at the logic level are parallel structures with all the components being active at the same time. At the program level, these are described as serial devices executing instructions in sequence.

At the program level, it is possible to name certain quantities, use abbreviations and make decisions. At this point, it is worth mentioning that the program level is linguistic in nature whereas the logic level is not.

5. The PMS Level (Processor Memory Switch Level)

The PMS level describes the system in terms of processing units, memory elements, peripheral devices and switching systems. The computer system is viewed in terms of its most aggregate behavior. The system is viewed as consisting of central processors, core memories, tapes, disks, input/output processors, communication lines, printers, tape controllers, graphics terminals, etc. The system is viewed as a processing medium and the information is measured in bits. Thus, the components have capacities and flow rates as their operating characteristics. All details of the program are suppressed.

The PMS description is one of the top levels of description of a computer system. Besides being a description of the overall structure of a computer system, the PMS description is also the description of the amounts of information held in various components, the flow of information among various components and the description of the control unit that accomplishes these flows.

6. The ISP Level (Instruction-Set Processor Level)

The ISP notation is meant to precisely describe the program level of a computer system. The behavior of a processor depends on the nature and sequence of its operations. This sequence is determined by means of a program in the main memory and a set of interpretation rules within the CPU. Thus, if the nature of operation and the interpretation rules are specified, the actual behavior of the processor is a function of the initial conditions and the particular program under consideration.

The ISP level gives the details of the machine's instruction cycles such as fetching the instruction, decoding, program counter incrementing, calculating the operand address, operand fetching and execution of the instruction.

The ISP description consists of a description of the entire instruction set of the processor. The details must include the format of instructions, various registers referenced by the instruction, and the interpreting rules of the instruction.

The above six levels of abstraction provide different ways to describe a digital system. In this thesis, the main topic of discussion is the register transfer level of abstraction and the language used is AHPL (A Hardware Programming Language). Also, the emphasis is on the AHPL simulator [8].

Chapter II

A HARDWARE PROGRAMMING LANGUAGE (AHPL)

A. AHPL Language and its Features

AHPL is a register transfer level language [1]. The use of AHPL as a hardware description language is based on the fact that many digital systems can be segmented into two distinct sections: (1) the control section and (2) the data section. The control section is responsible for causing register transfers within the data section. This is effected by sending control signals at appropriate times to the data section. In some cases, the fixed control sequence may be influenced by branching information that is received from the data section. Usually, in most applications, a single control signal is responsible for the transfer of the contents of a register (after a logical computation) into another register. The main topic of discussion here regards the various features of AHPL as a hardware design language. Other aspects of AHPL such as conventions and notations will not be discussed here since such information is readily available elsewhere [1]. Generally, all the bits in a register are treated uniformly and one can conveniently express the logical operations among registers in vector notations. The following AHPL example will illustrate the points noted above:

INPUTS:f,g

MEMORY:A[4], B[4], C[4], D[4]

1. $A \leftarrow ((B \wedge C) * f) \vee (D * g)$
2. $\rightarrow (g, \overline{g}) / (3, 10)$
3. $B \leftarrow D_{3,0:2}$
4. $\rightarrow (10)$

In the above example, statement #1 represents a register transfer and so does statement #3. In statement #1, if the variable f is 1, the logical AND of B and C registers is transferred to register A or if the variable g is 1, the contents of register D is transferred to register A. Variables f and g are not allowed to be 1 simultaneously. Also if f and g are 0 at the same time, the contents of register A do not change -- in other words, no transfer takes place.

In statement #2, if the variable g is 1, a branch occurs and the next statement to be executed is statement #3. If g is 0 ($\bar{g}$ is 1) the control proceeds to statement #10 in the sequence. This is a typical example of a conditional branch in an AHPL sequence. Statement #4 signifies an unconditional branch to statement #10.

In effect, the AHPL statements give a complete description of the digital system or a sequential network. It is possible to construct a detailed logic diagram of the hardware realization of both the control and data units, given the AHPL description. To illustrate the construction of a logic diagram from the AHPL description, the following example of an AHPL sequence is considered:

1. $Z \leftarrow X \vee A$

   $\rightarrow (a, \bar{a} \wedge b, \bar{a} \wedge \bar{b})/(2,3,4)$

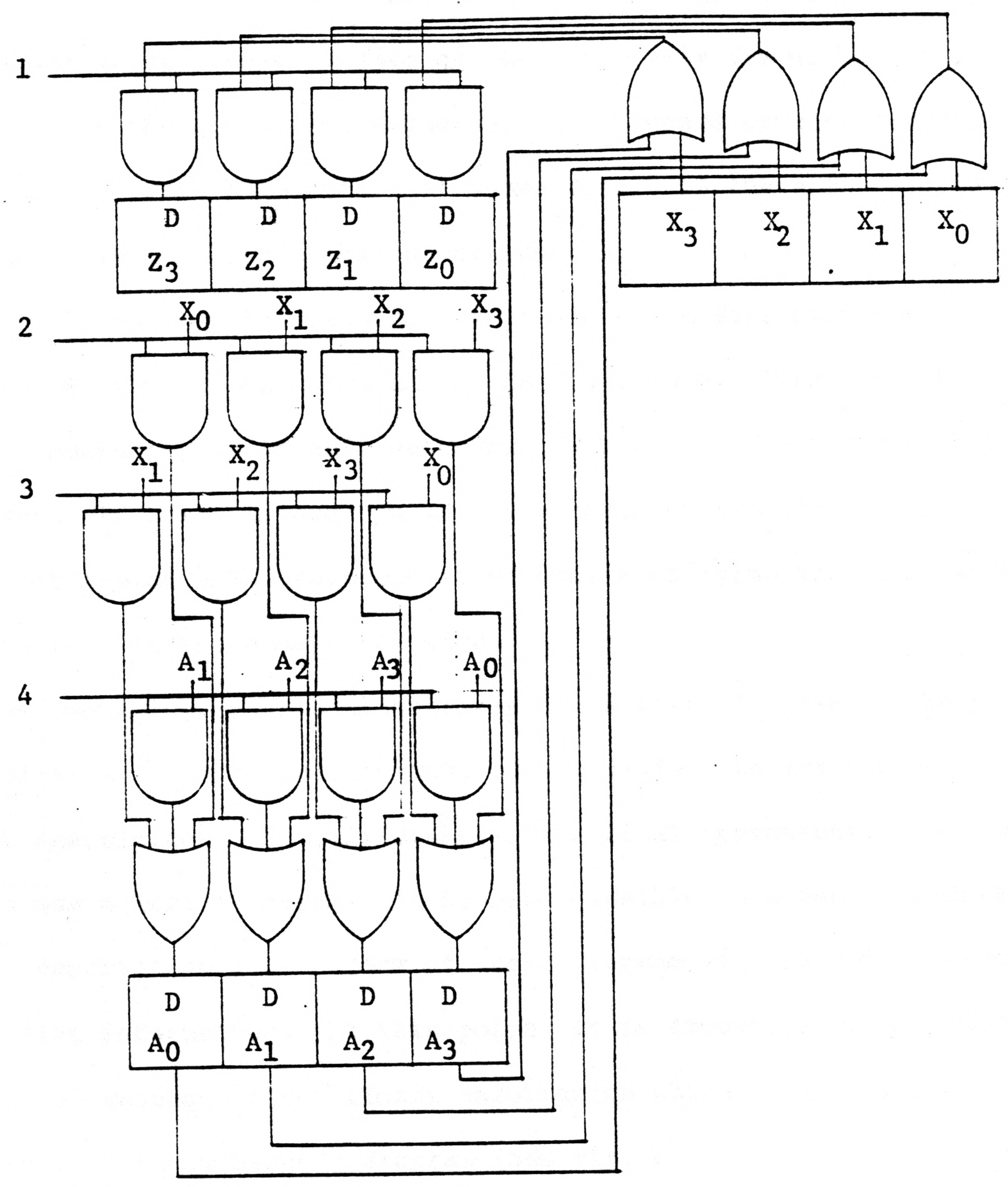
2. $A \leftarrow X$

   $\rightarrow (1)$

3. $A \leftarrow X_{1:3}, X_{0}$

4. $A \leftarrow A_{1:3}, A_{0}$

   $\rightarrow (1)$



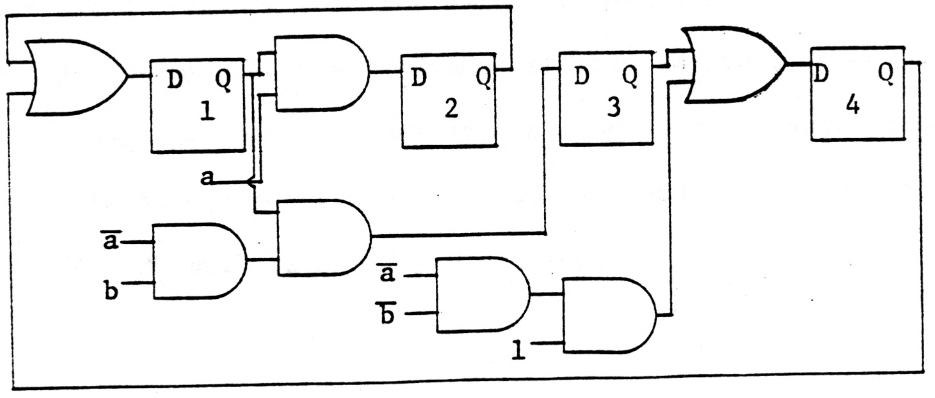


Figure 1. Control and Data Unit Realization of an AHPL Sequence

In the above example, construction of the logic diagram (see fig. 1) was quite simple because of the relatively few number of statements in the given AHPL sequence. The process of constructing such logic diagrams becomes almost unmanageable if the AHPL description contained a larger number of statements.

All the information required to completely define both the control and data section is available in the AHPL sequence. Every register transfer operation in an AHPL sequence takes place in one clock time. The branch (if there is one) following a register transfer operation in any particular AHPL statement is accomplished (simultaneously with the transfer) in the same clock time.

The preceding example illustrates how a logic diagram could be constructed from a specification of the design problem through an AHPL description. In this example, the final representation of the circuit was a logic diagram. It is also possible to directly translate an AHPL description into a form of logic diagram with gate details and wiring list information. At this point, it is important to note that the control sequence gives timing information which is not easily determined in the schematic diagram (see fig. 1).

Thusfar, the AHPL language was discussed in conjunction with elementary logic diagrams from the hardware design point of view and an example was given. In the remainder of this section, other aspects of the AHPL language such as combinational logic units, asynchronous systems and timing considerations will be reviewed. Finally, an advanced version of AHPL will be briefly discussed.

1. Combinational Logic Units in AHPL

The use of combinational logic functions in AHPL is similar to a function call in a programming language. The following AHPL transfer statement is considered as an example to introduce the idea of combinational logic functions:

$$\begin{aligned} C \leftarrow\ & (A_0 \wedge B_0) \oplus ((A_1 \vee B_1) \wedge (A_2 \vee B_2) \wedge (A_3 \vee B_3)), \\ & (A_1 + B_1) \oplus ((A_2 \oplus B_2) \wedge (A_3 \oplus B_3)), \\ & (A_2 + B_2) \oplus (A_2 \oplus B_3)\ ,\ (A_3 \oplus B_3) \end{aligned}$$

The expression on the right is quite complex. It is probably manageable if such an expression occurred only once in an AHPL description. If it were to occur a number of times, one would prefer not to write it each time. In such instances, the complex logic unit is defined in the beginning of the AHPL sequence and the unit is given a particular name. This name can then be substituted whenever a need arises to write the complex combinational function. In the above example, the expression on the right could be named LOGIC (A;B) and now the complex combinational transfer statement can be rewritten in the following way:

$$C \leftarrow \text{LOGIC }(A;B)$$

This convention can be viewed as similar to a function call in conventional programming languages. Such a notation is restricted in AHPL only to combinational logic networks.

A combinational logic unit may be referenced several times with the same arguments in an AHPL description. In such an instance, only one logic network is built with multiples of connections to its output. The notation for a given combinational function may also appear several times with different arguments. In such a case, one network can be constructed and shared instead of independently constructing the networks for each set of arguments. One method of sharing a complex combinational logic network is to declare buses for its arguments.

Quite often, a large digital system consists of several interconnected modules. It is important to consider such multimodular systems and the relation of each module to others within the system. It is quite possible that the data unit of one module might be triggered by control signals from more than one control section. It is quite important to be able to visualize the transfer and connection statements of one module and isolate them as belonging to one specific module. In these areas, the designer is often posed with the problem of defining the boundaries of the systems, particularly in situations where the systems are connected through communication channels and complicated networks.

Typically, the AHPL description of any particular module in a digital system consists of the name of the module, declarations, control sequence and an end statement. A declaration includes the following:

MEMORY (flip-flops, registers, etc.)

INPUTS

OUTPUTS

ONESHOTS

LABELS (used to rename portions of large registers)

BUSES

COMBUS (communications bus -- contents on this bus
may be defined by more than one module)

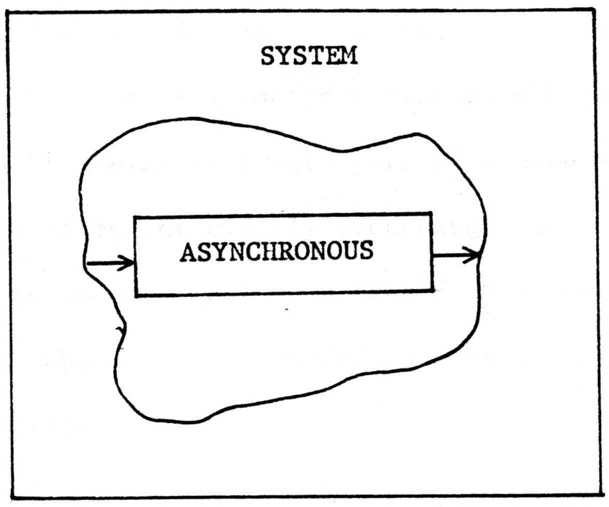
2. Asynchronous Systems

The AHPL descriptions studied thusfar have implied synchronous operations. All register contents were assumed to change values only on a common edge of the same clock pulse. This may not be the case for large systems with subsystems. Some logic levels may change asynchronously with the system clock. The physical distance between modules may be so large that it makes this synchronism impossible. This problem is particularly true in the case of high frequency clocks where the clock time is comparable to the propagation delay of the signal.

An asynchronous subsystem can be viewed as shown in figure 2. The design can be expressed in AHPL by identifying the asynchronous subsystem as a separate module (see fig. 2). The asynchronous signals are synchronized with the system clock before they are used in a given module. An asynchronous signal after synchronization is treated just as any other signal within the module. Thus a collection of modules, operating asynchronously with respect to one another, can also be described in AHPL.

3. Timing Considerations

It is possible to go through an AHPL sequence and find out if any step could be modified that would speed up the process. Interrelated branches can make this process more difficult. In cases where possible, the approach would be to eliminate the delay



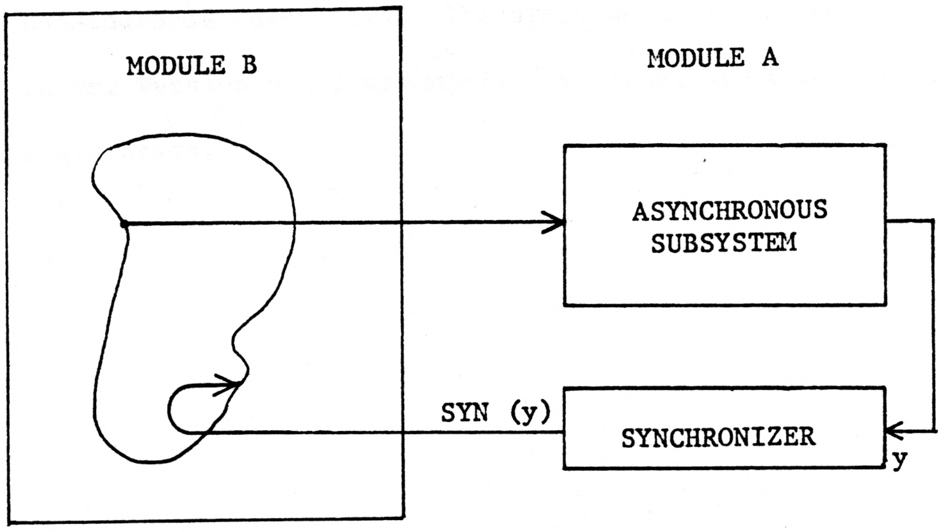


Figure 2. Asynchronous Systems in AHPL

associated with an individual step in the control sequence. If such a step is identified, it is specified in the corresponding AHPL step by a NO DELAY comment. The main idea is that the one clock time delay associated with an AHPL step can be eliminated if the transfer or branch part of a step is independent of the result of an immediately preceding step. The important point here is to note that some transfers do not require an additional clock time. These transfers can be accomplished in the previous clock time.

4. Advanced version of AHPL [5]

When the complexity of a system increases, it is useful to have a facility to invoke a set of register transfer operations in just the same way as combinational logic functions were invoked before. The advanced version of AHPL provides AHPL descriptions within descriptions. Also, the recent version of AHPL allows representation of sets of duplicate descriptions of considerable complexity. The structures permitted in this latest version are procedural, functional and combinational logic units.

Chapter III

AHPL SIMULATOR AND ITS FEATURES

In this section, the AHPL simulator [8] will be briefly reviewed. The main objective is to highlight the features of the AHPL simulator and its use as a design tool for the hardware designer. The actual operation of the simulator and various details regarding the syntax of AHPL to be followed for use in the simulator are available [8]. The AHPL simulator does not support descriptions of asynchronous systems. At the conclusion of this section, the shortcomings of the AHPL simulator are discussed and solutions to overcome these shortcomings are proposed.

A. AHPL Simulator As a Hardware Design Tool

In the last section, the use of AHPL as a design language and its various features were discussed. At this point, it is necessary to think of a simulator which could simulate the AHPL description and perform the register transfers, branches, etc. of a digital system, and display the contents of various registers, control signals and buses on a clock period by clock period basis.

The HPSIM (Hardware Program Simulator) and HPCOM (Hardware Program Compiler) are the two software packages designed to simulate and compile the AHPL description of a digital system. HPSIM is a function level simulator which simulates the AHPL description by performing the register transfers, branches, etc., on a clock time by clock time basis. HPCOM, on the other hand, is a program capable of listing the wire connections which specify the interconnections of the available integrated circuit parts. The AHPL as accepted by HPSIM and HPCOM is slightly different from the original version [1].

HPSIM was implemented on the CYBER 175 and DEC 10 computers, and it is written in Fortran. HPSIM 2 was implemented on the IBM 370 computer. The version of HPSIM which is discussed in this thesis is HPSIM 2. HPSIM 2 is an enhanced version of HPSIM and is more powerful and faster than HPSIM. To help the user debug his AHPL description, HPSIM 2 provides runtime error messages.

The input to HPSIM 2 is an AHPL description of a digital system followed by a communication section. In the communication section, the user assigns values to the external lines and specifies the names of registers and control signals that he needs to observe as outputs. In the communication section, it is possible to specify one of several options. Depending on the option specified, the output is available in different forms such as binary, octal or hexadecimal. If no option is specified, the output is in binary form by default. The option information, if included, should be the first specification in the communication section. The user can also specify the number of clock times for which the execution should continue. The simulator continues for as many clock times as specified unless a DEADEND is encountered. It is also possible to assign values to the external lines and this is similar to assigning data in programming languages. In terms of hardware, this could be written to imitate the I/O (Input/Output) behavior of a system so that it is not necessary to model the internal operation.

Before the simulation begins, the user can initialize the registers and memories. The user can specify the lines and registers that he needs to observe as outputs at each clock time. The corresponding module number to which each output belongs can also be specified. There is also a facility to indicate the registers and memories whose contents need to be observed at the end of the simulation and such information is specified in the DUMP section. The simulator may stay in wait loops for response from an external system. The outputs corresponding to such statements of wait loops can be suppressed by specifying them in the SUPPRESS section.

With the present simulator, seven standard functions are available. They are as follows:

ADD (Add function)

INC (Increment function)

BUSFN (Memory read function)

DCD (Decode function)

DEC (Decrement function)

COMPARE (Compare function)

ASSOC (Associative function)

If any of the above seven standard functions are used in the AHPL sequence, they must be declared as CLU (combinational logic units) units at the beginning of the AHPL description. User-defined CLU units are not allowed in the simulator at the present time.

More than one of the same kind of a standard function can be used in a module and, in such a case, all function names of that type must be declared with distinct alphanumeric suffixes. For example, if two separate adders are required in a circuit, the declarations would be as follows:

CLUNITS: $ADD_1[n_1](W,X)$; $ADD_2[n_2](Y;Z)$, where n_1 and n_2 are the number of bits of Adder 1 and Adder 2 respectively. CLUNITS may have CLUNITS as arguments.

Unclocked transfers which are denoted as $\leftarrow$ 0 are not distinguished from clocked transfers by HPSIM. The clocked transfer notation must be used in an AHPL description to be simulated using HPSIM. The simulation will be accurate subject to the condition that the output of a memory element is not used during the same clock period in which it is the target of an unclocked transfer.

Comments can appear anywhere in the AHPL sequence or in the communication section. Comments should be enclosed within double quotes and should not exceed a card limit.

B. An Example Simulation Using HPSIM 2

An example simulation of a small AHPL sequence is briefly reviewed here. The following AHPL sequence is considered in this example. The corresponding output file appears in the Appendix.

AHPL INPUT FILE

AHPLMODULE:CONTROLLER

EXINPUTS:Z;START.

MEMORY:A[4];B[4];C[4];D[4].

1 $\rightarrow (\overline{START})/(1)$.

2 B $\leftarrow$ A.

3 C $\leftarrow$ B; $\rightarrow$ (Z)/(5)

4 A $\leftarrow$ $(\overline{C})$; $\rightarrow$ (2).

5 D $\leftarrow$ C; $\rightarrow$ (2).

```
ENDSEQUENCE.
CONTROLRESET(1).
END.
```

<u>COMMUNICATION SECTION</u>

```
OPTION 7.
CLOCKLIMIT 10.
EXLINES START=0,1;Z=0#4,1.
OUTPUTS START;Z;A;B;C;D.
INITIALIZE A ← 3.
DUMP ALL.
```

After simulating the above AHPL sequence, the outputs of the registers A, B, C, D and the control signals START and Z are listed in the output file on a clock cycle by clock cycle basis (see the Appendix).

At the first clock time, all register and control variable values are zero except register A which is initialized to the value 3. Also, the variables START and Z are 0 at this time because of initialization. The module and step information indicate that module #1 is active and the next statement to be executed is statement #1. At clock time 2, the variable START becomes 1 as specified in the communication section of the file, and all other register contents do not change at this time. AHPL statement #1 is still not executed since START was 0 before. At the third clock time, START has already become 1 and the decision is made at AHPL step #1 and the wait loop is terminated. At the fourth clock time, statement #2 is executed and, correspondingly, the contents of register A are transferred to register B. At the fifth clock time, the contents of register B are transferred to register C, and this completes execution of AHPL statement #3. At the same time, a branch occurs based on the value of Z, which becomes a 1 at the end of clock time 5. So the next step to be executed is AHPL statement #4. Hence, at clock time 6 the contents of the complement of register C are transferred to register A, and this completes the execution of statement #4. At the same time, there is an unconditional branch to statement #2. At clock time 7, statement #2 is executed causing the transfer of contents of register A into register B. At the eighth clock time, the next executable statement would be statement #3, causing the contents of register B to transfer into register C. At this time, a branch occurs to statement #5 because Z is already 1 and stays high at this time. At clock time 9, the contents of register C are transferred into register D. This completes the execution of statement #5 and an unconditional branch to statement #2 occurs. At clock time 10, once again the contents of register A are transferred into register B, which completes execution of AHPL statement #2. In the above example, statement #'s 2, 3 and 4 are repeatedly executed and constitute an infinite loop.

The clock limit was specified as 10 in the communication section of the file and, therefore, the simulation stopped in 10 clock times. If the clock limit was specified as 1000, the simulation would have continued for 1000 clock times. If it were not for the unconditional branch to statement #2, the program would have probably encountered a DEADEND and would not repeat itself for several cycles. Whenever a program encounters a DEADEND, a relevant message to that effect is output, and the simulator stops even though the required number of clock times is not reached.

The above discussion is related to a single module design (see Appendix). The step information at any clock time indicates the next active AHPL statement in sequence and this is obvious from the computer printout (see Appendix) and the above discussion.

Finally, a DUMP of memories and registers is provided at the end, and this dump displays the contents of the memories and registers at the time when simulation stopped.

C. Problems With the Present Simulator

A close look at the simulator output (see Appendix), reveals that all the outputs specified in the communication section are printed out at every clock time. Though the designer has all the information dumped in the output file, such an output file is difficult to analyze. All such problems are briefly discussed below and at the conclusion of this section, solutions to these problems are proposed.

1. The control variables are listed along with the vectors, and it is difficult to identify certain level changes at certain clock times. This is because such information is presented simply in the form of 1's and 0's and there is no means of visualizing the information in graphical form relative to the system clock. Quite often, a graphical presentation is needed and preferred in most digital systems, because such a presentation

is more meaningful. The level changes with respect to the clock and also with respect to other control signals of any variable are more easily noticeable in a graphical picture. Any special Boolean relation between control variables can be better visualized with a graphical picture. Redundant control flip-flops may be detected and eliminated with the aid of a graphical picture. The convenience of graphical presentation does not exist with the original simulator.

2. With the original version of AHPL simulator, the status of control variables and vectors cannot be visualized in a selected group of clock pulses alone. It is probably of interest for the designer to select a certain number of clock times and study the status of some of these variables relative to the system clock. This process may be helpful to locate some faults (if any).

3. It might be of interest to observe the contents of registers after a certain condition is established in a given register, or after a certain Boolean relation is satisfied among certain control variables, or after a given register attains a critical value (maximum or zero). It may be helpful to observe the contents of registers and control variables after a certain combination of values of a given set of vectors and scalars is established. The present simulator does not support this feature.

4. Another possible area of interest is to know the number of times a register or set of registers reach a certain value and to

study the state of the machine or the system in the vicinity of those values that the register(s) attain. It can be difficult to scan and obtain such information with the existing simulator.

An immediate solution to the above problems (1 through 4), is to output only a few variables from the simulator. It is, in fact, possible to specify fewer number of outputs in the communication section of the input file to the simulator. The output file is probably better displayed for usual scanning because of the few number of outputs. There are two major disadvantages with this kind of a solution: (1) There is no control over the number of clock cycles of information printed out in the output file once a certain number is specified in the communication section. The visual scanning is in no way easier for a lesser number of outputs, and (2) once a specific number of outputs are listed out for a given number of clock times, and if at a later time a different set of vectors or scalars are required to be output, the communication section of the input file has to be changed and the simulator run again. Such a practice of changing the communication file often and running the simulator each time is obviously not desired.

A solution to the above problems (1 through 4) is proposed in this thesis. The solution proposed is to post-process the AHPL simulator output file. The major advantage is that all the required variable names can be specified in the communication section of the input file and the simulator can be run for a maximum number of clock times. The output file now contains all the specified variable values and for a maximum number of clock times. This output file can now be post-processed to have many features at the user's convenience.

D. Advantages of the Post-processor

1. The post-processor offers the convenience of observing the control variables in graphical form relative to the system clock. The list of control variables, the start clock time and the stop clock time can be specified for the post-processor in an interactive environment. The post-processor accepts this data as input and plots the control variables relative to the system clock. Due to the limitation in the size of the screen width, the states of control variables are not plotted for more than 25 clock times at a time.

2. It is possible to selectively display the number of times and also the particular clock times at which a given combination occurs. Once such information is available, it is possible to observe the state of the machine or the system in the vicinity of the given combination. It may be of interest to visualize the state of the machine a few clock times before or after the particular clock time at which a given combination occurs.

3. The post-processor can selectively display a subset of vectors, scalars or vectors and scalars within given bounds of

clock times. The user specifies the start and stop clock times and the list of scalars and/or vectors.

4. Step and module information is available at each clock time. This facility is built in the post-processor as an option. The module information specifies the active module and the step information specifies the next active AHPL statement.

The interested user can repeatedly use the above (1 through 4) features (some or all of them) depending upon his needs. The actual details regarding the usage of the post-processor are not discussed in this thesis. A User's Manual has been prepared [9]. This manual contains all the information necessary for the use and maintenance of the post-processor.

E. Examples to Illustrate the Use of the Post-processor

A simple example to illustrate the use of the post-processor is given below. The simulated output file is also attached (see the Appendix).

The following AHPL input file is considered in this example:

```
INPUT FILE

AHPLMODULE:CONTROLLER

MEMORY:A[4];B[4];C[4];D[4];Z[1];E[1];F[1].

CLUNITS:INC[4](A).

1      B  ←  A.

2      C  ←  B.

3      D  ←  C.

4      A  ←  INC (A)
```

```
5       Z  ←  &/(A).

6       E  ←  (Z̄)

7       F  ←  A[0] @ B[3]

8       →  (Z)/(10).

9       →  (1).

10      A  ←  (Ā)

        ENDSEQUENCE

        CONTROLRESET(1).

        END.
```

<u>COMMUNICATION SECTION</u>

```
OPTION 7.

CLOCKLIMIT 250.

OUTPUTS A;B;C;D;E;F;Z.

INITIALIZE Z  ← '1; E  ← '1; F  ← '1; A  ← '2;
B  ← '3; C ← '4; D  ← '5.

DUMP ALL.
```

The AHPL simulator is run with the above AHPL input file and now the output file is available, ready for post-processing. The following examples are considered to illustrate the uses of the post-processor:

<u>Example 1</u>

```
A = 1111

B = 0000

C = 0000

D = 0000

Z = 1
```

```
** NOW THE SELECTION OF VECTORS BEGINS **
A             1

ARE YOU INTERESTED IN THIS VECTOR (Y/N)  ?Y
THE NUMBER OF BITS OF THIS VECTOR=        4
ENTER NOW THE ABOVE # OF BITS( 1'S & 0'S)  AND HIT RETURN
1111
B         2

ARE YOU INTERESTED IN THIS VECTOR (Y/N)  ?Y
THE NUMBER OF BITS OF THIS VECTOR=        4
ENTER NOW THE ABOVE # OF BITS( 1'S & 0'S)  AND HIT RETURN
0000
C        3

ARE YOU INTERESTED IN THIS VECTOR (Y/N)  ?Y
THE NUMBER OF BITS OF THIS VECTOR=        4
ENTER NOW THE ABOVE # OF BITS( 1'S & 0'S)  AND HIT RETURN
0000
D       4

ARE YOU INTERESTED IN THIS VECTOR (Y/N)  ?Y
THE NUMBER OF BITS OF THIS VECTOR=        4
ENTER NOW THE ABOVE # OF BITS( 1'S & 0'S)  AND HIT RETURN
0000
** NOW THE SELECTION OF SCALARS BEGINS **
E       5

ARE YOU INTERESTED IN THIS SCALAR (Y/N)  ?N
F       6

ARE YOU INTERESTED IN THIS SCALAR (Y/N)  ?N
Z       7

ARE YOU INTERESTED IN THIS SCALAR (Y/N)  ?Y
ENTER NOW ITS BIT 1 OR 0 &HIT RETURN
1
```

Figure 3. Searching for a Given Combination of Values.

```
THE COMBINATION NEVER OCCURED

DO YOU WANT A GRAPHICAL DISPLAY OF SCALARS (Y/N) ?
```

Figure 3 (Continued)

It is required to observe the state of the system in the vicinity of the above combinations. The above values of the variables A, B, C, D and E are input to the post-processor and the post-processor returns the following message (see fig. 3):

"THE GIVEN COMBINATION NEVER OCCURRED".

An examination of the output file (see Appendix) reveals that the above given combination never occurs. Since the given combination never occurs, there is no question of observing the state of the machine in the vicinity of the given combination.

Example 2

It is required to observe the state of the machine in the vicinity where the register A attained the value 1111. The value of A=1111 is input to the post-processor.

The system has returned the following message (see fig. 4):

```
"# OF TIMES THE GIVEN COMBINATION OCCURS=5"

"THE GIVEN COMBINATION OCCURS AT THE FOLLOWING
CLOCK TIMES"

113  114  115  116  117
```

The clock times displayed above are decimal values.

Now, a group of clock pulses is selected before or after the above status and the state of the machine can be observed between the bounds of the clock times (see fig. 5). The variables selected are A, B and D. The clock time bounds are 100 and 113.

```
# TIMES THE COMBINATION OCCURS=        5
THE COMBINATION  OCCURS AT THE FOLLOWING CLOCK  TIMES
     113     114      115      116      117

DO YOU WANT A GRAPHICAL DISPLAY OF SCALARS (Y/N)  ?
```

Figure 4. Determining the Number of Times a Given Combination Occurred.

```
# OF CLOCK PULSES IN FILE =       117
YOU CAN NOW GET A TOP-TO-BOTTOM DISPLAY OF A SUBSET OF UARIABLES

DO YOU NEED MODULE & STEP DETAILS (Y/N) ?Y
START CLOCK TIME =?100
STOP CLOCK TIME=?113
** NOW THE SELECTION OF SCALARS BEGINS **
E              5

ARE YOU INTERESTED IN THIS SCALAR (Y/N)  ?N
F              6

ARE YOU INTERESTED IN THIS SCALAR (Y/N)  ?N
Z              7

ARE YOU INTERESTED IN THIS SCALAR (Y/N)  ?N
** NOW THE SELECTION OF UECTORS BEGINS **
A              1

ARE YOU INTERESTED IN THIS UECTOR (Y/N)  ?Y
B              2

ARE YOU INTERESTED IN THIS UECTOR (Y/N)  ?Y
C              3

ARE YOU INTERESTED IN THIS UECTOR (Y/N)  ?N
D              4

ARE YOU INTERESTED IN THIS UECTOR (Y/N)  ?Y
```

Figure 5. Studying the State of the Machine in the Vicinity of a Given Combination.

```
A, B, D, MDLE.STEP

1101 1100 1100 1.2
1101 1101 1100 1.3
1101 1101 1100 1.4
1101 1101 1101 1.5
1110 1101 1101 1.6
1110 1101 1101 1.7
1110 1101 1101 1.8
1110 1101 1101 1.9
1110 1101 1101 1.1
1110 1101 1101 1 2
1110 1110 1101 1 3
1110 1110 1101 1.4
1110 1110 1110 1 5
1111 1110 1110 1 6

DO YOU WANT TO SEARCH FOR ANY COMBINATION (Y/N) ?
```

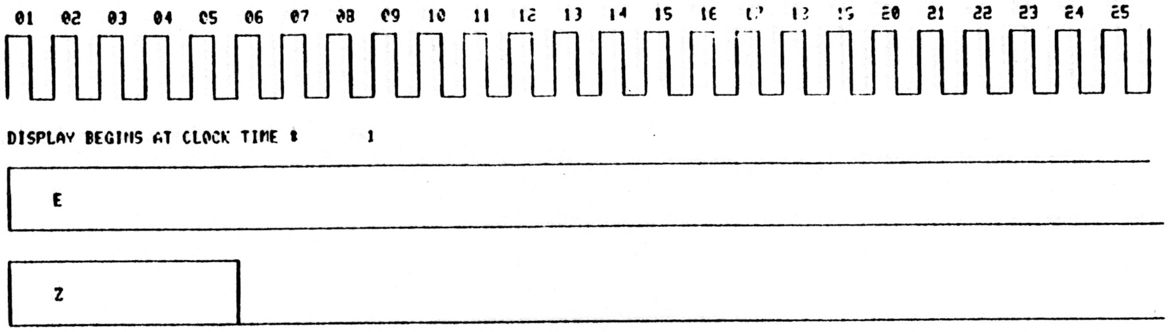
Figure 5 (Continued)

<u>Example 3</u>

It is required to have a graphical presentation of the control variables. The variables E and Z are selected and the bounds are clock times 1 through 55. The corresponding graphical picture is included (see fig. 6). The parameters that are to be input to the post-processor are the variable names E, Z and the clock time bounds, namely, 1 and 15.

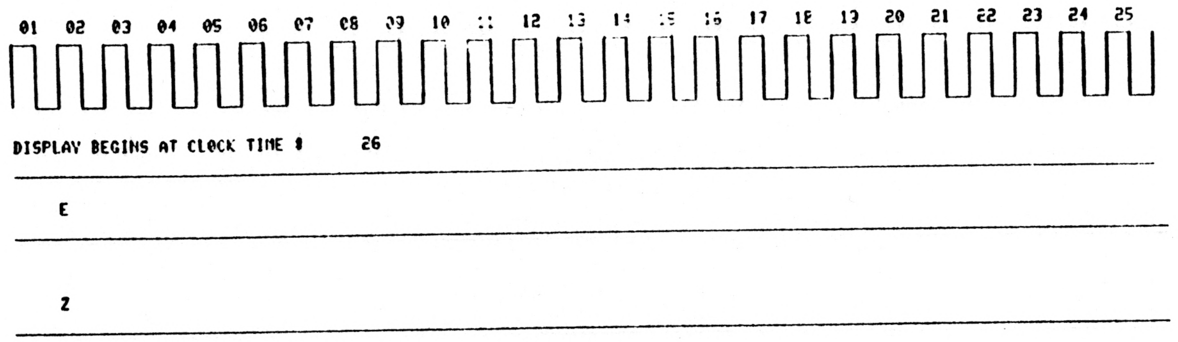
<u>Example 4</u>

It is required to selectively display a subset of the variables between clock times 21 through 40. The variables chosen are A, B, C, D, E and Z. The resulting computer display is included (see fig. 7). In this case, the parameters that are input to the post-processor are the variable names A, B, C, D, E, Z and the bounds of clock times, namely, 21 through 40.



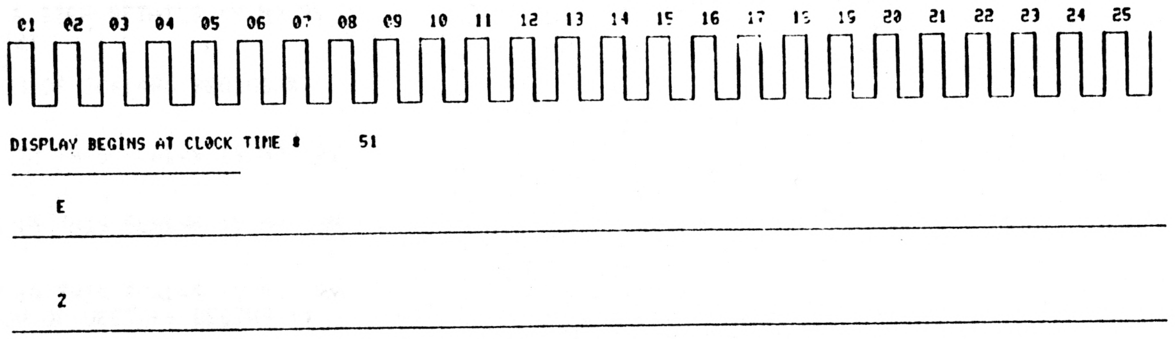
```
DISPLAY BEGINS AT CLOCK TIME :        1

        THERE IS MORE OUTPUT TO BE DISPLAYED
DO YOU WISH TO CONTINUE (Y/N)  ?
```

Figure 6. Graphical Display of a Subset of Scalars.



```
        THERE IS MORE OUTPUT TO BE DISPLAYED
DO YOU WISH TO CONTINUE (Y/N)  ?
```

Figure 6 (Continued)



```
DISPLAY BEGINS AT CLOCK TIME #      51
```

```
DO YOU NEED TO CONTINUE WITH MORE SELECTIONS (Y/N)  ?
```

Figure 6 (Continued)

```
# OF CLOCK PULSES IN FILE =      117
YOU CAN NOW GET A TOP-TO-BOTTOM DISPLAY OF A SUBSET OF VARIABLES

DO YOU NEED MODULE & STEP DETAILS (Y/N) ?Y
START CLOCK TIME =?21
STOP CLOCK TIME=?40
** NOW THE SELECTION OF SCALARS BEGINS **
E            5

ARE YOU INTERESTED IN THIS SCALAR (Y/N)   ?Y
F            6

ARE YOU INTERESTED IN THIS SCALAR (Y/N)   ?N
Z            7

ARE YOU INTERESTED IN THIS SCALAR (Y/N)   ?Y
** NOW THE SELECTION OF VECTORS BEGINS **
A            1

ARE YOU INTERESTED IN THIS VECTOR (Y/N)   ?Y
B            2

ARE YOU INTERESTED IN THIS VECTOR (Y/N)   ?Y
C            3

ARE YOU INTERESTED IN THIS VECTOR (Y/N)   ?Y
D            4

ARE YOU INTERESTED IN THIS VECTOR (Y/N)   ?Y
```

Figure 7. Selective Display of a Subset of Variables.

```
E;Z.A; B; C; D; MDLE.STEP

1 0 0100 0100 0100 0011 1.4
1 0 0100 0100 0100 0100 1.5
1 0 0101 0100 0100 0100 1.6
1 0 0101 0100 0100 0100 1.7
1 0 0101 0100 0100 0100 1.8
1 0 0101 0100 0100 0100 1.9
1 0 0101 0100 0100 0100 1.1
1 0 0101 0100 0100 0100 1.2
1 0 0101 0101 0100 0100 1.3
1 0 0101 0101 0101 0100 1.4
1 0 0101 0101 0101 0101 1.5
1 0 0110 0101 0101 0101 1.6
1 0 0110 0101 0101 0101 1.7
1 0 0110 0101 0101 0101 1.8
1 0 0110 0101 0101 0101 1.9
1 0 0110 0101 0101 0101 1.1
1 0 0110 0101 0101 0101 1.2
1 0 0110 0110 0101 0101 1.3
1 0 0110 0110 0110 0101 1.4
1 0 0110 0110 0110 0110 1.5

DO YOU WANT TO SEARCH FOR ANY COMBINATION (Y/N)  ?
```

Figure 7 (Continued)

Chapter IV

THE ALGORITHMIC DESCRIPTION OF THE POST-PROCESSOR

In this section, the design of the post-processor is reviewed. The overall operation of the post-processor is discussed at the generalized block diagram level. The algorithm is implemented in Fortran and the program [9] is well-documented and structured to aid in the use and maintenance of the system. The post-processor system is implemented on the Nova 6 minicomputer.

A. Description of the Algorithm

The following discussion refers to the flow chart of figure 8. The blocks in the flow chart are labelled a, b, c, etc., and as the discussion proceeds, references are made to the labelled boxes.

Block (a)

The first step obviously is to create the simulator output file. It may be recalled that before the simulation begins, there are two files that should be created. They are: (1) the AHPL description file, and (2) the AHPL communication file. All the various parameters such as the option number, the number of clock times, the set of variables whose values are to be output, etc., are specified in the communication file. It is necessary to specify option #7 (see page 17), since the system is built to work only for option #7. This option is more general, in that the module and step information are available at

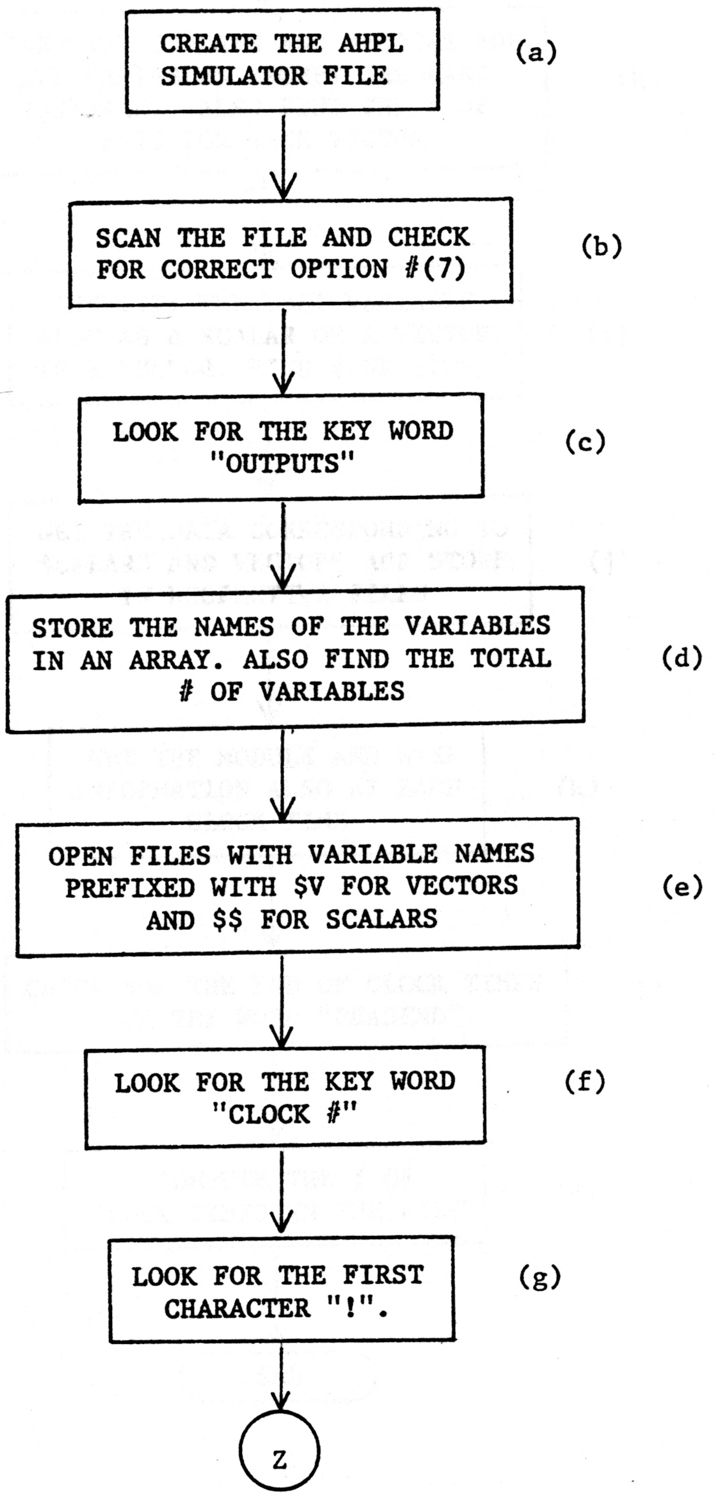

CREATE THE AHPL SIMULATOR FILE
(a)
SCAN THE FILE AND CHECK FOR CORRECT OPTION #(7)
(b)
LOOK FOR THE KEY WORD "OUTPUTS"
(c)
STORE THE NAMES OF THE VARIABLES IN AN ARRAY. ALSO FIND THE TOTAL # OF VARIABLES
(d)
OPEN FILES WITH VARIABLE NAMES PREFIXED WITH $V FOR VECTORS AND $$ FOR SCALARS
(e)
LOOK FOR THE KEY WORD "CLOCK #"
(f)
LOOK FOR THE FIRST CHARACTER "!".
(g)
Z

Figure 8. The Algorithm Implemented in the Design of the Post-processor.

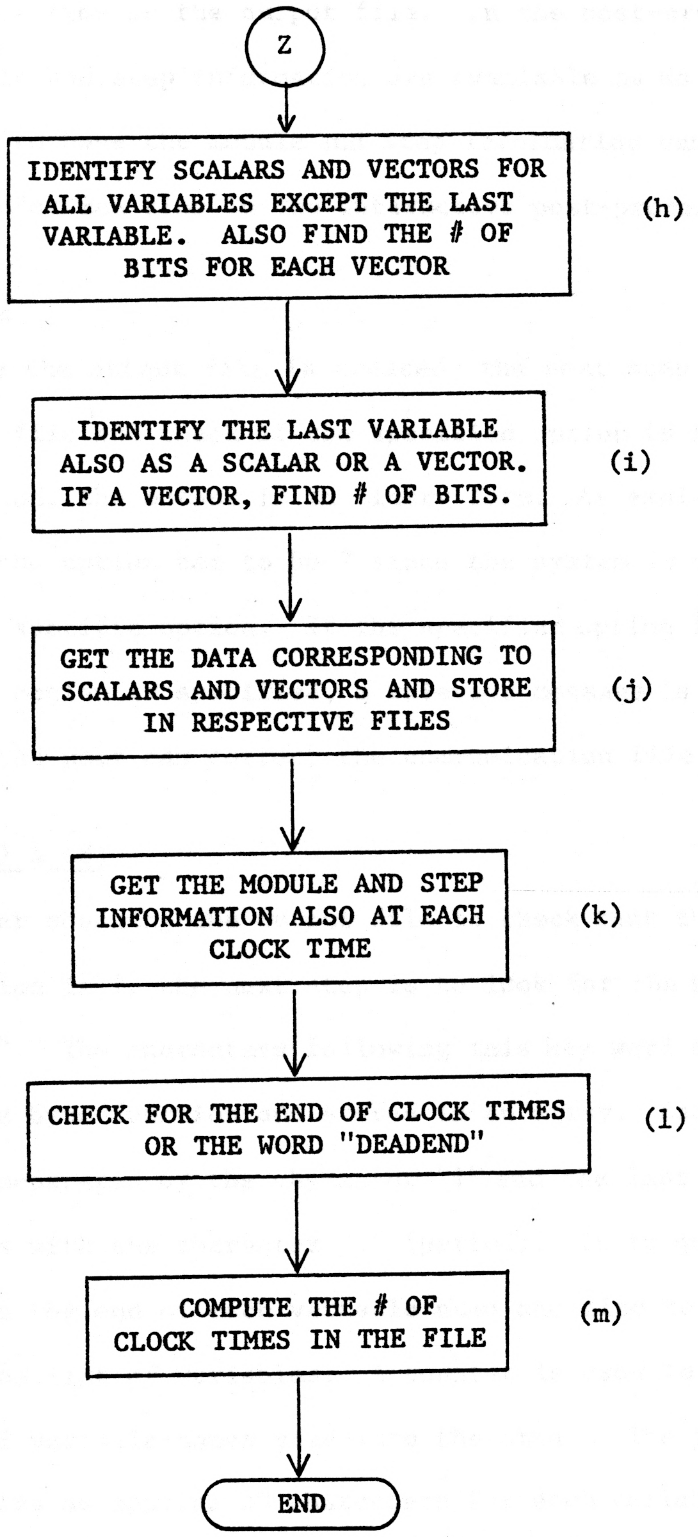

Z
IDENTIFY SCALARS AND VECTORS FOR ALL VARIABLES EXCEPT THE LAST VARIABLE. ALSO FIND THE # OF BITS FOR EACH VECTOR
(h)
IDENTIFY THE LAST VARIABLE ALSO AS A SCALAR OR A VECTOR. IF A VECTOR, FIND # OF BITS.
(i)
GET THE DATA CORRESPONDING TO SCALARS AND VECTORS AND STORE IN RESPECTIVE FILES
(j)
GET THE MODULE AND STEP INFORMATION ALSO AT EACH CLOCK TIME
(k)
CHECK FOR THE END OF CLOCK TIMES OR THE WORD "DEADEND"
(l)
COMPUTE THE # OF CLOCK TIMES IN THE FILE
(m)
END

Figure 8 (Continued)

each clock time in the output file. In the post-processor, the module and step information are available as an option. The need to have the module and step information can be specified at run time in the interactive post-processor.

Block (b)

Once the output file is created, the next step is to scan the file and check if the specified option is #7. With this option, the output is in binary form. As explained before, the option has to be 7 since the system is designed for this specific option. If the specified option is not 7, or if no option is specified, a relevant message is displayed so that the user can correct the communication file.

Block (c) & (d)

After scanning the output file to check that the specified option is 7, the next step is to look for the key word "OUTPUTS". The characters following this key word are read using the byte function and stored in an array. Each variable name is separated by the character "!" and the last variable name ends with the character "." (period). It is quite easy to recognize the end of each variable name and also to mark the end of the list of variables. A counter is used to count the number of variable names read into the array. The program accomodates as many as 20 characters for each variable name. This means that a variable name may not exceed 20 characters in length.

Block (e)

Having identified the variable names, the next step is to store the values of these variables in some form. A memory array could be declared in the program, and all the values of the variables can be stored in the array. There are a few problems in this approach. First of all, the memory array to be declared has to be quite large to accomodate all the variable values and for all the clock times. Neither the number of clock times, nor the number of variables can be predicted. Even if a largest possible memory array is declared, there is always a chance for some simulation to exceed the bounds specified in the declaration of the memory array. Assuming that a memory array can be declared to handle most cases, still the total memory required for the program is quite large.

To solve the problems discussed above, an alternative to the memory declaration scheme is proposed. A file is opened for each variable. The name of the file can be the name of the variable itself. All the files required to store the values of vectors can be prefixed with the characters "$V", and all the files required to store the values of scalars are prefixed with the characters "$$". This way, the scalar and vector files can be easily identified. A file is opened only if a variable exists. That is, the number of files opened is equal to the number of variables, unlike the case of declaring the memory where the memory size is fixed.

Also, there is no severe limitation on the number of clock times. Each file can accomodate a large number of clock times of information. The files can be closed after writing all the variable values for all clock times. At a later time when the values from these files are read, each file is referenced by means of a unique unit number. Each time a few lines are read from any file, the system has to return to the beginning of the file immediately. This is to ensure that the reading starts from the beginning of the file and the next time the file is referenced. If the system does not return to the beginning of the file, the reading would commence from a point where it was left the previous time. The other advantage of opening files for each variable is that the files are stored on the disk and the memory space is conserved.

Block (f)

Having created a file for each variable, the next step is to scan the file for the values of the variables. The key word to look for is "CLOCK #" (see Appendix). The output file contains both the description and communication files along with the values of variables at each clock time. The word "CLOCK #" occurs in the output file just before the values of the output variables for the first clock time. Therefore, the occurrence of the word "CLOCK #" marks the beginning of the variable values in the output file.

Blocks (g) & (h)

In the same line where the word "CLOCK #" occurs, the characters "!!!..." can be found (see Appendix). The number of these characters that can be found following the word "CLOCK #" is equal to the number of variables specified to be output in the communication file. A close look at the output file reveals this fact. The spacing between each of these characters "!!!..." determines whether each variable is a scalar or a vector. Starting from the first of these "!!!..." characters, it is easy to visualize that if the spacing between the nth and (n + 1)th of these characters is one blank space, then the nth variable is a scalar; otherwise, the nth variable is a vector. This rule holds for all but the last variable since the last of these "!!!..." characters cannot be compared with any other character. To determine if the last variable is a scalar or a vector, the first data line has to be read. The procedure is discussed in Block (i). Once a variable is identified as a vector, the number of bits associated with that vector has to be determined. A scalar, obviously, has just one bit. If the spacing between the nth and (n + 1)th of the characters "!!!..." is K (K not equal to 1), then the variable n is a vector of K bits. This is the simple algorithm implemented to determine the number of bits in a vector.

<u>Block (i)</u>

At this point, all the variables except the last variable are identified as vectors or scalars. To identify the last variable, the first data line has to be read. The column in which the last character "!!!..." occurs can easily be determined.

Now, if the next column in the data line contains a carriage return character, then the last variable is a scalar; otherwise, the last variable is a vector. If the last variable is a vector, the number of bits can be determined by keeping track of the number of columns after which a carriage return character occurs.

Block (j)

The next step is to read the AHPL output file, line by line, to get the data and store them in the respective files created for each vector and scalar. The column at which each vector and scalar begins is known. Having recognized the number of bits of each vector, that many columns of data are read for each vector. Of course, for a scalar only one column is read. This procedure is repeated for each line of data. Therefore, the data is extracted from the output file for vectors and scalars and the corresponding values stored in respective files.

Block (k)

Once the data values are read for any clock time, the next two lines contain the module and step information. Consequently, after reading any particular line for variable values, the next two consecutive lines have to be read to get the module and step information. The module and step information are stored in the files named "$$MODULE" and "$$STEP". The module number specifies the active module and the step number specifies the next active AHPL step.

Blocks (l) & (m)

The process of reading the data values, module information and step information can be continued until the last clock time is reached. At each clock time, values are read and a counter is incremented so that in the end the counter gives the number of clock times in the output file. The only problem left is to recognize the last clock time. Actually, the information regarding the number of clock times can be found in the communication file, but the problem is that the simulation may have halted before the specified number of clock times if the program encounters a DEADEND. For this reason, the only alternative is to scan the complete output file and determine the number of clock times by some other means. If the simulation runs for the number of clock times specified in the communication section, then after the last clock time a blank space and a carriage return are output. Thus, if any particular line contains only two characters, then it means that the simulation has continued for the number of clock times as specified in the communication section. In the event where a DEADEND is encountered, the simulator outputs a message to this effect (see Appendix). In such a case, the key word to look for is "DEADEND" or simply "DEAD". Therefore, either the word DEAD or a count of two characters signifies the end of the last clock time data, and the number of clock times in the simulator output file can be easily determined.

B. A Description of How to Use the Post-processor

The following description refers to figure 9. It may be recalled from figure 8 that the discussion of the flow chart terminated at a point

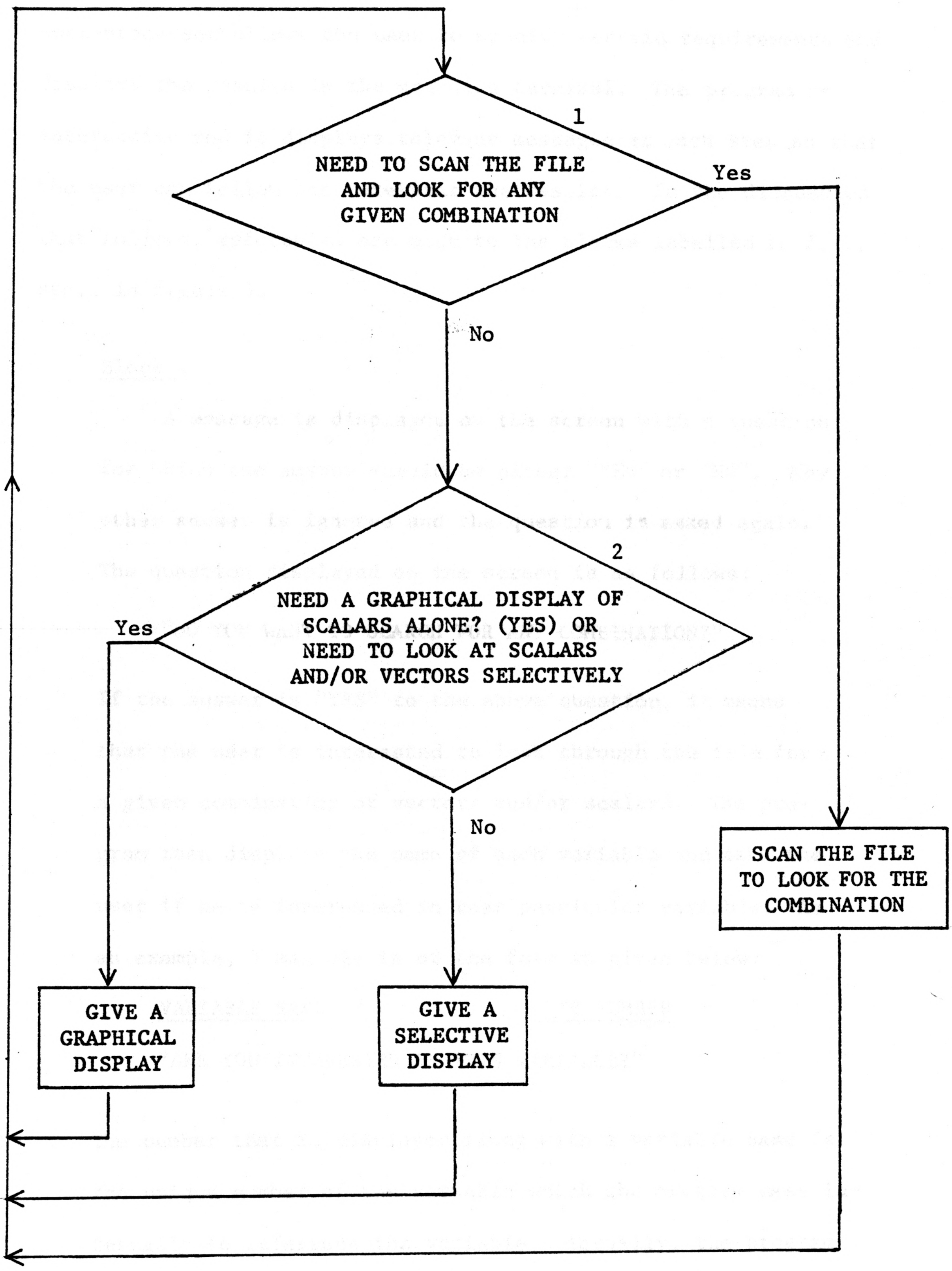

1
NEED TO SCAN THE FILE AND LOOK FOR ANY GIVEN COMBINATION
Yes
No
2
NEED A GRAPHICAL DISPLAY OF SCALARS ALONE? (YES) OR NEED TO LOOK AT SCALARS AND/OR VECTORS SELECTIVELY
Yes
No
SCAN THE FILE TO LOOK FOR THE COMBINATION
GIVE A GRAPHICAL DISPLAY
GIVE A SELECTIVE DISPLAY

Figure 9. Using the Post-processor

where all the data from AHPL output file was extracted and stored in different files. The data is now ready to be processed. The post-processor allows the user to specify certain requirements and displays the results in the graphics terminal. The program is interactive and it displays relevant messages at each step so that the user can follow and interpret the results. In the discussion that follows, references are made to the blocks labelled 1, 2, 3, etc., in figure 9.

<u>Block 1</u>

A message is displayed on the screen with a question for which the answer should be either "YES" or "NO". Any other answer is ignored and the question is asked again. The question displayed on the screen is as follows:

"DO YOU WANT TO SEARCH FOR ANY COMBINATION?"

If the answer is "YES" to the above question, it means that the user is interested to look through the file for a given combination of vectors and/or scalars. The program then displays the name of each variable and asks the user if he is interested in that particular variable. As an example, a message is of the form as given below:

<u>VARIABLE NAME</u> <u>ITS NUMBER</u>

"ARE YOU INTERESTED IN THIS VARIABLE?"

The number that is displayed along with a variable name is the unique number of the variable which the program uses internally to reference the variable. Actually, the program uses this unique number to open a file on the disk to store that particular variable's values. Later on, the program references the specific file belonging to any variable with the unique unit number.

Once the user specifies interest in a particular variable, the program allows the user to enter the value of that variable for searching the output file. If a variable is a scalar, the program accepts one bit (a 0 or 1) as the value of the scalar. If the variable is a vector, the program displays the number of bits and accepts that many bits of 1's and 0's for the value of the vector. In this way, all the variables of interest can be specified with the corresponding values of the variables. Once all the variables and their values are specified, the program scans through the file and returns with one of the following:

(1) If the combination never occurs, a relevant message is displayed to that effect, namely:

"THE GIVEN COMBINATION NEVER OCCURRED".

(2) If the given combination occurs in the file, a message is displayed indicating the number of times the combination occurred and the corresponding clock times.

"# OF TIMES THE GIVEN COMBINATION OCCURS

"THE GIVEN COMBINATION OCCURS AT THE FOLLOWING CLOCK TIMES".

The above information (1) and (2) gives an idea as to which section of the file is of most interest to selectively observe a subset of variables or have a graphical picture of of the scalars. If there is no necessity to look for a given combination in the file, then the program continues through block 2 after having accepted a "NO" as the response to block 1.

Block 2

At this time, the user can choose a graphical display of the scalars or a subset of the scalars. If the graphical display is not desired, a choice can be made to observe the subset of scalars and vectors in a selected group of clock times.

If the graphical display is chosen, then the program accepts the list of scalars for which graphical display is required. The program also accepts as input the start and stop clock times for the graphical display. The program does not display the graphical picture of the scalars relative to the clock for more than 25 clock cycles at any one given time. This is due to the fact that there is better clarity on the screen for a fewer number of clock cycles at a time (see fig. 6). Also, there is a limitation set due to the size of the screen width. Twenty-five clock times is chosen as a compromise between the screen size and a reasonable amount of clarity. Although more than 25 clock cycles of information is not displayed at a time, the user is not limited to specify only 25 clock times at any instance. If the total number of clock times specified exceeds 25, then the display is presented in portions of 25 clock cycles or less. As each portion is displayed, the system waits for the user's response to continue with the other portions of the display.

Before the presentation begins, the absolute value of the start time is displayed on the screen. Also the relative number of the clock time is displayed on top of the clock signal itself. At any given time, as many as six variables can be displayed graphically, one below the other, relative to the clock. This limitation is again due to the size of the screen.

If a graphical display is not desired, a top-to-bottom display of a subset of scalars and vectors can be chosen. This is a numeric type display (see fig. 7). It is possible to choose a subset of scalars and/or vectors and a selected number of clock times for the top-to-bottom display. This facility enables the user to choose only those variables of interest and select a group of clock times.

The program displays the name of each variable along with a number unique to that variable. A choice can be made regarding the set of variables of interest. Also, the start and stop clock times can be specified. In a top-to-bottom display of the subset of variables, if the screen size is inadequate, the remainder of the display does not continue until indicated by means of controls on the keyboard of the graphics terminal. The details regarding the actual use of these controls are explained in the User's Manual [9].

Although the user can specify as many vectors and scalars required for the top-to-bottom display, there is a limitation set by the screen width. The user does not realize this at the time of specifying the list of variables. Actually, the program checks to see if the given subset of variables can be accomodated on the screen. If all of the variables cannot be accomodated on the screen, a message is displayed on the screen and the user can specify a smaller subset of the variables.

As part of the top-to-bottom display, the user can request the module and step information at each clock time. This facility is available as an option and can be specified to the system.

In the entire discussion regarding the software implemented in the post-processor, there were no details about the actual program. The program is lengthy and complex, for which reason it was believed that the detailed version should not be included in the thesis. The program is well-documented and structured and appears in the User's Manual [9]. Every effort has been made to explain any ambiguous portions. Comments are included in the program, wherever it was felt necessary, to aid the interested reader in better understanding.

The User's Manual is written with the assumption that the reader has a good background in the use of the AHPL language and the AHPL simulator. The version of AHPL as accepted by the simulator does not use all the characters as described in the original version [1]. There are quite a few variations in the symbol usage and the details are available in the User Manual [8].

Due to the requirements of the graphical display as part of the post-processor, the program is written to work only with the graphics terminal. Hence, only the graphics terminal can be used in conjunction with the post-processor.

Chapter V

CONCLUSION

The post-processor has been developed to support many features required after the simulation of an AHPL description. The post-processing technique is particularly advantageous in cases where the AHPL output file is large. The graphical display is most advantageous in recognizing the level changes of control signals at given clock times. The facility of displaying a subset of the variables within given bounds of clock times is very helpful in design error detection. To enhance the use of the post-processor in the area of design error detection, the facility to search the output file for a given combination of variables is incorporated. This facility of being able to search the file for a given combination is quite powerful. At the present time, the post-processor displays the contents of registers in binary form. Although binary form is quite suitable for most cases, it may be advantageous to have the display of registers in other different forms such as octal, hexadecimal and decimal. The decimal form of displaying the register contents may be useful in increment, decrement, add and subtract operations.

A P P E N D I X

```
  AHPL CIRCUIT DESCRIPTION MODULE NUMBER   1:      DATE: 06/07/82
   TIME: 10:49:45   (HPSIM/2.0 U OF ARIZ.)
-
         1           AHPLMODULE:CONTROLLER
         2           EXINPUTS:Z;START.
         3           MEMORY:A[4];B[4];C[4];D[4].
         4           1     => (^START)/ (1).
         5           2     B <= A.
         6           3     C <= B ; => (Z)/(5).
         7           4     A <= (^C) ;=> (2).
         8           5     D <= C ; => (2).
         9           ENDSEQUENCE
        10           CONTROLRESET(1).
        11           END.
 1HPSIM COMMUNICATION SECTION FOR THE ABOVE MODULES  DATE: 06/07/82
TIME: 10:49:46
-       12           OPTION 7.
        13           CLOCKLIMIT 10.
        14           EXLINES START=0,1 ;Z=0#4,1.
        15           OUTPUTS START;Z;A;B;C;D.
        16           INITIALIZE A <= 3.
        17           DUMP ALL.
-
  ::::: EXECUTION  OF  THE HPSIM  PROGRAM  MODULES
  WILL START BY EXECUTING THE STEPS SPECIFIED BY THE
  (CONTROLRESET)  STATEMENT, ALL THE  EXECUTED STEPS
  AND  THEIR MODULE  NUMBER WILL  BE  LISTED  BEFORE
  THEIR CORRESPONDING OUTPUTS ARE PRINTED :::::
0 ::::: EXECUTION WILL STOP AFTER   10 CLOCK PULSES :::::
 1AHPL FUNCTION LEVEL SIMULATOR OUTPUT IS LISTED BELOW :
-              START
               ! Z
               ! ! A
               ! ! !    B
               ! ! !    !    C
               ! ! !    !    !    D
  CLOCK #      ! ! !    !    !    !
          1    0 0 0011 0000 0000 0000
0              MODULE#   1
               STEP  #   1
          2    1 0 0011 0000 0000 0000
0              MODULE#   1
               STEP  #   2
          3    1 0 0011 0000 0000 0000
0              MODULE#   1
               STEP  #   3
          4    1 0 0011 0011 0000 0000
0              MODULE#    1
               STEP  #    4
          5    1 1 0011 0011 0011 0000
0              MODULE#    1
               STEP  #    2
          6    1 1 1100 0011 0011 0000
0              MODULE#    1
               STEP  #    3
          7    1 1 1100 1100 0011 0000
0              MODULE#    1
               STEP  #    5
          8    1 1 1100 1100 1100 0000
0              MODULE#    1
               STEP  #    2
          9    1 1 1100 1100 1100 1100
0              MODULE#    1
               STEP  #    3
         10    1 1 1100 1100 1100 1100
0              MODULE#    1
               STEP  #    5

  ::::: PROGRAM REACHED THE CLOCKLIMIT. INTERPRETER STOPS :::::
 1HPSIM OUTPUT, DUMP OF MEMORIES AND REGISTERS:
- A        <   0:   0>  1100

  B        <   0:   0>  1100

  C        <   0:   0>  1100

  D        <   0:   0>  1100


R;
```

```
  AHPL CIRCUIT DESCRIPTION MODULE NUMBER    1:    DATE: 06/28/82
   TIME: 16:34:13    (HPSIM/2.0 U OF ARIZ.)
-
                 AHPLMODULE:CONTROLLER
                 MEMORY:A[4];B[4];C[4];D[4];Z[1];E[1];F[1].
                 CLUNITS:  INC[4]  (A).
                 1      B <= A.
                 2      C <= B.
                 3      D <= C.
                 4      A <= INC(A).
                 5      Z <= &/A.
                 6      E <= (~Z).
                 7      F <= A[0] @ B[3].
                 8      => (Z)/(10).
                 9      => (1).
                 10     A <= (~A).
                        ENDSEQUENCE
                        CONTROLRESET(1).
                 END.
 1HPSIM COMMUNICATION SECTION FOR THE ABOVE MODULES    DATE: 06/28/82
   TIME: 16:34:14
-                OPTION 7.
                 CLOCKLIMIT 250.
                 OUTPUTS A;B;C;D;E;F;Z.
                 INITIALIZE Z <= '1;E <= '1;F <= '1;A <= '2;
    B<= '3;C<= '4;D <='5.
                 DUMP ALL.
-
  ::::: EXECUTION  OF  THE HPSIM  PROGRAM  MODULES
  WILL START BY EXECUTING THE STEPS SPECIFIED BY THE
  (CONTROLRESET)  STATEMENT, ALL THE  EXECUTED STEPS
  AND  THEIR MODULE  NUMBER WILL  BE  LISTED  BEFORE
  THEIR CORRESPONDING OUTPUTS ARE PRINTED :::::
0 ::::: EXECUTION WILL STOP AFTER  250 CLOCK PULSES :::::
 1AHPL FUNCTION LEVEL SIMULATOR OUTPUT IS LISTED BELOW :
-             A
              !    B
              !    !    C
              !    !    !    D
              !    !    !    !    E
              !    !    !    !    ! F
              !    !    !    !    ! ! Z
  CLOCK #     !    !    !    !    ! ! !
          1   0010 0011 0100 0101 1 1 1
0             MODULE#    1
              STEP  #    2
```

```
        2    0010 0010 0100 0101 1 1 1
0            MODULE#    1
             STEP  #    3
        3    0010 0010 0010 0101 1 1 1
0            MODULE#    1
             STEP  #    4
        4    0010 0010 0010 0010 1 1 1
0            MODULE#    1
             STEP  #    5
        5    0011 0010 0010 0010 1 1 1
0            MODULE#    1
             STEP  #    6
        6    0011 0010 0010 0010 1 1 0
0            MODULE#    1
             STEP  #    7
        7    0011 0010 0010 0010 1 1 0
0            MODULE#    1
             STEP  #    8
        8    0011 0010 0010 0010 1 0 0
0            MODULE#    1
             STEP  #    9
        9    0011 0010 0010 0010 1 0 0
0            MODULE#    1
             STEP  #    1
       10    0011 0010 0010 0010 1 0 0
0            MODULE#    1
             STEP  #    2
       11    0011 0011 0010 0010 1 0 0
0            MODULE#    1
             STEP  #    3
       12    0011 0011 0011 0010 1 0 0
0            MODULE#    1
             STEP  #    4
       13    0011 0011 0011 0011 1 0 0
0            MODULE#    1
             STEP  #    5
       14    0100 0011 0011 0011 1 0 0
0            MODULE#    1
             STEP  #    6
       15    0100 0011 0011 0011 1 0 0
0            MODULE#    1
             STEP  #    7
       16    0100 0011 0011 0011 1 0 0
0            MODULE#    1
             STEP  #    8
       17    0100 0011 0011 0011 1 1 0
0            MODULE#    1
             STEP  #    9
        18     0100 0011 0011 0011 1 1 0
0              MODULE#     1
               STEP   #    1
        19     0100 0011 0011 0011 1 1 0
0              MODULE#     1
               STEP   #    2
        20     0100 0100 0011 0011 1 1 0
0              MODULE#     1
               STEP   #    3
        21     0100 0100 0100 0011 1 1 0
0              MODULE#     1
               STEP   #    4
        22     0100 0100 0100 0100 1 1 0
0              MODULE#     1
               STEP   #    5
        23     0101 0100 0100 0100 1 1 0
0              MODULE#     1
               STEP   #    6
        24     0101 0100 0100 0100 1 1 0
0              MODULE#     1
               STEP   #    7
        25     0101 0100 0100 0100 1 1 0
0              MODULE#     1
               STEP   #    8
        26     0101 0100 0100 0100 1 0 0
0              MODULE#     1
               STEP   #    9
        27     0101 0100 0100 0100 1 0 0
0              MODULE#     1
               STEP   #    1
        28     0101 0100 0100 0100 1 0 0
0              MODULE#     1
               STEP   #    2
        29     0101 0101 0100 0100 1 0 0
0              MODULE#     1
               STEP   #    3
        30     0101 0101 0101 0100 1 0 0
0              MODULE#     1
               STEP   #    4
        31     0101 0101 0101 0101 1 0 0
0              MODULE#     1
               STEP   #    5
        32     0110 0101 0101 0101 1 0 0
0              MODULE#     1
               STEP   #    6
        33     0110 0101 0101 0101 1 0 0
0              MODULE#     1
               STEP   #    7
        34    0110 0101 0101 0101 1 0 0
0             MODULE#    1
              STEP  #    8
        35    0110 0101 0101 0101 1 1 0
0             MODULE#    1
              STEP  #    9
        36    0110 0101 0101 0101 1 1 0
0             MODULE#    1
              STEP  #    1
        37    0110 0101 0101 0101 1 1 0
0             MODULE#    1
              STEP  #    2
        38    0110 0110 0101 0101 1 1 0
0             MODULE#    1
              STEP  #    3
        39    0110 0110 0110 0101 1 1 0
0             MODULE#    1
              STEP  #    4
        40    0110 0110 0110 0110 1 1 0
0             MODULE#    1
              STEP  #    5
        41    0111 0110 0110 0110 1 1 0
0             MODULE#    1
              STEP  #    6
        42    0111 0110 0110 0110 1 1 0
0             MODULE#    1
              STEP  #    7
        43    0111 0110 0110 0110 1 1 0
0             MODULE#    1
              STEP  #    8
        44    0111 0110 0110 0110 1 0 0
0             MODULE#    1
              STEP  #    9
        45    0111 0110 0110 0110 1 0 0
0             MODULE#    1
              STEP  #    1
        46    0111 0110 0110 0110 1 0 0
0             MODULE#    1
              STEP  #    2
        47    0111 0111 0110 0110 1 0 0
0             MODULE#    1
              STEP  #    3
        48    0111 0111 0111 0110 1 0 0
0             MODULE#    1
              STEP  #    4
        49    0111 0111 0111 0111 1 0 0
0             MODULE#    1
              STEP  #    5
        50   1000 0111 0111 0111 1 0 0
0            MODULE#    1
             STEP  #    6
        51   1000 0111 0111 0111 1 0 0
0            MODULE#    1
             STEP  #    7
        52   1000 0111 0111 0111 1 0 0
0            MODULE#    1
             STEP  #    8
        53   1000 0111 0111 0111 1 0 0
0            MODULE#    1
             STEP  #    9
        54   1000 0111 0111 0111 1 0 0
0            MODULE#    1
             STEP  #    1
        55   1000 0111 0111 0111 1 0 0
0            MODULE#    1
             STEP  #    2
        56   1000 1000 0111 0111 1 0 0
0            MODULE#    1
             STEP  #    3
        57   1000 1000 1000 0111 1 0 0
0            MODULE#    1
             STEP  #    4
        58   1000 1000 1000 1000 1 0 0
0            MODULE#    1
             STEP  #    5
        59   1001 1000 1000 1000 1 0 0
0            MODULE#    1
             STEP  #    6
        60   1001 1000 1000 1000 1 0 0
0            MODULE#    1
             STEP  #    7
        61   1001 1000 1000 1000 1 0 0
0            MODULE#    1
             STEP  #    8
        62   1001 1000 1000 1000 1 1 0
0            MODULE#    1
             STEP  #    9
        63   1001 1000 1000 1000 1 1 0
0            MODULE#    1
             STEP  #    1
        64   1001 1000 1000 1000 1 1 0
0            MODULE#    1
             STEP  #    2
        65   1001 1001 1000 1000 1 1 0
0            MODULE#    1
             STEP  #    3
        66   1001 1001 1001 1000 1 1 0
0            MODULE#    1
             STEP  #    4
        67   1001 1001 1001 1001 1 1 0
0            MODULE#    1
             STEP  #    5
        68   1010 1001 1001 1001 1 1 0
0            MODULE#    1
             STEP  #    6
        69   1010 1001 1001 1001 1 1 0
0            MODULE#    1
             STEP  #    7
        70   1010 1001 1001 1001 1 1 0
0            MODULE#    1
             STEP  #    8
        71   1010 1001 1001 1001 1 0 0
0            MODULE#    1
             STEP  #    9
        72   1010 1001 1001 1001 1 0 0
0            MODULE#    1
             STEP  #    1
        73   1010 1001 1001 1001 1 0 0
0            MODULE#    1
             STEP  #    2
        74   1010 1010 1001 1001 1 0 0
0            MODULE#    1
             STEP  #    3
        75   1010 1010 1010 1001 1 0 0
0            MODULE#    1
             STEP  #    4
        76   1010 1010 1010 1010 1 0 0
0            MODULE#    1
             STEP  #    5
        77   1011 1010 1010 1010 1 0 0
0            MODULE#    1
             STEP  #    6
        78   1011 1010 1010 1010 1 0 0
0            MODULE#    1
             STEP  #    7
        79   1011 1010 1010 1010 1 0 0
0            MODULE#    1
             STEP  #    8
        80   1011 1010 1010 1010 1 1 0
0            MODULE#    1
             STEP  #    9
        81   1011 1010 1010 1010 1 1 0
0            MODULE#    1
             STEP  #    1
```

```
         82    1011 1010 1010 1010 1 1 0
0              MODULE#    1
               STEP  #    2
         83    1011 1011 1010 1010 1 1 0
0              MODULE#    1
               STEP  #    3
         84    1011 1011 1011 1010 1 1 0
0              MODULE#    1
               STEP  #    4
         85    1011 1011 1011 1011 1 1 0
0              MODULE#    1
               STEP  #    5
         86    1100 1011 1011 1011 1 1 0
0              MODULE#    1
               STEP  #    6
         87    1100 1011 1011 1011 1 1 0
0              MODULE#    1
               STEP  #    7
         88    1100 1011 1011 1011 1 1 0
0              MODULE#    1
               STEP  #    8
         89    1100 1011 1011 1011 1 0 0
0              MODULE#    1
               STEP  #    9
         90    1100 1011 1011 1011 1 0 0
0              MODULE#    1
               STEP  #    1
         91    1100 1011 1011 1011 1 0 0
0              MODULE#    1
               STEP  #    2
         92    1100 1100 1011 1011 1 0 0
0              MODULE#    1
               STEP  #    3
         93    1100 1100 1100 1011 1 0 0
0              MODULE#    1
               STEP  #    4
         94    1100 1100 1100 1100 1 0 0
0              MODULE#    1
               STEP  #    5
         95    1101 1100 1100 1100 1 0 0
0              MODULE#    1
               STEP  #    6
         96    1101 1100 1100 1100 1 0 0
0              MODULE#    1
               STEP  #    7
         97    1101 1100 1100 1100 1 0 0
0              MODULE#    1
               STEP  #    8
         98   1101 1100 1100 1100 1 1 0
0             MODULE#    1
              STEP  #    9
         99   1101 1100 1100 1100 1 1 0
0             MODULE#    1
              STEP  #    1
        100   1101 1100 1100 1100 1 1 0
0             MODULE#    1
              STEP  #    2
        101   1101 1101 1100 1100 1 1 0
0             MODULE#    1
              STEP  #    3
        102   1101 1101 1101 1100 1 1 0
0             MODULE#    1
              STEP  #    4
        103   1101 1101 1101 1101 1 1 0
0             MODULE#    1
              STEP  #    5
        104   1110 1101 1101 1101 1 1 0
0             MODULE#    1
              STEP  #    6
        105   1110 1101 1101 1101 1 1 0
0             MODULE#    1
              STEP  #    7
        106   1110 1101 1101 1101 1 1 0
0             MODULE#    1
              STEP  #    8
        107   1110 1101 1101 1101 1 0 0
0             MODULE#    1
              STEP  #    9
        108   1110 1101 1101 1101 1 0 0
0             MODULE#    1
              STEP  #    1
        109   1110 1101 1101 1101 1 0 0
0             MODULE#    1
              STEP  #    2
        110   1110 1110 1101 1101 1 0 0
0             MODULE#    1
              STEP  #    3
        111   1110 1110 1110 1101 1 0 0
0             MODULE#    1
              STEP  #    4
        112   1110 1110 1110 1110 1 0 0
0             MODULE#    1
              STEP  #    5
        113   1111 1110 1110 1110 1 0 0
0             MODULE#    1
              STEP  #    6
```

```
          114    1111 1110 1110 1110 1 0 1
 0               MODULE#    1
                 STEP   #   7
          115    1111 1110 1110 1110 0 0 1
 0               MODULE#    1
                 STEP   #   8
          116    1111 1110 1110 1110 0 1 1
 0               MODULE#    1
                 STEP   #  10
          117    1111 1110 1110 1110 0 1 1
 0               MODULE#    0
                 STEP   #   0
   ::::: PROGRAM REACHED A COMPLETE DEADEND. INTERPRETER STOPS :::::
  1HPSIM OUTPUT, DUMP OF MEMORIES AND REGISTERS:
 - A      < 0:   0> 1111

   B      < 0:   0> 1110

   C      < 0:   0> 1110

   D      < 0:   0> 1110

   Z      < 0:   0> 1

   E      < 0:   0> 0

   F      < 0:   0> 1


R;
```

BIBLIOGRAPHY

1. F. J. Hill and G. R. Peterson, Digital Systems: Hardware Organization and Design. 2nd ed., Wiley, New York, 1978.

2. F. J. Hill, "Introducing AHPL," Computer, vol. 7, pp. 28-30, Dec. 1974.

3. F. J. Hill, "Updating AHPL," IEEE Proceedings of the International Symposium on Computer Hardware Description Languages and their Applications, Sept. 1975, pp. 22-29.

4. F. J. Hill and Z. Navabi, "Extending Second Generation AHPL Software to Accomodate AHPL III," IEEE Proceedings of the 4th International Symposium on Computer Hardware Description Languages, Oct. 1979, pp. 47-53.

5. F. J. Hill, R. E. Swanson, M. Masud and Z. Navabi, "Structure Specification with a Procedural Hardware Description Language," IEEE Transactions on Computers, vol. C-30, Feb. 1981, pp. 157-161.

6. D. P. Siewiorek, C. G. Bell and A. Newell, Computer Structures: Principles and Examples. McGraw Hill, 1982.

7. S. G. Shiva, "Computer Hardware Description Languages - A Tutorial," Proceedings of the IEEE, vol. 67, Dec. 1979, pp. 1605-1615.

8. F. J. Hill, R. Swanson and Z. Navabi, User Manual for AHPL Simulator (HPSIM 2) and the AHPL Compiler (HPCOM). Dept. of Elec. Engg., University of Arizona, Jan. 1979.

9. J. H. Tracey and P. S. Madhavan, User Manual for the Post-processor of AHPL Simulator. Dept. of Elec. Engg., Kansas State University, 1982.

10. Y. Chu, "Why do we need Computer Hardware Description Languages," Computer, vol. 7, Dec. 1974, pp. 18-22.

11. J. P. Hayes, "CHDL's in Design Automation Systems," Proceedings of the IEEE International Symposium on Computer Hardware Description Languages, 1975, pp. 53-69.

12. J. L. Houle, "New Applications of CHDL's," Proceedings of the IEEE International Symposium on Computer Hardware Description Languages, 1975, pp. 76-91.

ACKNOWLEDGEMENTS

I wish to thank my thesis supervisor, Dr. James H. Tracey, for his excellent guidance during the years of my study and suggestions during the preparation of this thesis. I also wish to thank my committee members Dr. Virgil Wallentine and Dr. Richard Gallagher for their suggestions and support.

I would also like to thank Dr. Nasir Ahmed for his comments and support.

Also, thanks to my wife Vinatha for her superb typing of this thesis.

A POST-PROCESSING SYSTEM FOR AN AHPL SIMULATOR

by

PUNDI SREENIVASAN MADHAVAN

B.S., Madras Institute of Technology, 1976
M.S., University of Madras, 1980

AN ABSTRACT OF A MASTER'S THESIS

submitted in partial fulfillment

of the requirements for the degree

MASTER OF SCIENCE

DEPARTMENT OF ELECTRICAL ENGINEERING

KANSAS STATE UNIVERSITY

Manhattan, Kansas

1982

ABSTRACT

The main objective of this work, "A Post-processing System For An AHPL Simulator," is to enhance the use of the AHPL simulator in hardware design applications. The original AHPL simulator provides an output file which is difficult to analyze. The analysis of the output file is not convenient, particularly for larger number of variables and clock times. The original version of the simulator does not support features such as graphical presentation, display of a subset of the variables in given bounds of clock times and searching the output file for a given combination values of variables. All of the above-mentioned features are implemented in the post-processor. In the simulation of the AHPL sequence, from a design standpoint, the post-processor offers significant support.