MULTIMODULE SIMULATION TECHNIQUES FOR CHIP LEVEL MODELING

by

Chang H. Cho

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Electrical Engineering

APPROVED:

J. R. Armstrong, Chairman

J. G. Tront

D. W. Luse

March 1986

Blacksburg, Virginia

MULTIMODULE SIMULATION TECHNIQUES FOR CHIP LEVEL MODELING

by

Chang H. Cho

J. R. Armstrong, Chairman

Electrical Engineering

(ABSTRACT)

A design and implementation of a multimodule chip-level simulator whose source description language is based on the original GSP2 system is described. To enhance the simulation speed, a special addressing ("sharing single memory location") scheme is used in the implementation of pin connections. The basic data structures and algorithms for the simulator are described. The developed simulator can simulate many digital devices interconnected as a digital network. It also has the capability of modeling external buses and handling the suspension of processes in the environment of multimodule simulation. An example of a multimodule digital system simulation is presented.

ACKNOWLEDGEMENTS

I sincerely wish to thank Dr. James R. Armstrong for his continuous encouragement and suggestions during the preparation of this thesis. I would also like to thank Drs. Tront and Luse for serving as my committee. I would like to express my deepest gratitude to my parents in Korea. Last but not least, I would like to thank my wife Changsook and two sons Kyunghoon and Sunghoon for their encouragement and patience.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

1. INTRODUCTION

With the increasing complexity of digital system design, logic simulation has been extensively used in both design verification and diagnostic considerations of digital systems. At the initial stage of logic simulation history, it was possible to model a digital system at gate level and simulate its gate level logic and timing. However, the advent of LSI or VLSI has rendered the use of the gate level model impractical for the modeling and simulation of large digital systems. As a result there is currently a great deal of interest in development of chip level models for LSI or VLSI devices [1,2].

Research for the chip level modeling and simulation has been carried out at Virginia Tech. under Dr. J. R. Armstrong since 1976. The first chip level simulator was completed in 1978 by Marvin Ellis [3]. It was capable of simulating a single device, with no provisions for connecting the device into a system. This simulator, known as GSP (General Simulation Program), consists of an assembler that translates an assembly level modeling language used in describing the device and the simulator program itself. The system that added multimodule simulation capability to the GSP was completed by Donald Devlin in 1981 [4]. The multimodule simulator has been extensively used since then [5,6,7,8].

Although the original GSP system has been a useful tool for the research of chip level simulation, its disadvantages have also been disclosed. First, since the source description is translated into an intermediate microcode language, it imposes extra time upon interpreting the microcode at runtime. Secondly, it was difficult for the modelers to read and code the source description because it was written in an assembly level language. As a result the second version of GSP, GSP2, was developed by James Kerr in 1984 [9]. It adopted a high-level block-structured language as the source description which is compiled directly into host (Digital Equipment Corporation's VAX-11) machine code to act as runtime routines for the simulator program. It obviously included features to overcome the disadvantages of the GSP described above, but it was again a single module simulator. This thesis describes the design and implementation of a multimodule chip-level simulator whose source description language is based on the original GSP2.

In the chip level simulation, each individual chip model in the system being simulated is created in isolation [1]. Therefore, for the complete modeling of a multi-chip digital system, the interconnection among the chips should be also described. The connection information will be used in simulating the interaction among the chip models at runtime. To design a multimodule chip level simulator, it should be first determined how to incorporate the connection information into the simulator. The following two methods can be considered.

1. The connection information is incorporated into each chip model when it is compiled. In this case, a preprocessor analyzes the interconnection and provides each chip model with necessary connection information. The advantage of this method is that the preprocessor assigns the memory locations to connected pins and can make the pin variables in the same connection share the same memory location. This "sharing single memory location" scheme enhances the simulation performance because it needs only O(n) time and memory space compared with $O(n^2)$ needed by the complete fanout tracing scheme (n denotes the number of connected pins). However, it has the disadvantage that all chip models have to be recompiled if the interconnection is changed.
2. The connection information is directly incorporated into the executable image. The GSP1 system adopted this method [4]. In this case, the chip models need not be recompiled even if the interconnection is changed. However, it has two disadvantages. First, a signal change on a source pin should be propagated to all destination

pins (complete fanout tracing) because each connected pin has its own memory location. It needs $O(n^2)$ time and memory complexity. Secondly, the connection information should be incorporated into the executable image whenever the simulation is restarted, which also reduces the simulation speed.

The GSP2 system adopts the first method to get higher simulation speed. A compiler ("connect compiler") is used as the preprocessor that analyzes the interconnection. To prevent frequent recompilation of chip models, the connect compiler is designed to thoroughly check the validity of the interconnection. In the GSP2 system, the user would generate a connection description file and two or more module description files to simulate a multi-chip digital system. The connection description file describes the interconnection among the chips. The module description file describes the behavior and the timing of an individual chip. The connect compiler compiles the connection description file and generates two files, the connection topology file and the connect file. The connection topology file contains all connection information needed by the chip models. Whenever the module description compiler compiles a chip model, it reads the connection information, incorporates it into the chip model, and constructs a multimodule invocation mechanism through which the chip model can interact with other chip models.

The connect file contains VAX-11 assembly codes which assign the memory locations where the values of connected pins are stored at runtime.

The main reason that the original GSP2 is not a multimodule simulator is that the connect compiler does not generate sufficient connection information needed by the chip models. It does not generate the connection topology of the system being simulated. Therefore, the module description compiler has no way of knowing where a connected pin in a module is connected. It prevents any signal change on a connected pin from invoking other modules. The second problem of the original GSP2 is that it cannot model external buses which are essential to the simulation of a multimodule digital system. In the original GSP2, if two or more pins are connected, they share the same memory location. It means that when a connected pin changes its value, all pins connected to it are automatically updated to the new value. The advantage of this "sharing single memory location" scheme was already pointed out earlier in this chapter. However, the scheme inherently disables modeling of an external bus because the signal change of any bus pin is copied to other pins connected to it without any operation among the bus pins. The last main problem is in the implementation of the WAIT statement which is used in suspending a process. The original GSP2 simulator works well if the suspended process is not invoked again before the waiting time expires. However, it destroys the simulation flow if another event invokes the process before the waiting time expires. This is because the simulator cannot access the proper entry point to the processor in this situation.

To solve the problems of the original GSP2 system, the connect compiler, the module description compiler, and the simulator have been designed as follows:

1. A new syntax for the connection description file has been designed. It forces the modeler to describe each connected pin with module name, pin name, pin size, connection range, pin type, and bus type (optional).

2. The connect compiler has been designed to generate all necessary information needed by the chip models. It reads the connection description file, determines the type of each connection, assigns proper memory locations to all connected pins, and writes all connection information (connection type, connection range, connection topology, memory locations, and bus type) into the connection topology file in a regular format that can be recognized by the module description compiler. It adopts the following special connection scheme for bus pins.

- A bidirect bus pin is assigned two different memory locations, an input port and an output port.
- Input pins and input ports of bidirect pins in a bus share the same memory location.
- Output pins and output ports of bidirect pins in a bus are not connected, i.e., they keep their own memory locations.

Since the "sharing single memory location" scheme is still used for pins other than bus pins, it enables the modeling of buses without much affecting the O(n) time and memory space complexity.

3. The module description compiler makes a proper data structure for multimodule invocation and generates codes for bus functions. It reads the list of connection definitions from the connection topology file and constructs two connection tables, the normal connection table and the bus connection table. For each pin in the normal connection table, the compiler makes a linked list that contains all pins connected to it. On the other hand, if a pin in the bus connection table appears on the left hand side of an assignment statement, the compiler generates code for a proper bus function.

4. The simulator has been designed to perform multimodule invocation and correct WAIT operations at runtime. Whenever a connected pin changes its value, the simulator traverses the linked list constructed by the module description compiler and invokes all the modules in the linked list. It also uses a proper data structure that can handle infinite entry points, performing correct WAIT operations without exceptions.

5. A processor system has been simulated to verify the multimodule simulation capability of the developed simulator.

Chapter 2, chapter 3, and chapter 4 details the design and implementation of the connect compiler, the module description compiler, and the simulator, respectively. Chapter 5 shows an example of modeling and simulation of a multimodule digital system. Chapter 6 describes the further improvements that could be made to the GSP2 system. Chapter 7 concludes and summarizes the work accomplished.

2. THE CONNECT COMPILER

2.1 SYNTAX OF THE CONNECTION DESCRIPTION FILE

In the GSP2 system, a separate file (connection description file) is used in describing the pin connections among the chips in the system being simulated. Most pin connections in a digital system are basically simple wirings. However, external buses that exist in almost all digital systems should be also modeled. Therefore, the syntax of the connection description file is designed so that both types of connections can be described. The syntax for a connection description file is given by the following diagram:

```
connection description file:
    __>'CONNECT'>______________________>'ENDCONN'>__>>
                 |__>connection>__|
                  |____<';'<____|
```

The syntax for the nonterminal "connection" is

```
connection:
     ____> pindef >____> pindef >______>>
                   |__< ',' <____|
```

The syntax for the "pindef" nonterminal is

```
pindef:
__>identifier>__>':'>__>identifier>__>__________________>
                                      |__>size_range>__|

____>'INPUT'>_______>______________________________>>
 |__>'OUTPUT'>____|    |_>':'>___>'BUSAND'>___>__|
 |__>'BIDIRECT'>__|            |_>'BUSOR'>___|
```

The first "identifier" gives the name of the module to which the pin vector belongs. The second "identifier" is the name of the pin vector. The "size_range" specifies the size and the connected portion of the pin vector. The "size_range" is followed by the type of the pin. In the GSP2 language, three types of pin variables are defined:

1. INPUT - pins which take values from other modules
2. OUTPUT - pins which send values to other modules
3. BIDIRECT - pins that can both receive and send values

Thus, the modeler is to select a pin type that corresponds to the data flow of the pin in the connection. The syntax of "pindef" is completed by an optional bus attribute, BUSAND or BUSOR. BUSAND represents a wired-AND logic and BUSOR represents a wired-OR logic.

Using the syntax described above, we can model any intermodule connection in digital systems. However, there is no method of describing an external source signal that is applied to two or more input pins in the simulated system. This is because the signal does not belong to any module in the system and therefore the user cannot specify the name of the module in which the signal resides. In this case, only the pins that reside in the system are written in the connection statement. This is in a sense a conflict because every pin in the connection has the type of INPUT, i.e., there is no source signal. To resolve the problem, the connect compiler assumes that the first pin written in the connection statement represents the external signal. Therefore, once the modeler writes a pin definition at the beginning of this type of connection, he must use it as the one that represents the external signal throughout the simulation. This pin is defined as a representative pin.

2.2 DETERMINATION OF CONNECTION TYPES

In the GSP2 system, two connection types are defined as following:

- A normal connection is a connection that has only one source signal in it.
- A bus connection is a connection that has two or more source signals in it.

The connect compiler determines the connection types by counting the number of source signals in the connection. According to the definitions of pin types, only OUTPUT or BIDIRECT pins can be used as source signals. However, in case that only INPUT pins exist in a connection, the first pin (representative pin) in the connection is considered as a source. Therefore, the compiler considers the connection as a normal connection without outputting an error message. This connection is defined as TYPE 1 to be distinguished from other normal connections which are defined as TYPE 2. In fact, a TYPE 2 connection consists of one OUTPUT pin and one or more INPUT pins. There is a restriction in writing a TYPE 2 connection statement - the OUTPUT pin must be written first. Although this rule imposes a restriction on the modeler, it greatly reduces the burden of the connection description compiler. On the other hand, there is no restriction on writing a bus connection. If there are more than one OUTPUT pin or at least one BIDIRECT pin in a connection, it is considered as a bus connection. For the consistency, a bus connection is defined as TYPE 3. Figure 1 shows the algorithm of determining the connection types. It is notable that the connect compiler detects the following invalid connections.

1. There is only one pin.
2. There are only OUTPUT pins.

3. There are only one BIDIRECT pin and INPUT pins.
4. There are only one BIDIRECT pin and OUTPUT pins.

The bus attributes BUSAND and BUSOR can be included only in pin definitions in a bus connection (TYPE 3). Therefore, the modeler must know the type of the connection before he writes it into the connection description file. If a pin in a normal connection (TYPE 1 or TYPE 2) contains a bus attribute or a pin in a bus connection does not contain a bus attribute, the compiler outputts a proper error message.

There is another restriction on the syntax of "pindef". According to the syntax diagram, a pin is defined by both the name of the module that the pin resides in and the name of the pin itself. Therefore, the modeler can use the same pin name for the pins in different modules. This is true when a normal connection is described. However, this cannot be done in case of a bus connection because the module description compiler uses only pin names when it generates the code for bus operations (it will be explained in chapter 3).

Figure 2 shows an example of a connection description file. The first connection in the example illustrates a TYPE 1 connection because it contains only input pins. It represents a single bit external signal "clock" that is connected

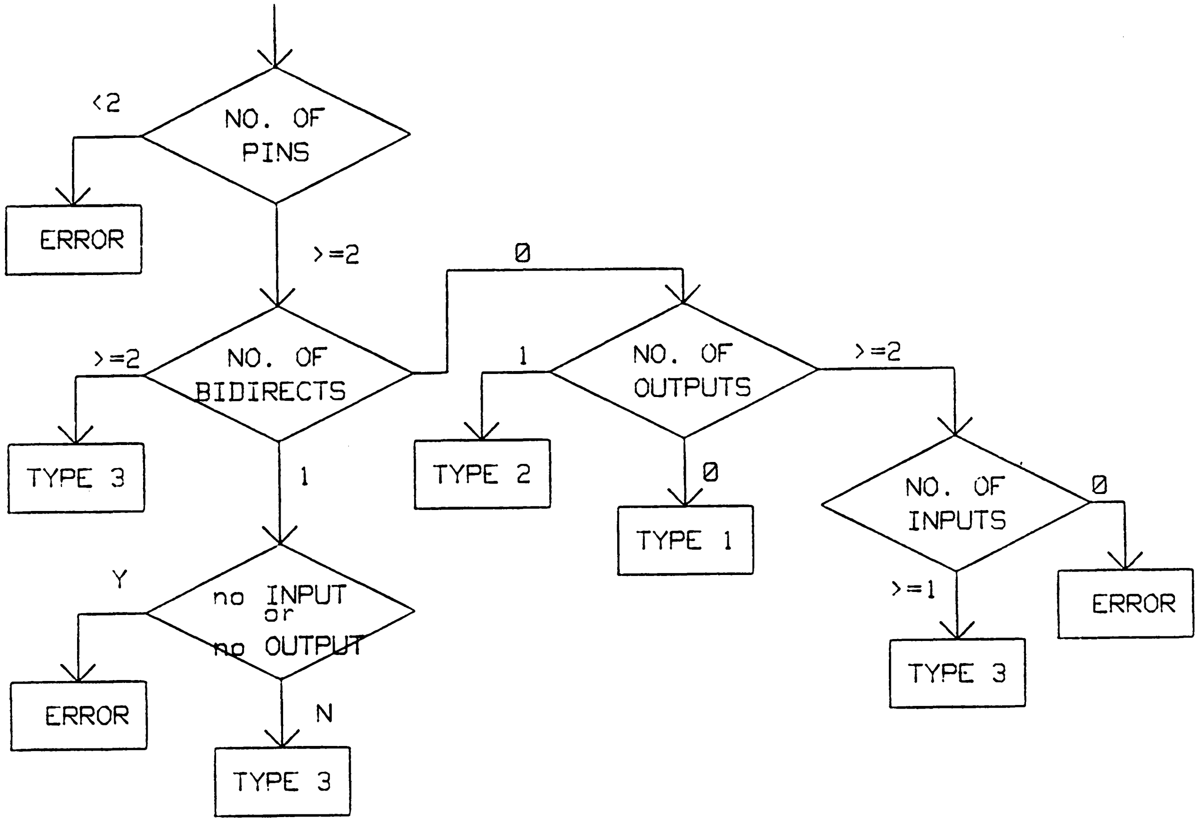

Figure 1. Algorithm of Determining Connection Types

to modules A, B, and C. Since there is no method of describing the external signal, the modeler must use the representative pin (A: clock) whenever he refers to the external signal. The second and the third connection show a subranged connection, i.e., the least significant two bits of an output pin "control" of module A are connected to the pin "siga" of module B and its most significant 2 bits are connected to "sigb" of module C. They are TYPE 2 connections. The fourth and the fifth connection represent bus connections (TYPE 3).

```
CONNECT

   A: clock [0|1,0|1] INPUT,
   B: clock [0|1,0|1] INPUT,
   C: clock [0|1,0|1] INPUT;

   A: control [0|4,0|2] OUTPUT,
   B: siga [0|2,0|2] INPUT;

   A: control [0|4,2|2] OUTPUT,
   C: sigb [0|2,0|2] INPUT;

   A: data_line [0|16,0|16] BIDIRECT: BUSAND,
   B: data_bus [0|16,0|16] BIDIRECT: BUSAND;

   A: busa [0|4,0|4] OUTPUT: BUSOR,
   B: busb [0|4,0|4] OUTPUT: BUSOR,
   C: busc [0|4,0|4] INPUT: BUSOR

ENDCONN
```

Figure 2. A Connection Description File

The fourth connection represents a wired-AND logic and the fifth connection represents a wired-OR logic.

2.3 PIN CONNECTION SCHEME

As described in section 2.1, a connected pin is defined by module name, pin name, pin size, connection range, pin type, and bus type (optional). The connect compiler assigns a single byte to each bit of a pin vector. This "one bit per byte" scheme is used to enable a subranged connection. If the value of a pin bit is logic 1, "true_byte" (11111111) is stored in the memory location assigned to the bit. If the value of a pin bit is logic 0, "false_byte" (00000000) is stored. Currently, only these two values are used in representing pin bit values. In case that other states (for example, "high impedance" state or "unknown" state) are required in the future, the states can be represented by arbitrary values between "true_byte" and "false_byte". The connect compiler assigns proper memory locations to a pin vector depending on its size and type. Whenever a pin definition is encountered in a connection statement, the connect compiler first checks if the pin was previously declared or not. If so (it means that memory locations were already assigned to the pin vector), no more memory location is assigned to the pin vector. Otherwise, the compiler checks the type of the pin. If it is an input or an output pin, the compiler assigns a single memory location to each bit of the pin vector. Therefore, the number of the memory locations assigned to the pin vector is same as its size. On the other hand, if it is a bidirect pin, it assigns two memory locations to each bit of the pin vector. In other words, a bidirect pin vector is assigned two ports, an input port, and an output port. Actually, the connect compiler first assigns the memory locations for the output port and then assigns the memory locations for the input port. Therefore, the total number of memory locations assigned to a bidirect pin vector becomes twice the size of the pin vector. The module description compiler uses the output port as an argument of a bus operation and stores the result at the input port. Figure 3 shows the memory locations allocated to the pins when the connect compiler compiles the following connection statement.

```
MODA: pina[0|4,0|4] BIDIRECT: BUSAND,
MODB: pinb[0|4,0|4] BIDIRECT: BUSAND,
MODC: pinc[0|4,0|4] INPUT: BUSAND,
MODD: pind[0|4,0|4] OUTPUT: BUSAND
```

After the connect compiler finishes assigning memory locations to all pins in a connection statement, it determines the type of the connection by counting the number of source pins in the connection (figure 1). Then, the compiler begins to connect the pins in two ways. If the connection is a normal connection (TYPE 1 or TYPE 2), every pin in the connection shares the same memory locations. If the connection is a bus connection (TYPE 3), input pins and input ports of bidirect pins share the same memory locations. However, output pins and output ports of bidirect pins are not connected, i.e., they keep their own memory locations. Figure 4 shows the changed memory locations for the pins in the figure 3 after they are connected.

2.4 THE CONNECTION TOPOLOGY FILE

The next thing for the connect compiler to do is to write the connection information into the connection topology file in

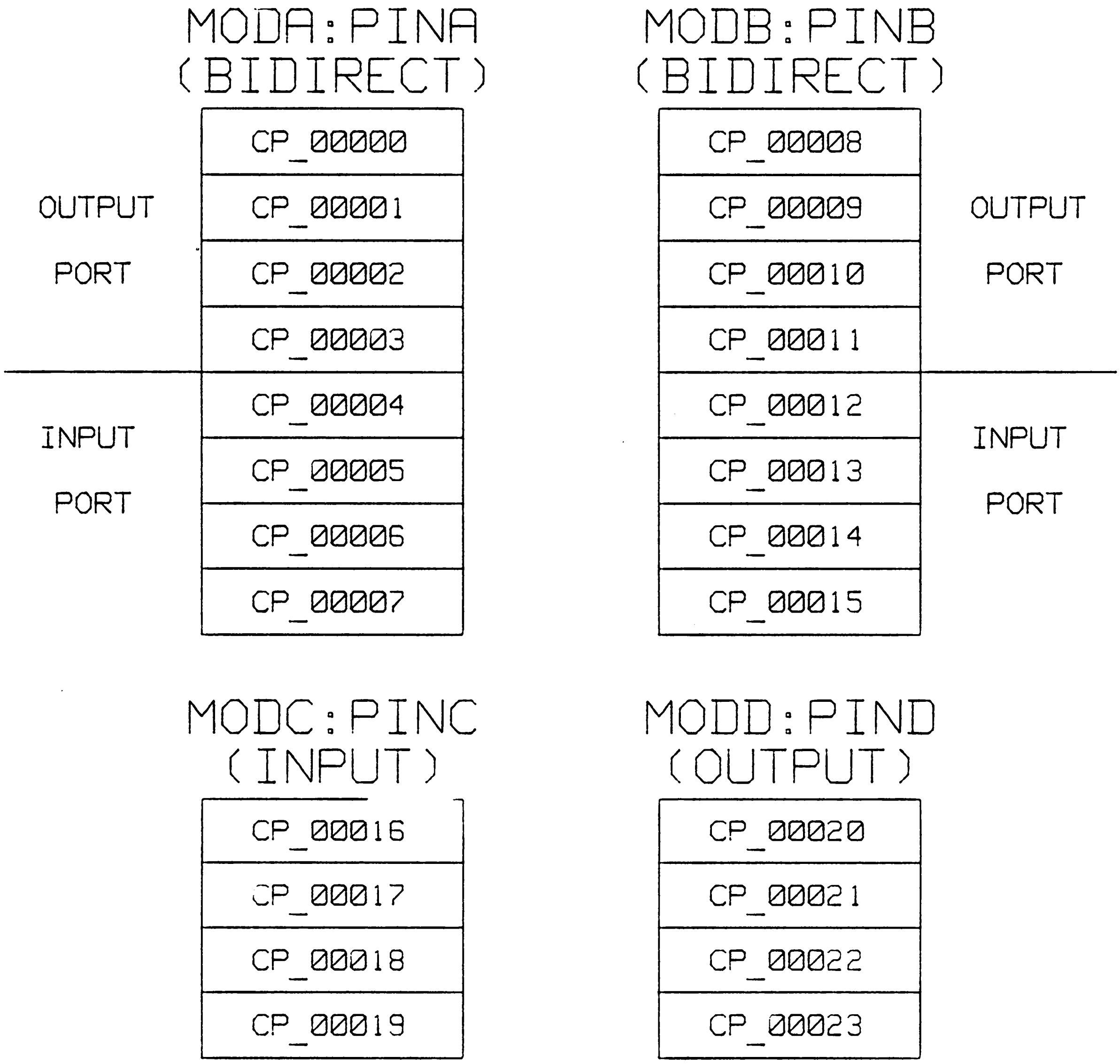

Figure 3. Memory Locations for Pins Before Connection

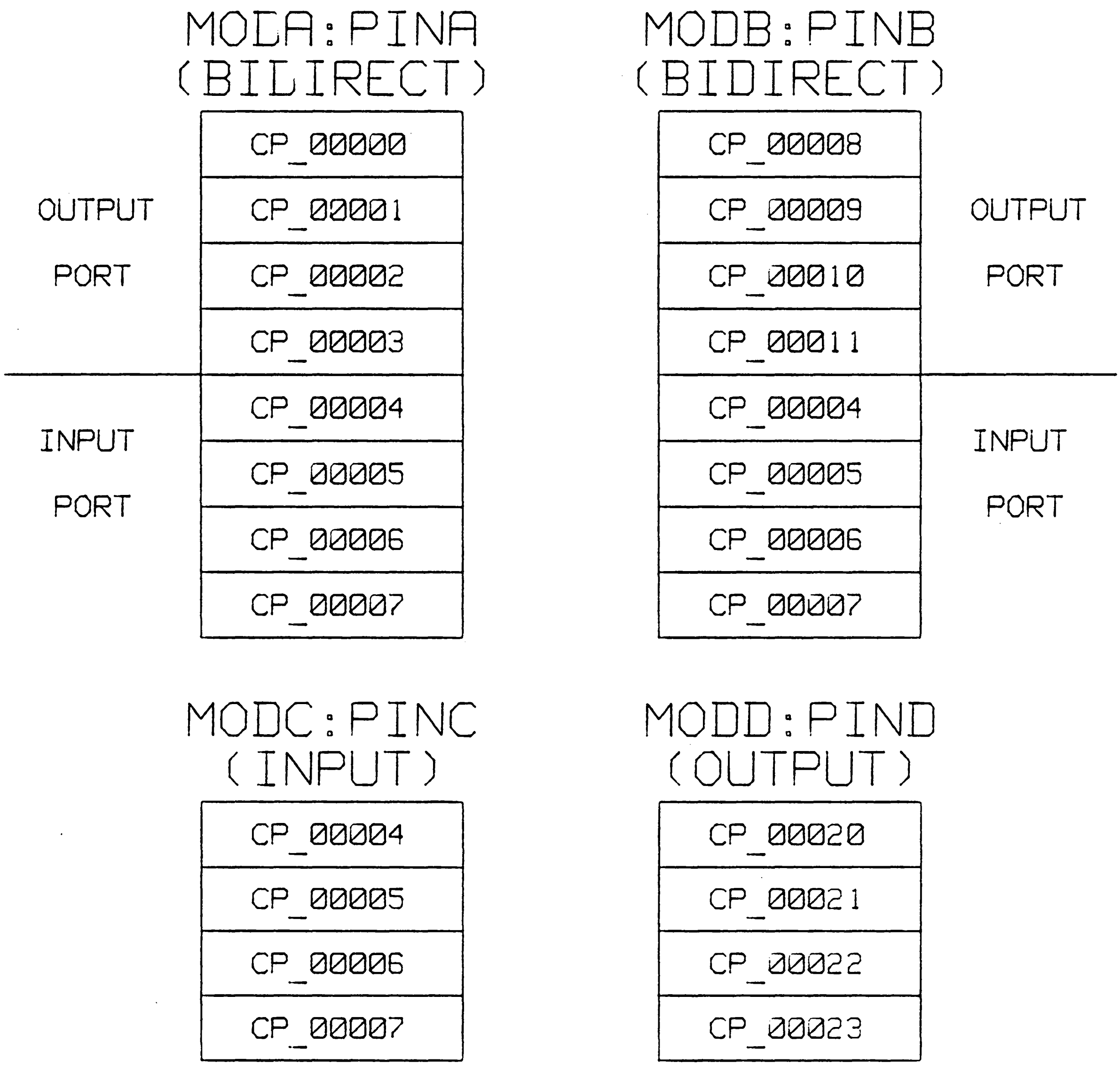

Figure 4. Memory Locations for Pins After Connection

a regular format that can be easily read by the module description compiler. Connection types, pin types, and bus types are converted into integers before they are written into the connection topology file. Figure 5 shows the format of the connection topology file. For each connection, the string "**" is written followed by three connection data - connection type, number of pins in the connection, and bus type. Next, pin definitions in the connection are sequentially generated. Each pin definition starts with 7 integers - length of module name, length of pin name, pin type, LSB number and length of entire vector, and LSB number and length of connected portion. These integers are followed by module name, pin name, and memory locations (optional). For any pin in a normal connection, memory locations are not written. For an output or input pin in a bus connection, the memory locations assigned to the pin vector are written. For a bidirect bus pin, the compiler first generates the memory locations of its output port and then generates those of its input port. The above sequence is repeated until all connections are processed. The file ends with the string "//".

```
**
(connection type)
(number of pins in the connection)
(bus type)


(length of module name) (length of pin name) (pin
type) (LSB number of entire vector) (length of entire
vector) (LSB number of connected portion) (length of
connected portion)

(module name)
(pin name)
(memory locations)


(length of module name) (length of pin name) (pin
type) (LSB number of entire vector) (length of entire
vector) (LSB number of connected portion) (length of
portion)

(module name)
(pin name)
(memory locations)


(REPEAT)


**

(REPEAT)


**

(REPEAT)


//
```

Figure 5. Format of the Connection Topology File

3. THE MODULE DESCRIPTION COMPILER

3.1 CONSTRUCTION OF CONNECTION TABLES

The module description compiler reads the connection information from the connection topology file whenever it compiles each individual module in the simulated system. As explained in chapter 2, the generated connection information covers the whole interconnection in the system. However, to a specific module, all connection information is not useful. As the number of modules increases, the ratio of the necessary portion to the whole information will be much decreased. Therefore, the module description compiler should extract the information needed by the specific module being compiled from the whole connection information. This scheme will much save the simulation time and the memory space at runtime.

The module description compiler constructs two tables, the normal connection table and the bus connection table. As mentioned in section 2.4, the connection topology file contains a sequence of connection definitions. The first information that is written in each connection definition is its type (figure 5). Consequently, the module description compiler can determine the type of each connection without reading the remaining connection data. Any TYPE 1 or TYPE 2 connection becomes a candidate that can be put into the normal connection table. Any TYPE 3 connection becomes a candidate that can be put into the bus connection table. This section describes the algorithm of constructing the two tables.

3.1.1 THE NORMAL CONNECTION TABLE

Before discussing the algorithm of constructing the normal connection table, it is necessary to determine what kind of connections are inserted in it. As an example, let us consider the following connection statement.

```
A: siga [0|4,0|4] OUTPUT,
B: sigb [0|4,0|4] INPUT,
C: sigc [0|4,0|4] INPUT
```

It represents that a 4-bit output pin "siga" of module A is connected to "sigb" of module B and "sigc" of module C. If "siga" changes its value, the values of "sigb" and "sigc" are automatically updated to the new value. As far as this connection is concerned, the module A is the "master" module ("siga" is the source pin in the connection) and the modules B and C are the slave modules. Since it is obvious that neither module B nor module C can invoke any event block in other modules in the connection, the connection information is not put into the normal connection table when the module B or the module C is compiled. However, when the module A is compiled, it is put into the normal connection table because "siga" has the possibility of invoking event blocks in other modules. (it might not invoke any event block in other modules, but the module description compiler has no way of knowing it during the compilation of module A). Generally speaking, the normal connection table contains every normal connection definition whose source signal resides in the module being compiled.

As mentioned in chapter 2, there are two rules of describing normal connections:

- The first pin in a TYPE 1 connection represents an external source signal.
- The first pin in a TYPE 2 connection must have the type of OUTPUT.

The above two rules can be summarized in a single statement - the first pin in a normal connection is certainly a source signal. This information helps the module description compiler determine whether to put a normal connection into the normal connection table or not.

When the module description compiler encounters a TYPE 1 or TYPE 2 connection, it first compares the module name of the first pin with the name of the module being compiled (it knows that the first pin is a source). If two module names are unmatched, the compiler skips the remaining connection data and begins to process the next connection definition. If they are matched, the compiler puts the pin name and the length of the pin name into an entry of the normal connection table. Then the compiler makes a linked list that includes all pins other than the source pin. Since each cell of the linked list contains pin variables in other modules, it contains additional fields to store the module name and the length of the module name. Finally, the linked list is attached to the entry of the table where the source pin is stored. Figure 6 shows the data structure of a normal connection table whose first entry corresponds to the connection statement in this section.

3.1.2 THE BUS CONNECTION TABLE

The basic data structure of the bus connection table is same as that of the normal connection table. The only difference is that the bus connection table contains much more information than the normal connection table. This is because the bus connection table should contain all connection information needed to implement bus functions. Since all pins in a bus connection are used in bus functions, any bus connection which contains a pin that resides in the module being compiled is put into the bus connection table.

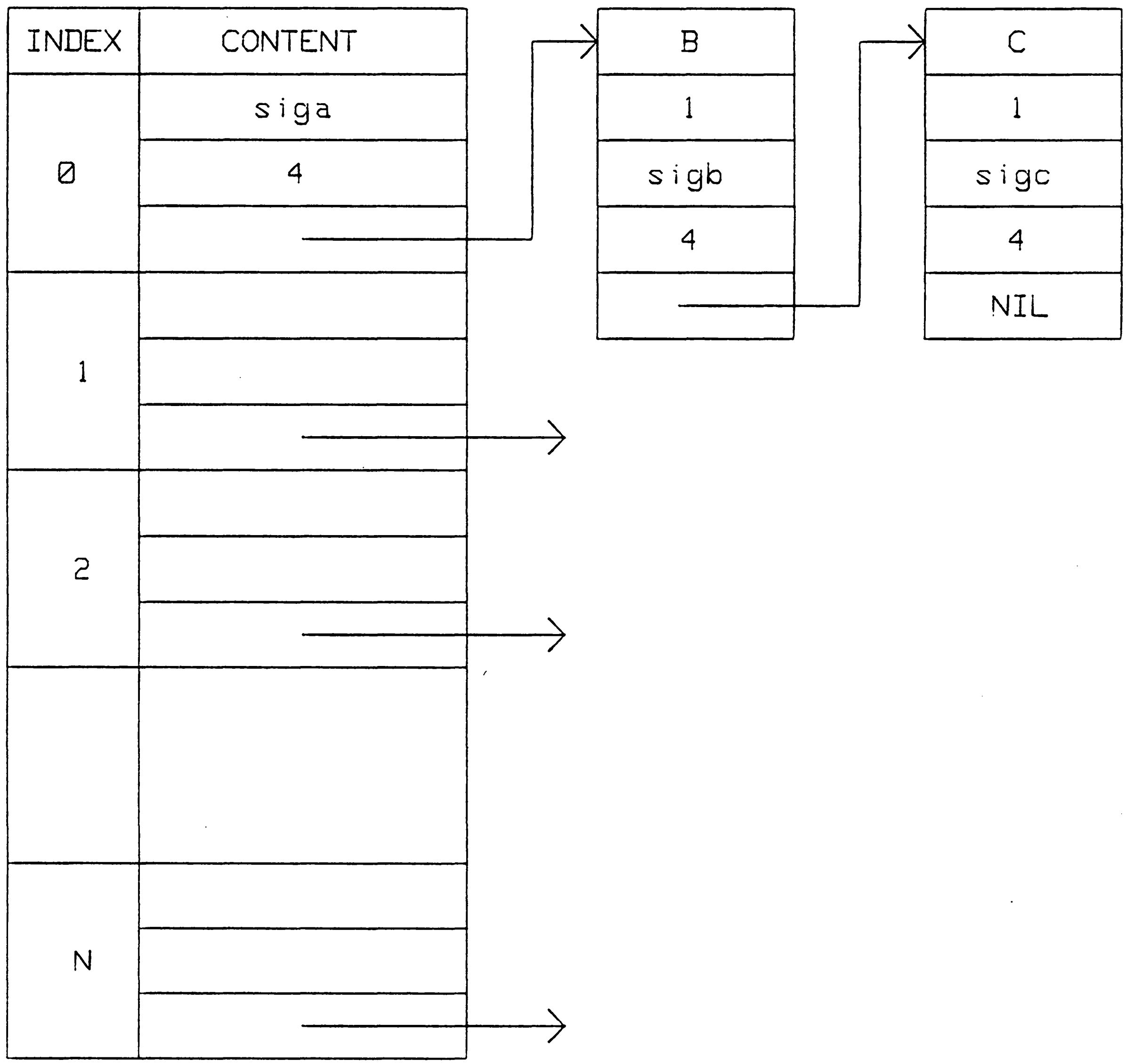

Figure 6. Data Structure of A Normal Connection Table

When the module description compiler recognizes a TYPE 3 connection, it sequentially reads the pin definitions in the connection. For each pin definition, the compiler first compares the module name of the pin with the name of the module being compiled. If they are not matched, the compiler constructs a cell and puts the pin definition into the cell. If the cell is the first one, it creates a linked list. Otherwise, the cell is attached to the built-in linked list. If the module names are matched, the compiler puts the pin definition into an entry of the bus connection table. This sequence is repeated until all pins in the connection are processed. If a pin that resides in the module being compiled is found during the sequence, the compiler attaches the linked list to the entry of the table. Otherwise, the compiler throws out the linked list, i.e., nothing is put into the table.

Each entry of the bus connection table consists of the following fields:

1. the bus type (1: BUSAND, 2: BUSOR)
2. the pin type (1: INPUT, 2: OUTPUT, 3: BIDIRECT)
3. the pin name
4. the length of the pin name
5. the LSB number of the entire vector
6. the length of the entire vector

7. the LSB number of the connected portion of the pin vector
8. the length of the connected portion of the pin vector
9. the memory locations where the values of the pin bits of an input or output pin vector are stored.
10. the memory locations where the values of the pin bits of the output port of a bidirect pin are stored.
11. the memory locations where the values of the pin bits of the input port of a bidirect pin are stored.
12. the pointer to the head of the linked list.

It is notable that there are three fields that can store memory locations. For an input or output pin, the field 9 is used. For a bidirect pin, the fields 10 and 11 are used. Such a data structure is known as a "variant record" [10].

Each cell of a linked list has the same fields as an entry of the bus connection table except that it has additional fields to store the module name and the length of the module name and it does not contain the field to store the type of the bus connection.

Figure 7 shows the data structure of a bus connection table. It is assumed that the first entry of the table is created when the compiler compiles the module MODA and reads the following connection statement.

```
MODA: pina [0|2,0|2] BIDIRECT: BUSAND,
MODB: pinb [0|2,0|2] OUTPUT: BUSAND,
MODC: pinc [0|2,0|2] INPUT: BUSAND
```

3.2 DECLARATION OF CONNECTED PINS

In the GSP2 language, all pins (connected or not) within a module must be declared in the block PINS of the corresponding module description file. According to the syntax of the pin declaration block [9], a pin is defined by its name, size, and type. Since the connected pins that have been declared in the connection description are declared again, the module description compiler checks if both declarations are the same. If they are not the same, the compiler outputts an error message. If they are the same, the compiler uses the type of the pin to determine the number of memory references to be generated. For an input or output pin, it generates a single memory reference. For a bidirect pin, it generates two memory references - one for the output port and the other for the input port. Since pin vectors are stored in a "one bit per byte" scheme, the memory reference for a pin is the address of a list of addresses to the individual bytes for the pin. The memory reference is actually the address where the address to the LSB of a pin vector is stored. This scheme implies that one level of indirect addressing (indexed by bit

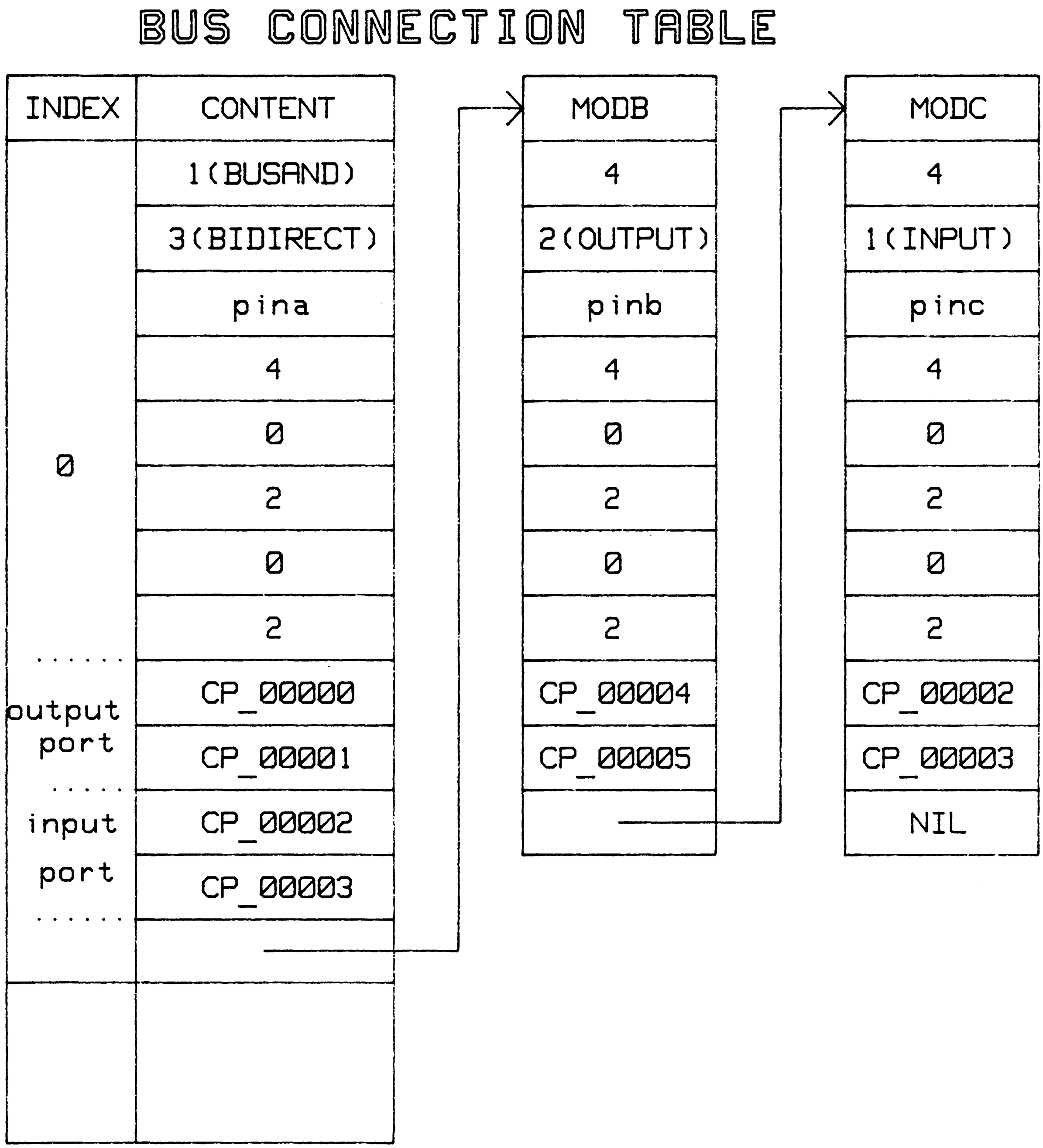

Figure 7. Data Structure of A Bus Connection Table

number from the memory reference) is used to access each bit. Therefore, the memory reference, the LSB number, and the length are needed to access a pin vector. The module description compiler puts all data related to the pin vector into an entry of the symbol table where all variables are to be stored. Figure 8 shows the indexed indirect addressing scheme for a bidirect pin BUSA[0|4]. It is assumed that the pin is included in a bus connection that has the bus attribute BUSAND. Thus, two memory references are assigned to the pin and every bit of the pin vector is initialized to "true_byte" (11111111).

Up to this point, the pins within the module being compiled have been declared. However, for a bus function which is actually an AND or OR function, the bus pins outside the module should be also declared. Since they cannot be declared by the modeler, they are internally declared by the compiler. As discussed in the previous section, each entry of a bus connection table contains the definition of a pin within the module and a pointer to a linked list that consists of the other pins in the connection. Consequently, the compiler can declare all the pin variables involved in a bus function by putting the pin definitions in the linked list into the symbol table. However, there is a problem to be considered. Since each entry of the symbol table has no field of storing the module name in a pin definition, there

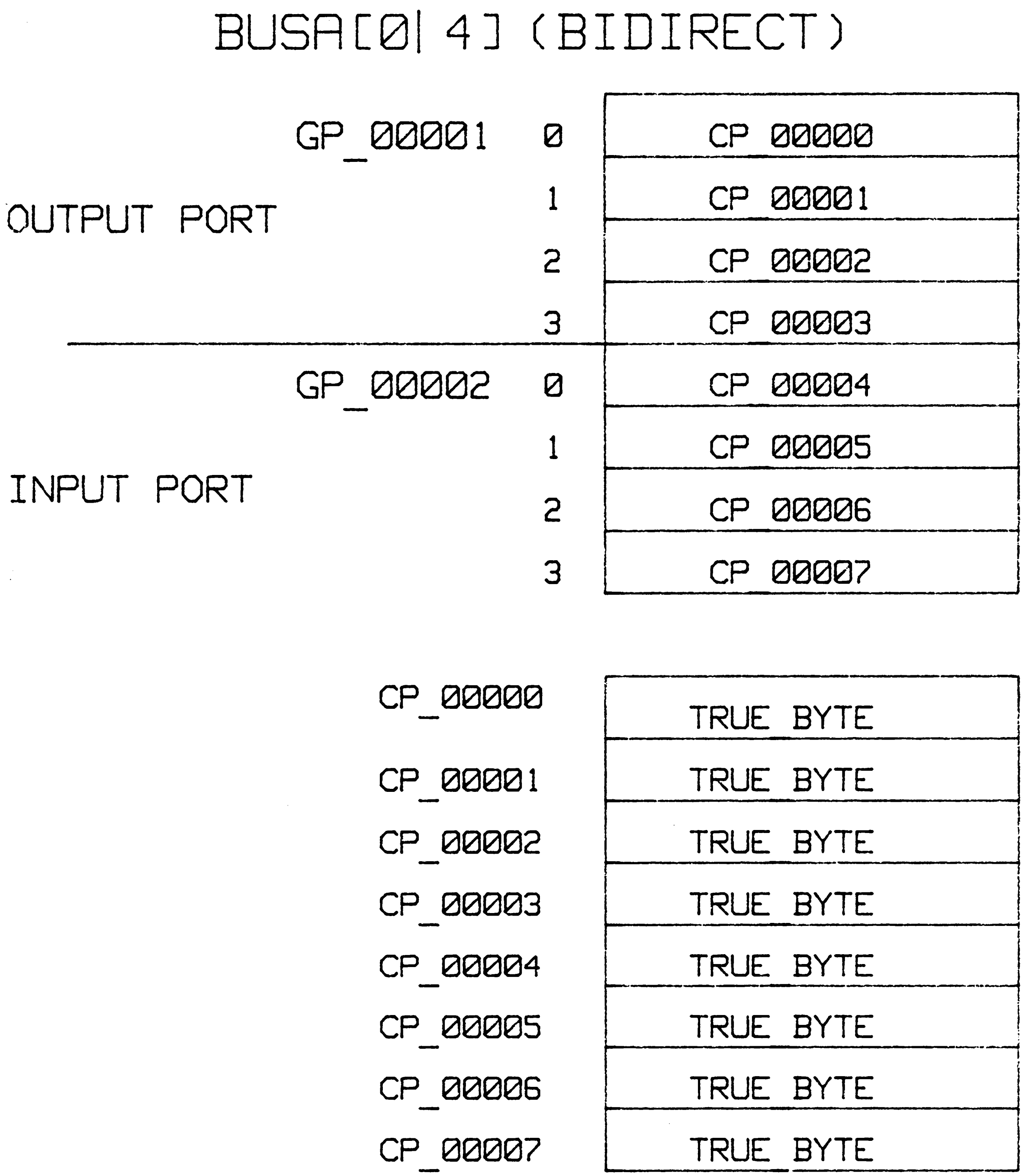

Figure 8. Indexed Indirect Addressing Scheme for Pin Variables

is no method of distinguishing two pins that have the same pin name and reside in different modules. It imposes a restriction on the modeler when he writes a bus connection statement in the connection description file, i.e., any bus pin should have a unique pin name (section 2.2).

3.3 MULTIMODULE INVOCATION MECHANISM

In each individual module description file, there exist one or more event blocks in which the behavior and the timing of a chip (or a module) are described. An event block definition has the following form:

```
EVENT event_block_name ON (event expression);
        (block of executable statements)
```

The event expression describes the condition on which the event block is invoked. In the GSP2 system, an event is defined as a signal change with time delay. When an event causes the event expression to be "true", the statements within the block are sequentially executed, which usually induces other signal changes. These events could invoke other event blocks in the module or event blocks in other modules. In fact, an event invokes every event block whose event expression becomes true due to the signal change. Therefore, there should be a mechanism through which every global variable can access the names of the event blocks that include the variable in their event expressions.

The event activation table is a data structure used for the purpose. Each entry of the table contains a pointer to a linked list that contains the names of the event blocks related to a specific variable. Whenever the module description compiler encounters a variable in an event expression, it attaches the name of the event block to the linked list for the variable (if it is the first event block, a linked list is created and attached to the next available entry of the table). When the compiler finishes processesing all event blocks in a module, each linked list will contain the names of the event blocks related to a specific variable. For a non-pin variable or an unconnected pin variable, it becomes the final linked list. However, for a connected pin, it is not complete because it does not contain the names of the event blocks in other modules. Since the original GSP2 considers it as the final linked list, it fails to access the event blocks in other modules. It is the main reason that the original GSP2 cannot be a multimodule simulator.

Since each individual module description file is compiled separately, the names of the event blocks in other modules cannot be directly accessed during the compilation phase. It can be done only after all the modules in the simulated system are compiled and linked together. Since each entry of the symbol table has the field to store the index into the event activation table, the simulator can access any event block if both the module name and the pin name are given. Therefore, the module description compiler constructs the final activation lists for connected pins by attaching the names of the extramodule pins. Which pin names to attach to a linked list depends on the type of the connection.

1. In a TYPE 1 connection, the first pin in the corresponding connection statement is considered as an external source signal. Therefore, other pin names are attached to the linked list for the representative pin.
2. In a TYPE 2 connection, the OUTPUT pin is the only source signal in it. Therefore, the INPUT pins are attached to the linked list for the OUTPUT pin.
3. In a TYPE 3 connection, there are more than one source signal in it. For example, let us consider a bus connection that consists of two bidirect pins (figure 9). Since a bidirect pin can be used as both a source and a destination, the linked list for one bidirect pin will contain the name of the other pin and vice versa. It makes an endless loop as the following sequence.
   a. The first cell (event block EB1) in the linked list for "pina" is invoked.
   b. Since the next cell contains a pin name (MODB, pinb), the simulator accesses the linked list for "pinb".

c. The first cell (event block EB2) in the linked list for "pinb" is invoked.

d. Since the next cell contains a pin name (MODA, pina), the simulator accesses the linked list for "pina".

e. Go to step a.

Therefore, any connected bus pin should not be included in an event expression. Although it becomes a disadvantage of the GSP2 system, it will not much affect the modeling of digital systems. We can find such an example in an interconnection (data bus) between a CPU and a RAM. In this case, the signal propagated through the bus does not invoke any operation for itself. Only when the read/write signal is applied to a RAM, "memory read" or "memory write" can be accomplished.

Each cell of an event activation list contains the following fields:

1. the boolean value that determines the type of information to be stored in the remaining fields. If it is true, the field 2 is meaningless. Otherwise, the fields 3 and 4 are meaningless.
2. the name of an event block.
3. the name of a module.
4. the name of a pin.
5. the pointer to the next cell.

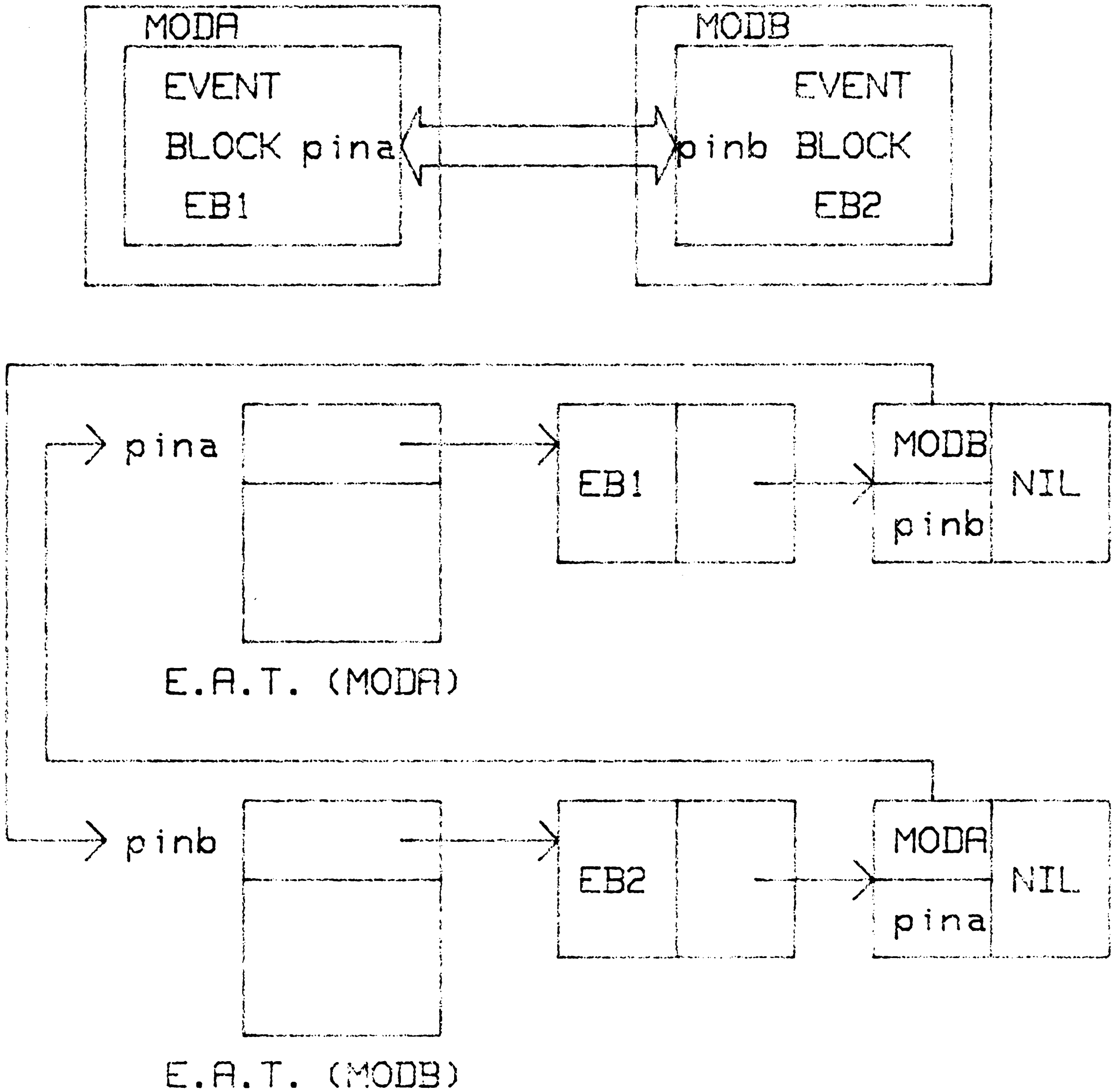

Figure 9. Event Invocation Mechanism for Bidirect Bus Pins

Figure 10 shows a schematic diagram of a modeling of a digital system. In the figure, there are three interconnections whose corresponding connection statements will have the following form (for the convenience, the size and the connection range of the pin vectors are not included).

```
MODA: a INPUT,    MODB: f INPUT;
MODA: c OUTPUT,   MODB: g INPUT;
MODA: e OUTPUT,   MODB: i INPUT
```

In the first connection (TYPE 1), the first pin "a" is the representative pin of it. The remaining two connections are TYPE 2. Figure 11 shows the event activation tables constructed during the compilation of each module in figure 10.

3.4 IMPLEMENTATION OF BUS FUNCTIONS

3.4.1 BUS FUNCTIONS

Before discussing the implementation of bus functions, it is useful to discuss the real hardware structure of buses. A bus has been already defined as a connection that has more than one source signal in it (section 2.2). A multiple-source bus is an electrical connection used to convey data

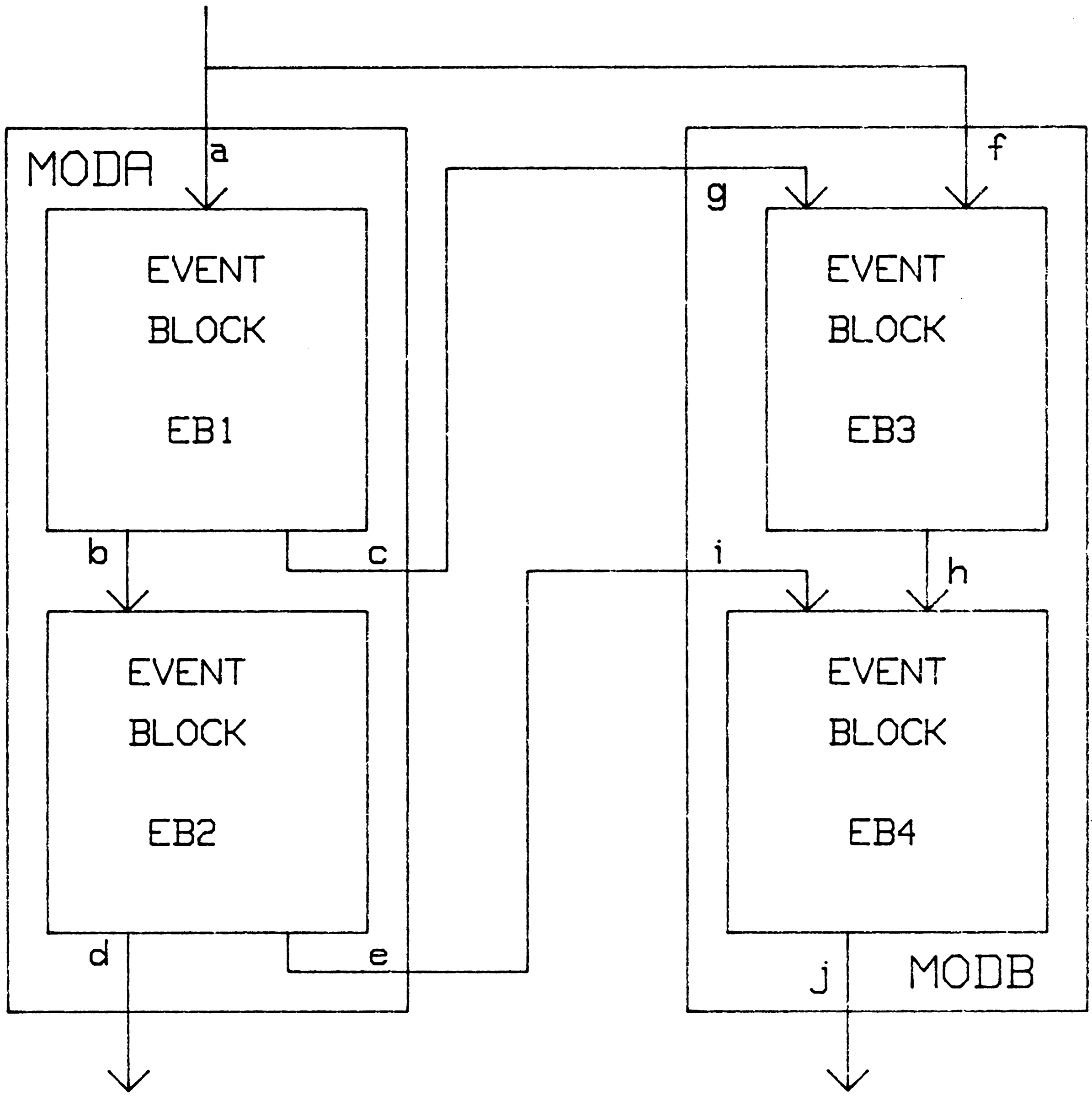

Figure 10. Modeling of A Two-module Digtal System

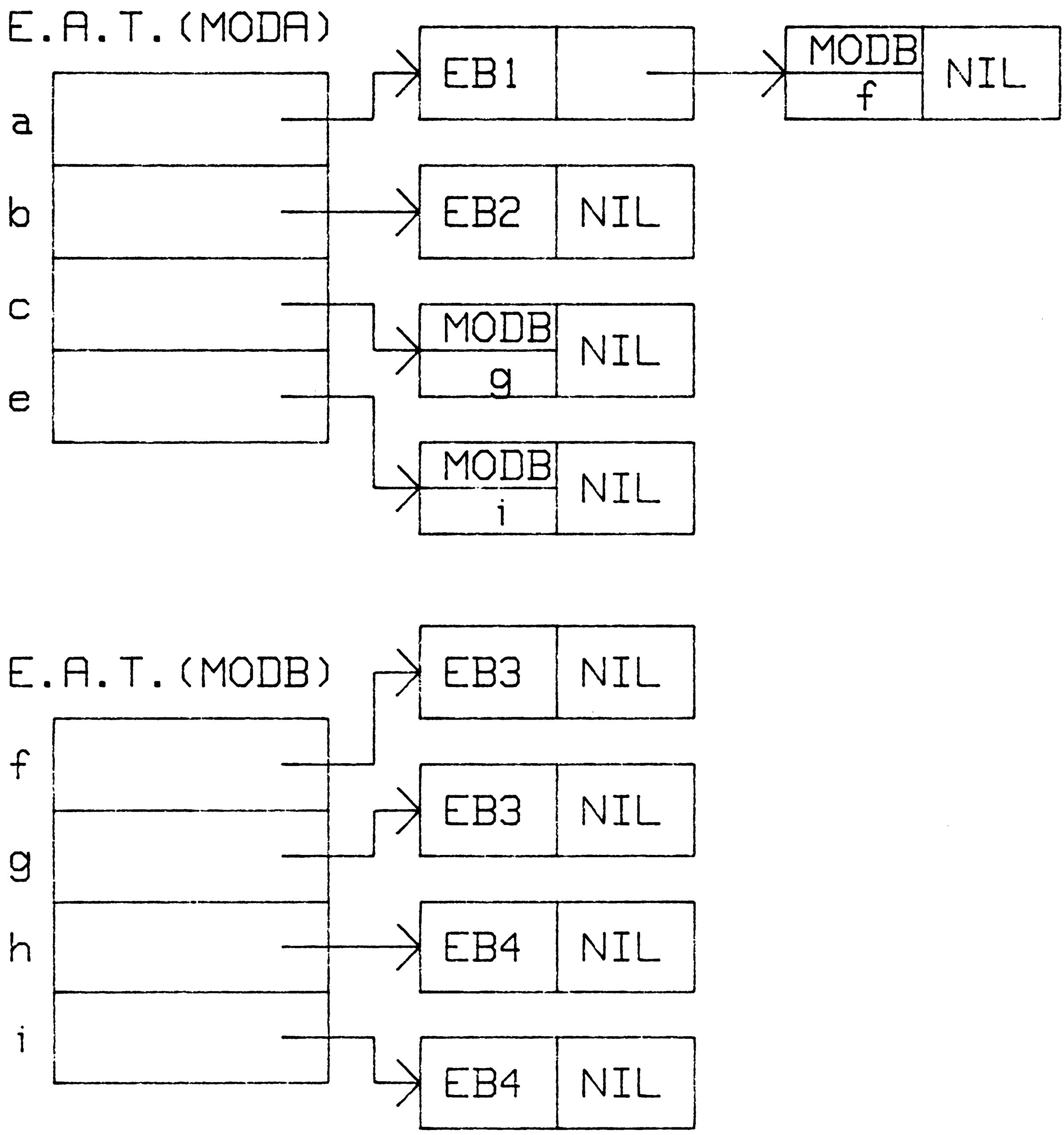

Figure 11. Event Activation Tables

signals which can originate from many sources, but the bus assumes the data values of only one source at a time. The data flow in a bus can be unidirectional or bidirectional. A bidirectional bus can be constructed with bus buffers to control the direction of data flow. Nowadays, there are two types of bidirectional buses, the open-collector bus and the tri-state bus. The open-collector bus performs the bit-by-bit wired-AND (or wired-OR) function among the source data bits. However, the tri-state bus performs no logic function: if one tri-state driver is logic 1 and another logic 0, the bus value becomes in an "unknown" state because the bus value depends on the current limiting safety devices incorporated into the circuits [11].

In the GSP2 system, the module description compiler internally implements the bus functions by generating the codes for AND or OR functions among the source pins. An open-collector bus can be completely modeled by this scheme. However, a tri-state bus cannot be completely modeled because the GSP2 language provides neither the "high impedance" state nor the "unknown" state. When two source signals drive a tri-state bus at a time, the compiler will perform a regular AND or OR function without considering an "unknown state". A high impedance state is simply described by logic 1. Therefore, the modeler should be careful to analyze the simulation result of a digital system that contains tri-state buses.

The compiler performs a bus function as the following sequence.

1. If an output or bidirect pin is encountered on the left hand (destination) side of an assignment statement, the compiler traverses the bus connection table. If the same pin is found in the table, it is considered as a bus pin. The compiler goes to the first entry of the table that has the same pin name.

2. The compiler traverses the linked list attached to the entry to find the arguments for the bus function. Output pins and output ports of bidirect pins become the arguments for the bus function.

3. The compiler checks the type of the bus. If it is BUSAND (BUSOR), it generates the code for an AND (OR) function.

4. The result of the bus function is stored at input pins and input ports of bidirect pins. The compiler actually updates only the first one of the destinations because they are connected together by the connect compiler (refer to figure 7 in section 2.3).

5. The compiler traverses the remaining entries of the bus connection table. If an additional entry that has the same pin name is found, (it is possible if the pin appears in more than one connection statement), the compiler repeats the step 2-4. The sequence is repeated until the end of the bus connection table is reached.

Figure 12 shows the basic concept of the implementation of the bus function.

3.4.2 CODE GENERATION

The code generator module ("prim.pas") in the GSP2 compiler generates VAX-11 MACRO codes to implement the semantics of the bus function. They are written in the module description macro files. The generated code is in the form of calls to the built-in macros. The sequence is as follows:

1. Temporary memory locations are generated for the arguments for the bus function. The macro _vect_source_subr together with the start bit numbers and the connection ranges of the arguments are generated to copy the current values of the arguments to the temporary memory locations.

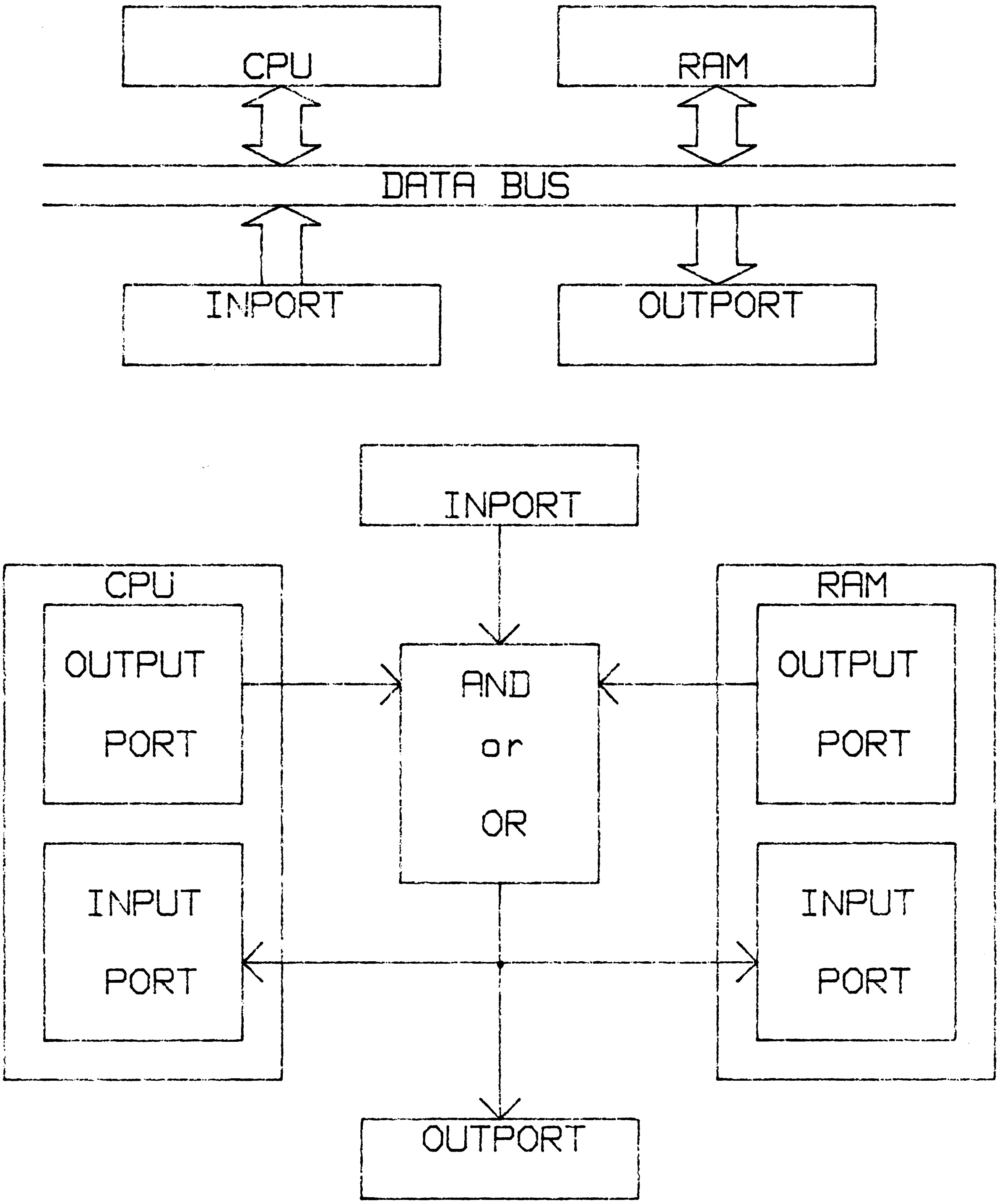

Figure 12. BASIC CONCEPT OF THE BUS FUNCTION

2. The macro _and or _or is written depending on the type of the bus. The temporary memory locations become the formal parameters for the macro.

3. The macro _copy_to_subvect or _propagate_to_subvect is generated to copy the result of the bus function. If the assignment statement that has caused the bus function has no PROP clause (it represents a time delay), _copy_to_subvect is used. In this case, the result is immediately copied to the memory locations of a destination pin. If the statement contains the PROP clause, a temporary memory location is generated to store the result. The temporary memory location will be the formal parameter for the macro _propagate_to_subvect. The result value will be copied to the destination pins during the simulation after the delay time expires.

4. THE SIMULATOR

4.1 OUTLINE OF THE GSP2 LINKER/SIMULATOR

4.1.1 COMPILED-CODE SIMULATION

In the GSP2 system, the source description is translated into VAX-11 machine code to act as run-time routines for the simulator program which is also written in VAX-11 machine code. As mentioned in chapter 1, this technique, known as the compiled-code simulation, is useful to get high simulation speed. The connect compiler generates a macro file ("connect file") which assigns the memory locations where the values of the connected pins are stored. Whenever the module description compiler compiles a chip model, it generates a macro file that contains macro codes corresponding to the chip model and a name file that contains the name of the chip model. Actually, macro files contain macro instructions which are defined in the GSP2 macro library.

Figure 13 shows how the GSP2 linker constructs an executable image file to be run on the VAX. The GSP2 linker invokes the VAX-11 assembler to sequentially assemble the macro files into object files. Whenever a chip model is assembled, the module name merger reads the name file corresponding to the

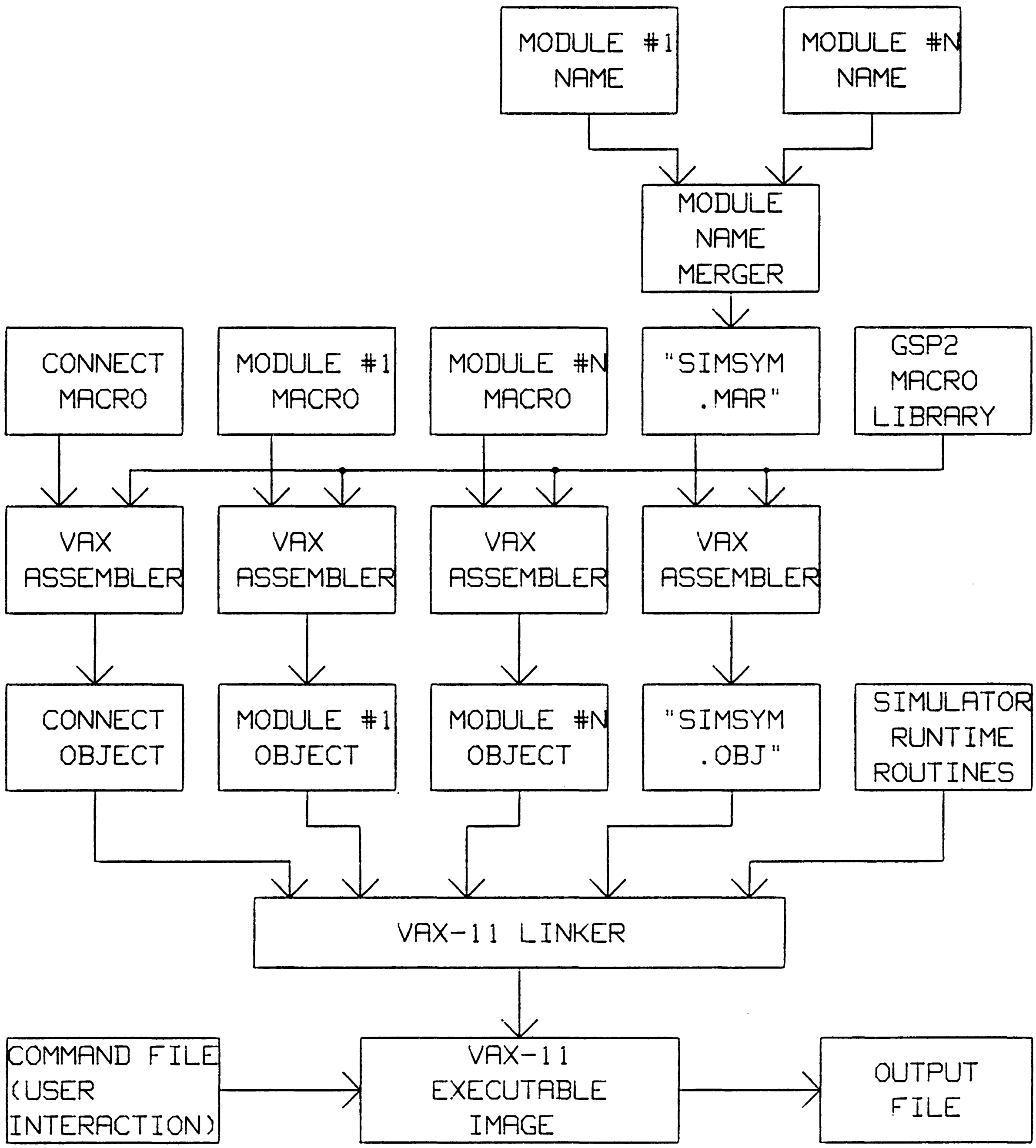

Figure 13. Constrcution of Executable Image File

model and writes the model name into the macro file "simsym.mar". Therefore, after all chip models are assembled, "simsym.mar" will contain the names of all chip models. Since the entry point to each chip model is defined by its name, the file is used as the data structure through which each chip model can be accessed. "Simsym.mar" is also assembled into the object file "simsym.obj". The GSP2 linker then invokes the VAX-11 linker to link the connect object file, the chip model object files, "simsym.mar", and the simulator runtime routines into an executable image file to be run on the VAX. Once the image file is constructed, the simulator runtime routines become the "master" that executes the chip models.

4.1.2 SIMULATOR RUNTIME ROUTINES

The simulator runtime routines are divided into four major modules. These are the monitor, the queue manager, the command interpreter, and the display module. The various modules of the simulator are controlled by the monitor module. The queue manager is responsible for entering and removing events from two time queues, the forward time queue (simply time queue) and the history queue. The time queue is a data structure which holds future events in the order of simulation time. The history queue remembers a fixed amount of past events. The command interpreter is invoked by the monitor if necessary. It receives user input in the form of GCL (GSP2 command language) commands, parses them, and interacts with the simulator's runtime symbol table or the queue manager, depending on the command. The display module is used to output information in various format to the user's terminal and/or the simulation output file.

4.1.3 PROGRESSION OF GSP2 SIMULATION

The monitor module considers each event block code as a process and controls the progression of simulation by invoking, suspending, and terminating the processes. The monitor module processes the events contained in the time queue in the order of simulation time. In the GSP2 system, the following three types of events are defined.

1. Signal Change Event - if an assignment statement with PROP clause which represents a delay time is executed, the signal change is put into the time queue (simulation time = current simulation time + delay time).
2. Reactivated Event - if a WAIT statement is executed, the address of the next statement is put into the time queue (simulation time = current simulation time + waiting time).

3. Break Event - if the GCL command BREAK is encountered, the event is put into the time queue with the simulation time specified by the user.

If a signal change event comes out of the time queue, the monitor updates the variable to the new value and executes every event block whose event expression becomes true because of the signal change. If a reactivated event comes out of the time queue, the monitor executes the event block that contains the WAIT statement from the stored entry point (see section 4.5). If a break event comes out of the time queue, the simulation halts and the command interpreter module is called to accept GCL commands from the terminal.

The invoked event block(s) usually contain one or more assignment statements with the PROP clause or a WAIT statement, which will in turn schedule future events at the time queue. This sequence is repeated until the time queue is empty or a break event is encountered.

4.2 SIMULATOR DATA STRUCTURES

4.2.1 THE RUNTIME SYMBOL TABLE

This data structure is built by the monitor module in its initialization phase. It is implemented as a binary tree which is ordered by the names of the simulated modules. Each node in this tree has three fields in it.

1. the pointer to the node's left child.
2. the pointer to the node's right child.
3. the module name which is actually the entry point to the module.

Each module contains five data structures, the variable symbol tree, the runtime event activation table, the event block symbol tree, the runtime bus connection table, and the procedure symbol tree. In this section, the first four data structures which are related to the sections 4.3 - 4.5 will be explained.

4.2.1.1 THE VARIABLE SYMBOL TREE

The variable symbol tree contains information on each variable in the module. It is ordered by the variable name. Each node in this tree has following fifteen fields.

1. the pointer to the node's left child.
2. the pointer to the node's right child.
3. a byte that indicates where the value of the variable is stored. If it is "true_byte", the field 5 is used. If it is "false_byte", the field 4 is used.

4. the address where the value of the variable other than the input port of a bidirect pin is stored.
5. the address where the value of the input port of a bidirect pin is stored.
6. the variable name, 31 character string.
7. a byte which specifies the data type of the variable.
8. a byte containing the MSB number of a vector.
9. a byte containing the LSB number of a vector.
10. the number of words in a memory variable.
11. a byte which determines whether the signal change on the variable is to be displayed in the output file or not.
12. the base address of the module's event activation table.
13. an index into the event activation table.
14. the base address of the bus connection table
15. the name of the module that the variable belongs to.

4.2.1.2 THE RUNTIME EVENT ACTIVATION TABLE

The runtime event activation table is an array of linked lists. Its basic structure is same as the compile-time event activation table (see section 3.3). However, each cell of the linked lists has additional fields (5-7) to store the temporary values which are used in the implementation of the SETUP function (see section 4.4). There are eight fields in each cell.

1. a byte that determines the type of information to be stored in the remaining fields. "False_byte" indicates that the cell contains the information of an event block that resides in the same module. Therefore, only the fields 2 and 8 are meaningful. "True_byte" indicates that the cell contains the information used in accessing the event blocks in other modules. Therefore, the fields 3-8 are meaningful.
2. a pointer to a node of the event block symbol tree.
3. the module name, 31 character string.
4. the name of a connected pin, 31 character string.
5. the old value of the connected pin.
6. the new value of the connected pin.
7. the pointer to the variable symbol node for the connected pin.
8. the pointer to the next cell.

4.2.1.3 THE EVENT BLOCK SYMBOL TREE

The event block symbol tree is ordered by the name of the event block. Each node in this tree has five fields.

1. the pointer to the node's left child.
2. the pointer to the node's right child.
3. the starting address of the event block code.
4. the address of the event block's local stack pointer.

5. the name of the event block, 31 character string.

The purpose of each of these fields will be discussed in section 4.5 where the implementation of the WAIT statement is discussed.

4.2.1.4 THE RUNTIME BUS CONNECTION TABLE

The runtime bus connection table is implemented as a two-level linked list (figure 14). Each cell of the first level linked list contains the pointer to the next cell and the pointer to the head cell of the second level linked list. Each second level linked list contains the destination pins involved in a bus connection (the reason why the source pins are not included will be explained in section 4.4). Each cell of the second level linked list has the following six fields. The fields 3-5 are used in the implementation of the SETUP function (see section 4.4).

1. the name of the module where the bus pin resides.
2. the name of the bus pin.
3. the old value of the bus pin.
4. the new value of the bus pin.
5. the pointer to the variable symbol node for the bus pin.
6. the pointer to the next cell.

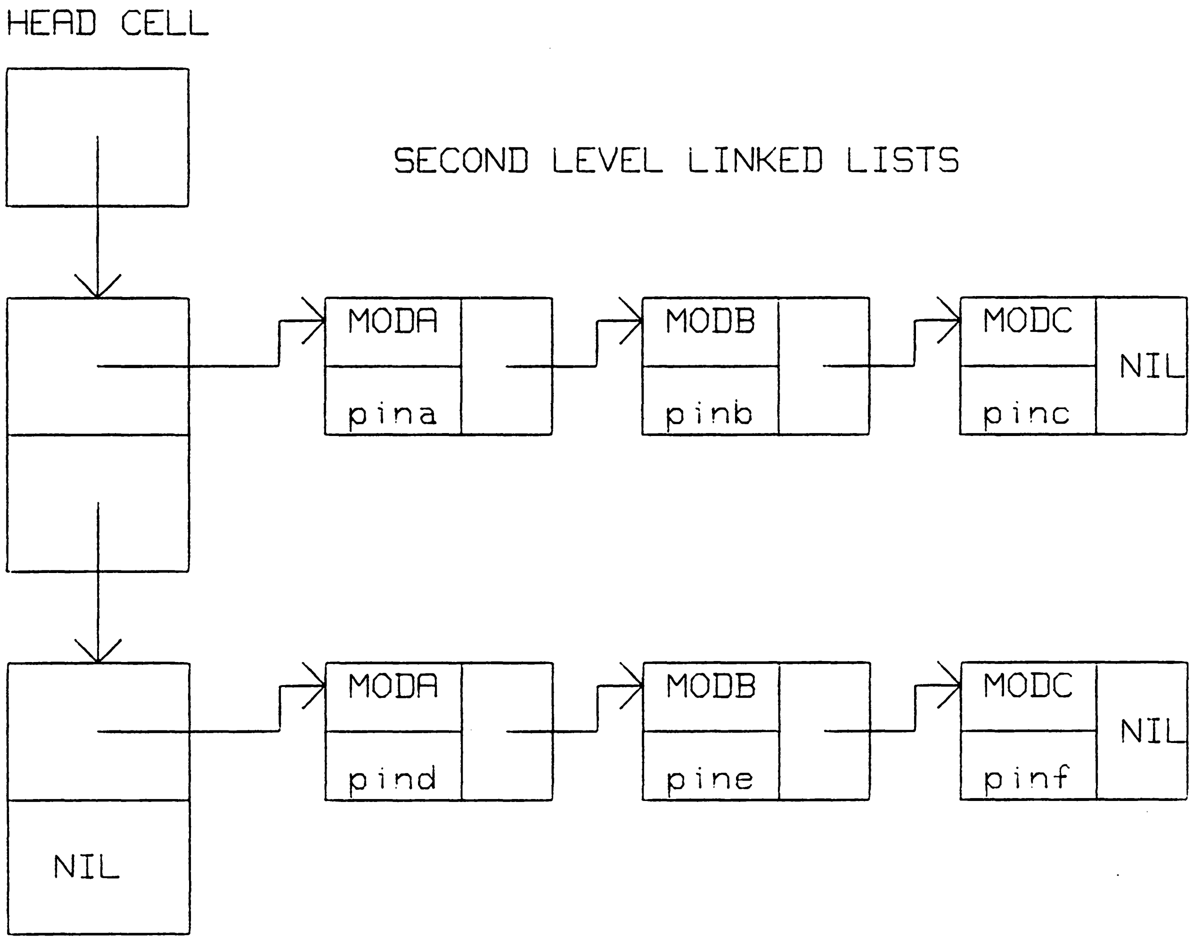

Figure 14. Data Structure of the Runtime Bus Connection Table

4.2.2 THE TIME QUEUE

This data structure is a time-ordered list of events. Actually, it is implemented as a binary tree that is ordered by the simulation time. It has following thirteen fields.

1. the pointer to the node's left child.
2. the pointer to the node's right child.
3. a byte that indicates the type of the event. As mentioned in section 4.1.3, there are three types of events. For a signal change event, the fields 4-9 are used. For a reactivated event, the fields 4 and 10-14 are used. For a break event, only the field 4 is used.
4. the simulation time the event is to occur.
5. the address where the value of the variable is stored.
6. the pointer to the variable symbol tree node for the variable.
7. the new value the variable is to be given.
8. the LSB number of a vector subrange.
9. the length of a vector subrange.
10. the program counter (see section 4.5).
11. the stack pointer (see section 4.5).
12. the frame pointer (see section 4.5).
13. the argument pointer (see section 4.5).

4.3 MULTIMODULE INVOCATION SCHEME

The monitor module controls the progression of simulation by invoking, suspending, and terminating the execution of the event blocks in the system being simulated. Figure 15 is a simplified flow chart which shows how the monitor controls the progression of the simulation. In this section, it is discussed how the monitor module finds all event blocks to be invoked by a signal change event.

When a signal change event comes out of the time queue, the monitor should invoke every event block whose event expression becomes true because of the signal change. The monitor uses the runtime event activation tables for this purpose. There exists only one event activation table for each module (chip model). Each entry of the table has a pointer to a linked list, which contains the information about the event blocks to be invoked by a specific variable within the module. Each cell of a linked list contains one of the two kinds of information. If the event block to be invoked is in the same module that the variable belongs to, it contains the pointer to a node of the event block symbol tree. Otherwise, it contains the name of a connected pin and the name of the module that the pin belongs to. The monitor can access the event blocks in other modules by using these two data items.

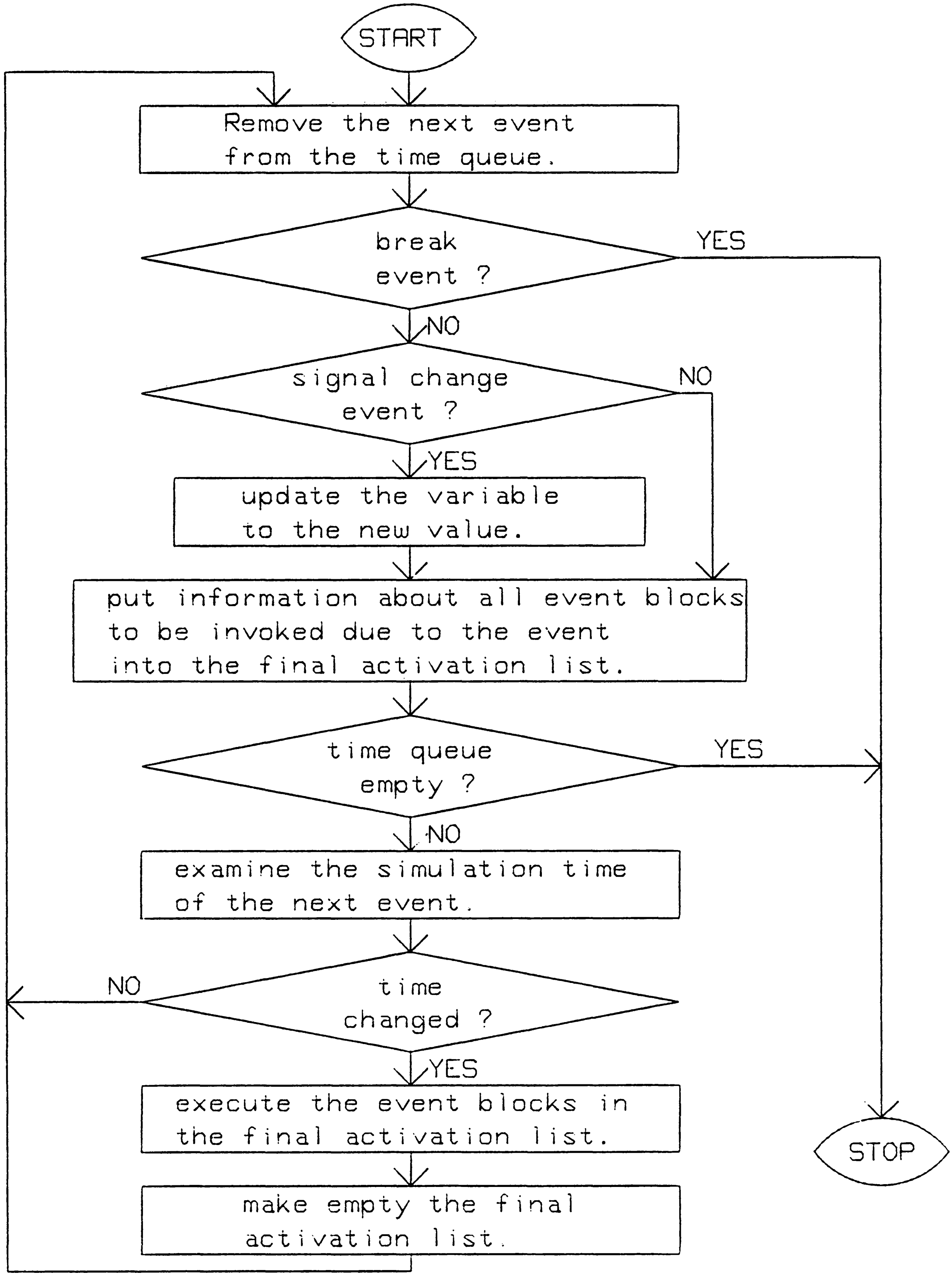

Figure 15. The Simulation Loop (for processing events)

The monitor performs the function using the following algorithm.

1. If a signal change event comes out of the time queue, access the variable symbol node for the variable (the pointer to the node is stored in field 6 of the time queue entry for the event).
2. Read the index (field 13 of the node) into the event activation table within the module that the variable belongs to. If it is a negative number (it means that there is no event block to be invoked due to the signal change), the algorithm terminates. If it is 0 or a positive number, compute the address of a table entry by adding the index to the base address (field 12 of the node) of the table.
3. Go to the first cell of the linked list attached to the table entry.
4. If field 1 of the cell contains "false_byte", access a node of the event block symbol tree (the pointer to the node is stored in field 2 of the cell). Then, read the starting address (field 3 of the node) and the stack pointer (field 4 of the node) of the event block and store them in the final activation list.[1] If field 1 of

---

[1] The final activation list is the data structure that contains the information about all event blocks to be invoked at the same simulation time.

the cell is "true_byte" (it happens only when the event is a signal change on a connected source pin), the cell contains the name of a destination pin connected to it and the name of the module that the pin belongs to. In this case, the following sequence is used.

a. Traverse the module tree (see section 4.2.1) until a node of the same module name is found. Access the root of the variable symbol tree within the module.

b. Traverse the variable symbol tree until a node of the same variable name is found.

c. Read the index into (field 13 of the node) the event activation table within the module that the variable belongs to. If is a negative number, the algorithm terminates. Otherwise, compute the address of a table entry by adding the index to the base address (field 12 of the node) of the table.

d. Go to the first cell of the linked list attached to the table entry.

e. Since a destination pin cannot invoke any event block in other modules, the field 1 of the cell must be "false_byte". Store the starting address of the event block in the final activation list.

f. If the cell is the last one in the linked list, go to step 5. Otherwise, go to the next cell.

g. Go to step e.

5. If the cell is the last one in the linked list, the algorithm terminates. Otherwise, go to the next cell.
6. Go to step 4.

Figure 16 shows an example of the multimodule invocation scheme. In the example, the followings are assumed.

- An external source signal is connected to "pina" of "MODA" and "pinb" of "MODB".
- The pin variable "pina" is included in the event expression of an event block within the module "MODA".
- The pin variable "pinb" is included in the event expression of an event block within the module "MODB".

The numbers around the arrows indicate the order of the execution.

4.4 IMPLEMENTATION OF SETUP FUNCTION

The SETUP function is used to monitor the past history of a variable. The SETUP has two arguments. The first argument is the name of a variable whose history is to be monitored. The second argument is an integer specifying how long it is to have been unchanged. Therefore, to implement the SETUP function, the simulator should retain the past values of variables. The history queue is used for this purpose. It

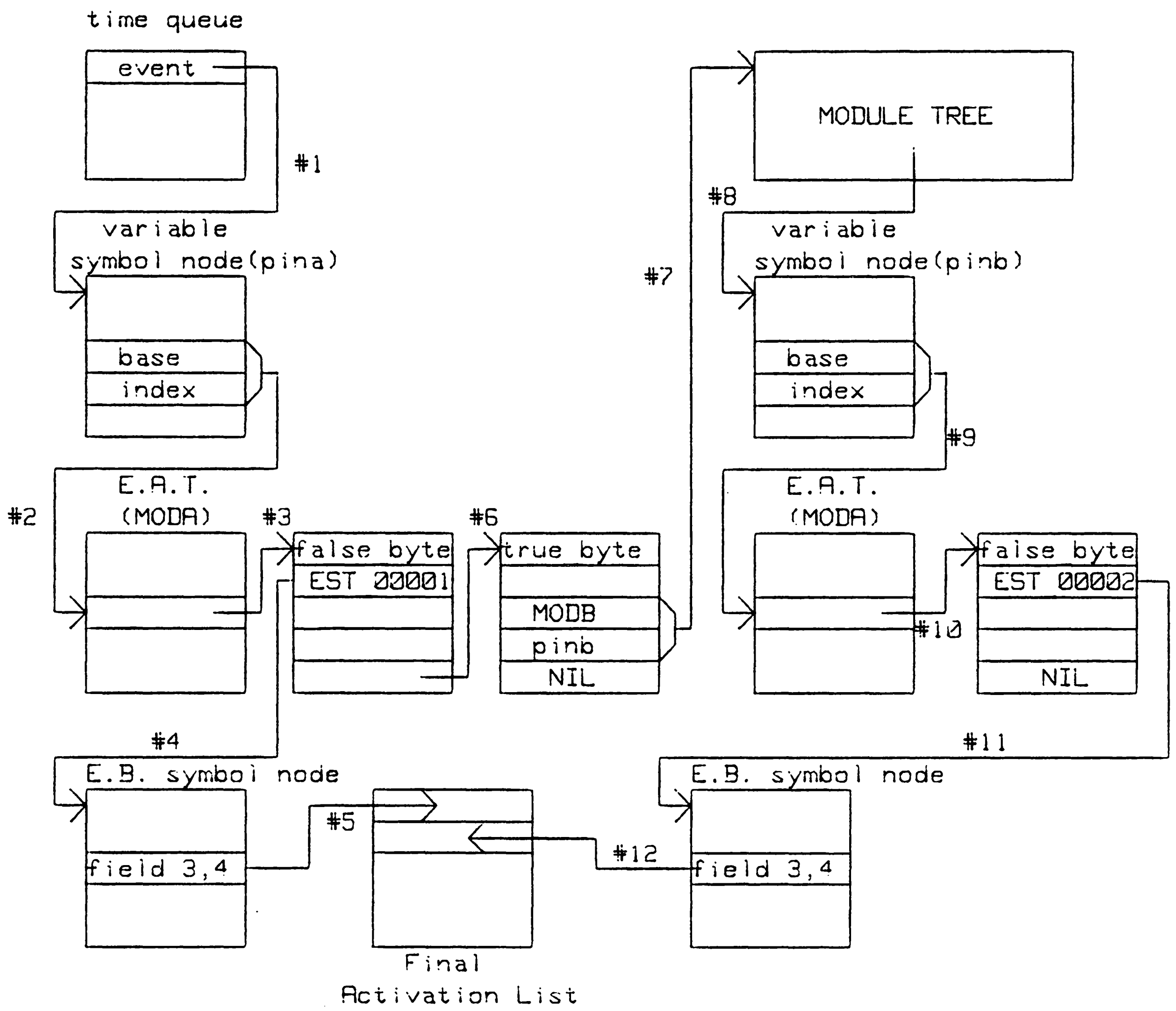

Figure 16. The Multimodule Invocation Scheme.

is implemented as a doubly-linked list [12]. Each cell of the linked list contains four data items, the simulation time, the old and new value of the variable, and the address where the value of the variable is stored. When the SETUP function is encountered during the execution of an event block, the queue manager searches the history queue to check if the variable has been stable for the specified time.

Whenever a signal change event comes out of the time queue, the monitor inserts four data items for the variable into the time queue. As described in section 4.2.2, each entry of the time queue contains the simulation time (field 4), the new value the variable is to be given (field 7), and the address where the variable is stored (field 5). Therefore, the only thing for the monitor to do is to save the old value of the variable before it is updated to the new value. The monitor can easily do the above operation for non-pin variables or unconnected pins. However, when a signal change event on a connected pin comes out of the time queue, the task of the monitor becomes complex. This is because the monitor should obtain data items for all pins connected to the scheduled pin. Figure 17 shows how the monitor obtains the data items and inserts them into the time queue.

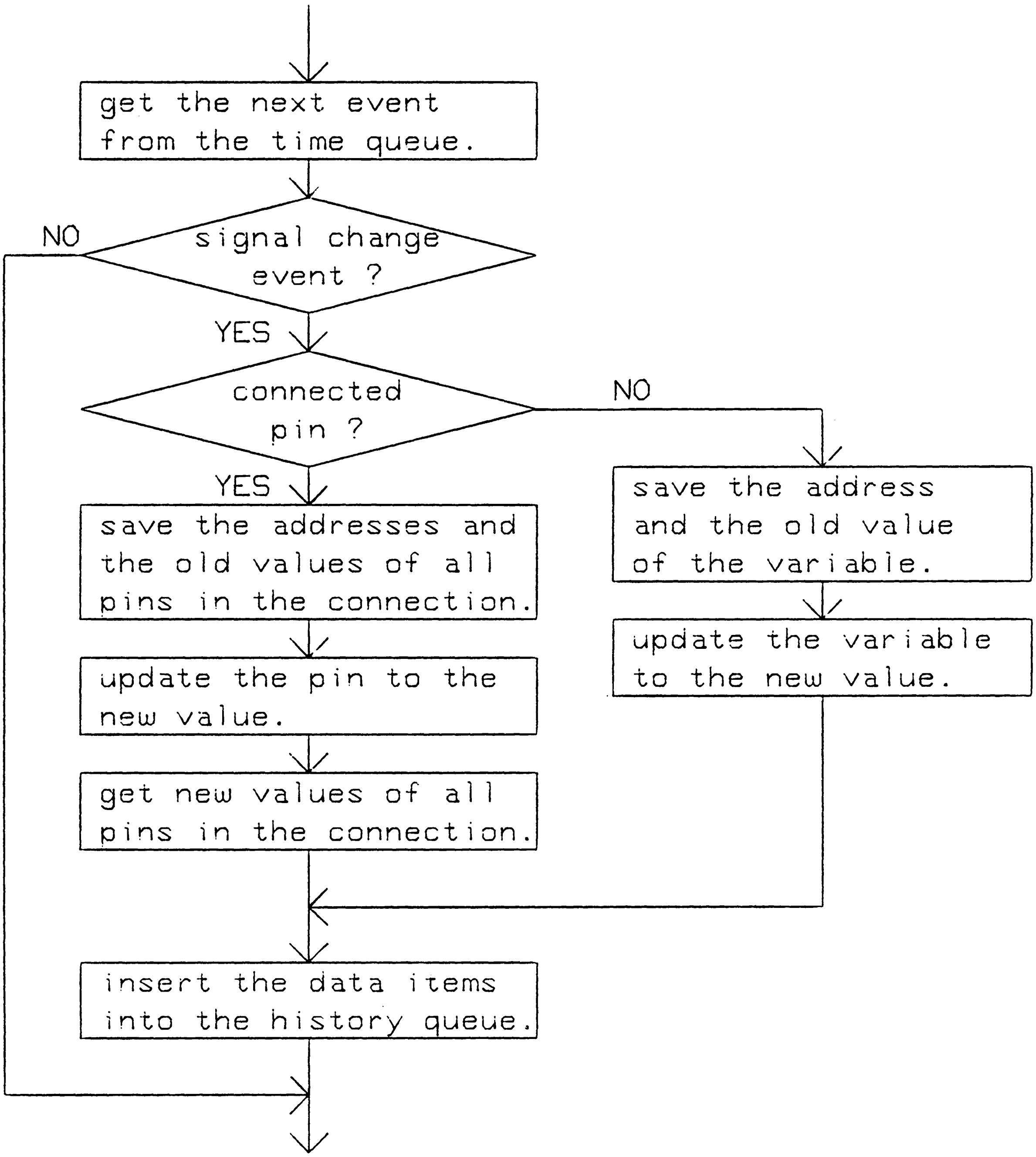

Figure 17. Implementation of SETUP Function

The monitor obtains and saves the data items of all connected pins using two data structures, the runtime event activation table and the runtime bus connection table.

If the next event (signal change) is on the source pin involved in a normal connection, the following algorithm is used.

1. Access the variable symbol node for the pin (the pointer to the node is stored in 6 of the time queue entry for the event).
2. Compute the address of an entry of the runtime event activation table by adding the index into the table to the base address of the table.
3. Go to the first cell whose first field contains "true_byte" (it means that the cell contains the name of a pin connected to the scheduled pin).
4. Traverse the module tree until a node of the same module name is found. Access the root of the variable symbol tree within the module.
5. Traverse the variable symbol tree until a node of the same pin name is found.
6. Read the address where the addresses for the value of the pin vector are stored (it is stored in field 4 of the node).

7. Obtain the value of the pin vector and stores the value in field 5 of the cell.
8. If the cell is the last one in the linked list, the algorithm terminates. Otherwise, go to the next cell whose first cell contains "true_byte".
9. Go to step 4.

The similar algorithm is used to process the addresses and new values of the connected pins.

If the event (signal change) is on a bus pin used as a destination, the monitor performs the similar algorithm using the runtime bus connection table. The table does not contain source pins because they are not connected, i.e., they keep their own memory locations (see section 2.3).

4.5 IMPLEMENTATION OF WAIT STATEMENT

The WAIT statement is used in suspending the execution of an event block for the specified waiting time. When the waiting time expires, the event block should be executed from the statement following the WAIT. Therefore, the monitor should be able to access the correct entry point when it processes a reactivated event. In the original GSP2, the monitor accesses wrong entry points in some cases. Therefore, a new scheme is adopted for the complete implementation of the WAIT statement.

In the original GSP2 system, each node of the event block symbol tree has a field that stores the starting address of the corresponding event block. In addition, it has a field that stores the entry point for the event that invokes the event block next time. When the monitor executes an event block until the the end of the event block is reached, the starting address of the event block is stored in the field as the entry point for the next event. On the other hand, when the monitor encounters a WAIT statement during the execution of an event block, the address of the statement following the WAIT is stored in the field. Now, suppose that the monitor encounters a WAIT statement during the execution of an event block. In this case, the monitor will store the address of the statement following the WAIT in the field and schedule a reactivated event into the time queue. If the next event that invokes the event block is the reactivated event, the monitor can access the correct entry point. A problem arises when the next event is a signal change event. In this case, the monitor is obliged to access the entry point to be used for the reactivated event and therefore executes the event block from the statement following the WAIT. In other words, the monitor does not execute the statements between the head of the event block and the WAIT statement. It will cause a undesired simulation result.

To eliminate the problem mentioned above, the monitor uses the following scheme.

1. The monitor does not use the event block symbol node to define the entry point for the next event. Instead, each entry of the time queue contains four fields[2] to store the entry point for a reactivated event.
2. Whenever a WAIT is encountered in an event block, the monitor inserts the address following the WAIT into the corresponding time queue entry.
3. If a signal change event comes out of the time queue, the monitor always uses the starting address that is stored in the corresponding event block symbol node. If a reactivated event comes out of the time queue, the monitor uses the entry point stored in the corresponding time queue entry.

The new scheme enables the user to correctly model the situation where an unexpected external event can disturb a

---

[2] In fact, an entry point is defined by four addresses, the program counter, the stack pointer, the argument pointer, and the frame pointer [13]. This is because WAIT can exist in a procedure which can be called by one or more event blocks.

waiting process. It is an advantage of the GSP2 over other hardware description languages [14,15], where the unexpected event is cancelled or held. Now, suppose that the modeler wants the event block not to be invoked again before the waiting time expires. The modeler can model the case by using the statements ARM and DISARM as shown in the following event block.

```
EVENT event_block_name ON (event expression);
  BEGIN
    (statements)
    DISARM event_block_name;
    WAIT waiting_time;
    ARM event_block_name;
    (statements)
  END
```

4.6 CPU TIME MEASURING ROUTINE

A CPU time measuring routine is inserted into the simulator as a means of measuring the simulation speed of the GSP2 system. It will be useful to predict the simulation time for a complex digital system.

The time measuring routine uses two callable VAX routines, LIB$INIT_TIMER and LIB$STAT_TIMER [16]. When LIB$INIT_TIMER is called, the timer which measures the CPU time is initialized. When LIB$STAT_TIMER is called, the timer stops and the measured time is returned. To make the measurement accurate, LIB$INIT_TIMER is placed in the simulator such that it can be executed with the GCL GO command. LIB$STAT_TIMER is placed such that it can be executed as soon as the simulation stops. There are following three cases that the simulation stops.

1. when the time queue is empty.
2. when the simulation time exceeds the maximum simulation time specified by the user.
3. when the GCL BREAK command is executed.

Therefore, LIB$STAT_TIMER is inserted into the simulator at the corresponding three positions.

5. AN EXAMPLE OF MULTIMODULE SIMULATION

As an example of modeling and simulation of a multimodule system with the GSP2, a simple processor system is demonstrated. As shown in figure 18, the system consists of a clock generator, a simple processor unit, a RAM, two parallel I/O ports, and a UART for serial I/O. The processor unit MARK2 is similar to the late 1940's processor MARK1 [17], which had the capability of executing only seven instructions. This chapter shows the complete simulation procedure, from modeling to simulation.

5.1 SYSTEM MODELING

It is assumed that the simulated system uses a memory-mapped I/O addressing scheme. There are five address lines that control the chip selection for either memory access or I/O. The highest two address lines determine the chip selection logic, which is incorporated into the model of the processor. The RAM chip is selected with a LOW signal on the highest address line. An I/O chip is selected with a HIGH signal on the line. The next highest address line determines which type of I/O is used. A LOW signal indicates parallel I/O while a HIGH signal indicates serial I/O. Thus, the address map is defined as follows:

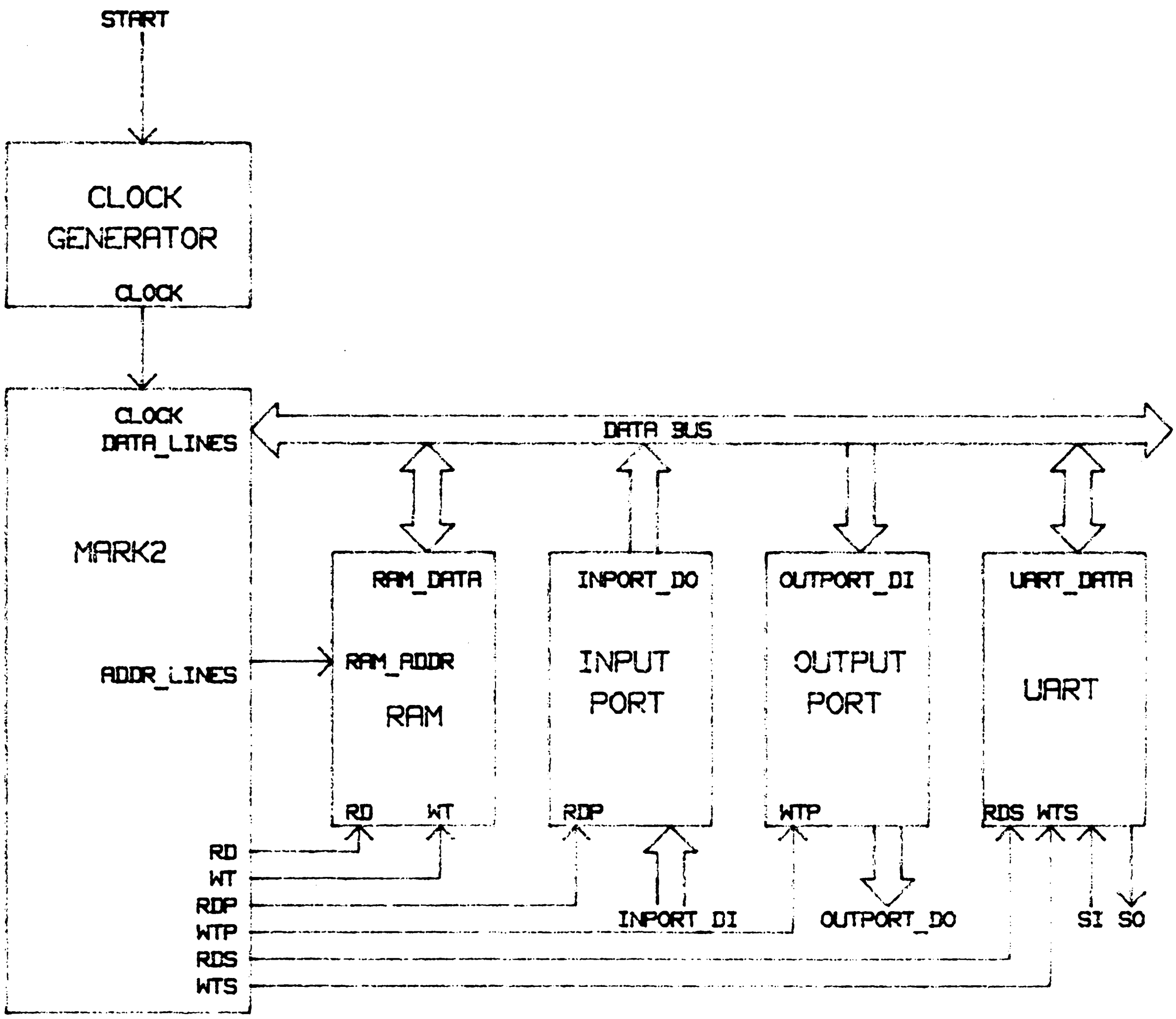

Figure 18. A Processor System

address	selected chip
0XXXX	RAM
10XXX	parallel I/O
11XXX	serial I/O

5.1.1 THE SYSTEM INTERCONNECTION

The interconnection among the chips are described in a connection description file (figure 19). There are seven normal (TYPE 2) connections and a bus (TYPE 3) connection in the file. The second connection statements in the file represents the system data bus which is implemented in a wired-AND logic circuit.

5.1.2 THE CLOCK GENERATOR

Figure 20 shows the modeling of the clock generator. The event block clock_enable is invoked with the rising edge of the external signal "start" and generates the system clock. The combination of the statements WAIT and LOOP is very useful in this type of oscillating circuits. The clock period is defined by the waiting time "clock_width" and the number of clock cycles are defined by the index of the LOOP statement "no_of_clock". It is notable that the duty cycle of the clock can be determined by the delay time parameters of the two assignment statements. In this example, the duty cycle of the clock is 50%.

5.1.3 THE PROCESSOR UNIT

The word length of the MARK2 is 8 bits long which allows for the address length of 5 bits and instruction opcodes of 3 bits. The seven instructions and their opcodes are defined as follows:

1. JMP (000) - absolute jump
2. JRP (001) - jump relative to program counter
3. LDN (010) - load negative value of the operand into accumulator
4. STO (011) - store the contents of accumulator
5. SUB (100 or 101) - subtract the operand from the value of accumulator and store the result in accumulator
6. CMP (110) - compare accumulator against zero; if negative skip the next instruction, else execute it
7. STP (111) - stop

Figure 21 shows the modeling of the processor. The processor model contains an event block and three procedures. The event block "cycle" is invoked with the rising edge of the signal "clock". The processor fetches, decodes, and executes an instruction in a single clock cycle. The first four statements in the event block represents the instruction fetch phase. The procedure @read_mem is called in this phase. The use of the WAIT statement in the procedure effects the suspension required for memory or I/O access. The decoding phase is modeled by the CASE statement whose individual branch defines the execution of the decoded instruction.

The procedure @select represents the chip selection logic. It is called by the procedures @read_mem and @write_mem for selecting a proper chip for the read or write operation.

5.1.4 THE RAM UNIT

The RAM model contains three event blocks in it (figure 22). The event blocks read_mem and write_mem represents the memory read and the memory write operation, respectively. The event block float_ram represents that the output enable circuit of the RAM chip is disabled when the signal "rd" makes a 0 to 1 transition.

5.1.5 THE PARALLEL PORTS

The parallel ports "inport" and "outport" have similar modeling structures (figure 23). The only difference is that sources of input data and the destinations of the output data are different each other. The event block par_input (par_output) is invoked when the signal "rdp" ("wtp") makes a 0 to 1 transition. If the input data has been stable for 10 unit times, the port transmits the input data to the output with the delay of 40 unit times. When the signal "rdp" (or "wtp") makes a 1 to 0 transition, the output circuit of the port is disabled.

5.1.6 THE UART

The module UART (figure 24) shows a modeling method of serial-to-parallel or parallel-to-serial data conversion. The event blocks load_reg and ser_out represents the parallel-to-serial data conversion. The event blocks read_bit, read_data, and uart_float represents serial-to-parallel data conversion. As in the modeling of the clock generator, the internal clock of the chip is modeled with the statements LOOP and WAIT.

5.2 CONSTRUCTION OF EXECUTABLE IMAGE FILE

Before the module description files are compiled, the connection description file ("dacon.con") is compiled using the following command:

```
$ gspconn dacon <CR>
```

In fact, the connect compiler generates the connection topology file "dacon.icn" and the connect file "dacon.mar". The file "dacon.icn" is used by the module description compiler and the file "dacon.mar" is linked with other chip models by the GSP2 linker. The module description files are sequentially compiled using the following commands:

```
$ gspcomp clock   dacon <CR>
$ gspcomp mark2   dacon <CR>
$ gspcomp ram     dacon <CR>
$ gspcomp inport  dacon <CR>
$ gspcomp outport dacon <CR>
$ gspcomp uart    dacon <CR>
```

Next, the module is linked with the simulator with the following commands:

```
$ gsplink <CR>
Connect: dacon   <CR>
Module: clock    <CR>
Module: mark2    <CR>
Module: ram      <CR>
Module: inport   <CR>
Module: outport  <CR>
Module: uart     <CR>
Module: <CR>
Image:  dasys    <CR>
```

After the above sequence is finished, the GSP2 linker creates the executable image file "dasys.exe".

5.3 SIMULATION

5.3.1 TEST PROGRAM

To verify the modeling of the simulated system, a test program is created. The program is written in a separate file "ram.dat" and is loaded into the memory "mem" of the module RAM during the initialization phase of the simulation. In this example, the following test program is used.

addr.	contents	comment
0000	B10000010	LDN parallel port (data: 00000011)
0001	B10000100	SUB parallel port (data: 00001100)
0010	B01100011	STO memory (location: 1100)
0011	B01100010	LDN memory (location: 1100)
0100	B01011100	SUB memory (location: 1011)
0101	B10000011	STO parallel port
0110	B11000011	STO serial port
0111	B01100011	STO memory (location: 1100)
1000	B00000111	STP
1001	B00000000	
1010	B00000000	
1011	B00000001	data
1100	B00000000	
1101	B00000000	
1110	B00000000	
1111	B00000000	

The program first loads the negative value of the parallel input data (+3) and then subtracts another parallel input data (+12) from it (the parallel data are scheduled during the intialization phase of the simulation). The program stores the result (-15) in memory location 1100. Next, the program negates the result and subtracts the value (+1) at memory location 1011 from it. The result (+14) is sequentially written in the parallel port, the serial port, and the memory (location: 1100).

5.3.2 THE GCL COMMAND FILE

Figure 25 shows the GCL command file "dasys.gcl" that consists of the following four parts:

1. The simulation parameters, the maximum simulation time and the history parameter (for SETUP function), are specified.
2. The contents of the file "mem.dat" is loaded into the internal memory.
3. The external signals and the data inputs are scheduled.
4. The range of data items to be displayed in the output file is specified.

There is another GCL command file (figure 25) that contains the GCL commands used in examing the memory contents. They are contained in a separate file "exram.gcl" to be used only when necessary.

5.3.3 SIMULATION

The executable image file "dasys.exe" is invoked with the following command:

```
$ gspsim dasys <CR>
  (banner message)
GCL>
```

After that, two command files "dasys.gcl" and "exram.gcl" are invoked to set up the system and display the contents of the memory. The character "@" is used in invoking the command files. Finally, the simulation starts with the GCL command GO. The simulation result is displayed in the output file shown in appendix A.

```
CONNECT

clock_generator: clock[0|1,0|1] OUTPUT,
mark2: clock[0|1,0|1] INPUT;

mark2: data_lines[0|8,0|8] BIDIRECT: BUSAND,
ram: ram_data[0|8,0|8] BIDIRECT: BUSAND,
uart: uart_data[0|8,0|8] BIDIRECT: BUSAND,
inport: inport_do[0|8,0|8] OUTPUT: BUSAND,
outport: outport_di[0|8,0|8] INPUT: BUSAND;

mark2: addr_lines[0|5,0|5] OUTPUT,
ram: ram_addr[0|5,0|5] INPUT;

mark2: rd[0|1,0|1] OUTPUT,
ram: rd[0|1,0|1] INPUT;

mark2: wt[0|1,0|1] OUTPUT,
ram: wt[0|1,0|1] INPUT;

mark2: rds[0|1,0|1] OUTPUT,
uart: rds[0|1,0|1] INPUT;

mark2: wts[0|1,0|1] OUTPUT,
uart: wts[0|1,0|1] INPUT;

mark2: rdp[0|1,0|1] OUTPUT,
inport: rdp[0|1,0|1] INPUT;

mark2: wtp[0|1,0|1] OUTPUT,
outport: wtp[0|1,0|1] INPUT

ENDCONN
```

Figure 19. Modeling of the System Interconnection

```
MODULE clock_generator

LITERALS
  no_of_clock = '10';
  clock_width = '500'

PINS
  start[0|1] : INPUT;
  clock[0|1] : OUTPUT

DECLARE
  count: INTEGER

EVENT clock_enable ON @RISE(start);
  LOOP count := 1 TO no_of_clock BY 1 DO
    BEGIN
      clock := #B1 PROP clock_width / 2;
      clock := #B0 PROP clock_width;
      WAIT clock_width
    END
  ENDLOOP

ENDMOD
```

Figure 20. Modeling of the Clock Generator

```
MODULE mark2

LITERALS
  ! instruction register fields !
  opcode = '0|3'; address = '3|5'; sign_bit = '7|1';
  ! instruction codes !
  jmp = '#B000'; jrp = '#B001'; ldn = '#B010';
  sto = '#B011'; sub1 = '#B100'; sub2 = '#B101';
  cmp = '#B110'; stp = '#B111';
  ! time parameter !
  read_period = '150';
  write_period = '200'

PINS
  clock[0|1] : INPUT;
  data_lines[0|8] : BIDIRECT;
  addr_lines[0|5],rd[0|1],wt[0|1],rds[0|1],
  rdp[0|1],wts[0|1],wtp[0|1] : OUTPUT
DECLARE
  ir[0|8],acc[0|8],pc[0|5],md[0|8],mar[0|5],
  rd_signal[0|1],wt_signal[0|1] : REGISTER VALUE 0

PROCEDURE @select(COPY addr[0|5] : REGISTER)
  BEGIN
    rd := rd_signal PROP 10;
    rds := @AND(addr[3|1],@AND(addr[4|1],rd_signal))
           PROP 10;
    rdp := @AND(@NOT(addr[3|1]),
                @AND(addr[4|1],rd_signal)) PROP 10;
    wt := wt_signal PROP 10;
    wts := @AND(addr[3|1],@AND(addr[4|1],wt_signal))
           PROP 10;
    wtp := @AND(@NOT(addr[3|1]),
                @AND(addr[4|1],wt_signal)) PROP 10;
  END
PROCEDURE @read_mem(COPY addr[0|5]: REGISTER;
                    MOD data[0|8]: REGISTER)
  BEGIN
    addr_lines := addr;
    rd_signal := #B1;
    @select(addr);
    rd_signal := #B0 PROP read_period;
    WAIT read_period;
    @select(addr);
    data := data_lines
  END
```

Figure 21. The Processor Model (continued)

```
PROCEDURE @write_mem(COPY addr[0|5],data[0|8]: REGISTER)
  BEGIN
    addr_lines := addr;
    data_lines := data;
    wt_signal := #B1;
    @select(addr);
    wt_signal := #B0 PROP write_period;
    WAIT write_period;
    @select(addr);
    data_lines := #HFF PROP 10
  END

EVENT cycle ON @RISE(clock);
  BEGIN
    @read_mem(pc,ir);
    mar := ir[address];
    pc := @ADD(pc,#B00001);
    WAIT 50;
    CASE ir[opcode] OF
      jmp: pc := mar;
      jrp: pc := @ADD(@SUB(pc,#B00001),mar);
      ldn: BEGIN
             @read_mem(mar,md);
             acc := @NEG(md)
           END;
      sto: @write_mem(mar,acc);
      sub1,sub2:
           BEGIN
             @read_mem(mar,md);
             acc := @ADD(acc,@NEG(md))
           END;
      cmp: IF acc[sign_bit] = #B1
             THEN
               pc := @ADD(pc,#B00001)
           ENDIF;
      stp: BEGIN
             WRITE TERMINAL 'processor halted';
             WRITE LOG 'processor halted';
             PAUSE
           END
    ENDCASE
  END

ENDMOD
```

Figure 21. The Processor Model

```
MODULE ram

PINS
  rd[0|1],wt[0|1],ram_addr[0|5] : INPUT;
  ram_data[0|8] : BIDIRECT

DECLARE
  mem[0|5,0|8] : MEMORY

EVENT read_mem ON @AND(@NOT(ram_addr[4|1]),@RISE(rd));
  ram_data := mem[ram_addr,0|8] PROP 100

EVENT write_mem ON @AND(@NOT(ram_addr[4|1]),@RISE(wt));
  mem[ram_addr] := ram_data PROP 100

EVENT float_ram ON @AND(@NOT(ram_addr[4|1]),@FALL(rd));
  ram_data := #HFF PROP 10

ENDMOD
```

Figure 22. The RAM Model

```
MODULE inport

PINS
  inport_di[0|8],rdp[0|1] : INPUT;
  inport_do[0|8] : OUTPUT

EVENT par_input ON @RISE(rdp);
  IF @SETUP(inport_di,10)
    THEN
      inport_do := inport_di PROP 40
  ENDIF

EVENT float_input ON @FALL(rdp);
  inport_do := #HFF PROP 10

ENDMOD
```

```
MODULE outport

PINS
  outport_di[0|8],wtp[0|1] : INPUT;
  outport_do[0|8] : OUTPUT

EVENT par_output ON @RISE(wtp);
  IF @SETUP(outport_di,10)
    THEN
      outport_do := outport_di PROP 40
  ENDIF

EVENT float_output ON @FALL(wtp);
  outport_do := #HFF PROP 10

ENDMOD
```

Figure 23. Modeling of Parallel Ports

```
MODULE uart

PINS
  si[0|1],rds[0|1],wts[0|1] : INPUT;
  uart_data[0|8] : BIDIRECT;
  so[0|1] : OUTPUT VALUE #B1

DECLARE
  preg[0|8] : REGISTER;
  icntr: INTEGER

EVENT load_reg ON @RISE(wts);
  BEGIN
    WAIT 10;
    IF @SETUP(uart_data,10)
      THEN
        BEGIN
          preg := uart_data;
          so := #B0 PROP 90
        END
    ENDIF
  END

EVENT ser_out ON @FALL(so);
  BEGIN
    LOOP icntr := 1 TO 8 BY 1 DO
      BEGIN
        WAIT 100;
        preg := @SHIFT(preg,1);
        so := CARRY
      END
    ENDLOOP;
    so := #B1 PROP 100
  END
```

Figure 24. The UART Model (continued)

```
EVENT read_bit ON @FALL(si);
  BEGIN
    WAIT 50;
    IF @SETUP(si,50)
      THEN
        LOOP icntr := 1 TO 8 BY 1 DO
          BEGIN
            WAIT 100;
            preg[0|1] := si;
            IF icntr < 8
              THEN
                preg := @SHIFT(preg,-1)
            ENDIF
          END
        ENDLOOP
    ENDIF
  END

EVENT read_data ON @RISE(rds);
  uart_data := preg PROP 40

EVENT uart_float ON @FALL(rds);
  uart_data := #HFF PROP 5

ENDMOD
```

Figure 24. The UART model

Command File "dasys.gcl"

```
set timelimit = 10000
set history = 20
set module = ram
load mem RAM.DAT
set module = clock_generator
dep start = B1 AT 50
set module = inport
dep inport_di = B00000011 AT 150
dep inport_di = B00001100 AT 650
trace module clock_generator
trace module mark2
trace module ram
trace module inport
trace module outport
trace module uart
```

Command File "exram.gcl"

```
set module = ram
set cbase = D
ex mem[0]
ex mem[1]
ex mem[2]
ex mem[3]
ex mem[4]
ex mem[5]
ex mem[6]
ex mem[7]
ex mem[8]
ex mem[9]
ex mem[10]
ex mem[11]
ex mem[12]
ex mem[13]
ex mem[14]
ex mem[15]
set cbase = B
```

Figure 25. The GCL Command Files

6. FURTHER RESEARCH AREA

6.1 MULTI-STATE SIMULATOR

Since the implemented GSP2 simulator provides only two logic states 1 and 0, it has the following drawbacks [18].

1. While the initial values of some signals in the real hardware systems are in "unknown" states, all variables are initialized to logic 1 or 0 in the GSP2 system.
2. When the timing specifications of a model are not satisfied, the only thing that can be done in the GSP2 simulation is to output a proper error message. However, with a point of view of the chip level fault simulation, the error needs to be propagated to the output pins.
3. There is no method of distinguishing a "high impedance state" from logic 1 or 0.
4. The exact modeling of a tri-state bus is impossible (see section 3.4.1).

Two things should be considered to solve the problems mentioned above.

First, two additional states, the high impedance state ('Z') and the unknown state ('X'), should be incorporated into the simulator. It means that each bit of a vector should be represented by at least two bits. For example, we can assign the values 00, 01, 10, and 11 to '0', 'Z', 'X', and '1', respectively. Since each bit of a pin variable is assigned a byte ("one bit per byte" scheme), the two states can be incorporated into the pin variables (see section 2.3). However, it is impossible for a register or memory variable since its whole vector is stored in a single memory location. Therefore, the "one bit per byte" scheme should be also applied to register variables and memory variables.

The next thing to be considered is how to handle the two states when they are encountered in arithmetic and logic functions or control statements. This can be handled by either the modeler [15] or the simulator itself [19]. In the former case,the modeler will create the procedures that can handle the two states in addition to logic 1 or 0. In the latter case, the built-in arithmetic and logic functions should be able to handle the four states. The former case is preferred to the latter to give the modeler more flexibility. Figure 26 shows the expected procedure (that would be created by the modeler) that can handle the following three-value AND function.

	0	1	X
0	0	0	0
1	0	1	X
X	0	X	X

As shown in the figure, the simulator should be changed such that it can allow at least the following things:

1. Each bit of a vector can be indexed by an integer value.
2. The control statements CASE and IF-THEN-ELSE can handle 'X' or 'Z' states.

6.2 HANDLING OF CONNECTION INFORMATION

As mentioned in chapter 1, there are two methods to incorporate the connection information of a simulated system into the simulator. One method is to generate the connection information before the chip models are compiled. In contrast, the other method is to generate it during the simulation after the chip models are compiled and linked together. Since the disadvantage of the first method is the advantage of the second method and vice versa, which method to select depends on the purpose of the simulation. When the system model has been fixed and is repeatedly simulated, the first method is the better since it increases the simulation speed. However, if the main purpose of the simulation is to design a digital

```
PROCEDURE @three_value_and(COPY vector_size: INTEGER;
                           COPY a,b: REGISTER;
                           MOD c: REGISTER)
  DECLARE
    i: INTEGER

  FOR i := 1 TO vector_size DO
    CASE a[i] of
      0: c[i] := #B0;
      1: IF (b[i] <> #BX)
           THEN
             c[i] := @AND(a[i],b[i])
           ELSE
             c[i] := #BX
         ENDIF;
      X: IF b[i] = #B0
           THEN
             c[i] := #B0
           ELSE
             c[i] := #BX
         ENDIF
    ENDCASE
  ENDFOR
```

Figure 26. Three-value AND function

system using the built-in chip models, the second method is the better since the modeler can easily change the interconnection among the chip models.

If the second method were adopted, the GSP2 system would be changed as follows:

1. The connection description file is not needed any more.
2. The user will give the connection information through some GCL commands.

3. The simulator should have the capability of creating the data structure to be used in the multimodule simulation. This function of the simulator would be similar to that of the connect compiler.

7. CONCLUSION

A multimodule simulator using a hardware description language has been developed. The "single memory allocation" scheme used for the connected pins increases the simulation speed compared to the "complete fanout tracing" scheme. The simulator has the capability of modeling external buses and handling the suspension of processes in the environment of multimodule simulation. A multimodule digital system has been modeled and simulated as the verification of the implemented simulator. Suggestions have been made for the extension of the simulator to a multi-state simulator.

APPENDIX A. SIMULATION RESULT -- OUTPUT FILE

```
GSP2 Simulator -- Revision 0.00 -- on
26-FEB-1986 04: 20: 03.75
No events to process (time queue empty).
GCL> @DASYS.GCL
GCL> set timelimit = 10000
GCL> set history = 20
GCL> set module = ram
GCL> load mem RAM.DAT
GCL> set module = clock_generator
GCL> dep start = B1 AT 50
GCL> set module = inport
GCL> dep inport_di = B00000011 AT 150
GCL> dep inport_di = B00001100 AT 650
GCL> trace module clock_generator
GCL> trace module mark2
GCL> trace module ram
GCL> trace module inport
GCL> trace module outport
GCL> trace module uart
GCL> @EXRAM.GCL
GCL> set module = ram
GCL> set cbase = D
GCL> ex mem[0]
    * RAM: MEM = 10000010
GCL> ex mem[1]
    * RAM: MEM = 10000100
GCL> ex mem[2]
    * RAM: MEM = 01100011
GCL> ex mem[3]
    * RAM: MEM = 01100010
GCL> ex mem[4]
    * RAM: MEM = 01011100
GCL> ex mem[5]
    * RAM: MEM = 10000011
GCL> ex mem[6]
    * RAM: MEM = 11000011
GCL> ex mem[7]
    * RAM: MEM = 01100011
GCL> ex mem[8]
    * RAM: MEM = 00000111
GCL> ex mem[9]
    * RAM: MEM = 00000000
GCL> ex mem[10]
    * RAM: MEM = 00000000
GCL> ex mem[11]
```

```
    * RAM: MEM = 00000001
GCL> ex mem[12]
    * RAM: MEM = 00000000
GCL> ex mem[13]
    * RAM: MEM = 00000000
GCL> ex mem[14]
    * RAM: MEM = 00000000
GCL> ex mem[15]
    * RAM: MEM = 00000000
GCL> set cbase = B
GCL> GO
CPU TIME IS INITIALIZED.
Time 50 - event CLOCK_GENERATOR: CLOCK_ENABLE has occurred
 Pins and Global Registers Before CLOCK_ENABLE Processed:
    * CLOCK = 0
    * START = 1
 Other Active Events at This Time:
Time 300 - event MARK2: CYCLE has occurred
 Pins and Global Registers Before CYCLE Processed:
    * ACC = 00000000
    * ADDR_LINES = 00000
    * CLOCK = 1
    * DATA_LINES = 11111111
    * DATA_LINES(IN) = 11111111
    * INPORT_DO = 11111111
    * IR = 00000000
    * MAR = 00000
    * MD = 00000000
    * OUTPORT_DI = 11111111
    * PC = 00000
    * RAM_DATA = 11111111
    * RAM_DATA(IN) = 11111111
    * RD = 0
    * RDP = 0
    * RDS = 0
    * RD_SIGNAL = 0
    * UART_DATA = 11111111
    * UART_DATA(IN) = 11111111
    * WT = 0
    * WTP = 0
    * WTS = 0
    * WT_SIGNAL = 0
 Other Active Events at This Time:
Time 310 - event RAM: READ_MEM has occurred
 Pins and Global Registers Before READ_MEM Processed:
    * DATA_LINES = 11111111
    * DATA_LINES(IN) = 11111111
    * INPORT_DO = 11111111
    * OUTPORT_DI = 11111111
    * RAM_ADDR = 00000
    * RAM_DATA = 11111111
```

```
    * RAM_DATA(IN) = 11111111
    * RD = 1
    * UART_DATA = 11111111
    * UART_DATA(IN) = 11111111
    * WT = 0
 Other Active Events at This Time:
Time 460 - event RAM: FLOAT_RAM has occurred
 Pins and Global Registers Before FLOAT_RAM Processed:
    * DATA_LINES = 11111111
    * DATA_LINES(IN) = 10000010
    * INPORT_DO = 11111111
    * OUTPORT_DI = 10000010
    * RAM_ADDR = 00000
    * RAM_DATA = 10000010
    * RAM_DATA(IN) = 10000010
    * RD = 0
    * UART_DATA = 11111111
    * UART_DATA(IN) = 10000010
    * WT = 0
 Other Active Events at This Time:
Time 510 - event INPORT: PAR_INPUT has occurred
 Pins and Global Registers Before PAR_INPUT Processed:
    * DATA_LINES = 11111111
    * DATA_LINES(IN) = 11111111
    * INPORT_DI = 00000011
    * INPORT_DO = 11111111
    * OUTPORT_DI = 11111111
    * RAM_DATA = 11111111
    * RAM_DATA(IN) = 11111111
    * RDP = 1
    * UART_DATA = 11111111
    * UART_DATA(IN) = 11111111
 Other Active Events at This Time:
Time 660 - event INPORT: FLOAT_INPUT has occurred
 Pins and Global Registers Before FLOAT_INPUT Processed:
    * DATA_LINES = 11111111
    * DATA_LINES(IN) = 00000011
    * INPORT_DI = 00001100
    * INPORT_DO = 00000011
    * OUTPORT_DI = 00000011
    * RAM_DATA = 11111111
    * RAM_DATA(IN) = 00000011
    * RDP = 0
    * UART_DATA = 11111111
    * UART_DATA(IN) = 00000011
 Other Active Events at This Time:
Time 800 - event MARK2: CYCLE has occurred
 Pins and Global Registers Before CYCLE Processed:
    * ACC = 11111101
    * ADDR_LINES = 10000
    * CLOCK = 1
```

```
    * DATA_LINES = 11111111
    * DATA_LINES(IN) = 11111111
    * INPORT_DO = 11111111
    * IR = 10000010
    * MAR = 10000
    * MD = 00000011
    * OUTPORT_DI = 11111111
    * PC = 00001
    * RAM_DATA = 11111111
    * RAM_DATA(IN) = 11111111
    * RD = 0
    * RDP = 0
    * RDS = 0
    * RD_SIGNAL = 0
    * UART_DATA = 11111111
    * UART_DATA(IN) = 11111111
    * WT = 0
    * WTP = 0
    * WTS = 0
    * WT_SIGNAL = 0
 Other Active Events at This Time:
Time 810 - event RAM: READ_MEM has occurred
 Pins and Global Registers Before READ_MEM Processed:
    * DATA_LINES = 11111111
    * DATA_LINES(IN) = 11111111
    * INPORT_DO = 11111111
    * OUTPORT_DI = 11111111
    * RAM_ADDR = 00001
    * RAM_DATA = 11111111
    * RAM_DATA(IN) = 11111111
    * RD = 1
    * UART_DATA = 11111111
    * UART_DATA(IN) = 11111111
    * WT = 0
 Other Active Events at This Time:
Time 960 - event RAM: FLOAT_RAM has occurred
 Pins and Global Registers Before FLOAT_RAM Processed:
    * DATA_LINES = 11111111
    * DATA_LINES(IN) = 10000100
    * INPORT_DO = 11111111
    * OUTPORT_DI = 10000100
    * RAM_ADDR = 00001
    * RAM_DATA = 10000100
    * RAM_DATA(IN) = 10000100
    * RD = 0
    * UART_DATA = 11111111
    * UART_DATA(IN) = 10000100
    * WT = 0
 Other Active Events at This Time:
Time 1010 - event INPORT: PAR_INPUT has occurred
 Pins and Global Registers Before PAR_INPUT Processed:
```

```
    * DATA_LINES = 11111111
    * DATA_LINES(IN) = 11111111
    * INPORT_DI = 00001100
    * INPORT_DO = 11111111
    * OUTPORT_DI = 11111111
    * RAM_DATA = 11111111
    * RAM_DATA(IN) = 11111111
    * RDP = 1
    * UART_DATA = 11111111
    * UART_DATA(IN) = 11111111
 Other Active Events at This Time:
Time 1160 - event INPORT: FLOAT_INPUT has occurred
 Pins and Global Registers Before FLOAT_INPUT Processed:
    * DATA_LINES = 11111111
    * DATA_LINES(IN) = 00001100
    * INPORT_DI = 00001100
    * INPORT_DO = 00001100
    * OUTPORT_DI = 00001100
    * RAM_DATA = 11111111
    * RAM_DATA(IN) = 00001100
    * RDP = 0
    * UART_DATA = 11111111
    * UART_DATA(IN) = 00001100
 Other Active Events at This Time:
Time 1300 - event MARK2: CYCLE has occurred
 Pins and Global Registers Before CYCLE Processed:
    * ACC = 11110001
    * ADDR_LINES = 10000
    * CLOCK = 1
    * DATA_LINES = 11111111
    * DATA_LINES(IN) = 11111111
    * INPORT_DO = 11111111
    * IR = 10000100
    * MAR = 10000
    * MD = 00001100
    * OUTPORT_DI = 11111111
    * PC = 00010
    * RAM_DATA = 11111111
    * RAM_DATA(IN) = 11111111
    * RD = 0
    * RDP = 0
    * RDS = 0
    * RD_SIGNAL = 0
    * UART_DATA = 11111111
    * UART_DATA(IN) = 11111111
    * WT = 0
    * WTP = 0
    * WTS = 0
    * WT_SIGNAL = 0
 Other Active Events at This Time:
Time 1310 - event RAM: READ_MEM has occurred
```

```
 Pins and Global Registers Before READ_MEM Processed:
    * DATA_LINES = 11111111
    * DATA_LINES(IN) = 11111111
    * INPORT_DO = 11111111
    * OUTPORT_DI = 11111111
    * RAM_ADDR = 00010
    * RAM_DATA = 11111111
    * RAM_DATA(IN) = 11111111
    * RD = 1
    * UART_DATA = 11111111
    * UART_DATA(IN) = 11111111
    * WT = 0
 Other Active Events at This Time:
Time 1460 - event RAM: FLOAT_RAM has occurred
 Pins and Global Registers Before FLOAT_RAM Processed:
    * DATA_LINES = 11111111
    * DATA_LINES(IN) = 01100011
    * INPORT_DO = 11111111
    * OUTPORT_DI = 01100011
    * RAM_ADDR = 00010
    * RAM_DATA = 01100011
    * RAM_DATA(IN) = 01100011
    * RD = 0
    * UART_DATA = 11111111
    * UART_DATA(IN) = 01100011
    * WT = 0
 Other Active Events at This Time:
Time 1510 - event RAM: WRITE_MEM has occurred
 Pins and Global Registers Before WRITE_MEM Processed:
    * DATA_LINES = 11110001
    * DATA_LINES(IN) = 11110001
    * INPORT_DO = 11111111
    * OUTPORT_DI = 11110001
    * RAM_ADDR = 01100
    * RAM_DATA = 11111111
    * RAM_DATA(IN) = 11110001
    * RD = 0
    * UART_DATA = 11111111
    * UART_DATA(IN) = 11110001
    * WT = 1
 Other Active Events at This Time:
Time 1800 - event MARK2: CYCLE has occurred
 Pins and Global Registers Before CYCLE Processed:
    * ACC = 11110001
    * ADDR_LINES = 01100
    * CLOCK = 1
    * DATA_LINES = 11111111
    * DATA_LINES(IN) = 11111111
    * INPORT_DO = 11111111
    * IR = 01100011
    * MAR = 01100
```

```
    * MD = 00001100
    * OUTPORT_DI = 11111111
    * PC = 00011
    * RAM_DATA = 11111111
    * RAM_DATA(IN) = 11111111
    * RD = 0
    * RDP = 0
    * RDS = 0
    * RD_SIGNAL = 0
    * UART_DATA = 11111111
    * UART_DATA(IN) = 11111111
    * WT = 0
    * WTP = 0
    * WTS = 0
    * WT_SIGNAL = 0
 Other Active Events at This Time:
Time 1810 - event RAM: READ_MEM has occurred
 Pins and Global Registers Before READ_MEM Processed:
    * DATA_LINES = 11111111
    * DATA_LINES(IN) = 11111111
    * INPORT_DO = 11111111
    * OUTPORT_DI = 11111111
    * RAM_ADDR = 00011
    * RAM_DATA = 11111111
    * RAM_DATA(IN) = 11111111
    * RD = 1
    * UART_DATA = 11111111
    * UART_DATA(IN) = 11111111
    * WT = 0
 Other Active Events at This Time:
Time 1960 - event RAM: FLOAT_RAM has occurred
 Pins and Global Registers Before FLOAT_RAM Processed:
    * DATA_LINES = 11111111
    * DATA_LINES(IN) = 01100010
    * INPORT_DO = 11111111
    * OUTPORT_DI = 01100010
    * RAM_ADDR = 00011
    * RAM_DATA = 01100010
    * RAM_DATA(IN) = 01100010
    * RD = 0
    * UART_DATA = 11111111
    * UART_DATA(IN) = 01100010
    * WT = 0
 Other Active Events at This Time:
Time 2010 - event RAM: READ_MEM has occurred
 Pins and Global Registers Before READ_MEM Processed:
    * DATA_LINES = 11111111
    * DATA_LINES(IN) = 11111111
    * INPORT_DO = 11111111
    * OUTPORT_DI = 11111111
    * RAM_ADDR = 01100
```

```
    * RAM_DATA = 11111111
    * RAM_DATA(IN) = 11111111
    * RD = 1
    * UART_DATA = 11111111
    * UART_DATA(IN) = 11111111
    * WT = 0
 Other Active Events at This Time:
Time 2160 - event RAM: FLOAT_RAM has occurred
 Pins and Global Registers Before FLOAT_RAM Processed:
    * DATA_LINES = 11111111
    * DATA_LINES(IN) = 11110001
    * INPORT_DO = 11111111
    * OUTPORT_DI = 11110001
    * RAM_ADDR = 01100
    * RAM_DATA = 11110001
    * RAM_DATA(IN) = 11110001
    * RD = 0
    * UART_DATA = 11111111
    * UART_DATA(IN) = 11110001
    * WT = 0
 Other Active Events at This Time:
Time 2300 - event MARK2: CYCLE has occurred
 Pins and Global Registers Before CYCLE Processed:
    * ACC = 00001111
    * ADDR_LINES = 01100
    * CLOCK = 1
    * DATA_LINES = 11111111
    * DATA_LINES(IN) = 11111111
    * INPORT_DO = 11111111
    * IR = 01100010
    * MAR = 01100
    * MD = 11110001
    * OUTPORT_DI = 11111111
    * PC = 00100
    * RAM_DATA = 11111111
    * RAM_DATA(IN) = 11111111
    * RD = 0
    * RDP = 0
    * RDS = 0
    * RD_SIGNAL = 0
    * UART_DATA = 11111111
    * UART_DATA(IN) = 11111111
    * WT = 0
    * WTP = 0
    * WTS = 0
    * WT_SIGNAL = 0
 Other Active Events at This Time:
Time 2310 - event RAM: READ_MEM has occurred
 Pins and Global Registers Before READ_MEM Processed:
    * DATA_LINES = 11111111
    * DATA_LINES(IN) = 11111111
```

```
    * INPORT_DO = 11111111
    * OUTPORT_DI = 11111111
    * RAM_ADDR = 00100
    * RAM_DATA = 11111111
    * RAM_DATA(IN) = 11111111
    * RD = 1
    * UART_DATA = 11111111
    * UART_DATA(IN) = 11111111
    * WT = 0
 Other Active Events at This Time:
Time 2460 - event RAM: FLOAT_RAM has occurred
 Pins and Global Registers Before FLOAT_RAM Processed:
    * DATA_LINES = 11111111
    * DATA_LINES(IN) = 01011100
    * INPORT_DO = 11111111
    * OUTPORT_DI = 01011100
    * RAM_ADDR = 00100
    * RAM_DATA = 01011100
    * RAM_DATA(IN) = 01011100
    * RD = 0
    * UART_DATA = 11111111
    * UART_DATA(IN) = 01011100
    * WT = 0
 Other Active Events at This Time:
Time 2510 - event RAM: READ_MEM has occurred
 Pins and Global Registers Before READ_MEM Processed:
    * DATA_LINES = 11111111
    * DATA_LINES(IN) = 11111111
    * INPORT_DO = 11111111
    * OUTPORT_DI = 11111111
    * RAM_ADDR = 01011
    * RAM_DATA = 11111111
    * RAM_DATA(IN) = 11111111
    * RD = 1
    * UART_DATA = 11111111
    * UART_DATA(IN) = 11111111
    * WT = 0
 Other Active Events at This Time:
Time 2660 - event RAM: FLOAT_RAM has occurred
 Pins and Global Registers Before FLOAT_RAM Processed:
    * DATA_LINES = 11111111
    * DATA_LINES(IN) = 00000001
    * INPORT_DO = 11111111
    * OUTPORT_DI = 00000001
    * RAM_ADDR = 01011
    * RAM_DATA = 00000001
    * RAM_DATA(IN) = 00000001
    * RD = 0
    * UART_DATA = 11111111
    * UART_DATA(IN) = 00000001
    * WT = 0
```

```
 Other Active Events at This Time:
Time 2800 - event MARK2: CYCLE has occurred
 Pins and Global Registers Before CYCLE Processed:
    * ACC = 00001110
    * ADDR_LINES = 01011
    * CLOCK = 1
    * DATA_LINES = 11111111
    * DATA_LINES(IN) = 11111111
    * INPORT_DO = 11111111
    * IR = 01011100
    * MAR = 01011
    * MD = 00000001
    * OUTPORT_DI = 11111111
    * PC = 00101
    * RAM_DATA = 11111111
    * RAM_DATA(IN) = 11111111
    * RD = 0
    * RDP = 0
    * RDS = 0
    * RD_SIGNAL = 0
    * UART_DATA = 11111111
    * UART_DATA(IN) = 11111111
    * WT = 0
    * WTP = 0
    * WTS = 0
    * WT_SIGNAL = 0
 Other Active Events at This Time:
Time 2810 - event RAM: READ_MEM has occurred
 Pins and Global Registers Before READ_MEM Processed:
    * DATA_LINES = 11111111
    * DATA_LINES(IN) = 11111111
    * INPORT_DO = 11111111
    * OUTPORT_DI = 11111111
    * RAM_ADDR = 00101
    * RAM_DATA = 11111111
    * RAM_DATA(IN) = 11111111
    * RD = 1
    * UART_DATA = 11111111
    * UART_DATA(IN) = 11111111
    * WT = 0
 Other Active Events at This Time:
Time 2960 - event RAM: FLOAT_RAM has occurred
 Pins and Global Registers Before FLOAT_RAM Processed:
    * DATA_LINES = 11111111
    * DATA_LINES(IN) = 10000011
    * INPORT_DO = 11111111
    * OUTPORT_DI = 10000011
    * RAM_ADDR = 00101
    * RAM_DATA = 10000011
    * RAM_DATA(IN) = 10000011
    * RD = 0
```

```
    * UART_DATA = 11111111
    * UART_DATA(IN) = 10000011
    * WT = 0
 Other Active Events at This Time:
Time 3010 - event OUTPORT: PAR_OUTPUT has occurred
 Pins and Global Registers Before PAR_OUTPUT Processed:
    * OUTPORT_DI = 00001110
    * OUTPORT_DO = 00000000
    * WTP = 1
 Other Active Events at This Time:
Time 3210 - event OUTPORT: FLOAT_OUTPUT has occurred
 Pins and Global Registers Before FLOAT_OUTPUT Processed:
    * OUTPORT_DI = 11111111
    * OUTPORT_DO = 00001110
    * WTP = 0
 Other Active Events at This Time:
Time 3300 - event MARK2: CYCLE has occurred
 Pins and Global Registers Before CYCLE Processed:
    * ACC = 00001110
    * ADDR_LINES = 10000
    * CLOCK = 1
    * DATA_LINES = 11111111
    * DATA_LINES(IN) = 11111111
    * INPORT_DO = 11111111
    * IR = 10000011
    * MAR = 10000
    * MD = 00000001
    * OUTPORT_DI = 11111111
    * PC = 00110
    * RAM_DATA = 11111111
    * RAM_DATA(IN) = 11111111
    * RD = 0
    * RDP = 0
    * RDS = 0
    * RD_SIGNAL = 0
    * UART_DATA = 11111111
    * UART_DATA(IN) = 11111111
    * WT = 0
    * WTP = 0
    * WTS = 0
    * WT_SIGNAL = 0
 Other Active Events at This Time:
Time 3310 - event RAM: READ_MEM has occurred
 Pins and Global Registers Before READ_MEM Processed:
    * DATA_LINES = 11111111
    * DATA_LINES(IN) = 11111111
    * INPORT_DO = 11111111
    * OUTPORT_DI = 11111111
    * RAM_ADDR = 00110
    * RAM_DATA = 11111111
    * RAM_DATA(IN) = 11111111
```

```
    * RD = 1
    * UART_DATA = 11111111
    * UART_DATA(IN) = 11111111
    * WT = 0
 Other Active Events at This Time:
Time 3460 - event RAM: FLOAT_RAM has occurred
 Pins and Global Registers Before FLOAT_RAM Processed:
    * DATA_LINES = 11111111
    * DATA_LINES(IN) = 11000011
    * INPORT_DO = 11111111
    * OUTPORT_DI = 11000011
    * RAM_ADDR = 00110
    * RAM_DATA = 11000011
    * RAM_DATA(IN) = 11000011
    * RD = 0
    * UART_DATA = 11111111
    * UART_DATA(IN) = 11000011
    * WT = 0
 Other Active Events at This Time:
Time 3510 - event UART: LOAD_REG has occurred
 Pins and Global Registers Before LOAD_REG Processed:
    * DATA_LINES = 00001110
    * DATA_LINES(IN) = 00001110
    * INPORT_DO = 11111111
    * OUTPORT_DI = 00001110
    * PREG = 00000000
    * RAM_DATA = 11111111
    * RAM_DATA(IN) = 00001110
    * RDS = 0
    * SI = 0
    * SO = 1
    * UART_DATA = 11111111
    * UART_DATA(IN) = 00001110
    * WTS = 1
 Other Active Events at This Time:
Time 3610 - event UART: SER_OUT has occurred
 Pins and Global Registers Before SER_OUT Processed:
    * DATA_LINES = 00001110
    * DATA_LINES(IN) = 00001110
    * INPORT_DO = 11111111
    * OUTPORT_DI = 00001110
    * PREG = 00001110
    * RAM_DATA = 11111111
    * RAM_DATA(IN) = 00001110
    * RDS = 0
    * SI = 0
    * SO = 0
    * UART_DATA = 11111111
    * UART_DATA(IN) = 00001110
    * WTS = 1
```

```
 Other Active Events at This Time:
Time 3800 - event MARK2: CYCLE has occurred
 Pins and Global Registers Before CYCLE Processed:
    * ACC = 00001110
    * ADDR_LINES = 11000
    * CLOCK = 1
    * DATA_LINES = 11111111
    * DATA_LINES(IN) = 11111111
    * INPORT_DO = 11111111
    * IR = 11000011
    * MAR = 11000
    * MD = 00000001
    * OUTPORT_DI = 11111111
    * PC = 00111
    * RAM_DATA = 11111111
    * RAM_DATA(IN) = 11111111
    * RD = 0
    * RDP = 0
    * RDS = 0
    * RD_SIGNAL = 0
    * UART_DATA = 11111111
    * UART_DATA(IN) = 11111111
    * WT = 0
    * WTP = 0
    * WTS = 0
    * WT_SIGNAL = 0
 Other Active Events at This Time:
Time 3810 - event RAM: READ_MEM has occurred
 Pins and Global Registers Before READ_MEM Processed:
    * DATA_LINES = 11111111
    * DATA_LINES(IN) = 11111111
    * INPORT_DO = 11111111
    * OUTPORT_DI = 11111111
    * RAM_ADDR = 00111
    * RAM_DATA = 11111111
    * RAM_DATA(IN) = 11111111
    * RD = 1
    * UART_DATA = 11111111
    * UART_DATA(IN) = 11111111
    * WT = 0
 Other Active Events at This Time:
Time 3960 - event RAM: FLOAT_RAM has occurred
 Pins and Global Registers Before FLOAT_RAM Processed:
    * DATA_LINES = 11111111
    * DATA_LINES(IN) = 01100011
    * INPORT_DO = 11111111
    * OUTPORT_DI = 01100011
    * RAM_ADDR = 00111
    * RAM_DATA = 01100011
    * RAM_DATA(IN) = 01100011
    * RD = 0
```

```
    * UART_DATA = 11111111
    * UART_DATA(IN) = 01100011
    * WT = 0
 Other Active Events at This Time:
Time 4010 - event RAM: WRITE_MEM has occurred
 Pins and Global Registers Before WRITE_MEM Processed:
    * DATA_LINES = 00001110
    * DATA_LINES(IN) = 00001110
    * INPORT_DO = 11111111
    * OUTPORT_DI = 00001110
    * RAM_ADDR = 01100
    * RAM_DATA = 11111111
    * RAM_DATA(IN) = 00001110
    * RD = 0
    * UART_DATA = 11111111
    * UART_DATA(IN) = 00001110
    * WT = 1
 Other Active Events at This Time:
Time 4300 - event MARK2: CYCLE has occurred
 Pins and Global Registers Before CYCLE Processed:
    * ACC = 00001110
    * ADDR_LINES = 01100
    * CLOCK = 1
    * DATA_LINES = 11111111
    * DATA_LINES(IN) = 11111111
    * INPORT_DO = 11111111
    * IR = 01100011
    * MAR = 01100
    * MD = 00000001
    * OUTPORT_DI = 11111111
    * PC = 01000
    * RAM_DATA = 11111111
    * RAM_DATA(IN) = 11111111
    * RD = 0
    * RDP = 0
    * RDS = 0
    * RD_SIGNAL = 0
    * UART_DATA = 11111111
    * UART_DATA(IN) = 11111111
    * WT = 0
    * WTP = 0
    * WTS = 0
    * WT_SIGNAL = 0
 Other Active Events at This Time:
Time 4310 - event RAM: READ_MEM has occurred
 Pins and Global Registers Before READ_MEM Processed:
    * DATA_LINES = 11111111
    * DATA_LINES(IN) = 11111111
    * INPORT_DO = 11111111
    * OUTPORT_DI = 11111111
    * RAM_ADDR = 01000
```

```
    * RAM_DATA = 11111111
    * RAM_DATA(IN) = 11111111
    * RD = 1
    * UART_DATA = 11111111
    * UART_DATA(IN) = 11111111
    * WT = 0
 Other Active Events at This Time:
Time 4460 - event RAM: FLOAT_RAM has occurred
 Pins and Global Registers Before FLOAT_RAM Processed:
    * DATA_LINES = 11111111
    * DATA_LINES(IN) = 00000111
    * INPORT_DO = 11111111
    * OUTPORT_DI = 00000111
    * RAM_ADDR = 01000
    * RAM_DATA = 00000111
    * RAM_DATA(IN) = 00000111
    * RD = 0
    * UART_DATA = 11111111
    * UART_DATA(IN) = 00000111
    * WT = 0
 Other Active Events at This Time:
PROCESSOR HALTED
GCL> @EXRAM.GCL
GCL> set module = ram
GCL> set cbase = D
GCL> ex mem[0]
    * RAM: MEM = 10000010
GCL> ex mem[1]
    * RAM: MEM = 10000100
GCL> ex mem[2]
    * RAM: MEM = 01100011
GCL> ex mem[3]
    * RAM: MEM = 01100010
GCL> ex mem[4]
    * RAM: MEM = 01011100
GCL> ex mem[5]
    * RAM: MEM = 10000011
GCL> ex mem[6]
    * RAM: MEM = 11000011
GCL> ex mem[7]
    * RAM: MEM = 01100011
GCL> ex mem[8]
    * RAM: MEM = 00000111
GCL> ex mem[9]
    * RAM: MEM = 00000000
GCL> ex mem[10]
    * RAM: MEM = 00000000
GCL> ex mem[11]
    * RAM: MEM = 00000001
GCL> ex mem[12]
```

```
    * RAM: MEM = 00001110
GCL> ex mem[13]
    * RAM: MEM = 00000000
GCL> ex mem[14]
    * RAM: MEM = 00000000
GCL> ex mem[15]
    * RAM: MEM = 00000000
GCL> set cbase = B
GCL> QUIT
```

REFERENCES

1. Armstrong, J. R., "Chip Level Modeling and Simulation", Simulation, Oct. 1983, pp. 141-148.
2. Armstrong, J. R., "Chip Level Modeling of LSI Devices", IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, Oct. 1984, pp. 288-297.
3. Ellis, M. D., A General Purpose Digital Network Simulator, Masters Thesis, EE Department, VPI and SU, 1979.
4. Devlin, D. E., A Chip Level Multimodule Logic Simulator, Masters Thesis, EE Department, VPI and SU, June 1981.
5. Armstrong, J. R. and F. G. Gray, Microprocessor Self-test, Final Report for RADC Contract F-30602-80-c-0200, October 1981.
6. Puthenpurayil, V., Functional Level Modeling of Digital Devices, Masters Thesis, EE Department, VPI and SU, September 1982.
7. Sathe, S. K., Functional Level Fault Simulation in LSI Devices, Masters Thesis, EE Department, VPI and SU, June 1982.
8. Gupta, A. K. and J. R. Armstrong, "Functional Logic Modeling and Simulation on P.C.", Proceedings of the 22nd Design Automation Conference, July 1985, pp. 133-135.

9. Kerr, J. M., Compiled Code, Chip Level Simulation Using High Level Hardware Description Language, Maters Thesis, EE Department, VPI and SU, June 1984.

10. Jensen, K. and N. Wirth, PASCAL User Manual and Report, Springer-Verlag, New York, 1974.

11. G. L. Langdon, Jr., Computer Design, Computech Press Inc., San Jose, California, 1982.

12. Horowitz, E. and S. Sahni, Fundamentals of Data Structures in PASCAL, Computer Science Press, 1984.

13. VAX Architecture Handbook, Digital Equipment Corporation, 1981.

14. N.2 ISP' User's Manual, Endot Inc., August 1985.

15. VHDL User's Manual, Document No. IR-MD-065-1, August 1985.

16. VAX-11 Utilities Reference Manual, Digital Equipment Corporation, 1983.

17. Siewiorek, D. P., Bell C. G., and A. Newell, Computer Structures: Principle and Examples, McGraw-Hill Inc., 1982.

18. Szygenda, S. A. and E. W. Thompson, "Digital Simulation in a Time-based Table Driven Environment", Computer, Vol. 8, No. 3, pp. 24-48, March 1975.

19. Alia, G., Ciompi, P., and E. Martinelli, "LSI Components Modeling In a Three-valued Functional Simualtion", Proceedings of the 15th Design Automation Conference, June 1978, pp. 428.

The vita has been removed from the scanned document