THE SLIDE SIMULATOR:
A FACILITY FOR THE DESIGN AND ANALYSIS OF COMPUTER INTERCONNECTIONS

by

Arthur H. Altxnan & Alice C. Parker

DRC-18-23-80

August 1980

Department of Electrical Engineering
Carnegie-Mellon University
Pittsburgh, PA 15213

This research has been supported by the U.S. Army Research Office, under grants #DAAG29-76-G-0224 and #DAAG29-78-G-0070, and by the Department of Electrical Engineering, Carnegie-Mellon University.

Table of Contents

1. Introduction 2
2. The Nature of I/O and Interface Operation 3
3. Novel Constructs of SLIDE 4
    3.1. The Process 4
    3.2. The DELAY Statement and Parallel Statement Execution 5
    3.3. Other I/O Related Primitives 5
    3.4. Requirements For a I/O Hardware Descriptive Language 6
4. An Overview of the Simulator 6
    4.1. Operational Overview 7
    4.2. Simulator Commands 9
5. The Multi-Level Simulator: Structure and Function 13
    5.1. Data Structures of the Simulator 13
    5.2. The Module Interconnections 15
6. The SLIDE Simulator: Structure and Function 18
    6.1. SLIDE Simulation Concepts 18
    6.2. General Functioning in SLIDE Simulations 19
7. The Simulation Test Case 20
    7.1. Peripheral Device Description 21
    7.2. Simulation Test Results 24
8. Conclusions and Future Research 27
9. Acknowledgments 27
L Test Simulation Traces 30

List of Figures

Figure 4-1: Example Interconnection Scheme 7-
Figure 4-2: Producing a Runnable SLIDE Simulator 9
Figure 4-3: Results of example ADD commands 9
Figure 5-1: **Partitioning of a Digital Device in the Multi-level Simulator** 13
Figure 5-2: **Representation of an Inverter Module** 15
Figure 5-3: **Representation of Device Interconnection by Wire** 15
Figure 7-1: **SLIDE Simulator Test Case** 21
Figure 7-2: **SLIDE Process Structure of the Peripheral Device Model** 21
Figure 7-3: **Daisychaihing of UNIBUS NPG Line** 25
Figur* 9-1: **First Test Run : Trace 1** 31
Figure 9-2: **First Test Run : Trace 2** 31
Figure 9-3: **First Test Run : Trace 3a** 31
Figure 9-4: **First Test Run : Trace 3b** 31
Figure 9-5: **First Test Run : Trace 3c** 31
Figure 9-6: **First Test Run : Trace 3d** 31
Figure 9-7: **First Test Run : Trace 3e** 31
Figure 9-8: **Second Test Run : Trace 1** 31
Figure 9-9: **Third Test Run : Trace 1** 31
Figure 9-10: **Third Test Run : Trace 2** 31

Abstract

Interconnection design can have a profound effect on the price and performance of a digital system. This paper will describe a new simulation facility that is designed to allow the user to describe an interconnection strategy, and simulate the behavior of the interconnected system.

The user first writes a description of the interconnections using the hardware descriptive language SLIDE. SLIDE has been specifically designed for interconnection and interface description; the UNIBUS, for example, has been described in SLIDE. The description is then compiled into SIMULA code, and linked by the user with other modules which probabilistically model the hardware that would drive the interconnections. The simulation then proceeds under interactive user control.

Thus, this simulator provides the capability to devise, debug, and evaluate digital interconnection schemes.

L Introduction

Multiprocessing research and development has caused an increased interest in I/O and Interconnection architectures. In fact, it has become important to document, simulate, and even formally verify portions of entire systems, including their interconnections.

Naturally, the more detailed the interconnection description becomes, the more accurate the simulation can be and the more information the description can contain. With current simulation techniques, interconnections and their interfaces can be described and simulated accurately at both the gate and circuit levels. Unfortunately, these techniques have precluded simulation of large interconnected systems, since the resulting simulation programs have been overly large and slow.

For this reason, existing lowHevel descriptive languages do not provlde the kind of hardware description needed for the above task; a description at a higher level is needed. An obvious solution is to use a register-transfer language.

There is strong motivation for constructing behavioral descriptions of I/O hardware.[1] Certainly, behavioral descriptions can convey the overall operation of interfaces to the reader better than structural descriptions, since much of the unnecessary detail is eliminated Simulations proceed more rapidly, and can encompass larger systems.

Research is currently underway at Carnegie-Mellon University to use such a behavioral language called SLIDE[2] for interface and interconnection and simulation description. The SLIDE simulator is the subject of this paper.

The SLIDE simulator allows description and simulation of interconnection schemes like the UNIBUS[3] and the D-bus [1J. Eventually behavioral descriptions of processors and other functional units will be linked to the interconnection descriptions, so that entire systems can be interconnected and simulated.

In the process of designing SLIDE, the following design goals were kept in mind:

'BehevioreJ descriptions differ from structure! discretions because they describe only ths functions of the hordwer* •nd not the hardware itself. Storage location! and reftster-transfers which •xist *m* ths hardware msy bo absent from the behevforsl description; control hsrdwsrs ts implicit rsthsr thsn •xpiicit A discussion of this distinction is found in (21

[2] 8UDE (formerly GLIDE) is in acronym for Structured Unfusfe for Interfece Description snd Evsiustion [17J

[3] UNIBUS is s ref tstered trsdemsrfc of Dtf its* Equipment Corporation.

-To provide a language which could be used to describe interface hardware behaviorally In a stand-alone fashion; the language should not depend on timing diagrams or state diagrams for completeness.

-To provide a language which is simple and not overburdened with obscure constructs and primitives. It should be logically consistent in semantics and syntax.

Previous research in the area of interface and I/O description has been done [5, 11, 12, 15, 16]. However, this research has either produced gate level languages, circuit level languages, or incomplete proposals such as the port descriptions of Bell and Newell [5] and Curtis [7]. A recent proposal [12] does have some useful control constructs which are similar to those of SLIDE. Further discussion of these topics can be found in [14].

There are already a number of general-purpose hardware descriptive languages; some, like ISPS [3, 4] and DDL [9] have been exercised and a software base exists. Therefore it is difficult to justify the development of yet another hardware descriptive language and simulator. However, for reasons of either efficiency, ease of use or missing semantics, current hardware descriptive languages do not provide the capabilities needed for the interconnection descriptions being considered at CMU.

Section 2 of this paper describes the problems associated with I/O and interface descriptions, and section 3 introduces the required capabilities of an I/O simulator. Section 4 is an overview of the simulator, sections 5 and 6 present salient features of the simulator, and section 7 contains results from an example simulation.

2. The Nature of I/O and Interface Operation

Consider the following example system configuration which illustrates some basic aspects of interface and I/O operation. A single bus connects a device, controller, CPU, and memory. The device controller is reading data in and writing it to memory; at the same time, the CPU is executing a program, and therefore accessing memory for instructions and data. At any time, either the device controller or the CPU might be transferring information across the bus. At the same time, either or both might be requesting the bus for future transactions. Between the two devices there are four processes, two for bus requests and two for bus transactions. At any time a maximum of three can be executing (only one bus transaction can occur at a time). This illustrates an inherent property of interface and I/O operation - complex control flow. More precisely:

-There can be multiple sequences of events executing concurrently and independently of each other.

-An event in one sequence can alter the execution order of another sequence.

-The time steps between events can be different for different sequences.

-The onset of execution of one sequence can initiate or terminate another sequence.

Each sequence of events in reality represents an independent control environment or a finite state machine; we shall refer to these sequences hereafter as *processes.*

Thus, a language designed to describe this genre of control flow must possess powerful and unconventional control constructs. The simulator must provide:

-Priority orderings between processes.

-Timing dependencies, timeouts, and data I/O at fixed bit rates.

-Execution of interprocess synchronization primitives such as *signal* and wait.

-Initiation, termination and suspension of processes.

-Event sequencing internal to a process.

-Communication between processes.

In addition to these, the simulator should execute operations common to I/O such as bit manipulation, code conversion, FIFO buffering, parity and error checking, synchronous I/O, and combinational logic[4]

3. Novel Constructs of SLIDE

This section introduces the novel constructs of the SLIDE language. These include the *process* (a construct for nonprocedural execution), the *delay* statement (a timeout construct)* and other I/O related primitives.

3.1. The Process

In the same way that routines are the central unit of execution in most programming

[4]A moro oxUnstv* ditcuMton of those primitfcwt CM bo found in [13].

languages, processes are the central unit of execution in SLIDE descriptions.^ A process is an independent executing environment — a piece of hardware such as a device controller or a bus arbiter. Within each process, variables (registers, lines, etc.) can be declared, and other processes (called *subprocesses)* can be defined Consequently, a SLIDE description is composed of layers of nested processes (much like an ALGOL program is composed of layers of nested routines).

A SLIDE description consists of one main *process* which syntactically encompasses all other subprocesses (much like an ALGOL program consists of one main program which encompasses all subroutines). Variables global to the entire description are declared within the main process. Variables which are local to a subprocess are declared within that subprocess.

Since each process describes a piece of hardware, each is an independent executing environment, and all processes which are executing do so in parallel. Processes which need to communicate with each other can do so by using global variables (e.g. by asserting a shared line).

Processes are started (called *Initialization)* nonprocedural^. When each process (except for the main process) is defined, the conditions-under which it is to be initiated are given. A *priority mechanism* exists which can be used to allow some processes to terminate execution or mutually exclude execution of others.

3.2. The DELAY Statement and Parallel Statement Execution

Aside from the usual statements such as *assignment, if-then-else, loops,* and *subroutines,* SLIDE has a powerful *delay* statement which allows delays and timeouts to be described. This statement is used to delay the execution of a process until some condition occurs and/or a timeout occurs. Within a process, complex statements can be executed in parallel or sequentially.

3.X Other I/O Related Primitives

Otter I/O related primitives which have been incorporated into the SLIDE language are those to:

-describe transitions (from low-to-high or high-to-low) as well as levels (low or high).

5We UN "description" h»r» and not "profram" to amphaaiza that SLIDE *describes* tha opanrttotr of a piccr of hardware. Corr*spondimfly, w* UN "axacuta" to moan "tha operation of tha actions daacribad."

-declare combinational logic via the *comb* declaration. The output of this logic is expected to change asynchronously after the input changes.

-declare synchronous lines via the *sync* declaration.

-declare FIFO buffers via the *buffer* declaration.

-declare associative memory tables via the *table* declaration.

-do I/O related operations such as packing and unpacking bit slices.

3.4. Requirements For a I/O Hardware Descriptive Language

This section has discussed some of the constructs in SLIDE which make it useful for describing I/O hardware. We feel that these constructs should necessarily be included in any I/O hardware descriptive language. To summarize, these constructs include:

-A nonprocedural executing environment such as the SLIDE process. Nonprocedurality and priorities are important in I/O hardware descriptions where many processes do not execute *until some condition becomes true.* Examples of this are interrupts, bus arbiters, device controllers, etc.

-A delay and timeout construct such as the SLIDE delay statement. This goes hand in hand with the nonprocedurality discussed above. It allows a process to delay execution until some condition becomes true, subject to a timeout condition.

-An ability to specify actions in parallel as well as sequentially. The need for this is obvious, and most hardware descriptive languages provide this.

-I/O related primitives such as those discussed in Section 3.3.

4. An Overview of the Simulator

The SLIDE simulator is a tool that will answer a number of questions a systems designer might raise with respect to an interface or interconnection strategy. These questions include:

-Does the design function as planned

-Is the design sensitive to wirelengths and other timing dependencies

-What is the effect of occurrence of exception conditions on operation of the proposed design

-What are the bottlenecks which limit speed of operation of the design

-What is the average device latency and is the maximum allowable latency exceeded

*Is resource allocation with the proposed strategy vulnerable to starvation and/or deadlock problems, and does it add an "acceptable" amount of overhead or not

It should be noted here that some properties of interconnection strategies (deadlocks, for example) may only be discovered through formal verification procedures, such as that proposed by Wallace [18]. However, simulation does provide a design aid which, if properly constructed and used, can provide a designer significant assistance in answering the above questions.

4.1. Operational Overview

The SLIDE slmulator is a subset of a more powerful, multi-level simulator currently under construction at Carnegie-Mellon. This latter simulator is somewhat similar to one reported on by Hill [10]. It is designed to allow a user to simulate digital systems whose components may be any combination of gate and register-transfer level modules.

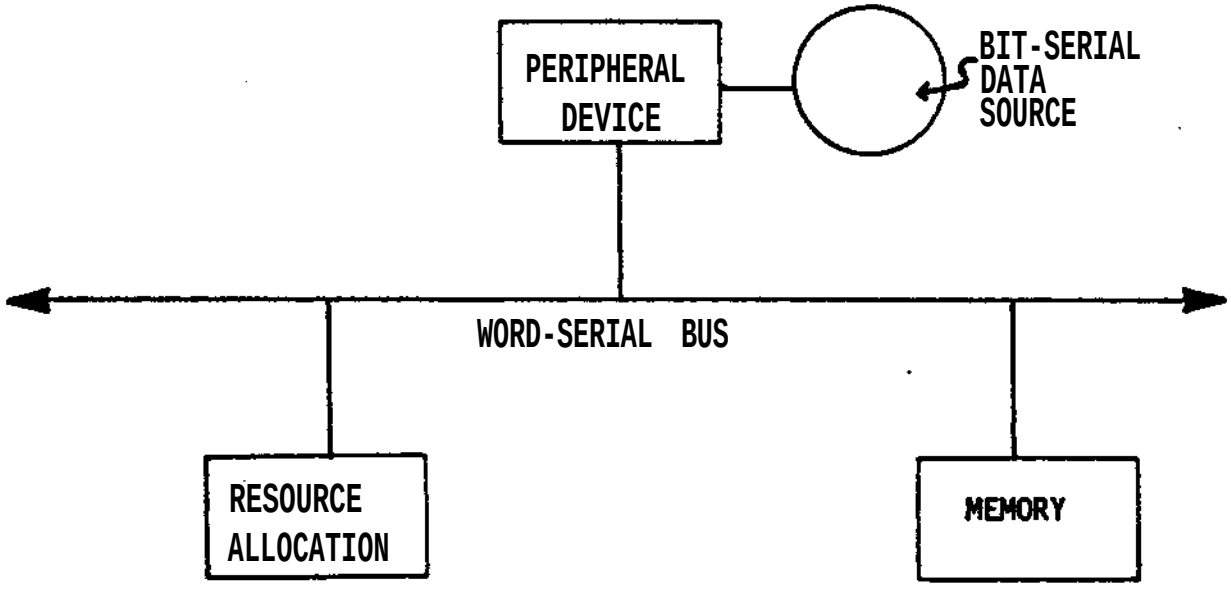
The language of choice for the simulator was SIMULA-67, an ALGOL based discrete-event simulation language [6J At present, the user can write descriptions of digital hardware in SIMULA and incorporate them into the simulator environment. Then, through interactive commands, the user can wire together various functional modules, and run simulations on the digital system thus modelled.

The SLIDE simulator uses this simulator as a "core", allowing the user to simulate interconnections and interfaces. We illustrate the SLIDE simulator and its relationship to the core by the following example.

Suppose the user wants to simulate an interface between a bit-serial data link and a word-serial bus. Data comes from a source attached to the data link. It is transferred through the interface to the word-serial bus, and into a memory module on the bus. A priority arbitration device is responsible for resource allocation on the word-serial bus. This example is shown in Figure 4-1.

To model this system, the user would write four SLIDE descriptions or modules: one to describe the interface, one the source, one the memory, and one the resource allocator. As shown in Figure 4-2 for a two-description case, the user would compile each SLIDE source file, then enter the compiler outputs into a preprocessor. The preprocessor would produce a single SIMULA source program that, when linked with a pre-compiled core, would be a runnable simulator. It would contain four functional modules, one for each SLIDE description. The user could then wire together the modules via interactive commands, and proceed to test

Figure 4-1: Example Interconnection Scheme



the behavior of the configuration.

One interesting aspect of the preprocessor [8] is that the SIMULA code generated from the SLIDE modules does not have to be properly structured syntactically. For example, declarations can be generated and inserted, on the fly, as the SLIDE descriptions are compiled. The preprocessor will clean up and reorder the SIMULA code for the SIMULA compiler.

Various properties of each SLIDE description may be parameterised. This is done by allowing numbers in a SLIDE description to be replaced by special identifiers called *simulation time parameters* (STP). STP's are bound interactively, by the user,for each instance of a SLIDE functional module in the simulator.

4.2. Simulator Commands

The capabilities of the SLIDE simulator are only partly reflected in the commands available to the user at present. This is because the SLIDE simulator is still under development, especially at the user interface. The following list of currently implemented commands therefore does not represent the maximum performance level of the program.

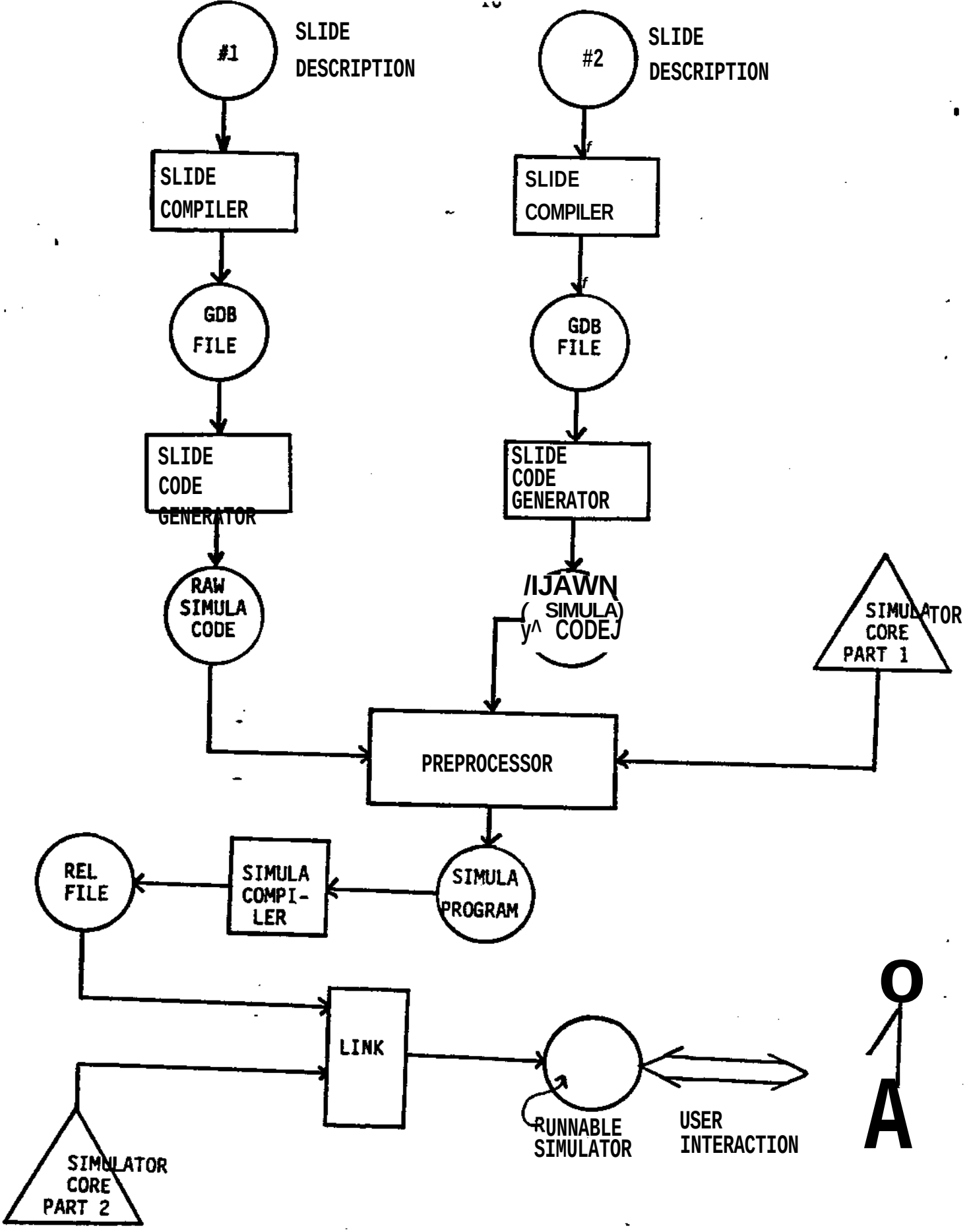
ADO <label>: <device> <parameters> <name> <name> ... <nam*>;

e. g. ADO BILL: INVERTER Ul U2;
ADD SUE : DELAY U2 U3;

The ADD command creates the data structures for the ports of <device>. The resulting module is labelled to distinguish it from other instances of <device> in the simulator. <parameters> is optionally used by non- SLIDE functional modules, for passing of device-related parameters. The remainder of the command field does the wiring for the ports of <device>. A one-to-one correspondence exists between each <name> and a port of the device, by the left to right position of <name>. For each name, a wire model is created and labelled with <name>. Then it is connected to the corresponding port. So for BILL in the above example, Wl is the name of a wire that is connected to port 1. W2 is the name of a wire that is connected to port 2, and to port 1 of SUE. The conceptual results of these two ADO commands are shown in Figure 4-3.

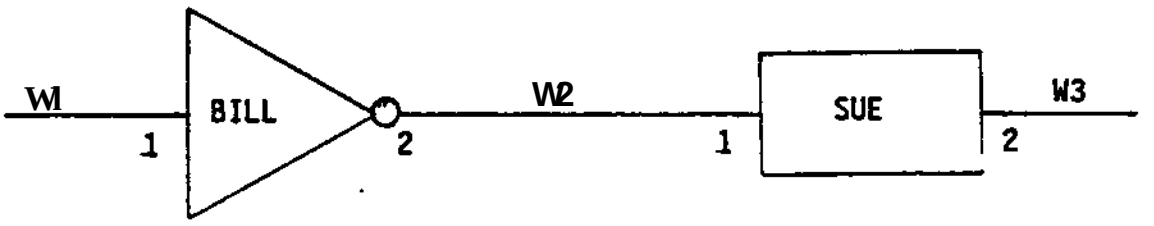
ALL

This command prints out the accumulated connection information from all the ADD's that have been done so far.



Figur* 4-2: Producing a Runnabie SLIDE Simulator

Figur* 4-3: Results of example ADO commands



DUMP <filename>

This command dumps out the accumulated connection information to a file called <filename>.

GET <filenam#>

This command retrieves the connection information in <filename> and implements it.

So the user could build up a test interconnection using AD&s% then scam it using DUMP. At any time* even on a different simulation run, he or she could recreate the interconnections by GETting the appropriate file.*

PROBE <wire>

e. g, PROBE ABC
-PROBE XYZ

This command turns on a trace for the named wire. Whenever this wire is written to, the state of the wire and the current simulated time will be output to the terminal. -PROBE <wire> turns off the trace on <wire>.

WHAT<label>
m. g. UHAT SUE

This command causes the entire state of the device <label> to be output.

SIMULATE

This causes the actual functional model of each device specified in the ADD commands to be created. It is at this point that any simulation time parameters specified in the original SLIDE description are bound by the user.

GO <number>

Run the simulator for <number> microseconds.

UNTIL <number>

Run the simulator until simulated time equals <number>.

FREEZE

This command causes the core image of the simulation program to be saved. This allows easy restarts for simulation tests that start at a certain point, but then are personalized via various parameter combinations, for example.

This brief overview of the SLIDE simulator is followed first by a description of the core simulator and then by a discussion of the SLIDE simulator itself.

5. The Multi-Level Simulator: Structure and Function

The multi-level simulator provides a decentralized, dynamically alterable environment for the interconnection and simulation of digital systems. This simulator structure is based on the discrete event-driven simulation primitives provided by SIMULA-67 [6J

A simulation program written in SIMULA uses special coroutines called *processes.* For a *process* to execute, it must be scheduled by placing it in a special linked list called the *event list.* The list is ordered by the simulation time associated with each process. The process at the front of the list is due to resume execution, and the time associated with this process is considered to be the "current" simulation time. A process can be removed from the list and rescheduled at a later time. Thus, time moves ahead in jumps, and the simulation is event-driven.

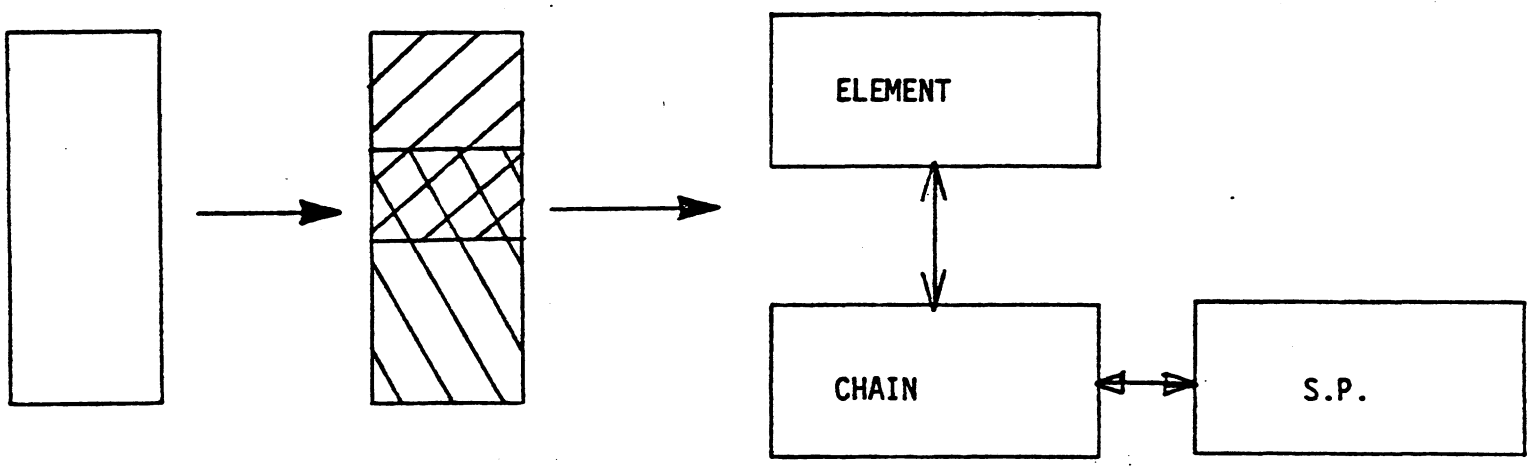
5.1. Data Structures of the Simulator

The structure of this simulator is characterized by three classes of dynamic data structures and the operations performed on them. These are: The *Element,* the *Chain* and the *Simulation Process* (SP). Together, the Element, Chain and the SP completely specify the behavior of a given digital device or circuit module.

The Element and the Chain constitute the common external characteristics of the device modules; the internal structural and functional differences have been abstracted away. The SP contains the information internal to a device module - the meat of a device description. It is a general data structure which references a number of SIMULA processes. Figure 5-1 illustrates the partitioning concept.

The Element is designed as a passive vehicle for the interconnection of device modules. Each Element contains a set of records representing the "ports"[9] of a device. Ports can be connected to other ports, and the state of the data represented by the ports of a device module indicates the externally visible state of the device.

Figure 5-1: Partitioning of a Digital Device in the Multi-level Simulator



The active part of the interconnection mechanism resides in the Chain. One Chain is associated with each element; the chain is responsible for the actions and reactions of a device with respect to its ports. The Chain also acts as an intermediary between the SP and the Element, which do not interact directly. The chain is implemented as a coroutine, whose actions are performed in zero time and *are* invisible to the SIMULA scheduler.

The partitioning of the digital circuit device modules into internal behavior and external structure has an advantage* The external descriptions of, say a flip-flop and a microprocessor, have many common features, and the internal differences are not visible during the interconnection proess. Thus modules of arbitrary complexity and level of abstraction may be interconnected in a consistent fashion because of the structuring of device models in the simulator. An example of a simple device module representing an inverter is given in Figure 5-2 . The SP is shown as SIMULA code, the Element is represented as an abstract data structure, and the Chain is illustrated in its role as interface between the two.

5.2. **The** Module Interconnections

Modules can be interconnected in two ways in this simulator - directly and via special *wire* modules. Direct connections allow the data records at one port to be directly accessible by all connected ports. This concept is useful, for example, when a CPU module has been described as a number of submodules, whose ports exist only as an abstraction. (A Z80 CPU has been described in this manner [8]). This method of interconnection allows structured design and pretesting of each submodule.

The more common connection method for large system simulations is the wire connection, meant to correspond to the usual physical interconnection of digital devices. A wire can be considered a degenerate case of a digital module, having no SP or explicit ports. Through its chain, the wire houses the actual wire data, identification information and procedures which allow its logical behavior to be modelled An example of this type of connection is an open-collector bus, since the data presented to the bus wires at a port might not reflect the actual logic values on the bus. Thus, a wire connection is modelled as the direct access of ports to the same wire device, not to each other. Wire types are distinguished by logical behavior, data representation , and synchrony.

The user can interactively wire together the components of his system, producing a structure like that shown in Figure 5-3 . Changes in the system being simulated can be made by adding components without halting program execution.

During execution, the SP*s may write to ports by depositing data in the appropriate wire

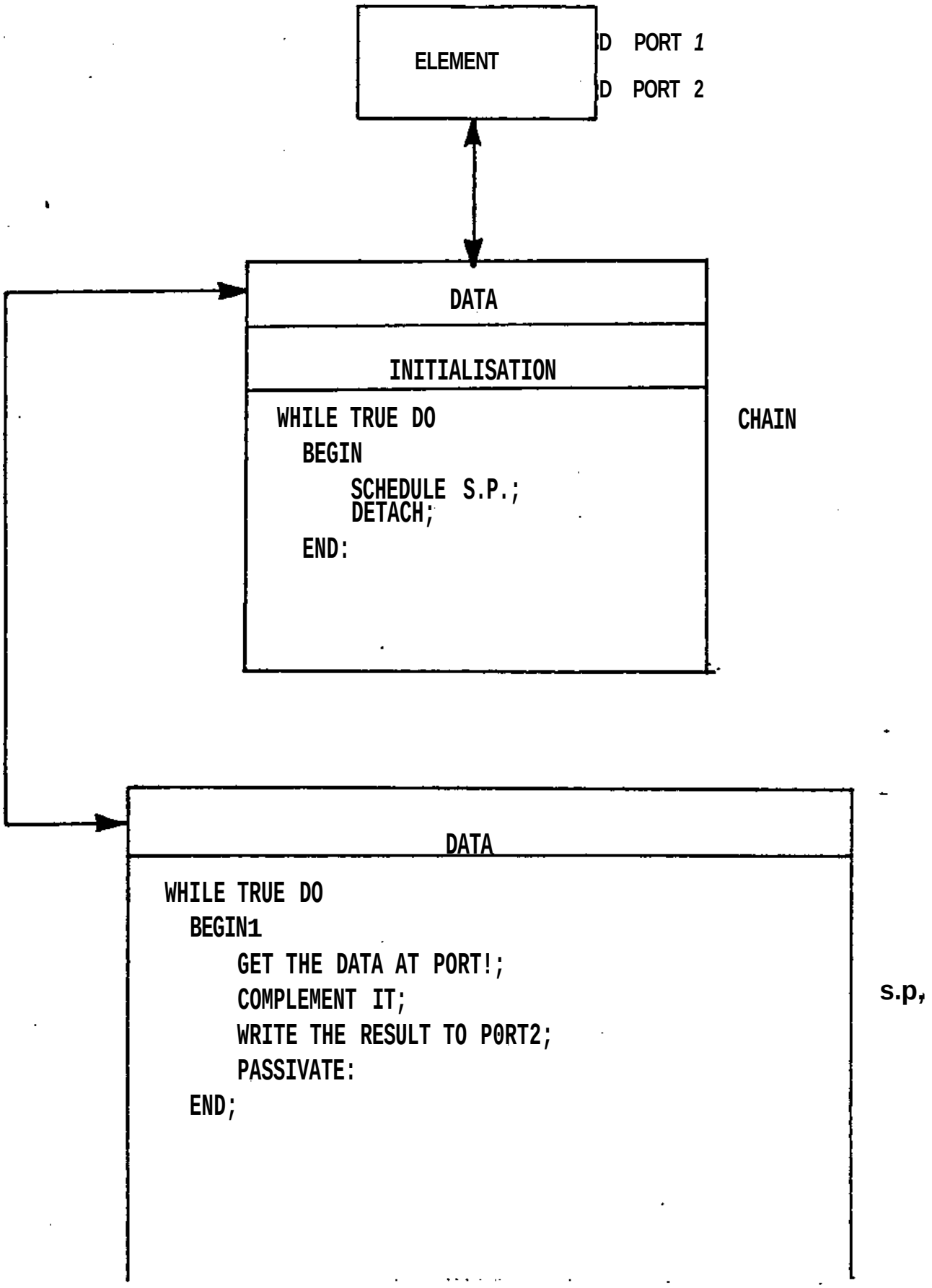
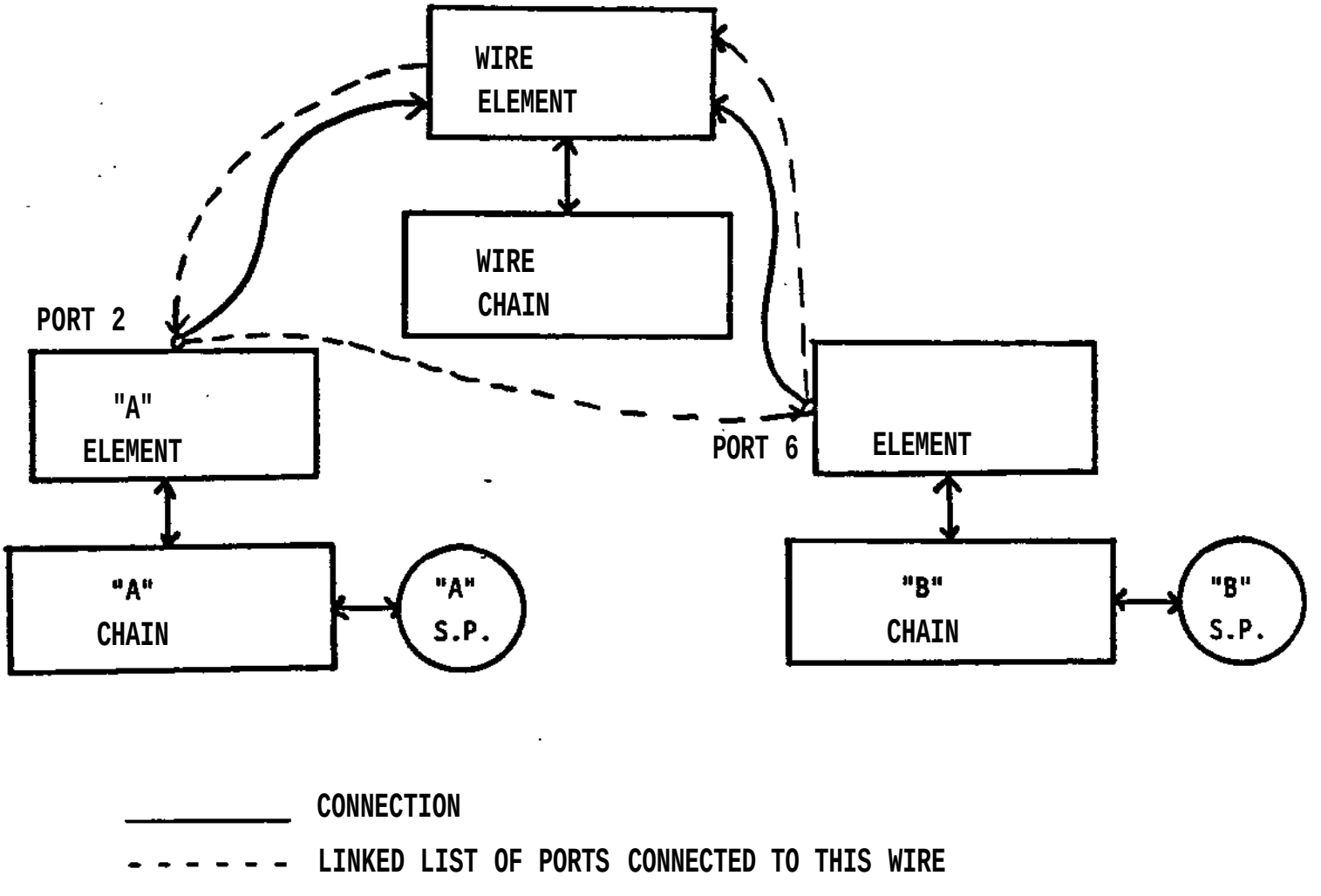


Figure 5-2: Representation of an Inverter Module

Figure 5-3: Representation of Device Interconnection by Wire



device, via a series of references. The wire Chain coroutine is called, and it proceeds through a linked list of ports associated with the wire, activating the Chain associated with each port. This gives each relevant device the opportunity to schedule processes, and it proceeds in zero simulation time. Delays are introduced only by the scheduling of processes related to a device module and by explicit delays within each device module's SP.

6. The SLIDE Simulator: Structure and Function

At this point the reader should recall the example that was used in the overview of the simulator in Section 4. To properly explain what happens after the test interconnection has been built and the GO or UNTIL commands have been issued requires that the underlying concepts of SLIDE simulation within the multi-level simulator core be explained

6.1. SLIDE Simulation Concepts

Although SLIDE devices will look like any other device to the core simulator, in terms of their structure and their interconnection to other devices, there are significant complications caused by the fact that SLIDE device modules are simulating the behavior of SLIDE descriptions.

First, as we have said, each SLIDE module is a mapping of an independent SLIDE description, each having its own SLIDE process structure. The process structure and hence the particular scheduling constraints of a given SLIDE device are independent of all other SLIDE devices. The multi-level simulator has already been seen to run in a decentralised fashion; each device executes its own actions autonomously. Then it is consistent to make each SLIDE device responsible for the scheduling of its associated processes. Therefore, the SP of a SLIDE device has the control structures to handle the scheduling task.

Second, SIMULA processes as such are not endowed with sufficient scheduling data structures to correctly model SLIDE processes. A SLIDE process within a SLIDE description is a member of a complex priority tree specified by the static nesting of the process in the description and by explicit priority numbering. Initiation and termination of a SLIDE process must always be referenced to the priority tree. This is clearly beyond the scope of a SIMULA process. The concept of termination itself , implying the automatic garbage-collection and recreation of a process and all its subprocesses, does not exist with SIMULA processes; it must be built up.

Third, each SLIDE scheduler within a SLIDE module needs to be invoked as the result of many different events. For example, a process can be started or terminated due to a change in a SLIDE variable. A process can also be initiated or terminated as a side-effect of some

other process initiation/termination, due to the priorities, even if these processes do not explicitly communicate at all. The SLIDE DELAY statement adds another complication by allowing a process to reschedule itself at any time, and to be woken up on some set of conditions[6] (possibly subject to a timeout). Hence extra control structures are needed to allow a simulating SLIDE device to evoke the scheduler at all significant events.

Fourth, we must consider the timing task in the simulation of a SLIDE description. Maintaining time fidelity between communicating SLIDE processes is crucial for interconnection simulations. Fortunately however, it is not necessary for every SLIDE action to occur at the absolutely correct simulation time. Part of the compilation process includes insertion of *proper* delays at correct points with respect to significant events, to reflect the true passage of time. By not having to insert delays after non-significant events, such as execution of statements having no direct or side-effect on process initiation or termination, the simulation is speeded up at no cost to modelling accuracy.

Fifth, The multi-level (core) simulator concept of "wire" does not constitute a complete solution to the modelling problem for SLIDE lines. Although[1] some SLIDE lines (those declared in a SLIDE *main* process) are global, some SLIDE lines are entirely local device descriptions. These lines must be modelled within the SP. Even though each global line corresponds to a port of the SLIDE module, so that "wires"* are used for communication across global lines, some behavior of the global line is modelled internally to the SP. What results then is is a distribution of line representation between the exterior and interior of a SLIDE module. The necessity for this split in representation becomes more evident when one realises that SLIDE lines are functionally more powerful than "wires". The actions of SLIDE lines can have wide effects on SLIDE process execution, for example. As we shall see, this implies that SLIDE lines be modelled as special SIMULA processes, and this goes beyond the multi-level simulator "wire", which is designed to be independent of the SIMULA time axis.

6.2. General Functioning in SLIDE Simulations

With these important concepts presented, we now describe the general functioning of a SLIDE device simulation. After variable initialisation, all SLIDE processes are given a chance to begin executing if they can. According to the semantics of the SLIDE language, a process may start executing whenever

Uts declared initialisation conditions become true, AND

---

[6] Thew conditions art arithmetic tnd/or lofic txprttstòftt of SLIDE vtriabfes

2.all the processes of which it is a subprocess are executing, AND

3.no process at the same "process level" is executing and has a higher priority.

Processes that could have started but for the second two points above are put into a special linked list. As each SLIDE process starts its execution, it checks the list to see if any of the members can start up as a result of its initialisation. Also at this point, any executing process which is at the same "process level" ,but of a lower priority than this process, is terminated.

As a SLIDE process executes its actions, variables such as lines and registers will be accessed. When a variable is written to, it is responsible for checking if any of the relevant expressions of it is contained in are now true,[7] and if they are, evoking the SLIDE scheduler.

It is possible that a process, by altering the data on a SLIDE variable, may cause its own termination, for example, by allowing a higher priority process to start. If we are to ensure an orderly execution of the SLIDE scheduler in such cases, the actions of the SLIDE variable in this instance should be implemented as a special SIMULA process, as opposed to a procedure called up by the SLIDE process. The variable is then free to cause what changes It pleases on the SIMULA time axis without needing to return to a calling process that may have been garbage-collected.

As each SLIDE process completes its actions *(terminates),* it is entered into the special linked list mentioned above. The list is once again checked to see if any of its members can begin execution.[0]

It should be noted that whenever a member of the list is being checked, it will be removed from the list if its Boolean initiation conditions are found to be false. This ensures that members of the list are only those that are likely to be started due to side-effects.

7. The Simulation Test Case

One test case for the simulator now exists. SLIDE modules have been written, compiled, interconnected and simulated in order to demonstrate the simulator. This example is only one

---

[7] Relevant oxproasions ara thoss which appaar in ths initiaJsaation conditions of a procass, or in a DELAY atatomont

*In tormmatmf, tho old inatanco of • procoaa ts farbafo-colloctod, and a now instance of tho process ia created. It is this now motanco that ontora tho Rat.

of a genre of simulations that can be done with the SLIDE simulator.

The example contains the following configuration: (see Figure 7-1)

-The PDP-11 UNIBUS

-A UNIBUS CPU.

-A Peripheral Device attached to the bus.

-A small CMOS memory attached to the bus.

-A synchronous data link connected to the peripheral device and to a black box - the source of the synchronous data.

The data link uses an SDLC-like protocol. The peripheral device converts received synchronous data bits into 16 bit words, and writes them to the memory over the bus.

To expedite the test, the following simplifications were made:

-The CPU consists only of the processor status words and the bus arbitrator.

-The data link protocol is always in information-transfer format.

-No error checking is done on the data link.

» The omitted details could be included; SLIDE could even be used for CPU description although it is not intended for that purpose.

7.1. Peripheral Device Description

We focus now on the SLIDE description of the peripheral device module, which is the most active module in the simulation. As shown in Figure 7-1, the peripheral device has two independently executing functional sides. It has an input side, which is responsible for pulling data off the synchronous line according to the protocol *flag, address* data* flag.* It also has a UNIBUS interface side, which receives a 16 bit data word from the input side. The bus side is responsible for transferring data words to memory according to the UNIBUS protocol.

The operation of this peripheral device was modelled with the SLIDE process structure shown in Figure 7-2.

On the bus interface side, we have processes BMASTER and PASSGRANT, statically nested inside a dummy process. BMASTER is one level of priority higher than PASSGRANT, which merely passes grants along the bus when the peripheral device has not made a bus request

Figure 7-1: SLIDE Simulator Test Case

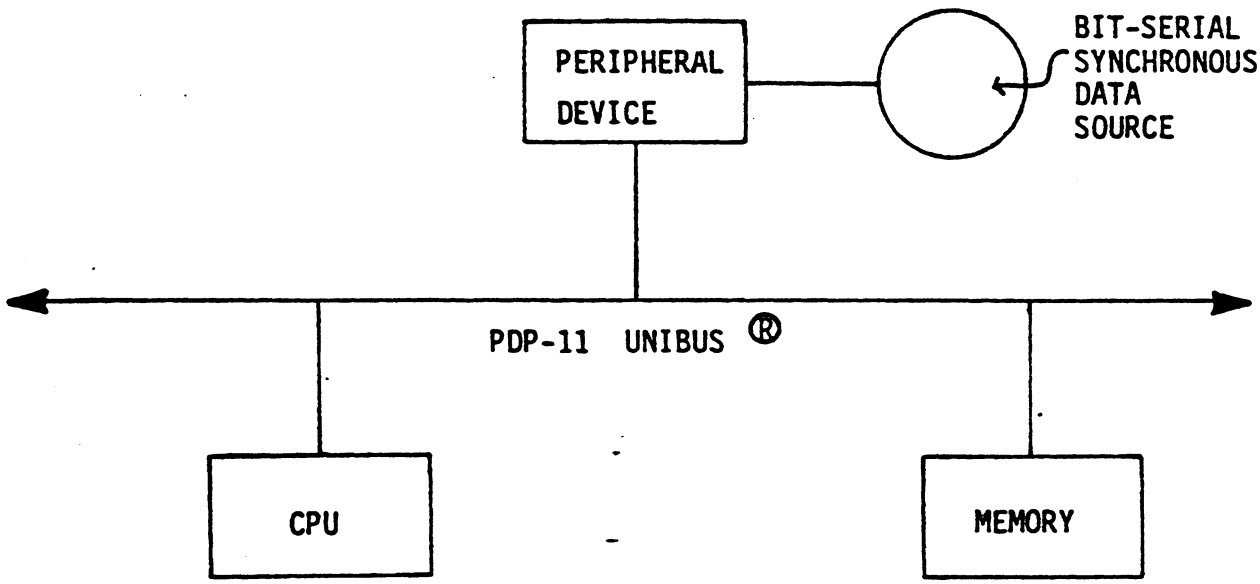
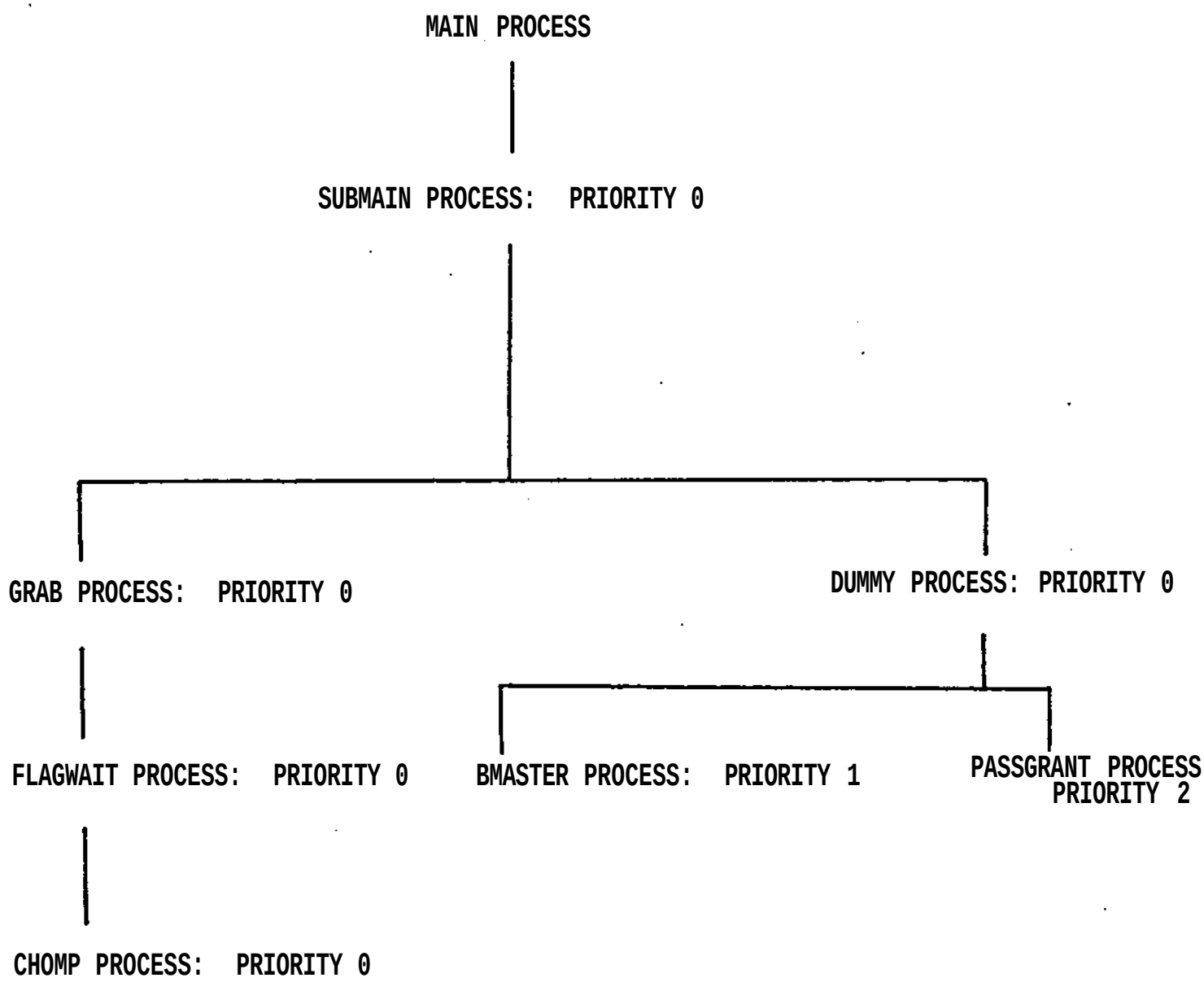


Figure 7-2t SLIDE Process Structure of the Peripheral Device Model



Thus, PASSGRANT is always active unless BMASTER is active, since BMASTER has higher priority[9] .

When CHOMP has assembled a word of data, BMASTER is initiated and PASSGRANT is terminated. When the transfer is completed, BMASTER terminates itself and PASSGRANT starts again.

The SUBMAIN and DUMMY processes were used to encapsulate local SLIDE variables; DUMMY isolated the bus interface side of the peripheral device (BMASTER, PASSGRANT) from the data link side (GRAB, FLAGWAIT, CHOMP). MAIN, SUBMAIN and DUMMY contain no actions except those embedded in subprocesses; thus they are "active" but asleep (not in a busy-wait). This is achieved by using the "DELAY WHILE 1" instruction, which puts an active process to sleep indefinitely.

GRAB, FLAGWAIT and CHOMP input data off the synchronous line, delete inserted zeros and detect flags and addresses. When the peripheral device address is recognized, the bus interface process is modified and 16-bit data words are assembled BMASTER requests the UNIBUS when data begins to be assembled.

Other devices connected are BLBOX, the black box that generates data for the synchronous line, and DELAY, a generalized delay gate with a variable delay parameter, logic type and output bit width. In our example, we created a 75 nanosecond delay of open-collector logic type to simulate the UNIBUS skew on the MSYN and SSYN lines.

7.2. Simulation Test Results

We began the simulation run by instantiating and interconnecting SLIDE modules. Then, the simulation was started and values were traced.

Figure 9-1 shows the interconnection commands used to create the example configuration. First, an ARBIT device is created from the SLIDE module of the same name, and it is called *AA1", distinguishing it from other ARBIT devices we may want to use. At this point, the user has already been given a list of the ports available for interconnection and their names, e.g., "B8SY corresponds to port 6". At interconnection time, the user may rename these ports to avoid confusion between multiple copies of devices, and list the names of ports, in order to the interconnector. Ports are then connected by name correspondance. So, ARBIT device AA1 might have a BBSY port (port 6) which we will name BBSY1. ARBIT device AA2 might

[9] The initiation condition, for PASSGRANT arc "INIT PASSGRANT WHEN 1"

have a BBYS port which we will name BBSY2. (Thus the instantiated devices and their interconnections may not have names corresponding to labels in the SLIDE modules themselves.) The ports which are named at interconnection time are given in terms of increasing port numbers from left to right

In our example (Figure 9-1), "A" causes a wire labelled "A" to be created and port 1 of ARBIT device AA1 to be connected to it. "D" causes port 2 to be connected to wire TT. On line two of this example, we add a DEVICEB module named AB2 to the system model. (This is the SLIDE module for the peripheral device.) By typing "A" at the port 1 position of AB2, we connect it to wire A.

The most interesting connection of wires is the daisy-chain of bus grant wires on the UNIBUS. DEVICEB and MEM0RY(AC3) each have a grant-in and grant-out line, at ports 9 and 10 and 10 and 11 respectively. The bus arbiter has the grant line emanating from port 19. By connections shown in the example (Figure 9-1), we achieve the setup shown in Figure 7-3.

The ADD command causes devices to be connected at the *element* level. The SIMU command then causes the *chain* and *simulation process* for each device to become present. The connections that have been made are checked for compatibility at this point and any parameters given in the original SLIDE module are given values at this point. In our example in Figure 9-1, we see that .PER was the period of the sync line, J\DDR is the SDLC address, and .TOPME is the address to put the first word in memory.

The trace facility command, PROBE (PR) caused each wire being probed to output its state whenever it is written to, whether the value on the wire had changed or not.

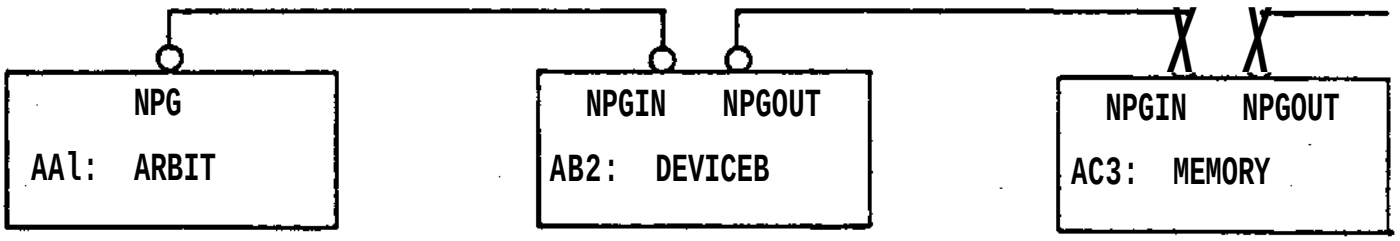
Once these preliminary commands were executed, we ran the simulation for .239 usec with the UNTIL 0.239 command.

Figure 9-2 shows the tracing on the sync line, called "INTO*. It had a period of 30 nanoseconds for this example, so every period the wire got written to, and a trace output resulted. This trace output printed the current simulation time, along with the wire name. On the next line, the logic type, bit width and current wire value were displayed This figure shows data on the wire coming after the flag and address. Note the zero insertion after 5 one's, which adheres to the SDLC protocol.

Bus connections were displayed by showing all device ports that are connected to each wire, along with the values each of them are putting on the bus.

Figures 9-3 to 9-7 shows the peripheral device actions over the UNIBUS. The device first gets control of the bus from the bus arbitrator, then acts as a bus master to write data to

Figur* 7-3: Daisychaining of UNIBUS NPG Line



the bus memory (bus slave).

In order to illustrate the power of the simulator, a second simulation was run, with the period of the sync line adusted above the UNIBUS data rate. Because of inadequate protocol or buffering between parts of the device interface, every other data word was lost, as shown in Figure 9-8.

A third test was run to illustrate the effect of a stuck line on the UNIBUS. MSYN was stuck high and then the memory device attempted a read, and raised SSYN. The peripheral device operated normally until it delayed waiting for SSYN to be lowered. The memory held SSYN and waited for MSYN to go down; MSYN was stuck, and the bus hung up. The results of this are seen in Figures 9-9 to 9-10.

8« Conclusions and Future Research

The purpose of the research described in this paper was to demonstrate that behavioral simulation of interconnections is possible and useful. While the man-machine interaction of the simulator is still under development, the simulator itself has been demonstrated as a viable CAO tool.

Future efforts involve testing the simulator on less-detailed, multiprocessor configurations! using the simulator to test proposed bus standards for the National Bureau of Standards, and increasing the capabilities of the simulator to include central processor simulations. Likely candidates for testing are ARPANET-like structures and parts of the CM* multiprocessor bus structure.

9. Acknowledgments

The authors wish to acknowledge the efforts of John Wallace, Rich Bollinger and Mario Barbacci in helping mahe this project a success.

References

[1]
ANSI Technical Committee X3T9.
USA. Contribution to ISO TC97/SC13 For a Small Computer-to-Peripheral Bus Interface Standard
December,
197a

[2]
BarbacciM
Automated Exploration of the Design Space for Register Transfer (RT) Systems.
PhD thesis, Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa., November, 1973,

[3]
BarbacciM, Barnes,(L, Cattell^, Siewiorek,D.
The Symbolic Manipulation of Computer Descriptions ; The ISPS Computer DescriptionLanguage.
Technical Report, Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa., March, 1978.

[4]
BarbacciM, NagleA
The Symbolic Manipulation of Computer Descriptions ; ISPS Application Note: An ISPS Simulator.
Technical Report, Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa., March, 197a

[5]
Bell,C, Grason^, NewellA
Designing Computers and Digital Systems Using PDP-16 Register Transfer Modules.
Digital Press, Digital Equipment Corp., Maynard, Mass., 1972.

[6]
Birtwistle, G. et al.
SIMULA BEGIN.
Petrocelli/Charter, 641 Lexington Ave., MY.CJ1Y. 10022, 1973.

[7]
Curtis,D.
IDS» An Interface Description System.
November,
1975.
Unpublished note, ALCOA.

[8]
DeBenedictis, Erik P.
Multilevel Simulator.
Master's thesis, Dept. of Electrical Engineering, Carnegie-Mellon University, Pittsburgh, Pa. 15213, 1979.

[9]
Dietmeyer, D.
Logic Design of Digital Systems.
Allyn and Bacon, 1978.

[10]
Hill, D. and van Cleemput, W.
SABLE: A Tool for Generating Structured, Multi-level Simulations.
In *Proceedings of the 1979 Design Automation Conference.* IEEE and ACM, 1979.

[11]
Knoblock, At Loughry, D. and Vissers, C
Insight Into Interfacing.
***IEEE Spectrum* 12(5)50-57, May, 1975.**

[12]
Marino, Edward.
Computer Interface Description.
In *Proceedings of the 17th Annual Technical Symposium,* national Bureau of Standards and ACM, June, 1978.

[13]
Parker, A.
A Generalized Approach to Digital Interfacing.
PhD thesis, Electrical Engineering Department, North Carolina State University, May, 1975.

[14]
Parker, A., and Wallace, J.
The Development of GLIDE: A Hardware Descriptive language for Interface and I/O Port Specifications.
November,
1978-
Research Report, Electrical Engineering Department, Carnegie-Mellon University.

[15]
Sorensen, Ib Holm.
System Modeling.
Master's thesis, Computer Science Department, University of Aarhus, Denmark, March, 1978.

[16]
Vissers, C.
Interface, A Dispersed Architecture.
In *Proceedings of the Third Annual Symposium on Computer Architecture^* pages 98-104. ACM SIGARCH and IEEE Computer Society, 1976.

[17]
Wallace, J. and Parker, A.
SLIDE: An I/O Hardware Descriptive Language.
1979.
to be published in the Proceedings of the 1979 International Symposium on Hardware Descriptive Languages, Palo Alto, CA, October.

[18]
Wallace, J.
SIGNALS: A Proposed Extension to GLIDE.
Research note, Electrical Engineering Department, Carnegie-Mellon University, Feb., 1979.

I. Test Simulation Traces

This appendix contains the simulator traces for the test runs described in Section 7.

```
IEXEC SDT.C
[19:27:56]
LINK! TOadinQ
CLNKXCT SDLC EXecutionl
SLTDE/Mu^ti-T.evei S1titU^ator version 1.0
Welcome and Good Luck!

•GET TFST2
»SIMU
%Simulation time parameters for AHBjT may be bound now
tiinished
%Simulatfon time Dara»eters for aB7 • DEVICES ipdv be bound now
•PER:=30
%finished
%Sii»ulation ti»e oaraneters for RC3 t MEMORY mav b* hound now
•TOPME:=10
tfinished
%Slnulation ti™e D«raoet*rs for A[)4 • BLROx mav be bound now
•PER:=30
•AD1:=100
%finished
JALL
AA:
  I   »RBIT A n M5yN SSY?! TNTR C PA PB SACK BBSY ?N1T ACLO DCLC
      NPP BK4 BR5 BR6 BH7 NPG BG4 RG5 BG6 BG7 PSW READY;
AB2:  nEvirEB A n MSYN" SSYN* C SACK BBSY NPP NP" NPG2 INTr;
ACii  MEMOpY A n MSYN sgyN TNTR C SAC* BBSY "PR NPG2 NpG3,
AD4;  »LBOx INTO?
AE5j  riEtAY/DELAY(75)/kGGir(4)        MSYN MSYNB;
AF6:  nELAY/DELAYf75)/L0Girf4)         SSYN SSYNB;
•PR A
•PR D
•PR MSYN
•PR MSYNB
•PR SSYN
•PR SSYNP
•PR C
•PR SACK
•PR BBSY
•PR UPP
•PR NPG
•PR NPG2
•PR NPG3
•PR INTO
•UNTIL 0.239
—>        0.030US IMTU
 LOGICs   6 SlZEs   o PERinns      30.000US
VALUES ON WlpE-
0
VALUES ALONG B'lS:
••>       O.OfiOus INTO
 LOGICs   6 SlZEs   0 PERlHDs     30.000US
VALUES ON wIpE-
1
VALUES AT.ONG-BUS
```

Figure 9-1: First Test Run : Trace 1

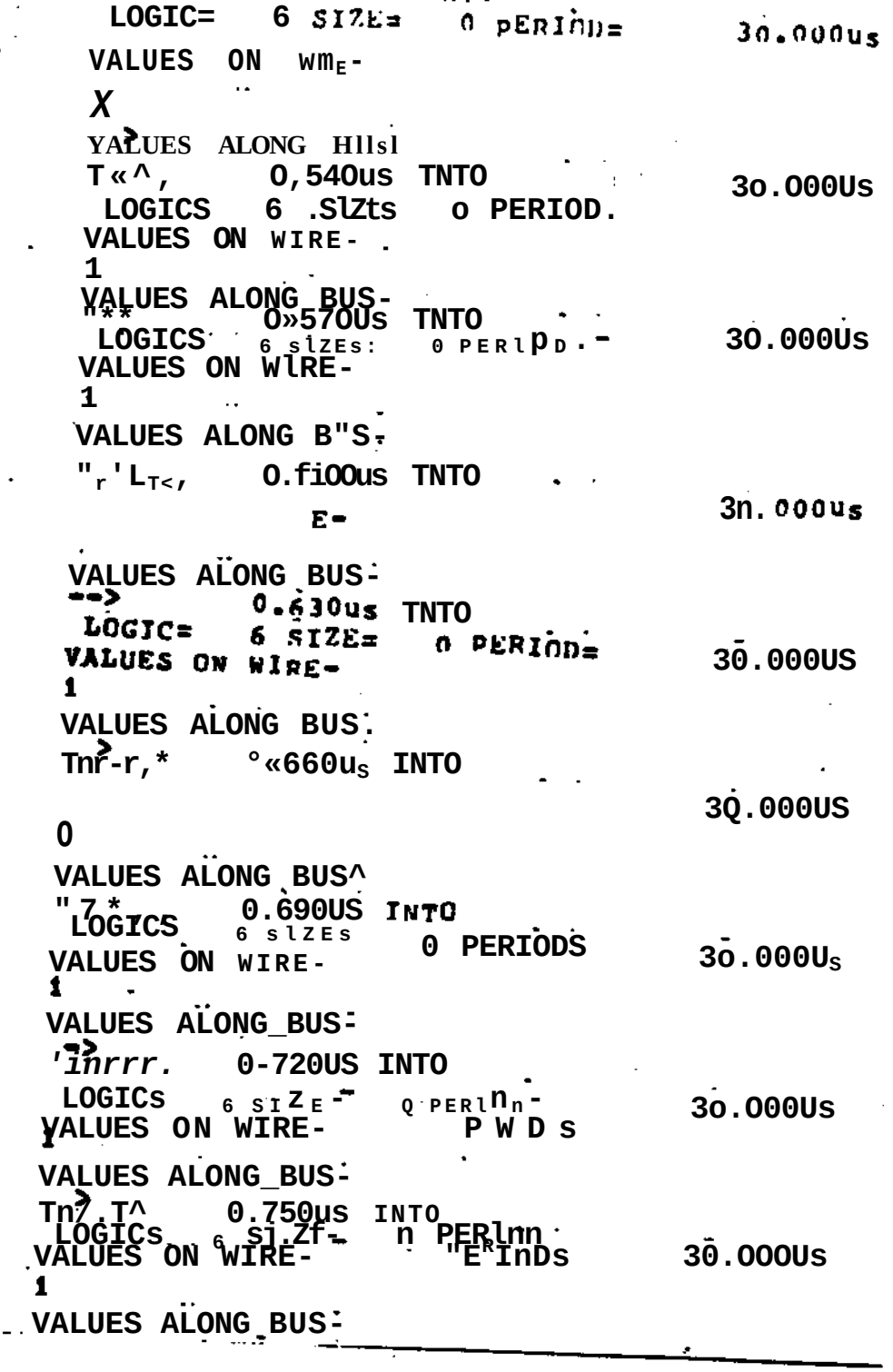


Figure 9*2: First Test Run : Trace 2

```
 LOGTC=    6 SlZts     n PERIOD=        30.000US
VALUES ON WlRE-
1
VALUES AT.UNG I)US-
-->         1.028us NpR
 LOGIC:    4 SIZE=     0 PERIOD=        0.OOnuS
VALUES ON WlRE-
1
VALUFS ATUNG H»1S-
AC3    9
0
AB2    8
1
AA1   14
0
—>          1.028us NPG
 LOGIC*    « Sl7fcs    n PERIOD=        n.nOOuS
VALUES ON WIRE-
1
^VALUES ATONG BUS-
-->         1.029us SACK
 LOGIC=    4 SIZE=     0 PERIOD=        0.OOOUs
VALUES UN WIRE-
l
VALUES AT.ONG BUS-
AC3    7
0
AB2    6
1
AA1    9
0
-->         1.029us NpR
 LOGIC5    4 SIZE=     0 PERIOD=        0.OOOUs
VALUES ON WIRE-
0
VALUES AT.UNG BUS-
ACS    9
0
AB2    8
0
AA1   14
0
-->         1.030US NPG
 LOGIC*    6 SlZfcs    0 PERIOD=        0.OOOUs
VALUES ON WIRE-
0
VALUES AT.ONG RUS-
—>          1.031US BUSY
 LOGICS    4 siZEs     o PERlODs=       0.OOOUS
VALUES ON WlRE-
1
VALUES AT.UNG BUS-
ACS    ft
0
AB2    7
1
AA1   10
0
```

Figure 9-3: Rrst Test Run : Trace 3a

```
—>        l.032us A
 LOGTC=    4 siZEs   17 PERIOD=          n.OOOuS
VALUES ON WIPE-
OOOOOOOOOOOOOOOOi
VALUES AT.ONG 8US1
AC3    1 .
OOOOOOOOOOOOOOOOu
AB2    1 .
OOOOOOOOOOOOOOOOi
AA1    1 .
OOOOOOOOOOOOOOOOO
lLOGIC*   4.032uS n 15 PERIODS          O.OOOuS
VALUES ON WIRE-
liin n iTitoi mi.
VALUES AT.UNC, BUS-
AC3. 2 -    -
OOOOOOOOOOOOnOOO
AB2    2 -
111111H1 1101 101
AA1    2 -
OOOOOOOOOOOOOOOO C
—>        l.n us
 LOGTC= . 4 S2E=    i PERIODS          O.OOOus
VALUES ON WIRE-
10
VALUES AT.ONG BUS-
AC?    6
00
AB2    5
10
AA1-   6
OQ
•.>        1.033us   SACK 0 PERIOD=          0.OOOUS
 LOGIC=    4 slZEs
VALUES ON VURE-
0
VALUES AT.ONG B«S-
AC3    7
0
AB2    6
0
AA1    9
0
—>        I.185US MSYNR
 LOGICS . 4 SlZEs    0 PERlODs          O.OOOUs
VALUES ON WIRE-
1
VALUES AT.ONG HUS-
AES    2          —>        1.260US MSYN m
0                   LOGTCs    4 SlZEs    n PEPlODs          O.OOOUs
AB2    3          VALUES ON WlRE-
1                  1
                   VALUES AT.ONG B'lS-
                   AES    1
                   1
                   AC3    3
                   0
                   AA1    3
                   0
```

Figure 9-4: First Test Run : **Trace 3b**

```
—>          1.760US SSYN
 T.OGIC=    4 SlZEs    0 PEPInO=          O.OOPUs
VALUFS ON WiRE-
1
VALUFS AT.ONG B"S-
AF6    1
0
AC3    4
1
AM     4
0
--->        1.135us SSYNB
 LOGTCs     4 SlZEs    o PEPlOD=          0.OOOUs
VALUES ON WlRE-
1
VALUES AT.UNG BUS-
AF6    2
1
AB2    4
0
--->        1.136us MSYNR
 LOGTC=     4 SI7.E=    o pERIPDs         0.OOOUs
VALUES ON WIPE-
0
VALUES AT.UNG QUS-
AE5    ?
0
AB2    3
0
—>          i.i36us n
 LOGTCs·    4 siZKs   IS pERlblis        0.OOOUs
VALUES ON WIR_E-
OOOOOOOOnOnOOOOO·
VALUES AT.UNG BUS-
AC3    7 -    -
OOOOOOOOOOOOnOOO
AB2    2 -    -
OOOOOOOOnuOOnOOO
AAl    2 -
0000000000000000 -
```

Figure 9*5} First Test Run : Trace 3c

```
—>          1.4 Itus MsYN
 LOGICS    4 SIZE*    0 PE I D S          0.OUOUs
VALUES ON WIPE-
0
VALUES AT.ONG BUsl
AE5    1
0
AC3    3
0
AA1    3
0
—>          1.412US SSYN
 LOGICS    4 SIZEs    0 PERInDs           0.OOOUs
VALUES ON WIRE-
O
VALUES ATONG B"5-
AF6    t
0
AC3    4
0
AAl    4
0
--->        1.43flUs A
 LOGIC5    4 SIZEs   17 PERIODS           0.OOOUS
VALUES ON WIRE-
OOOOOOOOnOOOOOOOOO
VALUES ALONG BUSl
AC3    1
OOOOOOOOnOOOOOOOOO
*B2    1
OOOOOOOOnOOOOOOOOO
*At    1
OOOOOOOOnOOOOOOOOO
—>          1.438US    t pEPlPDs          O.OOOUs
 LOGIC2    4 SIZEs
VALUES ON WIPE-
00
VALUES AT.UNG BUS-
AC3    6
00
AB2    S
00
AAl    6
00>         1.439us BRSY
 LOGIC35    4 SIZEs    0 pERlODs          0.OOnUS
VALUFS ON WIPE-
0
VALUES AT.UNG BUS-
ACS    8
0
AB2    7
0
AAl   10
0
                —>          1.487US SSYNP
                 LOGICS    4 SIZEs    o PERlODs          O.OOOUs
                VALUES ON WIPE-
                0
                VALUES AT.ONG BUS-
                AFS    2
                0
                AB2    4
                0
```

```
—>        1.S04U5
•WHAT AC1
IState Of ACi
NPG3
 LOGICS  6 SlZfc=    0 PERliiDa        o.noous
VALUFS ON WlRF-
0
VALUFS AT.ONG B"S-
NPG2
 LOGICs  6 jflZGs    0 PERlPO=         O.OOOUs
VALUES ON WIPE-
0
VALUES AT.UNG RUS-
NPP
 LOGICs   4 siZEs    0 PERlPDs         O.OOOUs
VALUES ON WlRE-
0
VALUFS ATONG B"S-
AC3   9
0
AB2   8
0
AA1   14
0
STATE OF M :
OOOOOOOOOOOOOOO
OOOOOOOOOOOOnOOO
OOOOOOOOOOOOnOOO
OOOOOOOOOOOOOOOO
OOOOOOnOnOOOnOOO
OOOOOOOOOOOOOOOO
OOOOOOOOOOOOOOOO
OOOOOOOOOOOOOOOO
OOOOOOOOnOOOOOOO
11111111111011n1
OOOOOOOOnOOOnOOO
—>        6.979US
fWHAT AC?
tstate Of ACi
NPG3
 LOGICs  6 5IZE=    n PERinD=         O.OOOUs
VALUFS ON WlRE-
0
VALUES AT.ONG RUS-
STATE OK H"J"*
OOOOOOOOOOOOOOOO
OOOOOOOOOOOOOOOO
OOllioilooionooi
11111111iiio11ni
0011101lnoioOOOi
11111111f110f101
ooinoi looiooont
111111111110]101
0011101100100001
lllllllllllOt101
OOOOOOOOOOOOOOOO
»EX1T
40 Garbaoe collections exerutëri d_ur ir!q 8322 m_s

End of SjM^Lf^ DrooraHt pxpcntξnn
```

Figur« 9-7: First Test Run : Trace 3e

```
.EXEC SDLC
[18:10:00]
LINK:   Loading
[LNKXCT SDLC Execution]
SLIDE/Multi-level Simulator version 1.0
Welcome and Good Luck!!

#GET TEST2
#SIMU
%Simulation time parameters for AA1 : ARBIT may be bound now
%finished
%Simulation time parameters for AB2 : DEVICEB may be bound now
.AD1:=100
.PER:=20
%finished
%Simulation time parameters for AC3 : MEMORY may be bound now
.TOPME:=10
%finished
%Simulation time parameters for AD4 : BLBOX may be bound now
.PER:=20
.AD1:=100
%finished
#PR A
#PR D
#PR MSYN
#PR MSYNR
#PR SSYN
#PR SSYNR
#PR C
#PR SACK
#PR BBSY
#PR NPR
#PR NPG
#PR NPG2
#PR NPG3
#PR INTO
#UNTIL 0.239
-->        0.020us INTO
 LOGIC=    6 SIZE=    0 PERIOD=       20.000us
VALUES ON WIRE-
0
VALUES ALONG BUS-
-->        0.040us INTO                          STATE OF M :
 LOGIC=    6 SIZE=    0 PERIOD=       20.000us  0000000000000000
VALUES ON WIRE-                                 1111111111101101
1                                               0000000000000000
VALUES ALONG BUS-                               1111111111101101
-->        0.060us INTO                          0000000000000000
 LOGIC=    6 SIZE=    0 PERIOD=       20.000us  1111111111101101
VALUES ON WIRE-                                 0000000000000000
                                                1111111111101101
                                                0000000000000000
                                                1111111111101101
                                                0000000000000000
```

Figure 9-8: Second Test Run : Trace 1

```
-«>          0.0000S MSYN
 LOGIC=    4 SI7E=    0 PERIC =          0.000us
VALUFS UN WIRE-
1
VALUFS AT.ONG BUS-
AE5   1
0
AC1   3
0
AA1   3
1
I->          0'.000US n
 LOi;TC=    4 SI7ES   1 5 PERIOD=          0.000us
VALUFS UN WIPE*
0000000000000000
VALUFS AT.ONG nns-
AC3   2
0000000000000000
Ab?   7
OOOOOOOOOOOtiOOOO
AA1   7
0000000000000000
-->          o' OOOuS SSYN
 LOtiTCs    4 SI7E=    0 IPERIOD=          O.OOOus
VALUFS ON WIRE-
1
VALUFS AT.OMC LJI1S-
AF6   1
0
AC3   4
1
AA1   4
0
-->          O.OiOijS TUTG
 LOGIC=     6 SIZE=    0 PERIOD=          iO.QOOus
VALUFS UW WIRE-
0
YALUFS AT,ONG BUS-
-->          0.060US TNTG
 LOC;TC=    6 SIZE=    0 PERIOD=          3O#0C0us
VALUFS OM WIRE-
ii
YALUFS AT.ONG BUS-
-->          0.07SUS MSYNP
 LOGIC=     1 SIZE=    0 PERIOD=          O.OOOus
VALUFS UN WIRE-
1
VALUFS bT,ONG BUS-
AE5   7
t
AB2   3
0
—>           O.O75US SSYNP
 LOGTCs     4 5l7h:=   O pERIOD=          0.000us
VALUFS OM WIRE-
1
VALUFS ATONG BUS-
AF6   7
1
AB2   4
0
```

Figure 9-9: Third Test Run : Trace 1

Figure 9-10: Third Test Run : Trace 2

```
•->          1.0J4uS 5AfK
 L.OCiTC=    4 .^I7L=    0 Ph.Pini;s        o.ooou?;
VALUFS »»i WTPVJ-
n
VALUFS AT.C|NC BT1S*
AC3   7
0
AH/   n
0
AA1   ^
0
:->          1.ObOuS INTO
 LnGiC=     K SIZE=    0 pfcPXODs        iO.OOOu^
VALUFS ON W_TPE-
0
VALUFS AT.t1N(; tM.lS-
•->          t .O80US TNTO
 I.nGTC=    6 si7fc;=    0 PE^innr        jo.ooous
VALUFS UM WTPh-
0
VALUFS ATajM(; BI1S-
```

```
.cTATF HF M :
OOOOnoOuOOOOOOOOO
OOOOQOOuOuOOOOOO
OOOOOOOOOOnOOOOO
OOOOOOOOOOOOOOOO
OOOOnOOOOOOOOOOO
OOOOOOOuOOOOOOOO
OOOOOOOOOOOOOOOO
OOOOQOOOOOOOOOOO
OOOOOOOOOOOOOOOO
OOOOOOOOOOOOOOOO
OOOOOOOOOuOQOOOO
>KX1X
```