SLIDE: AN I/O HARDWARE DESCRIPTIVE LANGUAGE

by

Alice C. Parker & John J. Wallace

DRO18-24-80

August 1980

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

** Bell Laboratories
Warrenville-Naperville Rd,
Naperville, IL 60540

This research has been supported by the U.S. Army Research Office under grants #DAAG29-76-G-0224 and #DAAG29-78-G-0070.

Table of Contents

List of Figures

Abstract

This paper describes the SLIDE language - a hardware-descriptive language designed for description of input/output, interfaces, and interconnected digital systems. The language allows description of asynchronous, concurrent processes which can communicate. Timing and synchronization mechanisms are capable of being described as well. Current research applications include the use of the SUOE language for I/O simulation and for verification of synchronization mechanisms.

L Introduction

Recently, multiprocessing research and development has caused an increased interest In I/O and interconnections. In fact, it has become important to document, simulate, and formally verify 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 present hardware-descriptive language techniques, interconnections and their interfaces can be described accurately at both the gate and circuit levels.

Unfortunately, this low level of description is not adequate for all applications. Sheer size and speed of the execution of simulation programs have precluded simulations of large interconnected systems. Verification of system behavior is difficult at this level since function must be recognized by the verifier from the description; also, the unstructured nature of the hardware may make some aspects of the verification indeterminate. Finally, low level interface and interconnection descriptions contain details which the reader does not need and which tend to obscure his understanding of the behavior of the hardware. For these reasons, existing gate-level descriptive languages do not provide the kind of hardware description needed for the above tasks; a description at a higher level is needed. An obvious solution is to use a register-transfer language.

Furthermore, there is strong motivation for constructing *behavioral* register-transfer descriptions of I/O hardware.[1] Certainly, behavioral descriptions can convey to the reader the overall operation of interfaces better than structural descriptions, since much of the unnecessary detail is eliminated. Simulations proceed more rapidly, and can encompass larger systems. Verification is possible since the behavior is explicit while the implementation is hidden and the description can be structured.

Research has been done at Carnegie-Mellon University to produce such a behavioral language called SLIDE[2] for interface and interconnection description; this language is the subject of this paper. There are two current projects which involve SLIDE and have provided motivation for its development:

1. The development of a SLIDE simulator. This simulator is now capable of simulation of interconnection schemes like the UNIBUS[3] and the D-bus [3J Eventually behavioral descriptions of processors and other functional units will be linked to the interconnection descriptions and entire systems will be able to be simulated*

2. The verification of aspects of interface behavior and module-to-module communication. The goals of this project are to determine aspects of SLIDE which enhance or impede verification attempts as well as to develop assertion and verification techniques [25].

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

- To provide a language which can 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 and there are already a number of general-purpose hardware descriptive languages. It is always

---

*Behavioral description! differ from structural descriptions because they describe only the functions: of the hardware and not the) hardware itself. Storage locations and register-transfers which exist in the hardware may be absent from the behavioral description; control hardware is implicit rather than explicit A discussion of this distinction it found in [4]

2SLIDE (formerly GUD€) *ṁ* an acronym for Structured Lanfuage for Interface Description and Evaluation

^UNIBUS ia • registered trademark of Digital Equipment Corporation

difficult to justify the development of yet another hardware descriptive language. For many reasons, 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 introduces the required capabilities of an I/O descriptive language. Section 3 surveys the field of hardware description; the intent of that section is to discuss the applicability of other hardware-descriptive languages to the problems presented in Section 2. Then, Sections 4 and 5 present salient features of SLIDE. An example SLIDE description of the UNIBUS™ [12] is included as Appendix A. The SLIDE BNF is given in Appendix B.

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 from an I/O device 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 separate sequences of control, 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*.

In light of the above example, we now describe what we believe to be the fundamental aspects of interconnection behavior which an I/O descriptive language must model. These are:

1. Concurrent execution of multiple processes.

2. Contention for shared resources between processes.

3. Critical sections of execution within processes.

4. Simultaneiety of execution both within and across processes.

5. Processes which behave in a subordinate fashion with respect to other processes.

6. The possession of behavior by the interconnections themselves.

All of the above, except the last, *are* analogous to the fundamental behavior of multiple software processes executing on a multiprocessor (which is not surprising).

Shared resource contention and concurrent execution imply the more specific mechanisms of *mutual exclusion, synchronization, arbitration, priority allocation,* and *preemption of resources.* The shared resource in the above example is the bus; the four control flows represent concurrent processes; mutual exclusion results when one of the contending processes is actually given the bus resource for a data transaction and the others are excluded.

Synchronization occurs when two concurrently executing processes must communicate. Arbitration and priority allocation occur because the processes contend for the shared resource. Preemption occurs, for example, when a specific device process fails to release the bus after a specified time period. The bus is preempted, and the errant process may be terminated or suspended.

Critical sections of execution refer to sequences of events which cannot be suspended or terminated, but are within a process which could otherwise be interrupted. For example, a device transacting over the bus may be preempted by a faster device needing bus access only at the completion of a transfer protocol sequence.

The fourth item, *simultaneiety,* means that actions may occur at the same instant of time. It also means that any side effects from simultaneous actions may conflict. For example, the assertion of a control line while synchronizing with a process may not produce the expected effect if another input to the process is simulaneousiy asserted.

Subordinate processes (subprocesses) occur in hardware in the following manner: A subordinate process is an independent executing environment which has certain dependencies on a more global process. The global process retains the ability to allow the subordinate process to start, and execution of the subordinate process is meaningless when the global process is not executing. For example:

A global "flag" process looks for a "flag" sequence of bits arriving over an interconnection. Once the flag arrives, the subordinate "data" process begins using the following sequence of bits as data. At the same time, however, the global process continues its search for a terminating flag sequence in the following bit sequence, and stops the "data" process when this occurs.

Thus, in our model of interconnection behavior, we need the idea of a *subordinate process**

The most fundamental realization which will allow general-purpose interconnection description is that interconnections have behavior. The use of a tri-state bus implies certain semantics about the behavior of the bus. The delay through a crosspoint switch is part of the behavior; the gating in other interconnection strategies is also part of the interconnection behavior, for example. In summary, this means that interconnections have implied storage, time delays and Boolean functions associated with them, just as storage is associated with variables in a conventional HDL

There is another consideration which should be mentioned here; it is a fundamental aspect of interconnection *description.* This is the notion of viewing interconnection structures as canonical data structures, which is relatively new to hardware descriptive languages. This view, however, would allow us to treat declaration and replication of buses, crossbar switches and daisy chains, for example, with the same ease we treat lists, arrays and bit vectors in many programming languages.

In conclusion, a language designed to describe this genre of interconnection behavior must possess powerful and unconventional control constructs. In particular, semantics should exist to allow:

1. Prioriy orderings between processes.

2. Initiation, termination and suspension of processes.

3. Interprocess synchronization primitives such as *signal* and *wait*

4. Communication between processes.

5* Description of timing dependencies, timeouts, and data I/O at fixed bit rates.

In addition to these, language primitives should exist which allow description of operations common to I/O such as bit manipulation, code conversion, FIFO buffering, parity and error checking, manipulation of synchronous I/O data, and modeling of combinational logic

3. Related Hardware Description Research

A complete survey of the field of hardware description is not possible within the scope Of this paper. However, this section is intended to perform the following functions:

- To identify key developments and research results which affected the development and final form of SLIDE.
- To describe parallel developments in the field which have similarities to SLIDE, either in form or application.
- To motivate the necessity of SLIDE as a solution to the interconnection description problem, in light of the descriptive capabilities of other currently available languages.

The survey we now present begins with a discussion of requirements for hardware description, presents some current techniques for interface and interconnection description, and briefly describes the capabilities and limitations of ISPS, AHPL, and DDL

It is interesting to note that two excellent comparisons of hardware descriptive languages (HDLs) (which were published some time ago [6J[15]) revealed some desirable properties of general-purpose HDLs. Barbacci enumerated these as:

* Readability.
* Familiarity with naming and usage conventions.
- Use across several levels of detail.
- Simplicity; small number of language primitives.
- Extensibility.
- Fidelity to the system organization.
- Timing and concurrency.
- Syntactic simplicity.
- Ability to separate data and control.

These were chosen presumably because of his emphasis on the applicability of the language for *description* of digital systems. Figueroa, interested in design automation! also found some of the above properties important; and in addition discussed:

- Modularity.

- Block structuring, including the existence of global and local variables.

* Facilities for functions, subroutining and macros.

- Facilities for specification of parallel processes.

* Facilities for determining and controlling process interaction explicitly.

Barbacci did point out that the languages he surveyed (CDL, APDL API and ISP) lacked arbitration as a primitive function. He defined *timing and concurrency* as "parallel actions", and explained "At the RT level, concurrent activities are described by allowing them to be activated simultaneously (i.e. under the same conditions)". The SLIDE design goals and the model for interconnection behavior presented here reflect may of the properties deemed desirable by Barbacci and Figueroa.

Formal specification techniques for description of interface and interconnection operation do exist. We can subdivide these into:

- The state diagram approach.

* The flowchart approach.

* The formal language approach.

Flowcharts and state diagrams have been used to describe a number of buses - including the IEEE-488 bus (see [17]). Figures 3-2 and 3-1 illustrate the use of these techniques. However, languages useful for bus and port descriptions have been slower to be adopted.

Bell and Newell [10] laid the groundwork for the semantics of interface description with their requirments for port description. These requirements helped form the basis for the SLIDE language semantics. In the interim, interface standards committees began struggling with the description problem. Curtis, working with the Purdue Workshop on Industrial Computer Systems, Data Transmission and Interface Committee, proposed IDS, an Interface Description System [11]. The system involved the use of the PMS notation, (from Bell and Newell) at the top level, and the use of a new laguage for description at the programming and register-transfer levels. In addition, the system was to cover other levels of description as well, but these were not defined at the time. The new language Curtis proposed was a version of ISP with features of AHPL and with necessary timing constructs **added** (Most of which were being added to the ISPL version of ISP [7]).

At the same time Vissers had been developing a language based on APL with timing

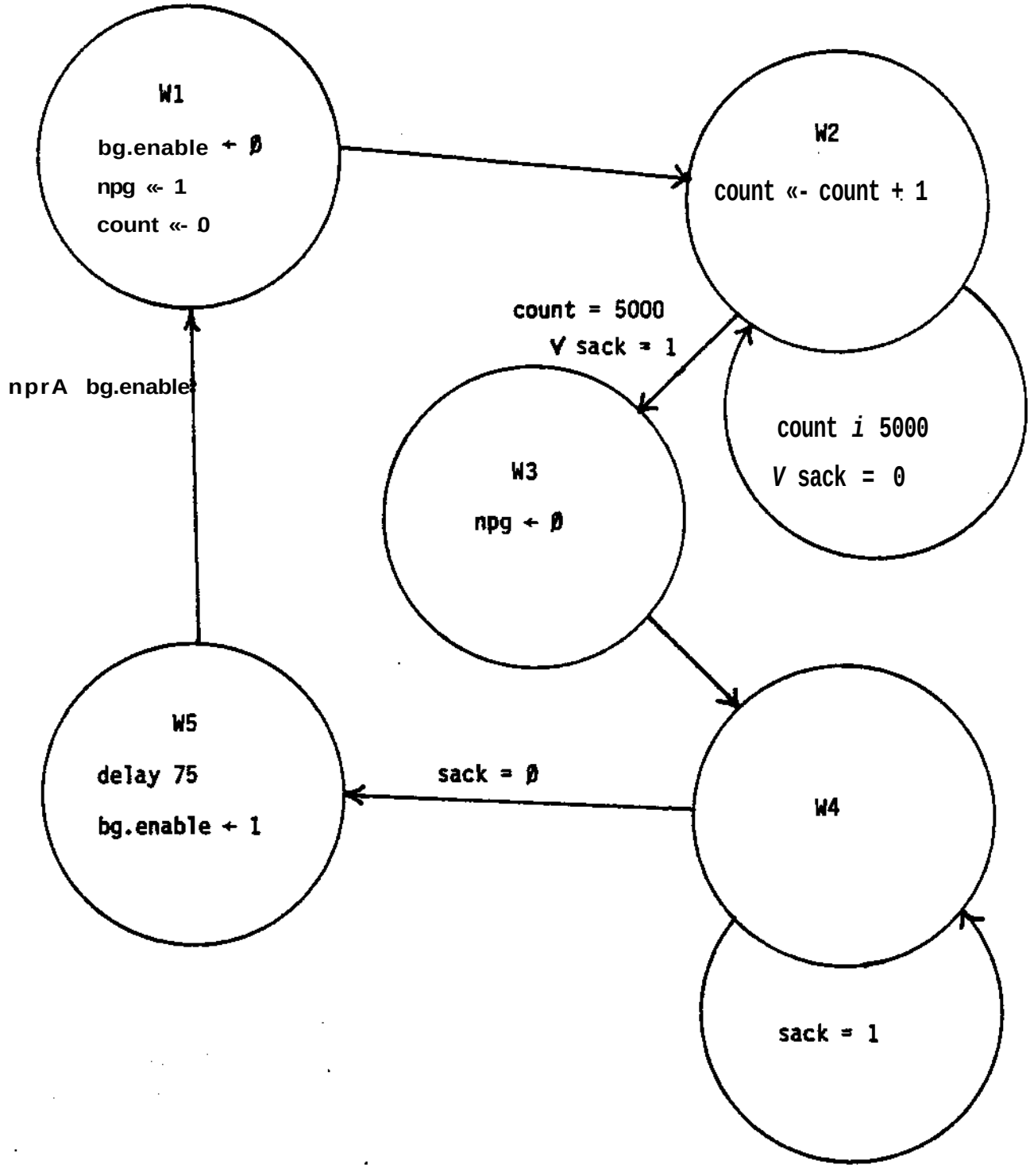

Figure 3-1: The UNIBUS ™ Non-Processor Grant Process, Oescribed in State Diagram Form

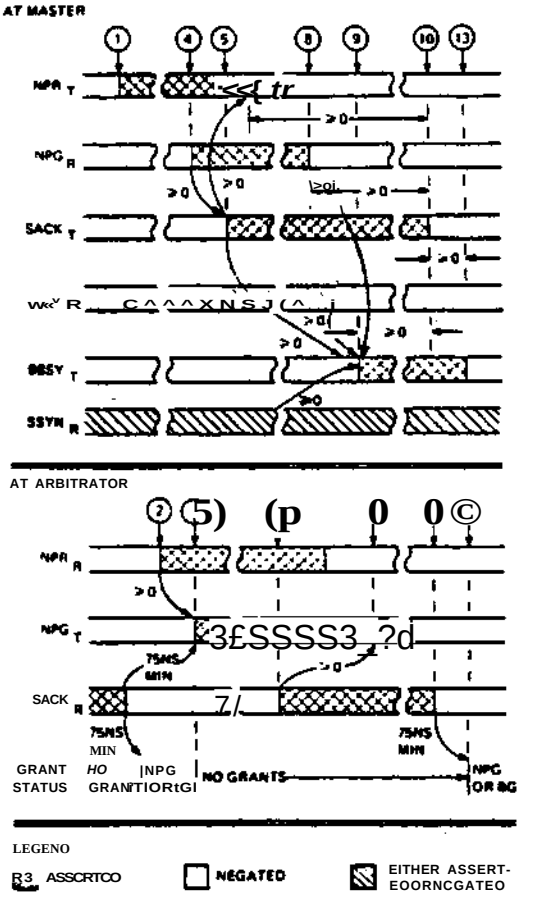

Figure 3-2: Timing of the UNIBUS ™ Non-Processor Grant Process
[12]

constructs added which is used to produce a formal description of the state diagrams describing interfacing functions. SDLC and the IEEE-488 interface bus have been described

using this approach [24, 17} The formalism of this approach allows one important application - functional simulation. Vissers has intentionally restricted the coverage of the language to the gate and register transfer levels, with the added timing. An example of the language is shown in Figure 3-3, which illustrates the states and transitions of an example taken from the UNIBUS. In this example the granting process for non-processor requests on the UNIBUS is described, just as in Figures 3-2 and 3-1.

```
      v  npr.grant
[11  ul:-M~npr ~bg.enable)/ul
£2]       bg.enable*8
[31       npg*B
[4]       npg*4
[5]       counte-B
•£61      *w2
[7]  u2: -*(count + 5888 A sack - 8,count-5888 V sack-l)/u2fu3
£81       countecount+1
£9]      -»w2
£10]M3:  npg*8
Ell]w4* -»(sack«B,sack-l)/u5,u4
£121MSI delay 75
[13]      bg.enabled.
£14]     -*M!
```

Figure 3-3: The UNIBUS ™ Non-Processor Grant Process, Described Using Vissers' Language

The automaton, or process, is named nprgrant. There are five major states in this process, Wl - W5. State Wl is executed when there is a non-processor request and busgrants are enabled (nprAbg.enable). The next state is W2, where there is a delay until the selection acknowledge line (SACK) is raised by the device receiving the non-processor grant (npg), or until 5000 time units elapse, whichever comes first. Then, the non-processor grant line is released, and state W4 is reached. State W4 is a delay state until selection acknowledge is released. Then state W5 is entered to reenable grants* The next state is again WI.

Although Vissers' descriptive language itself did not provide features adopted in SLIDE, the fact that the approach existed and examples were published greatly increased our understanding of the necessary requirements for interface description.

More recently, Marino proposed a hierarchical descriptive system for computer interfaces, MPLID [20]. This language system has more modularity than most computer hardware descriptive languages and the semantics of some example statements resemble

SLIDE primitives. However, an overall control structure which reflects the aspects of interconnection behavior we discussed earlier is lacking. The latest publication in this area [23] describes a System Modelling Language based on BCPL [22]. This language has three interesting features - the ability to declare processes (concurrency), mailboxes for interprocess communication (synchronization), and uninterruptable code sequences (critical sections). The language is designed to model systems at a higher function level than the SLIDE language.

Along similar lines, SOL (Simulation Oriented Language) [18], based on ALGOL-60, does not describe hardware structures explicitly. However, it does allow concurrent processes, timing, subordinate processes, and shared-resource contention to be dealt with explicitly. One other system-level language, ASPOL [19] also reflects these important aspects of interconnection behavior. Concurrent processes can be declared, and processes can have priorities of execution. *Set* and *Wait* primitives are provided for synchronization. While these languages provide synchronization and process-level constructs, they do not provide the level of detail possessed by register-transfer languages.

At this point, a short analysis of AHPL, DDL and ISPS will be made. This analysis is done not only to determine how well these languages reflect the model of interconnection behavior we have already described, but also to assess how well these languages achieve the goals and expected applications of SLIDE.

Clearly, AHPL [16] and DDL [13] could be used to accurately describe and simulate interconnection behavior. However, the main reasons these two languages are not suitable for the applications SLIDE has been designed for are the following:

- Descriptions are designed to represent hardware structure rather than behavior and become large quickly.

- The fundamental aspects of interconnection behavior are implicit in the descriptions. They must be structured by the user in terms of available primitives.

- An overall notion of interconnection behavior in terms of the model discussed previously is lacking in the language.

However, there are advantages to these languages. AHPL*III* [16] does address the "structuring of interconnections" issue referred to in section 2 better than SLIDE does. Both languages provide some basic primitive operations on data which are more general and powerful than SLIDE primitives.

A comparison of SLIDE with ISPL [5] resulted from an attempt to compile SLIDE descriptions (then GLIDE) into ISPL so that interfaces and interconnections could be

automatically designed and simulated with existing software [21} At this point, SLIDE adopted more of an ISP-like syntax for compilation purposes.

At this point SLIDE has all of the features of ISPS with the exception of procedure parameters, the case construct, and some arithmetic and logic operations. On the other hand, most of the SLIDE features to be highlighted in this paper are not present in ISPS[4]. The major shortcomings of ISPS with respect to interconnection description are at two extremes - the lowest level hardware operations and the overall process structuring. For example, attempts to describe the UNIBUS in ISPS have resulted in inability to synchronize between concurrent processes and failure to model the open-collector behavior, timing, and transitions of signals on the bus itself.

Thus, the current languages *are* either at too high a level, or synchronization mechanisms and processes must be implicit and are lost in the details of the descriptions.

4* 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.

4.1. The Process

In the same way that routines are the central unit of execution irr most programming languages, processes are the central units of execution in SLIDE descriptions.[5] 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 *subproeesses)* 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 subproeesses (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. Processes nested at the same level model either concurrent process execution

---

[4] Mh# most recent version of ISPS [8] has incorporated lomt SLIDE features for process synchronization

[5] ^Ws> us* "description" here and not "profram" to emphasize that a SLIDE description *describes* the operation of a piece of hardware. Correepondmfly, we use "execute" to mean "the operation of the actions described*

or processes competing for shared resources. Subprocesses are the subordinate processes we described in the introduction.

Since each process describes the operation of a piece of hardware, each one is an asynchronous executing environment. Processes which need to communicate with each other can do so by using global variables (e.g. by asserting a shared line) or in the future by using *signals* (see Section 6.2).

Processes are started (called *Initiation)* 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. This will be discussed more fully in Section 5.1.

4.2* 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. Both these constructs will be discussed in Section 5.3.

4.3. Other I/O Related Primitives

Other 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).
- declare combinational logic via the comb declaration.
- 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.

4.4. 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.*

-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 4.3.

S. The SLIDE Language

This section discusses more fully the constructs introduced previously. We will concentrate more on the semantics of these constructs than the actual SLIDE syntax. Therefore, we will be loose with the syntax, introducing it as we go along. The complete SLIDE language syntax is described in [26], and in Appendix B.

5.1. Processes

Processes are the central unit of execution in a SLIDE description. They are initiated nonprocedurally and are independent executing environments. A process definition consists of:

1. An *init* declaration which specifies under what conditions the process starts executing (is initiated).

2. Declarations of registers, lines,[6] combinational logic, etc which are local to the process.

3. Definitions of local subroutines.

4. Definitions of subprocesses.

5. The executable statements for the process.

Each process has an explicit priority. Informally, a process starts executing when (1) the process it is a subprocess of is executing, (2) its initiation conditions are true, and (3) no process at the same subprocess level with a higher priority is executing. When a process

[6] Lines ar* interconnections tuch as address and data lines, bus-rtqusst and bus-grant fines, ate

starts executing, all process which are at the same subprocess level, have a lower priority, and are executing are *terminated.*

Priorities can be used to time-order the execution of processes. For example, assume we have 3 processes, *A*% *B,* and C, no two of which can execute concurrently. Also, *A* is to always execute as soon as its initiation conditions become true; 8 is to execute when its initiation conditions become true, but only if *A* is idle; and C can execute only if *A* and 8 are idle. This can be done by giving *A* priority o/ 8 priority 1, and C priority *2.* Then as soon as /Ts initiation conditions become true, it will start executing, terminating 8 or C if they were executing. When *A* finishes executing, it will restart if its conditions are still true. If not, 8 may start if its conditions are true. If not, C may start.

Going back to our example about processes *A*% 8, and C above, if terminating a process once it has started executing is undesirable, a 1-bit variable can be used which prevents other processes from starting while any process is executing. This simple type of synchronization mechanism allows for mutual exclusion and critical sections of process execution can be protected from preemption termination.

A detailed example follows. Assume we are writing a SLIDE description for a disk controller interfaced to a UNIBUS. The controller is to do a transfer operation to a memory location over the UNIBUS whenever the *datarcady* line rises from logical 0 to logical 1. The controller is to reset (i.e. stop any on-going transfer and reset itself) whenever the *initline* rises from logical 0 to logical 1. Part of a SLIDE description for the controller is in Figure 5-2. The example system configuration is shown in Figure 5-1. Lines 1 and 2 specify the conditions under which the *Dreset* and *Dtransfer* processes start executing. *Reset* is given a higher priority than *transfer* since a reset should terminate any on-going transfer operation.

The expressions "initline EQL /" and "dataready EQL /" are true at the moment the line rises from 0 to 1; not before or afterwards. These have different semantics than the expressions Initline EQL 1" and "dataready EQL 1" which are true whenever the lines are logical 1.

If many controllers, each with its own *transfer* and *reset* processes, are to be connected to a UNIBUS$_9$ the overall structure of the resulting SLIDE description is shown in Figure 5*3. Here we have an arbiter, a set of devices A, B, C and D, and their interconnections. A description of the arbitration process is shown in Figure 5-4.

'Noto that 0 (ztro) it htfhttt priority} 1 it ntxt hifhtst; ttc

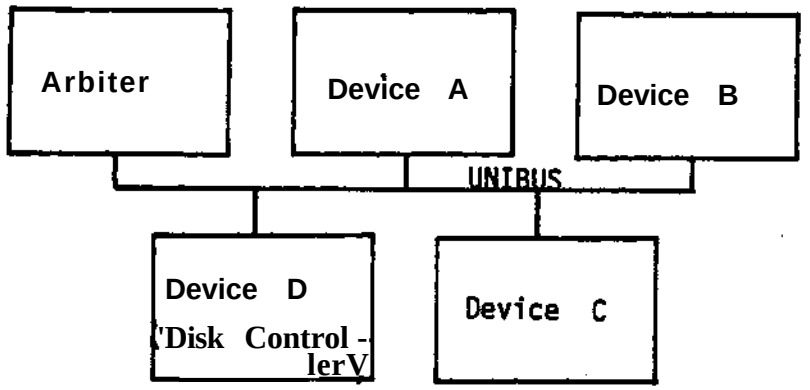

Figure 5-1: The Overall Example System configuration

```
[1]  INIT DresetrO     WHEN fnltline EQL A
[2]  INIT Dtransferil UHEN dataready EQL A

[3]  PROCESS Dreset;
[4]      BEGIN

         here we reset the controller

[5]      END*

[6]  PROCESS Dtransfen

         declarations local to transfer

[7]      BEGIN

         do the transfer operation here

CSJ      ENOl
```

Figure 5-2: Reset and transfer processes

```
MAIN PROCESS UNI BUS;
    global bus declarations
    IN1T larblter:O WHEN initline EQL /;
    INIT arbiter:1 WHEN TRUE;
    INIT DresettO WHEN initline EQL A
    INIT Dtransfertl WHEN dataready EQL A
    INIT AresettO WHEN initline EQL A
    INIT Atransfertl WHEN adataready EQL A
          •

    other devce processes are initiated similarly

    PROCESS arbiter;
          •
          BEGIN
          arbitrate the bus
          END
    PROCESS larbiter*
          •
          BEGIN
          initialize devices in the bus
          END
          t
    PROCESS Dreset*
          •
          BEGIN
          reset the disk controller
          END
          •
    PROCESS Dtransfer*
          •
          BEGIN
          transfer data on the unibus
          END
          •
    other processes are similar

    BE6IN          IdaIn process execution!
          DELAY WHILE TRUE          {Idle forever!
    END*
```

Figure 5-3: Bus description with n device controllers

```
PROCESS arbiter;
  EXT REGISTER

     psw<15:0>;  ! program status word I

  COMB pri<> : = psw<7:5>;     ! processor priority I

! The process is always executing, hence the "WHILE TRUE")

BEGIN
     WHILE TRUE DO
     BEGIN
          IF npr THEN       !If there is a non-processor!
                            Irequest, grant the bus tol
                            !the requesting device!
          '    BEGIN
               npg «- /;
               DELAY 5000 UNTIL sack EOL/NEXT
               npg «- \;
               END
!    If there is a level 7 request and the processor priority
     is at a lower level:
!
          ELSE IF (br7 AND pri LSS 7 AND ready) THEN
               BEGIN        IGrant the bus to level 7 devices!
               bg7«-/;
               DELAY 5000 UNTIL sack EQL/NEXT
               bg7«-\;-
               END
          ELSE IF (br6 AND pri LSS 6 AND ready) THEN
!              .     service bus request 6
               .
               .
           ELSE IF (prr) THEN
I              .
               .     Service processor request for bus
I
           NEXT
           DELAY UNTIL sack EQL 0 NEXT

           ! Wait to begin arbitration again I
           DELAY 75         ! Skew I
           END           End of WHILE TRUE statement!
     END;                End of arbiter!
```

Figure 5-4: UNIBUS arbiter process showing mutual exclusion

5.2. Hardware

The variables in a SLIDE description represent pieces of hardware. These can be registers, arrays of memory, synchronous and asynchronous lines, combinational logic, FIFO buffers, and associative tables. This section discusses each of these and their usage.

5.2.1. Hardware Declaration

Registers, asynchronous lines, and arrays of memory can be declared with a *hardware* declaration. Any hardware declared within a process is local to that process. It can be accessed within that process and its subprocesses using ALGOL-like scope rules. The notation is similar to the ISPS notation described in [8, *S*\ For example:

```
OCAL LINE d<15sO>, a<17:0>:
```

declares two 16 and 18 bit wide TTL Open Collector Active Low bus segments, one named d_f and one named a. These correspond to the data and address segments of the UNIBUS, respectively. Because the lines are typed, writing to a typed line implies certain behavior. Writing to an open collector line, for example, implies an an attempt to set or reset the line, but may not result in a change in state of the line unless other devices wired to the line cooperate. For this reason, hardware declarations which *seem* low-level are needed in order to model abstract behavior.

```
EXT REGISTER PSU<15tO>*
```

is declared in the *arbiter* process, and declares a single 16 bit register, external to (but accessible by) the arbiter. The arbiter needs this access to determine processor priority when granting bus requests.

```
MOS REGISTER mem[1023:0]<15:0>:
```

declares a IK by 16 bit MOS memory.

5.2.2. Synchronous I/O

Description of synchronous I/O (i.e. I/O transfer which occurs at a fixed rate) is difficult because of the different implementations of hardware which perform synchronization. SLIDE allows a limited description of synchronous I/O with the *sync* declaration. For example:

```
SYNC 6 10000 LINE tapel<8:0>*
```

declares the synchronous transfer of 9 bits in parallel from/to a line named *tapel* at a rate of one transfer per 10000 clock pulses. The use of a variable declared as synchronous carries with it an implicit wait for the data to synchronize.

We can test for a *time-ordered* sequence of values on synchronous line(s) as in the following example. If s is declared as follows:

SYNC e 50000 LINE s o t

then the statement below delays execution until ***s*** takes **on** the **values of 5 ones followed by a zero:**

DELAY UNTIL s EQL |l|l|l|l|l|0|t

This reads ***delay until s equals the time-ordered sequence of values*** **1,** ***U U U U*** **then 0$**

5.23. FIFO Buffers

FIFO buffers (also known as queues) can be declared with the *buffer* declaration. An assignment to a buffer puts the item at the end of the buffer. **An** assignment **from a buffer removes** the first item. Overflow and underflow **can be tested for** via **the** ***iferror*** **statement discussed** in Section 5.3.2.

5.2.4. Combinational Logic

Combinational logic can be declared with the *comb* declaration. **For example:**

COMB npgbotsnpga and not nprai

declares combinational logic within the device "a" interface to pass **a grant to the next device "h" whenever a grant** has been asserted and "a" device has not requested the bus*

5.2.5. Tables

Code conversion is often done by table lookup. (Of course, **hardware logic is also used for** this purpose and can be described in SLIDE with COMB declarations.) SLIDE **has** a special declaration for associative memory tables. For example:

```
TABLE grey <2:0><2:0>
        '000=>'000,
        '001=>'001,
        '010»>'011f
        '011=>'010,
        '100=>'110,
        '101=>'111,
        '110=>'101,
        '111=>'100;
```

is a grey code conversion table specified in binary." The **table above* named grey, takes 3**

---

[8] **The <teby atatanani ia diacucaad further in Section 5.3.1.**

[9] **In SLIDE, • sinf (• quoto O indicates • binary number; a pound aifn (•) indkataa an octal number.**

bits as input and produces 3 bits as output. The last 8 lines specify the conversions. A table can be accessed with the *encode* and *decode* unary operators. For example:

EHGCgrey) '101

has the value'111, and:

OEC(grey) '111

has the value '101. (DEC is the inverse of ENC.)

5.2.6. Specifying Bit Slices

An important property of SLIDE is that arbitrary bit slices of a variable can be accessed There are two ways to do this. First, *variable-namt<L:j>* references the *Uh* through *jth* bits of *variable-name* (i and *j* must be constants). Secondly, *variable-name<e(i)>* references the *i* bits of *variable-name* starting at bit position *e* U, the bit slice width, is a constant, but e, the starting bit position, can be an arbitrary expression). This allows easy expression of packing and unpacking operations.

5.27. Operators

Along with the operators discussed above such as *encode* and *decode,* SLIDE has other I/O related operators. These include logical, comparison, arithmetic, parity, concatenation, and formatting operators. The SLIDE operators are summarized in Figure 5*5.

bitwise logical operators:
OR, XQR, AND, EQV

pomoartson operators (evaluate to 1 for true* 0 for false):
EQL, NEQ, GTR, GEQ, LSS, LEQ

arithmetic operators:
•> -* *> */f* H00

format and concatcnatton operators:[10]
FMT (pattern), @

unary operators:
-, NOT (bitwise), ENC *(table),* DEC *(table)*,
PARE, PARC[11]

Figure 5*5: SLIDE operators

[10] The format (FMT) operator makaa tn arbitrary bit pattarn from two tourcaa.

[11] PARE Md PARO return tvtn/odd parity bit*

S3. SLIDE Statements

SLIDE has three more constructs of interest. These are: the *delay* statement, the *iferror* statement, and the ability to specify actions in parallel as well as sequentially.

5.3.1• Delay Statement

The *delay* statement is fairly general and allows for:

- delaying for a fixed period as in

```
DELAY 75;
```

which delays execution for 75 clock pulses. In the UMIBUS, this *delay* statement is often used in bus master descriptions, since the bus master must account for bus skews.

* delaying until some condition becomes true as in

```
DELAY WHILE initline EQL Is
```

which delays execution (and termination) of Dreset and other reset processes while the initHne is asserted on the UNIBUS.

- delaying until some condition becomes true subject to a timeout.

This timeout capability is very important. For example, assume a bus arbiter grants the bus to a process by raising the *busgrant* line. Within 5 microseconds, it expects the process to acknowledge by raising the *sack* line. If this does not occur, the arbiter should timeout, then lower the grant line to reset the bus. A SLIDE description of this is in Figure 5-6. (A "NEXT*[9] used as a statement delimiter forces sequential execution.) A DELAY-UNTIL-ELSE construct which is more complex allows for exception handling. (The ELSE statement is executed when a timeout occurs.)

```
[1]     npr •- / NEXT                    ! grant the bus !
[23     DELAY 5000 UNTIL sack EQL / NEXT!Clock is a 2 nanosecond 1
                                         Iclock!
[3]     npg 4- \t                        {lower the grant llnej
```

Figure 5-6: Delay statement example with timeout

5.3.2. Iferror Statement

Error conditions can arise in two cases:

1. when accessing a FIFO buffer if an overflow or underflow occurs

2. when using the encode or decode operators if an illegal operand is used

The existence of an error condition can be tested for with the *iferror* statement. An error condition exists if the last buffer or table access resulted in an error (the two cases above).

For example, assume we wish to extract a command from a command buffer named *combaf.* If the buffer was not empty, we process the command. Otherwise, we idle for 100 dock pulses. After this, we repeat. The SLIDE description to do this is in Figure 5*7.

```
[1]   WHILE TRUE DO
[2]     BEGIN
[3]        connand 4- combuf NEXT     1 get the next co»Mand I
[4]        IFERRQR
C5]             THEN DELAY 100
[61             ELSE BEGIN
                        •
                        •
                   process the command

171                     ENO*
[8]     END
```

Figure 5*7: If error statement example

5.3.3. Sequential and Parallel Execution

Process execution normally flows sequentially from statement to statement with the delimiter between statements being a "NEXT."[11] If two statements are to execute in parallel, a semicolon (;) is used for the statement delimiter.

Any degree of parallelism can be achieved by using the semicolon to indicate parallel* execution of arbitrarily complex statements. Two blocks of statements separated by a semicolon such as "BEGIN - END; BEGIN ~ END" execute in parallel. The BEGINs act as a *fork,* and the ENDs act as a *join.*

6* Conclusions and Future Research

SLIDE has proved itself general enough yet powerful enough to be useful as an I/O hardware descriptive language. We have written and simulated a SLIDE description of the UNIBUS, a non-trivial problem [1,2]. The nonprocedural and priority properties of the process *are* very powerful, allowing descriptions such as the Dreset (in Figure 5-2) to be written. This is not possible in other hardware descriptive languages such as ISPS*

6.1. Simulation and Verification

A SLIDE compiler and simulator exist [26]. The simulator is allowing research to proceed in studying bus structures, I/O, and multiprocessor communications. It will also provide a tool for teaching the above in an interactive fashion.

It is possible to parameterize a SLIDE description and then test the effect of these parameters on the hardware by simulation. These parameters *are* numbers whose values are not *bound* until simulation time. By using parameters, the effects of varying buffer sizes and timing can be studied.

In the summer of 1979, the verification aspects of SLIDE were studied [25]. A dialect of SLIDE, V-SLIDE, was designed and a mechanism for verifying the behavior of synchronization mechanisms was specified

6.2. Signals

A SLIDE description is an abstraction away from the details of hardware implementation. Since synchronization is basic to I/O, we plan to introduce two synchronization primitives, *signal* and *receive* [27]. These represent another step toward abstraction and are similar to the *P* and *V* semaphore operations [14]. For example, process *A* can send a signal called *s* to process *B* with the following statement:

```
SIGNAL(Bis)*
```

This is similar to the *P(s)* operation. Process 8 can delay until it receives the same signal from process *A* with:

```
DELAY UNTIL RECEIVE(A:s)i
```

TNs is similar to the *V(s)* operation. Process 0 can test whether a signal has been sent with:

```
IF RECEIVE(A*s)
        THEN ••9
        ELSE ...
```

There is no corresponding semaphore operation for this. Consequently signal and receive **are more** general then semaphores. It is easy to see how signals can be used to do process synchronization such as handshaking, bus requests and grants, etc.

The use of signals is an alternative to communicating using global variables. **The advantages** of using signals are:

L Communication between processes is explicit, cleaner, and consequently less error prone.

2. Signals are a more abstract primitive (i.e. they express what is to be done without expressing too much of the detail). A program which designs hardware from SLIDE descriptions has the freedom to determine the method of sending/receiving signals (assert high or low, etc.).

3. Because communication between processes is explicit, simulation and verification of SLIDE descriptions become more straight forward

6*3. Acknowledgements

We would like to acknowledge the assistance of some people whose efforts made this **work** possible: Bill Lyden wrote the original SLIDE compiler; Andy Nagle labored over the language; Mario Barbacci helped with the language design and the compiler; Art Altman implemented the simulator. Subrata Dasgupta provided useful comments on the manuscript Also, Steve Crocker and Bill Overman provided valuable feedback.

References

W

Altman, Arthur and Alice Parker.
The SLIDE Simulator: A Facility for the Design and Analysis of Computer Interconnections.
November,
1979.
Unpublished report.

[2]

Altman, A.
The SLIDE Simulator: A Design and Evaulation Tool for I/O and Interfacing Strategies.
Master's thesis, Electrical Engineering Department, Carnegie-Mellon University, 1979.

[3]

ANSI Technical Committee X3T9.
USA Contribution to ISO TC97/SC13 For a Small Computer-to-Peripheral Bus Interface Standard
December,
1978.

[4]

BarbacciM

Automated Exploration of the Design Space for Register Transfer (RT) Systems.
PhD thesis, Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, Psu, **November,** 1973.

[5]

Barbacci, M., et al.
ISP; A Notation to Describe a Computer's Instruction Sets.
***COMPUTER*, March, 1973.**

[6]

Barbacci, M.
A Comparison of Register Transfer Languages for Describing Computers and Digital Systems.
IEEE Transactions on Computers C-24(2):137-150, February, 1975.

C7]

BarbacciM
TheSymbolicManipulationofComputerDescriptions:ISPLCompilerandSimulator.
Technical Report, Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa* April, 1976.

[8]
Barbacci,M., Barnes,G., Cattell,R., Siewiorek,D.
The Symbolic Manipulation of Computer Descriptions ; The ISPS Computer Description Language.
Technical Report, Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa., August, 1979.

[9]
Barbacci,M., Nagle,A.
The Symbolic Manipulation of Computer Descriptions ; ISPS Application Note: An ISPS Simulator.
Technical Report, Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa., August, 1979.

[10]
Bell, C., Newell, A.
Computer Structures: Readings and Examples.
McGraw-Hill Book Co., New York, 1971.

[11]
Curtis,D.
IDS, An Interface Description System.
November,
1975.
Unpublished note, ALCOA.

[12]
DEC Staff.
PDP11 Bus Handbook.
Digital Equipment Corp., Maynard, Mass., 1979.

[13]
Dietmeyer, D.
Logic Design of Digital Systems.
Allyn and Bacon, 1978.

[14]
Dijkstra, E.W.
Cooperating Sequential Processes.
Academic Press, New York, 1968, .

[15]
Figueroa, M.A.
Analyses of Languages for the Design of Digital Computers.
Technical Report, Coordinated Science Laboratory, Univ. of Illinois, Urbana, IL., May, 1973.

[16]

Hill, FJL and Peterson, G.R.

Digital systems: Hardware Organization and Design.
Wiley, 1978.

[17]

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

[18]

Knuth, 0. and McNeley, J.
A Formal Definition of SOL
***IEEE Transactions on Computers* C-13:409-414, August, 1964.**

[19]

MacDougall, M. and 1 McAlpine.
Computer System Simulation with ASPOL
In *Proceedings of the Symposium on the Simulation of Computer Systems.* ,
September, 1973.

[203

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

[21]

Nagle$_9$ A.
An Investigation of GLIDE - A Generalized Language for **Interface Description and Evaluation.**
Master's thesis, Dept. of Electrical Engineering, Carnegie-Mellon University, Sept, **1976.**

[22]

Richards, M.
BCPL: A Tool for Compiler Writers and Systems Programming.
In *Spring* *Joint Computer Conference.* , 1969.

[23]

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

[24]

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.

[25]
Wallace, John.
On Automatic Verification of SLIDE Descriptions.
Master's thesis, Department of Electrical Engineering, Carnegie-Mellon Univ., Pittsburgh, PA, 1979.

[26]
Wallace, *1*
The GLIDE/SLIDE Compiler.
Research note, Electrical Engineering Department, Carnegie-Mellon University, April, 1979.

[27]
Wallace, J.
SIGNALS: A Proposed Extension to GLIDE.
Research note, Electrical Engineering Department, Carnegie-Mellon University, Feb., 1979.

Appendix A

UNIBUS Operation

A Brief Summary

The UNIBUS can be divided into two major sections: the data-transfer section and the arbitration section. Within the data-transfer section are five sets of lines: the 18 address lines, the 16 data lines, the handshaking lines (MSYN, SSYN, interrupt), the control lines (which indicate direction of data transfer, byte transfer and read-modify-write), and parity error lines. Transfer is asynchronous (with handshaking), and can be between a processor and any other UNIBUS "slave" device, or between any UNIOUS "master" and any UNIBUS "slave" device. MSYN and SSYN *are* the handshaking lines used by the master and slave except when a device wishes to interrupt and the processor replies with SSYN. The bus master is responsible for delaying 75 ns. or 150 ns. to compensate for bus skew. It also delays 5-10 microseconds and times out if an expected signal does not arrive.

The arbitration section includes the bus-request lines, bus-grant lines, selection-acknowledge, and bus-busy lines. A device requests the bus over one of five request lines, which are ordered in priority. The bus is granted to one of the five levels of priority over one of five grant lines. Each grant line can be daisychained to more than one device, so that a requesting device electricaily closest to the processor receives the grant first. The device responds with the selection-acknowledge (SACK) line. Arbitration is overlapped with data transfer. For this reason, a master doing data transfer raises the bus-busy line. When data transfer is over, the new bus master can do data transfer whenever the old master has lowered the bus busy line.

```
!!
!

    UNIBUS - version 4 - Alice C. Parker - January 1980

This is a partial description of the UNIBUS operation. The daisy
chain request and grant mechanisms are described. There are 4
devices, all at bus request level 6: A, B, C, and D, where A is
electrically closest to the processor. Processes are included for A
and B (as bus masters) and memory (as bus slave). Processes are
also included for system reset of each device, and for the arbiter.

!
!!

!Some questions are still to be resolved about the UNIBUS behavior.
 These include how long the arbiter holds everything at 0 during
 initialization and what the devices on the UNIBUS actually do
 during initialization. Other issues include the exact timing of
 the lowering of SACK by bus masters, and the lack of a terminator
 to assert SACK if a requesting device fails to do so. Also, it
 is not clear how to put a device into its programmable state.
 Furthermore, the SLIDE description does not include power-up, AC-LOW,
 and power-down sequences. It is not clear how the processor
 communicates with the arbiter, and what happens when a parity error
 occurs.
!

MAIN PROCESS unibus;
!
Here we have declared the MAIN process. Comments are
delimited by exclamation points.

We declare a 1 ns clock throughout.
!
CLOCK 1;

!
    general declarations:
    COMB declares combinational logic.
     This means that there exists logic to continually
     evaluate the following expressions.
!
COMB  true<> := 1,
    false<>  := 0;

!
    unibus lines and definitions: These lines are common to all devices.
!
OCAL LINE  ! these lines are open collector active low !
    a<17:0>, ! address lines !
    d<15:0>, ! data lines !
    msyn<>,      ! master sync !
    ssyn<>,      ! slave sync !
    intr<>,      ! processor interrupt line !
    c<1:0>,      ! control lines (c1 and c0) !
    pa<>, pb<>;  ! parity lines !

COMB  dati<>   := '00,       ! data in to master (a binary 00) !
    datip<>  := '01,       ! data in to master then pause !
    dato<>   := '10,       ! data out from master !
    datob<>  := '11,       ! data out from master (byte) !

    p.error<>: = NOT pa AND pb,  ! parity error !
    no.error<>: = NOT pa AND NOT pb;     ! no parity error !
```

```
    sackO,      ! selection acknowledge !
    bbsyO,      ! bus busy (for data section) I
    initlineO,  ! system reset line!
    ac.loO, ! AC power low line !
    dc.loO; ! DC power low line I
    I
     These lines are used for communication between the
     initialization logic and the bus masters for each device
     They are not UNIBUS lines
    I
LINE
    abusyO, mbusyO, bmbusyO;
```

```
!
    These are lines from the external devices A,B,C and D to
    the device interfaces described here.
    They are not UNIBUS lines,
!
LINE    a.ready<>, b.readyO, c.readyO, d.readyO;

!
    The following statements declare the daisy chain mechanism for non-processor
    requests. Device A is wired closest to the processor, then B, then
    C, then D.
!
OCAL LINE
    npraO,      ! request line for device A I
    nprbO,      ! request line for device B !
    nprcO,      ! request line for device CI
    nprdO,      ! request line for device D !
    npg<>;      ! grant line for non-processor requests!

COMB  npr<>   : = npra OR nprb OR nprc OR nprd,    ! bus request for npr!
    npgaO  : = npg,                  ! grant for device A !
    npgbO  : = npga AND NOT npra,    ! grant for device B !
    npgcO  : = npgb AND NOT nprb,    ! grant for device C I
    npgdO  : * npgc AND NOT nprc;    ! grant for device D !

!
    The following statements declare the daisy chain mechanism for priority level 6
    requests. Device A is wired closest to the processor, then B, then
    C, then D.
/
OCAL LINE
    br6a<>,     ! request line for device A !
    br6b<>,     ! request line for device B !
    br6c<>,     ! request line for device C !
    br6d<>,     ! request line for device D I
    bg6<>;      ! grant line for level 6 requests I

COMB  br6<>   : = br6a OR br6b OR br6c OR br6d_t    ! bus request 6 I
    bg6a<> : = bg6,                  ! grant for device A !
    bg6b<> : = bg6a AND NOT br6a.    ! grant for device BI
    bg6c<> : = bg6b AND NOT br6b,    ! grant for device CI
    bg6d<> : = bg6c AND NOT br6c;    ! grant for device DI

OCAL LINE
    br7O,
    bg7O,
    br5<>,
    bgSO,
    br4<>,
    bg4<>;
I
    The daisy chain mechanisms for BR4, BR5_t and BR7 are similar
    but omitted here.
I
I
    These are not part of the UNIBUS but are included so the
    processor can communicate with the arbiter
I

LINE
    prK>, prg<>;        ! Processor request/grant lines !
EXT REGISTER  readyO;! grant enable -- the processor disables BR grants at
                times, such as immediately after honoring an
                interrupt. I

I
```

The following INIT statements set up the initiation conditions for the processes named in the statements.
The processes executing under normal conditions have a priority of 1 (e.g. arbiter: 1). The processes overriding these for system reset have a higher priority, (e.g. larbiterO).

```
INIT arbiter:1 WHEN true;
    !
     The bus arbiter runs always.
    !

INIT amaster:1 WHEN a.ready EQL /;
    !
     This is the process for device A as bus master. It starts when device
     A (external to this description) has data ready to send (indicated by
     a positive transition (/) on the a.ready line).
    !

INIT bmaster:1 WHEN b.ready EQL /;
    !
     The same for device B and the other devices. (The others are not
     described for brevity.)
    !

INIT memory:1 WHEN  msyn EQL /  AND  a<17:0> LSS .topmem;
    !
     This is the process for memory as bus slave. It starts when an
     address in its range has been decoded and msyn is asserted.
     .topmem is a parameter which can be set after compilation
     of the SLIDE program.
    !

INIT cpu:1 WHEN true;
    !
     The processor is always running. It periodically
     checks the status of the interrupt logic.
    !

!
Each of the following processes overrides (preempts) the companion
processes "arbiter", "amaster", etc. whenever the arbitrator
initializes the bus because of their higher priorities.
!

INIT larbiter:0  WHEN initline EQL/ ;

INIT lamaster:0  WHEN initline EQL/AND abusy EQL 0 ;

INIT lbmaster:0  WHEN initline EQL/AND bmbusy EQL 0 ;

INIT lmemory:0   WHEN initline EQL/AND mbusy EQL 0 ;

PROCESS larbiter;

    BEGIN            ! Begin arbiter initialization !
        npg ← 0;
        bg7 ← 0;
        bg6 ← 0;
        bg5 ← 0;
        bg4 ← 0;
        DELAY   .longtime
    END;         ! End larbiter (arbiter initialization)!

PROCESS lamaster;

    BEGIN            ! Begin amaster initialization !
                    !set internal storage !
        br6a ← 0;     !locations to 0 also !
        sack ← 0;     !at option of device !
        d<15:0> ← 0;
        intr ← 0  next
        DELAY UNTIL initline EQL 0
```

```
END;          !End of amaster initialization!

!
!     Other initialization processes are similar
```

```
PROCESS arbiter;
  !
     This is the bus arbiter process.  There are 5 types of requests
     which are (in descending order of priority): non-processor, bus request
     7, bus request 6, bus request 5, and bus request 4.  BR7 - BR4 inter-
     rupt the processor

     Note that we can "grant ahead," Le. grant the bus to a second process
     while a first is still using it.  (The second process does not actually
     use the bus until bbsy goes low.)
  !

  !
     The following storage locations are in the processor, and
     are external to the arbiter process:
  !
  EXT REGISTER

     psw<15:0>;   ! program status word !

  COMB pri<> : = psw<7:5>;     ! processor priority !

! The process is always executing, hence the "WHILE TRUE"!

BEGIN
     WHILE TRUE DO
     BEGIN
          IF npr THEN       !If there is a non-processor!
                            !request, grant the bus to!
                            !the requesting device!
               BEGIN
               npg •- /;
               DELAY 5000 UNTIL sack EQL/NEXT
               npg •• \;
               END
!    If there is a level 7 request and the processor priority
     is at a lower level:
!
          ELSE IF (br7 AND pri LSS 7 AND ready) THEN
               BEGIN        !Grant the bus to level 7 devices!
               bg7W;
               DELAY 5000 UNTIL sack EQL/NEXT
               bg7«-\;
               END
          ELSE IF (br6 AND pri LSS 6 AND ready) THEN
!                .     service bus request 6
                 .
                 .
!
          ELSE IF (br5 AND pri LSS 5 AND ready) THEN
!                .
                 .     service bus request 5
!
          ELSE IF (br4 AND pri LSS 4 AND ready) THEN
!                .
                 .     service bus request 4
!                .
          ELSE IF (prr) THEN
!                .
                 .     Service processor request for bus
!
          NEXT
          DELAY UNTIL sack EQL 0 NEXT
          ! Wait to begin arbitration again !
          DELAY 75          ! Skew !
```

```
    END          End of WHILE TRUE statement!
END;             End of arbiter!
```

```
PROCESS amaster;
 !
    This process describes the procedure device A goes through to
    interrupt the processor.  Device A is capable of bus mastership
    and has priority level 6.
 I

 COMB interrupt.vector<> : =  #320;

 BEGIN
    !
     The device requests bus mastership by raising its bus request line.
     This filters through the daisy chain for priority level 6 and
     subsequently causes the granting mechanism to be initiated.
     !
    br6a «- /;                  ! request the bus ...!
    DELAY UNTIL bg6a EQL /  NEXT  !... & wait until it is granted !
    sack * / NEXT  br6a «- \;  ! ack the grant & drop the request!
     !
      Now we wait until the bus is free and then grab it.
     I
    DELAY UNTIL bbsy EQL 0  NEXT
    bbsy  - /  NEXT
     !
     We now load the data lines with our interrupt vector, then wait until
     the previous slave is done with the data lines (ssyn EQL 0), then
     we raise the interrupt line.
     !
    d<15:0> <- interrupt.vector;
    DELAY UNTIL ssyn EQL 0  NEXT
    intr«- /  NEXT
     t
      Below, now start 2 parallel sections of code.  The first waits until
      the grant is dropped and then drops the selection acknowledge.
      The second interrupts the processor.
     I
     BEGIN
       DELAY UNTIL bg6a EQL 0  NEXT
       sack* \
     END;

     BEGIN
       DELAY UNTIL ssyn EQL/  NEXT
       d «- 0  NEXT     ! release data tines I
       intr •- \
     END  NEXT

     bbsy •• \       ! release the bus I

 END !of amaster!;
```

```
PROCESS bmaster;
 I
    This process describes the procedure device B goes through to do a
    read-modify-write transfer with memory (a dati followed by a dato).
    Device 8 might be, for example, a real time clock.  It is capable
    of bus mastership and has priority level 6 (which really doesn't
    matter since this is a non-processor request).
 I

 EXT REGISTER bdatareg< 15:0>, baddrreg<17:0>;  ! registers in device B !

 BEGIN
    !
     The device makes a non-processor request by raising its request line.
     This filters through the daisy chain for non-processor requests and
     subsequently causes the granting mechanism to be initiated.
    I
    nprb «- /;                  ! request the bus ...!
    DELAY UNTIL npgb EOL /  NEXT  !... & wait until it is granted !
    sack •• /NEXT  nprb «- \  NEXT      ! ack the grant & drop our request!
    I
     Now we wait until the bus is free and then grab it.
    !
    DELAY UNTIL bbsy EQL 0  NEXT
    bbsy  -  /   NEXT
    !
     We now start 2 parallel sections of code.  The 1 st waits until
     the grant is dropped and then drops the selection acknowledge.
     The 2nd does a dati then a dato with memory.
    !•
    BEGIN
      DELAY UNTIL npgb EOL 0  NEXT
      sack«-  \
    END;

    BEGIN
      a<17:0> «- baddrreg<17:0>;  c<1:0> •- datip  NEXT
      DELAY 150;                  ! front end deskew!
      DELAY UNTIL ssyn EOL 0  NEXT! ssyn deskew!
      msyn *• /;  DELAY 10000 UNTIL ssyn EQL /  NEXT
      DELAY 75  NEXT          ! data deskew I
      bdatareg<15:0> •• d<15:0>;  msyn •• \  NEXT
      DELAY 75;             ! tail end deskew I

      BEGIN
         !... process the data here... I
      END  NEXT

      d<15:0> «- bdatareg<i5:0>;  c<1:0> «- dato  NEXT
      DELAY 150;                  ! front end deskew!
      BEGIN            ! ssyn deskew I
         DELAY UNTIL ssyn EQL 0  NEXT
         DELAY 150
      END NEXT
      msyn •- /;  DELAY 10000 UNTIL ssyn EQL /  NEXT
      msyn •• \  NEXT
      DELAY 75  NEXT          ! tail end deskew I
      a «- 0;  c •• 0; d «• 0    •   ! release a and c I
                          I end both parallel sections I
    END NEXT

    bbsy «• \       I release the bus I

 END I of bmaster I;
```

```
PROCESS memory;
  !
     This process describes the procedure the memory controller goes through
     to respond as a bus slave.
  !

  CMOS REGISTER   m[0:.topmem]<15:0>; ! this is memory (16 bit) !

  BEGIN
     !
      There are 4 types of operations:
      1. dati  • master wants data from slave.
      2. datip - same as dati except 1st part of a read-modify-write
            sequence (followed by a dato or datob).
      3. dato  - master is writing to slave.
      4. datob - master is writing a byte to slave.
     !
     If c EQL dati OR c EQL datip THEN
          BEGIN
            d •• m[a];
          ssyn «- /;
          DELAY UNTIL msyn EQL \ NEXT
          d * 0
          END
     ELSE IF c EOL dato THEN
          BEGIN
            m[a] «• d;
          ssyn *- /;
          DELAY UNTIL msyn EQL \
          END
     ELSE ! if c EQL datob THEN !
          BEGIN
            IF a<0>
          THEN m[a]<15:8> «- d<15:8>   ! high byte !
          ELSE mla]<07:0> - d<07:0>;   ! low byte !
          ssyn •• /;
          DELAY UNTIL msyn EQL \
          END
          NEXT
     ssyn «- \
  END ! of process memory !;
```

```
PROCESS cpu;

    REGISTER
        interrupt.flagO,
            !
             Rag bit shared by the processor
             interrupt logic and the CPU
            I
        vector<15:O>;

            |
             register shared by the processor interrupt
             logic and the CPU; used to store the
             interrupt vector
            I

    INITpro.int:1 WHEN intr EQL /;
    !
     This logic handshakes over the UNIBUS and catches the
     interrupt transaction
    |

    PROCESS    pro.int;
    I
     This process describes the procedure the processor
     interface goes through to respond to an interrupt
    !

    BEGIN
        DELAY 75 NEXT
        interrupt.flag + 1 ; ready «• 1 ;
        vector «• d ;        ! load the interrupt vector I

        ssyn •- / NEXT

        DELAY UNTIL intr EQL 0 NEXT

        ssyn «-\

    END;          ! End of processor.interrupt!
!
 NOTE:  Ready will have to be reset by the CPU after
 approximately 4 bus cycles
I
BEGIN

    i System initialization I
    I Begin the bus request Section I

    prr «- / NEXT            ! Raise the processor request flag !

    DELAY 100000 UNTIL prg EQL  1
    ELSE initline«- /
    NEXT BEGIN
        prr •• \ ; sack «• / NEXT
        DELAY UNTIL bbsy EQL 0 NEXT
        bbsy«- / NEXT

    ! begin initialization procedure I

        DELAY 5000 NEXT initline «• /
    END
    NEXT
    sack«- \ NEXT
    initline «• \ NEXT
    DELAY 75
```

```
    ! end initialization procedure !

END;     ! End process CPU !
BEGIN ! main process unibus !
      WHILE true DO NOP
      ! must have a dummy loop to keep this process from terminating !
END   ! of unibus description !
```

APPENDIX B

I. The GLIDE IV BNF

ONF for GLIDE IV - (^vision 8) - June 1979
John J. Wallace, Alice C. Parker
Carnegie-Mellon University, Dept of Electrical Engineering

This is the official GLIDE IV reference.

NOTES:

1. There is no syntactic difference between upper and lower case. All reserved words are written in upper case here.

2. The meta-symbol & refers to the null production.

3. The meta-symbols {...} refer to ZERO OR ONE occurance of the enclosed constructs. The meta-symbols {...}* refer to ZERO OR MORE occurances of the enclosed constructs. These were used for the following reasons:

   -The resulting psuedo-BNF is more natural.

   -The number of productions is reduced.

   -The productions more naturally fit the intended design of the GLIDE IV compiler and the intended output.

The following are left to the compiler implimentation:

-The form of "comments."

-The number of unique characters recognized for keywords and user variables.

I. PROGRAM COMPOSITION:

```
<program>  :;« MAIN <proces9>

<proces9>  ::» PROCESS <simple-name>;
                 <declars>
                 <ini ts>
                 <subrs>
                 <processes>
                 BEGIN <stmts> ENO

<dcclarD>    ::=  { <dcclar>; Iv<
<inits>      ::=  I <init>; hv
<subrs>      ::«  I <subr>; I w          "
<processe3>  : : •» \ <process>; 1ft

<subr>  ::- SUfin <oi mpl o-nomc>;  CHIC IN <otmto> END
```

II. INTERFACE STRUCTURING:

```
<declar> ::-     <hard-declar>
              I <tbl-cleclar>
              I <cIk-dec Iar>
              I <combinator ia!>
<hard-declar> ::= I EXT I  { <logic> } REGISTER <id-name-Mst>
                  I BUFFER <id-name-Iist>
                  I  \ SYNC  <num> } { <logic> } LINE <id-name-1 i st>

        <locjic> ::» TTL  I OCAL  I OCAH  I ECLAL  I ECLAH  I TRISTATE
                    I MOS  I CMOS  I TRANSF  I DIFBI  I DIFSND  I DIFREC
                    I HLT  I HIR[b]

<id-name-l ist> ::= <id-name> I , <id-name> \it
        <id-name>  ::» <simp!e-name> <length> <width>
           <!enc|th> ::« [ <range> ]  I &
           <uidth>   ::» < <rongc> > I <>[b]
                <range> ::= <num>:<num> I <num>

<tbl-declar>  ::» TABLE <simple-name> <uidth> <uidth>
                        <tbl-entry-list>
<tbl-entry-! i3t> ::= <tbl-entry> {,[f] <tbl-entry> Jit
           <tbl-entry>  ::« <num> »> <num>

<clk-dcclar>  ::« CLOCK <num>[7]

<combinational>  ::« COMB <comb-list>
   <comb-list> ::= <comb> I , <comb> I it
        <comb>  ::« <simple-name> <uidth> :• <exp>

<»nit>  ::= INIT <simple-name> :  <nuro> UHEN <exp>
```

---

[5] Thes« ara user definable.

[6] The angle brackots are part of (he width aa in <7O>.

[7] A clock declaration may only occur in the main process.

HI. STATEMENTS:

```
<stmts> ::= <stmt> I <scp> <3tmt> 1 w
        <scp> ::= NEXT I ;

<stmt> ::» <label> <a-3tmt>

<label> ::• <simple-name>: I &

<a-9tmt> ::» <delay-stmt>
           | <calI-stmt>
           | <nop-stmt>
           | <br-stmt>
           | < loop-stmt>
           | <assign-stmt>
           | <i f-stmt>
           | <i fcrror-stmt>
           | DEGIN <stmts> END
           | &

<delay-stmt> ::« DELAY <num> <exit-clause> <else-clause>
               I DELAY <num>
               I DELAY <exit-clause>

   <exit-clause> ::= UNTIL <exp> I UHILE <exp>
   <else-clause> ::» ELSE <stmt> I &

<call-stmt> ::» CALL <3imple-name>

<nop-9tmt> ::- NOP

<br-stmt> ::» BR <simple-name>

<loop-stmt> ::« <loop-clause> <loop-exit> DO <stmt>
              I <loop-clause> DO <stmt> <loop-exit>

   <loop-clau3e> ::« LOOP <num> TIMES 1 &
   <loop-exit>   ;:» <exit-clause>    I &

<as3ign-strat> ::* <id> _ <exp>
                 I <id> _ <trans>
                 I <id> I ?

<if-stmt> ::« IF <exp> THEN <stmt> <else-clause>

<jferror-stmt> ::= IFERROR THEN <stmt> <else-clause>
```

IV. EXPHESSIUNS

```
<exp> :*:» <clterm> I <disjunct> <dterm> 1ft
        <disjunct> ::- OR I XOR

<dtcrm> ::= <cterm> ( <conjunct> <cterm> 1ft
        <con]unct> ::= AND I EQV

<cterm> ::« <aexp>
          I <aexp> <comp-op> <aexp>
          I <icl>   <eq-op>   <trans>
          I <icl>   <eq-op>   <stjnch-value>
   <comp-op> ::* LSS I LEQ I GTR I GEQ I EQL I NEQ
   <eq-op>   ::« EQL I NEQ
   <trans>   ::« / I \
   <synch-voluc> ::» : <num> : { <num> : 1ft•

<aexp> ::« <tcrm> ( <aop> <term> I ft
        <aop> ::- + I -

<term> ::= <fact> I <mop> <fact> ) v*
        <mop> ::• Vc I / I ttOO

<fact> ::» <pr'imary> I <forma t> <primary> } >v
        <format> ::« FMT [ I * I / }ft]  I
<primary> ::= f <unary> Ift<simple>

<9imple> ::« <num> I <»d> I ( <exp> )

<unary> ::- - I NOT I PARE I PARO
          I ENC ( <simple-name> ) I DEC ( <simple-name> )

<id> ::« <simple-name> <id-length> <id-width>
   <id-length> ::« [ <exp> ] I &
   <id-width> ::« < <nuw>:<num> >  I  < <exp> >
                I < <exp>(<num>) > I <> I &**

<s»mple-namc> ::« <alpha-char> ( .<name-char> I ft
   <name-char> ::= <alpha-char> I <digit> I .
        <alpha-char> ::= A I B ... I Z

<num> ::« <decimal> I //<octal>*l f<binary>
   <dccimal> ::= <diciit> i <digit> 1ft
   <octal>   ::« <oit>   I <oit>   I ft
   <binary>  ::= <bit>   \ <bit>   1ft »

<digi t> ::= 1 I 2 ... 1 3 1 0
```

"S«« previous footnote about widths.

<oi t> ::- 1 I 2 ... I 7 I 0
<bi t> ::- 0 I 1