SLIDE: AN I/O HARDWARE DESCRIPTIVE LANGUAGE[1]

by

John J. Wallace* and Alice C. Parker**

DRC-18-16-79

May 1979

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

** Dept. 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.

[1] SLIDE (formerly GLIDE) is an acronymn for Structured Language for Interface Description and Evaluation

6-[illegible]-72
(1 2 r JL
DP[illegible] - 18-16-77

Table of Contents

1. 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. At present, 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 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 detail which the reader does not need and which tends to obscure his understanding of the behavior of the hardware. For these reasons, existing low level descriptive languages do not provide the kind of hardware description needed for the above tasks. The obvious solution is to use a behavioral 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 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 and the description can be structured.

Research is currently underway at Carnegie-Mellon University to produce such a behavioral language called SLIDE 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 will allow description and

[1] Bohavioral descriptions 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.

simulation of interconnection schemes like the UNIBUS^ and the D-bus[ANSI ??]. 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.

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

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

- The language should be simple and not overburdened with obscure constructs and primitives. It should be logically consistent in semantics and syntax.

There are already a number of hardware descriptive languages; some, like ISPS[Barbacci 78a, Barbacci 78b] and DDL[Dietmeyer 78] have been exercised and a software base exists. Therefore it is difficult to justify the development of yet another hardware descriptive language. For many reasons, some of which were described above, 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, introduces the required capabilities of an I/O descriptive language, and presents the past efforts at this type of description. Then, sections 3 and 4 present salient features of SLIDE.

2. The Nature of I/O and Interface Operation

Consider the following system configuration which illustrates the basic nature 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.

[2] UNIBUS is a registered trademark of Digital Equipment Corporation.

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. In particular, semantics should exist to allow:

- Priority orderings between processes.
- Description of timing dependencies, timeouts and data I/O at fixed bit rates.
- 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, language primitives should exist which allow operations common to I/O such as bit manipulation, code conversion, FIFO buffering, parity and error checking, synchronous I/O, and combinational logic.[1]

Previous research in the area of interface and I/O description has been done[Bell 72, Knoblock 74, Marino 78, Sorensen 78, Vissers 76]. However, this research has either produced gate level languages, circuit level languages, or incomplete proposals such as the port descriptions of Bell and Newell[Bell 72] and Curtis[Curtis 75]. A recent proposal[Marino 78] does have some useful control constructs which are similar to those of SLIDE.

---

[1] A more oxtonsive discussion of these primitives can be found in [Parker 75].

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 languages, processes are the central unit of execution in SLIDE descriptions,[1] A process is an independent executing environment — a piece of hardware such as a device controller or a bus arbitrator. 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 on one main *process* which 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) or by using *signals* (see section 5.2).

Processes are started (called *initialization)* nonprocedurally. When each process (except for the main process) is defined, the conditions under which it is to be initialized are given. A *priority mechanism* exists which can be used to allow some processes to terminate execution or mutually excludo execution of others. This will be discussed more fully in section 4.1.

3.?. 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

---

[1] We use "description" here and no! "program" to emphasize that a SLIDE description *describes* the operation of a piece of hardware. Correspondingly, we use "execute[1]* to mean "the operation of the actions described.[1]*

timeout occurs. Within a process, complex statements can be executed in parallel or sequentially. These constructs will be discussed in section 4.3.

3.3. Oihor I/O Rolatod 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 *comp* 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.

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 arbitrators, 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* The SLIDE Languaee

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 is

described in [Parker ??].

4.1. Processes

Processes are the central unit of execution in a SLIDE description. They are initialized 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 initialized).
2. Declarations of registers, lines,[1] 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 initialization conditions are true, and (3) no process at the same subprocess level with a higher priority is executing. When a process 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 initialization conditions become true; *B* is to execute when its initialization conditions become true, but only if *A* is idle; and C can execute only if *A* and *B are* idle. This can be done by giving *A* priority 0,[2] *B* priority 1, and C priority 2. Then as soon as /Ts initialization conditions become true, it will start executing, terminating *B* or C if they were executing. When *A* finishes executing, it will restart if its conditions are still true. If not, *B* may star! if its conditions are true. If not, C may start.

A detailed example follows. Assume we are writing a SLIDE description for a disk controller. The controller is to do a transfer operation whenever the *dataready* line rises from logical 0 to logical 1. The controller is to reset (i.e. stop any on-going transfer and

---

[1] Lines arc interconnections ouch as address and data linos, bus-request and bus-grant lines, etc.

[2] Note that O is highest priority; i is next highest: etc.

reset itself) whenever the *sysreset* line rises from logical 0 to logical 1. Part of a SLIDE description for the controller is in figure 1. Lines 1 and 2 specify the conditions under which the *reset* and *transfer* processes start executing. *Reset* is given a higher priority than *transfer* since a reset should abort any on-going transfer operation.

```
[1]   INIT reset:B     UHEN sysreset  EQL /;
[21   INIT transfer:!  UHEN dataready EQL /;
        .
        .
        .
[3]   PROCESS reset;
C4~3    BEGIN
            .
            .
        here we reset the controller


[53     ENDJ

[8)   PROCESS transfer;
            .
        declarations local to transfer
            .
[7]     BEGIN

        do the transfer operation here
            .
[83     END;
```

Figure 1: Reset and transfer processes

The expressions "sysreset 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 "sysreset EQL 1" and "dataready EQL 1^H which are true whenever the lines are logical 1.

IS many controllers, each with its own *transfer* and *reset* processes, are to be connected to a bus, the overall structure of the resulting SLIDE description is shown in figure 2.

In our example about processes *A,* S, 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

```
MAIN PROCESS bus;
        .
        .
   global bus declarations
        .
        .
   INIT device1:1 WHEN TRUE;
   INIT device2:1 WHEN TRUE;
        .
        .
        .
   INIT devicen:1 WHEN TRUE;

   PROCESS device1;
           .
           .
        INIT reset:0    WHEN sysreset  EQL /;
        INIT transfer:1 WHEN dataready EQL /;

        PROCESS reset;
           .
           .

        PROCESS transfer;
           .
           .

        BEGIN !device1!
           WHILE TRUE DO NOP;    !idle forever!
        END;

        .
        .
   other device processes are similar
        .
        .

BEGIN !main process!
   WHILE TRUE DO NOP;    !idle forever!
END;
```

Figure 2: Bus description with n device controllers

from starting while any process is executing. This is done in the *arbitrator* process in the UNIBUS description which is included in [Parker ??]. The *arbitrator* process is reproduced in figure 3.

In the arbitrator example, the 1 bit open collector variable, ***bg.enable***, is initially set to 1 by

```
PROCESS arbitrator;
    !
     Process arbit is the bus arbitrator.  It gives the bus
     to the request of highest priority.
    !
    oc bg.enableo; ! bus grant enable — 1 bit !

  • INIT npr.grant:1 UHEN npr AND bg.enable;    ! non processor req !
    INIT br7.grant:2 UHEN br7 AND bg.enable;    ! bus request level 7 !
    INIT brG.grant:3 UHEN brG AND bg.enable;    ! bus request level 6 !
    INIT br5.grant:4 UHEN br5 AND bg.enable;    ! bus request level 5 !
    INIT br4.grant:5 UHEN br4 AND bg.enable;    ! bus request level 4 !

    PROCESS npr.grant;        ! gives a non processor grant !
       BEGIN

            bg.enable «- 0;   ! disable bus grant enable immediately !

            grant the bus here

       END;
       .
       .
    the other grant processes are similar
       .
       .

    BEGIN ! executable statements for process arbitrator !
       bg.enable «- 1;        ! enable bus grants !
       delay until the bus is granted
    END;   ! of process arbitrator !
```

Figure 3: UNIBUS arbitrator process demonstrating mutual exclusion

the *arbitrator* process. This allows the grant process *(npr.grant, br7.grant$_f$* etc.) with the highest priority to start executing. Immediately each grant process sets *bg.enable* to 0 which excludes any other grant process from becoming initialized.

4.2. 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.

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.

4.2.1. Hardware Declaration

Registers, asynchronous lines, and arrays of memory can be declared with a *hardware* declaration. The notation is similar to the ISPS notation described in [Barbacci 78a, Barbacci 78b]. For example:

```
TTL OC data.bus<7:0>, address.bus<7:0>;
```

declares two 8 bit wide TTL open collector bus segments, one named data.bus, and one named address.bus.

```
ECL INT count.register<7:0>;
```

declares a single 8 bit ECL logic register which is internal to this process.

```
MOS INT mem[1023:0]<15:0>;
```

declares a 1K by 16 bit MOS memory which is internal to this process.

4.2.2. Synchronous I/O

Description of synchronous I/O (i.e. I/O 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 tape1<8:0> @ 10000;
```

declares the synchronous transfer of 9 bits in parallel from/to a bus named tape1 at a rate of once 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 s<> @ 50000;
```

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

```
DELAY UNTIL s EQL |1|1|1|1|1|0|;
```

this reads *delay until s equals the time-ordered sequence of values 1, 1, 1, 1, 1, then 0.*[1]

---

[1]The delay statement is discussed in section 4.3.1.

4.2.3. 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 4.3.2.

4.2.4. Combinational Logic

Combinational logic can be declared with the *comp* declaration.[1] For example:

```
COMP absum<8:0> i= a<7:0> + b<7x0>;
```

declares combinational IORJC to evaluate the sum of a and 6. Since a and 6 are both 8 bits wide, *absum* is declared to be 9 bits wide. The effect is similar to a function Collr *every* time *absum* is used in an expression, the sum of a and 6 is used instead.

4.2.5. 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 comp declarations.) SLIDE has a special declaration for associative memory tables. For example:

```
TABLE grey <1:0><1I0>
        'f00=>'!00,
        'f01=>'!01,
        'f10=>'fII,
        'fII=>'f10j
```

This is a grey code conversion table specified in binary.^ The table above, named *grey,* takes 2 bits as input and produces 2 bits as output. The last 4 lines specify the conversions. A table can be accessed with the *encode* and *decode* unary operators. For example:

```
ENC(grey) 'f10
```

has the value '11, and:

```
DEC(grey) 'fII
```

has the value '10.

---

[1] "Comp" stands for "compound"

[2] In SLIDE, a singlo quote (') indicates a binary number; a pound sign («) indicates an octal number.

4.2.6. 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, Ht/arta6/e-name<i:j>" references the *ith* through *jth* bits of *variable-name {L* and *j* must be constants). Secondly, Hi/arta6/e-fia/7ie<e(i)>w references the *i* bits of *varcable-name* starting at bit position *e (i,* the bit slice width, is a constant, but e, the starting bit position, can be an arbitrary expression).

4.2.7. 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 4.

bituise logical operators:
OR, XOR, AND, EQV

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

arithmetic operators:
+, -, *, /, MOD

format and concatenation operators:
FfiT (pattern)[1], @

unaru operators:
-„ NOT (bitwise), ENC *{table)*, DEC *{table)*,
PARE, PARO[2]

Figure 4: SLIDE operators

4.3. 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.

---

[1] The format oporaior makes an arbitrary bit pattern from two sources.

[2] PARE and PARO return even/odd parity bits.

4.3.1. Dolay Statomont

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

- delaying for a fixed period as in

```
DELAY 100;
```

which delays execution for 100 clock pulses.

- delaying until some condition becomes true as in

```
DELAY WHILE code<2:0> EQL #7;
```

which delays execution while the 3 bits of *code* are equal to octal 7.

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

This timeout capability is very important. For example, assume a bus arbitrator grants the bus to another process by raising the *busgrant* line. Within 500 clock pulses, it expects the process to acknowledge by raising the *ack* line. If this does not occur, the arbitrator should timeout, then raise the *sysresct* line for 100 clock pulses. A SLIDE description of this is in figure 5. (A "NEXT" used as a statement delimiter forces sequential execution.) If 500 clock pulses elapse before *ack* is raised high, the statements [3] through [7] are executed.

```
[11     busgrant <- / NEXT        ! grant the bus !
[2]     DELAY 500 UNTIL ack EQL /
[33           ELSE BEGIN          ! do this if a timeout !
[43                sysreset •- / NEXT        ! reset the bus !
[51               DELAY 108 NEXT
[6]               sysreset 4- \
[7]           END
```

Figure 5: Delay statement example with timeout

4.3.2. 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 *combuf.* If the buffer was not empty, we process the command. Otherwise, we idle for 100 clock pulses. After this, we repeat. The SLIDE description to do this is in figure 6.

```
tu    UH1LE TRUE DO
[2]     BEGIN
[33         command «- combuf NEXT          ! get the next command !
[43         IFERROR
[SJ               THEN DELAY 180
[61               ELSE BEGIN
                         •
 •                       •
                       process the command
                         •
                         •
E7J                    END
[8]       END
```

Figure 6: Iferror statement example

4.3.3. Sequential and Parallel Execution

Process execution normally flows sequentially from statement to statement with the delimiter between statements being a "NEXT." If two statements are to execute in parallel, a semicolon (5) 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.*

5. Conclusions c:nd Future Research

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

5.1. Simulation and Verification

a SLIDE compiler exists [Wallace ??a], and a simulator Is being written. The simulator will allow 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, timing, etc. can be studied.

In the summer of 1979, we plan to study the verification aspects of SLIDE. That is: (1) is verifying the operation of hardware described in SLIDE possible, and (2) what types of SLIDE descriptions are easily verified.

5.2. 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* [Wallace ??b]. These represent another step toward abstraction and are similar to the *P* and *V* semaphore operations. For example, process *A* can send a signal called *s* to process *B* with the following statement:

```
SIGNAL(B:s);
```

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

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

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

```
IF RECEIYE(A:s)
        THEN ...
        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:

i. 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.

5.3. Acknowledgements

We would like to acknowledge the assistance of some people whose efforts made this work possible: Bill Lyden wrote the origional SLIDE compiler; Andy Nagle labored over the language; Mario Barbacci helped with the language design and the compiler; and Art Altman is currently working on the simulator. Also Steve Crocker and Bill Overman provided valuable feedback.

References

[ANSI ??] ANSI Technical Committee X3T9.
U.S.A. Contribution to ISO TC97/SC13 For a Small Computer-to-Peripheral Bus Interface Standard-
December 12, 1978.

[Barbacci 78a] 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., March 1978.

[Barbacci 78b] 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., March 1978.

[Bell 72] Bell,C, Grason,J., Newell,A.
Designing Computers and Digital Systems Using PDP-16 Register Transfer Modules.
Digital Press, Digital Equipment Corp., Maynard, Mass., 1972.

[Curtis 75] Curtis,D.
IDS, An Interface Description System.
Unpublished note, ALCOA.

[Dietmeyer 78]
Dietmeyer, D.
Logic Design of Digital Systems.

Allyn and Bacon, 1978.

[Knoblock 74] Knoblock,D., Loughry, D., and Vissers,C.
Insight Into Interfacing.

IEEE Spectrum :, November 1974.

[Marino 78] Marino, Edward.
Computer Interface Description.
In *Proceedings of the 17th Annual Technical Symposium,* pages . national Bureau of Standards and ACM, June, 1978.

[Parker 75] Parker, A.
A Generalized Approach to Digital Interfacing.
PhD thesis, Electrical Engineering Department, Norih Carolina State University, May, 1975.

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

[Sorensen 78] Soresen, Ib Holm.
System Modeling.
Master's thesis, Computer Science Department, University of Aarhus, Denmark, March, 1978.

[Vissers 76] 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.

[Wallace ??a] Wallace, J.
The GLIDE Compiler.
Research note, Electrical Engineering Department, Carnegie-Mellon University, April, 1979.

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