3.3 Interface and I/O Protocol Descriptions[1]

Technical Report CRI-85-22
October 30, 1985

Alice C. Parker and Nohbyung Park

---

[1]Preprint from Advances in CAD for VLSI, Vol. 7: Hardware Descriptive Languages, Ed. R. Hartenstein, 1985.

3.3.1. Introduction

Recent multiprocessing research and development has spawned an increased interest in input/output and interconnections. It has become important to document, simulate, and formally verify entire systems, including their interconnections.

Naturally, the more detailed an interconnection description becomes, the more accurate the simulation can be and the more information the description can contain. With conventional hardware-descriptive language techniques, interconnections and their interfaces can be described accurately at both the gate and circuit levels.

This low level of description is not adequate for all applications, however. Sheer size and execution time of simulation programs have precluded simulations of large interconnected systems at the gate level, even when hardware accelerators are used. Verification of system behavior is difficult at this level since abstract function must be derived by the verifier from the description; also, the unstructured nature of low-level hardware descriptions 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 or her understanding of the behavior of the hardware. For these reasons, 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. Such a description is often referred to as *behavioral*.

Behavioral descriptions differ from structural descriptions because they describe only the functions intended by the designer and not the hardware structure. Storage locations and register-transfers which exist in the hardware may be absent from the behavioral description; control hardware is implicit rather than explicit.

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, the implementation is hidden, and the description can be structured.

The Nature of I/O and Interface Operation

Consider the following example system configuration (Figure 3-7) 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 events in another concurrent sequence,
- the time steps between events can be different for different sequences, and
- 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 each of these sequences hereafter as a *process*. The general meaning of process is an independent executing environment residing in a piece of hardware such as a device controller or a bus arbiter.

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

- concurrent execution of multiple communicating processes,
- contention for shared resources between processes,
- critical sections of execution within processes,
- simultaneity of execution both within and across processes,

- processes which behave in a subordinate fashion with respect to other processes, and
- interconnections which exhibit behavior.

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).

Concurrent execution and *shared resource contention* 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 must occur when two concurrently executing processes 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, but only at the completion of a transfer protocol sequence.

Simultaneity 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 simultaneously asserted.

A ***subordinate process*** (subprocess) is an independent execution 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. An example is given later in this chapter.

In addition, interconnections exhibit *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, writing a "high" signal to a *wired-or* line does not

mean that the line will contain a "high" value; it depends on whether other devices are asserting "low" on the same wire. In general, 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 hardware descriptive language.

There is another consideration which should be mentioned here; it is an aspect of interconnection *description*. This is the notion of viewing interconnection structures as data structures. This view, allows us to treat declaration and replication of buses, crossbar switches, and daisy chains with the same ease we treat lists, arrays, and bit vectors in many programming languages.

A language designed to model the aspects of interconnection behavior described above must possess powerful and unconventional control constructs. In particular, semantics should exist to allow the following:

- priority orderings between processes,
- initiation, termination, and suspension of processes,
- interprocess synchronization primitives such as *signal* and *wait*,
- communication between processes, and
- description of timing dependencies, time-outs, and data I/O at fixed bit rates.

These semantics must include some notion of nonprocedurality to reflect the situation where many processes do not execute *until some condition becomes true*. Also required is the ability to express actions in parallel as well as sequentially.

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

It is interesting to note that two excellent comparisons of hardware descriptive languages (HDLs) which were published some time ago [Barbacci 75], [Figueroa 73] revealed some desirable properties of general-purpose HDLs. Barbacci cited the following:

- 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, and
- ability to separate data and control.

These were chosen presumably because of his emphasis on the *description* of digital systems. Barbacci 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 condition)." Figueroa, focusing on design automation, also found some of the above properties important; and in addition discussed the following:

- modularity,
- block structuring, including the existence of global and local variables,
- facilities for functions, subroutines, and macros,
- facilities for specification of parallel processes, and
- facilities for determining and controlling process interaction explicitly.

We now present a brief survey of languages and descriptive techniques which gives a historical perspective to the I/O description problem. Formal specification techniques for description of interface and interconnection operation have been used in the past. We subdivide these into the following:

- the state diagram approach,
- the flowchart approach, and
- the formal language approach.

Flowcharts, timing, and state diagrams have been used to describe a number of buses including the IEEE-488 bus (see [Knoblock 75]). Figs. 3-1 and 3-2 illustrate the use of these techniques. However, languages useful for bus I/O and port descriptions have been adopted more slowly.

Bell and Newell [Bell 71] laid the groundwork for the semantics of interface languages with their port semantics. Curtis, working with the Purdue Workshop on Industrial Computer Systems, Data Transmissions, and Interface Committee, proposed IDS, an Interface Description System [Curtis 75]. The system involved the use of the PMS (Processor-Memory-Switch) notation (from Bell and Newell) at the top level, and the use of a new language 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 the ISP (Instruction Set Processor) language with features of AHPL and with necessary timing constructs added. (Most of these have later been added to the ISPL version of ISP [Barbacci 76]).

At the same time Vissers had developed a language based on APL with timing constructs added which could be used to produce a formal description of the state diagrams describing interfacing functions. SDLC (Synchronous Data Link Controller) and the IEEE-488 interface bus had been described using this approach [Vissers 76], [Knoblock 75]. Vissers had 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 Fig. 3-3, which illustrates the states and transitions of an example taken from the UNIBUS. In this example the granting process for nonprocessor requests on the UNIBUS is described, just as in Figs. 3-2 and 3-1.

The automation, or process, is named npr.grant. There are five major states in this process: *W1-W5*. State *W1* is executed when there is a nonprocessor request and busgrants are enabled (nprbg.enable). The next state is *W2*, where there is a delay until the selection acknowledge line (SACK) is raised by the device receiving the nonprocessor grant (npg), or until 5000 time units elapse, whichever comes first. Then the nonprocessor grant line is released, and state *W4* is reached. State *W4* is a delay state until the selection acknowledge is released. Then state *W5* is entered to reenable grants. The next state is again *W1*.

Marino proposed a hierarchical descriptive system for computer interfaces, MPLID [Marino 78]. This language system was modular. However, an overall control structure which reflected the aspects of interconnection behavior we discussed earlier was lacking. Another publication in this area, [Sorensen 78], described a system modeling language based on BCPL [Richards 69]. This language had three interesting features: the ability to declare processes (concurrency), mailboxes for interprocess communication (synchronization), and uninterruptible code sequences (critical sections). The language was designed to model systems at a high function level.

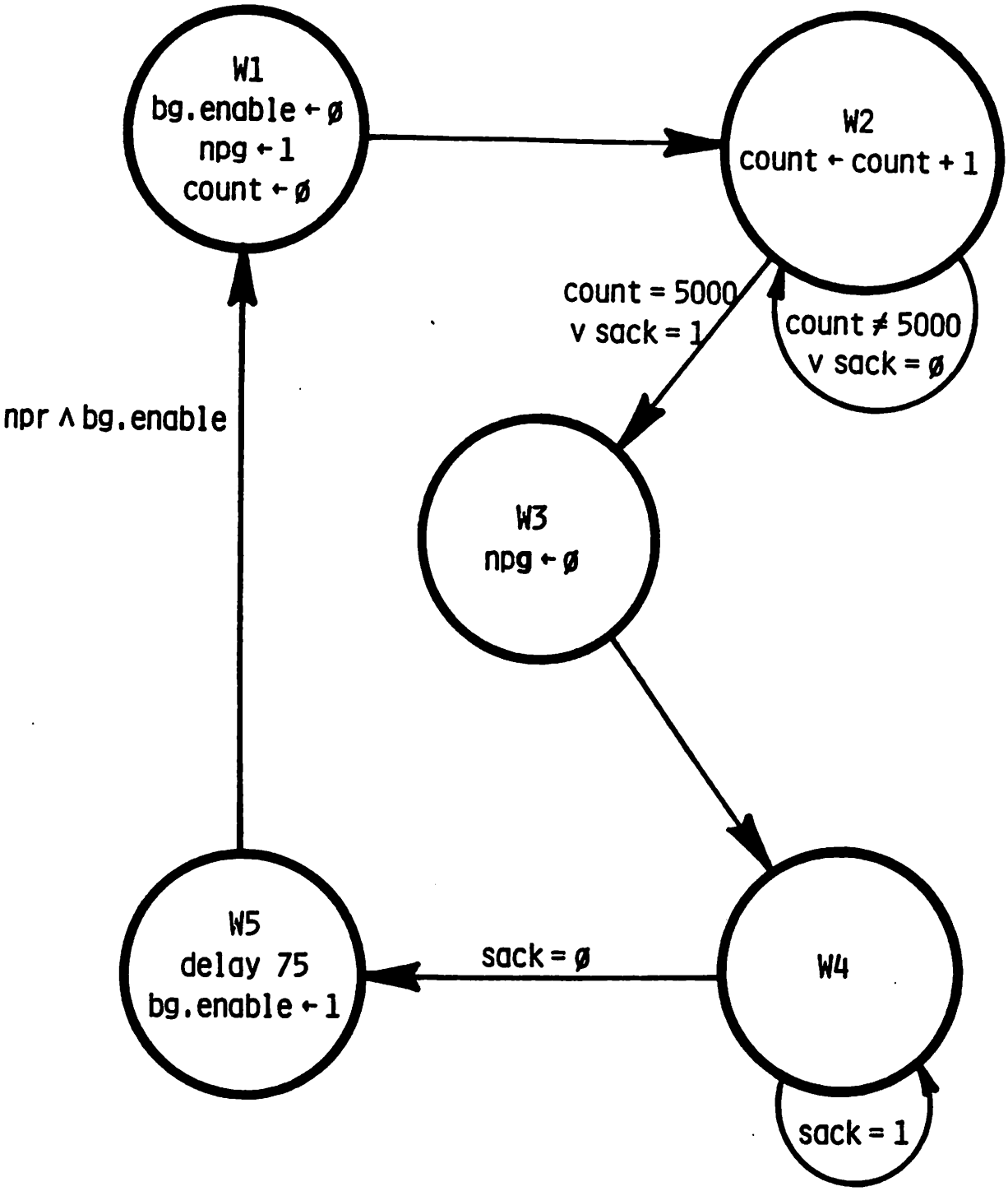

Figure 3-1: The UNIBUS[tm] nonprocessor grant process, described in state diagram form.

Along similar lines, SOL (simulation oriented language) [Knuth 64], based on ALGOL-60, did not describe hardware structures explicitly. However, it did allow concurrent processes, timing, subordinate processes, and shared-resource contention to be dealt with explicitly. One other system-level language, ASPOL [MacDougall 73], also reflected these important aspects of interconnection behavior. Concurrent processes could be declared, and processes could have priorities of execution. *Set* and *Wait* primitives were provided for

synchronization. While these languages provided synchronization and process-level constructs, they did not provide the level of detail possessed by register-transfer languages.

AHPL III [Hill 79] and DDL [Dietmeyer 78] could also be used to accurately describe and simulate interconnection behavior. However, the main reason these two languages were not entirely suitable were

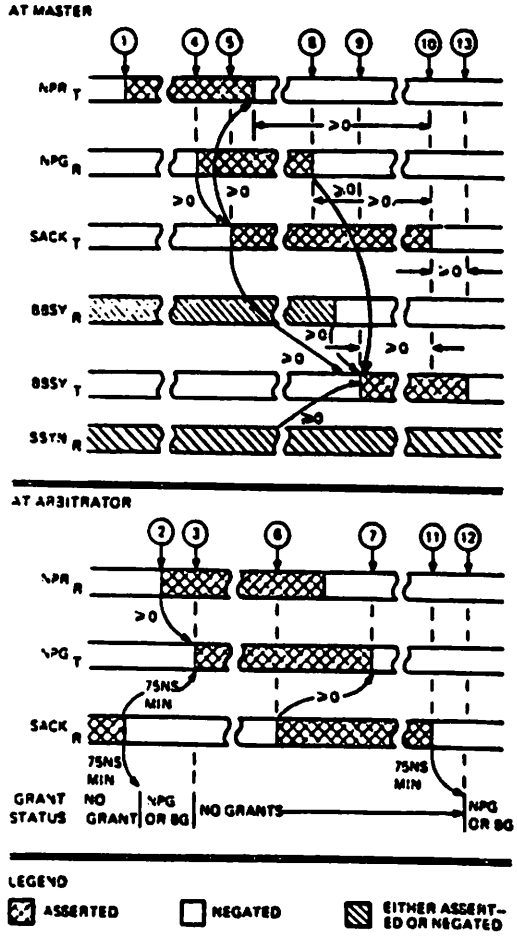

Figure 3-2: Timing of the UNIBUS[tm] nonprocessor grant process:

- Descriptions represented hardware structure rather than behavior and became large quickly.
- Fundamental aspects of interconnection behavior were implicit in the descriptions. They had to be constructed by the user with available primitives.
- An overall notion of interconnection behavior in terms of the model discussed previously was lacking in the languages.

However, there were advantages to these languages. AHPL III [Hill 79] addressed

```
      ∇npr.grant
[1] w1:→(~npr∨~bg.enable)/w1
[2]     bg.enable←0
[3]     npg←0
[4]     npg←1
[5]     count←0
[6]     →w2
[7] w2: →(count ≠ 5000 ∧ sack = 0,count=5000 ∨ sack=1)/w2,w3
[8]     count←count+1
[9]     →w2
[10]w3: npg←0
[11]w4: →(sack=0,sack=1)/w5,w4
[12]w5: delay 75
[13]    bg.enable←1
[14]    →w1
```

Figure 3-3: The UNIBUS[tm] nonprocessor grant process, described using Vissers' language.

the "structuring of interconnections" issue. Both languages provided some basic primitive operations on data which were general and powerful. Thus, past languages were either at a very high level, or synchronization mechanisms and processes had to be implicit and were lost in the details of the descriptions.

The remainder of this chapter is divided into sections dealing with timing issues (Section 3.3.2), concurrency (Section 3.3.3), protocols (Section 3.3.4), exception handling (Section 3.3.5), and electrical characteristics (Section 3.3.6). Each of these sections presents the descriptive task in general terms, followed by specific examples of selected hardware descriptive languages.

3.3.2. Timing Issues

The major descriptive issues involved with timing include

- pulse widths,
- clocks,
- delays, and
- propagation delays.

Pulse widths are sometimes important in controlling/sensing mechanical devices, such as disk controllers. Most hardware descriptive languages describe pulses by changing the value on a port, delaying a fixed period of time (to be described below), and then changing the value again. For example,[1] in SLIDE [Parker 81], the description of the generation of a 34

[1] For our examples, we will adhere to the convention of reserved words in capital letters, and user labels in lower case whenever possible.

nanosecond pulse on a connection called **sync** would be

```
sync ← 1 NEXT
DELAY 34 NEXT
sync ← 0
```

NEXT implies sequencing. The DELAY statement will be explained in more detail later.

Clocks are provided in many hardware description languages used for I/O. Usually, however, these clocks are single phase, used primarily to synchronize sequences of data coming from the outside world. In some languages with clocks, delays and other timing information are specified in clock *ticks* rather than in seconds. In SLIDE, clocks can be declared by the statement

```
CLOCK 23;
```

which specifies a single-phase clock with 23 nsec cycle time. In the SAKURA language [Suzuki 82], the clock is implicit, and time is measured in ticks.

Delays are often inserted into hardware descriptions. In many cases, the delay is merely a command to a simulator to advance the current time; in such cases the language is more like a simulation language than a descriptive one. In some cases, however, the designer might wish to specify a delay between two operations which occur in sequential fashion. Such a delay is shown graphically in Figure 3-4. This figure shows a *control and timing graph*. The graph contains an arc, representing a time range, labeled *delay*. Two other arcs are connected to this range through points, which have no time duration. These arcs represent the operations before and after the delay. Point 1 represents the termination of operations prior to the delay. Point 2 represents the start of operations after the delay. The graphical representation explicitly shows that the earlier operations have all terminated prior to the delay, and the later operations will not begin until the delay is completed. Such a delay facility is found in SLIDE, with the DELAY statement.

Propagation delays through functional units or interconnections are modeled in a similar fashion. Usually occurring after an operation, a delay statement models the functional delay. Interconnect delays are often modeled by delay statements before writing to an outside world interconnect, or after reading from one. In some representations, like timed Petri nets, delays can be attached to transitions. Graphically, function propagation delays and wire propagation delays are shown in Figure 3-5. This figure illustrates a function, *f1*, an input value *a1*, and an output value *a2*. The time range during which value *a2* exists begins some delay δf_1 after the point during which the value *a1* begins to exist.

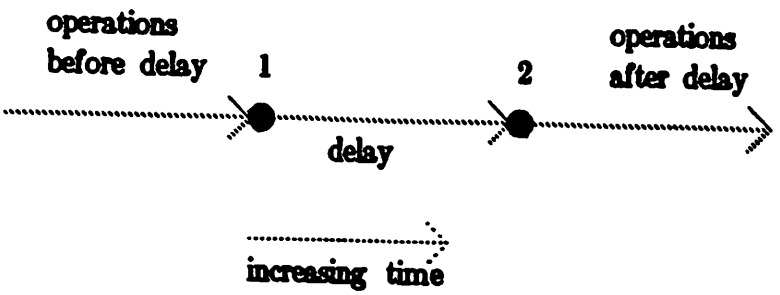

Figure 3-4: Delay illustrated with a control and timing graph.

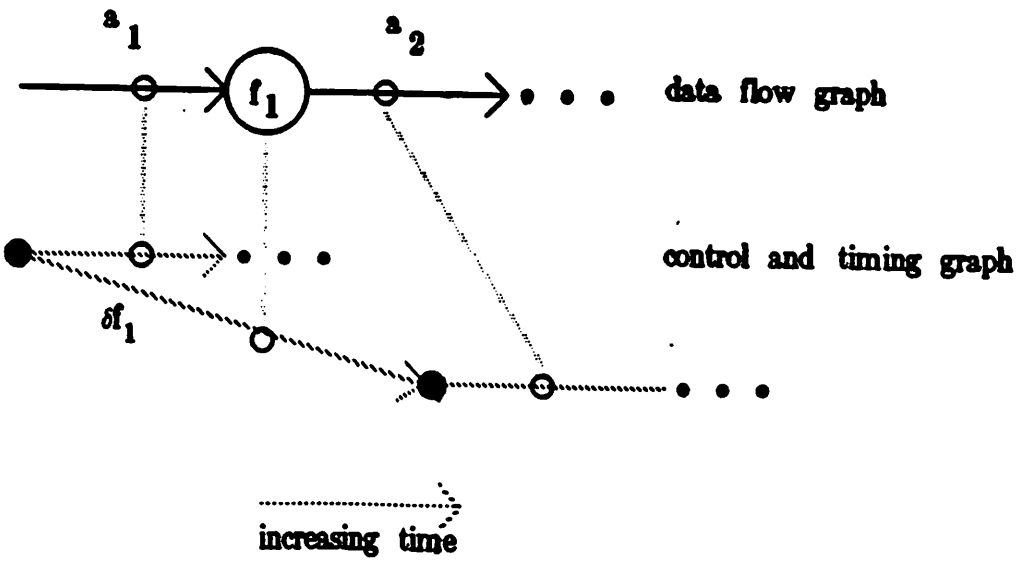

Figure 3-5: Modeling a propagation delay.

The SARA (System Architects' Design Apprentice) system [Razouk 80] represents systems in three modeling domains, a control graph, a data graph, and an interpretation. The control graph contains tokens, arcs which direct the flow of tokens, and nodes which represent events. The data graph is a structural view of the system, and the interpretation describes the behavior of each module in the data graph, using an extended version of PL/1 which includes timing.

In SARA the interpretation contains the statement

@DELAY=3 @msec;

which models the propagation delay of the **process** description in which the delay statement is found.

3.3.3. Concurrency

Concurrency is the dominating issue in modeling I/O and interfaces. Descriptions of concurrency involve

- asynchronous signals (e.g. reset),
- synchronization,
- communicating processes,
- subprocesses, and
- competing processes.

Asynchronous signals are signals (perhaps values or wires) which trigger immediate actions in the hardware. The classic example of an asynchronous signal is a reset line, which is not synchronized with a clock and which causes an immediate "escape" to a new state. SLIDE uses such a signal in a limited capacity. Processes can be initiated asynchronously (nonprocedurally) whenever the INIT expression becomes true. For example,

INIT transfers WHEN $(a = 3) \wedge (b < 5)$

states that the **transfers** process should be started at the precise moment that the expression $(a = 3) \wedge (b < 5)$ becomes true. The action of initiating a process can terminate other processes, allowing SLIDE to model "escapes".

FLEX [Comer 85], achieves a similar effect with *triggers* and *input wire specifications*. These statements have the form

[trigger-name|input-wire-expression] → statement;

Execution of these statements is delayed until the input wire expression or expression associated with the trigger-names becomes true. The trigger becomes inactive once handled. The value of the input-wire-expression does not change as a result of being handled.

Synchronization of multiple processes can be handled in a number of ways. Processes communicate in HDL descriptions by reading/writing shared (global) variables, wires or registers, or by messages. If communication is performed through shared variables, then *semaphores* or *signal-wait* commands allow access to the variables to be controlled. *Busy-waiting* is used to delay execution, or execution is specifically delayed through a *delay* statement. In SLIDE, a process can suspend itself with a statement of the form

[DELAY UNTIL|WHILE]<condition>

which waits for a condition (on a shared register or on external wires) to become true (UNTIL) or false (WHILE). This DELAY suspends its own process execution, while the INIT statement

specifies the conditions under which another process can begin. Statements following the DELAY are suspended; statements following the INIT are processed as usual. [Parker 81] also proposed two synchronization primitives, *signal* and *receive*.

SAKURA has guarded statements which suspend execution until the guard becomes true, similar to the SLIDE DELAY statement. The syntax is

WHEN Event → Statement

Events are associated with changing Boolean identifiers, and can be designated as UP, DOWN or CHANGE. For example,

WHEN init UP → initialized ← TRUE

specifies that when **init** is raised, **initialized** gets set to TRUE. A statement with a set of guarded commands implies that the first event which occurs selects only the associated commands to be executed.

Path Expressions ([Lauer 80], [Campbell 74], [Anantharaman 84]) express a coarser grain of synchronization. Though normally used to describe software processes, they are also applicable to hardware descriptions, as shown in [Anantharaman 84]. This paper gives a simple example with two reader processes R_1 and R_2 and a writer process W:

path R_1 + W end,
path R_2 + W end.

This states that either one or both of the reads could occur, or the write could occur, but read and write are mutually exclusive. The temporal part of Flex allows specification of path expressions.

S_A^* [Dasgupta 82] provides an abstraction called the *synchronizer* with operations AWAIT and SIG. Variables are declared of type BIT as follows: (taken from [Dasgupta 82]).

SYNC ibuffer_full:BIT(0)

and used as follows:

AWAIT ibuffer_full

.
.
.

SIG ibuffer_empty

where the process is waiting for the **ibuffer_full** signal and then signaling **ibuffer_empty**.

[Piatkowski 82] describes a state architecture notation (SAN) which provides for communicating finite state machines (FSMs). FSMs change state upon the arrival of pulsed inputs (inputs defined only at discrete times). As the FSMs change state, the outputs are pulsed. This notation is shown in Figure 3-6, taken from [Piatkowski 82]. **cs** refers to current state, **nts** to next state, the **x**'s are inputs and the **ntzs**'s and **ntzp**'s are outputs.

```
FNS : list next state information

  cs / xp.1, xp.2,... / xs.1, xs.2,... ⇒ nts ;
  cs / xp.1, xp.2,... / xs.1, xs.2,... ⇒ nts ;
   .
   .
END;

FOUTP : list pulsed output information
  cs / xp.1, xp.2,... / xs.1, xs.2,... ⇒ ntzp.1, ntzp.2,... ;
  cs / xp.1, xp.2,... / xs.1, xs.2,... ⇒ ntzp.1, ntzp.2,... ;
   .
   .
END;

FOUTS : list static output information
  cs  ⇒ ntzs.1, ntzs.2,... ;
  cs  ⇒ ntzs.1, ntzs.2,... ;
   .
   .
END;
```

Figure 3-6: Finite-state machine specification format.

In SAKURA, processes communicate by reading and writing a shared storage element called a *node*, through their own bidirectional ports. Two parallel processes, reading and writing, are described in this fragment of a SAKURA description, taken from [Suzuki 82].

```
PAR{
   DO -- Reading process
    WHEN Read Req UP:
     Data Av ← FALSE;
    WHEN Read Req DOWN:
     Data Av ← ~rp
 ENDLOOP
  //
 DO --Writing process
    WHEN Write Req UP:
     Space Av ← FALSE;
    WHEN Write Req DOWN:
     Space Av ← ~(wp MOD size) + 1
 ENDLOOP}
```

Execution is held up until the WHEN event occurs, which allows parallel processes to be started or resumed.

SARA describes process communication through **send** and **receive** signals which describe state transitions. Each process is described with a separate control graph and a signal causes state transitions by controlling the movement of tokens through each control graph. FLEX describes process communication by allowing multiple processes, each with its own flow of control, to execute in parallel. The process can share global variables, or communicate through the *trigger* and *input-wire-expressions* described earlier. Mutual exclusion in shared variable access is provided via uninterruptible atomic actions, which can be any normal action (including a procedure).

An example of FLEX [Comer 85] is the segment

```
PAR
 DO alive → reset_idle(alive.up_signals)OD;
 DO timer → inc_idle()OD;
RAP;
```

Where **alive** and **timer** are triggers, **reset_idle** and **inc_idle** are procedure calls, and the two DO statements represent parallel processes, indicated by the PAR...RAP construct[2]. **reset_idle** is invoked when the **alive** trigger must be handled, and **inc_idle** invoked when the the **timer** trigger must be handled. Actual trigger expressions would have been given earlier in the trigger declarations.

Concurrent Prolog has been used for hardware description and simulation [Suzuki 85]. Concurrent processes are subgoals to be repeated, and communicate through shared parameters. The direction of data flow is indicated with a "?" after the variable which is to receive the value. For example (taken from [Suzuki 85]),

```
G:-A(x),b(x?)
```

describes two concurrent processes **A** and **B**, and a shared variable **x**. A value is assigned to **x** by **A** and is used by **B**. A *commit* operator does not allow variables to be accessible by other processes until all subgoals which might affect the value of the variable succeed within a process. Concurrent Prolog deals with time as a shared variable, like any other.

---

[2] In FLEX, parallel processes are actually intended to be executed sequentially, since the target machine being described is a uniprocessor. This is a limitation of the target machine rather than the language itself.

Behavior Expressions (BE's) [McFarland 83] describes the behavior of a single *process* by describing all possible sequences of events. Events can be reads or writes to the external world. Predicates are used to describe the conditions under which events occur. Thus, process communication can be described implicitly by specifying the read and write events. The Behavior Expression

$$[R(x_{in}):true \cdot W(x_{out}):(x_{out})=1 \wedge x_{in}(j_{in})=1)]^{*}l.$$

repeatedly reads x_{in} and writes a 1 to x_{out} only if the value it has just read is a 1. If the last value read is not a 1, it does not write anything, but begins the next repetition immediately. *l* is a loop counter.

Subprocesses are the required subordinate processes we describe in the Introduction. Subprocesses are processes started (initiated) by outer processes. A SLIDE description consists of one *main process* which syntactically encompasses all other subprocesses (much like an ALGOL program consists of one main program which encompasses all subroutines). Variables global to the entire description are declared within the main process. Variables which are local to a subprocess are declared within that subprocess. Processes nested at the same level model either concurrent process execution or processes competing for shared resources. 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 by using *signals*.

Processes are started (called *initiation*) nonprocedurally. When each process (except for the main process) is defined, the conditions under which 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. Subprocesses terminate when the outer processes in which they are declared are terminated.

By the same token, subprocesses can only start when their outer processes are executing. An example of this is shown below. Process **flag.detect** detects a flag of six continuous one bits with preceding and succeeding zero bits. Once a flag is detected, it allows process **serial.to.parallel** to start. However, flag detect continues to execute, and when the end flag is detected, terminates process **serial.to.parallel.**

```
PROCESS flag.detect;
    INIT serial.to.parallel:0 WHEN flag;

    BEGIN
    (end flag detection is performed here)
    END;

PROCESS serial.to.parallel;
    BEGIN

    END;
```

The zero in the INIT statement indicates the priority of the **serial.to.parallel** process.

Competing processes exist when processes compete for a shared resource, such as a hardware module or bus. Path expressions allow description of mutually exclusive processes as in the following:

```
PATH reset + normal END;
```

which states that either the **reset** or **normal** process is to execute but not both concurrently. Predicate Path Expressions (PPEs) [Andler 79] allow process execution only when the predicate associated with a process is true. For example,

```
PATH <reset [init] | normal [~ init]>END
```

describes the same **reset** and **normal** processes, but the conditions under which each could execute are now specified. The | specifies exclusive selection.

In SLIDE, each process has an explicit priority. Informally, a process starts executing when: 1) the process of which it is a subprocess 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 starts executing, all sibling processes which are at the same subprocess level, have a lower priority, and are executing are terminated (i.e. killed).

Priorities can be used to time-order the execution of processes. For example, assume we have three processes, *A*, *B*, and *C*, no two of which can execute concurrently. *A* is to always execute as soon as its initiation conditions become true; *B* is to execute when its initiation conditions become true, but only if *A* is idle; and *C* can execute only of *A* and *B* are idle. This can be done by giving *A* priority 0, *B* priority 1, and C priority 2. Then as soon as *A*'s initiation 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 start if its conditions are true. If not, *C* may start.

In our example, 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 allows for mutual exclusion, and critical sections of processes can be protected from preemption (termination).

A detailed example follows. The example system configuration is shown in Fig. 3-7. 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 internal **dataready** 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 Fig. 3-8. Lines 1 and 2 specify the conditions under which the **Dreset** and **Dtransfer** processes start executing. **Dreset** is given a higher priority than **Dtransfer** since a reset should terminate any on-going transfer operation.

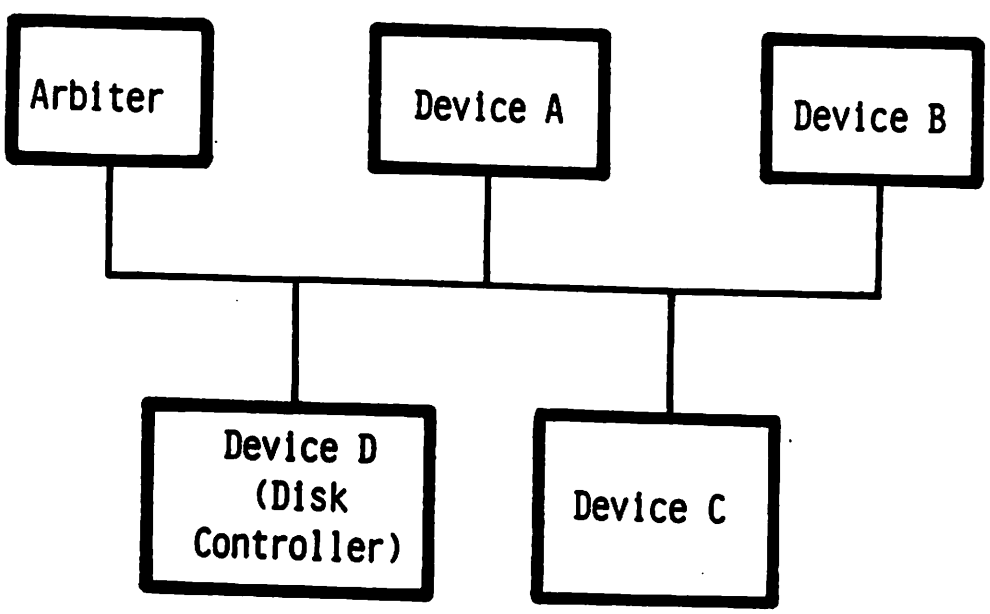

Figure 3-7: The example system configuration.

The expression **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 contain logical 1.

If many controllers, each with its own **transfer** and **reset** processes, are to be connected to a UNIBUS, the overall structure of the resulting SLIDE description is shown in Fig. 3-8. Here we have an arbiter, a set of devices, and their interconnections. Notice that the arbiter can always be *initiated* as long as it is not being initialized via the **arbiter** PROCESS, since the **arbiter** PROCESS *initiation* condition is always true.

```
MAIN PROCESS UNIBUS;
   global bus declarations
   INIT Iarbiter:0 WHEN. initline EQL /;
   INIT arbiter:1 WHEN TRUE;
   INIT Dreset:0 WHEN initline EQL /;
   INIT Dtransfer:1 WHEN dataready EQL /;
   INIT Areset:0 WHEN initline EQL /;
   INIT Atransfer:1 WHEN adataready EQL /;
        .
   other device processes are initiated similarly
        .
   PROCESS arbiter;
        .
        BEGIN
        arbitrate the bus
        END
        .
   PROCESS Iarbiter;
        .
        BEGIN
        initialize devices on the bus
        END
        .
   PROCESS Dreset;
        .
        BEGIN
        reset the disk controller
        END
        .
   PROCESS Dtransfer;
        .
        BEGIN
        transfer data on the unibus
        from disk to memory
        END
        .
   other processes are similar

   BEGIN          !Main process execution!
        DELAY WHILE TRUE          !idle forever!
   END;
```

Figure 3-8: The overall structure of an example SLIDE description.

3.3.4. Protocols

The study of protocols includes sequential synchronous data, handshaking, latching and buffering. The focus in this chapter is on description of *hardware* protocols, although some concepts are common to both hardware and software protocols.

Sequential, synchronous data is data arriving or departing to the outside world sequentially, at a fixed rate. 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. Ada has been used to describe such situations as a history of values [Barbacci 84]. *Step signals* correspond to transient values, and *time signals* are sequences of step signals, one step signal per unit time. Time signals can be examined at any point in the past, or at the current time. This is done by keeping a record of step signals and time signals as part of the pin manager package. Thus, all values which appear on pins are recorded according to the specifications found in a "pin manager" package.

Piatkowski [Piatkowski 82] also describes this situation. The State Architecture Notation supports streams of input and output data. Only the most recent data is accessed, however. Stream input and output data has reserved names. For example, **XS.1** is static input 1 (A static input is one where values only change immediately following each sample time), and **ZS.1** is static output 1. Pulsed variables (e.g. **XP.1**) are defined only at the variable's sample times.

SLIDE allows a limited description of synchronous I/O with the SYNC declaration. For example,

SYNC @ 100000 LINE Tape 1<8:0>

declares the synchronous transfer of 9 bits in parallel from/to a line named **tape 1** at a rate of one transfer per **100000** 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). for example if *s* is declared as

SYNC @ 50000 LINE s<>

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|1|0|.

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

Handshaking is not usually dealt with as a primitive. In most HDLs which describe I/O, handshaking consists of setting and resetting shared variables or interconnections. SARA supports description of handshaking in this manner, as does Piatkowski's SAN.

SLIDE supports handshaking by the values in shared registers or on shared wires, in the same way that synchronization is performed. An example handshaking sequence is shown in figure 3-9.

```
[1]     npr ← / NEXT          ! grant the bus !
[2]     DELAY 5000 UNTIL sack EQL /NEXT!Clock is a 2 nanosecond!
                                  !clock!
[3]     npg ← \;     !lower the grant line!
```

Figure 3-9: An example handshaking sequence.

The left arrow indicates wire **npg** is being written to. The / and \ symbols will be explained later.

Latching and buffering are also usually implicit in reads or writes to and from external variables, registers or wires. In FLEX, however, such operations are modeled in a more complex fashion. Output wires can be latched. Normally, wires output values for only one clock cycle, but latched wires must be explicitly changed. Such wires are declared as follows (taken from [Comer 85]):

d:OUTPUT TO LATCH;

where **d** is the name of the wire. Wires can be buffered, also. Buffering here implies FIFO storage, and in Flex up to three outputs can be stored at a time and later accessed. For example (also from [Comer 85]):

g:ARRAY [0..7] OF INPUT FROM BUFFER (3);
h:ARRAY [0..7] OF OUTPUT TO BUFFER (3);

This is similar to the SLIDE BUFFER declaration. An assignment to a named buffer in SLIDE puts the item at the end of the buffer, and assignment from a buffer removes the first item.

Many HDLs are able to model external signals and protocols more accurately by describing signal transitions or edges. Such transitions are modeled in SAKURA with the identifiers **UP** and **DOWN**, in temporal logic with up and down arrows, in FLEX with **change**, and in SLIDE by / and \.

3.3.5. Exception Handling Unique to I/O

Three major types of exception handling are important to I/O:

- data errors,
- missing data or missing signals, and
- asynchronous signals which signal exceptions or invoke exception-handling procedures.

SLIDE has operators that allow lookup tables to be accessed, so that data can be coded. For example,

```
TABLE gray  <2:0><2:0>
   '000 ⇒ '000,
   '001 ⇒ '001,
   '010 ⇒ '011,
   '011 ⇒ '010,
   '100 ⇒ '110,
   '101 ⇒ '111,
   '110 ⇒ '101,
   '111 ⇒ '100;
```

is a gray code conversion table specified in binary[3].

SLIDE also has a parity bit generation and checking operation.

Time-out capability is very important. For example, assume a bus arbiter grants the bus to a process by raising a **busgrant** line. Within 5 μs it expects the process to acknowledge by raising the **sack** line. If this does not occur, the arbiter will time-out, then lower the grant line to reset the bus. A SLIDE description of this is in Fig. 3-9. (A NEXT 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 time-out occurs.)

Timeouts are supported in SAN by declaring clocks. The **start** input starts the timer. If the clock runs without being reset until it times out, it pulses out a **timeout** signal to all inputs given in the clock specification.

[3] In SLIDE, a single quote (') indicates a binary number; a pound sign (#) indicates an octal number.

3.3.6. Electrical Characteristics

SLIDE supports some technology information. For example,

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

declares two 16-and 18-bit wide TTL *Open Collector Active Low* bus segments, one named **d**, and one named **a**. These correspond to the data and address segments of the UNIBUS, respectively. Because the lines are *typed*, writing to a line implies certain behavior. Writing to an open collector line, for example, implies 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 may seem low level, are needed in order to model abstract behavior.

References

[Anantharaman 84]
Anantharaman, T. S., et. al.
Compiling Path Expressions Into VLSI Circuits.
Technical Report CMU-CS-85-102, Carnegie-Mellon University, August, 1984.

[Andler 79] Andler, S.
Predicate Path Expressions: A High-Level Synchronization Mechanism.
PhD thesis, Department of Computer Science, Carnegie-Mellon University, August, 1979.

[Barbacci 75] 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.

[Barbacci 76] Barbacci,M.
The Symbolic Manipulation of Computer Descriptions: ISPL Compiler and Simulator.
Technical Report, Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa., April, 1976.

[Barbacci 84] Barbacci, M., Grout, S., Lindstrom, G., Maloney, M., Organick, E., and Rudisill, D.
Ada as a Hardware Description Language: An Initial Report.
Technical Report CMU-CS-85-104, Dept. of Computer Science, Carnegie-Mellon University, December, 1984.

[Bell 71] Bell, C., Newell, A.
Computer Structures: Readings and Examples.
McGraw-Hill Book Co., New York, 1971.

[Campbell 74] Campbell, R.H., and Habermann, A.N.
The Specification of Process Synchronization by Path Expressions.
Lecture Notes in Computer Science. Volume 16.*Operating Systems.*
Springer Verlag, 1974, pages 89-102.

[Comer 85] Comer, D.E., and Gehani, N.H.
Flex: A High-Level Language for Specifying Customized Microprocessors.
IEEE Transactions on Software Engineering SE-11(4):387-396, April, 1985.

[Curtis 75] Curtis,D.
IDS, An Interface Description System.
November, 1975.
Unpublished note, ALCOA.

[Dasgupta 82] Dasgupta, S.
Computer Design and Description Languages.
Advances in Computers.
Academic Press, 1982, pages 91-154.

[Dietmeyer 78] Dietmeyer, D.
Logic Design of Digital Systems.
Allyn and Bacon, 1978.

[Figueroa 73] Figueroa, M.A.
Analyses of Languages for the Design of Digital Computers.
Technical Report, Coordinated Science Laboratory, Univ. of Illinois, Urbana, IL., May, 1973.

[Hill 79] Hill, D. and vanCleemput, W.
SABLE: A Tool for Generating Structured, Multi-level Simulations.
In *Proceedings of the 1979 Design Automation Conference.* IEEE and ACM, 1979.

[Knoblock 75] Knoblock, D., Loughry, D., and Vissers,C.
Insight Into Interfacing.
IEEE Spectrum 12(5):50-57, May, 1975.

[Knuth 64] Knuth, D. and McNeley, J.
A Formal Definition of SOL.
IEEE Transactions on Computers C-13:409-414, August, 1964.

[Lauer 80] Lauer, P. E.
Project on the Design and Analysis of Highly Parallel Distributed Systems.
Technical Report, the University of Newcastle upon Tyne, December, 1980.

[MacDougall 73] MacDougall, M. and J. McAlpine.
Computer System Simulation with ASPOL.
In *Proceedings of the Symposium on the Simulation of Computer Systems.* September, 1973.

[Marino 78] Marino, E.
Computer Interface Description.
In *Proceedings of the 17th Annual Technical symposium.* ACM and National Bureau of Standards, June, 1978.

[McFarland 83] McFarland, M. and Parker, A.
An Abstract Model of Behavior for Hardware Description.
IEEE Transactions on Computers C-32(7):621-637, July, 1983.

[Parker 81] Parker, A. and Wallace, J.
SLIDE: An I/O Hardware Description Language.
IEEE Transactions On Computers C-30(6), June, 1981.

[Piatkowski 82] Piatkowski, T. F., Ip L., and He, D.
State Architecture Notation and Simulation: A Formal Technique for the Specification and testing of Protocol Systems.
Computer Networks 1982(6):397-417, 1982.

[Razouk 80] Razouk, R., and Estrin, G.
Modeling and Verification of Communication Protocols in SARA: The X.21 Interface.
IEEE Transactions on Computers C-29(12):1038-1052, December, 1980.

[Richards 69] Richards, M.
BCPL: A Tool for Compiler Writers and Systems Programming.
In *Spring Joint Computer Conference.* 1969.

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

[Suzuki 82] Suzuki, N., and Burstall, R.
Sakura: A VLSI Modelling Language.
In Artech House (editor), *Proceedings of the Conference on Advanced Research in VLSI.* Dedham, Mass., 1982.

[Suzuki 85] Suzuki, N.
Concurrent Prolog as an Efficient VLSI Design Language.
Computer 18(2):33-40, February, 1985.

[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.