

Language Barriers

Hardware Design?



The Need for Consistency Between the VHDL Language Constructs and the Underlying Hardware Design

s a subset of computer programming languages, hardware design languages (HDLs) must be necessarily precise and unambiguous. Then, hardware descriptions may be developed in an HDL and executed on a host computer to yield accurate and unambiguous results. The character of HDLs is defined by the syntactical rules that constitute a manifestation of the grammar and the semantics of the language constructs that define the meaning or effect following execution.

In addition, an HDL must encapsulate the complete knowledge of all hardware processes within its scope of modeling. There are key desirable properties of an HDL, and these properties are referred to

VHDL's understanding of the underlying meaning of entity is incomplete and, in part, erroneous.

here as "consistency" between the HDL and the underlying hardware process. Thus, the grammar and the semantics of each of the language constructs must be synthesized carefully, taking into consideration every possible and legitimate interaction between the individual constructs, and guaranteeing that accurate and unambiguous results are generated following the execution of any legitimate program. Conversely, the lack of consistency would imply a poor HDL in which the descriptions of hardware systems would be nonintuitive, unrealistic, and accompanied by high probability of errors and flaws. Such HDLs would also be difficult to learn and teach.

This article develops a set of fundamental principles underlying HDLs, starting from physics, reality, and first principles. It then critically analyzes the leading HDL—VHDL. Although a significant body of VHDL reflects superb design, it suffers from a number of fundamental flaws, which, unless corrected, will severely cripple its effectiveness and usability in the future. A primary goal of this article is to offer suggestions for modifications to key syntactic and semantic constructs of VHDL to ensure its continued success as the state-of-the-art HDL into the next century.

An HDL Primer

In the early days of computer design, to relieve the designer of the clerical and administrative burden of design, computer-program aids were developed, with one of the first examples being IBM's software for wire list preparation [1]. Then, tools to capture the interconnection of high-level computer system building-blocks such as the PMS [2] were developed. Gradually, and as early as the 1970s, HDLs emerged as a practical vehicle to facilitate the design of complex hardware in the same way that assembly language programming must yield to high-level general-purpose programming languages for developing large complex programs.

All HDLs constitute a subset of the computer-programming language designs. Like all computer-programming languages, if the intent of an HDL is known (i.e., one has the knowledge of the characteristics of the underlying process(es) that need to be expressed in the language) and the expectations following execution of a program on a computer(s) are specified, one may develop a basic set of (not necessarily canonical) language constructs and associated semantics that satisfy the intent. Clearly, the semantics will be a function of the underlying process(es) that are modeled in the language and the fundamental characteristics of the host computer(s) on which the executable programs are executed. The desirable properties of such a language would include accuracy of representation, precision of the execution results, faithfulness to the intent, simplicity, fast execution, and "naturalness" [3]; i.e., the appropriateness of its elements relative to the underlying process(es).

There is another very important requirement of all HDLs. A language encapsulates the complete knowledge of all processes

within its scope of modeling. That is, it must permit a programmer to express, accurately and precisely, every possible executable program that the programmer may design without violating any of the syntactic and semantic rules laid down by

the language. Clearly, the syntax and semantics together must encapsulate the total character of the underlying execution. Thus, the language designer is required to carefully synthesize the grammar and the semantics of each of the language constructs, taking into consideration every possible and legitimate interaction between the individual constructs and guaranteeing that accurate and unambiguous results are generated following the execution of any legitimate program. Any failure in this guarantee may seriously jeopardize the language-design effort.

In addition, for greater acceptance by the community and effectiveness of usage, the language designer is required to make every effort to realize straightforward and simple-to-use language constructs. The guideline to the language designer includes the precise purpose of the language, which, in turn, imparts to it a mandate, a well-defined set of characteristics, a clear scope, and a precise set of limitations. In this article, the intent is to capture the hardware design process faithfully and accurately and to yield fast and accurate results following the execution of the programs written in HDL.

The immediate question is: What are HDLs? Fundamentally, the goal of an HDL is to model in a host computer any hardware system—synchronous or asynchronous—at any given level of abstraction, as faithfully and accurately as possible. HDLs must facilitate hardware descriptions that are natural and easily comprehensible to the designer. Once developed, the model or the hardware description serves as an accurate replica of the original (or intended) hardware—whether already built or currently under development. Thus, the results obtained from executing the model on the host computer must match those of the actual hardware system. The model may then be either functionally simulated for design correctness and performance estimation, provided as an input to a tool for checking formal correctness, or used as an input to a hardware-synthesis tool. Clearly, without a host computer, HDLs are of limited value.

Thus, HDLs allow a hardware designer to describe accurately the target hardware design or an existing hardware system. The primary goal is to execute the description on a host computer so as to detect design flaws and errors and yield performance estimates. The execution of a hardware description on a host computer mimics the operation of the actual hardware and is termed *functional simulation*. Thus, the role of the host computer is to emulate the target hardware. The description may imply ease in the comprehension of an otherwise large and complex design, serve as a documentation for the design, and facilitate teaching the hardware design process and computer architecture. Other uses of hardware descriptions in HDLs include the ability to perform fault simulation and test generation to aid in developing re-

liable designs, timing verification to quickly ensure the timing accuracy of the designs, formal verification of designs, and automatic synthesis of the target hardware. Conceivably, a description of a hardware at a given level of detail may not

be ideally suited for all of the different uses. For example, while accurate delays are essential for functional simulation and timing verification, current synthesis tools are limited to hardware descriptions with only zero-delay assignments. This underscores the need to know the basic philosophical aims of an HDL and all of its potential uses, prior to the design of the HDL.

To summarize, the objectives of HDLs are to:

- ◆ Provide a means of describing a circuit either through a structural model (i.e., employing the physical elements utilized to implement the circuit) or a behavior model (i.e., the behaviors of the physical models are emulated in software).
- ◆ Permit the use of multiple models of the same circuit, to facilitate simulation, synthesis, etc.
- ◆ Provide a foundation for simulating circuits by encapsulating the concept of timing including delays.
- ◆ Provide a foundation for defining a circuit completely, independent of a specific implementation. The definition will constitute a starting point for developing an implementation for a specific technology (hardware/software) today as well as future realizations in advanced technologies.

HDLs are the key to understanding computer hardware and software. They are involved from the conception of any new computer or hardware design to the final implementation and testing of the IC. The US Government and industry, world-wide, has spend over hundreds of millions of dollars in developing HDLs, including VHDL. HDLs are taught in every major electrical and computer engineering department at universities around the world, and any company engaged in electronic design must utilize some form of an HDL. The field of HDL represents a unique interaction of a number of disciplines from EE and CS; namely, language design, hardware design and computer architecture, compilers, language environments, simulation, distributed algorithms, and parallel processing.

An effort to understand the basics, starting from the first principles, promises a number of additional advantages. First, it reveals the reason for each specific language construct, leading to a better appreciation of how to use it to realize the maximum benefit. As a result, one evolves into a superior decision maker and develops superb digital designs. Second, one develops a deeper understanding of the whats and whys of the underlying limitations. As circuits become faster and more complex in the future, one is better equipped to address the problems and may even initiate changes to update the HDLs. Third, knowledge of the fundamentals provides a continuity of understanding through the hundreds of publications in the literature on HDLs. Fourth, fundamentals are very convenient since they are usually

The difficulty with shared variables has finally surfaced in the multithreaded implementations of VHDL.

few in number, and they reflect a highly condensed and crystallized form of knowledge. Fifth, virtually all of the HDLs to date, including Verilog HDL and VHDL, execute only on uniprocessors, implying excruciatingly long simulation times

for large hardware systems. One of two key reasons is the lack of an intricate and correct weaving of concurrency into the HDLs. Last, should a new HDL come around in the future, one that is based on new principles superseding today's time-based and event-driven simulation principles to one conversed in the basics, it offers little resistance to understanding and mastering it.

Fundamental Requirements of HDLs

The key requirements of HDLs emanate from the fundamental characteristics of hardware, which, in turn, are defined through the following concepts of (1) entity, (2) connectivity, (3) concurrency, and (4) timing, while the designer's current view of hardware embraces the notion of (5) hierarchy.

Entity

The concept of entity is fundamental in this universe and also to our understanding of the universe. Although the universe and all knowledge about it may be one continuous thread to the creator, our understanding is that the universe, at any level of abstraction, consists of entities. The American Heritage Dictionary [4] defines the word entity as "The fact of existence, being. Something that exists independently, not relative to other things." Thus, an entity exists and its existence is guaranteed independent of all other entities. Because it exists, an entity must be self-contained; i.e., its behavior, under every possible scenario, is completely defined within itself. Because the entity exists independent of all other things, its behavior is known only to itself. Unless the entity shares its behavior with a different entity, no one has knowledge of its unique behavior. Conceivably, an entity may interact with other entities. Under such conditions, its behavior must include the scope and nature of the interactions between itself and other entities.

In the discipline of digital systems, hardware is organized through entities, at any level of abstraction. Examples include an AND gate at the gate level, an ALU at the register-transfer level, and the instruction decode unit at the architecture level. For each entity, its behavior must include all that is relevant at that level of abstraction, such as the logic, timing, control, exceptions, etc. Since every entity is independent of others and no one entity has explicit knowledge of another entity's behavior, conceivably, each entity may possess its own notion of timing, including clocks, timing constraints, delays, etc. Thus, for an HDL to successfully describe a number of such entities constituting a system, it must necessarily be capable of supporting asynchronous behavior. Philosophically, asynchrony appears to be a manifestation of the concept of entity. However, one may argue that asynchrony is the more basic of the two, and the point is therefore debatable.

Connectivity

Although they are independent, entities may choose to interact with one another. In general, a single entity all by itself is probably not all that interesting. A single gate is clearly not interesting. A single, stand-alone, un-networked computer, though inspiring a decade ago, is hardly considered exciting today. In the event that entities interact with one another, the scope and nature of entity A's interactions with entities B, C, ..., must be completely specified (i.e., for all possible scenarios) within the behavior of A. The interactions between the specific entities may be captured through connectivity information. In general, entity A may establish a connection with entity B for a limited duration, then establish another connection with entity C for a specific duration. However, in the current hardware design paradigm, the connectivity information is generally permanent and is therefore static. If hardware modules evolve to acquire mobility in the future, appropriate HDLs may need to be designed. A key rule of connectivity between entities is as follows. Where A is connected to B via a link, neither A nor B can forcibly access the other's internal behavior. B can read only what A asserts on the link, and vice versa. There is no limitation on the nature of the information that may be placed on the link, extending from a simple integer to a complex data structure. However, it is logical to expect both A and B to understand each other's asserted data, without ambiguity. For digital hardware, a link between two entities represents the wire that connects the corresponding hardware modules. At a minimum, the link must contain, explicitly or implicitly, the electrical voltage values and the times at which the voltages are asserted, starting from time 0 up to the current simulation time. There are no fundamental limits placed on the direction of information flow in the links. A link may be permanently unidirectional either from left to right or right to left, or bidirectional. Given the static nature of the connectivity in hardware, a link must be declared unidirectional, along with the direction, or bidirectional, statically.

Concurrency

Most research articles on HDL accurately observe that hardware is concurrent. This is indeed true. Concurrency is defined [4] as simultaneous occurrence. From the first principles, since every entity exists independent of others, entities must necessarily be concurrent. Therefore, in the digital design discipline, if a hardware system consists of N entities at a given level of abstraction, the potential concurrency is N . Where a host computer has available to it a total of N distinct processors, in theory, all N entities are potentially simultaneously executable, thereby achieving fast simulation of the entire system. However, the nature of concurrency in the HDLs is complex, requiring a careful analysis of the fundamentals.

Consider a simple circuit consisting of three interconnected AND gates, G1 through G3, organized through upper and lower circuits, as shown in Fig. 1. Consider first the upper circuit and that a logical 1 value is asserted at both inputs, A and B, of G1. At first sight, to one familiar with HDLs, it may appear that gate G1 will execute first for the given external input stimulus, and possibly generate an output, following which G2 will execute. In real-

ity, however, as soon as power is turned ON, both gates G1 and G2 start to function simultaneously, and this is a key basis for the claim that hardware is concurrent. Electrically, the gates operate continuously. It is not true that the gates are inactive when no signal is asserted at their inputs and that they "wake up" when a new signal is asserted at their input port(s). In Fig. 1, G2 is not even aware that it is connected to G1 and G1 does not know that its input ports are primary (i.e., external signals may be asserted on them). For our convenience, we may view hardware through its activity, but that view is not reality. The placement of the gate instances in a gate-level HDL in any order and the use of the guarded statements in the register-transfer level HDLs reflect their recognition of the concurrency of hardware.

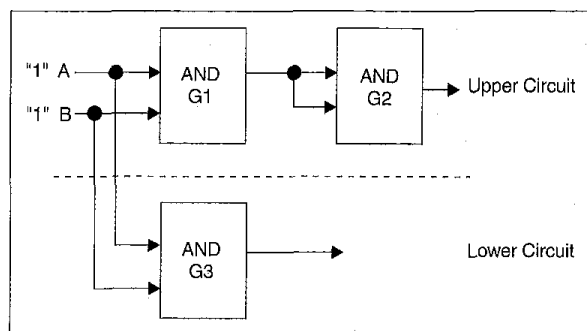
The recognition that hardware is concurrent by itself is not of much value. The issue of concurrency gains importance when the hardware description is required to be executed on a host computer. For a given hardware system, if all of its constituent entities are potentially simultaneously executable, then, assuming the availability of adequate computing resources, they may all be actually executed simultaneously on multiple processors, thereby simulating the entire hardware system quickly. Fast simulation is important since it can help us detect and fix errors and design flaws, quickly and cheaply, before developing an expensive, full-blown prototype. Of course, where the host computer is a single, sequentially executing computer, as in the case of all of the gate-level, register-transfer level, and architectural-level HDLs, any discussion of concurrency in the HDLs is significantly limited.

To better appreciate concurrency, this article will first focus on a different view of the execution of hardware descriptions under simulation, termed *event-driven simulation*. The event-driven simulation technique is the current choice of simulation due to its efficiency. A simple understanding is presented here, and for further details the reader is referred to [5]. Also, two sources of concurrency in HDL simulation are described here.

Consider

$$\vec{O}_{t2} = f(\vec{I}_{t1}, \vec{S}_{t1}), \quad (1)$$

where the output vector \vec{O} at time $t2$ of a sequential hardware entity is expressed as a function of the input vector \vec{I} and state vector \vec{S} at time $t1$. Where $t1 = t2$, the output is considered to be



1. Understanding concurrency in hardware.

generated instantaneously. However, according to relativistic physics, any action or output is limited by the speed of light and, in the digital hardware discipline, the output of an entity requires a finite time, termed *propagation delay*, to be realized at the output port. Thus, for digital designs, $t1 \neq t2$. In the digital design discipline, instantaneous output has been modeled through zero delays and it has been the source of inconsistency and much grief. Also, $t2$ cannot be less than $t1$. Therefore, $t2$ must be greater than $t1$, which is consistent with reality.

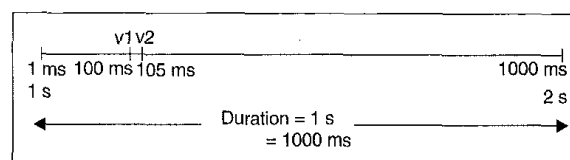
In Eq. (1), consider that a new input signal I_1 is asserted at the input port of the entity. Assume that an output O_1 is generated. Where the output is identical to its previous value, the event-driven simulation considers the case to be insignificant since there is no change at the output of the entity. In the case that O_1 is different from its previous value, the event-driven simulation considers the case significant and the new value is propagated to other entities that are connected to the entity in question. Thus, in event-driven simulation, only changes in the logical value of input and output ports of the entities are important.

In Fig. 1, assume that the circuit is powered, the circuit is stable, and that the starting point of the circuit operation is considered as time 0. Clearly, new signals are asserted at the input ports of gate G1 and it must be executed. In contrast, gate G2 does not have a new signal asserted at its input at time 0, so it need not be executed. Thus, at time 0, only gate G1 needs to be executed and the potential concurrency is 1. Consider that gate G1 generates a new signal value at time 10, which then is asserted at the input of gate G2. Also assume that a new signal value is asserted at the primary input ports of gate G1 at time 10. There is no activity between time 0 and time 10, so none of the gates need to be executed and concurrency is 0. At time 10, however, both G1 and G2 must be executed, implying a potential concurrency of 2. This describes the first source of concurrency. That is, if the host computer had two separate processors available, they could simultaneously execute G1 and G2 and accelerate the overall task of simulation. It is pointed out that despite G2's dependence on G1 for signal values, both G1 and G2 may be executed simultaneously at time 10.

In Fig. 1, now consider both the upper and lower circuits. At time 0, gates G1 and G3 both receive new signals at their input ports. They do not depend on one another for signal values. Therefore, they may be executed simultaneously, yielding a potential concurrency of 2 at time 0. Thus, if a host computer makes two distinct processors available, G1 and G3 may be executed simultaneously, thereby speeding up the simulation. This describes the second source of concurrency.

Timing

Time is defined [4] as the nonspatial continuum in which events occur in apparently irreversible succession from the past through the present to the future. In the digital design discipline, the correct functioning of systems is critically dependent on accurately maintaining the relative occurrence of events, thereby underscoring the importance of timing. Barbacci [6] correctly observes that the behavior of computer and digital sys-



2. The concept of universal time.

tems is marked by sequences of actions or activities, while Baudet et al. [7] wisely view the role of time in HDLs as an ordering concept for the concurrent computations. Piloty and Borrione [8] propose the BCL time model for HDLs where real time is organized into discrete instants separated by a single time unit, and the beginning of each time unit contains an indefinite number of computation "steps" identified with integers greater than zero. Steps provide only a before/after relationship. The concept of "delta delay" in VHDL [9] is derived [10] from Piloty and Borrione's notion of steps.

At a given level of abstraction, although each entity may have its own notion of time, for any meaningful interaction between entities A and B, both A and B must understand at the level of a common denominator of time. This is termed as *universal time*, assuming that the system under consideration is the universe. Otherwise, A and B will fail to interact with one another.

Thus, at any given level of abstraction in hardware, the entities must understand events in terms of the universal time, and this time unit sets the resolution of time in the host computer. Consider a hardware module A with a unique clock that generates pulses every second connected to another hardware module B whose unique clock rate is a millisecond. Figure 2 shows a timing diagram corresponding to the interval of length 1 second between 1 second and 2 seconds. Figure 2 superimposes the 1000 intervals, each of length 1 ms, corresponding to the clock of B. Clearly, A and B are asynchronous. Module A is slow and can read any signal placed on the link every second. If B asserts a signal value $v1$ at 100 ms and then another value $v2$ at 105 ms, both within the interval of duration 1 second, A can read either $v1$ or $v2$, but not both. The resolution of A, namely 1 second, does not permit it to view $v1$ and $v2$ distinctly. Thus, the interaction between A and B is inconsistent. If A and B were designed to be synchronous (i.e., they share the same basic clock), A would be capable of reading every millisecond and there would be no difficulty. In reality, microprocessors that require substantial time to generate an output of a software program are often found interfaced asynchronously with hardware modules that generate results more quickly. In such situations, the modules and the microprocessor understand the universal time; i.e., they are driven by clocks with identical resolutions, although the phases of the clocks may differ, thereby causing asynchrony.

Thus, the host computer, which is responsible for executing the hardware descriptions corresponding to the entities, must use the common denominator of time for its resolution of time. When the host computer is realized by a uniprocessor, the underlying scheduler implements this unit of time. When the host computer is realized by multiple independent processors, each local sched-

uler, associated with every one of the processors, will understand and implement this unit of time.

Timing Delays. An analysis of the manufacturer's timing specifications of gates and higher-level hardware modules reveal that the propagation delay plays a key role. Propagation delay is defined [11] as the time between the specified reference points on the input and output voltage waveforms with the output changing from one defined level (high or low) to the other defined level. For the output changing from logical low to logical high, the propagation delay is referred to as rise time and denoted by tp_{LH} , while for the output changing from logical high to low, it is referred to as fall time and denoted by tp_{HL} . The values of the rise and fall delay may differ from one another by as much as a factor of 3. Thus, hardware is unique in that, unlike in queuing theory, the delay is a function of the state of the input stimulus.

The behavior-level HDLs such as ADLIB-SABLE [12] and VHDL [9] incorporate propagation delays. ADLIB-SABLE includes anticipatory timing delays that result in an inconsistency. VHDL includes inertial and transport delays. While the concept of inertial delays is based on the pre-emptive scheduling principle, the concept of transport delays is observed to suffer from inconsistency and is addressed in detail in [13].

Asynchronous Timing Behavior and Asynchronous Interactions. For an HDL to successfully model the asynchronous behavior of a hardware system, the fundamental requirements are as follows. First, the language must be able to model the intrinsic, concurrent nature of hardware. Second, the language must enable the hardware system description to synchronize, during execution, with respect to an external, obviously asynchronous, signal. To realize this, it may be necessary to suspend the sequential execution of the hardware system until a specific timing condition involving the external signal is satisfied. Third, the language must permit the hardware system to include asynchronous delays in the course of its execution. This will enable the hardware description to pause at any point during its execution and for any arbitrary length of time, in universal time units. While the "waitfor" construct of ADLIB-SABLE constitutes the original effort to model asynchronous interactions, the "wait" construct in VHDL reflects a precise, accurate, and adequate language construct. Fourth, the language must enable the hardware system description to assign signal values to output and bidirectional ports asynchronously, with respect to other entities.

Timing Constraints. Along with the propagation delays, timing constraints between two or more signals constitute the complete manufacturer's timing specifications of gates and higher-level hardware modules. The Texas Instruments TTL Databook [11] lists setup, hold, minimum pulsewidth, and maximum clock frequency as the key timing constraints. The gate-level HDLs, register-transfer-level HDLs, and the architectural HDL completely lack any explicit language constructs to model timing constraints.

The difficulty with algorithms for accurate, distributed, event-driven simulation is yet another reason for the problem of executing VHDL models in parallel.

The ADLIB [14] language lacks explicit constructs to express constraints and leaves it to the user's ingenuity to model them through the high-level language constructs. The ASSERT construct permits the specification of timing constraints in the Conlan [8] HDL but is cryptic and nonintuitive. In contrast, VHDL utilizes the "signal attributes"

and other language constructs and permits a convenient specification of timing constraints.

To express timing constraints, fundamentally, the language must permit an entity to access the complete (or partial, as appropriate) history of every signal, up to the current simulation time, that serves as an input to it. At a minimum, the set of logical values and the corresponding assertion times of every input signal must be available to the entity. The entity generates the values for its output signals, so by definition, it has complete access to the history of all of its output signals.

Timing constraints may involve one, two, or more signals. For a single signal, issues such as minimum high and low durations are of concern. Where constraints involve two or more signals, they may be reorganized into sets of checks with each set involving two signals. Fundamentally, timing constraints may assume one of two forms. Consider two signals, S1 and S2. First, relative to a specific instant of time $T1$ in S1, it may be necessary to check the past behavior of S2; i.e., before $T1$. The notion of setup check in a flip-flop constitutes an example wherein one focuses on the active clock edge and examines whether the D input has been stable for setup time units prior to the clock edge. Second, relative to $T1$ of S1, it may be required to check the future behavior of S2; i.e. beyond $T1$. Clearly, this is impossible at time-instant $T1$ since any behavior is known, with certainty, only up to the present; i.e. $T1$. The future is unknown at the present. Therefore, the verification must be realized at an appropriate future time. The issue of hold time check, again in a flip-flop, constitutes an example. Relative to the active clock edge, the D input must remain stable hold time units into the future.

Hierarchy

The hierarchical design methodology is an effective technique to address the increasing size and complexity of hardware systems. In this approach, the entire system may be expressed at different levels of abstraction. At the higher levels, the description is condensed and the behavior is comprehensible at the high level, while at the lower level, the description is less compact and the details are increasingly visible. The basic character of hierarchy is that at each level of abstraction, the description of the hardware system must be complete, self-contained, consistent, and natural relative to the actual hardware at that level. That is, the behavior must be understandable to the designer without resorting to the lower-level details. In general, the resolution of time is

coarser and the simulation time decreases as one advances to the higher levels of the design abstraction.

In the hardware design process, efficient debugging often requires one subsection, say A, of the hardware system, S, to be represented at a lower level, while the remainder of the system (S-A) is represented at the higher level. Thus, while the designer is able to examine the functioning of A at a greater detail, the overall simulation is fast since the remainder of the system is executed at the higher level. Under these circumstances, two issues surface. First, although the individual timing resolutions of A and (S-A) may differ, the simulator must ensure that they both understand the common denominator of time to facilitate interaction between them during the simulation. Second, the elaboration of A at the lower level of hierarchy relative to (S-A) may be achieved either statically or dynamically. Under static elaboration, the subsection A must be identified a priori and elaborated before initiating the simulation. Under dynamic elaboration or zooming [8], the user is permitted to elaborate, during simulation, select subsections of the hardware system at a lower level. The motivation for dynamic elaboration may lie in the observations and intermediate results that warrant further examination of the subsection in greater detail.

Critical Analysis of VHDL Language Constructs

In this section, VHDL is examined in the light of the fundamental characteristics of behavior-level HDLs. First, we will concentrate on its key features and then on its limitations. We will not focus on enumerating every detail of the language, since it is extensive and has been covered in [9] as well as in a number of books [16, 17].

Characteristics of VHDL Relative to "Entity" and "Concurrency"

VHDL utilizes the word entity to define the primary hardware abstraction. Although its mention occurs earlier, chronologically speaking, in the VHDL Draft Request For Proposal (DRFP) [18], the use of entity is a significant step in the right direction. VHDL defines an entity: It represents a portion of hardware design that has well-defined inputs and outputs and performs a well-defined function.

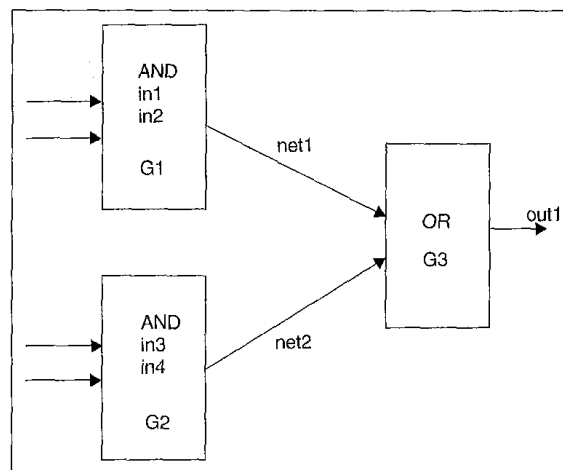
An entity is organized into two main parts—entity declaration and entity body. An entity declaration specifies a unique name for the entity and contains the common interface that is shared by one or more entity bodies corresponding to the entity declaration. The entity declaration has three subparts: an entity header, which enumerates the input, output, bidirectional ports, any generic parameters; an entity declaration that contains all items that are common to all corresponding entity bodies; and an entity statement part that contains assertions, procedure calls, and process statements [9] that are also common to all entity bodies. The second and third subparts merely reflect the common items between all entity bodies corresponding to the entity declaration. Unless a set of statements are organized explicitly through a process, all statements in VHDL are concurrent. By this token, the assertion, procedure, and process

statements in the entity statement part are all concurrent. The ports—in, out, and inout—may assume any complex data structure and are strongly typed.

The body of an entity is specified through the keyword architecture, and each entity body, corresponding to a unique entity name, carries a unique identifier. Each body may be expressed either through a sequential or concurrent set of statements. According to [9], a sequential representation is referred to as behavior. The concurrent set of statements of an entity body may assume either the form of a dataflow description or a structural organization; i.e., in terms of lower-level entities that are referred to as components. Thus, while the dataflow and behavior style descriptions may be utilized either for an entity at the highest or lowest level of abstraction, the structural style of entity body is suited for describing a static hierarchical digital design. Consider the simple circuit in Fig. 3 and assume that it constitutes an entity named Simple_circuit with input ports labeled in1, in2, in3, and in4, and output port out1. Figures 4 through 6 present three different entity bodies corresponding to the Simple_circuit.

Figure 4 presents an entity body written in the behavior style. The use of the process clause forces all of the three statements to be executed sequentially. The process is triggered for execution when any of the input ports in1 through in4 receive a new value. First, the variable temp1 receives an assignment followed by the variable temp2, and finally an assignment is made to the output port out1 after a delay of 10 ns. Clearly, a complete hardware system may be represented entirely as a behavior description.

Although the Simple_circuit consists of three concurrent gates, the behavior description in Fig. 4 reflects a black-box representation with four inputs in1 through in4, a single output out1, and an input-output functional relationship. The black-box is triggered for execution when any of the inputs change, since, in theory, a new value may be generated at the output. Thus, when the entity body "Behavior" is initiated for execution, it corresponds to a specific value of simulation time, say T_1 . In the absence of any "wait" statement that suspends execution, all of the statements in the entity body are assumed to exe-



3. Modeling a simple circuit in ADLIB-SABLE.

ecute instantaneously at the simulation time. Any signal assignment statement is also executed instantaneously, but the actual assignment of the value to the signal may be deferred. If the entity body contains a "wait," then all consecutive statements starting with the first one and up to "wait" are executed instantaneously, at the simulation time. Execution is suspended at the "wait," and when the entity body resumes execution later at a different simulation time, say T_2 , the statements following the "wait" and up to the subsequent "wait" are again executed instantaneously, corresponding to T_2 .

Consider the following scenario. Assume that new values are asserted at input ports in1 and in4 at times 0 and 10 ns, respectively. Assume further that new values are generated at the output port as a result of these input signals. The sequential process in Fig. 4 is initiated for execution at simulation time 0 ns since in1 is updated. A new value is assigned to temp1 first, and then a new value is generated for the signal out1. The assignment to out1 is scheduled for $0 + 10 = 10$ ns. Given that there are no activities between the current simulation time and 10 ns, the scheduler jumps the current simulation time to 10 ns. There are two activities—assigning the new value at out1 and re-initiating the entity body behavior for execution corresponding to the new signal value at in4. The scheduler may choose to perform the two activities in any order. Consider that it completes the assignment to out1 first, assuming that it has not been pre-empted. When the process is executed for the current simulation time of 10 ns, the new value at input port in4 is consumed, a new value is asserted to temp2, and then the new value is scheduled for asser-

```
architecture Behavior of Simple_circuit is
begin
  process (in1,in2,in3,in4)
    variable temp1, temp2: BIT;
  begin
    temp1 := in1 and in2;
    temp2 := in3 and in4;
    out1 <= temp1 or temp2 delay 10 ns;
  end process;
end Behavior;
```

4. A sequential behavior entity body.

```
architecture Dataflow of Simple_circuit is
  signal net1, net2: BIT;
begin
  L1: net1 <= in1 and in2 after 5 ns;
  L2: net2 <= in3 and in4 after 5 ns;
  L3: out1 <= net1 or net2 after 5 ns;
end architecture Dataflow;
```

5. A concurrent dataflow entity body.

```
architecture Structure of Simple_circuit is
  component and2
    port (a,b: in BIT; c: out BIT);
  end component;
  component or2;
    port (a,b: in BIT; c: out BIT);
  end component;
begin
  G1: and2 port map(in1, in2, net1);
  G2: and2 port map(in3, in4, net2);
  G3: or2 port map(net1, net2, out);
end architecture Structure;
```

6. A concurrent structural entity body.

tion at out1 at time $10 + 10 = 20$ ns. When the scheduler advances the current simulation time to 20 ns, it will perform the assertion at out1, assuming that it has not been pre-empted. Figure 7(a) presents a graphical view of the activities as a function of simulation time.

Figure 5 presents a dataflow description where all of the statements are concurrent; i.e. they may potentially execute simultaneously. Thus, the order of occurrence of the statements is not important. Two internal signals, net1 and net2, are defined and the first two statements correspond to assignments to the internal signals. An assignment may be made to net1 when either in1 or in2, or both, experience a change. Similarly, when either in3 or in4, or both, are subject to change, an assignment to net2 is realized. An assignment is made to output port out1 when either of net1 or net2 experiences a change in value. As with the behavior style description, a complete hardware system may be represented entirely as a dataflow description.

Although every statement in the entity body is concurrent in theory, in practice, their execution is controlled by a single, centralized scheduler. Thus, the implication of concurrency is severely diminished. The reason for the use of a centralized scheduler is to maintain consistency and guarantee correctness. In Fig. 5, the semantics of the statement at label L1 are that following any change in in1 or in2, at a simulation time T_i , an assignment is scheduled to be asserted to net1 after a delay of 5 ns from T_i . Similar are the semantics of the statements at labels L2 and L3. Assume that both L1 and L2 are executed at time 0. Thus, net1 and net2 are scheduled to receive new values at $0 + 5 = 5$ ns and $0 + 5 = 5$ ns, respectively. In the absence of external control, L3 may initiate its execution utilizing only the newly asserted value at net1, although the correctness argument requires that updated values at both net1 and net2 at $t = 5$ ns must be considered. As a result, execution of L3 may yield erroneous results. The role of the scheduler is to ensure that all updates to all signals corresponding to a specific simulation time are completed before activities for a future simulation timestep are permitted. In Fig. 5, if L1, L2, and L3 are scheduled for activation at simulation times T_1 , T_2 , and T_3 , respectively, first the minimum of these time values T_{\min} is computed and then only the corresponding statement(s) is allowed to execute. Following its completion, the minimum is computed again and only the corresponding statement(s) is permitted to execute, and this continues until there is no new activity.

Consider that the entity body "Dataflow" is subject to the same input signals as with the sequential entity body "Behavior." At simulation time 0 ns, the entity body is initiated for execution. Only the statement at label L1 executes since in1 receives an update. As a result, a new value is scheduled for assertion on net1 at time $0 + 5 = 5$ ns. The scheduler advances the current simulation time to 5 ns and completes the assignment to net1, assuming that it has not already been pre-empted. At this point, the entity body is initiated again and the statement at label L3 is executed since net1 has received an update. Upon execution of L3, a new value is scheduled to be asserted to out1 at time $5 + 5 = 10$ ns. Next, the scheduler advances the current simulation time

to 10 ns. The entity body is re-initiated and the statement at label L2 is executed. As a result, a new value is scheduled for assertion at net2 at $10 + 5 = 15$ ns. Also, the scheduler completes the assignment to out1 assuming that it has not been pre-empted. The scheduler advances the current simulation time to 15 ns and completes the assignment to net2. The entity body is re-initiated and the statement at label L3 is executed. A new value is scheduled to be asserted at out1 at time $15 + 5 = 20$ ns. A pictorial view of the activities as a function of simulation time is shown in Fig. 7(b). The fact that the sequence of activities in Fig. 7(b) is more detailed than in Fig. 7(a) reflects the fact that the Dataflow entity body represents more details about the input-output function in contrast to the Behavior entity body.

Figure 6 presents a structural entity body where a top-level entity may be expressed in terms of the lower-level constituent components. Since the components are in essence entities, they may, in turn, be expressed through behavior, dataflow, or structural descriptions. Thus, the structural is the most general and versatile description and closely resembles the instantiation scheme inherent in ADLIB-SABLE and Ada as an HDL [15]. In Fig. 6, the Simple_circuit is expressed through three gates, G1, G2, and G3, that relate to two types of components—*and2* and *or2*. Both *and2* and *or2* are entities with their respective entity bodies. In the structural description, first the entity declarations are re-stated and then the specific instances of the components are enumerated. The statements in the structural description are concurrent; i.e., the entity bodies corresponding to gates G1 through G3 may execute simultaneously, in principle. In practice, however, the execution semantics are similar to that discussed earlier for Fig. 5.

The "VHDL Entity" as an Incomplete Entity and Difficulties with Concurrency

Although VHDL employs the word entity to describe the basic hardware abstraction, its understanding of the underlying meaning of entity is incomplete and, in part, erroneous. An en-

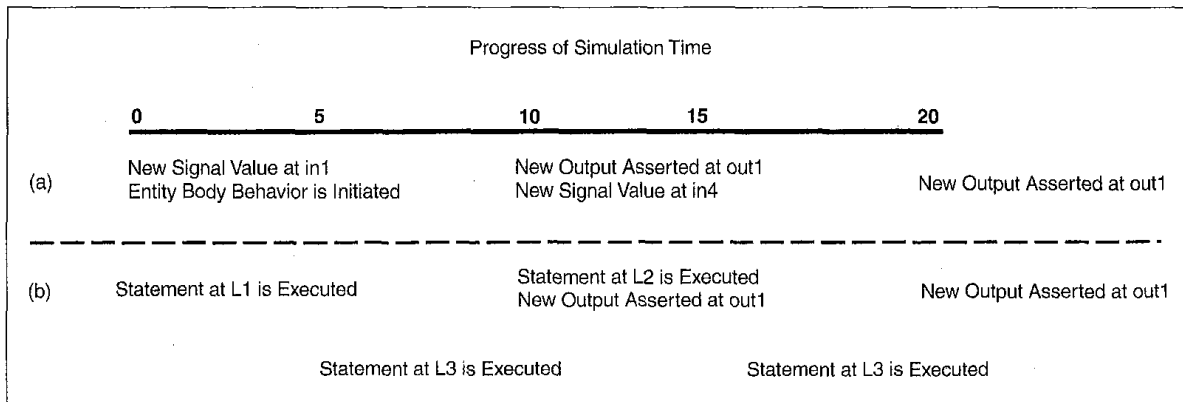
A cardinal rule of language design is never to permit ambiguity. The VHDL DRFP had reiterated it.

tity, by definition, must be self-contained, independent, and concurrent. Yet, many of VHDL's syntactical constructs and semantics violate this basic premise. The IEEE Standard VHDL Language Reference Manual [9] does not detail the

characteristics of an entity. Even the article [10] in which the original VHDL architects introduce the core philosophy underlying VHDL lacks any mention of the key characteristics of an entity. They defined a design entity to consist of an interface and one or more bodies, with the interface specifying the external characteristics of the entity and each body representing an alternate design approach consistent with those characteristics.

For instance, the inclusion of global signals in a structure description implies that there is an implicit connection of every component instantiated in the entity body. Since every component, in turn, is an entity, the notion of global signals contradicts the self-contained characteristic. By virtue of being a global signal, any update to it by a component instance must be visible to all the component instances. This requires either every instance to have explicit communication with every other instance or a centralized agent to control the activities of all instances. Upon examination of a real hardware system, it is clear that the behavior of a hardware system is controlled only by those signals that are explicitly connected to it. In fact, there is no known way to design hardware without complete knowledge of all signals, whether input, output, or both input and output.

The issue of global variables in dataflow and structure descriptions of entity bodies, labeled "shared" variables, pose the same problems as global signals. An added difficulty is that global variables not only require an immediate assignment, but that the updated value must be visible instantaneously to every entity or every concurrent statement. There is no known physical system, let alone hardware, that can achieve these semantics. In essence, global variables do not reflect any aspect of hardware and do not belong in an HDL, even if they imply programming convenience. It must be remembered that VHDL is a hardware description language, not a programming language. The diffi-



7. Timing of the execution entity bodies in VHDL.

culty with shared variables has finally surfaced in the multithreaded implementations of VHDL [19].

Because it violates a basic characteristic of entity, unlike the actual operation of the target hardware system, VHDL descriptions of hardware modules, in their current form, may not be executed concurrently on a real parallel processor.

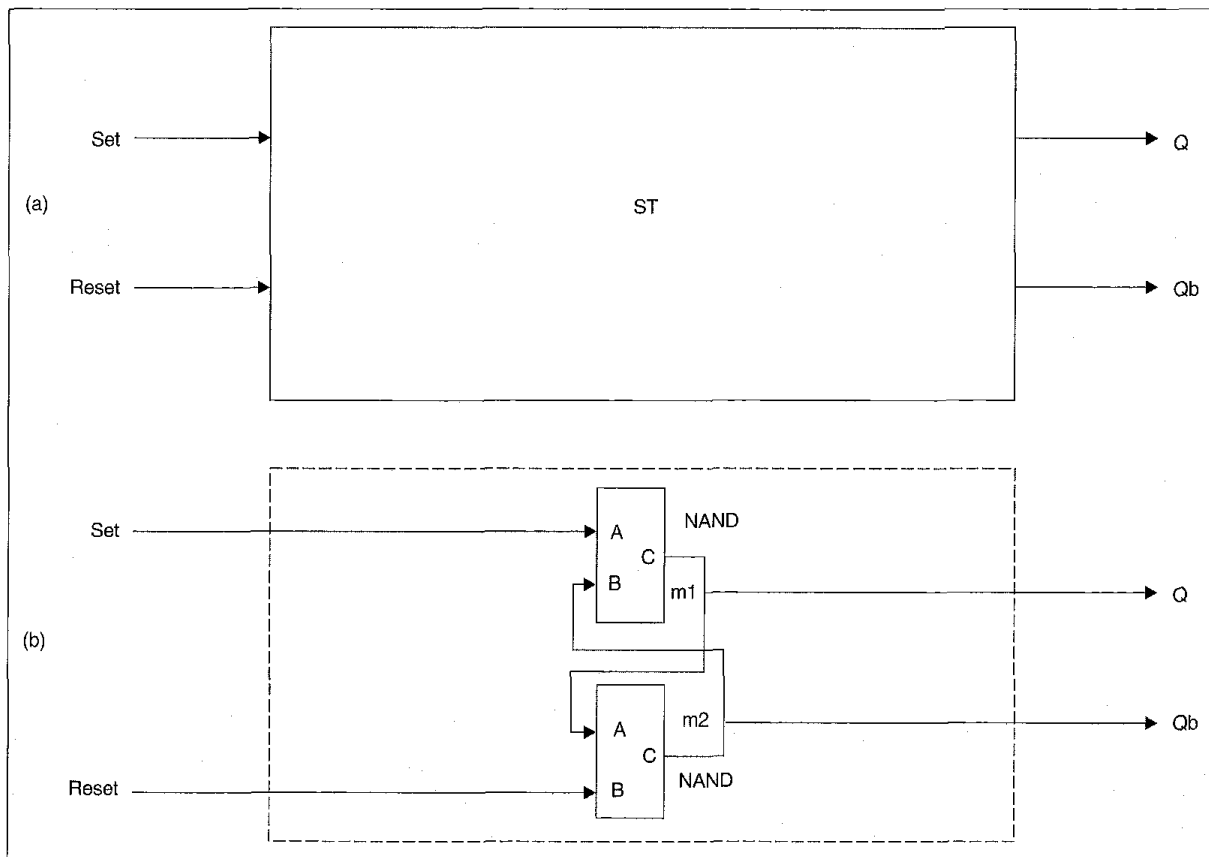
The VHDL architects viewed VHDL as a total concurrent language and needed to invent a special mechanism (process) to allow sequential statements. Any statement, unless encapsulated within a process, is viewed as a concurrent statement. Although the purpose is to reflect hardware that is basically a concurrent process, the purpose is easily achieved through the structure description where the lower-level constituent components are, in turn, also entities. Conceptually, it is unclear whether there is a unique role for dataflow descriptions. Any concurrent statement in a dataflow description must reflect a hardware process at some level of abstraction and, if so, it may easily be represented as an entity. At the pragmatic level, the individual concurrent statements in a dataflow description cannot be executed simultaneously since they are too fine-grained, they are not self contained (i.e., they may depend on global variables and signals), and they do not necessarily have their own notion of time. The

We recommend that delta delays be eliminated from VHDL.

added fact that VHDL is controlled by a centralized scheduler relegates the effort of designing individual concurrent statements even more unnecessary. Such redundancies

and unnecessary constructs, especially, without clear justification of their purpose, tend to make VHDL bulky, slow in compilation and especially execution, unnecessarily complex, and difficult to learn. There is yet another reason for the problem of executing VHDL models in parallel, and it lies in the difficulty with algorithms for accurate, distributed event-driven simulation. The reader is referred to [20] for more details.

In the Ada [21] programming language, whose task concept constitutes the basis for the entity concept in VHDL, a task specification is first declared and then followed by a task body. While the declaration ties the task identifier name with the task, the task body contains the execution semantics. Ada requires that a task specification and the corresponding task body share the same identifier. Ada does permit the creation of instances of a task type, all of which share the exact same task body. In contrast, in VHDL, any number of entity bodies, each different, may be associated with a single entity declaration. This is a violation of requirement (5) in the DRFP that mandates that any Ada syntax, if borrowed, must be adopted exactly. Furthermore, upon analysis, it is unclear



8. Port declarations in a VHDL entity.

as to the precise purpose and role of the entity declaration. In essence, an entity declaration enumerates the input, output, and bidirectional ports of an abstract hardware system. Clearly, the ports, by themselves, do not reflect the hardware system since a completely different hardware system may have identical input, output, and bidirectional ports. For instance, consider a two-input AND gate and a two-input OR gate. Except for the different entity names, the corresponding entity declarations are indistinguishable and essentially convey nothing important.

Consider an alternate strategy where the entity declarations of VHDL are eliminated and an entity is redefined to include the port descriptions, etc., as well as the body of the architecture. This newly defined entity is self-contained and one can synthesize multiple instances of it, as necessary, similar to comtypes in ADLIB-SABLE. A convention is introduced to name the new entities as X_Y, where X refers to the entity declaration and Y the architecture in the current VHDL standard. Thus, the designer can immediately recognize that two entities ADDER_GATES and ADDER_BEHAVIOR both refer to the same abstract hardware system—an adder. The designer would also understand that the entities are two distinct realizations of the basic adder. In the current VHDL, both the name in the entity declaration and the architecture identifier are required during static elaboration time. The result of this simplification would imply a simple yet strongly justified VHDL that would greatly facilitate quick and precise learning without sacrificing functionality.

Characteristics of VHDL Relative to "Connectivity"

A given hardware system is represented in VHDL at the highest level through an entity. Where the description of the corresponding entity body is either a dataflow or a sequential process, no further lower-level composition details are available, implying that the use of connectivity information is irrelevant. In contrast, when the body of the entity is expressed through structure (i.e., through components, that are, in turn, defined as entities), the binding of the "formal" ports of the components to the "actual" ports of the higher-level entity constitutes connectivity. In VHDL, the connectivity is achieved implicitly (i.e., by matching the formal with the actual parameters), unlike in ADLIB-SABLE

```
entity rsff is
  port (set, reset: in BIT; q, qb: buffer BIT);
end rsff;

architecture netlist of rsff is
  component nand2
    port (a, b: in BIT; c: out BIT);
  end component;
begin
  U1: nand2 port map (set, qb, q);
  U2: nand2 port map (reset, q, qb);
end netlist;
```

9. Entity declarations for an RS latch in the book VHDL.

```
entity rsff is
  port (set, reset: in BIT; q, qb: inout BIT);
end rsff;
```

10. Entity declarations for an RS latch in the book A Guide to VHDL.

where the user develops an explicit SDL program or in Ada as an HDL where the user synthesizes an explicit interconnection database. Thus, VHDL serves as a vehicle to both represent hardware behavior and express connectivity.

Consider the Simple_circuit in Fig. 3 and observe the connectivity description in VHDL as presented in Fig. 6. The highest level of abstraction in VHDL is the tuple {Simple_circuit, Structure}. In Fig. 6, the fact that the port identifiers "net1" and "net2" occur in the port maps of G1, G2, and G3 constitutes an implicit connection between G1 and G3, and G2 and G3. The primary input and output ports deserve different treatment in ADLIB-SABLE, Ada as an HDL, and VHDL.

The Difficulty with "Connectivity" in VHDL

Unlike in ADLIB-SABLE and Ada as an HDL, where the nets interconnecting the hardware modules are explicitly stated, in VHDL the nets are implied. A net is a physical wire(s) and the explicit style is a natural way of specifying the connectivity. In addition, a potential advantage of the explicit style may be in allowing the designer to specify its electrical characteristics and other relevant properties.

In virtually every HDL, the ports are specified as either input, output, or bidirectional. This is logical, natural, and based on our current knowledge, one cannot design hardware with any different kind of port. Yet, VHDL proposes the use of buffer as a port type without carefully justifying the need. Consider an RS latch at two levels of abstraction—behavior and structure, as shown in Figs. 8(a) and 8(b), respectively.

As required in VHDL, an entity declaration must be synthesized for the RS latch. While the ports Set and Reset are unmistakably of the input or "in" mode, there is confusion with the ports Q and Qb. Although, logically, they must be of mode "out," in the book titled *VHDL*, the author [16] states that Q and Qb must be of mode "buffer" and argues the following. The port Q of the entity declaration must be mapped on to the structure level, where Q is both an input to one of the two NAND gates and an output from the other NAND gate. Since labeling them as "inout" is likely to be viewed as an outright contradiction of common knowledge, Q and Qb are labeled as mode buffer. This is consistent with the VHDL semantics that mandate that a formal parameter (in this case, the ports of the NAND gates) of mode "in" may be mapped only onto an actual parameter (in this case, the ports of the entity declaration latch) of modes "in," "inout," or "buffer." Figure 9 presents the code segment. In another book, titled *A Guide to VHDL*, the authors Mazor and Longstraat [17] characterize the ports Q and Qb of mode "inout" as shown in Fig. 10.

While the use of mode "inout" for Q and Qb in Fig. 10 is clearly wrong, the use of buffer in Fig. 9 is unjustified. The reason for this confusion and ambiguity clearly rests on the shoulders of the VHDL architects. A cardinal rule of language design is never to permit ambiguity. The VHDL DRFP had reiterated it in mandate (1). A number of issues are raised through this scenario. First, the ports Q and Qb of a RS latch are clearly outputs, and there should simply be no questions about it. Regardless of its reasons, VHDL cannot require designers to specify Q and Qb as anything but of mode "out."

```
L1: S1 <= 1 after 10 ns;
L2: S2 <= 1 transport after 10 ns;
```

11. Delays in VHDL.

```
L1: wait on clock;
L2: wait until clock = '1';
L3: wait for 10 ns;
```

12. Syntactical usage of wait in VHDL.

Second, an entity declaration, according to VHDL, is expected to define a high-level interface that is not only applicable to any architecture body, but is independent of the lower-level details. While the issue of "buffer" vs. "inout" is raised when the architecture is of type structure, it would not have been raised for an architecture of type behavior. In the behavior architecture, the state vector of the latch would be encapsulated through a variable. Thus, VHDL's objectives are self-contradictory. Third, this scenario further strengthens the criticism that the notion of entity declaration has been borrowed from Ada without fully understanding its role and purpose in VHDL. Fourth, VHDL's proposal of "buffer" as a mode is unnatural in actual hardware systems and leaves the designers vulnerable and confused.

The VHDL language reference manual forbids a formal "in" port to be connected to an actual "out" port. This is understandable since such a mapping will leave the input port of the lower-level component undefined. However, such a restriction is unnecessary and harmful since one may encounter a formal "in" port connected to a formal "out" port and an actual "out" port, as is the case with the latch. Such types of connections are absolutely correct and ubiquitous in hardware, and VHDL cannot treat them as illegal. The VHDL compiler must check to verify that whenever a formal "in" port is connected to an actual "out" port, there is also a connection with a formal "out" port. Ironically, according to the language reference manual, such a checking is done in VHDL anyway whenever "buffer" mode is used to ensure that there is a single driving source. Therefore, at any given level of abstraction in the design hierarchy, the ports must be defined uniquely and correctly, independent of the lower-level implementation.

Characteristics of VHDL Relative to "Timing"

To achieve the timing description of both synchronous and asynchronous hardware systems, VHDL utilizes signal assignment statements, wait statements for synchronizing between signals, and timing assertions.

Signal Assignments. A signal assignment statement assumes the form, "S <= 1 after 10 ns;" where S represents a signal of a specific data type, integer in this case, and, upon execution at the current simulation time T, a 1 is

scheduled to be asserted at S at time (T + 10) ns. Unlike ADLIB-SABLE, which implements a simple anticipatory scheduling, the current VHDL IEEE Standard 1076-1993 proposes two different types of delays—inertial and transport, to address inertial delays found in digital devices and line delays associated with buses in digital systems. For accuracy in the simulation results, the VHDL simulator also implements the pre-emptive semantics for the inertial delays. Thus, VHDL achieves the timing accuracy described in Ada as an HDL while utilizing explicit timing constructs as in ADLIB-SABLE [22] to facilitate ease of comprehension among designers.

The statements corresponding to the labels L1 and L2 in Figure 11 reflect inertial and transport delays. Under inertial delays, the assignment of 1 to signal S1 may not ultimately be realized due to pre-emption by a more recent assignment. In contrast, nothing can prevent the assertion of 1 to the signal S2 at 10 ns beyond the current simulation time.

Wait Statements. A key to VHDL's capability of describing asynchronous interactions lies in its design of the "wait" construct that is inspired by the "waitfor" construct of ADLIB-SABLE. Clearly, the goal of wait is to suspend the sequential execution of an entity until a specific timing condition is satisfied.

Figure 12 presents three different syntactical uses of wait. Under label L1, the statement waits until the signal clock is subject to an event. The statement at label L2 pauses until the value of the clock signal changes to 1. Last, the statement at label L3 suspends the sequential execution for 10 ns from the current simulation time and resumes the execution of the subsequent sequential statements thereafter.

Timing Assertions. Unlike ADLIB-SABLE and Ada as an HDL, where the synthesis of the timing assertions is complex and nonintuitive to the designer, the VHDL design includes elegant language constructs to check for timing assertions. While the timing assertions in DABL are perhaps more straightforward and easily noticed in that they are declared up front within the

```
Process (Clk)
L1: if (Clk = '0' and Clk'EVENT)
L2:   ASSERT (Clk'LAST_EVENT >= clock_high_width)
      REPORT "Minimum clock high width violated"
      SEVERITY ERROR;
L3: if (Clk = '1' and Clk'EVENT) THEN
L4:   Qout <= Din after propagation_delay;
L5:   ASSERT (Clk'LAST_EVENT >= clock_low_width)
      REPORT "Minimum clock low width violated"
      SEVERITY ERROR;
L6:   ASSERT (not Din'EVENT) and (Din'LAST_EVENT >= setup_delay)
      REPORT "Setup violation"
      SEVERITY ERROR;
      end if;
L7: wait for hold_delay;
L8:   ASSERT (Din'Stable(hold_delay))
      REPORT "Hold violation"
      SEVERITY ERROR;
      ...
end process;
```

13. Synthesizing timing assertions in VHDL.

behavior model, the ability to access the history of signals in VHDL and to comparatively evaluate their relative timing provides greater flexibility to the designer, not only to verify the timing assertions but also to describe the timing behavior more intuitively.

The key attributes of signals include 'EVENT, 'LAST_VALUE, 'LAST_EVENT, 'STABLE(T), and 'DELAYED(T). For a given signal *S*, the operation S'EVENT returns true if *S* is subject to a transition at the current simulation time. The operation S'LAST_EVENT returns the elapsed simulation time, relative to the current simulation time, since the most recent transition of *S*. Even where S'EVENT is true, the S'LAST_EVENT returns the time interval between the current simulation time and the time of the previous transition of *S*. The operation S'LAST_VALUE returns the previous signal value prior to the current value of *S*. The operation S'STABLE(T) is designed to accept a time interval *T* as an argument and determine whether *S* incurs no transitions in the interval—(NOW, NOW-T), where NOW refers to the current simulation time. The operation returns a synthesized signal of duration *T*, starting at the current simulation time and extending toward the past. It has a true value when *S* lacks transitions in the interval *T* and false otherwise. The operation S'DELAYED(T) accepts a time interval *T* as an argument and returns a new signal that is delayed by *T*, relative to the original signal *S*. Since *S* is known with certainty from the origin, i.e., 0 simulation time, up to the current simulation time, the new signal is defined from $0 + T = T$ time units up to (current simulation time + *T*) time units. Clearly, the aim of this operation is to permit the user to view and manipulate the past behavior of *S*, beyond the last event, information on which may be obtained through the S'LAST_EVENT and S'LAST_VALUE attributes. It is critical to note that, at the current simulation time, the newly synthesized signal cannot offer the user a view into the future; i.e., beyond the current simulation time. It is also important to note that if the new signal is synthesized solely for the purpose of verifying timing relationships, it may not correspond to reality, and the VHDL description may be viewed as unnatural and nonintuitive.

Figure 13 presents the synthesis of timing assertions in VHDL for the four types of timing checks usually encountered in flip-flops and representative of timing checks in hardware systems. The statements at labels L2 and L5 verify that the high and low durations of the clock pulse meet the minimum requirements. The statement at L4 reflects the actual function of the flip-flop. The statement at label L6 verifies the setup requirement. The current simulation time (or NOW) is advanced to the positive clock edge and the state of the Din input between NOW and setup_delay time units prior to NOW is examined. The verification of hold time cannot be accomplished at NOW since it requires the examination of the state Din hold_delay time units into the future. The statement at label L7 suspends

Knowledge of the transactions may help greatly in debugging VHDL programs and in verifying the correctness of the VHDL environment.

the entity body for hold_delay time units and allows the simulation scheduler to advance the current simulation time. When the entity body is re-initiated, the statement at label L8 examines the state of Din signal to determine whether the hold condition is satisfied.

Timing Inadequacies in VHDL

Difficulties with the Delta Delays. According to the original VHDL architects [10], VHDL's model of time is derived from the BCL time model in Conlan [8]. In the BCL time model, the real time is organized into discrete instants separated by a single time unit and the beginning of each time unit contains an indefinite number of computation "steps" identified with integers greater than zero. The discrete instants and computation steps correspond to the macro- and a micro-time scale in VHDL.

The BCL time model poses several conceptual difficulties. First, given that a host computer is a discrete digital system, it cannot accommodate an indefinite number of steps within a finite time unit. Second, although the individual computation "steps" must imply some hardware operation, they do not correspond to discrete time instants, which are utilized by the underlying discrete event simulator to schedule and execute the hardware operations. Thus, the computation "steps" may not be executed by the simulator and, as a result, they may not serve any useful purpose. It is also noted that, fundamentally, in any discrete event simulation, the timestep or the smallest unit through which the simulation proceeds is determined by the fastest subsystem or process. For accuracy, this requirement is absolute. Assume that this timestep is T_m . If, instead of T_m , a timestep *T* is used deliberately ($T > T_m$), the contributions of the fastest subsystem or process cannot be captured in the simulation, leading to errors in interactions, and eventually incorrect results. Third, the dual time scales implied by the BCL model are inconsistent with the concept of universal time, explained earlier in this article.

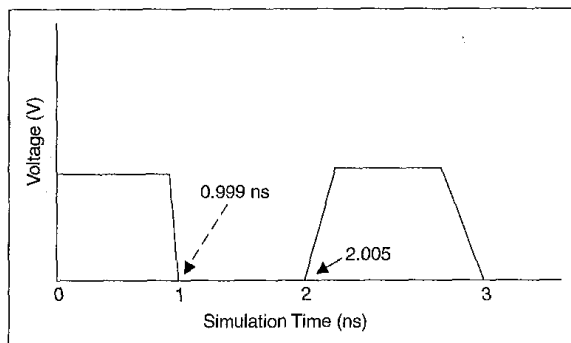
A manifestation of the macro- and micro-time scales in VHDL is the notion of delta delays. In theory, VHDL allows signal assignments with zero delays; i.e., the value is assigned to the signal in zero macro-time units but some finite, delta, micro-time units. The actual value of delta is inserted by the VHDL compiler, transparent to the user.

The first difficulty with delta delay is that the VHDL language reference manual [9] does not state how a value for the delta is selected. This is an important question since VHDL may return different results corresponding to different choices of the delta, as illustrated through Fig. 14.

Figure 14 presents a signal waveform. Assume that the value of delta is 1 ps. When the current simulation time is either 1 ns or 2 ns, VHDL safely returns the value 0 for the signal value. However, where the delta value is 5 ps, VHDL will return the value 0 corresponding to the current simulation time of 2 ns but fail to

return a definite value corresponding to the current simulation time of 1 ns. Since the signal waveform is realized at runtime (i.e., as the entities execute during simulation) and as the VHDL compiler must select a value for the delta delay at compile time, it is difficult to ensure the absence of ambiguous results.

The second difficulty is that, fundamentally, any attempt to simulate an asynchronous circuit with zero-delay components, under discrete event simulation, is likely to lead into ambiguity. In its aim to simulate digital designs with zero-delay components through delta delays, VHDL incurs the same limitation. Consider, for example, the RS latch in Fig. 8(b) and assume that both NANDs are 0 ns delay gates and that they execute concurrently in a VHDL structural description. Assume that the initial value at Q and Qb are both 1 and that the values at set and reset input ports are both 1. At simulation time 0, both gates execute and generate the following assignments: (1) a value 0 is assigned at Q at time $0 + 0 = 0$ ns and (2) a value 0 is assigned at Qb at time $0 + 0 = 0$ ns. Assume that there is a race between (1) and (2) and that (1) is executed infinitesimally earlier. As a result, the lower NAND gate is stimulated and it generates an assignment: (3) a value 1 is assigned at Qb at $0 + 0 = 0$ ns. Upon examining (2) and (3), both assignments are scheduled to affect the same port, Qb,



14. Impact of the choice of delta value on simulation results.

```

architecture X of Y is
  signal a, b: resolve BIT .. ;
begin
  PROC1: process
    variable c: int;
    begin
      for i in 0 to 1000 loop
S1:   a <= b + c;
S2:   b <= a + c;
      end loop;
    end process;

  PROC2: process
    variable d: int;
    begin
      for i in 0 to 7 loop
S3:   a <= b + d;
S4:   b <= a + d;
      end loop;
    end process;
end X;

```

15. An inconsistency with delta delays in VHDL.

at the same exact time, 0 ns, one armed with a "0" and another armed with a value "1."

The third difficulty is that one can construct any number of example scenarios in VHDL where the result is inconsistency and error. Consider the process, PROC1, shown in Fig. 15. While not critical to this discussion, it is pointed out that PROC1 does not include a sensitivity list, which is permitted by the VHDL language [9]. As an example of usage of a process without a sensitivity list, the reader is referred to page 57 of [9].

The statements S1 and S2 are both zero-delay signal assignments. While S1 updates the signal "a" using the value of signal "b" and the variable "c," the statement S2 updates the signal "b" using the value of the signal "a" and the variable "c." To prevent ambiguity of assignments to the signals "a" and "b," the VHDL compiler inserts, at compile time, a delta delay of value delta1, say, to each of S1 and S2. Thus, S1 is modified to $a <= b + c$ after delta1, and S2 is modified to $b <= a + c$ after delta1. For every iteration, the subsequent assignments to "a" and "b" are realized in increments of delta1. That is, first (NOW + delta1), then (NOW + delta1 + delta1), and so on. These are the micro-time steps in the micro-time scale and we will refer to them as delta points. Between the two consecutive macro-time steps, the VHDL scheduler may only allocate a maximum but finite number of delta points, which is a compile-time decision. Conceivably, the designer may choose a value for the number of iterations such that, eventually, the VHDL scheduler runs out of delta points. Under these circumstances, VHDL will fail. Thus, the idea of signal deltas, transparent to the user, is not able to be implemented.

Fourth, the notion of delta delay, in its current form, poses a serious inconsistency with VHDL's design philosophy of concurrency. Consider Fig. 15, where the processes PROC1 and PROC2, by definition, are concurrent with respect to one other. The two sets of statements—{S1,S2} in PROC1 and {S3,S4} in PROC2—both affect the signals "a" and "b" and "resolve" constitutes the resolution function, as required by VHDL. The statements S1, S2, S3, and S4 are all zero-delay signal assignments, so delta delays must be invoked by the VHDL compiler. Since the dynamic execution behavior of processes are unknown a priori, the VHDL compiler may face difficulty in choosing appropriate values for the delta delay in each of the processes. In this example, however, logically, the VHDL compiler is likely to assign a very small value for the delta delay (say delta1) in PROC1, given that it has to accommodate 1001 delta points. In contrast, the VHDL compiler may assign a modest value for the delta delay (say delta2 where $\text{delta2} > \text{delta1}$) in PROC2, given that only 8 delta points need to be accommodated. As stated earlier, assignments to the signals "a" and "b" will occur from within PROC1 at (NOW + delta1), (NOW + delta1 + delta1), and so on. From within PROC2, assignments to the signals "a" and "b" will occur at (NOW + delta2), (NOW + delta2 + delta2), etc. By definition, a resolution function resolves the values assigned to a signal by two or more drivers at the same instant. Therefore, here "resolve" will be invoked only when $(\text{NOW} + m \times \text{delta1}) = (\text{NOW} + n \times \text{delta2})$, for some integer values "m" and "n." In all other cases, assignments to the signals "a" and "b" will occur either from

within PROC1 or PROC2. Thus, the values of the signals "a" and "b," from the perspectives of processes PROC1 and PROC2, are uncoordinated, implying ambiguity and error.

In summary, the desire to use zero-delay assignments necessitates the inclusion of delta delays in VHDL. In turn, delta delays foster ambiguity, uncertainty, and provide a false sense of accuracy under zero-delay assignment usage. We recommend that delta delays be eliminated from VHDL and that designers first determine the universal time for a given hardware system and then utilize realistic inertial and transport delay values, as appropriate, in the VHDL description of the hardware modules. The VHDL scheduler has been carefully designed to ensure the accurate description of realistic hardware descriptions. Last, the synthesis tools must be enhanced to accept realistic VHDL descriptions.

Inadequacy of the Transport Timing Delay. The difficulty with the transport timing delay semantics is a serious one and is addressed in [13] in detail.

Limitations of Signal Attributes. The primary goal of the signal attributes is to facilitate specifying the timing assertions in hardware systems accurately, unambiguously, and easily. If one or more of the attributes assist in simplifying timing descriptions, that would be an added bonus. Timing assertions may involve examining the high or low pulsewidth of a single signal or the relative timing between two or more signals. In either case, what is required is access to the values and the assertion times of the values of the signals. VHDL does provide the value of a signal at the current simulation time. It is pointed out that a signal contains only the history of values and times from the origin to the current simulation time. What would have been ideal is for VHDL to have designed an attribute function that would accept a past time value T as an argument and yield the logical value of the signal under question corresponding to T . To a designer, such an attribute would have constituted a natural scheme to view into the history of signals.

According to the VHDL language reference manual, the operation S'STABLE(T) involves the STABLE attribute, accepts a time value T , and returns a signal whose value is either true or false, depending on whether any events have occurred on signal S in the past T time units starting at the current simulation time. In essence, S'STABLE(T) yields a single value. Therefore, to label it a signal appears unnecessary.

In essence, the S'DELAYED(T) operation, as discussed earlier, allows the designer to view into the past behavior of the signal, for an interval of T time units, starting at the current simulation time. Clearly, the synthesis of the new signal S'DELAYED(T) occurs at runtime and is very time consuming since it requires examining the history of the signal from the origin simulation time to the current simulation time and their manipulation. For the purpose of checking the representative timing assertions, the DELAYED operator appears to be redundant. As discussed earlier, the operators—STABLE(T),

While a significant body of VHDL reflects elegant design, it suffers from a number of serious flaws.

LAST_EVENT, and LAST_VALUE—are adequate. The DELAYED operator does possess a unique capability in that it allows the designer to view into a signal's history to any point in the past, up to the origin. Perry [16] proposes a technique to

verify the hold time of a D-type flip-flop, utilizing the DELAYED attribute. He delays the clock signal by hold time and utilizes it to activate a process hold time units into the future, relative to the current simulation time. The process verifies whether the value on the D input is stable for hold time units following the active clock edge. While perfectly legitimate, the solution requires the compute- and memory-intensive operation involving the DELAYED operator. In essence, the DELAYED operator allows for projecting a reference point in the future. When the simulation time has advanced in the future to the reference point, target signals, etc., may be examined. This scheme is unnatural for the following reason. If one wants to project a reference point in the future, one uses a timer in a hardware system. A sequential behavior description cannot model a timer since, by definition, a timer is concurrent. The goal, however, is correctly and easily realized by simply suspending the behavior description by executing a wait for T time units, at the end of which the description is reawakened.

A number of signal attributes—'ACTIVE, 'LAST_ACTIVE, 'QUIET(T), 'TRANSACTION(T), etc.—are included in VHDL to allow a designer to view the transactions. By definition, transactions are uncertain, for they may be pre-empted. Therefore, it appears unnatural for VHDL to include such attributes. Also, one has to be very careful when interpreting such results, since they are subject to change due to pre-emption. Since the total number of transactions may be significantly larger than the total number of events, operations such as S'QUIET(T) and S'TRANSACTION(T) are expected to be even more compute- and memory-intensive. Clearly, knowledge of the transactions may help greatly in debugging VHDL programs and in verifying the correctness of the VHDL environment. VHDL would be better served if such attributes were either invoked only under a special compilation and execution mode, say debugging, or removed from the language and placed under the VHDL simulation environment.

Characteristics of VHDL Relative to "Hierarchy"

The structural style of entity body description is ideally suited for describing a hierarchical digital design in VHDL. The description is expressed in terms of lower-level component instances, and components, in turn, are entities with the corresponding entity bodies assuming either dataflow, behavioral, or structural descriptions. Consider that in VHDL, the highest level of abstraction is expressed as an entity named *microprocessor*. The entity body of microprocessor is described in structure style that includes a declaration of the components—register, memory, ALU, control, and program counter—and their instantiations

along with the appropriate port map. Associated with each of register, memory, ALU, control, and program counter, in turn, are entity declarations and the corresponding entity bodies. The entity body of ALU is again expressed as a structure that includes a declaration of the component—adder, and its instantiation with the appropriate port map. Associated with adder is an entity declaration and the corresponding entity body is again described through a structure that includes a declaration of the components—primitive gates—and their instantiation accompanied by port maps. Associated with the primitive gates are entities. Since the gates lack further structural decomposition, the corresponding entity bodies must be expressed either as dataflow or behavior descriptions.

Limitations of Hierarchy in VHDL

Since VHDL is designed on top of Ada and since the constructs in Ada are rich enough to support selective elaboration of subsets of the hierarchy at runtime, as illustrated in [15], it is logical to expect the state-of-the-art VHDL to address the concept of zooming.

Conclusions

This article has presented a set of fundamental principles underlying HDLs and has critically analyzed VHDL, which is the leading HDL. While a significant body of VHDL reflects elegant design, it suffers from a number of serious flaws. The suggestions provided in this article for modifications to key syntactic and semantic constructs of VHDL will help ensure its continued success as the state-of-the-art HDL into the next century. HDLs are the key to understanding computer hardware and software and are involved from the conception of any new computer or hardware design to the final implementation and testing of the IC. A systematic, comprehensive, elaborate, and detailed discussion of the necessary VHDL language redesign efforts, however, is the subject of a future paper.

Sumit Ghosh currently serves as the associate chair for research and graduate programs in the Computer Science and Engineering Department at Arizona State University. His research interest is in fundamental yet practical problems from hardware description languages and a number of other disciplines within computer science and engineering.

Norbert Giambiasi has been a Professor at the University of Aix-Marseille since 1981. He is the author of a book on CAD and over 150 international publications. He has been the scientific manager of more than 50 research contracts with E.S Dassault, Thomson-Cimsa, Bull, Siemens, Cnet, Usinor, and others. His current interests are in the formal specification of hybrid simulation models, distributed simulation, discrete event simulation of hybrid systems, CAD systems, and design automation.

References

1. P.W. Case, H.H. Graff, and M. Klopmok, "The recording, checking, and printing of logic diagrams," *Proc. Eastern Joint Computer Conference*, 1958, pp. 108-118.
2. C.G. Bell and A. Newell, "The PMS and ISP descriptive systems for computer structures," *Proc. AFIPS Conference*, SJCC, Reston, Virginia, 1970, vol. 36, pp. 351-374.
3. T.W. Pratt, *Programming Languages Design and Implementation*, Englewood Cliffs, NJ: Prentice-Hall, 1984.
4. William Morris (ed.), *The American Heritage Dictionary of the English Language*. Boston, MA: Houghton Mifflin Company, 1981.
5. E. Debenedictis, S. Ghosh, and M.-L. Yu, "An asynchronous distributed discrete event simulation algorithm for cyclic circuits using data-flow network," *IEEE Computer*, vol. 24, no. 6, pp. 21-33, June 1991.
6. M.R. Barbacci, "A comparison of register transfer languages for describing computers and digital systems," *IEEE Trans. Comp.*, vol. C-24, no. 2, pp. 137-150, Feb. 1975.
7. G.M. Baudet, M. Cutler, M. Davio, A.M. Peskin, and F.J. Rammig, "The relationship between HDLs and programming languages," *VLSI and Software Engineering Workshop*, June 1982, Port Chester, NY, pp. 64-69.
8. R. Piloty and D. Borriore, "The Conlan Project: Concepts, implementations, and applications," *IEEE Computer*, vol. C-24, no. 2, pp. 81-92, February 1985.
9. The Institute of Electrical and Electronic Engineers, IEEE Standard VHDL Language Reference Manual, ANSI/IEEE Std 1076-1993, New York: IEEE, April 14, 1994.
10. M. Shahdad, R. Lipsett, E. Marschner, K. Sheehan, and H. Cohen, "VHSIC hardware description language," *IEEE Computer*, vol. C-24, no. 2, pp. 94-103, Feb. 1985.
11. The Engineering Staff of TI Inc., *The TTL Databook for Design Engineers*, Texas Instruments Incorporated, Dallas, Texas, 1976.
12. D. Hill, Language and Environment for Multi-level Simulation, Technical Report 185, Computer Systems Lab., Stanford University, 1980.
13. P.A. Walker and S. Ghosh, "On the Nature and Inadequacies of Transport Timing Delay Constructs in VHDL," *IEEE Trans. Computer-Aided Design*, vol. 16, no. 8, pp. 894-915, Aug. 1997.
14. D. Hill, Adlib Users Manual, Technical Report 177, Computer Systems Lab., Stanford University, 1979.
15. S. Ghosh, "Ada as a hardware description language," *IEEE Design and Test of Computers*, vol. 5, no. 1, pp. 30-42, Feb. 1988.
16. D.L. Perry, *VHDL*. New York: McGraw Hill, Inc., 1993.
17. S. Mazor and P. Longstraat, *A Guide to VHDL*. Boston: Kluwer Publishers, 1993.
18. Department of the U.S. Air Force, Draft Request for Proposal F33615-83-R-1003, VHSIC Hardware Description Language (VHDL), Sources Sought Synopsis PMRE 82-116, September 9, 1982.
19. R. Goering, "VHDL shared variables create dissent — Spec divides EDA," *EETimes*, No. 937, January 20, 1997.
20. P.A. Walker and S. Ghosh, "Asynchronous, distributed event driven simulation algorithm for accurate execution of vhdl descriptions on parallel processors," *Proc. 32nd IEEE/ACM Design Automation Conference*, June 12-16, 1995, San Francisco, CA.
21. US Department of Defense, Reference Manual for the Ada Programming Language, Government Printing Office, Washington, DC 20402, 1980.
22. D. Hill, "Beginnings of ADLIB/SABLE," *IEEE Design and Test of Computers*, pp. 58-60, September 1992.

CD ■