Shared Variables in VHDL 1993:
A Peek Into the Past and a Preview of the Future

Stephen A. Bailey
Vantage Analysis Systems
sbailey@viewlogic.com

John C. Willis
Systems Technology & Architecture
IBM Corporation
willis@vnet.ibm.com

1. Abstract

During the balloting of VHDL 1076-1993¹, one new language feature accounted for a significant portion of ballot comments and negative votes. That feature was shared variables. As a result of general dissatisfaction with the 1076-1992/A [VHDL92A] language implementation and the inability of the language design team to agree on a resolution, 1076-1992/B [VHDL92B] removed all mutual exclusion semantics, thereby creating something more appropriately termed global variables. Once again, shared variables dominated the issues raised in the balloting of 1076-1992/B. However, the VHDL Analysis and Standardization Group (VASG) anticipated the dissatisfaction and promised the balloters that should the B draft of the language pass ballot, a working group would be formed and chartered with the sole task of resolving the shared variable implementation in VHDL. The Shared Variables Working Group was formed in February 1992. This paper will address the working group’s progress to date, provide an historical context, review requirements for shared variables and preview the proposed shared variable language revision.

2. A Short History of Shared Variables

This section enumerates some of the original requirements for shared variables from the 1076-1992 language revision process, the implementation of shared variables in 1076-1992/A language definition and the reasons for removal of that implementation of shared variables from the 1076-1993 language definition.

2.1. The Original Requirements for Shared Variables

The original requirements for shared variables reflected diverse needs. Among the original requirements [VHDL92REQ] were:

- A need to share data between processes without the need for explicit modeling of the access to the shared data. Essentially, shared variables were a modeling convenience in which the shared variables are not part of the physical hardware. (e.g. model statistics and trace gathering.)
- A need to communicate dynamically sized data between processes efficiently such as queues, stacks, etc. VHDL signals cannot have an access type and unconstrained ports become fixed-size at elaboration time.
- A need to translate from other languages, such as Verilog and C, which have global variables.
- A need to model stochastic processes. For example, stochastic processes are often useful when simulating the environment that the hardware is to operate in. (e.g.

---

¹ At the time, the proposed standard was 1076-1992. The official standard identifier changed to reflect the year that the standard was officially approved.
pseudo-random number generators, queuing models)

2.2. What went wrong in 1076-1992/A
In the domain of Communicating Sequential Processes (CSP) languages, there are 3 well-known approaches to providing mutual exclusion for shared resources. In order to fully appreciate the problems with the 1076-1992/A language definition, a review of these three approaches appears in Table 1.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Pros</th>
<th>Cons</th>
<th>Analogy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Semaphores</td>
<td>⇒ Simple language design</td>
<td>⇒ Easy to miss the release of resources</td>
<td>Roughly analogous to goto's including non-local goto's.</td>
</tr>
<tr>
<td></td>
<td>⇒ Very flexible to use</td>
<td>⇒ Deadlock very difficult to avoid</td>
<td>Shared resource may be locked in scope/region and released in another. No relationship between lock and release is required.</td>
</tr>
<tr>
<td>Critical Regions</td>
<td>⇒ Ensure all shared resources are released</td>
<td>⇒ Statement-level distributes knowledge of access points throughout code</td>
<td>Equivalent to a functional or imperative language solution.</td>
</tr>
<tr>
<td></td>
<td>⇒ Can be deadlock free</td>
<td>⇒ Information hiding may be difficult</td>
<td>No relationship between shared resource and the operations on it are defined in the language. Users must explicitly program lock and release with full understanding.</td>
</tr>
<tr>
<td>Monitors</td>
<td>⇒ Group shared resources with their operations</td>
<td>⇒ User of shared resource may not have the flexibility of use desired -- only what the definer of the shared resource offers.</td>
<td>The &quot;object-oriented&quot; approach. Monitors relate the operations on shared resources with those resources. Maximizes information hiding and encapsulation. Frees users from the need to explicitly lock and release (including understanding all the implications thereof).</td>
</tr>
<tr>
<td></td>
<td>⇒ Critical regions are implicit; user of shared resource need not program critical region</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>⇒ Can also be deadlock free</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1. Approaches to Mutual Exclusion in Language Design

2.3. 1076-1992/A: Critical Regions
The first version of VHDL 1992 provided shared variables with mutual exclusion semantics through a form of critical regions called access statements. The following sections describe this initial implementation and some of the problems associated with it.

2.3.1. Access Statement
In VHDL 1992/A, the access statement defined a critical region of code for one or more shared variables. Some of the more important characteristics of the access statement were:
♦ A shared variable access list, analogous to a process sensitivity list, declared which shared variables needed to be locked before execution of the critical region could proceed.
♦ Limitations were placed on the types of statements that could be executed in a critical region. For example, wait statements were not permitted in critical regions.
♦ To prevent deadlock situations, critical regions could not be accessed from within another critical region.
♦ Because the access statement did not resolve the issue of more than one process containing an access-typed variable referencing the same shared data (alias), shared variables could not be access-typed.

Figure 1 uses a fragment of code to demonstrate 1076-1992/A access statements. As a result of
the mutual exclusion provided by access statements, shared variable SV1 will always have a value of -2, 0, or +2. The addition operation (first access statement) or the subtraction operation (second access statement) will always execute atomically.

The purpose of mutual exclusion semantics is to ensure the integrity of shared data. Without the mutual exclusion assured by the access statement, the SV1 in P1 may be consistently referenced on the right hand side, then the subtraction statement in P2 might execute completely, then the increment and store phase of the sequential statement in process P1 might complete, assigning a sequence of ascending, even values to SV1. With more complex data structures than a single variable and more complex, composite types, even more bizarre results are possible in the absence of mutual exclusion.

```
ARCHITECTURE foo OF e IS

  SHARED VARIABLE SV1 : Natural := 0;

BEGIN

  P1: PROCESS
     BEGIN
       ACCESS (SV1)
       SV1 := SV1 + 2;
     END ACCESS;
     WAIT ON Sig1;
     END PROCESS P1;

  P2: PROCESS
     BEGIN
       ACCESS (SV1)
       SV1 := SV1 - 2;
     END ACCESS;
     WAIT ON Sig1;
     END PROCESS P2;

END foo;
```

Figure 1. Example use of Access Statements

2.3.2. Problems with Access Statements

Balloting of 1076-1992/A raised issues concerning access statements ranging from concerns that the access statement failed to satisfy the original requirements to concerns that the language definition was ambiguous. Issues raised include:

- Problems, primarily ambiguities, surrounding the shared variable access list:
  - If the shared variable access list contains an indexed name, sliced name or record element, what constitutes the longest static prefix?
  - Does the language specify the order in which the access list's shared variables are locked?
  - Restrictions against nesting of access statements (to prevent deadlock) violated principles of information hiding, making access statements difficult to use within subprograms.
  - Restrictions against access-typed shared variables left a key requirement for dynamically sized data structures unsatisfied.

2.4. Shared Variable Revision 1076-1992/B

The ballot resolution team looked at the problems and issues with the A draft's implementation of shared variables and, due to time constraints and the inability to agree on a better solution, decided to provide the simplest resolution possible, global variables without any form of mutual exclusion.

Because members of the language design team neither had the time to flesh out a new language implementation nor would user pressures permit omission from the language, they chose a solution that many hated and few liked. Since all the issues with the A draft concerning shared variables involved mutual exclusion semantics, the solution was to eliminate all mutual exclusion semantics from the language. This solution required implementors to define their own mutual exclusion semantics for multi-threaded simulators and parallel simulators. In the absence of language-mandated mechanisms for mutual-exclusion, users were likely to find that VHDL models using shared variables were at best not portable between simulators or even versions of the same simulator.

The result of balloting the 1076-1992/B draft was predictable. Shared variables were once again the center of attention, primarily from those who understood the long-term ramifications of a parallel language design with
shared variables but completely lacking any mechanism for mutual exclusion. Mutual exclusion is primarily a concern for multi-threaded simulators, parallel simulators, synthesis systems and formal verification systems. Such tools with support for shared variables were apparently not widely available to the 1076-1992/B language designers or ballot group, hence the problem was initially not widely appreciated.

However, since the result of the balloting was predicted, and no one wanted the 1992 standardization held up indefinitely for a complete fix to the shared variable problem, the B draft was distributed to balloters with a cover letter promising the formation of a working group to address this single language issue. Thus was born the Shared Variables Working Group.

3. Shared Variable Revision
This section covers the progress of the Shared Variable Working Group. First, the results of a comprehensive requirements gathering and prioritization effort are presented. This is followed by the current proposal for revising shared variables in VHDL -- complete with an example.

3.1. Prioritized Requirements for Shared Variables
This section summarizes the requirements and priorities agreed upon by vote of the shared variable working group. The full text of the requirements document is contained in [SVWG194] and the complete prioritization vote results can be found in [SVWG294].

Identification and prioritization of the language design requirements was an extremely important undertaking since there has always been disagreement over what are the important design issues that shared variables are intended to help solve. This activity also helped greatly in providing focus to the language design team. The following summarizes the requirements that the implementation of shared variables should satisfy:

- Permit shared variable declarations within packages, entities, architectures and blocks.
- Make the characteristic of “shared” manifest in the declaration of the shared variable. (Not needed by language since shared variables can only be declared in a “concurrent” context.)
- Provide algorithmic determinism. VHDL ’87, with a very few exceptions, is a strictly deterministic language. The ’93 implementation of shared variables makes VHDL an entirely non-deterministic language (when shared variables are used). If shared variables provided algorithmic determinacy, then it would be possible for different simulation runs (possibly on different simulators or platforms) to get different results, but each set of results might be correct given the design’s algorithm.
- Support for dynamically-sized data that is shared between processes was re-affirmed following the requirement from the initial 1076-92 requirements process. This permits structures such as queues, stacks and trees that can grow and shrink as required during the execution of the model.
- Although much discussion has taken place as to whether the working group’s efforts should define a replacement for 1076-1993’s shared variables or complement them, the working group emphatically voted to maintain the original charter to replace the 1993 implementation of shared variables.
- The implementation should not arbitrarily restrict the inherent parallelism of VHDL.

Since the working group voted on the priorities of requirements by specifying a value in the range of -3 to +3, it was possible for a requirement to be voted as being dangerous or undesired. The language design team was instructed to avoid satisfying requirements that received a consensus negative priority vote. The requirements in this category include:

- Provide access-typed shared variables. Although dynamically-sized shared variables are desired, the working group realized that this requirement could be met without allowing multiple processes to maintain local pointers to shared data.
- Require strict determinism or allow unbounded non-determinism.
 Placement of arbitrary read/write restrictions on shared variables within a delta cycle (one write, infinite reads as long as reads occur first; no reads if any writes, etc.).

 Remove shared variables from the language.

 3.2. How are Shared Variables to be Revised to meet these requirements?
 This section first discusses the general approach and its overall benefits for meeting the requirements. Details of the current proposed implementation appear, concluding with an example.

 3.2.1. Shared Variables Implemented with Monitors for Mutual Exclusion
 Monitors were chosen as the basis for the implementation of shared variable mutual exclusion semantics. This choice is based upon monitor's superior ability to support information hiding and encapsulation. Specifically:

 - Monitors fit nicely with VHDL's current visibility rules and block model. They can also be implemented to exist comfortably with VHDL's typing system.
 - Monitors group shared data with the operations on that data.
 - Monitors provide algorithmic determinism by associating legal operations with shared resources and by providing an implicit lock and release mutual exclusion semantics based on entry to and exit from the monitor operations. It is impossible to modify shared data outside of the monitor operations.
 - Because of their superior ability to encapsulate the shared data, monitors support dynamically-sized data. The access-typed variables essential for providing dynamically-sized data with VHDL are completely encapsulated within the monitor and can only be accessed by the monitor operations. The pointer values cannot be transferred to local processes through the monitor subprogram interfaces. Of course, the implicit mutual exclusion semantics will ensure the integrity of access values within the monitor.

 It should be obvious that a monitor-based implementation of shared variables is not upward compatible with the existing 1993 implementation. Therefore, the approach is consistent with the requirement for replacing the existing implementation.

 Monitors do not arbitrarily restrict the inherent parallelism in the language or of designs modeled in VHDL. Sequential execution is limited only to the scope of the execution of a monitor operation. Monitor operations are defined by the user. Therefore, the user has complete control of the critical region.

 Monitors do not require strict determinism. The language implementation can leave the order of process access to shared resources via the monitor undefined. It only needs to guarantee that one and only one process gains access (if the access can change the state of the shared resource) to the shared resource at the same time.

 As enumerated above, monitors provide an excellent technical approach for meeting the requirements as they were prioritized by the working group.

 3.2.2 Proposed Language Design
 Monitors are introduced as part of the type system. First a monitor type must be defined. Then instances of (shared) variables using the monitor type may be created. Finally operators defined by the monitor type may be applied to the shared variable.

 type_definition ::=  
   scalar_type_definition  
   | composite_type_definition  
   | access_type_definition  
   | monitor_type_spec_definition  
   | monitor_type_body_definition

 Definition of a monitor type requires a monitor type specification of the form:

   monitor_type_spec_definition ::=  
     MONITOR  
     monitor_spec_declaration_list  
     END MONITOR

 8.21
The monitor specification may contain element declarations, subprogram specifications and subprogram bodies:

\[
\text{monitor_spec_declaration_list ::= element_declaration ''| subprogram_specification ''| subprogram_body''}
\]

The element declarations are only visible within the subprograms specified within the monitor type, and then only by selection (more on this later). The monitor subprograms define the abstract operators on the monitor type. They may include overloaded operators, explicit functions and explicit procedures. Subprogram bodies appearing in the monitor specification are intended to be relatively brief, however bodies for any of the monitor's subprogram specifications may appear in the monitor specification.

Monitor type specifications may appear in:
- packages declarative parts
- package body declarative parts
- entity declarative parts
- architecture declarative parts
- block declarative parts
- generate declarative parts

The ability to place (monitor) subprogram bodies within packages and entities is not strictly consistent with VHDL-1076-93. Ideally, the monitor specification would only contain the monitor's interface (subprogram specifications for the monitor's operators). From a practical standpoint, many contemporary VHDL compilers generate code for each design unit in isolation (separate compilation), preventing the inlining of the monitor subprogram bodies found in a package body. To avoid the need for such compilers to experience the overhead of actual calls to potentially small monitor subprogram bodies, the current proposal permits subprogram bodies within the monitor specification and thus package specification or entity.

Similarly, moving element declarations to the monitor specification allows a compiler using separate compilation to determine the size of an object of the monitor type based only on the specification, potentially generating more efficient code.

The (optional) monitor type body provides for the introduction of additional subprogram specifications and subprogram bodies. Subprogram specifications first appearing in the monitor body are only visible within other subprograms belonging to the monitor type and subsequent to the (private) subprogram specification.

\[
\text{monitor_type_body_definition ::= MONITOR BODY ''| subprogram_specification'' | subprogram_body'' | subprogram_body_declaration_list'' | END MONITOR BODY''}
\]

The monitor body may contain subprogram specifications and subprogram bodies:

- package body declarative parts
- architecture declarative parts
- block declarative parts
- generate declarative parts

Each monitor type body must have a corresponding and preceding monitor type specification.

Monitor type specification and bodies may only appear as part of a monitor type declaration, for example:

```
\begin{verbatim}
TYPE tree_type IS
  MONITOR
  ...
  END MONITOR tree_type;
\end{verbatim}
```

possibly followed by:

```
\begin{verbatim}
TYPE tree_type IS
  MONITOR BODY
  ...
  END MONITOR tree_type;
\end{verbatim}
```

There are several restrictions on the monitor type’s (abstract) operators, represented by subprogram specifications declared within the monitor type specification.

Foremost, a monitor subprogram must have exclusive access to all shared variables of monitor type referenced in the caller's association list (actual parameters of monitor type), thus assuring exclusive access to all visible element declarations. An implicit
sequence of blocking P (pass) operators prefaces each monitor subprogram and a corresponding sequence of V (from the Dutch word for free) operators prefaces each return from monitor subprograms. The order in which a monitor subprogram attempts to gain access to shared variables and frees shared variables is not defined by the language.

Interface declaration items or the return type associated with subprograms appearing in the monitor type specification may not be or contain an access type. This preserves the encapsulation of monitor data representations.

Subprogram bodies contained in the monitor type specification or body may not contain a wait statement, either directly or by call. In general, a compiler cannot detect that a wait statement may be encountered while executing a monitor subprogram. For example, the monitor subprogram may call a procedure located in another package which in turn may conditionally call other procedures. Thus this behavior is an error that may need to be detected during execution. The intent of this restriction is to reduce opportunities for deadlock.

Elements declared within the monitor type specification are only visible, by selection, within subprograms declared by the monitor type specification or monitor type body. Thus at most one type of monitor may appear in the interface declaration of any monitor subprogram and that type must be the associated monitor type. This provides monadic, dyadic, triadic and larger abstract operators as long as all shared variable interface objects are of the same monitor type. This restriction (all shared variable parameters to a monitor operation must be of the same monitor type) enables optimizations in multi-threaded and parallel simulator implementations, but at the expense of some more powerful constructs involving multiple monitor types.

Type marks denoting a monitor type may only appear in:
- shared variable declarations
- shared variable interface declarations (subprograms)
- elements of array type definitions
- elements of record type definitions

Specifically, type marks denoting a monitor type may not appear within an access type.

The VHDL 1076-93 keyword \texttt{SHARED} must preface all shared variable and shared variable interface declarations for documentation clarity and internal consistency checking. All shared variable and interface declarations must be of a monitor type and only shared variables may be monitor-typed. Shared variable interface declarations may only appear in subprogram interfaces. All actuals associated with such interface declarations must be shared variables and are passed in and out (according to the designated mode) by reference, not value.

For example, a shared variable object declaration might be:

\begin{verbatim}
SHARED VARIABLE database: TreeT;
\end{verbatim}

and an interface declaration might be:

\begin{verbatim}
PROCEDURE insert ( 
  SHARED VARIABLE db: INOUT TreeT; 
  VARIABLE element: DBRecord );
\end{verbatim}

Shared variable declarations may appear in:
- Package specification declarative part
- Package body declarative parts
- Entity declarative parts
- Architecture declarative parts
- Block declarative parts

3.2.3. Example Usage

Skeleton code from a database application will help to illustrate how monitors work.

A monitor type called \texttt{TreeT} is initially declared, perhaps in a package:

\begin{verbatim}
TYPE TreeT IS MONITOR 
  VARIABLE root : AccessToElement;
  VARIABLE size : Integer;

  PROCEDURE add ( 
    SHARED VARIABLE t : INOUT TreeT;
    VARIABLE e : IN ElementT;
    VARIABLE r : OUT boolean);

  FUNCTION locate ( 
    SHARED VARIABLE t : IN TreeT;
    VARIABLE k : IN KeyT 
  ) RETURN ElementT;

  PROCEDURE merge ( 
    SHARED VARIABLE t1,
\end{verbatim}
4.0 Conclusions

4.1. Current Working Group Status

The technical writer (John Willis) is iterating on a technical proposal (language change specification). The technical proposal is being circulated for review and comment by the technical committee (Victor Berman, Jacques Roulliard, Chuck Swart, Vijay Vaidyanathan, John Willis, Paul Menchini and Steve Bailey). When the technical committee is satisfied by the LCS, the proposal will be circulated to the shared variable working group. Once consensus has been achieved that the proposal is technically solid and can be incorporated into the 1993 LRM, the design will be handed off to the technical editor (Paul Menchini). The technical editor will draft a modified LRM that will be distributed for ballot later this year.

4.2. Should I use Shared Variables Today? If so How?

The use of shared variables as currently defined in 1076-1993, should not be done without carefully considering the consequences. Things to be considered when using the current implementation of shared variables include:

♦ Unpleasant surprises await those who use 1993 shared variables without problems on one vendor's simulator and port their models to another simulator! VHDL does not define the order of process execution. If the model depends on this ordering (more likely to happen with the current shared variable implementation), different results are likely even on strictly single-threaded simulators.

♦ Multi-threaded and parallel simulator vendors will need to define their own level of mutual exclusion. Once again, this will be a likely area of inadvertent model dependency on a single simulation vendor's product.

♦ If a multi-threaded or parallel simulator vendor does not provide mutual exclusion semantics, you will have no guarantee to
the integrity of the data your shared variables reference.

Even with these very real problems in the 1076-1993 implementation of shared variables, we understand that many users will be enticed to rely on them due to the fact that they will not be able to model some things without them. We would like to caution that users who decide to exploit the 1993 version of shared variables do so carefully. Following the advice below will lead to more portable models and reduce the effort of moving to the revised implementation of shared variables.

- Practice your own encapsulation and information hiding by declaring your shared variables in package bodies and defining subprograms in the package specification that manipulate the shared data directly. Ensure that processes only access shared data indirectly through your own "monitors."

- If the order of access to shared data is important, use signals in combination with shared variables to guarantee that order. (This modeling guideline will apply for the revised implementation as well.)

- If your simulation vendor offers a multi-threaded simulator, parallel simulator or you plan to use one in the future, ask that vendor how they implement mutual exclusion. The vendor's implementation will become an implicit part of your model's behavior and you must take this into account if you ever port to a different simulator with a different mutual exclusion model.

- Limit your use of shared variables to those pieces of the design that will not have a physical manifestation in your manufactured product. For example, use them to model the test bench or operational environment or to collect useful model statistics that will help you in debugging or improving your designs.

4.3. When will the Amended LRM become a Standard?

It is always difficult to predict such things with a high level of accuracy due to the dependency on volunteer help and the naturally optimistic tendency engineers and computer scientists have for schedules.

However, the technical team's best estimate is that the revision will be available during the 1st half of 1995. We plan to go to ballot by the end of this year with another quarter or two needed to respond to ballot comments and proceed through the IEEE's standardization process. If substantial flaws are found during the ballot process, subsequent ballots may be needed.

Bibliography


References

Monitors:


CSP:


C.A.R. Hoare, “Communicating Sequential Processes,” in Concurrent Programming,
Edited by Narain Gehani and Andrew D. McGettrick, Addison-Wesley, 1988

P and V Operators: