Submitted for publication in *IEEE Design and Test of Computers*.

Protected Shared Variables in VHDL: IEEE Std 1076a

Peter J. Ashenden
Department of Computer Science
The University of Adelaide
Adelaide, SA 5005
Australia
peter.ashenden@computer.org

Philip A. Wilsey
Dept. ECECS, PO Box 210030
The University of Cincinnati
Cincinnati, OH 45221-0030
USA
phil.wilsey@uc.edu

Abstract

The VHDL Standard current allows concurrent access to variables shared between processes, but does not define any semantics for concurrency control. The IEEE 1076a Shared Variables Working Group has developed a form of monitors, called protected types, to provide mutually exclusive access to shared variables. This article identifies the problems that can arise from unprotected concurrent access to shared variables, and reviews the idea of monitors, which forms the basis of the proposed language change. It then describes protected types, gives some guidelines on using them for hardware modeling, and includes an example to illustrate their use.

Index terms: VHDL, shared variables, protected types, monitors

VHDL is a standard hardware description language for modeling the behavior and structure of digital systems. A system is modeled as a hierarchy of component instances, interconnected with signals. Within the primitive components of the hierarchy, behavior is expressed using processes. A process encapsulates a body of sequential statements together with some state information represented using local variables. Processes communicate with one another using the interconnecting signals.

The original VHDL standard did not allow sharing of variables between process instances, despite this being requested in the preliminary language design phase. During the restandardization process started in 1990 (see sidebar, *Standardizing VHDL*), users identified the lack of shared variables as a deficiency in the languages. Hence, the VASG added shared variables as a requirement for the revised language standard.

The VASG subcommittee in charge of language design developed a mechanism called *access statements* and included it in the draft standard [6]. However, there were sufficient negative ballots that the VASG resolved to remove access statements. They decided to allow shared variables in the revised language, but not to specify what happens if multiple processes concurrently access a shared variable. Further, they set up a new Working Group to resolve the issue. That group has completed its work, and the language changes are about to be balloted.

In this article, we show a variety of problems that can arise from unprotected concurrent access to shared variables, and review the idea of *monitors*, which forms the basis of the proposed language change [7]. We then describe the new language feature, *protected types*, and give some guidelines on using them for hardware modeling. We also include an example to illustrate their use.

Shared Variables in VHDL-93

VHDL models that we write are elaborated down to a collection of processes interconnected by nets of signals. Normally, the only way for one process to communicate with another is through the signals. However, when we are modeling at a high level of abstraction, there are cases when communication by signals may be cumbersome. If two or more processes simply need to share data, we could express the communication using a variable that is shared by the processes. Each process could assign new values to the variable, and the changes would be seen by other processes reading the variable.

VHDL-87 did not allow variables to be shared in this way. VHDL-93 does allow shared variables, provided they are declared to be shared, as the following example illustrates:

```
shared variable counter : natural := 0;
```

Using the keyword **shared** documents the fact that there may be several processes with direct access to the variable. Those processes may read and write to the shared variable without restriction. However, the standard does not define the result of two processes concurrently accessing the shared variable.

The scope and visibilty rules of the language determine which processes can access a variable. VHDL's rules are similar to those of programming languages such as Pascal and Ada. A variable is declared in a declarative part of a construct, such as an architecture body, a process or a subprogram. The scope of the variable extends from the declaration to the end of the construct, including any nested constructs. Within its scope, a variable is visible unless hidden by a declaration with the same name as the variable inside a nested construct.

The implication of these rules is that, if we declare a variable within a process, it is only accessible within the process. Hence, it cannot be shared. If we declare a variable in an architecture body or block statement, it is accessible to all processes within the architecture body or block statement. Hence, it must be declared as a shared variable. Furthermore, if we declare a variable in a package declaration, it is accessible to any process in the model, and so must be declared as a shared variable.

Problems with Unprotected Shared Variables

Problems with shared variables can arise when the processes in the model are executed on a parallel computer, such as a mutiprocessor workstation or a parallel supercomputer. They can also occur on a single-processor computer if the simulation kernel preemptively switches between processes. The simplest problem is that two processes trying to update a shared variable might interfere with each other, resulting in an unpredictable final result for the variable. Suppose, for example, that two processes try to increment the above counter. Each process executes the statement

```
counter := counter + 1;
```

This statement typically involves reading the variable's value, adding one to it, then storing the result back into the variable's memory location. If one process completes this sequence before the other starts, the variable is incremented by two, as expected. However, if both processes read the initial value before either performs the write, the variable is only incremented by one. Since the model writer has no control over the interleaving of memory access from multiple processes, the result is non-deterministic.

More complex problems arise when the shared variable is not simply a scalar, but is instead a composite or dynamic data structure requiring complex update operations. In these cases, interference between processes can put the data structure in an inconsistent state, resulting in lost or corrupted data.

To date, model writers using shared variables have not had to deal with these problems, since the only simulators that implement shared variables do not run on parallel computers. However, the potential performance gains from parallel processing are sufficient to motivate development of parallel simulation tools. Parallel simulation has been an active research area for some time [9], and at least one vendor has announced a commercial parallel simulation tool. There is no guarantee that models that give desired results on today's sequential simulators will work on the forthcoming parallel simulators.

Mutual Exclusion and Monitors

The key to avoiding interference between processes concurrently accessing a shared variable is *mutual exclusion*. A process must acquire mutually exclusive access when it needs to read or update the variable using some sequence of instructions. While the process is performing those instructions, no other process is allowed access to the variable. This rule is enforced by the language implementation.

A programming language that supports mutual exclusion must provide a way of expressing which sequences of statements are to be performed with mutual exclusion. Such sequences are often called *critical regions*. Much of the research into concurrent programming languages has been centered on this issue. (See Gehani and McGettrick [4] for a survey.)

The simplest approach is to associate a lock with each shared variable. Before entering a critical section, a process must wait for the lock to be released, then set the lock. Testing the lock and setting it must be performed as an indivisible operation, and usually relies on hardware support from the host machine. After completing the critical section, the process releases the lock. The main problem with this approach is that a programmer may forget to release the lock, causing the shared variable to become inaccessible. Furthermore, the separation of the lock and release operations makes programs hard to understand and maintain.

Programming language researchers in the 1970s developed several more abstract forms of mutual exclusion mechanisms. One of these, developed by Hoare [5], was subsequently incorporated into a number of experimental programming languages. The idea is to encapsulate a shared variable so that it is not directly accessible to processes wishing to use it. The shared variable is encapsulated in a *monitor*, and processes must invoke procedures called *monitor operations* to access or update the shared variable. By definition, the monitor operations form critical regions. Only one process is allowed to execute operations of a given monitor at a time. While one process executes an operation, other processes that invoke any operation of the monitor are blocked. When the first process exits the operation (or suspends within it) one other process is allowed to continue.

The advantage of basic monitors is that they make concurrent programs using shared variables easier to understand and maintain. However, the idea is complicated by the need to allow processes to wait within monitor operations. The semantics that determine which processes then resume and in what order are complicated. Furthermore, if monitors are nested within monitors, the semantics become even more convoluted. For these reasons, monitors in their general form have not gained favor as a general purpose concurrent programming paradigm [1]. Languages based on remote procedure call and on message passing libraries have become more widely used. Despite these facts, some recent languages (e.g., Ada-95 [8]) still use simplified forms of monitors for protecting access to shared variables.

Protected Types

The IEEE Shared Variables Working Group decided to follow the approach of Ada-95, and specified a limited form of monitors, called *protected types*. The VHDL approach is somewhat simpler that that of Ada-95, since it simply addresses mutual exclusion, not communication between processes through shared variables.

In 1076a VHDL, we declare a protected type with a *protected type definition* in a type declaration. The syntax rule is:

```
protected_type_definition ::=
      protected_type_declaration
    | protected_type_body
```

The protected type declaration specifies the interface of the protected type. It contain *methods*, subprograms that will be used by processes to access the shared variable with mutual exclusion. The syntax rules are:

protected_type_declaration ::=
  protected
    protected_type_declarative_part
  end protected [*protected_type*_simple_name]

protected_type_declarative_part ::=
  { protected_type_declarative_item }

protected_type_declarative_item :=
   subprogram_declaration
  | attribute_specification
  | use_clause

Note that these rules introduce a new reserved word, "**protected**," into the language. A simple example of a protected type declaration is:

```
type shared_counter is protected
    procedure reset;
    procedure increment ( by : integer := 1 );
    impure function value return integer;
end protected shared_counter;
```

This declares a type for a shared counter with methods (i) to reset the counter value to zero, (ii) to increment the counter by some amount, and (iii) to read the value of the counter. We can declare a shared variable to be of this type using a shared variable declaration, for example:

```
shared variable event_counter : shared_counter;
```

Processes use the name of the shared variable as a prefix to a method name to identify the shared variable on which the method is invoked. For example:

```
event_counter.reset;
event_counter.increment (2);
assert event_counter.value > 0;
```

In each case, the process acquires mutually exclusive access to event_counter before executing the body of the method. While the process is executing the method, other processes that try to invoke any method on the same shared variable must wait. When the first process finishes executing its method, it releases mutually exclusive access to the shared variable. One of the waiting processes may then resume. The language definition makes no statement about the order in which waiting processes are chosen for resumption. An implementation may, for example, use first-in, first-out queuing, but that is not required.

A protected type body specifies the implementation details of a protected type. The syntax rules are:

protected_type_declaration ::=
    protected body
        protected_type_body_declarative_part
    end protected body [*protected_type*_simple_name]

protected_type_body_declarative_part ::=
    { protected_type_body_declarative_item }

protected_type_body_declarative_item :=
      subprogram_declaration
    | subprogram_body
    | type_declaration
    | subtype_declaration

| constant_declaration

| variable_declaration

| file_declaration

| alias_declaration

| attribute_declaration

| attribute_specification

| use_clause

| group_template_declaration

| group_declaration

Note that we can include variable declarations in a protected type body. These variables constitute the data stored in a shared variable of the protected type. We must also include subprogram bodies for the methods declared in the protected type declaration. Items declared within a protected type body are not visible outside the protected type, so the only way a process can access the items is by using the methods of the protected type.

A possible implementation of the shared counter protected type is:

```
type shared_counter is protected body

    variable count : integer := 0;

    procedure reset is
    begin
        count := 0;
    end procedure reset;

    procedure increment ( by : integer := 1 ) is
    begin
        count := count + by;
    end procedure increment;

    impure function value return integer is
    begin
        return count;
    end function value;

end protected body shared_counter;
```

The specification of protected types in VHDL includes a number of rules governing their use. In summary, they are:

- A protected type declaration and body together form a declarative region, but only method names in the protected type declaration are visible outside the protected type.

- If a protected type is declared in a package declaration, the protected type body must be declared in the corresponding package body. In other cases, the protected type body must be declared in the same declarative region as the protected type declaration.

- Only variables and variable-class subprogram parameters can be of protected types. Actual values of protected type subprogram parameters are passed by reference.

- Shared variables must be of protected types; other variables may be of protected types.

- Protected types cannot be used as elements of files, as elements of composite types, nor as the designated types of access types.

- Variable assignment is not allowed for protected type variables. As a consequence, a protected type variable must not have an initial value expression in its declaration.

- The equality ("=") and inequality ("/=") operators are not predefined for protected types.

- A protected type method must not include or execute a wait statement.

- Declarations in a protected type declaration are elaborated when the protected type declaration is elaborated. Declarations within a protected type body are elaborated once for each variable of the protected type, at the time of elaboration of the variable.

Guidelines for Use

There are two modeling requirements that lead to the inclusion of shared variables in VHDL. The first is a requirement in behavioral modeling to describe passive shared objects that are accessed concurrently by different parts of a design. An example of such an object is a register file in a pipelined CPU, represented simply as an array of bit-vectors. The register file must be accessed by the processes representing the operand fetch stage and the result write-back stage. At a lower level of abstraction, the register file would be implemented as a component with read and write ports. Read and write operations would take place over signals, and would conform to some signalling protocol. However, at the behavioral level of abstraction, the register file may be more simply described as a protected type with read and write methods. The mutual exclusion afforded by the protected type ensures that reads and writes to not interfere with each other.

The second requirement motivating shared variables is a requirement for instrumentation of models. A shared variable can be used to collect information about the aggregate behavior of processes within a design over the course of a simulation run. Each process updates the variable using update methods as events of interest occur. Mutual exclusion ensures that concurrent updates do not conflict. An example of this use is a multiprocessor computer system with cache memories attached to each processor. A shared variable might be used to collect sharing statistics for each block in the address space. Each process representing a cache would invoke a method to update the statistics when the sharing state of a block changes.

Shared variables using protected types are not intended as a general purpose inter-process communication mechanism. Thus they are not appropriate for use in system-level modeling to model communication between system components. Experience in concurrent programming languages has shown that remote procedure call and message passing are more appropriate paradigms for that purpose. Hence, VHDL protected types do not provide mechanisms as seen in other monitor-based languages for waiting and signalling within monitors. This is not to say that system-level communication is not required. Indeed, there are other efforts in progress addressing the issue [2].

While protected types bear some similarities to classes in object-oriented languages, they are not intended to fulfill that role. Classes provide encapsulation of data, allowing definition of abstract data types. Protected types also provide encapsulation, but the reason is simply to prevent uncontrolled concurrent access to the data. The preferred mechanism for defining abstract data types in VHDL is to use types and operations defined in packages. Another significant distinction is that classes include mechanisms for inheritance, whereas VHDL protected types do not. There is currently much interest in defining object-oriented extensions to VHDL [3], and the IEEE is in the preliminary phases of preparing a trial use standard.

One final point to note in using VHDL protected types is the potential for deadlock. It is possible to write a model in which two processes block waiting for mutual exclusion over shared variables, and can never resume. To illustrate the possibility, consider an extension of the shared counter example. Suppose the protected type is augmented to include a method to copy the value from one counter to another. The method is declared in the protected type declaration as

```
procedure copy ( variable from : in shared_counter );
```

The method implementation is declared in the protected type body as:

```
procedure copy ( variable from : in shared_counter ) is
begin
    count := from.value;
end procedure copy;
```

Two shared variables are declared as

```
shared variable a, b : shared_counter;
```

Now consider what might happen if process P_1 executes the statement

```
a.copy(b)
```

and process P_2 executes

```
b.copy(a)
```

A possible interleaving of execution involves P_1 acquiring access to a and P_2 acquiring access to b, before either reaches the body of the method. Note that passing a variable of protected type simply involves passing a reference; it does not involve acquiring access to the variable. When P_1 reaches the invocation of the from method within the copy method, it tries to acquire access to b. Since P_2 already has access to b, P_1 is blocked. Similarly, when P_2 reaches the invocation of the from method, it tries to acquire access to a. P_1 already has access to a, so P_2 block. Neither process can proceed, and execution is deadlocked.

While this is a contrived example, it illustrates one situation under which deadlock can arise. The language definition does not prohibit such situations, nor does it require that an implementation detect or resolve deadlock. When writing models using protected types, we must take care not to introduce the potential for deadlock.

A Modeling Example

This example illustrates use of shared variables for instrumenting a behavioral model of a multiprocessor cache memory system. The instrumentation records for each memory block the number of read misses for shared data, read misses for private data, and write misses.

The package declaration cache_instrumentation includes a protected type declaration for the instrumentation data structure. The protected type includes methods for logging read and write misses and for dumping the data to a file. The package also declares a shared variable of the protected type for collecting the data. The implementations of the data structure and the methods are described in the protected type body, which is declared in the package body. The protected type body includes an array of records, one for each block of the multiprocessor's memory space. Each record contains counters for the different kinds of misses to be logged.

The individual caches are described by a behavioral architecture body, which includes a cache controller process. The process invokes the instrumentation methods when cache miss events occur. The complete multiprocessor system is described by an architecture body that generates multiple instances of a processing element, each with an attached cache. Since each cache instance includes a cache controller process, there are multiple cache controller processes that may concurrently access the instrumentation variable in the instrumentation package. Furthermore, the multiprocessor model includes a process that periodically invokes the dump_log method. All of these processes share access to the instrumentation variable, so mutual exclusion is required to prevent interference.

```
package cache_instrumentation is
    use work.cache_types.all;
    type shared_counters is protected
        procedure log_read_miss ( block_number : block_range;  is_shared : boolean );
        procedure log_write_miss ( block_number : block_range );
        procedure dump_log ( file log_file : std.textio.text );
    end protected shared_counters;
    shared variable cache_counters : shared_counters;
end package cache_instrumentation;

package body cache_instrumentation is
    type shared_counters is protected body
        type counter_record is record
            shared_read_misses,
            private_read_misses,
            write_misses : natural;
        end record counter_record;
```

```
        type counter_array is array ( block_range ) of counter_record;

        variable counters : counter_array := (others => (0, 0, 0));

        procedure log_read_miss ( block_number : block_range;  is_shared : boolean ) is
        begin
            if is_shared then
                counters.shared_read_misses := counters.shared_read_misses + 1;
            else
                counters.private_read_misses := counters.private_read_misses + 1;
            end if;
        end procedure log_read_miss;

        procedure log_write_miss ( block_number : block_range ) is
        begin
            counters.write_misses := counters.write_misses + 1;
        end procedure log_write_miss;

        procedure dump_log ( file log_file : std.textio.text ) is
            use std.textio.all;
            variable L : line;
        begin
            for block_number in block_range loop
                write ( L, string'("Block ") );
                write ( L, block_number );
                write ( L, string'(":  shared read misses = ") );
                write ( L, counters.shared_read_misses );
                . . .
                writeline ( log_file, L );
            end loop;
        end procedure dump_log;

    end protected body shared_counters;

end package body cache_instrumentation;


architecture behavioral of cache is

    use work.cache_instrumentation.all;
    . . .

begin

    cache_controller : process is
        . . .
    begin
        . . .
        if hit = '0' then
            if read = '1' then
                cache_counters.log_read_miss ( current_block_number, shared = '1' );
            else
                cache_counters.log_write_miss ( current_block_number );
            end if;
        end if;
```

```
            . . .
        end process cache_controller;

        . . .

    end architecture behavioral;


    architecture system of multiprocessor is

        . . .

    begin

        PE_array : for PE_index in 0 to num_PEs generate

            PE : entity work.processor(behavioral)
                    port map ( . . . );

            L2_cache : entity work.cache(behavioral)
                    port map ( . . . );

            . . .

        end generate PE_array;

        log_controller : process is
            use work.cache_instrumentation.all;
        begin
            wait for 10 ms;
            cache_counters.dump_log ( std.textio.output );
        end process log_controller;

        . . .

    end architecture system;
```

Acknowledgements

Numerous people were involved in the IEEE Shared Variables Working Group over the course of its activities. Their work in analyzing requirements, surveying alternative solutions, and refining the details of the protected type approach lead to the current draft standard. Stephen Bailey as chair of the Working Group ensured that a good solution resulted from the effort. John Willis as author of the Language Change Specification ensured the conceptual integrity and completeness of the solution. Paul Menchini as LRM editor carefully integrated the language change specifications into the VHDL Standard document. We would like to thank all three for reviewing a draft of this article:

References

[1] G. R. Andrews and F. B. Schneider, "Concepts and Notations for Concurrent Programming," *ACM Computing Surveys*, vol. 15, no. 1, pp. 1–43, 1983.

[2] P. J. Ashenden and P. A. Wilsey, "Considerations on System-Level Behavioural and Structural Modeling Extensions to VHDL," *Proceedings of VHDL International Users Forum Spring 1998 Conference*, Santa Clara, CA, pp. 42–50, 1998.

[3] P. J. Ashenden, P. A. Wilsey, and D. E. Martin, "SUAVE: Painless Extension for an Object-Oriented VHDL," *Proceedings of VHDL International Users Forum Fall 1997 Conference*, Arlington, VA, pp. 60–67, 1997.

[4] N. Gehani and A. D. McGettrick, Eds., *Concurrent Programming*. Wokingham, UK: Addison-Wesley, 1988.

[5] C. A. R. Hoare, "Monitors: An Operating System Structuring Concept," *Communications of the ACM*, vol. 17, no. 10, pp. 549–557, 1974.

[6] IEEE, *Draft Standard VHDL Language Reference Manual*. Draft Standard P1076-1992/A, New York, NY: IEEE, 1992.

[7] IEEE, *Shared Variable Language Change Specification (PAR 1076A)*. IEEE DASC P1076a Working Group, http://vhdl.org/vi/svwg/lcs/lcs.htm, 1996.

[8] ISO/IEC, *Ada 95 Reference Manual*. International Standard ISO/IEC 8652:1995 (E), Berlin, Germany: Springer-Verlag, 1995.

[9] G. D. Peterson and J. Willis, "A Taxonomy of Parallel VHDL Simulation Techniques," *Proceedings of VHDL International Users Forum Fall 1996 Conference*, Menlo Park, CA, 1995.

Sidebar

Standardizing VHDL

VHDL stands for VHSIC Hardware Description Language. The US Department of Defense originally developed the language to help solve its electronic component procurement problems. Parts supplied to defense agencies were to be documented in VHDL, reducing the costs of developing and maintaining complex parts with long life-cycles.

As a means of promulgating the language, the Department of Defense turned it over to the IEEE for standardization. The IEEE has a well-defined process for developing a standard, starting with a Project Authorization Request (PAR) put forward by a sponsoring Technical Committee or a Standards Coordinating Committee. The sponsor forms a Working Group to draft the standard document. The sponsor also forms a pool of balloters, who review the draft standard and vote upon it. Balloters who vote negative must provide proposed changes that, if adopted, would allow them to reverse their votes. If there are sufficiently many negative votes, the revised draft is recirculated for further ballot. When the balloters accept the draft, it is forwarded to the IEEE Standards Board for final approval and adoption as a standard. Subsequently, the standard must be reviewed every five years and be reaffirmed, revised, or rescinded.

Standardization of the VHDL was sponsored jointly by the Design Automation Technical Committee and Standards Coordinating Committee 20. The sponsors formed the VHDL Analysis and Standardization Group (VASG) in 1985 as the Working Group responsible for developing the draft IEEE standard from the previously developed language definition. In 1987, IEEE Std 1076-1987 was approved. The language defined by that standard is informally known as VHDL-87.

In 1990, the VASG started the process of reviewing the standard. Various subcommittees collected requirements from users for improvements to the language, designed language extensions, validated the extensions, revised the standard document, and conducted ballots. Due to the complexity of the task, they were not able to provide a revised draft to the Standards Board in time for approval in 1992. Hence the current version of the standard is IEEE Std 1076-1993 [1], defining the language known as VHDL-93.

One of the requirements that the VASG was unable to meet satisfactorily in VHDL-93 was provision of share variables with well-defined concurrency control. Hence, the VASG formed a separate Working Group chartered to develop shared variable language mechanisms, and to draft changes to the standard to incorporate the mechanisms. The revised standard, IEEE Std 1076a, will replace the existing standard.

References

[1] IEEE, *Standard VHDL Language Reference Manual*. Standard 1076-1993, New York, NY: IEEE, 1993.