

The Risks and Opportunities with Shared Variables

Vijay Vaidyanathan
Zycad Corporation,
47100 Bayside Parkway,
Fremont, CA 94538, USA

VIUF Fall 1993

Abstract

This paper discusses some well known problems associated with shared variables and examines their relation to VHDL modelling, the emerging VHDL92 standard, and what a user can do to minimize the risks associated with using this powerful new feature.

1 Introduction

One of the most significant (and debated) modifications to the VHDL language in the 1992 revisions, was the addition of shared variables. In the final version of the VHDL92 Language Reference Manual (LRM), shared variables were retained, but the LRM specifically avoided associating any semantics with concurrent access to shared variables. There is currently a shared variable Working Group that is chartered with finding a satisfactory solution to the problem. It is clear that some form of shared variables are a reality in the language.

Unfortunately, the lack of a completely specified semantics for shared variables opens the door to an abuse of the construct. This paper aims to educate the user about some of the potential hazards of using shared variables, but points out that the addition of this power-

ful new class of object gives the sophisticated user tremendous new expressive power.

1.1 Process Communication in VHDL

The 87 version of the VHDL Language Reference Manual disallows shared variables. It does this by specifying that the only declarative regions where a variable may be declared are in the process declarative regions and in subprogram declarative regions. The effect of this, is that variables may be read (and written) only from within the process that they are declared in, and are not visible to any other process. Processes that need to communicate values between each other are required to do so using signals. Thus, signals are the shared resource that processes may jointly read or write (and therefore use for synchronization) in order to pass values between processes.

The designers of the language enforced a strict mechanism by which values of signals are made unavailable to all processes until that phase of the simulation cycle is complete (i.e. all processes hit a wait). This is the familiar delta cycle minimum delay that is associated with a signal write. If this were absent, a process that read the value of a signal could not be sure that the value that it read was reli-

able. In fact, it could not even be certain that the boolean condition ($S1 = S1$) would evaluate to TRUE (where $S1$ is a SIGNAL). This is because, if it were not for the delta cycle delay, it is possible that another (concurrent) process wrote to signal $S1$ after the LHS of the boolean expression above was evaluated, but before the RHS was evaluated.

1.2 The shortcomings with Shared Signals

There are two basic properties of signals that make them difficult to use in a variety of applications.

The first of these is the fact that there is always a delta delay associated with writing to a signal. Thus, the value of a signal is not immediately available after it has been written to. As a result of this, successive writes to the signal may not create the desired effect. For instance:

```
p: process (s)
  begin
    s <= s+1;
    s <= s+1;
  end process;
```

This does not increment s by 2 everytime it is invoked, but increments s only by 1 (the first assignment is lost). While the above example may seem artificial, there are certain modelling situations (as we shall see) where this delta delay does get in the way.

The second basic "problem" with signals is that they are inherently expensive objects. Most VHDL implementations would take considerably more space to represent a signal than the corresponding variable. Writing to a signal is also typically more expensive than writing to a variable.

As we shall see, the introduction of shared variables into the language certainly solves the

first of these problems, but may not provide a satisfactory solution to the second.

This paper will first describe some of the opportunities that this new feature offers, followed by some new problems that it introduces.

2 The Opportunities

In this section we examine some of the applications of this new construct.

2.1 Improved modelling power

The sophisticated VHDL user will now be able to write models that were previously extremely difficult to write, if not altogether impossible.

2.1.1 Stack manipulation

Consider the simple (albeit artificial) case of a component that acts as "lower case string reverser". (A simple twist on the unbounded buffer problem with a single reader and writer).

This component is sensitive to a string signal "StringIn", and a control signal "Print". When the Print signal goes to TRUE, it prints out, in reverse order, all the lower case characters that were passed to it in the StringIn signal.

```
entity reverser is
  port (InString : in string (1 to 10);
        Print : in boolean);
end reverser;
```

Let us think about the most natural way to implement this. It is conceivable that there be two separate processes, one to print when asked to, and one to accumulate the string signals. The simplest way to accumulate the

string signals would be to push each lowercase character on a stack. The printer would then simply pop each element off the stack and print it.

This turns out to be tricky to implement in VHDL (without shared variables). The stack itself needs to be shared between the two processes. Thus it has to be a signal. While this is bad enough, consider what happens in the printer process. It loops until the stack is empty, popping successive characters off the stack. However, all of this happens in the same delta cycle. If the stack pointer is implemented as an array index, then decrementing the stack pointer has no effect on the value of that signal, and hence this will loop forever.

```
-- This process does not work.
-- It tries to dump out the contents
-- of a stack, clearing the stack
-- while doing so.

print_it : process(Print)
begin
  if (Print) then

    -- NextFree is the index into the
    -- array that implements
    -- the stack i.e. the stack pointer,
    -- points to the next
    -- free entry in the stack

    while NextFree /= MinStack loop

      NextFree <= NextFree -1;
      --      ^--- Error!!

      PrintChar(CharStack(NextFree));
    end loop;
  end if;
end process;
```

This is not all that artificial an example. It is quite conceivable that such a stack be cre-

ated for debugging, and that a test signal controlled by a testbed, dumps the contents of the debugging stack for perhaps a nightly regression run of the model.

One solution in this case to work around the absence of shared variables would be to artificially glob the processes together and use a local (non shared) variable. In general, however, that is a poor strategy since globbing processes together decreases the available parallelism in the model, possible leading to a degradation of simulation performance in simulators that can exploit the inherent parallelism in VHDL.

However, with the presence of shared variables, the modeller can implement the stack as a shared variable array, and the value of the variable will then be available to both processes.

2.1.2 Modelling Temperature of a component

Another simple example of an application that is hard to implement without shared variables is to model temperature of a component. Typically, the modeller would like to model the temperature of a component by counting the level of activity in the component processes of the circuit. This is hard to do with signals for the reasons specified above. For instance, if an n-bit register is implemented by "n" processes, and we wish to count the number of 0-1 transitions and the number of 1-0 transitions, all "n" processes need access to the shared variable that maintains this count. Since it is quite possible that two or more bits change state in the same delta cycle, a shared signal would not be able to maintain this count accurately.

2.1.3 Instrumenting the model: gathering statistics

Another similar and simple application is for instrumenting the model for data gathering.

For instance, the developer of a RISC processor model may be very interested in gathering statistics about the number of times a particular instruction is inserted into the instruction pipeline and comparing it with the number of times it actually ends up getting executed. Depending on the model, it is quite possible that the variables that maintain these statistics need to be updated from different processes.

2.2 “Low cost” signals

One of the reasons that VHDL users have asked for shared variables is for the ability to declare “light weight” or “low cost” signals. The classic example that is often stated is that of a memory. Typically, a memory component may be modelled by a READ process and a WRITE process, and the memory itself modelled as an array. Now for most simple applications, the actual model is simple enough that there are no delays etc that need to be associated with writing to memory. However, the memory array still needs to be implemented as a signal, since the READ and WRITE processes both need access to the array.

Worse, if two different processes need to write to the array (e.g. perhaps a separate process updates the parity bit), the signal array may need to be defined to be of a resolved type, and a resolution function supplied.

While it is true that this is sometimes unnecessary, and that a shared variable may provide a simpler and more efficient solution, the user is strongly cautioned (as we will discuss below) not to assume that shared variables are necessarily less expensive than signals. As we shall see, it is quite conceivable that in certain implementations under certain conditions, a shared variable is as expensive, if not more expensive, than a signal.

2.3 Automatic Translation from other HDLs

One of the original reasons for introducing this construct into the language was in order to make the automatic translation of models from other HDLs to VHDL more feasible. However it is unclear that the lack of shared variables represents the sole hurdle in this process. It is not clear to this author whether such automatic translation tools will become a reality in the near future.

2.4 Efficient algorithms on Multiprocessor implementations

One of the more interesting applications of shared variables is that it will allow the user to more easily exploit the advantages of multiprocessor implementations of behavioral models in VHDL. With the increasing popularity and availability of multiprocessor workstations and increasing Operating System support for multithreading, there is now an opportunity for dramatic improvements in simulation performance by exploiting the concurrency that is inherent in VHDL. We show below a very simple example of a sorting algorithm that exploits this concurrency.

The best possible sorting algorithm on a uniprocessor implementation takes $O(n \log n)$ time. The following algorithm however, the RankSort algorithm, is slower on a *sequential implementation* and is $O(n^2)$. On the other hand, the Ranksort algorithm is actually a very good concurrent algorithm, and in fact is $O(n^2/p)$ where p is the number of available processors. In the artificial case, when the number of available processors is unbounded, we can actually use as many as n processors to sort an n element array, yielding a complexity of $O(n)$ which is better than even the best possible sequential bound of $O(n \log n)$.

Even under more realistic situations with

bounded number of processors, Ranksort performs remarkably well.

The Ranksort algorithm works as follows: the “rank” of an element in an array of elements (assumed unique, for simplicity) is defined as the number of elements in that array that are less than that element. Thus, to compute the rank of an element, we loop over the array, maintaining a count of the number of elements that are less than the element whose rank we wish to find. The simplicity of the ranksort algorithm lies in the fact that the rank of an element is simply the final index of that element in the resulting sorted array! The ranksort algorithm works by letting each element in the array “find its own rank”, say n , and then simply “inserting itself” into the n th element of the final array.

This is a particularly good algorithm because it is a relaxed algorithm. There is no need for any synchronization between any of the “ n ” processes: each process “ n ” finds the rank of element “ n ” and inserts that element into the appropriate position in the final array. If all “ n ” processes run in parallel, the rank-finding step takes $O(n)$ and the insert step takes constant time, therefore completing the sort in linear time!

Since this is a relaxed algorithm it would be far more efficient to use shared variables to sort the array rather than signals with resolution functions.

3 The Risks

Having spent some time describing the opportunities available with shared variables, we now deal with the risks that they pose.

VHDL is a concurrent language. When you write a VHDL model, you are, in effect, writing a concurrent program, and therefore should be aware of the problems associated with that.

3.1 The Shared Resource Problem

The underlying problem with concurrency and shared variables is that concurrent processes need concurrent access to the shared resource (in this case, a VHDL variable). Although this is an issue with signals as well, the “sharing” of signals is solved in VHDL by the introduction of 2 mechanisms:

The first, is the delta delay between the time that a signal is assigned a value and the time that the value is available to be read. Thus, the boolean condition ($S1 = S1$) will always evaluate to TRUE in every VHDL implementation.

The second is the rule that requires a resolution function for every signal that is driven from more than one process. Thus, in effect, the user is allowed to specify exactly what happens when two or more conflicting values are written into a signal. Further, the LRM specifically describes exactly when the resolution function is called as well as when that signal takes the resulting value.

When a variable is shared between two processes, the user does not expect to see a delta delay in the variable assignments (in fact, that is specifically why signals are unsuitable for some situations). This leads to the problem of deciding exactly what happens when there is conflicting access to a shared variable. Consider the following VHDL fragment:

```
p1: process
  begin
    :
      shared_var := value;
    :
  end process;

p2: process
  begin
    :
```

```

    if (shared_var = value) then
      :
    end if;
    :
  end process;

```

Since *p1* and *p2* are executed in parallel, there are only two possibilities: (1) the fragments are always executed in different delta cycles, or, (2) the fragments do sometimes execute in the same delta cycle.

If (1) is true then this program will run consistently. On the other hand, if (1) is not true, then this program will run erratically, since the condition within the *if* may or may not be true depending on whether the variable assignment in *p1* has been executed or not.

In some implementations, this behavior may be arbitrary but repeatable. For instance, even a sequential implementation that ran every process in textual order without preemption may always schedule *p1* before *p2* and therefore the condition will always be true. On the other hand, that same implementation would produce different behaviour if *p2* appeared textually before *p1*.

This then demonstrates the first problem with shared variables: it introduces a new level of non-determinism and non-portability.

3.2 The Arbitrary Interleaving Assumption

When modelling concurrent processes, there is a very important assumption that must be made. This is called the Arbitrary Interleaving assumption.

It is based on the fact that VHDL processes are concurrent processes with independent threads of control. As a result, processes proceed at an unspecified rate, and in an unspecified order within each delta cycle. Essentially, this assumption states that we must assume that the constituent primitive actions

that make up each concurrent process may be arbitrarily interleaved in time.

Let us take an example:

Consider the case of two concurrent processes that manipulate a shared resource, e.g. a hard disk.

```

writer: process
  begin
    :
    seek(sector_to_write);
    write_data(out_buffer);
    :
  end process;

reader: process
  begin
    :
    seek(sector_to_read);
    in_buffer := read_data;
    :
  end process;

```

The arbitrary interleaving assumption states that all of the following sequence of events must be assumed to be legal and possible.

```

seek(sector_to_write);
write_data(out_buffer);

seek(sector_to_read);
in_buffer := read_data;

```

which produces reasonable behaviour. However, another invocation of this same code may produce the sequence:

```

seek(sector_to_read);
seek(sector_to_write);

write_data(out_buffer);
in_buffer := read_data;

```

which would do the expected action for the writer, but the reader would get incorrect data;

or:

```
seek(sector_to_write);
seek(sector_to_read);

in_buffer := read_data;
write_data(out_buffer);
```

in which case the data would get to the reader as expected, but the writer would not get expected results. In fact, the following interleaving:

```
seek(sector_to_write);
seek(sector_to_read);

write_data(out_buffer);
in_buffer := read_data;
```

is a legal interleaving, but would satisfy neither the reader nor the writer.

The designer must write models that produce predictable behaviour in the presence of the arbitrary interleaving assumption. If the model does not take this into account, it is quite conceivable that the model behaves radically differently in one implementation from than in another even if both implementations are legal VHDL compliant implementations. Worse, the same model may behave radically differently during successive runs of the exact same VHDL model.

There is another property that is significant here, called "Finite Progress". A scheduling algorithm satisfies the finite progress criterion if every process that is eligible to run is guaranteed to be scheduled to run within some finite period of time (however long). Finite progress is a very useful property, but unfortunately few implementations guarantee it. In order to guarantee finite progress, the VHDL

scheduling kernel must be capable of preempting running processes, which is something that the LRM does not require an implementation to do. Thus, the VHDL modeller is unfortunately required to live with the Arbitrary Interleaving assumption without having the luxury of relying on Finite Progress.

This combination makes things even more difficult than the disk access example indicates. Consider the following case: Here, one process writes a value to a shared variable, while another process reads it. Recognising the fact that the Arbitrary Interleaving assumption may cause the value to not yet get written, the modeller decides to busy wait on the shared variable until the value does get written:

```
p1: process
begin
  :
  shared_variable := value;
  :
end process;

p2: process
begin
  :
  if shared_variable = value then
    print "a";
  else
    -- naive wait for shared_variable
    -- to reach value
    while shared_variable /= value
      -- do nothing, busy wait!
    loop end loop;

    -- Aha! shared var reached
    -- expected value
    print "b";
  end if;
  :
end process;
```

Under the arbitrary interleaving assumption, it is possible that the condition in *p2* be tested only after *p1* has executed the assignment. In that case, the test will succeed (and “a” printed). However, if the test occurs before the assignment, the `else` fragment will be executed.

In this case, the `else` fragment attempts to accommodate for this possibility by busy waiting on the value of the shared variable. Unfortunately, different implementations may choose to do different things at this point.

A sequential (uniprocessor) implementation that does not do preemptive scheduling would never run process *p1* until process *p2* blocked. However, processes *p2* is in a loop waiting for process *p1* to assign to the shared variable. As a result, the simulation would infinitely loop at this point.

On the other hand, an implementation that satisfies the Finite Progress criterion (or one where *p1* was scheduled on another processor), would eventually schedule *p1* and therefore eventually *p2* would be able to exit the busy wait loop (and print “b”).

Thus, we see that this apparently simple example could unpredictably print “a”, print “b”, or infinitely loop, and none of these would be illegal VHDL behavior!

3.3 Data Atomicity

The Arbitrary Interleaving assumption has another effect that can sometimes lead to remarkably unexpected results. The assumption states that the processes may be arbitrarily interleaved, but does not state what the minimum granularity of each interleaving may be. To be more specific, it does not state what the smallest primitive operation is, which cannot be further subdivided, and hence interleaved.

On a native compiled code preemptively scheduled solution, or in a multiprocessor solution, this is often extremely small, usually a

single hardware instruction and sometime even a single clock cycle on the underlying hardware. If the architecture that this runs on is a shared memory machine, then hardware support is required to arbitrate between the competing demands for access to the shared memory.

On the other hand, on a non preemptive uniprocessor sequential implementation, the interleaving is practically non existent, and a process is normally run until it voluntarily blocks (i.e. waits).

An interesting effect of this poorly defined data atomicity is that sometimes the primitive operation is smaller than one expects, leading to apparently bizarre simulation results. Consider the following case:

```
p1: process
begin
  :
  shared_time_variable := v1;
  :
end process;
p2: process
begin
  :
  shared_time_variable := v2;
  :
end process;
```

It is then reasonable to expect that there are two possible scenarios under the Arbitrary Interleaving assumption. As a result, it may be reasonable to expect that once both processes have run, the value of the shared time variable be either *v1* or *v2*. However, in the above case, its is quite possible that the final value be neither *v1* nor *v2*.

The reason for this is that often, variables of type `TIME` are represented as 64 bit numbers. Hence, the variable assignments above, may be translated into 2 machine instructions, one to load the higher 32 bits and one to load

the lower 32 bits. The primitive operations are now the 32 bit load instructions, and the following is a possible interleaving:

```
load high32bits of v1 to
  high 32 bits of shared_time_variable

load high32bits of v2 to
  high 32 bits of shared_time_variable

load low32bits of v2 to
  low 32 bits of shared_time_variable

load low32bits of v1 to
  low 32 bits of shared_time_variable
```

As a result, the value of the shared time variable now contains the top 32 bits from v2 and the lower 32 bits from v1, resulting in a value that is neither v1 nor v2!

This kind of problem is caused by the lack of data atomicity. The above situation would not pose a problem if the writing of all 64 bits could be constructed as an atomic operation that could not be interleaved. This is the principle behind the identification of regions of code that require non interleaved, non interruptible, exclusive access to shared resources, called *critical regions*.

3.4 Critical Regions

A critical region is simply a contiguous fragment of code that requires uninterrupted exclusive use of a shared resource. By specifying a critical region, the modeller is identifying portions of the code that cannot be interleaved with other regions of code that require access to that same shared resource.

In the disk access example, the seek-read sequence forms a critical region. similarly, so does the seek-write region.

If a model contains concurrent processes that access shared variables, there are usu-

ally critical regions present. If the critical regions are not identified, arbitrary interleaving may cause unpredictable behavior. Thus, for any model containing critical regions to behave predictably, the modeller must take precautions to identify these critical regions to the implementation so that these regions are not arbitrarily interleaved.

The current version of the LRM specifically avoids specifying any such mechanisms. We shall therefore try and keep the following discussion as broad as possible.

4 Some other Hazards

Finally, there are a few other problems with shared variables. For instance, the LRM does not disallow the possibility of shared variables of an access type. However, declaring a shared variable of an access type has the potential to cause immense complications.

Intuitively, this is due to the fact that synchronization mechanisms are defined on the shared variable and not on the data it refers to. However the data referred to by the shared variable, is, for all practical purposes, a shared resource as well. We shall not dwell on these issues due to a lack of space, but it is the opinion of this author that modellers who are interested in predictability and portability should refuse to declare shared variables of access types, and that vendors issue appropriate warnings when such constructs are encountered.

5 Some Solutions

As we have seen above, most of the complications caused by shared variables boil down to the ability of the implementation to provide the modeller with constructs to identify critical regions.

In this section, we shall examine some solutions towards this end. We shall not go into great detail with any of them, because until the shared variable Working Group reaches a consensus, it is unclear which of these solutions are most easily implemented.

However, all of these solutions have the following in common: They all require that the user specify (either explicitly or implicitly) the region of code that form a critical region. These methods differ only in terms of expressive power in defining these critical regions as well as in “ease of use”.

For instance, some methods are extremely simple to describe (semaphores and spinlocks). On the other hand, these are far more immune to programmer error. Some other methods (e.g. Monitors) are far more elegant and much less prone to programmer error. However, monitors require syntactic modifications to the language and are also initially slightly more difficult to understand.

5.1 Spinlocks

A spin lock is a simple synchronization mechanism. Intuitively, a spin lock represents the state of a shared resource or a critical region. The shared resource is either being used (i.e. Locked) or not being used (i.e. Unlocked). Another way of viewing this is that program flow is within the critical region (i.e. Locked) or outside the critical region (i.e. Unlocked).

At any given time, the lock itself maintains a state (locked, or unlocked). To modify this state, there are just two operations supported on a spin lock: *Lock* and *UnLock*.

The *Lock* operation changes the state of the lock from unlocked to locked. The *UnLock* operation changes the state of the lock from locked to unlocked. These operations are provided by the underlying implementation and are guaranteed to be executed atomically.

However, the *Lock* operation has another important property. If the state of the lock is already locked, then the lock operation does not return until another process does an *UnLock*. At which time, the *Lock* operation sets the state back to locked and then returns to the caller of the *Lock*.

This gives us a way to demand exclusive control over a critical region. For instance, the disk access example would be recoded as follows:

```
USE work.spinlocks.all;

-- there is a declaration
-- of spin lock (e.g.
-- variable disk : spinlock)
--

writer: process
begin
:
Lock(disk);
seek(sector_to_write);
write_data(out_buffer);
UnLock(disk);
:
end process;

reader: process
begin
:
Lock(disk);
seek(sector_to_read);
in_buffer := read_data;
UnLock(disk);
:
end process;
```

This would ensure that if one of the processes were to hit the *Lock* after the other had already executed the *Lock*, the second process would have to wait until the process that had the lock performed an *UnLock* (i.e. exited the critical region).

It is important to note that the spin lock operations must be implemented by the underlying implementation, since the operations must be atomic. Further, the actual implementation details depend on the underlying scheduling algorithm and therefore cannot be implemented by the user. In fact, the implementation of spin locks will necessarily be different on different implementations or platforms.

For instance, an implementation may provide the following VHDL package declaration, but its body must be “built in”, like package STANDARD.

Note that the *only* legal operations on spinlocks should be through the provided *Lock* and *UnLock* operators. Directly assigning to a spinlock will probably lead to unpredictable results.

```
package spinlocks is

  type spinlock is (Locked, UnLocked);

  procedure Lock(L: inout spinlock);
  procedure UnLock(L: inout spinlock);

end spinlocks;
```

Although spin locks form a very simple mechanism, it has several shortcomings.

First, it is easy to make an error in this. Common errors include inadvertently swapping the lock and unlock, forgetting to execute an *UnLock* (perhaps due to a complicated *if* or *case* statement.)

Another problem with spin locks is that it is the responsibility of the user to associate a unique spin lock with each critical region. If the user uses the same lock for several critical regions that don't really have conflicting resource needs, it will lead to a degradation in performance because it has the effect of “sequentializing” the processes. On the other hand, if the user associates too few locks with

the resources, then the behaviour will once again be unpredictable since regions requiring exclusive access may be arbitrarily interleaved.

Intuitively, these problems arise from the fact that a spin lock is not necessarily linked to the critical region automatically. The responsibility to do so, as well as the mapping, is the responsibility of the implementor.

Finally, as the above implementation details show, the underlying scheduler must provide the functionality of the *Lock* operation. If the scheduler needs to suspend the locking process until the resource is freed (in the absence of finite progress). If a process suspends because of the unavailability of the spin lock, then the scheduler must maintain a list of processes that are waiting for this lock, and run the processes as an when the lock becomes available. This has the effect of sequentializing the execution of processes that may otherwise have run concurrently.

These were the implementation overheads that were alluded to in earlier paragraphs. Depending on the scheduling kernel, this may cause a significant overhead, and as a result, shared variables in certain implementations may turn out to be computationally expensive.

5.2 Semaphores

Semaphores are a more powerful construct than spin locks, and address some of the problems with spin locks.

Much like spin locks, there are 2 semaphore operations. However, these operations are performed directly on the shared resource which in our case is the shared variable. (More precisely, the operations are performed on a semaphore associated with the shared resource.) The two operations are often called *Wait* and *Done*. These operations must be implicitly declared whenever a shared variable is declared. Hence, this mechanism is harder to

supply in the form of a VHDL package.

In [PM] the interesting observation was made that this could be done by defining the Wait and Done operations to operate on the 'PATHNAME of a shared variable

The Wait operation has the effect of granting access to the shared variable if available, and suspending a process on that shared variable otherwise. The Done operation has the effect of releasing the shared variable for access by another process, and scheduling the next process waiting for that shared resource.

The implementation of a semaphore is simple, but we shall not describe it here. There are several excellent textbooks that describe semaphores in great detail.

Although semaphores are more useful than spin locks in some situations, their implementation is often quite different, and they share the problem of leaving the burden of matching the Wait and Done semaphores upto the user.

5.3 Monitors

One of the most elegant solutions to the synchronization problem is through the use of Monitors.

A Monitor is a package of shared data and operations that are defined on that shared data. The only code fragments that are allowed access to the shared data are the operations defined in the monitor. Thus, the user is not permitted to define a shared variable and then operate on it directly. All operations must be performed through one of the operations on the monitor.

Since the user is disallowed from directly performing the operation (e.g. incrementing a shared variable), the monitor can automatically control exclusive access to that shared variable (by defining the entire operation to be a critical region). This relieves the user from the responsibility of identifying and controlling access to critical regions.

On the other hand, monitors are harder to use, and require syntactic modifications to the language. Although monitors are the technically superior approach, there has been some resistance to the introduction of monitors into the language on those grounds.

6 Should you use Shared Variables?

First, if the user is interested in repeatable predictable and portable VHDL models, it is dangerous to use shared variables without some form of synchronization.

In fact, there are only two possibilities in any model: (1) There is never a reader of a shared variable in the same delta cycle as an (only) writer of that shared variable. (2) There is one or more writers and one or more readers of a shared variable during the same delta cycle.

If (1) is true, then shared variables may be used safely without any synchronization mechanism. However, in that case, so can a signal! The only remaining advantage then of using shared variables without critical region protection is for the "low cost" advantage of a shared variable over a signal. As we have pointed out, there are situations in which this advantage may not exist.

If (1) is not true and (2) is true, then there is an arbitrary interleaving that could cause different behaviour from one run to the other, and therefore you must use some form of critical region protection or synchronization.

7 Conclusions

The most important observation is that without the use of synchronization primitives, almost any use of shared variables will lead to unpredictable and non-portable behaviour.

However, with the judicious use of these synchronization primitives, there is a great deal of expressive power at the disposal of the sophisticated user.

[HO85] Hoare, C. A. R., Communicating Sequential Processes Comm. of the ACM 21, pp. 666-677. (1985)

References

- [PM] Menchini, Paul *Personal Communication*. (1993).
- [AND91] Andrews, G.R., Concurrent Programming: Principles and Practice. Benjamin/Cummings. (1991).
- [BENARI82] Ben-Ari, M., Principles of Concurrent Programming. Prentice Hall Intl. (1982).
- [BH73] Brinch Hansen, P., Operating System Principles. Prentice Hall. (1973)
- [CRS92] Snow C.R., Concurrent Programming Cambridge Computer Science Texts. (1992)
- [BL93] Lester, Bruce P., The Art of Parallel Programming Prentice Hall. (1993)
- [CON63] Conway M.E., A Multiprocessor System Design. *Proceedings of the AFIPS Fall Joint Computer Conference*, pp. 139-146. (1963)
- [DI67] Dijkstra, E.W., Co-operating Sequential Processes. *Programming Languages*, F. Genyus ed., Academic Press, pp. 43-112. (1967)
- [HO72] Hoare, C. A. R., Towards a theory of parallel programming. *Operating System Techniques*, pp. 61-71. Academic Press. (1972)
- [HO74] Hoare, C. A. R., Monitors, an operating system structuring approach. Comm. of the ACM 17, pp. 453-455. (1974)