Modeling Shared Variables in VHDL *

Jan Madsen Jens P. Brage

Department of Computer Science
Technical University of Denmark
DK2800 Lyngby, Denmark

Abstract

A set of concurrent processes communicating through shared variables is an often used model for hardware systems. This paper presents three modeling techniques for representing such shared variables in VHDL, depending on the acceptable constraints on accesses to the variables. Also a set of guidelines for handling atomic updates of multiple shared variables is given.

1 Introduction

It is often desirable to partition a computational system into discrete functional units which cooperates to solve a given task. In order to be able to cooperate, it is necessary for the functional units to communicate information; the communication can be based on various models, one of which is shared variables.

The primary characteristic of shared variables is that multiple functional units may access a given variable for reading and writing; between updates, the shared variable retains the most recently written value. For the class of shared variables considered here, multiple *simultaneous* write accesses are not permitted.

Shared variables can be used for several different purposes, at various levels of abstractions. A few examples of these uses are:

- In the language model Synchronized Transitions [7], shared variables are used as the medium of communication between a set of *transitions*, i.e., guarded, atomic variable assignments.
- During high-level synthesis from a procedural defined functional unit in VHDL (i.e., a high-level **process**), the sequential behavior is transformed into a set of concurrent functional units. As VHDL allows an output to be updated from several points in the sequential code, this leads to a requirement for a given output to be updated from several of the resulting functional units [1]. This is naturally modeled as a shared output variable.
- The parasitic capacitances inherent in the CMOS technology allows buses to store values on the signal lines. Shared variables is a reasonable model for this behavior.

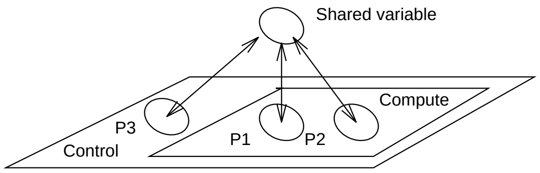

Figure 1: Inter process communication through shared variables.

Control:
  Process P3:
    var $\leftarrow$ 15;
    suspend until var = 1;
    var $\leftarrow$ 0.

Compute:
  Process P1:
    forever
      suspend until var $\neq$ 1 $\wedge$ odd(var);
      var $\leftarrow$ var * 3 + 1.

  Process P2:
    forever
      suspend until var $\neq$ 0 $\wedge$ even(var);
      var $\leftarrow$ var / 2.

Listing 1: Pseudo code to test 15 for wondrousness [3].

Listing 1 shows a simple example of an algorithm [3, pp. 400 – 401] which uses a shared variable for communication between independent processes (see figure 1).

In many cases it is interesting to be able to model the shared variable concept in VHDL. As figure 1 illustrates, a shared variable exists, in a sense, independently of the design hierarchy. This creates problems for the modeling in VHDL.

One possible approach might be to use the global vari-

*This research has been sponsored by the Danish Technical Research Council.

ables of VHDL'93 [4];[1] however, the global variables are primarily intended for system level modeling, permitting any process in the hierarchy to access a few select variables. In hardware models, where the shared variable concept is the main communication mechanism, this kind of global visibility is unacceptable; all communication should be explicitly stated in the interface declaration.

This paper presents three different models for shared variables; using the built-in support for **register**s in VHDL, modeling hierarchical shared output variables and a model providing full shared variables. Finally some usage guidelines for the latter model is presented.

2 Using the register Based Shared Variables

VHDL partly supports shared variables through the special signal kind called **register** [6, pp. 209–211]. A **register** signal retains its last value when all its drivers have been disconnected (i.e., when **null** is assigned to the signal driver).

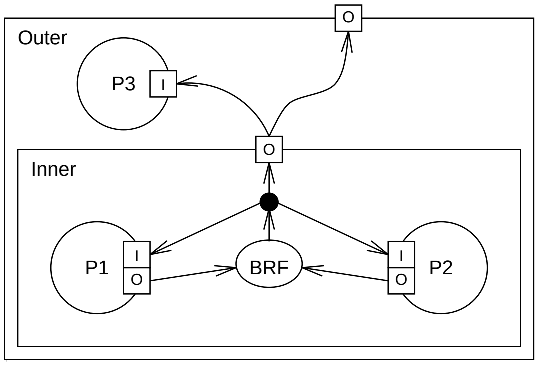

Figure 2: Modeling shared variables using the **register** construct.

```
subtype VariableType is ValueType;
type VectorType is array (INTEGER range < > ) of VariableType;
function Resolver (Input : VectorType) return VariableType;
subtype SharedVariableType is Resolver VariableType;
procedure SharedWrite (signal SharedVar : out VariableType,
                              Value : in ValueType);
```

Listing 2: Type definitions for **register** based shared variables.

Having multiple sources for the signal makes a bus resolution necessary (see listing 2; the `ValueType` may be any user type). However, as only one driver may be connected (non-**null**) at a given time, the bus resolution function is very simple, as illustrated in listing 3. A shared variable may be updated through the procedure shown in listing 4.

[1] While global variables have been included in the standard, the access mechanism has not been finalized, effectively making global variables unavailable at the time of writing.

```
function Resolver (Input : VectorType) return VariableType is
begin
  assert Input'LENGTH=1 or NOW=0 NS
    report " Multiple drivers"  severity ERROR;
  return Input (Input'LEFT);
end;
```

Listing 3: The bus resolution function for **register** based shared variables.

```
procedure SharedWrite (signal SharedVar : out VariableType,
                              Value : in ValueType) is
begin
  SharedVar < = Value;
  wait for 0 NS;
  SharedVar < = null;
end;
```

Listing 4: Procedure for updating a **register** based shared variable.

The '**wait for** `0 NS`' ensures that the shared variable will be updated before the driver is disconnected again.

The main limitation of using the **register** construct is that a hierarchy of drivers for the signal cannot be modeled (as a resolution function cannot return **null**), i.e., it is possible to propagate a value up through the hierarchy for reading, but *not* for writing. This situation is shown in Figure 2: Process `P1` and `P2` share a variable `Var` which is a signal of kind **register**. `Var` may only be written from within the entity `Inner`. The signal may however be read in process `P3` which is at the next hierarchical level.

Another problem is due to the restriction that **null** cannot be used to initialize signal drivers (the **null** literal is always of an access type). To work around this problem, the implicit loop of each process have to be translated into an explicit loop. Listing 5 shows the structure of a process statement, in which all output drivers are first set to **null** and the actual execution is then placed in the explicit infinite loop.

```
...
signal Var : SharedVariableType;
...
process
begin
  Var < = null;
  ...
  loop
    ...
    SharedWrite(Var,Var + 1);
    ...
  end loop;
end process;
```

Listing 5: The structure of a process statement using the **register** based shared variables (in the examples the `ValueType` is assumed to be an integer).

3 Modeling Shared Output Variables

The main limitation of the **register** based shared variables is that hierarchical output drivers cannot be handled. If this is an unacceptable restriction, a slightly more complicated solution must be chosen. In this case the shared variables are modeled by a **record** (listing 6) containing the value as well as drive information. Encoding the drive information within the signal enables the resolution function to propagate disconnection status.

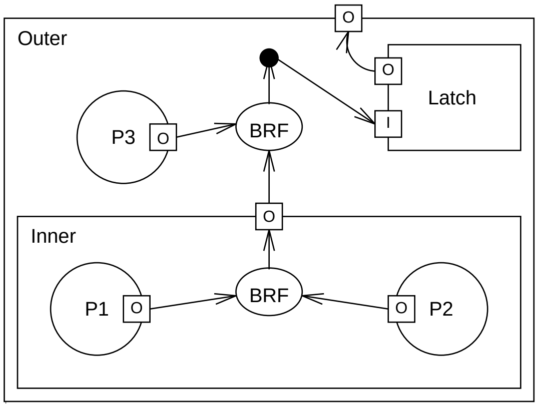

Figure 3: Modeling shared output variables.

As seen from listing 7, it is now necessary for the resolution function to scan the complete list of inputs; this was

```
type DriveType is (Invalid, Valid);
type VariableType is record
                         Value : ValueType;
                         Drive : DriveType;
                       end record;
type VectorType is array (INTEGER range <> ) of VariableType;
function Resolver (Input : VectorType) return VariableType;
subtype SharedVariableType is Resolver VariableType;
procedure SharedWrite (signal SharedVar : out VariableType,
                          Value : in ValueType);
```

Listing 6: Type definitions for shared output variables.

```
function Resolver (Input : VectorType) return VariableType is
  variable ValidSeen : BOOLEAN := FALSE;
  variable Result : VariableType;
begin
  Result := (DefaultValue,Invalid);
  for I in Input'RANGE loop
    case Input(I).Drive is
    when Valid =>
      assert not ValidSeen
        report "Multiple sources" severity ERROR;
      ValidSeen := TRUE;
      Result := VariableType'(Input(I).Value,Valid);
    when Invalid =>
      null;
    end case;
  end loop;
  return Result;
end;
```

Listing 7: The bus resolution function for shared output variables. `DefaultValue` is the desired default value for the `ValueType`.

```
procedure SharedWrite (signal SharedVar : out VariableType,
                       Value : in ValueType) is
begin
  SharedVar <= VariableType'(Value,Valid);
  wait for 0 NS;
  SharedVar <= VariableType'(DefaultValue,Invalid);
end;
```

Listing 8: Procedure for updating a shared output variable.

```
entity Latch is
  port (Shared : in VariableType;
        Output : out ValueType);
end Latch;

architecture Sticky of Latch is
begin
  process (Shared)
    variable StoredValue : ValueType := DefaultValue;
  begin
    if Shared.Drive = Valid then
      StoredValue := Shared.Value;
    end if;
    Output <= StoredValue;
  end process;
end Sticky;
```

Listing 9: Memory function for the shared output model.

implicitly handled by the simulator kernel in the previous model. Notice that in case of no `Valid` driver, the resolution function will propagate an `Invalid` value. The procedure for shared write (listing 8) is much the same as before, only the drive information is now explicitly set in the signal assignment.

For the **register** based shared variable, the language semantics provides the necessary memory function for shared variables. In the shared output model, this function must be provided explicitly; listing 9 shows an output latch which performs this function. The latch must be instantiated at the topmost hierarchical level and also provides the type conversion to `ValueType` (see Figure 3).

From listing 10 it can be seen that the explicit loops in the processes of the previous model is no longer required, as the initialization can be made in the signal declaration.

```
...
signal Var : SharedVariableType := (DefaultValue,Invalid);
...
process
begin
  ...
  SharedWrite(Var, 42);
  ...
end process;
```

Listing 10: The structure of a process statement using the shared output variables.

4 Modeling Full Shared Variables

The shared output model can easily be extended to handle hierarchical writing *and* reading, permitting full shared variables to be modeled; this is illustrated in Figure 4. The main extension is the inclusion of a weak driving value,[2] in addition to the `Valid` and `Invalid` values, as illustrated in listing 11.

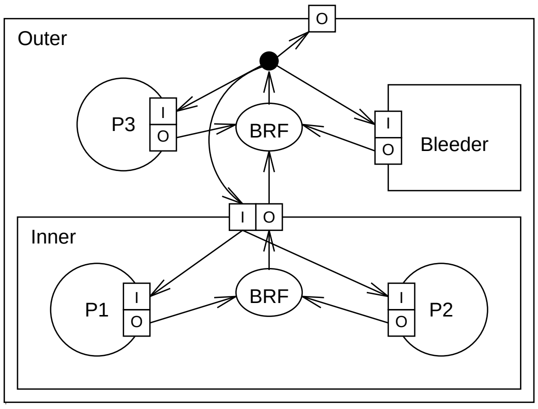

Figure 4: Full shared variables.

```
type DriveType is (Invalid, Weakvalid, Valid);
type VariableType is record
                        Value : ValueType;
                        Drive : DriveType;
                     end record;
type VectorType is array (INTEGER range <> ) of VariableType;
function Resolver (Input : VectorType) return VariableType;
subtype SharedVariableType is Resolver VariableType;
procedure SharedWrite (signal SharedVar : out VariableType;
                        Value : in ValueType);
function SharedRead (signal SharedVar : in VariableType)
         return ValueType;
```

Listing 11: Type definitions for full shared variables.

Having introduced the `Weakvalid` drive value, the resolution function has to be extended to correctly handle the relative strength of the input signals. Listing 12 shows the extended resolution function, which ensures that a `Valid` signal value will override a `Weakvalid` value, as a `Weakvalid` value overrides an `Invalid` value. The strongest value present will thus propagate up through the hierarchy.

As in the previous model, the strong value originates from the `SharedWrite` procedure. The `Weakvalid` value is generated by a 'bleeder' component, which is instantiated once for each shared variable (the resolution function checks for the occurrence of more than one bleeder per shared variable). The bleeder component, see listing 13, may be placed anywhere in the hierarchy and serves to maintain the value of the shared variable between updates.

[2]This is inspired by shared variable implementations in CMOS [5].

```
function Resolver (Input : VectorType) return VariableType is
  variable ValidSeen : BOOLEAN := FALSE;
  variable WeakSeen : BOOLEAN := FALSE;
  variable Result : VariableType;
begin
  Result := (DefaultValue, Invalid);
  for I in Input'RANGE loop
   case Input(I).Drive is
   when Valid =>
     assert not ValidSeen
       report " Multiple sources"  severity ERROR;
     ValidSeen := TRUE;
     Result := VariableType'(Input(I).Value,Valid);
   when Weakvalid =>
     assert not WeakSeen
       report " Multiple bleeders"  severity WARNING;
     WeakSeen := TRUE;
     if (not ValidSeen) then
       Result := VariableType'(Input(I).Value,Weakvalid);
     end if;
   when Invalid =>
     null;
   end case;
  end loop;
  return Result;
end;
```

Listing 12: The bus resolution function for the full shared variables.

```
entity Bleeder is
  port (Shared : inout SharedVariableType :=
                 (DefaultValue,WeakValid));
end Bleeder;

architecture Bloody of Bleeder is
begin
  process (Shared)
    variable StoredValue : ValueType := DefaultValue;
  begin
    if Shared.Drive = Valid then
      StoredValue := Shared.Value;
    end if;
    Shared <= VariableType'(StoredValue,Weakvalid);
  end process;
end Bloody;
```

Listing 13: Memory function for the full shared model.

Listing 14 shows the `SharedRead` function which is used to sample the value of a shared variable, hiding the implementation details.

Listing 15 shows a full-scale example using this model for shared variables. The `Inner` structure contains two processes, which cooperate to produce a sequence of numbers from a given seed value, using a shared variable to store the values. If this sequence results in the number 1, the original seed value was a wondrous number. The third process, placed higher in the hierarchy, places the original seed value on the shared variable and checks for wondrousness. When wondrousness has been detected, it stops `P1` and `P2` by setting the variable to 0.

```
function SharedRead (signal SharedVar : in VariableType)
        return ValueType is
begin
  return SharedVar.Value;
end;
```

Listing 14: Procedure for reading the full shared variables.

```
entity Inner is
  port (Var : inout SharedVariableType :=
           (DefaultValue, Invalid));
end inner;

architecture Wondrous of Inner is
begin
  P1: process
  begin
    wait until SharedRead(Var) /= 1 and
             SharedRead(Var) mod 2 /= 0;
    wait for 5 NS;
    SharedWrite(Var,SharedRead(Shared)*3 + 1);
  end process;

  P2: process
  begin
    wait until SharedRead(Var) /= 0 and
             SharedRead(Var) mod 2 = 0;
    wait for 5 NS;
    SharedWrite(Shared, SharedRead(Var) / 2);
  end process;
end;

entity Outer is
  port ( Output : out ValueType);
end outer;

architecture Wondrousness of Outer is
  component Bleeder
    port ( Shared : inout SharedVariableType);
  end component;
  component Inner
    port (Var : inout SharedVariableType);
  end component;
  signal Var : SharedVariableType := (DefaultValue, Invalid);
begin
  P3: process
  begin
    SharedWrite(Var,15);
    wait until SharedRead(Var) = 1;
    assert FALSE report " Is wondrous" severity NOTE;
    wait for 2 NS;
    SharedWrite(Var,0);
    wait;
  end process;

  Shared: Bleeder port map (Var);
  c1: Inner port map (Var);
  Output < = SharedRead(Var);
end;
```

Listing 15: Example showing the use of the full shared variables.

5 Using Shared Variables

The example shown in the previous section is fairly simple, as it does not require atomic updates of multiple shared variables. This is due to the fact that the processes in the example algorithm only updates a single shared variable each, and that the algorithm fulfills an exclusive write condition, i.e., for no value of the shared variable state, is more than one process trying to update the same shared variable.

If a design does not fulfill these conditions, certain additional steps must be taken to avoid indeterministic behavior. If the exclusive write condition is not fulfilled, the specific timing of the processes will affect the execution of the algorithm. In this case guard conditions on the `SharedWrite` may be sufficient; an example of this is the process in listing 16, which is a variant of process `P1` from listing 15.

If atomic update of *multiple* shared variables is required, e.g., in order to exchange the contents of two shared variables, a possible solution is given in listing 17. The **wait** which updates the shared variables is no longer elided, but provided explicitly in the user's code.

```
P1: process
begin
  wait until SharedRead(Shared) mod 2 /= 0;
  wait for 5 NS;
  if SharedRead(Shared) /= 0 then
    SharedWrite(Shared, SharedRead(Shared)*3 + 1);
  end if;
end process;
```

Listing 16: The process `P1` is now also sensitive to the value 0, as is process `P3` (listing 15). In order to resolve the resulting multiple writes, a guard condition has been added on the `SharedWrite` call.

```
signal Req : SharedVariableType := (DefaultValue, Invalid);
signal Ack : SharedVariableType := (DefaultValue, Invalid);
 ...
process
begin
 ...
 Req < = VariableType' (Ack.Value,Valid);
 Ack < = VariableType' (Req.Value,Valid);
 wait for 0 NS;
 Req < = VariableType' (DefaultValue, Invalid);
 Ack < = VariableType' (DefaultValue, Invalid);
 ...
end;
```

Listing 17: Implementing a swap operation by explicit inclusion of a **wait** in the user code.

An alternative solution is to arbitrate access to a set of shared variables by using some synchronization scheme. One such scheme [2] can be based on the atomic swap implemented by the `SharedSwap` shown in listing 18.

```
procedure SharedSwap (signal SV1 : inout SharedVariableType;
                      signal SV2 : inout SharedVariableType) is
begin
  SV1 < = VariableType' (SV2.Value,Valid);
  SV2 < = VariableType' (SV1.Value,Valid);
  wait for 0 NS;
  SV1 < = VariableType' (DefaultValue, Invalid);
  SV2 < = VariableType' (DefaultValue, Invalid);
end;
```

Listing 18: The atomic swap operation for use as a synchronization primitive.

6 Conclusion

This paper demonstrates that designs utilizing shared variables can adequately be modeled in VHDL. Using the bus-resolution function of VHDL allows the mechanics of implementing shared variables to be hidden from the user.

Three different implementations of shared variables have been presented:

- Using the **register** construct of VHDL, a model for handling non-hierarchical shared variables has been given [6].
- An improved model which allows hierarchical updates was then presented; this model does not provide for read accesses within the hierarchy.
- This limitation was then lifted in the last model, which provides shared variables with full hierarchical access.

The three models correspond to different levels of requirements for shared variable behavior. Finally, some guidelines for handling atomic update of multiple shared variables has been given for the full shared variable model.

References

[1] Jens P. Brage. *Foundations of a High-Level Synthesis System*. PhD thesis, Technical University of Denmark, 1993.

[2] E.W. Dijkstra. Co-operating sequential processes. In F. Genuys, editor, *Programming Languages*. Academic Press, 1965.

[3] Douglas Hofstadter. *Gödel, Escher, Bach: An Eternal Golden Braid*. Penguin Books, 1979.

[4] The Institute of Electrical and Electronics Engineers, Inc., 345 East 47th Street, New York, NY 10017, USA. *IEEE Standard VHDL Language Reference Manual; IEEE Std. 1076 – 1992*, 1993.

[5] Alain J. Martin. Synthesis of asynchronous VLSI circuits. In J. Staunstrup, editor, *Formal Methods for VLSI Design*, chapter 6. North-Holland, 1990.

[6] Ken Scott. Anomalies in VHDL and how to address them. In Randolph E. Harr and Alec G. Stanculescu, editors, *Applications of VHDL to Circuit Design*, chapter 7, pages 197–228. Kluwer Academic Publisher, 1991.

[7] Jørgen Staunstrup. *A Formal Approach to Hardware Design*. Kluwer Academic Publisher, 1994.