MemoryGenericPkg User Guide

User Guide for Release 2024.11

By

Jim Lewis

SynthWorks VHDL Training

Jim@SynthWorks.com

http://www.SynthWorks.com

Table of Contents

1 MemoryGenericPkg Overview 3
2 Basic Usage 3
  2.1 Package References 3
  2.2 Creating a MemoryID 3
  2.3 NewID: Sizing the Memory Model 3
  2.4 MemWrite: Writing to the Memory Model 4
  2.5 MemRead: Reading from the Memory Model 4
  2.6 Putting the basic pieces together 4
3 Method Reference 5
  3.1 MemoryIDType 6
  3.2 NewID 5
  3.3 MemWrite 7
  3.4 MemRead 7
  3.5 MemErase 8
  3.6 GetAlertLogID 8
  3.7 FileReadH and FileReadB 8
  3.8 FileWriteH and FileWriteB 9
4 A Better Way of Calling NewID? 10
5 Data Structure 11
6 Compiling MemoryGenericPkg and Friends 12
7 About MemoryGenericPkg .... **Error! Bookmark not defined.**
8 Future Work .... **Error! Bookmark not defined.**
9 About the Author - Jim Lewis .... **Error! Bookmark not defined.**

1 MemoryGenericPkg Overview

The MemoryGenericPkg provides an efficient, sparse memory structure and an API that simplifies the creation of memory models. The memory data structure reduces the amount of memory used during a simulation and hence improves performance.

There are two primary instances of MemoryGenericPkg: MemoryPkg_X and MemoryPkg_NoX. MemoryPkg_X stores 0, 1, X, and U in the memory, and hence, uses two bits per memory bit. MemoryPkg_NoX stores 0 and 1 in the memory and uses 1 bit per memory bit.

MemoryPkg is currently the same as MemoryPkg_X. The rest of this document will refer to the usage of MemoryPkg. The usage of MemoryPkg_X or MemoryPkg_NoX is identical.

2 Basic Usage

2.1 Package References

Using MemoryGenericPkg requires at a minimum the following package reference:

```
use osvvm.MemoryPkg.all ;
architecture Test1 of tb is
```

That said, it is recommended to reference the OSVVM context declaration.

```
library OSVVM ;
  context osvvm.OsvvmContext ;
```

2.2 Creating a MemoryID

A MemoryID is a handle to the memory object that is created internal to MemoryPkg. MemoryID has the type MemoryIDType. MemoryIDType is a regular type and can be held in a signal or variable.

```
architecture Model of Sram is
  signal MemoryID : MemoryIDType ;
   . . .
```

2.3 NewID: Construct the Memory

NewID allocates and initializes (sizes) the memory data structure. It must be called before any read or write operations are used.

```
MemoryID <= NewID(
             Name      => SRAM'instance_name,
             AddrWidth => Address'length,
             DataWidth => Data'length) ;
```

Note, here SRAM is the entity name. If you want to separate verification components to share the same memory model, simply give them the same name, AddrWidth, and DataWidth in the call to NewID. OTOH, if you don't want to share memories between separate verification components, then use a unique name – such as is generated by ‘instance_name.

2.4 MemWrite: Writing to the Memory Model

The procedure MemWrite writes a value to a storage location in the memory data structure. If this is the first write to a block of storage elements, the block of storage elements will be allocated before the write starts.

```
MemWrite(MemoryID, Address, Data) ;
```

2.5 MemRead: Reading from the Memory Model

The function MemRead reads a value from a storage location.

```
Data <= MemRead(MemoryID, Address) ;
```

2.6 Putting the basic pieces together

The following puts together the above pieces into a basic SRAM model. Note timing in the form of output propagation delays and input timing checks have not been added to the model. While these are necessary to complete memory model they are a separate issue.

Note that both RamWriteProc and RamReadProc both use wait statements at the beginning of the process. This allows NewID, which runs during initialization to complete before either MemWrite or MemRead can run. This is essential since the memory does not exist until NewID is called.

```
library ieee ;
  use ieee.std_logic_1164.all ;
  use ieee.numeric_std.all ;
library OSVVM ;
  context osvvm.OsvvmContext ;

Entity SRAM is
port (
  Address : in     std_logic_vector (19 downto 0) ;
  Data    : inout  std_logic_vector (15 downto 0) ;
  nCE     : in     std_logic ;
  nOE     : in     std_logic ;
  nWE     : in     std_logic
) ;
end SRAM ;
Architecture model of SRAM is
  signal MemoryID : MemoryIDType ;
  signal WriteEnable, ReadEnable  : std_logic ;
begin
  MemoryID <= NewID(
                Name      => SRAM'instance_name,
                AddrWidth => Address'length,
                DataWidth => Data'length) ;

  WriteEnable <= not nWE and not nCE ;
```



```
  RamWriteProc : process
  begin
    wait until falling_edge(WriteEnable) ;
    MemWrite(MemoryID, Address, Data) ;
  end process RamWriteProc  ;

  ReadEnable <= not nCE and not nOE ;

  ReadProc : process
  begin
    wait on Address, ReadEnable ;
    case ReadEnable is
      when '1' =>       Data <=  MemRead(MemoryID, Address) ;
      when '0' =>       Data <= (Data'range => 'Z') ;
      when others =>    Data <= (Data'range => 'X') ;
    end case ;
  end process ReadProc ;
end model ;
```

3 Memory API Reference

The intent of the API is to make creation of efficient memory models easy. Further more, to make the creation of shared memories that are in separate components easy.

Behind the simple API are some rather interesting data structures. There is a brief discussion of the data structures at the end of this document. The important thing to note is that you do not need to understand the data structures to be able use MemoryPkg to create efficient memories.

3.1 NewID and Supporting Types

3.1.1 NewID: Data Structure Constructor

NewID allocates and initializes (sizes) the memory data structure. It must be called before any read or write operations are used.

```
impure function NewID (
  Name                : String ;
  AddrWidth           : integer ;
  DataWidth           : integer ;
  ParentID            : AlertLogIDType          := OSVVM_MEMORY_ALERTLOG_ID ;
  ReportMode          : AlertLogReportModeType  := ENABLED ;
  Search              : NameSearchType          := PRIVATE_NAME ;
  PrintParent         : AlertLogPrintParentType := PRINT_NAME_AND_PARENT
) return MemoryIDType ;
```

NewID also creates an AlertLogID for the data structure using the value of the name parameter and the value of ParentAlertLogID. If AddrWidth is greater than 38 bits, a warning is produced. AddrWidth is limited to being 41 bits. This creates an array that is 2**AddrWidth-BLOCK_SIZE in length.

3.1.2 **MemoryIDType**

MemoryIDType is defined as follows.

```
type MemoryIDType is record
  ID : integer_max ;
end record MemoryIDType ;
```

3.1.3 **AlertLogReportModeType**

```
type AlertLogReportModeType is (DISABLED, ENABLED, NONZERO) ;
```

When DISABLED, an AlertLogID does not print in ReportAlerts (text reports) or WriteAlertYaml (Yaml/HTML reports). When NONZERO, an AlertLogID does not print in ReportAlerts, but prints in WriteAlertYaml. Hence, NONZERO shortens the text reports to what is essential. When ENABLED, an AlertLogID prints in ReportAlerts and WriteAlertYaml.

None of these settings impact printing in Alert or Log.

DISABLED is intended to be used for data structures that are not used for self-checking and may produce errors on parameter checks. These errors are not lost as they are still accumulated in the parent ID.

3.1.4 **AlertLogPrintParentType**

```
type AlertLogPrintParentType is (PRINT_NAME, PRINT_NAME_AND_PARENT) ;
```

When PRINT_NAME, Alert and Log print the name of the AlertLogID. When PRINT_NAME_AND_PARENT, Alert and Log print the parent ID name, the ID separator, and then the ID name.

3.1.5 **NameSearchType**

```
type NameSearchType is (
    PRIVATE_NAME, NAME, NAME_AND_PARENT, NAME_AND_PARENT_ELSE_PRIVATE) ;
```

When NewID is called, first a search of existing IDs is done. If there is a match, that ID is returned, if there is not, a new one is created. Matching is determined by the ID, ParentID, and Search parameters. If the Search parameter is PRIVATE, no searching is done and a new ID is created. If the Search parameter is NAME, then any ID whose name matches is returned. If the Search parameter is NAME_AND_PARENT, then any ID whose name and ParentID match or NAME and stored Search parameter is NAME is returned.

Hence, to share a memory, construct the data structure in one entity (such as a verification component) and look up the data structure in another entity (such as the test sequencer). This is illustrated below:

```
-- Construct Data Structure in VC
entity Axi4Memory is
. . .
Initialize : process
  variable MemID : MemoryIDType;
begin
  -- construct
  MemID := NewID(
    Name       => MEM_NAME,
    AddrWidth  => ADDR_WIDTH,
    DataWidth  => DATA_WIDTH,
    Search     => NAME  ) ;
  . . .
```

```
-- Look up data structure in test sequencer
architecture Test1 of TestCtrl is
. . .

MemTestProc : process
  variable MemID : MemoryIDType;
begin
  -- look up
  MemID := NewID(
    Name       =>  MEM_NAME,
    Addrwidth  =>  ADDR_WIDTH,
    DataWidth  =>  DATA_WIDTH,
    Search     =>  NAME ) ;

  MemRead (MemID, A1, D1) ;
  MemWrite(MemID, A1, D1+5) ;
  . . .
```

3.2 MemWrite (and Write)

MemWrite writes to a memory location. If the corresponding block of storage is not yet allocated, then it will be allocated before the write.

```
procedure MemWrite (
  ID    : MemoryIDType ;
  Addr  : std_logic_vector ;
  Data  : std_logic_vector
) ;
```

If any address bit is an 'X', an alert will be signaled and the write is ignored. If any data bit is an 'X', the current policy makes the entire data word all 'X's. An alert of severity failure is generated if the memory was not already constructed with NewID.

It is an error if the Addr contains more bits than AddrWidth (the initialized size of memory) and those bits are non-zero.

3.3 MemRead (and Read)

MemRead reads from a memory location. It is available as either a function or procedure.

```
procedure MemRead (
  ID    : in MemoryIDType ;
  Addr  : in  std_logic_vector ;
  Data  : out std_logic_vector
) ;
impure function MemRead (
  ID    : MemoryIDType ;
  Addr  : std_logic_vector
) return std_logic_vector ;
```

If the address has an 'X' in it, an alert will be signaled and the value returned will be all 'X's. If corresponding block of storage is not allocated, the value returned will be all 'U's. If the block of storage is allocated, but the value has not been written to yet, the current policy returns all 'X's.   An alert of severity failure is generated if the memory was not already constructed with NewID.

It is an error if the Addr contains more bits than AddrWidth (the initialized size of memory) and those bits are non-zero.

3.4 **MemErase**

MemErase deallocates all block of storage blocks. However, it does not deallocate the base structure of the memory.   MemErase is useful for erasing the memory between tests.

```
procedure MemErase (ID : in MemoryIDType) ;
```

3.5 **GetAlertLogID**

GetAlertLogID returns the AlertLogID associated with a memory.

```
impure function GetAlertLogID (ID : in MemoryIDType) return AlertLogIDType ;
```

3.6 **FileReadH and FileReadB**

FileReadH and FileReadB are designed to mimic Verilog system functions $readmemh and $readmemb.

```
procedure FileReadH (    -- Hexadecimal File Read
  ID           : MemoryIDType ;
  FileName     : string ;
  StartAddr    : std_logic_vector ;
  EndAddr      : std_logic_vector
) ;
procedure FileReadH (
  ID           : MemoryIDType ;
  FileName     : string ;
  StartAddr    : std_logic_vector
) ;
procedure FileReadH (
  ID           : MemoryIDType ;
  FileName     : string
) ;

procedure FileReadB (    -- Binary File Read
  ID           : MemoryIDType ;
  FileName     : string ;
  StartAddr    : std_logic_vector ;
  EndAddr      : std_logic_vector
) ;
procedure FileReadB (
  ID           : MemoryIDType ;
  FileName     : string ;
  StartAddr    : std_logic_vector
) ;
procedure FileReadB (
  ID           : MemoryIDType ;
  FileName     : string
) ;
```

The file shall contain only of the following:

- White space (spaces, new lines, tabs, and form-feeds)
- Comments (either //, #, or --)
- @ character (designating the adjacent number is an address)
- Binary or hexadecimal numbers

Addresses specified in the call to FileReadH or FileReadB provide both an initial starting address and a range of valid addresses for memory operations. Addressing advances from StartAddr to FinishAddr. If FinishAddr is greater than StartAddr, then the next address is one larger than the current one, otherwise, the next address is one less than the current address.

Addresses may also be specified in the file in the format '@' followed by a hexadecimal number as shown below. There shall not be any space between the '@' and the number. The address read must be between StartAddr and FinishAddr or a FAILURE is generated.

```
@hhhhh
```

Values not preceded by an '@' character are data values. Data values must be separated by white space or comments from other values. ReadMemH requires hexadecimal values compatible with hread for std_ulogic_vector. ReadMemB requires values compatible with read for std_ulogic.

In the current implementation, if more digits are read than are required by the memory, left hand characters will be dropped provided they are 0. If fewer digits are read than are required by the memory, 0 characters are added to the left hand side. Note that this behavior may change in the future to support stricter conformance between data in the file and the data width of the memory.

3.7 **FileWriteH and FileWriteB**

FileWriteH and FileWriteB are designed to mimic Verilog system functions $writememh and $writememb.

```
procedure FileWriteH (    -- Hexadecimal File Write
  ID           : MemoryIDType ;
  FileName     : string ;
  StartAddr    : std_logic_vector ;
  EndAddr      : std_logic_vector
) ;
procedure FileWriteH (
  ID           : MemoryIDType ;
  FileName     : string ;
  StartAddr    : std_logic_vector
) ;
procedure FileWriteH (
  ID           : MemoryIDType ;
  FileName     : string
) ;
```

```
procedure FileWriteB (    -- Binary File Write
  ID           : MemoryIDType ;
  FileName     : string ;
  StartAddr    : std_logic_vector ;
  EndAddr      : std_logic_vector
) ;
procedure FileWriteB (
  ID           : MemoryIDType ;
  FileName     : string ;
  StartAddr    : std_logic_vector
) ;
procedure FileWriteB (
  ID           : MemoryIDType ;
  FileName     : string
) ;
```

Address and data values are written one per line. Address values are preceded by an @ character. Values that are X or values that correspond to a block of storage elements that have not been allocated will not be written out.

4 Generics and Instances of MemoryGenericPkg

4.1 Generic Interface

The generic interface for MemoryGenericPkg is shown below.

```
package MemoryGenericPkg is
  generic (
--    type MemoryBaseType ;
    function SizeMemoryBaseType(Size : integer) return integer ; -- is <> ;
    function ToMemoryBaseType  (A : std_logic_vector ; Size : integer) return
MemoryBaseType ; -- is <> ;
    function FromMemoryBaseType(A : MemoryBaseType   ; Size : integer) return
std_logic_vector ; -- is <> ;
    function InitMemoryBaseType(Size : integer) return MemoryBaseType -- is <>
  ) ;
```

MemoryBaseType is currently a subtype of integer_vector. The original intent was to use either integer_vector or bit_vector but that was too much at the bleeding edge of tool support.

4.2 Instances of MemoryGenericPkg

There are two primary instances of MemoryGenericPkg: MemoryPkg_X and MemoryPkg_NoX. MemoryPkg_X stores 0, 1, X, and U in the memory, and hence, uses two bits per memory bit.

```
use work.MemorySupportPkg.all ;
package MemoryPkg_X is new work.MemoryGenericPkg
  generic map (
--    MemoryBaseType      => MemoryBaseType_X,
    SizeMemoryBaseType  => SizeMemoryBaseType_X,
    ToMemoryBaseType    => ToMemoryBaseType_X,
    FromMemoryBaseType  => FromMemoryBaseType_X,
    InitMemoryBaseType  => InitMemoryBaseType_X
```

```
  ) ;
```

MemoryPkg_NoX stores 0 and 1 in the memory and uses 1 bit per memory bit.

```
use work.MemorySupportPkg.all ;
package MemoryPkg_NoX is new work.MemoryGenericPkg
  generic map (
--    MemoryBaseType      => MemoryBaseType_NoX,
    SizeMemoryBaseType  => SizeMemoryBaseType_NoX,
    ToMemoryBaseType    => ToMemoryBaseType_NoX,
    FromMemoryBaseType  => FromMemoryBaseType_NoX,
    InitMemoryBaseType  => InitMemoryBaseType_NoX
  ) ;
```

The storage policies of these packages is defined by the type MemoryBaseType (currently fixed at integer_vector) and the generic functions, SizeMemoryBaseType, ToMemoryBaseType, FromMemoryBaseType, and InitMemoryBaseType.

There is also a package instance named MemoryPkg_orig. This implements the "original" storage policies that MemoryPkg used. The storage policy for this package uses a single integer value and uses the left most bit of the integer to indicate whether the value is an X or not. As a result, only 31 bits can be stored in the memory and if an X is stored in any bit, they all read back as an X. Its usage is not recommended – instead use MemoryPkg_X or MemoryPkg_NoX.

5 A Better Way of Calling NewID? Not yet.

As an ordinary type, MemoryID could be a constant and NewID called in the constant declaration as follows:

```
constant MemoryID : MemoryIDType :=
           NewID(
             Name       =>  Axi4Memory'instance_name,
             Addrwidth  =>  Addr'length,
             DataWidth  =>  Data'length ) ;
```

One big benefit of this is that NewID is called during elaboration of the design before MemRead or MemWrite run.

While the tools we test with allow this, unfortunately the language does not. We plan to address this in the next revision of the language. See: https://gitlab.com/IEEE-P1076/VHDL-Issues/-/issues/75

6 Data Structure

MemoryGenericPkg contains a singleton data structure that allows a simple API to be created to a memory data structure. The singleton data structure allows a dynamic array of memory objects to be created. Don't worry the complexity of this is hidden from users of the package.

When NewID is called, an additional element is added to the dynamic array. This element implements a sparse memory data structure. NewID returns a value of MemoryIDType. This value is the handle to the memory.

The sparse memory data structure used by each element in the dynamic array of memories consists of an array of pointers to a block of storage elements. A block of storage elements is nominally an array 1024 values.   This structure is shown in Figure 1.

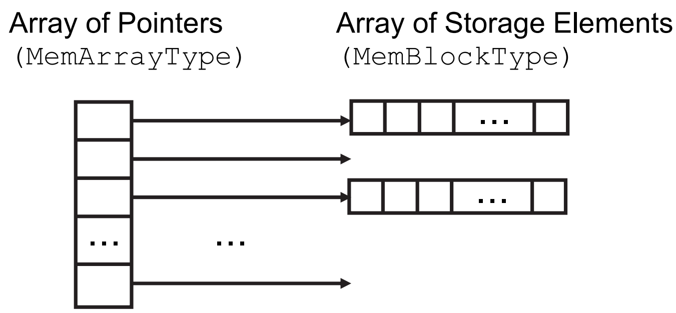

Figure 1. Allocation Structure of MemoryGenericPkg

The Array of Pointers is created when the memory is initialized by the call to NewID. A block of storage is allocated when a write operation writes to a location within the block. This data structure minimizes workstation memory usage when only a portion of the RAM model is used.

Currently storage elements are type integer.   Storing a std_logic_vector into an integer requires a policy. The current storage policy is that if there is an 'X' in any bit of the std_logic_vector, then the value is stored as -1. An uninitialized value is integer'left. As a result, that allows up to 31 bits of std_logic_vector to be stored into a storage element. The future plan is for the storage element type and storage policy to be determined by package generics.

7 What about the older revisions of MemoryPkg?

Versions of MemoryPkg before 2021.06 were solely protected type based.   That use model is still supported, however, it is deprecated and not talked about here.   If you need the documentation for it, see the subdirectory ProtectedTypes_the_old_way in the Documentation repository.

8 About MemoryGenericPkg

MemoryGenericPkg is part of the OSVVM Utility library.

MemoryGenericPkg was created by Jim Lewis of SynthWorks.   Please support our work by buying your VHDL Training from SynthWorks.

MemoryGenericPkg.vhd is a work in progress and will be updated from time to time. Caution, undocumented items are experimental and may be removed in a future version.

9 Compiling and Using OSVVM Utility Library

Reference all packages in the OSVVM Utility library by using the context declaration:

```
library OSVVM ;
  context osvvm.OsvvmContext ;
```

Compilation order for OSVVM Utility Library is in OSVVM_release_notes.pdf. Rather than learning this, we recommend using the OSVVM compilation scripts.

OSVVM Utility library is released under the Apache open source license. It is free (both to download and use - there are no license fees). You can download it from osvvm.org or from our development area on GitHub (https://github.com/OSVVM/OSVVM).

If you add features to the package, please donate them back under the same license as candidates to be added to the standard version of the package. If you need features, be sure to contact us.

We also support the OSVVM user community and blogs through http://www.osvvm.org. Interested in sharing about your experience using OSVVM? Let us know, you can blog about it at osvvm.org.

10 About the Author - Jim Lewis

Jim Lewis, the founder of SynthWorks, has thirty plus years of design, teaching, and problem solving experience. In addition to working as a Principal Trainer for SynthWorks, Mr Lewis has done ASIC and FPGA design, custom model development, and consulting.

Mr. Lewis is chair of the IEEE 1076 VHDL Working Group (VASG) and is the primary developer of the Open Source VHDL Verification Methodology (OSVVM.org) packages. Neither of these activities generate revenue.

Please support our volunteer efforts by buying your VHDL training from SynthWorks.

If you find bugs these packages or would like to request enhancements, you can reach me at jim@synthworks.com.