MemoryPkg User Guide

User Guide for Release 2016.01

By

Jim Lewis

SynthWorks VHDL Training

Jim@SynthWorks.com

http://www.SynthWorks.com

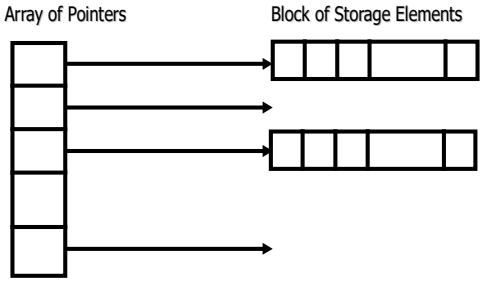
Table of Contents

1 MemoryPkg Overview

The MemoryPkg provides a protected type and methods to that both simplify the creation of memory models and make them efficient with respect to memory usage.

2 Data Structure

MemoryPkg stores values in a sparse memory data structure. The data structure consists of an array of pointers to a block of storage elements. A block of storage elements is nominally an array 1024 values. This is shown below.



The Array of Pointers is created when the memory is initialized and the size is set. 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.

3 Basic Usage

3.1 Package References

Using MemoryPkg requires the following package references:

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

3.2 Declaring the Memory Object

Memory is modeled using a data structure stored inside of a memory object. The memory object is created by declaring a shared variable of type MemoryPType, such as ptRam shown below.

```
architecture Model of Sram is
 shared variable ptRam : MemoryPType ;
 . . .
```

```
begin
```

3.3 MemInit: Sizing the Memory Model

Memory models are not sized by the package itself. Instead, the memory model is sized dynamically using the method MemInit.

```
ptRam.MemInit(AddrWidth => Address'length,  DataWidth => Data'length) ;
```

3.4 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.

```
ptRam.MemWrite(Address, Data) ;
```

3.5 MemRead: Reading from the Memory Model

The function MemRead reads a value from a storage location.

```
Data <= ptRam.MemRead(Address) ;
```

3.6 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 MemInit, which runs during initialization to complete before either MemWrite or MemRead can run. This is essential since the memory is not initially sized.

```
library ieee ;
  use ieee.std_logic_1164.all ;
  use ieee.numeric_std.all ;

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
  shared variable ptRam : MemoryPType ;
  signal WriteEnable, ReadEnable  : std_logic ;
begin
  ptRam.MemInit(AddrWidth => Address'length,  DataWidth => Data'length) ;

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

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

  ReadEnable <= not nCE and not nOE ;

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

4 Method Reference

4.1 MemInit

The size and width of the memory is set by MemInit.

```
procedure MemInit ( AddrWidth, DataWidth  : in  integer ) ;
```

4.2 MemWrite

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 ( Addr, Data  : in  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.

4.3 MemRead Procedure

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

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

4.4 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 ;
```

4.5 Deallocate

Deallocates deallocates all memory allocated by a memory model. After calling Deallocate, MemInit must be called before the memory is usable again.

```
procedure deallocate ;
```

4.6 SetAlertLogID

MemoryPkg uses the AlertLogPkg to generate errors. By default, it will report errors to OSVVM_ALERTLOG_ID. SetAlertLogID allows your memory models to have any errors reported to be associated with a particular memory model.

```
procedure SetAlertLogID (A : AlertLogIDType) ;
procedure SetAlertLogID (
  Name : string ;
  ParentID : AlertLogIDType := ALERTLOG_BASE_ID ;
  CreateHierarchy : Boolean := TRUE
) ;
```

4.7 GetAlertLogID

GetAlertLogID returns the AlertLogID associated with a memory model.

```
impure function GetAlertLogID return AlertLogIDType ;
```

4.8 FileReadH and FileReadB

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

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

4.9 FileWriteH and FileWriteB

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

```
procedure FileWriteH (    -- Hexadecimal File Read
      FileName     : string ;
      StartAddr    : std_logic_vector ;
      EndAddr      : std_logic_vector
) ;
procedure FileWriteH (FileName : string ;  StartAddr : std_logic_vector) ;
procedure FileWriteH (FileName : string) ;
procedure FileWriteB (    -- Hexadecimal File Read
      FileName     : string ;
      StartAddr    : std_logic_vector ;
      EndAddr      : std_logic_vector
) ;
procedure FileWriteB (FileName : string ;  StartAddr : std_logic_vector) ;
procedure FileWriteB (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.

5 Compiling MemoryPkg and Friends

Use of MemoryPkg.vhd requires use TextUtilPkg.vhd, TranscriptPkg.vhd, and AlertLogPkg.vhd. The compile order is: TextUtilPkg.vhd, TranscriptPkg.vhd, AlertLogPkg.vhd, and MemoryPkg.vhd. Compiling the packages requires VHDL-2008.

6 About MemoryPkg

MemoryPkg was developed and is maintained by Jim Lewis of SynthWorks VHDL Training. While it is new to OSVVM, it has been part of SynthWorks' VHDL training classes since 2006.

Please support our effort in supporting MemoryPkg and OSVVM by purchasing your VHDL training from SynthWorks.

MemoryPkg is free (both to download and use - there are no license fees). It is released under the Perl Artistic open source license. You can download it from either osvvm.org or http://www.synthworks.com/downloads. It will be updated from time to time.

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. I blog about the packages at http://www.synthworks.com/blog. We also support the OSVVM user community and blogs through http://www.osvvm.org.

Find any innovative usage for the package? Let us know, you can blog about it at osvvm.org.

7 Future Work

MemoryPkg.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.

8 About the Author - Jim Lewis

Jim Lewis, the founder of SynthWorks, has twenty-eight 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.