# UVM's MAM to the Rescue

Michael Baird
Willamette HDL, Inc.
Beaverton, OR USA
mike@whdl.com

***Abstract-*** **A UVM Register Model provides an abstract representation of both registers and memories in a hardware block. Memories are mapped with a base address and a range representing its size. Memory accesses through the register model are then done via an offset from the base address. This works well for simple random access to memory. However many designs use local memory for temporary buffers for i/o or other activities. To avoid clashes in these types of designs, the buffer locations must be managed carefully. UVM provides a set of utility classes collectively know as the Memory Access Manager (MAM) to ease this burden. This paper discusses the use of the MAM to test a 32-channel DMA controller and the results.**

## I. Introduction

A Universal Verification Methodology (UVM) Register Model provides an abstract representation of both registers and memories in a hardware block. The register model provides for two major conveniences for registers, mirroring of registers and translation from an abstract register name to a physical address. While registers are mirrored in the register model memories are not. Memories are mapped with a base address and a range representing its size. Memory accesses through the register model are then done via an offset from the base address.

This works well for simple random access to memory. However certain designs use local memory for temporary buffers for i/o or other activities. Examples might be a LAN controller buffering packets in memory, or a DMA controller. To avoid clashes in these types of designs, the buffer locations must be managed carefully. This can be yet another burdensome and sometimes error prone chore for the verification engineer.

UVM provides a set of utility classes collectively know as the Memory Access Manager (MAM) to ease this burden. The MAM provides the ability to allocate and manage buffers without overlap or knowledge of the exact locations of the buffers. This provides flexibility for when designs change and portability to other designs. This paper describes what capability the MAM provides and discusses the use of the MAM to test a 32-channel DMA controller and the results.

## II. Memory

A memory is modeled by extending the base class uvm_mem. A memory is declared in a register model with a base address which is an offset determined by the register map in the register block which contains the memory. Memories may be mapped to multiple buses using multiple maps within the block where the memory is declared. The specification of the memory includes a width declared in bits, a range which is the size of the memory declared in

terms of the number of bytes and an access type which is one of Read-write (RW), Read-only (RO) or Write-only (WO). Unlike registers, memories are not mirrored in the register model because of memory usage in the simulator.

A location in the memory is accessed with an offset from the base address of the memory. You could think of the offset as the "local" address within the memory. Using this offset or local address is useful as the base address of the memory may be changed as the design matures or there may be multiple instances of a memory used in the design resulting in multiple different base addresses for memories in the design. Changes to the base address do not affect the local address and hence do not affect the verification code that accesses the memories.

There are six types of accesses supported for memories.

- *read(), write()*. Single location access either through the front-door or back-door. Address is the offset within the memory
- *peek(), poke()*. Single location access through the backdoor. Address is the offset within the memory.
- *burst_read(), burst_write()*. Block access of contiguous addresses. Address is the offset within the memory for the starting location of the block access.

## III. Memory Access Manager

Many designs use temporary buffers within a local memory for i/o or other activities. An example is the source and destination buffers used by a DMA controller. This example is illustrated later in this paper. Another example is a Media Access Controller (MAC) which uses temporary buffers in memory to store incoming and outgoing Ethernet frames. The allocation and deallocation of memory buffers, as well as abstracting their location details (starting address, ending address etc.) within the test code is yet another burdensome "housekeeping" chore for a verification engineer. The UVM Register Model provides a set of utility classes collectively known as the Memory Allocation Manager (MAM) to ease this burden.

The MAM lets us request a contiguous range of memory of a specific size. The MAM allocates a region of the size requested within the specified memory. We can then treat that region like an independent memory, using a zero-based address for accesses. Operations done on the MAM region are automatically translated into corresponding operations to/from the actual memory by the MAM. This allows us to manipulate the memory regions or buffers without worrying about where they actually reside in the bus address space.

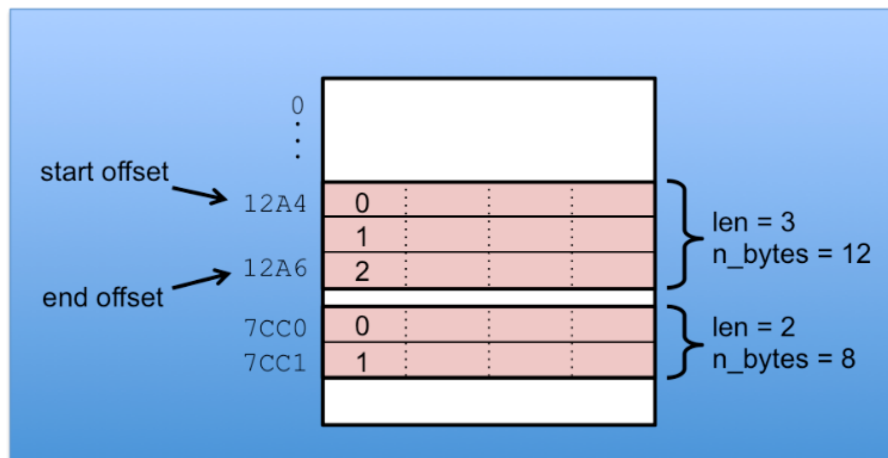

Figure 1. MAM example with 2 allocated regions

You may interact with the MAM in a simple high-level manner or with much more detailed control of the features.

## IV. 4. MAM Utility Classes

**uvm_mem_mam**

The *uvm_mem_mam* is a MAM utility class that provides methods for requesting and releasing memory regions. The base class *uvm_mem* has a property mam of type *uvm_mem_mam*. Since memories in the Register Model are derived from *uvm_mem*, the use of the MAM is available on any memory object.

***uvm_mem_mam* API**

- *function uvm_mem_region* ***request region****(int unsigned n_bytes, uvm_mem_mam_policy alloc = null, string fname = "", int lineno = 0)****;***

  Creates a region of size *n_bytes* with the start address determined by the *alloc* argument. If the default of *null* is used, the region location is randomly allocated. Returns an *uvm_mem_region* object that is described in the next section.

  You can change the allocation of the buffers from a "random" scheme (the default) to some other scheme, such as packing the buffers, by deriving your own *uvm_mem_mam_policy* object for use when requesting buffers.

- *function uvm_mem_region* ***reserve_region*** *(bit [63:0] start_offset, int unsigned n_bytes, string fname = "", int lineno = 0)****;***

  Creates a region of size *n_bytes* at a fixed location with the start address of *start_offset* from the base address of the memory. Returns an *uvm_mem_region* object.

- *function void* ***release_region****(uvm_mem_region region);*

  Releases a previously allocated memory region. When you are done with a region you should release it as this allows the memory address range to be available for subsequent requests. There is no automatic garbage collection of regions as is the case with SystemVerilog classes.

- *function void* ***release_all_regions****(uvm_mem_region region);*

  Forcibly release all allocated memory regions.

- *function uvm_mem_region* ***for_each****(bit reset = 0);*

  Iterate over all the currently allocated regions.

- *function uvm_mem* ***get_memory****();*

Get the memory where the region resides.

Example usage:

```
uvm_mem mem0 = system_block.mem0;
// request buffers, size in BYTES
uvm_mem_region src_buf = mem0.mam.request_region(num_words*4);
uvm_mem_region dst_buf = mem0.mam.request_region(num_words*4);
```

```
...
mem0.mam.release_region(src_buf);  // release src_buf region
mem0.mam.release_all_regions();    // release all regions
...
// reserve buffers
uvm_mem_region src_buf = mem0.mam.reserve_region(32'h1000,
                                                  num_words*4);
uvm_mem_region dst_buf = mem0.mam.reserve_region(32'h3000,
                                                  num_words*4);
...
mem0.mam.release_all_regions(); // release all regions
```

**uvm_mem_region**

For each allocated memory region there is an *uvm_mem_region* descriptor object created. The *request_region()* and *reserve_region()* methods of the *uvm_mem_mem* class return the descriptor object. Access to the allocated memory region is done by calling the API methods of the descriptor object.

***uvm_mem_region* API**

Access methods. Same as memory access methods:

- *read(), write()*, *peek(), poke()*, *burst_read(), burst_write()*.

For each allocated memory region there is a *uvm_mem_region* descriptor object created. The *request_region()* and *reserve_region()* methods of the *uvm_mem_mem* class return the descriptor object. Access to the allocated memory region is done by calling the API methods of the descriptor object.

```
uvm_mem_region src_buf = mem0.mam.request_region(num_words*4);
...
src_buf.write(status, 1, 32'h1234);
src_buf.read(status, 1);
```

Introspection methods

- *get_start_offset()*. Get the start offset address relative to the memory.
- *get_end_offset()*. Get the end offset address relative to the memory.
- *get_len()*. Get the size of the region in consecutive locations (not necessarily bytes).
- *get_n_bytes()*. Get the size of the region in bytes.
- *get_memory()*. Returns the handle to the *uvm_mem* object where the region resides.

## V. Example Circuit Device Under Test and Bus Structure

The diagram below illustrates the example circuit used to illustrate the MAM. The Device Under Test (DUT) is a 32-channel DMA controller, the WISHBONE DMA/Bridge IP core from opencores.org. The chip is implemented with a WISHBONE bus interface, which is defined at opencores.org. The DMA chip is both a master and a slave on the WISHBONE bus. It acts as a master for DMA transfers and as a slave for access to its registers.

Additional devices on the WISHBONE bus are two instances of the WISHBONE Slave Memory from opencores.org. They are both configured as 1Mbyte slave memories.

The WISHBONE bus itself is a fairly simple System on Chip (SoC) bus. It supports single reads and writes, block reads and writes and read-modify-write operations. The implementation for the example circuit has a straight forward non-prioritized interrupt structure, is implemented with synchronous handshaking, with 32-bit address and 32-bit data.

The WISHBONE bus interface is a Bus Functional Model (BFM) style interface with methods for doing bus transactions and bus monitoring. A separate interface for the reset logic is not shown in the diagram below.

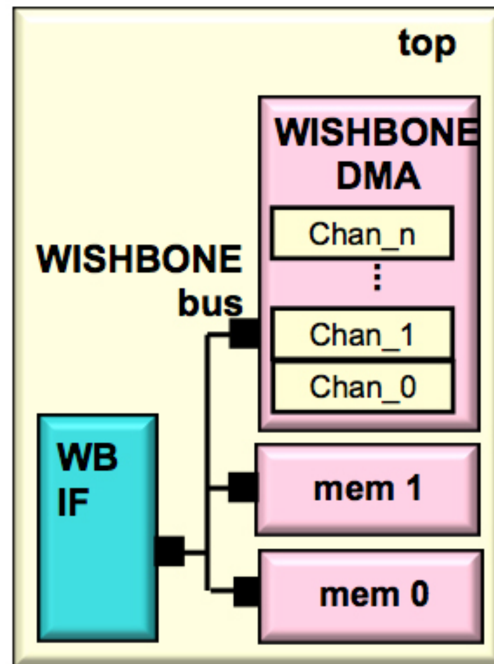


Figure 2. DMA DUT with Slave Memories

The DMA DUT has a number of registers that must be first initialized (interrupt status, interrupt mask etc.) before a DMA transfer may be initiated. A DMA transfer is set in the DUT by writing a number of registers to set the source buffer location, destination buffer location, size of transfer and so forth in a particular channel. The DMA is initiated by writing a Channel Status Register (CSR). The DMA DUT is capable of 32 simultaneous DMAs.

## VI. 6. Example Circuit Testbench

The example circuit testbench is illustrated in the diagram below. The UVM testbench consists of WISHBONE bus agent with a driver, monitor and sequencer. Additionally the agent has "interface" or "API" sequences for generating WISHBONE read and write transactions. The WISHBONE environment encapsulates:

- WISHBONE bus agent
- Scoreboard that mirrors the slave memories
- WISHBONE bus adapter which is used to translate between the WISHBONE bus specific transactions and the register model generic transactions
- Predictor, which updates the register database, based upon WISHBONE bus transactions broadcast by the monitor.

A base test class (*test_dma_base*) creates and instantiates the register model and distributes the handle of the register model to the testbench components and sequences that access the register model. Distribution was via the configuration database. Multiple tests were generated that were derived from the base test class *test_dma_base*. These tests used a number of different sequences for testing the DUT.

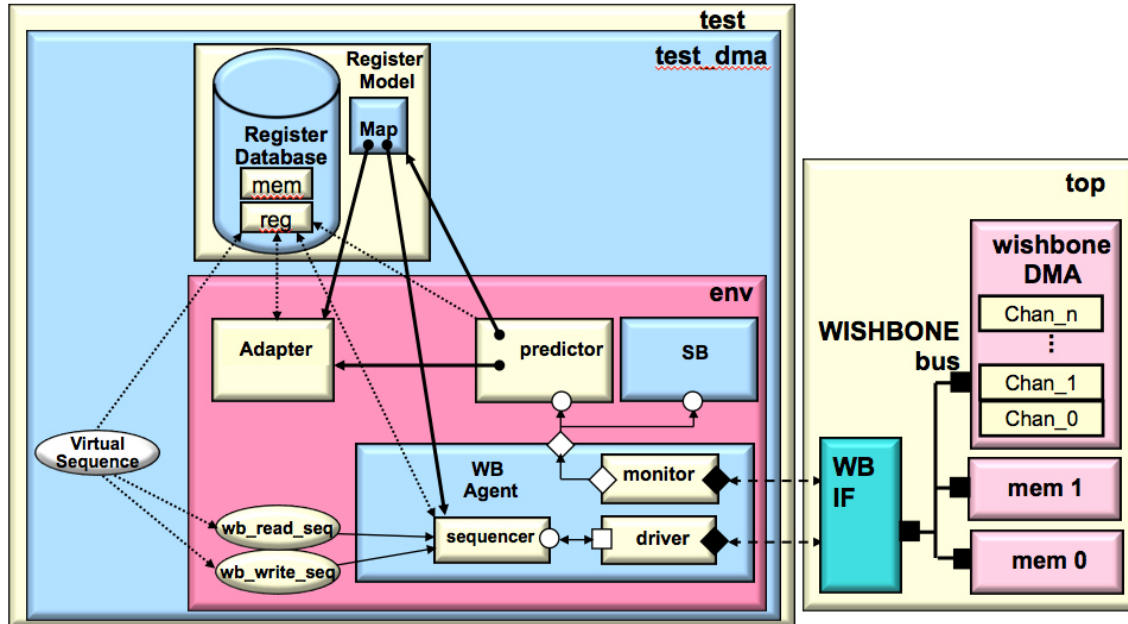


Figure 3. UVM DMA Testbench

## VII. NON MAM BASED TESTS

This section will describe the original non-MAM implementation, pointing out the flaws. The next section will show a MAM-based approach.

Initially the buffers for the DMA transfers were set up and managed "by-hand". The code below shows the *dma_transfer()* task. This task requires four inputs:

1. DMA channel number.
2. Source buffer's start address (absolute address).
3. Destination buffer's start address (absolute address).
4. Size of the DMA transfer (in 32-bit words) .

The body of the task illustrates the steps in a by-hand setup of a DMA transfer.

1. Initialize data for the source buffer.
2. Initialize the source buffer. Note this does not use the register model but accesses the memory directly through the bus agent.
3. Set up the DMA channel registers and start the DMA by writing the CSR register. Note that the source and destination addresses are supplied to the *dma_setup()* task
4. Wait for an interrupt from the DMA chip and service the interrupt
5. Verify the data in the destination buffer is correct.

```
class dma_simple_seq extends wb_dma_reg_base_seq;
. . .
// cause a dma transfer on a channel
virtual task dma_transfer(
              int chan_num,  // channel number
              int srce_addr, // source buffer addr
              int dest_addr, // dest buffer addr
              int tot_sz, //size of transfer, 32-bit words
              bit[2:0] ch_priority = 0, // channel priority
              int chk_sz = 0
              );
  uvm_status_e txn_status;
  init_srce_data(tot_sz); //  initialize data for source buffer
  // initialize the source buffer (does not use register model)
  wb_blk_write(srce_addr, srce_data, tot_sz,txn_status);
  // Set up the registers in DMA channel and start DMA
  dma_setup(chan_num, srce_addr, dest_addr, tot_sz);
  wait_for_dma_complete(chan_num); // wait for interrupt
  // verify the data in the destination buffer
  check_data(dest_addr, tot_sz);
endtask
. . .
```

Even with relatively few transfers, manually setting up the buffer addresses for each transfer quickly becomes quite tedious. In the code below, inside the *body()* task, the *dma_transfer()* task (shown in the code above) is called with hand generated source and destination buffer addresses. Simply changing the size of the transfers can be tedious because checks need to be made to make sure there are no buffer overlaps or buffers that spill beyond the end of the memory space. Introducing concurrent transfers, as shown below with the *fork-join* construct, further acerbates the problem, as now buffer overlap must be meticulously checked.

Assigning the base addresses of memories to variables that are set from configuration information and expressing the buffer locations as an offset from the base can mitigate the issue of portability of source code.

```
class dma_simple_seq extends wb_dma_reg_base_seq;
. . .
// cause a dma transfer on a channel
virtual task dma_transfer(
              int chan_num,  // channel number
              int srce_addr, // source buffer addr
              int dest_addr, // dest buffer addr
              int tot_sz, //size of transfer, 32-bit words
              bit[2:0] ch_priority = 0, // channel priority
              int chk_sz = 0
              );
  uvm_status_e txn_status;
  init_srce_data(tot_sz); //  initialize data for source buffer
  // initialize the source buffer (does not use register model)
  wb_blk_write(srce_addr, srce_data, tot_sz,txn_status);
  // Set up the registers in DMA channel and start DMA
  dma_setup(chan_num, srce_addr, dest_addr, tot_sz);
  wait_for_dma_complete(chan_num); // wait for interrupt
```

```
  // verifiy the data in the destination buffer
  check_data(dest_addr, tot_sz);
endtask
. . .
```

## VIII. USING THE MAM IN TESTS

This section describes using the MAM for managing the DMA buffers. The code below shows the *dma_transfer()*task. This task requires two inputs:

1. DMA channel number.
2. Size of the DMA transfer (in 32-bit words) .

Note that in this approach (using the MAM) the dma_transfer task does *not* require the addresses for the source and destination buffers as inputs. This is a significant difference between using the MAM and the by-hand approach described in the previous section.

The code in the body of the task does the following:

1. Declare *uvm_mem_region* handles for each of the buffers
2. Request a memory region for the source buffer; checking to ensure a region was allocated.
3. Request a memory region for the destination buffer; checking to ensure a region was allocated.
4. Initialize data for the source buffer.
5. Initialize the source buffer.
6. Set up the DMA channel registers and start the DMA by writing the CSR register. Note that the source and destination addresses are supplied to the *dma_setup()* task using the MAM introspection method *get_start_offset()* to retrieve the buffer offset in the memory.
7. Wait for an interrupt from the DMA chip and service the interrupt.
8. Verify the data in the destination buffer is correct.
9. Release the buffers.

Note that the by-hand approach actually has four fewer steps in the *dma_transfer()* task. These extra four steps in the MAM approach, however, are automation steps, which eliminate the by-hand address calculations for the buffers. Worth it!

```
class dma_simple_reg_mem_seq extends wb_dma_reg_base_seq;
. . .
// cause a dma transfer on a channel
virtual task dma_transfer(
            int chan_num,  // channel number
            int tot_sz,    // size of transfer, 32-bit words
            bit[2:0] ch_priority = 0, // channel priority
            int chk_sz = 0
            );
  uvm_status_e txn_status;
  uvm_mem_region src_buf, dest_buf; //buffer descriptor handles

  // Ask the MAM for a block of memory inside mem0
  src_buf  = mem0.mam.request_region(tot_sz*4);
```

```
  if(src_buf == null) // check for buffer not allocated
    `uvm_error("DMA", "DMA buffer allocation error")
  // Ask the MAM for a block of memory inside mem1
  dest_buf = mem1.mam.request_region(tot_sz*4);
  if(dest_buf == null) // check for buffer not allocated
    `uvm_error("DMA", "DMA buffer allocation error")

  init_srce_data(tot_sz); //  initialize data for source buffer
  // Seed the source buffer
  src_buf.burst_write(txn_status, 0, srce_data);

  // Perform the DMA transfer
  dma_setup(chan_num,
     // use introspection method get_start_offset()
     // to get offset address of the buffer (start address)
     mem0.get_address(src_buf.get_start_offset()),
     mem1.get_address(dest_buf.get_start_offset()),
     tot_sz);

  wait_for_dma_complete(chan_num);
  // Verify that the destination buffer data matches the source
  check_data(dest_buf, tot_sz);
  // release buffers
  mem0.mam.release_region(src_buf);
  mem1.mam.release_region(dest_buf);
endtask
. . .
```

Below is a snippet from the simulator output when simulating the above code. It shows the message output from the *request_region()* method illustrating the random address location of the allocated buffers.

```
UVM_INFO @ 830: reporter [RegModel] Attempting to reserve
['h0000000000017a8a:'h0000000000017a91]...

UVM_INFO @ 830: reporter [RegModel] Attempting to reserve
['h000000000001b554:'h000000000001b55b]...
```

Taking an approach that uses the MAM for allocating DMA transfer buffers leads to the ability to quickly and with considerably fewer headaches, develop more complex tests, and tests that are more portable. In doing buffer allocation and management by-hand, the base address of the buffer and the size of the buffer are the two main factors that determine where buffers are allocated. Using the MAM, the code developer does not need to take the base address into account, only size of the transfer. However, in some sense the size of the transfer only matters relative to is there enough buffer space to allocate the buffer.

In the code below, inside the *body()* task, the *dma_transfer()* task (shown in the code above) is called without the need to supply buffer addresses as was the case in the by-hand code. Introducing concurrent transfers, as shown below with the *fork-join* construct does not complicate the buffer allocation from the test writer's point of view. The only real concern could be running out of buffer space with too many concurrent DMA transfers. Changing the size of the

transfers can be done simply by changing the size value on the *dma_transfer()* call without regard to buffer overlap issues, as was the case in the by-hand approach.

```
task body;
  super.body();
  // do dma transfers
  dma_transfer(0, // channel number
                8  // transfer size in 32-bit words
                );
  // do transfers in parallel
  fork
    dma_transfer(1, 16);
    dma_transfer(2, 100);
    . . .
  join
  dma_transfer(3, 8);
endtask
```

## IX. Other MAM benefits

Besides the avoidance of managing buffer locations by-hand there are several other benefits to using the MAM.

**Portability of sequence code.** In the by-hand approach the base address of the memories is required in order to make the buffer address calculation. The MAM approach only requires a handle to the register model memory where in the buffer is to be allocated. This leads to more portable code in the sequences.

**Randomization.** In the by-hand approach using randomization for the size and number of buffers is problematic. Using MAM randomization of both the size and number of buffers is easily supported. In the example code shown above in section 8 we could randomize the DMA channel numbers, the size of the DMA transfers and even the number of concurrent DMA transfers.

## X. Conclusions

UVM's MAM comes to the rescue of the verification engineer when testing designs that use local buffers such as the DMA controller illustrated in this paper. Managing the allocation of these buffers can be tedious, time-consuming, and error prone. Using MAM to the manage buffer allocation results in tests that are more portable, easier to maintain, easier to change, less error prone and much less time consuming to manage than the by-hand approach. Using MAM enables tests to be more quickly created and scaled.

## References


[1] Baird, Mike and Schwartz, Kurt. Introduction to UVM Student Guide. Portland: Willamette HDL Inc. 2014 (www.whdl.com)
[2] Baird, Mike and Schwartz, Kurt. Advanced UVM Student Guide. Portland: Willamette HDL Inc. 2014 (www.whdl.com)
[3] Universal Verification Methodology (UVM) User's Guide. Accellera www.accellera.org