

USING the VMM MEMORY ALLOCATION MANAGER FOR DMA VERIFICATION

Author

Syed Sheeraj Ghouse, Architect
SASKEN Communication Technologies Limited, Bangalore

Abstract

In DMA-based designs, the proper functionality revolves around the correctness in allocation of memory space for DMA Descriptors and DMA buffers. This correctness is critical in ensuring the valid functionality of DMA-based designs. To verify such designs, one might choose a static allocation approach. However, this approach lacks the randomness in the address values and might miss some of the critical bugs in the design. The VMM's Memory Allocation Manager (MAM) provides a solution for this problem. MAM adds the randomization capabilities to the buffer, descriptor allocation in verification and thus may uncover additional bugs in the design.

This paper walks the user through the structure and components of MAM. The specific use case scenarios of this application in the context of DMA verification are described. This paper also explains the customizations done to get some custom features. It concludes with the advantages of using VMM MAM and highlights the other areas where MAM can be used.

Keywords: VMM, MAM, DMA

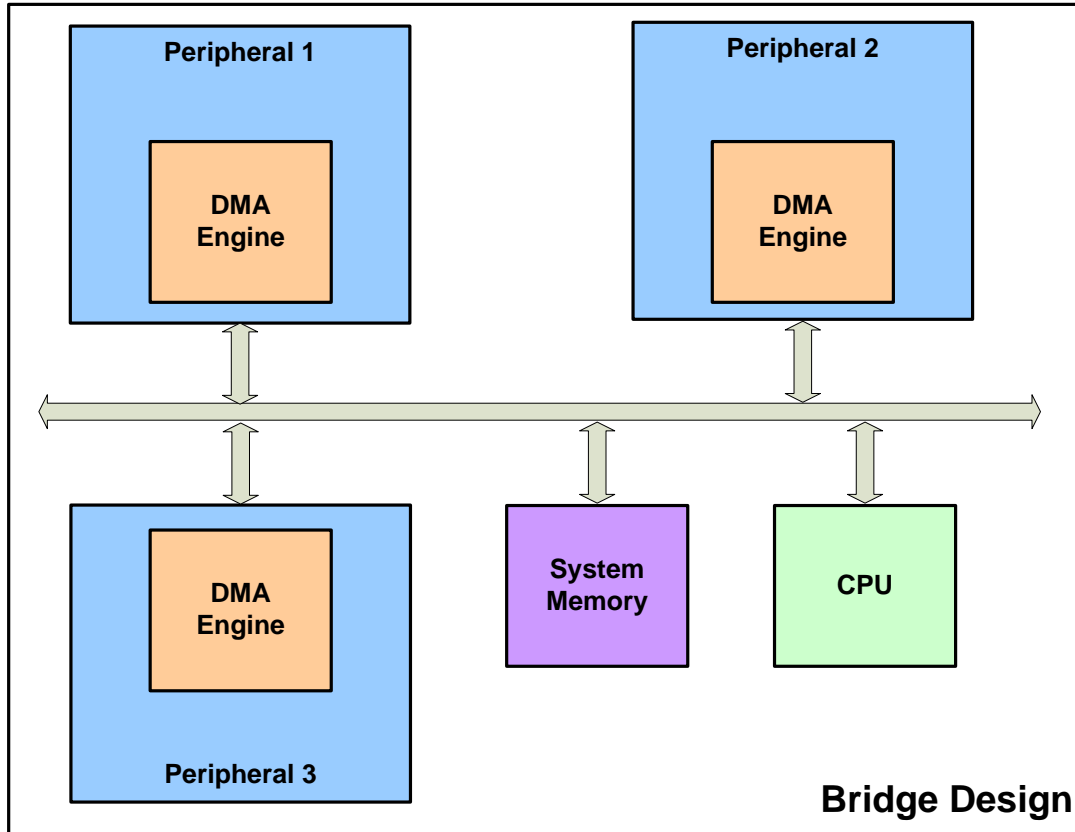
1. Introduction

DMA (Direct Memory Access) is an essential feature of all modern system/chip architectures as it allows the components to transfer data without intervention from the CPU. The DMA engine, descriptors and buffers handling and interface of the DMA engine to the interconnect are the main aspects in the verification of DMA. To verify the descriptors and buffers handling, one might choose a static approach in which the memory space for DMA descriptors and DMA buffers will be pre-defined in a static fashion and this space will be re-used to some extent. For example, as and when the descriptor and buffers are required to be allocated, the required amount of space will be used and for the subsequent request the next consecutive location will be used as the starting address. This results in using a restricted portion of memory space rather than using the full available memory space. So this approach will result in suppressing the buffer allocation problems like address aliasing.

The simulation needs to decide the number of descriptors and the number, size of buffers dynamically in a random fashion. To accommodate this one might try to add some randomization to the above approach. But that turns out to be complex.

2. Verification Requirements for a DMA based design

This paper uses a bridge design as an example to explain the usage model of MAM. The bridge design uses a DMA-based mechanism for data transfer between various peripherals. The peripherals are connected through a common on-chip bus protocol.



The CPU programs the descriptor chains into the system memory and informs the DMA engine to process the same. After completing the data transfer the DMA engine informs the CPU through an interrupt.

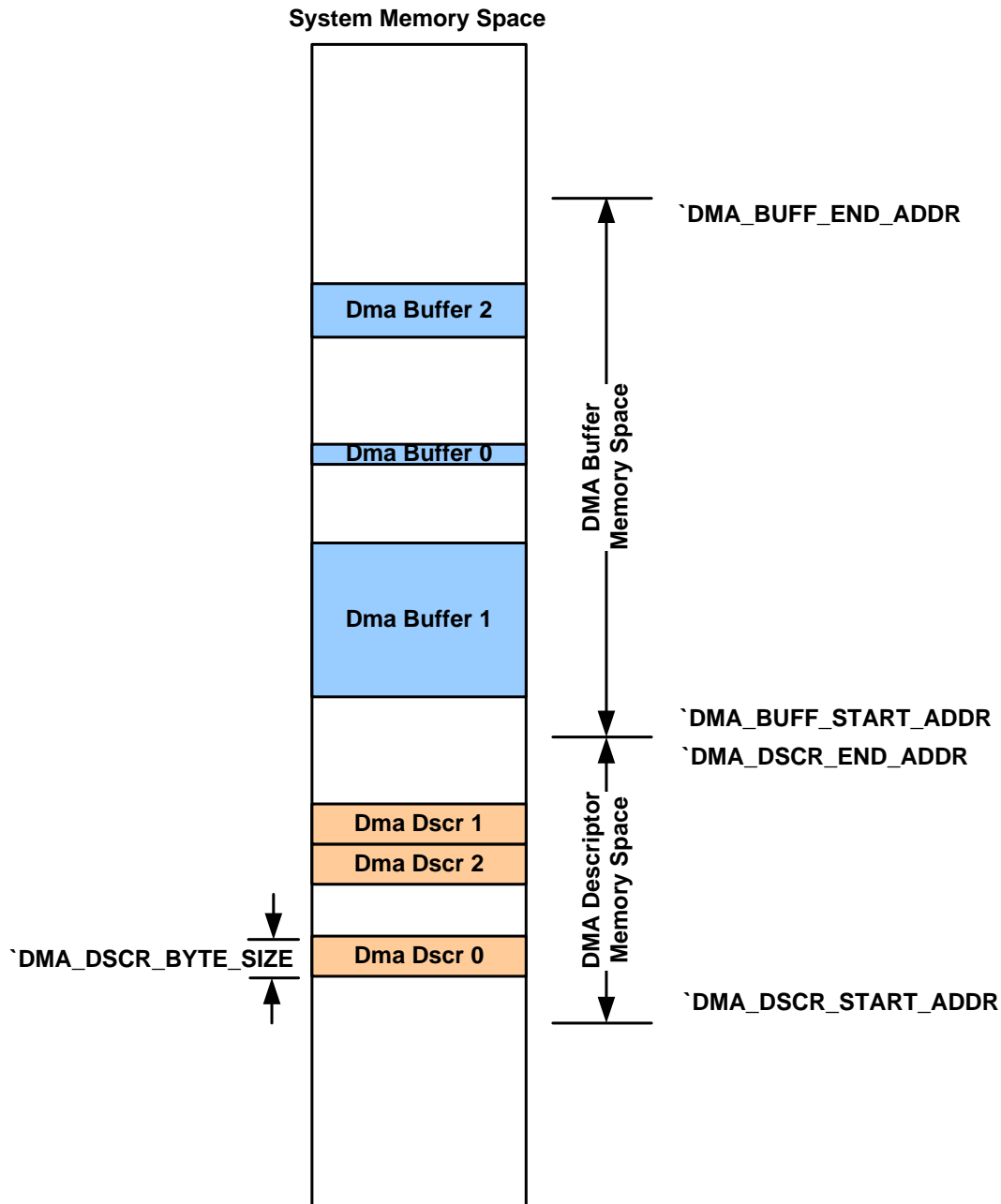


Figure 2. Allocating Random Descriptors and Buffers

The `DMA_DSCR_START_ADDR` compiler directive defines the start address of the Descriptor region. The `DMA_DSCR_END_ADDR` compiler directive defines the end address. `DMA_DSCR_BYTE_SIZE` defines the descriptor size in bytes. Similarly, `DMA_BUFF_START_ADDR` defines the start address of Buffer region and `DMA_BUFF_END_ADDR` defines the end address.

The CPU has to allocate different sizes of DMA buffers for data transfer between the peripherals and also the required DMA descriptors for the same. To better verify the DMA controller, the allocation of DMA buffers and DMA descriptors should be random over the available space as per the system memory map. Additionally, there shall be no overlap between the buffers and descriptors for the peripherals for proper functionality.

In the next two sections, we will see the structure of MAM and then its usage model to meet the above-mentioned verification challenge.

3. Structure of VMM MAM

VMM MAM contains the following four class definitions.

- Region Class: **vmm_mam_region**
- Configuration Class: **vmm_mam_cfg**
- Allocator Class: **vmm_mam_allocator**
- Memory Allocator Class: **vmm_mam**

In this paper, the details of these classes are restricted to what a user needs to know from the usage perspective. For more details, please refer to RAL User Guide (http://vmmcentral.org/pdfs/vmm_regabstraction_guide.pdf)

3.1 VMM MAM Region Class: *vmm_mam_region*

A region is an allocated memory space for a DMA Descriptor or for a DMA Buffer. This class represents the allocated memory region for a single descriptor or a single buffer.

get_n_bytes() : This method returns the numbers of bytes in each addressable location in the region..

get_len() : This method returns the length of the allocated space. To get the exact length in terms of bytes, the return value of this method needs to be multiplied with the value returned by the **get_n_bytes** method.

get_start_offset() : This method returns the starting address of the allocated region in the memory. Similarly, **get_end_offset** returns the end offset of the region.

release_region() : This method can be used to free-up the allocated space in the memory after the completion of DMA transfer. This will allow the space to be subsequently allocated for re-use by subsequent DMA transfers.

3.2 VMM MAM Configuration Class: *vmm_mam_cfg*

This class defines the configuration of a Memory Allocation Manager instance. This class contains the properties, like the starting address (**start_offset**), end address (**end_offset**) for the entire available memory space for either DMA Descriptor or for DMA buffers. It also contains properties like **n_bytes** (number of bytes in a single memory location) and **locality** (specifying the way in which the allocation needs to happen).

The **locality** can take two possible values, NEARBY (see: 5. Customizations done over the standard package) and BROAD. The NEARBY locality represents the allocation mechanism where multiple **vmm_mam_region**'s will be allocated in consecutive locations. The BROAD locality means the allocation will be at random locations.

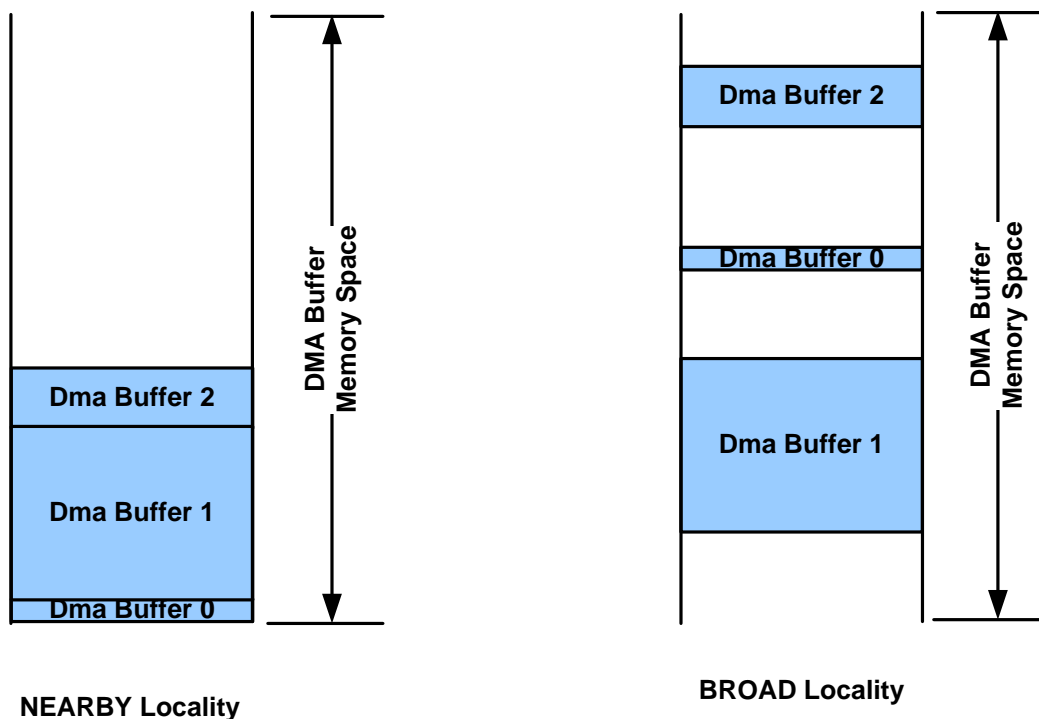


Figure 3. NEARBY and BROAD locality

The above defined properties are declared random. So one can randomize the **start_offset** and **end_offset** properties of the MAM configuration as well to map to multiple possible configurations

3.3 VMM MAM Allocator Class: *vmm_mam_allocator*

This class is the MAM allocator class, which it allows the user to add additional constraints on the allocation of regions (e.g. start offset alignment or region within a memory page) and ensures that there is no overlap when allocating the regions.

3.4 VMM MAM Class: *vmm_mam*

VMM Memory Allocation Manager Class is the main class and encapsulates the configuration information (**vmm_mam_cfg**) and the allocator class (**vmm_mam_allocator**). It maintains a queue of allocated regions. This queue grows and shrinks as and when regions are requested and released respectively. There can be multiple instances, each responsible for managing a different memory or available area.

This class contains the following methods to carry out the tasks like requesting a region, reconfiguration of the memory allocator and releasing a region.

new (string name, vmm_mam_cfg cfg): This method creates a new instance of the **vmm_mam**. During construction, the configuration information about the **vmm_mam** class needs to be passed as an argument.

reconfigure (vmm_mam_cfg cfg): This method performs the reconfiguration of **vmm_mam** class. The reconfiguration can only change the **start_offset** and **end_offset**. It can not change the **n_bytes** property of the **vmm_mam_cfg**. This method also checks whether already allocated regions fall within the newly specified **start_offset** and **end_offset**.

request_region (int unsigned n_bytes, vmm_mam_allocator alloc, bit [63:0] user_start_offset): This method is used to request a **vmm_mam_region** somewhere in the entire available memory range. While requesting the region, the user needs to pass the information regarding the number of bytes to be allocated for this region. By default, the **alloc** argument is assigned 'null'. If the user wants the region to be allocated at a specific location, then he can pass the third argument **user_start_offset**.

get_region (bit [63:0] start_offset): This method can be used to get a handle of the **vmm_mam_region** when the **start_offset** of the region is known. The return value can be used to free-up the allocated region by calling **release_region** method.

release_region (vmm_mam_region region): This method is used to release a previously-allocated region.

release_all_regions(): This method is used to release all the previously-allocated regions.

reserve_regions(): This method is used to reserve a memory buffer of the specified number of bytes starting at the specified offset in the memory

4. Using VMM MAM

4.1 Declaration of VMM MAM class objects

```
vmm_mam_cfg  m_oDmaDscrMamCfg;  
vmm_mam_cfg  m_oDmaBuffMamCfg;  
vmm_mam      m_oDmaDscrMam;  
vmm_mam      m_oDmaBuffMam;
```

Figure 4. Declaration of Configuration and MAM objects

We have requirement of allocation for DMA Descriptor and DMA buffers. So let us configure two VMM MAM instances, each managing the allocation memory in their respective available system memory areas.

4.2 Configuration and Construction of VMM MAM class objects

First you need to configure each MAM with details such as start offset, end offset, and number of bytes per memory location.

```
m_oDmaDscrMamCfg = new();  
m_oDmaDscrMamCfg.n_bytes = `DMA_DSCR_BYTE_SIZE;  
m_oDmaDscrMamCfg.start_offset = `DMA_DSCR_START_ADDR / `DMA_DSCR_BYTE_SIZE;  
m_oDmaDscrMamCfg.end_offset = `DMA_DSCR_END_ADDR / `DMA_DSCR_BYTE_SIZE;  
m_oDmaDscrMamCfg.locality = vmm_mam::NEARBY;  
  
m_oDmaDscrMam = new("DmaDscrMam", m_oDmaDscrMamCfg);
```

Figure 5. Configuration and Construction of DMA Descriptor MAM

Let us assume the descriptors always will be of size 16 bytes (`DMA_DSCR_BYTE_SIZE`) and it shall be at the 16 byte aligned addresses. So the **n_bytes** of VMM MAM configuration for DMA Descriptors shall be 16 bytes. In addition to the **n_bytes**, the **start_offset**, **end_offset**, **locality** needs to be specified for the DMA descriptor MAM configuration class object.

```

m_oDmaBuffMamCfg = new();
m_oDmaBuffMamCfg.n_bytes = 1;
m_oDmaBuffMamCfg.start_offset = `DMA_BUFF_START_ADDR;
m_oDmaBuffMamCfg.end_offset = `DMA_BUFF_END_ADDR;
m_oDmaBuffMamCfg.locality = vmm_mam::BROAD;

m_oDmaBuffMam = new("DmaBuffMam", m_oDmaBuffMamCfg);

```

Figure 8. Configuration and Construction of DMA Buffer MAM

The minimum size of DMA buffers will be 1 byte and buffers can be at any byte address. So the **n_bytes** of VMM MAM configuration for DMA buffers shall be 1. Similar to DMA Descriptor MAM configuration, specify the **start_offset**, **end_offset** and **locality** for DMA buffer MAM configuration class object.

After configuration, construct the VMM MAM instances for DMA descriptors and DMA buffers. Once VMM MAM is constructed it is ready to use for allocating and releasing regions.

4.3 Requesting a region in VMM MAM

```

vmm_mam_region oDmaDscrMamRegion;
vmm_mam_region oDmaBuffMamRegion;
bit [31:0]      bvDscrStartAddr;
bit [31:0]      bvBuffStartAddr;
int unsigned    nDscrCnt;
Int unsigned    nCurrentBuffLen;

// Allocate Buffers and Descriptors for the entire DMA Descriptor Chain
for (int i = 0; i < nDscrCnt; i++) begin

    oDmaDscrMamRegion = m_oDmaDscrMam.request_region(`DMA_DSCR_BYTE_SIZE);
    bvDscrStartAddr = oDmaDscrMamRegion.get_start_offset() *
                    `DMA_DSCR_BYTE_SIZE;

    // Calculate (or) Randomly select the required Buffer Length
    .....
    .....

    oDmaBuffMamRegion = m_oDmaBuffMam.request_region(nCurrentBuffLen);
    bvBuffStartAddr = oDmaBuffMamRegion.get_start_offset();

    // Formulate the Descriptor Content
    .....
    .....

    // Write the Descriptor in Memory at Location bvDscrStartAddr
    .....
    .....

end

```

Figure 9. Allocating Regions using VMM MAM

For allocating the regions, one needs to call the **request_region** method, specifying the number of bytes to be allocated. The **request_region** returns a handle to the allocated region, from which the start address can be obtained. This upon multiplying with **n_bytes** property of the configuration, gives the actual start address of the region in the memory space if byte-level granularity is used. If you use MAM through RAL, you don't need to worry about the physical address.

After completion of DMA operation, one can pass the **start_offset** to the **get_region** (see 5. Customizations done over the standard package) method and get a handle on the allocated region and then call **release_region** on that instance to free up the space in the system memory.

5. Customizations done over the standard package

In the **request_region** method, the NEARBY locality logic was added. The available locality logic in **request_region** method is BROAD. This gives the random allocation facility. The NEARBY locality which was added mimics the static approach of memory allocation and is very useful in bring-up of testbench which uses MAM.

In addition one extra argument called **user_start_offset** has been added for controlling the start address of allocated region exactly by the user (if required). This is much useful in getting coverage of allocating the descriptor and buffer at its maximum range of available address space.

get_region method was added to the VMM MAM class which takes the **start_offset** as argument and returns the handle of the **vmm_mam_region**. The returned handle can be used when calling the **release_region** method. **get_region** and **release_region** are very much required when the available space for descriptor and buffers are small in the system memory map and need to simulate for thousands of transactions for stress testing.

6. Advantages of using VMM MAM

Just by switching the locality configuration from NEARBY type to BROAD type, the VMM MAM allocates the descriptors and buffers randomly over the entire allocated space. This helps to quickly uncover the problems of aliasing in design as well as architecture if any.

The bugs related to address value handling in the design can be uncovered easily. If the design misses to decode the upper bits of address appropriately, then it is going to result in overwrite of data in the buffer or descriptor which finally end-up in either data integrity issue or wrong processing of buffers by the DMA engine. VMM MAM provides

the right platform for randomization of size and location of DMA buffers and DMA descriptors.

Once VMM MAM is configured, integrated and brought up properly, then a user can work at the higher abstraction level rather than working at the level of locations for DMA descriptors and buffers. From our experience, this gives a definite improvement over the productivity. The user will be working at the abstraction level of just specifying the total byte count to be transferred using the DMA engine. The rest of the details like how many buffers has to be allocated and how many descriptors need to be used for this and where exactly these descriptors and buffers sit in the system memory can be handled in the testbench using random transaction generator and the VMM MAM.

In verification environment, the VMM MAM works as a co-processor to the CPU model in providing the required randomization of descriptor, buffer allocation.

7 Conclusion

We discussed about the limitations of doing a static way of allocation for DMA descriptors and DMA buffers. Following that, we have seen the structure and usage model of VMM MAM class. The usage model is very simple - just configure, construct and start using it. We also discussed about the advantages of using VMM MAM highlighting some of the data points from our experience.

VMM MAM can also be considered for modeling some architecture where the buffer allocation happens between various ports. The available common buffer space needs to be allocated and reused for traffic flowing between multiple ports. To model this kind of architecture, VMM MAM can be used.

In Platform SOC projects, there will be a need for having configurable memory map for various purposes. To achieve this random configurability in the system memory map also one can use VMM MAM.

VMM MAM is an easy to use vmm library class and is the right choice for adding randomization to the allocation of buffers and descriptors in DMA based design verification and any other requirement of memory space allocation. MAM helps in dynamic allocation and management of memory space.

8 References

- [1] VMM Primer - Using the Memory Allocation Manager by Janick Bergeron at <http://www.vmmcentral.org>
- [2] VMM Register Abstraction User Guide from Synopsys
- [3] Verification Methodology Manual for SystemVerilog by Janick Bergeron, Eduard Cerny, Alan Hunter and Andrew Nightingale



SASKEN Communication Technologies Limited, Bangalore (INDIA)

www.sasken.com

©2010 SASKEN Communication Technologies Limited. All rights reserved