VMM PRIMER

Using the Memory Allocation Manager

Author(s):

Janick Bergeron

Version 1.1 / May 28, 2006

Introduction

The VMM Memory Allocation Manager is an element of the VMM Environment Composition application package that can be used to manage the dynamic allocation of memory regions in a large RAM. It is similar to C's malloc() and free() routines. It is useful for managing memory buffers required by designs to store temporary, such as DMA buffers. These memory buffers are usually configured as a buffer address value (pointer) through a descriptor or a linked list in the design.

This primer is designed to learn how to configure a memory allocation manager, and how to use it in a verification environment to properly configure a design that requires memory buffers. Other primers cover other aspects of developing VMM-compliant verification assets, such as transactors, verification environments and the Register Abstraction Layer package.

This primer assumes that you are familiar with VMM. If you need to ramp-up on VMM itself, you should read the other primers in this series.

The design used to illustrate how to use the VMM Memory Allocation Manager is the OpenCore Ethernet Media Access Controller (MAC). Despite being a fairly complex design, this primer focuses on a single aspect of its functionality: the transmit DMA buffers. A complete verification environment that can verify this design requires many more elements than a memory allocation manager. This primer will only show the portions of the environment that make use of the VMM Memory Allocation Manager class.

This document is written in the same order you would incorporate a memory allocation manager in a verification environment and verify your design. As such, you should read it in a sequential fashion. You can use the same sequence to use a memory allocation manager and use it to verify your design.

The DUT

Figure 1 shows the (partial) structure of the verification environment surrounding the Ethernet MAC. The firmware emulation transactor configures and controls the registers inside the design through a RAL model. The DMA memory, not being mapped in the DUT address space, is not included in the RAL model. The firmware emulation transactor accesses the DMA memory directly through backdoor read()/write() methods provided by the RAM model itself.

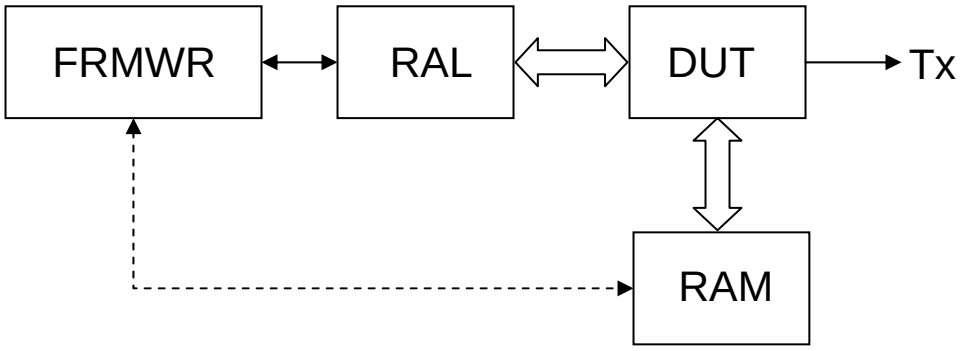

Figure 1: Partial Verification Environment

The firmware must first initialize the DUT as follows:

- All Transmit Buffer Descriptors are marked as “not ready”.
- Transmission is enabled.

To transmit a frame, the firmware must use the following procedure:

- The address of the frame in the RAM is specified in the Transmit Buffer Descriptor.
- Store the frame in consecutive locations in the RAM.
- The number of bytes in the frame is specified in the Transmit Buffer Descriptor.
- The Transmit Buffer Descriptor is marked as “ready”.
- Wait for a “Buffer Transmitted” interrupt. A Transmit Buffer will now be available and automatically marked as “not ready” by the DUT.

The DUT contains a programmable number of Transmit Buffer Descriptors (TxBD), up to 128, as defined by the TX_BD_NUM register. Each TxBD is located in consecutive memory addresses and has the structure partially by the following RALF specification.

SubEnv/oc_ethernet.ralf

```
block oc_ethernet {
   bytes 4;
   ...
   register TX_BD_NUM {
      field TX_BD_NUM { bits 8; }
   }
   ...
   regfile TxBD[128] @... +2 {
```

```
        register TxBD_CTRL {
           bits 32;
           ...
           field WR;
           ...
           field RD;
           field LEN { bits 16; }
        }
        register TxPNT {
           bits 32;
           field PTR { bits 32; }
        }
     }
     ...
}
```

There are two ways DMA buffers can be allocated: statically or dynamically.

With static buffer allocation, buffers are allocated when the TxBD are initialized. They are large enough to accommodate the largest possible frames to be transmitted and are reused to transmit different frames.

With dynamic buffer allocation, a new buffer is allocated for every frame to be transmitted. Buffers are sized according to the need of each frame. Each time, the TxBD is updated to refer to the newly allocated buffer.

An actual device driver would probably use a static buffer allocation approach. This approach minimizes the buffer allocation problem and restricts the memory usage to a well-defined area. But this approach is unlikely to expose several categories of functional bugs in the processing of transmit buffers. It would also require several different runs of randomly-allocated static buffers to cover different areas of the addressable space. Therefore, to exercise more of the addressable space and maximize the number of different buffer locations and sizes, a dynamic buffer allocation strategy will be used.

Step 1: Configuration

A Memory Allocation Manager will be used to manage the allocation of DMA buffers in the RAM. As such, it must be configured coherently with the configuration of the RAM itself.

The environment configuration descriptor contains an instance of the RAM model configuration descriptor. It must also contain an instance of the MAM configuration descriptor. Both are randomized and constrained to be coherent.

SubEnv/VIPs/wishbone/config.sv

```
class wb_cfg;
   typedef enum {BYTE, WORD, DWORD, QWORD} sizes_e;
   rand sizes_e port_size;
   ...
endclass: wb_cfg

class wb_slave_cfg extends wb_cfg;
   rand bit [63:0] min_addr;
   rand bit [63:0] max_addr;
   ...
endclass: wb_slave_cfg
```

SubEnv/blk_env.sv

```
class test_cfg;
   ...
   rand wb_slave_cfg ram;
   rand vmm_mam_cfg  mam;
   ...
   constraint test_cfg_valid {
      ...
      mam.n_bytes      == 1;
      mam.start_offset == ram.min_addr;
      mam.end_offset   == ram.max_addr;       ...
   }
   ...
endclass: test_cfg
```

When the test configuration descriptor is randomized in the extension of the vmm_env::gen_cfg() method, the configuration of the memory allocation manager will match the configuration of the memory it manages: they will both have the same number of bytes per address and the same address range.

SubEnv/blk_env.sv:

```
   virtual function void gen_cfg();
      bit ok;
      super.gen_cfg();

      ok = this.cfg.randomize();
      ...
   endfunction: gen_cfg
```

Note that the memory allocation mode and locality remain unconstrained. This will hopefully uncover bugs when DMA buffers are reused, located in adjacent locations or in widely different addresses.

Step 2: Buffer Initialization

The TxBD must be initialized to the “not ready” state before transmission is enabled. This is accomplished in the extension of the vmm_env::cfg_dut() step. Once the buffer descriptors have been initialized, transmission can be enabled.

SubEnv/eth_subenv.sv

```
virtual task configure();
   ...
   ral.TX_BD_NUM.set(cfg.n_tx_bd);
   ...
   begin
      int bd_addr = 0;
      ...
      repeat (cfg.n_tx_bd) begin
         ...
         ral.TxBD[bd_addr].RD.write(status, 0);
         ...
         bd_addr++;
      end
      ...
   end
   ...
   ral.TXEN.write(status, 1);
   ...
endtask: configure
```

The DUT is now ready to transmit frames. It is waiting for the first TxBD to be specified as “ready”.

Step 3: Frame Transmission

The firmware emulation transactor is responsible for initiating the transmission of frames it receives from the frame generator. When a frame and a transmit buffer are available, a DMA buffer of the appropriate length needs to be allocated and the frame data transferred to the memory corresponding to the newly allocated buffer.

SubEnv/eth_subenv.sv

```
class frmwr_emulator extends vmm_xactor;
   ...
   local task tx_driver();
      logic [7:0] bytes[];

      forever begin
         eth_frame fr;
         ...
```

```
            if (!drop) begin
               vmm_mam_region bfr;
               ...
               bfr = this.dma_mam.request_region(fr.byte_size());
               ...
               len = fr.byte_size();
               bytes = new [len + (4 - len % 4)];
               fr.byte_pack(bytes);
               for (int i = 0; i < len; i += 4) begin
                  this.ram.write(bfr.get_start_offset() + i,
                                 {bytes[i], bytes[i+1],
                                  bytes[i+2], bytes[i+3]});
               end
               ...
            end
         end
      endtask: tx_driver
      ...
   endclass: frmwr_emulator
```

Of course, once the frame is laid in the DMA buffer, all that remains is to update the buffer descriptor to point to the new DMA buffer and mark it as “ready”.

SubEnv/eth_subenv.sv

```
class frmwr_emulator extends vmm_xactor;
   ...
   local task tx_driver();
      logic [7:0] bytes[];

      forever begin
         eth_frame fr;
         ...
         if (!drop) begin
            vmm_mam_region bfr;
            ...
            bfr = this.dma_mam.request_region(fr.byte_size());
            ...
            len = fr.byte_size();
            bytes = new [len + (4 - len % 4)];
            fr.byte_pack(bytes);
            for (int i = 0; i < len; i += 4) begin
               this.ram.write(bfr.get_start_offset() + i,
                              {bytes[i], bytes[i+1],
                               bytes[i+2], bytes[i+3]});
            end
            ...
            ral.TxBD[bd_addr].PTR.write(status,
                                        bfr.get_start_offset());
            ...
            cfg.dma_bfrs[bd_addr] = bfr;
            ral.TxBD[bd_addr].LEN.set(len);
```

```
            ral.TxBD[bd_addr].RD.write(status, 1);
            ...
         end
      end
   endtask: tx_driver
   ...
endclass: frmwr_emulator
```

Step 4: Frame Completion

To avoid memory leakage, and allow a greedy memory allocation mode to allocate previously-used memory, it is necessary to free buffers once the frame they contained has been transmitted.

This is implemented in the interrupt service routine. When an interrupt signals that a frame has been transmitted, all of the Transmit Buffer Descriptors that are newly marked as “not ready” are assumed to have been transmitted. The memory allocated for this DMA buffer is then released.

SubEnv/eth_subenv.sv

```
class frmwr_emulator extends vmm_xactor;
   ...
   local task service_irq();
      ...
      forever begin
         ...
         if (TXE || TXB) begin
            ...
            while (...) begin
               ...
               ral.TxBD[bd_addr].RD.read(status, RD);
               ...
               if (RD) break;
               ...
               this.dma_mam.release_region(
                                       this.cfg.dma_bfrs[bd_addr]);
            end
         end
         ...
      end
   endtask: service_irq
   ...
endclass: frmwr_emulator
```

Step 5: Congratulations

You have now completed the integration of a VMM Memory Allocation Manager in a verification environment. You can now exercise more functionality of the design and have the opportunity to uncover more functional bugs through the random allocation of DMA buffers..

You may consider reading other publications in this series to learn how to write VMM-compliant command-layer transactors, verification environments or integrate a Register Abstraction Layer model or a Data Stream Scoreboard