2015 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

UVM's MAM to the Rescue

Michael Baird

Willamette HDL, Inc.

Agenda

- The Setup
- The Issue
- The Rescue
- MAM Details
- Contrast
  - Doing it the hard way
  - Doing it the MAM way
- Conclusions

Memories in the UVM Register Model

- Memories are mapped into the UVM register model with a base address and a range
  - Locations are accessed using an offset from the base address
    - Base address determined by register map
  - Register model translates this address to a physical address
  - Not mirrored in the register model
- Six types of accesses supported:
  - `read(), write(), peek(), poke(), burst_read(), burst_write()`

Issues with Multiple Temporary Buffers

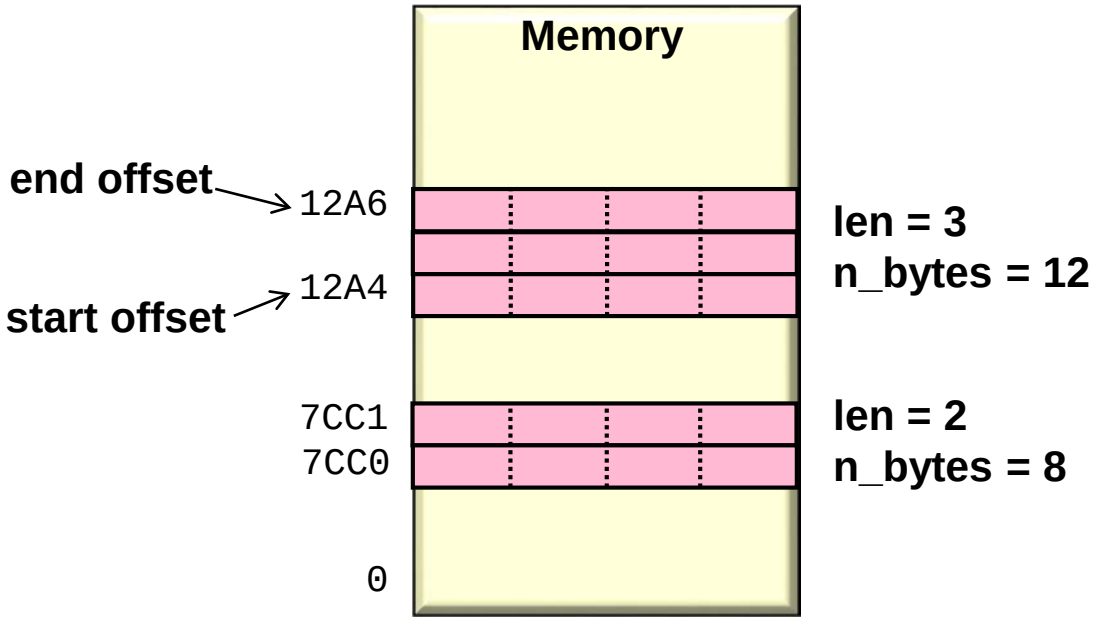
- Certain designs use local memory for temporary buffers
  - LAN controller buffers for incoming/outgoing packets
  - DMA controller source & destination buffers
- Managing buffers can be a burdensome, error prone task for verification engineer
  - Allocation/deallocation of buffers
  - Abstracting location details (start/end address etc.) within test code
  - Avoiding buffer overlap, or memory overflow

UVM's MAM to the Rescue

- Memory Access Manager (MAM)
  - Set of utility classes
  - Service requests for a contiguous range of memory of a specific size
    - Allocates a region of size requested in specified memory
    - No user knowledge of exact location of buffers required
    - Manages buffer location avoiding overlap

Allocated Region

- Allocated region treated as if an independent memory
  - Operations on region automatically translated into corresponding operations on actual memory



MAM Utility Class – uvm_mem_mam

- uvm_mem_mam
  - Provides methods for requesting and releasing memory regions

- Memories in the register model are modeled using the uvm_mem base class
  - Every uvm_mem object has a property mam of type uvm_mem_mam
    - Allows for use of MAM on any memory in the register model

uvm_mem_mam API

`request_region()`
- Creates a region at a random location of specified size

`reserve_region()`
- Creates a region at a specified location and size

`release_region()`
- Releases previously allocated memory region

`release_all_regions()`
- Forcibly releases all memory regions

`get_memory()`
- Get the memory where the region resides

Example uvm_mem_mam usage

```
// create handle to memory in register model
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
```

MAM Descriptor Class: `uvm_mem_region`

- 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 methods. Same as memory access methods

  `write(), burst_write(), read(), burst_read()`

  `peek(), poke()`

uvm_mem_region API – Access Methods

- Call the API methods on the uvm_mem_region 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);
```

uvm_mem_region API – 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

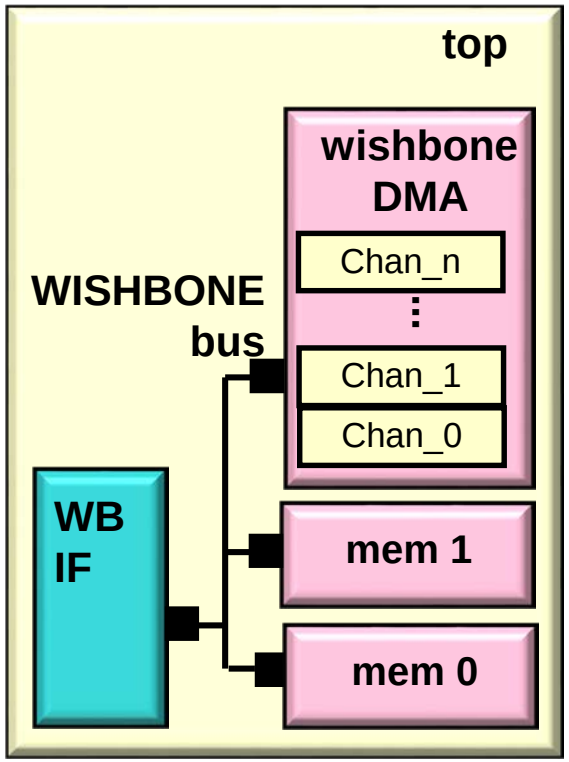
`get_memory()`
- Returns the handle to the `uvm_mem` object where the region resides

Example Circuit For Illustrating MAM usage

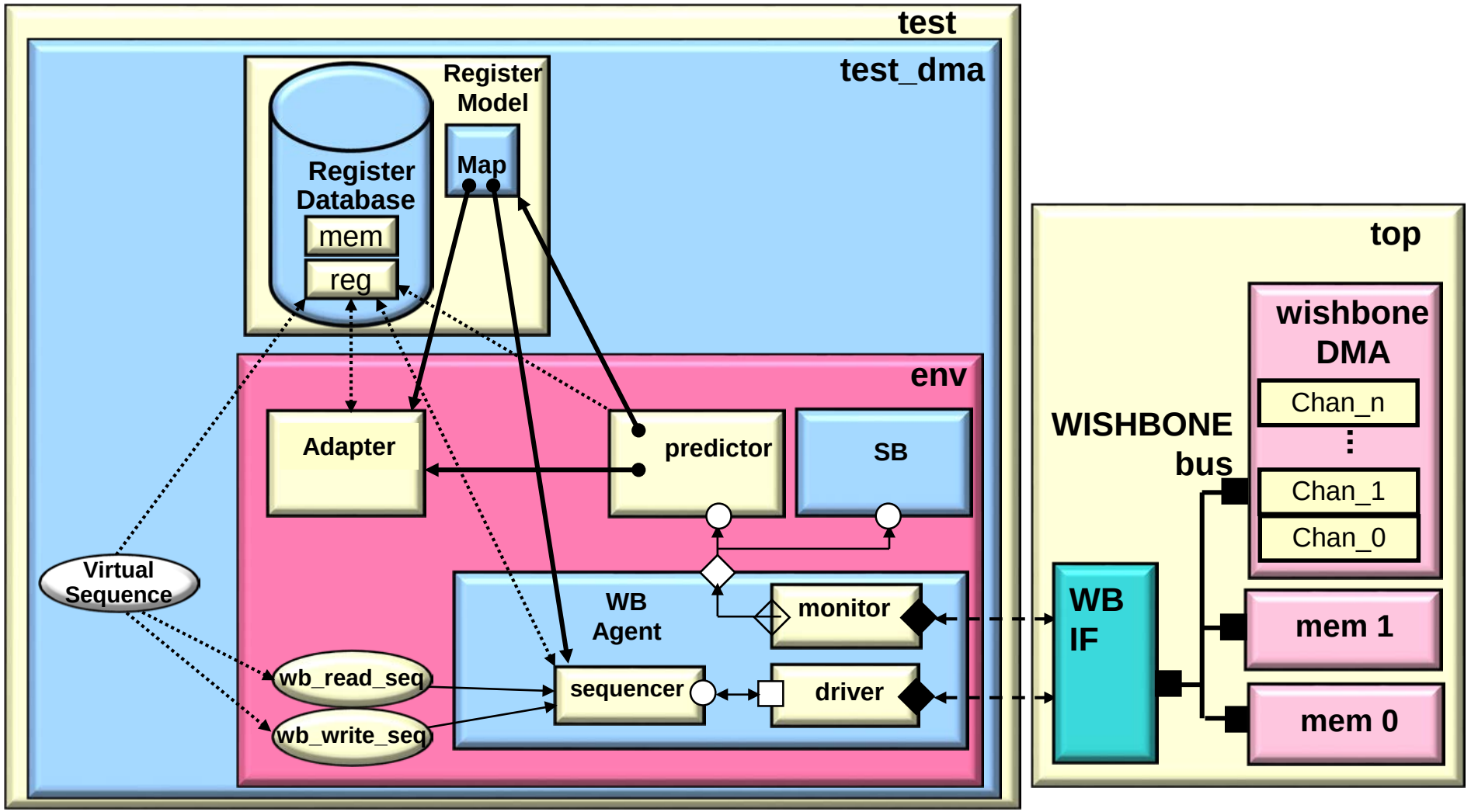
- 32 channel DMA controller
  - WISHBONE DMA/Bridge IP core from opencores.org
  - WISHBONE bus interface
    - Master for DMA transfers
    - Slave for access to its registers
- Two instances of the WISHBONE Slave Memory from opencores.org
  - Configured as 1 Mbyte slave memories

DMA DUT

- DMA chip initialization:
  - Initialize general registers
- DMA transfer:
  - Initialize channel registers
    - Source buffer address
    - Destination buffer address
    - Size of transfer
  - Initiate transfer by writing the CSR register



Testbench


test
test_dma
Register Model
Register Database
mem
reg
Map
env
Adapter
predictor
SB
Virtual Sequence
WB Agent
monitor
sequencer
driver
wb_read_seq
wb_write_seq
top
WISHBONE bus
wishbone DMA
Chan_n
Chan_1
Chan_0
WB IF
mem 1
mem 0

Non-MAM Based Tests "by-hand" Approach Steps

- DMA transfers executed by the `dma_transfer()` task in a virtual sequence
  - Four inputs to the task:
    - DMA channel number
    - Source buffer's start address (absolute address)
    - Destination buffer's start address (absolute address)
    - Size of the DMA transfer (in 32-bit words)

dma_transfer()

```
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
```

Tedious!

- Only takes a few transfers with manual setup for each transfer to become quite tedious!
  - Care must be taken to ensure no buffer overlap
  - Simply changing buffer size results in checking to ensure:
    - No buffer overlap
    - No spilling beyond the memory space
  - Introducing concurrent transfers further acerbates the problem

MAM Based Tests Steps

- DMA transfers executed by the `dma_transfer()` task in virtual sequence
  - Inputs reduced from four to just two
    - DMA channel number
    - Size of the DMA transfer (in 32-bit words)

dma_transfer()

```
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")
```

dma_transfer() (cont.)

```
  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
```

Simulator Output

- Output message from `request_region()`

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

UVM_INFO @ 830: reporter [RegModel] Attempting to reserve
['h0000000000001b554:'h0000000000001b55b]...
```

Note the random address location of allocated buffers

Calling dma_transfer()

- A simple set of calls to dma_transfer() illustrates ease of use with MAM

```
task body;
  // do dma transfers
  dma_transfer(0,    // channel number
          8 );       // transfer size in 32-bit words
  // do transfers in parallel
  fork
    dma_transfer(1, 1600);
    dma_transfer(2, 1000);
    . . .
  join
  dma_transfer(3, 800);
endtask
```

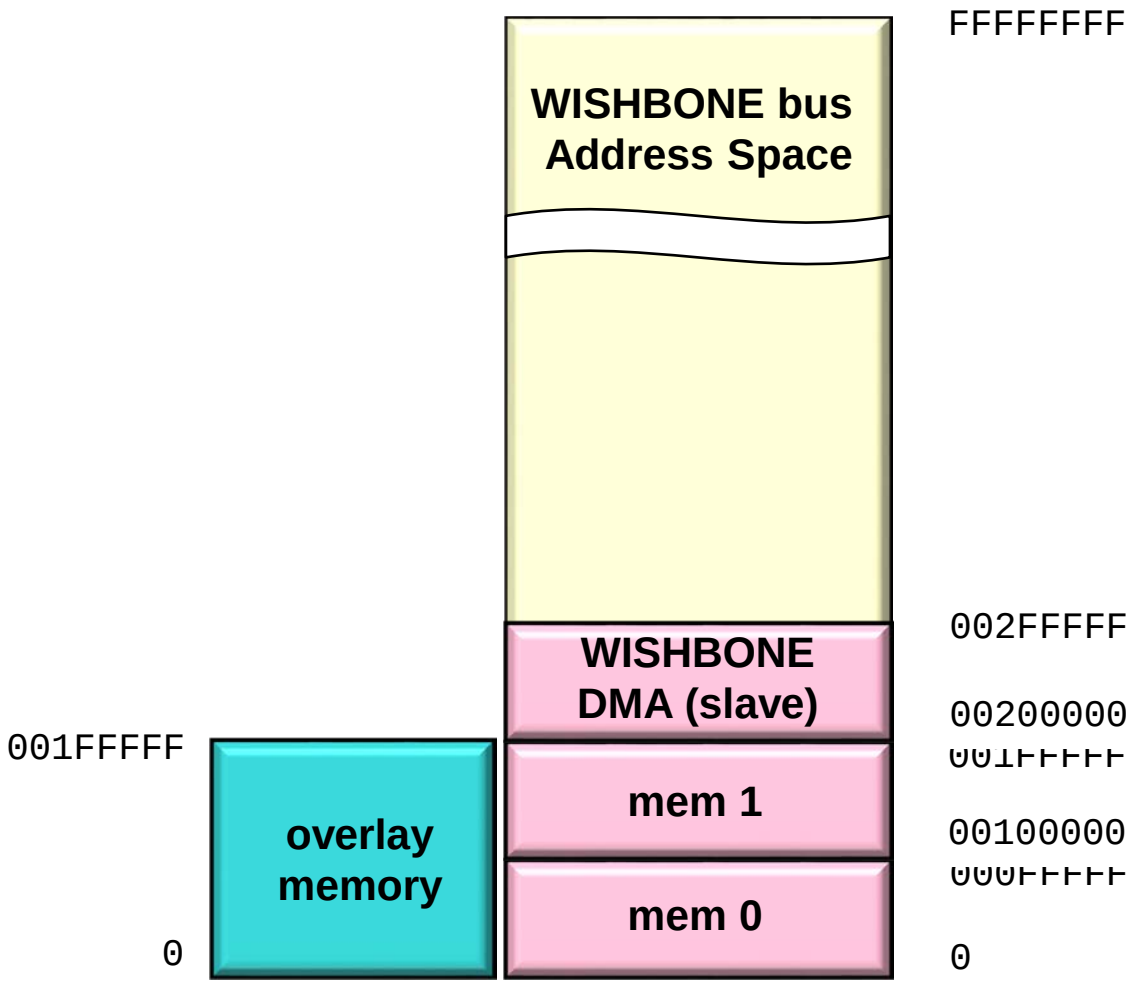
No need to supply buffer addresses

Concurrent transfers? No problem - unless you run out of memory

Bonus Coverage

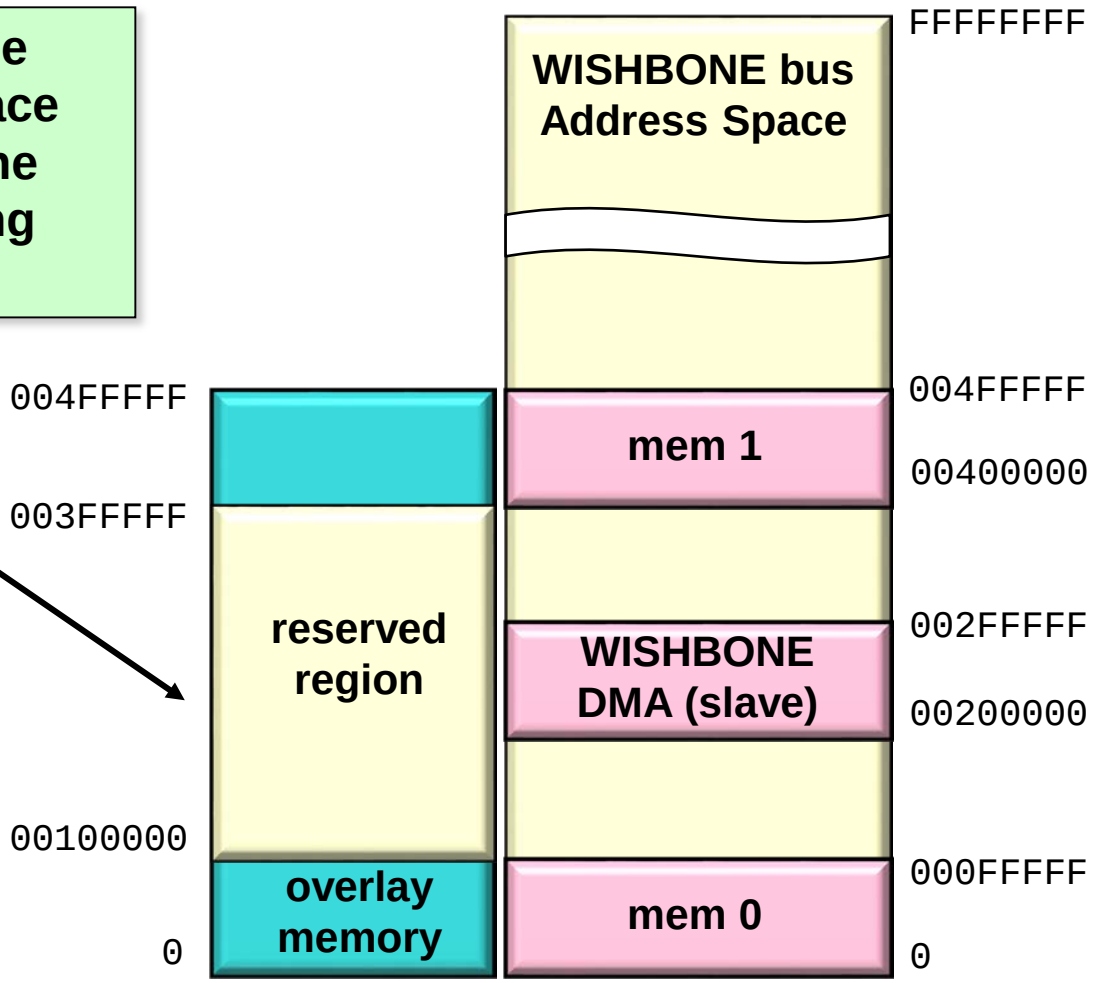
- What if it is desirable to allocate buffers across multiple different memories?
  - Don't want to specify a specific memory
    - Issue: You have to request or reserve a region using the `mam` property of that memory
- One solution is to declare an "overlay" memory
  - Spans full address occupied by different physical memories
  - Use the `mam` of the overlay memory to randomly allocate a buffer using `request_region()`

"Overlay" Memory with Contiguous Address Space



"Overlay" Memory with Non-contiguous Address Space

For each "hole" in the memory address space reserve a region in the overlay memory using `reserve_region()`



MAM Benefits

- Avoidance of by-hand buffer management
- Portability of stimulus code
  - Only need the handle to the register model memory where buffer is to be allocated
- Randomization
  - Randomization for the size and number of buffers
    - By-hand: Problematic
    - MAM: Easily supported

Conclusions

- UVM's MAM to the rescue when testing designs that use local buffers!
- Using the MAM resulted in tests that were:
  - Easier to maintain
  - Easier to change
  - More portable
  - Less error prone
  - Much less time consuming to generate
  - More easily and quickly scaled