The Art of Portable and Reusable UVM Shared System Memory Model Verification Methodology across Multiple Verification Platforms

Roman Wang
Advanced Micro Devices, Inc., Shanghai, China. +8613482890029, Roman.Wang@amd.com
Thomas Bodmer
Advanced Micro Devices, Inc., Sunnyvale, USA. 1-408-203-2988 Thomas.Bodmer@amd.com
Beryl Chen
Advanced Micro Devices, Inc., Sunnyvale, USA. Beryl.chen@amd.com

Abstract— Shared system memory is an essential design element of an IP or SoC, so the shared system memory model (SMM) becomes really important and indispensable to ensure the verification quality. Different teams or groups will deploy different SMM per different requirements using different languages. In addition, SMM is used in multiple verification platforms, such as UVM IP stand-alone, UVM ComboWhacker (UVM CW, a kind of AMD specific subsystem verification), virtual FPGA (vFPGA, a kind of simulation environment using FPGA netlist and glue logics), and SoC full chip. With more verification IPs (VIP) integration in verification, the question regarding how to make in-house SMM and VIP Built-in SMM work together is raised. All of these will introduce big reuse challenges. In the AMD southbridge group, a portable and reusable SMM verification methodology which had been qualified across different verification platforms in multiple projects for three years. It really shifts left our project schedule with low maintenance and integration efforts (an approximate 96% effort reduction), and high verification quality. In this paper, we will present our successful story and solution experience for anyone wishing to take a similar approach.

Keywords—UVM, System Memory Model(SMM), IP Standalone, Combo Whacker(CW), Virtual FPGA(vFPGA)

I. Introduction

The shared system memory is an essential design element in both IP and SoC. A comprehensive system memory model (**SMM**) is critical to simplify the design verification process while ensure the quality. In different verification platforms, SMM is used to create test sequences, predictor, scoreboard, etc. With more and more companies adopting the UVM methodology to develop their verification environment from IP level to SoC, maximal reusability is always the goal. When trying to make vertical or horizontal reuse of verification environment across IP, SoC, teams and projects, SMM reusability can be great challenges.

- Different verification teams or groups deploy and adopt the different SMM written by different languages, such as Verilog/VHDL, System Verilog, E, Vera, or C++/C.
    1. It's really difficult to make different SMMs work together in a specific verification platform.
    2. Only few SMM implements the memory allocation features. Ex. It can allocate exclusive regions.
    3. Potential PLI/DPI usage will affect verification performance too much.
- Different 3rd part UVM VIP provides the slave responder with different built-in SMM implementation.
- The legacy SMM could not easily work with UVM memory front door operation APIs.
- The flexible debug abilities and specific errors injection are all required.

The generic UVM SMM developed by AMD south bridge group addresses all the above concerns. In this paper, this portable and reusable UVM SMM is presented. It can be reused without any changes across various verification platforms, such as IP standalone, UVM CW, vFPGA, and SoC full chip. It also had been qualified in multiple successful projects for more than three years. In today's fast-pace semiconductor industry, in order to accelerate the verification process, lots of third-party UVM verification IPs (VIP) are integrated to reduce verification effort while maintaining the quality. However, different VIP implies different interface protocols. This proposed SMM is also protocol independent to all required UVM VIPs. With our successful story and solution experience, this paper can benefit you a lot.

Firstly, it attempts to dissect the idea of each key component with a big picture in typical UVM environment and try to explain how it works via answering several potential questions. Section III depicts the portable solutions across different verification platforms. Section IV presents the debug capabilities for quick bug-hunting. Section V summarizes our successful story in projects and future extensions. Finally, this paper concludes in Section VI by sharing what we benefit from this verification methodology.

II. PORTABLE AND REUSABLE UVM SYSTEM MEMORY MODEL VERIFICATION METHODOLOGY

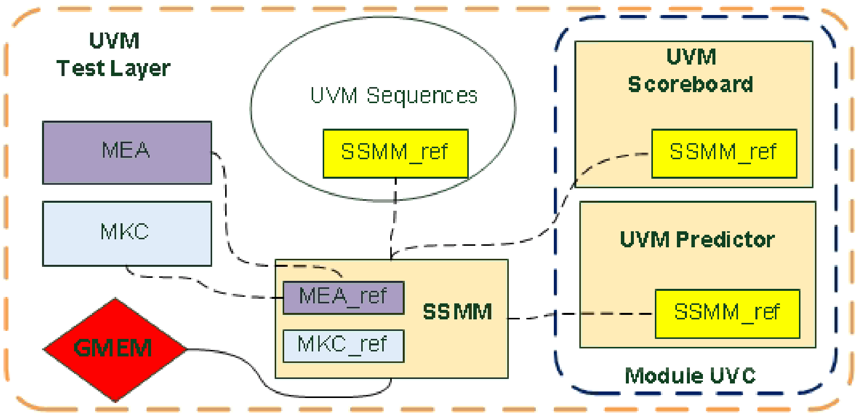

Figure 1. Big picture of key components in the typical UVM environment

The portable and reusable UVM system memory model verification methodology is made of five key concepts: UVM shared system memory export (**SSMM**), UVM generic save restore memory model (**GMEM**), UVM unique slave memory sequence (**USMSEQ**), UVM memory knobs container (**MKC**), and UVM memory export adapter (**MEA**). In Figure1, it presents the SSMM, MEA, MKC and GMEM are created in the UVM test layer. The unique SSMM reference is residing in the UVM sequences and components (such as, scoreboard and predictor). The EMA and MKC also have reference in SSMM. In next sections, we will present the details of each. Before we convey the details, it's necessary to understand few prerequisites first.

A. *Prerequisites*

➶ *System Memory Address Alignment*

The data and address alignment concepts are described well in the computer architecture knowledge. In this paper, let's take AMBA bus protocol as an example. The AMBA memory address could be aligned or unaligned when you want to access the data size bigger than BYTE. The Word aligned address requires the address [1:0] to be zero. When we allocate the data buffer, it's usually expected to return the aligned starting address for the designer to easily implement. This methodology predefined several alignments types for memory allocation.

Alignment type	***Description***
0	Half-Word alignment (16bits)
1	Word alignment (32bits)
Up to 12	1024 Words alignment (4KB)

Table1. Information of alignment in AMBA protocol

➶ *System Memory Segments*

In a complex SoC design, it usually defines several different types of memory attached on the system bus. They are shared with most of IPs and play the different roles per different production application requirements. On verification point of view, we can mark every type of memory as a unique memory segment which has the exclusive predefined address range. The secure related verification is becoming really important for Internet of Things (IoT) production. To meet the ARM Trust Zone concept, memory segment could be defined as a secure or non-secure domain as well. Figure 2 presents a clear picture of the relationship between system memory and memory segments.

In IP-level verification, it usually uses a unique memory segment to achieve the whole memory map for verification. Improper memory setting may introduce reuse problems from IP to SOC vertical reuse, because IP has specific accessible memory space which may cover multiple segments in SoC. It's better for IP level verification to set the exact accessible memory segments (defined in SoC memory map per project).

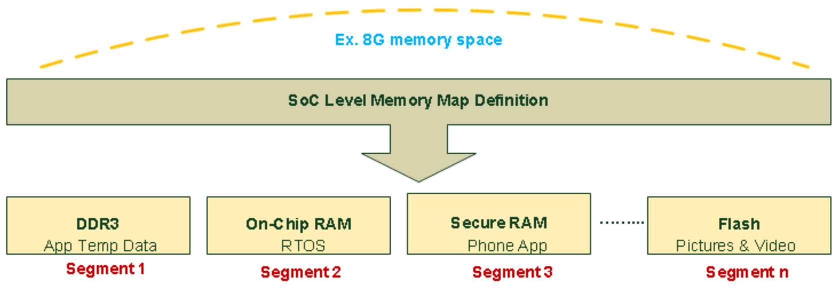

Figure 2. Concept of memory segments in a SoC

B. UVM Generic Save Restore Memory Model

The GMEM is a SystemVerilog class and can provide a singleton instance. It is designed to save and restore memory data with the address as index. It also provides necessary protected backdoor access APIs for SSMM usage (not for end-users) and DPI functions to support the legacy and vFPGA verification platform.

C. UVM Shared System Memory Export

The SSMM is a SystemVerilog class and can provide a singleton instance. It's designed as an export proxy to provide multiple friendly easy to use APIs for end-users to communicate GMEM. It's built in the base test layer and its reference will be passed down through the hierarchical verification environment. Verification engineers can use it to create random sequences, predictor, scoreboard, etc.

Current SSMM can support a set of key features as below:

- ✓ Built-in shared dynamic memory allocator manager which adopts the UVM MAM concept. It helps to allocate exclusive memory region and it's better for user to de-allocate the region when it's useless. At the report_phase of test, it will free all allocated regions automatically.
- ✓ User defined memory segments with more attributes, such as segment name, base address, access type, size, secure type, etc.
- ✓ User defined memory allocation policy to control the starting address alignment.
- ✓ Built-in memory adapter library to support different bus protocols: industrial standard protocol such as in-house bus protocols, APB, AHB, AXI4, PCIe, etc.
- ✓ Back-door access to the memory model (GMEM or Verilog/VHDL legacy memory model).
- ✓ Front-door access using built-in abstract UVM MEM model.
- ✓ Comprehensive error injection mechanism.
- ✓ Multiple outstanding requests with timeout control and out-of-order transaction completion.
- ✓ Response item (post or non-post) control, response latency and burst response delay control.
- ✓ Data audit for memory domain or secure check and nice debug utilities.
- ✓ Functional coverage.

D. *UVM Unique Slave Memory Sequence*

USMSEQ is a unique parameterized UVM sequence which is protocol independent and designed to co-work with SSMM to support DMA traffic. The unique means it could execute on any slave UVM VIP's sequencer without any change. This sequence could be vertically and horizontally reused. It can not only support different response type from request type, but also support the same types of them. For example, they use the response with request type to ease driver implementation in VIP. It has a local mailbox to get the request broadcasted from monitor and a SSMM reference to handle advance features, such as multiple outstanding request, errors injection, response control with latency, etc.

E. *UVM Memory Knobs Container*

MKC is an UVM object and has types of local associate arrays to support knob layer control. For example, it supports outstanding and response latency knobs. The knobs can control constraint by refining the weights. MKC is built in base test layer and its referent is passed down to hierarchical environment. The knobs are configured in test layer and can also be overridden via UVM command line processor.

F. *UVM memory export adapter*

MEA is an UVM object and derives from uvm_reg_adapter. In general, the verification engineer creates the register adapter (which derives from uvm_reg_adapter as well) to build the UVM register layering. The MEA is only working for the UVM_MEM operation, and the bus2reg and reg2bus usage model are reverse comparing with UVM_REG. To make things simpler, we create separated memory export adapters, and the legacy UVM_REG adapter usually exists in the UVM VIP package as a built-in feature.

G. *Main Featured key APIs in SSMM*

The typical APIs are listed in the Appendix1. User can call such APIs in UVM sequences or components.

H. *Typical Use Model Study*

To help the audience better understand this verification methodology, we try to present the details for several potential questions.

- *How to create SSMM?*

    1. Instance and initialize the user defined mem_export_cfg if system memory segment is not unique. For example, the IP stand-alone uses the default unique memory segment and SoC need the multiple different memory segments based on system memory map definition.
    2. In the base test layer, we declare the mem_export and gmem instance. It uses the config_mem_export to create built-in memory allocation manager and abstract memory model.
    3. User may set the max_alloc_size with a proper value to avoid memory allocation abuse.

```
uvm_mem_export mem_export;
uvm_gmem    gmem;
virtual function void build_phase (uvm_phase phase);
   super.build_phase(phase);
   gmem = uvm_gmem::get(this);
   uvm_resource_db#(uvm_gmem)::set("*","gmem", gmem, this);
   mem_fe = uvm_mem_fe::get();
   mem_fe.config_mem_fe(); // use the default setting and unique one memory segment
   uvm_config_db#(uvm_mem_fe)::set(this,"*", "mem_fe",mem_fe);
   mem_fe.set_max_alloc_size(70000); // setting the max allocation size by bytes
```

Figure 3. Code example to create the memory export and gmem

- *How the USMSEQ works with a specific UVM Slave VIP?*

    Firstly, let's talk about the traditional use case. When we verify the DUT master to preform DMA traffic, we integrate the slave VIP and enable its auto-responder feature with a built-in memory model. The slave responder will process the request, communicate memory model to save and restore data, and then simply send back the response to DUT.

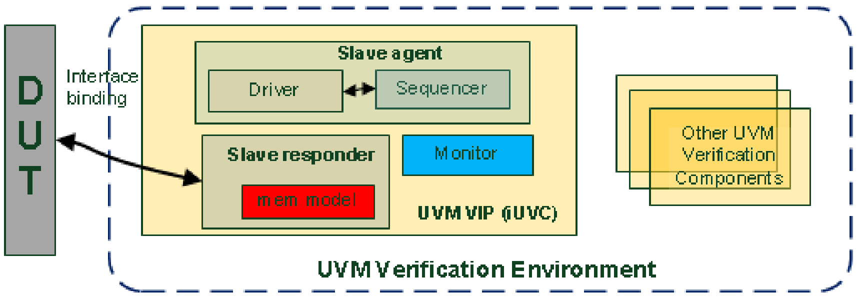

Figure 4. Traditional built-in memory model in UVM VIP

The VIP slave driver usually provides more comprehensive protocol features than simple slave auto-responder, so our solution is intent to use the VIP slave driver with mem_export and unique slave sequence to achieve full bus protocol. The figure 5 depicts a clear picture of the work flow using USMSEQ and UVM slave VIP.

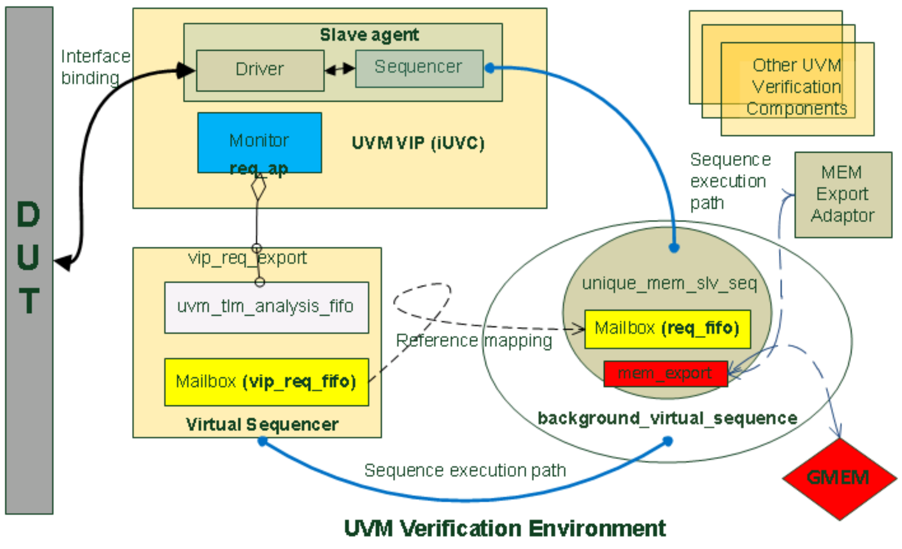

Figure 5. How USMSEQ works with other Slave UVM VIP?

The virtual sequencer defines uvm_analysis_export, uvm_tlm_analysis_fifo and mailbox. The uvm_analysis_export is connected to UVM VIP monitor's request analysis port (req_ap) and subscribes the request transaction from the DUT master bus. It's also connected to uvm_tlm_analysis_fifo's analysis_export and puts request transactions into tlm_fifo. The mailbox is used as a bridge to pass data between the virtual sequencer and the unique memory slave sequence. When on-the-fly reset happens, these FIFOs should be flushed automatically.

```
mailbox #(vip_request_transaction) vip_req_mx_fifo;
uvm_analysis_export   #(vip_request_transaction) vip_req_export;
uvm_tlm_analysis_fifo #(vip_request_transaction) vip_req_tlm_fifo;
function new (string name="global_vsqr", uvm_component parent=null);
   super.new(name, parent);
   vip_req_mx_fifo        =  new();
   vip_req_export         =  new("vip_req_export");
   vip_req_tlm_fifo       =  new("vip_req_tlm_fifo",this);
```

Figure 6. Code example in virtual sequencer

The background virtual sequence defines all free running sub-sequences in the background, such as unique memory slave sequences, interrupt service sequences, etc. It assigns the virtual sequencer's mailbox reference to unique memory slave sequences, gets the request transaction from virtual sequencer's uvm_tlm_analysis_fifo and puts it back to its local mailbox. In the fork/join_none block, the unique memory slave sequence is executing on the VIP's slave sequencer which has reference in the virtual sequencer. In figure8, the request and response transaction are different types, so it configures the mem_export using mem_fe.set_req_rsp_diff_hdl() API when building mem_export in the test layer.

```
uvm_sys_mem_slv_seq #(vip_request_transaction, vip_response_transaction)    vip_mem_slv_seq;

virtual task body();
     vip_mem_slv_seq =  uvm_sys_mem_slv_seq #(amd_axi4_request_transaction,
                                  amd_axi4_response_transaction)::type_id::create("sata_axi4_rd_slv_seq");
    vip_mem_slv_seq.req_fifo = p_sequencer.vip_req_mx_fifo;
    ……
    fork
       forever begin
          p_sequencer.vip_req_tlm_fifo.get(vip_req_tr); //blocking
          p_sequencer.vip_req_mx_fifo.put(vip_req_tr);
       end
       vip_mem_slv_seq.start(p_sequencer.vip_slv_sqr, this);
    join_none
```

Figure7. Code example of creating memory slave sequence in background virtual sequence

The unique memory slave sequence defines a mailbox to get the request transaction and mem_export to cooperate with GMEM. It sends the request items to GMEM by calling mem_export.interwork2gmem, and returns the proper response depending on VIP. At last, it sends the specific response item to the UVM VIP using `uvm_send(). In the VIP, the slave driver can get the response via get_next_item(), and process response item on the bus interface. To make a better trace of the response, it adds one variable named rsp_id to count response number and print in the log file.

```
class uvm_sys_mem_slv_seq #(type REQ = uvm_sequence_item, type RSP = uvm_sequence_item )
extends uvm_sequence #(REQ,RSP);
mailbox      #(REQ)   req_fifo;
uvm_mem_fe           mem_export;
REQ                        m_req_item;
RSP                        m_rsp_item;
`uvm_object_param_utils_begin(uvm_sys_mem_slv_seq#(REQ,RSP))
`uvm_object_utils_end
const static string type_name = {"uvm_sys_mem_slv_seq#(REQ)"};
virtual task body();
if(!uvm_config_db#(uvm_mem_fe)::get(null,get_full_name(),"mem_fe", mem_fe))
`uvm_error("RSRCNF", "mem_fe resource not found");
    req_fifo.get(m_req_item);
    if(mem_fe.rsp_req_same_hdl)
    m_req_item.copy(mem_fe.interwork2gmem(m_req_item));
    else begin
    m_rsp_item = RSP::type_id::create({"m_rsp_item",$sformatf("[%d]",rsp_id)});
    m_rsp_item.copy(mem_fe.interwork2gmem(m_req_item));
    m_rsp_item.set_id_info(m_req_item);
    end
    if(mem_fe.rsp_req_same_hdl)
    `uvm_send(m_req_item)
    else
    `uvm_send(m_rsp_item)
```

Figure 8. Code example in unique memory slave sequence

The uvm_env integrates all UVM VIPs, interface UVCs (iUVC) and other module UVCs (mUVC). The related analysis_port and analysis_export are connected in the connect_phase.

```
function void connect_phase(uvm_phase phase);
   super.connect_phase(phase);
   vip_inst.mtr.req_ap.connect(vsqr.vip_req_export);
   vsqr.vip_req_export.connect(vsqr.vip_req_tlm_fifo.analysis_export);
```

Figure 9. Code example of connection the tlm_fifo between VIP and virtual sequencer

- *How does the memory export adapter handle the protocol specific?*

  Memory export adapter is built in the base test layer and its reference is passed to SSMM using set_adapter function. Figure 10 depicts the detailed memory export adapter working flow. When the unique memory slave sequence gets the request item, the sequence will call the SSMM.interwork2gmem function and pass down the request reference. In the interwork2gmem function, it first calls the adapter.bus2reg to translate the bus request item into gmem_item (which has few elements, such as address, data, etc.). It uses the get_item function to get the gmem_item and communicates with GMEM using the backdoor_item2gme function. At last, it calls adapter's bus2reg function to generate the bus response item and return. In appendix 2, it lists the elements supported in original uvm_reg_item, and customizes few of them to support our solutions, such as the error injection.

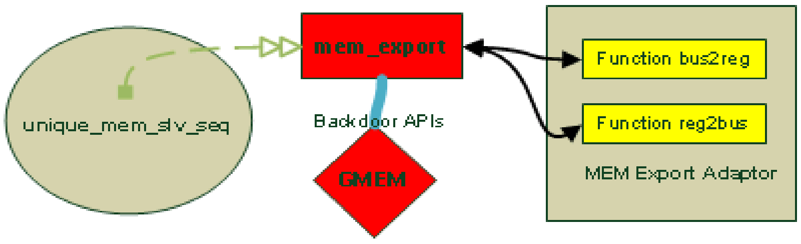

Figure 10. How the protocol adapter working with memory slave sequence

```
   reg_item.element_kind = UVM_MEM;
   m_adapter.m_set_item(reg_item);
   m_adapter.bus2reg(tr,rw_item);
   reg_item = m_adapter.get_item();
   if(backdoor_item2gmem(reg_item)) begin
   ……..
   return m_adapter.reg2bus(rw_item);
endfunction
```

Figure 11. Code example of interworking with gmem

- *How to achieve back-door access to memory model (GMEM or Verilog/VHDL legacy memory model)?*

  The memory backdoor access is an important method, it improves the efficiency of verifying memories since it can access memory locations with little or zero simulation time. It's usually used for initialized data buffer to prepare DMA traffic or get data from memory for checking or prediction. In the legacy verification environment, the integrator may instance the Verilog or VHDL legacy memory model. When we adopt this system memory methodology in the legacy environment, they still use the legacy model and require backdoor access using SSMM. UVM methodology provides some methods for uvm_mem to support the backdoor access, user could call the add_hdl_path_slice function to set the memory location path, and then directly call the peek (backdoor-read) or poke (backdoor-write) tasks. SSMM also provides the proxy APIs named as the peek_legacy and poke_legacy tasks to do the same. Users can also adopt four backdoor access functions (backdoor_write2gmem, backdoor_read4gmem, backdoor_batch_read4gmem, backdoor_bitstream_write2gmem and backdoor_batch_write2gmem) to save or restore data in GMEM directly.

- *How to achieve front-door access using built-in UVM MEM model?*

  When we verify a salve IP which has internal shared memory or external memory interface, it usually adopts a master VIP to exercise random memory traffic using an SSMM memory allocation manager. SSMM provides the proxy APIs (write/read function) to call the uvm_mem memory write/read functions to perform the front door access. In the legacy UVM register adapter's reg2bus function, it's required to implement few codes to support memory operation, such as burst, write or read, etc. There is no need to update the bus2reg function for prediction, because we don't save the data in the abstract memory model (derives from uvm_mem). In addition, the proxy APIs will call the SSMM's backdoor APIs to save or restore data in GMEM as the second copy, and it will be very useful for data checking.

In data fabric verification, it usually integrates several master VIPs on the slave ports of fabric and the slave VIPs (or memory controller with memory model) on the master ports of fabric. Every master VIP is attached with the similar UVM memory export layering, the problem is that we have multiple different physical interfaces. To address this challenge, SSMM provides the uvm_mem_export_cfg class for the user to define different information to create multiple maps. Each map will add proper memory segments using add_mem function, because each master has pre-defined accessible memory segments based on the SoC system memory map definition.

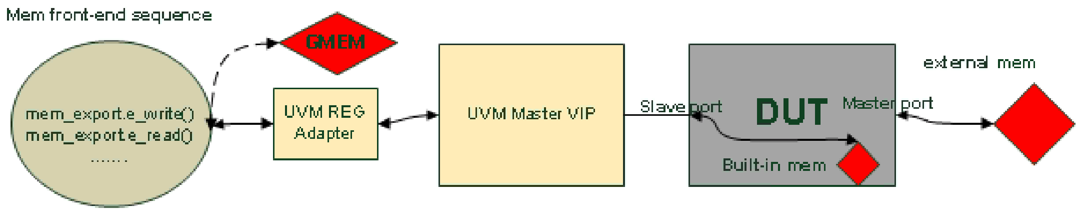

Figure 12. Front-door use model in IP level

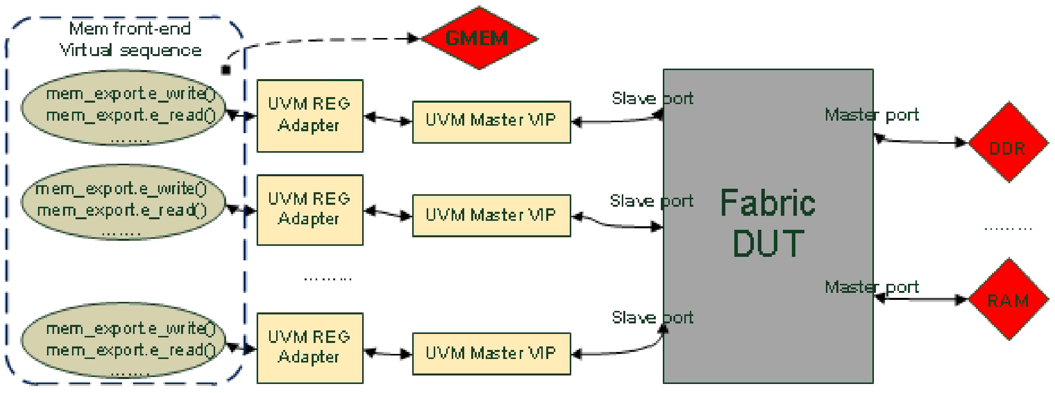

Figure 13. Front-door use model in SoC level

- *How to support multiple outstanding requests?*

  To improve performance, most of today's design supports multiple outstanding requests during DMA traffic. However the simple auto-responder could not support it. Verification engineers may use the pending outstanding request number to create functional coverage with other points. Users can set the outstanding request number and timeout threshold (which ensures no simulation halt), and this feature can be disable or enable by calling SSMM APIs in the UVM components or sequences. Its value also can be changed on-the-fly. In the unique memory slave sequence, it has two concurrent threads at the beginning of body task, the one is waiting for the pending request number till it's equals to the setting, another is to count the time ticks and break first thread when reaching the timeout threshold. It also records the current pending request number in SSMM, user could call specific API to get it and audit with other verification points. For example, DUT design needs to ensure there is no pending outstanding request when specific events or scenarios. When the outstanding request number reaches the setting, UVM VIP should assert the ready signal until it's allowed to accept more. To support this, UVM VIP must support external ready control mechanism, otherwise you have to force the ready signals when the outstanding request number reaches the setting.

- *How to achieve out-of-order transaction completion?*

  We implement the generic FIFO and instance it in the unique memory slave sequence. This FIFO can reorder the pending request and pop up the first request item to process.

- *How to achieve errors injection?*

  Errors injection, detection, and recovery are really important to verify the design robustness. Our methodology is protocol independent, so the idea is to deploy the different memory export adapters for protocol specific error injections requirements and coordinate SSMM. The SSMM provides several APIs to enable different error injection types, add user defined random delays in multiple errors, etc. The user can enable and disable the error injection feature during running time. The disable operation will clean all local settings, so the user should set the specific error injection before the next enable operation. It executes one time error injection by default.

Here is an example of the featured error injection for the AXI4 protocol.

Reserved Error Types for AXI4	*Descriptions*
1: AXI_RESP_EXOKEY	mem_export. Enable_inject_rsp_error(1, 2,2,1); // Two errors injection, delay_burst_num =2 and enable rand_delay
2: AXI_RESP_SLVERR	mem_export. Enable_inject_rsp_error(2, 2,2,1); // Two errors injection, delay_burst_num = 2 and enable rand_delay
3: AXI_RESP_DECERR	mem_export. Enable_inject_rsp_error(3, 2,2,1); // Two errors injection, delay_burst_num = 2 and enable rand_delay
4: RUSER[0]=1’b1& RRESP=OKAY	mem_export. Enable_inject_rsp_error(4, 2,2,1); // Two errors injection, delay_burst_num = 2 and enable rand_delay

5: Match Specified ID	mem_export.add_err_address(`adddress1, `mask1);
	mem_export.add_err_address(`adddress2, `mask2);
	mem_export. Enable_inject_rsp_error(5, 2,2,1);
	// Two errors injection, delay_burst_num = 2 and enable rand_delay

6: Match Specified Address	mem_export.add_err_ID(`ID1, `mask1);
	mem_export.add_err_ID(`ID2, `mask2);
	mem_export. Enable_inject_rsp_error(5, 2,2,1);
	// Two errors injection, delay_burst_num = 2 and enable rand_delay

Table 2. Error injection types for AXI4 Protocol

- *How to achieve post or non-post response item and latency control?*

  In the PCIe protocol, the memory write is the post operation, and that means no response is needed. In this case, we should change the provides_responses bit in the memory export adapter dynamically. User can enable or disable global no response dynamically. SSMM will audit the adapter.provides_responses status and m_global_no_rsp bit to decide if sending back the response item or not. To mimic the real system, it requires several latency cycles before returning the response. The random latency cycles can be controlled by latency knobs or user can set a global value by calling set_global_rsp_latency_dly API dynamically.

- *How to achieve a random burst response delay?*

  For DMA traffic, IP usually allocates several exclusive memory regions, issues multiple burst requests and slave VIP will return the burst response per its request. The burst response happens between two whole responses (a whole response may contain several beats) and belongs to the same memory region. For example: AXI READ burst response (which contains several beats) + random delay + AXI READ burst response (which contains several beats).

```
  // Ex.  Reserve 300 Bytes staring from ‘h9000_0000 address in memory segment 1, the reserving is for “GPU”
   if( !mem_fe.reserve('h90000000,300,1,”GPU”))
    `uvm_error("mem_fe.reserve","mem_fe.reserve failed!!!!\n")
 // Ex. Allocation 4KB with alignment==12 in segment 2, and it’s for USB3.
   St_addr_usb3 = mem_fe.alloc(1024*4,12,2,"USB3");
// Ex. Allocation 2KB with alignment==12 in random segments way, and it’s for SATA.
   St_addr_sata = mem_fe.alloc_rand(1024*2,5,"SATA");
// Ex. Free allocated region for USB3, but not clean related GMEM
   status = mem_fe.free(St_addr_usb3);
// Ex. Free allocated region for USB3, and clean related GMEM with all x data
   status = mem_fe.free(St_addr_usb3,1);
// Ex. free_all
   mem_fe.free_all();
// Ex. Erase 1KB starting from 'h90000000 address with random data
   mem_fe.erase2rand('h90000000,1024);
```

Figure 14. Code example of using memory export in UVM sequences

- *How to make use of the built-in memory allocation manager?*

In a real OS operation system, for specific applications, it’s required to allocate multiple exclusive memory regions located in random system memory segments. In the design verification world, we really have the same requirements. For example, IP DMA traffic need to allocate the reserved initial data buffer, command table buffer, data exchange buffer, etc. All of them must be exclusive and can be cleaned to improve simulation performance when they are useless. As appendix1 shows, SSMM provides multiple APIs to support different memory allocation and free requirements (one region or all of them). The free operation will delete the allocated region in the abstract memory model, and also supports an option to clean the related GMEM buffer with x data. It provides two APIs to support only erase GMEM with all zero or rand data. The user can call them in sequences any time.

III. PORTABLE SOLUTIONS ACROSS DIFFERENT VERIFICATION PLATFORMS

A. IP UVM Stand-alone Verification Platform

The IP-level stand-alone verification platform is designed to verify a specific IP with multiple UVM VIPs. Figure 15 depicts an overall UVM IP stand-alone test bench. The IP RAL model, GMEM, and mem export are built in the test layer, and pass down these references through hierarchical UVM components via uvm_configure_db. The module UVC will get their references and assign to the module configure object which is shared between UVM sequence and components. All interface UVCs are acting in active mode and communicating with DUT. When the IP module is vertically reused to the high layer, they will be configured as passive mode and only make the monitor active to spy the bus and predictor, scoreboard for self-checking.

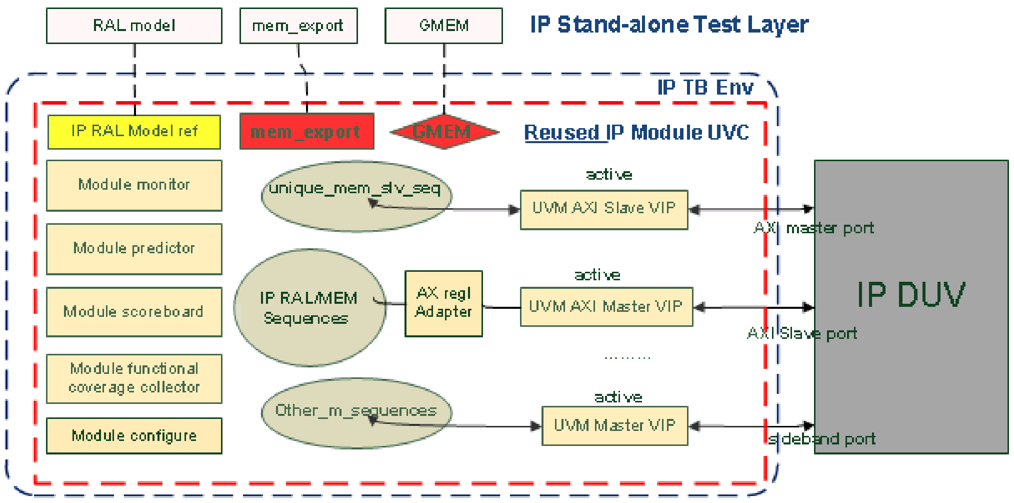

Figure 15. Methodology adoption in UVM IP standalone environment

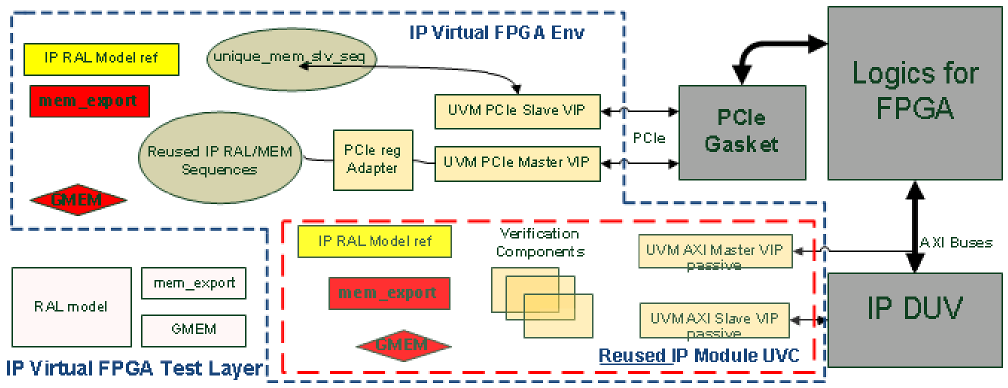

Figure 16. Methodology adoption in IP vFPGA environment

B. IP vFPGA Verification Platform

The IP vFPGA verification platform is designed to verify IP (especially for new design or big design changes) in cycle-based simulation before real FPGA verification. Figure 16 depicts an overall IP vFPGA test bench. When the FPGA verification finds the potential design bug, vFPGA can reproduce, review, and fix it on this platform. Besides of design netlist, vFPGA platform also has more synthesible glue logics are involved, such as PCIe gasket. We integrate the PCIe master and slave UVM VIP, implement PCIe RAL layering to reuse the IP-level RAL sequences. The reusable IP level module UVC is also integrated in the IP vFPGA environment to achieve IP prediction, functional checking, RAL model prediction, and functional coverage collection. IP RAL model, GMEM, and mem export are built in the IP vFPGA test layer, and pass down these references through hierarchical UVM components (such as IP vFPGA environment and reused IP level module UVC) via uvm_configure_db to ensure they are consistent.

C. IP or Mega-IP UVM CW SoC Verification Platform

In a SoC design, the different design teams will deliver the IP (or Mega-IP), the integration team integrates the separated IP (or Mega-IP) into a bigger IPs and attaches them on the data fabric which is the central block to ensure the high performance system bus traffic . The IP UVM CW verification platform is designed to verify the separated IP's basic functions and connectivity in the SoC design database. It only cares about the data path verification between IP and system data fabric, so other unnecessary IPs will be replaced with the shells (or called stub). It's helpful to accelerate the compile and run time. The Mega-IP looks like a kind of big IP which integrates more IPs inside. Figure 17 depicts an overall IP or Mega-IP UVM CW test bench. On the data fabric, we adopt in-house or 3rd part UVM VIPs and implement others, like UVM RAL layering and reused IP level module UVC which are similar to what the IP vFPGA verification platform does. IP RAL model, GMEM, and MEM Export are built in UVM CW test layer, and pass down references through hierarchical UVM components using uvm_config_db..

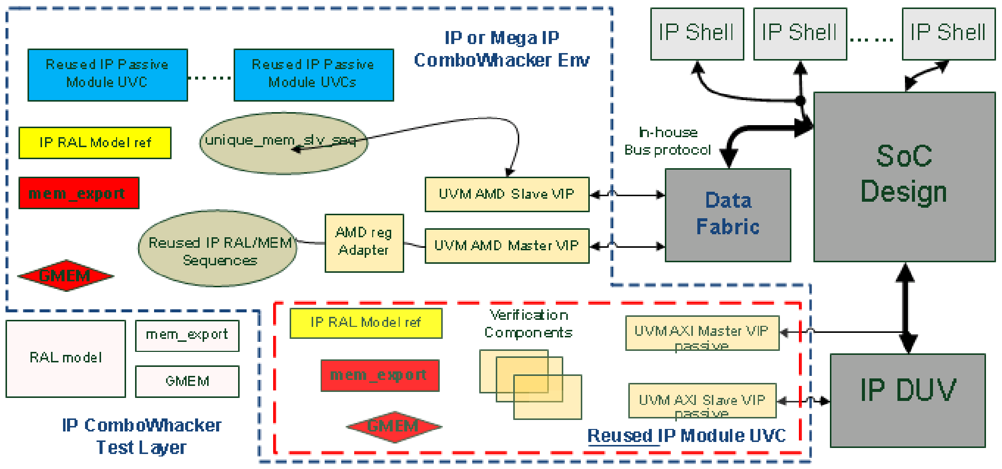

Figure 17. Methodology adoption in IP UVM CW SoC environment

D. SoC Full Chip UVM Verification Platform

When all of the IP are ready and integrated into a SoC database, IP UVM RAL sequence libraries and legacy C++ tests can be reused to verify the functions of SoC. To speed up the simulation, we use the UVM VIPs to replace the CPU complex design which actually presents the shell. In SoC verification, it adopts the real memory controllers with the Verilog/VHDL or VIP memory model, and we only need to capture the write request and fill data into GMEM. The TLM2APIs are designed to connect legacy C++ test cases with UVM master VIP. We have lots of legacy C++ tests which adopt the legacy C++ memory model, in order to align with this methodology we create several DPIs to make communications between C++ and UVM memory models. The UVM Slave VIP's monitors spy the data bus around fabric and broadcast write request information to unique memory slave sequence, and then use mem_export to write data into GMEM. At the same time, the data will be written to C++ memory model via DPIs. In this way, the data will be consistent between different memory models. All IP's module UVCs are reused in SoC level to do the self-checking, IP level functional coverage collection and RAL model update. IP RAL model, GMEM, and MEM Export are built in UVM test layer, and pass down these references through hierarchical UVM components using uvm_config_db.

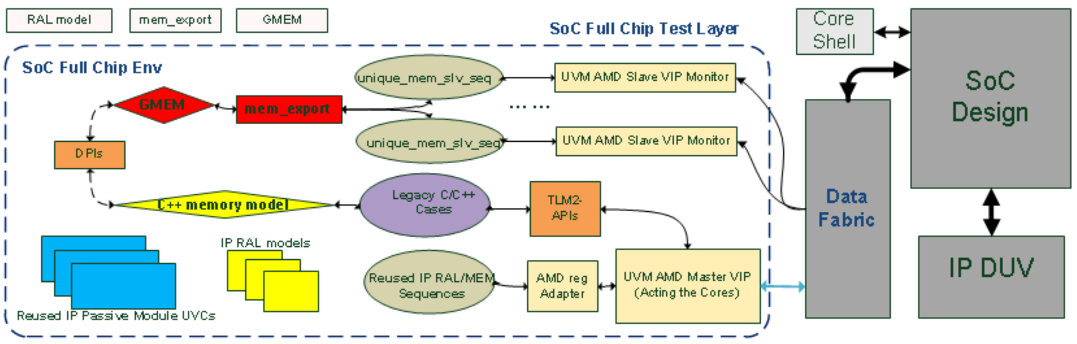

Figure 18. Methodology adoption in SOC Full Chip environment

IV. DEBUG

Debug is always painful for most of verification engineers who usually take more than 50% time on the debugging, such as design, verification environment, and test cases. Our solution offers several mechanisms to help understand the information of memory allocation, data in GMEM, error injection, outstanding requests, latency, and so on. In the verification environment, mem_export reference could be got via uvm_config_db in both UVM components and sequences. The user can call the specific debug API to get proper information. Besides API capability, necessary debug related information with different verbosities is developed. For example, UVM_LOW gives the debug messages that indicate major events. UVM_HIGH gives the debug messages of methodology implementation and the developer could use them to fix bugs. The user could set the proper verbosity to get different level debug information via UVM command processor. The following table presents an example of the dump() function to show all allocated memory regions.

```
                              ....... All Allocated/Reserved Regions .......
 "SATA CMD TABLE1"  had got allocated region:  ['h83291540:'h83291550] in "DDR4 Segment"
 "SATA Data Buffer"     had got allocated region :  ['h8dadecc0:'h8daf2130]   in "DDR4 Segment"
 "Firmware Buffer"         had got reserved region:   ['h13124300:'h1312f000]  in "RAM1 Segment"
```

V. SUCCESS STORIES

Our portable and reusable UVM shared system memory model verification methodology has been across numerous UVM projects in our group for more than three years. Typically we can see such projects obtaining an approximate 96% effort reduction compared with creating the memory model methodology from scratch and reusing them during adopting the UVM VIPs with different protocols across different verification platforms. As we can see in the following table, when we verify two IPs which adopt the same protocol of UVM VIP, the verification effort in the second IP is significantly reduced with benefit from the portable and reusable capability of our solution. Furthermore, the experienced engineers can easily understand how to integrate and use this methodology by reading detailed user guide, and eventually reduce more time in the next project. With the help of flexible debug capability, verification engineers could find the bugs in hours comparing to get it more than one day without such help. Outside the box, the verification engineers keep raising more interesting requirements to inspire us to enhance this solution to accelerate project execution.

IP/Mega-IP Level	Jobs	Efforts per person
	Methodology Integration	0.5 day
1st IP	Smoke test with debug	1 day
	Methodology Integration	0.4~0.6 hour
2nd IP	Smoke test with debug	1 hour
UVM CW/SoC Level	Jobs	Efforts per person
	Methodology Integration	1 day
1st SoC	Smoke test with debug	4 day
	Methodology Integration	1 hour
2nd SoC	Smoke test with debug	12 hours

To address more interesting verification challenges, we need to continue the development of this methodology. We are currently planning to include following improved extensions later.

- To refine all error injections using extension uvm_object declared in uvm_reg_item.
- To support user-defined backdoor access.
- To support error injection with random in multiple error types at a time.
- Even better debug aiding mechanisms.
- Support more potential bus protocol UVM VIPs.
- Support DDR4 abstract model

VI. CONCLUSION

In this paper, we present a portable and reusable UVM shared system memory model verification methodology and its typical use model. Lots of friendly APIs in SSMM make people easily deploy the verification requirements. The flexible debug information shows a clear picture of what engineers want to see and reduce the debug time. The export adapter library provides a quick connection to the different bus protocols. The protocol independent unique memory salve sequence bridges the gap in adoption of different protocol VIPs. Based on our numerous successful UVM projects' experience, this methodology can be portable and reusable across multiple verification platforms: UVM IP or Mega-IP stand-alone, UVM CW, vFPGA, and SoC full chip to dramatically reduce verification efforts. The current is not the end, and it is keeping evolving.

ACKNOWLEDGMENT

I would like to thank my wife (Liangliang Li) and the AMD south bridge team (Davis.Wan and Leo.zhang) for their continued support. AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

REFERENCES

[1] UVM1.2 Reference Manual. Accellera www.accelera.org

[2] IEEE 1800-2009 SystemVerilog

[3] http://www.accellera.org/apps/org/workgroup/uvm/

APPENDIX

1. Appendix1.Key APIs list in memory export

Table Column Head	
API Name	***Description***
function void **config_mem_fe** (uvm_mem_fe_cfg cfg = null, longint unsigned size = ‘h8000_0000_0000_0000, // This is to use the default only one segment, if user doesn’t config the segments. Int unsigned width_of_mem = 8, string access = “RW”, uvm_reg_addr_t seg_base_addr = ‘h0, uvm_endianness_e endness = UVM_LITTLE_ENDIAN, bit map_byte_addressing = 1);	User hooks the pre-defined the memory information, SSMM built-in abstract memory model will be created using such information. If user doesn’t define cfg object, it will use the unique one segment for all memory sizes and treat it as a non-secure domain.
Function void set_max_alloc_size(int unsigned max_alloc_z);	User hook to change the default allocation size, such as 1024B.
function void set_adapter(uvm_reg_adapter adptr);	Set specific adapter to handle different protocols
function bit reserve (uvm_reg_addr_t addr = 0, int unsigned n_bytes = 4, int unsigned segment = 0, string requester = “Roman”);	Reserve a specific memory buffer
function uvm_reg_addr_t alloc (int unsigned n_bytes = 4, int unsigned alignment = 2, int unsigned segment = 0, string requester = “Roman”);	Allocate a random buffer
function uvm_reg_addr_t alloc_rand (int unsigned n_bytes = 4, int unsigned alignment = 2, string requester = “Roman”);	All domains are visible for this API
function bit alloc_chained_list (output uvm_reg_addr_t addr_list[], output int unsigned buf_list[], input int unsigned n_bytes = 4, // Total input int unsigned alignment = 2, input int unsigned segment = 0, input string requester = “Roman”);	Advanced allocation to make big buffer split into several continuous chain.
function bit free(uvm_reg_addr_t st_addr, bit is_clgmem = 1);	Free specific buffer
function free_all(bit is_clgmem = 1)	Free all buffers
function void erase2rand(uvm_reg_addr_t st_addr, int unsigned n_bytes = 1);	Clean specific buffer as random data
function void dump();	Dump all allocation information
function void backdoor_write2gmem(uvm_reg_addr_t addr = 0, bit [127:0] data = 0, int unsigned n_bytes_of_data = 8, uvm_endianness_e endness = UVM_LITTLE_ENDIAN);	Backdoor access to GMEM
function void backdoor_bitstream_write2gmem (input uvm_reg_addr_t addr = 0, ref bit bitstream[]);	Backdoor batch access to GMEM
function void backdoor_batch_write2gmem (input uvm_reg_addr_t addr = 0, ref byte unsigned data[]);	Backdoor access to GMEM
function bit backdoor_batch_read4gmem (input uvm_reg_addr_t addr = 0, ref byte unsigned data[], input int unsigned n_bytes = 1);	Backdoor batch access to GMEM
function void enable_outstanding_req();	Enable outstanding request feature
function void set_outstanding_req_num(int unsigned num = 1);	Set outstanding numbers
function void enable_global_no_rsp();	Enable bypass response
function void set_global_rsp_latency_dly(time dly = 1ns);	Set global latency
function void enable_inject_rsp_error(int err_type, int num_of_err_burst =1, int delay_burst_num = 0, bit is_rand_delay = 0);	Enable specific error injection type
function void add_err_address(bit [63:0] addr, bit [63:0] mask = ‘1);	Set error injection address list
function void set_outstanding_request_timeout(time to = 1us);	Set outstanding request timeout
function void enable_out_of_order();	Enable out of order feature
function void disable_out_of_order()	Disable out of order feature
function void dump_data(uvm_reg_addr_t addr, int bytes, bit is2file = 0);	Dump data to a file

2. Appendix2. *How to use the uvm_reg_item in uvm_reg_adapter*

What's uvm_reg_adapter class provoides?	***How we use it per customized requirements?***
Function void m_set_item(uvm_reg_item item);	**We use this function to communicate between adapter and SSMM**
function uvm_reg_item get_item();	**We use this function to communicate between adapter and SSMM**
bit provides_responses;	Set this bit in extensions of this class if the bus driver provides separate response items. **We use this for post write in PCIe protocol.**
What's uvm_reg_item class provoides?	***How we use it per customized requirements?***
Uvm_elem_kind_e element_kind;	Kind of element being accessed: REG, MEM, or FIELD. See <uvm_elem_kind_e>. **We only use the UVM_MEM in this methodology.**
rand uvm_access_e kind;	Kind of access: READ or WRITE.
Uvm_reg_data_t value[];	The value to write to, or after completion, the value read from the DUT
uvm_reg_addr_t offset	For memory accesses, the offset address. For bursts, the ~starting~ offset address
uvm_status_e status	The result of the transaction: IS_OK, HAS_X, or ERROR
int prior = -1	The priority requested of this transfer **We use this item to support error injection feature.**
String bd_kind	If path is UVM_BACKDOOR, this member specifies the abstraction. Kind for the backdoor access, e.g. "RTL" or "GATES" **We use this item to support secure feature.**
String fname	The file name from where this transaction originated **We use this item to support error injection feature.**
int lineno	The file name from where this transaction originated **We use this item to support error injection feature.**
uvm_object extension;	Handle to optional user data, as conveyed in the call to write(), read(), mirror(), or update() used to trigger the operation.