Breaking Down Barriers: Achieving Seamless Protocol Conversion with UVM Component Layering

Name: Santosh Mahale
Organization: Marvell India Pvt. Ltd.
Job Title: Senior Staff Verification Engineer
Email ID: smahale@marvell.com
Mobile no: +91 7350686748

Name: Shantanu Lele
Organization: Marvell India Pvt. Ltd
Job Title: Senior Staff Verification Engineer
Email ID: slele@marvell.com
Mobile no: +91 9901908181

Table of Contents

- **Abstract:**

  Increasing computing needs and maturing technology changes demands change in how various blocks in System on chip (SoC) are connected. The Advanced Micro controller Bus Architecture (AMBA) bus protocols is an example of interconnect specifications from ARM that standardizes on chip communication mechanisms between various functional blocks (or IP) for building high performance SOC designs. Over the years AMBA protocols as well have undergone updates and some new protocols are also introduced on the way. Due to this even if an IP does not have significant logic change, to be compliant with latest SoC architecture, IP's input or output protocol gets updated. This calls for verification changes which must be done along with design update. Common proposal for this kind of change is to add a bridge to do conversion between two protocols. UVM component layering concept enables conversion of one complicated protocol transaction items into other type of protocol transaction items. With layering component, we can develop efficient reusable verification component to replace RTL block under development which does protocol conversion. This allows left shifting verification framework development effort which can reduces time to market with parallel development.

- **Related Work:**
  Consider a typical system where there is a processing element, usual IO devices, hardware accelerator to ease processing load and MMU to oversee memory management. Over the time processor architecture evolves and there is need of higher bandwidth communication between these elements. An example is recent ARM processor having native CHI interface compared to older ARM processors where ACE or AXI was native interface. To cater increased bandwidth needs interconnect and subsystem which are near to processor must upgrade to newer protocols to maintain high bandwidth flow eventually. Take an example where system shown in Figure 1 Typical system architecture has upgraded CPU which forces hardware accelerator IP also to upgrade to newer interface specification. Hardware accelerator subsystem adapt to this change by adding a protocol conversion module within IP.

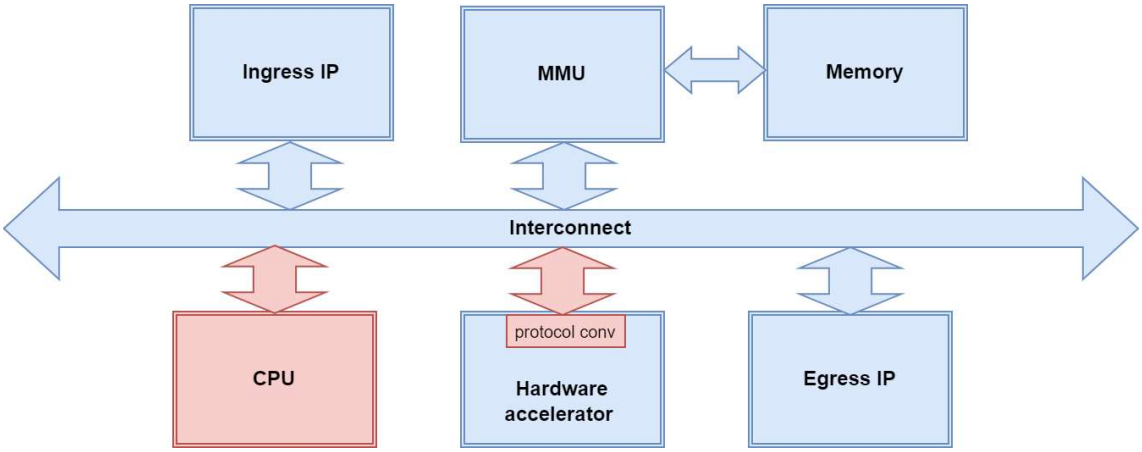

Figure 1 Typical system architecture

When Ips/Subsystems are upgraded in such a way, verification can be done in three ways.

1. We can verify subsystem without bridge/protocol converter, which means bridge verification must be done at full chip. This will increase risk of finding integration bugs at full chip which will be very late.
2. We can start verification with legacy IP only and later add bridge in subsystem testbench; which will duplicate work on input interface and all work done on legacy interface has to be thrown out.
3. The other approach is to use layering component which will take place of protocol convertor module and start developing verification collaterals such as tests & sequences which will allow bridge to be included in subsystem verification environment as soon as its ready without doing any redundant work.

- **Layering Component:**
  *A Layering component is slightly different component than usual UVM agent. Layered component allows verification engineer to code sequence in one protocol and execute it on completely different protocol driver. User sequence which will be coded on protocol **A** will be running on sequencer of type protocol **A**, instead of driving it on protocol **A** driver, UVM layering component first converts sequence item to protocol **B** and starts it on Protocol **B** sequencer which will eventually be run on protocol **B** driver.*

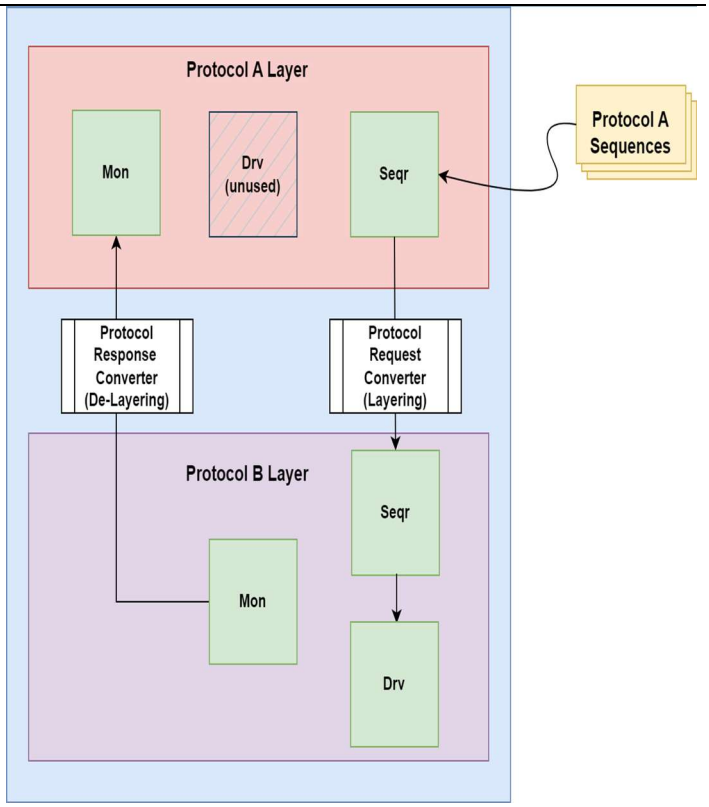

Figure 2 UVM Layering component

With layering component, we can start verification development on new protocol and sequence executed with new protocol will be converted to legacy protocol by layering component. Converted sequence is then executed on legacy sequencer. This way we can start verification without the availability of actual bridge module. All the test cases and sequence can be developed on new protocol. When Bridge development is completed, we can simply integrate bridge and start stressing it with already stabilized test & sequence suite.

- **Application:**

  Case Study 1: CHI2AXI Protocol Converter

CHI				AXI					
Opcode[6]	Opcode[5:0]	Opcode Field Value	Comment	READ	ArBURST	ArLOCK	Write	AwBURST	AwLOCK
0	4	4	ReadNoSnp	Y	INCR/FIXED	2'b00/2'b01	X	X	X
0	5	5	PCrdReturn	TERMINATED AT HN NO EFFECT on AXI(Credit management opcode)					
0	17	11	ReadNoSnpSep	Y	INCR/FIXED	2'b00	X	X	X
0	28	1C	WriteNoSnpPtl	X	X	X	Y	INCR/FIXED	2'b00/2'b01
0	29	1D	WriteNoSnpFull	X	X	X	Y	INCR/FIXED	2'b00/2'b01
1	4	44	WriteNoSnpZero	X	X	X	Y	INCR/FIXED	2'b00/2'b01

Table 1 CHI to AXI control attribute conversion

CHI									AXI							
MemAttr[1]	MemAttr[3]	MemAttr[2]	MemAttr[0]	SnpAttr	Likely Shared	Oder[1]	Order[0]	Comment	ArCache[3]	ArCache[2]	ArCache[1]	ArCache[0]	AwCache[3]	AwCache[2]	AwCache[1]	AwCache[0]
Device	Allocate	Cacheable	EWA													
1	0	0	0	0	0	1	1	Device nRnE	0	0	0	0	0	0	0	0
	0	0	1	0	0	1	1	Device nRE	0	0	0	1	0	0	0	1
	0	0	1	0	0	X	0	Device RE	0	0	0	1	0	0	0	1
0	0	0	0	0	0	X	0	Non-Cacheable Non Bufferable	0	0	1	0	0	0	1	0
	0	0	1	0	0	X	0	Non-Cacheable Bufferable	0	0	1	1	0	0	1	1
	0	1	1	0	0	X	0	Non-snoopable write back no-allocate	1	0	1	1	0	1	1	1
	1	1	1	0	0	X	0	Non-snoopable write back allocate	1	1	1	1	1	1	1	1

Table 2 CHI to AXI Memory attribute conversion

CHI		AXI
Order	Meaning for RN to HN	Meaning for AXI4 Request-Response
2'b00	NO Ordering Required	OOO
2'b01	RESERVED	RESERVED
2'b10	Request Order/Ordered Write Observation	INORDER/OOO
2'b11	Endpoint Order	INORDER/OOO

Table 3 CHI to AXI ordering attribute conversion

CHI		AXI	Comment
NS	Comment	AxPROT[1]	
1	non-secure	1	non-secure
0	secure	0	secure

Table 4 CHI to AXI protection conversion

CHI			AXI	Comment
Excl	Comment		AxLock	
1	Supported Opcodes with Exclusive 1. ReadNoSnp 2. WriteNoSnp		1	
0	Separate data & response are not supported with Excl as per protocol RespErr field in the response is used for conveying response for exclusive transaction		0	

Table 5 CHI to AXI Excl conversion

- **CHI2AXI Layering Architecture:**

  Figure 3 CHI to AXI Layering Internal Architecture shows the block diagram of CHI2AXI UVM layering component. This is complete reusable component which does conversion of CHI transaction items into AXI transaction items. CHI transactions are generated from CHI vip sequences and converted to AXI transactions from UVM layering component.

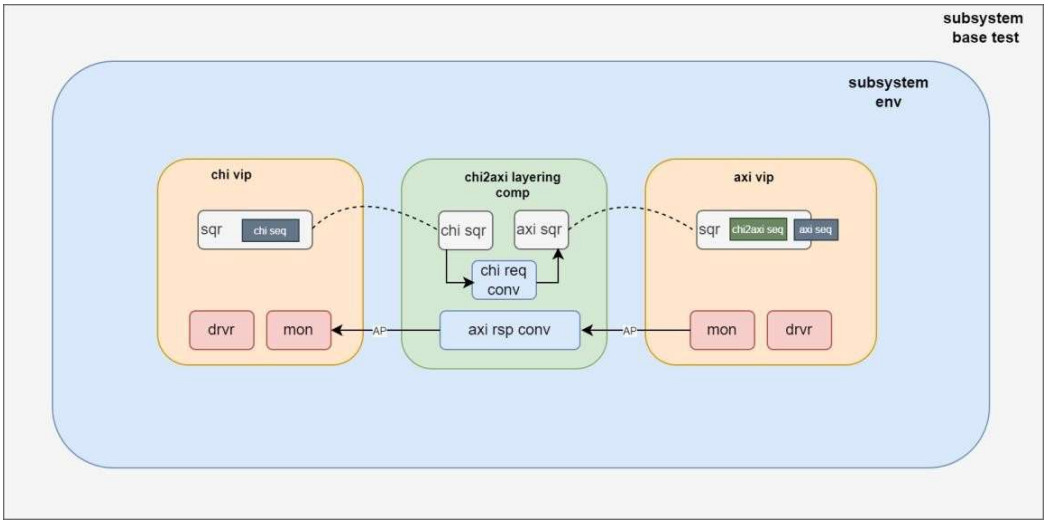

Figure 3 CHI to AXI Layering Internal Architecture

- **Flow Diagram:**

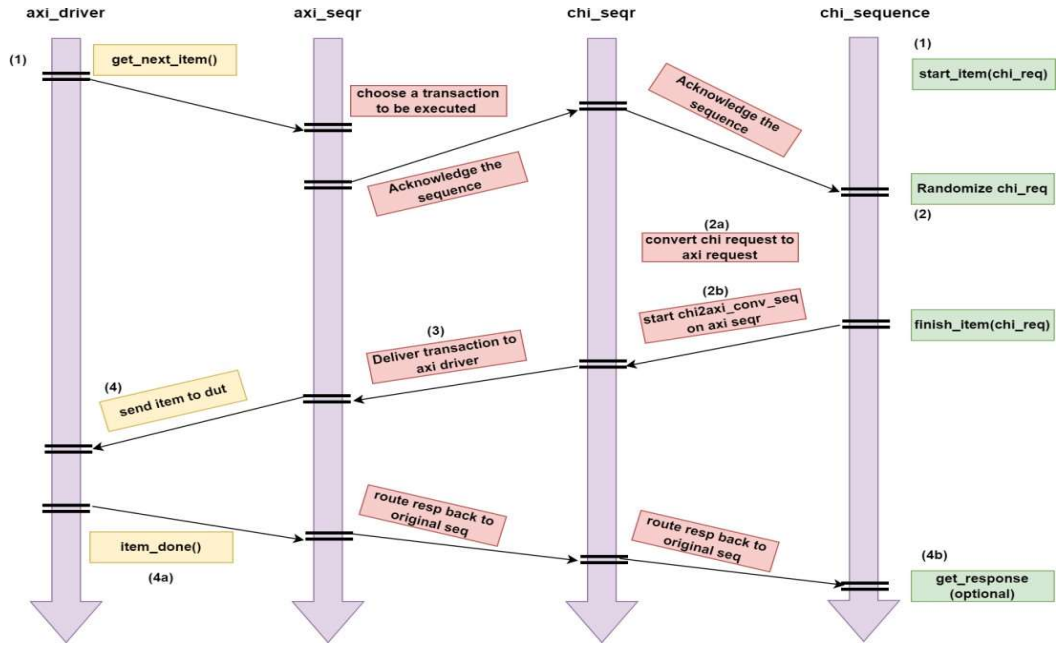

Figure 4 CHI to AXI Protocol Conversion Layering Flow

User will write all tests and sequence in CHI (newer)

1. At the start of run phase axi driver will be ready and waiting for sequence item to appear. Similarly, test will call a sequence which will randomize a sequence item.
2. A sequence item will be then start on chi sequencer which will first convert [(2a)] the CHI sequence item to AXI sequence item and instead of sending it to CHI driver will start converted axi sequence item on axi sequencer[(2b)].
3. Axi sequencer then pass on this item to driver which will drive the protocol level transaction.
4. Once done driver will inform sequencer completion of sequence item processing by calling item_done()[(4a)]. Optionally driver can also send response back to driver if get_response()[(4b)] is called by sequence.

- **Code Snippets:**
    - Test Case main phase:
      As shown in Figure 3 CHI to AXI Layering Internal Architecture, below main function where actual CHI sequence started in test case which is extended from subsystem base test.

```
virtual task main_phase(uvm_phase phase);
    phase.raise_objection(this);
```

```
            //start test sequence on the CHI sequencer inside layering component
            chi2axi_seq.start(env.layering_env.chi2axi_layering
            _comp.chi_sqr);
        phase.drop_objection(this);
    endtask : main_phase
```

- Top Level env :
  As shown in Figure 3 CHI to AXI Layering Internal Architecture, creating the Layering agent (chi2axi converter) handle in subsystem level env and setting the axi agent object required for the converter.

```
layering_env =
chi2axiact_pkg::env_c::type_id::create("layering_env",
this);
 layering_env.set_axi_agent(axi_master_agent);

//Analysis port connected to receive the response transaction from AXI VIP.
`AXI_RSP_CONNECT(axi_master_agent,layering_env)
```

- Layering Agent:
  As shown in Figure 3 CHI to AXI Layering Internal Architecture, chi2axi_layering_comp implementation :

```
class chi2axiact_layering_agent extends uvm_component;
    //Declaration of analysis port for chi/axi req/rsp transaction items
    uvm_analysis_port #(req_item_c) chi_req_ap;
    uvm_analysis_port #(rsp_item_c) chi_rsp_ap;
    uvm_analysis_imp_axi_rsp
    #(denaliCdn_axiTransaction,chi2axiact_layering_agent)
    axi_rsp_port;
    //local axi agent handle
    axi_agent_t axi_agent;

//Receiving axi rsp transactions from AP
virtual function void  write_axi_rsp(denaliCdn_axiTransaction
          axi_rsp);
   axi_trans_q.push_back(axi_rsp);
 endfunction

 virtual task run_phase(uvm_phase phase);
    chi2axiact_conv_seq chi2axi_seq;

    chi2axi_seq =
    chi2axiact_conv_seq::type_id::create("chi2axi_seq");

     // connect chi2axi conversion sequences to their respective upstream sequencers
      axi_sqr = axi_agent.seqr;
      forever begin
        chi_sqr.get_next_item(chi_transaction);

        chi2axi_seq.chi_trans = chi_transaction;
```

```
                // start the chi2axi conversion sequences
                 chi2axi_seq.start(axi_sqr);
                 chi_sqr.item_done();

                 // waiting for the axi response
                 wait(axi_trans_q.size());
                  $cast(axi_rsp_transaction,axi_trans_q.pop_front());
                 axi2chi_rsp_translator();
               end
        endtask
endclass
```

- Conversion Sequence:
  As shown in Figure 3 CHI to AXI Layering Internal Architecture, conversion sequence running in forever loop in Layering component class which is having actual conversion logic as per Table 1, Table 2, Table 3, Table 4, Table 5.

```
class chi2axiact_conv_seq extends uvm_sequence
#(denaliCdn_axiTransaction);
  `uvm_object_utils(chi2axiact_conv_seq);

  virtual task body();
      axi_trans =
denaliCdn_axiTransaction::type_id::create();
      start_item(axi_trans);
      chi2axi_req_translator();
      finish_item(axi_trans);
  endtask
endclass
```

- **Results:**
  - Initiated CHI Transaction:

```
LinkType                denaliChiLinkTypeT      32    DENALI_CHI_LINKTYPE_Rn2Hn
LinkInterface           denaliChiLinkInterfaceT 32    DENALI_CHI_LINKINTERFACE_FULL
Orientation             denaliChiOrientationT   32    DENALI_CHI_ORIENTATION_DOWNSTREAM
SpecVersion             integral                3     'h0
QoS                     integral                4     'ha
TgtID                   integral                11    'h1
SrcID                   integral                11    'ha
TxnID                   integral                12    'h1
ReqOpCode               denaliChiReqOpCodeT     32    DENALI_CHI_REQOPCODE_WriteNoSnpPtl
SnpOpCode               denaliChiSnpOpCodeT     32    DENALI_CHI_SNPOPCODE_SnpShared
Size                    denaliChiSizeT          32    DENALI_CHI_SIZE_WORD
Addr                    integral                64    'ha9ca6e430c8
SnpAddr                 integral                64    'h29f13b70f
NonSecure               integral                1     'h1
LikelyShared            integral                1     'h0
AllowRetry              integral                1     'h0
Order                   integral                2     'h0
MemAttr                 integral                4     'h0

Data                    da(integral)            4     -
  [0]                   integral                8     'h2b
  [1]                   integral                8     'hb
  [2]                   integral                8     'h7e
  [3]                   integral                8     'hd0
```

Figure 5 CHI Initiated Sequence Items

- Converted AXI transaction:

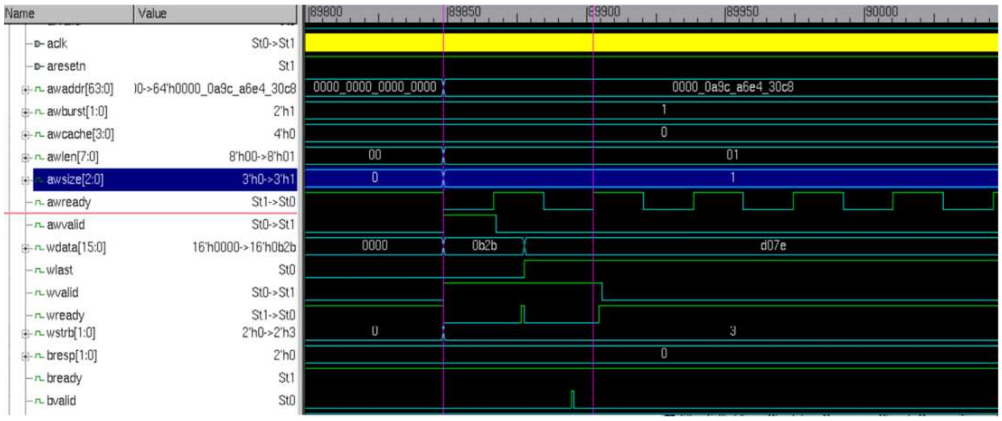

Figure 6 Converted AXI Interface fields

- **Case Study 2: Phy bypass component**

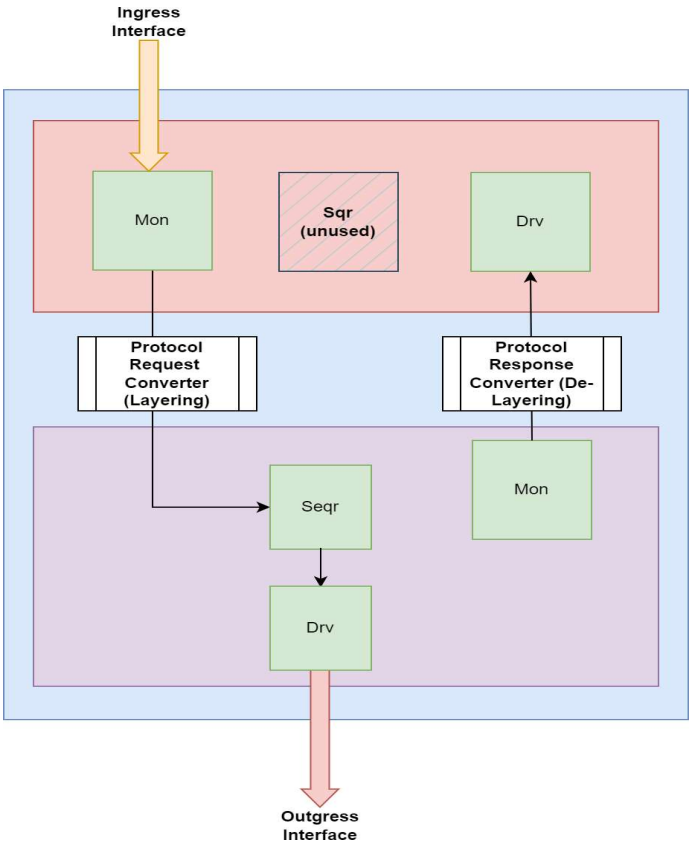

Figure 7 Phy bypass component

In typical SoC, for subsystems bring up; PHY block develops bottleneck for verification progress because long initialization and training time. PHY developers do provide bypass ideas but still it takes extra simulation time to get pass PHY module and do the actual transaction on output interface. UVM component layering can be used in such case to eliminate PHY complexity by catching PHY input transaction through a standard

monitor, which will call translation function to output protocol and driving converted transaction through output verification IP on output interface. With phy bypass component, we can simply skip allinitialization, training and calibration steps and directly go into data transfer phase.

- **Conclusions:**
  - UVM Component layering provides solution to develop completely reusable verification component.
  - UVM Component layering Left shift's verification cycle & providing opportunity to stress various scenarios at Sub-system level TB; while doing so, we can catch issues in incremental changes done is legacy designs.
  - Layering Component can be used at IP/Sub-System level without any limitation.
  - This methodology allows seamless integration of RTL as and when its ready. After availability of RTL, we can optionally use this Layering component as a transaction reference model.
  - This architecture can be extended to do conversion from any to any Bus Protocol and applications are endless.