A New Class Of Registers

M. Peryer
Mentor Graphics (UK) Ltd.,
Rivergate, London Road,
Newbury, Berkshire, RG14 2QB, United Kingdom

D. Aerne
Mentor Graphics Corp.,
8005 SW Boeckman Road,
Wilsonville, OR USA 97070- 7777

***Abstract*- The Universal Verification Methodology (UVM) register model provides a useful stimulus abstraction layer for register and memory access and for shadowing the content of hardware register content. However, the register model assumes the use of a simple parallel bus protocol, whereby a front door bus read or write is executed as a blocking transaction. This paper presents practical approaches to extending the reach of the register model abstraction layer to protocols which do not follow a simple completion model, based on the experience of developing a UVM library of Verification Intellectual Property (VIP).**

I. Introduction

The UVM register model is used to create a testbench side shadow of design hardware register content and to abstract accesses to registers and memory. The shadow model is built up from classes that describe memory regions or register hardware structures, encapsulating bit fields within registers, and registers within blocks. Memory blocks and registers are allocated address offsets within an address map model inside the block. The register model provides read() or write() tasks that can be called from a UVM sequence using either a memory or a register path, and an abstraction layer maps these into bus level transactions with the correct address. The shadow register model is kept up to date with the design hardware state based on the content of monitored analysis transactions for read or write accesses that are routed through a register predictor component. Memory accesses are not shadowed.

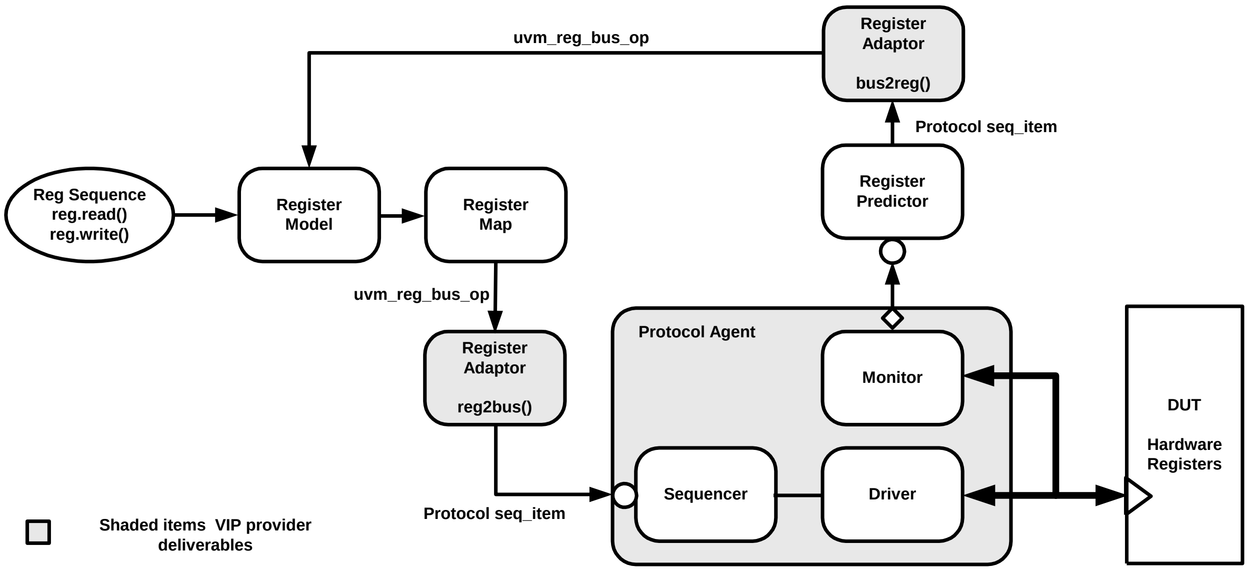

Figure 1 - Standard UVM register model integration

To integrate a register model into a UVM testbench, the user has to create a register model and then use an adapter class and a predictor to hook it into the testbench structure. This procedure is described in the UVM Cookbook[1]., and in the UVM User's Guide[2][3]. The adapter class is an extension of the uvm_register_adapter base class and it encapsulates the translation between a generic register read and write struct (uvm_reg_bus_op) to a target protocol agent specific sequence_item that is used to generate the bus transfer. The adapter is usually provided as part of the package for the bus agent by a VIP provider. Fig.1 is a representational diagram of the testbench integration of the UVM register model. The key SystemVerilog/UVM code fragments typically used in the integration procedure are shown in Fig.2.

```
// Function: build_phase
// Only elements shown are those used by register model integration
function void env::build_phase(uvm_phase phase);

  // Create and build the register model
  regs = test_reg_block::type_id::create("regs");
  regs.build();
  // Adapter for register bus
  reg2axi4 = reg2axi4_adapter #(axi4_rw_trans_t)::type_id::create("reg2axi4");
  // Register predictor
  reg_predictor = axi4_reg_predictor #(axi4_rw_trans_t)::type_id::create("reg_predictor", this);

endfunction

// Function: connect_phase
// Provides the register model map with a sequencer and a
// register bus adaptor
function void env::connect_phase(uvm_phase phase);

  // Assign target sequencer and adaptor to reg map
  regs.map.set_sequencer(env.master.m_sequencer, reg2axi4);
  // Assign register map and adaptor to the predictor
  reg_predictor.map     = regs.map;
  reg_predictor.adapter = reg2axi4;
  // Connect the predictor to the bus agent monitor analysis port
  env.master.ap.connect(reg_predictor.bus_item_export);

endfunction
```

Figure 2 - Key Code Fragments From Typical UVM Register Model Integration

The register layer abstracts the stimulus away from the protocol layer, which means that the stimulus can easily be retargeted to another protocol, as might happen during vertical stimulus reuse. The standard implementation and integration of the UVM register model assumes a blocking semantic for front door register or memory reads and writes. In other words, when a read() or a write() task call is made via the register model, it blocks further execution until the target protocol transfer completes. All bus protocols can support blocking memory and register transfers, but often this limits opportunities to improve testbench performance or to verify different aspects of the target protocol.

The most frequently encountered UVM register model limitation is the lack of support for burst access. For memory accesses, the UVM register model API supports burst reads and writes. From an API perspective this gives the appearance of supporting the transfer of blocks of words in a burst bus level transfer. Unfortunately, the register model breaks the bursts into a series of blocking single word transfers, since it cannot assume anything about the underlying bus transport layer.

Another example of its limitations would be using a register model as an abstraction layer on an I2C bus[4]. Since the I2C bus is serial, any read or write access will block for at least 18 I2C bus clock periods, and if the target device sends a NACK response, the access may have to be retried. A blocking semantic can be used in this case, with the NACK reflected back as the register access status field, but it means that the stimulus thread is blocked and has to handle the retry.

A more advanced example would be register and memory accesses over an AMBA® AXI™ bus[5]. The AXI protocol has separate read and write channels, allowing it to support concurrent read and write accesses. AXI also supports multiple outstanding transactions and slaves can provide out of order responses. AXI transfers can be supported using a blocking semantic, which would be fine for a simple register interface, although at higher levels of integration, the stimulus thread could be blocked whilst a front door access filters through several layers of interconnect, each adding several clock cycles of delay. In the case of memory mapped transfers the target design is unlikely be stressed and the testbench is likely to under-perform[6].

Other protocols, such as PCIe[7], send a request that can have several completion packets that have to be tied back to the originating request. In their native form, PCIe transfers frequently overlap with each other since the completion packets can arrive in any order and the PCIe root complex is free to send further requests to an endpoint.

The UVM register layer uses a uvm_reg_bus_op struct to generically describe a register or memory access. This struct only contains address, data and direction information whereas most advanced bus protocols use additional bus fields to qualify protocol accesses. For instance, the AXI protocol has bus fields associated with protection, caching and Quality of Service (QoS). Typically, the register adapter has to populate these fields with valid values, but it is difficult to set these to realistic values without maintaining system level context.

Fortunately, there are ways to extend the register model to allow different protocol aspects and advanced completion models to be supported.

II Extending the UVM register implementation

The UVM register implementation can be extended in several ways. The most obvious approach is to extend the uvm_reg base class, either by populating the various pre_/post_ write/read hook methods that are provided, or by registering call-backs. Both of these approaches are documented, but this approach has limited scalability. One of the original design issues with the UVM register model was to ensure that it was as light weight as possible to avoid consuming a vast amount of simulation process memory when modelling SoC scale register maps that could contain hundreds of thousands of registers. Extending the uvm_reg class hook methods, or adding call-backs will increase the footprint of the register model implementation. Despite these considerations, the fundamental problem with both these types of register model extension is that they are on the wrong side of the adaption layer.

The most effective way to implement an adaption layer for more advanced protocol semantics is to use the layered sequence design pattern[8]., to insert an additional protocol specific adaption layer between the register layer and the protocol agent sequencer, as shown in Fig.3. The main advantages of using the protocol adaption layering sequence are that it does not require any extensions to the standard register model; that there is a one-time cost in terms of memory overhead; and that as a long-running process it is able to keep track of system and protocol context allowing it to handle complex completion models.

The layered sequence contains a handle to an up-stream sequencer and uses its export methods to process up-stream protocol requests before sending them to the down-stream target sequencer, it then routes responses back to the up-stream sequencer. The protocol adaption layer receives uvm_reg_item sequence_items and converts them to the sequence_items for the target bus, this is broadly equivalent to the reg2bus() method of the uvm_reg_adapter, except that the uvm_reg_item is an object that contains some useful handles and more information on the transfer request than the simple uvm_reg_bus_op struct.

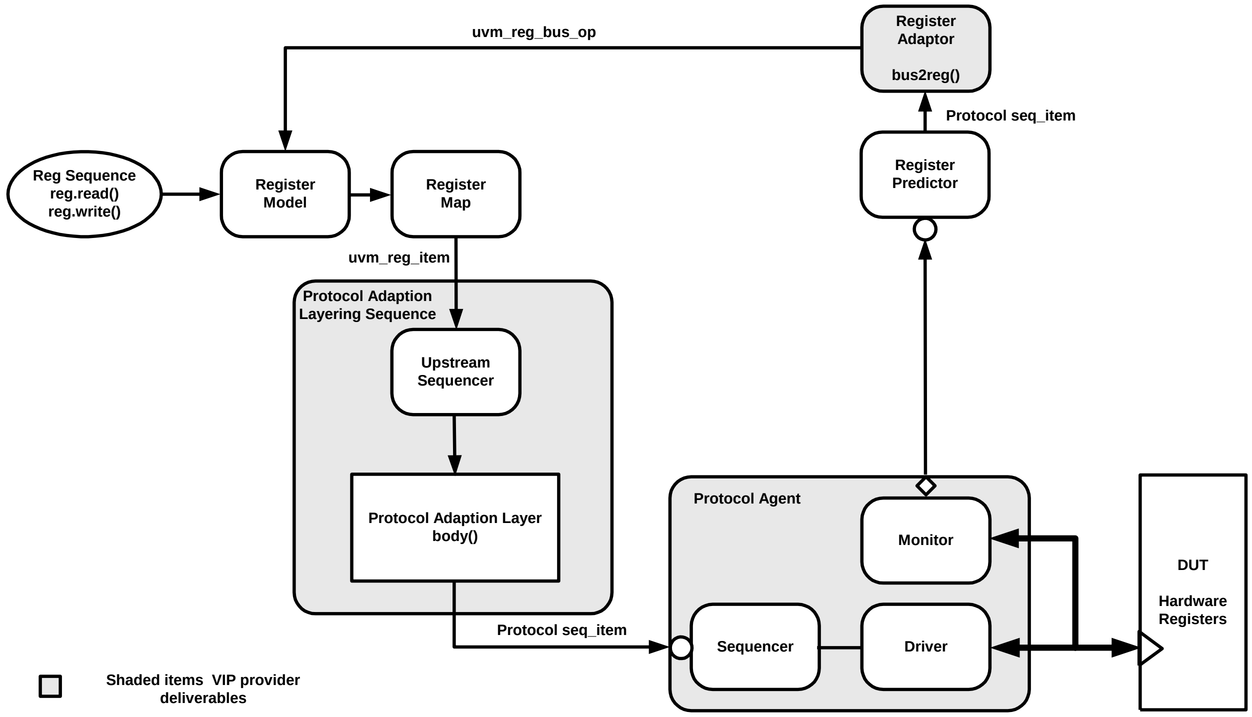

Figure 3 - Using a Protocol Adaption Layering Sequence With The Register Model

In order to use a protocol adaption layered sequence with the register layer, the integration of the register model has to be changed slightly. The adaption layer sequence and a uvm_sequencer parameterised with the uvm_reg_item has to be created during the UVM build phase, and a handle to the uvm_sequencer has to be assigned to upstream sequencer handle in the adaption layer sequence. During the UVM connection phase, the upstream sequencer has to be registered with the register model map using its set_sequencer() method, passing a null handle for the register adapter field. This last point is particularly important, since passing a null handle for the adapter means that the

UVM register model by-passes its normal adaption layer and sends a uvm_reg_item to the protocol layering sequence via its upstream sequencer. Finally, during the run_phase, the protocol adaption layering sequence has to be started on the protocol bus agent sequencer. These additions and modifications to the UVM testbench integration are illustrated in Fig.4.

```
// Function: build_phase
// Only elements shown are those used by register model integration
function void env::build_phase(uvm_phase phase);

  // Create and build the register model
  regs = test_reg_block::type_id::create("regs");
  regs.build();
  // Adapter for register bus
  reg2axi4 = reg2axi4_adapter #(axi4_rw_trans_t)::type_id::create("reg2axi4");
  // Register predictor
  reg_predictor = axi4_reg_predictor #(axi4_rw_trans_t)::type_id::create("reg_predictor", this);

  // Protocol adaption layering sequence up_stream sequencer component:
  reg_up_stream_sqr = uvm_sequencer #(uvm_reg_item)::type_id::create("reg_up_stream_sqr", this);
  // Protocol adaption layering sequence
  reg_layer_seq = axi4_reg_layer_seq #(PARAMS)::type_id::create("reg_layer_seq");
  // Assign up_stream sequencer handle
  reg_layer_seq.up_stream_sqr = reg_up_stream_sqr;

endfunction

// Function: connect_phase
// Provides the register model map with a sequencer and a
// register bus adaptor
function void env::connect_phase(uvm_phase phase);

  // Assign target sequencer to reg map, leaving adaptor handle as a null assignment
  regs.map.set_sequencer(reg_up_stream_sqr, null);
  // Assign register map and adaptor to the predictor
  reg_predictor.map     = regs.map;
  reg_predictor.adapter = reg2axi4;
  // Connect the predictor to the bus agent monitor analysis port
  env.master.ap.connect(reg_predictor.bus_item_export);

endfunction

// Task: run_phase
// During the run_phase, the protocol adaption layering sequence is started on the
// protocol agent sequencer
task env::run_phase(uvm_phase phase);

  reg_layer_seq.start(axi4_master.m_sequencer);

endtask
```

Figure 4 - Adaption Layer Integration (Additional and modified code in bold)

All other connections and components, including the predictor remain as per the normal UVM register integration. An example of using a protocol adaption layering sequence to the AXI protocol follows in the next section.

III Applying the extensions to the AXI protocol

A. AXI Protocol Overview

The AXI protocol has independent read and write channels and allows slaves to accept multiple requests on each channel and to respond to them in any order. The AXI protocol also supports burst transfers, allowing for efficient transfers of block data. In the case of read responses, the individual read beat responses from different threads can be interleaved. In order to stress an AXI memory interface it is important to be able to generate overlapping memory transfers, and this requires a non-blocking completion model for memory accesses made from a register sequence.

The default behavior for the AXI, protocol adaption layer is that it handles AXI reads and writes as blocking transactions. This means that only one, single word, read or write can take place on the bus at any point in time.

Memory region verification can easily be enhanced by using the protocol adaption layer sequence to take multiple word transfers from the register model burst_read/write() API and convert these to a burst transfer making block memory transfers much more realistic and efficient. The protocol layering sequence can also enable the support of non-blocking read and write transactions, using a SystemVerilog fork join_none construct to spawn a new thread. In the case of non-blocking writes, the response can be ignored, a process referred to as posting a write. In the case of non-blocking reads the user has to make a choice, either to receive the read response in the register sequence or to ignore it.

In the register region, a common verification scenario is that a hardware block has to be initialized by writing to a set of registers before it can be used. This type of initialization can be done via the UVM register model by making a number of set() calls to the configuration registers to make the initial register settings and then calling the update() method to complete the initialization with a series of writes to the modified registers. This process can be enacted with the register models default blocking completion model, but it can also extended to use a non-blocking completion semantic, allowing the stimulus thread to move on to other processing. In both blocking and non-blocking cases, the register model will be updated via the predictor and the protocol adaption layer handles the AXI sequence_items when they complete.

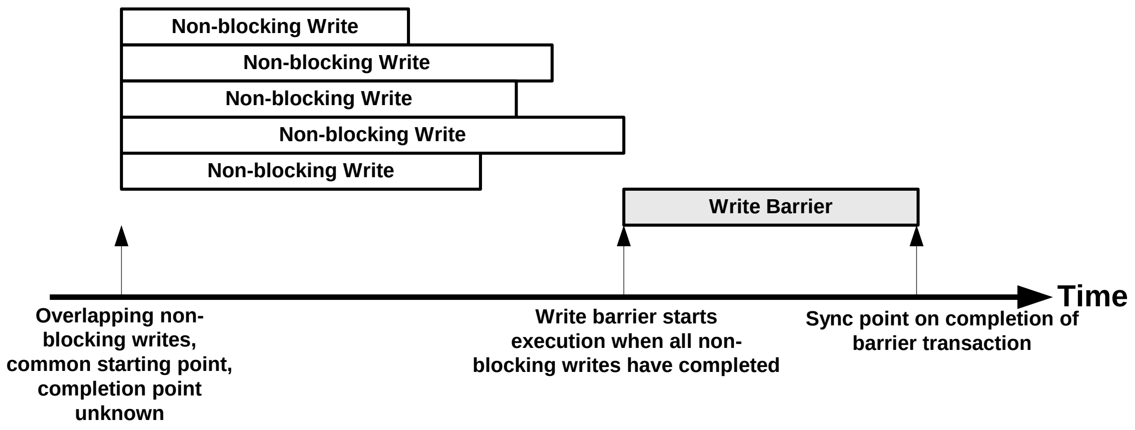
Non-blocking Write
Non-blocking Write
Non-blocking Write
Non-blocking Write
Non-blocking Write
Write Barrier
Time
Overlapping non-blocking writes, common starting point, completion point unknown
Write barrier starts execution when all non-blocking writes have completed
Sync point on completion of barrier transaction

Figure 5 - Write Barrier Transaction

When non-blocking write transfers are made, a stream of overlapping write transactions is sent to the hardware device. The transactions may be re-ordered by an on-chip bus interconnect or even re-ordered within the target hardware slave. The calling sequence has no visibility of the order in which the writes complete, or when they have all finished, therefore a barrier transaction is required to ensure that all the non-blocking writes have completed before progressing further with the test. Fig.5. illustrates the principle of a barrier transaction. To implement a barrier transaction, the protocol adaption layer has to keep track of all transactions in progress and only complete the barrier transaction when the transaction queue is empty.

In common with other protocols, the AXI bus has a number of fields that qualify transactions as having characteristics such as those related to protection, caching behavior and quality of service. The UVM register model is at too high a level of abstraction to be able to set these fields, therefore the field values are often set in the register adapter class during the conversion from the generic register struct to the bus specific sequence_item. When using the adaption layer, more application context information can be applied using resources such as a system address map that specify the valid settings for each of the bus fields. This is particularly important when dealing with systems that have secure locations with access protection mechanisms.

An example of how an AXI specific protocol adaption layering sequence would be implemented is described in the following section.

B. Implementation

The protocol adaption layering sequence receives uvm_reg_items from the register model via its up_stream sequencer. The uvm_reg_item contains a number of useful handles that enable interaction with the calling register sequence.

The register front-door API allows an extension object to be associated with any read() or write() call, and a handle to this object is available from the uvm_reg_item. The extension object can be used to encode whether the bus transaction semantic associated with the register layer read or write should be blocking, non-blocking or a barrier. The extension object can also be used to send other protocol specific information through to the adaption layer. As an example Fig.6 shows how such an extension object is implemented and how it might be used within a register sequence to specify different types of target bus accesses.

The uvm_reg_item also contains the handle of the parent register sequence which means that the adaption layer has the ability to return out of order response information back to the register sequence, either via the uvm_sequence_base class handle_response() method or by calling some other sequence specific method.

```
// typedef used within the extension object:
typedef enum {BLOCKING, NON_BLOCKING, BARRIER} axi4_access_e;

// Class: access_obj
// Used to extend the protocol in the register layer
class access_obj extends uvm_object;

axi4_access_e access_mode; // Determines completion semantic
int qos;                   // The QoS level

endclass: access_obj
//
// Examples of using the extension object within the register sequence:
//
uvm_status_e status;
uvm_reg_data_t wdata;
uvm_reg_data_t wburst[];
uvm_reg_data_t rdata;
uvm_reg_data_t rburst[];
access_obj extension = access_obj::type_id::create("extension");

reg_model.reg.write(status, wdata, .parent(this)); // Standard blocking write
reg_model.reg.read(status, rdata, .parent(this));  // Standard blocking read
extension.access_mode = NON_BLOCKING;
reg_model.reg.write(status, wdata, .parent(this), .extension(extension)); // Non-blocking write
extension.access_mode = BARRIER;
reg_model.reg.read(status, rdata, .parent(this), .extension(extension)); // Read barrier
extension.access_mode = NON_BLOCKING;
extension.requires_resp = 1;
extension.qos = 8;
rburst = new[16];
// Non-blocking read 16 word burst, returns response via response handler, allocated a QoS of 8
reg_model.mem.burst_read(status, `h200, rburst, .parent(this), .extension(extension));
wburst = new[8];
foreach(wburst[i]) begin
  wburst[i] = i + 5000;
end
// Non-blocking write 8 word burst, will overlap previous read, allocated a QoS of 8
reg_model.mem.burst_write(status, `h400, wburst, .parent(this), .extension(extension));
```

Figure 6 - Implementation And Useage Of Register Extension Object

Within the layering sequence, the body task is responsible for getting uvm_reg_items from the up_stream sequencer and converting them into AXI sequence_items, according to any protocol or completion semantic options present in the extension object. The first step in the process is to get hold of the address for the transfer. The second step is to split the transfer into either a read or a write transfer.

For a write transfer, a write transaction is created and its dynamic arrays for the write data, write data strobes and the write user data have to be sized to the number of words in the transfer and the burst length set, for a register write, the size would be ==1, for a memory write burst the size would be >= 1. The write data is copied into the write transaction from the uvm_reg_item. Then other fields in the write transaction such as AWPROT, AWCACHE and AWQOS are assigned values based on a lookup in the VIP address map object; from information in the extension object; or based on other context information available within the layering sequence. The final part of the adaption process is to schedule the transaction according to the semantic defined in the extension object. If the extension object handle is null, then a blocking transfer takes place. If the extension object handle is valid, then access options for non-blocking, barrier and blocking completion semantics are processed. For a non-blocking write access, the write transaction is pushed into a queue before a fork join_none block where the finish_item() method is called on the write transaction and when it unblocks, the transaction is deleted from the queue. Using the queue in this way allows the number of outstanding writes to be monitored. The barrier semantic option waits for the write transaction queue size to be zero before scheduling a blocking write transaction. (Note that the barrier transaction could also have been implemented as a wait for the write transaction queue to be of zero length but this does not necessarily tie in with the register model functionality). The blocking semantic option is included for completeness.

```
  // Function: response_handler
  // Takes non_blocking read response items and pushes them into a queue for processing
  //
  function void response_handler(uvm_object response);
    uvm_reg_item t;

    $cast(t, response);
    read_response_q.push_back(t); // Other options include populating a sparse memory array
  endfunction
```

Figure 7 - Register Sequence Response Handler

A read transfer is handled in a similar way to a write transfer, the main difference being that the read transfer does have a result that will be stored in the AXI sequence_item read data array. For a blocking or barrier semantic, the value of the read data array is copied back to the uvm_reg_item value array so that it can be handled in the calling sequence. In the case of a non-blocking memory read, a copy is made of the uvm_reg_item so that its value array can be assigned the content of the returned read data when the AXI read transaction completes. The uvm_reg_item is then returned to the parent register sequences response_handler() method. (See Fig.7. for a simple example of how such a response_handler() function might be implemented in the parent register sequence). This mechanism is used since the upstream sequencer would have deleted the uvm_reg_item from its queue by the time the AXI read transaction completes. Register reads update the register model using the prediction path, regardless of completion model, therefore any register value can be read from the register model using the get_mirrored_value() method, therefore register reads do not result in a call to the response handler.

When either the write or the read transaction handling completes, the item_done() call is made on the up_stream sequencer and the body() method loop starts its next iteration.

The code for the protocol adaption layering sequence is listed in Fig.8., Fig.9 and Fig.10.

```
  // Class: axi4_reg_adaption_layer_seq
  // AXI4 protocol adaption layering sequence example
  class axi4_reg_adaption_layer_seq extends uvm_sequence;

  // Upstream sequencer to which the register map is attached
  uvm_sequencer #(uvm_reg_item) up_stream_sqr;

  // Transaction queues that need to be cleared down for barriers
  read_t  read_q[$];
  write_t write_q[$];

  // System level address map
  addr_map address_map;

  extern task body;


  endclass: axi4_reg_adaption_layer_seq
```

Figure 8 - AXI4 Protocol Adaption Layering Sequence Class

```
task axi4_reg_adaption_layer_seq::body;
  access_obj access_key;
  uvm_reg_item item;
  uvm_reg_addr_t addr;
  uvm_reg_addr_t addr_array[];
  uvm_reg target_reg;
  uvm_mem target_mem;
  bit[3:0] wr_id;
  bit[3:0] rd_id;

  forever begin
    up_stream_sqr.get_next_item(item);
    if(item.element_kind == UVM_REG) begin
      $cast(target_reg, item.element);
      addr = target_reg.get_address(item.map);
    end
    else begin
      $cast(target_mem, item.element);
      addr = target_mem.get_address(.map(item.map)) + item.offset;
    end
    if((item.kind == UVM_WRITE) || (item.kind == UVM_BURST_WRITE)) begin // Write adaption
      write_t write_txn = write_t::type_id::create;
      write_txn.addr = addr;
      write_txn.data_words = new[item.value.size()];
      write_txn.write_strobes = new[item.value.size()];
      write_txn.wdata_user_data = new[item.value.size()];
      foreach(write_txn.data_words[i]) begin
        write_txn.data_words[i] = item.value[i];
        write_txn.write_strobes[i] = '{1,1,1,1};
      end
      write_txn.burst_length = item.value.size() - 1;
      write_txn.burst = AXI4_INCR;
      write_txn.size = AXI4_BYTES_4;
      write_txn.id = wr_id;
      wr_id++;
      write_txn.prot = address_map.get_prot(addr);
      write_txn.cache = address_map.get_cache(addr);
      if(item.extension == null) begin // Standard register access
        write_txn.qos = 1;
        start_item(write_txn);
        finish_item(write_txn);
      end
      else begin
        $cast(access_key, item.extension);
        write_txn.qos = access_key.qos;
        case (access_key.access_mode)
          NON_BLOCKING: begin
                          start_item(write_txn);
                          write_q.push_back(write_txn);
                          fork
                            begin
                              finish_item(write_txn);
                              foreach(write_q[i]) begin
                                if((write_q[i].addr == write_txn.addr)
                                   && (write_q[i].id == write_txn.id)) begin
                                  write_q.delete(i);
                                  break;
                                end
                              end
                            end
                          join_none
                        end
          BARRIER: begin
                     wait(write_q.size() == 0);
                     start_item(write_txn);
                     finish_item(write_txn);
                   end
          BLOCKING: begin
                     start_item(write_txn);
                     finish_item(write_txn);
                   end
        endcase
      end
    end
```

Figure 9 - AXI Protocol Adaption Layering Sequence body() method implementation (Part 1)

```
     else begin // Read adaptation
        read_t read_txn = read_t::type_id::create;
        read_txn.addr = addr;
        read_txn.data_words = new[item.value.size()];
        read_txn.resp_user_data = new[item.value.size()];
        read_txn.burst_length = item.value.size() - 1;
        read_txn.burst = AXI4_INCR;
        read_txn.size = AXI4_BYTES_4;
        read_txn.id = rd_id;
        rd_id++;
        read_txn.prot = address_map.get_prot(addr);
        read_txn.cache = address_map.get_cache(addr);
        if(item.extension == null) begin  // Standard register access
          read_txn.qos = 1;
          start_item(read_txn);
          finish_item(read_txn);
          foreach(read_txn.data_words[i]) begin
            item.value[i] = read_txn.data_words[i];
          end
        end
        else begin
          $cast(access_key, item.extension);
          read_txn.qos = access_key.qos;
          case (access_key.access_mode)
            NON_BLOCKING : begin
                             start_item(read_txn);
                             read_q.push_back(read_txn);
                             fork
                               begin
                                 automatic uvm_reg_item tmp;
                                 tmp = item;
                                 finish_item(read_txn);
                                 foreach(read_q[i]) begin
                                   if((read_q[i].addr == read_txn.addr)
                                      && (read_q[i].id == read_txn.id)) begin
                                     read_q.delete(i);
                                     break;
                                   end
                                 end
                                 // Only return data if the access is to a memory location
                                 if(tmp.kind == UVM_MEM) begin
                                   foreach(read_txn.data_words[i]) begin
                                     tmp.value[i] = read_txn.data_words[i];
                                   end
                                   tmp.offset = read_txn.addr;
                                   tmp.parent.response_handler(tmp);
                                 end
                               end
                             join_none
                           end
            BARRIER: begin
                       wait(read_q.size() == 0);
                       start_item(read_txn);
                       finish_item(read_txn);
                       foreach(read_txn.data_words[i]) begin
                         item.value[i] = read_txn.data_words[i];
                       end
                     end
            BLOCKING: begin
                        start_item(read_txn);
                        finish_item(read_txn);
                        foreach(read_txn.data_words[i]) begin
                          item.value[i] = read_txn.data_words[i];
                        end
                      end
          endcase
        end
      end

      up_stream_sqr.item_done(); // Unblock the calling parent register sequence
```

Figure 10 - AXI Protocol Adaption Layering Sequence body() method implementation (Part 2)

IV Applying the register extensions to PCIe

The PCIe protocol uses different categories of memory space which are mapped as part of the system level initialization that results from the PCIe bus enumeration process. Different types of transfer request packets are used when accessing configuration space, IO space or 32 bit or 64 bit memory space. Memory access packets have the scope to transfer large numbers of words, and the target of the request may return several completion packets as the result of a single request. For instance a PCIe read of 4K words could result in four completion packet responses each containing 1K words. In the protocol, the requests are decoupled from the completions so the requests can overlap with each other and their associated completion packets can be interleaved.

If the standard form of register adaption layer is used, then the register adapter has to be configured with the memory mapping for each PCIe endpoint in the system in order to form the right type of transfer packet. Since PCIe is typically used to transfer blocks of memory, the register model burst limitation breaks block transfers down into single word transfers, making for inefficient and ineffective verification.

Using a PCIe specific adaption layering sequence means that the entire PCIe system memory map can be taken into account, and that memory transfers can be made using the full payload specified in the memory read or write burst API. The user can also use an extension object to specify whether write transfers are non-blocking and posted, or whether a read transfer is blocking and whether the resultant data is required or not.

In the PCIe protocol, there are other layers that deal with the handling of credit tokens to throttle transfers. The protocol layering sequence can be used to handle the transfer of credits and to take credit levels into account when requesting transfers. This impacts the accuracy of the performance modelling of PCIe transfers, and can be implemented without the register sequence user having to know. However, the extension object can be used to interact with the credit token handling, allowing the potential to introduce errors at that level.

V Conclusion

The standard implementation of the UVM register layer provides a useful abstraction layer for users wishing to write reuseable register and memory mapped stimulus. However, this implementation is inefficient for burst transfers and exercises only a limited sub-set of the functionality of all but the simplest of bus transport layers. Using a protocol adaption layering sequence is a practical way of overcoming these limitations as illustrated by the AXI example and by the PCIe example discussion.

Acknowledgment

I should like to acknowledge the critical feedback that I have received on the content of this paper from my colleagues in the Questa Verification IP development team.

References

[1] Verification Academy – UVM Cookbook: https://verificationacademy.com/cookbook/registers/integrating.
[2] Universal Verification Methodology (UVM) 1.1 Users Guide – Accellera, May 18, 2011
[3] Universal Verification Methodology (UVM) 1.2 Users Guide – Accellera, October 8, 2015
[4] I2C bus specification and user manual – Rev.6, - NXP Semiconductors, 4 April 2014
[5] AMBA AXI and ACE Protocol Specification - ARM - IHI 0022E (ID022613) – February 22, 2013
[6] A. Yehia, "Boosting Simulation Performance of UVM registers in high performance systems" – DVCon 2013 Proceedings
[7] PCI Express 4.0 Base Specification – PCI SIG – February 19, 2014
[8] M. Peryer, "Seven Separate Sequence Styles Speed Stimulus Scenarios," – DVCon 2013 Proceedings