2023 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION INDIA

UVM Sequence Layering for Register Sequences

Effective Reusability

Muneeb Ulla Shariff, Mirafra Technologies Pvt Ltd, Bangalore, Karnataka, India
(muneebullashariff@mirafra.com)

Sangeetha Sekar, Mirafra Technologies Pvt Ltd, Bangalore, Karnataka, India
(sangeethasekar@mirafra.com)

Ravi Reddy, Roche Sequencing Solutions, Santa Clara, California, USA
(ravi.reddy@roche.com)

***Abstract*—Universal Verification Methodology (UVM) Sequence Layering enables protocol-independent test scenario development by adding an intermediary layer between the sequence and sequencer flow. The additional layer allows higher-level coding, enhances code reusability, and scalability. This paper showcases the application of UVM Sequence Layering on RAL register sequences, leveraging Cadence VIP, and demonstrates two use cases for PCIe and SPI protocols. The implementation of adapter layer sequence integration required minor RAL model modifications. With the introduction of the PCIe and SPI adapter layer sequences, finer-granularity was achievable with existing register sequences, resulting in faster verification turnaround time with minimal additional effort.**

Keywords—UVM(Universal Verification Methodology); RAL(Register Abstraction Layer); VIP(Verification Intellectual Property)

I. Introduction Of UVM Sequence Layering

Sequence layering facilitates the development of test scenarios that are protocol independent. The high level sequence is protocol-independent and the protocol conversion can be done by the intermediate layer along with additional processing, before passing it to the lower protocol-level sequencer, which pushes them forward to the driver.

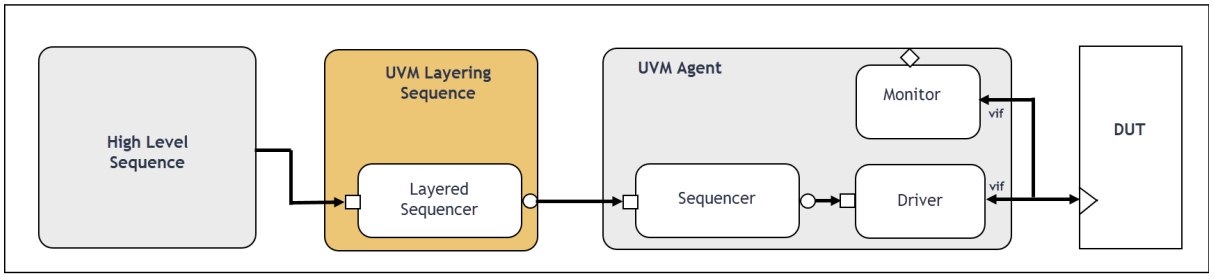

Figure 1. Concept Diagram for UVM Sequence Layering

We can apply this concept while working with the RAL model, as depicted in Figure 2. The protocol-adapter-layer sequence replaces the traditional RAL adapter's reg2bus() function and converts the register transactions to the protocol specific sequences, handles complicated processing and then starts the protocol-specific sequence(s) on the protocol-agent's sequencer.

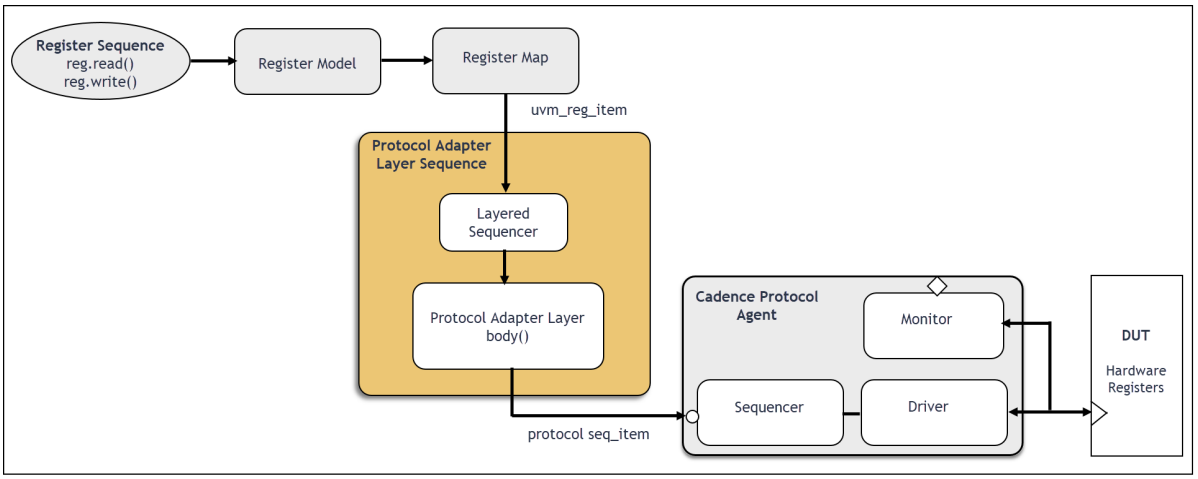

Figure 2. Protocol Adapter Layering Sequence with the RAL Model

II. IMPLEMENTATION

1. In order to use a protocol adapter layer sequence with the register layer, the integration of the RAL model has to be changed slightly.

```
//------------------------------------------------------------------------------
// Class: user_env
// User environment class
//------------------------------------------------------------------------------
class user_env extends uvm_env;
  `uvm_component_utils(user_env)
  ....
  ....
  // layered sequencer
  uvm_sequencer #(uvm_reg_item) reg_layered_sqr;

  // adapter layer sequence
  user_reg_adapter_layer_seq user_reg_layer_seq;

  //------------------------------------------------------
  // Function: build_phase
  // Used for creating the required components
  //------------------------------------------------------
  function void build_phase(uvm_phase phase);
    ...
    ...
  // creating the user regmodel
    user_regmodel_h = user_regmodel::type_id::create("user_regmodel", this);
    user_regmodel_h.build();
    user_regmodel_h.lock_model();
```

Creating the RAL model

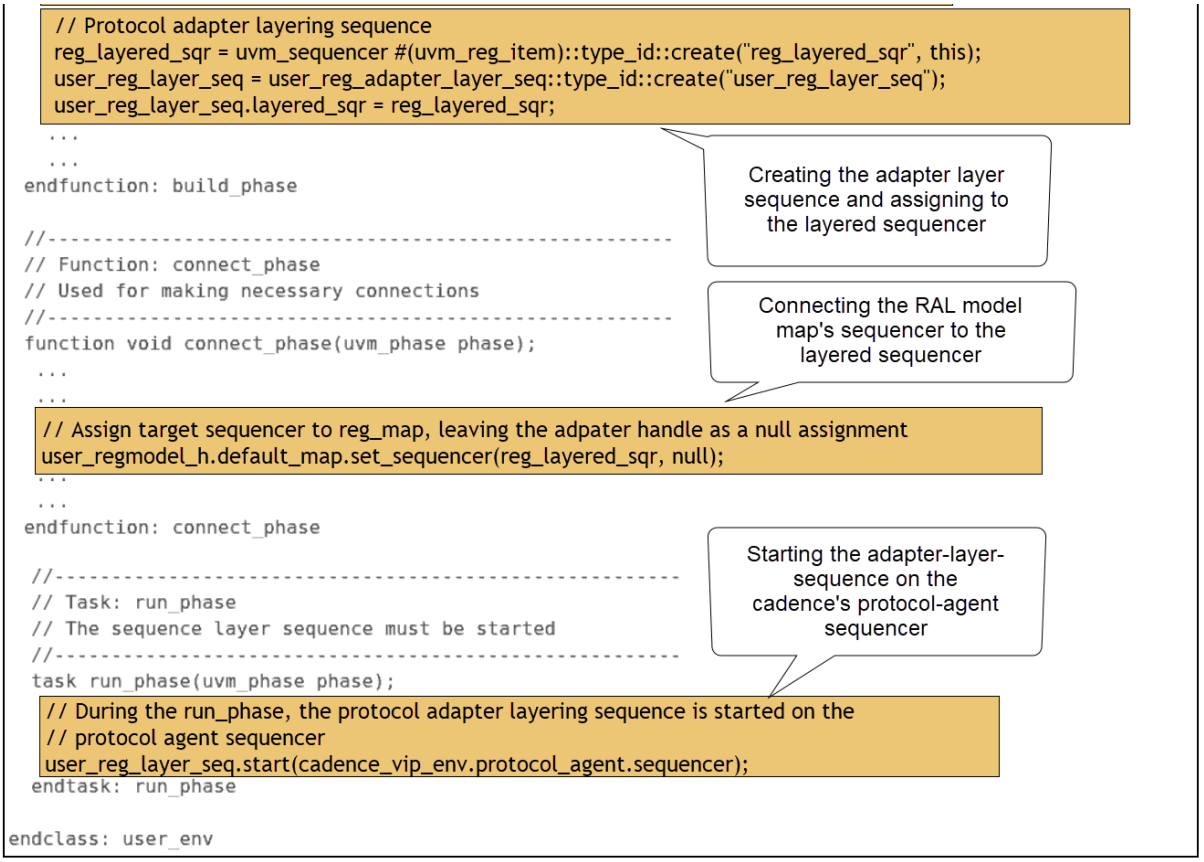

Figure 3. Adapter Layer Sequence Integration (Showing only the additional and modified code) [1]

2. ***Use case for PCIe protocol:*** The user_reg_layer_seq is a protocol based sequence that the user has to write. As shown in Figure 4, pcie_reg_adapter_layer_seq is the user_reg_layer_seq written based on the PCIe protocol in perspective. [2][3]

```
//------------------------------------------------------------------------------
// Class: pcie_reg_adapter_layer_seq
// Sequence layer for replacement for adapter's reg2bus function
//------------------------------------------------------------------------------
class pcie_reg_adapter_layer_seq extends pcie_link_base_seq;
  `uvm_object_utils (pcie_reg_adapter_layer_seq)

  // Variable: layered_sqr
  // Used for getting  the uvm_reg_item from reg_sequence
  uvm_sequencer #(uvm_reg_item) layered_sqr;

  uvm_reg_item item;
  uvm_reg target_reg;
  uvm_reg_addr_t reg_addr;
  uvm_reg_data_t reg_wr_data;
  uvm_reg_data_t reg_rd_data;

  // Variable: reg_write_seq
  // Used for PCIe MMIO register writes, which are posted writes
  blocking_mem_write_seq reg_write_seq;

  // Variable: reg_read_seq
  // Used for PCIe MMIO register reades, which are non-posted reads
  blocking_mem_read_seq reg_read_seq;

 // Constructor
 function new(string name = "pcie_reg_adapter_layer_seq");
    super.new(name);
 endfunction: new

 // Body Task
 task body();

   `uvm_info(get_type_name(),$sformatf("Inside th  cie_reg_adapter_layer_seq\n"), UVM_LOW);

   forever begin

     // Getting the uvm_reg_item from reg_sequence via layered sequencer
     layered_sqr.get_next_item(item);

     `uvm_info(get_type_name(),$sformatf("Got the item from reg sequence\n"), UVM_HIGH);

     if(item.element_kind == UVM_REG ) begin
         $cast(target_reg, item.element);
     end
     else `uvm_fatal(get_type_name(),"Its not an reg access");

     // Get the address and data
     reg_addr = target_reg.get_address(item.map);
     reg_wr_data = item.value[0];
```

Cadence PCIe Base Sequence

Get the item from reg.write() or reg.read() methods

```
// PCIe MMIO register read
if (item.kind == UVM_READ) begin
  begin
     `uvm_do_with(reg_read_seq,{
        function_nb == 0;
        bar_nb == 0;
        // offset address inside BAR0
        address == reg_addr;
     })
  end
  // Get the read data from payloadAccumulated array
  // Each iteration gets 8bits of read data
  for (int i=0;i<reg_read_seq.payloadAccumulated.size();i++) begin
     reg_rd_data[(i*8) +: 8] = reg_read_seq.payloadAccumulated[i];
  end

  // Returning back the read value to reg sequence
  item.value[0] = reg_rd_data;
end
```

Starting the Cadence PCIe non-posted read sequence

```
// PCIe MMIO register write
else begin
  begin
     `uvm_do_with(reg_write_seq,{
        function_nb == 0;
        bar_nb == 0;
        // offset address inside BAR0
        address == reg_addr;
        // 32 bits data = 4bytes
        Data.size() == 4;
        Data[0] == reg_wr_data[7:0];
        Data[1] == reg_wr_data[15:8];
        Data[2] == reg_wr_data[23:16];
        Data[3] == reg_wr_data[31:24];
     })
  end
end
```

Starting the Cadence PCIe posted write sequence

```
// Returning back control to the layered sequencer
layered_sqr.item_done();
end

endtask: body

endclass : pcie_reg_adapter_layer_seq
```

Returning back the control to the reg.read() or reg.write() methods

Figure4: PCIe Protocol Adapter Layer Sequence Implementation With Cadence PCIe Sequences

3. ***Use case for SPI protocol:*** In our design, every register accessed via the SPI interface should be a 80 bits transfer. In other words, every reg.write() or reg.read() needs to be converted to a 80 bits SPI transfer. In-order to support byte-granularity we need to send multiple transactions but the traditional RAL adapter's reg2bus() function will be called only once for each reg.write() or reg.read(). Hence, using the SPI adapter layer sequence, we can achieve byte-granularity with existing register sequences.[4]

```
//------------------------------------------------------------------------------
// Class: spi_reg_adaption_layer_seq
// Sequence layer for replacement for adapter's reg2bus function
//------------------------------------------------------------------------------
class spi_reg_adaption_layer_seq extends cdnSpiUvmSequence;
  `uvm_object_utils (spi_reg_adaption_layer_seq)

  // Variable: layered_sqr
  // Used for getting  the uvm_reg_item from reg_sequence
  uvm_sequencer #(uvm_reg_item) layered_sqr;

  uvm_reg_item item;
  uvm_reg target_reg;
  uvm_reg_addr_t reg_addr;
  uvm_reg_data_t reg_wr_data;
  uvm_reg_data_t reg_rd_data;
  // 80bits = 10bytes
  bit[7:0] spi_mosi_data_array[10];
  bit[7:0] spi_miso_data_array[10];

  // Variable: reg_write_seq
  // Used for SPI write transfers
  spi_write_seq reg_write_seq;

  // Variable: reg_read_seq
  // Used for SPI read transfers
  spi_read_seq reg_read_seq;

  // Constructor
  function new(string name = "spi_reg_adaption_layer_seq");
     super.new(name);
  endfunction: new

  // Body Task
  task body();

    `uvm_info(get_type_name(),$sformatf("Inside the spi_reg_adaption_layer_seq\n"), UVM_LOW);

    forever begin

      // Getting the uvm_reg_item from reg_sequence via layered sequencer
      layered_sqr.get_next_item(item);

      `uvm_info(get_type_name(),$sformatf("Got the item from reg sequence\n"), UVM_HIGH);

      if(item.element_kind == UVM_REG ) begin
          $cast(target_reg, item.element);
      end
      else `uvm_fatal(get_type_name(),"Its not an reg access");

      // Get the address and data
      reg_addr = target_reg.get_address(item.map);
      reg_wr_data = item.value[0];
```

Cadence SPI base sequence

Get the item from reg.write() or reg.read() methods for registers accessed via SPI interface

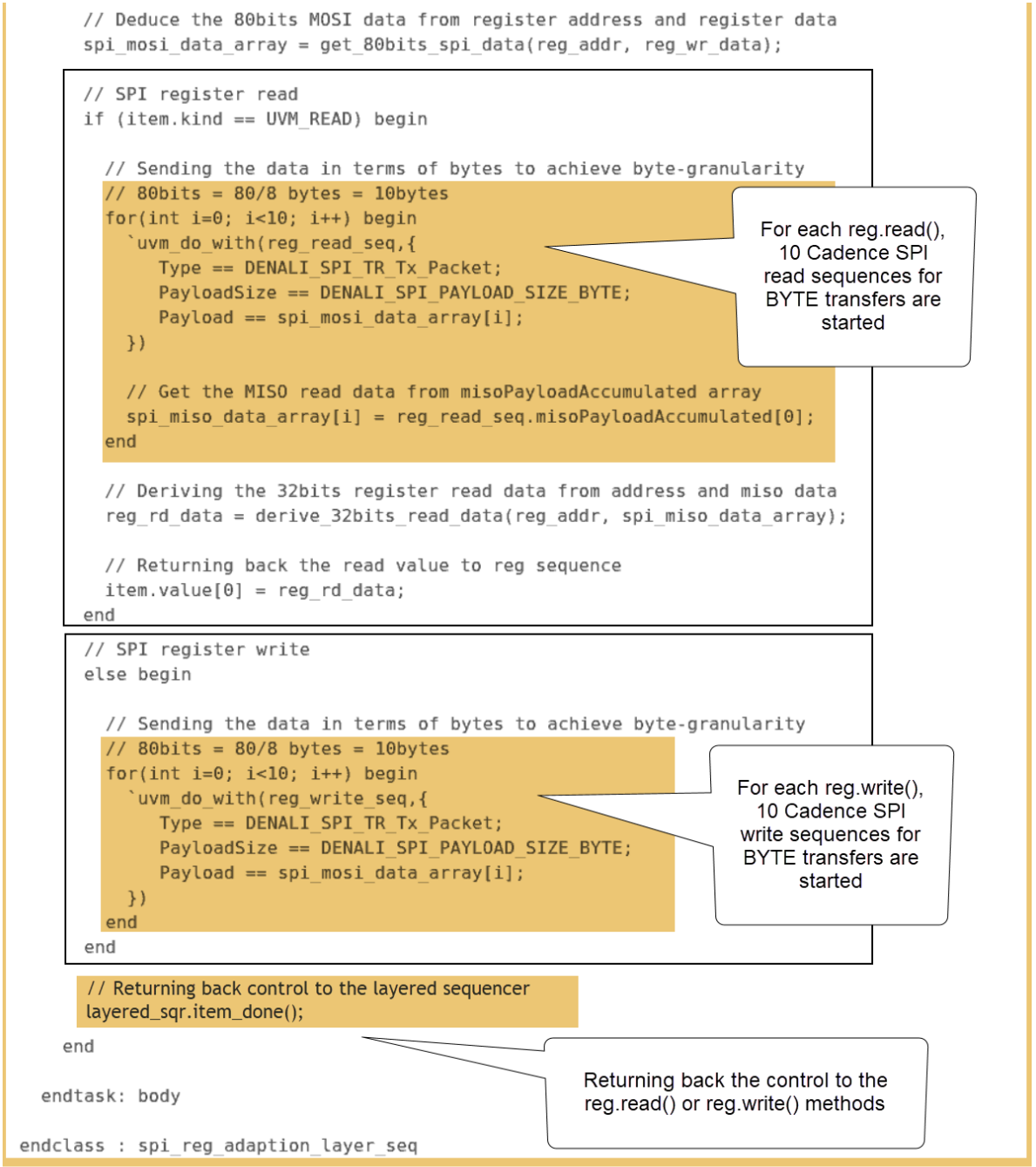

Figure5: SPI Protocol Adapter Layer Sequence Implementation With Cadence SPI Sequences

III. Results

With the implementation of UVM Sequence Layering for RAL, engineers were able to reuse register sequences with a faster verification turnaround time without modifying the testbench. The development of sequence_adapter_layer required only 8% extra effort but resulted in a 70% time-saving while bringing up the testbench and running tests. The overall activity effort to re-write sequences and develop custom verification components was just around 100-man hours, including the use of Cadence VIP for PCIe and SPI interfaces with excellent support from the Cadence VIP team. This demonstrated the effectiveness of UVM Sequence Layering in improving code reusability and scalability.

IV. Conclusion

In conclusion, the paper demonstrates the successful application of Universal Verification Methodology (UVM) Sequence Layering on RAL register sequences, using Cadence VIP for PCIe and SPI protocols. The introduction of the sequence adapter layer resulted in faster verification turnaround times with minimal additional effort. This approach significantly improved code reusability and scalability, making it a valuable technique for complex verification environments.

Acknowledgment

I would like to gratefully acknowledge the critical feedback and support that I received on the content of this paper from my colleagues, Mayukh Majumdar, Vishwanath Anathakrishnan, Priya Anathakrishnan.

References

[1] M. Peryer, D. Aerne, "A New Class Of Registers," - DVCon US 2016
[2] Verification Academy – UVM Cookbook: https://verificationacademy.com/cookbook/registers/integrating
[3] Universal Verification Methodology (UVM) 1.2 Users Guide – Accellera, October 8, 2015
[4] UVM Tutorial for Candy Lovers – 16. Register Access Methods – ClueLogic