Adapting the UVM Register Layer for Burst Access

M. P. Villalpando
General Dynamics Mission Systems
8201 E McDowell Rd, Scottsdale, AZ 85257
mark.villalpando@gd-ms.com

***Abstract*- The Register Abstraction Layer (RAL) built into Universal Verification Methodology (UVM) is an essential tool for properly and efficiently verifying register access. One of the limitations of the UVM RAL is the absence of support for register burst access. While UVM 1.2 does contain methods for bursting UVM memories, a verification engineer is limited to only single access when handling modeled UVM registers. Without these methods, it requires more effort from the verification engineer to properly test and verify a design bus's burst capabilities across individual memory registers. Usually this requires a monitor, scoreboard, and sequences which are separate from those provided in single access UVM RAL.**

This paper aims to extend the functionality of the UVM RAL by describing a method to allow the engineer to frontdoor, burst read or write access using the built in *uvm_reg.write()* and *uvm_reg.read()* methods. This will allow the verification engineer to properly simulate burst access across any series of UVM registers defined in his or her *uvm_reg_map*. With this solution the UVM test capabilities do not suffer, the user still has single access support, the core UVM components are not modified, and custom UVM register model will not require changes. Most importantly, no separate verification components will be required to test burst access across the bus.

I. Introduction

A. Background

UVM RAL uses a variety of components to create a mirrored and predicted database of a register map (or multiple) inside the UVM Testbench. Each register in the design is modeled in the testbench using a class object. This object contains all fields and attributes as it does in the RTL, and is accessible by any UVM component in the environment. This simulation model allows verification engineers to easily assign coverage to registers, check mirrored or desired values, or command the configuration bus with write or read commands. Register Abstraction is truly a power tool in the UVM arsenal, but is also limited in some aspects.

Using RAL, each register in the design is model using a *uvm_reg* base class. This class contains predefined methods such as front door/backdoor reads and writes, coverage handling, and retrieving address or size information. These methods are local to each register defined in a *uvm_register_map* meaning the user has the ability to poll information from any register in simulation. For more information on the RAL hierarchy and *uvm_*reg methods see references [1] [2].

The time consuming *uvm_reg* RAL methods which actually create bus transactions are the front door accesses: *uvm_reg.read()* and *uvm_reg.write()*. Each of these methods is a single access transfer to and from the RTL bus during simulation. This means a test writer can only write to or read from a single register per method call. This is extremely inefficient in simulation because it does not utilize a burst capability of the hardware bus which is a very common feature of FPGA designs. It also creates a hole in the functional coverage, as the burst capability of an address range is never properly verified. In order to properly test, a verification engineer would have to create non-UVM methods to drive burst access and manually update the mirrored model stored in simulation. This can become extremely cumbersome when dealing with thousands of registers.

Luckily, this limitation of the UVM RAL can be overcome without having to modify UVM library code. By using some of the built in RAL capabilities, it is possible to modify the single access reads and writes method calls to transport multiple data words which can be encoded and decoded for burst access. Fig 1. Illustrates the proposed solution, which utilizes an extension object class to transport the burst data.

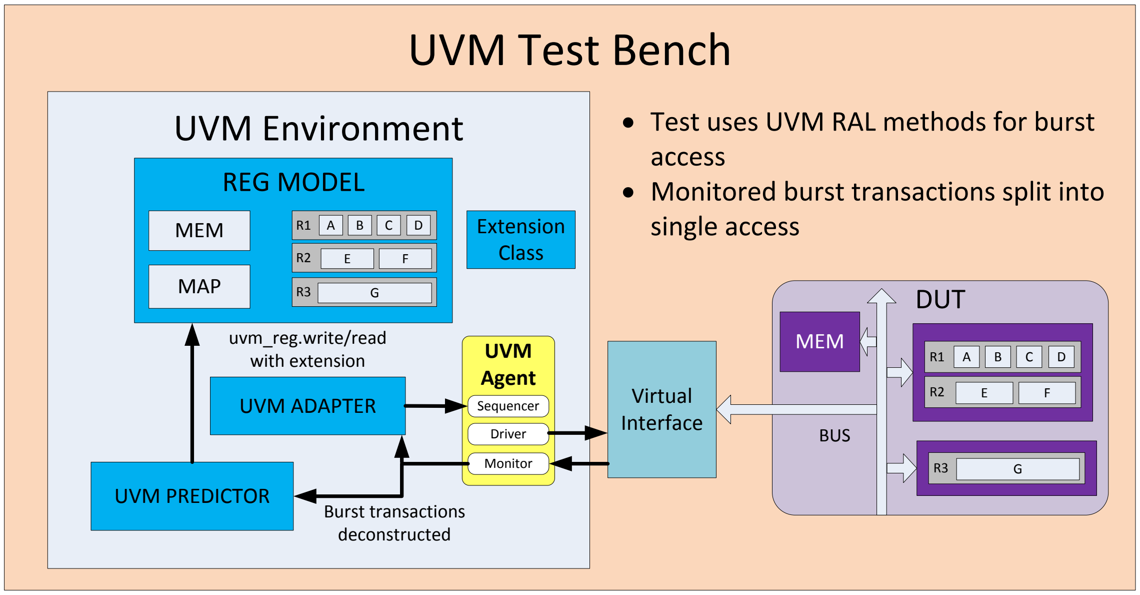

Figure 1. UVM Testbench with RAL – Burst Access Method

B. *Memory Burst Access*

The question may arise: "I can already burst to memories with UVM RAL, why do I need to burst to registers? Can't I just define them as *uvm_mem*?" The answer here really depends on the verification plan its RAL goals. UVM memories are limited in RAL in that they are not actually modeled in the testbench. Rather, a *uvm_mem is* simply a placeholder for a range of addresses. Data which transfers to and from memories are never stored in a mirrored or desired model in simulation and therefore it impossible for a scoreboard to predict or check these values without manually modelling the information on each address. If the verification goal is to match observed values of registers to predicted or desired values, *uvm_mem* objects will not suffice.

C. *Sample Protocol*

The protocol example presented in this paper is a custom shared bus with a valid bit, and a data bus. Fig. 2 and Fig. 3 show the burst read and write operations on the bus. The address is automatically incremented by a controller in the design. In this example, the bus will interface to a 16-bit memory map with 16-bit addresses.

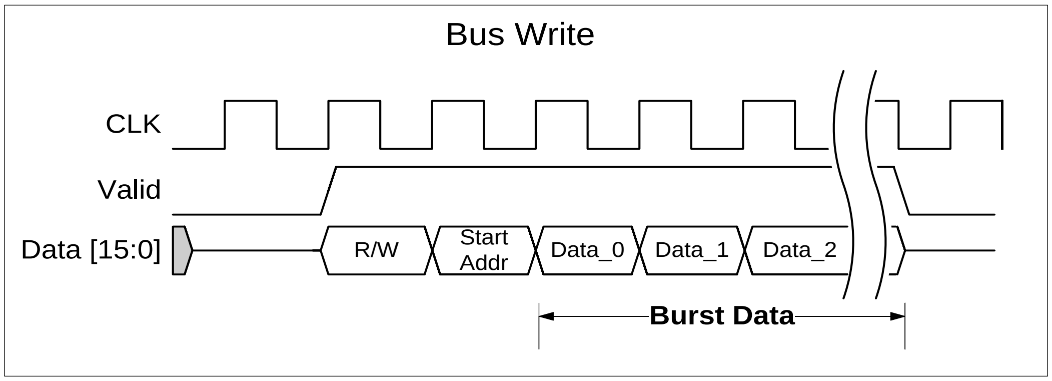

Figure 2. Sample Configuration Protocol – Burst Write

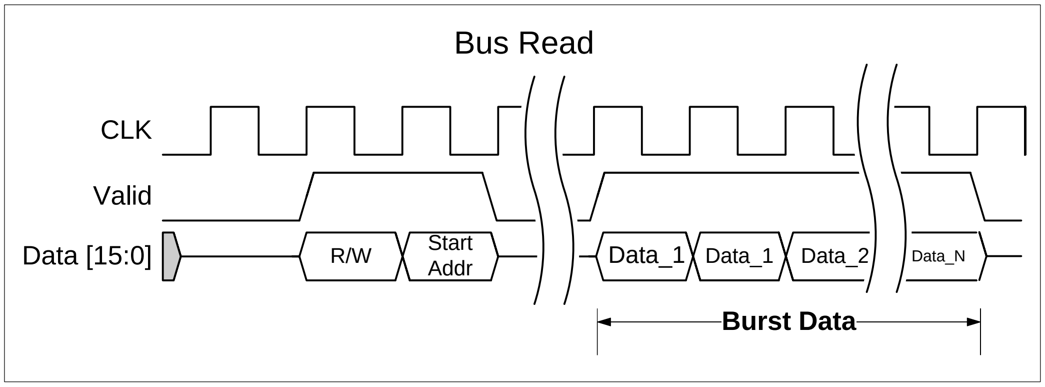

Figure 3. Sample Configuration Protocol – Burst Read

II. METHOD

A. *Extension Object*

Most UVM register commands allow passage of an Extension object to provide extraneous information for the bus transaction. Attributes such as destination ID codes, transmission delays, and other bus controls can all be transmitted through the Extension object and decoded in the Register Adapter.

This extension object is a critical piece of the burst access workaround method. Instead of transmitting our data through the *data* argument in our UVM commands, all the burst data will be stored within the extension object. For the method in Fig. 4, each data word contains two bytes of information. A queue is used to store an unlimited number of words in each transaction and allows for easy, ordered extraction by the Register Adapter. This method allows the test writer to bypass the single data word per register transaction governed by the data argument by simply storing the burst information in an extension object's local queue.

```
class cfg_info extends uvm_object;
  `uvm_object_utils(cfg_info)

  rand int unsigned  transmit_delay;     // Creates transaction to transaction delay
  rand logic  [ 5:0] device_id;      // Destination Device
  rand logic  [15:0] burst_data [$]; //Burst Data to be written
  rand logic  [7:0] burst_length;   //Length of burst write or read

  constraint c_transmit_delay  {
    transmit_delay  > 4;                // Minimum back-to-back latency
    transmit_delay <= 10;
  }
 function new (string name = "");
    super.new(name);
  endfunction
endclass
```

Figure 4. Extension Object

B. *Register Adapter*

The Register Adapter is the key ingredient to a functional RAL design. Its main purpose is to provide two virtual functions *reg2bus()* and *bus2reg()* which will convert bus transactions to register transactions and vice versa. The function shown in Fig. 5, r*eg2bus(),* is triggered by any UVM method which attempts to transmit commands to the RTL bus in simulation. This function simply reconstructs a bus transfer object from a *uvm_reg_bus_op* object which is created from internal UVM elements. To adapt this function for burst access it needs to decode the provided Extension object and store the burst data in a bus transfer object data queue.

```
virtual function uvm_sequence_item reg2bus( const ref uvm_reg_bus_op rw );

  uvm_reg_item item      = get_item();
  cfg_transfer cfg_trans  = cfg_transfer::type_id::create("cfg_trans");
  cfg_info cfg_ext          = cfg_info::type_id::create("cfg_ext"); //Extension object handle

  cfg_trans.dir                              = (rw.kind == UVM_READ) ? READ : WRITE;
  cfg_trans.address[15:0]                = rw.addr[15:0];

  //Copy extension object attributes to the bus transfer object
  if (item.extension != null) begin
    $cast(cfg_ext, item.extension);
    cfg_trans.transmit_delay       = cfg_ext.transmit_delay;
    cfg_trans.device_id              = cfg_ext.device_id;
    cfg_trans.length                    = cfg_ext.burst_length;
  end

  //Populate bus transfer object queue with burst data stored in ext object
  if (rw.kind == UVM_WRITE) begin
     cfg_trans.wr_data                  = cfg_ext.burst_data;
  end
  return cfg;
endfunction: reg2bus
```

Figure 5. Register Adapter – *reg2bus()* function

As shown in Fig. 5, this method does not require use of *rw.data* like in normal single access operation. Instead, the entire burst data queue (*cfg_ext.burst_data*) is passed to the transfer object which the BFM virtual interface will parse and drive on the bus.

The alternate function, *bus2reg()*, is triggered by the UVM Predictor when it receives a bus transaction from the corresponding Monitor. Because the goal is to not modify the existing UVM library code, this function can behave as though it receives a single transaction. The only important aspect is that it pops the top word from the data queues and uses that to build its response. The burst handling will be covered in the Monitor.

C. Monitor

The purpose of the Monitor in this scenario is to provide a compatible transfer object to the Register Predictor which will use the Register Adapter's *bus2reg()* method to convert the bus transaction to a register transaction and update the mirrored register model.

The Register Predictor does not recognize burst access observations, so the Monitor has to convert a burst transaction into multiple single access transactions so they can easily be digested by the Predictor. As show in Fig 6., rather than transmitting the transfer object with all the burst data once, the monitor will cycle through the transfer object queues and publishes a number of transfers equal to the total length of the burst size. This works in conjunction with the Register Adapter, as its *bus2reg()* function simply grabs the data from the top of its queue and returns its *uvm_reg_bus_op* object. So long as the Register Predictor receives a *uvm_reg_bus_op* object for each data word, every register address accessed in the burst will have its corresponding model updated with the observed value.

```
//-------------------------------------------------------------------------
// Monitor's Collect Transactions from the Bus Task
//-------------------------------------------------------------------------
virtual protected task collect_transactions();
  cfg_transfer transfer;
  cfg_transfer trans_predictor; //Modified transaction to be sent through the Anaylsis Port

  forever begin
    transfer = new();
    lcb_config.v_lcb_if.collect_transaction(transfer); //Virtual Interface task gathers Transfer Object
    $cast(trans_predictor, transfer.clone);

     if (transfer.length > 0) begin
        while (transfer.length > 1) begin
          $cast(trans_predictor, transfer.clone);
          ap_cfg_transfer.write(trans_predictor);

          // Update master transaction
          if (transfer.dir == READ)
            transfer.rd_data.pop_front(); //Cycle through observed read data
          else
            transfer.wr_data.pop_front(); //Cycle through observed write data

          transfer.address++;
          transfer.length--;
        end
          $cast(trans_predictor, transfer.clone);
          ap_cfg_transfer.write(trans_predictor);
      end
    end
  end
endtask : collect_transactions
```

Figure 6. Monitor – Collection Task

D. *Virtual Interface*

The Interface in this example is modeled as a bus functional model (BFM) meaning it will simulate all RTL level protocol when driving or collecting data. The only special consideration the virtual interface needs is the ability to decode the transfer object data queue when driving data, and encode the observed data into a queue when collecting data. It is not necessary for the Interface to pack and unpack this transfer objects if it's already handled in the driver and monitor.

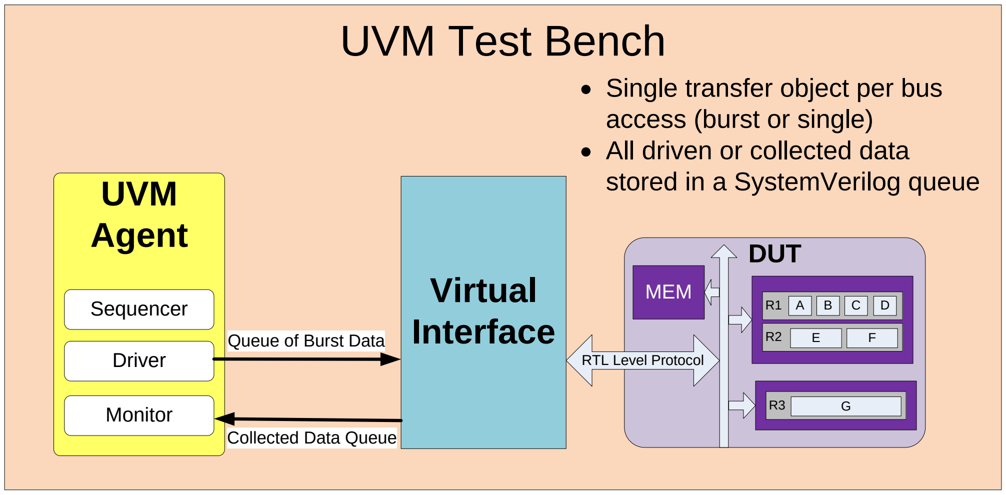

Figure 7. Virtual Interface Operation

E. Test Case

Once all the UVM and RAL elements have been properly configured, the verification engineer can drive and receive burst data through the bus by using the *uvm_reg.write()* and *uvm_reg.read()* methods. Fig 8. shows an extended test class which inherits the UVM component connections, the Register Block, and other configurations from its parent. The test creates an Extension object, populates it with random burst data of length 3 and calls the *uvm_reg.write()*, and *uvm_reg.read()*. The referenced register for each command is the one with the lowest address. This bus increments the target address for each consecutive read/write. For example, if Reg0 is at address 0x02, the commands will read/write address 0x02, 0x03, and 0x04 in that order. This is protocol specific and may require adapting for the user's own configuration bus.

```
class burst_reg_test extends base_test;
  `uvm_component_utils( burst_reg_test )

  cfg_info              m_cfg_ext;   //Extension Object handle
  uvm_status_e          m_status;

  rand logic [15:0] m_burst_data [$];

  constraint c_burst_size {
     m_burst_data.size() == 3;
  }

  function new( string name, uvm_component parent );
    super.new( name, parent );
  endfunction: new

  task main_phase( uvm_phase phase );
    super.run();
    phase.raise_objection( .obj( this ) );

    m_cfg_ext                        = new();
    m_cfg_ext.device_id              = 2;
    m_cfg_ext.transmit_delay         = 2;
    m_cfg_ext.burst_data             = m_burst_data;
    m_cfg_ext.burst_length           = 3;

    m_reg_block.reg0.write(status, 16'h00, .extension(m_cfg_ext));    //Burst Write, starting address is at reg0
    m_reg_block.reg0.read(status, 16'h00, .extension(m_cfg_ext));     //Burst Read, starting address is at reg0

    phase.drop_objection( .obj( this ) );
    global_stop_request();
  endtask: main_phase

endclass: burst_reg_test
```

Figure 8. Test Case Scenario

A. Conclusion

The UVM Register Abstraction Layer provides many of the tools necessary for complete memory map verification. Its built-in methods allow test writers to easily interact with the device under test while abstracting from bus level protocol. The main limitation of these built-in methods is the lack of burst access for modeled registers.

Despite the fact that register burst access is not explicitly supported in the UVM RAL, the method in this paper presents an opportunity for the verification engineer to adapt the included RAL components and methods without having to modify the UVM library code. Some of the modifications are dependent on the protocol of the bus, so no two solutions will be exactly the same. The methodology, however, should be consistent.

A. Advantages

It should always be a goal of the verification engineer to have tests which utilize the maximum speed of the RTL and all its efficient capabilities. Without using custom code, the UVM test does not tap into the burst access feature of register access which the design may have. Because simulation speed is critical for complex designs, it is likely the verification engineer will need to design tests using these burst access capabilities. The method described in this paper allows will inevitability speed up simulation time by using the RTL burst features.

What may not be immediately apparent is that by using built-in UVM methods, the verification engineer doesn't have to spend time developing his or her own methods to accomplish essentially the same task. Tests can become confusing and enigmatic when half of the methods used are UVM single access, which automatically update the mirrored model, and the other half are custom burst access sequences which do not update the model without much more extra code. By using all built-in UVM RAL methods test creation, and sequence creation become less complex and more reusable for future designs.

B. Limitations

Of course, this is not a perfect solution to all RAL problems. One obvious limitation is the lack of response when performing a *uvm_reg.read()*. Since UVM cannot transfer an Extension Object from the bus back to the test, the test will be unable to access the burst read data as it could in single access operation. The mirrored memory maps will still be updated thanks to the separation of transfer objects by the monitor, but the test won't have immediate access to all the read data without using a *uvm_reg.get()* on each register class. Another limitation is the lack of support for *uvm_reg.mirror()*. This is a handy method which will call a *uvm_reg.read()* and automatically compare bus observations with the UVM register model. Unfortunately, the comparison logic is all locked inside the Register Predictor, so without modifying that code it won't be possible to support burst access.

C. Further Modifications

The inclusion of the Extension Object for register methods provides a flexible way to communicate any type of data across the UVM RAL. Using this method it would certainly be possible to add more functionality, such as skipping certain addresses in a burst or switching ID parameters so long as the bus protocol can accept it. Until UVM contains the functionality in its library it is necessary to utilize this Extension object for all extra sequence needs.

REFERENCES

[1] VMM Central. https://www.vmmcentral.org/uvm_vmm_ik/files3/reg/uvm_reg-svh.html.

[2] Verification Academy. https://verificationacademy.com/cookbook/registers

[3] K. Shimizu, “UVM Tutorial for Candy Lovers – 9. Register Abstraction”. 2010. http://cluelogic.com/2012/10/uvm-tutorial-for-candy-lovers-register-abstraction.