Register model backdoor register access automation for a complex IP

Kilaru Vamsikrishna, kilaruv@cadence.com
Sushrut B Veerapur, sushrut@cadence.com
Dhruv Ashvinbhai Donga, donga@cadence.com

Abstract- This paper explores the concept of backdoor register access using UVM. UVM permits backdoor register access, which provides direct access to the DUT's RTL registers via provided HDL path. Backdoor register access has significant advantages over front door access because it needs no simulation time as it accesses the HDL data directly without requiring a bus transaction *(Eg: AXI-Lite READY/VALID handshaking)*. The approach used in this paper does not rely on the path specified in the vendorExtention's HDL path of IP-XACT. IP-XACT vendorExtentions are themselves limited in their ability to capture the backdoor paths of register's subfields and capturing multi-dimensional arrays of registers present on a given design. This paper explains about backdoor API and each API utilizes sv/uvm_reg methods in order to assign hdl_paths to registers and register's fields. These APIs help automate the flow providing the backdoor access mechanism for a complex serial protocol IP. These custom APIs have the capability to automate the process of providing the path for complex IP. This approach also allows easily assigning paths for multiple register banks whereas number banks can be n *(n = 1 to 1000x)*

I. INTRODUCTION

UVM Register model allows access to registers through:

- Front-door access
  - Front-door access tends to require a significant amount of simulation time. This is because a register write or read operation only finishes after completing the protocol handshaking process, which typically spans more than one clock cycle.
- Backdoor register access
  - Backdoor register access happens at zero simulation time. This approach helps in saving CPU time. Backdoor register accesses can be achieved in many ways.

1) Backdoor register access through IP-XACT and Vendor toolset

The use of IP-XACT XML and vendor toolset has facilitated the automated creation of UVM_REG register files. This toolset decodes the IP-XACT XML's and transforms them into UVM_REG register documents. Additionally, it decodes the vendorExtensions tag within IP-XACT XM to extract a register array. This process includes configuring HDL paths for the array of registers defined. For the vendorExtensions that fall outside the IP-XACT standard, the toolset inaccurately captures the dimensions of the register array.
Vendor toolset supports the Hdlaccess through vendorExtensios but it has many limitations.

```
<ipxact:vendorExtensions>
       <cadence:extensions>
      <vendorExtention:array>
      <vendorExtention:a_from>0</vendorExtention:a_from>
      <vendorExtention:a_to>3</vendorExtention:a_to>

<vendorExtention:offset_calc>some_calculation_for_a</vendorExtention:offset_calc>
       <cadence:hdlAccess cadence:path=("reg[%0d]",a)/>
      </cadence:extensions>
</ipxact:vendorExtensions>
```

Limitations with the above approach:

- The IP-XACT XML and the vendor's toolset are unable to accommodate a mix of subfields and arrays due to their limitations.
- Managing or dividing the HdlPath instantiation hierarchy within IP-XACT XML becomes difficult when there aren't enough HdlPaths hierarchy for certain register implementations in the design.
- Some register field's hdlpaths are stored in bytes, while others are stored as separate bits.

Limitations of IP-XACT and vendor tool automation:

- vendorExtensions have limited capabilities when it comes to capturing the hidden paths of subfields in registers present in design.
- Capturing multi-dimensional arrays of registers present on a given design.
- Moreover, this flow is not completely automated.

2) The second method involves utilizing a custom API provided by SystemVerilog/UVM_REG with built-in functions.

II. INTRODUCTION TO UVM RAL

To access and verify large set of registers, register fields, and huge sizes of memories, some innovative approach is much required. UVM provides a core class library called UVM RAL (UVM Register Abstraction Layer) for managing and using registers. UVM RAL is an object-oriented model for registers inside the design. UVM RAL is a simpler method for accessing and verifying design memories and registers.
The RAL model replicates the design register, the unit by which the design register is modified, thereby updating the RAL model register with the same value. To achieve this, UVM utilizes functions to simultaneously update both the design and the RAL model register.
To access and verify large set of registers, register fields, and huge sizes of memories, some innovative approach is much required. UVM provides a core class library called UVM RAL (UVM Register Abstraction Layer) for managing and using registers. UVM RAL is an object-oriented model for registers inside the design. UVM RAL is a simpler method for accessing and verifying design memories and registers.

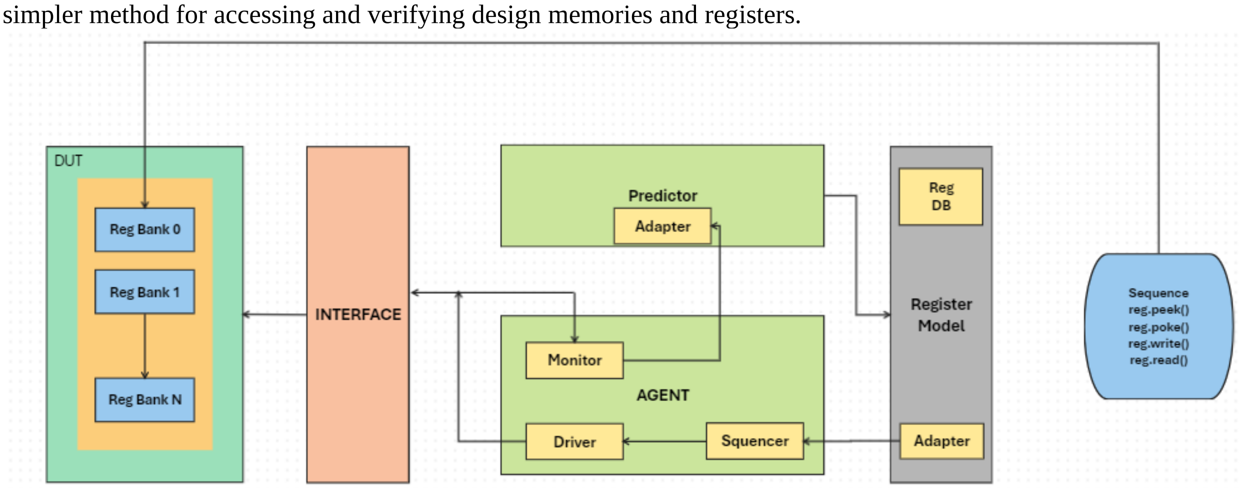

Figure 1: OVERVIEW OF UVM RAL BACKDOOR REGISTER ACCESS

UVM RAL:

The UVM RAL, as its title indicates, serves as a sophisticated object-oriented interface for interacting with design registers. This model replicates the functionality of the design registers, offering complete configurability. Its abstract nature allows for straightforward transition from the block level to the system level.

There are typically three levels of hierarchies in each RAL model:

- The highest hierarchy is uvm_reg_block, which typically contains an instance of each register (uvm_reg), but it may also contain an instance of another uvm_reg_block.
- uvm_reg imitates the design's register. Each register may have one or more uvm_reg_fields depending on its definition.
- *uvm_reg_field* represents the field of the register

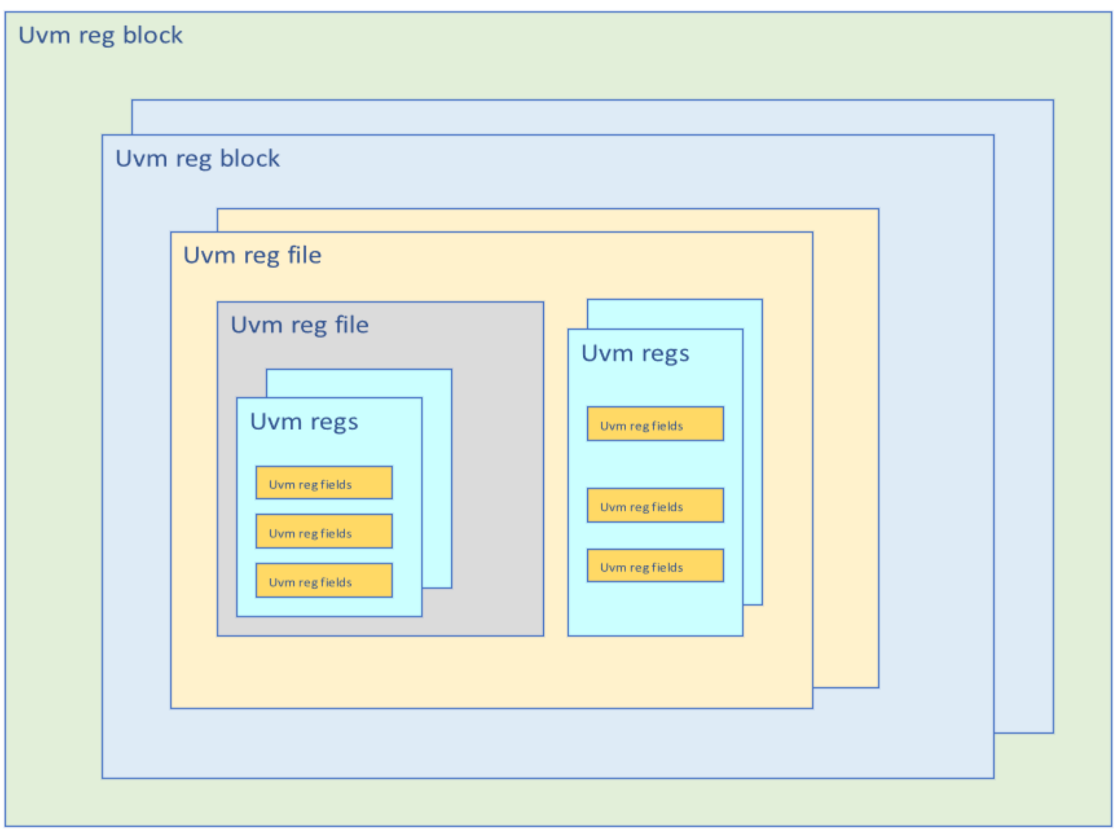

Figure 2: STANDARD UVM REGISTER HIERARCHY

Creating a uvm_reg_block in UVM (Universal Verification Methodology) involves defining a block of registers that may include multiple register files or registers organized hierarchically. Here's a step-by-step guide on how to create a uvm_reg_block, including defining a register block, adding registers and register files, and using them in a testbench.

To construct register model:

1. Define Register and Register File:
   - my_reg: Defines a register with a field.
   - my_reg_file: Defines a file containing the register my_reg.
2. Define Register Block:
   - my_reg_block: Extends uvm_reg_block and includes my_reg_file.
   - build_phase: Instantiates and configures my_reg_file, then adds it to the block.
3. Testbench:
   - Instantiation: Creates an instance of my_reg_block

This example provides a complete framework for defining a register block, including registers and register files, and using them in a UVM testbench:

```
//Define the Register and Register File
class my_reg extends uvm_reg;

 // Register Field
 uvm_reg_field my_field;

 function new(string name = "my_reg", int width = 32, uvm_component parent =
null);
  super.new(name, width, UVM_NO_COVERAGE, parent);
 endfunction

 function void build_phase(uvm_phase phase);
  super.build_phase(phase);
  // Create a 16-bit field
  my_field = uvm_reg_field::create("my_field", 16, UVM_NO_COVERAGE);

  // Add field to the register
  this.add_field(my_field);

  // Configure the field
  my_field.configure(this, 16, 0, UVM_REG_DEFAULT);
 endfunction

endclass
```

```
//Register File Definition
class my_reg_file extends uvm_reg_file;

 // Register instance
 my_reg reg1;

 function new(string name = "my_reg_file", uvm_component parent = null);
  super.new(name, parent);
 endfunction

 function void build_phase(uvm_phase phase);
  super.build_phase(phase);

  // Create and configure the register
  reg1 = my_reg::new("reg1", 32, this);
  reg1.configure(this, 32, 0, UVM_REG_ALL_ON);

  // Add register to the register file
  this.add_reg(reg1);
 endfunction

endclass
```

```
// Define the Register Block
class my_reg_block extends uvm_reg_block;

 // Register files
 my_reg_file reg_file1;

 // Constructor
 function new(string name = "my_reg_block", uvm_component parent = null);
  super.new(name, parent);
 endfunction

 // Build phase to create and configure the register file
 function void build_phase(uvm_phase phase);
  super.build_phase(phase);

  // Create the register file
  reg_file1 = my_reg_file::new("reg_file1", this);

  // Add the register file to the register block
  this.add_reg_file(reg_file1);

  // Configure the register block
  this.configure(this, 0, 0, UVM_REG_BLOCK_ALL_ON);
 endfunction

endclass
```

```
//Instantiate and Use the Register Block in the Testbench
module tb;

 // Instantiate the register block
 my_reg_block reg_block;

 initial begin
  // Create and configure the register block
  reg_block = my_reg_block::new("my_reg_block", null);

  // Access and manipulate registers through the block
  reg_block.reg_file1.reg1.my_field.write(16'hABCD);

  // Read from the field of the register
  uvm_reg_data_t data;
  reg_block.reg_file1.reg1.my_field.read(data);
  $display("Field value: %h", data);
 end

endmodule
```

UVM RAL Adapter:

Within the framework of UVM (Universal Verification Methodology) and the RAL (Register Abstraction Layer) framework, an "adapter" usually denotes a class or part that acts as a connector between the RAL model and the real-world hardware or simulation interfaces. This role is essential for converting complex register operations into simpler transactions and back again.

Role of an Adapter in UVM RAL:

1) Conversion of Registration Processes:
The RAL framework offers a broad overview of registers, fields, and processes. Nonetheless, the direct interaction with the hardware typically requires the use of more detailed protocols (such as AXI, AHB, APB etc.).

Adapter converts the abstract register operations (like read(), write(), or modify()) into protocol-specific transactions that can be understood by the hardware interface drivers.

2) Interface Bridging:

Various parts within a testbench could employ distinct interfaces or protocols. The RAL model must engage with these parts smoothly.

The adapter serves as a connection between the RAL model and these interfaces, making sure that register accesses are converted into the appropriate format for the specific interface being used.

3) Register to Protocol Mapping:

In the RAL model, the details of registers are simplified from the real-world hardware specifics. For these simplifications to be used in real transactions, they must be linked back to the hardware's protocol.

Adapter maps the register fields and values to the appropriate protocol transactions, handling tasks such as address translation, data formatting, and protocol-specific signaling.

Example in which you possess a UVM RAL model for a collection of control registers, and you're tasked with linking this to an AXI interface:

```
class my_ral_adapter extends uvm_object;
  // Interface or driver for low-level transactions
  virtual axi_if axi_interface;

  // Constructor
  function new(string name = "my_ral_adapter", uvm_component parent = null);
    super.new(name, parent);
  endfunction

  // Method to perform the conversion
  function void perform_write(reg_addr, reg_data);
    axi_transaction tx;
    tx.addr = reg_addr;
    tx.data = reg_data;
    axi_interface.do_write(tx);
  endfunction

  // Additional methods for other operations (e.g., read, modify)
endclass
```

In this example:

my_ral_adapter interfaces with an AXI interface and executes write tasks by transforming register actions into AXI transactions.

perform_write function converts the address of a register and its data into the format required for AXI transactions, and then transmits it via the AXI interface.

UVM Predictor:

The Register Predictor is a part of the Register Abstraction Layer (RAL) that plays a role in the DUT (Design Under Test) by modeling and forecasting the actions of registers. The RAL structure in UVM enables the creation, alteration, and confirmation of registers in an organized manner, and the Register Predictor improves this process by monitoring anticipated register values.

Role of UVM REG predictor in RAL:

1) The Register Predictor is set up as a component of the RAL configuration and is connected to the UVM register model. It begins with default values, usually mirroring reset values or the state specified in the testbench.

2) Whenever a writing operation is executed, the Register Predictor modifies its internal configuration to mirror the updated values. This guarantees that the predicted values align with the executed operations.
3) For read transactions, the Register Predictor can provide the predicted value based on its current state. This predicted value can then be compared with the value read from the DUT.
4) When there's a difference between what was expected and what happened, it suggests there might be a problem with how the DUT's register is managed or with the register operations. The Register Predictor assists in spotting these problems by pointing out differences.
5) The Register Predictor integrates seamlessly with the UVM register model and is used in conjunction with UVM register sequences, drivers, and monitors. It leverages UVM's extensive register modeling capabilities to predict and track register values effectively.

Here’s a basic piece of UVM code to define a RAL predictor for a register field:

```
class my_reg_predictor extends uvm_reg_predictor;

  // Constructor
  function new(string name = "my_reg_predictor", uvm_component parent = null);
    super.new(name, parent);
  endfunction

  // Override the predict method to provide register prediction
  function void predict();
    uvm_reg_data_t reg_val;

    // Example: Assuming we want to predict the value of register 'my_reg'
    // Replace my_reg with the actual register instance
    my_reg.get(my_agent.reg_map, reg_val);

    // Print the predicted value
    `uvm_info("PREDICTOR", $sformatf("Predicted value of register: %0h", reg_val), UVM_LOW)
  endfunction

endclass
```

The introduction to UVM RAL (Register Abstraction Layer) emphasizes its essential function in the verification of hardware using the Universal Verification Methodology approach. UVM RAL offers a systematic approach for representing and engaging with registers by simplifying the challenges linked to accessing and modifying registers. This simplification improves the creation of testbenches by providing a uniform, elevated interface to registers, which encourages reusability and ease of maintenance. Using UVM RAL, verification engineers can make their procedures more efficient, concentrate on ensuring the functionality of the system instead of dealing with the intricacies of register access, and guarantee more thorough and consistent testing of hardware designs.

III. IMPLEMENTATION

The initial phase of this process involves creating a UVM register model and the use of IP-XACT enables the generation of register models. After generating reg model, using std register model APIs get each and every register and it’s fields using get_all_fields method using uvm_reg_block. In UVM_REG_BLOCK, each register is linked to the appropriate hdl_path in the RTL design. These all fields is required because the initial step is to provide backdoor path for all fields which we are getting in backdoor_api file at zero simulation time. After getting all fields from a given address map, for all these fields set the backdoor path. For each regiser store the below info required for back door path access:

- regfile_name - (Through -- field.get_parent().get_regfile().get_name())

- reg_name - (Through -- field.get_parent().get_name())
- field_name - (Through -- field.get_name())
- offset - (Through -- field.get_lsb_pos())
- size – (Through -- field.get_n_bit())

These all will be used to set automative backdoor path. Afterwards, apply this information in the add_hdl_path built-in function. However, before doing so, delete the previous hdl_path by using the clear_hdl_path function. There are two approaches available for implementing a backdoor path

- Involves accessing all registers (add_hdl_path())
- Involves accessing all fields within a register (add_hdl_path_slice())

In complex IPs each and every fields contains useful information. So it's better to use second methods using fields (add_hdl_path_slices)
Here is the example of How to make APIs, How to set backdoor path and also How to utilize built-in APIs in detail
The add_hdl_path_slice function accepts inputs such as the HDL path, size of bits, position of bits, and is used to set individual bits or groups of bits within a register to a specific HDL path within the design.

- The first argument of add_hdl_path_slice is string name which is nothing but hdl_path field slice name. In this path from top to reg bank the path will be given by designers. In this path add regfile_name, reg_name and field name which are stored from reg model.
- If reg file is containing more than one reg file then in that case the parent regfile will be field_name.get_parent().get_regfile().get_regfile().get_name(); and concatenate both reg file in one string and then assign it to main string regfile_name which will be used in PATH to set HDL path. Same can be done for N no of reg files.
- Modify hdl path like: string PATH={TOP_PATH.regfile_name_reg_name_field_name}.
- The second argument of add_hdl_path_slice is referred as offset. Offset simply determines the starting bit position of the field. This offset value can be get through function get_lsb_pos(), Ex: int offset_val=field.get_lsb_pos();
- The third argument of add_hdl_path_slice is referred as size. Value of size can be easily determined through buiit-in function get_n_bit(), Ex : int no_of_bit=field.get_n_bit();
- All the above are appended to the function like below:

  foreach(fields[i]) parent_reg.add_hdl_path_slice(PATH,offset_val,no_of_bit,"RTL"); This will add hdl_path for all the fields which are present in IP-XACT/REG_BLOCK.

Refer below example for creation of API.

```
function automatic void backdoor_path(uvm_reg_block block);
  uvm_reg_field lookup_flds[string];
  uvm_reg_field fld_q[$];
  uvm_reg lookup_regs[string];
  ...
  block.get_fields(fld_q);
  foreach(fld_q[idx])
   lookup_flds[fld_q[idx].get_full_name]=fld_q[idx];
  ...
  foreach(fld_q[idx]) begin
    string hdl_path,field_name,reg_name,regfile_name,regfile_name_parent_1,..,regfile_name_parent_N;
    string lsb,size_of_field,top_rtl_path_hierarchy;
    ...
    lsb=fld_q[idx].get_lsb_pos();
    size=fld_q[idx].get_n_bits();
    reg_name=fld_q[idx].get_parent().get_name();
    field_name=fld_q[idx].get_name();
    regfile_name=fld_q[idx].get_parent().get_regfile().get_name();
    ..
    regfile_name_parent_N=fld_q[idx].get_parent()....get_regfile().get_name();
    ...
    if(lookup_flds.exists(fname)) begin
        uvm_reg parent_reg = lookup_flds[fname].get_parent();
        ...
        hdl_path={top_rtl_path_hierarchy,regfile_name,"_",reg_name,"_",field_name};
        parent_reg.add_hdl_path_slice(hdl_path,lsb,size);
    end
  end
endfunction
```

After creating API, This API needs to be added in every reg block through some scripting. Whenever new reg bank will be added, just include this API(need to change TOP RTL path based on newly added bank's hierarchy and guard it with newly added bank's switch so it won't affect older banks) and it will set backdoor path for newly added register bank so like this automation will be done.

```
class my_reg_block extends cdns_uvm_reg_block;
  `uvm_object_utils(my_reg_block)
  uvm_reg_map default_map;
  uvm_reg_map my_reg_map;
  rand my_reg_bank bank1;
  ...
  virtual function void build();
   my_reg_map = create_map("my_reg_map",
   `UVM_REG_ADDR_WIDTH'h0, 4, UVM_LITTLE_ENDIAN, 1);
   default_map = my_reg_map;
   bank1 = my_reg_bank::type_id::create("bank1", , get_full_name());
   bank1.configure(this);
   bank1.build();
   ...
   //Mapping my_reg_map

   my_reg_map.default_map.add_parent_map(bank1,`UVM_REG_ADDR_WIDTH'h0);
   my_reg_map.set_submap_offset(bank1.default_map,
  `UVM_REG_ADDR_WIDTH'h0);
   ...
   //Apply hdl_paths through API
   backdoor_path(this);
endclass
```

Once the HDL paths are set at 0 simulation time, one can use reg model backdoor register access. After, need to set path of uvm_path_e as UVM_BACKDOOR to enable backdoor register access and will have to make different task for backdoor write/read.

- For single register write/read, use built-in methods reg.write()/ reg.read() and inside that must pass third argument as UVM_BACKDOOR or can use built-in backdoor method reg.peek()/reg.poke()

Refer below code:

```
task reg_backdoor_write_read(string reg_name,
                             input bit[31:0] write_data,
                             output bit[31:0] read_data);
  uvm_reg my_reg;
  uvm_status_e status;
  uvm_path_e path= UVM_BACKDOOR;
  ...
  my_reg.write(status, write_data, path);
  #1;
  my_reg.read(status, read_data, path);
  ...
  if(status != UVM_IS_OK)
    reg_access_success=0;
  else
    reg_access_success=1;

 endtask
```

This is an automated way to add backdoor path for every register field in the register block.

IV. PERFORMANCE MEASUREMENTS

Conducted a study to compare the time it takes to access the front door versus the backdoor, by varying the number of times the UVM_REG register was accessed. This included testing both write and read operations on the register in both the front door and backdoor settings, with the number of register access attempts were 300, 2000, 10,000, 100000. The objective was to measure the CPU time required for these tests when the register access was performed in both the front door and backdoor configurations.

Number of Register access	CPU Time with Front Door register access (Mins)	CPU Time with Back Door register access (Mins)	Acceleration
Around 300+	9.20	6.30	**1.46x**
Around 2000+	14.47	6.41	**2.25x**
Around 10,000+	46.34	9.07	**5.10x**
Around 1,00,000+	367.84	16.72	**22x**

V. APPLICATIONS

In the future, whenever a new reg block is added, simply invoke the backdoor function that set up in a separate file. Using this method, easily set path for N no. of register banks (reg_bank_0 -> reg_bank_n. To achieve functional coverage for the registers, utilize backdoor register access, as it requires minimal CPU time and can improve performance.

VI. CONCLUSION

This paper demonstrates an automated process instead of manually adding backdoor register access for every reg block, eliminating the need to repeat the same steps for every reg block. The primary purpose of utilizing backdoor register access is to decrease the amount of CPU time required.