One Stop Solution for DFT Register Modelling in UVM

Rui Huang
Advanced Micro Devices, Inc., Rui.Huang@amd.com
No.2, Science Institute South Rd.
Haidian District, Beijing, China 100190

***Abstract*-The DFT (Design For Test) design is becoming more and more complex to satisfy test requirements for ultra-large-scale SoC (System on Chip). From the perspective of test access method, nowadays IEEE 1149.1 protocol is usually adopted along with IEEE 1687 and 1500 protocols, which enables easy and modular integration of DFT IP (Intellectual Property) into SoC. However, this approach makes the DFT test access network complex and it needs a series of complex shift operations to access a TDR (Test Data Register). It will be beneficial if abstracting DFT TDR access in RAL (Register Abstract Level), so that test writers can focus on test sequences and tests can easily migrate from block level to system level. This paper introduces a layered structure to model DFT TDRs and its access network, which is universal for different projects. Also this paper introduces a new way to model ultra-long length registers in UVM that requires smaller memory space in simulation than the current UVM RAL solution.**

I. INTRODUCTION

To model DFT TDR in UVM, the first challenge is how to model ultra-long length TDRs. In particular, some DFT TDRs' length, compared to the functional registers of a SoC, can be as long as thousands of bits. At some situations such as MBIST (Memory Built-In Self-Test) dumping or scan dumping, the TDRs' length can be even longer. The UVM RAL mainly targets at functional registers and subjects to its limitation when modelling the ultra-long length DFT TDR.

If we want to model DFT TDRs of a system using the current UVM RAL solution, we need to override the macro *UVM_REG_DATA_WIDTH* to the value identical to the length of the longest DFT TDR in the system. The width of the *uvm_reg_data_t* data type in UVM RAL is decided by the above-mentioned *UVM_REG_DATA _WIDTH* macro. The *uvm_reg_data_t* is constructed and used almost everywhere in the RAL-related components and objects. Furthermore, when constructing a TDR, every field of that TDR is modelled as *uvm_reg_field* that also profligately uses the *uvm_reg_data_t* type to store the field value, although every field of the DFT TDR is not long. As we can see the waste of storage in simulation is obvious.

This paper presents a new way to model ultra-long length DFT TDR using UVM, with a minor amount of work needed to override the methods of the base classes of UVM RAL.

In a complex DFT test access network, different protocol TDRs are hierarchically located in a network that is connected via IEEE 1687. To access a TDR, one or more levels 1687 SIBs (Segment Insertion Bit) have to be opened, and the length of DR (Data Register) chain varies with SIB values, which results in the second challenge: how to get the necessary information to convert the abstracted generic TDR access operation into a series of IEEE 1149.1 shift operations (herein after called JTAG operations) – and ideally this solution can be applied to different projects. Recently the author [1] introduced a novel method to model DFT TDR access network, where functional equivalent elements are created as the DUT (Design Under Test) and information required in converting an abstracted TDR access to JTAG operations is encoded into the TDR's address.

The disadvantage of [1] is that, when a TDR is accessed, its controlling SIBs are opened, desired value is shifted to the TDR, and then the SIBs are closed to their default values for the convenience of the next TDR access. However, in real test scenarios, it turns out that TDRs controlled by identical SIBs are accessed probably in tandem, which means that we can save some unnecessary SIBs opening and closing JTAG operations, and thus some test time in ATE test.

This paper tries to improve the efficiency of converting a generic TDR access operation to JTAG operations introduced by [1], it saves shift cycles by means of monitoring the status of current network SIBs and analyzing the TDR to be accessed so as to open and close the SIBs smartly.

A. *Structure of This Paper*

This paper introduces a layered structure to model DFT TDR. As shown in Figure 1, the DFT TDR layer is divided into two layers. In the Register Layer One, the generic DFT TDR access is converted into the generic *dft_reg_transaction*, and the *dft_reg_monitor* writes the observed *dft_reg_transaction* to the *dft_reg_predictor*. In the Register Layer Two, the generic *dft_reg_transaction* is converted into a series of *jtag_transaction*s, and the *dft_tdr_network* returns the *dft_reg_transaction* by observing *jtag_transactions* written by *the jtag_monitor*. In the Transaction Layer, the *jtag_transaction*s are passed to the *jtag_driver* to toggle JTAG interface.

Accordingly, this paper is divided into three sections. The first section is about how to model ultra-long length TDR in the Register Layer One, while the second section explains how to model DFT TDR access network in the Register Layer Two. The third part is results and discussion.

In each of the first and second parts, a general overview will be provided firstly, and then the detailed implementation will be elaborated with reference to examples.

II. Ultra-Long Length TDR Modelling in Register Layer One

B. *Idea Overview*

Regarding to UVM register modelling on ultra-long length TDRs, the author analyzed the methods being called during a register write and read process in UVM RAL, and found that the major limitations lies in the following two facts:

1. The register access methods of the *uvm_reg* class uses *uvm_reg_data_t* as the routine argument or return type for generic register access operations such as *uvm_reg::write()* and *uvm_reg::read()*.
2. Most methods in UVM RAL suppose that the dynamic array size of *rw.value* is one.

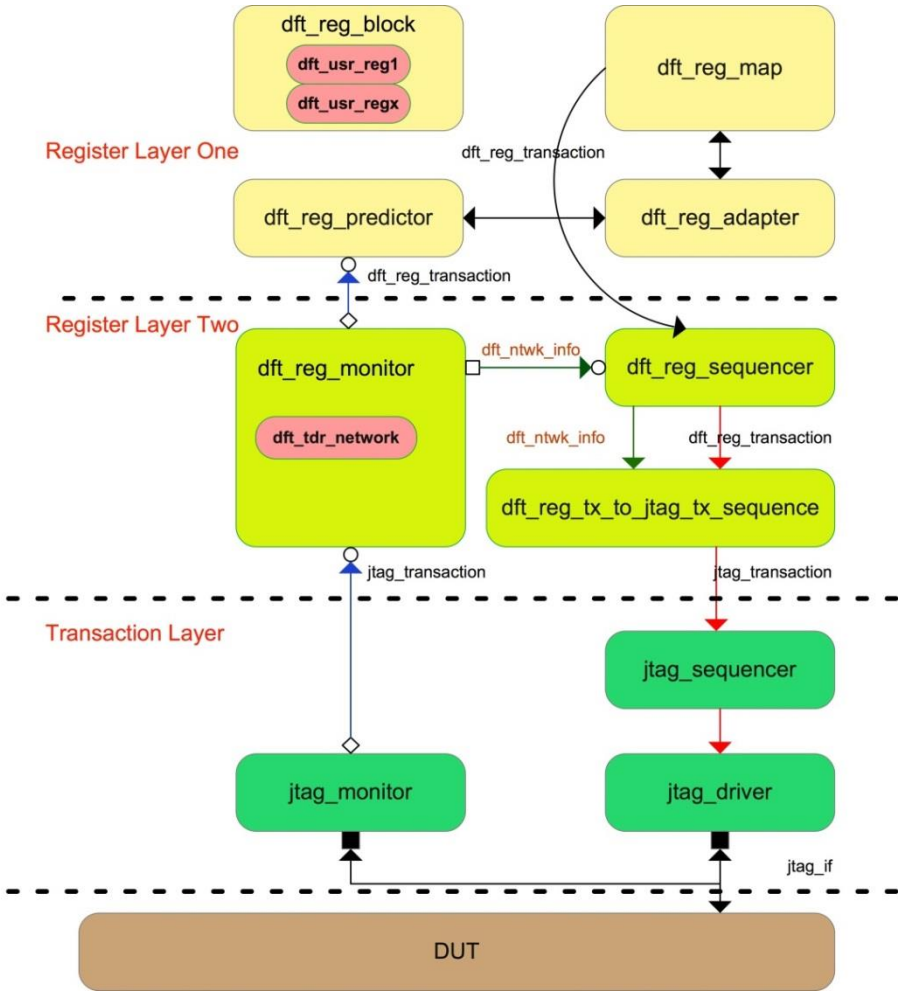


Figure 1. DFT TDR modelling block diagram.

To eliminate these limitations, the following steps can be taken:

1. As shown in Figure 2, the *dft_reg* class is defined as the base class when construing DFT TDRs. A set of register access methods with "dft" as prefix are added to replace the corresponding ones in *uvm_reg* class. This set of methods uses *dft_reg_data_t*, which is a queue of bit type, as routine argument and return type instead of *uvm_reg_data_t*. In this way, no matter how long the TDR length is, the queue size can dynamically fit to the TDR's length.
2. Modify the methods supposing the dynamic array size of *rw.value* to be one, so that they construct *rw.value* array according to the bit length of the data being processed, or they fetch the data in *rw.value* array by checking its size first.

Figure 3 illustrates how a generic DFT TDR write access is converted to a generic *dft_reg_transaction* in the view of routine arguments passing.

1. The value to be written in a TDR is passed to the *dft_reg::dft_write()* by the *value_q* argument of *dft_reg_data_t* type, instead of by the *uvm_reg::write() value* argument of *uvm_reg_data_t type*.
2. The *dft_reg::dft_set()* converts *value_q* data into several segments of *uvm_reg_data_t* type, which are filled to *uvm_reg_field::set()*.
3. The *dft_reg::do_write()* creates a *uvm_reg_item* object to store the *value_q* data by re-constructing the dynamic array (*rw.value*) with desired size.
4. The *dft_reg::do_write()* passes the written data to *dft_reg_map::do_write()* by argument *rw* of *uvm_reg_item* type.
5. The *dft_reg_map::do_bus_write()* converts the written data stored in the dynamic array to several *uvm_reg_bus_op* packages, which are passed to the *dft_reg_adapter*.
   By using the MSB (Most Significant Bit) of the address (encoded in Figure 4) as a flag bit that indicates the last package of the written data, the *dft_reg_adapter::reg2bus()* knows the time when all the written data are collected and to return the complete *dft_reg_transaction* that the *dft_reg_map::do_bus_write()* is going to send to the *dft_reg_sequencer*.
6. The *dft_reg_predict::write()* calls *dft_reg_adapter::bus2reg()* for several times until the MSB of the *uvm_reg_bus_op.addr* is set by the *dft_reg_adapter::bus2reg()* to indicate that the last package data has been converted.
7. The *dft_reg_predict::write()* creates a *uvm_reg_item* object and stores the *data* member of *uvm_reg_bus_op* structure to *rw.value* that is passed to *dft_reg::do_predict()*.
8. The *dft_reg::do_predict()* concatenates all the data in *rw.value* to *dft_reg_data_t* type, disassembles it into several *uvm_reg_data_t* type according to the TDR's field width, and then passes them to *uvm_reg_field::do_predict()*.

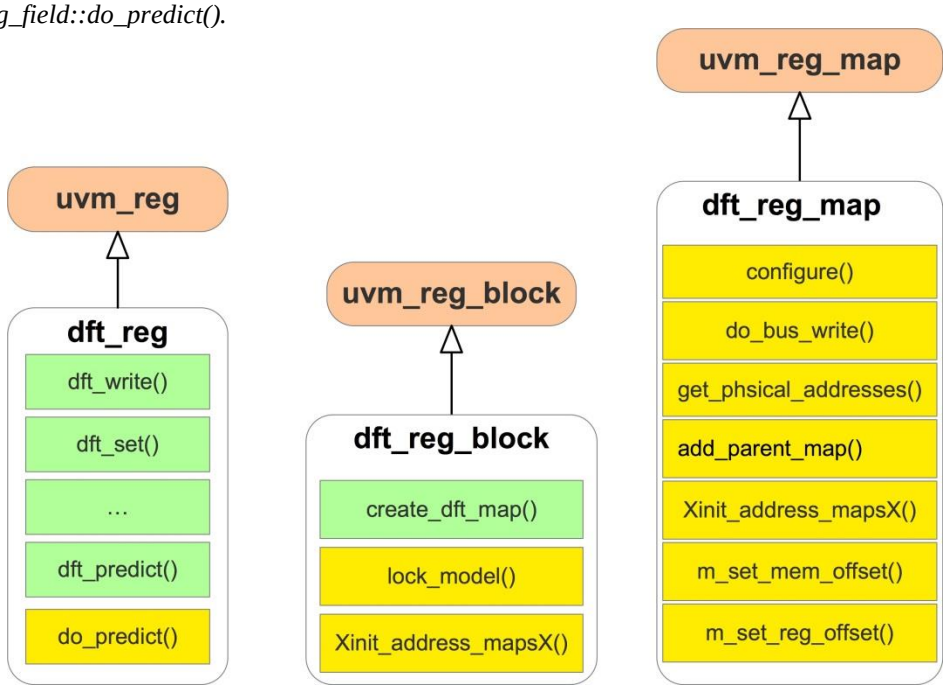


Figure 2. Class extension diagram.

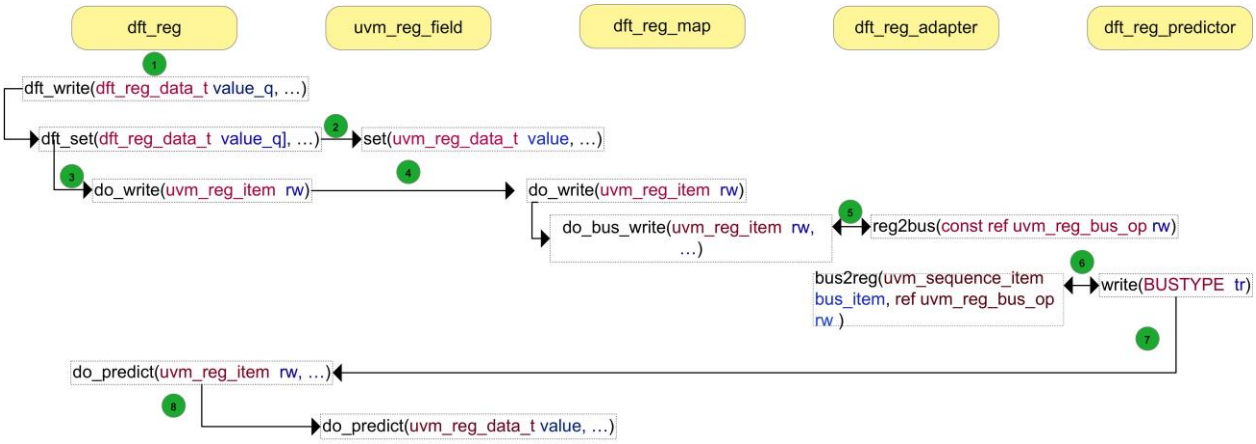


Figure 3. Routine argument passing process of write operation.

C. dft_reg Class Implementation

As shown in Figure 2, the methods with "dft_" prefix are newly added (shown in green) for DFT TDR access. To implement these newly added methods, we can copy the corresponding ones in the *uvm_reg* class and modify the input argument (or return type) to *dft_reg_data_t* instead of *uvm_reg_data_t*. Then the data type conversion is added when it is needed. Because the *uvm_reg::do_predict()* supposes the *rw.value* array size to be one, we need to use *dft_reg::do_predict()* to override it.

D. dft_reg_block Class Implementation

The *dft_reg_block* class is extended from the *uvm_reg_block* shown in Figure 2. In Figure 2, the green methods are newly added, and the yellow methods need to be overridden. Because many methods use local variables in *uvm_reg_block* class which cannot be seen by extended classes, we copy all codes in *uvm_reg_block* to *dft_reg _block* class and do the following changes:

1. Remove the fatal error check that *uvm_reg_block::max_size* should not be larger than *UVM_REG_DATA_WIDTH* in *lock_model()* function. This check is invalid for DFT TDR, because DFT TDR is configured through serial JTAG bus.
2. Enhance *Xinit_address_mapsX()* function to support *dft_reg_map* type.
3. Add *create_dft_map()* function to return *dft_reg_map* type register map.

E. dft_reg_map Class Implementation

The *dft_reg_map* class is extended from the *uvm_reg_map* shown in Figure 2. Similar to the *uvm_reg_block*, many methods of the *uvm_reg_map* use local variables, so we copy all codes in *uvm_reg_map* to *dft_reg_map* class and do the following changes:

1. Modify the *do_bus_write()* method to convert each element in *rw.value* to *data* member of *uvm_reg_bus_op* structure and set the MSB of the last *addr* member *of uvm_reg_bus_op* structure to one so as to indicate that all the written data have been transferred as shown in black arrow in Figure 5. Then *dft_reg_adpater::reg2bus()* returns a complete generic dft_reg_transaction to *do_bus_write()*.
2. Modify *get_physical_adresses()* to only returning signal address no matter how long the TDR width is.
3. Modify *top_map* to *dft_reg_map* type instead of *uvm_reg_map* type in *Xinit_address_mapX()*, *m_set_reg _offset()* and *m_set_mem_offset()*.
4. Modify *add_parent_map()* to supporting *dft_reg_map* type.
5. Modify local variable *m_parent* to *dft_reg_block* type, and modify *configure()* function accordingly.

F. dft_reg_predictor Class Implementation

The *dft_reg_predictor* class is extended from *uvm_reg_predictor*. Add *dft_map* variable of *dft_reg_map* type and modify *write()* task to let it call *dft_reg_adpater::bus2reg()* several times until it see the MSB of the *addr* member *of uvm_reg_bus_op* structure is set, which indicates that the observed *dft_reg_transaction* has been converted to several *uvm_reg _bus_op* packages, as show in red arrow in Figure 5. Then the *write()* task create a *uvm_reg_item* object to store the returned packages and pass the object to *dft_reg::do_predictor()*.

III. DFT Test Access Network Modelling

G. Idea Overview

Figure 6 is an example for DFT test access network. In Figure 7, a SIB is modelled as *sib_node*, and a D flip-flop is modelled as *reg_node*. The *out_update ()* method models the active clock edge that triggers the shift register bit during shift operation, while the *value_update ()* method models the active clock edge that triggers the update register bit during the update operation. By using the *sib_node* and *reg_node* we can constructs the elements of the test access network. The possible paths from TDI (Test Data Input) to TDO (Test Data Output) can be described by System Verilog conditional statements. As such, a functional equivalent dft_tdr_network is obtained.

H. dft_reg_transaction and jtag_transaction definition

Figure 8 shows the properties of the *dft_reg_transaction* and the *jtag_transaction* class.

In the *dft_reg_transaction*, the *read_not_write* indicates whether the transaction is a read or a write operation.

The *addr* is the TDR's encoded address, as shown in Figure 4.

The *wr_data_q* stores the data to be written.

The *rd_data_q* stores data returning by the *dft_reg_monitor*.

The *reg_length* indicates the TDR's length.

The extension is used to send side information to the *dft_reg_adapter*.

In the *jtag_transaction*, the *o_ir* stores the TDR's OPCODE (OPeration CODE) and *o_ir_length* is its size. The *o_dr* stores the data being written to the TDR and the *o_dr_length* is its size.

The *tdo_dr_queue*, *tdo_ir_queue*, *tdi_dr_queue*, and *tdi_ir_queue* store the data during shifting IR (Instruction Register) or shifting DR (Data Register) state monitored by the *jtag_monitor*.

Figure 4. TDR address encoding.

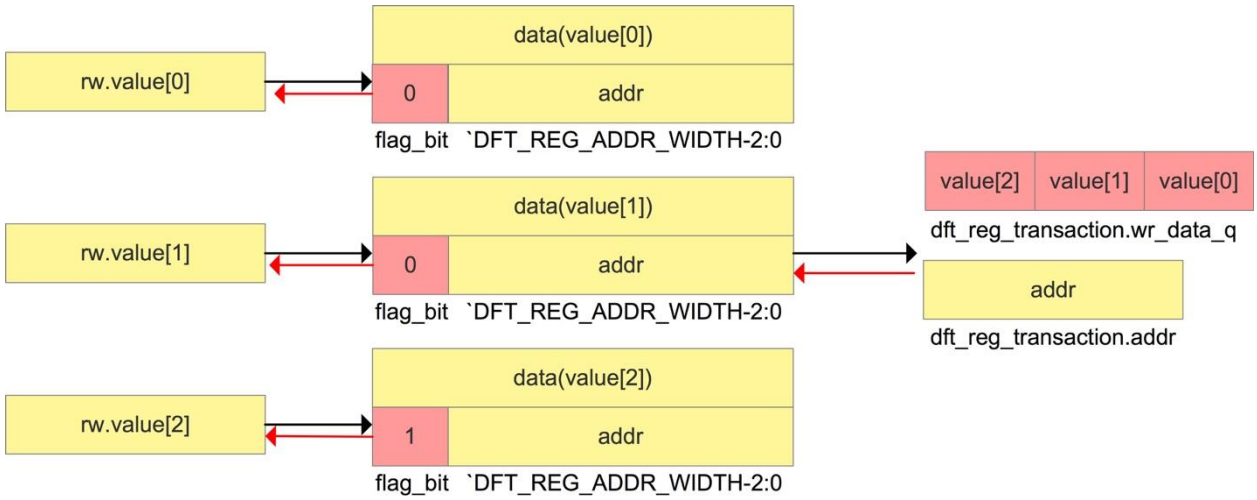


Figure 5. Data conversion process example.

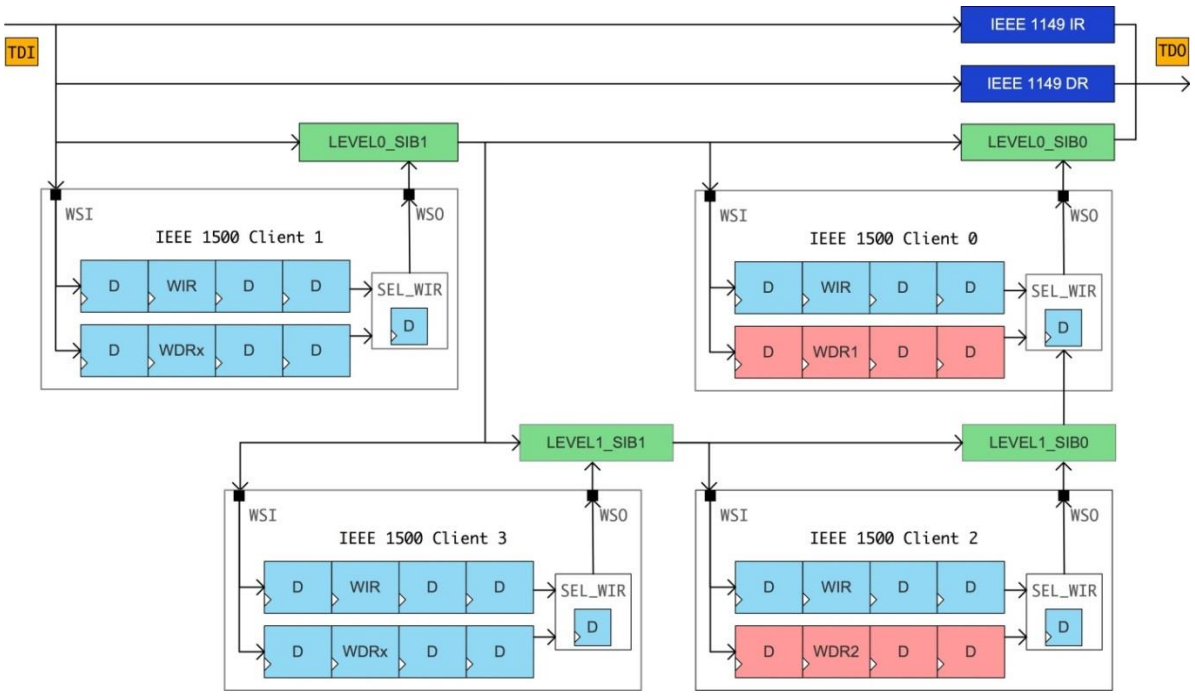
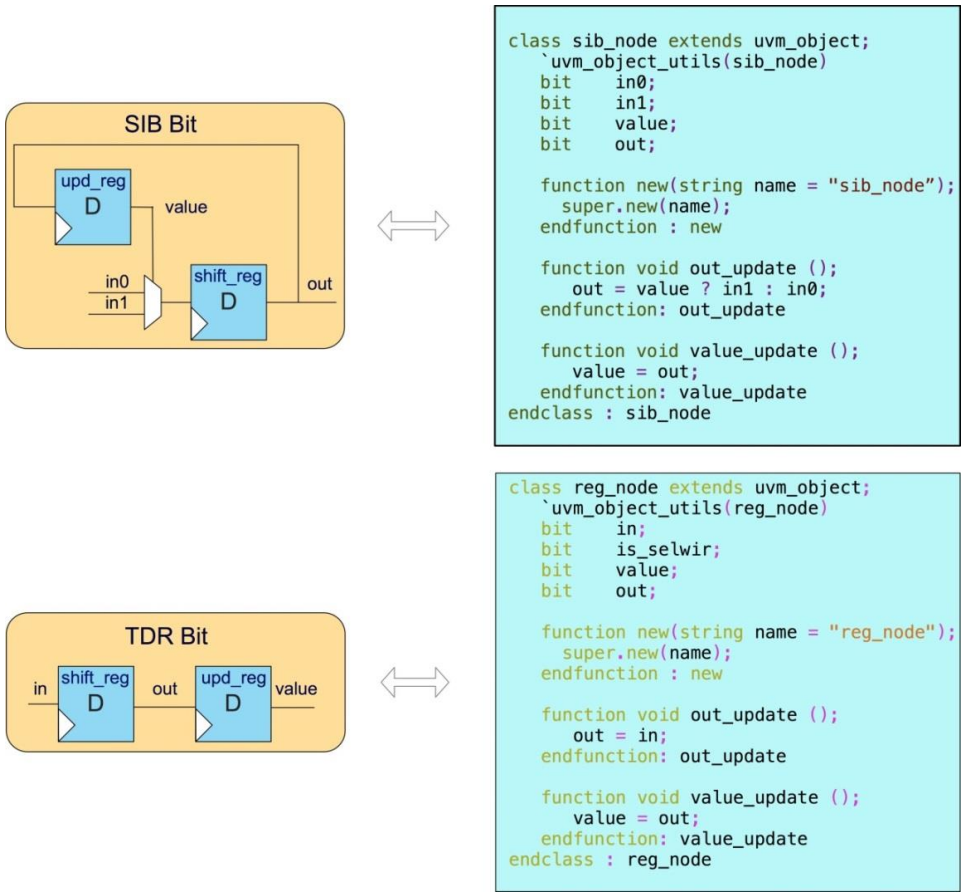


Figure 6. DFT test access network.




```
class sib_node extends uvm_object;
   `uvm_object_utils(sib_node)
   bit    in0;
   bit    in1;
   bit    value;
   bit    out;

   function new(string name = "sib_node");
     super.new(name);
   endfunction : new

   function void out_update ();
      out = value ? in1 : in0;
   endfunction: out_update

   function void value_update ();
      value = out;
   endfunction: value_update
endclass : sib_node
```

```
class reg_node extends uvm_object;
   `uvm_object_utils(reg_node)
   bit    in;
   bit    is_selwir;
   bit    value;
   bit    out;

   function new(string name = "reg_node");
     super.new(name);
   endfunction : new

   function void out_update ();
      out = in;
   endfunction: out_update

   function void value_update ();
      value = out;
   endfunction: value_update
endclass : reg_node
```

Figure 7. DFT test access network elements modelling.

```
class dft_reg_transaction extends uvm_sequence_item;
    `uvm_object_utils( dft_reg_transaction )

    bit                                 read_not_write;
    bit [`DFT_REG_ADDR_WIDTH-1:0]       addr;
    logic                               wr_data_q[$];
    logic                               rd_data_q[$];
    bus_reg_ext                         extension;
    int unsigned                        reg_length;
    ...
endclass: dft_reg_transaction
```

```
class jtag_transaction extends uvm_sequence_item;
    bit                                 o_ir[];
    rand  int unsigned                  o_dr_length;
    rand  int unsigned                  o_ir_length;
    bit                                 o_dr[];
   //tdo_dr_queue/tdo_ir_queue  store tdo data
    bit                                 tdo_dr_queue[$];
    bit                                 tdo_ir_queue[$];
    //tdi_dr_queue/tdi_ir_queue  store tdi data
    bit                                 tdi_dr_queue[$];
    bit                                 tdi_ir_queue[$];
    bit                                 chk_ir_tdo;
    bit                                 chk_dr_tdo;
    bit                                 exp_tdo_dr_queue[$];
    bit                                 exp_tdo_dr_mask_queue[$];
    bit                                 exp_tdo_ir_queue[$];
    rand  bit                           read_not_write;
    ...
endclass:jtag_transaction
```

Figure8. *dft_reg_transaction* and *jtag_transaction* properties.

The *chk_ir_tdo* and *chk_dr_tdo* are flags to inform the *jtag_driver* whether to check TDO cycle-by-cycle during shifting IR or shifting DR state. The *exp_tdo_dr_queue* is the golden data expecting the DUT TDO output during shifting DR state.

The *exp_tdo_dr_mask_queue* indicates which bit in *exp_tdo_dr_queue* needs not to be checked.

The *exp_tdo_ir_queue* is the golden data expecting the DUT TDO output during shifting IR state.

I. DFT TDR Encode

A DFT TDR's address is encoded in Figure 4. It composes of three segments, the first segment is the reserved flag bit for the *dft_reg_map* and the *dft_reg_predictor* that communicate with the *dft_reg_adapter* as mentioned in ultra-long length TDR modelling section. The second segment is the TDR's OPCODE, and the third segment is the TDR's location information in the test access network. In Figure 6, WDR1's OPCODE is 8'hFE and is controlled by LEVEL0_SIB1, so its address is encoded as 13'h0FE2. Similarly, the WDR2's OPCODE is 8'h36 and is controlled by LEVEL0_SIB0 and LEVEL1_SIB0, so its address is encoded as 13'h0365. For IEEE1149.1 type TDR, we can simply fill the third segment to zero.

J. dft_reg_tx_to_jtag_tx_sequence Implementation

The *dft_reg_tx_to_jtag_tx_sequence* is a virtual sequence, which gets the *dft_reg_transaction* from the *dft_reg_sequencer*, converts it into the *jtag_transactions* and sends them to the *jtag_sequencer*. The *dft_reg_tx_to_jtag_tx_sequence* decodes *dft_reg_transaction.addr* to get the TDR's location information in the network. Figure 9 shows how a generic *dft_reg_transaction* is converted to a series of *jtag_transaction*s. Because the *dft_reg_transaction.addr*[3:0] is 4'b0101, the TDR being access is controlled by LEVEL0_SIB0 and LEVEL1_SIB0, before shifting the OPCODE and written data, the LEVEL0_SIB0 and LEVEL1_SIB0 should be set first.

Figure 9. The conversion from the *dft_reg_transaction* to *jtag_transactions*

K. *dft_reg_network Implementation*

For the test access network in Figure 6, its elements can be modelled as shown in Figure 10. The IEEE 1500 client in the test access network can be divided into two types, the first is controlled by level 0 SIBs and the second is controlled by level 1 SIBs. Because we suppose only access one WDR in a IEEE 1500 client each time, it is unnecessary to model every IEEE 1500 client and all the WDRs in it. It only need to model the SEL_WIR, the WIR and a WDR of each type of IEEE 1500 client. The WDR's length is dynamic, which can be calculated by the *jtag_transaction.o_dr _length* and the current SIBs' value in the *dft_tdr_network*.

By a series of conditional judgments based on the possible paths between TDI and TDO in the test access network and the *jtag_transactions* observed by the *jtag_monitor*, the *dft_tdr_network* can return a generic *dft_reg _transaction* to the *dft_reg_monitor*, who passes it to the *dft_reg_predictor*.

In Figure 9, the step 1 to step 3 are used to open SIBs. If we want to access another WDR in the same IEEE1500 client, with the current solution, the five steps in Figure 9 are repeated. In fact the first three steps can skip if we do not close them in the step 5. As mentioned above, the WDRs controlled by identical SIBs are accessed in tandem with a high possibility. If the *dft_reg_tx_to_jtag_tx_sequence* knew each SIB's status in the network, then it could open and close SIBs smartly according to the current WDR being accessed, so as to save unnecessary SIB opening steps. In this way we can save the test time in ATE test.

To realize this, a *dft_ntwk_info* is defined. Figure 11 shows the properties of the *dft_ntwk_info* class.

A *uvm_blocking_put_port* is added in the *dft_reg_monitor* and a *uvm_blocking_put_imp* port is added in the *dft_reg_sequencer* shown in Figure 1. The *dft_ntwk_info* is passed from the *dft_reg_monitor* to the *dft_reg _sequencer* whenever the *dft_tdr_network* update SIBs. The *dft_reg_tx_to_jtag_tx_sequence* get current SIB's status from *dft_ntwk_info*, so that it can open or close SIBs as needed.

Suppose that we want to access the WDR2 in Figure 6 continuously for five times, it will take 840 shift cycles if using the method introduced by [1]. Now by adding a pair of uvm_blocking_put port and uvm_blocking_put_imp port to transfer the SIBs' status, it only takes 632 shift cycles. Roughly 25% test time is saved.

IV. Results and Discussion

This paper introduces a layered structure to model DFT TDR in UVM. The Register Layer One and the Transaction Layer can be generally used in different projects. For Register Layer Two, in contrast, the *dft_reg_network* component and *dft_reg_tx_to_jtag _tx_sequence* are project-specific, and users need to model them according to their project's test access network architecture.

At this stage, the *dft_reg_network* and *dft_reg_tx_to_jtag_tx_sequence* are based on the assumption that only one WDR is accessed each time. However, in the next step of work, it is possible to further improve DFT WDR configure efficiency by enhancing the test access network modelling so as to support broadcast mode, which means the WDRs in different IEEE1500 clients that have the same OPCODE can be accessed simultaneously.

Table I shows five simulation results using VCS.

The simulation scenario is as follows:

A TDR is defined to have X fields and each field is one bit. Instance 200 such TDRs and write these 200 TDRs in sequence.

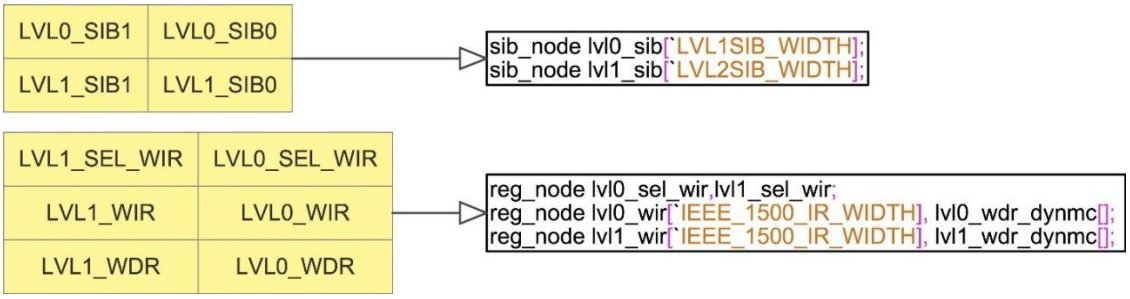


Figure 10. DFT access network elements mapping to defined classes.

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

  bit                    lvl0_sib[`LVL1SIB_WIDTH];
  bit                    lvl1_sib[`LVL2SIB_WIDTH];
  bit                    lvl0_sel_wir,lvl1_sel_wir;
  bit                    lvl0_wir[`IEEE_1500_IR_WIDTH];
  bit                    lvl1_wir[`IEEE_1500_IR_WIDTH];
  ...
endclass : dft_ntwk_info
```

Figure 11. *dft_ntwk_info* properties.

TABLE I
MEMORY USAGE OF WRITING 200 ULTRA-LONG LENGTH TDRS

	512	1024	2048	4096
Memory Saving Solution	2074.7 MB	2344.8 MB	2981.2 MB	4327.2 MB
Current UVM RAL Solution	2349.5 MB	2759.5 MB	4212.6 MB	9263.6 MB
Memory Saving	11.7%	15.0%	29.2%	53.3%

The row "Memory Saving Solution" shows the memory usage in simulation when using the method introduced by this paper, while the row "current UVM RAL solution" shows the results when using the current UVM RAL solution. The percentage of memory saved is calculated. We found that 11.7% to 53.5% memory is saved when the TDR's length ranges from 512 to 4096 bits.

Table II shows five simulation results using VCS.

The simulation scenario is as follows:

A TDR is defined to have X fields and each field is one bit. Instance 400 such TDRs and write these 400 TDRs in sequence.

The row "Memory Saving Solution" shows the memory usage in simulation when using the method introduced by this paper, while the row "current UVM RAL solution" shows the results when using the current UVM RAL solution. The percentage of memory saved is calculated. We found that 6.8% to 60.0% memory is saved when the TDR's length ranges from 512 to 4096 bits.

The results of Table I and Table II shows:

1. The memory usage of current UVM RAL solution expands significantly when the TDR's length larger than 2048 bits.
2. The benefit of memory saving becomes increasingly obvious when the TDR's length gets longer and the TDR numbers to be accessed increases.

TABLE II
MEMORY USAGE OF WRITING 400 ULTRA-LONG LENGTH TDRS

	512	1024	2048	4096
Memory Saving Solution	2335 MB	2960.3 MB	4066.6 MB	6421.2 MB
Current UVM RAL Solution	2504.2 MB	3542.1 MB	6451.5 MB	16064 MB
Memory Saving	6.8%	16.4%	37.0%	60.0%

REFERENCES

[1] R. Huang, "A universal DFT verification environment: filling the gap between function simulation and ATE test," DVCon 2016 (http://events.dvcon.org/events/proceedings.aspx?id=199-6).