One Stop Solution for DFT Register

Modelling in UVM

Rui Huang

Advanced Micro Devices, Inc., [Rui.Huang@amd.com](mailto: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 serial 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 two macros *UVM\_REG\_DATA\_WIDTH* and *UVM\_REG\_ADDR\_WIDTH* to the value identical to the length of the longest DFT TDR in the system. The width of the two basic data types in UVM RAL, namely *uvm\_reg\_addr\_t* and *uvm\_reg\_data\_t*, is decided by the abovementioned *UVM\_REG\_ADDR\_WIDTH* and *UVM\_REG\_DATA \_WIDTH* macros, respectively. These two data types are constructed, and are 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 (hereinafter 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.

## 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 several *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

## 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 *uvm\_reg\_item.value* is one.

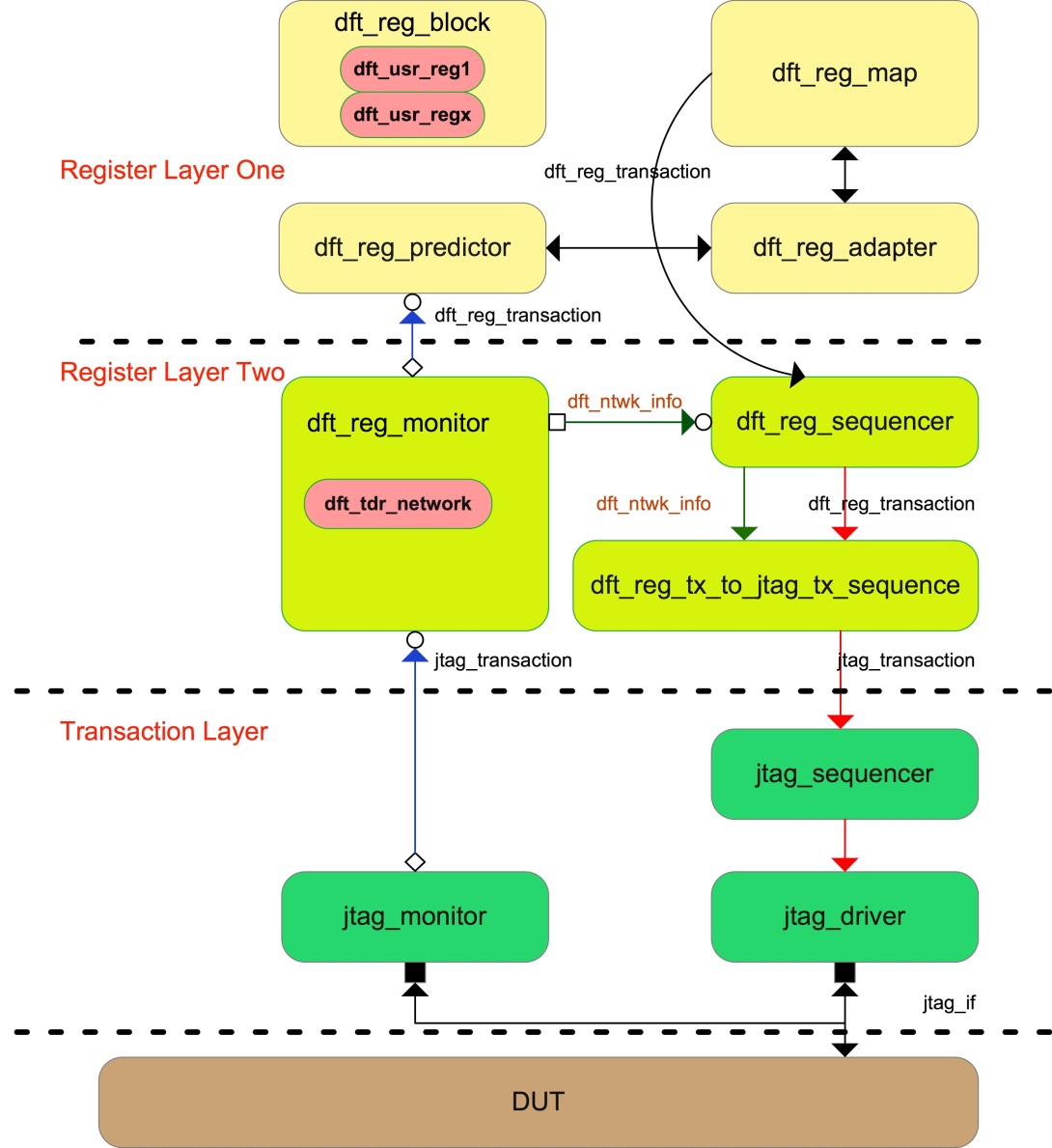


Figure 1. DFT TDR modelling block diagram.

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 *uvm\_reg\_item.value* to be one, so that they construct *uvm\_reg\_item.value* array according to the bit length of the data being processed, or they fetch the data in *uvm\_reg\_item.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 (*uvm\_reg\_item.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*.

1. 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.
2. The *dft\_reg\_predict::write()* creates a *uvm\_reg\_item* object and stores the *uvm\_reg\_bus\_op.data* to *uvm\_reg\_item.value* that is passed to *dft\_reg::do\_predict()*.
3. The *dft\_reg::do\_predict()* concatenates all the data in *uvm\_reg\_item.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().*

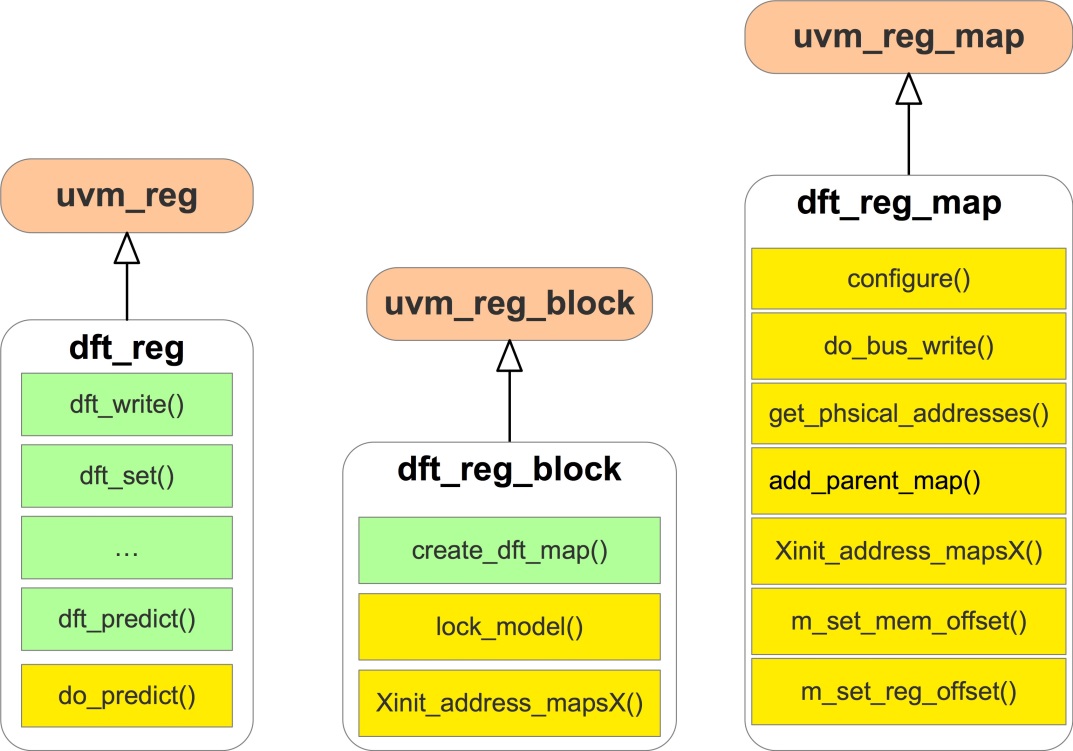


Figure 2. Class extension diagram.

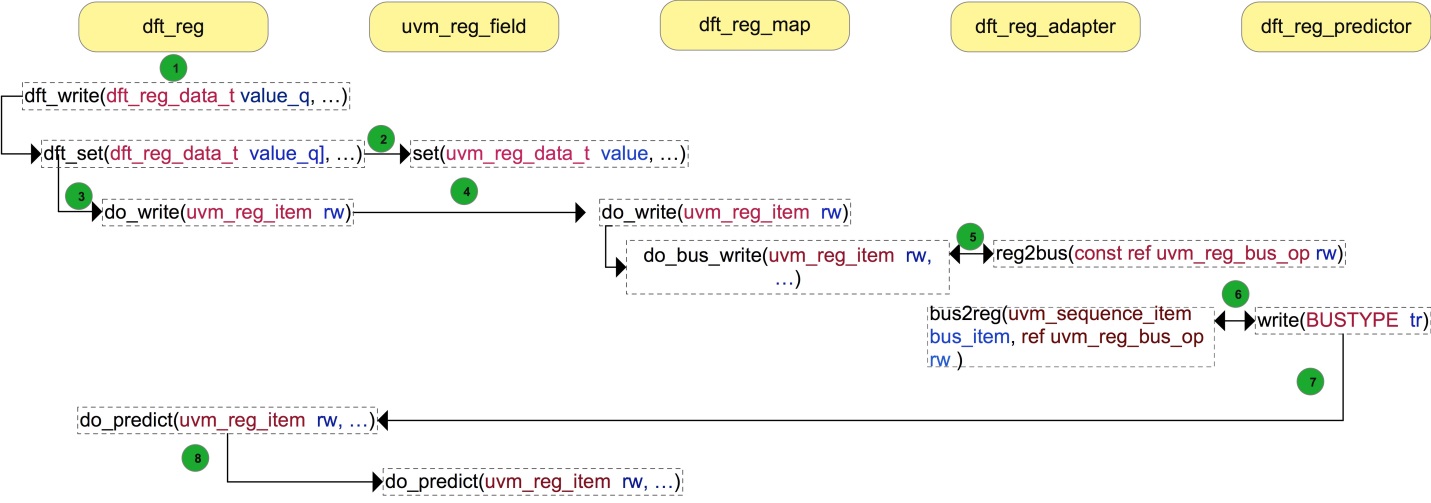


Figure 3. Routine argument passing process of write operation.

## 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 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 *uvm\_reg\_item.value* array size to be one, we need to use *dft\_reg::do\_predict()* to override it.

## 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.

## 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 *uvm\_reg\_item.value* to *uvm\_bus\_reg\_op.data* and set the MSB of the last *uvm\_bus\_reg\_op.addr* to one so as to indicate that all the written data have been transferred as shown in black arrow in Figure 4. Then *dft\_reg\_adpater::reg2bus()* returns a complete generic dft\_reg\_transaction to *do\_bus\_write().*

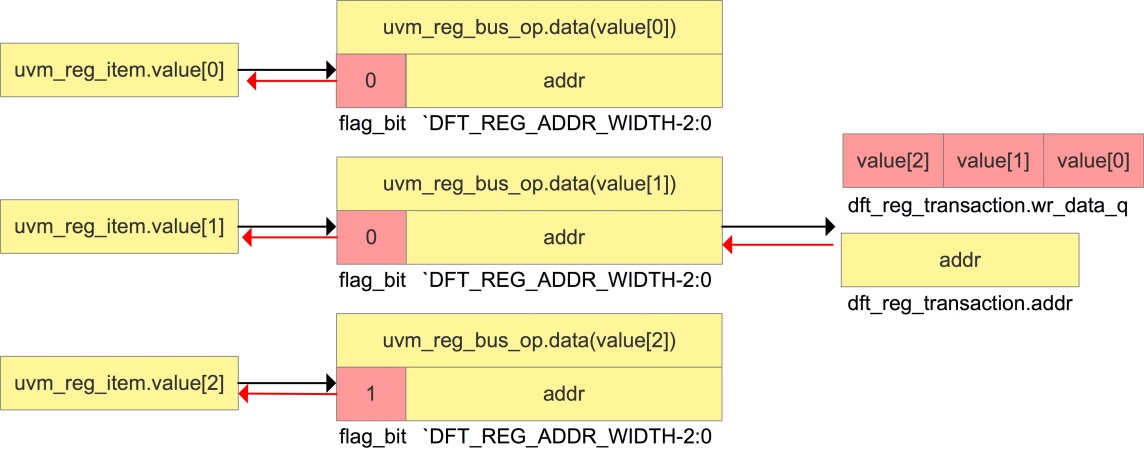


Figure 4. Data conversion process example.

1. Modify *get\_physical\_adresses()* to only returning signal address no matter how long the TDR width is.
2. 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().*
3. Modify *add\_parent\_map()* to supporting *dft\_reg\_map* type.
4. Modify local variable *m\_parent* to *dft\_reg\_block* type, and modify *configure()* function accordingly.

## 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 *uvm\_reg\_bus \_op.addr* 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 4. 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

## Idea Overview

Figure 5 is an example for DFT test access network. In Figure 6, 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.

## dft\_reg\_transaction and jtag\_transaction definition

Figure 7 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 8.

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 or shifting DR state monitored by the *jtag\_monitor*.

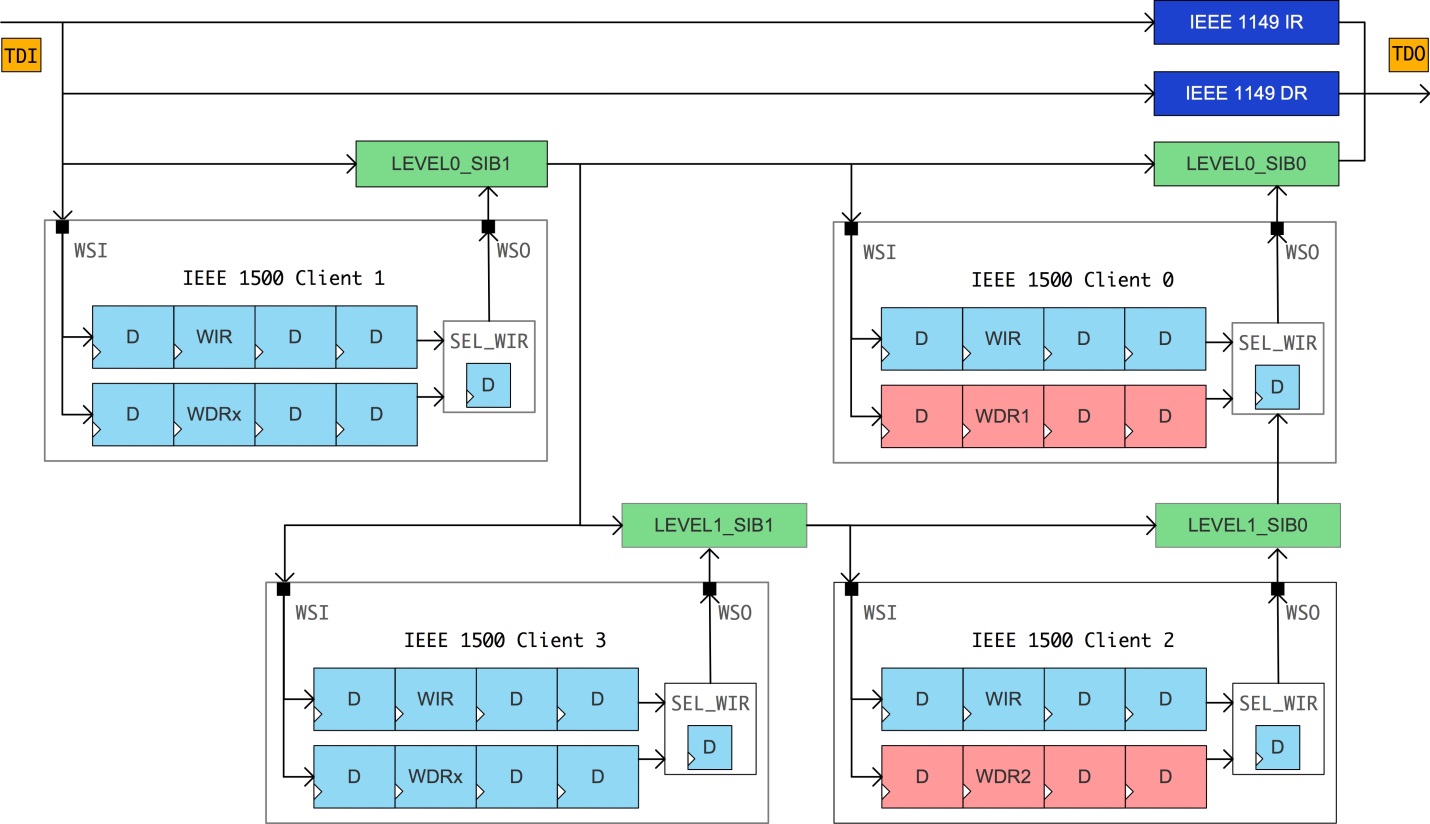


Figure 5. DFT test access network.

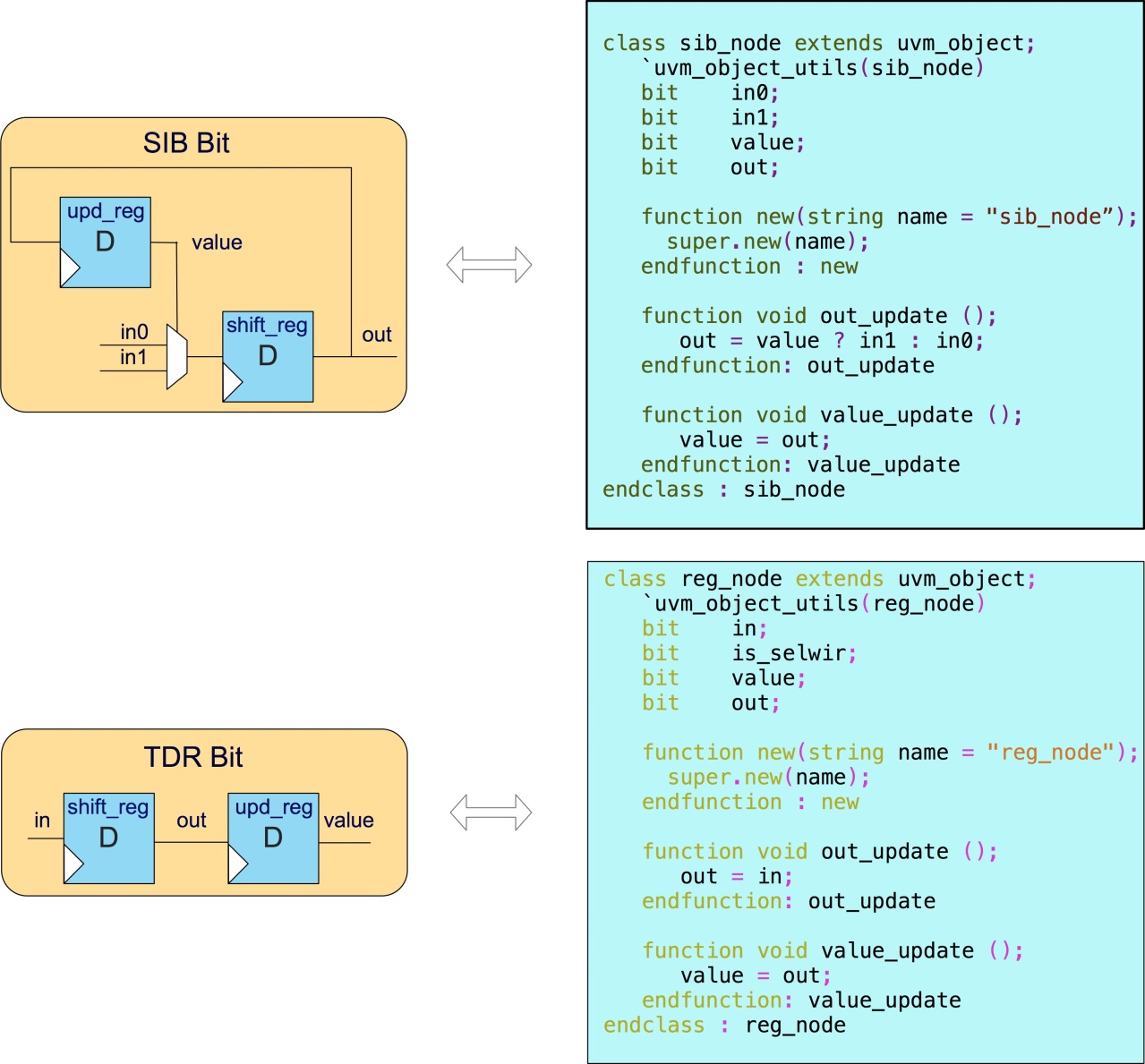


Figure 6. DFT test access network elements modelling.

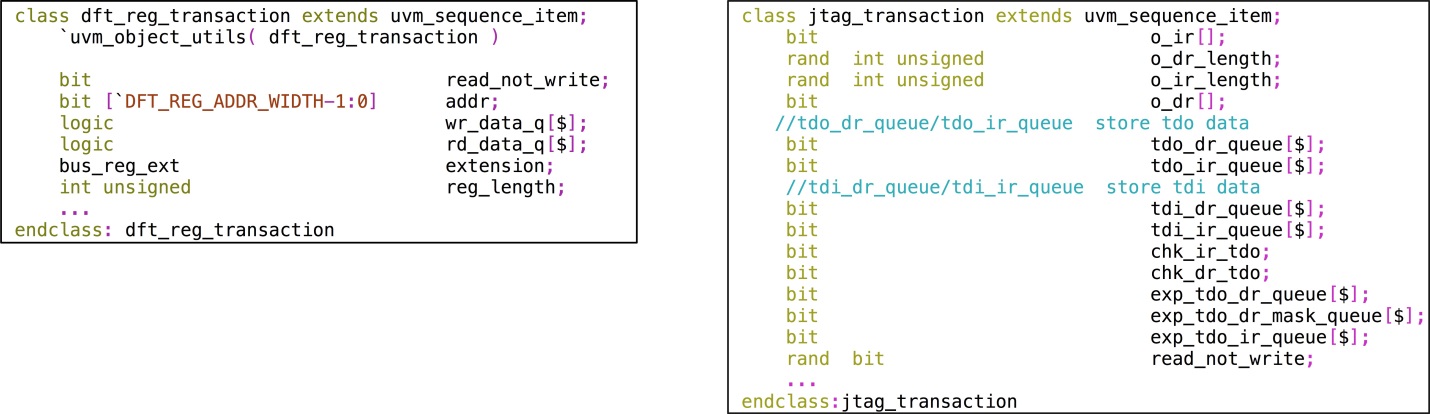


Figure7. *dft\_reg\_transaction* and *jtag\_transaction* properties.

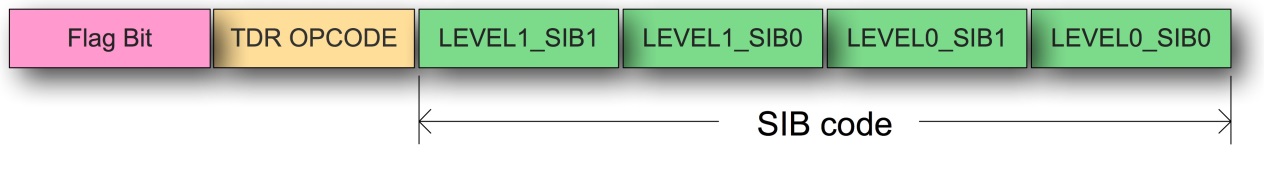


Figure 8. TDR address encoding.

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 (Instruction Register) 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.

## DFT TDR Encode

A DFT TDR’s address is encoded in Figure 8. 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 5, WIR1’s OPCODE is 8’hFE and is controlled by LEVEL0\_SIB1, so its address is encoded as 13’h0FE2. Similarly, the WIR2’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.

## 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 serial 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.

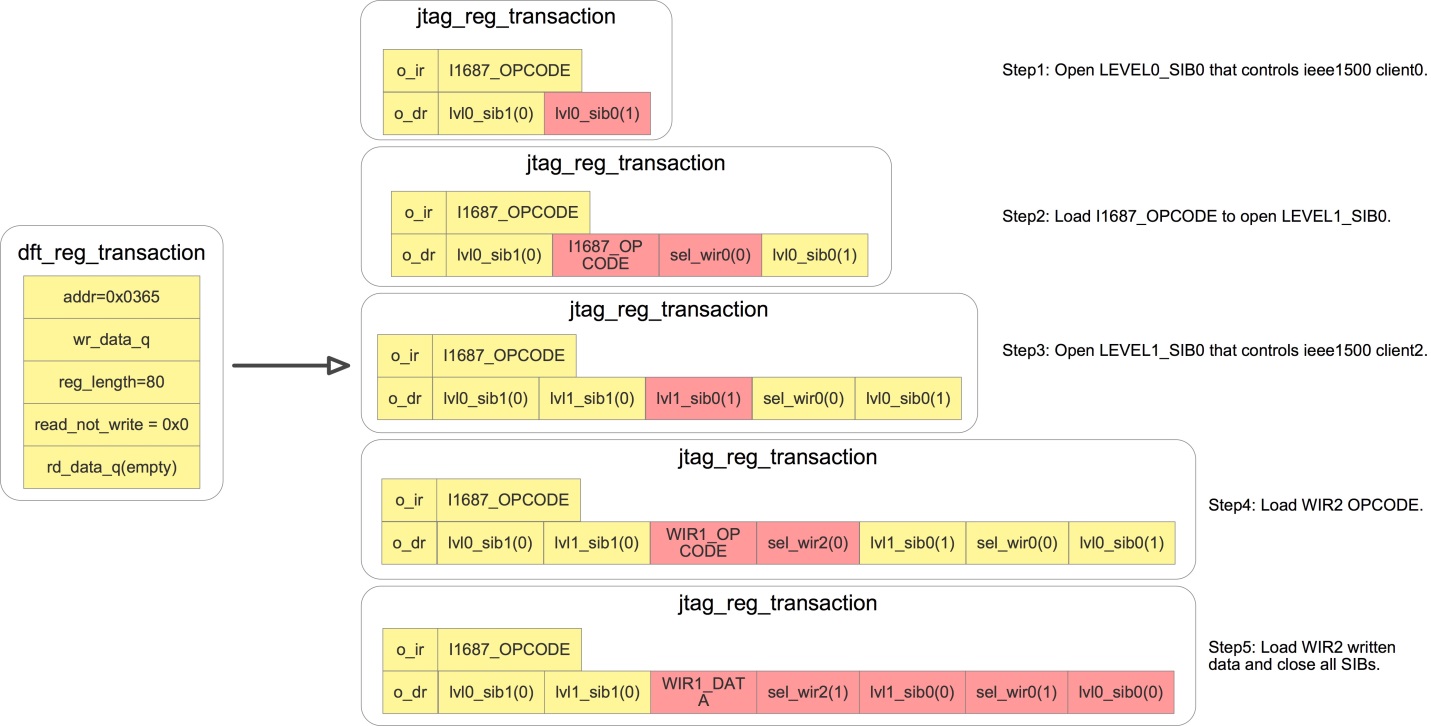


Figure 9. The conversion from the *dft\_reg\_transaction* to *jtag\_transactions*

## dft\_reg\_network Implementation

For the test access network in Figure 5, 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 a WDR in a IEEE 1500 client, 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 serial of conditional judgments based on the possible paths between TDI and TDO in the TDR 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 for the reason of restoring the default state of SIBs and SEL\_WIRs for the convenience of access next WDR. If we want to access another TDR 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 step5. 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 access, 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 WIR2 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 TDR 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 TDR is accessed each time. However, in the next step of work, it is possible to further improve DFT TDR configure efficiency by enhancing the test access network modelling so as to support broadcast mode, which means the TDRs 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.

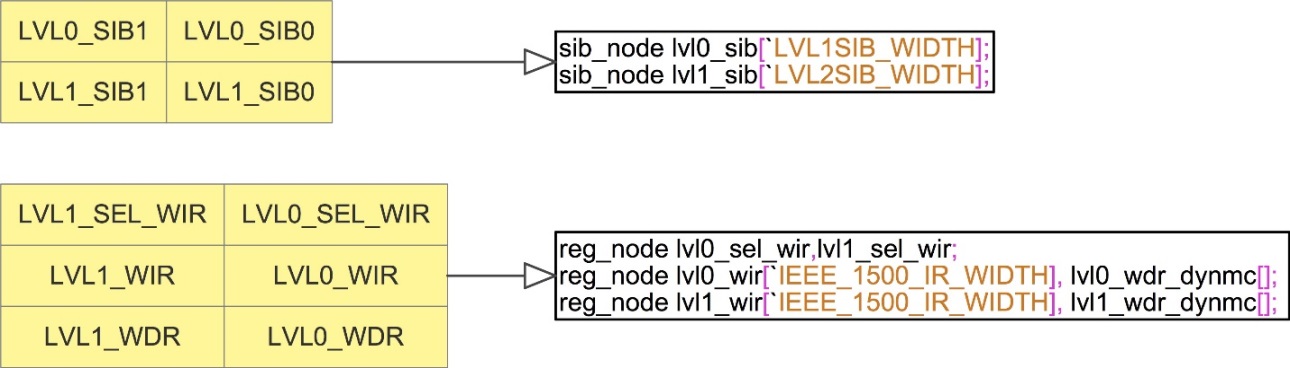
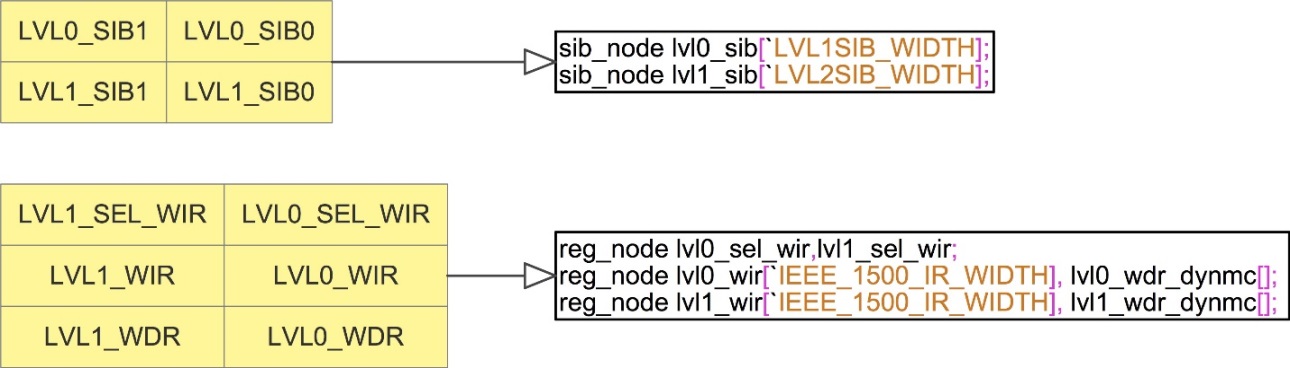


Figure 10. DFT access network elements mapping to defined classes.

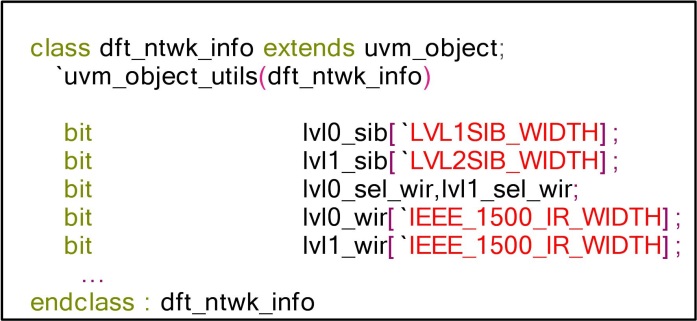


Figure 11. *dft\_ntwk\_info* properties.

Table I

Memory Usage of Writing 200 Ultra-Long Length TDRs

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 512 | 1024 | 2048 | 4096 |
| Memory Saving Solution | 2349.5 MB | 2759.5 MB | 4212.6 MB | 9263.6 MB |
| Current UVM RAL Solution | 2074.7 MB | 2344.8 MB | 2981.2 MB | 4327.2 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 “UVM RAL” 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 “UVM RAL” 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>).