**One Stop Solution for DFT Register Modelling in UVM**

Rui Huang

Advanced Micro Devices, Inc., [Rui.Huang@amd.com](mailto:Rui.Huang@amd.com).

**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. It will be beneficial if abstracting DFT TDR access in RAL, 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 we introduce a new way to model ultra-long length registers in UVM that requires smaller memory space than the current UVM RAL solution.

**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 thousand 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 to functional registers and has its limitation when modelling a DFT TDR of thousand bits. If we want to model DFT TDRs of a system, we need to override two macros (*UVM\_REG\_DATA\_WIDTH* and *UVM\_REG\_ADDR\_WIDTH*) to the value identical to the longest length of the 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 *UVM\_REG\_ADDR\_WIDTH* and *UVM\_REG\_DATA \_WIDTH* macros, respectively. These two data types are instantiated, and used almost everywhere in the RAL related components and objects. Furthermore, when constructing a register, every field of the register is modelled as the *uvm\_reg\_field*, which also profligately uses the *uvm\_reg\_data\_t* type to store the field value, although the width of each field of a register is not long. With the current UVM RAL solution, the side effect of storage waste in simulation is obvious.

This paper presents a resolution for storage issue in modelling ultra-long length DFT TDR using UVM, with minor 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, resulting 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 can be applied to different projects. Huang [1] introduced a novel method to model DFT TDR access network by creating the functional equivalent elements as the DUT and encoding a TDR’s location information in the network to its address using its instruction operation code (a.k.a. OPCODE) and controlling SIBs.

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

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

**Overview:** The DFT TDR layer is divided into two layers as shown in Figure 1. 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 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 jtag\_monitor. In the transaction layer, the *jtag\_transaction*s are passed to the *jtag\_driver* to toggle JTGA interface.

**Structure of This Paper**: This paper is divided into three parts. The first part is about how to model ultra-long length TDR in the register layer 1. The second part is how to model DFT TDR access network in the register layer 2. The third part  
is result and discussion .

In both of the first and second parts, first, a general  
overview will be provided, and then the detailed implementation will be elaborated with reference to an example.

**Ultra-Long Length TDR Modelling in Register Layer One**:

Idea Overview:

By analyzing the methods being called during a register write and read process in UVM RAL, the author find the major limitations in UVM register modelling on ultra-long length TDR 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 the dynamic array size of *uvm\_reg\_item.value*[] is one when process the uvm\_reg\_item pass by.

To remove these limitations, following steps are taken:

1. The *dft\_reg* class is defined as shown in Figure 2 as the base class when construing DFT TDRs. A set of register access methods which use “dft” as prefix are added to replace the correspond one in uvm\_reg class. This set of methods uses dft\_reg\_data\_t which is the 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 that suppose the dynamic array size of *uvm\_reg\_item.value*[] is one when process the uvm\_reg\_item pass by, to let them construct *uvm\_reg\_item.value*[] array according to the *dft\_reg\_data\_t* size or process the *uvm\_reg\_item.value*[] array by checking it’s 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 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 the flag bit that indicates the last package of the written data, the *dft\_reg\_adapter::reg2bus()* knows when all the written data are collected and when to return the real *dft\_reg\_transaction* that the *dft\_reg\_map::do\_bus\_write()* is going to send to 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 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 and then disassemble it into several uvm\_reg\_data\_t type according to the TDR’s field width and then pass them to uvm\_reg\_field::do\_predict().

**The dft\_reg Class Implementation:**

As shown in Figure 2, the methods have “dft\_” prefix are newly added for DFT TDR access. To implement this newly added functions and tasks, we can copy the corresponding one in uvm\_reg class and modify the input argument or return type to dft\_reg\_data\_t instead of uvm\_reg\_data\_t. Then add data type conversion when need. Because the uvm\_reg::do\_predict() supposes the *uvm\_reg\_item.value*[] array size is one, we need use dft\_reg::do\_predict() to override it.

The **dft\_reg\_block Class Implementation:**

The dft\_reg\_blcok class is extended form the uvm\_reg\_block as shown in Figure 2. Because many methods use local variables in uvm\_reg\_block class which cannot be seen by extended class, we copy all code in uvm\_reg\_blcok to dft\_reg\_block class and do following changes.

1. Remove the fatal error check that uvm\_reg\_block::max\_size should not 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.

The **dft\_reg\_map Class Implementation:**

The dft\_reg\_map class is extended form the uvm\_reg\_map as shown in Figure 2. Similar as the uvm\_reg\_block, many methods of the uvm\_reg\_map use local variables, so we copy all code in uvm\_reg\_map to dft\_reg\_map class and do following changes.

1. Modify the do\_bus\_write() method to convert each element in the data 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 to indicate all written data have been transferred as shown in black arrow in Figure 4. And then dft\_reg\_adpater::reg2bus() return a complete generic dft\_reg\_transaction to do\_bus\_write().
2. Modify get\_physical\_adresses() to only return 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 support dft\_reg\_map type.
5. Modify local variable m\_parent to dft\_reg\_blcok type and modify configure() function accordingly.

The **dft\_reg\_predictor Class Implementation:**

The dft\_reg\_predictor class is extended form 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 one, indicating the observed dft\_register\_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().

**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 is to model the active clock edge that triggers the shift register bit during shift  
operation, and the value\_update () method is to model 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 in the test access network. The different possible paths from TDI to TDO can be described by System Verilog conditional statements. As such, a functional equivalent dft\_tdr\_network is get.

**The dft\_reg\_transaction and jtag\_transaction definition:**

Figure 7 shows the properties of dft\_reg\_transaction and jtag\_transaction class.

In the dft\_reg\_transaction, the read\_not\_write indicates whether the transaction is a generic read or write operation.

The addr is the TDR’s encoded address as shown in Figure8.

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 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 shift IR or DR state monitored by the jtag\_monitor.

The chk\_ir\_tdo and chk\_dr\_tdo are flags to indicate jtag\_driver whether to check TDO cycle-by-cycle during shift IR or DR state. The exp\_tdo\_dr\_queue is the golden data expecting the DUT TDO output during shift DR state, which is used by the jtag\_driver to check the TDO data on the fly.

The exp\_tdo\_dr\_mask\_queue indicates which bit in exp\_tdo\_dr\_queue needs not to check.

The exp\_tdo\_ir\_queue is the golden data expecting the DUT TDO output during shift IR state, which is used by

jtag\_driver to check the TDO data on the fly.

**DFT TDR Encode:** A DFT TDR’s address is encoded as Figure 8. It composes three segments, the first segment is the reserved flag bit for dft\_reg\_map and dft\_reg\_predictor communicate with 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. 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.

**The dft\_reg\_tx\_to\_jtag\_tx\_sequence Implementation:** The dft\_reg\_tx\_to\_jtag\_tx\_sequence is a virtual sequence, which get the dft\_reg\_transaction from dft\_reg\_sequencer, converts it into jtag\_transactions and send 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. In Figure 9, because the bit 3 to bit 0 of dft\_reg\_transaction.addr is 4’b0101, the dft\_reg\_tx\_to\_jtag\_tx\_sequence knows to access WIR2, it need to open LEVEL1\_SIB0 first, and then open LEVEL1\_SIB0, and last shift WIR2’s OPCODE and written. Figure 9 shows how a generic dft\_reg\_transaction is converted to a serial of jtag\_transactions.

**The 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\_trsaction to the dft\_reg\_monitor, who passes it to the dft\_reg\_predictor.

In Figure 9, the step1 to step3 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 shown in Figure 9. As mentioned above, the WDRs controlled by same 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 as show in Figure 1. The dft\_netwk\_info is passed from the dft\_reg\_monitor to the dft\_reg\_sequencer whenever the dft\_tdr\_network update SIBs.

Suppose we want to access the WIR2 continually five times, it takes 840 shift cycles not counting the FSM transaction cycles before passing the dft\_ntwk\_info to the dft\_reg\_tx\_to\_jtag\_tx\_sequence, now it only takes 632 shift cycles. About 25% test time is saved.

**Results and Discussion:**

**References:**

[1] Huang R. (2016) *DVCon2016* <http://events.dvcon.org/events/proceedings.aspx?id=199-6>.

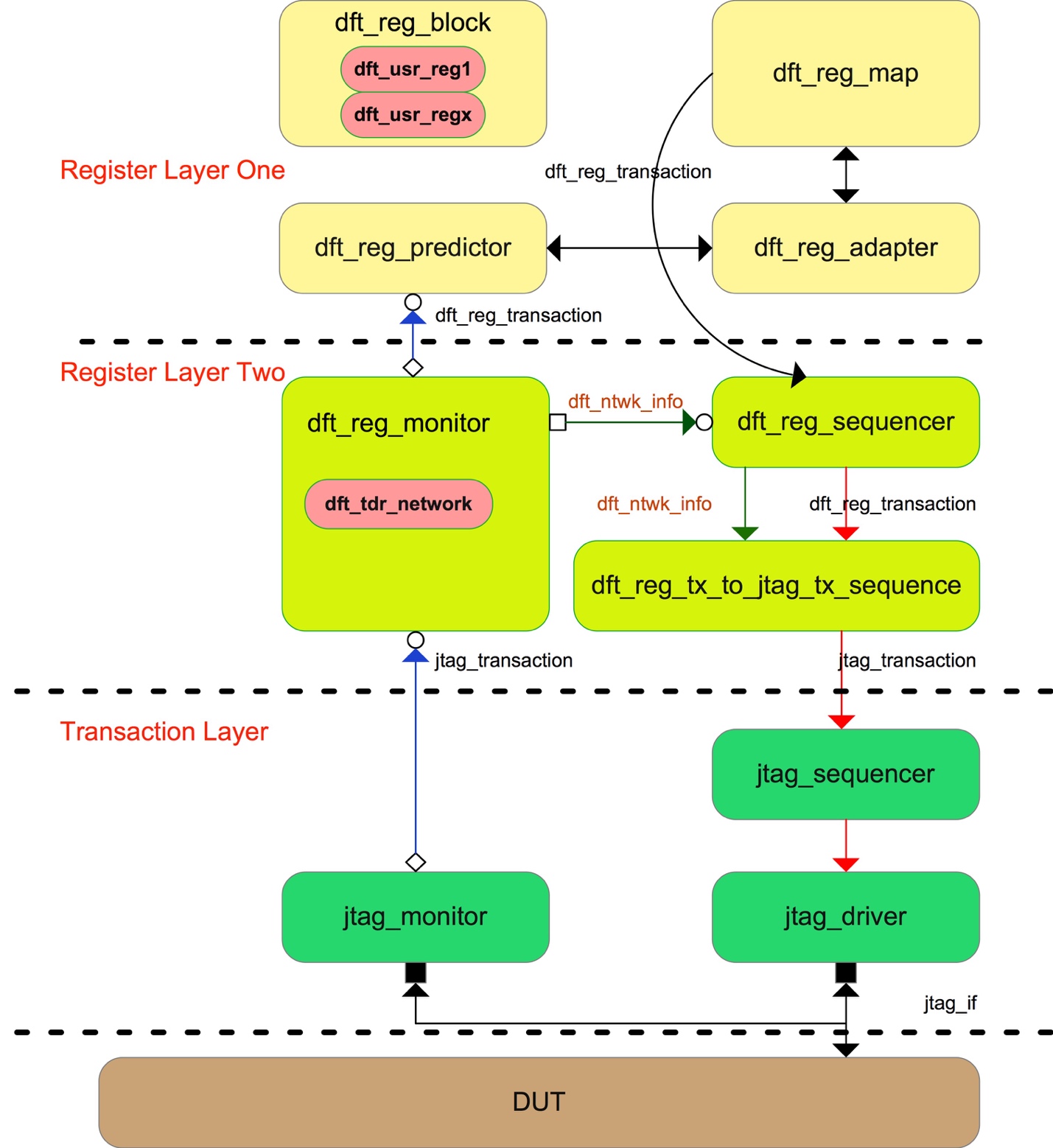


Figure 1 DFT TDR Modelling Block Diagram

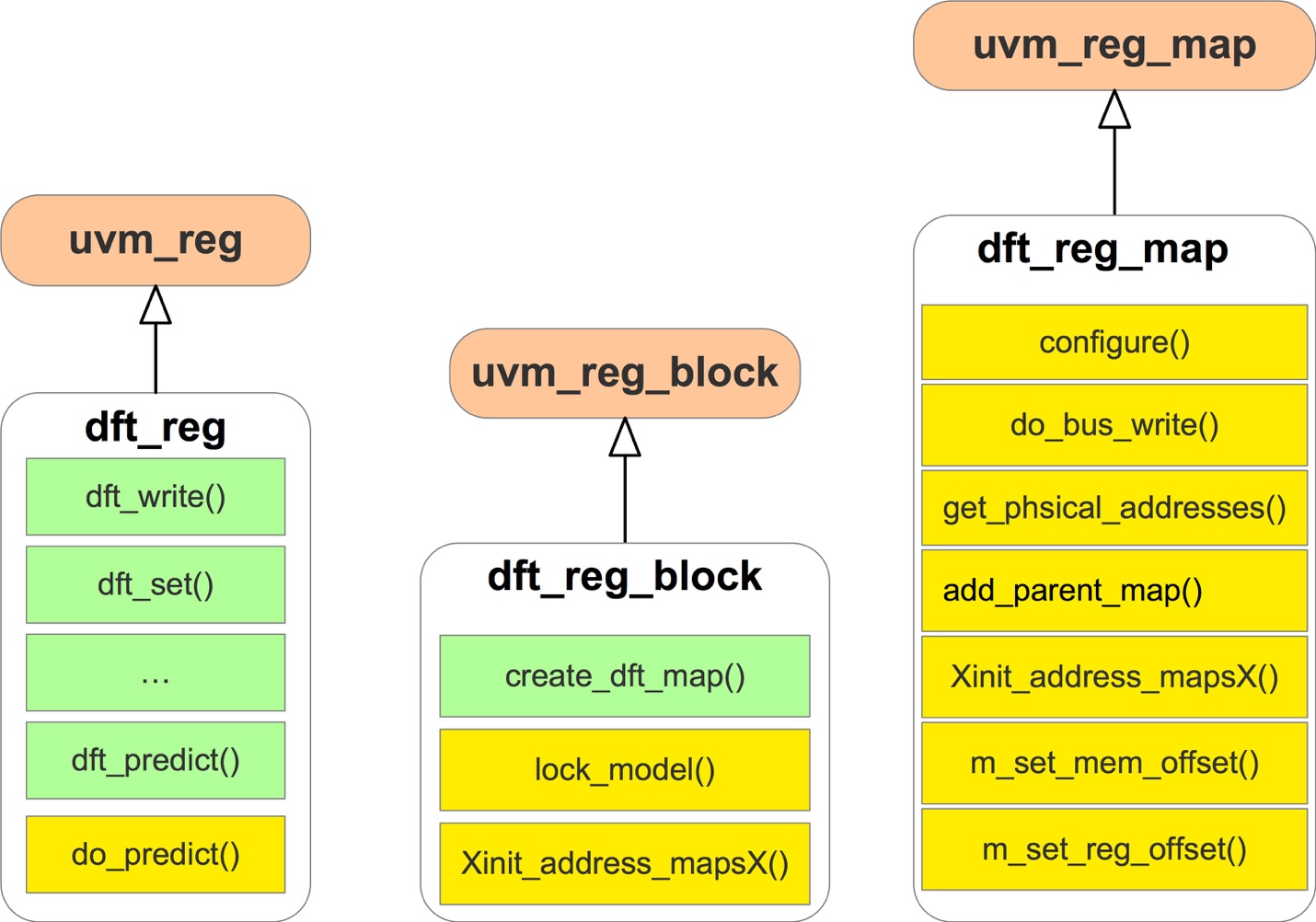


Figure 2

Figure 3

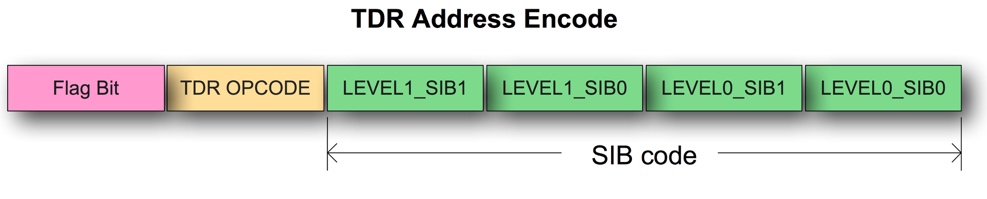


Figure 4

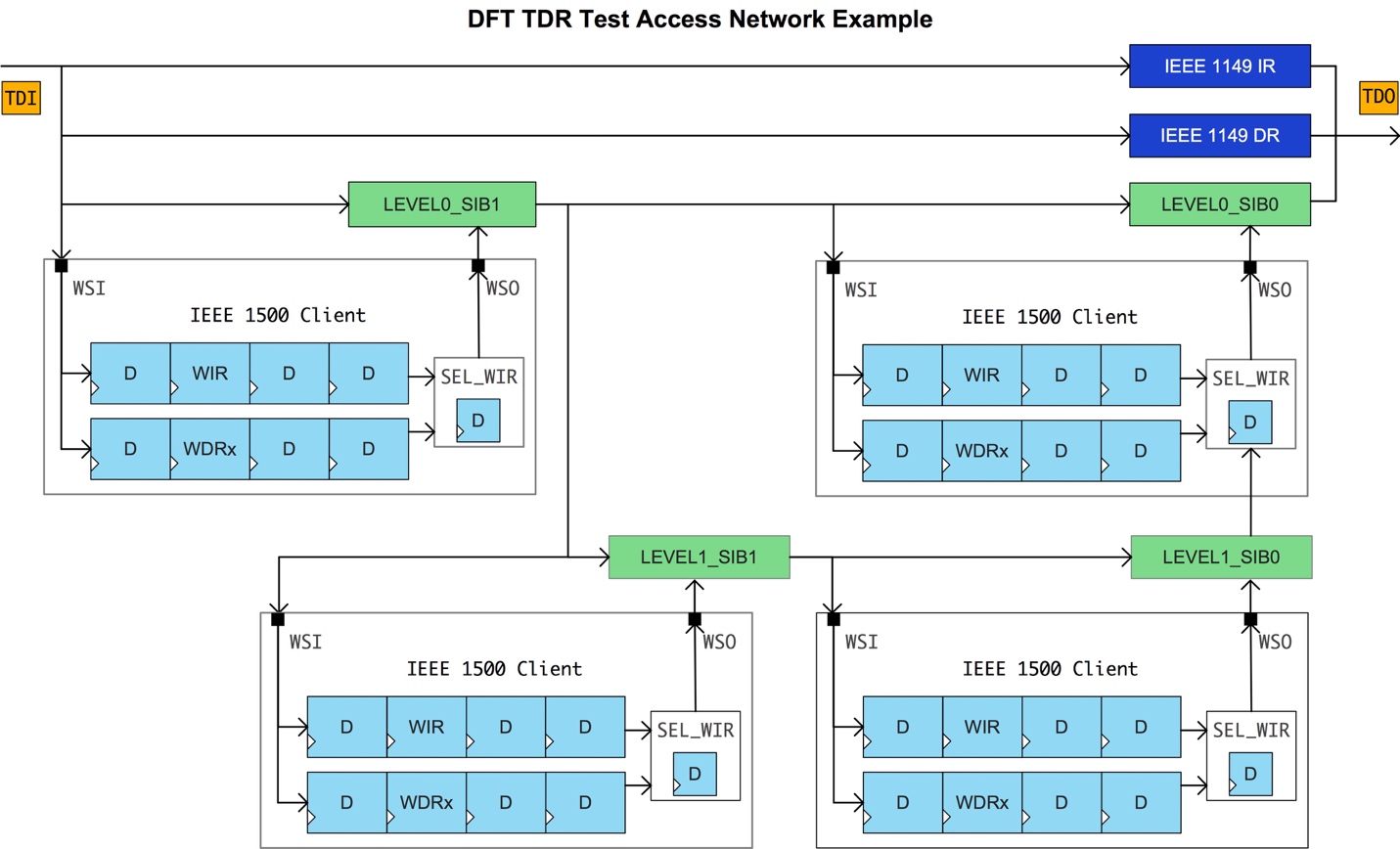


Figure 5

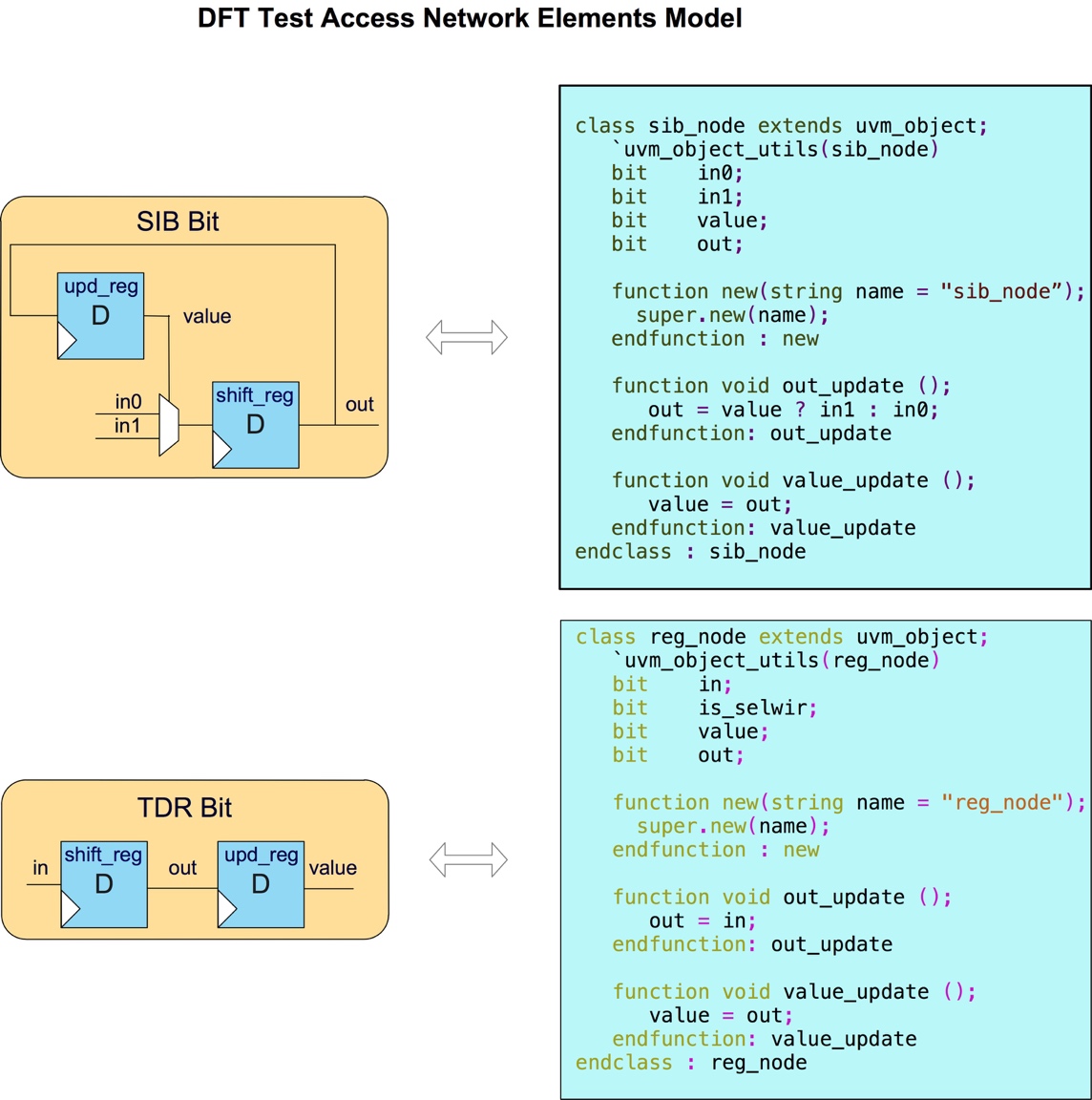


Figure 6

**Results and Discussion**:

The *dft\_tdr\_network* component and the *dft\_reg\_tx\_to\_jtag\_tx\_sequence* virtual sequence is project-specific. Users need to model the *dft\_tdr\_network* component and program the *dft\_reg\_tx\_to\_jtag\_tx\_sequence* according to their project’s TDR test access network architecture.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| UVM | 4096 | 8192 | 2048 | 1024 |
| virtual memory size | 955.2 | 1025.0 | 872.8 | 833.4 |
| resident set size | 651.1 | 720.8 | 568.9 | 529.2 |
| shared memory size | 1.8 | 1.2 | 1.2 | 1.2 |
| private memory size | 649.3 | 719.6 | 567.6 | 528.0 |
| major page faults | 512 | 521 | 0 | 0 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| My(200) | 512 (477 + 474.4) | 1024 | 2048 | 4096 |
| virtual memory size | 894.9 |  |  |  |
| resident set size | 590.9 |  |  |  |
| shared memory size | 1.2 |  |  |  |
| private memory size | 587.7 |  |  |  |
| major page faults | 0 |  |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| UVM (200) | 512 (362+357.2) | 1024 | 2048 | 4096 |
| virtual memory size | 985.9 |  |  |  |
| resident set size | 681.6 |  |  |  |
| shared memory size | 1.2 |  |  |  |
| private memory size | 680.4 |  |  |  |
| major page faults | 523 |  |  |  |

The author improved the method in [1] by adding a pair of *uvm\_blocking\_put*\_*port* and *uvm\_blocking\_put\_imp* port, which transfer the current access network status to *dft\_reg\_sequencer* whenever *dft\_tdr\_network* update its status, so that the *dft\_reg\_tx\_to\_jtag\_tx\_sequence* can use the network status to judge when it need to open or close SIBs to save unnecessary shift cycles.