**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 the IEEE 1687 and 1500 protocols, which makes DFT IP level TDRs can be integrated into SoC easily and modularly. While it causes the DFT test access network become complex and to access a TDR need a serial of complex shift operations. It is necessary to abstract DFT TDR access in RAL to let test writers focus on test sequences and ease tests migration 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 project and resolve the high storage cost issue of UVM when modelling ultra-long width registers.

**Introduction:** To model DFT TDR in UVM, the first challenge is how to model ultra-long width TDRs. In particular, some DFT TDRs’ width, comparing to the functional registers of a SoC, can be as long as thousand bits. At some extreme 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*, 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 TDR is modelled as *uvm\_reg\_field*, which also profligately uses the *uvm\_reg\_data\_t* type to store the field value, although the length of each field of a TDR is not long. With the current UVM RAL solution, the side effect of very high storage cost in simulation is obvious.

This paper presents a resolution for this storage cost issue in modelling ultra-long length DFT TDR using UVM, with minor work needed to override the functions and tasks 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: The abstracted generic DFT TDR access operations should be properly converted into a series of IEEE 1149.1 shift operations (hereinafter called JTAG operations) in an efficient and universal way, so that it can be used in differnt project and thereby save test time in ATE test. 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 OPCODE and controlling SIBs.

The disadvantage of [1] is that, in the *bus2reg* direction, each TDR’s 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 1, 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 2, the generic *dft\_reg\_transaction* is converted into several *jtag\_transaction*s, and the dft\_tdr\_network return the dft\_reg\_transaction by observation 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 1**:

Idea Overview:

By analyzing the functions and tasks being called during a UVM register write and read process in 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 tasks and functions 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 tasks and functions in UVM RAL suppose the dynamic array size of *uvm\_reg\_item.value*[] is 1 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 x 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 shown as Figure x, 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 x, the functions and tasks 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. Figure x shows how dft\_reg::dft\_write() is implemented comparing with uvm\_reg::write(). Since uvm\_reg::do\_predict() supposes the *uvm\_reg\_item.value*[] array size is one, so we need use dft\_reg::do\_predict() to replace it. The data process in dft\_reg::do\_predict() in shown in Figure x.

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

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

1. Modify do\_bus\_write() function 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 write data have been transferred as shown in Figure. 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 according.

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

The dft\_reg\_predictor class is extended form uvm\_reg\_predictor as shown in Figure x. 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 Figure x. 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:** 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.

A DFT TDR’s address is encoded as Figure x. 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 width 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. Figure x shows WIR1 and WIR2 address encoding example, because WIR1 is controlled by LEVEL0\_SIB1, the third segment is encoded as 4’b0010. because WIR2 is controlled by LEVEL0\_SIB0 and LEVEL1\_SIB0, the third segment is encoded as 4’b0101.

By using the sib\_node and reg\_node we can constructs a functional equivalent network as DUT. For the test access network in Figure x, it need to instance four sib\_nodes to model the corresponding SIBs, a fixed width IR and a WDR whose length is dynamic. By a serial of conditional judgments and jtag\_transactions observed by the jtag\_monitor, the dft\_tdr\_network can return a generic dft\_reg\_trsaction to dft\_reg\_monitor, who passes it to the dft\_reg\_predictor.

In the DFT TDR access network, a SIB bit and a TDR bit can be modelled as shown in Figure 18..  
dft\_tdr\_network uses sib\_node and reg\_node to construct a network equivalent to the DUT.  
And it only needs to model each 1500 client’s IR and a WDR (Wrapper Data Register) whose length is dynamic,  
which can calculate from jtag\_transaction coming from jtag\_monitor and current network chain length. It needs  
not to actually model every TDR, because each time only a TDR can be configured in a 1500 client.

The dft\_tdr\_network in Figure 1 reconstructs the same functional network as DUT. A TDR location information is encoded into its address as shown in Figure 4. In Figure 1, the *reg2bus* direction is shown in red lines, where *dft\_reg\_tx\_to\_jtag\_tx\_sequence* fetches *dft\_reg\_transaction*s, unpacks address, decodes SIB code, and then generates *jtag\_transactions* to *jtag\_sequencer*. For the *bus2reg* direction shown in blue lines, dft\_tdr\_network maintains network status using *jtag\_transactions* from *jtag\_monitor*. When *sib\_node* values hit SIB code in the *dft\_reg\_block*, the *dft\_reg\_monitor* writes a *dft\_tdr\_transaction* to *dft\_reg\_predictor*.

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.

**References:**

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

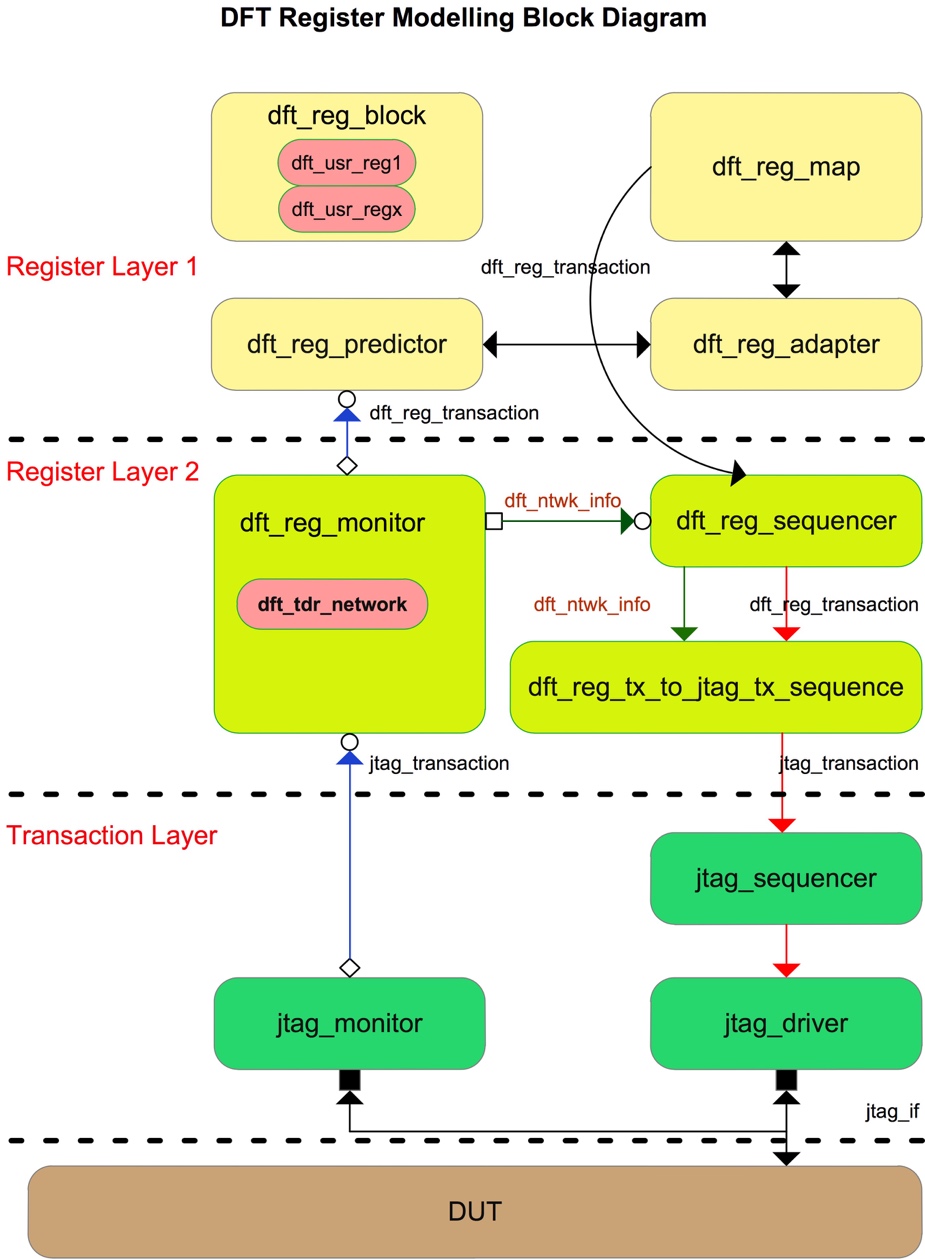


Figure 1

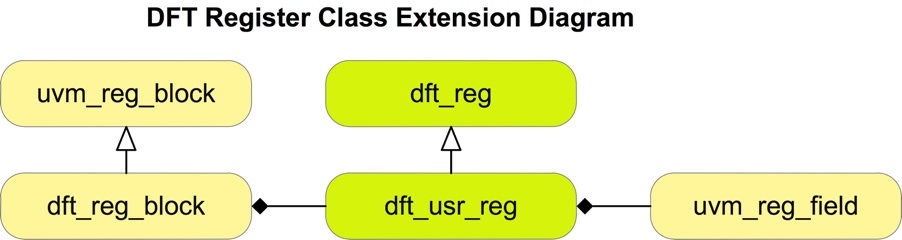


Figure 2

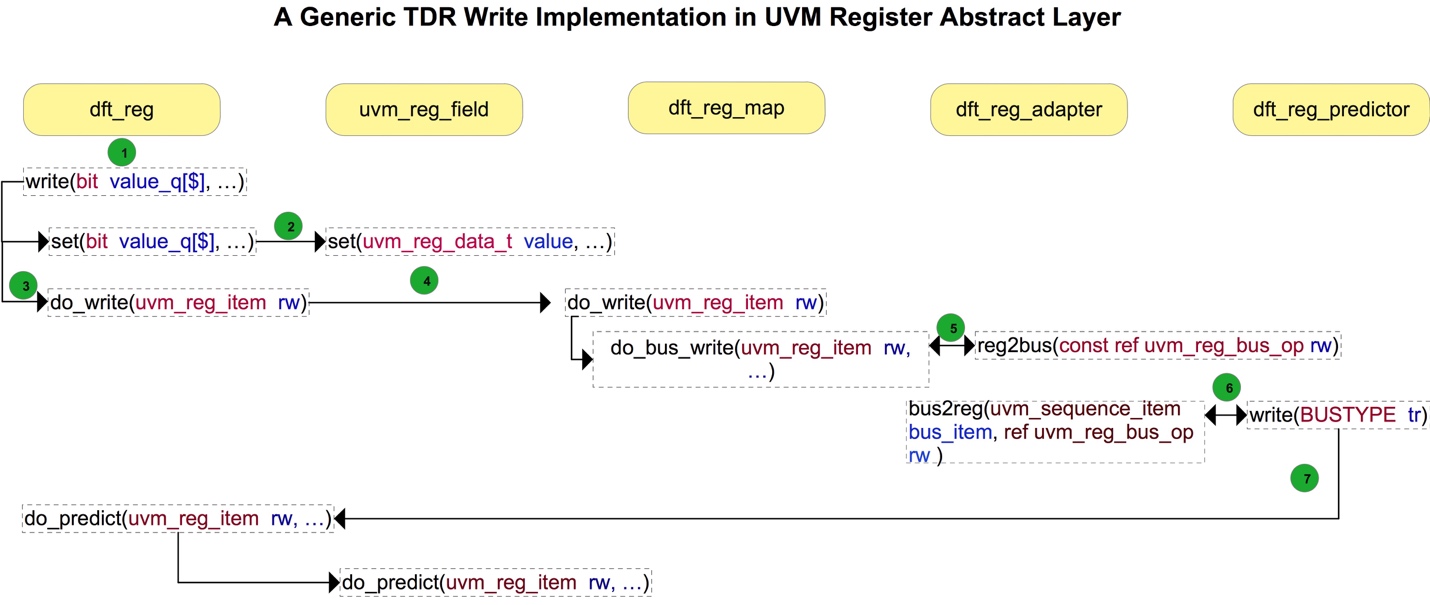


Figure 3

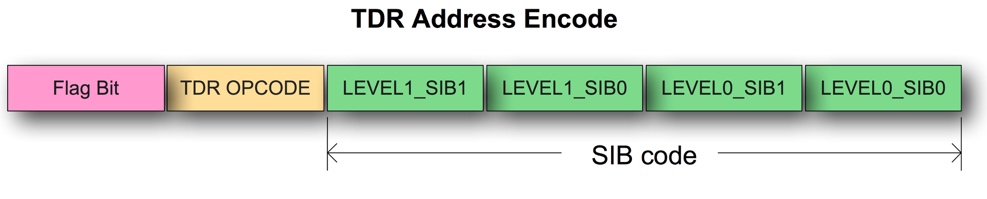


Figure 4

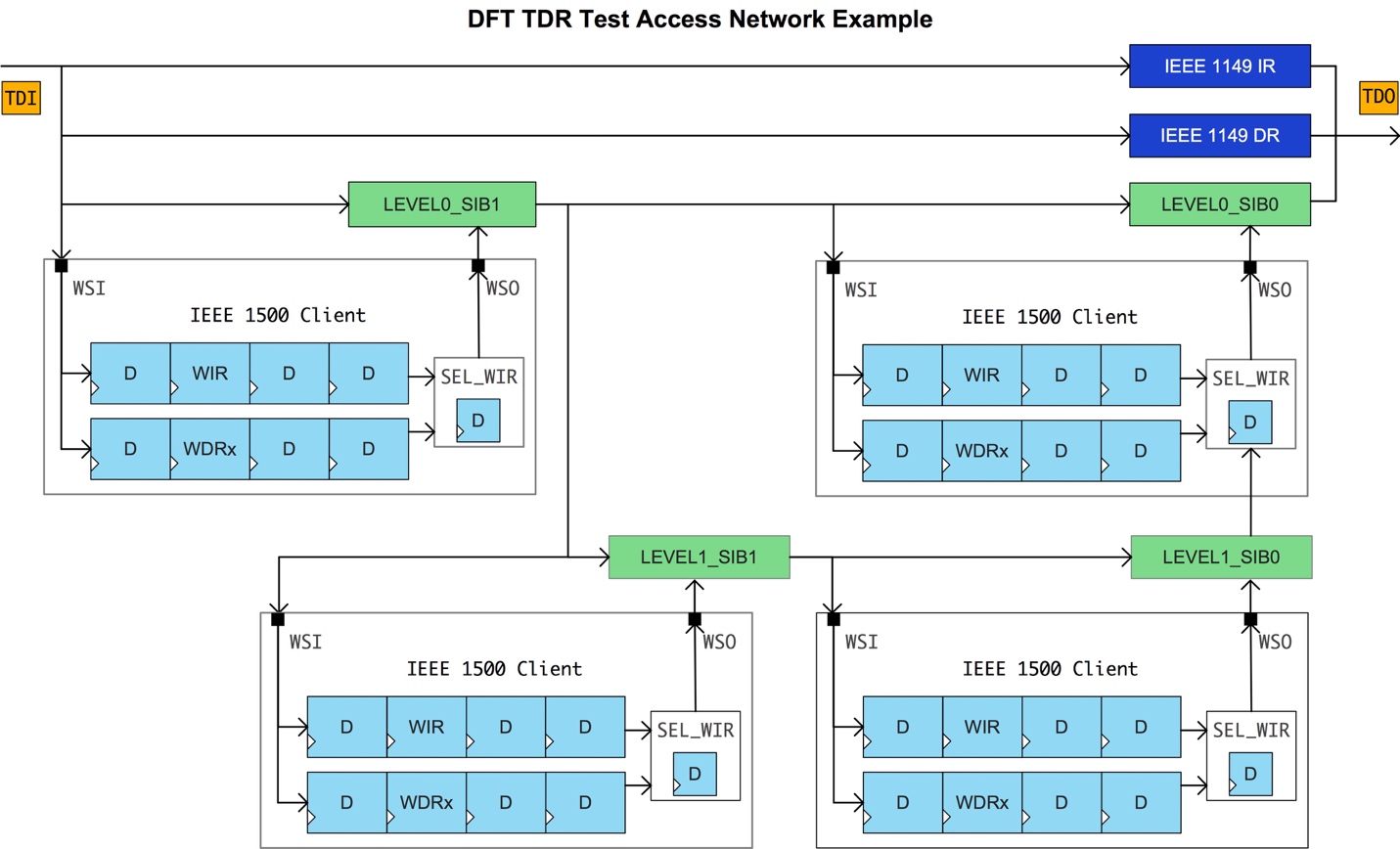


Figure 5

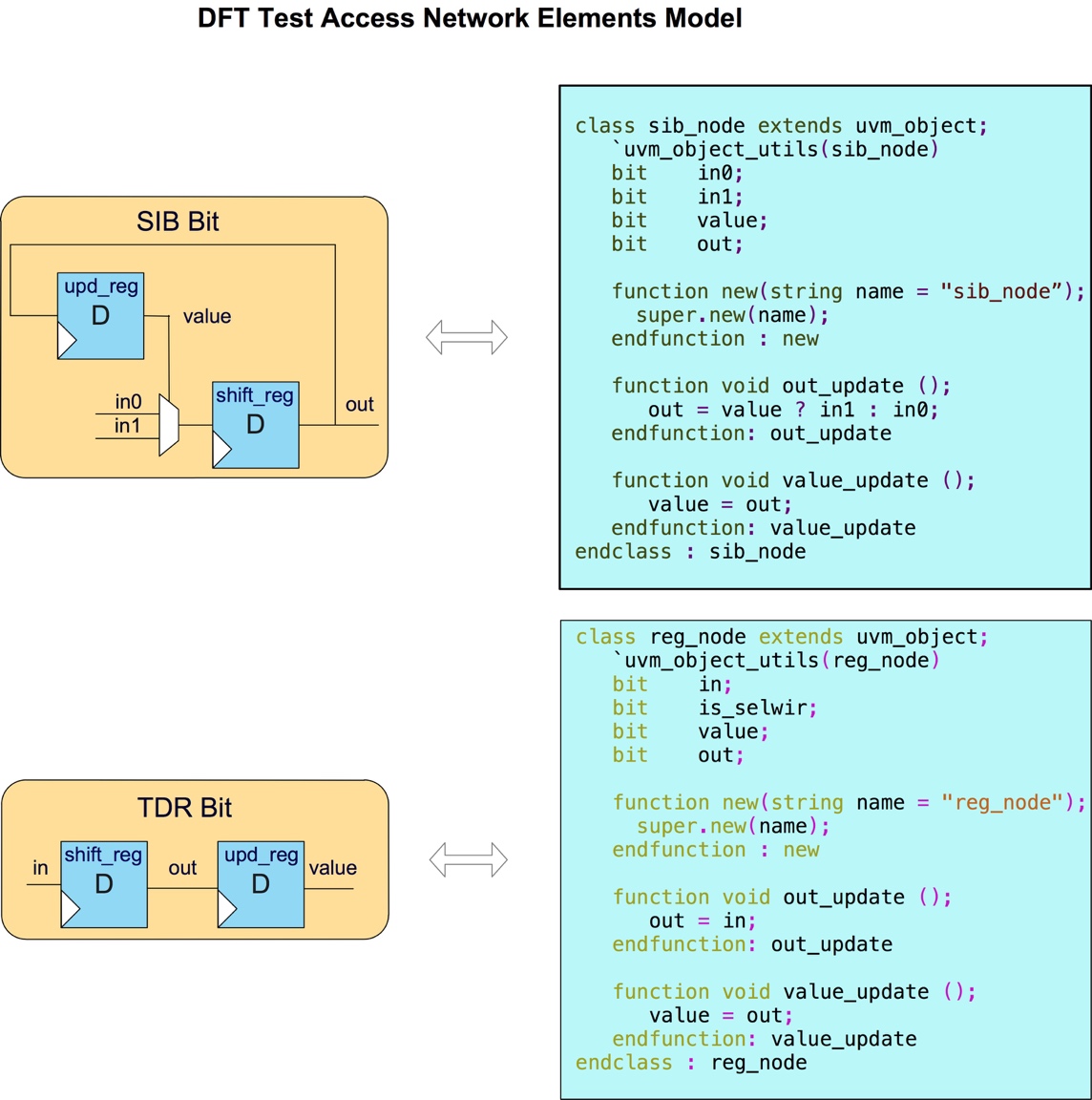


Figure 6

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.