A Universal DFT Verification Environment: Filling the Gap between Function Simulation and ATE Test

Rui Huang
No.2, Science Institute South Rd.
Haidian District, Beijing, China 100190

***Abstract*-The DFT (Design For Testability) design has become more and more complex accompanying the increasing scale of SoC (System on Chip). How to verify DFT logic completely in simulation and how to supply test patterns with high coverage to ATE (Automatic Test Equipment) test are important for post-silicon debug and yield increase. While verification methodology is evolving, innovating and entering the UVM (Universal Verification Methodology) era, DFT verification needs to keep pace to leverage the advantages of UVM, and thereby to increase test reusability, extendibility and function coverage, etc. This paper presents a general UVM-based DFT verification environment, which can be used from modular DFT verification to SoC DFT verification, and it can generate functionally equivalent STIL (Standard Test Interface Language) test patterns for ATE test during SoC simulation. This paper also presents a method to model hierarchically networked DFT TDR (Test Data Register) at RAL (Register Abstract Level) in the UVM environment to allow test writers focus on test sequences without taking care of the details in TDR read and write operations.**

I. Introduction

In DFT (Design For Testability) domain, the test patterns running on an ATE (Automatic Test Equipment) can be categorized into two types: scan related and non-scan related. The former can be generated using ATPG (Automatic Test Pattern Generation) tools, while the latter cannot. Like other function tests, these non-scan DFT function tests are normally created by design verification engineers using languages such as System Verilog or C++. However, ATEs need test patterns described by STIL (Standard Test Interface Language) or other test languages.

To fill the gap, there is usually a dedicated team to transfer function simulation to ATE test environment, or alternatively in-house automation flows are developed to enforce complex rules on test writing and register specification documentation, which are specific for a given environment and difficult to migrate.

This paper provides a universal and more efficient solution by introducing a UVM (Universal Verification Methodology) based DFT verification environment that naturally generates test patterns in STIL format during simulation and can be plugged into any UVM-based environment. This method applies to other formats that ATEs need as well.

For ultra-large-scale SoC (System on Chip), IEEE 1149.1 protocol alone cannot satisfy the DFT design requirements, therefore the IEEE 1687 and 1500 protocols are usually adopted to enable modular and hierarchical DFT test access, leading to challenges when writing test sequences at RAL (Register Abstract Level), as different protocol TDRs (Test Data Register) are hierarchically located in a network connected via IEEE 1687. To access a TDR, one or more levels 1687 SIBs (Segment Insertion Bit) gateways have to be opened and the length of DR (Data Register) chain varies with SIB values. The author also comes up with a general way to model hierarchically networked DFT TDR (Test Data Register) at RAL.

A. Structure of This Paper

This paper is divided into four parts. The first part is about how to build a UVM-based DFT verification environment that can generate STIL test patterns naturally. Then the second part will focus on the method of lifting DFT TDR to RAL. The third part answers how to verify that the generated STIL pattern works. The fourth part is the result discussion and conclusions.

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

II. UVM-Based DFT Verification Environment

B. Idea Overview

The STIL test pattern describes test stimulus using vectors which specify pad drive and measurement information (called STIL information hereinafter) in a time period.

A UVM test usually contains one or several sequences, which are finally broken down into streams of UVM sequence items (a.k.a transactions) and passed to UVM drivers. UVM drivers are normally used to drive and sample pads of DUT, meaning that they also contain the STIL information passing through. In fact, as to be demonstrated in this paper, the UVM drivers are the best supplier of STIL information.

With the precondition that any pad drive and sample are controlled by a UVM driver, which enforces no direct pad connection in the testbench (except for clock pads), simply by collecting all STIL information from the drivers and then writing them out according to the time stamp of STIL information, we can obtain complete test vectors of a certain UVM test when the simulation finishes.

Thus, we can categorize the pads of a SoC into the following types form DFT functional simulation perspective:

1) IEEE 1149.1 compliance on-chip TAP (Test Access Port). Hereinafter, it is simply called JTAG (Joint Test Action Group) interface as shown in Table I, which is the most important interface for DFT design. Please note that in Table I, ***read_not_write*** signal is not defined in IEEE 1149.1, as it is an internal signal only used in this environment, for more description please refer to Section C.5.
2) Clock pads, which are clocks that need to toggle in DFT functional simulation. See Section *D* for more description.
3) Reset pads. All the reset related pads are categorized into this type.
4) Other pads. Except for type 1) to 3) abovementioned, the remaining pads are categorized into this type. See Section E for more description.

In Figure 1, ***jtag_driver***, ***clock_driver***, ***reset_driver***, and ***pad_driver*** correspond to the above four pad types, respectively. The ***STIL_generator*** collects STIL information from these drivers and writes them to a STIL pattern file.

*C. **jtag_agent** Implementation*

In Figure 2, ***jtag_agent*** is composed of ***jtag_sequencer***, ***jtag_monitor***, and ***jtag_driver***, all of them configured through ***jtag_agent_configuration***.

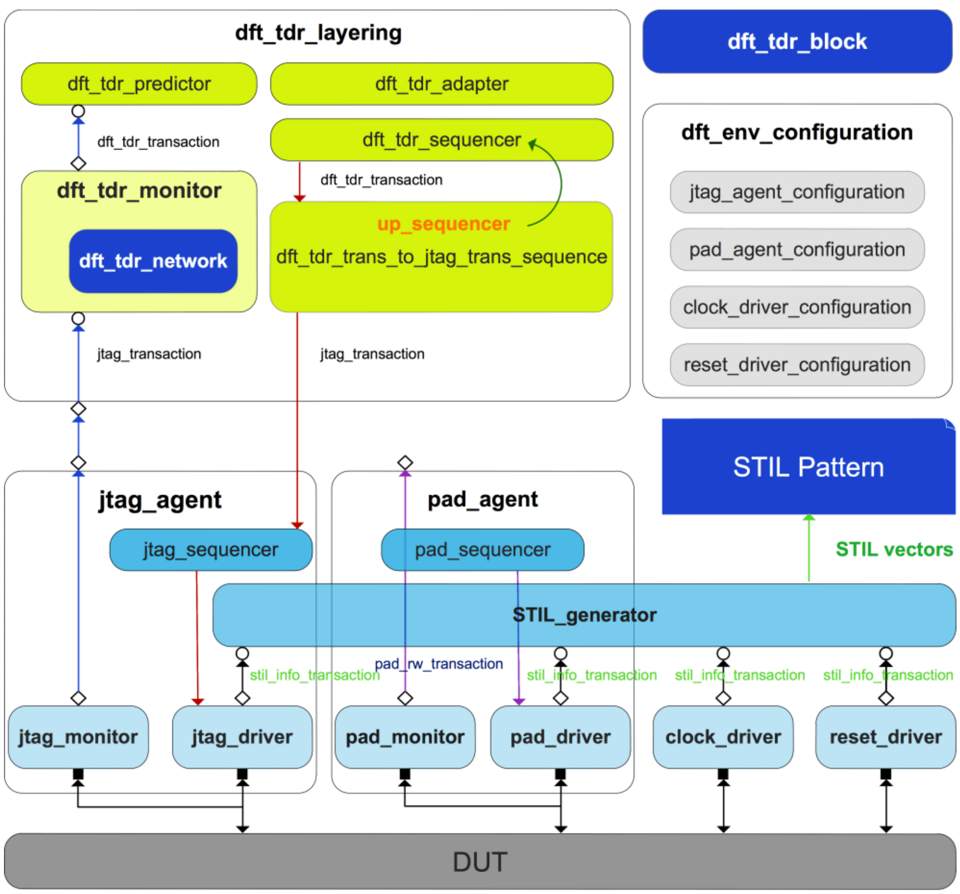


Figure 1. UVM-based DFT verification environment.

TABLE I

JTAG INTERFACE DEFINITION

JTAG Interface	
Pad Direction	Pad Name
input	TCK
input	TMS
input	TRST_L
input	TDI
output	TDO
input	read_not_write

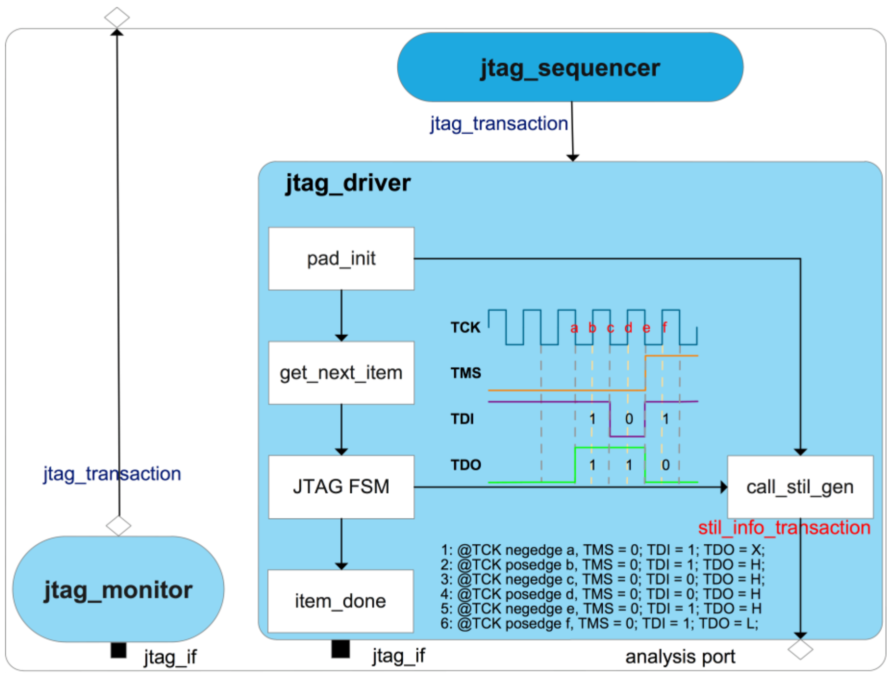


Figure 2. ***jtag_agent*** block diagram.

*C.1. **jtag_agent_configuration** Class*

Figure 3 shows properties and a key method (***pad_info_init*** ()) of ***jtag_agent_configuration*** class.

```
class jtag_agent_configuration extends uvm_object;
   `uvm_object_utils( jtag_agent_configuration )
   virtual jtag_if          jtag_vi;
   dft_register_block       reg_block;
   bit                      gen_stil_file;
   string                   stil_file_name;
   int                      tck_half_period;
   string                   pad_name[3];
   //0: input; 1: output; 2: inout
   int unsigned             pad_dir[3];
   function void pad_info_init();
      pad_name[0] = "TDI";
      pad_dir[0] = 0;

      pad_name[1] = "TMS";
      pad_dir[1] = 0;

      pad_name[2] = "TDO";
      pad_dir[2] = 1;
   endfunction: pad_info_init
   ...
endclass: jtag_agent_configuration
```

Figure 3. ***jtag_agent_configuration*** properties and ***pad_info_init*** () method

C.2. ***jtag_transaction*** *Class*

Figure 4 shows properties of ***jtag_transaction*** class.

o_ir is a dynamic array to store instruction operation code (a.k.a OPCODE) sending to DUT's IEEE 1149.1 FSM (Finite State Machine) IR (Instruction Register) and ***o_ir_length*** is its size.

o_dr is a dynamic array to store data sending to DUT's IEEE 1149.1 FSM DR (Data Register) and ***o_dr_length*** is its size.

tdo_dr_queue, ***tdo_ir_queue***, ***tdi_dr_queue***, and ***tdi_ir_queue*** store data during shift IR or DR state monitored by ***jtag_monitor***.

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.

exp_tdo_dr_queue is golden data expecting DUT TDO output during shift DR state, which is used by ***jtag_driver*** to check TDO data on the fly.

exp_tdo_dr_mask_queue indicates which bit in ***exp_tdo_dr_queue*** need not to check.

exp_tdo_ir_queue is golden data expecting DUT TDO output during shift IR state, which is used by ***jtag_driver*** to check TDO data on the fly.

read_not_write is a flag indicating ***jtag_monitor*** whether it is a read or write operation for current transaction. Please see Section C.5 for more details.

C.3. JTAG Interface Connection in Testbench

This paper categorizes pads of a SoC into four types, which are driven by different drivers, so the JTAG interface shown in Table I is driven by ***clock_driver***, ***reset_driver***, and ***jtag_driver*** as shown in Figure 5.

Figure 6 is ***jtag_if*** interface definition that does not contain all signals shown in Table I because of the categorization of pads. The rest signals are defined in ***clock_if*** and ***reset_if*** interfaces.

C.4. ***jtag_driver*** *Class*

IEEE 1149.1 protocol is implemented in ***jtag_driver***, which fetches every ***jtag_transaction*** sequence item from ***jtag_sequencer***, drives the JTAG interface's TDI and TMS, and samples TDO if ***chk_ir_tdo*** or ***chk_dr_tdo*** flag is on. ***exp_tdo_dr_queue*** and ***exp_tdo_ir_queue*** store the expected golden value, which also will be used as the golden measure information for TDO in the generated STIL pattern.

If the ***gen_stil_file*** knob is on, ***jtag_driver*** not only needs to drive and sample pads – it also converts such information to STIL information (handled by ***call_stil_gen*** () method), and then sends it to ***STIL_generator*** through an analysis port, which is an object of ***uvm_analysis_port*** class specialized with ***stil_info_transaction*** type.

```
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
```

Figure 4. ***jtag_transaction*** properties definition.

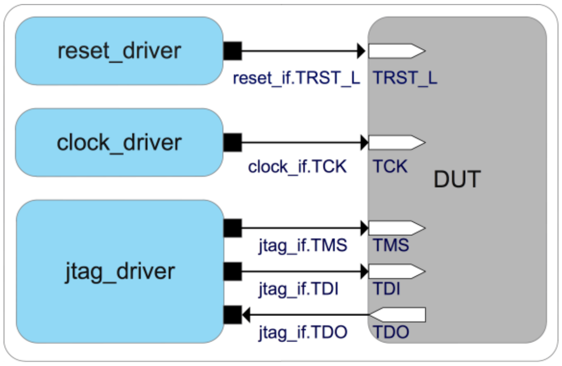


Figure 5. JTAG interface toplevel connection.

```
interface jtag_if( input bit tck, input bit trst);
   logic tdi;
   logic tdo;
   logic tms;
   logic read_not_write;
   ...
endinterface: jtag_if
```

Figure 6. Signals defined in ***jtag_if*** interface.

In Figure 2, let us suppose ***jtag_driver***'s FSM is in shift DR state and it is going to shift three bits 101 to DUT and sample TDO data during shift operation. The golden TDO data are three bits 110.

At TCK negative edge a, ***jtag_driver*** keeps TSM low to let DUT's FSM stay in shift DR state and drives TDI high to send out the first bit out. ***call_stil_gen*** () method converts this information as shown in line 1.

At TCK positive edge b, ***jtag_driver*** samples TDO and compares it with the golden value, which is one bit 1. ***call_stil_gen*** () method converts this information as shown in line 2.

At TCK negative edge c, ***jtag_driver*** keeps TSM low to let DUT's FSM stay in shift DR state and drives TDI low to send out the second bit out. ***call_stil_gen*** () method converts this information as shown in line 3.

At TCK positive edge d, ***jtag_driver*** samples TDO and compares it with the golden value, which is one bit 1. ***call_stil_gen*** () method converts this information as shown in line 4.

At TCK negative edge e, ***jtag_driver*** drives TSM low to let DUT's FSM go to exit1 DR state and drives TDI high to send out the last bit out. ***call_stil_gen*** () method converts this information as shown in line 5.

At TCK positive edge f, ***jtag_driver*** samples TDO and compares it with the golden value, which is one bit 0. ***call_stil_gen*** () method converts this information as shown in line 6.

*C.5. **jtag_monitor** Class*

There is a signal called ***read_not_write*** defined in JTAG interface shown in Table I, which is only used by ***jtag_monitor*** to indicate whether the current transaction is a write operation or read operation.

JTAG interface is a serial bus, while shifting TDI to a register, data stored in it is being shift out on TDO, so there is not a really so-called write or read operation.

Here, we define write operation and read operation in concept for RAL convenience.

Read operation: data being shifted in a register is the same as the data stored in it.

Write operation: data being shifted in a register is different with the data stored in it.

jtag_monitor monitors JTAG interface activity, sampling TDI or TDO according to ***read_not_write*** signal, composing ***jtag_transaction*** sequence items and then passing them to ***dft_tdr_laying*** as shown in the blue arrows of Figure 1.

D. Clock Pads Connection in Testbench

In STIL pattern file, the ***Timing*** block defines sets of "***WaveformTable***s". Each ***WaveformTable*** defines the waveforms to be applied to each signal used in a vector [1]. Because DFT function tests only use JTAG interface to configure TDRs, we define one ***WaveformTable*** in the generated STIL pattern file and use TCK's half period as ***WaveformTable***'s ***Period***. For other clocks describe it as the same frequency as TCK in STIL pattern file and connect them with desired frequency form ATE during post-silicon test. Therefore, ***clock_driver*** only need to drive TCK during simulation and other clocks are generated from testbench (this is the only exception that clock pads are allowed to drive from testbench in this environment).

As shown in Figure 7 for an example, the DUT has two PLL reference clocks and a bypass clock, which need active during simulation, named PLL1_REF, PLL2_REF, and BYPASS_CLK.

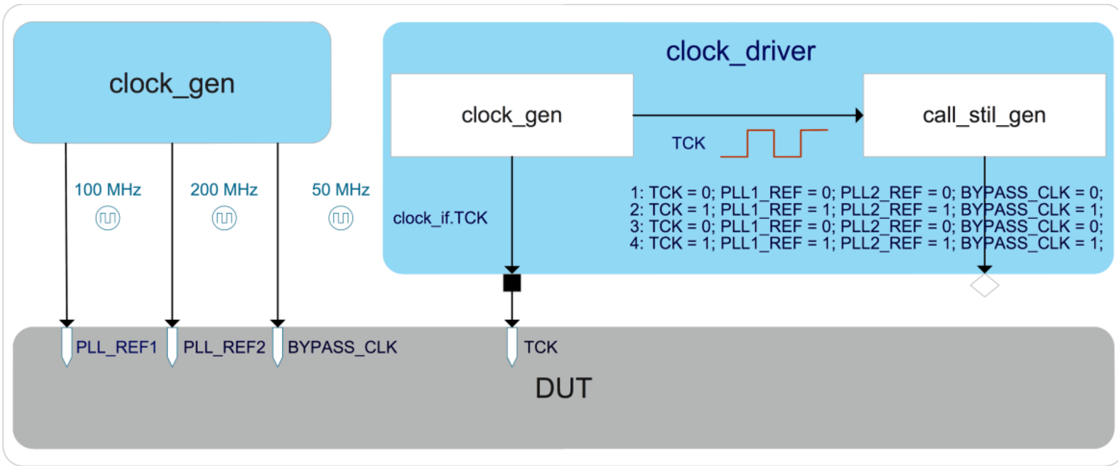


Figure 7. An example of clock pads connection in testbench.

clock_gen module in toplevel takes charge of these three clocks' toggle. TCK of JTAG interface is generated by ***clock_driver***.

If ***gen_stil_file*** knob is on, ***clock_driver*** needs to pass TCK drive information to the ***call_stil_gen*** () method at the same time it drives TCK, and ***call_stil_gen*** () method uses the TCK drive information as all active clocks' drive information and pass STIL information to ***STIL_generator*** as shown in Figure 7 line1 to line4.

For an ATE test, PLL1_REF, PLL2_REF, and BYPASS_CLK toggle information in the STIL pattern can be regarded as a placeholder to make post silicon engineers aware that these three clocks are reference clocks, so that they will not use the toggle information described in STIL patterns to driver reference clocks, but use clocks supplied by ATE with desired frequency.

E. ***pad_agent*** *Implementation*

Figure 8 shows components in ***pad_agent*** and the execution flow in ***pad_driver***, which fetches ***pad_rw_transaction*** from ***pad_sequencer***.

The pad type 4) defined in Section B, can be subgrouped according to their function or interface protocol. Take memory pads, GPIO pads, and scan control pads as examples, each of them could be put in a separate subgroup.

Figure 9 is an example of subgrouping pads according to their interface protocol to define ***pad_if*** interface.

In Figure 8, ***pad_init*** () method initializes all subgroups pads in turn at the beginning of ***run_phase*** task of ***pad_driver***, and ***call_stil_gen*** () method converts this information to STIL information and writes to ***STIL_generator*** through an analysis port.

Figure 10 displays all properties of ***pad_rw_transaction*** class.

grp_num is used to indicate ***pad_dirver*** which group of pads to drive.

in_data_queue stores data being driven by ***pad_driver***.

out_data_queue stores data being sampled by ***pad_driver***.

inout_data_queue stores data being driven or sampled by ***pad_driver***. An unknown bit in the queue indicates ***pad_driver*** the corresponding pad is in output mode and it will write the sampled pad value into the same location.

exp_out_data_queue and ***exp_inout_data_queue*** stores golden value to let ***pad_driver*** check on the fly and also the information for STIL pattern to measure pads value during a time period converting by ***call_stil_gen*** () method.

Please note these queue types should be logic instead of bit in order to store four state values.

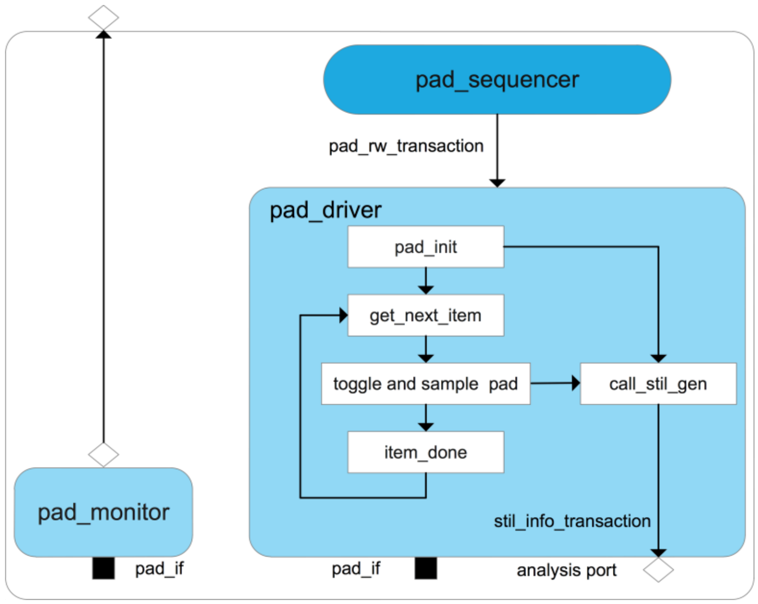


Figure 8. ***pad_agent*** block diagram.

```
interface pad_if( input bit clk);
    logic [`PAD_GRP0_IN_NUM-1:0]      pad_grp0_in;
    logic [`PAD_GRP0_OUT_NUM-1:0]     pad_grp0_out;
    logic [`PAD_GRP0_INOUT_NUM-1:0]   pad_grp0_inout;

    logic [`PAD_GRP1_IN_NUM-1:0]      pad_grp1_in;
    logic [`PAD_GRP1_OUT_NUM-1:0]     pad_grp1_out;
    logic [`PAD_GRP1_INOUT_NUM-1:0]   pad_grp1_inout;
    modport driver_mp(input pad_grp0_out, output  pad_grp0_in,
                      inout  pad_grp0_inout,input  pad_grp1_out,
                      output  pad_grp1_in, inout  pad_grp1_inout);
    modport dut_mp(output pad_grp0_out, input  pad_grp0_in,
                   inout  pad_grp0_inout,output pad_grp1_out,
                   input  pad_grp1_in, inout  pad_grp1_inout);
 endinterface: pad_if
```

Figure 9. An example of defining ***pad_if*** interface in subgroups.

```
class pad_rw_transaction extends uvm_sequence_item;
    int unsigned      grp_num;
    logic             in_data_queue[$];
    logic             out_data_queue[$];
    logic             inout_data_queue[$];
    logic             exp_out_data_queue[$];
    logic             exp_inout_data_queue[$];
    ...
endclass: pad_rw_transaction
```

Figure 10. ***pad_rw_transaction*** properties definition

*E.1. **pad_agent_configuration** Class*

Figure 11 is an example of ***pad_agent_configuration*** class, which has two subgroups of pads.

A DFT test needs to initialize every group's package name by calling the ***pad_info_init*** () method before the ***main phase*** objection and stores it in configuration database for ***pad_driver*** and ***STIL_generator*** fetch.

*F. **reset_driver** Class*

Figure 12 is an example of ***reset_driver*** that drives all resets signals defined in ***reset_if*** interface and ***call_stil_gen*** () method converts drive information to STIL information and writes to ***STIL_generator*** through an analysis port.

*G. **STIL_generator** Implementation*

The ***STIL_generator***, which extends from ***uvm_subscriber*** class specialized with ***stil_info_transaction*** type, has four analysis exports to connect with ***clock_driver***, ***reset_driver***, ***pad_driver***, and ***jtag_driver***'s analysis port separately. Since the ***uvm_subscriber*** class has only one built-in analysis export, the ***uvm_analysis_imp_decl*** macro needs to be used to declare analysis imp export and its associated ***write*** () method for the remaining analysis export [2].

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

   virtual pad_if          pad_vi;
   bit                     gen_stil_file;
   string                  grp0_in_name[`PAD_GRP0_IN_NUM];
   string                  grp0_out_name[`PAD_GRP0_OUT_NUM];
   string                  grp0_inout_name[`PAD_GRP0_INOUT_NUM];

   string                  grp1_in_name[`PAD_GRP1_IN_NUM];
   string                  grp1_out_name[`PAD_GRP1_OUT_NUM];
   string                  grp1_inout_name[`PAD_GRP1_INOUT_NUM];

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

   function void pad_info_init();
      grp0_in_name[0] = "MEMDATA0";
      grp0_in_name[1] = "MEMDATA1";
      ...

      grp1_in_name[0] = "GPIO0";
      grp1_in_name[1] = "GPIO1";
      ...
   endfunction: pad_info_init
endclass: pad_agent_configuration
```

Figure 11. ***pad_agent_configuration*** properties definition example

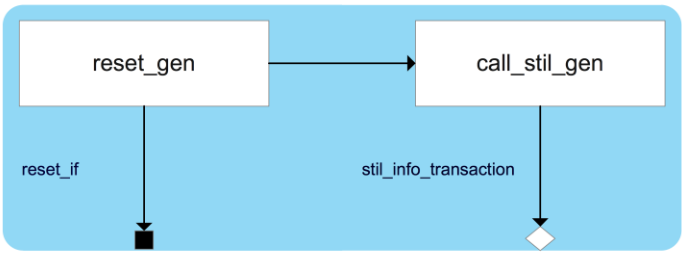


Figure 12. An example of ***reset_driver***.

stil_info_transaction is defined in Figure 13. ***stil_info*** is pads drive and measure information and ***comment info*** is the comment going to be printed out with the ***stil_info***.

In Figure 14, each driver's analysis port has its corresponding ***write*** () method, a semaphore which has only one key and a group of Ping-Pong buffers which have two variables, called ***ping_data_rdy*** and ***pong_data_rdy,*** to indicate Ping-Pong buffer status.

The ***stil_info_transaction*** written through a driver's analysis port is stored in a Ping-Pong buffer group, each buffer stores one ***stil_info_transaction***.

The ***STIL_generator*** needs to collect all ***stil_info_transaction*** coming from the same simulation time slot, to concatenate ***stil_info*** of every ***stil_info_transaction***, and to write them out as a single test vector. To make sure ***STIL_generator*** does not miss any ***stil_info_transaction*** from the same time slot, it has to suspend the ***run_phase*** task in ***STIL_generator*** until all other ***run_phase*** tasks finish. However, in UVM, because all ***uvm_component*** ***run_phase*** tasks are executed in parallel and the ***STIL_generator*** itself is an ***uvm_component***, there is no easy way to schedule the simulation events in ***STIL_generator***'s ***run_phase*** task to be executed after all other drivers' ***run_phase*** tasks finish.

To resolve this issue, a group of Ping-Pong buffers is introduced. The ***write*** () method always writes the ping buffer first and then the pong buffer, so the ping data and pong data come at different simulation time slots. Once a group of Ping-Pong buffers is full, which indicates the simulation has already moved forward, it will be the right time to collect all ping buffer data and write them out.

The ***run_phase*** task of ***STIL_generator***, as shown in Figure 14, always checks if there is at least one driver whose Ping-Pong buffer group is full. If the result is true, it will query each key of the semaphore belonging to the corresponding driver. Once it gets all the keys, it will then fetch all ping buffer data, update the Ping-Pong buffer groups (if both ping and pong buffers are empty, do nothing; if the ping buffer is full and pong buffer is empty, clear ***ping_data_rdy***; if both ping and pong buffers are full, copy the pong buffer data to the ping buffer and clear ***pong_data_rdy***), and put back all keys and write a test vector to STIL pattern.

```
class stil_info_transaction extends uvm_sequence_item;
  `uvm_object_utils( stil_info_transaction )
  string    stil_info;
  string    comment_info;
  string    report_id;
  ...
endclass: stil_info_transaction
```

Figure 13. ***stil_info_transaction*** properties.

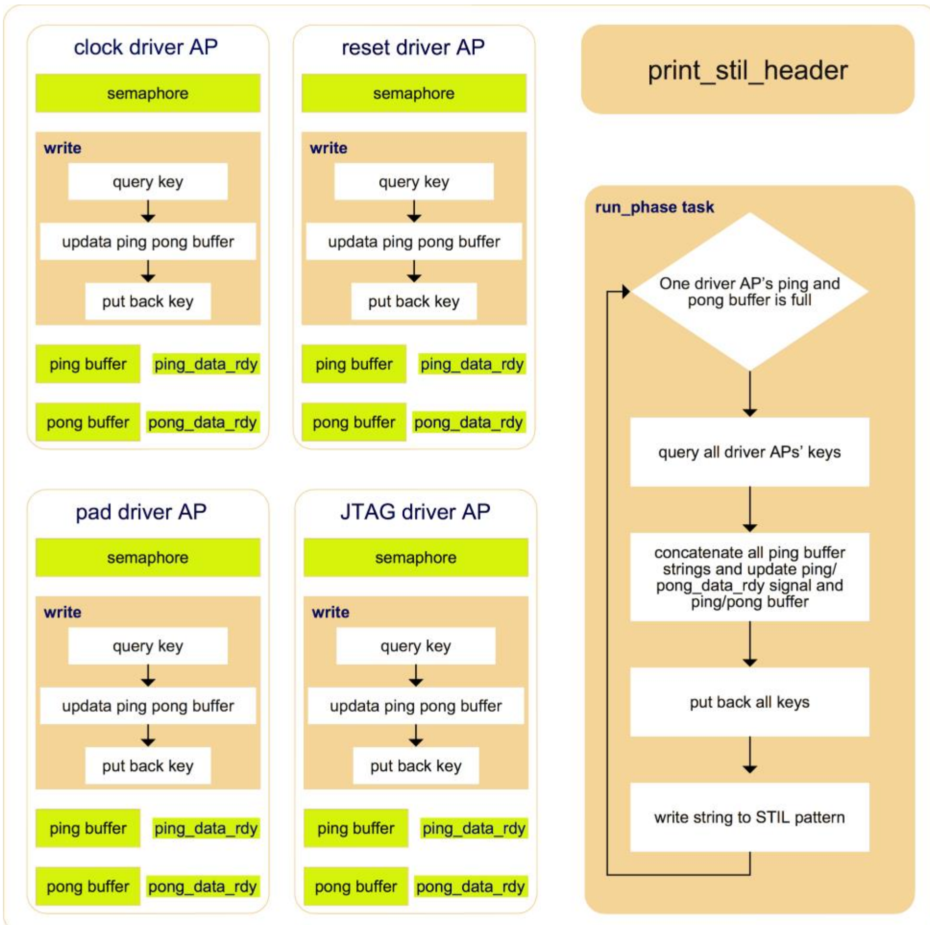


Figure 14. ***STIL_generator*** block diagram.

III. DFT TDR Abstraction

H. Idea Overview

For ultra-large-scale SoC, usually there is a group of TDRs, which are either IEEE 1500 or IEEE 1149.1 compliant, being used to configure a block of the DFT design. The TDR groups among different blocks are chained together using IEEE 1687 protocol. Figure 15 is an example of DFT TDR access network.

It is necessary to level up TDR access in RAL, so as to make it easy to migrate UVM tests developing from this UVM-based DFT verification environment among verification environments and tests from block to system levels. By doing this test writers can focus on test sequences as such, rather than the complex operation of accessing every TDR hierarchically located in the network.

For non-UVM-based environment, the normal way is to define a base class according to its protocol (for example, to define an IEEE1500 TDR base class and an IEEE 1149.1 TDR base class) and wrap up a TDR access operation inside its extension. When DFT access network changes, the wrapped-up access operation in each TDR class has to be updated accordingly. Such work is usually time-consuming. However, the method of modelling DFT TDR in UVM-based environment is rarely seen in literature to the author's knowledge.

This paper presents a neat and easy maintenance way to abstract TDR in UVM-based environment, as shown in Figure 16.

We can encode a TDR's location information into its address as shown in Figure 17 and model an equivalent TDR access network named as ***dft_tdr_network*** in ***dft_tdr_monitor*** as shown in Figure 16.

In Figure 1, the reg2bus direction is shown in red lines, where ***dft_tdr_trans_to_jtag_trans_sequence*** fetches ***dft_tdr_transaction***s, unpacks address, decodes SIB code to get TDR location information, and then generates ***jtag_transaction***s to ***jtag_sequencer*** [3]. For the bus2reg direction shown in blue lines, ***dft_tdr_network*** maintains network status using ***jtag_transactions*** from ***jtag_monitor***. When ***sib_node***s value hit SIB code in ***dft_tdr_block***, ***dft_tdr_monitor*** writes a ***dft_tdr_transaction*** to ***dft_tdr_predictor***.

In this way, the TDR class definition is very neat, and only needs to declare each bit field of it. Figure 19 is an example of a TDR class definition. When TDR access network changes, we only need to update ***dft_tdr_network*** and ***dft_tdr_trans_to_jtag_trans_sequence***, while all TDR class definitions do not need any update that can save a lot of test environment setup time.

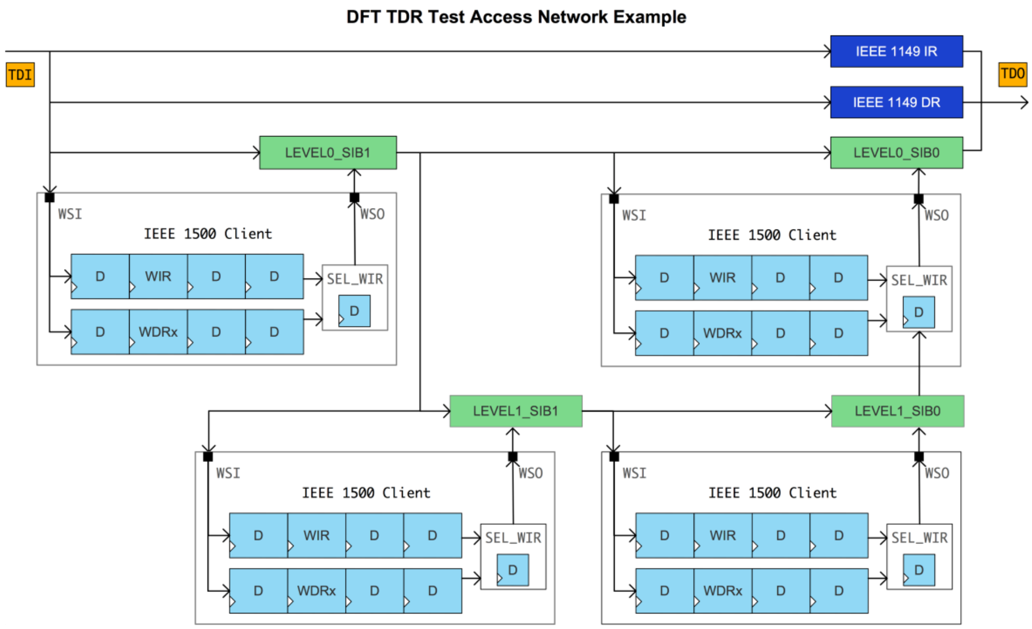


Figure 15. DFT TDR access network example.

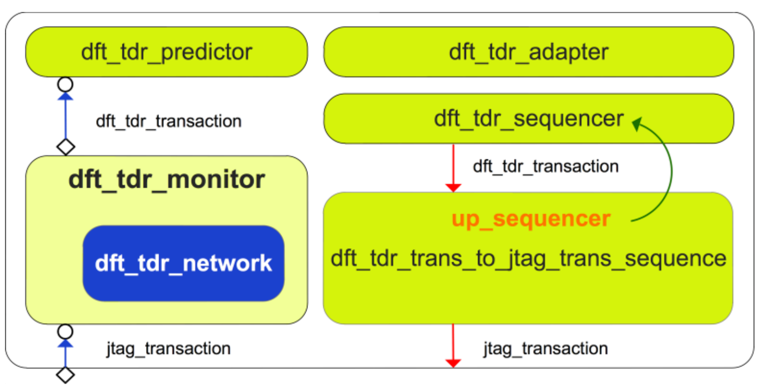


Figure 16. ***dft_tdr_layering*** block diagram.

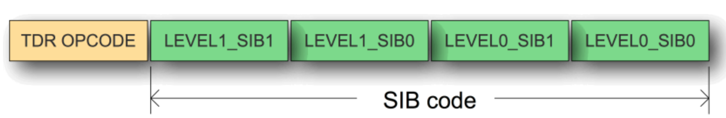


Figure 17. TDR address encode.

I. DFT TDR Access Network Modelling

In DFT TDR access network, a SIB bit and a TDR bit can be modelled as shown in Figure 18.

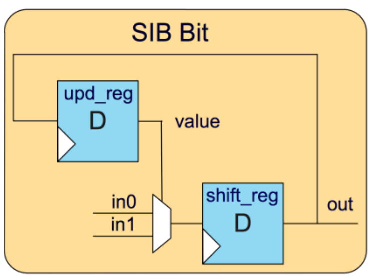
out_update () method is to model the active clock edge triggering the shift register bit during shift operation and ***value_update*** () method is to model the active clock edge triggering the update register bit during update operation.

dft_tdr_network uses ***sib_node*** and ***reg_node*** to construct an equivalent network as 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 need not actually model every TDR, because each time only a TDR can be configured in a 1500 client.

J. DFT TDR Class Definition

DFT TDR class definition is similar to other function registers, which extend from ***uvm_reg*** class. A bypass TDR that has only one bit field is defined in Figure 19 as an example.

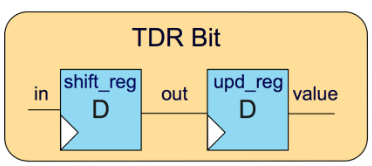



```
class sib_node extends uvm_object;
   `uvm_object_utils(sib_node)
   bit     in0;
   bit     in1;
   bit     value;
   bit     out;

   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 18. TDR access network element modelling.

```
class ieee1149_bypass_reg extends uvm_reg;
   `uvm_object_utils( ieee1149_bypass_reg )
   rand uvm_reg_field bypass;

   function new( string name = "ieee1149_bypass_reg" );
      super.new( .name( name ), .n_bits( `BYPASS_LENGTH ), .has_coverage( UVM_NO_COVERAGE ) );
   endfunction: new

   virtual function void build();
      bypass = uvm_reg_field::type_id::create( "bypass" );
      bypass.configure( .parent                  ( this ),
                        .size                    ( `BYPASS_LENGTH    ),
                        .lsb_pos                 ( 0),
                        .access                  ( "RW" ),
                        .volatile                ( 0    ),
                        .reset                   ( `BYPASS_RST_VALUE    ),
                        .has_reset               ( 1    ),
                        .is_rand                 ( 1    ),
                        .individually_accessible( 0    ) );
   endfunction: build
endclass: ieee1149_bypass_reg
```

Figure 19. DFT TDR definition example.

K. ***dft_tdr_transaction*** *Class and* ***bus_reg_ext*** *Class*

bus_reg_ext class is used for sending golden value to ***jtag_driver*** when doing register read or write in RAL. ***dft_tdr_adpter*** colons the extension information to the handle of extension in ***dft_tdr_transaction*** in bus2reg direction.

Figure 20 and Figure 21 show all properties of ***dft_tdr_transaction*** and ***bus_reg_ext*** class.

read_not_write indicates current register operation is a ***UVM_READ*** or ***UVM_WRITE*** kind.

address is the encoded TDR address.

If current register access kind is ***UVM_WRITE***, ***dft_tdr_adapter*** shift write data to ***wr_data_q***. If current register access kind is ***UVM_READ***, ***dft_tdr_adapter*** shift the default value of the register to ***wr_data_q***.

extension is an object of ***bus_reg_ext*** class. It is used to transfer the side information for TDO pad checking in RAL.

reg_length stores current register's length.

In bus2reg direction, if current register access kind is ***UVM_WRITE***, ***dft_tdr_adapter*** returns data in ***wr_data_q*** else returns data in ***rd_data_q***.

IV. STIL Test Pattern Verification

In order to verify the content and behaviours of the generated STIL file, we can use STIL Verify™ to generate a Verilog testbench and re-run simulation before delivering to ATE test engineers. The STIL file is verified if the simulation passes in STIL Verify™ generated Verilog testbench.

STIL Verify™ is a free verification utility provided by Mentor Graphics for checking the conformity of STIL files, which ensures that STIL files are syntactically correct, and features a Verilog testbench that allows EDA (Electronic Design Automation) and ATE tool developers to run and display STIL content in any Verilog simulator taking STIL file and DUT as input [4].

V. Discussion

In Figure 1, ***pad_agent*** is mostly a physical layer agent that only drives and samples pads directed by ***pad_rw_transaction***s, and has no knowledge about interface protocols, although it groups pads based on their interface protocols. If needed, the user can implement an upper layer agent to convert protocol-related transactions to ***pad_info_transaction***s and pass them down to ***pad_agent***.

For the sake of simplification, this paper focuses on describing how to build a verification environment that can convert UVM tests to test patterns for ATE test during simulation, and the common components such as coverage collectors and scoreboards are not shown, the user can easily implement them using sequence items coming from ***jtag_monitor***, ***dft_tdr_monitor***, and ***pad_monitor***.

Figure 22 is an example of building an upper layer agent that includes a scoreboard and a coverage collector using sequence items from ***pad_monitor***, above the ***pad_agent***. Inside ***scan_agnet***, scan related protocols are implemented in ***scan_trans_to_pad_rw_trans_sequence***, which converts each ***scan_transaction*** to a serial of ***pad_rw_transaction***s. ***scan_monitor*** collects ***pad_rw_transaction***s and converts them into ***scan_transaction***s.

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

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

Figure 20. ***dft_tdr_transaction*** properties definition.

```
class bus_reg_ext extends uvm_object;
  `uvm_object_utils(bus_reg_ext)
  bit       chk_ir_tdo;
  bit       chk_dr_tdo;
  logic     exp_tdo_dr_q[$];
  bit       exp_tdo_dr_mask_q[$];
  logic     exp_tdo_ir_q[$];
  ...
endclass : bus_reg_ext
```

Figure 21. ***bus_reg_ext*** properties definition.

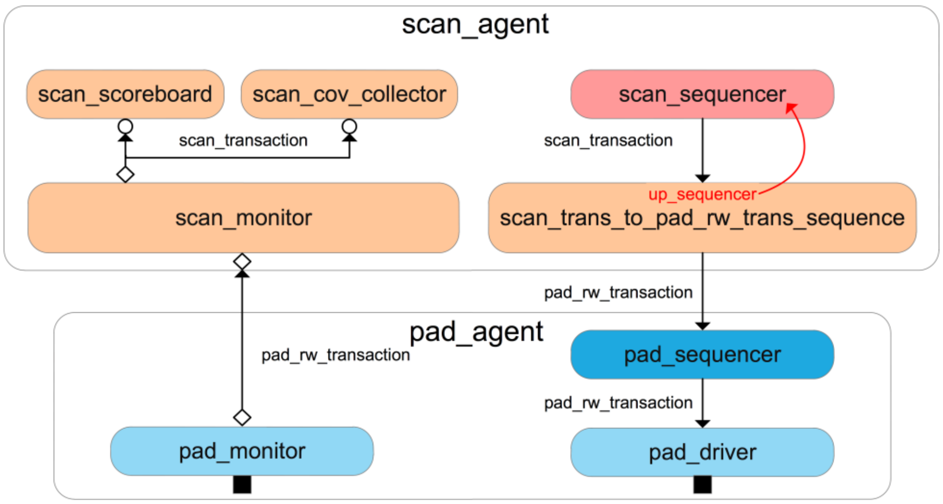


Figure 22. An example of building upper layer agent above ***pad_agent***

Because we enforce every pad drive and sample should be done by a driver except for reference clock pads and each driver passes STIL information to the ***STIL_generator*** whenever it drivers and samples a pad, the generated STIL pattern is functionally equivalent to its corresponding UVM test. Coverage statistics, which is gathered from coverage collectors to rank a UVM test, is also used to rate the generated STIL pattern.

VI. Conclusion

This UVM-based DFT environment can be easily adopted in most projects for DFT function verification by overriding ***dft_env_configuration***, grouping pads as shown in Section B, and defining related interfaces.

By modifying the ***call_stil_gen*** () method in each driver to transfer pad drive and measurement information to the format of the other test language required, this environment could also generate other format patterns ATE needs, not just the STIL format.

Using this method, it saves usually a team's work to translate DFT function tests to STIL patterns in a project, and more important, it avoids errors introduced in the manual translation process to save turnaround debug efforts.

The approach to lift TDR in RAL is also a general way and can be applied in most projects by modelling related ***dft_tdr_network*** and overriding ***dft_tdr_trans_to_jtag_trans_sequence***. Moreover, it makes it easy to migrate UVM tests developing from this UVM-based DFT verification environment among verification environments and tests from block to system levels.

This UVM-based DFT environment works well in an experiment project and the generated STIL test pattern files pass simulation using STIL Verify^{TM}, which indicates it could be applied in real projects.

The next step of work will be to use this environment in real projects and validate it in post-silicon debug.

References

[1] IEEE Standard Test Interface Language (STIL) for Digital Test Vector Data, Section 5. STIL orientation and capabilities tutorial (informative), p. 10.

[2] S. Sutherland, and T. Fitzpatrick, "UVM rapid adoption: a practical subset of UVM," *DVCON 2015*, p. 21.

[3] Mentor Graphics, Online UVM Cookbook (http://verificationacademy.com/cookbook), pp. 302–307.

[4] Mentor Graphics (https://www.mentor.com/products/silicon-yield/getting_started_stil).