### Abstract

# The DFT design is becoming more and more complex accompany with the scale increasing of SoC. How to verify DFT logic completely in simulation and supply test patterns to ATE test with highly coverage is important for post-silicon debug and yield increase.

# While verification methodology is evolving, innovating and entering UVM era, DFT verification need to keep in pace to leverage the advantage of UVM, therefore to increase test reusability, extendibility and function coverage etc.

# This paper presents a general UVM based DFT verification environment, which is usable from modular DFT verification to SoC DFT verification, and it can generate STIL test patterns for ATE test during SoC simulation.

# This paper also presents a method to model hierarchically networked DFT TDR in register abstract level in UVM environment to let test writers focus on test sequences not caring the details of TDR read and write.

### 1.0 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. These non-scan DFT function tests, like other 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.

# For ultra-large-scale SoC (System on Chip), IEEE 1149 protocol alone cannot satisfy DFT design requirements, so 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) have to be set and the length of DR (Data Register) chain varies with SIB values. The author also comes up with a general way to model complex DFT test access network.

### 1.1 Proceed of This Paper

# This paper is intended to divide into four parts. The first part is about how to build a UVM based DFT verification environment that can generate STIL test patterns innately. The second part is focus on how to lift DFT TDR to register abstract level. The third part is how to verify the generated STIL pattern is workable. The fourth part is about result discussion and conclusion.

# In the first and second part, each of them will be proceeded as following sections.

# Idea overview.

# Implementation details.

### 2.0 UVM Based DFT Verification Environment

#### 2.1 Idea Overview

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

# A UVM test usually contains one or several sequences; the UVM sequences are finally broken down into streams of UVM sequence items (a.k.a transactions) and passed to UVM drivers. UVM drivers are in born the best supplier of STIL information.

# If we restrict any pad drive and sample is controlled through a UVM driver, which enforces no direct pad connection in test bench, collect all STIL information from drivers and then write them out according to the time stamp of STIL information, we can get complete test vectors of a certain UVM test after simulation finish.

# So we divide pads of a SoC as following types for DFT functional simulation.

# IEEE 1149.1 compliance on-chip TAP (Test Access Port). Hereinafter it’s simply called JTAG (Joint Test Action Group) interface as show in Table 1, which is the most significant interface for DFT design. Please be noted that in Table 1, read\_not\_write signal is not included in IEEE 1149.1, it’s an internal signal used only in this environment, for more description please refer to section 2.2.5.

# Clock pads, which are clocks need toggle in DFT functional simulation. Please see section 2.3 for more description.

# Reset pads. All the reset related pads are categorized as this type. [Categorize all reset related pads into this type]

# Other pads. Except for type 1-3 mentioned above, the rest pads are categorized as one type. These pads may need have initial value or be toggled once in a while in simulation. Please see section 2.4 for more description.

# In Figure 1, jtag\_driver, pad\_driver, clock\_driver and reset\_driver corresponds to above pad types from 1 to 4. The STIL generator collects STIL information from these drivers and writes them to a STIL pattern file.

|  |  |
| --- | --- |
| JTAG Interface | |
| Pad Direction | Pad Name |
| input | TCK |
| input | TMS |
| input | TRST\_L |
| input | TDI |
| output | TDO |
| input | read\_not\_write |

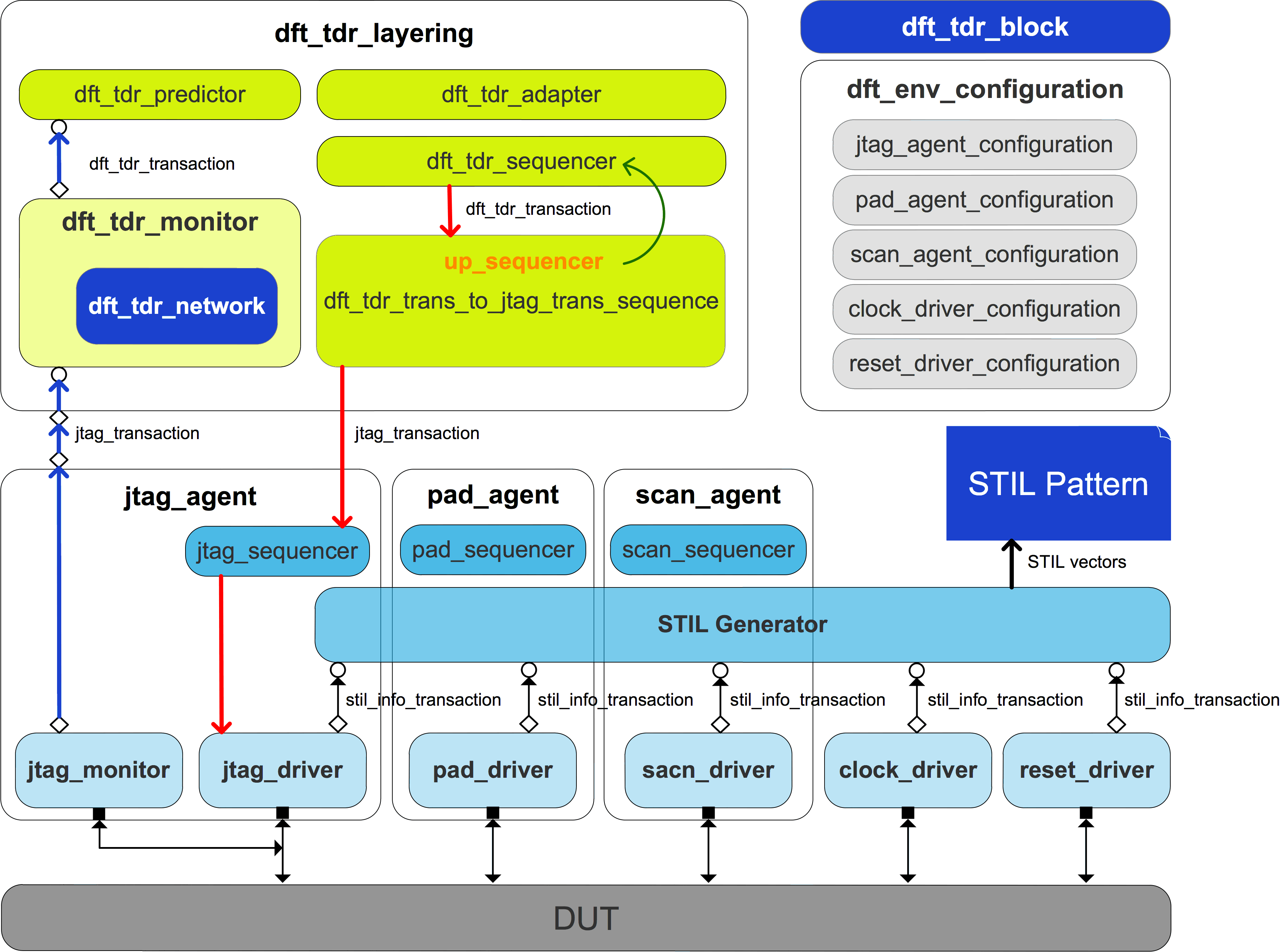
Table 1 – JTAG interface definition

Figure 1 – DFT verification environment

#### 2.2 jtag\_agent Implementation

In Figure 2, jtag\_agent is composed of jtag\_sequencer, jtag\_monitor and jtag\_driver, configured through jtag\_agent\_configuration.

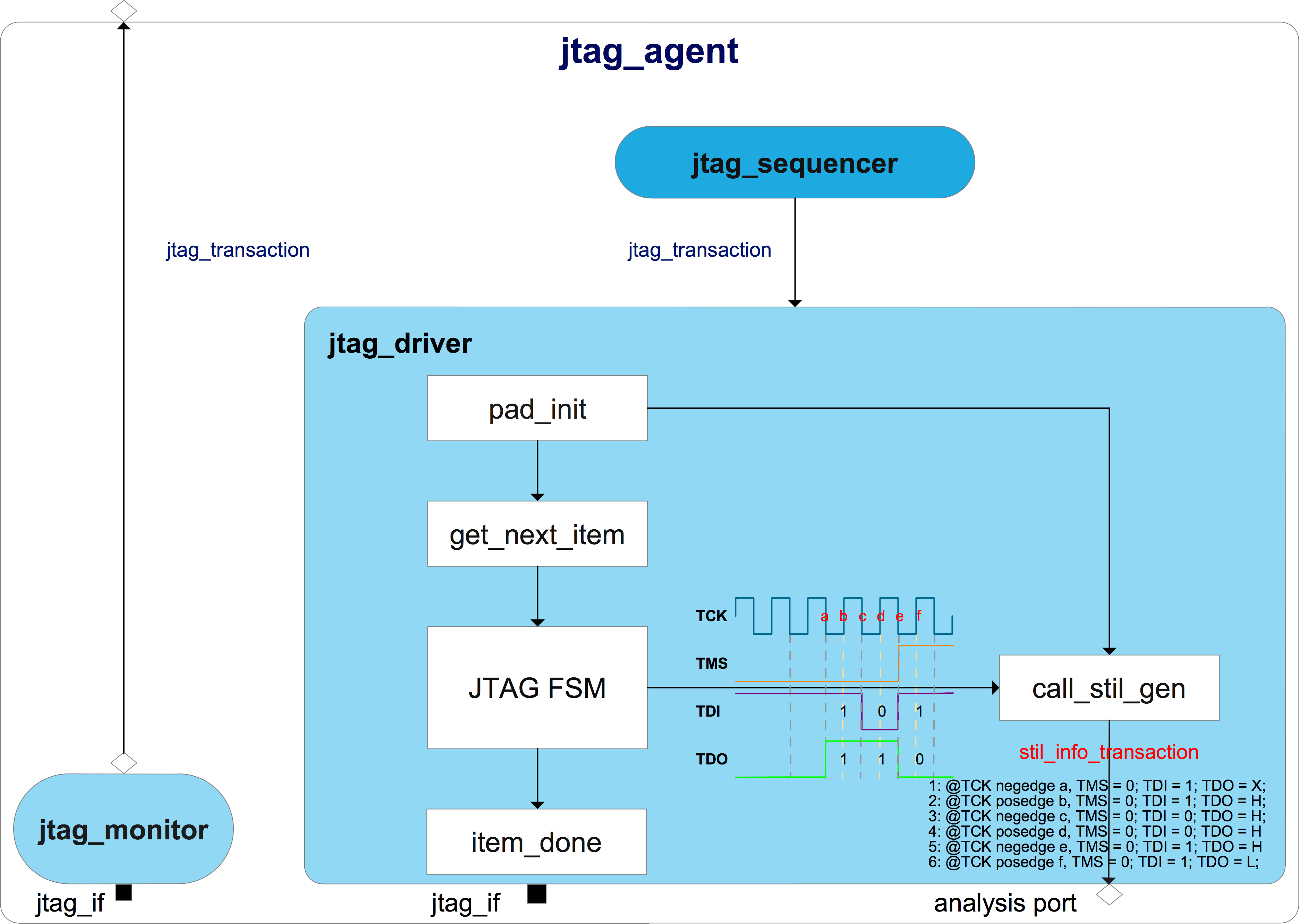


Figure 2 – JTAG agent

##### 2.2.1 jtag\_agent\_configuration

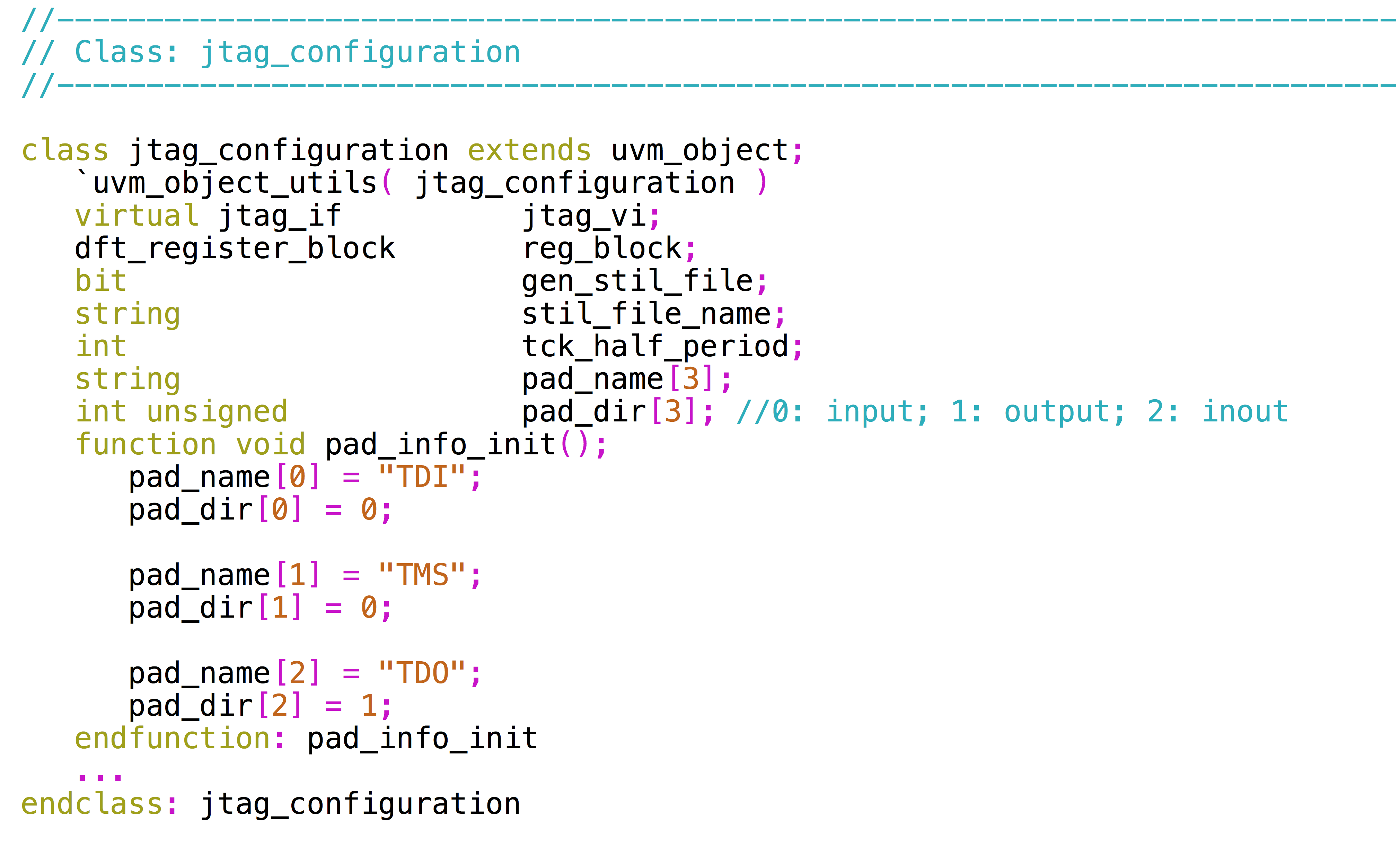


Figure 3 – jtag\_configuration

Figure 3 shows properties and an important function of jtag\_configuration class.

jtag\_vi is virtual JTAG interface.

reg\_block is TDR register model in RAL.

gen\_stil\_file is configuration knob to turn on STIL pattern generation during simulation.

stil\_file\_name is desired STIL pattern file name.

pad\_name array stores JTAG interface package name of a SoC and pad\_dir array indicate pads direction. STIL pattern files need this information to describe stimuli vectors.

DFT test must call pad\_info\_init function before UVM main\_phase objection and store it in configuration database for drivers and STIL generator fetch.

##### 2.2.2 jtag\_transaction class

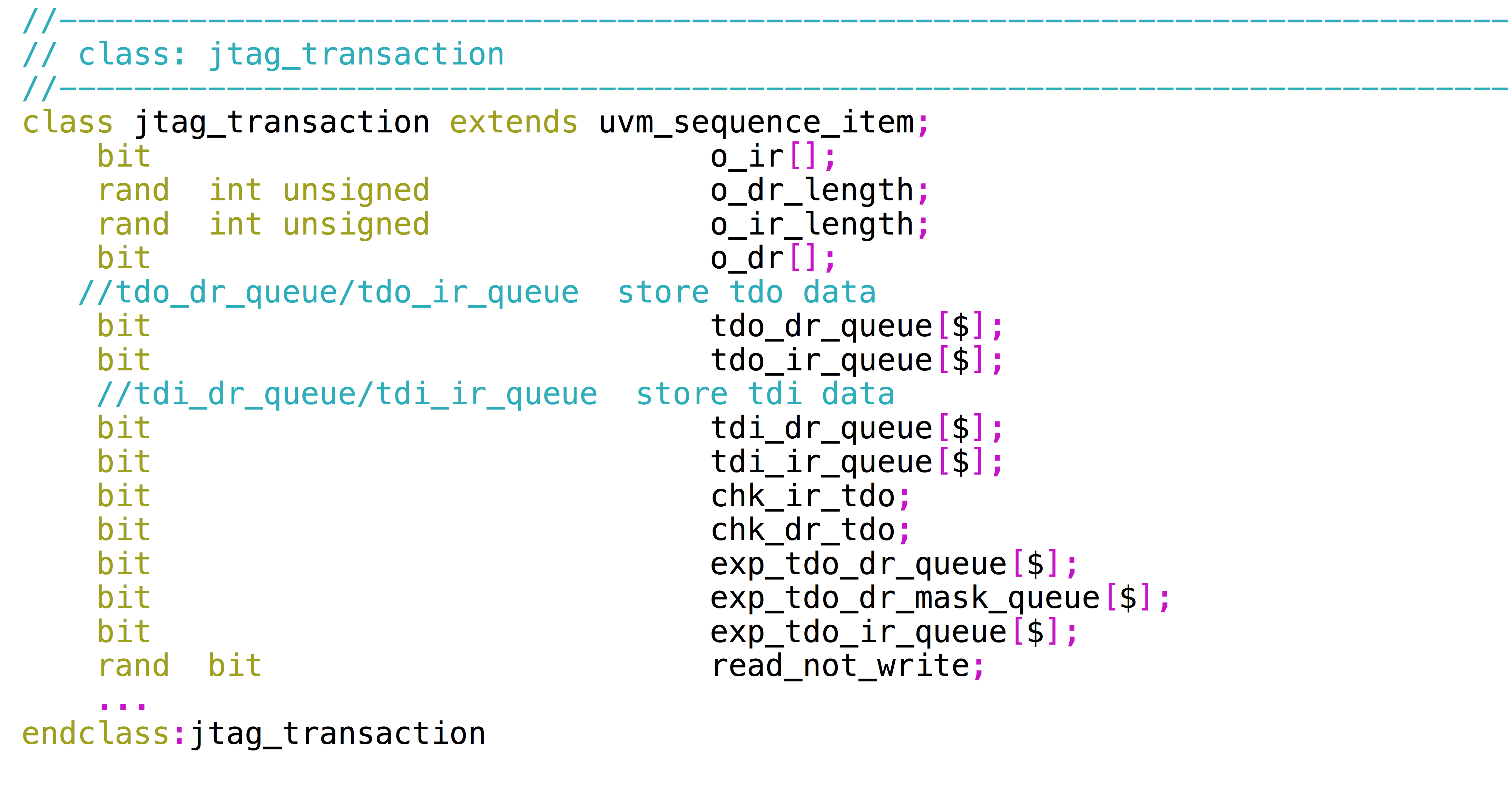


Figure 4 – jtag\_transaction

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 JTAG 1149.1 FSM Instruction Register (IR) and o\_ir\_length is its size.

o\_dr is a dynamic array to store data sending to DUT’s IEEE 1149.1 compliance FSM Data Register (DR) 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 monitoring 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 output during shift DR state, which is used by driver to check TDO data on the fly.

exp\_tdo\_dr\_mask\_queue indicates which bit in exp\_tdo\_dr\_queue is doesn’t care.

exp\_tdo\_ir\_queue is golden data expecting DUT output during shift IRstate, which is used by driver to check TDO data on the fly.

read\_not\_write is flag indicating jtag\_monitor whether it’s a read or write operation for current transaction. Please see Section 2.2.5 for more details.

##### 2.2.3 JTAG interface

This paper categorize pads of a SoC into 4 types which are driven by different drivers, so the JTAG interface showing in Table 1 is driven by clock driver, reset driver and JTAG driver as show in Figure 5.

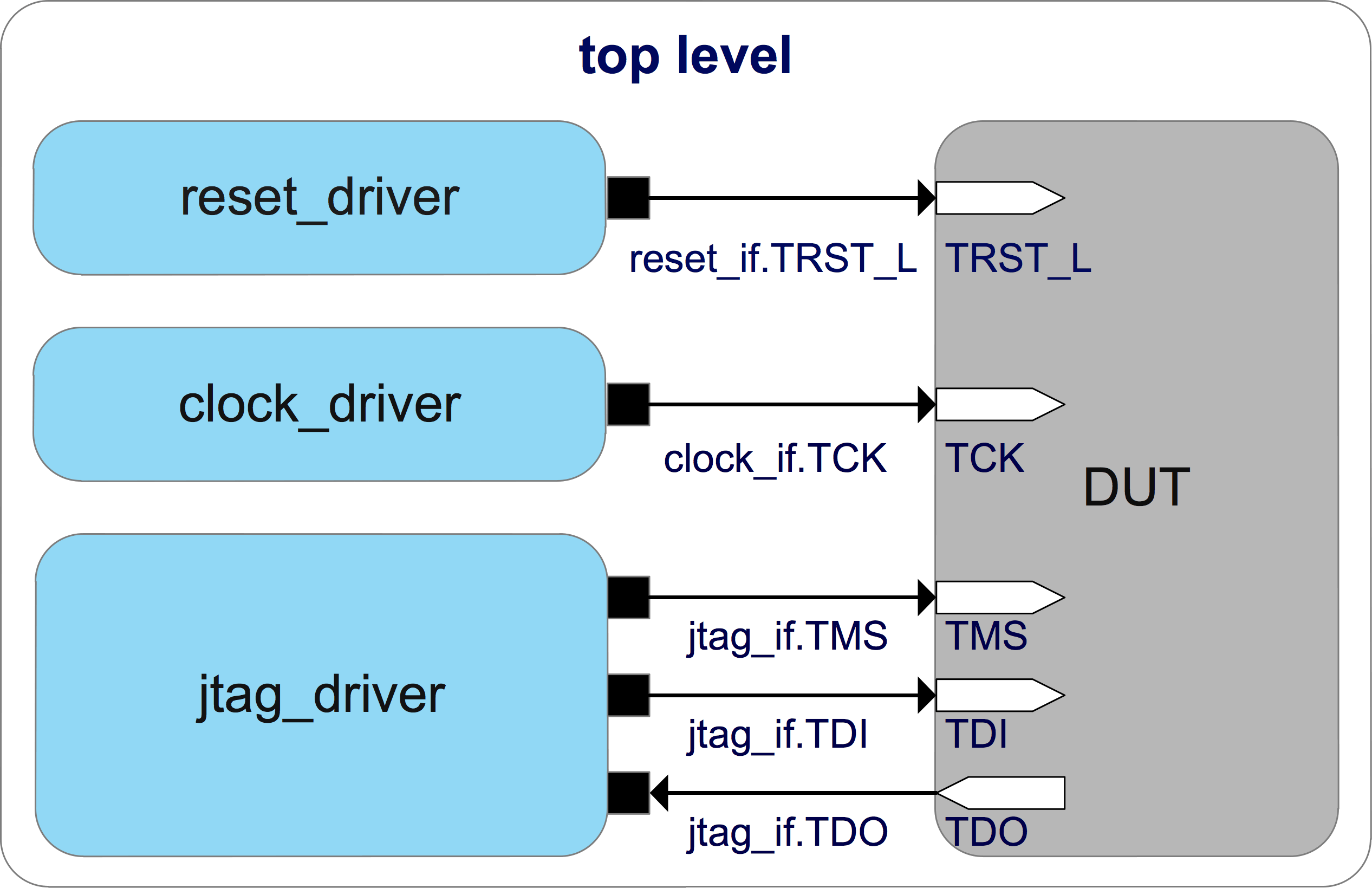


Figure 5 – JTAG interface toplevel connection

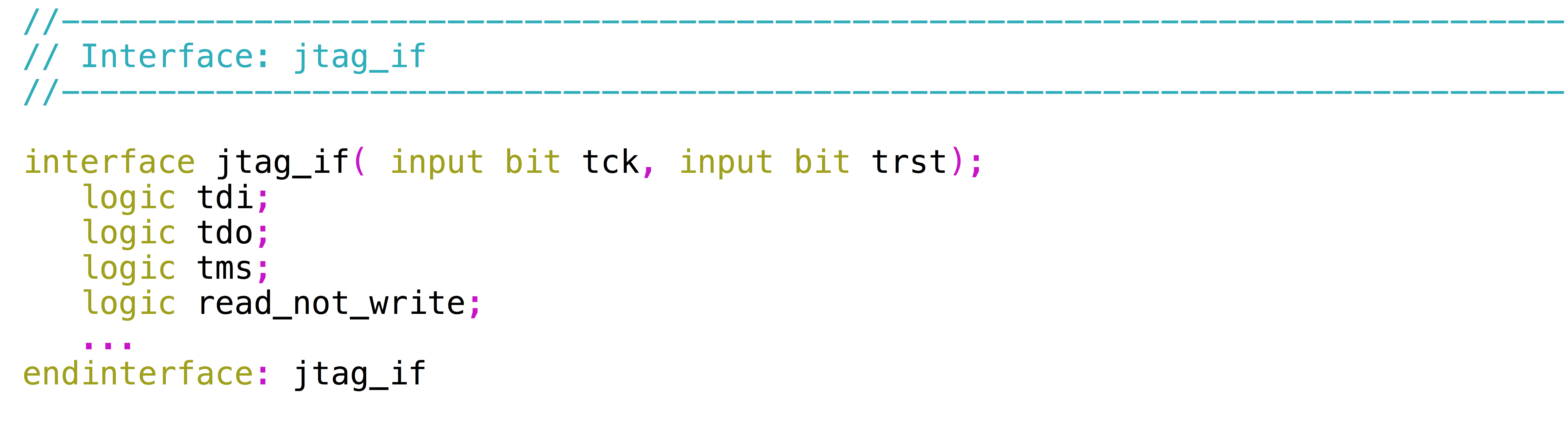


Figure 6 - jtag\_if

Figure 6 is jtag\_if definition that does not contain all signals showing in Table 1 because of the categorization of pads. Others signals definition are included in clock\_if and reset\_if.

##### 2.2.4 jtag\_driver class

IEEE 1149.1 protocol is implemented in jtag\_driver, which fetches every jtag\_transaction sequence item from jtag\_sequencer, toggles 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 a STIL pattern.

If the gen\_stil\_file knob is on, jtag\_driver not only need to toggle and sample pads but also need convert the toggle and sample information to STIL information(handle by call\_stil\_ge function), and then send to STIL generator through an analysis port, which is an object of uvm\_analysis\_port class specialized with stil\_info\_transaction type.

In Figure 2, let us suppose JTAG driver’s FSM is in shift DR state and is going to shift 3 bits 101 to DUT and sample TDO data during shift operation. The golden TDO data is 3 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 function converts this information as show in line 1.

At TCK positive edge b, JTAG driver samples TDO and compares it with golden value, which is 1 bit 1. call\_stil\_gen function converts this information as show 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 function converts this information as show in line 3.

At TCK positive edge d, JTAG driver samples TDO and compare it with golden value which is 1 bit 1. call\_stil\_gen function converts this information as show 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 function converts this information as show in line 5.

At TCK positive edge f, JTAG driver samples TDO and compare it with golden value which is 1 bit 0. call\_stil\_gen function converts this information as show in line 6.

##### 2.2.5 jtag\_monitor class

There is a signal called read\_not\_write defined in JTAG interface showing in Table 1, which only used by jtag\_monitor to indicate whether 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 define write operation and read operation in concept for register abstract level convenience.

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

Write operation: data being shift 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 show in blue arrows of Figure 1.

#### 2.3 clock\_driver Implementation

Figure 7 is an example of how to toggle clocks in this DFT verification environment.

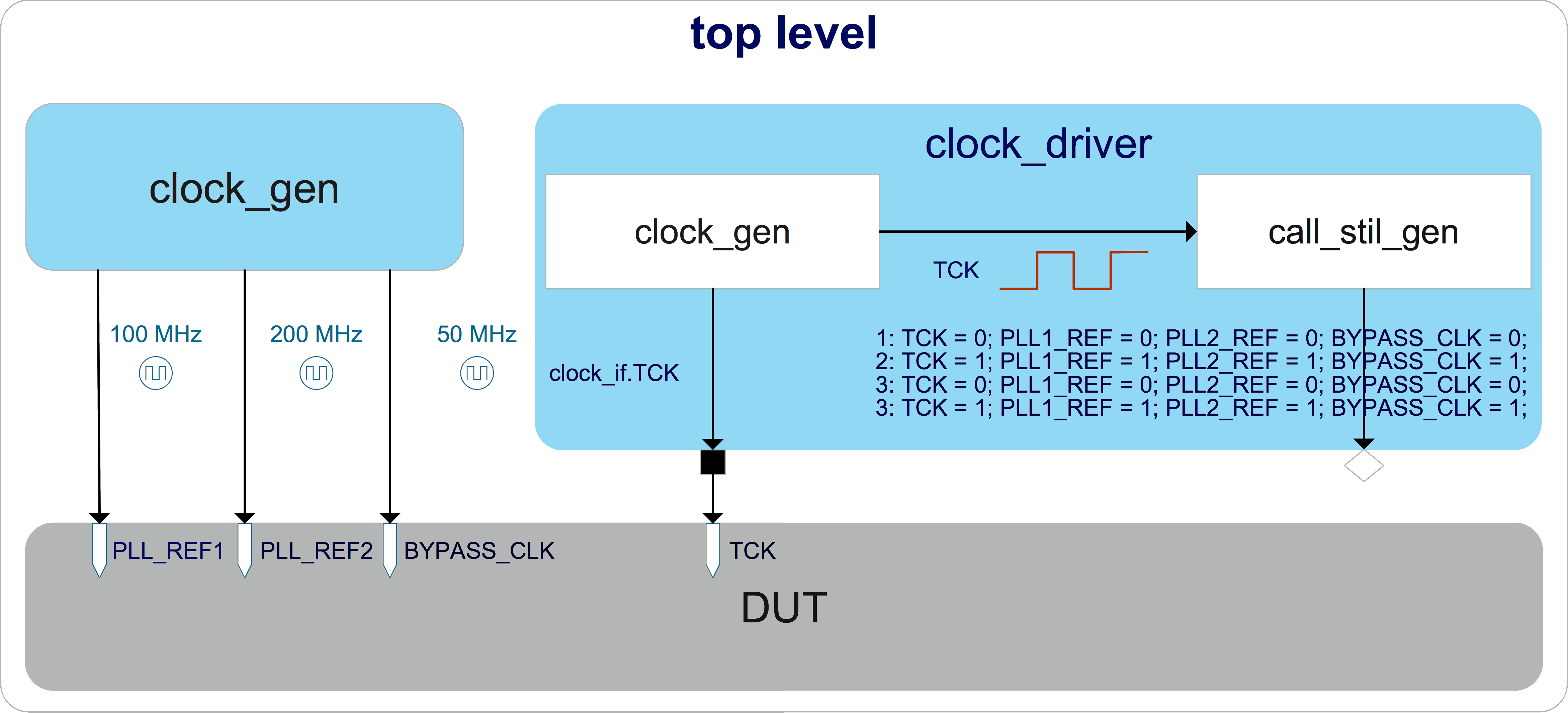


Figure 7 – clock toplevel connection

Suppose a DUT has two PLL reference clocks and a bypass clock need to active during simulation, which are PLL1\_REF, PLL2\_REF and BYPASS\_CLK as show in Figure 7.

clock\_gen in top level 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 need pass TCK toggle information to call\_stil\_gen function at the same time it drives TCK, and call\_stil\_gen need use TCK toggle information as all active clocks’ toggle information and pass STIL information to STIL generator.

Why do this? Does this imply all clocks run on a same frequency as TCK for ATE test?

For ATE test, PLL1\_REF, PLL2\_REF and BYPASS\_CLK toggle information in STIL pattern can be regarded as a placeholder to let post silicon engineers aware these three clocks are reference clocks, so that they will not use the toggle information described in STIL pattern to driver reference clocks, but use clocks supplied by ATE with desired frequency.

#### 2.4 pad\_agent Implementation

The pad type 4 defined in section 2.1, can be sub grouped 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 sub group.

Design pad interface and pad\_rw\_transaction as show in Figure 8 and Figure 9.

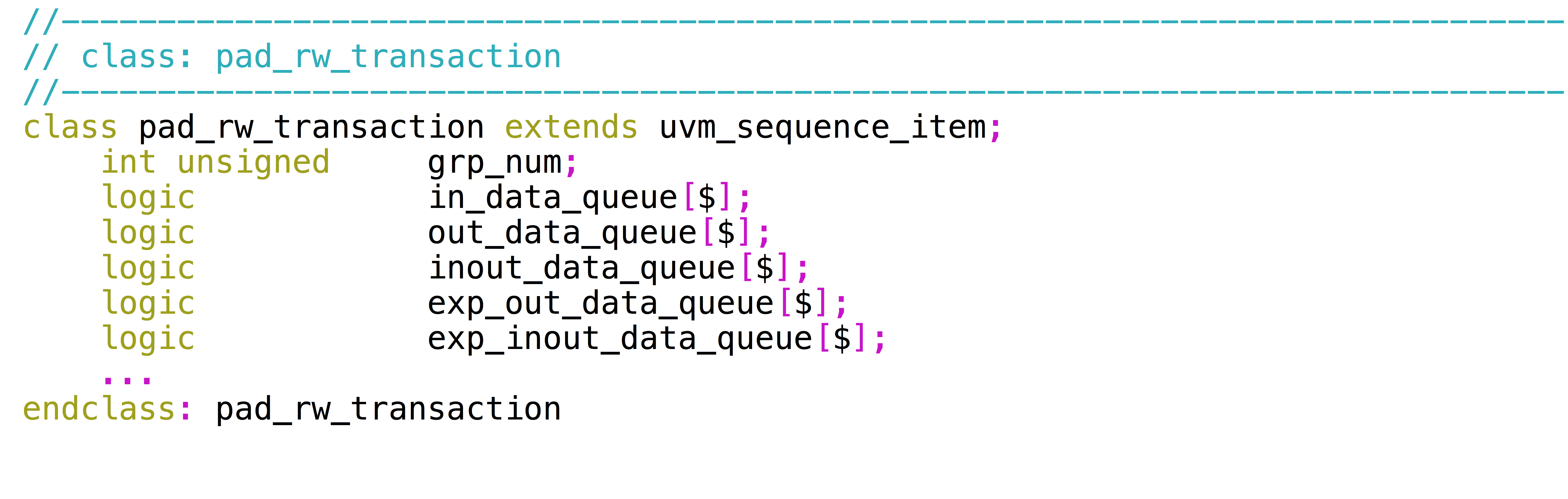


Figure 8 – pad\_rw\_transaction

Figure 8 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 driving by pad\_driver.

out\_data\_queue stores data being sampling by pad\_driver.

inout\_data\_queue stores data being driving or sampling 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 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 pad value during a time period converting by call\_stil\_gen function.

Please be noted these queue types should be logic instead of bit to store four state value.

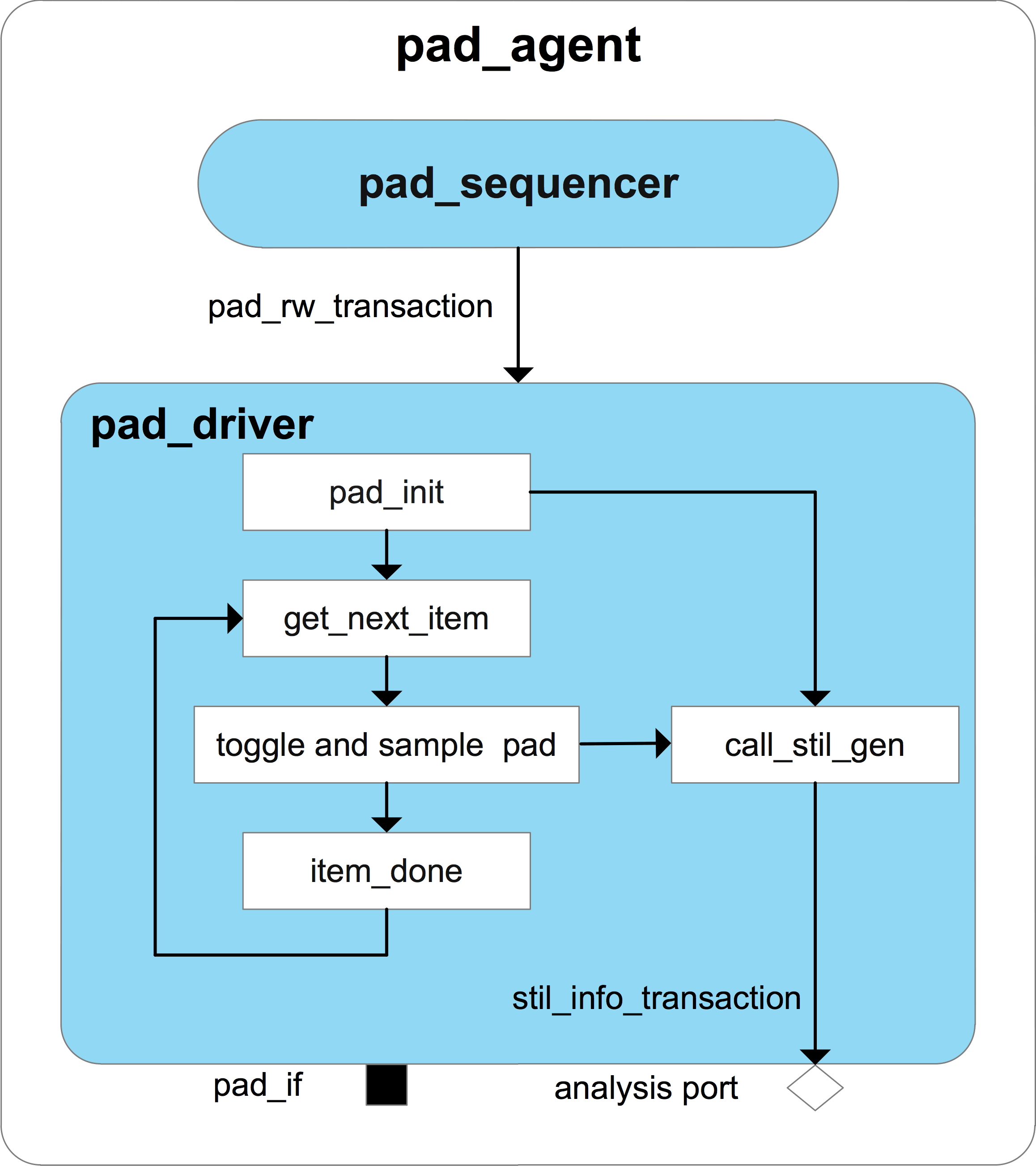


Figure 9 – pad\_agent

In Figure 9, pad\_init function will initialize all groups pads in turn at the beginning of run\_phase task, and call\_stil\_gen function will convert this information to STIL information and write to STIL generation through the analysis port.

##### 2.4.1 pad\_agent\_configuration

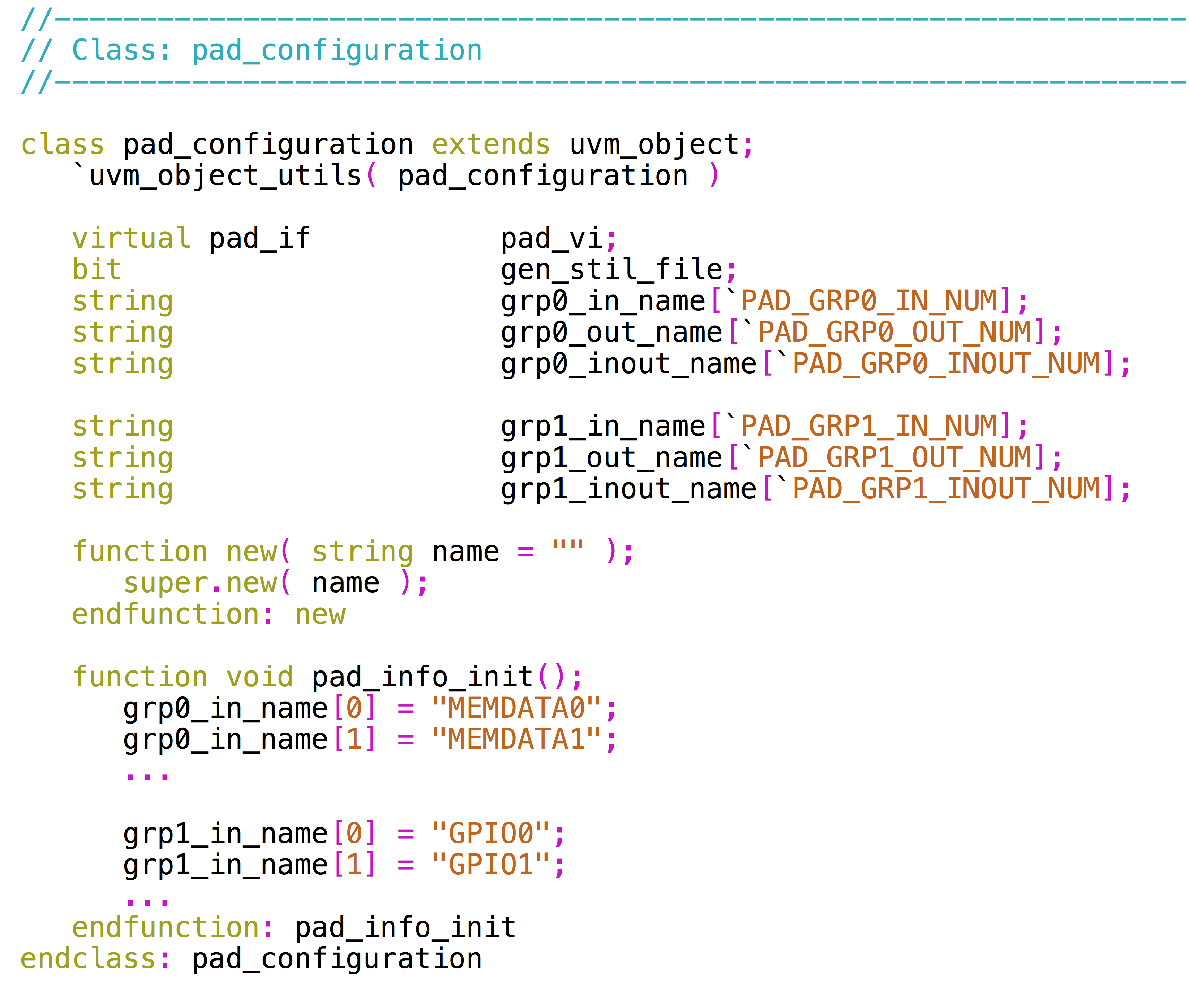


Figure 10 – pad\_configuration

Figure 10 is an example of pad\_configuration class, which has two sub groups of pad.

UVM test need initialize every group’s package name by calling pad\_info\_init function before main\_phase objection and store it in configuration database for pad\_driver and STIL generator fetch.

#### 2.5 reset\_driver class

In Figure 11, reset\_driver drives all resets signals defined in reset interface.

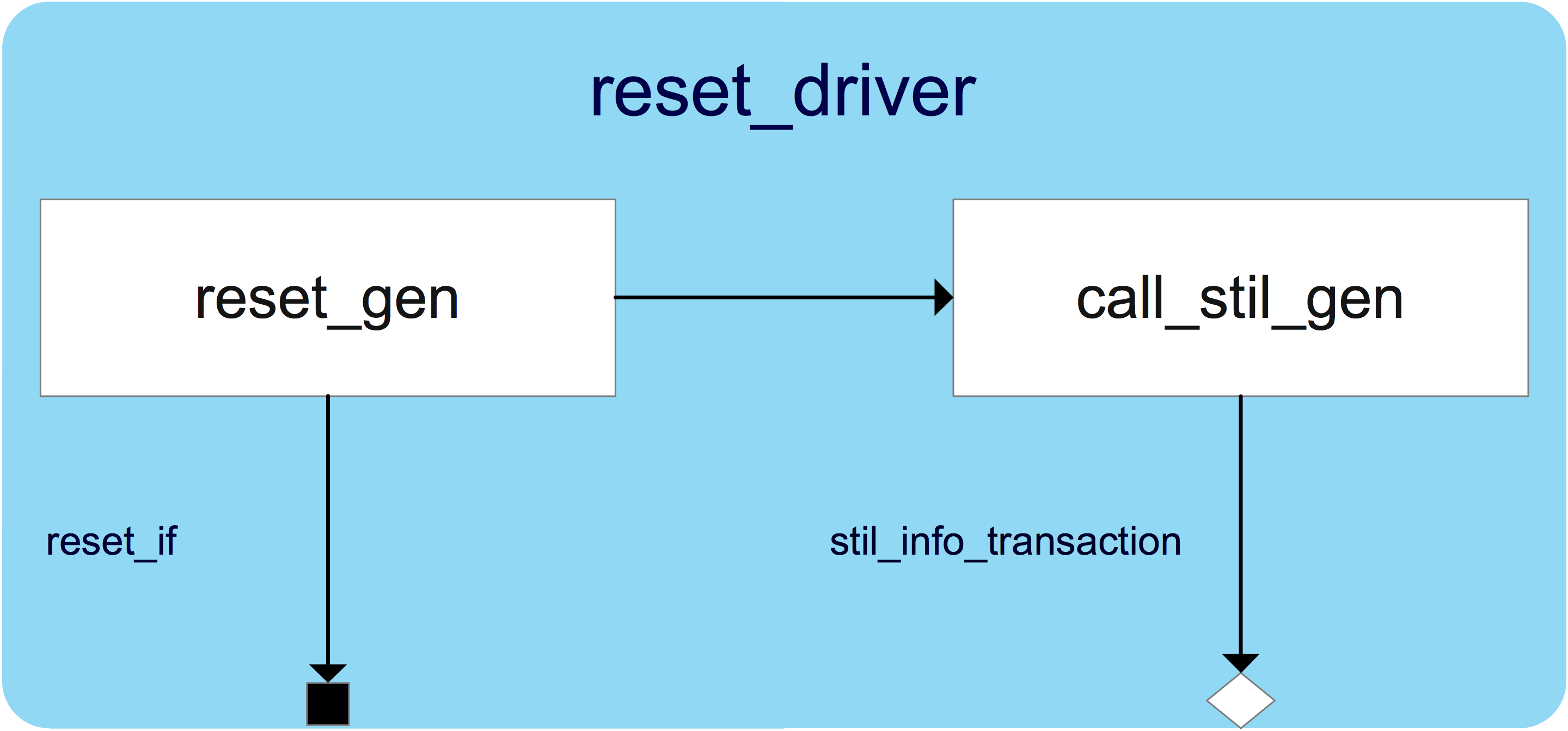


Figure 11 – reset\_driver

##### 2.5.1 reset\_if Class

Figure x is an example of reset\_if definition. BOZO.

#### 2.6 stil\_generator Class

BOZO:STIL file construction

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 uvm\_subscriber class has only one built-in analysis export, the `uvm\_analysis\_imp\_decl macro need to be used to declare analysis imp export and its associated write () method for the rest analysis export [2].

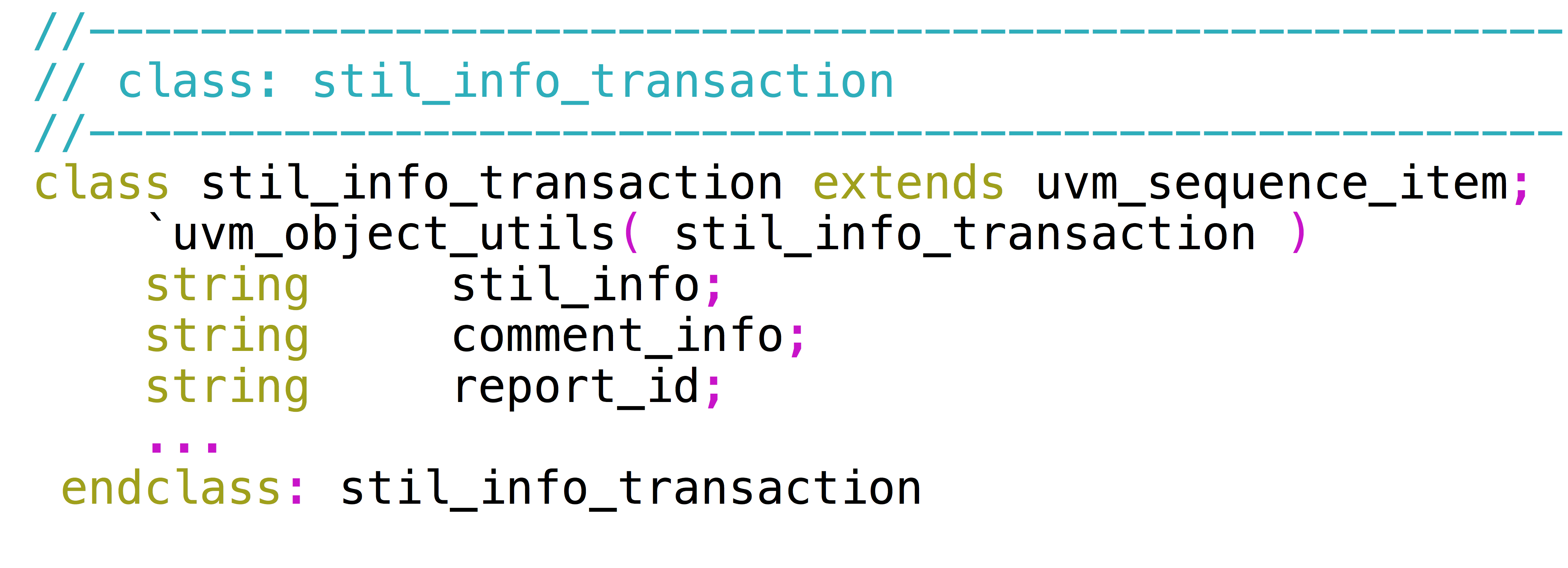


Figure 12 – stil\_info\_transaction

stil\_info\_transaction is defined in Figure 12. stil\_info is pads toggle and measure information and comment\_info is comment going to be printed out with the stil\_info.

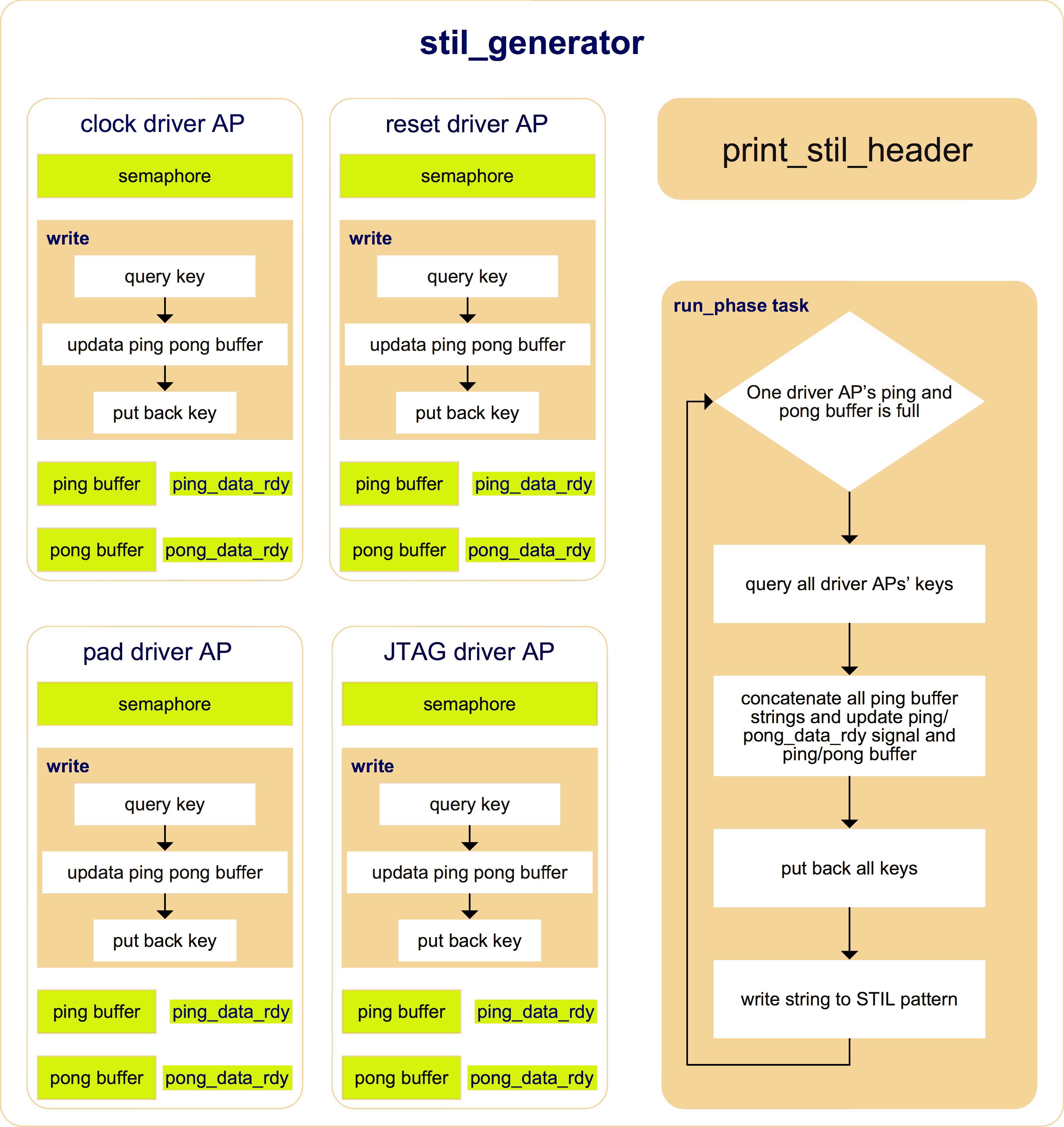


Figure 13 – STIL generator

In Figure 13, each driver’s analysis port has its corresponding write () method, a semaphore which has only one key and a group of ping-pong buffer which has 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 need collect all stil\_info\_transaction coming from a same simulation time slot, concatenate stil\_info of every stil\_info\_transaction and write them out as a single test vector. To make sure stil\_generator does not miss any stil\_info\_transaction from a same time slot, it need to suspend the run task in stil\_generator until all others run task finish. However, in UVM all uvm\_component run tasks are executed in parallel and stil\_generator itself is an uvm\_component, there is not an easy way to schedule the simulation events in stil\_generator run task to execute until all drivers’ run task simulation events finish.

To resolve this, a group of ping-pong buffer is introduced. The write () method always writes ping buffer first and then pong buffer, because ping data and pong data come at different simulation time slot, once a group of ping-pong buffer is full, which indicates the simulation has already moved forward, it’s the right time to collect all ping buffer data and write them out.

The run\_phase task as show in Figure 13 always check if a driver’s ping-pong buffer is full. If it’s true, it will query each key of the semaphore belong to corresponding driver, once get all the keys, fetches all ping buffer data, update ping-pong buffer (if both ping and pong buffer is empty, do noting; if ping buffer is full and pong buffer is empty, clear ping\_data\_rdy; if both ping and pong buffer is full, copy pong buffer data to ping buffer and clear pong\_data\_rdy.) and put back all keys and write a test vector to STIL pattern.

### 3.0 DFT TDR Abstraction

#### 3.1 Idea Overview

For ultra-large-scale SoC, usually there is a group of TDRs, which are either IEEE 1500 or IEEE 1149.1 compliance, being used to configure the DFT design of a tile or a large design block. The TDR groups among different tiles are chained together using IEEE 1687 protocol. Figure 14 is an example of DFT TDR access network.

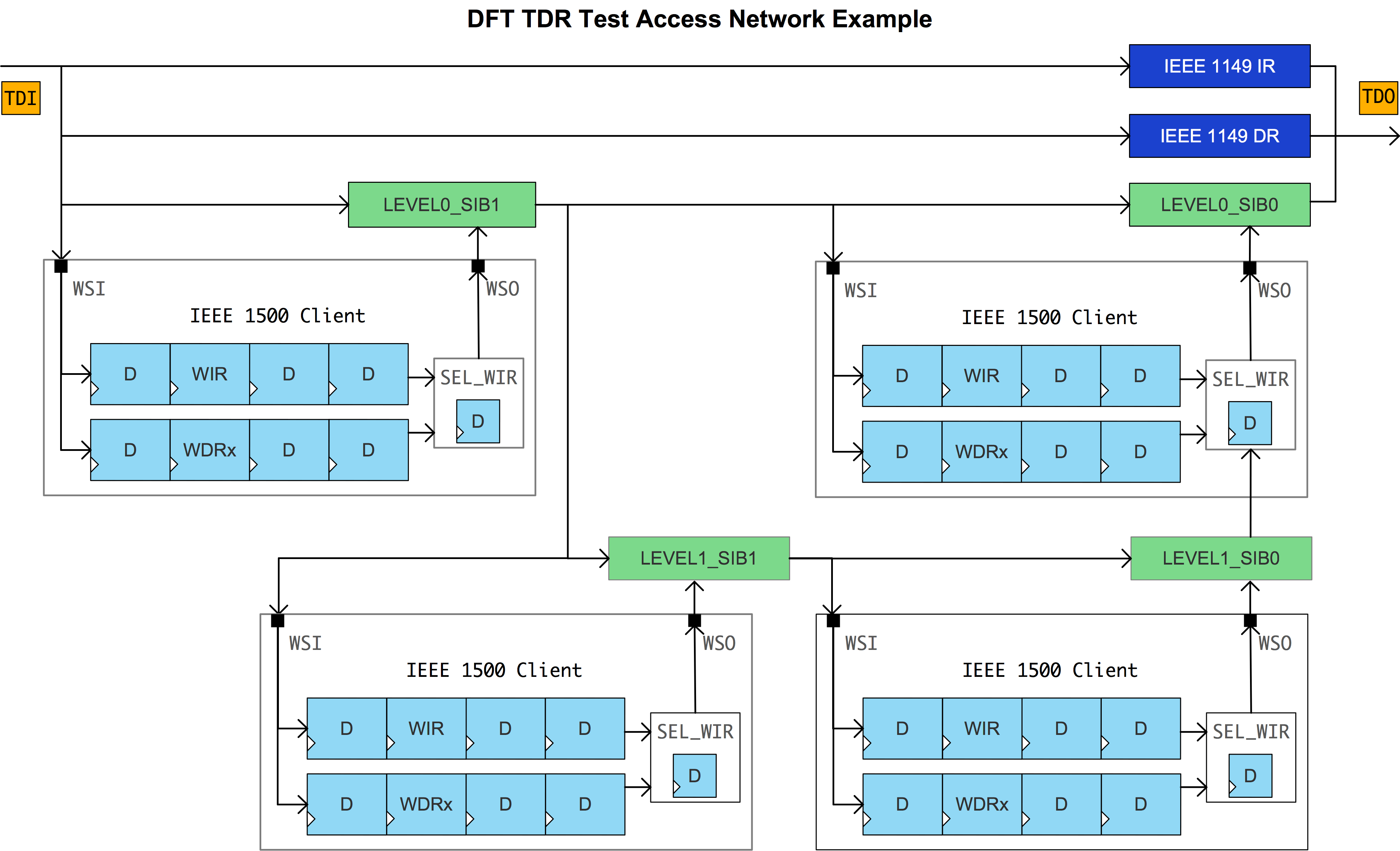


Figure 14 – example network

In order to hide the complex operation of accessing a TDR hierarchically located in the network, it necessary to level up TDR access in RAL.

Method of modelling DFT TDR in UVM based environment is rarely seen.

For none-UVM based environment, the normal way is to define a base class according to its protocol (for example, define an IEEE1500 TDR base class and an IEEE 1149 TDR base class) and wrap a TDR access operation inside its extension. When DFT access network changes, the wrapped access operation has to update accordingly. This is usually a time consuming work.

This paper presents a neat and easy maintain way to abstract TDR in UVM based environment as show in Figure 15.

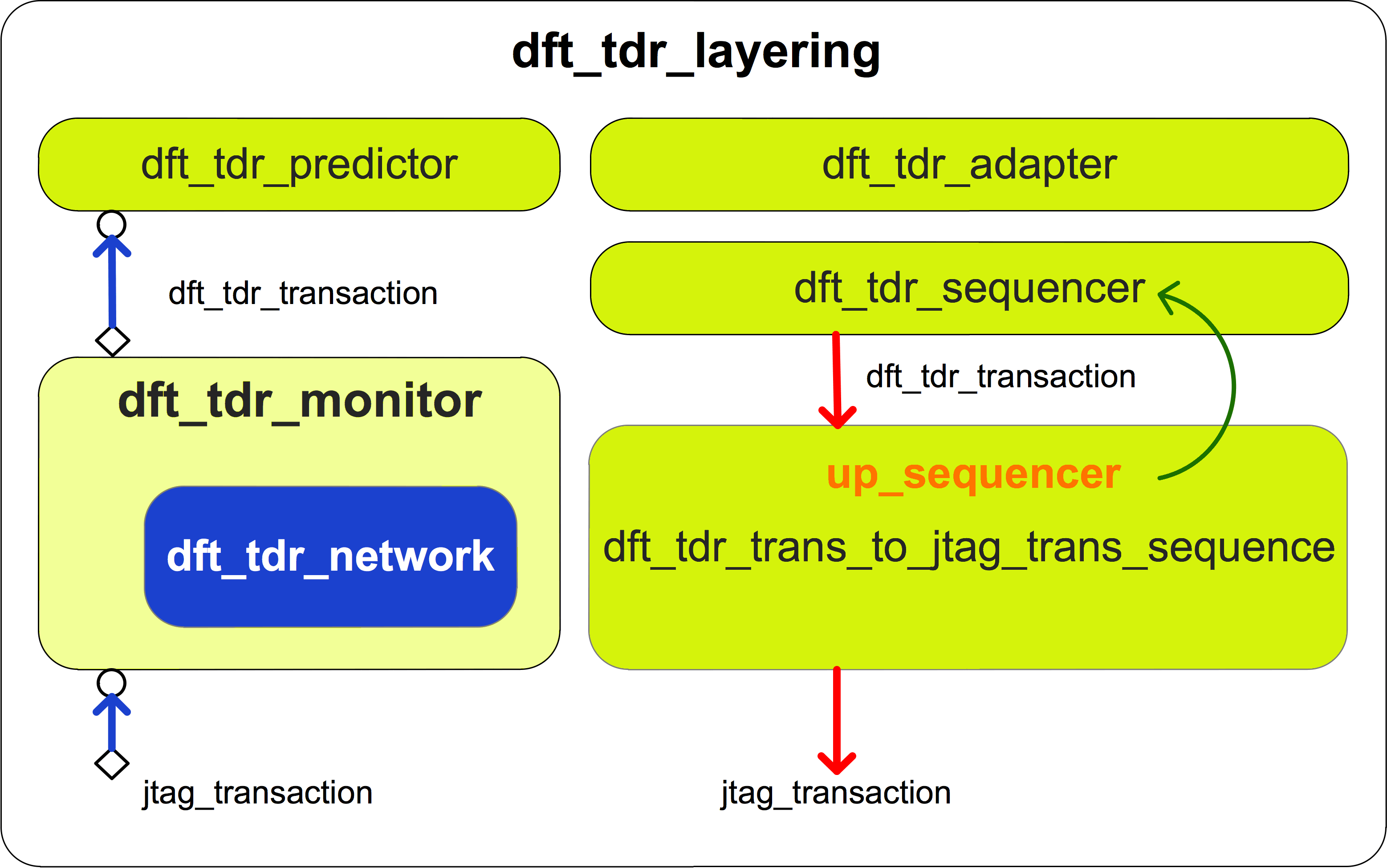


Figure15 – dft\_tdr\_layering

We can encode a TDR’s location information into its address showing in Figure 16 and model an equivalent TDR access network named as dft\_tdr\_network in dft\_tdr\_monitor.

In Figure 1, the reg2bus direction is shown in red lines, where dft\_tdr\_trans\_to\_jtag\_trans\_sequence fetches dft\_tdr\_transactions, unpacks address, decodes SIB code, and then generates jtag\_transactions to jtag\_sequencer [2]. 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 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, only need declare each bit field. Figure 18 is an example of a TDR class. When TDR access network changes, only need to update dft\_tdr\_network and dft\_tdr\_trans\_to\_jtag\_trans\_sequence, all TDR class definition don’t need any update which can save a lot of test environment setup time.

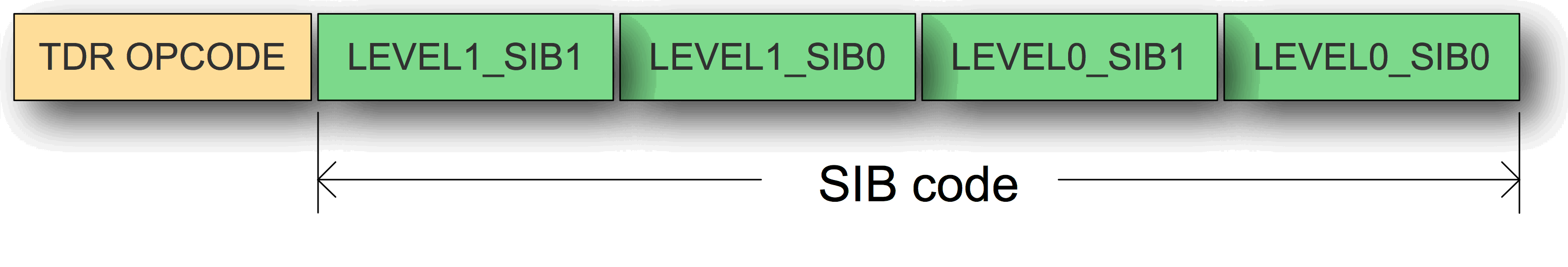


Figure16 – address encode

#### 3.2 DFT TDR Access Network Modelling

In DFT TDR access network, a SIB bit and a TDR bit can be modelled as Figure 17.

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 using sib\_node and reg\_node to construct an equivalent network as DUT.

In Figure 14, only need to model each 1500 client’s IR and a WDR which length is dynamic, which can calculate from jtag\_transaction coming from jtag\_monitor and current network chain length. It need not to actually model every TDR, because each time only a TDR can be configured in a 1500 client.

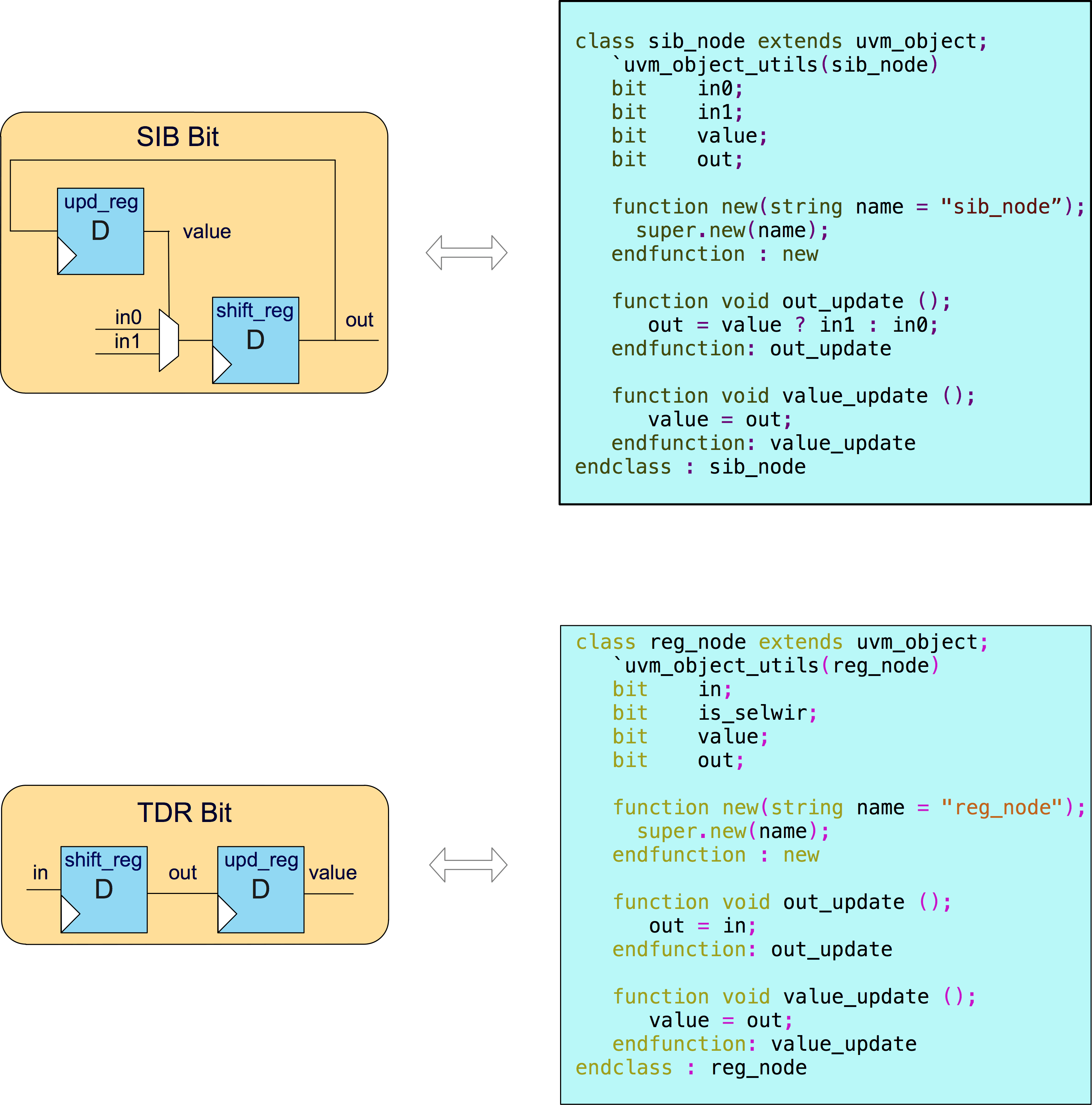


Figure17 – network element model

#### 3.3 DFT TDR Class Definition

DFT TDR class definition is similar as other function registers, which extend from uvm\_reg class. A bypass TDR which has only one-bit field is defined as Figure 18.

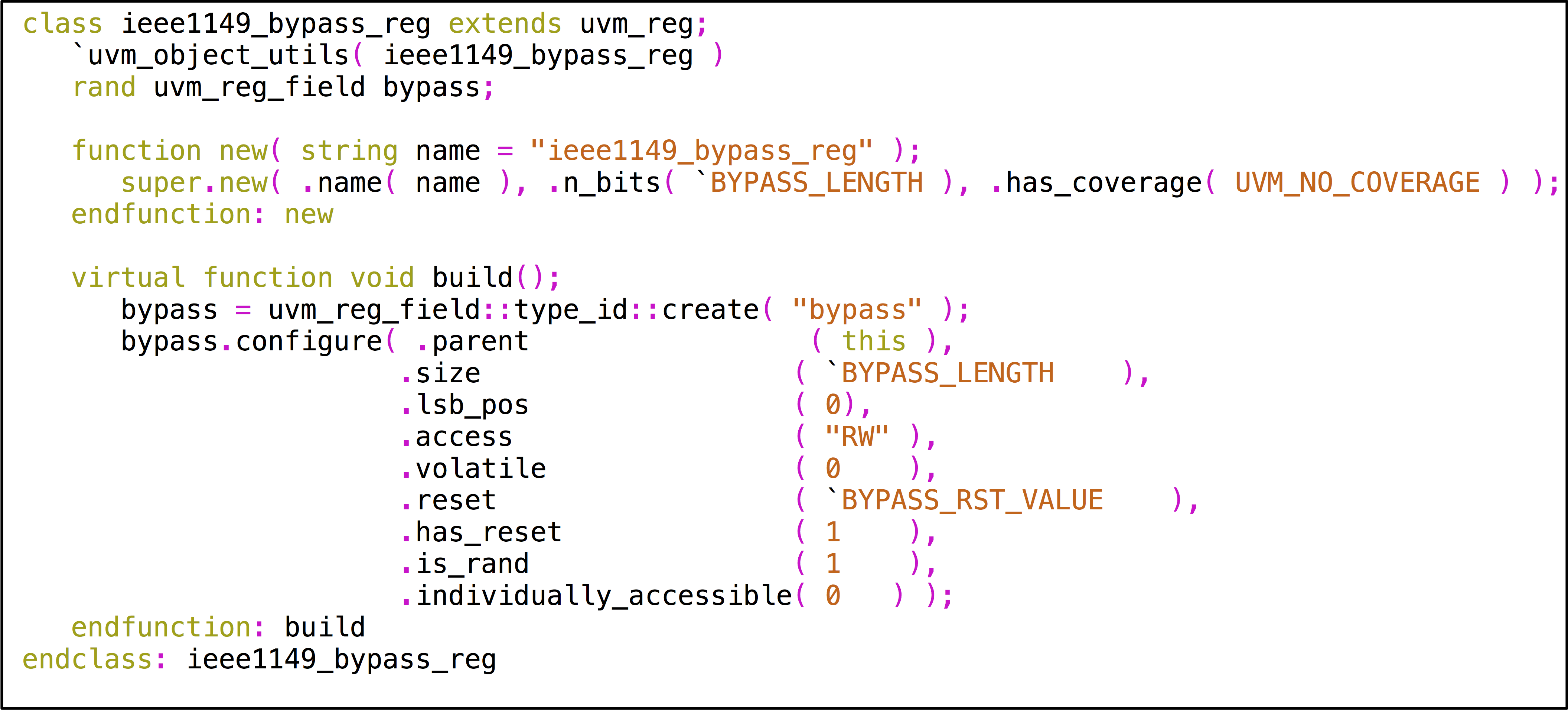


Figure18 – example TDR definition

#### 3.4 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.

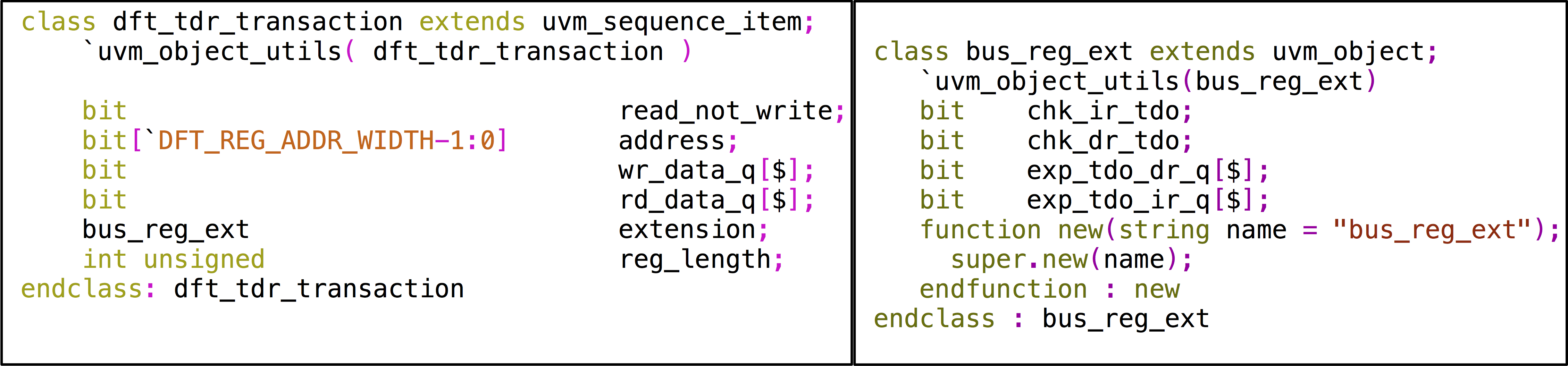


Figure19 – dft\_tdr\_transaction

Figure 19 shows all properties of dft\_tdr\_transaction class.

read\_not\_write indicates current register operation is a UVM\_READ or UVM\_WRITE type.

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 of the register to wr\_data\_q.

extension is an object of bus\_reg\_ext class, which 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 rd\_data\_q.

### 4.0 STIL Test Pattern Verification

In order to verify the content and behavior of generated STIL file, we can use STIL VerifyTM to generate a Verilog testbench and re-run simulation before deliver to ATE test engineers.

STIL VerifyTM 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 Electronic Design Automation (EDA) and ATE tool developers to run and display STIL content in any Verilog simulator taking STIL file and DUT as input [3].

### 5.0 Discussion

In Figure1, pad\_agent is mostly a physical layer agent though it group pads based on their protocols. If needed, users can implement an upper layer to convert protocol related transactions to pad\_info\_transactions and pass down to pad\_agent.

This paper focus on describing how to build a verification environment, which can convert UVM tests to test patterns for ATE test during simulation, so coverage collectors and scoreboards and are not show, users can easily implement them using transactions coming from jtag\_monitor, dft\_tdr\_monitor and pad\_monitor.

Since the generated STIL pattern is equivalent to its corresponding UVM test, a high quality test having high coverage decide a high quality test pattern.

### 6.0 Conclusion

This DFT environment can be easily adopted in most projects for DFT function verification by overriding dft\_env\_configuration, grouping pads as show in section x and defining related interfaces.

By modifying call\_stil\_gen function in each driver to transfer pad toggle and measurement information to the format of other test language required, this environment could also generate other format patterns ATE need.

Using this method, it saves several man months for translating DFT function test to STIL patterns in a project and avoid errors introduced in this process.

The way to lift TDR in RAL is also a general way and can be apply in most projects by modelling related dft\_tdr\_network and overriding dft\_tdr\_trans\_to\_jtag\_trans\_sequence.

[This abstraction mechanism allows the migration of verification environments and tests from block to system levels without any modifications.]

This UVM based DFT environment works correctly in an experiment project and the generated STIL test pattern files pass simulation using STIL VerifyTM, which indicated it could be apply in real projects.

The next step is to put this method in real projects and validate it in post-silicon debug.

**7.0 References:**   
[1] UVM Cookbook, Section Sequences/Layering, 302–307, Cited on 8 September 2015, <http://verificationacademy.com/cookbook>.

[2]

[3]https://www.mentor.com/products/silicon-yield/getting\_started\_stil