### 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 and 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 to let test writers focus on test sequences not caring the details of TDR read and write.

### 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.

# …point issues need to resolve…

# … review previous work…

# … Important to resolve these issues…

# … What this paper did …

### Proceed of This Paper

# This paper is intended to divide into two 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.

### UVM Based DFT Verification Environment

#### 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 finally break down into streams of UVM sequence items (a.k.a transactions) and pass to UVM drivers. UVM drivers are in born the best supplier of STIL information.

# If we restrict any pad drive and sample should be 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 a 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 a internal signal used only in this environment, for more description please refer to section x.

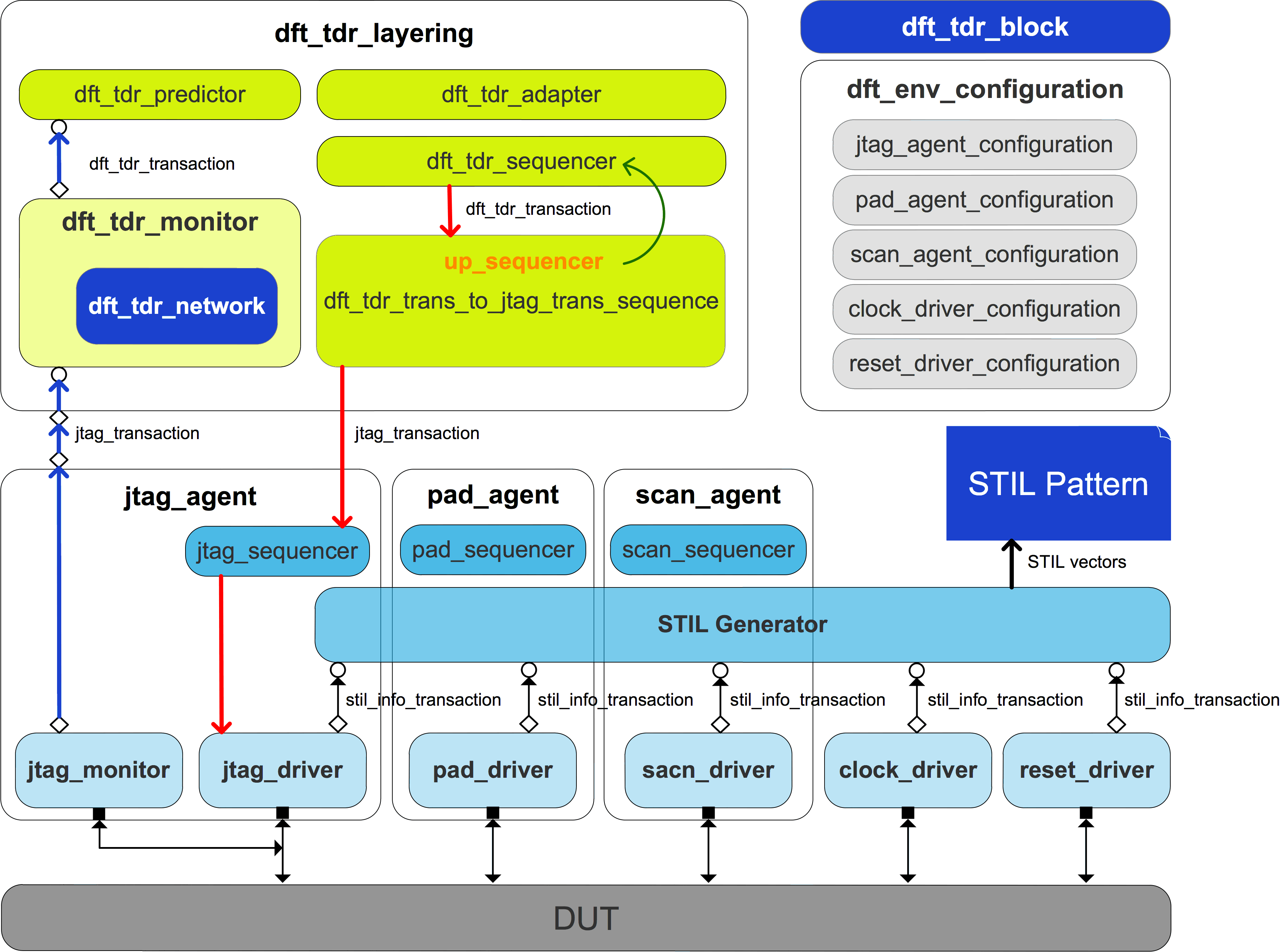
# Clock pads. It’s the clocks need toggling in DFT functional simulation. Please see BOZO for more description.

# Reset pads. All the reset related pads are categorized as 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 BOZO for more description.

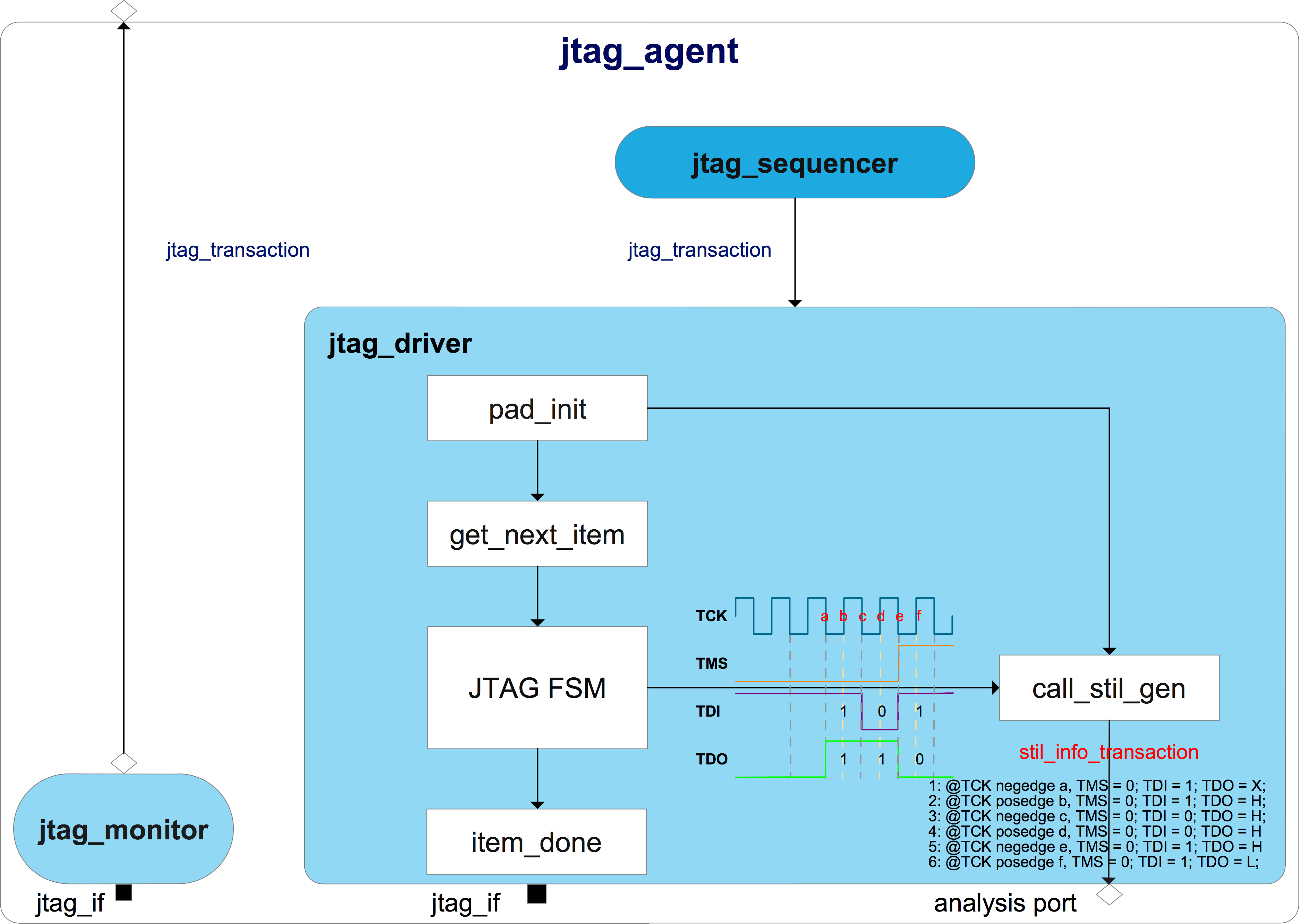
# In Figure 1, jtag\_driver, pad\_driver, clock\_driver and reset\_driver is corresponding to above pad types from 1 to 4. The STIL generator collects STIL information from these drivers and writes them to a STIL pattern file.

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

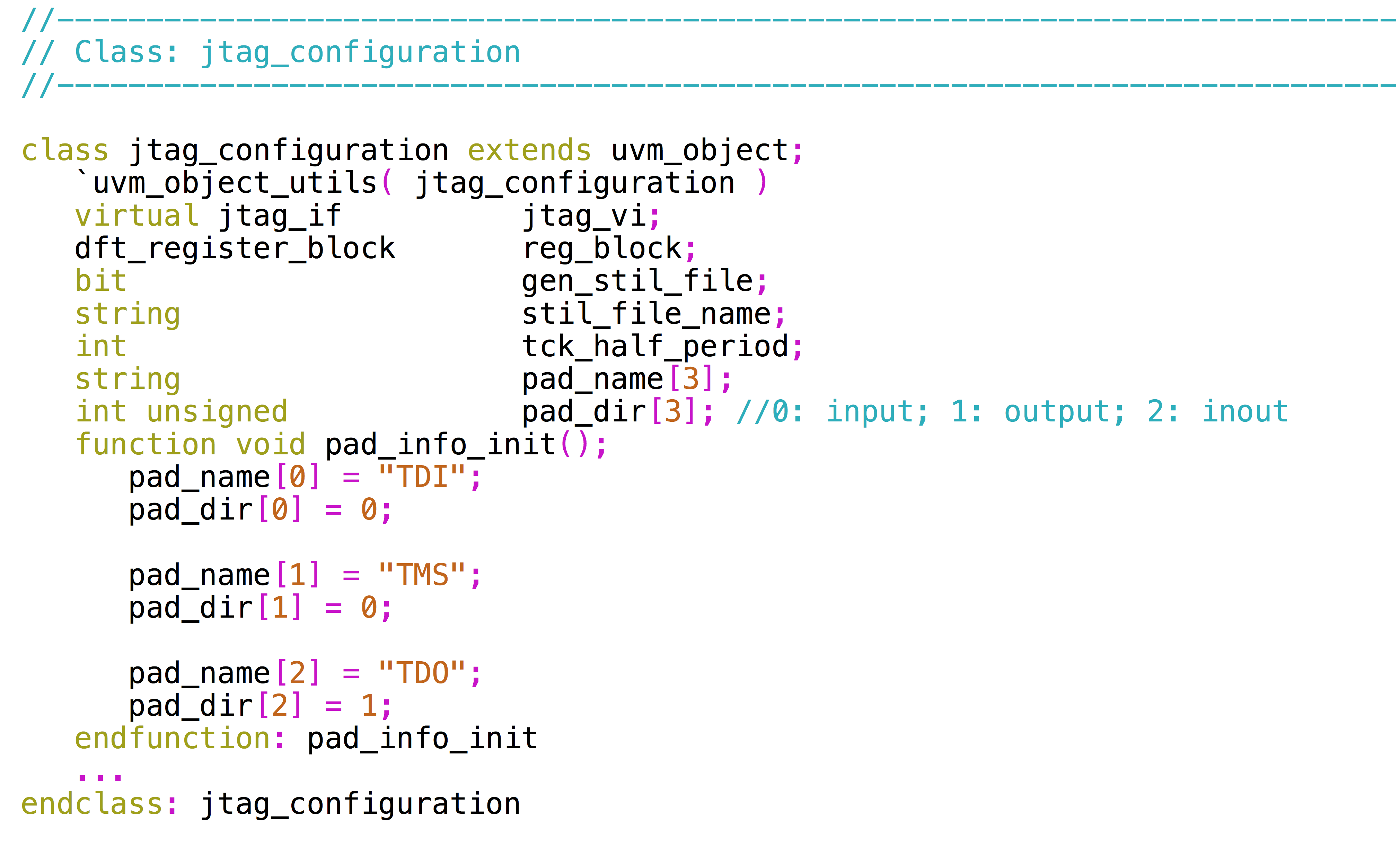


#### jtag\_agent Implementation

jtag\_agent is composed by jtag\_sequencer, jtag\_monitor and jtag\_driver, which configuration can be get from jtag\_agent\_configuration.

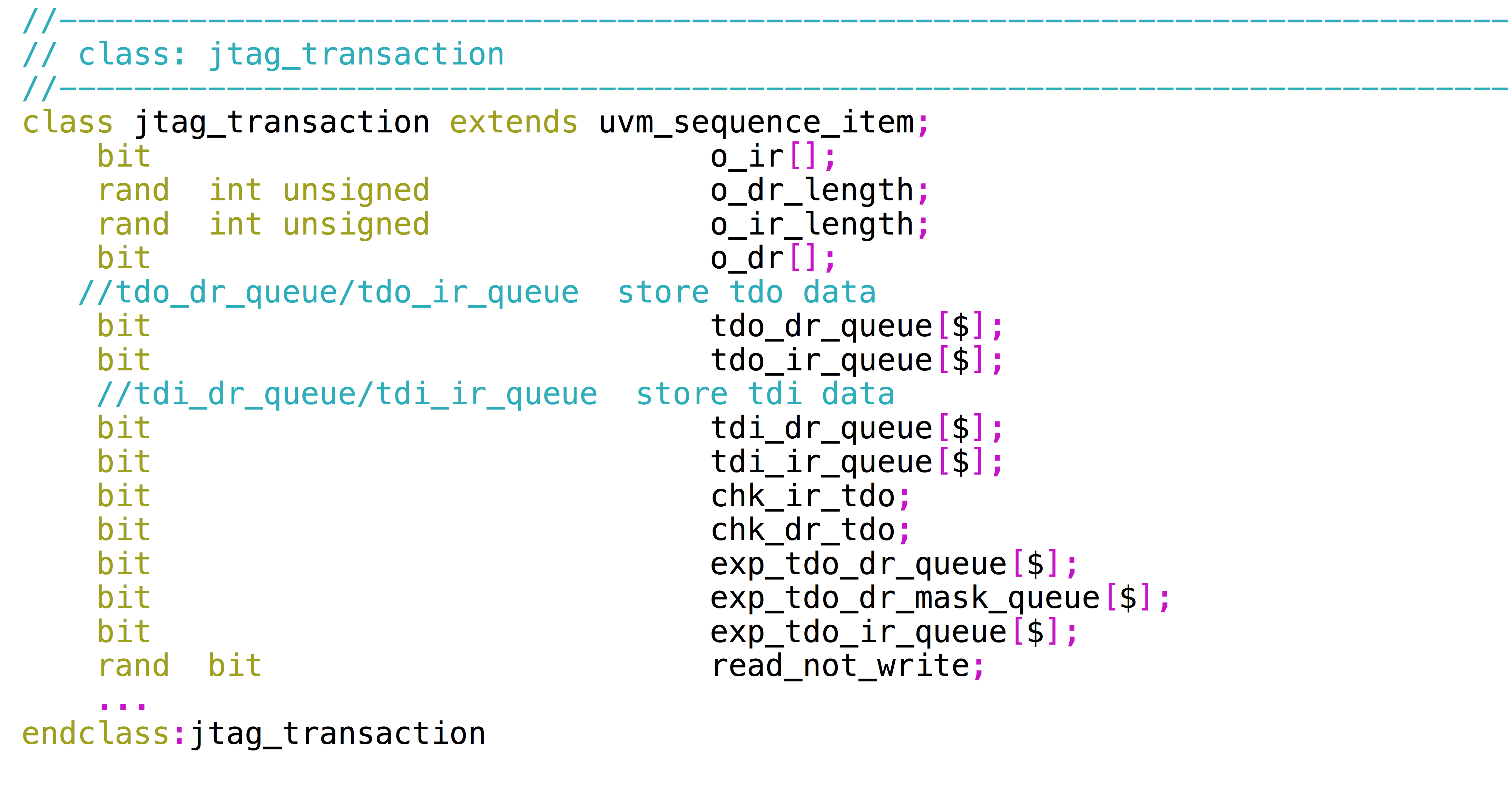


##### jtag\_agent\_configuration



BOZO: describe property usage.

##### jtag\_transaction class



o\_ir is a dynamic array to store instruction operation code(a.k.a OPCODE) to be sent to DUT JTAG 1149.1 FSM IR(Instruction Register).

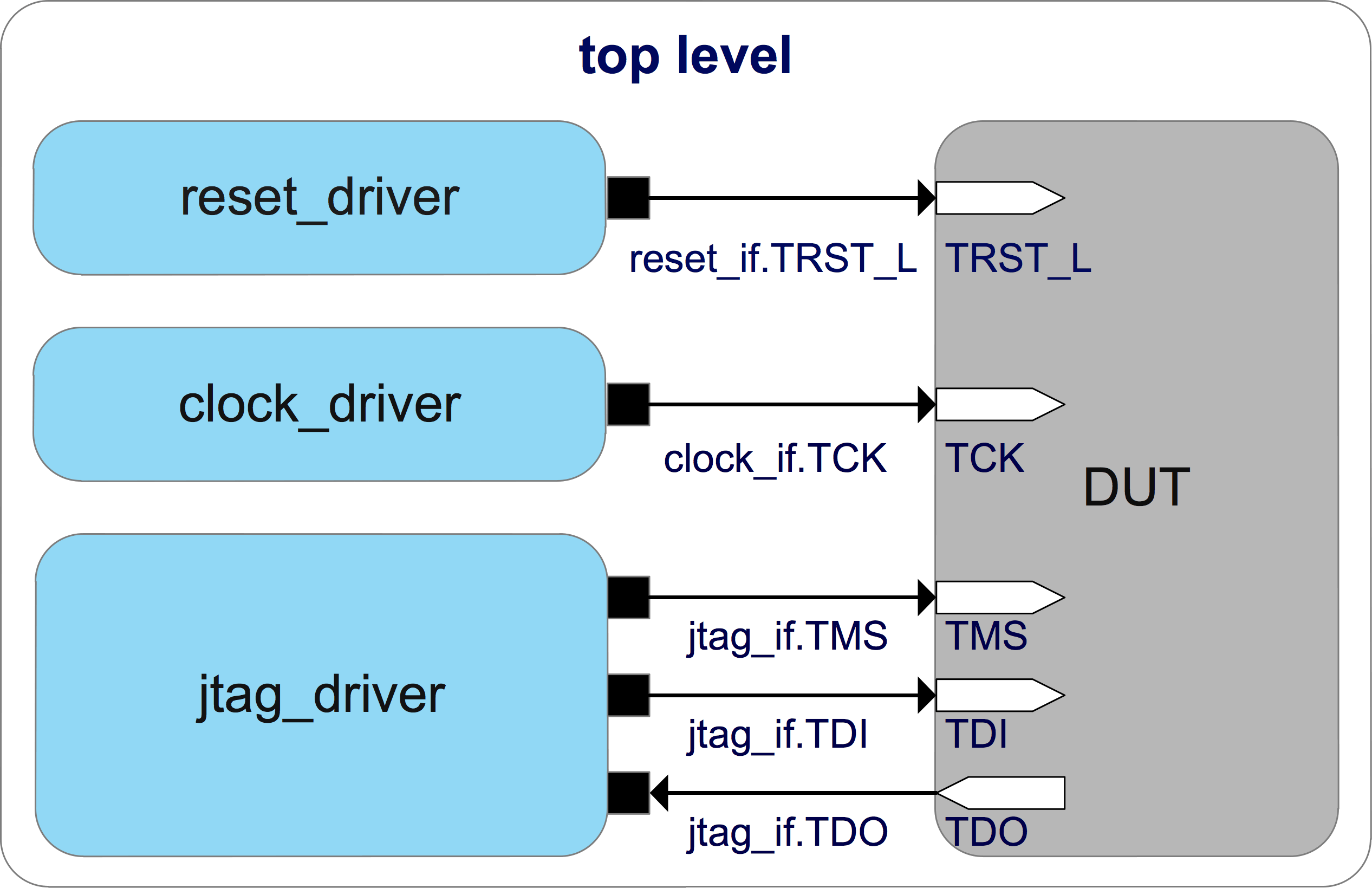
o\_dr is a dynamic array to store data to be sent to DUT’s IEEE 1149.1 compliance FSM DR(Data Register).

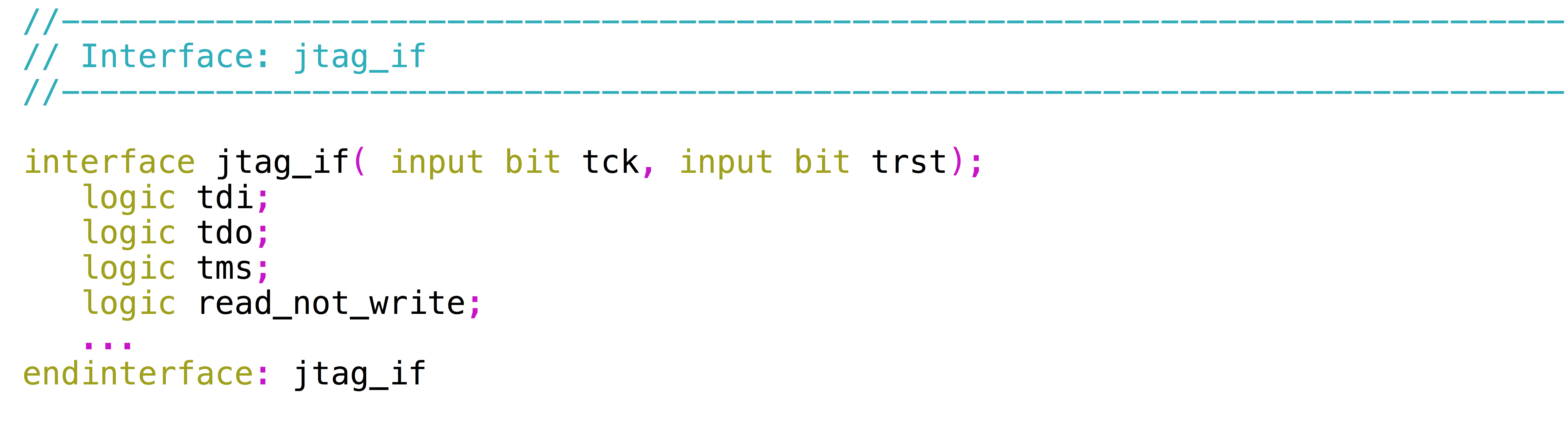
chk\_ir\_tdo and chk\_dr\_tdo are flags to indicate jtag\_driver whether to check TDO cycle by cycle during shift IR or DR state.

##### JTAG interface

Since this paper categorize pads of a SoC into 4 types which are driven by different drivers, a JTAG interface is driven by clock driver, reset driver and JTAG driver as show in Figure x.

jtag\_if is defined in Figure x.





##### 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, exp\_tdo\_dr\_mask\_queue and exp\_tdo\_ir\_queue stores 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 STIL generator through an analysis port, which is an object of uvm\_analysis\_port class specialized with stil\_info\_transaction type.

In Figure x, suppose JTAG driver’s FSM is in shift DR state, it’s 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, keep TSM to low to let DUT’s FSM stay in shift DR state; drive TDI to high. Such information is converted to STIL information as show in line 1 by call\_stil\_gen function.

At TCK positive edge b, sample TDO and compare it with golden value which is 1 bit 1 and keep TDI and TMS value. Such information is converted to STIL information as show in line 2 by call\_stil\_gen function.

At TCK negative edge c, keep TSM to low to let DUT’s FSM stay in shift DR state; drive TDI to low. Such information is converted to STIL information as show in line 3 by call\_stil\_gen function.

At TCK positive edge d, sample TDO and compare it with golden value which is 1 bit 1 and keep TDI and TMS value. Such information is converted to STIL information as show in line 4 by call\_stil\_gen function.

At TCK negative edge e, drive TSM to low to let DUT’s FSM go to exit1 DR state; drive TDI to low. Such information is converted to STIL information as show in line 5 by call\_stil\_gen function.

At TCK positive edge f, sample TDO and compare it with golden value which is 1 bit 0 and keep TDI and TMS value. Such information is converted to STIL information as show in line 6 by call\_stil\_gen function.

##### jtag\_monitor class

There is a signal called read\_not\_write defined in JTAG interface 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.

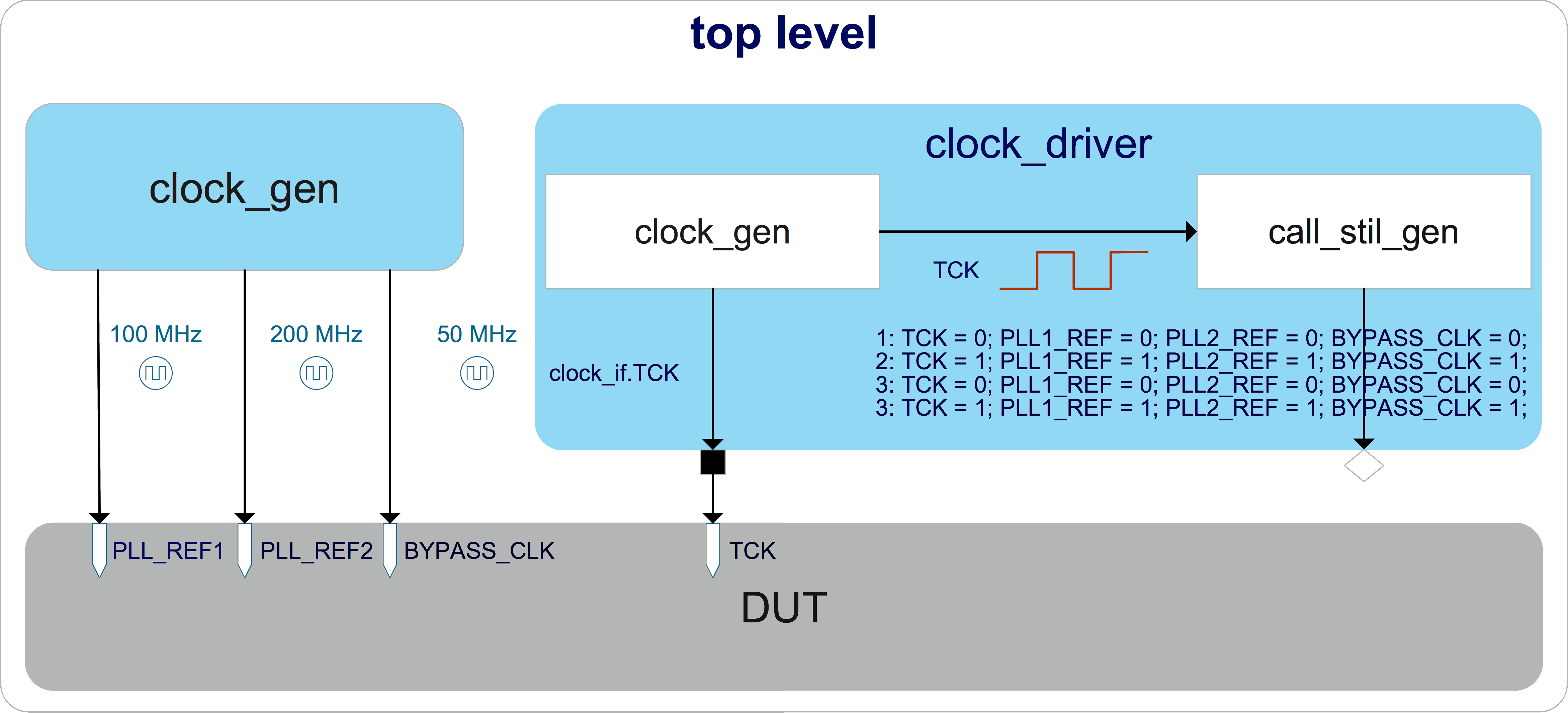
Read operation: data being shift in a register is same as the data stored in it.

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.

#### clock\_driver Implementation

Figure 4 is an example of how to toggle clocks in this DFT verification environment.



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 show in Figure 4

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.

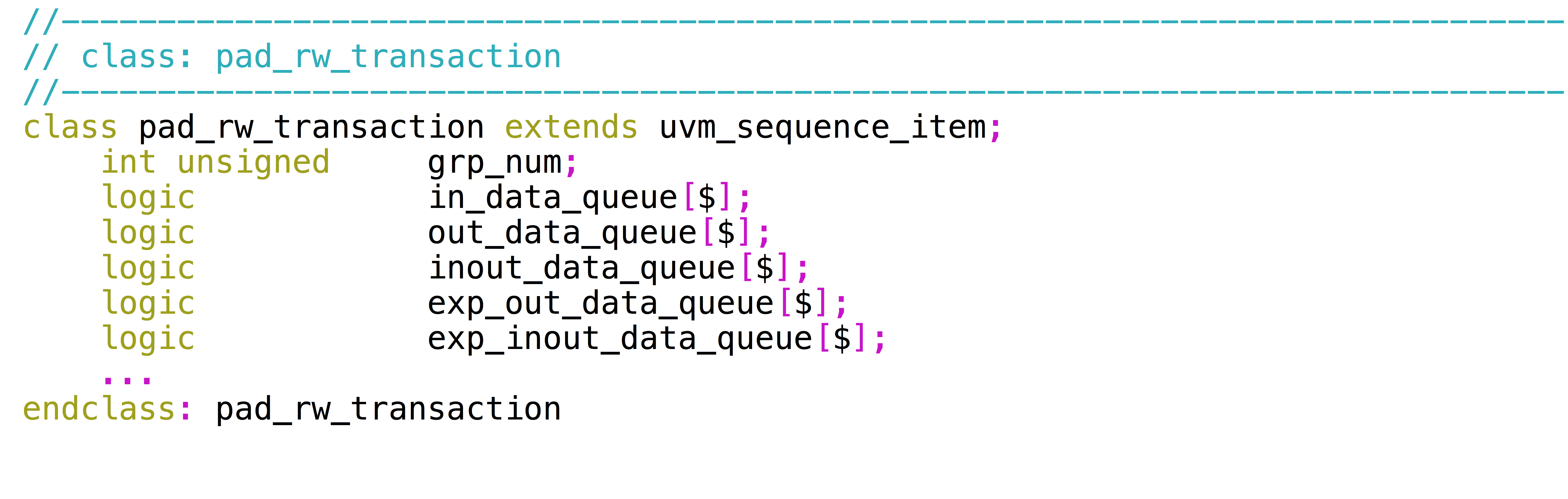
Why do this? Does that imply all clocks run on a same frequency as TCK for ATE test?

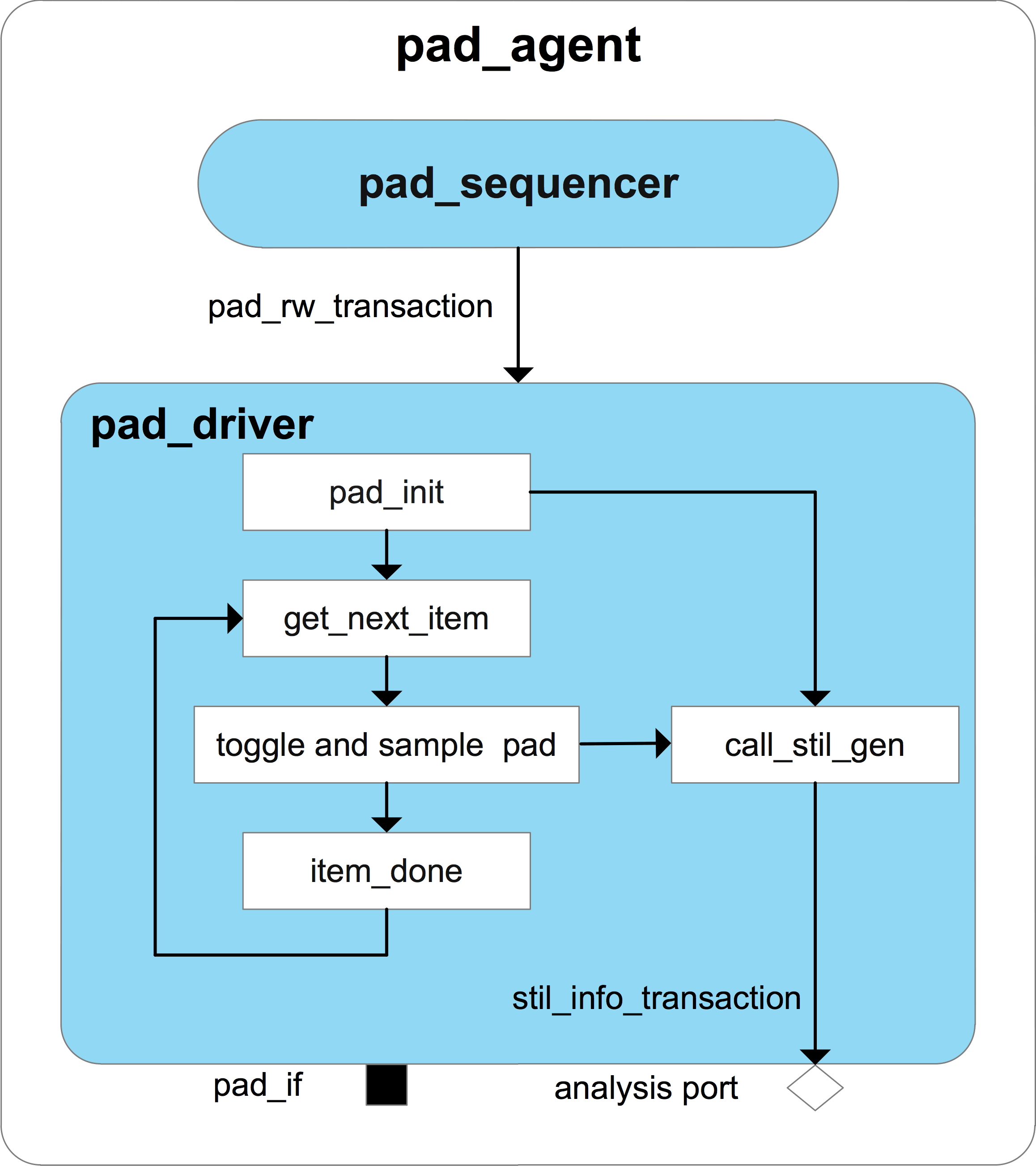
For ATE test, PLL1\_REF, PLL2\_REF and BYPASS\_CLK toggle information in STIL pattern can be regarded as a place holder to let post silicon engineers aware these three clocks are reference clocks, they will not use the toggle information described in STIL pattern to driver reference clocks, but use clocks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

#### pad\_agent Implementation

##### The pad type 4 defined in section xx, can be grouped according to their function. Take memory pads, GPIO pads and scan control pads as examples, each of them can be put in a group. So pad interface and pad\_rw\_transaction is designed as show in Figure x and Figure y.

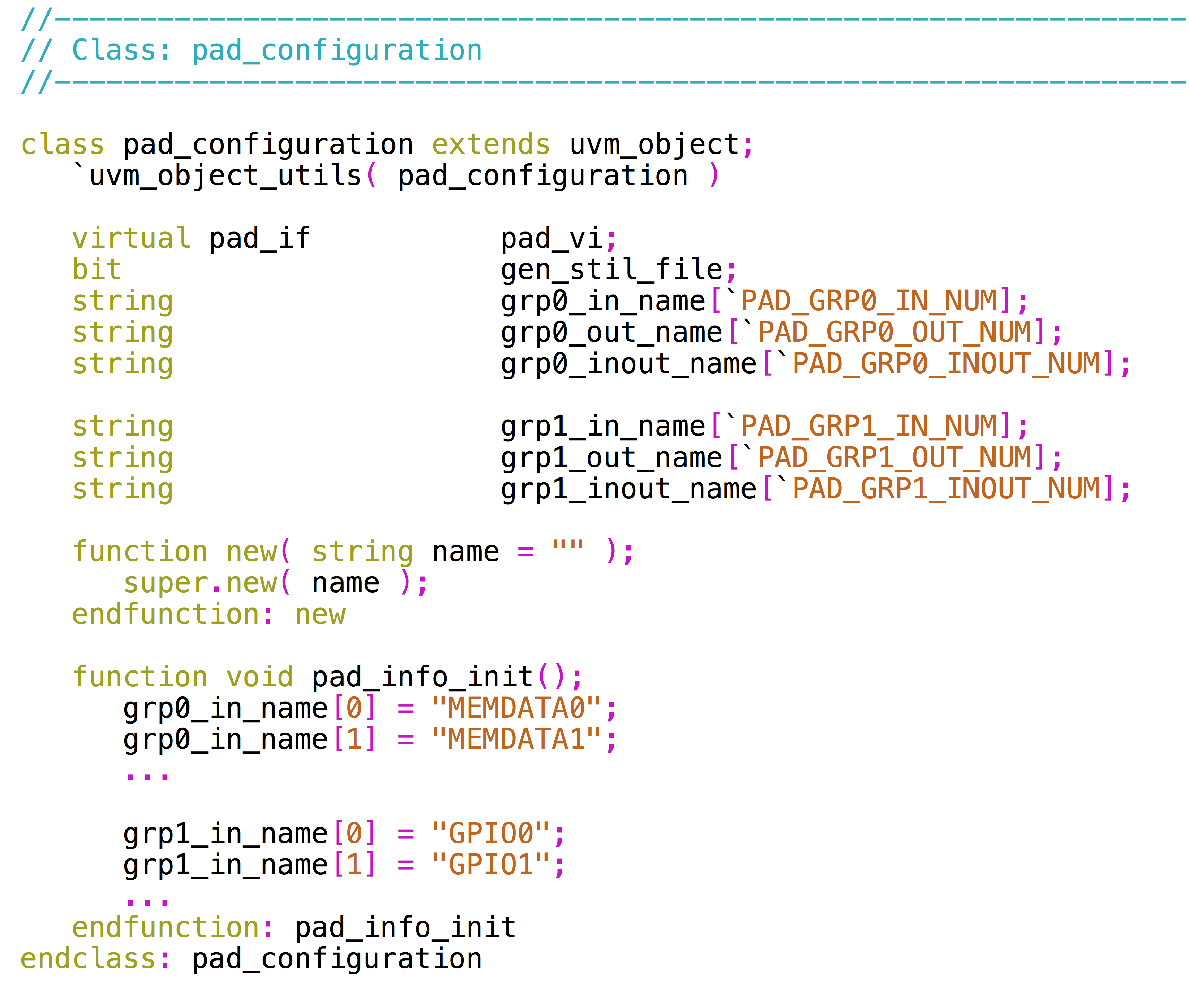
grp\_num is used to indicate pad\_dirver which group of pads is to drive.





pad\_init function will initialize all groups pads in turn at the beginning of a test, and call\_stil\_gen function will convert this information to STIL information and write to STIL generation through the analysis port as show in Figure x.

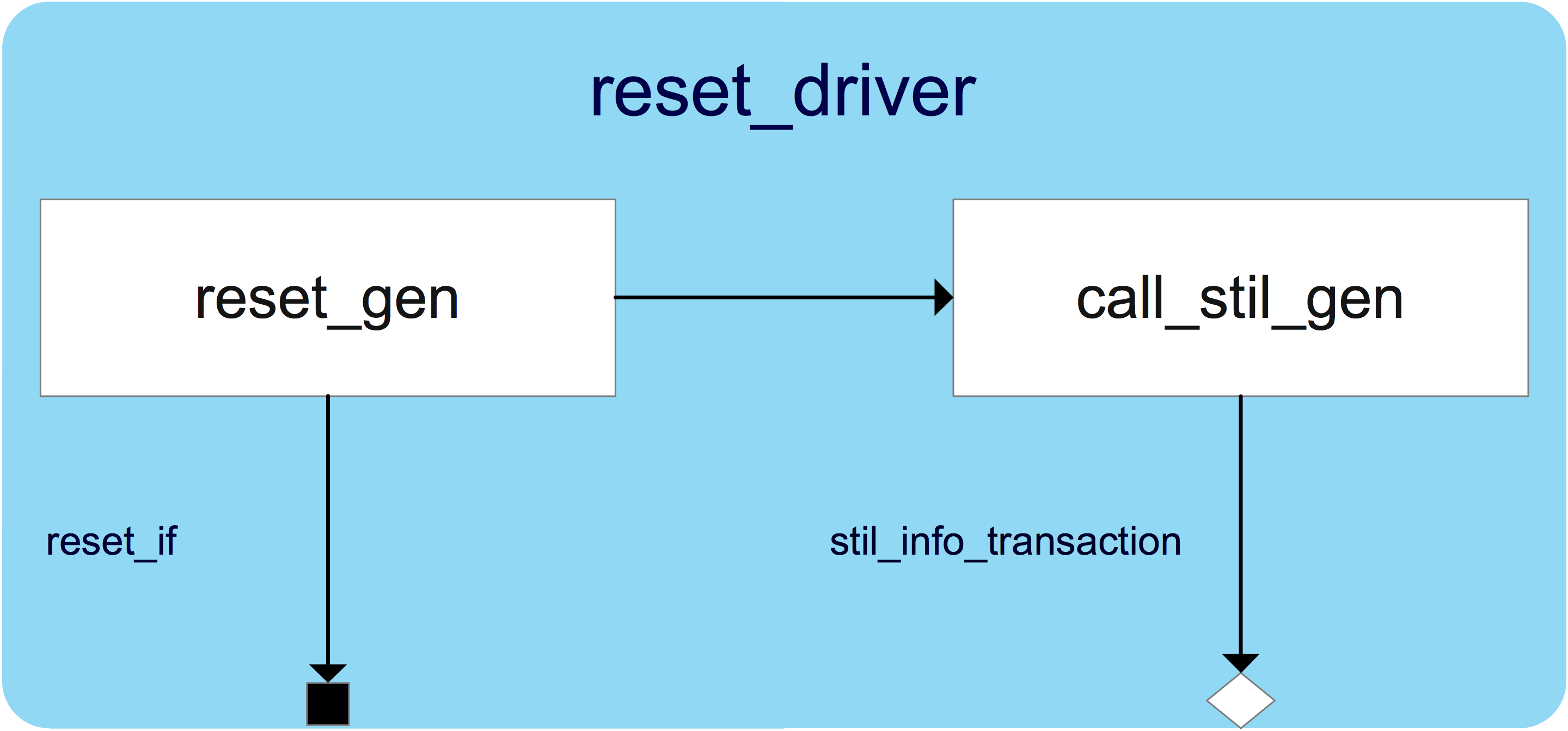
##### pad\_agent\_configuration



Store every pad’s name in pad\_configuration through pad\_info\_init function at the beginning of a test, which will be used in call\_stil\_gen function of pad\_driver to generate STIL information and STIL generator to print STIL header information.

#### reset\_driver class

Similar as clock\_driver, reset\_driver drive all resets signals defined in reset interface as show in Figure x.

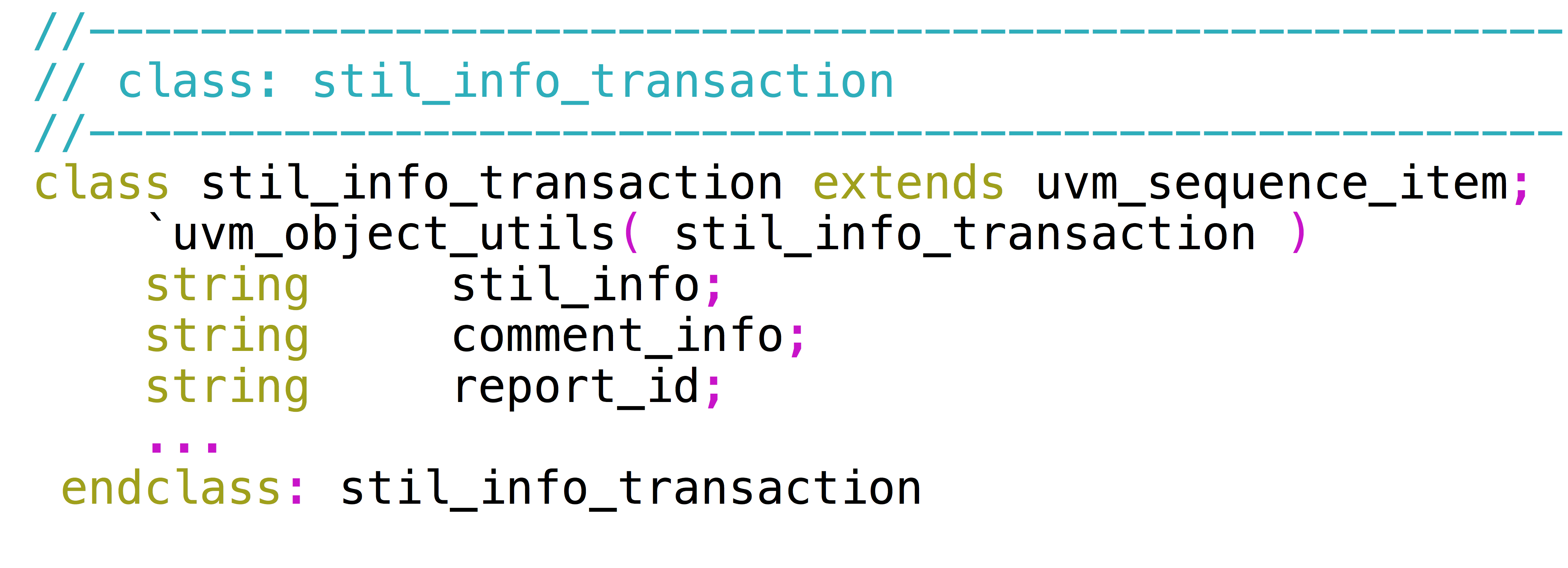


#### stil\_generator Class

STIL file construction

The stil\_generator, which extends from uvm\_subscriber class specialized with stil\_info\_transaction type, has 4 analysis exports to connect with clock\_driver, reset\_driver, pad\_driver and jtag\_driver’s analysis port. 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].

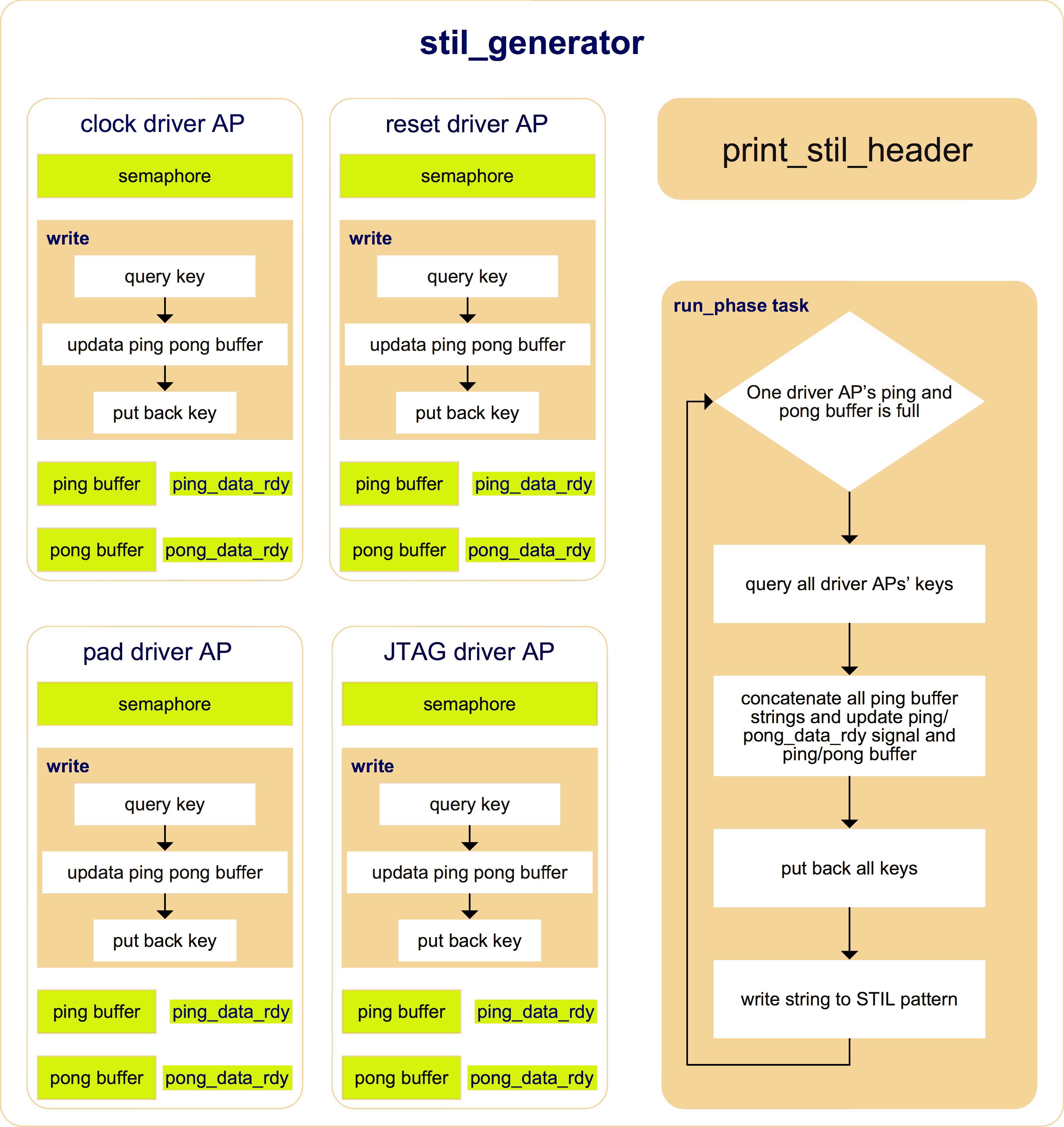
stil\_info\_transaction is defined as Figure x. stil\_info is pads toggle and measure information and comment\_info is comment going to be printed out with stil\_info.



In Figure x, each driver’s analysis port has its corresponding write () function, 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 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’s need to suspend the run task in stil\_generator until all others run task finish. But in UVM all uvm\_component run tasks are executed in parallel and stil\_generator itself is a 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.

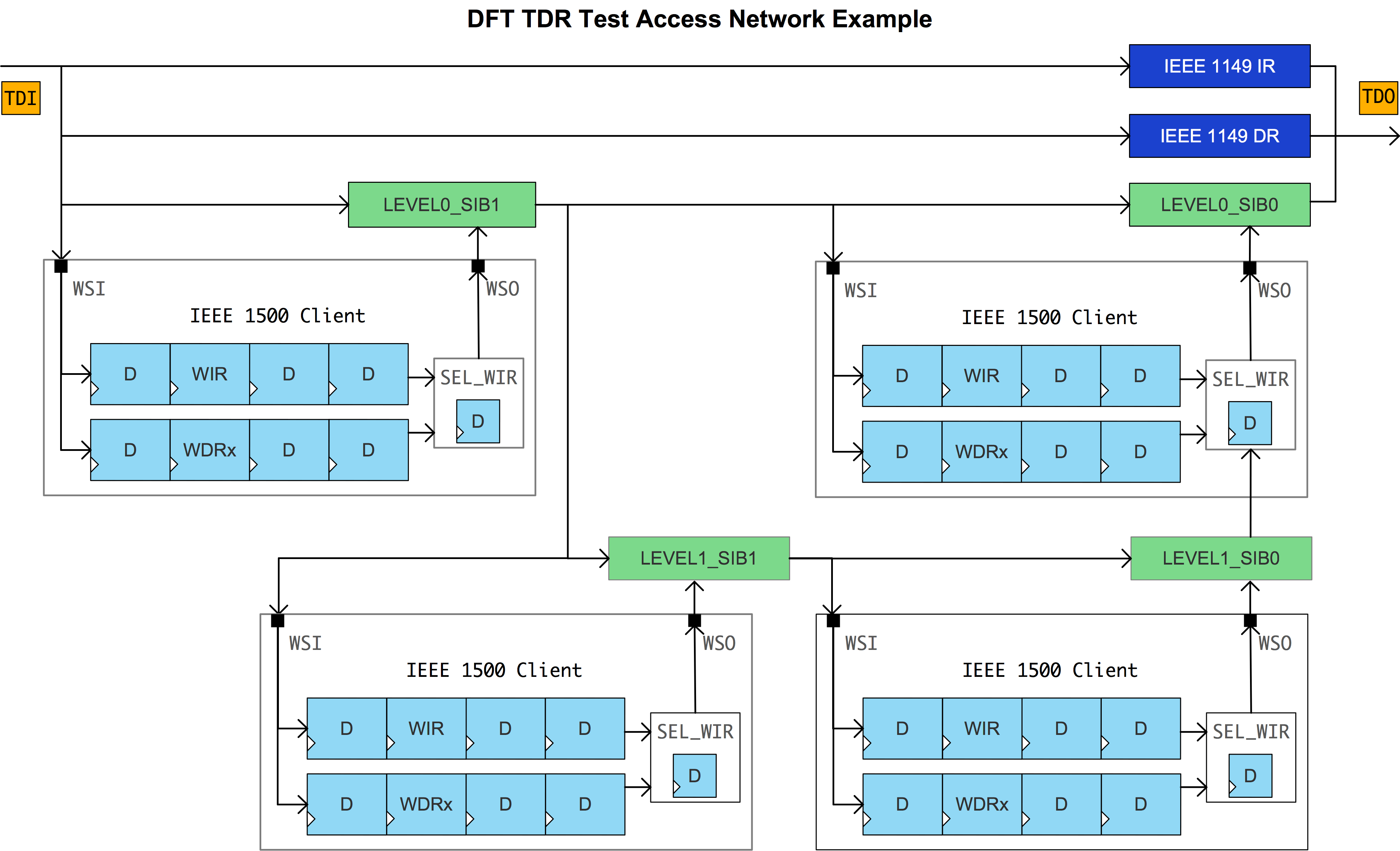
To resolve this, a group of ping-pong buffer is introduced. The write () function always write ping buffer first and then pong buffer, because ping data and pong data come at different simulation time slot, once a group of ping-pong buffer is full, which indicates the simulation has already moved forward, it’s the right time to collect all ping buffer data and write them out. The run\_phase task as show in Figure x 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.); put back all keys and write a test vector to STIL pattern.



### DFT TDR Abstraction

#### 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, as show in Figure x.

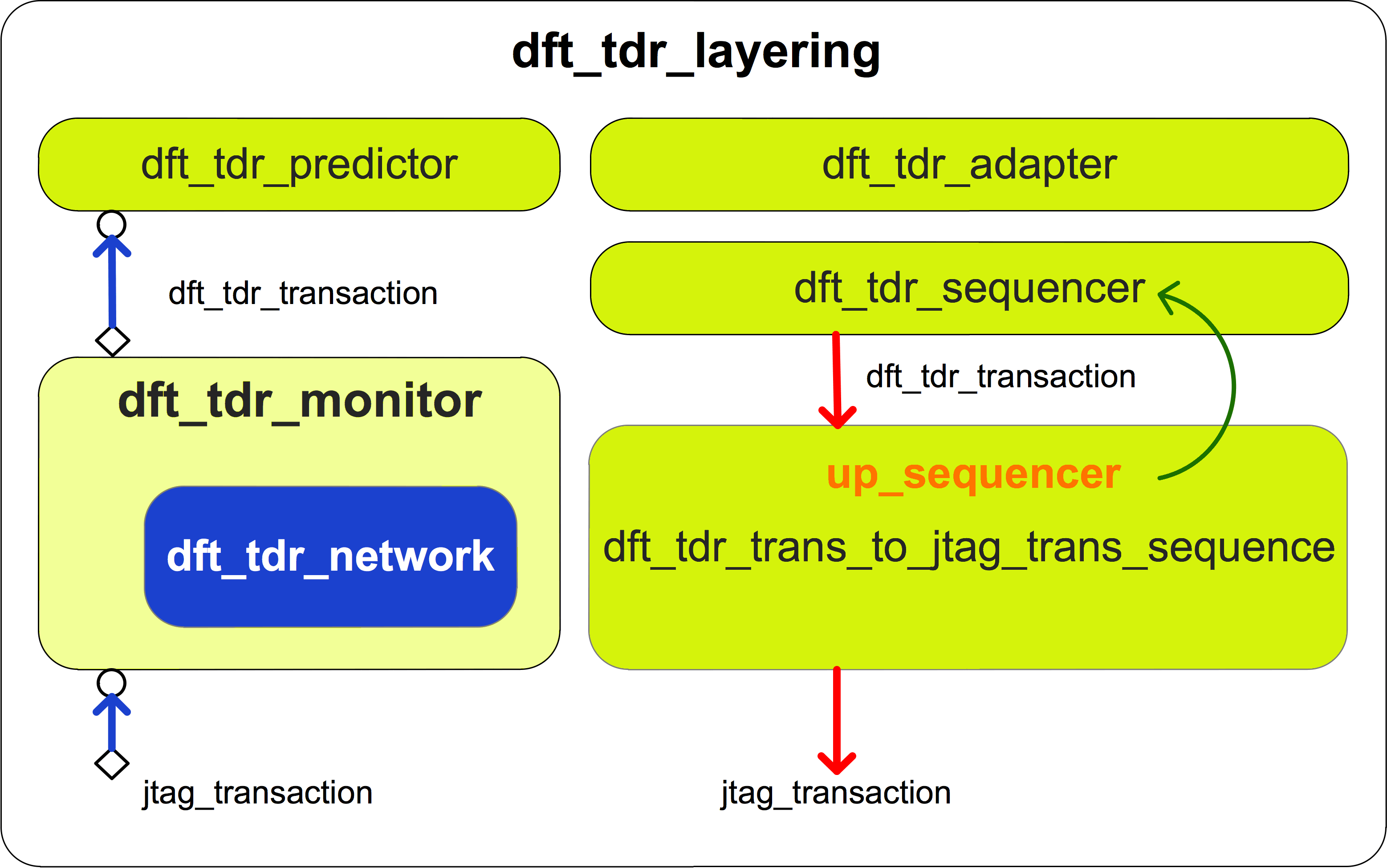


In order to hide the complex operation of accessing a TDR hierarchically located in the network, it’s necessary to level up TDR access in RAL.

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 a IEEE1500 TDR base class and a 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 be updated accordingly.

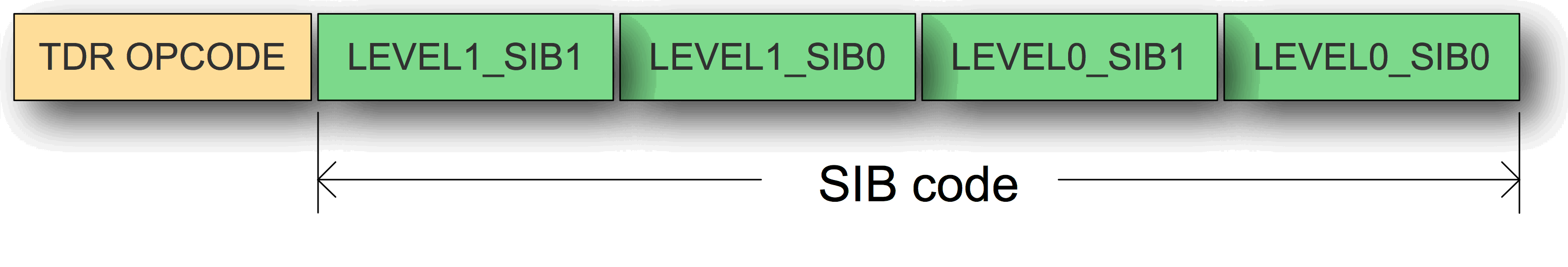
This paper presents a neat and easy maintain way to abstract TDR in UVM based environment as show in Figure x.



We can encode a TDR’s location information into its address showing in Figure x 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.

The benefit of this method is when TDR access network changes, only need to update dft\_tdr\_network and dft\_tdr\_trans\_to\_jtag\_trans\_sequence, all TDR classes don’t need any update which can save a lot of test environment setup time when a project product moving forward to next generation.

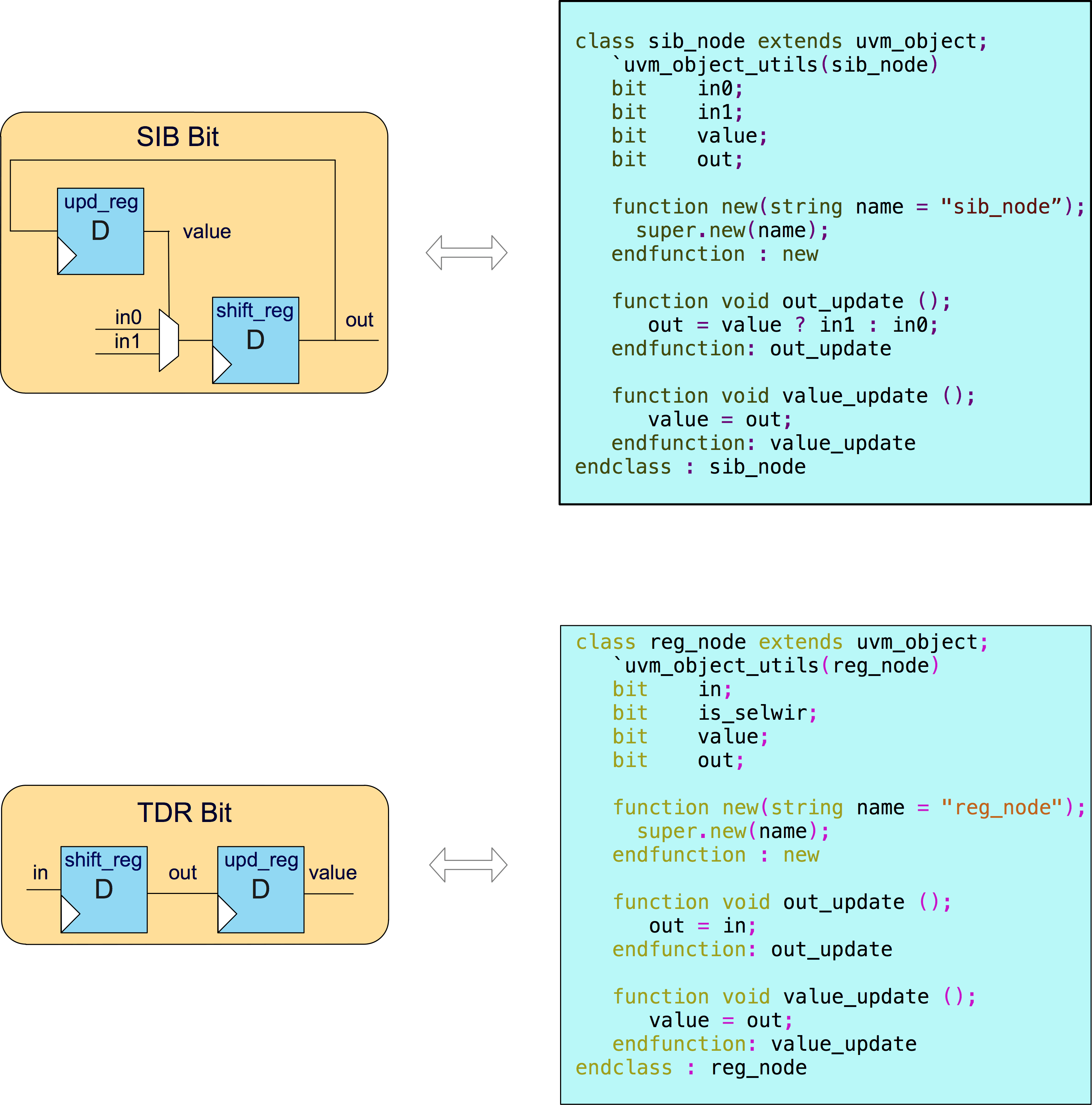


##### DFT TDR Access Network Modelling

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

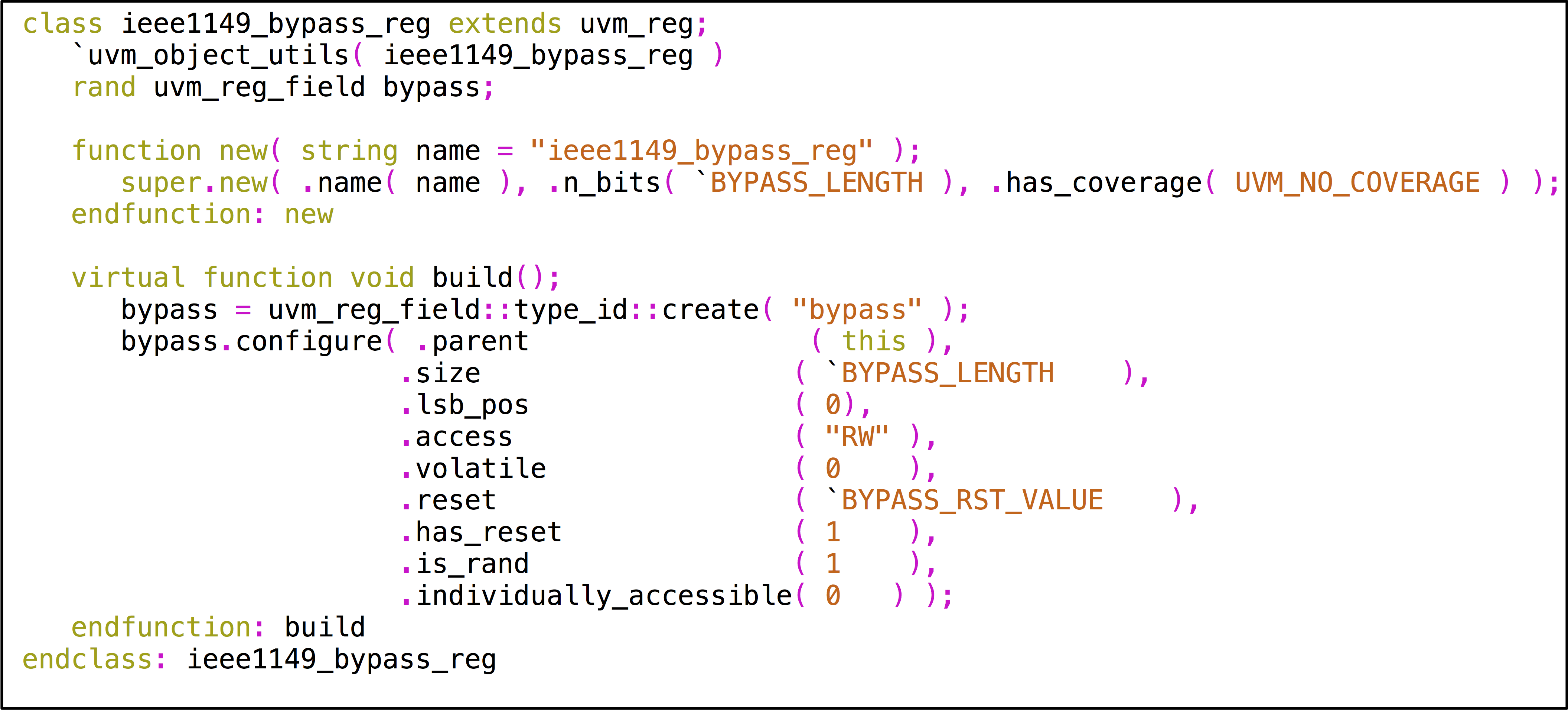
out\_update () function is to model the active clock edge triggering the D flip flop (shift register) during shift operation and value\_update () function is to model the active clock edge triggering the D flip flop (update register) during update operation.

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



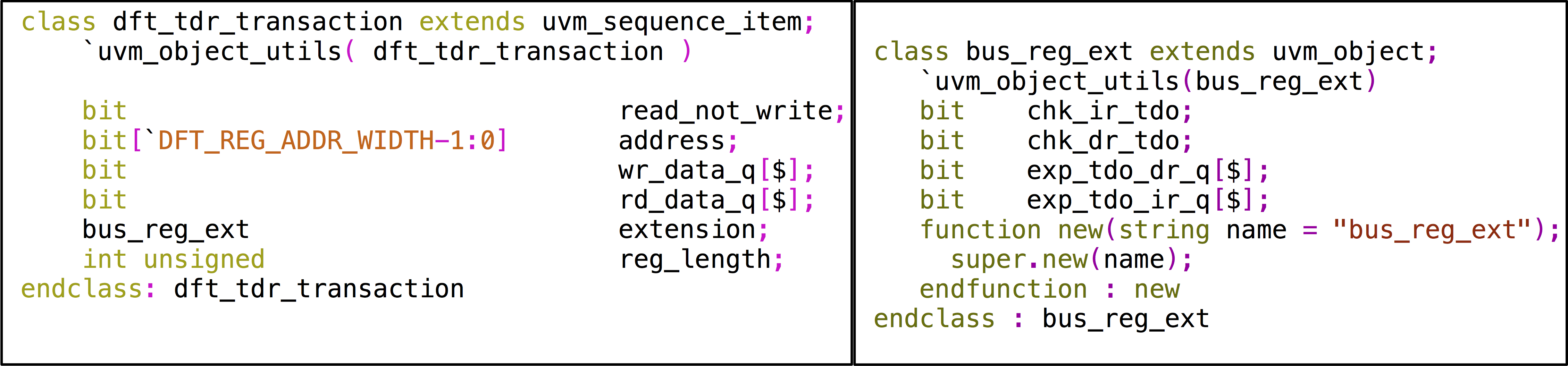
#### DFT TDR Class Definition

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



#### 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.



### STIL Test Pattern Verification

### Conclusion

discussion: other agent, such scan agent, how to implement?