UVM-Simulationsmodell eines JTAG-Interfaces

Serin Varghese

June 9, 2017

UVM-Simulationsmodell eines JTAG-Interfaces

TECHNISCHE UNIVERSITÄT
CHEMNITZ

Masters in Micro and Nano Systems
Fakultät für Elektrotechnik und Informationstechnik
Professur Schaltkreis- und Systementwurf
Technische Universität Chemnitz

This is to certify that the work entitled "UVM-Simulationsmodell eines JTAG-Interfaces "is a bonafide work carried out by (**Varghese, Serin John; *Immatrikulation Nr. 328799***) is a contribution to Schaltkreis- und Systementwurf, Masters in Micro and Nano Systems, Technische Universität Chemnitz.

Betreuer/Prüfer	*Betreuer/Prüfer*
Dipl.-Ing. Marcel Putsche	Dipl.-Ing. Thomas Horn
SSE Department,	SSE Department,
TU Chemnitz	TU Chemnitz,

Date :
Place : Chemnitz

Acknowledgement

Abstract

[1]

[1]Keywords: Universal Verification Methodology, JTAG, verification component, TAP controller, boundary scan

Contents

1 Abbreviations used

- UVM - Universal Verification Methodology
- OVM - Open Verification Methodology
- DUT - Device Under Test
- BSDL - Boundary Scan Description Language
- DR - Data Register
- IR - Instruction Register
- TAP - Test Access Port
- TCK - Test ClocK input
- TDI - Test Data Input
- TDO - Test Data Output
- TMS - Test Mode Select
- TRST - Test ReSeT input
- VC - Verification Component
- IP - Intellectual Property
- PCB - Printed Circuit Board
- IC - Integrated Circuit
- FSM - Finite State Machine
- BSR - Boundary Scan Register
- RTL - Register Transfer Level
- DUT - Device Under Test
- IDE - Integrated Development Environment

2 Introduction

2.1 Introduction to Boundary Scan and JTAG

Boundary Scan is a method of testing interconnects on PCBs and internal IC sub-blocks. This standard is defined in IEEE 1149.1 For boundary scan tests, additional logic is added to the device. The boundary scan cells are placed between the core logic and the ports.

JTAG is an established technology(and industry standard) with a potential that is only now becoming fully realised. Connection testing and In System Programming(ISP) are the two applications most often associated with JTAG, but it has far more to offer.

2.1.1 Background

JTAG was initially conceived to address difficulties in testing circuits using the traditional 'bed-of-nails' approach. Modern packaging technologies like BGA and Chip Scale Packaging limit and in some cases eliminate physical access to pins. JTAG overcomes this problem, by placing cells between the external connections and the internal logic of the device. With the cells configured as a shift register, JTAG can be used to set and retrieve the values of pins(and nets connected to them) without physical access.

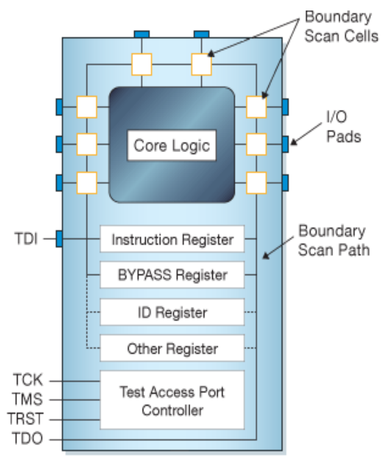

Figure 1: JTAG Block Diagram

There is also an option to sample the data values as they pass between the core logic and the pins during the normal operation of the device.

The JTAG interface adds four extra pins to each device:

- TDI to input data to the device
- TDO to output data from the device
- TMS to control what should be done with the data
- TCK clock signal to synchronize everything

If a circuit contains more than one JTAG-compliant device, these can be linked together to form a JTAG chain. In a JTAG chain the data output from the first device becomes the data input to the second device; the control and the clock signals are common to all the devices in the chain. Fig. provides a representation of a simple JTAG chain containing three devices.

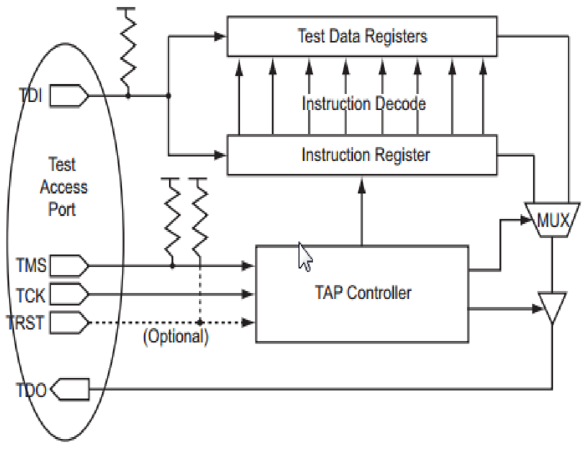

Figure 2: Simple JTAG Device1

2.1.2 Test Access Port (TAP)

Each test logic function is accessed through the TAP. The five pins associated with the TAP are listed on the 1 with their corresponding descriptions. Four pins - TMS, TCK, TDI, and TDO - are always required for JTAG operation. The fifth pin, TRST, is optional. These pins are dedicated pins - used only with the test logic.

Table 1: Test Access Port Descriptions

Port	Description
Test Mode Select (TMS)	Serial Input for the test logic control bits. Data is captured on the rising edge of the test logic clock (TCK). An internal pull-up resistor is present in dedicated mode but not in flexible mode.
Test Clock Input (TCK)	Dedicated test logic clock used serially to shift test instruction, test data, and control inputs on the rising edge of the clock, and serially to shift the output data on the falling edge of the the clock.
Test Data Input (TDI)	Serial input for instruction and test data. Data is captured on the rising edge of the test logic clock. This pin is equipped with an internal pull-up resistor.
Test Data Output (TDO)	Serial output for test instruction and data from the test logic. TDO is set to an Inactive Drive state (high impedance) when data scanning is not in progress.
Test Reset (TRST)	Active-low input which asynchronously resets the test logic. This pin is equipped with an internal pull-up resistor.

TRST overrides the behavior of the TMS and TCK. In other words, asserting TRST resets the TAP controller regardless of the the states of the TMS and TCK. Also, if TAP controller is held in the reset state, the state machine remains in the 'Test Logic Reset' condition.

2.1.3 TAP Controller

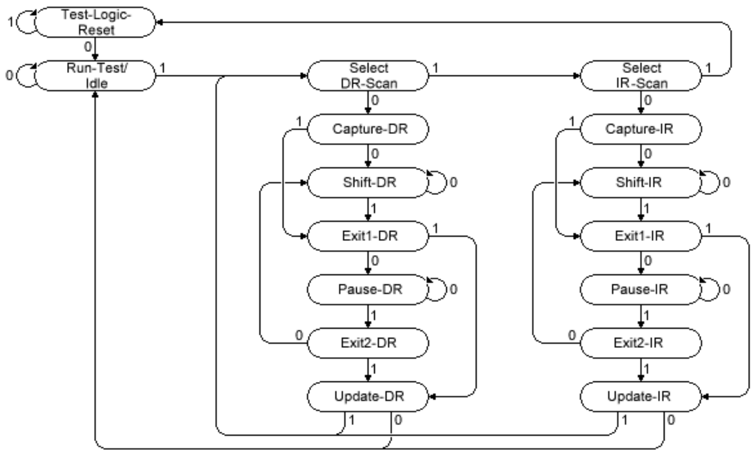

Figure 3: TAP CONTROLLER

The 16 states of the TAP controller finite state machine are shown in the fig **??**. The 1s and 0s shown adjacent to the state transitions represent the TMS values that must be present at the time of a rising edge at TCK for a state transition to occur. In the states that include the letters -IR, the instruction register operates. In the states that include the letters -DR, the test data registers operates (bypass, boundary scan).

By default, upon power up(or when TRST is asserted) the TAP controller enters the Test-Logic-Reset state. The TAP controller also has an inherent property for automatically reaching this state when the TMS signal is held high for atleast 5 clock signals.

The operation of each state is explained below:

- **Test-Logic-Reset**
  All test logic is disabled in this controller state enabling the normal operation of the IC.
- **Run-Test-Idle**
  In this controller state, the test logic in the IC is active only if certain instructions are present. For example, if an instruction activates the self test, then it is executed when the controller enters this state. The test logic in the IC is idle otherwise.
- **Select-DR-Scan**
  This controller state controls whether to enter the Data Path or the Select-IR-Scan state.
- **Select-IR-Scan**
  This controller state controls whether or not to enter the Instruction Path. The controller can return to the Test-Logic-Reset state otherwise.
- **Capture-IR**
  In this controller state, the shift register bank in the Instruction register parallel loads a pattern of fixed values on the rising edge of TCK. The last two significant bits must always be '01'.
- **Shift-IR**
  In this controller state, the instruction register gets connected between TDI and TDO, and the captured pattern gets shifted on each rising edge of TCK. The instruction available on the TDI pin is also shifted in on to the instruction register.

- **Exit1-IR**
  This controller state controls whether to enter the Pause-IR state or Update-IR state.
- **Pause-IR**
  This state allows the shifting of the instruction register to be temporarily halted.
- **Exit2-IR**
  This controller state controls whether to enter either the Shift-IR state or Update-IR state.
- **Update-IR**
  In this controller state, the instruction in the instruction register is latched to the latch bank of the Instruction Register on every falling edge of TCK. The instruction becomes the current instruction once it is latched.
- **Capture-IR**
  In this controller state, the data is parallel-loaded into the data registers selected by the current instruction on the rising edge of TCK.
- **Shift-DR, Exit1-DR, Pause-DR, Exit2-DR and Update-DR**
  These controller states are similar to the Shift-IR, Exit1-IR, Pause-IR, Exit2-IR and Update-IR states in the Instruction Path.

2.1.4 Registers

- **Instruction Register** The instruction register allows an instruction to be shifted into the design. The instruction is used to select the test to be performed or the test data register to be accessed or both. Optionally, the instruction register allows examination of design-specific information generated within the component.

  Each IR cell in the Instruction Register has a shift-register stage and a latch stage for fault isolation of the board-level serial test data path.

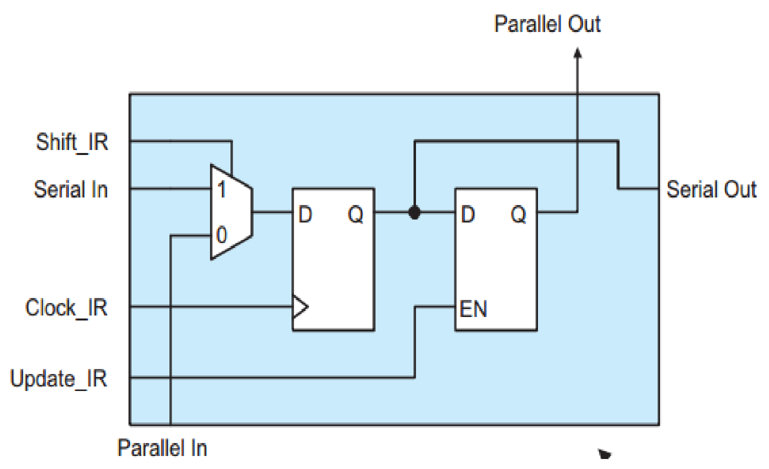

Figure 4: Instruction Register

Table 2: Supported Instructions

Instruction	**IR-Code (IR3-IR0)**	**Instruction Type**	**Description**
EXTEST	0000	Mandatory	Allows testing of off-line circuitry and board-level interconnections
SAMPLE/PRELOAD	0001	Mandatory	Allows a snapshot of the normal operation of the component to be taken and examined
IDCODE	0010	Optional	32-bit hard-wired Manufacturer ID, part number, and version number
BYPASS	1111	Mandatory	Provides minimum-length (1-bit) serial path between TDI and TDO pins of component when no test operation of that component is required
INTEST	XXXX	Optional	Allows testing of on-chip system logic while component is assembled on the board

- **Data Register**

  I Boundary-Scan Register
  The boundary-scan register allows testing of circuitry external to a component, for example, board interconnect or external components that do not conform to this standard. The register also permits the system signals flowing into and out of the system logic to be sampled and examined without causing interference with the normal (nontest) operation of the on-chip system logic. Optionally, additional test functions may be supported - for example, testing of the on-chip system logic.

  II Bypass Register
  This provides a single-bit serial connection through the circuit when none of the other test data registers is selected. This register can, for example, be used to allow test data to flow through a particular device to other components in a product without affecting the normal operation of the particular component.

  III Device Identification Register
  This is an optional test data register that allows the manufacturer, part number, and variant of a component to be determined.

2.2 Introduction to Verification Methodologies:

2.2.1 Classical verification vs Constraint based verification:

With the increasing complexity of the digital systems, comes the need to have smarter ways to verify the functionality of the designed DUT. Initially the digital were tested with tediously written test-benches and then observing the respective waveform. The higher complexity did no longer allow for the manual checks and there was a need to automate the verification methods.

Verification planning and management involves identifying the features of the DUT that need to be verified, prioritizing those features, measuring progress, and adjusting the allocation of verification resources so that verification closure can be reached on the required timescale. The mechanics of verification can be accomplished using static formal verification in the context of UVM focuses on the simulation-based verification environment.

There are two contrasting approaches to coverage-driven verification in current use. "Classical" constrained random verification starts with random stimulus and gradually tightens the constraints until coverage goals are met, relying on brute power of randomization and compute server farms to cover the state farms to cover the state space. More recently, graph-based stimulus generation (also known as Intelligent Testbench) starts from an abstract description of the legal transitions between the high-level states of the DUT, and automatically enumerates the minimum set of tests needed to cover the paths through this state space. For many application, graph-based stimulus is able to achieve high coverage in far fewer cycles than "classical" constrained random. UVM directly supports constrained random, whereas graph-based stimulus generation requires a separate, dedicated tool. Stimulus generated from the graph-based approach can be executed on a UVM verification environment.

Functional coverage and code coverage measure different things. Code coverage measures the execution of the actual RTL code (which must therefore exist before the code coverage can run at all). The collection of code coverage information, including statement and branch coverage, state coverage, and state transition coverage, is largely automatic. Functional coverage, on the other hand, attempts to measure whether the features described in the verification plan have actually been executed by the DUT. The feature to be measured have to be decided from the specification and implementation of the design to create the verification plan, and so functional coverage can be considered as a qualitative measure of DUT code execution.

The best practice is to create a verification plan that consists of a list of features to be tested as opposed to a list of direct test descriptions. All stakeholders in the verification process should contribute to the identification and prioritization of features in the verification plan, since this feature set will form the foundation for the subsequent verification process.

2.2.2 Advantages of Functional coverage

Functional coverage helps to identify

- the features in the verification plan that have been successfully tested
- the features in the verification plan that have yet to be tested
- the proportion of the features that have been tested and thus how close the verification process is to completion
- the set of tests that provide maximum coverage using the minimum number of CPU cycles

In contrast, in traditional directed testing methodology, the absence of further bugs being detected is taken as evidence that verification is nearly complete. This may overcome some scenarios in which the DUT might fail.

2.2.3 What is UVM?

Introduction to UVM

UVM is a methodology for functional verification using SystemVerilog, complete with a supporting library of SystemVerilog code. UVM stands for Universal Verification Methodology. It was created by Accelera based on the OVM(Open Verification Methodology) version 2.1.1.

It is basically a methodology for the functional verification of digital hardware, primarily using simulation. The hardware or system would be typically be described using Verilog, SystemVerilog, VHDL or SystemC at any appropriate abstraction level. This could be behavioral, Register-transfer level, or gate level. UVM is explicitly simulation-oriented, but UVM can also be used alongside assertion-based verification, hardware acceleration or emulation.

History

In December 2009, a technical subcommittee of Accellera - a standards organization in the electronic design automation (EDA) industry - voted to establish the UVM and decided to base this new standard on OVM 2.1.1, a verification methodology developed jointly in 2007 by Cadence Design Systems and Mentor Graphics. In February 21, 2011, Accelera approved the 1.0 version of UVM. It included a Reference Guide, a Reference Implementation in the form of SystemVerilof base class library, and a User Guide.

Checkers, Coverage and Constraints

Constrained random verificaiton relies on Checkers, Coverage and Constraints and these are supported by explicit features of the SystemVerilog language.

Firstly, checkers ensure functional correctness. Nothing is gained by throwing more and more random stimulus into a design to take functional coverage to ever higher levels unless the design-under test is being checked automatically for functional correctness. Checkers can be implemented by SystemVerilog assertions or using regular procedural code. UVM provides mechanisms and guidelines for building checkers into the verification environment and for logging reports.

Secondly, coverage provides a measure of the functional completeness of the testing, and tells us when we have met the goals set out in the verification plan, and thus when you have finished simulating. SystemVerilog offers two separate mechanisms for functional coverage collection; property-based coverage (cover directives) and sample-based coverage (covergroups). Both can be used in a UVM verification environment. The specification and the execution of the coverage information is intimately tied to the verification plan, and many simulation tools are able to annotate coverage information onto the verification plan document, facilitating tight management control.

Thirdly, constraints provide the means to reach coverage goals by shaping the random stimulus to push the DUT into interesting corner cases. Without shaping, random stimulus alone may be insufficient to exercise many of the deeper states of the DUT. Constrained random stimulus is still random, but the statistical distribution of the vectors is shaped to ensure that interesting cases are reached. Systemverilog has dedicated language features for expressing constraints, and UVM goes further by providing mechanisms that allow constraints to be written as a part of a test rather than embedded within verification components. this and other features of UVM facilitate the creating of reusable verification components.

Verification Reuse
UVM facilitates the constructino of verification environments and tests, both by providing reusable machinery in the form of a library of SystemVerilog classes, and alos by providing a set of guidelines for best practice when using SystemVerilog for verification.

Verification productivity can be enhanced by reusing verification components, and this is an an important objective of UVM. Verification reuse is enabled by having a modular verification environment where each component has clearly defined responsibilities, by allowing flexibility in the wat in which components are configured and used, by having a mechanism to allow imported components to be customized to the application at hand, and by having well-defined coding guidelines to ensure consistency.

The architecture of UVM has been designed to encourage modular and layered verification environments, where verification components at all layers can be reused in different environments. Low-level driver and monitor components can be reused across multiple DUT. The whole verification environment can be reused by multiple tests and configured top-down by those tests. Finally, test scenarios can be reused from application to application. This degree of reuse is enabled by having UVM verification components able to be configured in a very flexible way without modification to their source code. This flexibility is built into the UVM class library.

3 Objective and Specifications

Development of a Universal Verification Methodology environment of the JTAG interface. It will contain the following functionality:

- Existing and verified IP core of a JTAG interface
- Execution of the following Instructions:
  EXTEST, INTEST, SAMPLE/PRELOAD, BYPASS, IDCODE according to IEEE1149.1 standard
- Enhanced test bench(es) to fully test the DUT

The simulation runs as well as the occurring challenges are documented.
IDE used: Questa®Advanced Simulator, Mentor Graphics

4 Verification Environment

Blocks in UVM

We have a DUT and to test the functionality we have to simulate it. To achieve this, we will need a block that generates sequences of bits to be transmitted to the DUT. This block in UVM is called the *Sequencer*.

Usually the sequencer is unaware of the communication bus and the physical connections to the DUT. The sequencer is responsible only for generating generic sequences of data and then it is sent to another block that has direct access to the physical pins of the DUT. This block that interacts directly with the DUT is called the *Driver*.

While the driver maintains activity with the DUT by feeding it data generated from the sequencers, it does not validate the applied stimuli. We need a block that will listen to the communication between the driver and the DUT. This block is called the *Monitor*. Monitors sample the inputs/outputs of the DUT.

The monitor tries to make a prediction of the expected result and send the prediction and result of the DUT to another block of UVM. This block, the *Scoreboard*, compares and evaluates these data from the monitor.

All these blocks together constitute a typical system used for verification and the same structure is used in UVM testbenches. This is represented in fig.**??**

Usually the sequence, the sequencer, the driver and the monitor compose an *Agent*. An agent together with the scoreboard constitute an *Environment*. All these blocks are controlled by a greater block denominated by *Test*. The test block controls all the blocks and sub blocks of the testbench. By changing just a few lines of code, we could add, remove and override blocks in our testbench and build different environments without rewriting the whole test.

UVM classes

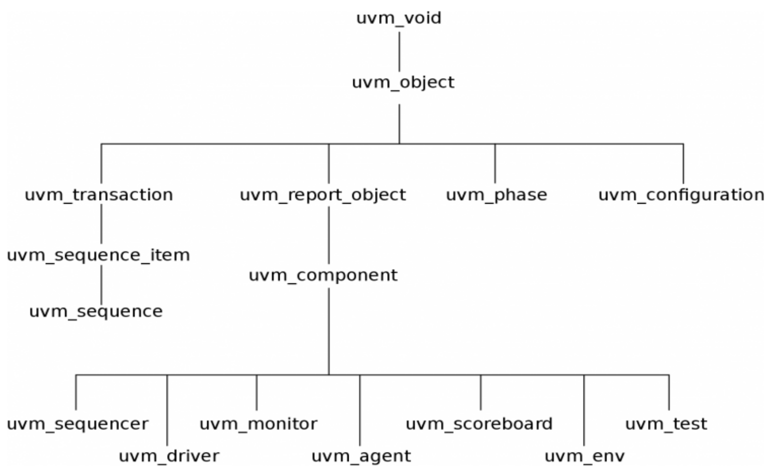

Figure 5: UVM Class Tree

The re-usability is one of the great advantages of UVM. This is mainly due to the concept of classes and objects from SystemVerilog.

In UVM, all the above mentioned blocks are represented as objects that are derived from the already existing classes.

A class tree of the most important UVM classes can be seen in the fig.**??**.

The data that travels to and from the DUT is stored in *uvm_sequence_item* and *uvm_sequence*. The sequencer is derived from the *uvm_sequencer*, the driver is derived from the *uvm_driver* and so on.

UVM Phases

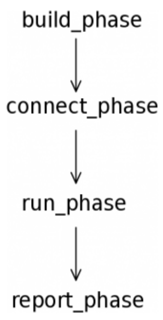

Figure 6: Instruction Register

All the above mentioned classes have simulation phases. Phases are ordered steps of execution implemented as methods. When we derive a new class, the simulation of our testbench goes through these different steps in order to construct, configure and connect the testbench.

- Build Phase:
  The build phase is used to construct components of the hierarchy. For example, the build phase of the agent class will construct the classes for the monitor, for the sequencer and for the driver.
- Connect Phase:
  The connect is used to connect the different sub components of a class. Using the same example, the connect phase of the agent connects the driver to the sequencer and the monitor is connected to an external port.
- Run Phase:
  The run phase is the main phase of the execution. This is where the actual code of a simulation will execute.
- Report Phase:
  Finally, the report phase is the phase where the results of the simulation are displayed.

UVM Macros

Macros are an important aspect of UVM. These macros implement some useful methods in classes and in variables. Though they are optional to use, using them simplifies the process of code development and testing.

The most common ones are:

- ‘uvm_component_utils
  This macro registers the new class type. It is used when deriving new classes like a new agent, driver, monitor and so on.
- ‘uvm_fie‘ld_init
  This macro registers a variable in the UVM factory and implements some functions like copy(), compare() and print().
- ‘uvm_info
  This is a very useful macro which we have used to print messages from the UVM environment during simulation time.

4.1 Device Under Test

The device under test here is a JTAG TAP controller. adopted from ...

5 Testing and Problems faced

5.1 Phase I: Basic communication

In this phase, the DUT that we use is a dummy DUT. Its only work is to print out the data as it receives it. The interface of the DUT is similar to that of an 8-bit memory block with data address and data registers. A transaction is defined and a sequence is passed to a sequencer. This sequence then calls on to the driver to access the DUT. The environment used to develop the code is EDAPlayground. EDAPlayground is an online simulator where there is no download required to run the code. For initial testing, this was easier to use as the samples codes are readily available. The work-flow is given below:

- Classes for sequence, sequencer and driver are written
- UVM builds, connects and runs these classes
- A transaction is created
- A randomized and constrained sequence (address and data) is sent to the DUT
- Sequencer sends the sequence to the Driver and waits for item_done signal from the driver
- Driver toggles these data on to the DUT through the interface and then sends item_done signal to the sequencer
- UVM reporting is activated as long as the system does not receive a reset signal
- Repeat sending sequences to the driver and observe the signals

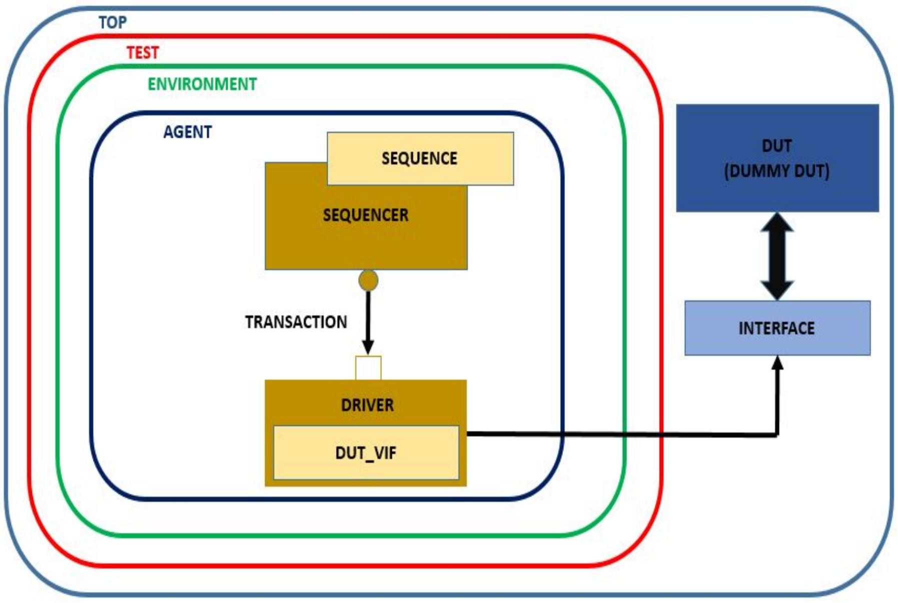

Figure 7: Basic Communication Environment

5.1.1 Testing Results for Phase I

The system reset is held low and normal operations are let to run. Randomized data and address is sent. From the waveform in Fig.??, we can see that these signals are toggling with the input.

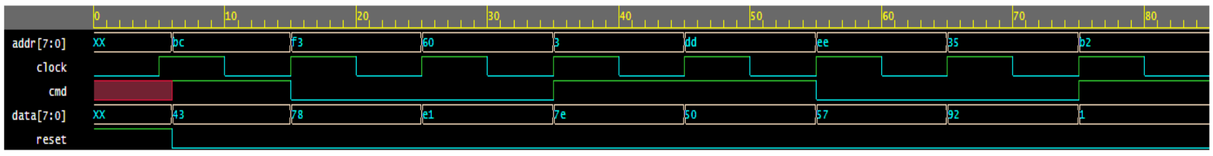

Figure 8: Phase I testing

The input parameters are randomized using the default *randomize()* command in SystemVerilog. The input to the DUT here is a 8-bit data bus and the constraint on the value of the input is from 0 to 255.

5.1.2 Testing Conclusion for Phase I

With this we have established a one-directional communication with our DUT. Also, the UVM Phases(build, connect, run and report) were tested. Our further developments have been built upon this basic building blocks. Here, we have not yet connected a monitor and scoreboard block. This would be included in phase III.

5.2 Phase II: Adding JTAG DUT

Phase 2 development was built over the existing architecture in Phase 1. Here we have added

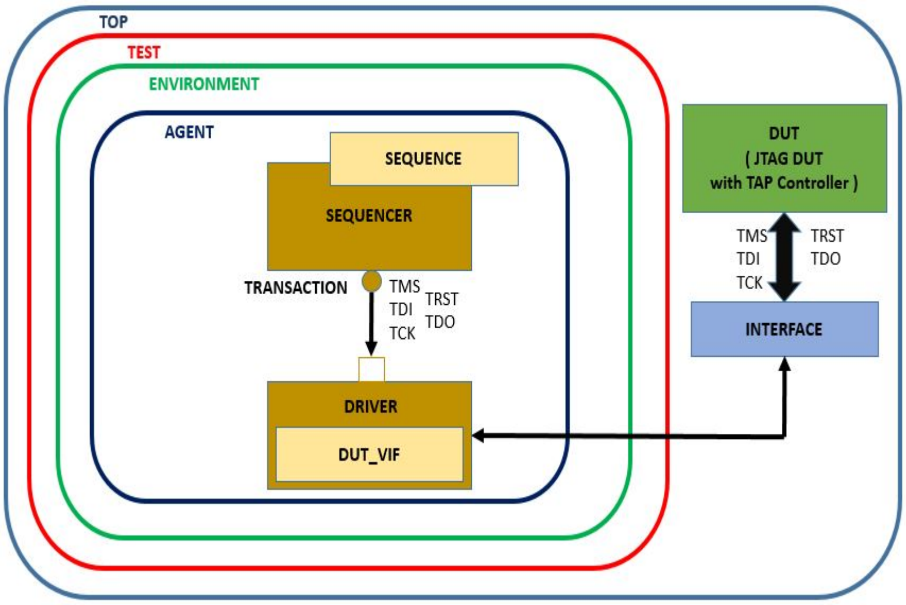

Figure 9: Environment after JTAG DUT addition

5.2.1 Testing Results for Phase II

5.2.2 Testing Conclusion for Phase II

5.3 Phase III: Adding Monitor and Scoreboard

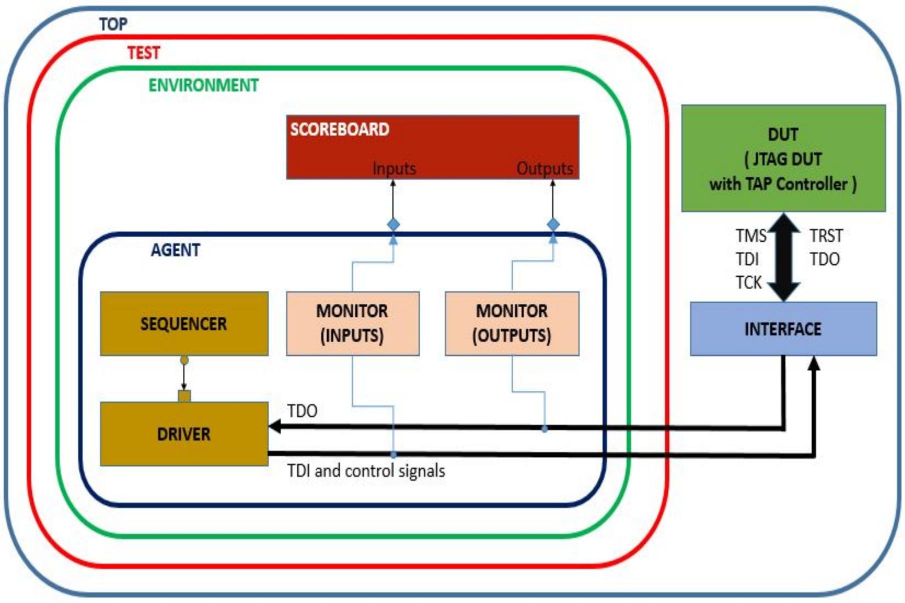

Figure 10: Complete Environment with Monitors and Scoreboard

5.3.1 Testing Results for Phase III

5.3.2 Testing Conclusion for Phase III

6 Results

6.1 Instruction: IDCODE

6.2 Instruction: BYPASS

6.3 Instruction: EXTEST

6.4 Instruction: INTEST

6.5 Instruction: SAMPLE/HOLD

References

[1] *The Test Access Port and Boundary-Scan Architecture*, Colin M Maunder, et al., IEEE Computer Society Press, Los Alamitos

[2] *IEEE Std 1149.1-1993*, IEEE Standard Test Access Port, and Boundary-Scan Architecture, IEEE, Inc., New York

[3] *The Boundary-Scan Handbook*, Kenneth P. Parker, Kluwer Academic Publishers, Norwell

[4] *High-Level Guide to JTAG*, `https://www.xjtag.com/about-jtag/jtag-high-level-guide/`

[5] *IEEE Standard 1149.1*, `https://www.microsemi.com/document-portal/doc_view/130050-ac160-ieee-standard-1149-1-jtag-in-the-sx-rtsx-sx-a-ex-rt54sx-s-families-app-not`

[6] *JTAG - General Description of the TAP Controller states*, `https://www.xilinx.com/support/answers/3203.html`

[7] *IEEE Standard Test Access Port and Boundary Scan Architecture*, IEEE-SA Standards Board, 14 June 2001

[8] *Coverage driven Verification Methodology*, `https://www.doulos.com/knowhow/sysverilog/uvm/easier_uvm_guidelines/coverage-driven/`

[9] *UVM Verification Primer*, John Aynsley `https://www.doulos.com/knowhow/sysverilog/uvm/tutorial_0/`

[10] *Accellera's UVM User's Guide 1.1*

[11] *Accellera's UVM 1.1 Class Reference*

[12] *Verification Academy's UVM Cookbook*

[13] *SystemVerilog for Verification: A Guide to Learning the TestBench Language Features*, Chris Spear

[14] *Comprehensive Functional Verification: The Complete Industry Cycle*, John Goss

[15] *UVM guide for Beginners* `https://www.colorlesscube.com/uvm-guide-for-beginners/`

[16] *EDA Playground* `https://www.edaplayground.com`