# UVM-Simulationsmodell eines JTAG-Interfaces

Serin J. Varghese

TECHNISCHE UNIVERSITÄT
CHEMNITZ

Fakultät für Elektrotechnik und
Informationstechnik

RESEARCH PROJECT REPORT

---

# UVM-Simulationsmodell eines JTAG-Interfaces

---

*Author*:
Serin J. VARGHESE
Immatrikulation Nr. 428459

*Supervising Professor*:
Prof. Dr.-Ing. habil. Göran HERRMANN

*Supervisors*:
Dipl.-Ing. Marcel PUTSCHE
Dipl.-Ing. Thomas HORN

Submitted to the Fakultät für Elektrotechnik und Informationstechnik
Professur Schaltkreis- und Systementwurf
in partial fulfillment of the requirements for the degree of
Master of Science in Micro and Nano Systems

Chemnitz, Germany
October, 2017

# Acknowledgement

I would first like to thank my thesis supervisor, Dipl.-Ing. Marcel Putsche, whose office was always open whenever I ran into a trouble spot or had a question about my research or writing. He consistently allowed this paper to be my own work, but steered me in the right direction whenever he thought I needed it.

I would also like to thank the experts who were involved in the validation survey for this research project: Prof. Dr.-Ing. habil. Göran Hermann and Dipl.-Ing. Thomas Horn. Without their passionate participation and input, the validation survey could not have been successfully conducted. I am thankful for them to have allowed me to have taken up this Research Project which I have found to be very rewarding.

I am grateful to my brother and my parents, who have provided me through moral and emotional support in my life. I am grateful to my other family members and friends who have supported me along the way. A special thanks to Ms. Priyanka Kamthe, who had to bear my frustration and listen to long talks on how my project was progressing. I thank all of them for their unwavering belief and confidence in me.

**Abstract**

A verification component is designed for a device with a JTAG interface. We have selected a full adder module with JTAG capability as our device under test. The instructions of Idcode, Bypass, Sample/Preload, Extest and Intest are implemented. All the implemented instructions are IEEE 1149.1 standard compliant. This verification component is designed with the use of the Universal Verification Methodology. Using the modules of the UVM environment, we have given the DUT a set of constrained stimulus and observed the response. The designed VC has the capability to introduce errors to understand how the VC would react to runtime errors. The errors, if any, are printed out on the console. This VC gives us an advantage of reusability wherein this full adder module can be replaced by any other module and the tests can be repeated with little effort. In this project we have designed the advanced DUTs with JTAG capability and verification environment, tested the working of the JTAG instructions and finally compared the expected data with the one that is actually observed.

1

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

# Contents

# 1 Abbreviations used

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

# 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.[4]

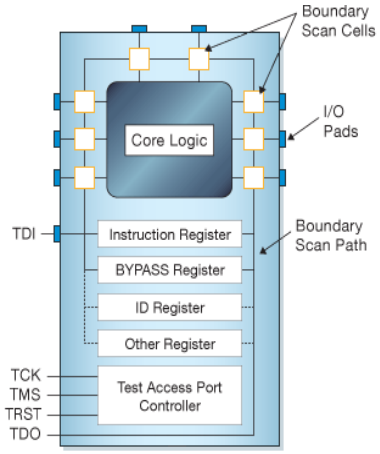


Figure 1: JTAG Block Diagram [18]

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.[4]

### 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 in Table 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.[5]

Table 1: Test Access Port Descriptions[5]

| 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.[5]

### 2.1.3 TAP Controller

The 16 states of the TAP controller finite state machine are shown in the Fig. 2. 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).[20]

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[20]

- **Test-Logic-Reset**
  All test logic is disabled in this controller state enabling the normal operation of the IC.

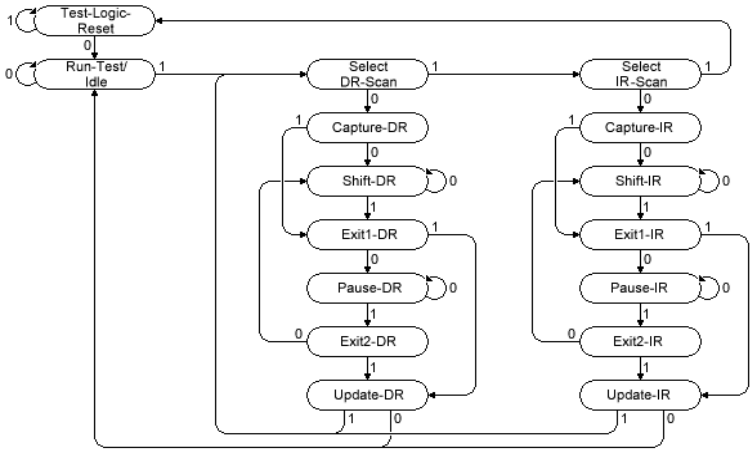


Figure 2: TAP CONTROLLER [19]

- **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.[20]

### 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.[5][21]

Table 2: Supported Instructions[5]

| 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.[5]

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.[5]

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.[5]

## 2.2 Introduction to Verification Methodologies:

### 2.2.1 Classical verification vs Constraint based verification:[8]

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.[8]

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.[8]

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.[8]

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.[8]

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.[8]

### 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.[8]

### 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.[9]

**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 SystemVerilog base class library, and a User Guide.[22]

**Checkers, Coverage and Constraints**
Constrained random verification relies on Checkers, Coverage and Constraints and these are supported by explicit features of the SystemVerilog language.[9]

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.[9]

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.[9]

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.[9]

**Verification Reuse**

UVM facilitates the construction 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.[9]

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.[9]

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.[9]

# 3 Literature Survey

**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*.[15]

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.

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.[15]

**UVM classes**

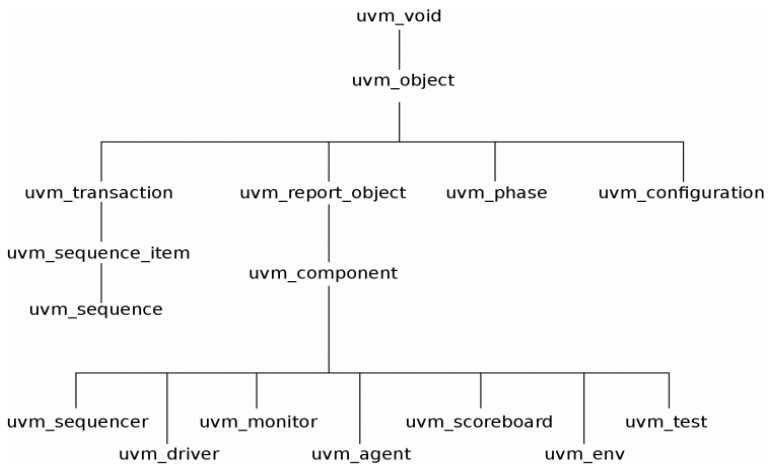


Figure 3: UVM Class Tree[15]

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.3.

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.[15]

**UVM Phases**[15]

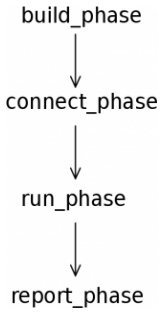


Figure 4: Instruction Register[15]

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.[15]

- 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.[15]

**UVM Macros**[15]

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.[15]

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_field_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.[15]

# 4 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

# 5 Developed Modules:

## 5.1 Device Under Test with JTAG capability

The designed DUT is JTAG compliant. The objective of our project is to develop a UVM Test for a JTAG interface. We would design a basic full adder with JTAG capabilities. The full adder module can be replaced with any other DUTs and the same testbench would implement the same JTAG tests accurately.

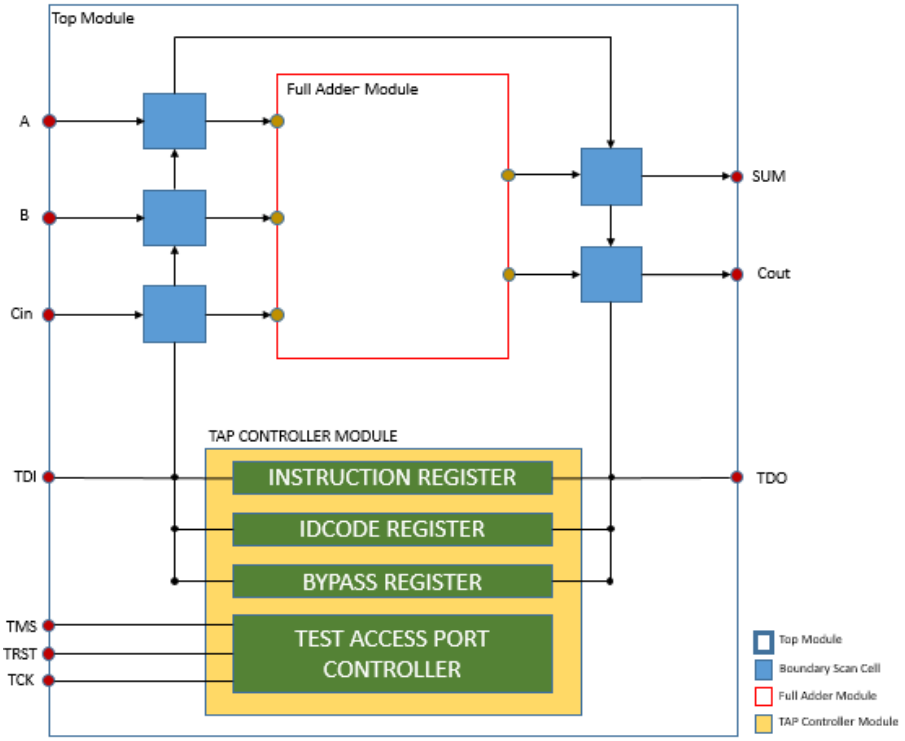


Figure 5: Developed Design Under Test

### 5.1.1 Boundary Scan Cells Modules

**Located in:** *InputCell.v* and *OutputCell.v*

The boundary-scan test architecture provides a means to test interconnects between integrated circuits on a board without using physical test probes. It adds a boundary-scan cell that includes a multiplexer and latches to each pin on the device. Boundary-scan cells in a device can capture data from pin or core logic signals, or force data onto pins. Captured data is serially shifted out and externally compared to the expected results. Forced test data is serially shifted into the boundary-scan cells. All of this is controlled from a serial data path called the scan path or scan chain. Figure 1 depicts the main elements of a boundary-scan cell. By allowing direct access to nets, boundary-scan eliminates the need for a large number of test vectors, which are normally needed to properly initialize sequential logic. Tens or hundreds of vectors may do the job that had previously required

thousands of vectors. Potential benefits realized from the use of boundary-scan are shorter test times, higher test coverage, increased diagnostic capability and lower capital equipment cost.

Data is passed serially through the Boundary Scan Registers which help in debugging the state of the inputs and the outputs.

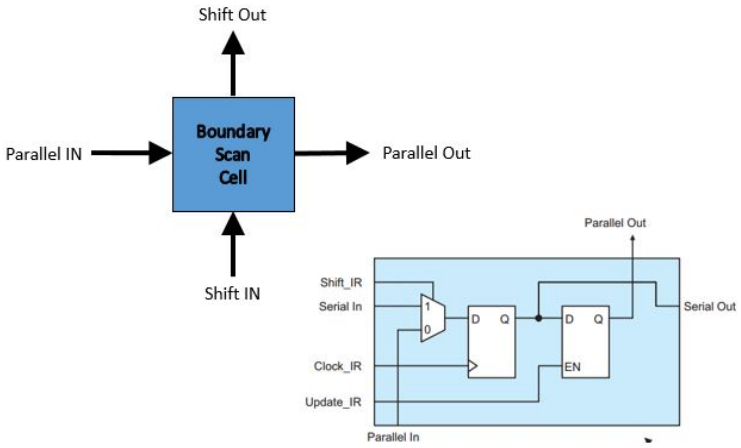


Figure 6: Boundary Scan Cell [17]

**Output Boundary Scan Cell:** This is the Boundary Scan Cell that is connected to the output side of the Full Adder Module. When the DUT is running in JTAG mode, then the TDI and TDO are connected between the boundary scan registers and this path is called the 'Scan Path'. The JTAG mode, the value in the instruction register decides the flow of information from the Scan Path. In normal operation mode, these boundary scan cells pass the Full Adder outputs to the outputs of the DUT.

```
module OutputCell( FromCore, FromPreviousBSCell, CaptureDR, ShiftDR,
UpdateDR, extest, TCK, ToNextBSCell, TristatedPin);
input  FromCore;
input  FromPreviousBSCell;
input  CaptureDR;
input  ShiftDR;
input  UpdateDR;
input  extest;
input  TCK;
output ToNextBSCell;
output TristatedPin;
}
```

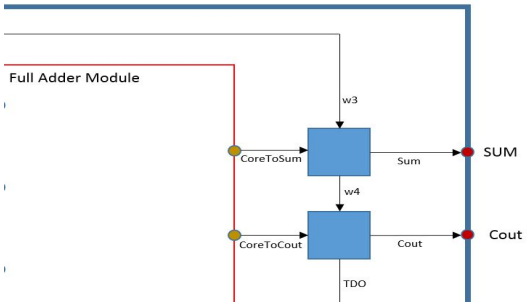


Figure 7: Output Section of the DUT

**Input Boundary Scan Cell:** This is the Boundary Scan Cell that is connected to the output side of the Full Adder Module. In normal operation mode, these boundary scan cells pass the input given to the DUT to the Full Adder input ports. The JTAG mode, the value in the instruction register decides the flow of information from the Scan Path.

```
1  module InputCell( InputPin, FromPreviousBSCell, CaptureDR, ShiftDR,
2  UpdateDR, TCK, ToNextBSCell, ToCore);
3  input  InputPin;
4  input  FromPreviousBSCell;
5  input  CaptureDR;
6  input  ShiftDR;
7  input  UpdateDR;
8  input  TCK;
9  output ToNextBSCell;
10 output ToCore;
```

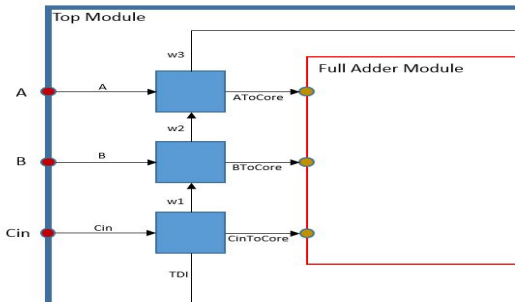


Figure 8: Input Section of the DUT

In addition to the parallel in, parallel out, serial in and serial out lines, the captureDR, ShiftDR, UpdateDR and TCK pins are also passed to this module from the TAP controller module. These signals indicate the Boundary Scan of the mode in which the DUT is operating in.

InputCell.v and OutputCell.v are the files that contain the input boundary scan cell and output boundary scan cell respectively.

### 5.1.2 TAP Controller Module

**Located in:** *tap_top.v*

The TAP controller module controls all the operation of the DUT when in the JTAG test operation mode. The TCK, TDI, TRST and TMS are the inputs to this module. The TDO pin is the output from this module. The TAP controller also contains the instruction register, the BYPASS register and the IDCODE register.

When the instruction register contains the instruction for IDCODE operation, the TDI and TDO are connected between the IDCODE registers. On every falling edge of the TCK signal, the IDCODE register is shifted out bit by bit to the TDO pin.

The BYPASS register is a one bit register that is connected between the TDI and TDO pin when the BYPASS instruction is selected. So the TDO follows the TDI with one clock cycle delay.

The operation of the TAP controller is controlled by the TMS pin. Fig. 2 shows how the TAP controller states change. The Finite-State Machine for the TAP controller has been implemented and tested.

### 5.1.3 Full Adder Module

**Located in:** *full_adder.v*

The full adder module implements the basic operations of the Full adder. It has three inputs and Sum and Carry outputs. This module is written in the full_ addder.v file.

Ports of the full adder:

- A - input
- B - input
- Cin - input
- Sum - output
- Cout - output

```
1  module full_adder(
2    input_a,
3    input_b,
4    input_cin,
5
6    output_sum_o,
7    output_cout_o
8  );
9
10 input input_a;
11 input input_b;
12 input input_cin;
13
14 output output_sum_o;
15 output output_cout_o;
16
```

```
17 reg output_sum;
18 reg output_cout;
19
20 assign output_sum_o = output_sum;
21 assign output_cout_o = output_cout;
22
23 always @(*) begin
24   {output_cout, output_sum} <= input_cin + input_a + input_b;
25 end
26
27 endmodule
```

## 5.2 UVM Modules

This contains all the modules that are required for the verification environment

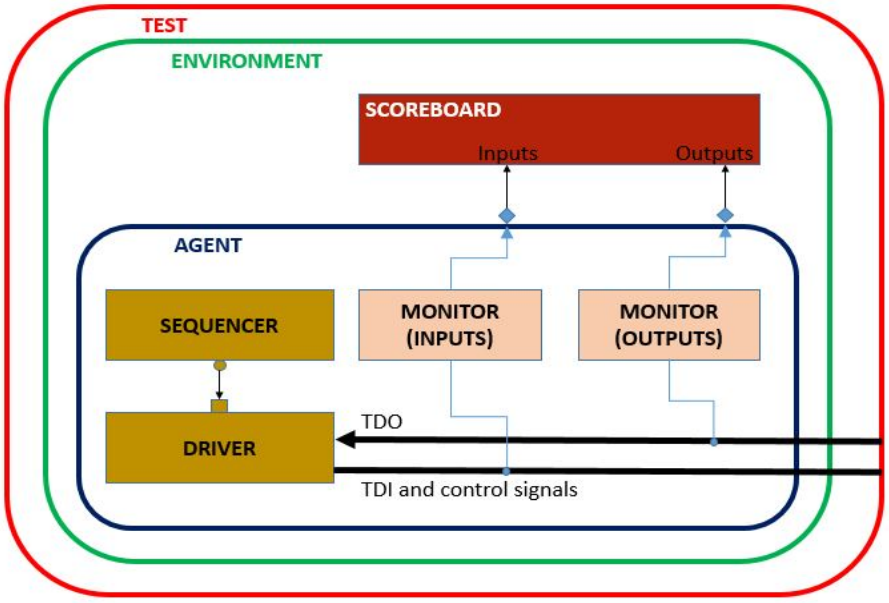


Figure 9: Designed Verification Environment

Each of these blocks are explained in detail below. It also contains actual snapshots of the code for easier understanding.

### 5.2.1 Top Block

**Located in:** *testbench.sv*

Generally the development of the DUT is done independently of the development of the testbench environment. The testbench top module connects the DUT to the verification components. A virtual interface is defined and added to the database so that all the modules that have access to the DUT can invoke it from the database.

```
module top;
  import uvm_pkg::*;
  import my_testbench_pkg::*;

  // Instantiate the interface
  dut_if dut_if1();

  // Instantiate the DUT and connect it to the interface
  dut dut1(.dif(dut_if1));

  // Clock generator
  initial begin
    dut_if1.TCK = 0;
    forever #5 dut_if1.TCK = ~dut_if1.TCK;
  end

  initial begin
    // Place the interface into the UVM configuration database
    uvm_config_db#(virtual dut_if)::set(null, "*", "dut_vif", dut_if1);
    // Start the test
    run_test("my_test");
  end

  // Dump waves
  initial begin
    $dumpfile("dump.vcd");
    $dumpvars(0, top);
  end

endmodule
```

The Top contains the following:

- DUT Instance
- Interface instance
- Clock Generator block
- start the test
- set config_db
- waveform dump logic

### 5.2.2 Test Block

**Located in:** *my_testbench_pkg.svh*

The test file is derived from the uvm_test class. The test defines the test scenario for the testbench. It contains the environment, configuration properties, class overrides etc. A sequence can also be connected from this block. This test block runs when the run_test() function is called. Here, the test also defines the environment.

```

```

```
class my_test extends uvm_test;
  `uvm_component_utils(my_test)

  my_env env;

  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction

  function void build_phase(uvm_phase phase);
    env = my_env::type_id::create("env", this);
  endfunction

  task run_phase(uvm_phase phase);
    // We raise objection to keep the test from completing
    phase.raise_objection(this);
    #10;
    `uvm_warning("", "Task Started! Ready for Lift-off!")
    // We drop objection to allow the test to complete
    phase.drop_objection(this);
  endtask
endclass: my_test
```

### 5.2.3 Environment Block

**Located in:** *my_testbench_pkg.svh*

The environment is derived from the uvm_env class. The environment defines the Agent and the Scoreboard in the build phase. In the connect phase, the outputs from the monitor are connected to the scoreboard.

```
class my_env extends uvm_env;
  `uvm_component_utils(my_env)

  my_agent agent;
  jtag_scoreboard mem_scb;

  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction

  function void build_phase(uvm_phase phase);
    agent   = my_agent::type_id::create("agent", this);
    mem_scb = jtag_scoreboard::type_id::create("mem_scb", this);
  endfunction

  function void connect_phase(uvm_phase phase);
    agent.mon_before.mon_ap_before.connect(mem_scb.sb_export_before);
    agent.mon_after.mon_ap_after.connect(mem_scb.sb_export_after);
  endfunction : connect_phase
```

```
endclass: my_env
```

### 5.2.4 Agent Block

**Located in:** *my_testbench_pkg.svh*

The build phase of the driver executes the following commands:

- Instantiate the ports that are used to connect to the monitor
- Instantiate the sequencer block
- Instantiate the agent block
- Instantiate the monitor block for reading TDI
- Instantiate the monitor block for reading TDO

The connect phase of the driver executes the following commands:

- The seq_item_port of the driver is connected to the seq_item_export of the sequencer
- The uvm_analysis_port of TDI monitor is connected to the uvm_analysis_port of the agent
- The uvm_analysis_port of TDO monitor is connected to the uvm_analysis_port of the agent

The agent also contains the following blocks:

- Sequence or Transaction Block
- Sequencer Block
- Driver Block
- Monitors Block

These blocks and their snapshots are explained more in detail further.

```
class my_agent extends uvm_agent;
  `uvm_component_utils(my_agent)

  uvm_analysis_port#(my_transaction) agent_ap_before;
  uvm_analysis_port#(my_transaction) agent_ap_after;

  jtag_monitor_before  mon_before;
  jtag_monitor_after   mon_after ;
  my_driver driver;
  uvm_sequencer#(my_transaction) sequencer;

  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction

  // Build Phase
  //The build phase instanciates all the modules built inside the agent
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);

    agent_ap_before = new("agent_ap_before", this);
```

```
    agent_ap_after  = new("agent_ap_after", this);;

    sequencer = uvm_sequencer#(my_transaction)::type_id::create("sequencer
       ", this);
    driver = my_driver ::type_id::create("driver", this);
    mon_before = jtag_monitor_before::type_id::create("mon_before", this);
    mon_after = jtag_monitor_after::type_id::create("mon_after", this);
  endfunction : build_phase

  // Connect Phase
  //interconnection between the modules
  function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    driver.seq_item_port.connect(sequencer.seq_item_export);
    mon_after.mon_ap_after.connect(agent_ap_after);
    mon_before.mon_ap_before.connect(agent_ap_before);
  endfunction

  // Run Phase
  task run_phase(uvm_phase phase);
    // We raise objection to keep the test from completing
    phase.raise_objection(this);
    begin
      my_sequence seq;
      seq = my_sequence::type_id::create("seq");
      seq.start(sequencer);
    end
    // We drop objection to allow the test to complete
    phase.drop_objection(this);
  endtask
endclass: my_agent
```

**Sequence or Transaction Block**
**Located in:** *my_sequence.svh*

The first step in testing our RTL design is to decide what kind of transaction is to be passed to the Driver. The transaction is designed by extending the uvm_ sequence_ item class. This includes the information needed to model the communication between the UVM components.

```
class my_transaction extends uvm_sequence_item;

  `uvm_object_utils(my_transaction)

  rand bit tms;
  rand bit tdi;
  rand bit trstn;
  rand bit tdo;
  rand bit A;
  rand bit B;
  rand bit Cin;
```

```
12
13    function new (string name = "");
14      super.new(name);
15    endfunction
16  endclass: my_transaction
```

**Sequencer Block**
**Located in:** *my_sequence.svh*

After a basic transaction has been defined, the verification environment will need to generate a collection of them and get them ready to be sent to the driver. This is the job of the sequencer. Sequencer is extended from the uvm_ sequence and its main job is to generate multiple transactions. Sequences are an ordered collection of transactions and they shape transactions to our needs and also generate as many as we need. We could also constrain the range of randomization to the valid range to reduce simulation time in invalid values. These transactions are then transferred to the driver module.

```
1   class my_sequence extends uvm_sequence#(my_transaction);
2
3     `uvm_object_utils(my_sequence)
4
5     function new (string name = "");
6       super.new(name);
7     endfunction
8
9     integer numberOfRequests = 0;
10
11    task body;
12      numberOfRequests = 150 + `DATA_LENGTH;
13      repeat(numberOfRequests)
14      begin
15        req = my_transaction::type_id::create("req");
16
17        start_item(req);
18        if(!req.randomize())
19        begin
20          `uvm_warning("", "Randomization failed!")
21        end
22        // Waiting for the driver to send the item_done() command
23        finish_item(req);
24      end
25    endtask: body
26  endclass: my_sequence
```

**Driver Block**
**Located in:** *my_sequence.svh*

The role of the driver block is to directly interact with the DUT. The driver pulls transactions from the sequencer and sends them repetitively to the signal-level interface. This interaction will be observed and evaluated by another block, the monitor. The driver toggles the TMS and the TDI pins to traverse through the TAP controller. The values of the TDI are shifted into the IR or the DR register depending on the state of the TAP controller. The driver module is extended from the uvm_ driver class. The run phase of the driver does the following:

- Gets a sequence item from sequencer
- Drive the sequence item to the DUT
- Wait for a possible few clock cycles for the DUT to respond
- Tell the sequencer that the current process is complete
- Ask the sequencer to send the next sequence item

The config_db places the defined virtual interface in the database so that it can be accessed by the driver module. Using the similar process, the interface can be loaded into any block which accesses the DUT directly.

**UVM Driver Methods:**

**get_next_item**

This method blocks the driver till a sequence_item is available at the sequencer

**item_done**

The non-blocking item_done()method will return a null pointer if there is no sequence_item available in the sequencer.

**Tests implemented in the driver:**

- Bypass Instruction
- Idcode Instruction
- Intest Instruction
- Sample/Preload Instruction

```
1  class my_driver extends uvm_driver #(my_transaction);
2    `uvm_component_utils(my_driver)
3    virtual dut_if dut_vif;
4
5    function new(string name, uvm_component parent);
6      super.new(name, parent);
7    endfunction
8
9    function void build_phase(uvm_phase phase);
10     // Get interface reference from config database
11     if(!uvm_config_db#(virtual dut_if)::get(this, "", "dut_vif", dut_vif))
12     begin
13       `uvm_error("", "uvm_config_db::get failed")
14     end
15     else
```

```
    begin
      `uvm_warning("", "Configuration database successfully accessed!")
    end
  endfunction

  task run_phase(uvm_phase phase);(...)
  endtask

  virtual function void compareForBypass();(...)
  endfunction: compareForBypass

  virtual function void compareForIdcode();(...)
  endfunction: compareForIdcode

  virtual function void printForExtestFullAdder();(...)
  endfunction: printForExtestFullAdder

  virtual function void printSamplePreload();(...)
  endfunction: printSamplePreload

  virtual function void printIntest();(...)
  endfunction : printIntest

  virtual function void printExtest();(...)
  endfunction

endclass: my_driver
```

**Monitors Block**

**Located in:** *my_sequence.svh*

The monitor is derived from the uvm_monitor. Monitor is a passive block that observes the communication of the DUT with the verification environment. The monitor also returns an error if the response of the DUT does not match with the expected results. It is passive because it does not drive any signals to the DUT. The monitor samples the DUT signals through the virtual interface and converts the signal level activity to transaction level activity.

Monitor uses TLM ports to point to the DUT signals. There are two monitors that have been defined in out verification environment. One monitor is used to sample the inputs that are driven from the driver to the DUT( TDI ). The second monitor samples the response of the DUT (TDO) and converts in into transaction level activity. All these are written to the scoreboard.

```
class jtag_monitor_before extends uvm_monitor;
  `uvm_component_utils(jtag_monitor_before)

  uvm_analysis_port#(my_transaction) mon_ap_before;

  virtual dut_if dut_vif;

  reg   [ 1  :  0  ] clock_value ;
  integer tdiScan =0;
```

```
10
11    function new(string name, uvm_component parent);
12      super.new(name, parent);
13    endfunction: new
14
15    function void build_phase(uvm_phase phase);
16      super.build_phase(phase);
17      mon_ap_before = new("mon_ap_before", this);
18      if (! uvm_config_db #(virtual dut_if) :: get (this, "", "dut_vif",
          dut_vif)) begin
19        `uvm_error (get_type_name (), "DUT interface not found")
20      end
21    endfunction: build_phase
22
23    task run_phase(uvm_phase phase);
24      my_transaction sa_tx;
25      sa_tx = my_transaction::type_id::create(.name("sa_tx"), .contxt(
          get_full_name()));
26
27      //Writing the data at every toggling of the TDI pin
28      forever begin
29        @(posedge dut_vif.TCK)
30        begin
31          if(startValiadation_bypass)
32          begin
33            sa_tx.tdi = dut_vif.TDI;
34            // This instruction writes the data to the scoreboard
35            mon_ap_before.write(sa_tx);
36            validationBufferTDI_bypass  [  tdiScan  ]=dut_vif.TDI;
37            tdiScan++;
38          end
39        end
40      end
41    endtask: run_phase
42  endclass: jtag_monitor_before
```

```
1   class jtag_monitor_after extends uvm_monitor;
2     `uvm_component_utils(jtag_monitor_after)
3
4     uvm_analysis_port#(my_transaction) mon_ap_after;
5
6     virtual dut_if dut_vif;
7
8     reg   [  1  :  0  ] clock_value ;
9     integer tdoScan_bypass =0;
10    integer tdoScan_idcode =0;
11    integer tdoScan_intest =0;
12    integer tdoScan_extest =0;
13
14
15    function new(string name, uvm_component parent);
```

```
16      super.new(name, parent);
17    endfunction: new
18
19    function void build_phase(uvm_phase phase);
20      super.build_phase(phase);
21      mon_ap_after = new(.name("mon_ap_before"), .parent(this));
22      if (! uvm_config_db #(virtual dut_if) :: get (this, "", "dut_vif",
          dut_vif))
23      begin
24        `uvm_error (get_type_name (), "DUT␣interface␣not␣found")
25      end
26    endfunction: build_phase
27
28    task run_phase(uvm_phase phase);
29      my_transaction sa_tx_after;
30      sa_tx_after = my_transaction::type_id::create(.name("sa_tx_after"), .
          contxt(get_full_name()));
31
32      forever begin
33        @(negedge dut_vif.TCK)
34        begin
35          if(startValiadation_bypass == 1)
36          begin
37            sa_tx_after.tdo = dut_vif.TDO;
38            mon_ap_after.write(sa_tx_after);
39            validationBufferTDO_bypass  [  tdoScan_bypass  ]=dut_vif.TDO;
40            tdoScan_bypass++;
41          end
42
43          if(startValiadation_idcode == 1)
44          begin
45            sa_tx_after.tdo = dut_vif.TDO;
46            mon_ap_after.write(sa_tx_after);
47            validationBufferTDO_idcode  [  tdoScan_idcode  ]=dut_vif.TDO;
48            tdoScan_idcode++;
49          end
50
51          if(startValiadation_intest == 1)
52          begin
53            sa_tx_after.tdo = dut_vif.TDO;
54            mon_ap_after.write(sa_tx_after);
55            validationBufferTDO_intest  [  tdoScan_intest  ]=dut_vif.TDO;
56            tdoScan_intest++;
57          end
58        end
59      end
60    endtask: run_phase
61  endclass: jtag_monitor_after
```

### 5.2.5 Scoreboard Block

**Located in:** *my_sequence.svh*

The scoreboard module is extended from the uvm_scoreboard class. uvm_scoreboard is inherited by uvm_component. The signals from the monitors are connected to the scoreboard.

```
class jtag_scoreboard extends uvm_scoreboard;
  `uvm_component_utils(jtag_scoreboard)

  uvm_analysis_export #(my_transaction) sb_export_before;
  uvm_analysis_export #(my_transaction) sb_export_after;

  uvm_tlm_analysis_fifo #(my_transaction) before_fifo;
  uvm_tlm_analysis_fifo #(my_transaction) after_fifo;

  my_transaction transaction_before;
  my_transaction transaction_after;

  function new(string name, uvm_component parent);
    super.new(name, parent);
    transaction_before = new("transaction_before");
    transaction_after = new("transaction_after");
  endfunction: new

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    sb_export_before = new("sb_export_before", this);
    sb_export_after  = new("sb_export_after",  this);

    before_fifo = new("before_fifo", this);
    after_fifo  = new("after_fifo",  this);
  endfunction: build_phase

  function void connect_phase(uvm_phase phase);
    sb_export_before.connect(before_fifo.analysis_export);
    sb_export_after.connect(after_fifo.analysis_export);
  endfunction: connect_phase

  task run();
    forever begin
      before_fifo.get(transaction_before);
      after_fifo.get(transaction_after);
    end
  endtask: run
endclass: jtag_scoreboard
```

# 6 Testing and Debugging Phases

## 6.1 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 complete environment is shown in Fig. 10. 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

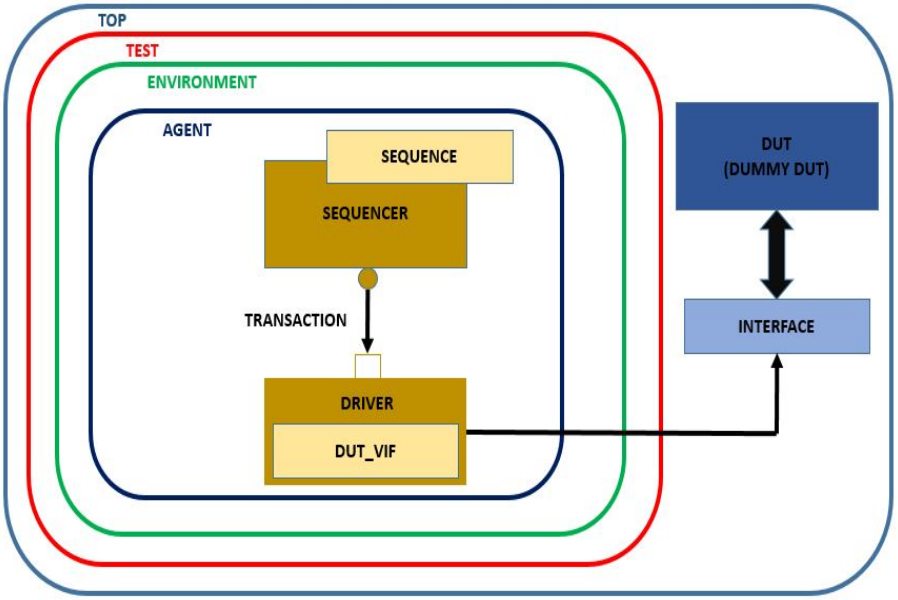


Figure 10: Phase 1 Testing Environment
Here the UVM environment is connected to the designed DUT

### 6.1.1 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.11, we can see that these signals are toggling with the input. The

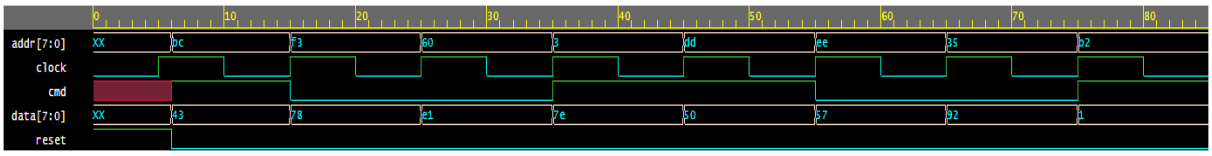


Figure 11: Phase I testing

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.

### 6.1.2 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.

## 6.2 Phase II: Adding TAP controller DUT

Phase 2 development was built over the existing architecture in Phase 1. Here we have replaced the dummy DUT with a DUT that has a TAP controller inbuilt into it. The development environment was now changed from the EDAPlayground to the QuestaSim IDE. The codes from the EDAPlayground examples however do not run as it is on QuestaSim. The environment variables for the UVM library have to be properly defined and we have to take care of the ìncludes in our program. *import uvm_pkg::*;* has be included as can be seen from the files in the project. The representation of the environment is shown in Fig. 12.

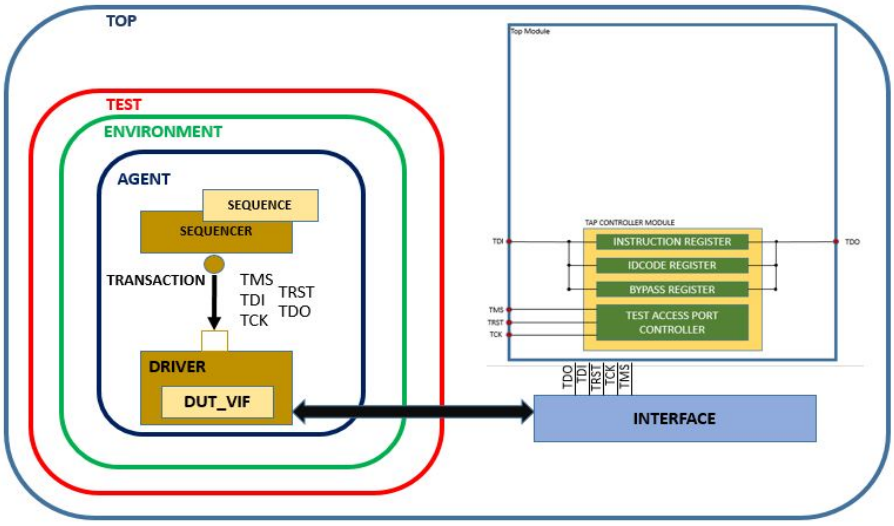


Figure 12: Phase II Environment: TAP Controller DUT added

### 6.2.1 Testing Results for Phase II

**BYPASS Instruction test**

Steps in which the BYPASS Instruction is executed:

- Move through the TAP controller states and reach the SHIFT IR state.
- Shift-in the BYPASS instruction(4'b1111) into the Instruction register. This means that the TAP controller remains at the SHIFT IR for 4 clock cycles. The value of TMS will be held to 0. TDI will be held to 1 to send in 4'b1111 into the IR. This will inform the TAP controller that the BYPASS instruction is to be executed
- Move out of the Instruction Register state and navigate through the TAP controller states to reach the SHIFT DR state.
- Now for every clock cycle, TDO will follow the data from TDI with a delay of one clock cycle. This one clock cycle delay is because of the addition of 1-bit Bypass register in the chain between TDI and TDO. In the VC we pass random signals to TDI and compare the same with the signals observed at TDO.

The selection of the Bypass instruction is done by declaring *'define BYPASS_INSTR*.
Note: Do not run two tests simultaneously.
The tests for the bypass instruction are added in the run phase of the driver. The comments in the code mention exactly what each step does. Variable data stream length can be defined. A global variable *DATA_LENGTH* is defined in the *my_sequence.svh* file. The value will correspond to the length of the data stream that is sent from the TDI to the TDO.

The values of the TDI and TDO are stored by the monitor and then written into the scoreboard. The function *compareForBypass()* compares the TDI and TDO values and the result is printed in the console of the QuestaSim during simulation.

To introduce error into the bypass instruction the variable *introduceErrorBypass* should be set to 1. This introduces a stuck-at-1 error at TDO pin. The function *compareForBypass()* compares the two data streams and gives out an error on the console as the TDO and the TDI values do not match.

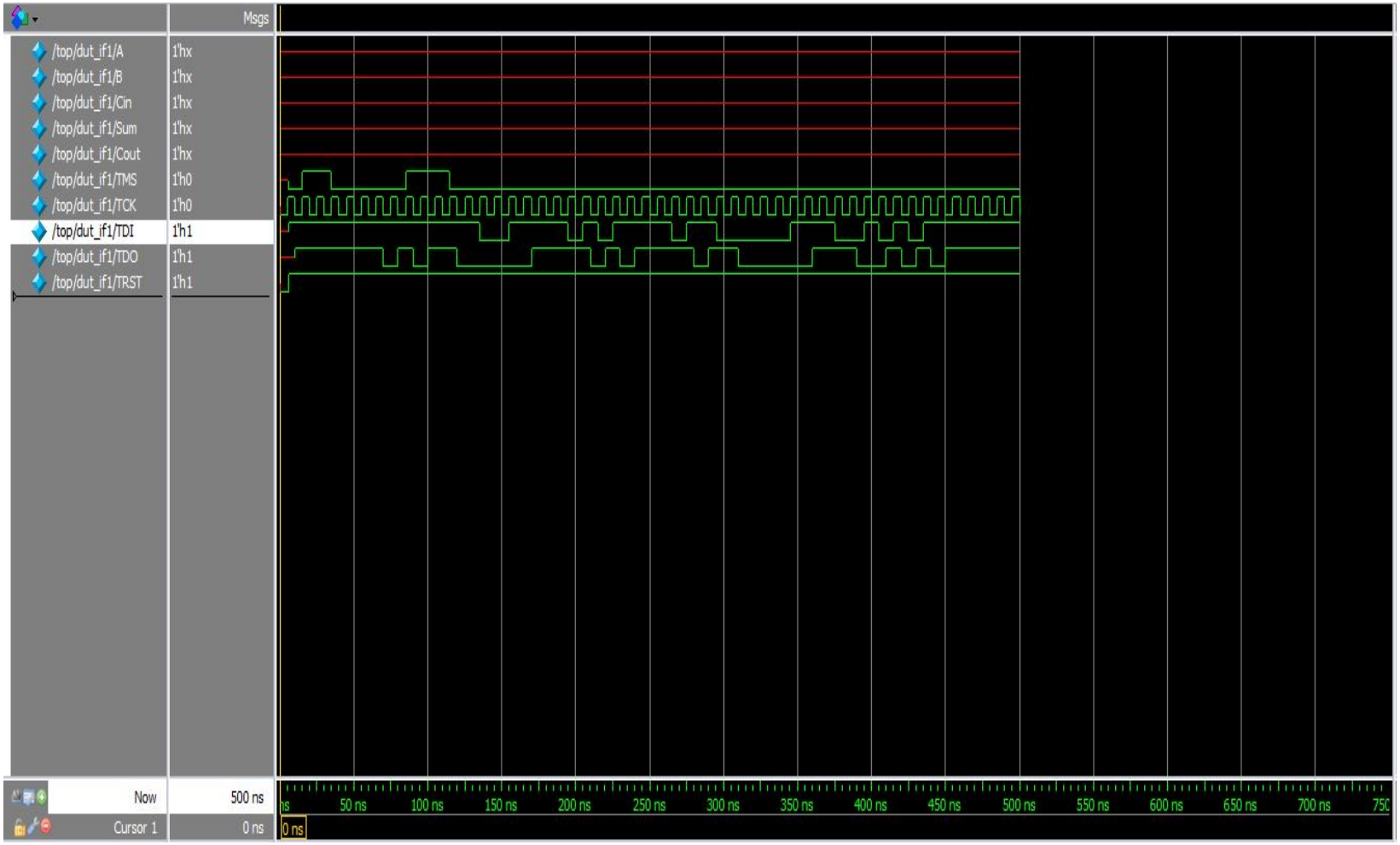


Figure 13: BYPASS testing.
The TDO follows the TDI with a one clock cycle delay

**IDCODE Instruction test**
Steps in which the IDCODE Instruction is executed:

- Move through the TAP controller states and reach the SHIFT IR state.
- Shift-in the IDCODE instruction(4'b0010) into the Instruction register. This means that the TAP controller remains at the SHIFT IR for 4 clock cycles. The value of TMS will be held to 0. TDI will be 0 for the 1st clock cycle. 1 for the next clock cycle and 0 for the 3rd and the 4th clock cycle. This will inform the TAP controller that the IDCODE instruction is to be executed
- Move out of the Instruction Register state and navigate through the TAP controller states to reach the SHIFT DR state.
- The IDCODE register is a 32-bit register which is defined in a TAP controller. This contains a unique ID that used to identify it. Now for every clock cycle in the SHIFT DR state, this IDCODE register is connected between the TDO and the TDI. The IDCODE is shifted out serially.

The selection of the Idcode instruction is done by declaring *'define IDCODE_INSTR.*
The tests for the IDCODE are also defined in the run phase of the driver. The comments in the code mention exactly what each step does. Here, variable data streams cannot be defined as the length of the IDCODE register is defined in the IEEE Standard 1149.1

The values of the TDI and TDO are stored by the monitor and then written into the scoreboard. The function *compareForIdcode()* compares the TDO values with the values of the IDCODE defined in the DUT . The result is printed in the console of the QuestaSim during simulation. Incase of an error, the IDCODE that is read by the verification environment is also printed out on the console.

To introduce error into the IDCODE instruction the variable *introduceErrorIdcode* should be set to 1. This introduces an error in the value that is read from the IDCODE register. The function *compareForIdcode()* compares the two data streams and gives out an error on the console as the TDO values and the IDCODE register values do not match. The waveforms are shown in Fig. 14

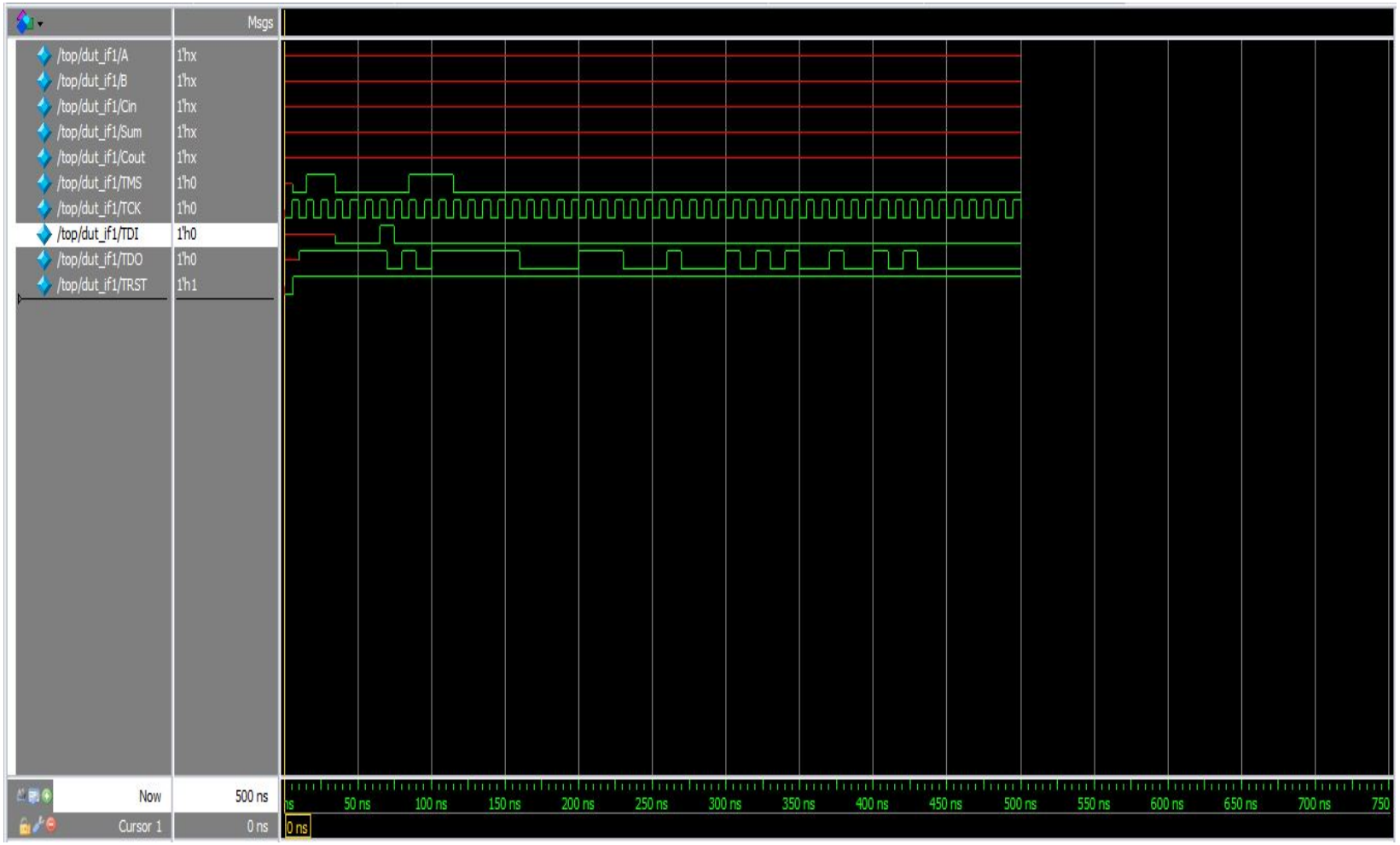


Figure 14: IDCODE testing
a. 32 bit IDCODE is shifted out to the TDO. This can be verified from the waveform.
b. The IDCODE of the DUT is h149511c3

### 6.2.2 Testing Conclusion for Phase II

The TAP controller states can now be navigated through easily. For the further development we have a concrete understanding of how the algorithm works. Also, we have tested the operations of the BYPASS and the IDCODE instructions. All the testing so far, however, has been done using the waveforms of the QuestaSim IDE. To improve the readability and usability of the testbench, we will be implementing monitors and scoreboards.

## 6.3 Phase III: Adding Monitor and Scoreboard

Two monitors are added to the UVM environment. The first is connected to the TDI input and the second is connected to the TDO output. These information are also written to the scoreboard. The changes that are made to the environment are as shown in Fig. 15.

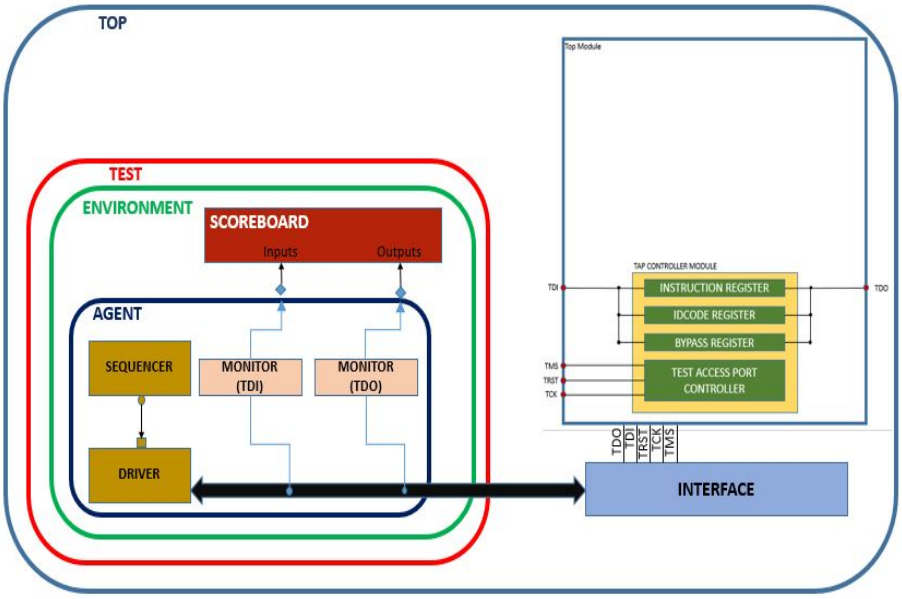


Figure 15: Phase III Environment: with Monitors and Scoreboard

### 6.3.1 Testing Results for Phase III

Functions are written that read the information that is received by both the monitors and compares the two. *compareForBypass() and compareForIdcode()* compares these information and invokes the uvm_error functions.

### 6.3.2 Testing Conclusion for Phase III

By this addition we have a complete testbench where the user does not have to physically check the waveforms for verification. The functions for comparison return the result on the console. The comparison starts when the *startValidation* flag is raised.

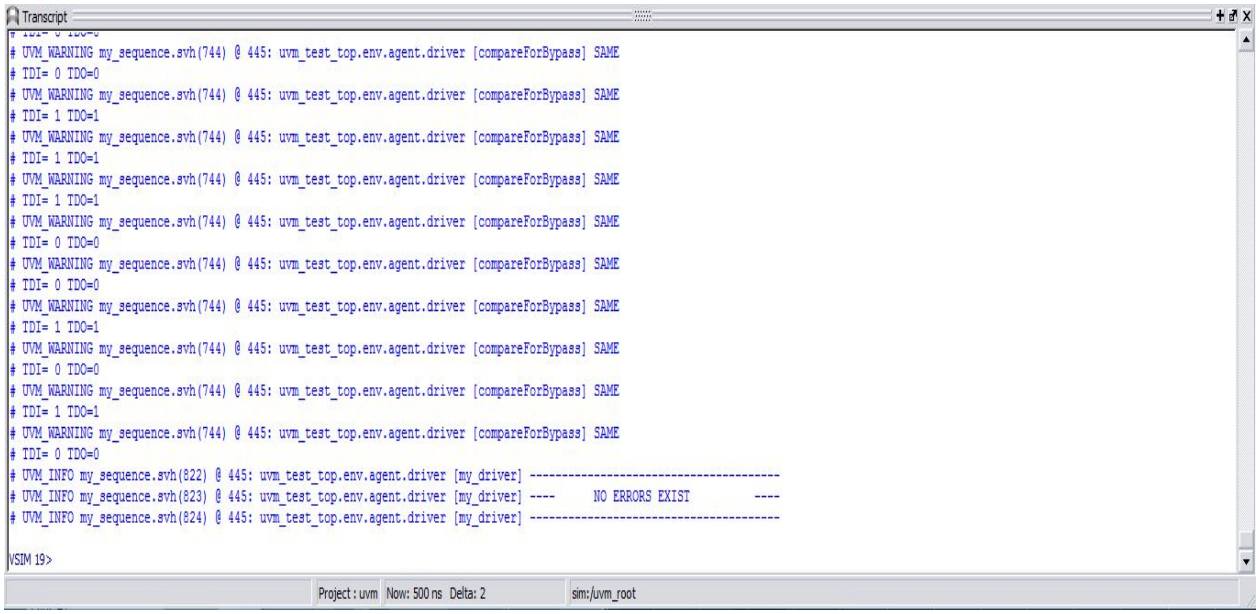


Figure 16: Complete Environment with Monitors and Scoreboard

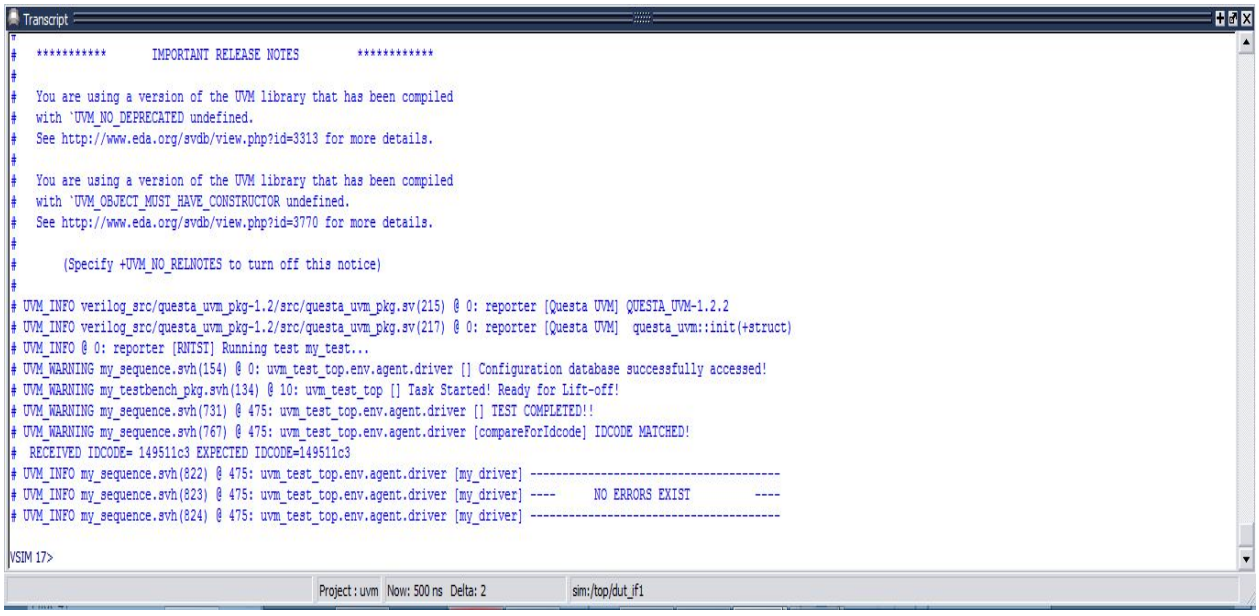


Figure 17: Complete Environment with Monitors and Scoreboard

## 6.4 Phase IV: Adding Boundary Scan registers

Boundary Scan Registers are connected to every input or output pin. These control the data traveling into and out of the pins. For testing the EXTEST, INTEST and the SAMPLE/PRELOAD, these BSRs are imperative. During normal operation, the data is bypassed from the BSR. During the SHIFT DR state, the all the BSRs are connected serially between the TDO and the TDI.

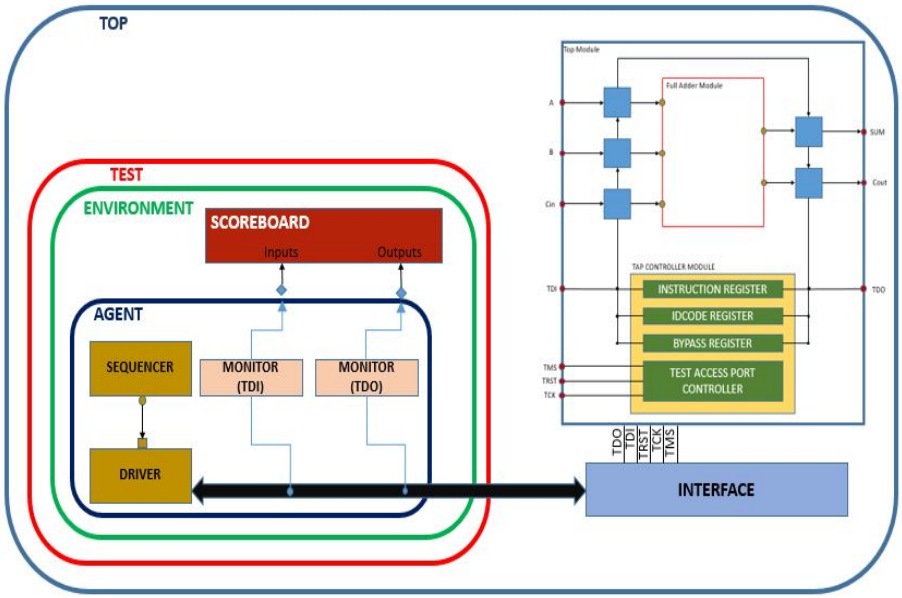


Figure 18: Complete Environment

### 6.4.1 Testing Results for Phase IV

**SAMPLE/PRELOAD Instruction test**

Steps in which the SAMPLE/PRELOAD Instruction is executed:

- Move through the TAP controller states and reach the SHIFT IR state.
- Shift-in the SAMPLE/PRELOAD instruction(4b0001) into the Instruction register. This means that the TAP controller remains at the SHIFT IR for 4 clock cycles. The value of TMS will be held to 0. TDI will be logic 1 for the 1st clock cycle and logic 0 for the next three clock cycles. This will inform the TAP controller that the SAMPLE/PRELOAD instruction is to be executed.
- Move out of the Instruction Register state and navigate through the TAP controller states to reach the SHIFT DR state.
- The Boundary scan cells are connected between the TDI and the TDO. The boundary scan cells are therefore filled-in(Preloaded) with the data stream from the TDI. At each clock cycle, the bits are shifted through.
- By accessing the data received on the TDO we read(sample) the data that was on the pins.

**EXTEST Instruction test**

Steps in which the EXTEST Instruction is executed:

- Move through the TAP controller states and reach the SHIFT IR state.
- Shift-in the EXTEST instruction(4'b0000) into the Instruction register. This means that the TAP controller remains at the SHIFT IR for 4 clock cycles. The value of TMS will be held to 0. TDI will be logic 0 for the 4 clock cycles of TCK. This will inform the TAP controller that the EXTEST instruction is to be executed.
- Move out of the Instruction Register state and navigate through the TAP controller states to reach the SHIFT DR state.
- The Boundary scan cells are connected between the TDI and the TDO. The boundary scan cells are therefore filled-in(Preloaded) with the data stream from the TDI. At each clock cycle, the bits are shifted through.
- By reading the data received on the TDO we read(sampled) the data that was on the pins.
- Move out of the Shift DR state and navigate through the TAP states and reach the SHIFT DR again.
- Shift out the Boundary Scan register data via TDO. The result verifies the working of the internal circuitry also.

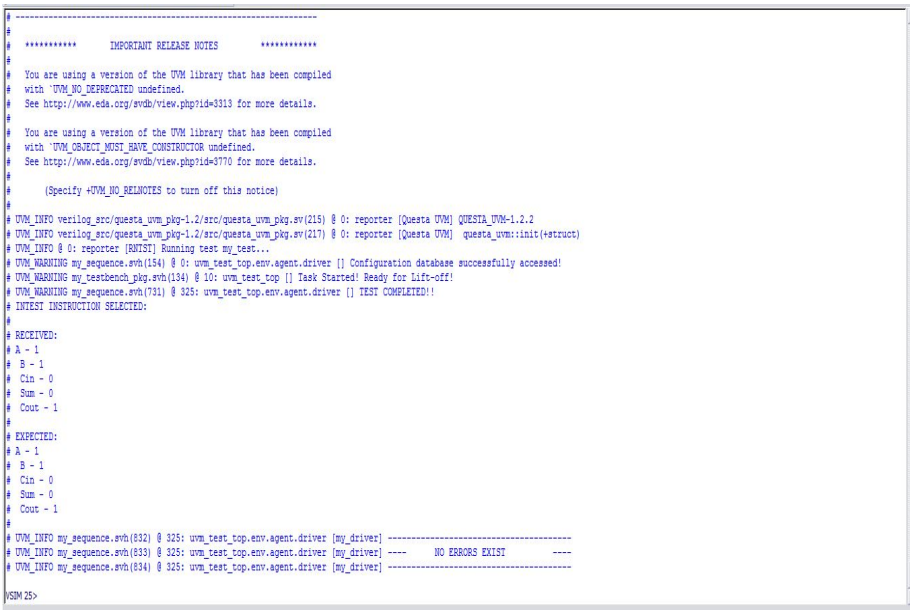


Figure 19: EXTEST Console Output

**INTEST Instruction test**

Steps in which the INTEST Instruction is executed:

- Move through the TAP controller states and reach the SHIFT IR state.
- Shift-in the INTEST instruction(4'b1001) into the Instruction register. This means that the TAP controller remains at the SHIFT IR for 4 clock cycles. The value of TMS will be held to 0. TDI will be logic 1 for the 1st clock cycle, logic 0 for the next two clock cycles and logic 1 for the 4th clock cycle. This will inform the TAP controller that the INTEST instruction is to be executed.

- Move out of the Instruction Register state and navigate through the TAP controller states to reach the SHIFT DR state.
- The Boundary scan cells are connected between the TDI and the TDO. The boundary scan cells are therefore filled-in(Preloaded) with the data stream from the TDI. At each clock cycle, the bits are shifted through.
- By reading the data received on the TDO we read(sampled) the data that was on the pins.
- Reach the UpdateDR state. In this state, the BSR passes the data stored in the BSR to the internal circuitry.

### 6.4.2 Testing Conclusion for Phase IV

The working of the boundary scan registers is tested. The BSR is successful in both parallel loading and serial transmission of the data. The Sample/Preload, Extest and the Intest operations, test all the conditions of operation of the BSRs.

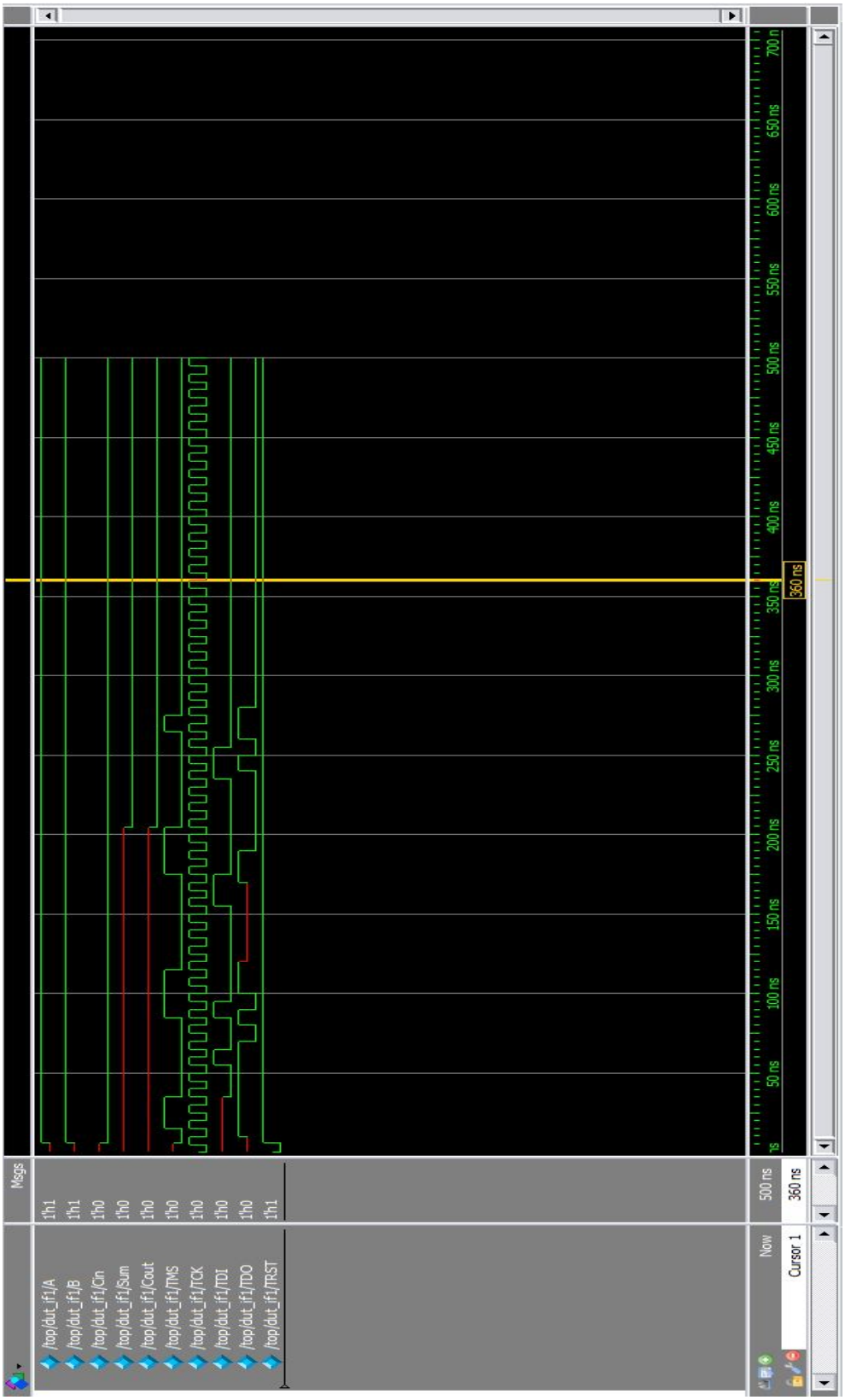


Figure 20: EXTEST testing

# 7 Conclusion

We have now implemented a Device under test with JTAG capabilities. This can now be further extended to any DUT that is designed by replacing the *full_adder.v* file with any file that has the DUT to be tested. Only *top_module.v* needs to be modified for the instantiation of the modules. *config.svh* file has been added. The user can change the configuration of the test by changing the values in the configuration file. This helps us isolate the DUT from the JTAG-interface. This makes the VC generic and can be easily replaced with any other DUT. The UVM environment would remain the same and we could test the complete functionality of the JTAG instructions.

The tests have been carried out to see the responses of the DUT to the BYPASS, IDCODE, SAMPLE/PRELOAD, INTEST and EXTEST instructions. The UVM environment compares the received and the expected values and gives an output on the console of the IDE. The user does not have to use the waveforms to manually verify the integrity of the signals. This reduces the time and effort that goes into testing of modules.

Provision has also been made to intentionally introduce errors into the testing modes. This gives us an idea of how the response would be when there is something wrong with the DUT. Also, this functionality helps cross verify our verification environment.

## 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

[17] *corelis.com* http://www.corelis.com/education/Boundary-Scan_Tutorial.htm

[18] *A technical overview of JTAG* https://www.xjtag.com/about-jtag/jtag-a-technical-overview/

[19] *How JTAG works* http://www.fpga4fun.com/JTAG2.html

[20] *JTAG - General description of the TAP Controller states* http://www.cnblogs.com/shangdawei/p/4753689.html

[21] *Analog and Mixed-Signal Boundary-Scan: A Guide to the IEEE 1149.4 Test Standard*, Adam Osseiran

[22] *Universal Verification Methodology* `https://en.wikipedia.org/wiki/Universal_Verification_Methodology`