UVM Based Verification of CAN Protocol Controller Using System Verilog

Suchika Lalit
P.G Students, Department of Electronics Engineering
Gujarat Technological University
Ahmedabad, Gujarat, India
suchikalalit@gmail.com

Mr. Ashish Prabhu
Sr. Verification Engineer at LSI Pune, India
p_ashishp@yahoo.com

***Abstract*—**Over the years, design complexity and size have stubbornly obeyed the growth curve predicted by Gordon Moore. The industry is migrating towards leading edge nodes, which can hold more than 100 Million gates. The chip makers want to pack as many functions possible in their SoCs and provide as many feature additions to gain market share. And, of course, all of those features need to be verified. Verification is currently the largest challenge facing the semiconductor industry in keeping pace with both the customer demand for features and our technical ability to add millions of gates to our chips. Verification quality is a must for functional safety in electronic systems. This paper describes the verification of CAN Protocol Controller using System Verilog. The CAN Controller functions as the interface between an application and the actual CAN bus. Taking this need in consideration, this paper describes flow from specification extraction to development of verification environment.

Keywords- *UVM, ASIC, VLSI, CAN, DUT*

*****

I. INTRODUCTION

During the last decades, several verification methodologies have been developed to ease the process of ASIC verification designs. EDA tool vendors usually develop these methodologies which in most cases are not compatible with tools from different vendors [4]. With the introduction of the Open Verification Methodology (OVM) which supports the use of SystemVerilog testbenches, need for verification became more standardized and hence, OVM paved way for Universal Verification Methodology (UVM) which has become an official Accellera standard supported by all EDA tool vendors today[4].

This research presents UVM based Verification process and methodology using SystemVerilog, explains verification strategy and reuse of design environment with reference to verifying the CAN Protocol controller (IP) core. Communication across a CAN bus starts with the application providing the CAN controller with the data to be transmitted. The CAN controller provides an interface between the application and the CAN bus. The function of the CAN controller is to convert the data provided by the application into a CAN message frame fit to be transmitted across the bus. A transceiver receives the serial input stream from the controller and converts it into a differential signal. The Physical connection of the CAN controller to the CAN bus is done with the CAN transceiver.

The Universal Verification Methodology (UVM) offers the most excellent structure to attain coverage driven verification. The coverage driven verification combines automatic test generation, self-checking testbenches and coverage metrics to significantly reduce the time spent verifying a Design Under Test (DUT).

II. VERIFICATION

In VLSI (Very Large Scale Integration) technology we design and make integrated chips. ASIC (Application Specific Integrated Chip) designing is a process in which RTL (Register Transfer Level) design is made using Hardware Description Language (HDL). Based on correct RTL respective chip is manufactured. If RTL contains errors or bugs the final chip does not work properly according to specified functionality. To make sure that RTL is working correctly according to specified functionality, verification is required. According to Moore's law number of transistor increases in the design every 18 months. As the number of transistors in the design increases so the errors in the design increases. Thus verification is one of the most important processes of ASIC flow which make sure the functional correctness of the design.

"Verification is a process used to demonstrate the functional correctness of a design in its implementation" [8].At every step of developing a chip we need verification. At each level we need some
level of verification. Basically verification covers the below things.

- What we specified is what we envisioned.
- What we design is what we specified.

- What we taped out is what the RTL described.
- What we manufactured is what we taped out.

Bug is error or misbehaving of design. It is unexpected behaviour of the design. No design in VLSI is bug free. Bugs found in early stages of verification costs very little. It is always best to find bugs in the design as early as possible. For complex designs synthesis takes lots of time. If we find bugs in later stage it costs more.

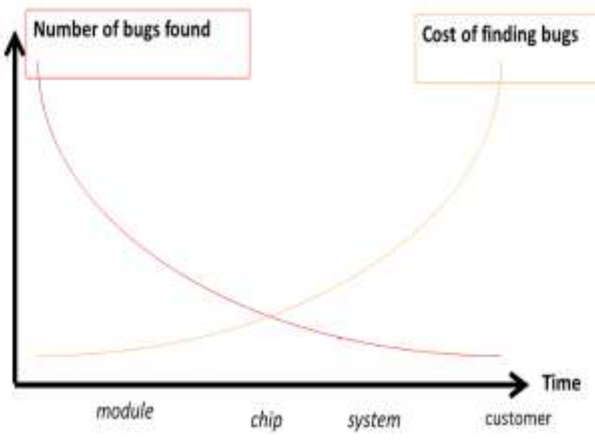


Figure 1 Number of bugs' vs time in verification flow

III. UVM

UVM (Universal Verification Methodology) was introduced in December 2009, by a technical subcommittee of Accellera. UVM uses Open Verification Methodology as its foundation. Accellera released version UVM 1.0 EA on May 17, 2010. UVM Class Library provides the building blocks needed to quickly develop well-constructed and reusable verification components and test environments. It uses system Verilog as its language. All three of the simulation vendors (Synopsys, Cadence and Mentor) support UVM today which was not the case with other verification methodology.

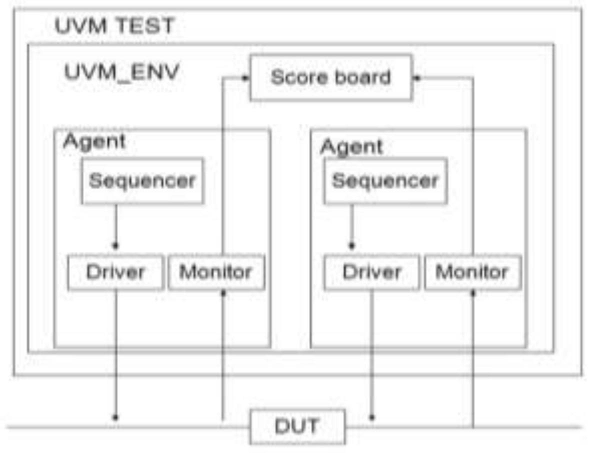


Figure 2 UVM Basic Component Model.

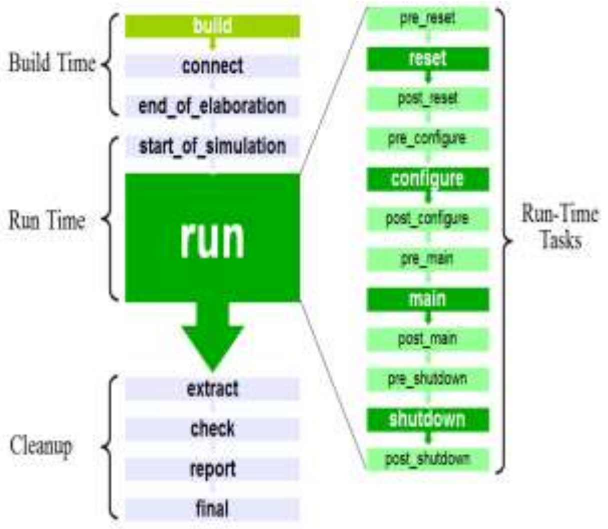


Figure 3UVM phases

IV. CAN PROTOCOL CONTROLLER DESIGN

The interface between the CAN serial bus and CAN application is provided by the CAN Controller Figure 3 shows a block diagram of CAN Protocol Controller with the pins and different blocks inside the controller.

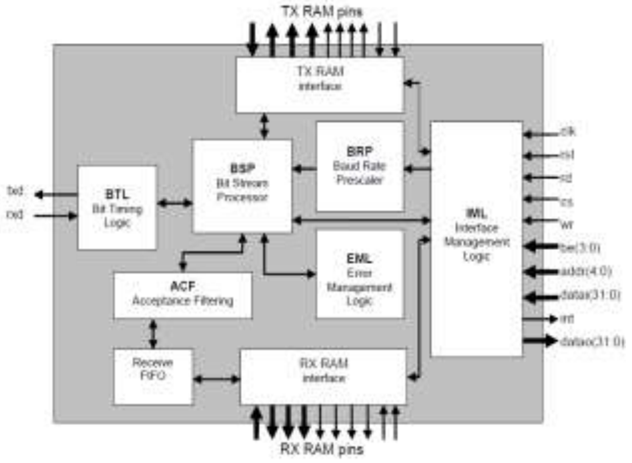


Figure 3 Block diagram of CAN controller

A. Description of the CAN controller blocks [7]

- Interface Management Logic (IML)

Interface management logic interprets commands from CPU, controls addressing of the CAN registers and provides interrupts and status information to the host microcontroller.

- Transmit Buffer (TXB)

Transmit buffer is an interface between the CPU and the Bit Stream Processor (BSP) that is able to store a complete message for transmission over the CAN network. This buffer is 13 bytes long, written to by the CPU and read out by the BSP.

- Receive Buffer (RXB, RXFIFO)

Receive buffer is an interface between the acceptance filter and the CPU that stores the received and accepted messages from the CAN-bus line. The Receive Buffer (RXB) represents a CPU-accessible 13-byte window of the Receive FIFO (RXFIFO), which has a total length of 64 bytes.

- Acceptance Filter (ACF)

Acceptance filter compares the received identifier with the acceptance filter register contents and decides whether this message should be accepted or not. In the event of a positive acceptance test, the complete message is stored in the RXFIFO.

- Bit Stream Processor (BSP)

Bit stream processor is a sequencer which controls the data stream between the transmit buffer, RXFIFO and the CAN-bus. It also performs the error detection, arbitration, stuffing and error handling on the CAN-bus.

- Bit Timing Logic (BTL)

Bit timing logic monitors the serial CAN-bus line and handles the bus line-related bit timing. It is synchronized to the bit stream on the CAN-bus on a ‘recessive-to-dominant’ bus line transition at the beginning of a message (hard synchronization) and re-synchronized on further transitions during the reception of a message (soft synchronization). BTL also provides programmable time segments to compensate for the propagation delay times and phase shifts.

- Error Management Logic (EML)

EML is responsible for the error confinement of the transfer-layer modules. It receives error announcements from the BSP and then informs the BSP and IML about error statistics.

V. DETAILED TESTBENCH ARCHITECTURE

For the verification process, UVM using System Verilog and Mentor Graphics QuestaSim is used to create the testbench environment. A testcase is developed with particular constraints that will limit the random stimulus generation.

The generator creates a programmable amount of random frames that will be inserted in the DUT (Design Under Test). The sequencer will take these frames and will transform them into signals (bytes) and will send them through the interfaces/driver. The scoreboard will predict the expected result from the driver and this result will be used by the checker to compare them with the received data from the DUT.

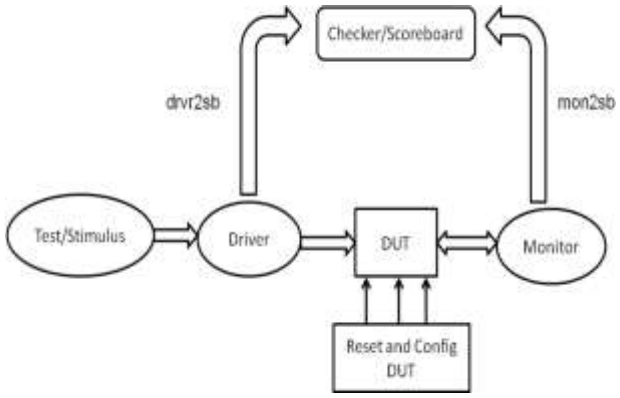


Figure 4: Detailed Testbench Environment

VI. TEST PLAN

Test plan is a document which contains all possible scenarios of test cases. Based on specifications we define all possible test cases and maintain a document for that. It is one of the most important steps of verification flow. Maximum number of test cases can find more bugs from the design. In industry as much possible time is spent in defining the test plan as according to test plan. Based on verification plan we implement all defined modules in verification plan in terms of code using System Verilog language.

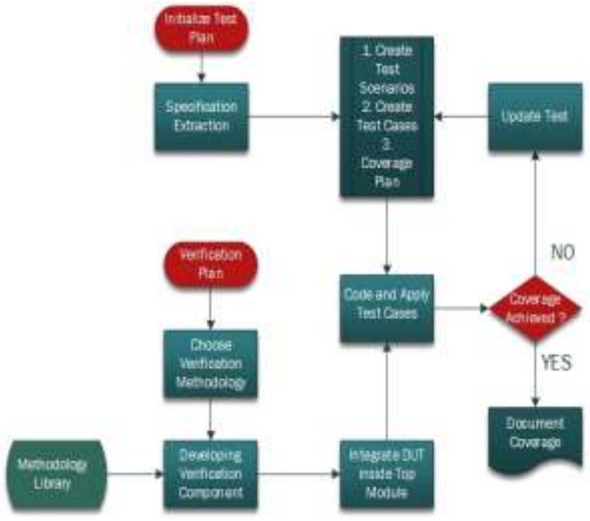

Initialize Test Plan
Specification Extraction
1. Create Test Scenarios 2. Create Test Cases 3. Coverage Plan
Update Test
NO
Verification Plan
Code and Apply Test Cases
Coverage Achieved ?
YES
Choose Verification Methodology
Document Coverage
Methodology Library
Developing Verification Component
Integrate DUT inside Top Module

Figure 5 Test Plan

VII. IMPLEMENTATION

To perform the verification, we need the complete and stable RTL design first. So as first task stable IP Core of CAN Protocol Controller is collected. The CAN Controller IP Core is provided by OpenCores [12] community which provides free IP Core. Compilation results show that the RTL code of CAN protocol controller is syntax and other compilation error free. It means it is ready to be functionally verified.

Table 1 Verification Component

Components	Parent Class	Description
top.sv	NA	top module contains instantiation of interface and CAN core.
interface.sv	NA	Interface block provides communication path between testbench and CAN core.
tr1_test.svh	can_base_test, uvm_test	tr1_test
can_env	uvm_env	Environment class has two components viz. Agent, Scoreboard.
can_scoreboard	uvm_scoreboard	Scoreboard provides output results comparison mechanism and contains function model of our design.
can_agent	uvm_agent	Agent provides three blocks namely Sequencer, Driver and Monitor. It also has connection between all three components and with blocks of Environment.
sequence_item	uvm_sequence_item	To create transactions, apply randomization to desired signals.
can_driver	uvm_driver	Driver converts transactions coming from sequence to signal level activities and applied them to CAN core via virtual interfaces.
can_monitor	uvm_monitor	Monitor collects results from the CAN core output ports via virtual interface and sends them to Scoreboard in form of transactions.

Scoreboard functionality is to compare all inputs to the relative outputs. And for that scoreboard will be connected to CAN functional model. Here, this functional model can be in any foreign language like C, C++, Python etc or it can be created in SystemVerilog. To connect CAN functional model to scoreboard we require DPI-C if the model is in C language.

VIII. SIMULATION RESULTS

A. tr1_test

Two basic sequences are applied to check the UVM environment for CAN protocol controller that are “reset and initialize”.

Simulation waveform for tr1_test testcase is as shown below.

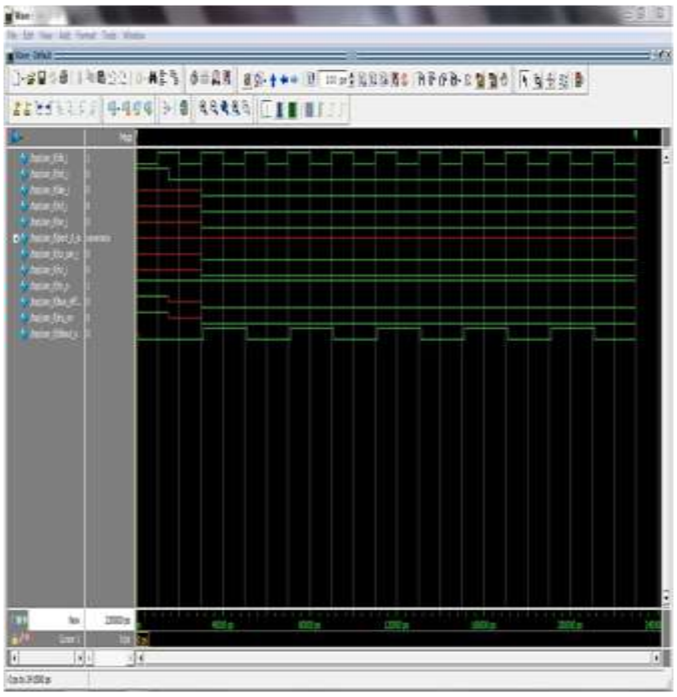
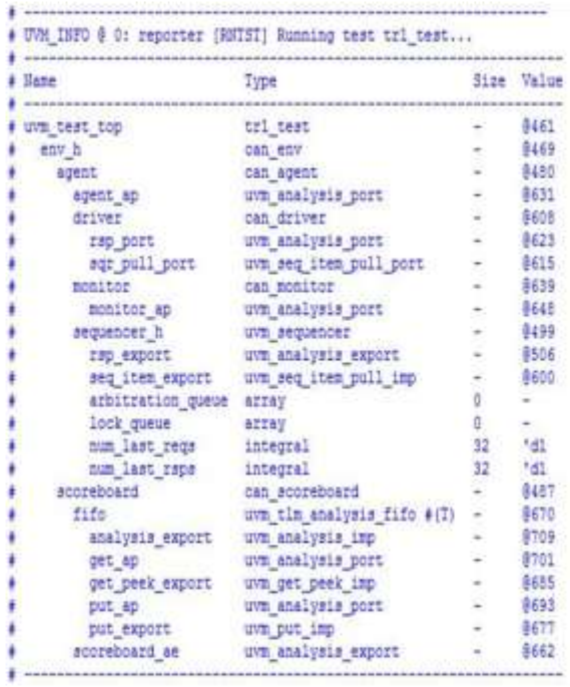


Figure 6 Verification Component Hierarchy



```
# ---------------------------------------------------------------
# UVM_INFO @ 0: reporter [RNTST] Running test tr1_test...
# ----------------------------------------------------------------
# Name                      Type                          Size  Value
# ----------------------------------------------------------------
# uvm_test_top              tr1_test                      -     @461
#   env_h                   can_env                       -     @469
#     agent                 can_agent                     -     @480
#       agent_ap            uvm_analysis_port             -     @631
#       driver              can_driver                    -     @608
#         rsp_port          uvm_analysis_port             -     @623
#         sqr_pull_port     uvm_seq_item_pull_port        -     @615
#       monitor             can_monitor                   -     @639
#         monitor_ap        uvm_analysis_port             -     @648
#       sequencer_h         uvm_sequencer                 -     @499
#         rsp_export        uvm_analysis_export           -     @506
#         seq_item_export   uvm_seq_item_pull_imp         -     @600
#         arbitration_queue array                         0     -
#         lock_queue        array                         0     -
#         num_last_reqs     integral                      32    'd1
#         num_last_rsps     integral                      32    'd1
#     scoreboard            can_scoreboard                -     @487
#       fifo                uvm_tlm_analysis_fifo #(T)    -     @670
#         analysis_export   uvm_analysis_imp              -     @709
#         get_ap            uvm_analysis_port             -     @701
#         get_peek_export   uvm_get_peek_imp              -     @685
#         put_ap            uvm_analysis_port             -     @693
#         put_export        uvm_put_imp                   -     @677
#       scoreboard_ae       uvm_analysis_export           -     @662
# ----------------------------------------------------------------
```

Figure 7 Verification Component Hierarchy

B. can_pkg Package File

```
package can_pkg;

// Include Package Items and Macros
import uvm_pkg::*;
`include "uvm_macros.svh"

// Define Sequencer, Include Sequence Items
`include "sequence_item.svh"
typedef uvm_sequencer#(sequence_item) sequencer;
```

```
// Sequences
`include "sequences/base_sequence.svh"
`include "sequences/reset_seq.svh"
`include "sequences/init_seq.svh"
`include "sequences/tr1_seq.svh"

// UVM Components
`include "can_driver.svh"
`include "can_monitor.svh"
`include "can_agent_config.svh"
`include "can_agent.svh"
`include "can_scoreboard.svh"
`include "can_env.svh"

// Base Test and Extended Tests
`include "test/can_base_test.svh"
`include "test/tr1_test.svh"

endpackage: can_pkg
```

IX. CONCLUSION

Verification plays an important role for the functional safety and understanding of electronic circuits. Literature survey is done to select the verification methodology as UVM. The Universal Verification Methodology (UVM) represents the latest member of a family of methodologies for functional verification of digital hardware. UVM was built on the principle of cooperation between EDA vendors and customers. It is based on SystemVerilog classes, and proven to be a powerful OOP technique with highly reusability. Due to the wide range of applications of CAN controller in automobile industry this protocol needs to be verified. The main objective of this project is to develop a generic verification environment in SystemVerilog by the UVM methodology. So here a verification environment is proposed for CAN Protocol Controller. Here layered testbench is developed where each layer has particular functionality. By using OOP concept different functionality are divided into different classes. Global class contains all global signals and signals which need to be randomized. Generator class performs all randomization and data generation operation. Driver class performs driving command to DUT. Monitor class monitors all activity of whole testbench. Scoreboard class keeps the track of passed and failed transactions. So here self-checking and generic environment is developed. According to specification testplan is developed which contains all possible test cases and scenarios.

REFERENCES

[1] G. Zarri, F. Colucci, F. Dupuis, R. Mariani, M. Pasquariello, G. Risaliti, C. Tibaldi, “On the Verification of Automotive Protocols”, 2006.

[2] Guo Jinyan, Hu Yueli, “The Design and Realization of CAN Bit Timing Logic” in Prime Asia 2010.

[3] Juan Francesconi, J. Agustin Rodriguez, Pedro M. Juli´an, “UVM Based Testbench Architecture for Unit Verification” in Argentine School of Micro-Nanoelectronics, Technology and Applications 2014.

[4] Jonathan Bromley, “If SystemVerilog Is So Good, Why Do We Need the UVM?”, 2013.

[5] Geng Zhong, Jian Zhou, Bei Xia, “Parameter and UVM, Making a Layered Testbench Powerful”, IEEE 2013.

[6] Philips Semiconductors. CAN Specification Version 2.0, Parts A and B [S].1992.

[7] Philips Semiconductors, SJA1000 Stand-alone CAN controller Datasheet, 2000.

[8] Chris Spear, SystemVerilog for Verification, A Guide to Learning the Testbench, MA, Springer, p.15(2006).

[9] Accellera, “Universal Verification Methodology (UVM) 1.1 User‟s Guide”, May 2011.

[10] Kuang-Chien (KC) Chen,” Assertion Based Verification for SoC designs”, 0-7803-7889-X/03 IEEE, Published on 2003.

[11] Yang Guo, Wanxia Qu, Tun Li, Sikun Li. “Coverage Driven Test Generation Framework for RTL Functional Verification” Published in Computer- Aided Design and Computer Graphics, 2007 10th IEEE international conference, pp. 321-326.

[12] OpenCores, Free IP Core Provider, www.opencores.org

[13] Overview and understanding of SystemVerilog and Verification Environment. www.systemverilog.in www.testbench.in

[14] Bhaumik Vaidya and Nayanpithadiya, “An Introduction to Universal Verification Methodology” published in Journal of Information Knowledge and Research in Electronics and Communication Engineering, 2012-13, pp. 420-424