COVERAGE Driven Verification of I2C Protocol Using System Verilog

Anuja Dhar, Ekta Dudi and Hema Tiwari
ECE Department, Jayoti Vidyapeeth Women’s University, Jaipur, India

Pallavi Atha
VLSI Department, Suresh Gyan Vihar University, Jaipur, India

ABSTRACT

The rapid advances in the integration technologies, enables fabrication of millions of transistors in single IC/Chip. So as the design grows, the verification techniques are required to meet the correctness of the design without exercising exhaustive input-output combination. This paper accommodates reusability of I2C protocol under various test environments, by the following System Verilog which support the complexities of the SoC designs. Here, The RTL Design of I2C is obtained from Opencore.org and its functional verification is carried by self, using System verilog completely wrap DUT. The whole verification done using system verilog Hardware description and Verification language (HDVL), simulated on Questa Sim 10.0b. The concept of seed is used to control the quality of random number generator (RNG) algorithm to run the same test case. The functional coverage found using Constraint random verification (CRV) approach is 100% and code coverage found is 86.88% & for DUT 93%.

Key words: I2C, verification, coverage, system verilog, DUT


1. INTRODUCTION

System verilog is a combined hardware description and hardware verification language. It has many features based on extensions to verilog. As it has many user friendly features and also it uses OOPS concepts this makes it very dynamic to verify the environment. System verilog mostly uses higher level abstraction. System verilog offers new procedural blocks that better convey the intended design structure. System verilog has its own assertion specification language that is useful for verifying...
Anuja Dhar, Ekta Dudi, Hema Tiwari and Pallavi Atha

properties of a design. This language also supports randomization process through which at every random value the design is checked. System verilog has a new feature called coverage. It traces all the functionality and line by line code execution of the environment for verification. Everything using these features is covered in this paper to verify our environment.

Using system verilog we had designed our I2c protocol. I2C is a two-wire, bi-directional serial bus that provides a simple and efficient method of data exchange between devices. It was invented by Philips semiconductor. It is a multi-master bus. Its specification are flexible, it can communicate with slow devices and can also use high speed modes to transfer large amounts of data. The system is comprised of two bus lines, SCL (Serial Clock) and SDA (Serial Data). It defines at 3 transmission speed at normal it is 100Kbps, at fast transmission speed it is 400 Kbps and at high speed it works at 3.3 Mbps

2. DUT- “I2C MASTER CORE”

In this paper DUT is I2C Master Core which is the core of I2C that gives commands to various slaves associated to it by Read and write functions. Hence we have verified the functionality of I2C master which correctly reads and write data for its slaves and resets when needed.

I2C bus consists of 2 active wires called SDA and SCL Line with a ground connection. The I2C bus specification states that the IC that initiates a data transfer on the bus is considered as the bus master, at the same time all the other ICs are regarded as to be bus slaves. All communication is control by using SCL line. So, there is a virtual bus slave from which our master is connected.

We have several states on the bus for transfer of data. I2C master comprises of control register, transmit register, status register, Receive register. In every states these registers gets activated one by one for transfer of data.

![I2C DUT Diagram](http://www.iaeme.com/IJARET/index.asp)
Before any transaction on the bus, a START condition needs to be generated on the bus so that all linked up devices get activated. The start condition indicates the beginning of data transfer. It is defined when high to low transition of SDA line and SCL line is high. In our specification we have master clock named as wb_clk_i and synchronous reset wb_rst_i which is set to active high and its width is 1. There is lower address bits named as wb_adr_i having 3 bit width and also 8 bit data named as wb_dat_i. The bus master generates a start signal in command register and sends the address of bus slave along with the indication whether the access is a read or write operation. The command register has 8 bit width and address 0x04. The bus slave will going to match the address sent with their own address and will produce an acknowledge signal which is sent back to bus master during this our status register activated. Here status register is of 8 bit width and has address 0x04. On successful reception of acknowledge signal the bus master transfer the data byte-by-byte basis and now transmit register gets activated, to write data to slave 0 or to read data from slave 1, the data is stored in transmit register and set WR bit if writing and set RD bit for reading. Transmit register has 8 bit width and address 0x03. After data transmission is completed bus master generates a STOP condition. So that bus slave will again come to idle state. The stop condition always denotes the end of transmission whether it is issued at the end or at the middle of transmission. We have 8 bit of receive register also which shows that last byte received via I2C.

In our specification we have external connections which are at the input and output side of scl and sda line. scl_pad_i and scl_pad_o are the serial clock line input and serial clock line output both having 1 bit width. sda_pad_i and sda_pad_o are the serial data line input and serial data line output both having 1 bit width.

3. VERIFICATION TEST BENCH COMPONENTS
The purpose of test bench is to determine the correctness of the design under test (DUT). This can be accomplished by the following step:

- Generate stimulus
- Apply stimulus to the DUT.
- Capture the response.
- Check the correctness.
- Measure progress against the overall verification goals.
Figure 2 Verification Environment

The key concept for verification methodology is layered test bench as shown in figure it actually helps to make the task easier by dividing the code into smaller pieces that can be developed separately. At the bottom is the signal layer that contains the DUT and the signals that connect it to the test bench. The next higher level is the command layer in which DUT's input are driven by the driver and DUT's output drives the monitor. Assertion also crosses the command/signal layer. Functional layer added, which feed down into the command layer. The generator block receives higher-level transaction such as read and writes, break them into individual commands. These commands sent to scoreboard that compares the commands from the monitor with those in the scoreboard. The functional layer is driven by the generator in scenario layer.

3.1. Interface
The DUT and Test bench belong to two different system Verilog instance worlds. The DUT belongs to static instance while test bench belongs to dynamic instance. The DUT's ports are connected to the instance of interface, in such way test bench communicate with the DUT through interface instance as shown in Fig2. Using a virtual interface as a reference or handle to the interface instance, the test bench can access the tasks, functions, ports and internal variables of the system Verilog interface.

Interface also has input, output and in-out ports. Only the variables or nets declared in the port list of an interface can be connected externally by name or position when the interface is instantiated, the original net lists using ports had this information that the compiler uses check for wiring mistake. Modport construct in an interface is used which group signals and specify directions. The driver modport allows us to connect a driver module and respectively the monitor module.
We added a clocking block that defines clock signals, and captures the timing and synchronization requirements of the blocks being modeled. We assemble those signals which are synchronous to particular clock, and make their timing explicit.

In above block, we declare a clocking block called CB that is to be clocked on the positive edge of the signal clk. Next line adds output signals to the clocking block such as: wb_adr_i, wb_dat_i, wb_rst_i and so on.

### 3.2. Transaction

The functional layer is driven by the transact or in the scenario layer. The scenario layer of our test bench orchestrates all these steps i.e. reset, read, and write with constrained-random values. Some input variables such as wb_clk_i, wb_rst_i, arst_i and so on are declared as rand so that these variables are distributed uniformly across valid values and few of signals that is wb_adr_i, wb_dat_i are declared as randc, by doing so no value is repeated with iteration until every possible values has been assigned. Constraint block is used here so that it restricts the range of the variables and defines the relation between variables. Here wb_dat_i range is between 10 to 230 for wb_adr_i equals to 2 and all other variables range are restricted in conditional manner.

### 3.3. Generator

This block comes under functional layer, which receives high level transaction such as read, write and break them into individual commands. And these commands are also sending to the scoreboard through mailbox. Data can be sent to a mailbox by placing a message in it using put () and thus retrieve by scoreboard.
3.4. Driver and Monitor

DUT first unpack the packet which is our signal and runs single commands i.e. read, write which is generated by the generator into actual input for DUT. The Driver sends these signals using Virtual Interface to reset and configure DUT. The DUT’s output drives the monitor and the monitor will keep track of scl and sda lines and also displays various messages according to the operations being performed like whether it is read or write operation. We define one mailbox here for the scoreboard and use put method to place data in it.

3.5. Scoreboard

Scoreboard comes under functional layer as it checks functionality through comparison. In our environment data coming from monitor and data randomly generated in generator are pass to scoreboard directly through to two respective mailbox which are declared in generator. Both the respective values are checked and compared to understand whether we are working on same data & address location. In scoreboard we also compare our results from golden values and set our goals for further improvement.

From the above snapshot we can see that both the mailbox (mbx_scb & mbx_mon) calls respective data & are compared using comparison command (==) to check our test/address location is successful or not. Hence Scoreboard is a checker which helps in checking the functionality of our environment.

3.6. Environment and Program

Environment class used to implement verification environment and Program block provides an entry point for the tests and contain the instance of environment class to access all the public declaration of environment class as shown below in snapshot. All methods in environment class are declared as virtual method and formalize the simulation steps using virtual methods which are used to control the execution of simulation. Below Constructor method declared with virtual interface as arguments, so that when the object is created in test case, new () method can pass the interfaces into environment class where they are assigned to the local virtual interface handle. build() and run() methods are used in environment so that objects like driver, monitor so on are constructed and then after calls all the above declared methods in a sequence order respectively. The test case calls this method, to start the simulation.

3.7. Top

This is the Top layer of the verification environment and work as the entrance of the simulation. Below we can see test, interface and DUT are instantiated and connected with each other with signal using dot mapping and include a simple clock generator.

4. TEST CASES FOR VERIFYING I2C ENVIRONMENT

The Program block defines the test scenario for the test bench and test cases are written in the program block to verify the functionality of DUT. Test case contains the instance of the environment class. This test case creates an Environment object and defines the required test specific functionality. Verification environment contains the declarations of the virtual interfaces. These virtual interfaces are pointed to the physical interfaces which are declared in the top module. These virtual interfaces are made to point to physical interface in the test case.
Coverage Driven Verification of I2C Protocol Using System Verilog

The verification of I2C can be divided into different test case category. Following are the test categories which are going to do exhaustive verification of I2C core.

<table>
<thead>
<tr>
<th>SL. NO.</th>
<th>TEST CASES</th>
<th>SIGNALS REQUIRED FOR OPERATION</th>
<th>SIGNAL CONDITION</th>
<th>EXPECTED OUTCOME</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>RESET</td>
<td>wb_rst_i</td>
<td>If ( wb_rst_i = 1 ), i.e. System should restart and reset all the signals. If ( wb_rst_i = 0 ) System works as required.</td>
<td>System should restart when reset condition is called.</td>
</tr>
<tr>
<td>2.</td>
<td>I2C Master Write Test</td>
<td>wb_rst_i, wb_cyc_i, wb_stb_i, wb_we_i, arst_i, scl_pda_i, sda_pda_i, wb_adr_i, wb_dat_i</td>
<td>Reset should pull low forcefully i.e. ( wb_rst_i = 0 ). For write condition, signals ((wb_cyc_i, wb_stb_i, wb_we_i, arst_i, scl_pda_i, sda_pda_i)) are to be marked as high. Address &amp; data values are passed randomly. Such as ( wb_adr_i ) (0 to 3) &amp; ( wb_dat_i ) (10 – 255) respectively. For Ex: ( wb_adr_i = 0 ), then ( wb_dat_i = 10 – 50 )</td>
<td>System should write the values randomly.</td>
</tr>
<tr>
<td>3.</td>
<td>I2C Master Read test</td>
<td>wb_rst_i, wb_cyc_i, wb_stb_i, arst_i, wb_we_i, wb_adr_i, wb_dat_i</td>
<td>Reset should pull low forcefully i.e. ( wb_rst_i = 0 ). For Read condition, signals ((wb_cyc_i, wb_stb_i, arst_i, ) &amp; ( wb_we_i = 0 ) are to be marked as high. At address ( wb_adr_i ) (2), read the same value that is written in the system.</td>
<td>System should read the value.</td>
</tr>
<tr>
<td>4.</td>
<td>Read After Write</td>
<td>wb_rst_i, wb_cyc_i, wb_stb_i, arst_i, wb_we_i, wb_adr_i, wb_dat_i</td>
<td>Reset should pull low forcefully i.e. ( wb_rst_i = 0 ). Repeat write condition keeping initially ( wb_we_i = 1 ) After 2 cycle ( wb_we_i = 0 ) for read condition.</td>
<td>System should write the value and then after read it.</td>
</tr>
</tbody>
</table>

Note: \( wb\_clk\_i = 1 \) throughout the verification environment

Waveform below explains the final test case (4) which covers all the cases that have been mentioned above in table.
From the below snapshot we have extended transaction named read after write in which we pass the signal’s value in constraint. In program block, create the handle rw_h of subclass and instantiated it by new construct. Then build the environment and pass the value of object rw_h into environment, after that run the environment.

5. COVERAGE PARAMETERS FOR I2C

Coverage is the heart of verification. Every design verification technique requires coverage metrics to gauge progress, assess effectiveness, and help to determine when the design is robust enough for tape out. The coverage tools gather the information during simulation and then process it to produce a coverage report.

![Flow Chart for Coverage](image)

The concept of coverage gets more complex, when we deal with the concepts of functional coverage, cover groups and cover points. The cover group is a construct which encapsulates the specification of coverage model. With the cover points we can generate cover report of our design and know the strength of verification. Bins are associated with set of values or a sequence of value transitions. If bin designates set of values the count is incremented every time to match one of the values of set.
Coverage Driven Verification of I2C Protocol Using System Verilog

Above snap of code shows that cover group is used in generator block which includes cover point wb_data_i, cover_point wb_addr_i, cover_point wb_rst_i to measure its address and data coverage. Cover point may be one or more in cover group, each coverage point includes a set of bins associated with its sampled values. We have sample values inside in a virtual task using sample(). We have used auto bin max which automatically creates bins, and here we define it equals to 4. There are bins which can be declared as ignore by using keyword ignore bins which exclude some values from coverage and in our environment address values 4,5,6 are ignored.

5.1. Types of Coverage

- **Code Coverage**: Code coverage is very helpful at identifying "holes" in verification. Code coverage can be considered as a quantitative measure of DUT code execution. Code coverage checks whether the lines of the DUT has been exercised or not, have all the states in the FSM has been entered or not, have all the paths within a block have been exercised or not, have all the branches in Case have been entered or not, have all the conditions in an if statement is simulated or not.

- **Functional Coverage**: Functional Coverage is a means of measuring what has been verified in a design in order to check off items in the verification plan. It is one phase of total coverage analysis methodology that includes assertion and code coverage. Functional coverage can be considered as a qualitative measure of DUT code execution.

5.2. Functional Coverage vs Code Coverage

Functional coverage are more preferred because 100% functional coverage indicates that all items in the test plan have been tested. Combines this with 100% code coverage and it indicates that testing is done. Code coverage cannot detect conditions that are not in the code and also many boundary conditions. So code coverage cannot be used exclusively to indicate we are done testing.

![Figure 6 Code Vs Functional Coverage](http://www.iaeme.com/IJARET/index.asp)
5.3. Coverage Reports

Coverage reports help us to analyse our output and help in further planning for various tape outs. With the use of cover groups we generate coverage report.

The command used to generate functional coverage report in questasim is:

```
vsim -cvgperinstance-c<module_top_name>
+UVM_TESTNAME=<test_case_name> -do "run -all; coverage report -details -file <file_name.txt>".
```

Coverpoint is added to wb_dat_i,wb_addr_i,wb_rst_i to calculate its coverage. Functional coverage report is shown below:

**Questa Covergroup Coverage Report**

Scope: /top_sv_unit/generator

<table>
<thead>
<tr>
<th>Covergroups / Instances</th>
<th>Total Bins</th>
<th>Covered Bins</th>
<th>Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Covergroup covPORT</td>
<td>9</td>
<td>9</td>
<td>100.00%</td>
</tr>
</tbody>
</table>

**Coverpoints / Bins**

| Coverpoint: cover_point_wb_data_i | 100.00% |
| Coverpoint: cover_point_wb_addr_i | 100.00% |
| Coverpoint: cover_point_wb_rst_i  | 100.00% |

The command used to generate code coverage report in questasim is:

```
vsim -c -do "coverage save -onexit <code_coverage_report_filename.ucdb> ;run -all;exit" -coverage-voptargs="+cover=bcfst" +test_name <module_top_name>
```

Code coverage report is shown below:

**Coverage Summary by Structure:**

<table>
<thead>
<tr>
<th>Design Scope</th>
<th>Coverage (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>top</td>
<td>83.62%</td>
</tr>
<tr>
<td>li</td>
<td>93.93%</td>
</tr>
<tr>
<td>DUT</td>
<td>91.70%</td>
</tr>
<tr>
<td>i</td>
<td>53.72%</td>
</tr>
<tr>
<td>top_sv_unit</td>
<td>91.13%</td>
</tr>
<tr>
<td>generator</td>
<td>100.00%</td>
</tr>
</tbody>
</table>

**Coverage Summary by Type:**

- Weighted Average: 86.53%
- Coverage Type: Statement: 74.00%, Branch: 89.59%, FEC Condition: 72.88%, Toggle: 91.56%, FSM State: 100%, FSM Transition: 86.53%

The above snippet shows code coverage report statement coverage covers 74 %, branch coverage covers 89.59%, FEC condition covers 72.88%, toggle coverage covers 91.56%, FSM state covers 100% and FSM transition covers 86.53%.
Weighted average coverage collected across our DUT is 86.88%. To improve our coverage we followed few basic steps such as -

- By using repeat function, we can increase run 100 times or more, that results to cover more code.
- In generator ignore some address (as per specification) using ignore bins to get accurate results.
- Create less bins, for capturing results.

6. RESULT

The System Verilog verification environment developed along with the complete flow of verification has been discussed. The proposed verification environment applies constrained random technique to fulfill the configuration of DUT. The verification is carried for both the READ and WRITE operation cycles and waveforms are generated as shown in figure. The functional coverage and code coverage is employed to make sure that DUT has realized all the expected functional requirements and line by line code covered respectively. Coverage metrics such as Code coverage, state coverage, branch coverage, FSM transition coverage, Toggle coverage and FSM state is also covered here. Cover points and cover bins are used here to estimate functional coverage.

REFERENCES


