Memory Subsystem Verification: Can it be taken for granted?

Shivani Upasani (shivani.upasani@lsi.com),
Prashanth Srinivasa (prashanth.srinivasa@lsi.com),
LSI India Research & Development Pvt. Ltd.,
Global Technology Park,
Marathahalli Outer Ring Road,
Devarabeesanahalli, Bangalore- 560103
India

Abstract— With the increasing number of processors and on-chip buses, the present day memory subsystems are more complex, making their verification a challenge. This paper talks about the complexities involved and innovative steps followed in developing efficient strategies to verify a memory subsystem. These steps enable us to locate performance bugs and reduce rework when the memory bank structure changes.

Acronyms:

AXI : AMBA Advanced eXtensible Interface
DMA : Direct Memory Access
DUT : Design Under Test
ECC : Error Correcting Code
IP : Intellectual Property
ISR : Interrupt Service Routine
RAL : Register Abstraction Layer
RMW : Read Modify Write
SoC : System on Chip
UVM : Universal Verification Methodology
VIP : Verification Intellectual Property
VMM : Verification Methodology Manual

I. INTRODUCTION

As the SoC complexity grows with the number of processors and on-chip buses, the requirement for on-chip memory with different number of banks and sizes is also growing. In addition, changes to the architecture during the development might often result in changes to the memory requirement, thus causing a need for a robust and reusable verification strategy. This paper talks about various complexities involved and the verification strategies adopted in verifying a memory subsystem.

The DUT/memory subsystem consists of multiple memory banks each having a third-party IP as the memory controller block with AXI interface. All these banks have a common register logic for control and status of the ECC functionality of all of the memories. Each memory bank is configurable in terms of number of sub-banks, data width, and address width.

Fig. 1 depicts the DUT. As shown in the diagram, the memory controller(s) are connected to different memory bank(s). Each memory bank can further consist of multiple sub-banks. The numbers of banks in the design are variable and can be extended depending on the system design requirement.

The need of the hour is a scalable environment, which can keep up with the changing design specifications.

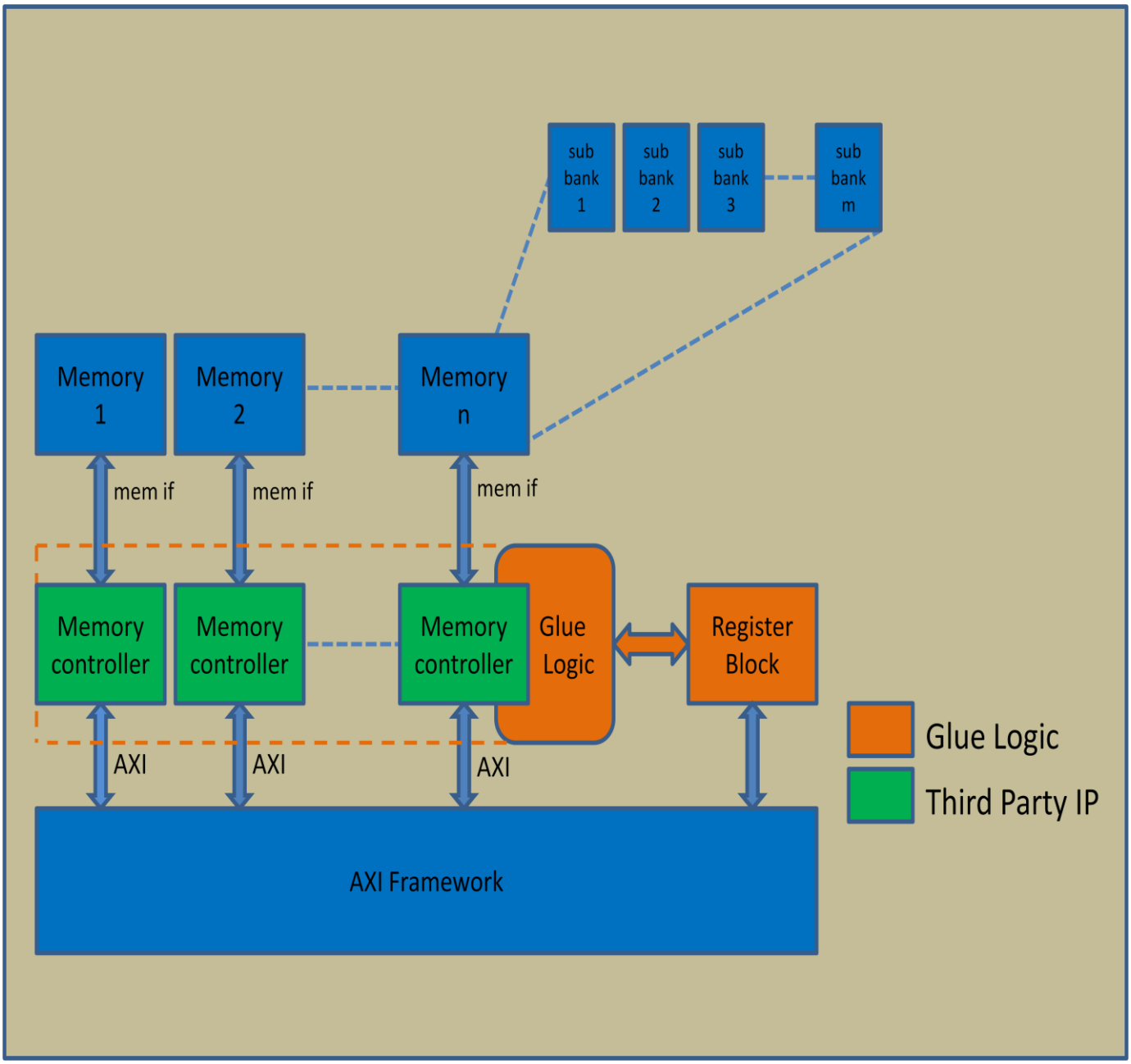

Figure 1. Design Under Test

For the verification of the memory subsystem, we adopted a coverage-driven verification methodology. Environment consists of generators for injecting random stimulus, RAL for register management, self-checking scoreboards, functional coverage models, utility scenarios like interrupt service routines, error injection mechanisms, etc. These are scalable and configurable based on the number of memory banks and address and data widths.

The following challenges are encountered during verification of the Memory subsystem.

1. There are two approaches to develop a scoreboard to test the memory subsystem. The traditional approach is to read the data directly from memory via backdoor after the

transaction is seen at the AXI end. The new approach is to use a reference model to verify the DUT.

2. In order to develop time-accurate checkers to predict the ECC behavior and interrupt set clear values, registers must be accessed. Reading the registers introduces a delay in these checks.

3. We developed a completely re-usable checker to verify the power mode functionality of each bank/sub-bank inside the memory subsystem. This checker performs the connectivity check and also a functional check to verify whether each bank has gone into correct low-power mode.

II. CHALLENGES FACED AND RECOMMENDATIONS

1. *Scoreboard Strategy*

The verification strategy for this DUT involves verifying the data integrity across AXI at one end to the memory at the other end. This is achieved by placing monitors across all the design interfaces. These monitors capture the traffic and post the transactions to the bank scoreboard for comparison. Backdoor access to the memory is used to confirm that the correct data is read from or written into the memory.

Fig. 2 depicts the verification strategy for the DUT.

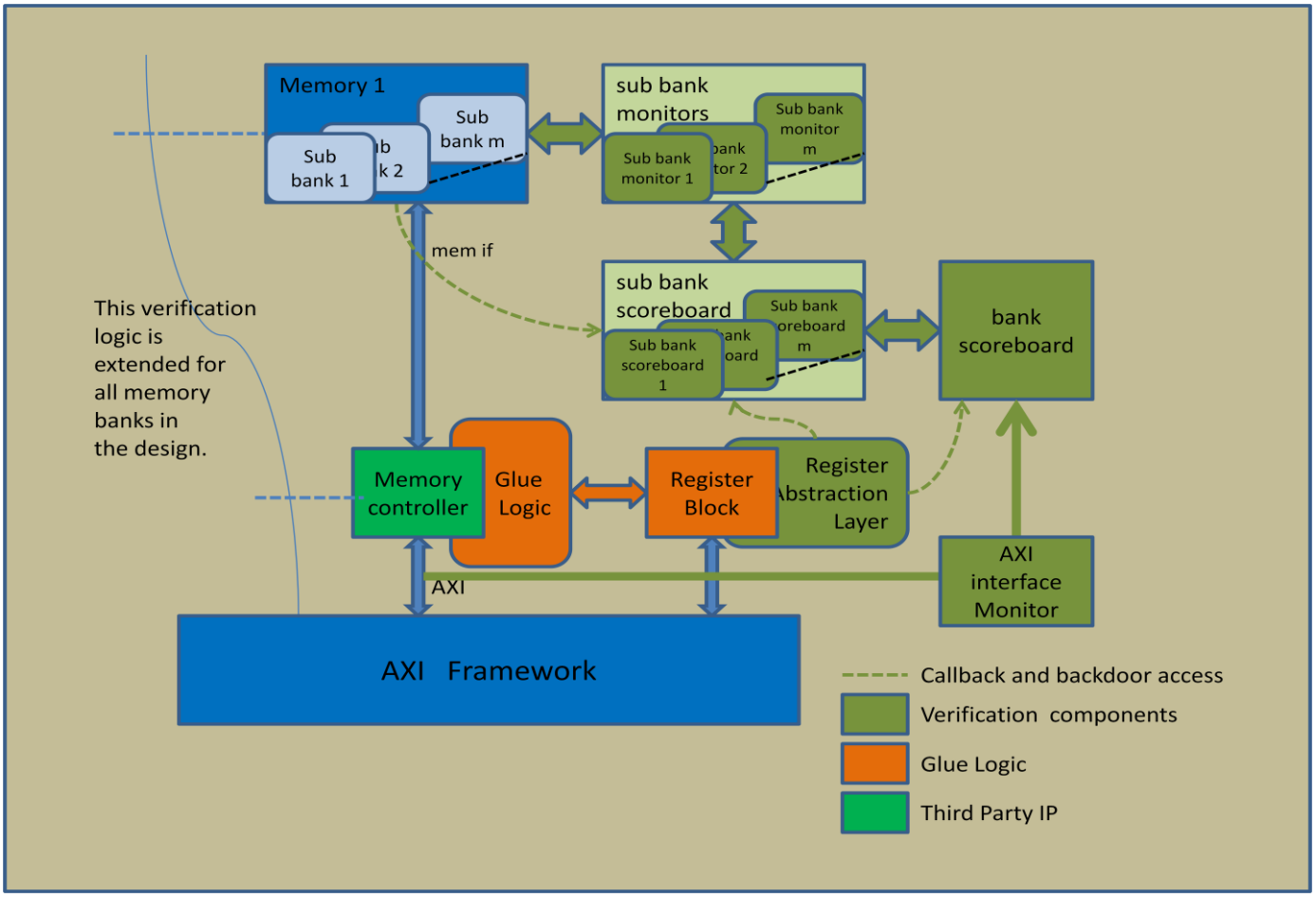

Figure 2. Verification Environment

When the transaction reaches the AXI end, it is taken up by the AXI interface monitor and sent to the bank scoreboard. This scoreboard predicts the number of transactions that are supposed to occur on the memory interfaces.

Each of the memory sub banks inside a memory slave is scoreboarded at a finer level by another scoreboard. For example: sub bank scoreboard 1, 2, etc.

For a write transaction, the bank scoreboard predicts the writes expected on the sub-bank interface. When the writes occur at the memory end, it is read from the memory via the backdoor and compared with the predicted write data.

For a read transaction, the bank scoreboard predicts the number of read accesses on the sub-bank interface. When those reads occur at the sub-bank interface, the data (accessed through backdoor) is updated to each of the predicted transaction. When this data reaches the AXI end, it is compared with each of the predicted data read.

If the AXI write transfer is of lesser size than the memory bus width, partial writes or RMW are done by the memory controller on the memory interface. One write access at the AXI end is translated to read followed by write sequence at the memory end. Using the write strobe signals the bank scoreboard determines if a RMW operation is expected for any AXI transaction. These transactions are then verified as any read and write transactions as explained in the above two paragraphs. We found issues with the DUT in the following cases:

a. Redundant reads found in case of wrap transfers.

In case of AXI wrap, read transfers, if the address wraps back to the same memory word boundary, two reads are initiated (first read for first beat and second read when it wraps back), though a single read is sufficient. If it wraps back after next memory word, second read is must as we have only single register to store memory read data. Only in case of it wrapping back immediately to same memory word, the second read is redundant.

For example, for a AXI WRAP transfer of length = 4, size = byte, and start_address = 0x22, the valid transfers are 0x22, 0x23, 0x20 and 0x21. The memory controller should have read the memory location once and given back the data at the AXI end, but we observed that one read is done at memory end at address 0x20 for 0x22 and 0x23. One more read is done at the same address 0x20 for 0x20 and 0x21.

b. Redundant reads are seen in back-to-back read and write transfers when read and write transfer width is less than the bus width.

In case of back-to-back read and write transfers with random delays from the AXI end, it is observed that due to the interleaving of reads and RMW at the memory controller end, redundant reads are observed to the address of read transfers.

c. Redundant reads seen in cases where ECC generation is disabled

In this case, we observe that for any write transfer with burst size less than the bus width, it is not necessary to perform the RMW operation. But the memory controller initiates these redundant reads.

The above issues are identified because of the reference model scoreboard approach, which actually predicts the *n* number of transfers that are expected on the memory interface depending on the AXI burst type, burst size, and burst length. If any extra read or write is seen on the interface and it is not expected in the scoreboard queue or array the scoreboard raises an error flag.

These redundant reads reduce the performance and the power efficiency of the memory subsystem. Therefore it is important to detect them in any memory subsystem.

In case of a conventional scoreboard, approaches like reading directly from the memory via backdoor or creating a byte-by-byte scoreboard for comparison are not helpful in locating these redundant reads.

In case of scoreboards which read data from memory via backdoor, these redundant reads are missed.

In case of scoreboards which use byte-by-byte address and data queue approach, the address and data queues are populated with data from read transfers seen on the interface. Redundant reads at the memory end are simply ignored or data gets over-written into the same address and data entry.

If we apply these scoreboard approaches to the examples stated above, the redundant reads are either ignored or over-written and will go un-noticed

Therefore even though the time and complexity requirement of taking the reference model approach is more as compared to the traditional approaches, in our case the effort proved to be useful in detecting two important performance and power efficiency issues in the design.

2. ECC and Interrupt Handling:

The memory subsystem supports ECC generation and check. The verification environment uses RAL extensively for verifying ECC functionality of the memory subsystem.

Fig. 3 depicts the interrupt/exception checker logic.

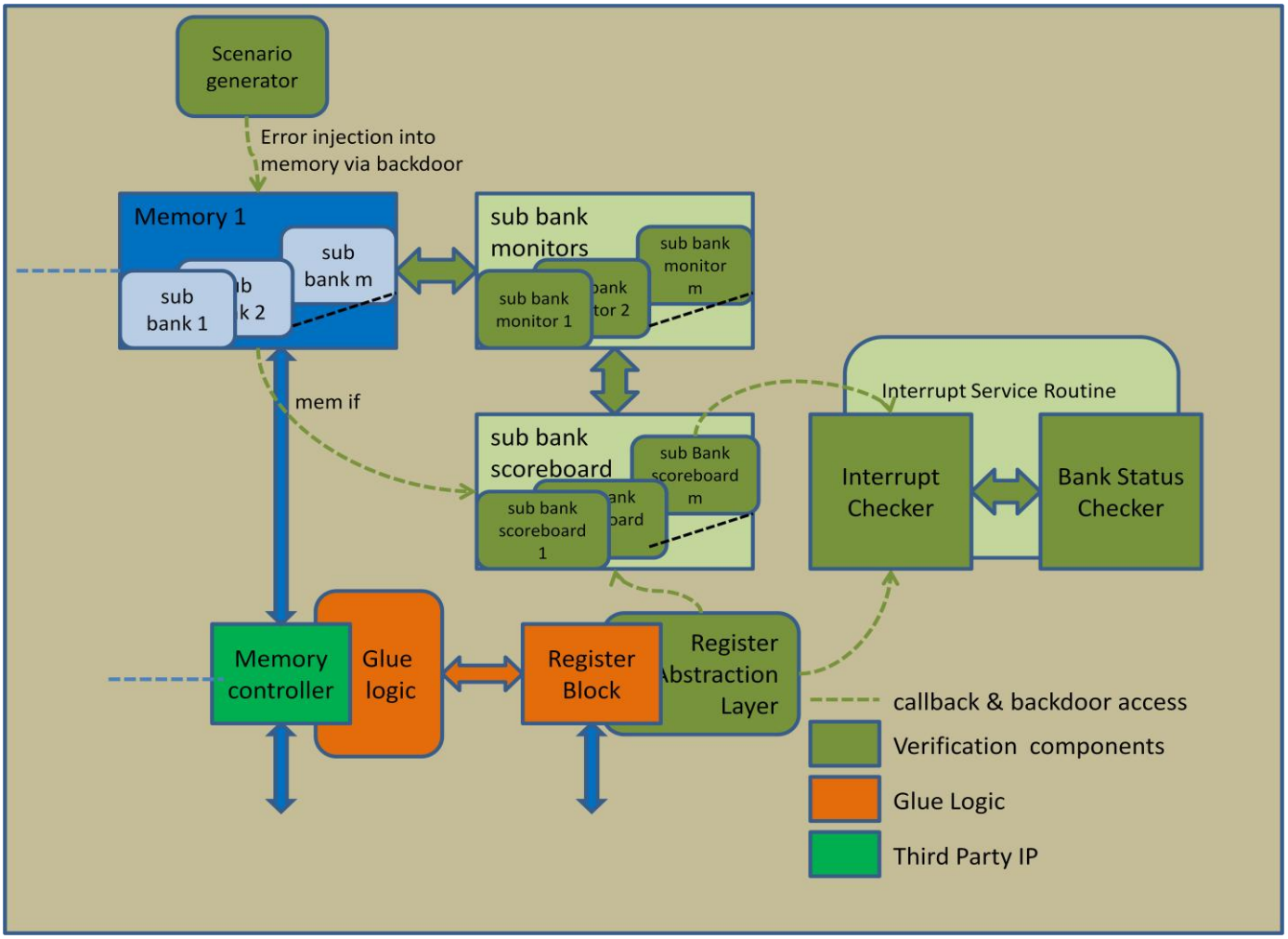

Figure 3. Interrupt and Exception Checker

The verification environment scenarios inject either single bit or double bit error using the backdoor into any address location and generate read or write (unaligned) accesses to that location. This results in read or RMW sequences at the memory end.

On detecting any read-modify-write transactions on the memory interface, the data which is read, is transmitted to the ECC reference model (an ECC generating algorithm) and compared with the read ECC value.

During this transfer, if any correctable ECC error (single bit error) occurs, the memory subsystem goes ahead and sends the corrected data to the master and the data integrity for this phase is done in a similar fashion as that of full write. In case of a double bit error, the data with double bit error is sent back to the master

In case of read transfers, the data read from the actual memory (which might have been corrupted by the environment itself), is transmitted to the ECC algorithm and type of ECC error is calculated. This ECC error information is then passed onto the interrupt checker. It waits for *n* number of clocks before it expects the DUT status registers to be updated with the correct error type

Depending on the interrupt or exception enables, the interrupt or exception is predicted. This is then compared with the actual interrupt or exception being generated by the DUT.

During the error being detected by the environment, the address, syndrome, read/write values, etc. are passed to the bank status checker. These checkers are instantiated per bank. This checker keeps track of the changing address and syndrome values in the individual banks depending on the sequence of errors being detected in each individual bank. It is a requirement that the address/syndrome values do not change if the previous error is not serviced.

We have modeled a system level ISR in our verification environment. On seeing any interrupt or exception, the ISR randomly waits and then clears the status register for any previously generated errors. If the ISR does not clear all the error bits, then the interrupt/exception remains set.

While implementing the checker/ISR for ECC we came across the following issues:

a. One major challenge in this checker is to get the expected value of the DUT status register almost at the time that it is updated in the DUT. We accomplished this using the RAL backdoor access to the status register which returns the value in zero simulation time.

b. The latest RAL comes with a feature where same register can be accessed in parallel using the backdoor access. This feature helps in accessing the status register from two classes simultaneously. One is to check the status being set on any ECC error detected. Second is to clear the status register from the interrupt service routine.

c. We also faced issues with the reusability of the checker since the register names are changing, so we started using parameterized RAL model register names. These are passed using the new function of the checker class and used internally for the functional check. If the register names change, they only need to be passed once and the code for the remaining check need not change.

With this approach we were able to locate couple of issues with the glue logic which is used to latch the ECC error information into the register block.

First issue is that the address and syndrome values latched in the memory bank status registers were changing before the previous error status was cleared from the Status register.

Second issue is seen in back to back error injection. When Single bit error is followed by a double bit error, the Status register fails to capture the double bit error status.

3. *Power mode verification*

The power modes supported by the memory models (Light Sleep (LS), Deep Sleep (DS), and Shutdown (SD)) are verified as a part of the memory subsystem verification. Two types of checks are done in this case – connectivity and functional.

Connectivity checks are assertion checks, which not only test the respective bank being put into low power but also that any other bank's low power mode is not set due to any short in the RTL. It also checks for the low power mode signals being driven correctly to all the sub-banks

It is important to confirm that no access occurs to the memory during low power modes and hence functional check becomes a necessity in this case.

Fig. 4 shows the pictorial representation of the power mode verification done for the memory subsystem.

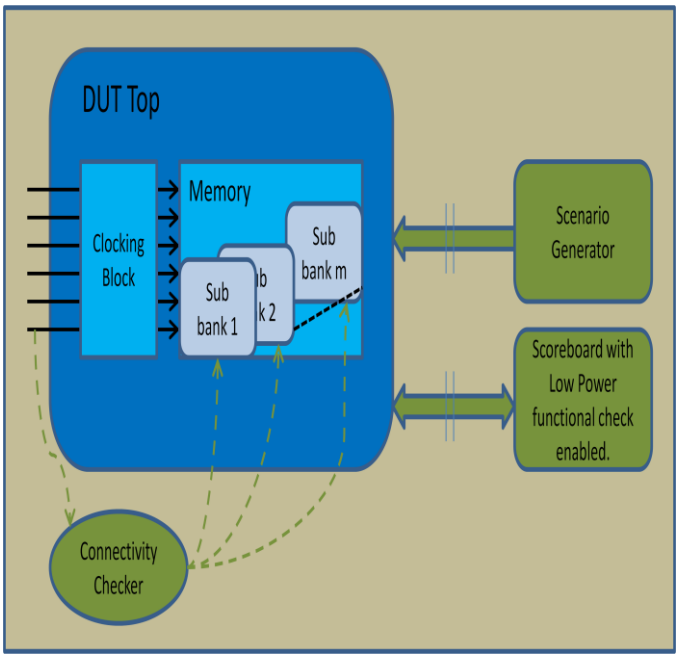

Figure 4. Power Mode Checker

If a write is initiated to the memory bank in case of any of the LS, DS, or SD mode, the write should not occur at the memory end. If a read in initiated to the memory bank in case of low power modes, then for LS, memory should return previously read data, and for DS and SD modes it should return 0.

In our verification environment we randomly drive the LS, DS, and SD modes of a particular memory bank while the other banks are still in operation. This confirms that while one memory is in power-down mode, the other memories are up and running.

In light sleep mode, if a write is initiated to the memory bank, then it does not get written. This is verified by reading from the same address location after the light sleep mode is inactive. If the memory is in light sleep mode and a read is initiated, then the memory continues to latch the previous read data onto the read data bus.

In case of deep sleep mode, if a write is initiated to the memory, it is not forwarded to the memory and in case of read transfer, the memory returns a value of 0.

In case of shut down mode, writes do not occur to the memory. For read transfers the data read is always 0 ('x' in case of timing simulations). The memory is initialized again and tested for any data transfer to make sure that it comes out of shut down correctly.

Thus, the memory power modes are verified in this subsystem.

III. RESULT AND CONCLUSION

1. Using the reference model scoreboard approach, we are able to find performance issues with the third party IP, which is doing redundant reads. This bug would have been missed if non reference model approach was used. The development for the reference model scoreboard took us close to three weeks and we are estimating an approximate effort of one week for developing the scoreboard using non reference model approach. Though the effort is three times more, it proved helpful in locating performance and power efficiency bugs in this design.

2. This environment can be scaled to support any memory bank configuration.

3. We successfully verified the low-power modes of the memory by carrying connectivity checks and functional checks.

IV. ACKNOWLEDGEMENT

We would like to thank LSI and DVCon for giving us the opportunity to present our work and share our experience in a diversified forum. We also express our gratitude to Dwaraka Jayendra, our manager for his guidance and help.

V. REFERENCE

[1] IEEE Standard for System Verilog - Unified Hardware Design, Specification,and Verification Language

[2] Verification Methodology Manual for SystemVerilog by Janick Bergeron, Eduard Cerny, Alan Hunter and Andrew Nightingale