Title: Advanced Techniques for AXI Fabric Verification in a Software/Hardware OVM Environment

Galen Blake
SMTS
Altera Corporation
Austin, TX

Steve Chappell
Solutions Architect
Mentor Graphics
San Jose, CA

ABSTRACT:

In this paper we present an architecture for verifying proper operation and performance of an AXI bus fabric in a dual-core ARM processor system using a combination of OVM and C software driven test techniques.

The end system being verified consists of a dual core ARM processor connected to an AXI bus fabric. Various peripherals connect to the fabric using both AXI and AHB bus interfaces. Confirming fabric connectivity and performance under different end user scenarios are among the key verification goals.

The dual core processor is configured to run a minimal Operating System (OS) designed to test basic operational features of the system including the peripherals and interfaces. Embedded C software libraries were developed to manage OVM sequences capable of driving fabric ports and checking their status. The Embedded C software is also capable of initiating direct fabric accesses using the AXI port which connects the processor to the Fabric. Both of these Embedded C software techniques are leveraged as part of the overall AXI fabric verification framework and reused for other major sub-system verification of the design.

Master peripherals access the fabric and slave peripherals respond using AXI, AHB or APB ports. Although the fabric is predominately AXI, some older peripherals still use AHB which is bridged to AXI within the fabric. APB is used for most peripheral register access and data transport for the slow peripherals. Bus access typical of a particular peripheral operation is modeled using highly configurable OVM sequences that drive the protocol specific verification IP.

An advanced graph based solution was deployed in the design of these port-level sequences. They provide the capability for checking full protocol compliance, an engine for continuous traffic generation, precise control and configurability for shaping the form and type of traffic needed to test the fabric. These characteristics are easier to construct, easier to analyze and review and are more efficient to

achieve coverage than constrained-random or directed OVM sequences.

In order to produce the same level of coverage to the port connecting the processor to the fabric, the graph based solution is also applied to the AXI port through the embedded C where the OVM sequence is replaced with a series of API calls.

During verification bring-up multiple port-level sequences were configured to generate traffic typical of end user peripheral operation. Various burst attributes including burst sizes and access types, along with typical data access rates for the peripheral were configured. Sequence operation was initiated from the Embedded C software.

A sub-system level OVM sequence layer is added to control the port level sequences. It is responsible for generating more complex traffic scenarios that mix and control the traffic on each peripheral sequence. The sub-system sequence must be able to create conditions that are typical of the overall fabric and system operations. This sequence must manage, control and synchronize the activity of each peripherals port-level sequences instructing them to generate transfers with a wide range detailed operations. This includes varying payload sizes, varying destination slave addresses, varying idle time between transfers and many others. This approach evaluates performance of the fabric running various normal and heavy traffic scenarios to extract actual performance characteristics. These are compared to predictions of architectural models acceptable system performance parameters. If any traffic conditions which lead to performance degradation outside the acceptable range are identified, proper action can be taken.

An advanced graph based solution was likewise deployed in the design of the sub-system level sequence to effectively manage and deterministically calculate that all of the important traffic scenarios are reached.

Some of the more interesting and challenging aspects of this work will be discussed:

- How to design highly configurable and adaptable port-level sequences
- Designing the architecture of the higher-level sequence and connection to the port-level sequences it manages
- Issues implementing traditional System Verilog coverage metrics in this type of an application
- Instrumenting the fabric to evaluate performance under different traffic scenarios

1. Introduction

Verifying each master to slave connection on an AMBA fabric is a reasonably straight forward task. Verifying that each port complies with both the standard defined protocol and any user defined conditions at every Master and at every reachable slave will increase the complexity of the verification task. Moreover, verifying the fabric will continue to function and maintain acceptable performance under normal and heavily loaded traffic conditions introduces several unique challenges.

Normal fabric operations will include bus transactions from multiple masters being sent to multiple slaves. Some masters may also have multiple transactions in flight.

Defining and controlling transactions on the fabric from each of the master and slave ports in a real system using the particular protocol for each of those ports can be an intractable problem particularly when we seek a high level of synchronization and control of that traffic. For example, it is not easy to send a packet to an Ethernet peripheral block and then predict with some precision exactly which types of AXI transactions might result much less on which clocks they will occur. There are several dependencies on the state of the block such as buffer conditions and packets already in flight. This can be further obscured by design specific implementation choices and register settings that are found in a 3rd party IP. Taking just this one peripheral example and its protocol into account, now imagine multiplying this across all the protocols found in a system.

To solve this challenge, we replace those peripheral blocks with VIP models for the connected protocol giving us much more precise control and dependable operation.

We then leverage this control to construct tests that mimic the normal flow of data from each peripheral. Then we model the normal and heavy traffic scenarios to match expected system operations. Verifying the functionality and performance under loaded traffic scenarios helps us determine there are no conditions that could lead to stalls or dead locks in the fabric, stalled Masters, stalled slaves or issues with performance degradation beyond tolerable limits.

2. Environment

The testbench environment is fully based on System Verilog and the OVM 2.1.1 library. The environment also includes significant embedded C software running on the CPU that performs chip level initialization, driver operations and many test control and monitoring operations.

There are a number of OVM based Verification IP (VIP) components that form the foundation of the test bench. Coupling the embedded C software into the verification environment means the testbench is also tightly bound to the CPU. Therefore, the CPU can be used to coordinate and check test bench activity. This is facilitated by a custom OVM based mailbox system with dynamic message passing.

The VIP models receive transactions directly from OVM sequences launched by an API in the embedded C SW and/or OVM sequences launched by the test bench. OVM Sequences launched by the CPU can include either directed tests, constrained random tests or more advanced sequences. A block diagram of the environment is shown in Figure 1 below. Some details such as the CPU-TestBench communication system are omitted for clarity.

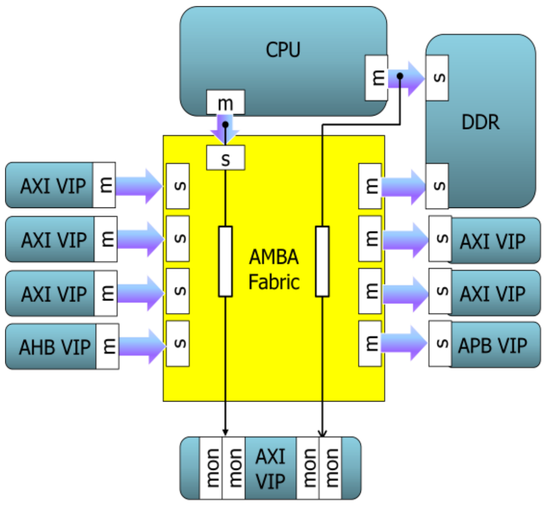


Figure 1 – Block Diagram

3. Beyond Constrained Random

Constrained Random Verification or CRV has been proven to increase productivity and find bugs missed by directed tests. Nevertheless there are limitations:

- Users are responsible to define a reasonable set of random variables and constraints. The definition of the variables and constraints is spread across many files. This sprawling structure of data is difficult to create, difficult to visualize, difficult to analyze, challenging to refine and hard to assert any level of precise control.

- Constraint solvers are proprietary and users are not assured of consistent results across simulation platforms.

- Discovery of interesting or important corner cases are randomly discovered and are subject to the odds of random convergence of multiple variables. Random coverage of the defined coverage space is not efficient. Some areas may be repeated many times before a new unexplored area is exposed.

- Finally, successful CRV also requires development of coverage models to measure test effectiveness which can be extremely difficult.

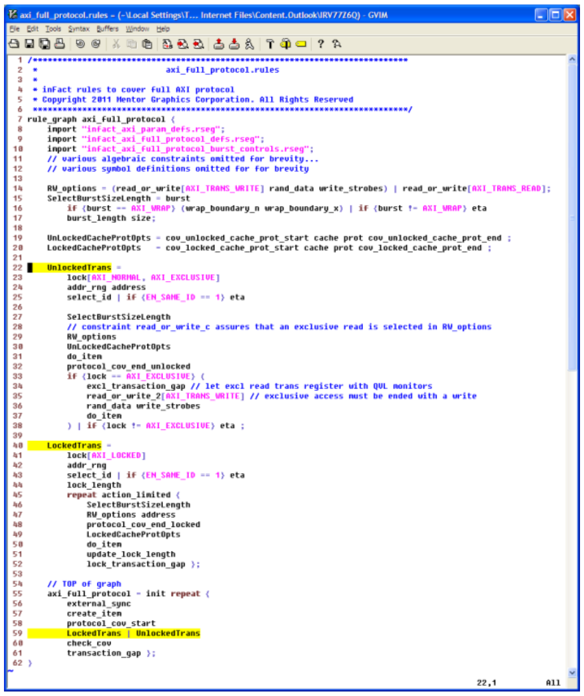
To overcome these limitations, we chose a graph based solution. Specifically we address the limitations of CRV in the following ways.

In order to address the coverage definition problem, we replace the random variable and constraint definition with an efficient and compact grammar that defines the coverage space in a single file. This file is compiled into a graph that makes it easy to visualize and analyze for correct and complete definition. This comprehensive view of the functional space we intend to cover gives us feedback on the parameters that are covered and those that are intentionally left out, as well as out of band features that may be selectively enabled and covered. Careful reviews of the graph can also give us feedback on any features of a protocol that we might have missed. An example of the grammar is shown in Figure 2 and an example of the graph it produces is shown in Figure 3.

This solution can be ported to any simulation platform assuring us of consistent results without any dependency on the constraint solver of the simulator.

Interesting and important cases are dependably covered efficiently without dependencies on random chance.

Finally, coverage checking can be built into the graph and it can improve coverage closure efficiency by testing the complete scope of the protocol in a minimal number of simulation clock cycles. Using this approach, we are assured that we cover the complete protocol space covering corner cases with high efficiency.


axi_full_protocol.rules - (~\Local Settings\T... Internet Files\Content.Outlook\IRV77Z6Q) - GVIM
File Edit Tools Syntax Buffers Window Help
22,1 All

```
/*************************************************************************
 *                        axi_full_protocol.rules
 *
 * inFact rules to cover full AXI protocol
 * Copyright 2011 Mentor Graphics Corporation. All Rights Reserved
 *************************************************************************/
rule_graph axi_full_protocol {
    import "infact_axi_param_defs.rseg";
    import "infact_axi_full_protocol_defs.rseg";
    import "infact_axi_full_protocol_burst_controls.rseg";
    // various algebraic constraints omitted for brevity...
    // various symbol definitions omitted for for brevity

    RW_options = (read_or_write[AXI_TRANS_WRITE] rand_data write_strobes) | read_or_write[AXI_TRANS_READ];
    SelectBurstSizeLength = burst
        if {burst == AXI_WRAP} (wrap_boundary_n wrap_boundary_x) | if {burst != AXI_WRAP} eta
        burst_length size;

    UnLockedCacheProtOpts = cov_unlocked_cache_prot_start cache prot cov_unlocked_cache_prot_end ;
    LockedCacheProtOpts   = cov_locked_cache_prot_start cache prot cov_locked_cache_prot_end ;

    UnlockedTrans =
        lock[AXI_NORMAL, AXI_EXCLUSIVE]
        addr_rng address
        select_id | if {EN_SAME_ID == 1} eta

        SelectBurstSizeLength
        // constraint read_or_write_c assures that an exclusive read is selected in RW_options
        RW_options
        UnLockedCacheProtOpts
        do_item
        protocol_cov_end_unlocked
        if {lock == AXI_EXCLUSIVE} (
            excl_transaction_gap // let excl read trans register with QVL monitors
            read_or_write_2[AXI_TRANS_WRITE] // exclusive access must be ended with a write
            rand_data write_strobes
            do_item
        ) | if {lock != AXI_EXCLUSIVE} eta ;

    LockedTrans =
        lock[AXI_LOCKED]
        addr_rng
        select_id | if {EN_SAME_ID == 1} eta
        lock_length
        repeat action_limited {
            SelectBurstSizeLength
            RW_options address
            protocol_cov_end_locked
            LockedCacheProtOpts
            do_item
            update_lock_length
            lock_transaction_gap };

    // TOP of graph
    axi_full_protocol = init repeat {
        external_sync
        create_item
        protocol_cov_start
        LockedTrans | UnlockedTrans
        check_cov
        transaction_gap };
}
~
```

Figure 2 – Protocol Grammar.

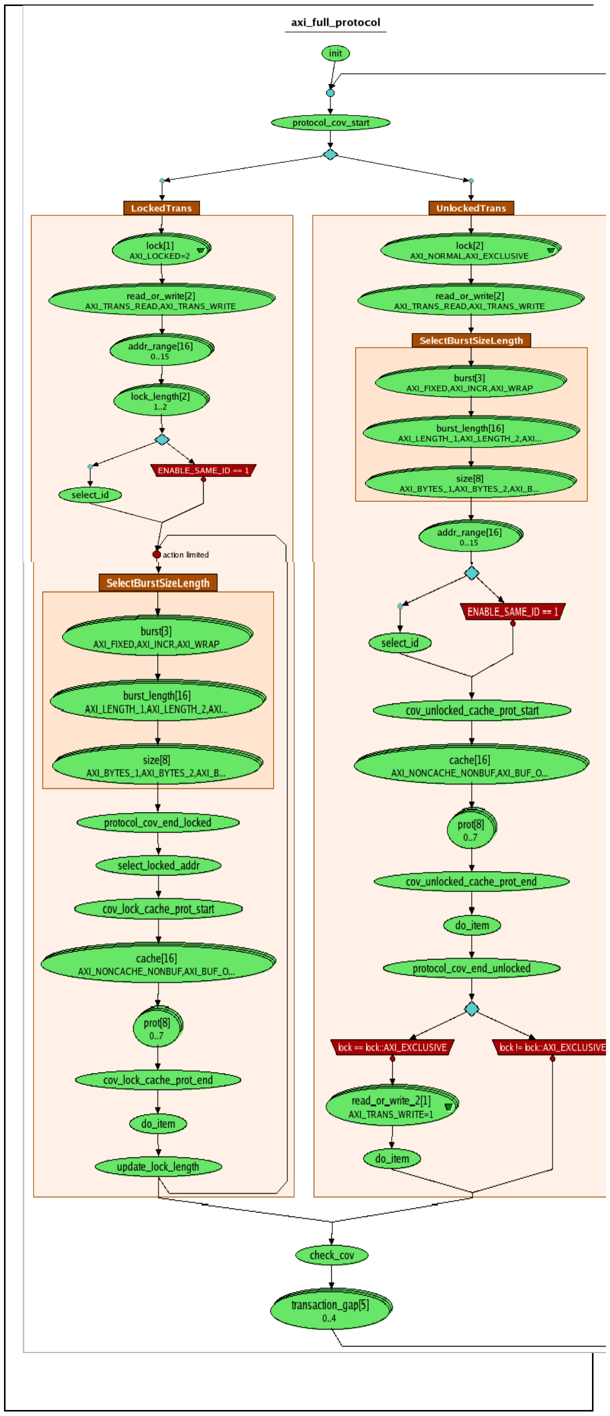


Figure 3 – Protocol Graph.

4. Testing Individual Ports and Paths of the Fabric.

Testing individual ports with specific protocols must cover all aspects of the protocol. Additionally, each master port must be tested to confirm that it can reach all accessible slave destinations. The slaves themselves may support a subset of a protocol or even be a different protocol altogether from the Master. For example, an AXI master could initiate a 64 bit transaction to a 32 bit APB slave. It is the job of the Fabric to split the original 64 bit transaction into 2x32 bit transactions and the test bench to track it. In this example the Master and Slave monitors report transactions based on the Master ID and Fabric ID using local scoreboards and analysis ports. A subsystem scoreboard subscribes to the local analysis ports for checking. Details of this checker are omitted for brevity.

The graph is used in the form of OVM sequence compatible with the VIP. This works for most masters but the graph can also be used to generate calls to the embedded C API giving us the ability to use a consistent approach to test the AXI master port on the CPU connecting it to the Fabric.

The graph based sequences can be used for protocol testing, path coverage and also generating high volumes of transactions. The graph based sequences also have numerous parameters used to activate or deactivate supported features of each individual port instance.

In addition to protocol and path tests, the graph can also be used to generate endless streams of traffic that can be controlled by the graph itself with a local perspective matching traffic expected from the normal peripheral the VIP has temporarily replaced.

Moreover, the local traffic controls and parameters in the graph can be extended to an external graph with a system perspective where there is awareness of the traffic conditions on all other ports. These controls give us the ability to define, control and synchronize the traffic conditions across the fabric.

5. Traffic Synchronization and Control.

The same principles that guide our choice of a graph based solution for bus protocols also apply to definition and control of traffic conditions. The local controls in each "protocol graph" that give the ability to shape traffic within that graph can also be dynamically controlled by this "traffic graph". Examples of these controls include graph parameters such as the number of idle clocks between transactions, the size of data and number bursts in a transaction. Graph parameters can be dynamically controlled to be a fixed number, a random range of numbers or a weighted random range of the numbers.

Additional controls are added for synchronization. For example, a protocol graph can be instructed to conduct a single transaction and stop until instructed to run the next transaction. It can be instructed to run specific numbers of transactions or run continuously until instructed to stop. This gives the traffic graph several different ways to control traffic.

Dynamically controllable graph parameters can be changed between transactions and even during a transaction at certain control points defined in the protocol graph. For example, before a transaction completes, it could check to see if there are any updates to the number of idle clocks between transactions prior to completing the transaction.

Most important, the traffic graph has the ability to simultaneously launch transactions on multiple ports that can be synchronized to start on the same clock. All these features can be used to produce any number of worst case scenarios to thoroughly examine the capabilities of the Fabric. An example traffic graph is shown in Figure 4.

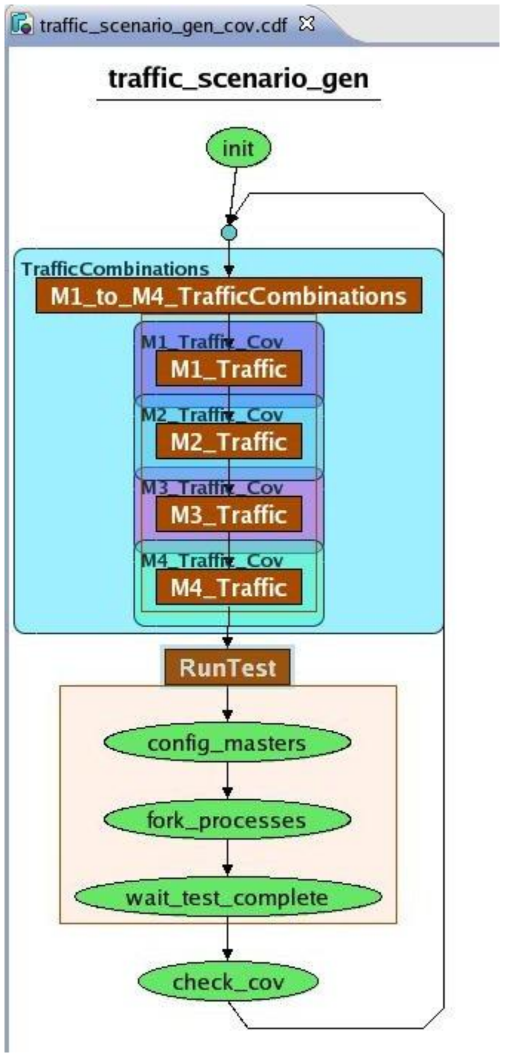


Figure 4 – Traffic Graph.

6. Modulating traffic.

Making effective use of the graph parameters described above gives us the ability to control and shape or modulate the traffic at each Master port. For example, the density of transactions can be adjusted to match the traffic conditions found on a peripheral that only has occasional traffic.

Transactions can be queued and released on multiple ports simultaneously or staggered in a very controlled manner. The size and type of transactions can also be controlled to match expected system operations. For example, the bandwidth of a slow peripheral device will not generate the same amount of traffic as a high speed peripheral device. Some peripherals may have very dense transactions for brief periods of time and then go quiet for a while. Some may have constant high density transactions. The shape of the traffic can be influenced by buffer sizes in the peripherals, the layout and arbitration defined in the fabric, bandwidth limitations at popular slaves like DDR, clock and clock ratio settings and interactions between multiple masters and slaves. The traffic graph needs the ability to modulate traffic in a manner that matches normal system operations described above. The protocol graph that is used to interact with the API in the embedded C also needs to implement the same level of control. An example of modulated traffic control with a normal traffic scenario on three masters is shown in Figure 5. Each box represents a series of nearly continuous bus transactions with very short (not pictured) idle cycles. A “heavy” traffic scenario would have more activity and less idle time between each series of transactions and an example is shown on Figure 6.

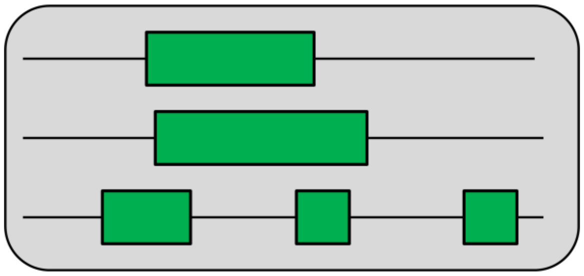


Figure 5 – Normal Traffic Modulation Scenario

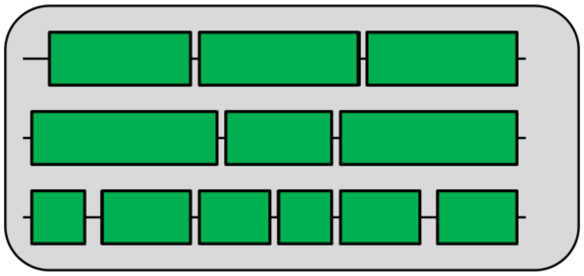


Figure 6 – Heavy Traffic Modulation Scenario

7. Tracking performance.

In addition to coordination and control, there also needs to be instrumentation to monitor performance of the fabric. We first check the “ideal” or unloaded latencies of each path to validate predictions of our architectural model. Next we use our architectural models to predict latencies under normal traffic and heavy traffic conditions. We use these predictions to define acceptable performance conditions.

We then take full advantage of the traffic graph to develop very large numbers of normal and heavy traffic scenarios ensuring that each of them maintain basic operations and that performance does not degrade below acceptable limits. Performance metrics include both bandwidth and latency. Basic operations and performance are monitored in flight with scoreboards used to track and report progress.

8. Findings and Conclusion

Using the graph based approach has improved design quality very early in the project. Protocol coverage is reached efficiently and traffic analysis has already achieved good results improving the design of blocks connected to the fabric.

The advantages versus constrained random have been proven. Fabric and System-level coverage goals have been more easily defined and achieved.

As this environment reaches maturity and small enhancements are added, our confidence increases that we have a fabric and a system that will operate correctly and that our system performance goals will be met.