Verifying Microprocessor Debug-Bus Connectivity Formally Using VC Formal

Vinayak Kamath
AMD, Inc.
Austin, Texas, U.S.A.
www.amd.com

Xiaolin Chen
Synopsys Inc.
Mountain View, California, U.S.A.
www.synopsys.com

ABSTRACT

A microprocessor debug-bus is a DFD (Design for Debugging) feature designed to pull debug state information from various locations on the chip and funnel it into the Debug State Machine (DSM). Since the debug-bus control-path and datapath signals get consolidated only at the end of the project, the debug-bus has to be continually verified to ensure end-to-end connectivity. Unlike simulation-based verification, formal verification-based connectivity checks are exhaustive and are not stimulus dependent. This makes debug-bus connectivity checking a good candidate for formal verification. We discuss a flow that extracts design intent from the RTL implementation in the form of assertions and proves them formally. We also discuss how the flow tailored the debug-bus RTL to meet Synopsys VC Formal's requirements. This connectivity check has been successfully used in numerous AMD microprocessor core development projects.

Table of Contents

Table of Figures

Table of Tables

1. Introduction

Formal verification is playing an increasingly important role in the overall improvement of the design quality and verification efficiency. Assertions and properties are now accepted as an integral part of verification. Chip designers prefer to exhaustively test their design, wherever possible. Formal apps have made it easier to apply formal verification to verification problems such as register checking, connectivity checking, equivalence checking and X-propagation, to name a few. In addition to all this, the availability of greater computing power has made formal verification a viable and practical verification tool.

The debug state machine(DSM) is a piece of DFT IP which allows complex triggering sequences to be programmed to provide flexible and powerful debug visibility during silicon bring-up. To collect debug data specific to the functional issue being investigated, microprocessor core states can be captured on an on-chip storage structure (such as the L2 cache) or sent off-chip. AMD uses DSMs in its CPUs, chipsets, southbridges, and graphics. Functional signals are routed to the DSM via the debug-bus through several stages of programmable multiplexing logic. Thus, the debug-bus provides a convenient way to observe signals within various functional blocks during silicon validation.

We have used VC Formal-based connectivity checking for verifying that all the functional signals connected to the debug-bus, which serve as debug data, can be correctly routed to the DSM. As there are hundreds of debug signals, guaranteeing connectivity of every single signal via simulation is cumbersome as it requires writing separate directed tests for each design block. In contrast, VC Formal can exploit the regularity of debug-bus architecture to prove end-to-end connectivity. This paper describes our experience of applying formal verification beginning with verification planning and implementation to signing off on debug-bus functionality.

2. Debug-bus design

The debug-bus is designed in the form of a ring around the core with stops in each of the design blocks, as shown in Figure 1. Each debug-bus stop contains a debug-bus muxing unit (DBMU) instance. Each instance can be individually configured to do one of two things:

a. Forward incoming debug-bus data from the previous stop
b. Send out local functional(debug) signals or BIST data

DBMUs are enabled and controlled by programming model-specific registers in the core. The debug-bus data is logged into a circular buffer in the DSM. Longer program traces can be collected by writing the debug-bus data into memory.

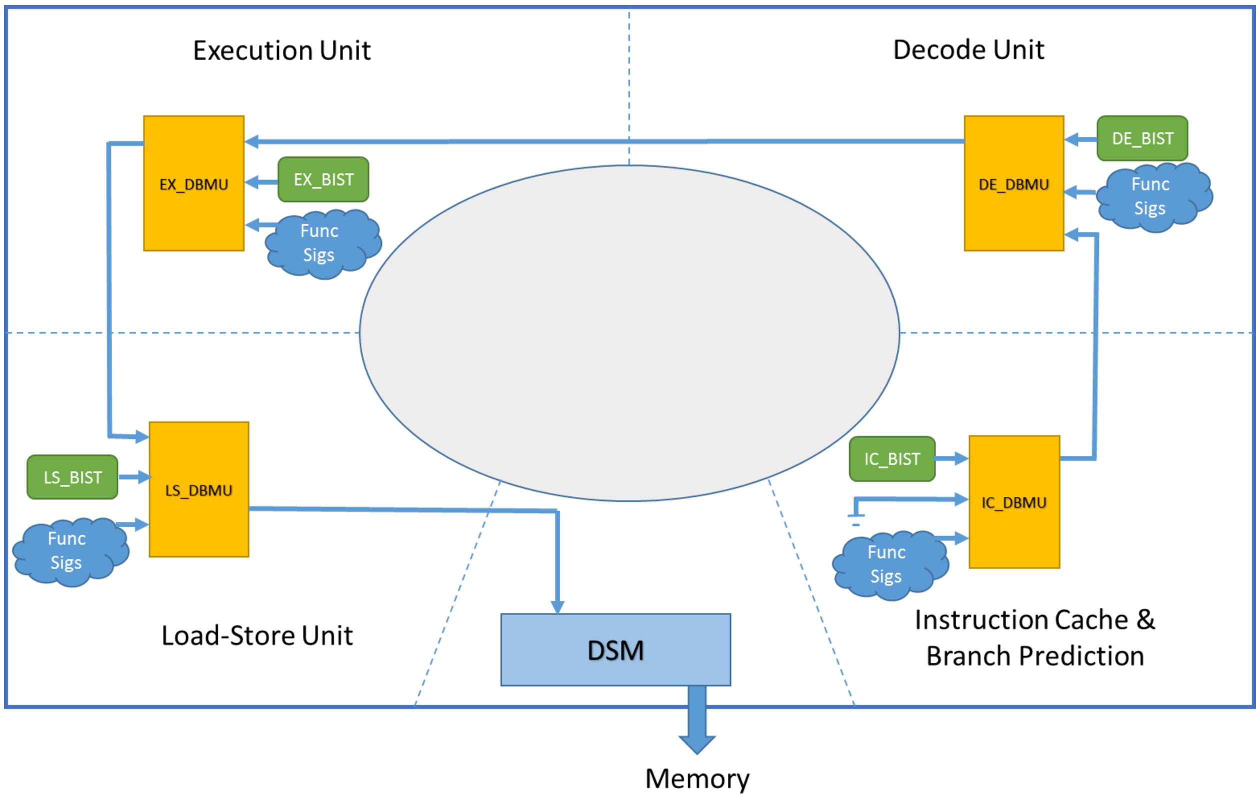

Figure 1: Microprocessor core debug-bus topology (datapath only)

2.1 Debug-bus structure

Each DBMU is 64-bits wide. However, debug signals can be of any arbitrary width. Even though only 64 bits of data can be observed per DBMU at any given time, the total debug data that fans into each debug-bus stop is much higher. So, the debug signals are funneled through a series of staging flops and muxes, as illustrated in Figure 2.

DbusSig_0, DbusSig_1, DbusSig_2 and DbusSig_3 are four debug signals connected to the DBMU. For the sake of simplicity, each functional signal, staging flops and the DBMU have been shown as a being only one-bit wide. DbusCtl_0, DbusCtl_1 and DbusCtl_2 represent three control signals used to select the debug signal that we want to observe at the DBMU output. The staging flops connected to debug-bus muxes are gated by the enable signal DBMU_data_en. This signal is also used to enable the DBMU. The MBIST input and incoming debug-bus data (from the previous stop) input of the DBMU have not been shown.

It is important to note that functional signals in each block are drawn from several sub-blocks. So the staging flops and muxes will not necessarily be in the same design hierarchy. Consequently, the path delays from a signal to the DBMU output can vary, even if they are located within the same processor block. For example, in Figure 2, from the time we apply the select value {DbusCtl_0=1'b0, DbusCtl_2=0}, it takes DbusSig_0 four cycles to appear at the DBMU output. This includes a single cycle consumed by each of the three staging flops I_Dbg_20, I_Dbg_10 and I_Dbg_00 and an additional cycle consumed by the DBMU. However, if we apply the select value {DbusCtl_0=1'b1, DbusCtl_2=0}, it takes only three cycles for the value of DbusSig_1 to appear at the DBMU output. This is because DbusSig_1 has to only pass through two staging flops. Thus, even though both DbusSig_0 and DbusSig_1 are present within the same block, they have different delays.

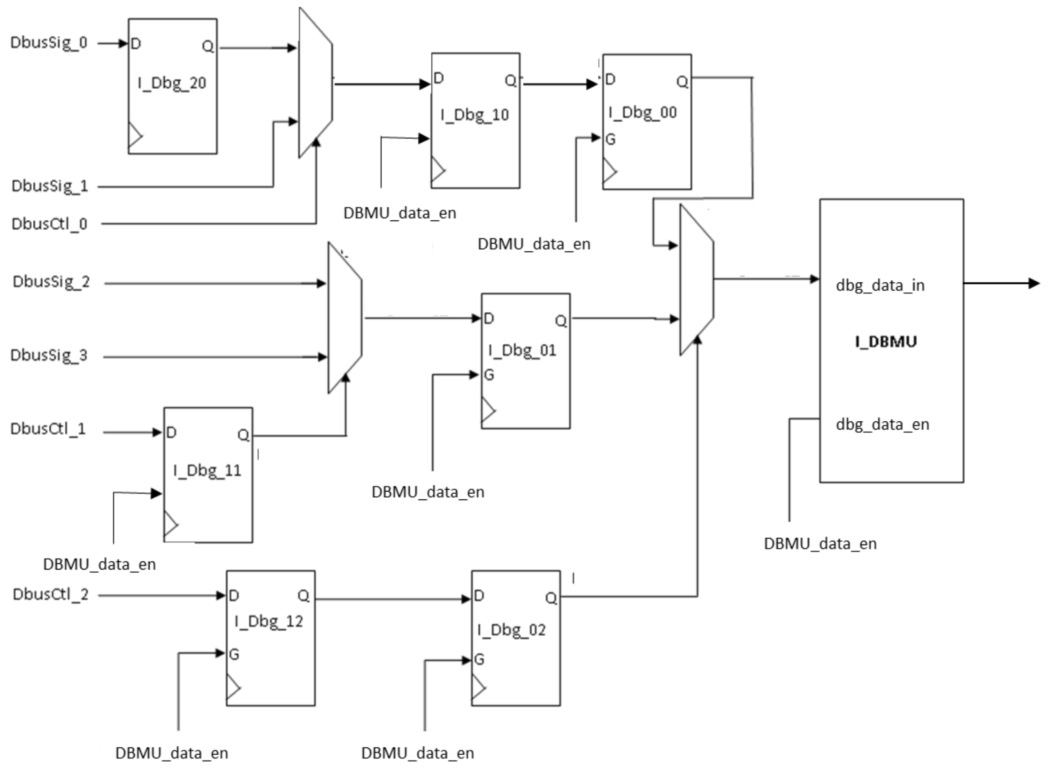

Figure 2: Block diagram of a debug-bus stop

2.2 Debug-bus verification

The intent of our debug-bus connectivity check is to verify that:

a. Clock, reset and enables have been correctly set up
b. All debug signals are connected to their respective DBMU outputs
c. Debug data is not corrupted because of unintended interaction with other datapath or control-path signals
d. The select value required to observe a debug signal matches with the specification

3. Verification Challenges

The debug-bus presents a unique set of challenges to the design and verification teams because of the cost of verification versus penalty for failure considerations. On one hand, the debug-bus is one of the most critical pieces of on-chip debug hardware available for silicon validation. The debug-bus functions much like a hardware assertion. Triggers can be defined as a function of one or more debug-signals. These triggers can be used to initiate specific actions. This allows us to gather design data whenever a specific state is reached. For example, if a bug causes deadlocks in a particular stage of the instruction pipeline, triggers can be setup to detect it and flush the pipeline to avoid the deadlock altogether or log relevant debug data to better understand the conditions leading up to the deadlock. Issues in the debug-bus directly impact bring-up time and effort. It forces engineers to rely on other analysis techniques that are more debug intensive, time consuming or less accurate.

On the other hand, being a design for debugging (DFD) feature, the debug-bus adds cost to a design but does not provide direct value to customers. The verification of the debug-bus is labor intensive, monotonous, and prone to mistakes if one decides to manually write directed tests to reach 100 percent connectivity coverage for all observable debug signals. Besides, the subset of signals that are used as debug signals undergoes changes constantly during the design process. This adds an additional burden on the verification infrastructure to keep up with design changes. Without a robust verification mechanism in place, these factors tend to push the verification of features like the debug-bus to a lower-priority. An easily deployable mechanism that requires little to no upkeep is an ideal solution for debug-bus connectivity checking.

3.1 Traditional verification of debug-bus

Traditionally, AMD has verified debug-bus connectivity using assertions. System Verilog assertions (SVAs) were manually written based on the specification. These assertions were then verified against the RTL through simulation. Stimulus was generated to enable the debug-bus and individually cover each assertion. Alternatively, assertions can be also generated from the RTL. In this case, in addition to verifying the generated assertions, we also have to ensure that the assertions correctly reflect the requirements outlined in the specification. These assertion-based verification flows are shown in Figure 3.

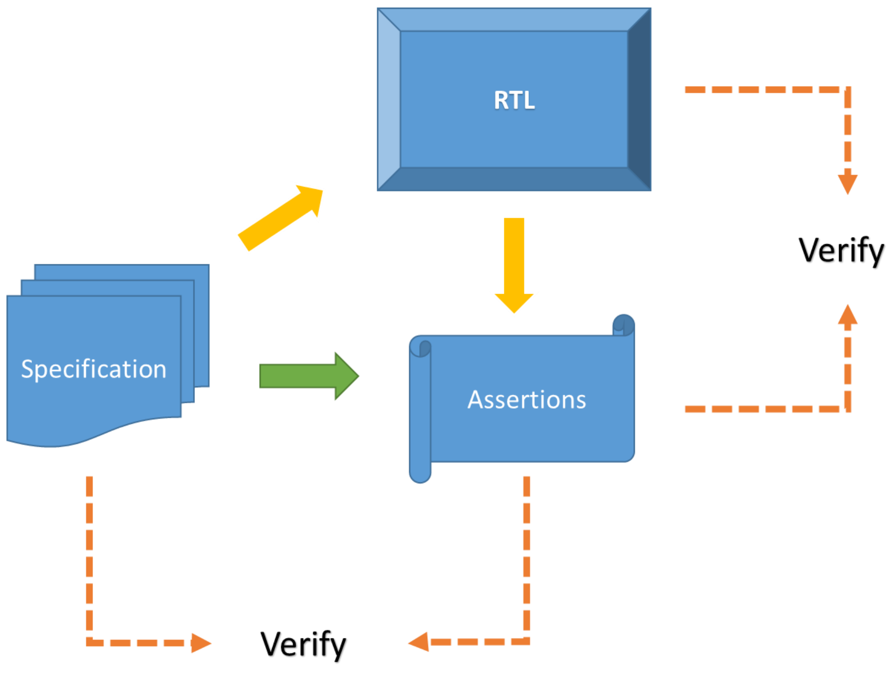

Figure 3: Assertion-based design verification

3.2 Drawbacks of simulation-based verification

From the designers' perspective, the biggest drawback of this approach was that majority of the debug-bus design effort was focused on creating and maintaining the assertions, not the RTL. Simple changes to the RTL such as adding a flop stage or control signal required them to update assertions. This often led to delays in implementing changes until the design was more stable (Oleg Petlin, 2014).

Verification engineers also had their share of issues with the overall debug-bus design flow. The assertions themselves were a source of problems because they were often poorly written, resulting in false positive simulation failures. Whenever design changes had to be made, debug-bus assertions had to be disabled until the RTL was stabilized. Not disabling the assertions could cause a large number of tests to fail when feedback to the design team was critical. So the assertions always trailed the final RTL release resulting in significant periods where the two would mismatch. Towards the end of the design cycle, gathering assertion coverage proved to be labor intensive and resource intensive. Directed tests had to be often written to close coverage holes. As a result, the maintenance cost was deemed too high and the turnaround time to verify the debug-bus was too long.

4. Formal verification-based debug-bus verification

4.1 Formal technology

Simulation-based verification testbenches have been the workhorse of design verification for several decades. Symbolic simulation, functional test generation, SAT, Boolean SAT, Arithmetic/Boolean SAT and ATPG-based methods are some examples of simulation-based verification methodologies. This is an event driven approach to the verification problem. It is extremely flexible and it can be applied to designs of all kinds and sizes. However, for multi-million and multi-billion transistor designs, it can be very difficult or impossible cover all possible cases, even within the legal input space. As a result, it can allow bugs to escape to silicon.

Formal verification has been gaining momentum as a complementary technology to simulation. It uses mathematical algorithms to prove or disprove properties that describe design behavior. SAT (Satisfiability), BDDs (Binary Decision Diagrams), interpolation, and induction are some examples of formal engines used to verify design properties. Unlike simulation, formal verification covers all the possible cases in the legal input space. Therefore, it is extremely efficient at targeting hard-to-reach states and preventing verification escapes.

4.2 Formal apps

A formal 'app' or application is a specialized or fully customized verification flow based on automated or script-based extraction of a pre-defined set of design properties from either a specification or RTL implementation. The extracted properties are verified by running formal verification tools.

In recent years, several formal apps have become available to solve a gamut of design verification problems. In addition to solving common design verification issues like register checking, connectivity checking and reachability analysis, they are being used for safety/security testing like fault analysis and detecting unauthorized accesses, for structural operation testing like clock-domain crossing checks, power management and X-propagation checks, protocol compliance, arithmetic precision and equivalence checking (Hogan, 2016).

4.2.1 VC Formal apps

Synopsys VC Formal currently offers several formal apps. They are:

- FPV : Formal Property Verification
  This is a property checking that serves as the foundation of all other apps. In this app mode, the user is expected to create all the properties that describe the requirements of a design functionality.
- AEP : Automatically Extracted Properties
  This app automatically extracts properties based on RTL structure.
- COV : Coverage Analysis
  This app analyzes code coverage using the same coverage goals as VCS simulation using formal techniques.
- CC : Connectivity Checking

This app checks the connectivity between user a specified source and destination for a specific enable condition.
SEQ: Sequential Equivalence Checking
This app verifies, on a cycle-by-cycle basis, if two RTL designs produce identical outputs given identical inputs.

In this paper, we describe the use of the Connectivity Checking app.

4.3 Motivation for using formal connectivity checking

Given the challenges of our traditional debug-bus verification flow, we decided to redesign it from the bottom up. Since the formal connectivity checking app provides the convenience of generating assertions directly from the RTL instead of having to write them manually, we decided to establish a consistent RTL coding style. This allows us to have a unified verification flow across all blocks connected to the debug-bus. This, in turn, significantly reduces the designers' burden of having to keep the assertions consistent with the RTL. The task of rewriting assertions is reduced to merely rerunning the assertion generation script. The assertions can be verified formally against the RTL using Synopsys VC Formal.

An analysis of the cone of influence of an average connectivity assertion revealed a modest complexity with a sequential depth from one to a maximum of ten. This proved the feasibility of using formal verification in order to exhaustively prove each assertion in the design and to eliminate the use of randomly generated stimulus and simulation coverage analysis (Oleg Petlin, 2014).

In general, the practice of generating assertions from RTL is of questionable value as the RTL serves as the specification and the implementation. To counter this, we compare the assertions against the specification to check for any anomalies or unintended dependencies. To further simplify this process, we convert the SVAs into a human readable form as discussed in the next chapter.

One of the primary advantages of automatically generated SVAs is the assertion quality. Every debug-bus assertion follows the same rules and coding style. Each assertion targets a single debug data signal, contains every flip-flop and mux control signal in the data path as part of the enable condition, and includes the exact number of clock cycles required for the data to make the transfer. In addition, the format for the condition, destination, and source is standardized resulting in a uniform product that is easy to read, reuse or reference in the future.

4.4 System Verilog Assertions(SVAs)

VC Formal generates a separate assertion for each datapath bit in the form of an implication. The SVAs take the general form

```
assert property((<select_condition>) [*N+1] |->
        (<destination> = $past(<source>, N))
```

where **`select_condition`** is the control signal value required to connect the source to the destination node and **`N`** is the path delay. The left-hand side expression is called the antecedent and the right-hand side expression is called the consequent. The implication construct specifies that the checking of a property is performed conditionally on the match of a sequential antecedent (Accellera, 2014). The implication evaluates to a true or false. If there is no match of the antecedent expression for a given source point, then the evaluation of the implication from that start points succeeds vacuously and returns true.

Consider the following assertions:

```
bind core_block core_block_cc_assertions I_core_block_cc_assertions ();
```

```
module core_block_cc_assertions;

//unit0.hier.DbusSigCnt[1][7] -->
unit0.unit_dbmu_top.DBG_LOCAL_DATA[7]

unit0_hier_DbusSigCnt_1_7__to__unit0_unit_dbmu_top_DBG_LOCAL_DATA_7:

assert property (@(posedge block1.CCLK) (

(block1.unit0.hier.DbusCtlSel[5] == 0) &&
(block1.unit0.hier.DbusCtlSel[4] == 0) &&
(block1.unit0.hier.DbusCtlSel[3] == 0) &&
(block1.unit0.hier.DbusCtlSel[2] == 0) &&
(block1.unit0.hier.DbusCtlSel[1] == 1) &&
(block1.unit0.hier.DbusCtlSel[0] == 0))[*3] |->
(unit0.unit_dbmu_top.DBG_LOCAL_DATA[7] ==
$past(unit0.hier.DbusSigCnt[1][7], 2)));


//unit0.hier.DbusSigCnt[1][6] -->
unit0.unit_dbmu_top.DBG_LOCAL_DATA[6]

unit0_hier_DbusSigCnt_1_6__to__unit0_unit_dbmu_top_DBG_LOCAL_DATA_6:

assert property (@(posedge block1.CCLK) (

(block1.unit0.hier.DbusCtlSel[5] == 0) &&
(block1.unit0.hier.DbusCtlSel[4] == 0) &&
(block1.unit0.hier.DbusCtlSel[3] == 0) &&
(block1.unit0.hier.DbusCtlSel[2] == 0) &&
(block1.unit0.hier.DbusCtlSel[1] == 1) &&
(block1.unit0.hier.DbusCtlSel[0] == 0))[*3] |->
(unit0.unit_dbmu_top.DBG_LOCAL_DATA[6] ==
$past(unit0.hier.DbusSigCnt[1][6], 2)));


//unit0.hier.DbusSigCnt[1][5] -->
unit0.unit_dbmu_top.DBG_LOCAL_DATA[5]

unit0_hier_DbusSigCnt_1_5__to__unit0_unit_dbmu_top_DBG_LOCAL_DATA_5:

assert property (@(posedge block1.CCLK) (

(block1.unit0.hier.DbusCtlSel[5] == 0) &&
(block1.unit0.hier.DbusCtlSel[4] == 0) &&
(block1.unit0.hier.DbusCtlSel[3] == 0) &&
(block1.unit0.hier.DbusCtlSel[2] == 0) &&
(block1.unit0.hier.DbusCtlSel[1] == 1) &&
(block1.unit0.hier.DbusCtlSel[0] == 0))[*3] |->
(unit0.unit_dbmu_top.DBG_LOCAL_DATA[5] ==
$past(unit0.hier.DbusSigCnt[1][5], 2)));

Endmodule
```

We have shown three assertions generated by the formal app for `block1` of the processor core. A comment identifying the source (debug signal) and destination(DBMU output) precedes each System Verilog assertion. For example, the first assertion covers the path from `unit0.hier.DbusSigCnt[1][7]` to `unit0.unit_dbmu_top.DBG_LOCAL_DATA[7]`. The path delay in all three assertions is two cycles. We convert these assertion into the following format to make it more readable.

(block1.unit0.hier.DbusCtlSel[5:0] == 6'b000010)

==

(DbusCtlSel[5:0] == 6'b000010) => DBMU[7] == unit0.hier.DbusSigCnt[1][7], Delay: 2

(DbusCtlSel[5:0] == 6'b000010) => DBMU[6] == unit0.hier.DbusSigCnt[1][6], Delay: 2

(DbusCtlSel[5:0] == 6'b000010) => DBMU[5] == unit0.hier.DbusSigCnt[1][5], Delay: 2

The assertions are grouped based on their selection condition. Each line clearly denotes the source, destination, selection condition, and path delay. This makes it very convenient to manually verify the generated assertions against the specification.

4.5 Formal verification flow for debug-bus

For performance reasons, connectivity checks were performed at the block level. In other words, each debug-bus stop, containing one DBMU, its control signals and its fan-in cone of debug data, was verified individually. Since a block-level check can identify the possibility of incoming debug data (from the previous stop) corrupting or depending on local debug-data, it is not necessary to repeat the checks at the core level. Such unintended interactions is revealed either through the SVAs or as failures of the formal proof.

The connectivity check methodology has been illustrated in Figure 4. We begin by setting up the tool. This includes enabling the FV application mode, setting the maximum runtime and telling the tool how to handle unresolved modules when reading in the design. All unresolved modules can be automatically set as blackboxes. Alternatively, we can selectively choose which modules we like to set as blackbox, including unresolved modules.

Once the tool is set up, the design is read in with any SVAs, if required. These could be manually written assertions or automatically generated from a previous run. We set up the clocks, reset and enable the debug-bus clock gaters using the `set_cc` command. We also identify the source and destination points of the datapath and the control-path signals. We set a limit on the sequential depth to limit resource usage and runtime. This also serves as the bounded depth.

After the initial setup, we begin the assertion generation process using the `gen_cc` command. This identifies all possible paths between the source and destination paths. Once complete, the result can be saved off as System Verilog assertions. The tool also reports if no path is found to the destination from a source point.

The formal analysis or ‘simulation’ is run from a user-defined initial state. The initial state of the formal model, also known as the reset state, represents the value held in all state elements in the design. There are two ways to specify reset. Either using the `create_reset` or `sim_force` and `set_constant` commands. We use the latter. Once the correct simulation state is set, running `check_cc` begins the formal proving process. The reset values assigned at the beginning are held constant during the formal analysis. This way, properties are not falsified by reset being asserted randomly during formal analysis. Running the formal proofs verifies all properties with usage attribute set to *assertion*. It obeys all constraints specified by properties with usage set to *assume*.

VC Formal marks all checked assertions as proven or falsified. A falsified assertion can be debugged by asking the tool to generate a test which causes the assertion to fail. This is very useful to identify issues in the tool setup, reset state and design bugs. If an assertion could not be tested (because the antecedent could not be satisfied), it is marked as vacuous. If an assertion can be hit, but cannot be proven or falsified, it is marked as non-vacuous.

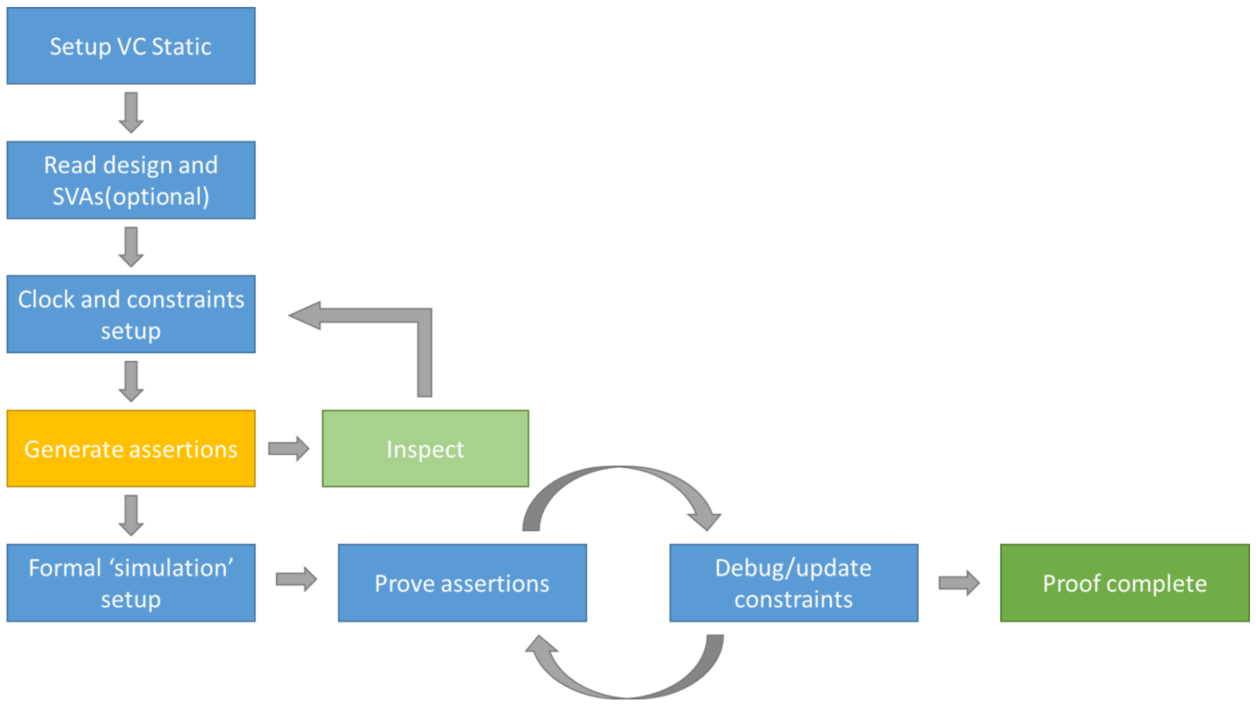

Figure 4: Debug-bus connectivity check verification methodology

5. Results

5.1 Verification challenges

VC Formal does not allow combinational logic other than muxes in the datapath. Since debug signals are drawn from different sub-blocks, the AND-OR gates that implement the mux are spread out across the design hierarchy. This is not an issue if the combined logic is functionally equivalent to a mux. However, because the blocks can lie on different physical tiles, staging flops are often inserted to meet timing requirements. As a result, AND-OR gates may not always form a mux and are reduced to distributed clusters of combinational gates. For example, in Figure 5, there are the two flops attached the legs of the OR gate. So this AND-OR combination no longer forms a mux.

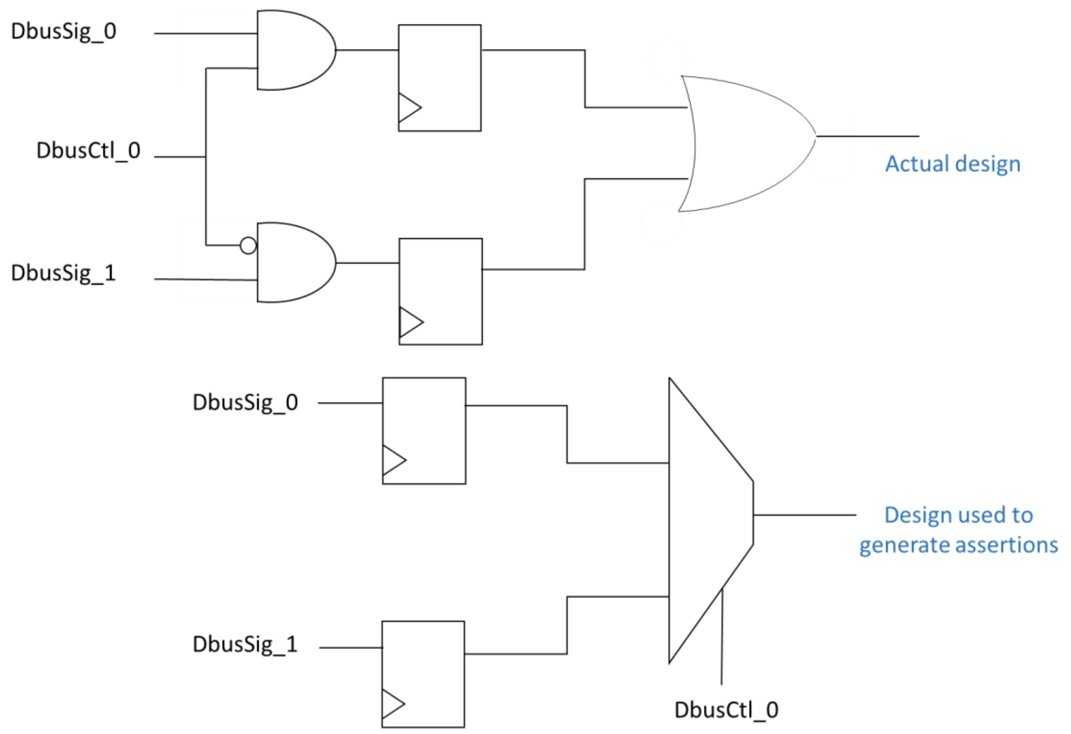

Figure 5: Generating assertions from RTL when datapath contains combinational logic

As a workaround, we break down the check into two steps. In the first step, we read in the design with a mux in place of the selection logic, as shown in Figure 5. In the second step, we prove the generated assertions on the original design. Since the modified design is not functionally equivalent to the original design, additional constraints are required to complete the formal proof. When the legs of the mux have unequal delays, there is additional verification complexity.

We had to work with Synopsys engineers to correctly generate assertions for muxes whose select signals were one-hot encoded.

5.2 Results

A summary of the connectivity checks performed on the four blocks show in Figure 1 have been summarized below in Table 1.

Table 1: CPU block-level debug-bus connectivity check results

Block	Paths verified	Enables checked	Assertion generation		Assertion proof	
			CPU time (s)	Peak memory (MB)	CPU time (s)	Peak memory (MB)
I-cache & BP	532	67	60	1166	204	29504
Decode Unit	911	115	192	992	512	19915
Load-Store Unit	928	116	188	1031	538	19851
Execution unit	1085	1074	130	1285	440	15166

Each path corresponds to an assertion or one bit of a debug signal. Each distinct enable value is counted separately. For example, if there is a one-bit mux select signal, it counts as two enable values if both inputs of the mux are being used.

5.3 Connectivity issues identified

Synopsys engineers extended the verification use of the formal app by adding structural linting capabilities. RTL structural issues were identified as debug-bus connectivity violations. While debug signals can cross block boundaries to reach a debug-bus merge unit, it is considered a path violation if the debug signal cannot reach the debug-bus within the functional unit. Name violations are triggered if either debug data or control signals are encountered which do not conform to the naming conventions. Control violations are triggered if the fan-in traversal from either a mux or flip flop enable port does not reach a control signal when the maximum search depth is encountered. Lastly, any combinational logic which affects the debug data signal is not allowed and will be flagged as a combination logic violation.

In addition to these issues, the debug-bus connectivity check uncovered two RTL bugs before product tape-out. In the first case, several debug signals buses were concatenated into a 200-bit wide debug signal. Some of the constituent debug signals were padded with zeros to make the signals byte-aligned. For instance, if a signal was only 6 bits wide, bit index 7 and index 6 were tied to zero. Because of accidentally padding an additional zero, the total bus-width came up to 201 bits. This meant that the most significant bit remained unconnected. This issue was identified during assertion generation because the tool could not find a path from this wire to the debug-bus. The second case was a similar issue, where 104 bits were connected to a 100-bit wide debug-bus. It is interesting to note that these issues do show up as bus-size mismatches in RTL lint, but can go unnoticed.

6. Conclusions

Formal verification is becoming increasingly user friendly. Assertion-based formal verification is a more reliable approach to exhaustively testing FSMs and control logic where the behavior can be accurately specified. Designs that are too big can be partitioned into more manageable blocks. The block can then be chained together into a sequence such that the first block drives the second block, the second block drive the third block, and so on. Each of the blocks can be verified individually and with constraints specific to that block.

Traditionally, debug-bus verification at AMD was performed through assertion-based simulation. Simulation-based verification of the debug-bus puts a lot of burden on the designers to write and maintain SV assertions. It requires co-ordination between the verification and design teams to avoid false failures because of incorrect and outdated assertions. Assertion coverage has to be collected to verify if all paths have been tested. Directed tests have to be written to target debug signals that are not verified by random stimulus.

In this paper, we discussed how we used VC Formal to perform debug-bus connectivity checks on four blocks within our microprocessor core. Each of these blocks within the core were verified individually with a separate set of constraints. The flow ensured that the clock, reset, and control logic were correctly setup and all debus signals could be observed over the debug-bus without any mutual interference. By adopting this flow, we could guarantee that all debug signals had been verified. There were two instances where newly added debug signals did not have a path to the debug-bus. We were able to identify both of them and provide immediate feedback to the designers.

The overhead of making the switch from simulation-based verification to formal verification was a one-time effort of setting up the tool. Even though this took a significant amount of time, since this flow could be reused across projects with only minor changes, this cost was justified.

This approach has proven to be reliable and convenient and it has encouraged us to explore other design verification problems that could be solved using formal verification. For our future projects, we would like to further reduce the turnaround time and include this in the suite of tests that are used to qualify check-ins.

7. References

Accellera. (2014, May 05). System Verilog 3.1a Language Reference Manual. Accellera's Extensions to Verilog®.

Hogan, J. (2016, February 2). *Jim Hogan on how "this is not your father's formal verification"*. Retrieved from Deep Chip: http://www.deepchip.com/items/0558-01.html

Oleg Petlin, A. M. (2014). Formal Verification of Debug Bus Connectivity: A Formal App Case Study. *SNUG.* Boston.