Accelerated, High Quality SoC Memory Map Verification using Formal Techniques

Cletan Sequeira, Rajesh Kedia
Texas Instruments India Pvt. Ltd.
Bangalore, India
c-sequeira@ti.com, r-kedia@ti.com

Lokesh Babu Pundreeka, Bijitendra Mittra
Cadence Design Systems, Inc.
Bangalore, India
lokeshp@cadence.com, bijitm@cadence.com

Abstract— *In today's world, SoCs are typically designed by integrating several existing in-house Intellectual Properties (IPs) and/or third party IPs provided by external vendors. Having many of these IPs and memory blocks integrated together forms a complex SoC memory map with multiple masters trying to access multiple slaves (IP or memory) simultaneously. The functioning of each of these slaves is tightly dependent on the correctness of the memory map. Hence it becomes critical to implement a correct SoC level memory map. With the sheer number of slaves, complex access policies and checks on valid/error responses for different types of address range implementations, verification of a SoC level memory map becomes an uphill task. In this paper, we present a novel approach towards verifying such a SoC level memory map using Formal techniques. We talk briefly about the previous work that has been done using traditional verification techniques and their shortcomings in verifying SoC level memory maps. We give a short introduction to the SoC design followed by a discussion on our strategy to use Assertion Based Verification IPs along with custom assertions to achieve comprehensive verification results. We also discuss issues and challenges faced in order to scale Formal verification to SoC Level. Finally, we share the results from the successful application of our methodology to verify the entire memory map on a complex SoC design.*

Keywords—Formal verification; Memory Map; SoC; ABVIP

I. INTRODUCTION

Now-a-days, System On Chips (SoCs) are being increasingly deployed in large number of applications and systems to implement automation rendering ease and convenience in many human activities; prime examples being smart mobile phones, automotives, medical electronics etc. This makes design implementation a fairly difficult task. With larger product space and product revisions, comes the requirement for larger feature integration on smaller die-sizes, smaller design turnaround times and lower power consumption. To address these needs, a typical SoC design contains several existing/legacy in-house Intellectual Properties (IPs) and/or third party IPs licensed from external vendors. Each IP in a SoC consists of multiple registers referred to as memory mapped registers (MMRs) that are used to configure/control/observe functionalities associated with that block. Integration of multiple embedded memory blocks is also increasing with the increasing complexity of the SoCs. The address range to access these memory mapped components for each IP and embedded memories is mapped into the SoC level addresses creating a SoC level memory map. In a typical functional scenario, having multiple masters trying to access multiple slaves (IP or memory) simultaneously, it becomes critical to implement a correct SoC level memory map so that the access requests intended for a certain slave IP always reaches the intended one and does not affect any other. The overall task of SoC memory map verification becomes complex due to the presence of large number of slaves, bus protocols having complex access policies and checks on valid/error responses for different types of address range implementations. Current approaches to verify such memory maps using directed CPU based assembly/C test-cases are time consuming and not sophisticated enough to catch critical bugs like address aliasing, protocol violation, etc. Formal Verification (FV) has always been efficient to address these types of verification problems and have been successfully deployed at sub-blocks like IPs, decoders, bridges, etc. thus gaining increased confidence over just C or assembly based tests. However, once these sub-blocks are integrated at the top level, it introduces a different level of complexity caused by unimplemented/unused/mirrored address spaces; interconnect logic, multiple masters and slaves and their interactions. Hence there was a strong motivation to deploy FV at the SoC level (with all components integrated) and exhaustively verify the memory map in its entirety. Our proposal involves using Assertion Based VIPs (ABVIPs) along with memory map specific assertions to address this verification problem using FV. The beauty of this method is that most of the memory map verification can be done using assertions without the need for complex/long SoC tests. This work significantly improves the confidence on the memory map verification and automation developed around it makes it easily portable and configurable for changes in the next devices.

II. PRIOR WORK

A. *C or Assembly language based functional tests*

Traditionally, SoC memory map have been verified using system level tests written in C or assembly language which exercise read and write to different address locations through on-chip masters (CPU/DMA). While this approach uses a very realistic and close to actual use case scenario, covering the complete memory map using this method is impractical and requires a huge effort on test development and runtime. Moreover, we may not be able to control bus masters to exercise all aspects of protocol. For example, we may not be able to precisely exercise a back to back read-write-read kind of a sequence on the bus through CPU since it is dependent on CPU pipeline and other internal states. Thus, we may miss many of the protocol related violations with this approach. Also, being able to run testcases in this environment will require the complete SoC hookup and the verification environment to be ready which will take its own time.

B. *Formal Verification (FV) for individual blocks + SoC tests*

To tackle the limitations posed by C or assembly language based SoC tests, there has been some work on deploying formal verification towards the individual components that build the memory map. Components such as bridges and decoders are verified at unit level using FV and once fully verified, they are used in SoC level tests. This approach provides much improved quality and confidence on the memory map but still lacks in assuring an exhaustive coverage on hookup of these individual blocks in the system. While each of the individual memory map component like bridges, decoders, memory mapped IPs are verified in entirety, there could be bugs introduced during the hookup of these components and the traditional SoC tests may not be sufficient to cover all such scenarios. As an example, consider one of these scenarios: “Writing to a reserved address space shouldn’t cause any impact to any functional address”. Verifying it using functional method requires writing a complex testcase having a flow similar to the below:

a. Initialize the complete range of implemented address space with some known values.

b. Write random values to all the reserved addresses.

c. Read back the implemented addresses and check that they retain their values from step-a.

Now-a-days, address space for microcontrollers spans across 32-bit or 64-bit range making implementation of such a testcase practically impossible. Hence there have been approaches where engineers don’t cover the complete address space. Rather they use various patterns like walking-1 and walking-0 to exercise all address bits, one at a time, providing a fair compromise between practicality and quality. But still, exhaustiveness is lacking in this approach as well.

C. *Using Emulation platform as accelerator*

Emulation platforms like Field Programmable Gate Arrays (FPGAs) have been used to target scenarios which are runtime intensive on simulation. Complete SoC memory map verification for today’s class of designs fall under such category. Hardware platforms have been found to be efficient for exercising such scenarios in a fairly exhaustive manner, but they come with their own limitations. Setting up the hardware platform and getting it to work happens late in the design cycle as compared to the simulation environment. Hence, if there is any bug caught on emulation platform, depending on the current state of design development, the cost of fixing the bug may be very high. Secondly, there may be some bugs which are sequence dependent and it may not be possible to exercise all possible combinations of sequences on a hardware platform too.

Given the above cited limitations with SoC memory map verification, there was a strong motivation to look for alternative techniques which could lead us to a better quality of verification within the given constraints of run-time, schedule and effort. Clearly, Formal Verification (FV) comes as a possible thought given its exhaustive nature. The power of FV lies within the fact that it explores the complete state space within the given constraints and comes up with a final PASS or FAIL status. This gives much more confidence than just covering some selected patterns as in directed/random tests. The initial concern on deploying FV here was the effort in developing the drivers for various masters and slaves, but availability of certain verification IPs from Cadence solved this concern and served as another factor in being able to pursue FV for memory map verification.

III. INTRODUCTION TO OUR SOC

Our SoC consists of a complex bus system comprising of 4 AHB masters (3 buses from Cortex-M4 (CM4) and one from the DMA) and several slaves which interact with each other through a common AHB fabric. Each of the master interfaces consists of a 32-bit address bus spanning across an address range of 4GB. The slaves available in our SoC follow either AHB or APB or TI custom protocol. There are bridges and decoders between the AHB master ports and some of the slaves as shown in the Figure -1 below. The AHB fabric follows a fixed priority arbitration scheme whenever more than 1 master access the same slave.

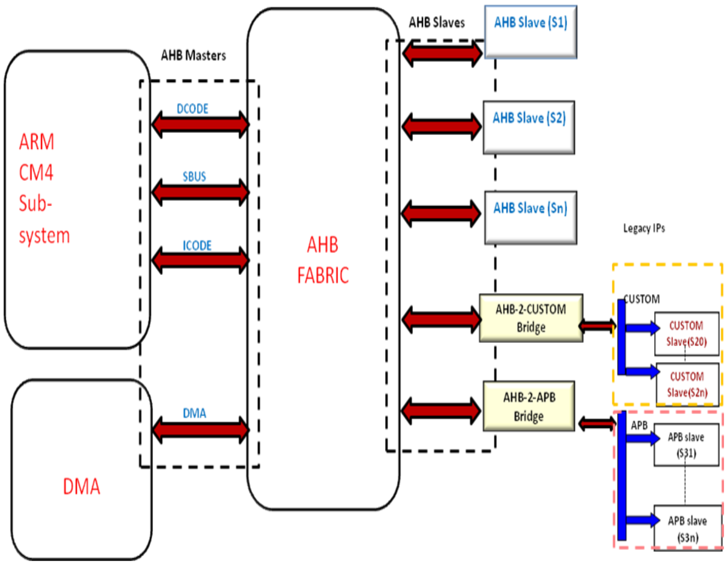

Fig.1. SoC Level Memory Map Block Diagram

The memory map in our SoC consists of the following types of address spaces:

1. Slaves which are accessible only by CPU and not by DMA (Error response when accessed by DMA).
2. Slaves that are accessible only by some CPU buses and gives an error response when accessed by others.
3. Slave that is mirrored at different addresses for 2 different masters.
4. Reserved address within the slaves which have no effect on write access and read zeros.
5. Illegal spaces in between the slaves which give an error response on access.
6. Slaves which allow access of certain size only and give error response for invalid size transactions.

IV. Proposed Methodology

In this section, sub-sections A and B give a background on the ABVIPs and the environment, while C and D focuses on the actual memory map verification. Sub-section E discusses various challenges that we faced during this work.

A. *Introduction to ABVIPs from Cadence*

Cadence provides off-the-shelf Assertion Based Verification IPs (ABVIPs) for standard ARM protocols like AHB (master and slave), APB, AXI that checks for the compliance to the ARM AMBA protocols. These ABVIPs served as foundation blocks in our verification methodology. A correct hookup and control of these ABVIPs can be used to verify protocol compliance of masters and slaves in a multi-master/multi-slave environment. It provides constraints on the input side of the device and assertions that check the outputs are as expected. Here the same property can act as a checker or BFM as explained in Figure 2 below.

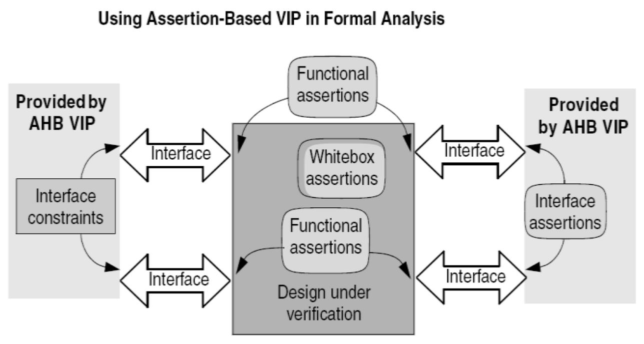

Fig.2 Using ABVIPs in Formal Analysis

Example: We define a property:

prop1: ((!hresetn_i) → *(hready));*

which is an AHB protocol specification. The property mentions that whenever hresetn_i is ‘0’, then hready is ‘1’. Now, if we say “*assert prop1*”, then it becomes an assertion and works as a checker to verify the hready behavior of slaves. Whenever hresetn_i is ‘0’, it will check if hready is ‘1’ else it will fail. On the other hand, if we say “*assume prop1*”, then it becomes a constraint and controls the driving of hready from the AHB slaves. Whenever hresetn_i goes ‘0’, it will force hready to ‘1’.

Based on the above example, if we control the slave and master properties precisely, we can make them act either as a driver or a checker.

B. *Hookup of ABVIPs to each of the master and slave*

1. The slave and master ABVIPs are clubbed together to form a complete set of AHB/APB properties.
2. Bind these combined AHB and APB ABVIPs to each AHB and APB slave respectively. For slaves following TI custom protocol, an ABVIP is coded in-house and hooked up at each custom slave. To ease the hookup, a perl script was written to automate it based on an input file.
3. The properties contained in the VIPs are controlled precisely (either as constraints or as assertions) depending on whether it is bound to a slave or master using IFV Tool Command Language (TCL) interface. This is also automated through scripts.
4. With the above steps, we have the complete setup ready to verify the protocol compliance of each of the interconnect logic which form a part of the memory map.

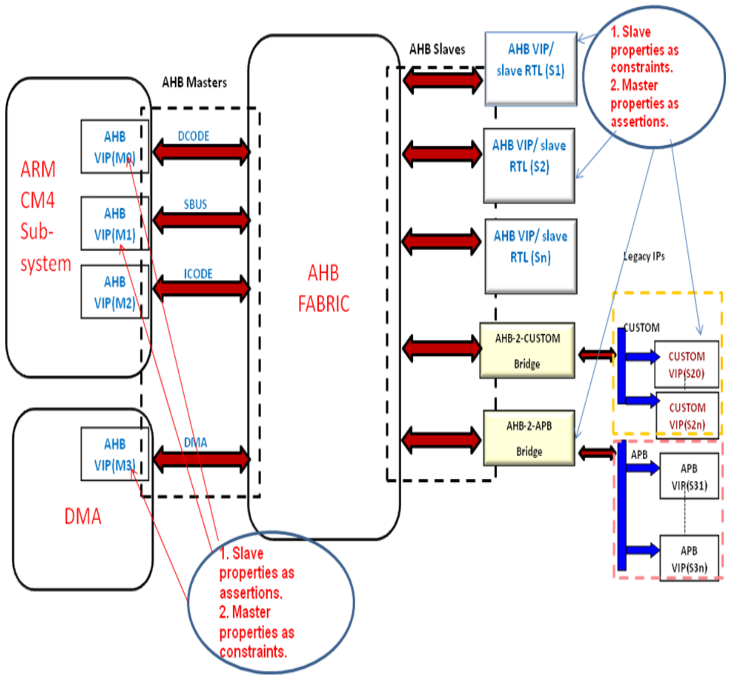

Fig.3. SoC Level Memory Map Verification Environment

We are reusing the golden RTL file directly in our verification flow and no changes are done to the golden RTL for hooking up the ABVIPs to each of the master or slave. The ABVIPs are bound to various masters and slaves using vunit construct available in Property Specification Language (PSL). There is a separate PSL file written for the hookup. An example of the vunit based ABVIP binding is given below for an AHB slave:

```
vunit vunit_abvipfv_master1_ahb_top (top_entity) {
--   Define local parameters
localparam ahb_master1_ABUS_WIDTH = 32 ;
localparam ahb_master1_AHB_MAX_WAIT_STATES = 3 ;
…
-- Hookup the VIP to signals inside the top_entity
ahb_full_monitor  ahb_full_monitor_MASTER1 (
 .hclk(HCLK),
 .hresetn(HRESETn),
 .haddr(u_CORTEXM4.HADDRD),
 .htrans(u_CORTEXM4.HTRANSD),
 .hwrite(u_CORTEXM4.HWRITED),
 .hsize(u_CORTEXM4.HSIZED),
 .hburst(u_CORTEXM4.HBURSTD),
…
);

--  Map local parameters to the VIP
defparam    ahb_full_monitor_MASTER1.ABUS_WIDTH    =
ahb_master1_ABUS_WIDTH;
defparam    ahb_full_monitor_MASTER1.DBUS_WIDTH    =
ahb_master1_DBUS_WIDTH;
…
}  //end of vunit definition
```

C. *Additional assertions to verify the memory map*

On top of the default protocol checks available as part of the ABVIPs, we wrote additional assertions to check for the SoC specific memory map features. The assertions were intended to cover the below features:

1. **Write access to each slave from each master at designated addresses:** We constrain each master to drive different pre-defined write data for each slave address range. On slave side, we have assertions which check that if there is a write access to the slave, it must contain only the pre-defined write data for that slave. This way we comprehensively confirm that the particular slave can be written only within the selected address range.

   Property:

   a. *({psel_i && penable_i && pwrite_i}|-> {pwdata_i == apb_slave_write_data})@(posedge pclk_i);* -- this property checks that whenever there is a write access on an APB slave, then pwdata_i should match the predefined write data for that slave.

   b. *({(hsel_i && hready_global_i && htrans_i[1] && hwrite_i);!hready_global_i[*]} |=>{(hwdata_i == ahb_slave_write_data)}) @(posedge hclk_i);* -- this property checks that whenever there is a write access on an AHB slave, then hwdata_i should match the predefined write data for that slave.

2. **Read access from each slave to each master at designated addresses:** We constrain each slave to drive different pre-defined read data. On master side, we have assertions which check that if there was a read access to a particular slave, on hready going high, the read data must contain the pre-defined read value for that slave. This way we comprehensively confirm that the particular slave can be read only under the selected address range.

   Property:

   ({AHM_Slave_Valid_Address_active && !hwrite && hready_i && htrans[1]; !hready_i[*];hready_i}|-> {hrdata_i== AHM_Master_Slave0_read_data}) @(posedge hclk_i); -- this property checks that whenever there is a read access from an AHB master to a particular slave, then hrdata_i should match the predefined read data corresponding to that slave.

3. **Mirroring of certain slaves at multiple addresses for different masters:** Different masters can access certain slaves at different addresses. e.g. Master M1 can access slave S1 in address range A-B while master M2 can access the same slave S1 in address range C-D. To verify this implementation, the read and write assertions described above were duplicated and different address ranges were specified for each master. This was taken care by the IFV tool TCL commands.

   TCL commands:

   constraint –add -pin dbus_MASTER.slave1_Valid_Address_LOW $slave1_mirror_start_add

   constraint -add -pin sbus_MASTER.slave1_Valid_Address_LOW $slave1_actual_start_add

4. **Reserved space checks:** In case of a reserved space, we need to ensure that writes are ignored and read data is always 0. This gets checked by having an assertion at master side which checks for the read data to be “0x00000000” for such addresses. The writes being ignored is checked by the slave assertions which expect only pre-defined write data on the bus.

   Property:

   ({AHM_Slave_Valid_Addres_active && !hwrite && hready_i &&htrans[1]; !hready_i[*];hready_i}|-> {hrdata_i== 0x00000000}) @ (posedge hclk_i); -- this assertion is similar to the read assertion described above with the read data being fixed to 0x00000000.

5. **Error response checks for illegal space between slaves:** There is an assertion at master side which checks for the response line to indicate error whenever ready goes high if the access was to an illegal address range.

Property:

{master_error_resp_active && hready_i && htrans[1]} |=> {hresp_i && !hready_i;hresp_i && hready_i})@(posedge hclk_i); -- this assertion checks for hresp_i signal to be '1' in case of illegal address ranges.

6. **Error response checks for invalid size access:** The assertion is similar to the earlier assertion for error response but this becomes active only when the master issues an access of invalid size.

   Property:

   {AHM_Slave_Valid_Adress_active && (hsize[1] | hsize[2]) && hready_i && htrans[1]; !hready_i[*];hready_i} |-> {hresp_i})@(posedge hclk_i); -- this assertion checks for hresp_i signal to be '1' in case of accesses of certain sizes for certain address ranges.

7. **Priority of each master when they are accessing the same slaves:**
   - Drive different write data from different masters.
   - Constrain *htrans* as sequential or non-sequential access for the master with highest priority.
   - Constrain all the masters to drive same slave address.
   - At slave side, check that write data from highest priority master is seen.
   - Repeat the above steps for all other masters by constraining the htrans of higher priority master to idle.

   This is implemented using a combination of the PSL properties and IFV TCL commands. The properties behave as checkers for which particular master is accessing the slave, while the TCL commands controls which masters are active and which are not.

D. *Putting it all together*

All the above checks for memory map require manual effort in coding the assertions, but once done, they are portable across different devices. They are also easily extensible for addition or removal of master or slaves in the system.

Having all the assertions coded and ABVIPs hooked up in the system, we are ready to proceed with the verification of the system level memory map. The design compile is done by black-boxing the modules which don't form a part of the memory map. At the first step, only the ABVIPs, bridges, bus-fabric and decoders are taken as real design. Incisive Formal Verifier (IFV) from Cadence is used as the formal verification tool. Firstly the protocol assertions are run and then the SoC memory map related assertions are run in IFV.

With this approach, we check the following critical functionalities of the SoC which are difficult to achieve using conventional approaches based on C/assembly tests,

1. Adherence of interconnect logic (bus-fabric, decoders, bridges) to AMBA (AHB and APB) and custom protocols.
2. Connectivity of bus fabric with all the masters and slaves.
3. Functionality of bus-fabric, bridges and decoders.
4. Accessibility and address mapping for each slave from various masters.
5. Priority checks for each master when they are accessing the same slave.
6. Mirroring of certain slaves at different addresses for different masters.
7. Error response checks in-case of invalid addresses.
8. Address aliasing.

E. *Challenges faced in scaling up FV to SoC Level*

While FV makes many things easier by not demanding a sophisticated verification environment and being able to exhaustively verify the design, it poses certain challenges during the initial bring-up, especially at SoC level due to the larger size of the design compared to unit level. The team involved in this work had prior experience on FV with IFV tool and PSL assertions and hence we are not listing the common issues related to syntax, etc. which might be seen by beginners. Instead we focus on the issues specific to porting FV to SoC level. Mentioned below are some of the challenges and issues faced by us during the development of the flow.

1. **Limit the state-space of the design**: In an SoC, there are lots of modules present but many of them may not have any influence on the functioning of the memory map. Hence a common practice in such cases is to black-box those modules so that the tool just sees the modules needed for intended functionality and thus limit the overall design size that the tool needs to analyze. Following similar practice, we selected only the modules which comprise the memory map i.e. bridges, decoders, bus-fabric, clock and reset controllers and all others were black-boxed. But eventually we ended up removing many modules from black-box list during the bring-up. An example reason being – "Even though we had constrained the clocks and resets at their source points, they were passing through some other modules for test/debug purposes or for some functional overrides and were causing assertions to fail." These modules needed to be read-in as actual RTL and not as black-box to get the flow working. Overall, it was an iterative process to find out the right set of modules for black-boxing.
2. **Choosing the right engine from the IFV tool**: IFV tool internally has many engines (algorithms) for solving the formal properties and each of them works better for certain kind of logic or assertions but not for all. Initially, we observed huge runtime and non-convergence due to use of default engines and it was an iterative process to find out the right set of engines

which works well with all assertions in the context of our design. We tried using the tool option (auto_dist) to automatically distribute each assertion for all engines and get the result from the best engine for that assertion; but it required multiple parallel jobs to be invoked on compute farm and was limiting the overall turn-around time due to limited slots per user. Our approach to select the engine involved running the non-converging assertions with the auto_dist option and list down the winning engine. We did this for few such properties and chose the engine that was chosen most. Using this engine as default enabled us to see more properties converging faster. Doing it iteratively, we chose "sword", "bow" and "hammer" [2] as the default engines.

3. **Design Constraints:** Certain constraints need to be defined for the memory map to be successfully seen by all masters. These constraints could be to disable test logic or reset or to prevent power down, etc. The device was given an initial reset pulse through tcl interface to reset the internal states and we added some of the constraints initially based on our understanding of the SoC and discussions with designer. But during assertion cleanup, there was significant time spent in refining the constraints as and when there were false failures seen. We didn't observe any time-out due to the missing constraints, and mostly it led to assertion failures which could be easily debugged using the counter-example provided by the tool.

The benefits and power of FV kept us focused and motivated to work on this flow despite the above challenges. Finally, we were able to solve these issues and achieve the verification comprehensiveness within acceptable time-frame.

V. AUTOMATION

As mentioned earlier, there has been automation done wherever possible to maximize the reuse and enable a faster turn-around time for future projects. The memory map information is captured in a format as shown below in an excel sheet.

MASTER	SLAVE	START ADDRESS	END ADDRESS	SIZE	Sel_Path
M2#M0	S31	SA31	EA31	SZ31	H1.pin1
M2#M0	S41	SA41	EA41	SZ41	H2.pin2
M0#M1#M3	S0	SA0	EA0	SZ0	H3.pin3
M3#M1	S1	SA1	EA1	SZ1	H4.pin4
M2	S1	SA1m	EA1m	SZ1m	H4.pin4

Table-1: Example of the input file for automation

The columns in the above table can be described briefly as:

- MASTER – Mentions which masters can access a particular slave.
- SLAVE – Mentions the particular slave being accessed by the masters.
- START ADDRESS – Mentions the valid start address for a particular slave.
- END ADDRESS – Mentions the end address for the slave.
- SIZE – Mentions the size of the slave in terms of address range (while this information is redundant given the start and end address, some implementation aspects require us to provide this information).
- Sel_Path – Mentions the hierarchical path for the select pin of the slave, needed for the automated binding of ABVIPs.

In the above table, the two highlighted rows show an example of slave S1 mirrored across different address ranges for different masters.

This Excel file containing the memory map information as described above is parsed by scripts which generate the PSL property files for binding the ABVIPs to the master and slaves. It also generates the appropriate TCL files which cover various memory map parameters that are used in the assertions.

VI. RESULTS

We caught three critical bugs in the bridge using this approach, out of which one was a corner case bug related to bridge functionality which would have been nearly impossible to catch using traditional approaches. We recorded a huge runtime advantage in comparison to the simulation-based flow. While it took just 16 hours to run ~1300 protocol and memory map assertions, it would have been in terms of days to complete the C/assembly based simulations that too with a limited coverage of features. Since FV does not need a testbench setup, we could catch bugs early during the development cycle and it worked as a first level exhaustive sanity check before releasing the RTL for further verification activities.

The summary of assertions for our SoC using this flow is listed in Table-2 below. While all the memory map assertions passed, there were some of the protocol assertions that showed as failing. These were analyzed and waived-off since they were some of the optional implementations. We also see in the results that including the protocol assertions increase the run-time significantly compared to without having them. This is because of the complexity of the assertions and state space explosion to prove them and it required large compute time for proof. But because the total run-times being within acceptable limits, we chose to get them passing rather than excluding them.

	Total Assertions	Assertions Passed	Assertions Failed	FV Run-time (hrs)
Protocol checks + memory map checks	1300	1268	32	16
Only memory map checks	500	500	0	5

Table-2: Summary of various assertions for SoC

Flow Deployment	Approximate bring-up effort
For the first device	~6 weeks
For subsequent devices in family	< 1 week

Table-3: Flow bring-up effort for various devices

A. *Examples of bugs that were caught using this flow*

1. "*If the first access from the CPU is to an address space that falls within the range for the custom bridge, then the access may get lost and not reach the slaves.*" – If we deployed only the traditional verification approaches, this bug wouldn't have been caught because of multiple reasons. Firstly, it is a corner case to be thought of as a scenario in the planning. Secondly, CPU has its own way of powering-up and first few accesses from CPU are not controllable. Hence, even if we have such a scenario in our plan, it is not possible to be exercised.

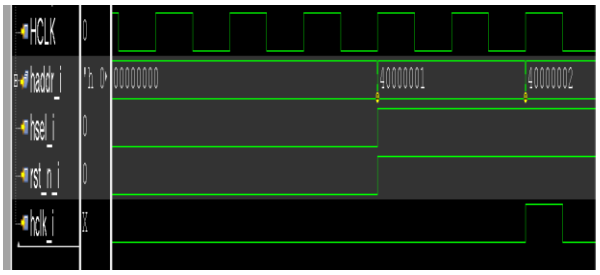

Fig.4. First transaction getting missed due to missing pulse

2. "*Inconsistency between access to odd versus even byte through bridge – The values for the unused bits during odd byte access and even byte access was not consistent. In one case, it was taking all 0s and in other case, it was duplicating the data.*" – Again this is a bug which doesn't have any direct functional impact but may have other implications like active power. This would have been impossible to catch with traditional approaches.

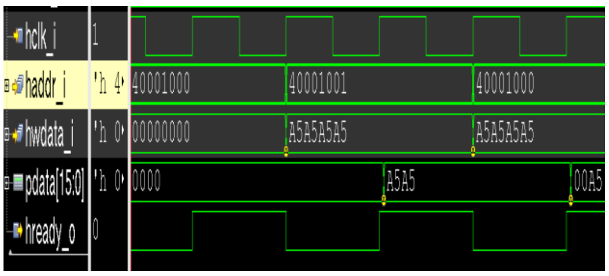

Fig.5. Inconsistency between even/odd byte accesses

3. "*There was a protocol violation in the way decoder was driving certain signals for the APB*". The default protocol assertions from the ABVIPs were able to catch this bug.

VII. ADVANTAGES

1. The ABVIPs binding with the appropriate module is done using vunit construct which allows this flow to work directly on the integrated RTL without any local changes.
2. Memory map checks can be done much early during the development phase which helps to find early bugs in decoders, bridges, interconnects, etc.
3. It gives more confidence on the verification quality because of the exhaustive nature of formal verification.
4. No disk-space issues because debug in IFV doesn't require waveforms to be dumped unlike in simulations.
5. Debug is quite simple and fast – the exact (or sometimes close) cone of signals related to the assertion get added to the waveform automatically.

VIII. LIMITATIONS

1. While the control space (address decoding, priority control, etc.) is exhaustively covered, a major limitation with the proposed methodology is that it doesn't cover the data space exhaustively because of the use of pre-defined data patterns for each master/slave.
2. This flow assumes that the individual IPs internal register map is verified at unit level and doesn't exercise the internal register map. This is not done due to capacity limitations of the tool and the effort in finding more constraints when actual slave RTL is integrated. Also, the focus on this work currently was on finding the SoC integration issues.
3. The pre-requisite for this flow to work is the availability of ABVIPs. For any IP being added in the system which doesn't have a corresponding ABVIP available, this flow demands for an extra effort in the development of the VIP.
4. We observed very large run-times for some of the complex assertions, which were related to protocol verification. Though most of them were able to converge for our design, but it may not always be the case for designs having more complex protocols or with much higher number of master/slaves. In such cases, we may not be able to see a PASS/FAIL status on these assertions and these may remain in the EXPLORED state.

IX. CONCLUSION AND FUTURE WORK

The proposed methodology is fully automated and can be reused for any ARM Cortex-M based device. It can be easily extended to other ARM processors which use more complex bus protocols like AXI. This approach can be a generic methodology that can be deployed across any design/protocol, given the availability of the corresponding ABVIPs. There is a known limitation of the data space not being exercised exhaustively and only control space being exhaustively verified. Sophisticated techniques like Formal scoreboarding can be used in conjunction with this approach to verify data integrity and is the next step in our plan. We also plan to replace each slave ABVIP with the corresponding RTL to gain further confidence on the whole system. Since this flow is based on Formal techniques, the inherent exhaustive nature of FV ensures higher confidence on the verification quality. This methodology successfully deploys formal verification to SoC level (other than just pin-multiplexing or connectivity checks) and can enable thoughts for deploying FV in multiple other SoC scenarios as well.

X. REFERENCES

1. Property Specification Language Reference Manual (http://www.eda.org/vfv/docs/PSL-v1.1.pdf)
2. Cadence Incisive Formal Verifier User Manual
3. Cadence Assertion Based Verification IP User Guide
4. AMBA™ 3 Specification Rev 1.0, http://www.arm.com