Experience of using Formal Verification for a Complex Memory Subsystem Design

Sujeet Kumar, Intel Technology India Pvt Ltd, Bangalore, (sujeet1.kumar@intel.com)
Vandana Goel, Intel Technology India Pvt Ltd, Bangalore, (vandana.goel@intel.com)
Hrushikesh Vaidya, Intel Technology India Pvt Ltd, Bangalore, (hrushikesh.vaidya@intel.com)
Ronak Sarikhada, Intel Technology India Pvt Ltd, Bangalore, (ronak.sarikhada@intel.com)

***Abstract*— Formal verification (FV) has been widely accepted as a verification approach for catching corner case design issues, it also speeds up the verification process of any subsystem [1]. Usage of formal verification for a complex memory subsystem design is not an easy task because of huge state space of the design, maintaining the different IP releases, and running developed test cases over time. In this paper, we discuss the best approaches followed for verifying a memory subsystem which includes a split based approach for unsolved properties, optimizing the engine parameters, connectivity check automation, testbench maintenance automation and cronjobs automation. Using these approaches, we found 8 critical RTL/architecture issues within a short span with limited resources. It also helped us to identify the problem ahead of the traditional functional verification.**

Keywords—Formal Verification, Automation of Formal Jobs, Connectivity Automation, Formal Regression.

I. INTRODUCTION

Formal verification is well known for verifying corner scenarios. Since random simulation-based verification is a non-ending task for complex designs, formal verification is required to gain full confidence.

With the latest advancement of different engines from different vendors, the solving capacity has been increasing. But there is still a limitation on verifying subsystem with multiple IPs.

With the current advancement in the formal domain, many formal apps are available like connectivity app, property check, sequence equivalence check (SEC) check, but they have the usage limitations. They do not support Intel specific file format, automation for running crons jobs and automated ways of creating CSV (comma-separated values) files from the designs.

In this paper, our sincere effort has been made to address these problems. The first problem is of invalid address spaces in the memory subsystem. We resolved this problem using split based approach, by constraining our design to only valid memory address space. Automation of the CSV generation (connectivity specification) of RTL design helped us to speed up the connectivity check which ideally requires lot of manual effort. In the register verification, tool vendors do not support intel specific register description language (RDL) files. We developed the script which addresses this issue by converting a register spec into a property-based check. For the issues like RTL changes, change in IP release paths on daily basis, we developed GUI based script which eased our process to maintain the file list and daily regression runs.

The formal test plan and execution of the memory subsystem is described in detail in Section II. The results of this exercise are discussed in Section III. The demonstrated results warranted the continuation of this methodology on future Intel Memory Subsystem. Authors believe that the proposed methodology can be incorporated in any subsystem verification flow to ensure effective usage of FV, and thus better verification confidence.

II. MEMORY SUBSYSTEM FORMAL EXECUTION

The test plan is essential to ensure all aspects of the product are covered and tested. As subsystem contains many IPs, the test plan helps brainstorm what verification technique will be used for individual IP and subsystem verification. Figure 1 represents the Memory Subsystem. Figure 2 provides a short summary of how FV is applied.

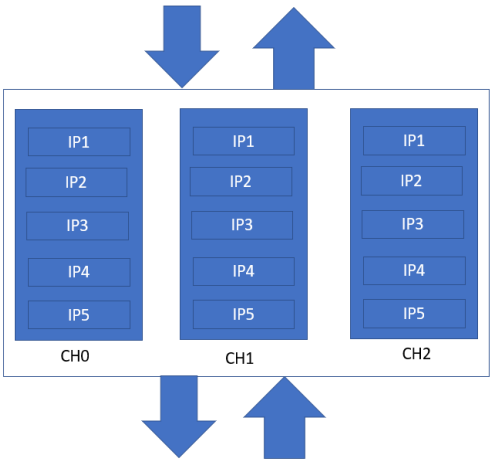

Figure 1: Memory MSS

Memory subsystem consists of five IPs, the role of these IPs is to ensure the routing of data from the processor to PHY.

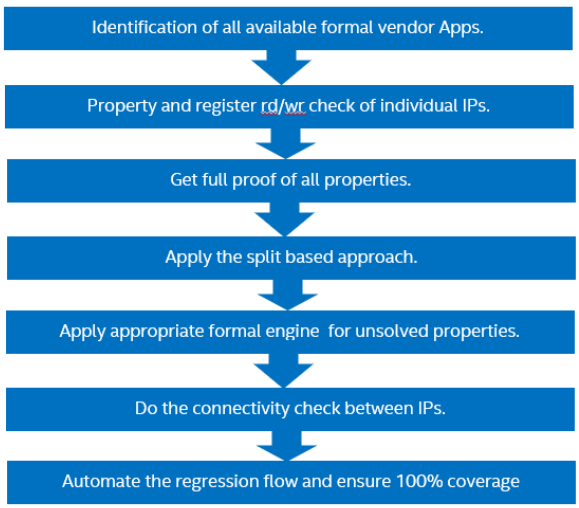

Figure 2: Formal Verification flow

2.1 SPLIT BASED APPROACH

FV task can be subdivided for complexity through case splitting. The problem can be defined as

$$f(x, y, z \ldots) = a_x + b_y + c_{z\ldots.}$$

Function is seen as dependent on its variables. Splitting is done in such a way that the range of x is kept random when y and z is constant and vice versa. By doing this, the problem complexity is minimized.

2.2 FV FLOW DETAILS

We followed the following steps to verify memory subsystem.

2.2.1. Identify all the available vendor apps:

We identified connectivity, register verification, property (SVA), sequence equivalence check (SEC) apps for this project.

1. *Connectivity App:* This app performs exhaustive verification of the static/structural, temporal and conditional connectivity of intellectual property (IP) blocks inside a system on chip *(SOC).*

2. *Register Verification App:* This app generates register checks from the definition table and performs exhaustive verification of the register read/write design of the IP as per the definition table.
3. *Property (SVA) App:* This app performs exhaustive property check.
4. *Sequence Equivalence Checking (SEC) App:* Sequential Equivalence Checking App provides a technique to verify the equivalence check between the designs.

2.2.2. Apps Usage:

1. **Connectivity App:** Connectivity specification is captured in the CSV format. Connectivity App automatically generates structural, behavioral and temporal connectivity checks and verifies the connectivity as specified in the CSV, as shown in Figure 3. IPs are black boxed to keep the verification analysis focused on only connectivity.

Syntax of writing connectivity properties as follows
CSV format: CONNECTION, <NAME OF CONNECTION>, <SOURCE SIGNAL>, <DESTINATION SIGNAL>
For example: CONNECTION, mss_ddrphy32_dfi0_ctrlreq, mss_lpddr0, dfi_ctrlreq, mss_ddrphy32_top, dfi0_ctrlreq

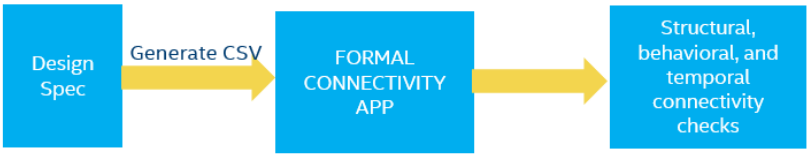

Figure 3: Usage of Connectivity App

2. **Register Verification App**: Control and status register (CSR) configuration and behavioral descriptions can be recorded in different formats: Excel/OpenOffice spreadsheets, comma-separated values (CSV) files or IP-XACT xml file. We used IP- XACT format for third party APB/AXI based IPs register verification. In the testbench setup, DUT is binded with ABVIP. ABVIP is assertion-based verification IP for APB/AXI AMBA bus protocol. ABVIP generates the traffic to the DUT. It also has embedded checks for the APB/AXI bus protocol. The register verification app automatically generates properties and captures register interaction, latency, and read/write semantics based on IP-XACT file. The setup of CSR app as shown in Figure 4

Figure 4: CSR App

3. **Property Based Verification**: In this approach, user defined properties are verified. There are two set of properties, ABVIP based properties which are generated by the tool and manual properties that captures IP specifications. For the manual property verification, we developed the assertions. The setup is shown in Figure 5.

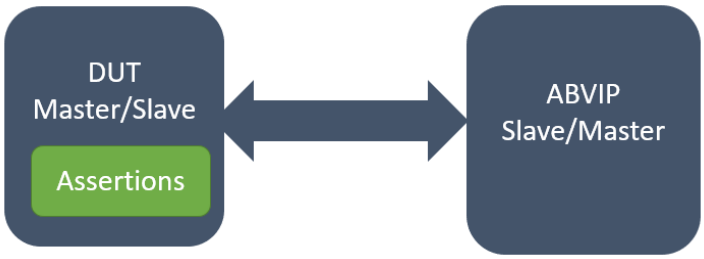

Figure 5: Property Based Verification

4. **Sequence Equivalence Check App:** In this design CH0, CH1, CH2 instances are identical with different coding styles. Using SEC app CH1/CH2 instance is verified against reference CH0. This saved us the time/resources to verify the CH1 and CH2. SEC app setup is shown in Figure 6.

Figure 6: SEC App

2.2.3 Split based Approach for Register Property Verification of Individual IPs:

For some inhouse IPs, register description format was not supported. Therefore we could not perform register verification of these IPs using CSR Apps. For such IPs, we developed a script to convert the register specifications to SVA based property file.

During the run, we did not get full proof for 2341 properties. It was found that these properties were being verified for invalid address spaces. To overcome this, we identified invalid address spaces for these properties and we constrained these properties only for valid address spaces. We call this approach as **Split Based Approach**. Split based approach is applied in floating point verification[2], we did not find any previous work on split based approach in the memory subsystem verification. Using split based approach, 1293 (55.23%) properties out of 2341 were solved. For the rest of the unsolved properties,1048 (44.76%), we applied different engines sequence with optimized engine parameters and got the required coverage.

IP name	Total number of properties	Number of undetermined properties						Undetermined solved using split
		RO		WO		RW		
		Before split	After split	Before split	After split	Before split	After split	
IP1	778	33	0	1	0	3	3	91.89%
IP2	652	90	72	18	0	171	63	51.61%
IP3	847	351	27	18	0	54	54	80.85%
IP4	2827	528	356	0	0	340	182	37.67%
IP5	2818	351	183	0	0	383	105	60.76%

Table 1. Split Based Approach Results

2.2.4 Connectivity Verification:

In integeration verification, connectivity of one IP to another IP is very important. For connectivity verification, connectivity CSV is required. Since generating CSV is a manual process, we automated the flow to read the connectivity spec and generate the CSV file. Automation helped a lot to reduce manual errors as well as time for developing the test cases by 5x.

2.2.5 Regressions:

Regression is very important to ensure design changes do not break the functionality, we automated the regression flow, and now tests are run on a regular basis.

2.2.6 Coverage App:

Coverage app is used to attained to the 100 % code coevarge. This gaves us the confidence on verification closure.

III. RESULTS

Determining the success of the deployment depends on number of issues found and time take to resolve. We got significant results using formal verification.

Category of bugs found

3.1 Performance Related: We found a performance bug in the design, which we would not have found through functional integration verification.

3.2. Connectivity Related: Four issues were found through connectivity verification early in the project.

3.3. IP/Reg Verif Related: Two crucial issues were found in the IP implementation. Same were later found in functional verification when scenarios were developed to test the features.

Table 2 compares the formal verification results with respect to functional verification results. Parameters used are listed in Table 3.

Parameter	**Formal**	**Functional**
Bugs found	$NB_{\_FV}$	$NB_{\_DV} = 3.5* NB_{\_FV}$
Resources	N_{R_FV}	$N_{R\_DV} = 3.5* N_{R\_FV}$
Test Bench Development Time	T_{D_FV}	$T_{D\_DV} = 3*T_{D\_FV}$

Table 2. Formal Vs Functional Comparisons

Parameter	**Depiction**
T_{D_FV}	Test development time in Formal Verification
T_{D_DV}	Test development time in Functional Verification
N_{R_FV}	Number of resources used in Formal Verification
N_{R_DV}	Number of resources used in Functional Verification
$NB_{\_FV}$	Number of bugs found in Formal Verification
$NB_{\_DV}$	Number of bugs found in Functional Verification

Table 3. Parameter Depiction

***** Formal verification started very late in the project.

Although Formal is started very late in the project, we got the significant result by this approach, and we expect, it can be deployed in any subsystem verification.

REFERENCES

[1] Erik Seligman, M Achutha KiranKumar, and Tom Schubert, “Formal Verification- An Essential Tool kit for the modern VLSI Design,” Elsevier Publications.

[2] C. Jacobi ; K. Weber ; V. Paruthi ; J. Baumgartner ,“Automatic formal verification of fused-multiply-add FPUs”,IEEE