2022 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE
MUNICH, GERMANY
DECEMBER 6 - 7, 2022

Automated Configuration of System Level C-Based CPU Testbench in Modern SoCs : A Novel Framework

Ruchi Misra, Chetan Kulkarni, Alok Kumar, Garima Srivastava,
Samsung Semiconductors India R&D, Bangalore,
ruchi.misra@samsung.com, chetan.k@samsung.com, kumar.alok@samsung.com, s.garima@samsung.com

Youngsik Kim, Seonil Brian Choi,
Samsung Electronics, Korea,

ys31.kim@samsung.com, seonilb.choi@samsung.com

***Abstract*—As the design size and complexity continues to increase, the task of verifying an SoC in 6–12 months becomes a challenge. Today, simulation time has dramatically increased from minutes and hours to days and weeks. This puts a lot of pressure on the methodology which we use to bring-up our SoC environment to enable the functional verification. One such issue comes when the testing environment has real CPU RTL and the configuration sequences have to be coded in C or Assembly level language. In order to bring up this C based environment for every design drop, there is a significant manual effort involved which is highly under-rated. In this paper we have illustrated two approaches to automate the process of coding or conversion of SV based sequences/specifications to C-based ones which enabled us to speed-up the functional verification closure in complex SoCs.**

Keywords— SoC, SV, TB, RTL, CPU, Simulation, Automation, Framework

I. INTRODUCTION

System-on-Chip design-verification is highly expensive and time-consuming process. The major challenges faced today in this field are reducing the area of chip, increasing the quality of chip, quality of test-bench, run-time reduction etc. Lot of efforts are being put to reduce time and resources by automating different processes involved in SOC cycle. The testbench files provide stimulus vectors to the design under test (DUT). The combination of DUT and testbench has to undergo sequential steps usually executed manually. The time taken by this verification step increases with the complexity of the design as well as the testbench. Apart from this, manual intervention introduces the possibility of human errors which might add more time to development cycle. One such process is the bring-up of C based testbench environment which includes manual coding or conversion of some of the design initialization sequences to C. The area of concern for the current work is manual coding of the C based design sequences which introduces the possibility of errors and is also very time consuming. We saw an opportunity to speed-up the C-based testbench environment bring-up and to eliminate the manual conversion of C files and hence went ahead with scripting the entire process which ended up giving good returns and proved beneficial during execution of the project cycle.

When it comes to verification of the SoC along with real RTL such as CPU, the input initialization sequences like clock/reset/boot sequence must be in assembly language or C-programming language. The design specifications or the register programming's are usually first coded in SV or UVM based testbench to start the stubbed CPU environment testing. Once we start incorporating the real CPU RTL Design, there arises the need for either the coding or conversion of SV based sequences to C.

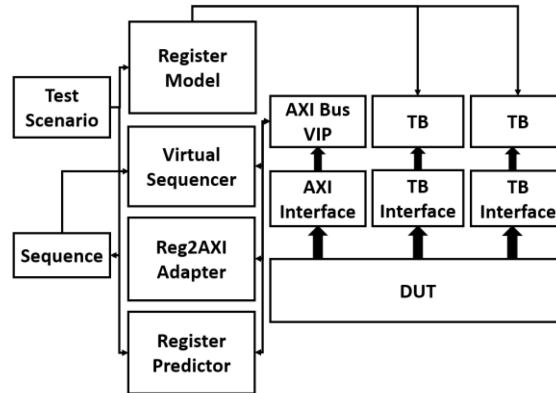

Figure 1: Typical Test-Bench Architecture

The SoC consists of various blocks such as CPU, clock generator and controller, power controller, memory unit, USB etc. All blocks are connected using AMBA bus protocols such as AXI, CHI, APB. The communication between all the blocks takes place through these buses. A typical testbench architecture is shown in Figure 1 above. The AXI Bus VIP connects to the DUT through AXI interface which is driven by UVM sequencers. Verification engineers use Register models to ease the stimulus generation and functional checking. A register model consists of a set of register definitions, register instances clustered under a register block and their address mappings into Design Under Test's (DUT) address space. The register model source files can be implemented manually by verification engineer or, the preferred way, generated from a textual register description such as an XML file, IP-XACT or a spreadsheet.

The Register abstraction layer that allows modelling and verification of hardware registers and memory blocks, provides read() or write() tasks that can be called from the sequence using either a memory or a register path and an abstraction layer maps these into bus level transactions with the correct address, some of the register library features are summarized as follows:

- Read and write for the registers and the memories as part of normal operation.
- Modeling registers, memory blocks and analyze the register activity.
- Checking the DUT registers and memories against a shadow device (with shadow register) at the scoreboard.
- Collect coverage on the registers that are created at the RAL model.
- Initialize the DUT registers and memories.
- Randomize the contents of DUT registers and memories.

II. RELATED WORK

There has been some work in the industry in past few years related to the automated framework which takes inputs from specifications given in IPXACT and generates C tests directly. However, the solution presented in this paper is aimed to provide a complete solution in terms of SoC verification based on SV and C based testbenches. The proposed framework dumps out both Spec based as well as SV based incremental code so that for derivative projects and minor changes, only the additional changes are converted to C based testbench.

III. METHODOLOGY

UVM offers power of randomization and constraints naturally, but verification teams have needs to create or reuse C programs also. These C programs may generate stimulus, check golden results and collect statistical data.

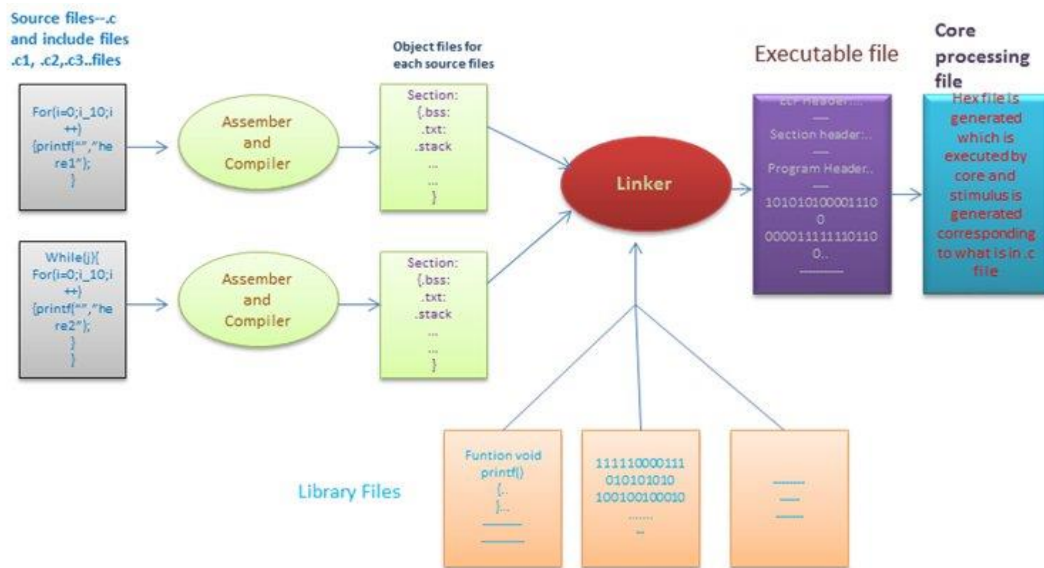

Figure 2: Files used in C based SoC TB

As shown in Fig 2, C source files after going through assembler and compiler, dump object files. Then linker creates executable files which sometimes takes library files also and finally generates hex files that is processed by cores.

Using SystemVerilog DPI-C is one way to connect these two SV/UVM and C worlds together. But the problem is that using DPI-C can sometimes be hard, slow and the DPI code has a close connection to a “scope” which can be a module instance, an interface instance, or the global root scope. Hence parallel coding of SV and C based sequences is done progressively for every RTL release manually during the verification cycle.

SoC verification engineers recognize the limitations of constrained-random testbenches, driving them to handwrite C test sequences to run on the processors for both simulation and hardware emulation, even though they are limited in fully exercising the SoC design. The performance of these verification platforms is not good enough to run a full operating system (OS), so these tests execute “bare-metal,” which adds a significant overhead to composition effort. Even though there is a lot of thought going in the industry in the direction of automating handwritten tests, it is a reality that still verification engineers are relying on legacy C code for many projects. The reason for this is that the C based testbenches need only one-time set-up effort, are more reliable and provide better coverage. Given the scope of derivative projects in current VLSI industry, it is certain that the C based testbenches and directed or manually created C based sequences are not going away anytime soon. Now, while the C based testbench needs manual intervention every design label to convert the corresponding changes in the Spec or SV

sequence, here comes the need of automation in reducing a big load off of verification engineers in terms of generation of testbench sequences and repetitive codes.

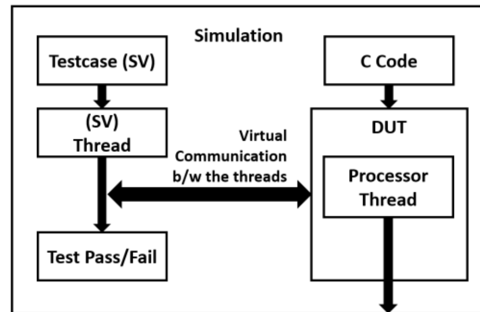

Figure 3: Parallel SV and C flow in simulation

The parallel existence of SV and C based testbench is explained in Fig 3 above. The SV testcase gives the control to C code and receives back the control before declaring test pass or fail. The stimulus is provided to the DUT through the C test which gets compiled and converted to assembly level. The parallel coding of SV and C based sequences is done progressively for every RTL release manually during the verification cycle. This paper illustrates two different approaches of automating the code or creation of these C files using the proposed Perl based framework.

A. Initial Conversion (Spec Based)

Once the design specification is available, the verification engineer with good C coding skills is supposed to create the same onto the C based testbench. The complete bringup of the C based infrastructure is needed when the first version of design is released. In this approach, we convert the complete boot/clock/reset sequence as per the specification into C-language. we have to convert all the tasks, function calls, class methods, conditions and all the methods of RAL model used such as register write, read, set-get methods etc and replicate the same code behavior in C-language.

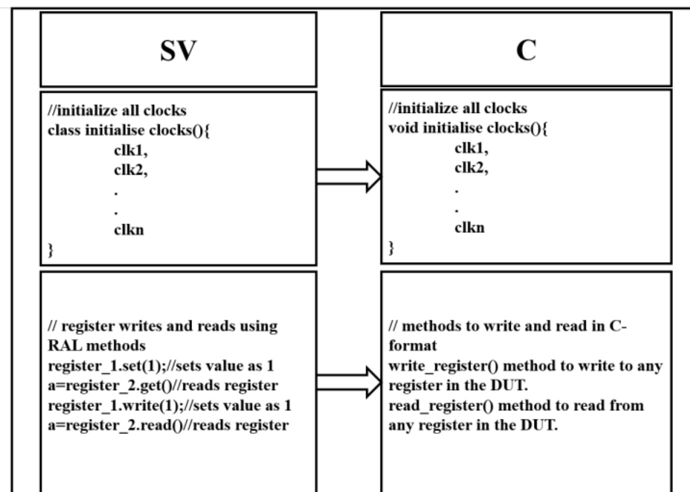

Figure 4: Code comparison between SV and C

B. Incemental Conversion (SV sequence based)

As SOC verification cycle progresses, there will be several software releases along with test bench and test sequence updates. There is a need to update the already converted C-file as and when there is update in corresponding sequence in SV file. In this approach, the conversion is only performed when there is an update in design specification or the corresponding SV-file. This update is converted in C-language syntax and added in proper position in C-file. The framework is able to handle such incremental updates which happen every design release. The script is smart enough to find the difference, add proper comments and even insert defines wherever needed. This incremental conversion usually happens during the entire SoC cycle till the design matures. An

example pseudo-code snippet is shown in Fig 4 which demonstrates how similar functions in SV and C differ in structure and syntax. The figure illustrates the conversion process in general by taking an example of clock initialization function and shows how register reads and writes differ.

IV. EXAMPLE APPLICATION : AUTOMATED BOOT SEQUENCE

While UVM tests run in simulation, C tests and sequences are more flexible. They can run in simulation on a CPU model, RTL or behavioral, along with the RTL design. They can also run as "bare metal" tests in an emulator, an FPGA prototype, and even the final chip. Thus, the C tests range from RTL simulation to hardware-software co-simulation to full system validation.

The flow diagram (Figure 5) shows the generic boot flow for an SoC. When the external reset of the chip is released, all the clocks and resets and memory are initialized. CPU reset is released, and CPU starts fetching instruction from first location of the memory. Resets of other blocks is released and powered up based on the sequence provided to CPU. We have to convert all these sequences included in this process such as clock initialization, reset initialization etc into C-language. This is where the automation has proven beneficial. The Perl based tool is capable of taking user inputs whether incremental or initial changes are needed and dumps out C code which can then be directly plugged into the testbench.

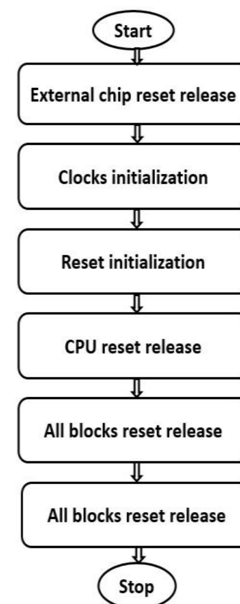

Figure 5: SoC boot flow

V. EXAMPLE APPLICATION: REGISTER UPDATES

Register definition generally starts with an architect scoping out a specification. Once the specification is completed the hardware engineer, software engineer, and verification engineers can begin coding different views of the registers described in the functional specification. Once we have a design, verification and software engineers can start running tests. Anytime a bug is discovered the specification must be changed and all the subsequent outputs must be changed accordingly. But due to other priorities the designer would have changed the code but not the document or vice versa. This process repeats itself many times over the course of the project. Bugs are only one source of change; marketing requests may also come in at any stage of the design cycle requiring the specification to change and all downstream code to be modified.

Hence one of the typical use-cases of this tool script is to convert the register access codes of SV to C. There are two ways of accessing design registers in a verification environment: front door and back door. Front door is by using the design register bus. This consumes cycles and follows the register bus protocol. Backdoor is a zero-simulation time access by mapping to the design register directly using the HDL path and allows quick configuration of registers. Thus, configuring by backdoor saves simulation time, especially useful for full-chip and sub-chip simulations where several registers need to be setup. Backdoor access also helps in uncovering address decode design bugs. Since the mirror register gets updated while doing front-door or back-door writes, writing a register in front door and reading it from the backdoor flags any mismatch between design and the mirror registers. Another important use of backdoor access is to know when a register field (RO bit) is updated by design and use that information without polling for this (RO) bit using the register bus. The example is an Interrupt status register bit (RO), Using the back door, we get to know that Interrupt has set and ISR procedure is executed as in the system. Now, these front door and backdoor accesses are very common in C based testbench and are often needed to be manually added by user every design label which can be easily done by the proposed tool.

Figure 6 below shows SV and C based Data types and most commonly used SV types which are directly compatible with the C types are shown here.

SYSTEMVERILOG TYPE	C Type
byte	char
int	int
longint	long long
shortint	short int
real	double
shortreal	float
chandle	void*
string	char*

Figure 6 : Comparison between SV and C Data Types

There are System Verilog-specific types, including packed types (arrays, structures, unions), 2-state or 4-state, which have no natural correspondence in C. For these the designers can choose the layout and representation that best suits their simulation performance. The representation of data types such as packed bit and logic arrays are implementation-dependent, therefore applications using them are not binary-compatible (i.e., an application compiled for a given platform will not work with every System Verilog simulator on that platform). Packed arrays are treated as one-dimensional, while the unpacked part of an array can have an arbitrary number of dimensions.

The framework should be able to handle such tricky issues like defines or includes in SV and C and different data-types in SV or C for example. There are multiple challenges while creating a C based code through the framework like synchronization of events, print statements etc. However, through multiple iterations and trials on different versions of design, the script of framework can be matured to provide an error-free C code.

VI. RESULTS

A. *Advantages of using the framework:*

- Removes the need to perform manual conversion or coding of C sequences of CPU system level testbench which is prone to human error as well as time consuming.
- Reduces the manual effort of C code change based on design changes, by approximately 1 day per week, in turn saving 52 days in a year.
- Eliminates the possibility of unnecessary debugs resulting due to error-prone coding or conversion of C code arising due to spec change.

B. *Engineering Man-hours saved*

Below bar graph compares the number of days taken to do the C coding or conversion with and without using the proposed framework and highlights the efficiency of the tool. We adapted the tool for a live project during the course of 8-10 months and observed that the bring-up efforts of the C-based environment was reduced significantly and it helped engineer focus on functional verification instead. The proposed tool is also recommended for future generation of projects with minor enhancements as needed.

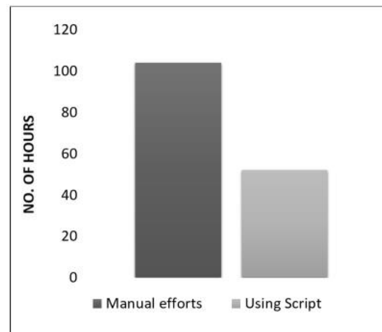

Figure 7: Comparison of efforts with and without the Automation

VII. CONCLUSION

Simulation of the RTL model is one of the first and mandatory steps of the design verification flow. Such a simulation needs to be repeated often due to the changing nature of the design in its early development stages and after consecutive bug fixing. Despite its relatively high level of abstraction, RTL simulation is a very time-consuming process, often requiring nightly or week-long regression runs. The paper describes the basic idea of the framework through a pilot done on the initialization sequences used in boot flow and provides some initial experimental results showing its effectiveness in improving RTL simulation performance in an automated way.

VIII. FUTURE WORK

Automation of the initialization sequences used in boot flow gave us confidence in the possibility of faster bring-up of the C based test environment thereby saving manual effort and considerable time. Taking this one step further, the same can be achieved for other flavors of the verification environment such as Power Aware simulations, UPF (Unified Power Format) based enablement, DFD (Design for Debug) and Emulation. The framework has the potential to scale-up to become a completely design spec-based sequence generator which would help in multiple dimensions of the SoC testing.

IX. REFERENCES

[1] C. Mapara and P. Nagarajan, ”Transaction Based Speedup for Simulation Replay,” 2018 19th International Workshop on Microprocessor and SOC Test and Verification (MTV), 2018, pp. 73-75, doi: 10.1109/MTV.2018.00024.

[2] T. B. Ahmad and M. Ciesielski, ”Fast time-parallel C-based eventdriven RTL simulation,” 17th International Symposium on Design and Diagnostics of Electronic Circuits Systems, 2014, pp. 71-76, doi: 10.1109/DDECS.2014.6868766