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,

YoungSik Kim, Seonil Brian Choi

SAMSUNG

accellera SYSTEMS INITIATIVE

Agenda

- Introduction
- Motivation
- A Typical Test-Bench Architecture
- Methodology
- Example Applications
- Results
- Conclusion
- Future Work
- Acknowledgement

Introduction

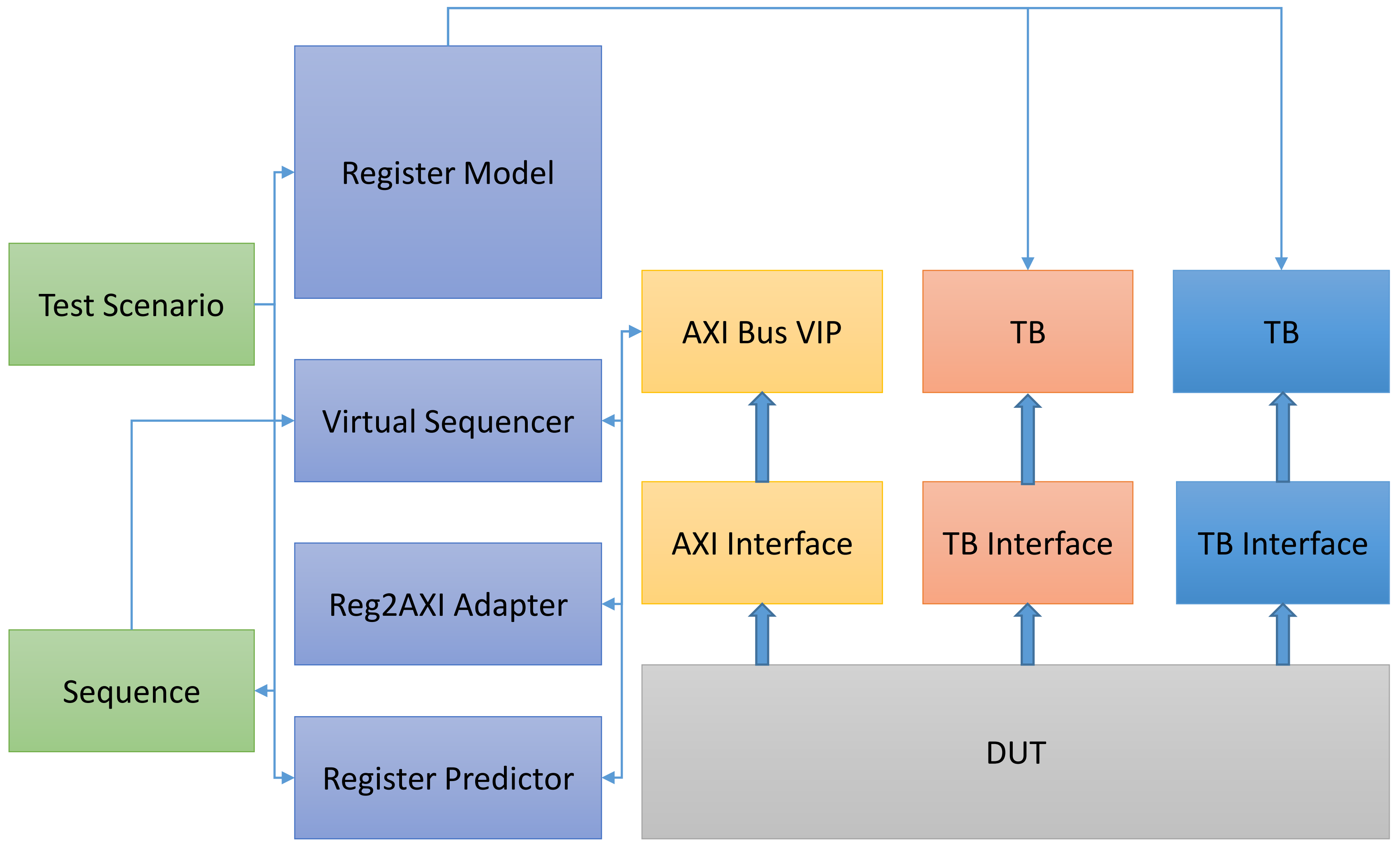
- Design and Complexity continues to increase for chipsets.
- Simulation Time has increased dramatically at SoC level from minutes and hours to days and weeks.
- This puts a lot of pressure on the methodology which we use to bring up our SoC environment.
- When the testing environment has real CPU RTL, the configuration sequences have to be coded in C.
- In order to bring up this C based environment for every design drop, there is a significant manual effort involved.
- Lets see two approaches to automate this 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.

Motivation

- 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.
- Bring-up of C based testbench environment which includes manual coding or conversion of some of the design initialization sequences to C.
- Manual coding of these C based design sequences introduces the possibility of errors and is also very time consuming.
- Opportunity to speed-up the C-based testbench environment bring-up and to eliminate the manual conversion of C files.

A 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 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.



SV and C Based Testbenches

- UVM offers power of randomization and constraints naturally, but verification teams have needs to create or reuse C programs also.
- C tests range from RTL simulation to hardware-software co-simulation to full system validation.
- These C programs may generate stimulus, check golden results and collect statistical data.
- C source files after going through assembler and compiler, dump object files.
- Linker creates executable files which sometimes takes library files also and finally generates hex files that is processed by cores.

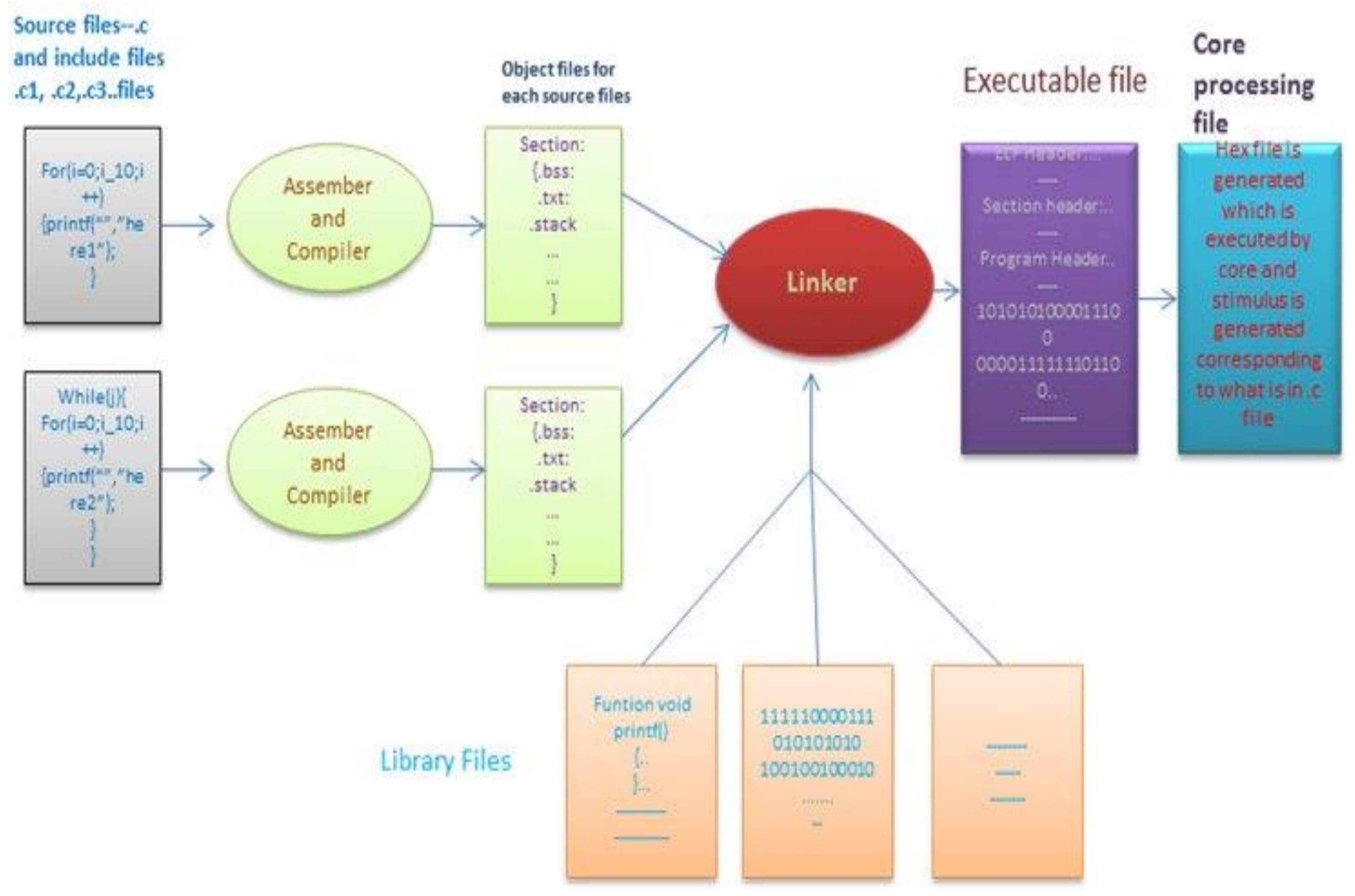
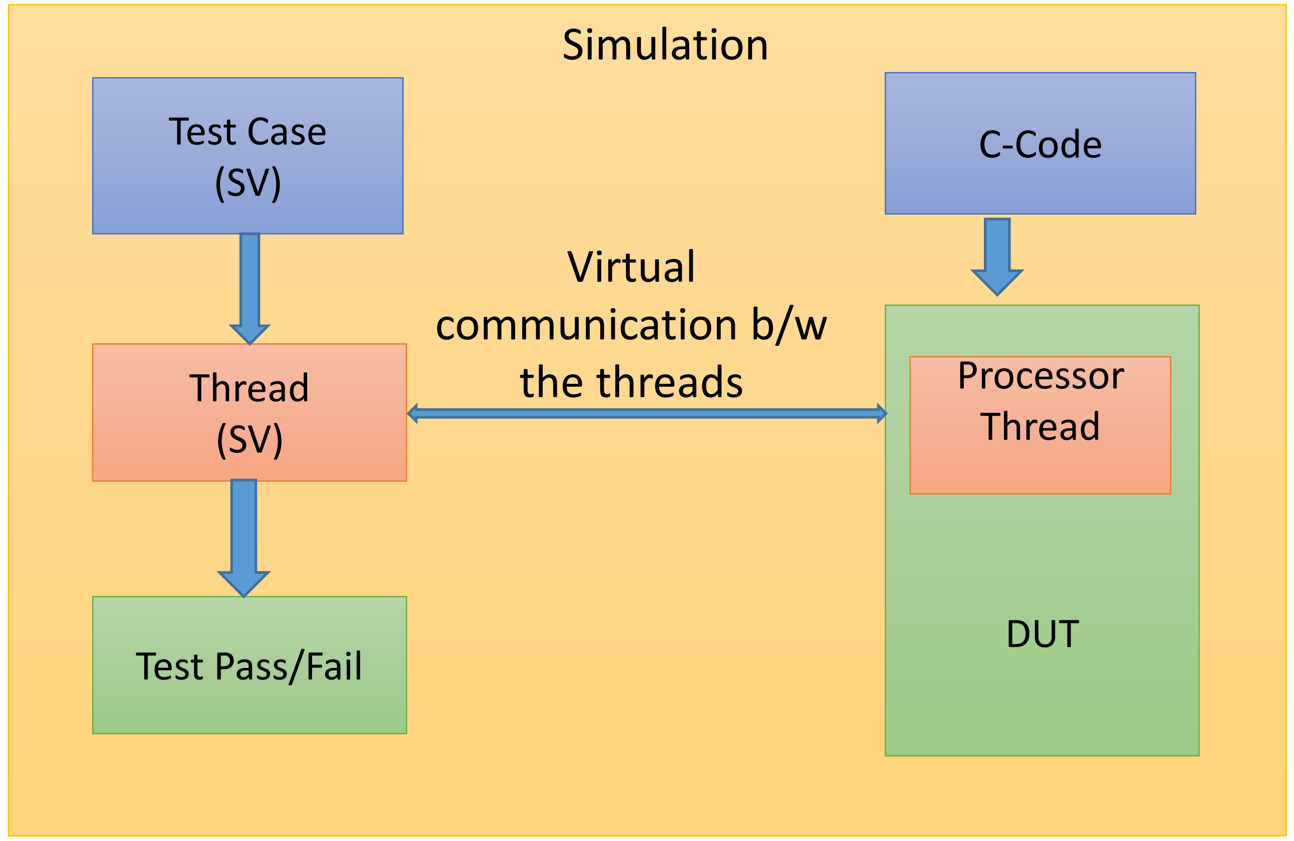

Source files--.c and include files .c1, .c2,.c3..files
For(i=0;i_10;i++) {printf("","here1"); }
While(j){ For(i=0;i_10;i++) {printf("","here2"); } }
Assember and Compiler
Assember and Compiler
Object files for each source files
Section: {.bss: .txt: .stack ... ... }
Section: {.bss: .txt: .stack ... ... }
Linker
Executable file
Section header..
Program Header..
101010100001110 0 000011111110110 0..
Core processing file
Hex file is generated which is executed by core and stimulus is generated corresponding to what is in .c file
Library Files
Funtion void printf() {.. }..
111110000111 010101010 100100100010

Figure 2: Files used in C based SoC TB

Parallel Existence of SV and C based Test-Benches

- 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.
- Lets see two different approaches of automating the code or creation of these C files using the proposed Perl based framework.



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

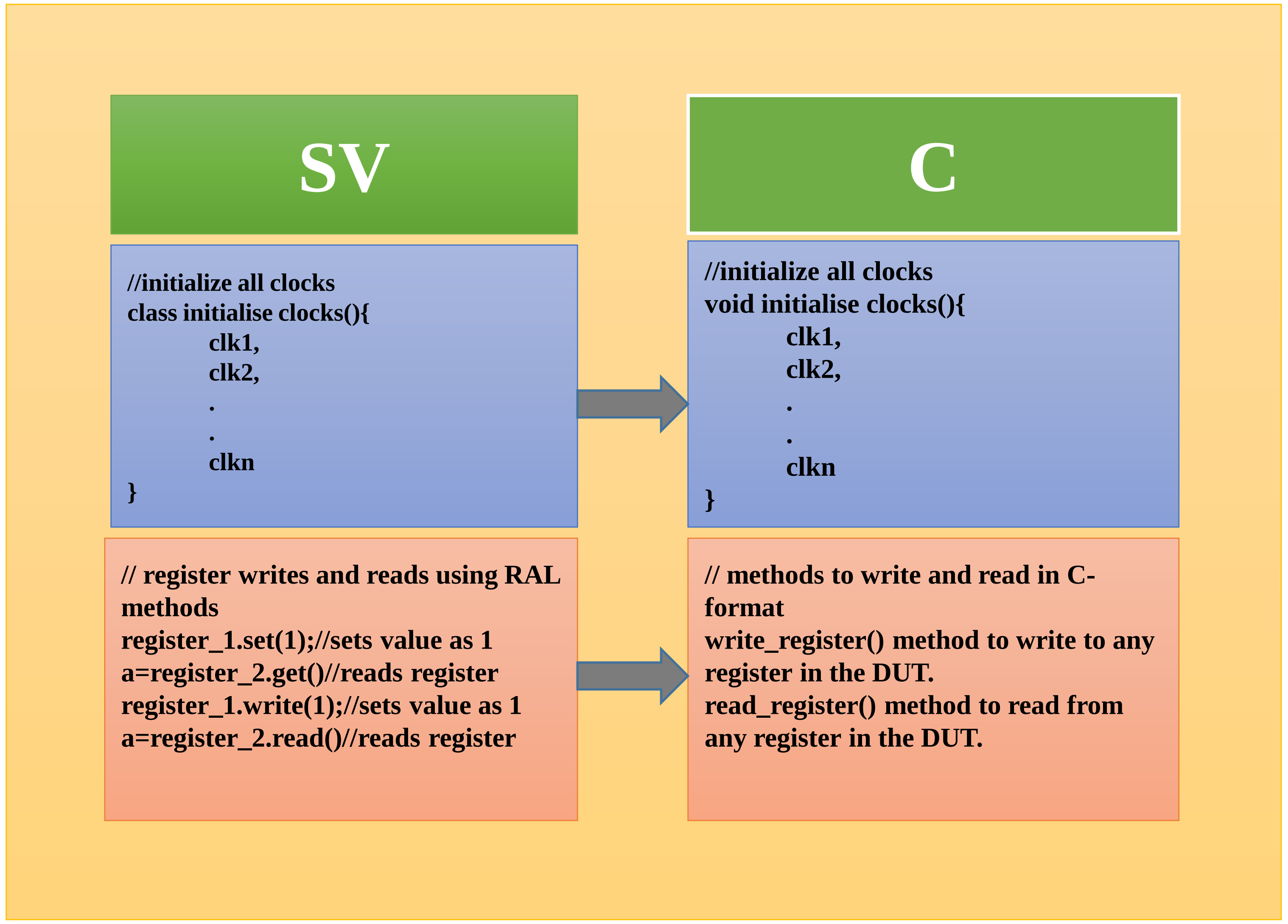
Connecting SV/UVM and C Worlds Together

- Using SystemVerilog DPI-C is one way
- But 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.
- 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.
- 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.

Methodology

- **Initial Conversion (Spec Based)**
  - ❖ when the first version of design is released.
  - ❖ we convert the complete boot/clock/reset sequence as per the specification into C-language.
  - ❖ 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.

- **Incremental Conversion (SV sequence based)**
  - ❖ 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.



```
//initialize all clocks
class initialise clocks(){
        clk1,
        clk2,
        .
        .
        clkn
}
```

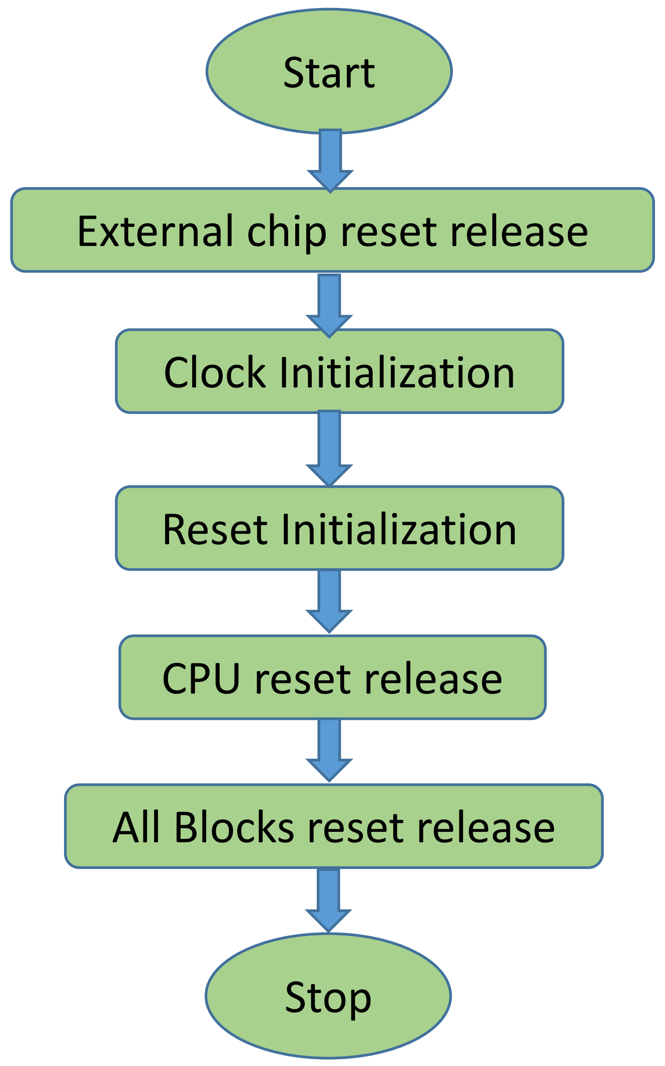
```
//initialize all clocks
void initialise clocks(){
        clk1,
        clk2,
        .
        .
        clkn
}
```

```
// register writes and reads using RAL
methods
register_1.set(1);//sets value as 1
a=register_2.get()//reads register
register_1.write(1);//sets value as 1
a=register_2.read()//reads register
```

```
// methods to write and read in C-
format
write_register() method to write to any
register in the DUT.
read_register() method to read from
any register in the DUT.
```

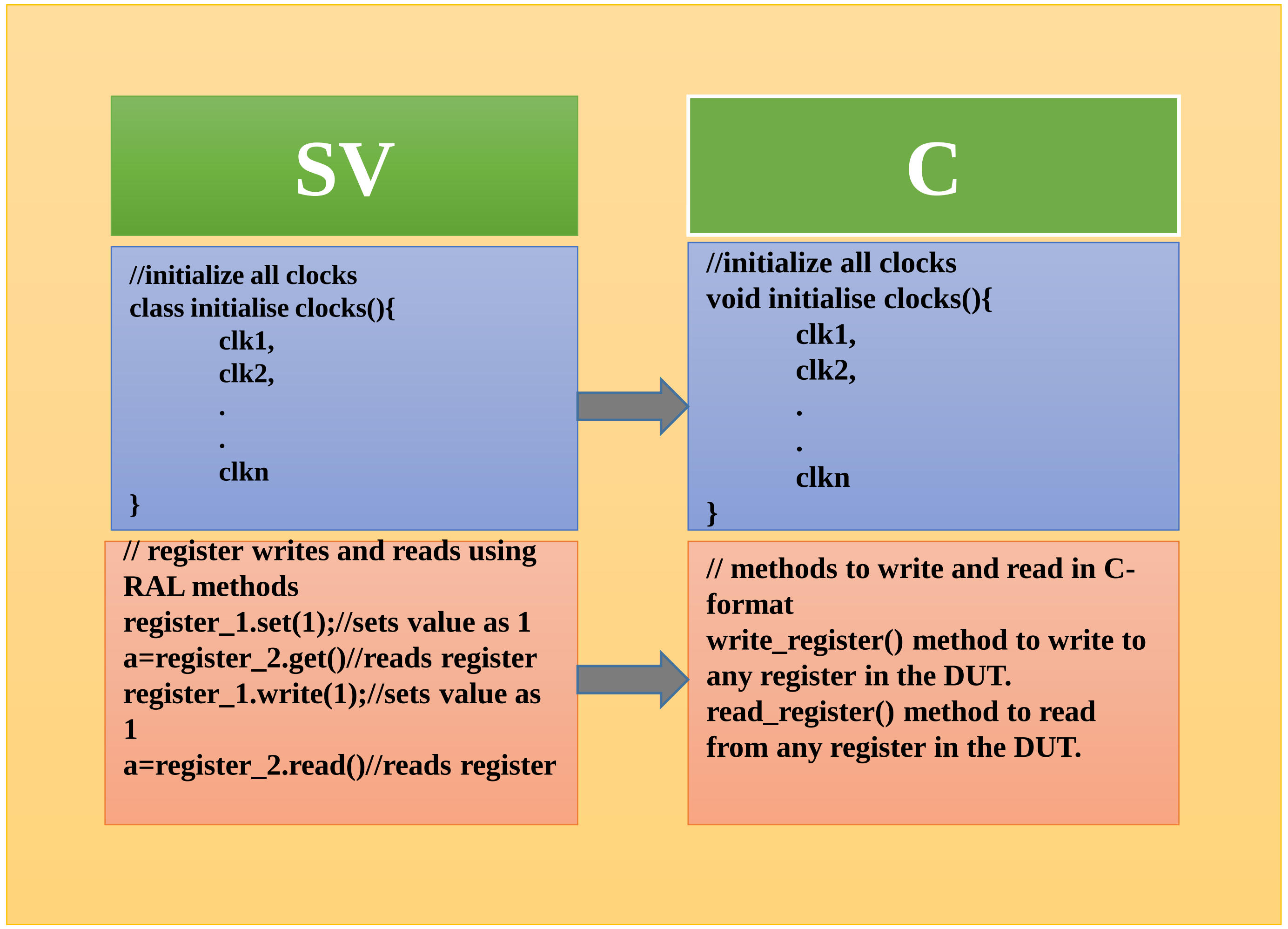
Example Application : Automated Boot Sequence

- When the external reset of the chip is released, all the clocks and resets and memory are initialized.
- CPU reset gets 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.
- The Proposed framework converts all these sequences such as clock initialization, reset initialization etc into C-language.



Example Application : Register Updates

- Another typical use-case of this tool script is to convert the register access code of SV to C.
- Two ways of accessing design registers in a verification environment: Frontdoor and Backdoor.
- 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.
- 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.



```
//initialize all clocks
class initialise clocks(){
        clk1,
        clk2,
        .
        .
        clkn
}
```

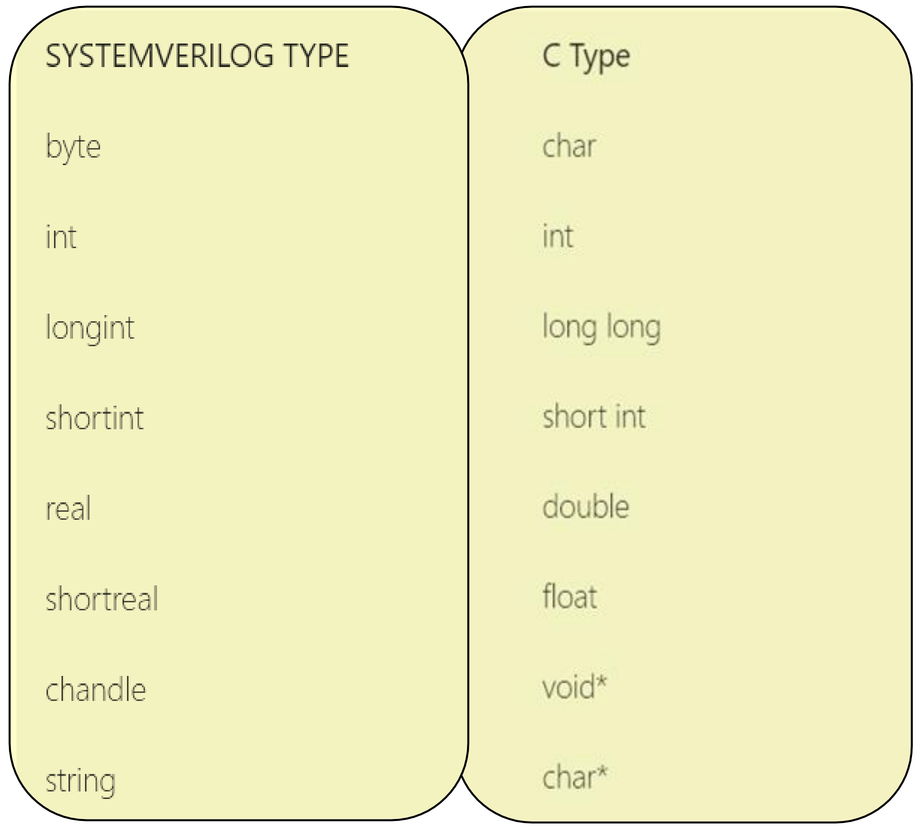
```
// register writes and reads using
RAL methods
register_1.set(1);//sets value as 1
a=register_2.get()//reads register
register_1.write(1);//sets value as
1
a=register_2.read()//reads register
```

```
//initialize all clocks
void initialise clocks(){
        clk1,
        clk2,
        .
        .
        clkn
}
```

```
// methods to write and read in C-
format
write_register() method to write to
any register in the DUT.
read_register() method to read
from any register in the DUT.
```

Common Challenges

- We can see here SV and C based Data types and most commonly used SV types which are directly compatible with the C 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 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.
- Synchronization of events in C is another challenge.
- Converting the print statements from SV to C should be handled.



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

Results

- The 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.
- 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.
- The tool was piloted for a live project during the course of 8-10 months.
- Eliminates the possibility of unnecessary debugs resulting due to error-prone coding or conversion of C code arising due to spec change.
- Recommended for future generation of derivative projects with incremental updates.

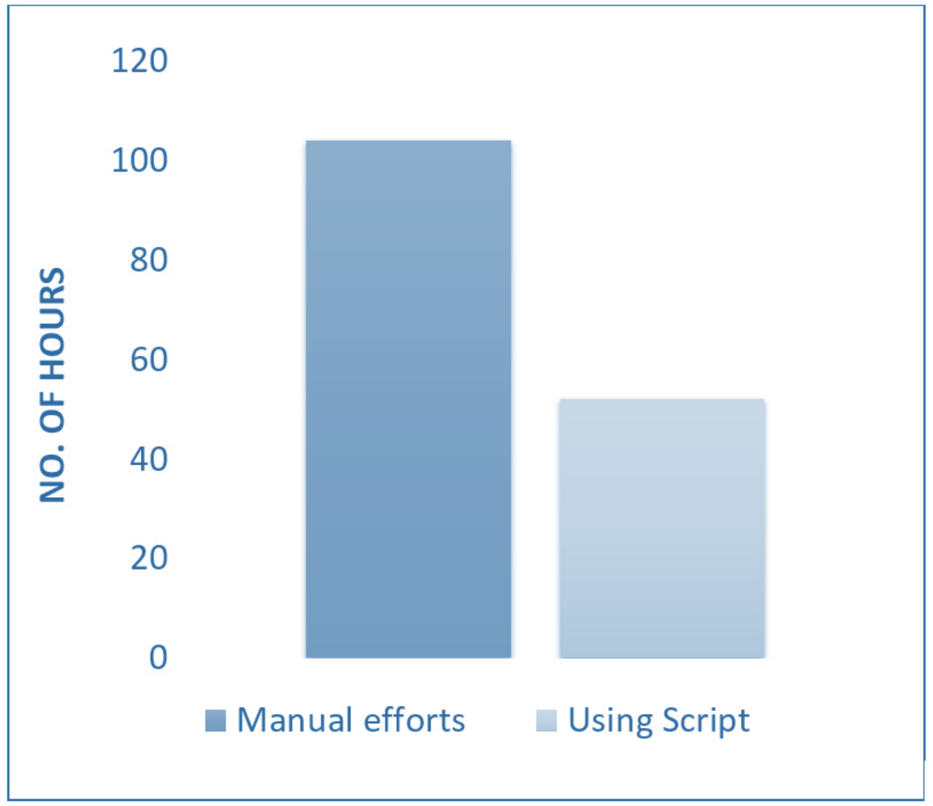


Figure 7: Comparison of efforts with and without the Automation

Conclusion

- 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.
- 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.

Future Potential

- Power Aware simulations
- UPF (Unified Power Format) based enablement
- DFD (Design for Debug) environment bring-up
- Emulation environment bring-up
- 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.

Acknowledgement

The authors would like to thank Samsung Semiconductors India Research for enabling the work mentioned in this paper. We would also like to thank DVCon Europe team for giving us the opportunity to participate in the conference and present our work.