SystemC Virtual Prototype: Ride the earliest train for Time-To-Market!

Shweta Saxena
Analog Devices Inc.
shweta.saxena@analog.com

Mahantesh Danagouda
Analog Devices Inc.
Mahantesh.danagouda@analog.com

***Abstract*- As the design complexities increase, it is evident that initial software development and validation must get the necessary head-start so that major software collateral can be developed much earlier in the IC design cycle to meet aggressive time-to-market windows. This paper focuses on how the time between design conception and actual development can be used for developing a Virtual Prototype of design to achieve dual goals of first silicon success and significant time-to-market reduction. It highlights how embedded software could run on a virtual model and be used for early driver development of multiple peripherals, software validation infrastructure and testing framework. Unlike conventional software development and validation methodologies like FPGA prototyping and emulation that are hardware dependent, a virtual prototype can be developed independent of hardware, thereby giving an opportunity to begin software development much before the hardware development has even begun.**

I. Introduction

The growing interest in multiprocessor system-on-chip (SOC) designs has introduced an equally complex software development and validation challenge. System design is increasingly being performed at higher levels of abstraction to deal with variety of issues. Early software development has now become a key system level design initiative, so our focus must shift from "Are we building the product, right?" to "Are we building the right product?". Virtual prototyping is a novel method to meet the expectation of first silicon success and tight schedules for aggressive time-to-market windows.

A Virtual Prototype or VP is a simulation model of a hardware device that mimics its functionality at an abstraction level, which allows running and interactively debugging the actual embedded software without modifications. Essentially it is a platform to deal with both hardware and software in a single model. It is coded in SystemC and data transfer is modeled using SystemC compatible TLM2.0.

Early software development channels as a part of hardware-software co-development thereby helping the software to develop much earlier, so it could be leveraged for hardware verification. Being independent of hardware, VP gives the flexibility of independent testing, allowing the software to be developed while the hardware is still under development. This way we can determine if we are building the right software early in the design cycle. This is a big advantage over traditional development, where the hardware design must be functionally healthy prior to the creation of a FPGA platform, which is then used for software development and validation. A VP maintains agility in the process of software development, helps gain acceleration in time to market and gives us the options to hit our aggressive time-to-market windows by utilizing the time between defining system design specification till the actual hardware is designed. It provides a medium to run an application much earlier than the actual silicon, but at speeds that are still comparable to real silicon which gives a considerable gain in time to make viable assumptions for system design and options to run applications based on actual use cases early in the IC design cycle.

VP also gives us an opportunity to do architectural exploration or inspect the what-if scenarios quite early. Architectural exploration deals with coming up with a suitable system architecture and distributing system tasks in the specification onto individual components and make viable assumptions to decide the hardware-software partitioning at the time of system definition.

II. Description

We developed a SystemC based VP of ADI's next generation audio codec and used it for early development and validation of the hardware-dependent software or HDS. HDS is the part of an operating system that varies across microprocessor boards and is comprised notably of device drivers and of boot code which performs hardware initialization.

A. Functional model development and Automated SystemC code generation

A typical SOC would host hundreds or thousands of registers and hand-coding them using SystemC takes lots of manual effort. Hence, we leveraged on automated SystemC code generation. The basic skeleton structure for the VP was generated using ADI's proprietary XML based internal tool that is used by the designers to define register maps, memories, all address spaces along with hierarchical block definitions and certain properties that are used to generate the RTL. The tool also supports a python-based plugin that parses register map definitions and memory address space to generate a matching hierarchical equivalent in SystemC. It also generates the entire register map package and stiches it appropriately. Functional SystemC models for individual blocks are then developed and instantiated within respective module nutshells in the generated code. The process involves creation of a xml-based file that defines different properties that aid in RTL generation. There are a set of properties that are used for the VP generation, but all these properties stem from a common design definition, hence ensuring that the register maps, hierarchical definitions, and memory address space is always same in RTL and VP. Every time the design is updated, automated processes run this tool to generate RTL and the VP as well, thereby making them hierarchically and structurally equivalent. The challenge to keep RTL and VP functionally equivalent was achieved by coordinating the functional changes done by the design team in periodic discussions and making same updates to the models. Some basic C test cases from the design verification environment or some common applications could be run on the VP and RTL both. Some of the applications that were developed to be run on the VP, were also used as tests for RTL validation. Figure1 shows the flow for the code automation.

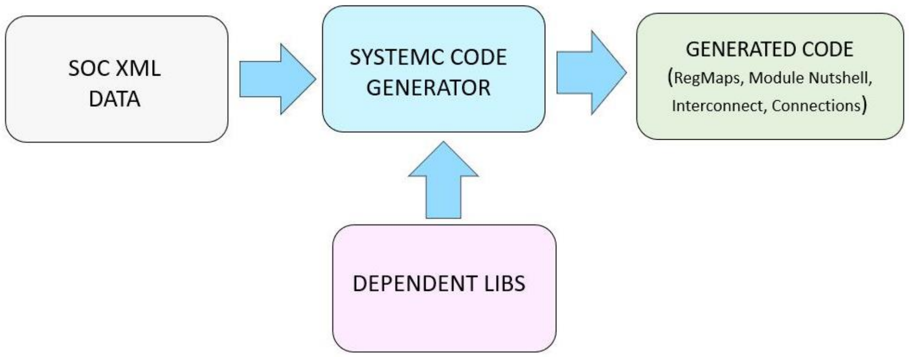

SOC XML DATA
SYSTEMC CODE GENERATOR
GENERATED CODE (RegMaps, Module Nutshell, Interconnect, Connections)
DEPENDENT LIBS

Figure 1: Automated SystemC code generation

Based on the requirement of software, this VP included a hybrid environment of loosely timed pure transactional models and near cycle accurate models to instill further confidence in the software validation. The design involved multiple cores(heterogenous), so there were different interrupt control mechanism and interface that comply with different processors and connect to their respective debuggers.

Design specific digital subsystem's individual blocks were modelled in-house. Because of the complexity involved in modelling the core's proprietary design, the core subsystem (core1 and core2) models were obtained from corresponding vendors. SystemC models for various design blocks in the digital subsystem were developed to begin software development, and in parallel, its validation. The data transfer uses TLM2.0 which gives up to 1000X speed gain in comparison to the normal RTL simulation.

The top-level wrapper (as shown in Figure 2) comprises of:

- Core subsystem with core1,
- Communication subsystem with models for i2c, i3c, spi, qspi, uart,
- Memory subsystem with models of memory DMA, SROM and SRAM
- Security subsystem with core2,
- System Interrupt Controller or SIC with core1
- Audio and control subsystems were not modelled.

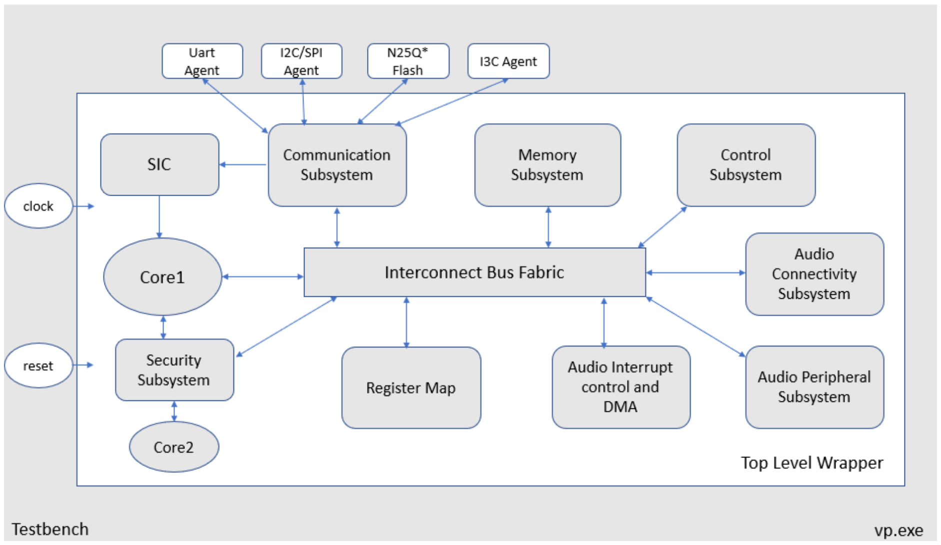


Figure 2: SystemC Virtual Prototype

B. Testbench, Application development and debugging

The VP was designed to be a ready to use executable for the software team. For this purpose, we included the top-level wrapper with generated SystemC code along with functional models and then instantiated it in a testbench which is also a SystemC module. It instantiates clock and reset generating modules that meet the design specifications. However, the clock used in the VP was pre-defined using the construct *sc_clock*() with a fixed frequency. The testbench is instantiated in the main function called the *sc_main*().

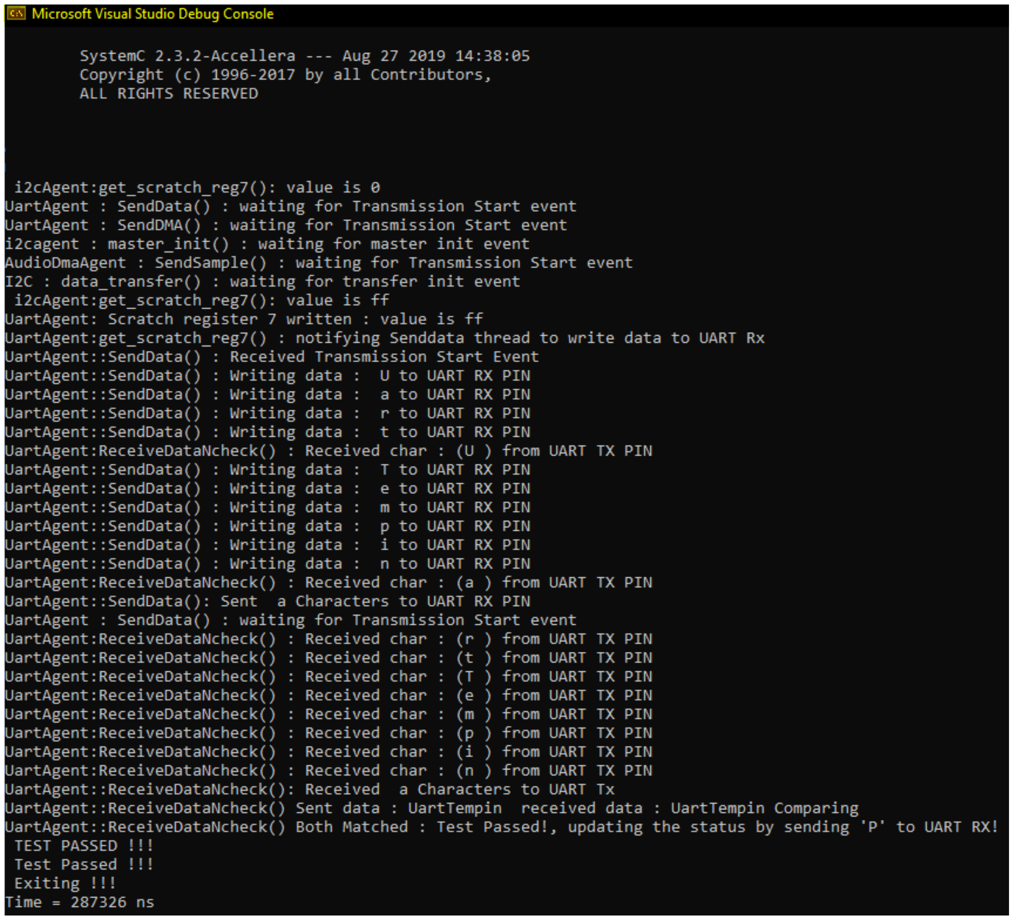
- *sc_main*() is a pre-defined function in SystemC analogous to the main() function in C/C++. The arguments required by the core1 and core2 configurations and other additional simulation related arguments can be provided as character arguments to this function. The path to the application image is also provided as an argument.
- *sc_start*() is a pre-defined function, which is called within sc_main() and is responsible for starting all the simulation threads as soon as it is called.

All this code is built together to generate a complete executable. Any application can then be built by individual core debuggers and run by loading it on core subsystem of the VP, by providing the binary file of that application as an argument to the VP executable. Command line arguments can control the execution of application and provide debugging options using an external debugger with seamless debugging experience between VP and hardware. To test any of the external SOC interfaces like I2C, SPI etc., we modelled agents that act as I2C master/slave depending on the configuration, so that all the functionalities associated with interfaces were thoroughly validated. We developed a mechanism to control the data flow from agents to the VP by using certain debug registers present in the main register map. The application can write to any of these debug registers to activate SystemC agent of choice and initiate data transmission from it to the peripheral under test. The agent monitors changes to these debug registers and behaves as configured. It initiates commands based on fixed values written to these registers.

The testbench envelops the connections of SystemC agents to their respective peripherals (see Figure 2). These agents were developed to provide a virtual communication interface with the VP, much like physical interface such as a UART interface with silicon or FPGA. A SystemC agent is a module which comprises of:

- Input and output ports that connect with corresponding output and input ports of the intended peripheral in the testbench e.g., RX and TX lines for a uart.
- A function to receive indication from the application to turn on and begin transmission.
- A function to send data as desired by the peripheral it is meant to communicate with.
- A function to receive data from peripheral and basic implementation for checking data sanctity.
- Functions to communicate (request and send responses) with a client-server application which in turn communicates with the external software testing framework.

The VP gives options to be run in a debug mode. The executable can be run from the Microsoft Visual Studio with the "-debug" option added along with application image (elf file) as an argument. This would launch the executable and it would wait for a debugger to attach. By separately launching the debugger, it can be attached to the waiting .exe and the application can then be debugged by stepping into the code through its main() function. The external debugger and the Visual Studio, both provide the options of setting breakpoints. When the application is running, the control can jump back and forth from the VP and the application via these breakpoints, thereby giving options to debug the application and VP together. This is very useful for the software team as they can create various testing scenarios and study the behavior of design in them. This greatly helps in reviewing existing code and adjusting if needed. Figure 3 shows an image of the Microsoft Visual Studio debug console log.



```
Microsoft Visual Studio Debug Console

        SystemC 2.3.2-Accellera --- Aug 27 2019 14:38:05
        Copyright (c) 1996-2017 by all Contributors,
        ALL RIGHTS RESERVED


 i2cAgent:get_scratch_reg7(): value is 0
UartAgent : SendData() : waiting for Transmission Start event
UartAgent : SendDMA() : waiting for Transmission Start event
i2cagent : master_init() : waiting for master init event
AudioDmaAgent : SendSample() : waiting for Transmission Start event
I2C : data_transfer() : waiting for transfer init event
 i2cAgent:get_scratch_reg7(): value is ff
UartAgent: Scratch register 7 written : value is ff
UartAgent:get_scratch_reg7() : notifying Senddata thread to write data to UART Rx
UartAgent::SendData() : Received Transmission Start Event
UartAgent::SendData() : Writing data :  U to UART RX PIN
UartAgent::SendData() : Writing data :  a to UART RX PIN
UartAgent::SendData() : Writing data :  r to UART RX PIN
UartAgent::SendData() : Writing data :  t to UART RX PIN
UartAgent:ReceiveDataNcheck() : Received char : (U ) from UART TX PIN
UartAgent::SendData() : Writing data :  T to UART RX PIN
UartAgent::SendData() : Writing data :  e to UART RX PIN
UartAgent::SendData() : Writing data :  m to UART RX PIN
UartAgent::SendData() : Writing data :  p to UART RX PIN
UartAgent::SendData() : Writing data :  i to UART RX PIN
UartAgent::SendData() : Writing data :  n to UART RX PIN
UartAgent:ReceiveDataNcheck() : Received char : (a ) from UART TX PIN
UartAgent::SendData(): Sent  a Characters to UART RX PIN
UartAgent : SendData() : waiting for Transmission Start event
UartAgent:ReceiveDataNcheck() : Received char : (r ) from UART TX PIN
UartAgent:ReceiveDataNcheck() : Received char : (t ) from UART TX PIN
UartAgent:ReceiveDataNcheck() : Received char : (T ) from UART TX PIN
UartAgent:ReceiveDataNcheck() : Received char : (e ) from UART TX PIN
UartAgent:ReceiveDataNcheck() : Received char : (m ) from UART TX PIN
UartAgent:ReceiveDataNcheck() : Received char : (p ) from UART TX PIN
UartAgent:ReceiveDataNcheck() : Received char : (i ) from UART TX PIN
UartAgent:ReceiveDataNcheck() : Received char : (n ) from UART TX PIN
UartAgent::ReceiveDataNcheck(): Received  a Characters to UART Tx
UartAgent::ReceiveDataNcheck() Sent data : UartTempin  received data : UartTempin Comparing
UartAgent::ReceiveDataNcheck() Both Matched : Test Passed!, updating the status by sending 'P' to UART RX!
 TEST PASSED !!!
 Test Passed !!!
 Exiting !!!
Time = 287326 ns
```

Figure 3: Visual Studio Console log

C. Unified Test Infrastructure for SystemC/FPGA/Silicon

The software team had a requirement to construct a unified testing interface for SystemC based VP, FPGA prototype and silicon so that the same test-suite would run across all three platforms seamlessly. Since the VP was available much earlier, they were able to develop this infrastructure quite early. A two-way communication channel needed to be established between the software test server and VP for processing requested command and respond back to the test server using interface's API. Test server would issue commands to VP to execute specified testcase and would keep on polling for the response from the VP. UART and I2C SystemC agents were used as control ports for exchanging command and response. The major challenge here was to establish communication channel between SystemC server and external test application. After thorough investigation, it was decided to use TCP/IP based control port (client/server model) interface for exchanging the data(command/response) between test server and VP. This client-server model is a C++ application which implements functions to send a specified length data to the VP and a function to receive back data from VP. Figure 4 shows the details of the flow.

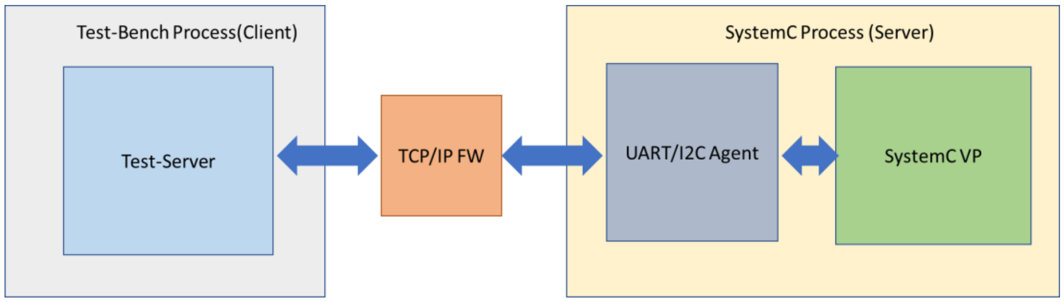


Figure 4: Unified Test Infrastructure

The SystemC process(server) could be any machine where SystemC model is running, and it is directly interfaced with the TCP/IP client through the SystemC agent in absence of actual physical interfaces. The client-server interface can be used to connect with a “test case parser” application running in the test PC. The software testing infrastructure in test PC creates a peripheral packet to be sent over to the VP via TCP/IP firmware as a byte stream. The packet includes information about the interface under test, a testcase number and some parameters. The application implements functions to decode this, selects the desired agent and sets the corresponding test application scenario to be run. Once the test run is completed, the selected control port agent returns the result to the test PC through the same TCP/IP client. Figure 5 below describes this flow in detail from the software perspective.

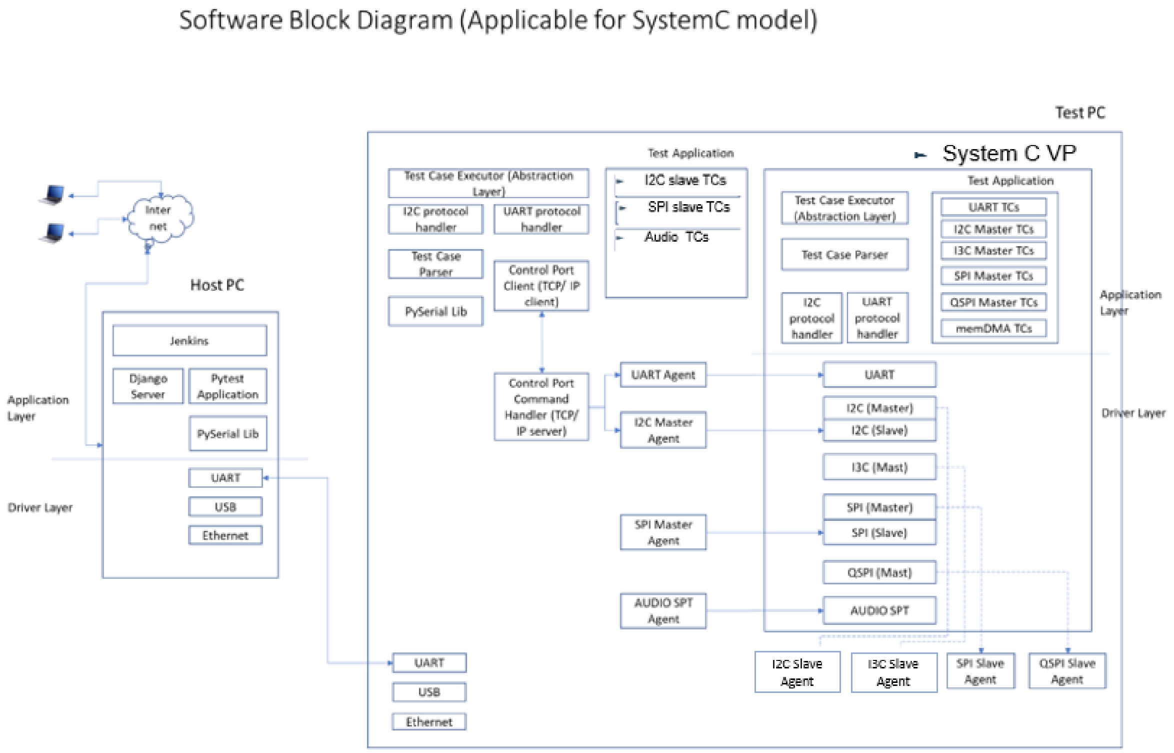


Figure 5: SW testing framework

This effort greatly helped validation team to develop test cases early and reuse the same testcases on FPGA based prototypes or Silicon. As the number of FPGA based prototypes available for a product is limited, it is always good to use VP in place of FPGA based prototypes where system timing is not a concern.

D. Feedback and issues found by SW team.

When bringing up a new platform, the primary development activity in software is the bring-up of the platform’s Software Development Kit or SDK and a reference application (bare metal or RTOS based) that leverages the SDK. To enable pre-silicon development, a FPGA is typically pressed into action. FPGA development schedules however, trail significantly behind the delivery of a Virtual Platform. The VP on this project allowed ADI engineers to design, develop, implement, and debug the SDK well ahead of FPGA availability. From a software perspective the development done on VP is no different than done on FPGA or on silicon. The Integrated Development Environment (IDE) is the same, the software sources are the same. The main difference is the interaction with the external agents.

To make the process of using the VP easier for the software team, the VP code was developed, and the executable was built using Microsoft Visual Studio solution. All the SystemC code repositories and library dependencies were accessible to the team so that every software engineer on the team could build the VP executable with updated code at their end and use it as they desire. This gave the team, the flexibility to have multiple users run different applications on it at their end. The software team ran applications for software drivers of SIC and the communication subsystem which varied from reading and writing various registers, testing loopback and DMA, checking the handling of all communication subsystem interrupts by the SIC driver. Some of these applications were re-used in the DV environment and validated software drivers were planned to be used to assist in developing DV test applications. Using VP, the software team also developed and validated the entire automated testing infrastructure much ahead of time.

We received a very positive feedback regarding the VP from our software team. Some key points were:

- **Ease of usage**: Since it is a software module and has no hardware dependency, it is very easy to use. Any external debuggers like Xtensa Explorer/ ARM DS / IAR debugger can connect to VP and provides seamless experience between actual hardware and VP. It gives powerful debugging features via Visual Studio (VS) and C++.
- **Debugging advantage**: Debugging with TLM models is much faster as compared to RTL, and it is possible to debug both device drivers and test code at the same time.
- **Impact on software validation schedule**: The software team developed and validated testcases for uart, Direct Memory Access and other communication subsystem peripherals (i2c, spi, i3c, qspi) without having to wait for the FPGA set up. They were also able to develop their testing infrastructure and established communication with external host PC without being dependent on the hardware.
- **RTOS**: Having the VP in place months in advance of the FPGA platform allowed the software team to bring up the RTOS, bare-metal and RTOS-aware device drivers and RTOS-based reference application. The team designed and developed inter-task communications and profiled RTOS-based implementation for code and data footprints and cycle count.

The VP was then used beyond its intended application of basic communication peripheral driver development and validation, with requests for more cycle accurate models to facilitate specific behavioral testing. Throughout the validation process, variety of basic issues were discovered in the communication peripheral drivers using VP e.g.,

- Uart driver couldn't handle subsequent transmission requests because of which the uart model in VP reported FIFO overflow errors. This prompted the review of driver level buffering.
- Uart driver was incorrectly handling the "set-break" operation in uart.
- During DMA, an interrupt handler was not correctly set for clear functionality which was debugged down to a single bit update using the VP.
- VP helped in identifying design issues like FIFO overflow/underflow mechanisms and the way of monitoring the FIFO itself.

E. Exploring UVM-SystemC with Elastic Testbench

Virtual prototyping is now being adopted at ADI with advent of automation and reuse in different projects across the company. One of the VP validation flows under exploration involves the use of Elastic Testbench or ETB. It makes use of the Accelera UVM-SystemC standard under the hood and provides an environment to configure required number of ports/sockets and would automate corresponding testbench agent's creation and connections. This configurability allows the ETB to drive scenarios to be tested on a VP with any number of TLM target sockets, input pins and output pins. It can also be configured to wait on response on certain output pin or pins which gives the flexibility of controlling the UVM reporting and self-checking mechanisms. UVM sequences can be written to create test scenarios that could be specific for validating block level models or a sub-system level use case. They can also be re-used in UVM based RTL verification environment. UVM-SystemC test environment supports a virtual sequencer, scoreboard, and reference model for result reporting. Alternatively, we have an optional setup to validate software (developed using VP) on to Emulators like Palladium to confirm the functionality. SystemC models are also being explored to be used as reference models for Datapath RTL verification.

We validated I3C, I2C and SPI block models using the ETB. Now it is being explored to create system-level use cases to be run on the VP. As a part of future work, options like constrained random verification and functional coverage that are supported by the UVM-SystemC standard are also under exploration. Figure 6 shows the set up for the ETB.

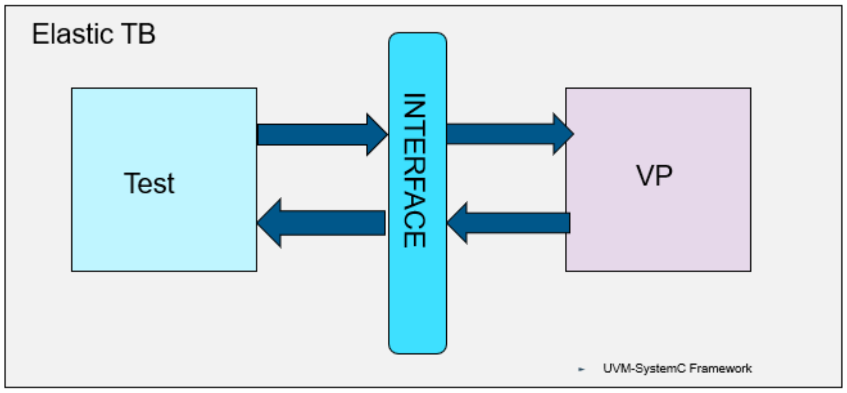


Figure 6: UVM-SystemC Elastic Testbench

III. CONCLUSION

The Virtual Platform was used for early development of the hardware dependent software (HDS) and its validation. Considering this was usually done once FPGA implementation is completed, it was a considerable gain in time and it also provided a medium for the software team to develop and validate their complete testing infrastructure well ahead in time. If the software development is entirely dependent on the FPGA, then we lose the initial project time that is needed for the RTL to be functionally correct for FPGA implementation. For larger and complex SOCs with multiple onboard processors, this is even more useful as FPGA bring up times are much longer given the design complexities. It is also not entirely straightforward to run applications involving multiple processors on the FPGA, which is a comparatively simpler task in the software oriented Virtual Platform. The key benefits are listed below.

A. Accelerate time to market and squeeze product development time.

The VP thus helped in development and validation of HDS, saving a generous 3 months in product development time. The reduction in time-to-market is directly proportional to the complexity of the SOC. The more complex the SOC, the larger is the gain in time saved. Following the current trend, we estimate an average saving of 6-12 months in time-to-market for more complex SOCs.

B. Agile process of the VP development

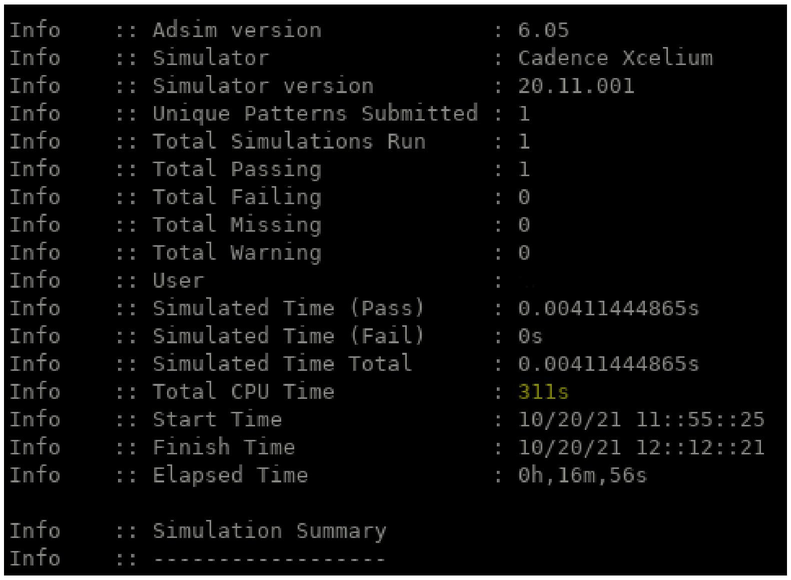
The VP followed the Agile development methodology. All model development was aligned with the software development and validation schedule and the availability of the design specification. Periodic discussions were done to keep this process aligned and phased releases of the VP were made as per the software schedule.

C. Component Reusability

Developed functional models can be re-used for subsequent projects, which saves considerable amount of time and effort involved in modeling.

D. Faster and easier navigation of simulation environment

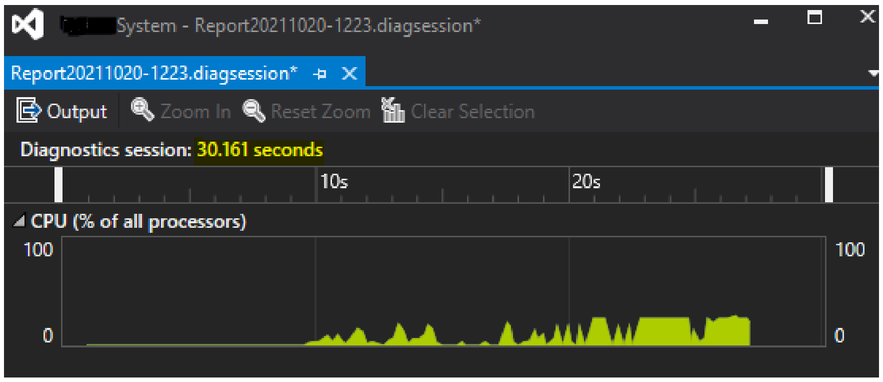
The VP offers an environment that is significantly faster than RTL simulations and provides powerful debug capabilities (through Microsoft Visual Studio), something that the software development team yearns for. Navigating through SystemC environment is way simpler than handling the UVM complexities associated with RTL environment.



```
Info    :: Adsim version                : 6.05
Info    :: Simulator                    : Cadence Xcelium
Info    :: Simulator version            : 20.11.001
Info    :: Unique Patterns Submitted : 1
Info    :: Total Simulations Run        : 1
Info    :: Total Passing                : 1
Info    :: Total Failing                : 0
Info    :: Total Missing                : 0
Info    :: Total Warning                : 0
Info    :: User                         :
Info    :: Simulated Time (Pass)        : 0.00411444865s
Info    :: Simulated Time (Fail)        : 0s
Info    :: Simulated Time Total         : 0.00411444865s
Info    :: Total CPU Time               : 311s
Info    :: Start Time                   : 10/20/21 11::55::25
Info    :: Finish Time                  : 10/20/21 12::12::21
Info    :: Elapsed Time                 : 0h,16m,56s

Info    :: Simulation Summary
Info    :: ------------------
```

Figure 7: RTL simulation speed


[redacted] System - Report20211020-1223.diagsession*

Diagnostics session: 30.161 seconds

Figure 8: SystemC simulation speed

E. Early Software validation and testing infrastructure development.

Software QA team developed and tested the software validation and testing infrastructure using the VP and got a 3+ months head start in software driver validation. Feedback from our software team, verbatim “Pre-silicon verification using SystemC opens a new world for software engineers since it is very approachable unlike the UVM/System Verilog environment. The simulation speed is fast for RTOS bring-up. Verification of software, done properly, is a difficult job. SystemC makes the effort much easier.”

In summary, virtual prototyping typically offers a solution for early system testing, but we applied it in getting a head-start for the development of hardware dependent software. In addition to early performance and architecture optimization, virtual prototype also enables a shift left of the entire verification and validation process. As a result, firmware and application software tests can be carried out much earlier and parallel to hardware development. In the future, fast, virtual product releases will move many other processes step up, that were previously only possible once product development was completed.

References

[1] https://www.accellera.org/activities/working-groups/systemc-verification
[2] https://www.accellera.org/activities/working-groups/systemc-verification/uvm-systemc-faq
[3] https://standards.ieee.org/standard/1666-2011.html