Responding to TAT Improvement Challenge through Testbench Configurability and Re-use

Kartik Jain, Renuka Devi Nagarajan, M Akhila, Component Design Engineer,

Mukesh Bhartiya, Engineering Manager, Intel Technology India Pvt. Ltd, Bangalore, India

Abstract— SoC integration is becoming increasingly complex and challenging due to the complex nature of IPs and the need to react to fast moving market expectations quickly. From a verification standpoint, the turnaround time can be much faster if the verification environment can handle higher levels of abstraction, enabling verification engineers to skip block level details of each VIPs/IPs and focus more on communication/interface between IPs. Also designers are cutting architecture exploration time by adopting System-C models, which is also pushing for a shorter time to first test.

In this dissertation we will introduce a verification methodology which delivers a configurable verification environment with multiple VIPs and enables connectivity to System-C drivers to reuse System-C unit tests with UVM SoC environment.

These methodologies focus on encapsulating core components with simple wrappers and significantly reducing dependencies of sequences and data checkers/monitors on transaction type.

This approach helps to improve TAT in two ways: Firstly, with the configurable test bench structure along with Fabric (AMBA) VIP wrapper scheme eliminates the need of re-establishing the verification environment from scratch for new SoC architectures. By using a pre-verified verification environment the user can focus more on test case stability rather than spending significant time on environment updates. Secondly, with the ability to switch between System Verilog and System-C drivers for the same interface, most of the block level tests can be reused at full chip level to quickly identify configuration/integration bugs.

This paper demonstrates a UVM based reconfigurable verification environment, built for a scalable SoC architecture. This paper highlights the creation of wrapper for primary fabric VIP, provision to have multiple drivers and transaction type independent end-to-end data checker. We achieved 2X reduction in time to first test as well as reduction in compilation and simulation time by 2X using the proposed techniques.

Keywords— Generic Test bench Structure; Fabric (AMBA) VIP configurable Wrapper; System-C drivers integration; Generic Base Sequences and Data Checker; UVM Advanced Features – Custom Phasing, UVM RAL;

Introduction

With SoCs integration challenges becoming more complex and TTM reducing, the TAT for verification is also becoming shorter. The verification environment has to be stable enough to absorb all the architectural updates at SoC level quickly. While the tests themselves are closely coupled to the actual design behavior; the environment, itself, should be minimally effected with these changes. There are many vectors which control the updates in the verification environment and we will be talking about those in the subsequent sections.

Figure 1 shows a generic SoC design which has a multilayered fabric (primarily AMBA based) connecting various commonly used IPs/blocks. The primary high speed fabric connects the core, memories and some basic peripherals like DMA etc. The other layer of fabric is connected to low speed peripherals viz. UART, I2C, I2S etc. SoC specific IPS can be either connected to the primary fabric or to another layer which talks to the primary fabric. Expected bandwidth and performance will govern such decisions.

The example here shows some commonly used VIPs like I2C, I2S, UART, SDIO and AMBA. Since the primary master of the system is connected to AMBA, the AMBA agent VIP is the primary controller of the sequence initiation. Multiple master agents' sequencer flow is controlled by UVM virtual sequencer which controls all the VIP sequencers and their transactions.

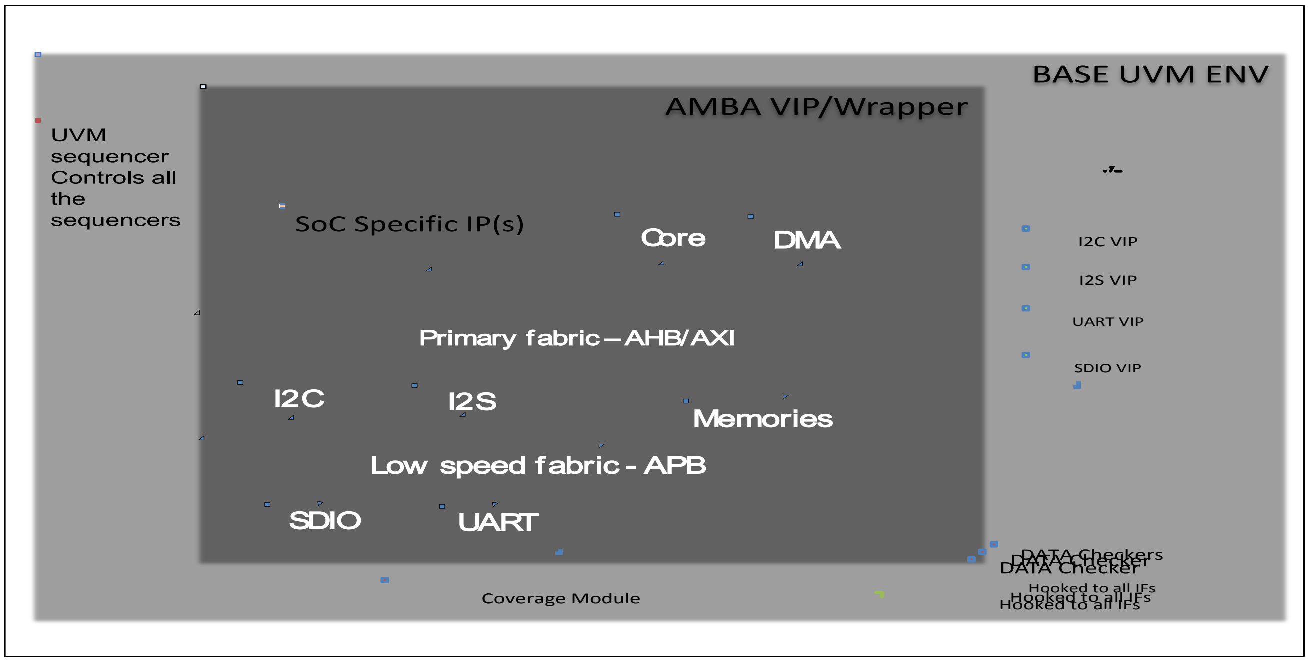

Figure.1 Baseline SoC architecture for testbench

The paper is organized as follows. Section II addresses the core areas that were targeted to get significant Re-usability and Re-configurability in SoC Verification Environment. The proposed implementation details for short TAT are explained in Section III followed by the results

Key Areas to Target for Re-usability and Re-configurability

Generic Verification Environment

During the initial phase of a project, the SoC architecture goes through many tweaks for power and performance. During this phase, the verification environment should be able to adapt to architectural changes quickly.

Fabric(AMBA) related Architectural Updates

Based on architectural experiments there could be change in Fabric (AMBA) configurations, which can essentially change the number of fabrics (AMBA) in SoC, number of masters and slaves in each fabric and other parameters. Main fabric (AMBA) VIP configuration is the most affected one and being the backbone of complete environment, it has major impact on verification. During architectural exploration phase; main fabric undergoes frequent changes either to meet required bandwidth or add new device support (peripherals). This will result in re-configuring many parameters of the AMBA VIP (viz. address map, slaves accessible to a particular master etc.) and hence touching the base configuration class again and again.

To minimize VIP configuration efforts due to such changes, a wrapper was created around the base AMBA configuration class which has simple function calls to configure the VIP w.r.t SoC configuration. It captures all prevalent parameters required to set the VIP and can be extended to add control to any other parameter of the VIP, as per requirement.

SoC's External Interface related Architectural Updates

Based on data transfer rates and sensor requirements, each SoC will have different set of interfaces to the external world. In order to ensure the connectivity and correctness of data transaction from

SoCs to board and vice versa, the corresponding VIP's will be connected which will mimic the behavior of on-board components.

The verification environment needs to have the capability of adding, removing and modifying the number of external components. Accordingly the test bench structure was created to take a SoC specific single input file and configure the pre-verified verification components. In SoC specific input file, user (verification Engineer) can specify the SoC requirements like number and type of fabrics/ external interfaces and their parameters.

Transaction Type Dependencies

Generic Base Sequences

Having VIPs in an environment will surely help build the environment quickly, but at the same time it's important to keep the environment immune to VIP changes. To build such a robust environment, we have limited ourselves to use any VIP specific transaction class object within only one sequence and its methods. Rest of the sequences and tests are extended from a generic base sequence. We have also implemented centralized objection handling mechanism as part of the base sequence to have a unified way of test flow

Generalised Data Checker

Typical data checkers are reliant on their transaction type on its analysis port. Transaction type of the checker can be any fabric (AHB/AXI/APB) or any other peripherals like UART, I2C, I2S etc., In order to address this issue we have decoupled the transaction type dependencies from the checker module. Base classes are developed to work on custom transaction type and transformation of VIPs transaction type is kept only at callback level.

Reusing Block level System-C tests

For new IPs, architects are moving towards adopting System-C for architecture exploration and developing functional models, which will normally be available for virtual prototyping in the early phases of the design cycle. These models come with extensive block level scenarios/ tests-cases regressed at functional model level. If SoC test-bench can reuse these models/ tests from unit level block environment, it will help fix some low hanging integration and configuration bugs in the IP.

To shorten time to first test and rerun some of the block level tests at full chip level, hooks were incorporated to choose between block level System-C driver and CRV based UVM driver. With this capability, basic block level tests can be run quickly to flush away basic configuration bugs.

Proposed Techniques for Faster TTM

Generic Verification Environment Implementation

Configurable wrapper to absorb Fabric(AMBA) architectural changes

AMBA VIP can be configured in many flavors with in-built monitors to watch for protocol violations and data sanity between master & slave. But this flexibility comes at the cost of configuring many VIP parameters with every change in fabric structure. Since the list of parameters is exhaustive, one need to be careful about every setting and any wrong configuration could lead to a false checker violation adding to simulation debug cycles.

This process consumes significant amount of time especially when fabric architecture under goes multiple changes during exploration phase and also may vary with project derivatives. To handle this challenge, a configurable wrapper was developed over AMBA VIP. The wrapper consists of simple function calls and encapsulates all VIP configuration specific details from user. This enables the user to limit all fabric related changes to one wrapper file

Let's assume a case where fabric gets a new AHB-Lite layer with 4 slaves all working on different clock and reset from the master. For this case below is the list of parameters that needs to be re-configured. Proper configurations of all parameters shown in Table-I below are required for VIP's expected functionality

S.N.	**Parameter/ Function to re-configure**
1	create_sub_cfgs(, ,) – Accepts number of different layer
2	Layer type configuration – AHB-Lite/ AHB
3	Number of slaves for new layer
4	Common clock & reset mode to define relation with master clock & reset
5	Master's address & data width
6	Per slave address & data width
7	Master active/ passive setting
8	Per slave active/ passive setting
9	Master address range
10	Per slave address range
11	Monitor configuration

Table I. Configuration Fabric Parameter

Instead of remembering all above parameters and settings, a onetime effort was put in to develop a wrapper that will make the environment less prone to configuration errors.

The wrapper takes input via the below 3 function calls as shown in Figure 2 and then configures the AMBA VIP under-the-hood.

set_new_layer ( ) - This accepts all layer related parameters such as layer type, number of masters & slaves, common clock & reset mode.

set_layer_master ( ) - This accepts master specific parameters such as active-passive mode, address & data width. Needs to call per master basis.

set_layer_slave ( ) - This accepts slave specific parameters such as active-passive mode, address & data width, address range. Needs to call per slave basis.

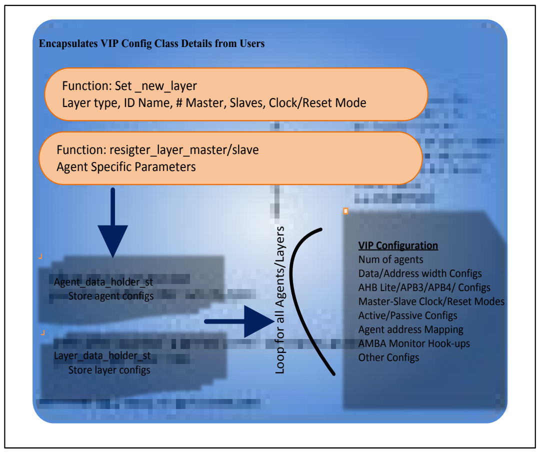
Encapsulates VIP Config Class Details from Users
Function: Set _new_layer
Layer type, ID Name, # Master, Slaves, Clock/Reset Mode
Function: resigter_layer_master/slave
Agent Specific Parameters
Agent_data_holder_st
Store agent configs
Layer_data_holder_st
Store layer configs
Loop for all Agents/Layers
VIP Configuration
Num of agents
Data/Address width Configs
AHB Lite/APB3/APB4/ Configs
Master-Slave Clock/Reset Modes
Active/Passive Configs
Agent address Mapping
AMBA Monitor Hook-ups
Other Configs

Figure.2 Configurable Fabric VIP Wrapper Structure

Apart from these, the wrapper needs some more inputs to map master-slave to layers. Wrapper holds all inputs in associate arrays which can be later queried to get component specific configuration for any debug purpose.

With this approach, verification environment can be easily tuned to any modification in fabric structure resulting in man-hour saving and prompt feedback to architecture team.

This also gives flexibility in terms of:

I. Constraining any major change in VIP configuration parameters within wrapper.

II. Turning-off VIP monitors on specific interfaces and instantiate custom checker.

Configurable TB to absorb External Interface changes

Each SoC will have a different set of communication mode, bandwidth, boot-up methods and power budget requirements. Based on those criteria the SoC will have a different architecture in all the aspects like fabric, peripheral types and numbers. The verification environment should be capable of handling these aspects of architectural changes. These issues were addressed as described below. Figure 3 shows a conventional system environment.

In a single input file, we will be specifying the architecture details as shown in Figure 4a. We have created a system environment to get the architecture inputs from a file and populate the corresponding verification environment.

Figure 4b shows the proposed configurable verification system environment generated by the SoC specific input file. This input file is created by user and has all the details of parameters which are needed by to match design characteristics. It also makes it easy for porting and single source control.

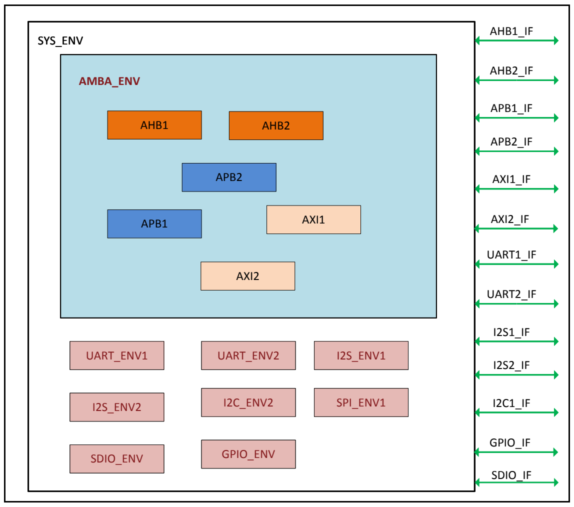

Figure.3 Existing Verification System Env Structure

```
AMBA VIP CONFIGURATION
int   layer_master          [0:`NO_OF_AMBA_LAYERS-1]     = {4,2,1,1,2,1,1,1,1,1,1,1,1};
int   layer_slave           [0:`NO_OF_AMBA_LAYERS-1]     = {5,1,1,1,5,5,1,1,5,5,1,2,2};
int   layer_clock_mode      [0:`NO_OF_AMBA_LAYERS-1]     = {1,1,1,1,1,1,1,1,1,1,1,1,1};
int   layer_reset_mode      [0:`NO_OF_AMBA_LAYERS-1]     = {1,1,1,1,1,1,1,1,1,1,1,1,1};
int   set_xaction_mon       [0:`NO_OF_AMBA_LAYERS-1]     = {1,1,1,1,1,1,1,1,0,0,0,1,1};

OTHER VIP CONFIGURATION:

  int uart_active[0:`NO_OF_UART_INSTANCES-1]                          = {1,1};
  int handshake_type[0:`NO_OF_UART_INSTANCES-1]                       = {0,0};
  int data_width[0:`NO_OF_UART_INSTANCES-1]                           = {8,8};
  int stop_bit[0:`NO_OF_UART_INSTANCES-1]                             = {1,1};
  int parity_type[0:`NO_OF_UART_INSTANCES-1]                          = {1,1};
  int baud_divisor[0:`NO_OF_UART_INSTANCES-1]                         = {1,1};
  int uart_receiver_buffer_size[0:`NO_OF_UART_INSTANCES-1]            = {16,16};
  int uart_enable_tx_rx_handshake[0:`NO_OF_UART_INSTANCES-1]          = {1,1};
  int uart_enable_dtr_dsr_handshake[0:`NO_OF_UART_INSTANCES-1]        = {0,0};
```

Figure.4A Snippet of SoC Specific Configuration input file

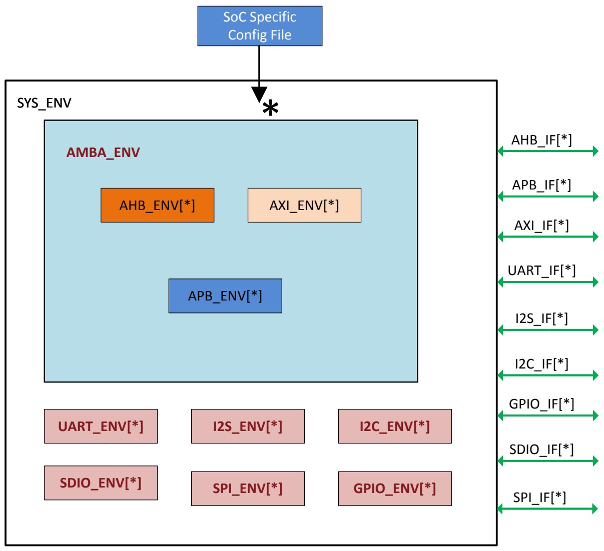

Figure.4B Configurable Verification System Env Structure

Developing Environment components independent of Fabric Transaction Type

Developing Generic Base Sequences (Independent of Fabric transaction Type)

In order to absorb frequent changes in architecture, we developed components that can work seamlessly with both AHB & AXI based systems and can easily accommodate new/mixed flavor of fabric. Sequences and tests reusability can be achieved only when they are not directly handling any specific type of transaction item. To improve reusability of sequences & tests, we extended all such components directly from UVM base classes and avoided any direct randomization of VIP AHB/ AXI transaction class. We implemented `uvm_do_* calls on VIP transaction class within our custom read and write tasks with appropriate arguments. All top-level sequences are calling these tasks without worrying about underlying master transaction type.

These tasks are implemented at leaf level and here we randomized AHB/ AXI transactions, based on task arguments. Any major fabric changes related to AHB, AXI or any other protocol will affect only leaf level tasks and that can be done in few working days; keeping rest of the sequences/ tests immune to such changes.

To have a unified way of test completion and avoid any dead-locks due to abrupt killing of sequences or tests, we also implemented objection handling mechanism as part of base sequence. This helps us to automate the objection raise-drop mechanism and users do not have to tackle this at different levels of sequences. Now every sequence raises its own objection at pre-start stage and drops it once its completed or killed.

Developing End–to–End Generic Data Checker (Independent of Fabric transaction Type)

AMBA VIP in-built checkers are good to check data integrity between master & slaves attached to AMBA bus but what about sanity of data appearing on output of attached peripheral?

Our verification environment required an end-to-end data checker that can handle in-order comparison between AMBA master and output of any peripheral attached anywhere in the system. This implies handling of many different transaction types from various VIPs and also system reference model output formatted in text.

Checker has to be generic enough to handle addition of any interface and hence works on its own transaction types. It is configurable to handle on-the-fly reset/ abort conditions, selectively turning off checker on any given interface, applying byte-mask before comparison and is able to dump data values at different levels for easy debugging.

To achieve this, it's needed to de-couple the checker from any type of VIP specific transaction and operate only on UVM analysis ports without any clock dependencies. Figure.5 depicts overview of checker operation

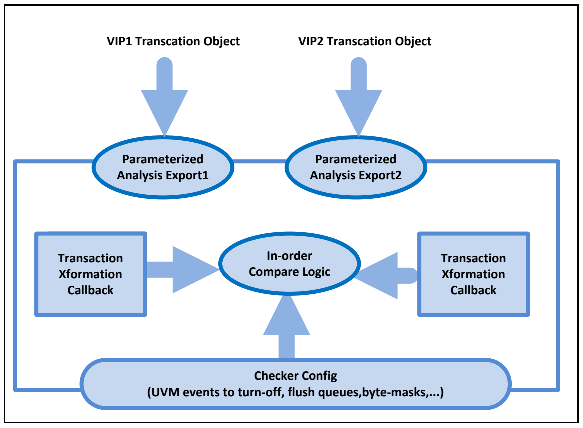
VIP1 Transcation Object
VIP2 Transcation Object
Parameterized Analysis Export1
Parameterized Analysis Export2
Transaction Xformation Callback
In-order Compare Logic
Transaction Xformation Callback
Checker Config
(UVM events to turn-off, flush queues,byte-masks,...)

Figure5.Checker Structure

As shown above, for any given interface checker accepts VIP specific transactions to xfer_convert callback class via UVM analysis ports and then passes on the transformed information to checker base. All analysis ports and transformation classes are parameterized and can be configured for given interface while instantiating checker. This enables checker base functionality to deal with its own pre-defined data format. Also UVM events and other configuration parameters are passed to handle resets, byte-masking etc.

Checker keeps pushing TX data in queues and performs comparison once RX data is available on analysis ports without depending on any clock. Callback hooks are provided to manipulate data before comparison, if required.

Hybrid Verification Environment manages between SystemC drivers and UVM drivers

One possible way to address TTM challenge is to get unified functional models across the front-end flow that can be used by both validation & virtual prototyping team. Re-using System-C unit tests with configurable UVM SoC environment is such an option.

One of the major components in System-C unit testbench is the system-C drivers. These System-C drivers that excite System-C models can communicate through TLM ports and are capable of driving logic values at pin level.

Verification environment had hooks for System-C drivers and was configurable to choose between System-C drivers and UVM drivers. UVM drivers were used to drive randomly generated stimulus whereas System-C drivers were used to re-run unit level tests at system level. This gave us flexibility to use UVM constraint random approach to focus on generating corner case scenarios at the same time reduced effort to migrate unit level System-C tests to UVM.

The drivers developed for Loosely-Timed(LT) and Approximately-Timed(AT) models are at transaction level and communicate to System-C models using TLM and drivers eventually becomes pin accurate when it starts feeding in to cycle accurate models. The connection between System-C driver and RTL depends on whether the System-C model is at transaction level or pin accurate level.

Support of TLM 2.0 in both System-C and UVM makes it fairly seamless to integrate System-C components with UVM environment. VCS-TLI supports auto-conversion of transaction payload class extended from generic TLI structure. For user defined payloads, conversion functions required by TLI have to be written. Complexity of conversion functions depends upon data types used in user defined payloads.

Integrating System-C pin accurate drivers are much simpler than writing conversion functions for user defined payload types. A sample TLI file needs to be created with the adaptor name same as the driver class name and then drivers are compiled along with the sample TLI file to dump out CPP header and SV file. The generated files take care of data type casting between System-C and SV and imports and implements DPI functions which help in cross language communication. These functions assign values to the driver signals based on the implementation and pass the same to the RTL as well.

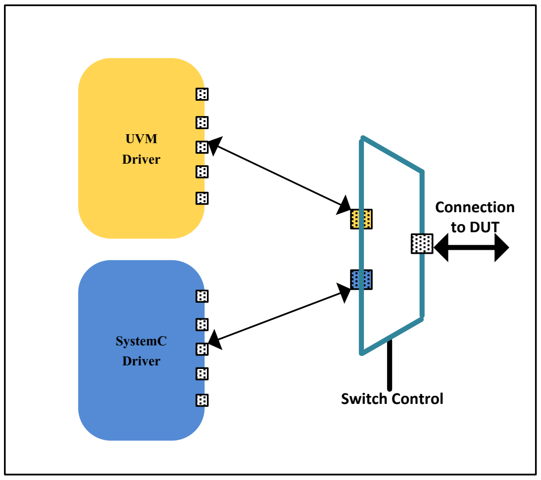

Figure6. System-C Driver Integration

Due to availability of System-C pin accurate drivers from unit level environment, we adopted 2nd method described above. This saved our effort of writing TLM-TLI ports and conversion functions essential for SC-SV communication.

This integration of System-C drivers gave us initial jump start in verification and helped to bring first test easily using existing unit level tests. Later in verification cycle, it was useful to replicate any unit level scenario at full chip level. This approach helped us by reducing directed test-case development to test basic functionality at system level. We could then focus our efforts on random scenarios generation using SV constraints and corner-case testing.

```
//SC Side
extern "C"
{
    void sv_call();
 }
//Based on this function call from SV, SC will start
the test.
 // SV Side
import "DPI-C" function void sv_call()
//Inside the sequence
sv_call()
```

Figure7. DPI call based handshake snippet

Other Approaches

Apart from approaches mentioned above, we also implemented some of the UVM advanced features as well as ways to optimize simulation performance & license usage. We have listed them as under:

UVM custom phase implementation & phase jumping

Implementation of custom phase, parallel to UVM main phase. This phase waits for few events trigger or timeout conditions to apply hardware reset to entire SoC and then jump back to UVM reset phase. This helps us to verify SoC behaviour on abrupt conditions.

UVM RAL

Used UVM RAL methodology at block level to get register coverage and understand UVM RAL flow.

Compile what you need

All VIP instances and relevant code are protected by separate defines. This gives the flexibility to compile only required set of VIPs as per test basis, resulting in lighter SIMV (simulation executable) and also optimize VIP license usage by not checking out all VIP licenses per tests.

Results and Comparison

The recommended techniques allow us to update the environment quickly and absorb the architectural changes. This in turn helps in bringing up the first test in initial few weeks of project. When compared to the previous projects, it reduced the initial environment setup time by 3-4X in terms of weeks. Figure8. compares the time taken to bring up the first test in verification environment

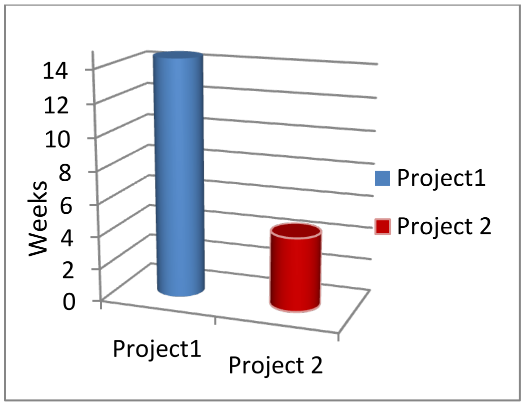

Figure8. Time to First test

Reduced compile and runtimes through selective compile and optimizing test-bench code post simulation profiling. This includes optimization in constraints blocks, avoiding any SoC status register polling and controlling amount of debug info output from test-bench.

Figure 9 shows the same:

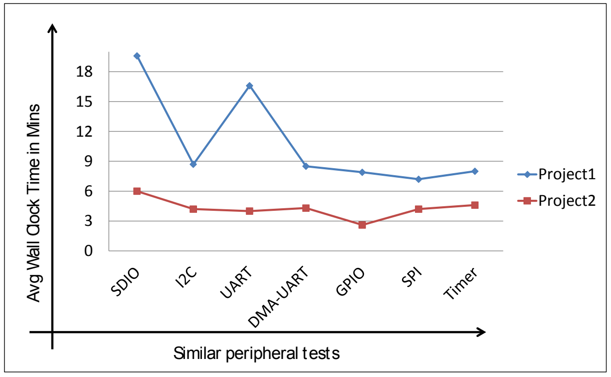

Figure9. Compile time and Runtime Comparison

Potential future enhancements

Implement configurable wrapper for other major environment components and if feasible try to generate configurations directly from XLSX using scripts.

Techniques to streamline top-level module which instantiates all VIPs and has VIPs monitor pin connections. As of now top module creation is manual and needs touch-up with changes in architecture.

References

[1] IEEE Standard System-C Language Reference Manual.

[2] UVM 1.1 User Guide.

[3] Synopsys AMBA VIP User Manual.