Scalable Test bench Architecture and Methodology for Faster Codec and Computer Vision Scenario Verification

Azhar Ahammad
Lead Engineer. Sr , Qualcomm Incorporated
Shreekara Murthy
Engineer, Qualcomm Incorporated

***Abstract*- The Video subsystem supports CODECs like HEVC, H264D, VP9D and AV1D. The Use cases of our video subsystem core include MCTF(Motion Compensated Temporal Filtering), visual special effect and other latest video features. Video core top level integration tests spread across multiple test suites for various categories like GPIO (General Purpose Input/Output), various categories of IRQs, a verity of automotive requirements like FUSA (Functional Safety), a bunch of AXI related functionalities, various spare register features, Clock controller modes and Debug subsystem related scenarios etc. Different chipsets are derived out of the existing architecture and new architecture comes with various updates around these categories of features. In these cases, the maintainability of the functional integration tests becomes a challenge test intent, the micro steps or sequences, checkers and coverage for each feature needs to be updated corresponding to the interface changes, tech node updates, new automotive requirements etc. The time involved in making these changes in the Testbench, identifying the targeted tests, and closing functional coverage on time is huge. This paper proposes an advanced methodology to resolve above challenges in top level integration verification by creating new scenarios and deploys an integration dashboard with functional coverage details. The architecture is implemented using synthesizable components and so can be ported over to any emulation platform for faster debugs. The new approach uses Feature Mapping Table (FMT) as a one-point configuration mechanism for this and generates test scenarios, checkers, and functional cover points accordingly and this avoids the manual. The configurability of the methodology makes it scalable. We have successfully implemented this methodology in our internal projects which resulted in 60% reduction in test update and coverage closure times.**

I. INTRODUCTION

Video subsystem integrates different blocks to perform control of video decode/encode processing and computer vision processing. The integration tests for the Video core are spread across multiple test suites for various categories. The complete list of the features and categories are mentioned below in Table 1, these features are integrated with the video core. There are close to 200+ feature characteristics which are to be verified. These characteristics were identified for one of our latest projects. With the current approach of running the functional or Formal connectivity checks all these features cannot be exhaustively verified, achieving complete toggle coverage for these features was not possible. The current approach checks the feature connectivity and the functionality. The tests at the core level can also be verified on Emulation platform to catch any bugs or functionality mismatch. Fig 1 represents the different features of our core.

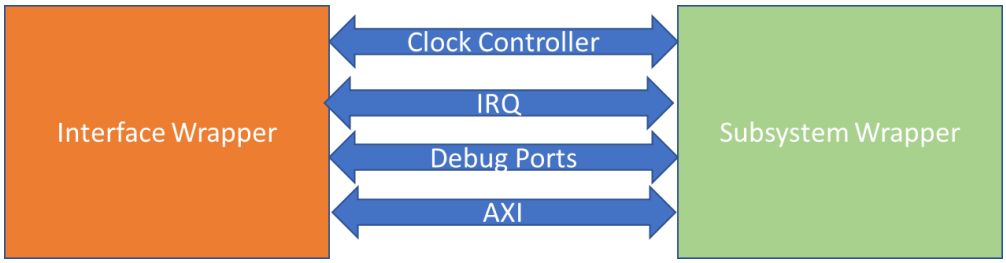

Fig 1: Features of the subsystem core

Feature Type	Description	Type of signals
AXI	AXI bus interface	Signal type
Debug SS	Debug SS is a hub of all debug related interface blocks	Signal type
Control Processor	Control processor is used used for various applications	Signal type
IRQ	Interrupt related signals for the RTL	Signal type
GPIO	General Purpose Input Output signals are used to program the DUT registers.	Signal type
FUSA	Functional Safety features corresponding to latest safety standards.	Signal type
Clock Controller	The Clock controller block is used to generate system clocks, reset signals and power control signals	Signal type

Table 1: Categories of Features

II. Problem Statements

The current approach to manage various categories of top-level integration tests is by running the functional tests in SV UVM environment or through formal connectivity checks. But complexity in maintaining proper checks at all the steps involved in a feature verification limits exhaustive verification of features. The problem with the current approach is,

- Maintainability of the functional integration tests becomes a challenge as the test intent, the micro steps or sequences, checkers for each feature need to be updated corresponding to interface changes, tech node updates, new automotive requirements etc.

- subsystem core top level has 300+ cover points and coverage point updates are manual for these design changes which makes it non-scalable across projects.

- The time involved in making these changes in the test bench, identifying the targeted tests, and closing functional coverage on time is huge (4-6 weeks).

Debugging of most of the top-level integration use case scenarios is difficult in emulation platforms since the test bench components are not synthesizable.

III. SOLUTION

This methodology aims to verify the feature sets with a generic flow which would minimize the DV efforts to run specific Testcases for targeted scenarios. The feature sets are first identified and fed as input to the Feature Mapping Table (FMT). The feature categorization is done based on the design requirements and can be updated easily across projects. The FMT can be seen below in **Table 2**, as seen in the Table the Source and destination signals need to be passed along with the conditions for the feature. The conditions include status registers based on which the Pseudo code will be generated.

type	tag_name	src_type	src_name	dest_type	dest_name	Condition
type name	Tag signal name	Src Name	Src Path	Dest Name	Dest Path	Conditions need to be input. Based on which the Pseudo code will be generated for the signals

Table 2: Feature Mapping Table Template

This methodology eliminates the coverage holes which was not earlier achieved through the functional tests. All associated values of the feature can be toggled by creating specific scenarios through functional tests. **Fig 2** represents the complete flow for the **Scalable Test bench Architecture for Top Level Integration Scenario Verification.**

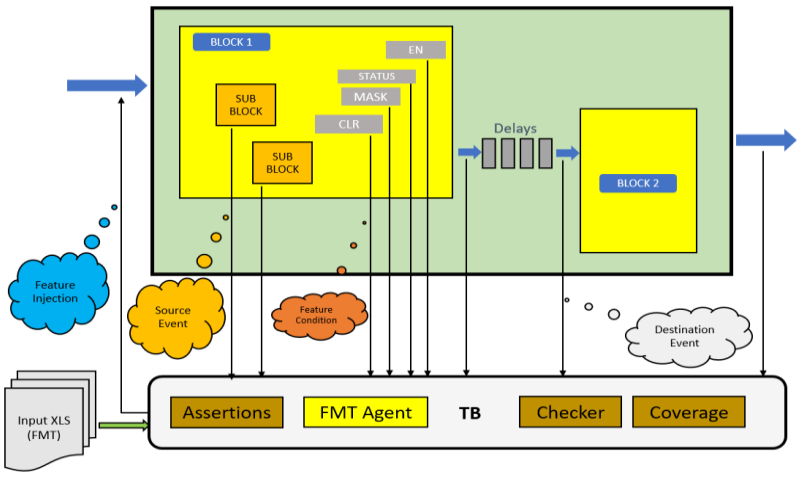

Fig 2: Test bench Architecture

IV. IMPLEMENTATION

In this new test bench architecture, the design specification and features are captured through the Feature Mapping Table (FMT). That is an XLS file which is fed to the Testbench Script which converts the input data to a Pseudo code which is integrated with the subsystem Testbench. This generates sequence of TB pseudo codes (TPC) along with the input and output (.in and .out) files. These files include the address of the register, data to be written or read by the register and the operation to be performed (Read/Write). The TPC Processing Engine (TPE) runs test sequences from the test case containing TPC. We can achieve auto generation of checkers and coverage associated with the features. The signals are tapped from the design and are instantiated hence the signals can be driven easily and toggled. The coverage dashboard gives the details regarding the coverage-points associated with the features. Below **Fig 3** is the representation of the flow.

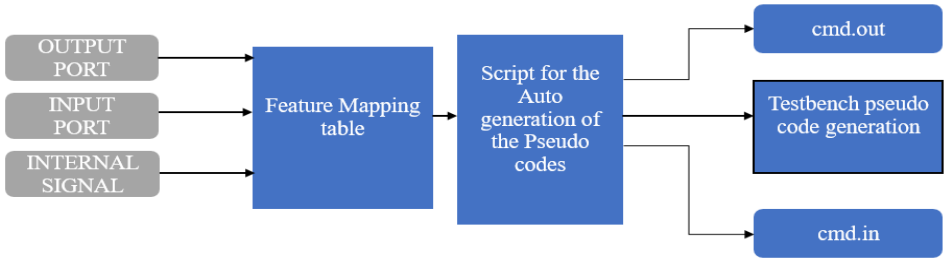

Fig 3: Top Level Integration Scenario Verification Testbench

As seen in **Fig 4** the Testbench master Wrapper consists of AHB Master, TB command memory, Command Parser, whereas the Slave Wrapper consists of TB Register Interface the AHB Slave. We have designed a separate arbiter logic for enabling the control for the implemented Testbench AHB master and the subsystem core AHB master. The main control logic is defined in the with Finite State machine(FSM), this controls the data flow. The RIF consists of all the details regarding the registers.

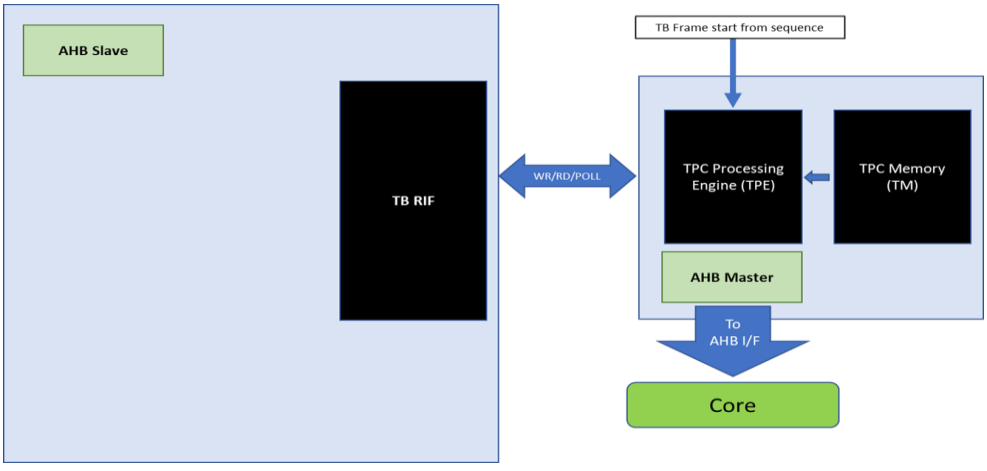

Fig 4: Implemented Testbench components

Based on the Opcodes the data transfer operations between the two masters can be controlled. The transactions are controlled through the Arbiter (i.e., Read/Write operation). We have so far defined 6 Opcodes for

both TB model and the core TB. More Opcodes can be added based on the design requirements. **Table 4** mentions the few Opcodes and the Operation associated with it.

No	Commands	Details	OPCODE
1	Testbench register Write	This is used for Write Operation of TB register.	Opcode Number
2	Testbench register Read	This is used for Read Operation of TB register.	Opcode Number
3	Data check for TB register data	This operation is used for data check of the received data on TB interface.	Opcode Number

Table 4: Opcodes

V. RESULTS

Benefits of this methodology are listed below.

- The effort on the verification time can be significantly reduced since the full flow is automated
- Easy maintainability and highly scalable across different projects since only FMT table needs to be reviewed and updated based on the design requirements.
- This approach gives us the complete configurable coverage dashboard information for integration verification review.
- Review of small steps covered in test sequence for verifying a feature will be easy since the dashboard captures all of them.

This architecture was implemented for one of our mobile chip project. The methodology was also verified for an automotive targeted project for the two features GPIO and IRQ. Complete functional coverage can be achieved with this approach. Since the test bench components are synthesizable the same architecture can be used on Emulation and Simulation platforms.

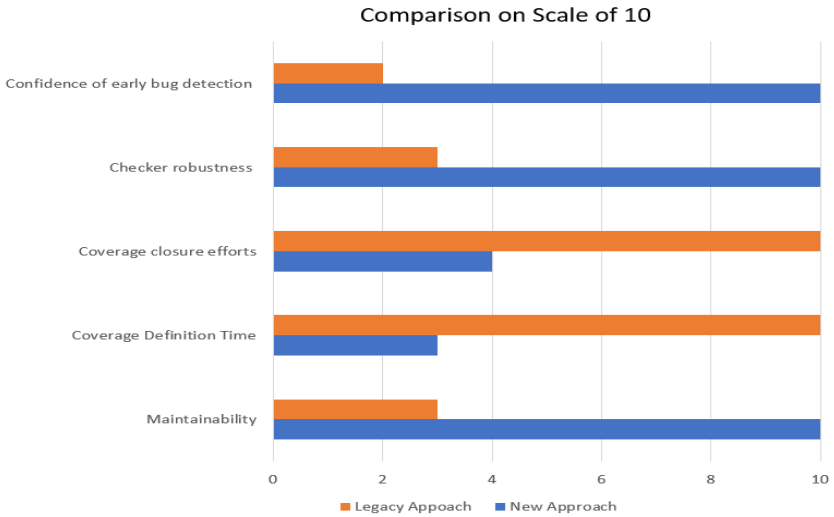

Fig 6: Comparison between the old and new approach

VI. Future Works

We plan to implement this architecture for all the integration scenarios. Enabling the debugger architecture to catch any bugs on Firmware platforms

VII. Conclusion

The proposed scalable test bench architecture with configurable coverage dashboard and easily maintainability across project changes gave significant improvement in the verification of integration features. The synthesizable code makes it suitable to use this on both simulation and emulation platforms which enables debugs of integration features seamless across platforms. The effort spent for achieving 100% functional coverage also can be reduced using this approach. The proposed design can be used in several categories for which functional or formal approach would take lot of time.

ACKNOWLEDGMENT

Authors would like to acknowledge Qualcomm Technologies, for providing the opportunity to carry out this experiment. Authors would also like to acknowledge each one who had supported in this activity