Adaptive UVM <-> AMOD Testbench for Configurable DSI IP

Krishnapal Singh, Nvidia, Hyderabad, India (*krishnapals@nvidia.com*)

Pavan Yeluri, Nvidia, Hyderabad, India (*pyeluri@nvidia.com*)

Ranjith Nair, Nvidia, Hyderabad, India (*ranjithn@nvidia.com*)

***Abstract*—The efforts required to create a functional verification framework scales up significantly with the increase in the complexity of hardware design. Moreover, for a new design, verification is obstructed by the availability of interfaces/ports, initial first-cut design. In this paper, we describe an approach where a pure C++ architectural model (AMOD) can be re-used ingeniously along with UVM modeling to verify the protocol-physical layer IP to shorten the IP unit level verification cycle. Protocol layer is less timing sensitive and requires transaction level accurate behavior to be met whereas PHY layer is more timing sensitive due to timing sequences, phase alignment marker insertions. This serves as the foundation of the described approach. This work is demonstrated using the work carried out at NVIDIA on the configurable MIPI DSI (Display Serial Interface) IP [1].**

***Keywords*—AMOD, C++, DPI, MIPI, DSI, PHY, DPHY, CPHY, UVM, SV, PPI, ECC**

I. Introduction

C/C++ has been in use as a preferred language to implement architecture behavioral model (AMOD). AMOD is used by the architects for algorithm/specification validation of the IP features. Also, the architectural model usage to enable early software development without being dependent on the RTL design availability has gained popularity in recent times. As the framework for cross-language communication of using C/C++ models along with system verilog modelling already exists which makes use of DPI (Direct programming interface) [2], one would assume that AMOD re-use in RTL verification would be widespread as well. However, the usage has been limited. Based on our exploration, a few of the general points understood for the limited use were:

- For designs which are less timing-sensitive and can be modelled almost completely using a transaction-based behavior modelling, the approach seems feasible. However, for designs where timing sensitive aspect is present as well, it becomes difficult to conceptualize and model interactive communication between C/C++ and system verilog.
- How do we ensure the correctness of the AMOD as this serves as the golden reference model if used for RTL verification?

In this paper, we attempt to address the above concerns and demonstrate the benefits of re-using the AMOD in RTL verification. The paper is organized as follows: Section II gives a brief overview on MIPI DSI and the layers present in DSI which helps to understand the functionality and complexity of the DSI IP and provides us the brief background understanding for Section III which details out the DSI testbench architecture. Section IV walks through the steps required to integrate AMOD in the UVM TB and explains the interactive communication between SV/UVM and AMOD during run-time through an example. Section V answers the concern mentioned above about how to ensure the correctness of AMOD functionality. Section VI specifies the results based on our experiments with DSI, section VII leads to the conclusion and section VIII mentions the scope of future enhancement.

II. MIPI DSI : A BRIEF OVERVIEW

MIPI DSI (display serial interface) specifies the high-speed serial interface between a host processor and a peripheral, such as a display module. It is widely used for display applications in smartphones, tablets, laptops and as well as in automotive market for dashboard displays and in-car infotainment systems. DSI has two variants which are described below:

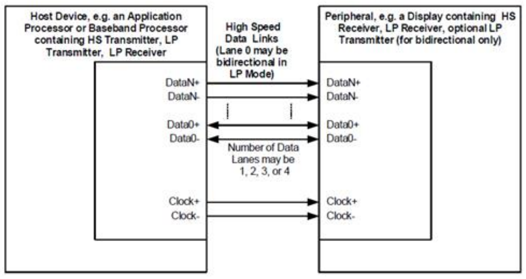

Host Device, e.g. an Application Processor or Baseband Processor containing HS Transmitter, LP Transmitter, LP Receiver
High Speed Data Links (Lane 0 may be bidirectional in LP Mode)
Peripheral, e.g. a Display containing HS Receiver, LP Receiver, optional LP Transmitter (for bidirectional only)
DataN+
DataN-
Data0+
Data0-
Number of Data Lanes may be 1, 2, 3, or 4
Clock+
Clock-

Figure 1 DSI Transmitter and Receiver Interface (D Option)

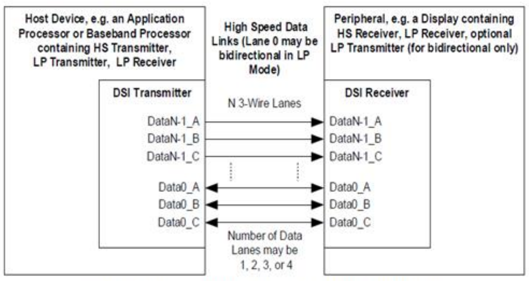

Host Device, e.g. an Application Processor or Baseband Processor containing HS Transmitter, LP Transmitter, LP Receiver
High Speed Data Links (Lane 0 may be bidirectional in LP Mode)
Peripheral, e.g. a Display containing HS Receiver, LP Receiver, optional LP Transmitter (for bidirectional only)
DSI Transmitter
N 3-Wire Lanes
DSI Receiver
DataN-1_A
DataN-1_B
DataN-1_C
Data0_A
Data0_B
Data0_C
Number of Data Lanes may be 1, 2, 3, or 4

Figure 2 DSI Transmitter and Receiver Interface (C Option)

DSI interface shown in Figure 1 (D option) has been in existence since 2009. It has a high-speed differential interface with one 2-wire clock lane and one or more 2-wire data lanes. The physical layer for this variant is known as DPHY. DSI interface shown in Figure 2 (C option) was first introduced first in 2014. It consists of the 3-wire lanes (which are also known as trios) and the clock information is embedded inside the 3-wire lane. The physical layer for this variant is known as CPHY.

The above figure describes the different layers of DSI:

Application layer: Provides the pixel data stream or the commands to the DSI controller.

Protocol layer: Forms the packets by appending (in case of transmitter) ECC (error correcting code), checksum and generates the header and the data payload information.

Lane management: Determines how the packet data should be distributed on the lanes. The lanes can be 1, 2, 3 or 4.

PHY layer: Mainly specifies the transmission medium characteristics, electrical signaling parameters, start of packet and end of packet signaling.

DSI Receiver (peripheral module) performs the reverse operations of those executed in the DSI transmitter (Application processor).

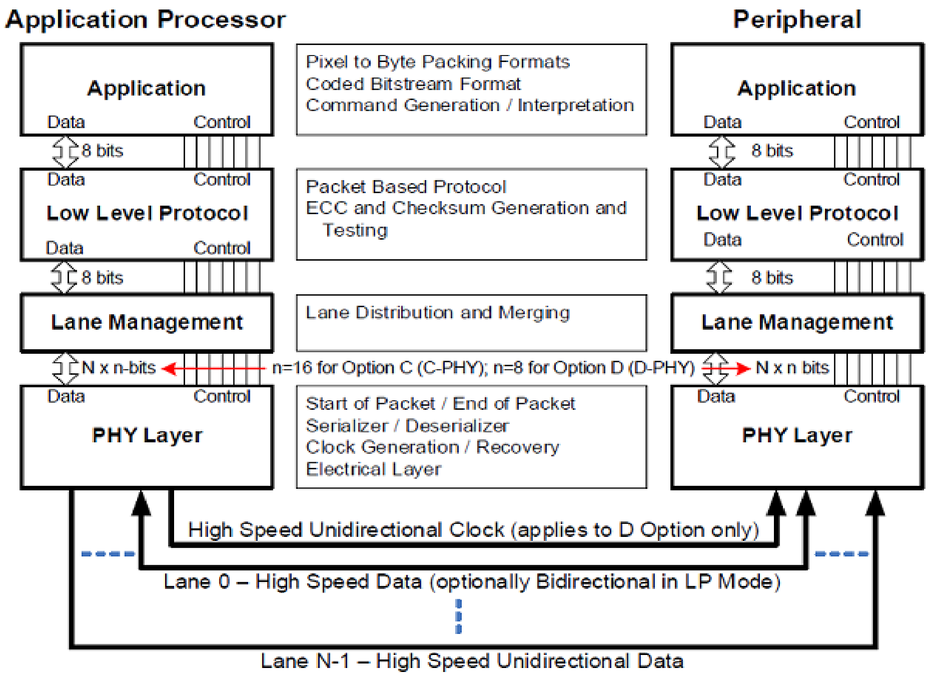

Application Processor
Peripheral
Application
Data
Control
8 bits
Low Level Protocol
Lane Management
N x n-bits
n=16 for Option C (C-PHY); n=8 for Option D (D-PHY)
N x n bits
PHY Layer
Pixel to Byte Packing Formats
Coded Bitstream Format
Command Generation / Interpretation
Packet Based Protocol
ECC and Checksum Generation and Testing
Lane Distribution and Merging
Start of Packet / End of Packet
Serializer / Deserializer
Clock Generation / Recovery
Electrical Layer
High Speed Unidirectional Clock (applies to D Option only)
Lane 0 – High Speed Data (optionally Bidirectional in LP Mode)
Lane N-1 – High Speed Unidirectional Data

Figure 3 MIPI DSI layers

III. DSI TESTBENCH ARCHITECTURE

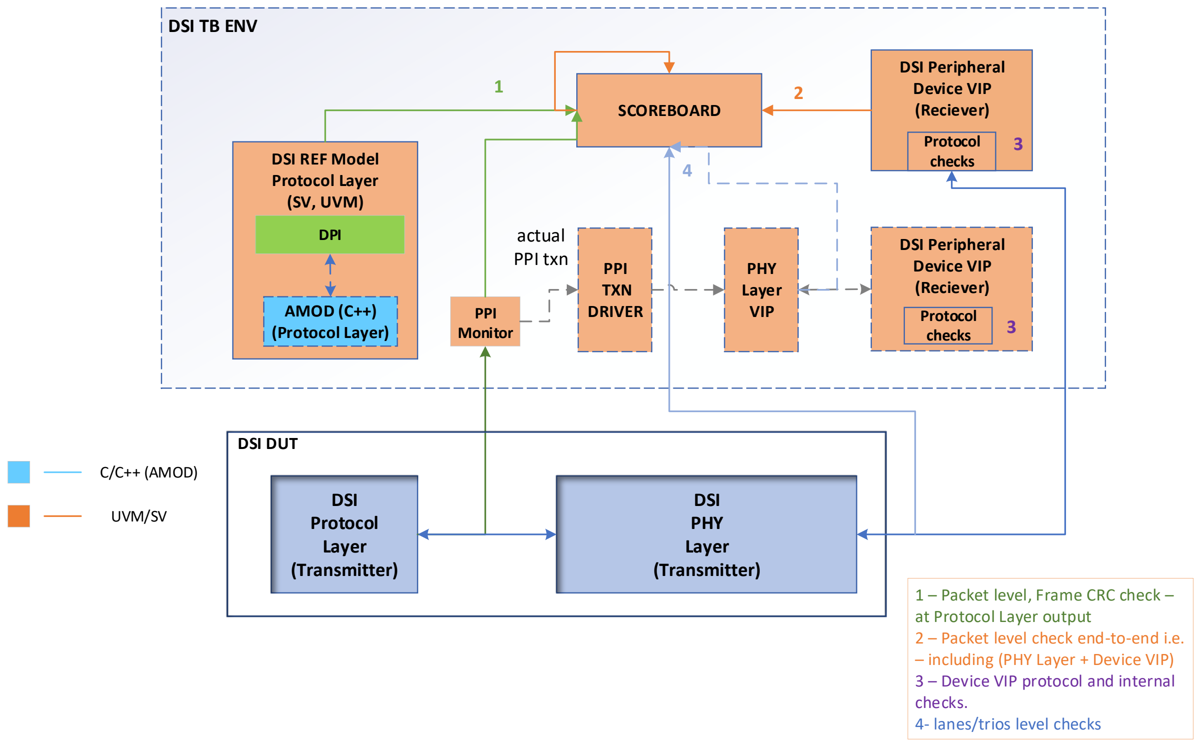

DSI TB ENV
SCOREBOARD
1
2
3
4
DSI Peripheral Device VIP (Reciever)
Protocol checks
DSI REF Model Protocol Layer (SV, UVM)
DPI
AMOD (C++) (Protocol Layer)
PPI Monitor
actual PPI txn
PPI TXN DRIVER
PHY Layer VIP
DSI Peripheral Device VIP (Reciever)
Protocol checks
DSI DUT
DSI Protocol Layer (Transmitter)
DSI PHY Layer (Transmitter)
C/C++ (AMOD)
UVM/SV
1 – Packet level, Frame CRC check – at Protocol Layer output
2 – Packet level check end-to-end i.e. – including (PHY Layer + Device VIP)
3 – Device VIP protocol and internal checks.
4- lanes/trios level checks

Figure 4 DSI Testbench Architecture

We will be using following term PPI – “Logical PHY-Protocol interface” in this paper, which is defined in MIPI DSI specification. This refers to the interface between DSI Protocol and PHY layer.

Following is the brief description and application of the testbench architecture:

- The AMOD implements the DSI protocol layer functionality which requires transaction-based accuracy to be met. The AMOD provides the transaction output expected at the protocol layer output.

- UVM model connects with the AMOD through DPI (Direct programming interface) [3]. It drives the AMOD input and collects the architectural model output and forms the expected PPI transaction. Expected PPI transaction is compared against actual PPI transaction received from Protocol layer output from DUT inside the SCOREBOARD for verifying protocol layer functionality correctness.

- Actual PPI interface transaction is used in the PPI TXN DRIVER to drive the input of the PHY layer VIP. PHY layer VIP implements the PHY layer functionality for transmitter written using UVM/System-Verilog as PHY layer involves timing sequences insertion. There are following two reasons for using actual PPI interface transaction and not expected PPI interface transaction to drive the input of the PHY layer VIP:
  - PPI output from DUT gives the data-rate/timing information to the PHY TXN DRIVER so that timing sequence insertion in the PHY layer VIP can be handled appropriately.
  - PPI transaction correctness is ensured with respect to transaction accurate handling inside the SCOREBOARD and timing sequence correctness is ensured inside the protocol and timing checks implemented inside the DSI Peripheral Device VIP (as mentioned in point below).

- DSI Peripheral Device VIP is hooked up at the PHY layer output from DUT and is used for checking the correctness of end-to-end functionality. It implements checkers for all the protocol and timing rules specified in the MIPI DSI specification.

- SCOREBOARD logic ensures the correctness of the functionality at-

- PPI interface.
- PHY layer output (lanes/trios).
- End-to-end functionality using DSI Peripheral Device VIP.

IV. INTEGRATING AMOD IN UVM TESTBENCH

The flow to provide the interface between system Verilog and C/C++ programming exists already [2]. We've listed down the steps below which we had followed for integrating DSI AMOD in UVM TB. These steps are generic and can be used for integrating any class based AMOD implementation in UVM TB:

- **AMOD implementation class**

```
/// DSI AMOD class
class dsiCore {
 public:
    dsiCore ();
    ~dsiCore();
    /// Function to process input pixels received and form the packets and store on the
PPI interface FIFO. Example function with 32-bit data and 1-bit valid signal
    void processPixel (const uint32_t data, bool valid);
private:
    /// PPI I/F Buffer
    queue <uint32_t> m_PPIData[2];   /// Buffer for Storing processed protocol layer
output PPI data.
};
```

- **Create wrapper functions to support AMOD member functions call from SV/UVM model**

```
#include "svdpi.h" /// include the system Verilog DPI header file
/// Create a static object for DSI AMOD class and initialize to NULL
static dsiCore* s_pDsiCore = NULL;
/// function to construct DSI AMOD class in UVM TB
extern "C" bool UVMSetDsiCore() {
    s_pDsiCore = new dsiCore ();
     return (s_pDsiCore != NULL) ? 1 : 0;
}
/// wrapper function for processing pixels
extern "C" void UVMProcessPixel (const uint32_t data, const bool valid) {
          s_pDsiCore -> processPixel(data, valid);
 }
```

- **Construct AMOD object in SV/UVM model**

```
import "DPI-C" function void UVMSetDsiCore ();  /// Import C/C++ function
/// Inside the build phase of the SV/UVM model
virtual function void build_phase (uvm_phase phase);
     super.build_phase(phase);
     /// AMOD constructor
     UVMSetDsiCore ();
 endfunction: build_phase
```

- **UVM/SV interactive communication during run-time – explained through uvm_callback.**
  Pixel interface used in the below example is an example input interface to the DSI controller which drives the pixel information to the DSI controller.

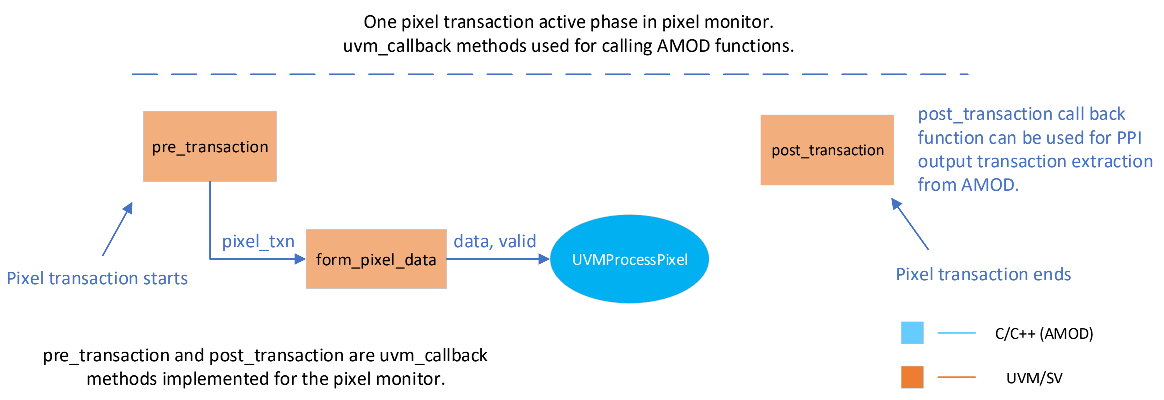

One pixel transaction active phase in pixel monitor.
uvm_callback methods used for calling AMOD functions.
pre_transaction
pixel_txn
form_pixel_data
data, valid
UVMProcessPixel
post_transaction
post_transaction call back function can be used for PPI output transaction extraction from AMOD.
Pixel transaction starts
Pixel transaction ends
pre_transaction and post_transaction are uvm_callback methods implemented for the pixel monitor.
C/C++ (AMOD)
UVM/SV

Figure 5 Interactive UVM/SV ←→ AMOD communication during run-time

The implementation logic for the callback method is as follows:

```
 /// Import the pixel processing wrapper function in UVM/SV model
  import "DPI-C" function void UVMProcessPixel (input int unsigned data, input bit valid);

 /// Extending callbacks for DSI pixel monitor
 class dsi_pixel_monitor_callback extends uvm_callback;
    /// Over-writing the pre_transaction task called at start of observed transaction.
    task pre_transaction (pixel_transaction tr);
       /// Form pixel data
       form_pixel_data (tr); /// to extract data and valid from pixel transaction

       /// Pixel processing in AMOD
       UVMProcessPixel(data, valid);
    endtask: pre_transaction

    /// Over-writing the post_transaction method called at the end of pixel transaction
    task post_transaction (pixel_transaction tr);
          /// Post transaction task can be used to process PPI output transaction from AMOD
    endtask: post_transaction
endclass: dsi_pixel_monitor_callback
```

V. ENSURING CORRECTNESS OF AMOD

An important question on the proposed approach we encountered was – "How is it ensured that AMOD is correctly implemented?". To serve as the golden reference model for RTL verification, it should be ensured that AMOD is implemented and validated appropriately.

For validating AMOD functionality, the following steps were followed:

- A standalone AMOD testbench written in C/C++ for validating the sanity features of the AMOD. C/C++ tests were written to program DSI and drive the input pixel interface by using appropriate function calls.
- For checking AMOD functionality correctness, we implemented data integrity checker by implementing a SINK model which does the reverse of the DSI controller operation i.e., takes the packet, strips-off the packet header and checksum information and extracts the output payload pixel data which can be compared with input pixel data.
- Figure 6 and the steps mentioned below details out the SINK model and data integrity checker implementation:

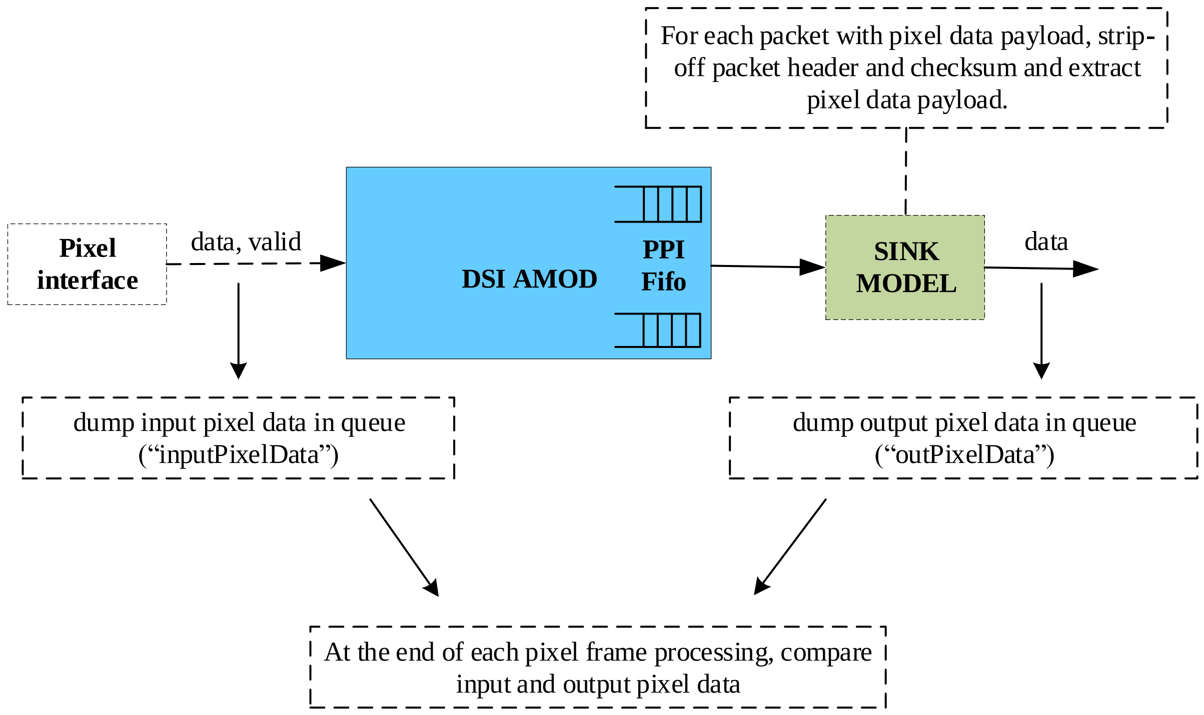

For each packet with pixel data payload, strip-off packet header and checksum and extract pixel data payload.
Pixel interface
data, valid
DSI AMOD
PPI Fifo
SINK MODEL
data
dump input pixel data in queue ("inputPixelData")
dump output pixel data in queue ("outPixelData")
At the end of each pixel frame processing, compare input and output pixel data

Figure 6 Sink model and data integrity checker for DSI AMOD

Steps used in the process:

- At the input Pixel interface, dump the valid pixel data into a queue (we'll refer it as the "inputPixelData").

- Implement SINK model at the PPI interface output, which does the following:
  - At the PPI interface of AMOD, for each packet with pixel data payload, strip-off the packet header and checksum information and extract the data payload. Dump the extracted data payload corresponding per packet in a queue. We'll refer it as "outputPixelData".

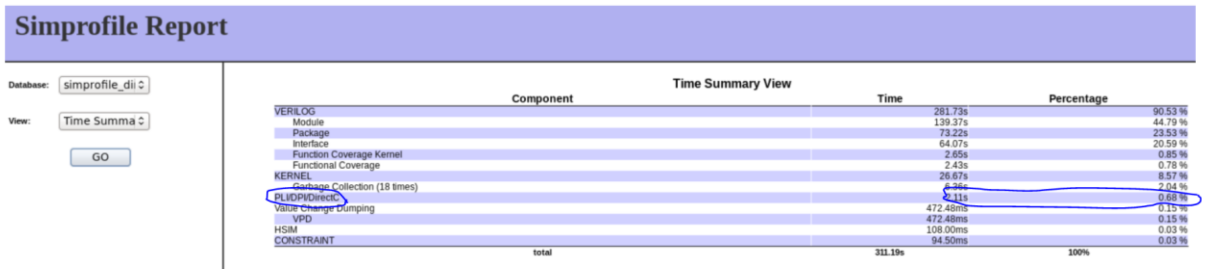
- For each pixel frame, do the comparison of "inputPixelData" vs "outputPixelData". This serves as data integrity check.

VI. RESULTS

Reuse of architectural model (AMOD) in IP unit verification provided us the following major benefits –

- Behavioral model creation efforts (for IP unit level verification) were reduced by ~50% as protocol layer functionality is directly reused from the architectural model. This, in turn, led to a reduction in overall IP functional verification cycle efforts by ~15%.
- Minimal efforts spent on fixing issues in protocol layer functionality modelling as the sanity feature correctness was already ensured in architecture verification. This proved to be highly beneficial during IP unit verification as we could focus more on finding design issues.

We also performed VCS simulation profiling in order to evaluate the performance impact DPI usage could have during run-time. Following analysis shown in Figure 7 below for an exhaustive pixel data transfer clearly indicates that DPI occupies a very minimal part of overall simulation print.



Simprofile Report

Database: simprofile_dii
View: Time Summa
GO

Time Summary View

Component	Time	Percentage
VERILOG	281.73s	90.53 %
Module	139.37s	44.79 %
Package	73.22s	23.53 %
Interface	64.07s	20.59 %
Function Coverage Kernel	2.65s	0.85 %
Functional Coverage	2.43s	0.78 %
KERNEL	26.67s	8.57 %
Garbage Collection (18 times)	6.36s	2.04 %
PLI/DPI/DirectC	2.11s	0.68 %
Value Change Dumping	472.48ms	0.15 %
VPD	472.48ms	0.15 %
HSIM	108.00ms	0.03 %
CONSTRAINT	94.50ms	0.03 %
total	311.19s	100%

Figure 7 Simulation profiling report for exhaustive pixel data transfer usecase for DSI

VII. CONCLUSION

As described in the current paper, the architectural model (AMOD) can be re-used in RTL verification as behavioral model with the use of DPI, even for the complex IP protocol. At NVIDIA, we have successfully used it for DSI IP unit level verification which benefited us in terms of effort reduction, early verification enablement and bring-up of end-to-end testing.

We strongly believe a similar approach can be utilized for other complex IP protocols as well where both transaction accurate and timing sensitive aspects are present.

VIII. FUTURE ENHANCEMENT

We have implemented and demonstrated AMOD functionality for protocol layer of DSI and re-used in verification. More pieces of IP can be developed in AMOD and re-used in RTL verification for further effort reduction. For example – for DSI CPHY, there is an encoder state and mapper stage which can be conceptualized to be implemented in AMOD, with only the timing sequence insertion required to be implemented in UVM/SV modelling.

REFERENCES

[1] https://www.mipi.org/specifications/dsi

[2] SystemVerilog Meets C++: Re-use of Existing C/C++ Models Just Got Easier – “John Aynsley Doulos john.aynsley@doulos.com”

[3] https://www.doulos.com/knowhow/sysverilog/tutorial/dpi/