Interfacing of Systemverilog and Systemc Using Transaction Level Modeling

Prem Kumar Lohani\textsuperscript{a}, Ranjani K\textsuperscript{a}, Ravishankar\textsuperscript{b}, Sundaresan C\textsuperscript{a}, Chaitanya CVS\textsuperscript{a}

\textsuperscript{a} School of Information Sciences, Manipal University, Manipal.

\textsuperscript{b} CEO, Whizchip Technologies Pvt. Ltd. Bangalore.

ABSTRACT

SystemVerilog and SystemC are extensively used for design and Verification in VLSI industry. This paper propose a method to combine SystemVerilog and SystemC code in a single hardware/software simulation which allows design teams to leverage abstract representations of system function as it increases system simulations speed. This paper also purposes a method of developing generalized Noise Channel Model to mimic the real timing scenarios in communication protocols and provide physical medium timing information to higher layers using SystemC.

Keywords: SystemVerilog; SystemC; TLM; Interface; VLSI; Verification.
1. Introduction

SystemC (SC) [6] and SystemVerilog (SV) [7] are two main standard system languages which are widely used in VLSI industry for design and verification. SystemC is a library developed in C++ [8], which is mainly used for System level and Hardware modelling and widely used by system architects. SystemVerilog is unified design and verification language which is built on Verilog with additional feature of object oriented programming concepts and widely popular among hardware engineers. SystemC and SystemVerilog have different approach to address system design process. They both provide standards for addressing system level design and verification challenges. System architect mainly require a flexible language that allow them to represent very complex hardware or software interactions starting from higher abstraction application levels to low level detailed description of each hardware and software components. On the other hand, hardware engineers require a language which is closer to HDL and also able to express high level system construct. Both designers introduce number of powerful methods which is needed for more accurate pre-silicon hardware verification and software validation. Proper Hardware verification and software validation are needed to reduce time to market and silicon re-spins. Each language addresses the needs of specific constituents in the system-design process. Following table represents main difference between SystemVerilog and SystemC for system design process [9].

<table>
<thead>
<tr>
<th></th>
<th>SystemC</th>
<th>SystemVerilog</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstraction Level</td>
<td>Events</td>
<td>Logic Gates</td>
</tr>
<tr>
<td>Architectural Design</td>
<td>System Level</td>
<td>Hardware Level</td>
</tr>
<tr>
<td>RTL –Gate level</td>
<td>Partially Supported.</td>
<td>Logic synthesis is possible.</td>
</tr>
<tr>
<td></td>
<td>(Need to modify for synthesis),</td>
<td></td>
</tr>
<tr>
<td>Verification</td>
<td>Co-Simulation</td>
<td>Supports ABV, constraint randomization and functional coverage</td>
</tr>
</tbody>
</table>

Table 1: Differences between SystemC and SystemVerilog

SystemC [6] builds on the system designer’s familiarity with C++ [8]. It extends ANSI Standard C++ with a C++ class library needed for system-level and HDL modelling. At the same time, SystemC offers Models of Computation (MOCs) that provide a bridge to more detailed representations including RTL. Using this approach, system-level designers can create detailed descriptions and perform cycle-accurate simulations at a speed that is nominally 1000X faster than equivalent RTL descriptions. SystemVerilog builds on the strengths of Verilog to allow hardware designers to represent more abstract system-level function in their designs. Using SystemVerilog’s interface between RTL and gate-level design, it accesses high-level descriptions written in C, C++, or even SystemC. As a result, hardware engineers can create detailed representations of complex systems, replacing abstract models with timing-accurate detail as individual block implementation proceeds. The ability to combine SystemVerilog and SystemC code in a single hardware/software co-simulation allow design teams to leverage abstract representations of system function as it is needed to speed system simulations. Indeed, both languages interoperate through an intermediate layer of abstraction: the Transaction-Level Models (TLMs) [10]. As a result, they allow designers to blend SystemC code and SystemVerilog RTL in co-simulation runs. Abstractions like TLMS provide a compact representation of complex functionality that runs dramatically faster in simulations than corresponding RTL models—even when using cycle-accurate transaction models.

Following are the advantages of cross language communication:

1. Abstraction Refinement.
2. Expansion of Verification IP Inventory.
3. Leveraging language strengths.

To provide interoperability i.e. component must be reusable and easy to use, each language component must agree on

1. Information of exchange (data type)
3. Direction of exchange (initiator or target)

The Open System C Initiative (OSCI) TLM community managed to create a standard on which models could be developed for re-use and interoperability. Communication protocol such as Ethernet [1], PCI express [2] or USB [3] provides exchange of information between devices. As per the OSI model [4], physical layer provides actual interconnection between devices. In physical layer due to thermal and environmental characteristics of the physical medium, noise is generated and distorts the information carried by the physical medium.

The physical medium noise constraints must be considered to provide the real time environment for communication protocol during simulation or verification. Consideration of noise or timing parameter during the simulation will result in more accurate verification of communication protocol. In general, physical layer involves relationship between a device
and a transmission medium, such as copper or fibre optical cable. The major functions and services performed by the physical layer are:

1. Establishment and termination of a connection to a communications medium.

2. Conversion between the representation of digital data in user equipment and the corresponding signals transmitted over a communications channel. These signals are operating over the physical cabling.

During the transmission, physical and electrical characteristics of devices and physical medium affect the information to be exchanged. These electrical and thermal characteristics generate noise parameters such as jitter [5], skew etc. Due to above mentioned parameters, timing violation occurs in the communication protocol and are responsible for data loss. The above effect of noise parameters are observed in real time. To accommodate such real time scenario during the simulation for comprehensive studies creates requirement of generalized noise channel model. In order to allow for comprehensive studies on the physical layer noise parameters, high-level modelling of the effects of noise can be used for the performance analysis in circuits and systems which are prone to errors. The modelling is based on a SystemC or SystemVerilog description because of its system level modelling capabilities and of its fine time granularity. The rest of the paper is organized as follows: Evolution of TLM standard is discussed in section 2. Section 3 discusses about interfacing SystemVerilog and SystemC. Noise channel model is discussed in section 4. Section 5 discusses about different noise parameters. Noise channel transaction format is discussed in section 6. Section 7 discusses the implementation of noise channel models. Results are discussed in section 8. Last section presents our conclusion.

2. Evolution of TLM standard

In the beginning there was C, then C++ [8]. For obvious reasons these widely known software languages were used to develop in-house system level models for doing high level performance and architectural analysis. The use model was very limited as the simulation behaviour was very far from the actual hardware. It lacks concurrency and timing. C++ extensions augmented with a “proof of concept” simulator was created and called SystemC that enables modelling these hardware behaviours in C++, and more. That worked very well. Failing to replace Verilog and VHDL for design and verification, SystemC found its niche use model in the architectural modelling domain. At that point architects using SystemC started to demand a standard that enables interoperability for fast platform composition, ease of use, and re-usability. Hence, the OSCI SystemC Transaction Level Modelling 1.0 and later 2.0 standards were created. TLM-2.0 standard [10] is targeted at the integration of Transaction-Level component models around an On-Chip communication mechanism. TLM-2.0 provides a standardized modelling interface between Transaction-Level Models of components. The primary focus of the SystemC TLM-2.0 standard is on speed and interoperability. Speed means being able to execute application software at as close to full speed as possible and TLM-2.0 sets very aggressive simulation performance goals in this respect. Interoperability means being able to integrate models from different sources with a minimum of engineering effort, and in the case of integrating models that use different bus protocols, to do so without sacrificing any simulation speed.

3. Interfacing SV_UVM-SC

The top level connection model between SystemVerilog and SystemC is shown in figure 1.

Figure 1: SV_UVM-SC TLM Connection Module

There are two main components at top level, which are developed in two different languages (SystemVerilog and SystemC) and both are communicating using Transaction Level Modelling (TLM). These components are described as follows: SV_UVM_GEN: This component generates transaction and developed in UVM [11] [12]. In the current paper, Ethernet L2 packet [1] has been considered as transaction type. USB [3] or PCI [2] type transactions can also be considered. Transactions are randomized in this component and sent out through TLM port. SC_PRINTER: This component receives transaction generated by SV_UVM_GEN component and developed in SystemC. Component receives Ethernet L2 packet transaction through TLM export.

3.1 UVM Connect:

UVMC [13] library provides TLM1 and TLM2 connectivity and object passing between SystemVerilog and SystemC models and components as shown in figure 2.
Some of the features of the UVMC:

- UVMC library is a separate optional package to the UVM.
- Existing TLM models in both System Verilog and SystemC can be reused in a mixed-language context without modifications.

While communicating verification components must agree on the data they are exchanging and the interface used to exchange that data. TLM [10] connections that are parameterized to the type of object (transaction) help components to meet the requirements mentioned above. Thus easing integration cost and increases their reuse. UVMC provides connect and connect_hier functions to make connections across SV-SC language.

SV TLM2:

```
svmc_tlm#(trans)::connect(port_handle, "lookup");
svmc_tlm#(trans)::connect_hier(port_handle, "lookup");
```

SC TLM2:

```
svmc_connect(port_ref,"lookup");
svmc_connect_hier (port_ref, "lookup");
```

**port_handle or ref** – specifies the handle or reference to the port, export, import, interface or socket instance to be connected.

**trans** - specifies the transaction type for unidirectional TLM1 ports, exports and imports. This is only needed by SV.

**lookup** - an optional lookup string to register for this port.

### 3.2 Implementation of UVM Connect:

The implementation of UVM connect is shown in figure 3. The description of internal process flow is discussed in the flowing section.

#### 3.2.1 Description of internal process flow:

![Internal Process Flow Diagram](image3)

**Figure 3: Internal Process Flow**
During the connection call the “target socket proxy” and “initiator socket proxy” are used. These proxies satisfy the requirements that ports and exports are connected in the native language prior to the communication across the language boundary. Target socket proxy acts as target for SV_UVM_GEN’s (producer’s) initiator socket. Initiator socket proxy on SC side serves as the source of the transaction for the consumer target socket. Initially Initiator socket proxy will be in the ideal state and waits for the bits arrival from the SV side. When the execution of run task in the producer (SV side) starts, it creates transaction object. This object arrives through the initiator socket of producer to the target socket of Target socket proxy. After transaction arrives, “do_pack” method of converter converts it to bits. This stream of bits is sent across the language boundary using standard DPI (Direct Programming Interface). Once the transaction is sent in a non-blocking fashion from the SV side, Target socket proxy waits for the return of bits. As soon the SystemC receives transaction bits, it creates a new object representing our transaction. This new object is unpacked by the converter using “do_unpack” method. Once converter has reconstituted the transaction object, blocking transport method will be called through the initiator socket and makes way to the target socket of consumer for execution. When it returns from “b_transport”, transaction is complete. Again transaction is packed into bits using converter “do_pack” method in the Initiator socket proxy. Packed bits are sent back from SystemC side using standard DPI, that freezes wait process for return bits in SV side and this happens in non-blocking fashion. After bits are sent, it loops back to the wait process for new bits for next transaction. On SV side it resumes the process from wait for return bits. Since return bits are available it is unpacked using “do_unpack” method and sent back to the originating call in producer. Above described process continues as transaction initiates.

3.2.2 UVMC Converter

If components were written in a common language, we could guarantee the exchange of compatible data simply by using the same transaction type. Such components are strongly typed. Any mismatch in type would be caught by the compiler. Now if two components were developed such that each agreed more or less on the content of the transaction, but this time their transaction definitions were of different types. This condition is always the case between components written in two different languages—they cannot possibly share a common transaction definition. To get such components talking to each other requires an adapter, or converter, that translates between the transaction types defined in each language. UVM Connect imposes very few requirements on the transaction types being conveyed between TLM models in SV and SC, a critical requirement for enabling reuse of existing IP. The more restrictions imposed on the transaction type, the more difficult it will be to reuse the models that use them.

Following are the advantages of UVMC converter:

1. No base classes required. It is not required that a transaction inherit from any base class to facilitate its conversion—in either SV or SC. The converter is ultimately responsible for all aspects of packing and unpacking the transaction object and it can be implemented separately from the transaction proper.

2. It is not required that the transaction provide conversion methods. The default converter used in SV will expect the transaction type to implement the UVM pack/unpack API, but we can specify a different converter for each and every UVMC connection we make. Our converter class may opt to do the conversion directly, or it can delegate to any other entity capable of performing the operation.

3. It is not required that the members (properties) of the transaction classes in both languages be of equal number, equivalent type, and declaration order.

3.2.3 Implementation of UVMC Converter:

In order to make communication between SV Ethernet L2 Packet Generator and SC Ethernet L2 Packet Printer [1] which has a user defined transaction (non-TLM generic payload), converters should be defined in both languages. Defining a converter involves implementing two functions—“do_pack” and “do_unpack”—either inside our transaction definition or in a separate converter class. Although the means of conversion are similar between SV and SC, differences in these languages capabilities cause differences in conversion. There are several options available for writing converters in SV and SC. Following option can be used for SV transaction: In-Transaction: This approach defines the conversion algorithm in transaction class itself. A transaction in UVM is derived from “uvm_sequence_item”, which defines the “do_pack” and “do_unpack” virtual methods for users to implement the conversion functionality. This option is the recommended option for SV-based transactions. To use this approach, authors declared and defined overrides for the virtual “do_pack” and “do_unpack” methods in our transaction class. virtual function void do_pack (uvm_packer packer); virtual function void do_unpack (uvm_packer packer); Most transactions in SV should be defined this way, as it is prescribed by UVM and it works with UVM Connect’s SV default converter. Converter Specialization: A separate class can also be defined for converting transaction type. In SC, conversion for a transaction is typically defined in a separate class called a “template specialization”. C++ allows specializing default converter implementation for a specific transaction type. When implementing the converter’s “do_pack” and “do_unpack” functions, we stream our transaction members to and from the “packer” variable, an instance of “uvmc_packer” that is inherited from an internal base class.

To pack fields into the “packer”, following syntax can be used. For example:

```cpp
packer<< t.cmd <<t.addr<<t.data;
```

To unpack fields from the “packer”, following syntax can be used. For example:
packer>> t.cmd >>t.addr>>t.data;

4. **Noise Channel**

The top level block diagram for the noise model is shown in figure 1.

![Block diagram of noise model](image)

There are two main components at the top level, which are developed in two different languages SystemVerilog (SV) [7] and SystemC (SC) [6], and both are communicating using Transaction Level Modelling (TLM) [10]. These components are described as follows: SV_Lane_Generator: This component generates lanes value as transaction and developed in UVM [11][12]. Transactions are randomized in this component and sent out through TLM port. SC_Noise_Channel: This component receives lane as transaction generated by SV_Lane_Generator. This component models noise channel in SystemC by computing noise parameter (such as jitter, skew) for each lane and update transaction with computed noise parameter. Component receives transaction through TLM export.

5. **Noise parameters**

There are five types of noise parameters which can cause effect during the transmission. They are:

1. **Transition Time**: Transition time (Rise time and fall time) is defined as time taken by the signal to change from 20% to 80% or 80% to 20% signal level respectively, of isolated edges.

2. **Jitter**: Jitter may be caused by the electromagnetic interference (EMI) and the crosstalk with carriers of other signal lane [10]. The measured jitter at the transmit and receive compliance point shall be less than the maximum Total Jitter, maximum Deterministic Jitter, Random Jitter and duty tolerance Jitter.

3. **Lane Skew**: Skew (or relative delay) can be introduced between lanes by both active and passive elements of a link. Skew is defined as the difference between the times of the earliest lane and latest lane for the one to zero transition of the alignment marker sync bits.

4. **Cable delay**: Cable delay is the amount of time required for cable to transmit signal to travel from the transmitter to the receiver. It can be computed as the ratio between the link length and the propagation speed over the specific medium.

5. **Return Loss**: Return loss is a measure of the reflected energy from a transmitted signal. These reflections are caused by impedance discontinuities in the channel. These discontinuities may be due to several things such as connectors, improper manufacture. Any energy that is reflected reduces the power of the transmitted signal. It is commonly expressed in positive dB The Larger the value, the less energy that is reflected.

6. **Noise channel transaction format**

Following figure represents transaction of noise channel.

<table>
<thead>
<tr>
<th>Lane 0</th>
<th>Lane 1</th>
<th>Lane 2</th>
<th>Lane 3</th>
<th>Lane0 Delay</th>
<th>Lane1 Delay</th>
<th>Lane2 delay</th>
<th>Lane3 Delay</th>
<th>Return loss</th>
</tr>
</thead>
</table>

![Transaction of Noise Channel](image)

Lane 0: Represents signal value on lane 0.
Lane 1: Represents signal value on lane 1.
Lane 2: Represents signal value on lane 2.
Lane 3: Represents signal value on lane 3.

Noise channel model computes total delay based on different noise parameter and update following variable.
Lane0_delay: Stores calculated total delay for lane 0.
Lane1_delay: Stores calculated total delay for lane 1.
Lane2_delay: Stores calculated total delay for lane 2.
Lane3_delay: Stores calculated total delay for lane 3.
Return loss: Stores Return loss.

7. Noise channel model implementation

Following flow chart describes noise channel model flow in implementation of blocking transport interface (b_transport).

![Flow Chart Noise Channel Model](image-url)

**Figure 6: Flow Chart Noise Channel Model**
7.1 Jitter model:

Accuracy of Jitter is specified in ppm. Jitter can vary between specified ranges. Calculations of jitter accuracy range in time unit is as follows:

\[
\text{Jitter accuracy} = \text{jitter} \times \frac{\text{ppm value}}{10^5}
\]

Following figure shows the flow chart for jitter model.

![Flow chart Jitter model](image)

7.2 Skew model:

The pseudo code to model skew on lanes is given below.

1. Set modify_skew = Maximum Skew.
2. Receive lanes
3. for (i=0 ; i<number of lanes; i++) {
4.  double x[i] = get_skew() (Calculate skew for each lanes)
5. }
6. Send lanes out.

get_skew() function is used to calculate skew for all lanes based on specified maximum lane to lane skew. Addition of calculated value of lane skew value must not be greater than specified Maximum lane to lane skew.

Following are the steps to generate random skew for each lane:
Figure 8: Flow chart Skew model

7.3 Return Loss model:

Return loss is a measure of the reflected energy from a transmitted signal. These reflections are caused by impedance discontinuities in the channel. These discontinuities may be due to several parameters such as connectors, improper manufacture. Any energy that is reflected, reduces the power of the transmitted signal.

It is commonly expressed in positive dB. The larger the value, the less energy that is reflected.

Return loss can be calculated using the following equation:

\[ RL = 20 \log \left| \frac{Zi - 100\Omega}{Zi + 100\Omega} \right| \]

\( Zi \) = input impedance of the cable at a given frequency.

The value of a RL is a negative number. The standards refer to all measured values in a positive format. For comparison to a specification, this conversion is accomplished by placing a negative sign in front of the equation.

\[ RL = -20 \log \left| \frac{Zi - 100\Omega}{Zi + 100\Omega} \right| \]

7.4 Cable delay model:

Cable delay is the amount of time required for cable to transmit signal to travel from the transmitter to the receiver. It can be computed as the ratio between the link length and the propagation speed over the specific medium.

\[ \text{Cable delay} = \frac{\text{channel length}}{\text{propagation speed}} \]

Channel length: it represents the length of channel or physical medium in meter.

Propagation speed: The propagation speed depends on the physical medium of the link (that is, fibre optics, twisted-pair copper wire, etc.) in meter per second (m/s)

8. Results and Description:

UVM generates randomized bit value for each lane as transaction and sends to SystemC noise channel model. SystemC models computes delay based on timing and noise parameters specified by user and updates total delay for each lane as transaction.

Following timing and noise parameters are considered during the simulation.
1. Rise time in ps
2. Fall time in ps
3. Jitter in ps
4. Jitter accuracy in ppm
5. Maximum lane to lane skew in ns
6. Channel length in meter and propagation speed in m/s for cable delay
7. Input impedance of cable at a given frequency in ohms for return loss

For the specified timing and noise parameter System C computes total delay for each lane to propagate corresponding lane value. Following description shows simulation results for specified value of noise parameters.

Number of lanes = 4
Rise time = 2.0 ps
Fall time = 1.0 ps
Jitter = 2.0ps
Jitter accuracy = 100000 ppm
Skew = 29.0 ns (Maximum lane to lane skew)

To calculate cable delay
Channel length = 10 m
Propagation speed = 200000000 (in m/s depends on channel)

To calculate return loss
Input impedance = 80.0 ohms (for a given frequency)

Initially all lanes are initialized to Zero value. We have generated nine packets of lane values and passed to SC model for delay computation.

SystemC model Calculates lane to lane skew for each lane in picosecond as per the algorithm described in section 7.2 during simulation.

<table>
<thead>
<tr>
<th>Lane</th>
<th>Calculated lane to lane skew in ps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lane 0</td>
<td>26569.7</td>
</tr>
<tr>
<td>Lane 1</td>
<td>1545</td>
</tr>
<tr>
<td>Lane 2</td>
<td>635.056</td>
</tr>
<tr>
<td>Lane 3</td>
<td>250.29</td>
</tr>
</tbody>
</table>

Table 1: Calculated lane to lane skew
System C Model Calculates Cable delay in picoseconds during simulation as per description in section 7.4

Table 2: Calculated transition time & jitter

<table>
<thead>
<tr>
<th>UVM Transmitted value</th>
<th>SC Received lane value</th>
<th>Transition on lanes</th>
<th>Calculated Jitter</th>
</tr>
</thead>
<tbody>
<tr>
<td>L3 L2 L1 L0</td>
<td>L3 L2 L1 L0</td>
<td>L3 L2 L1 L0</td>
<td>L3 L2 L1 L0</td>
</tr>
<tr>
<td>- - - -</td>
<td>0 0 0 0</td>
<td>- - - -</td>
<td>- - - -</td>
</tr>
<tr>
<td>1 1 1 0</td>
<td>1 1 1 0</td>
<td>R R R N</td>
<td>2.09 2.03 2.03</td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>1 0 0 0</td>
<td>N F F N</td>
<td>- 1.84 2.19</td>
</tr>
<tr>
<td>1 1 1 0</td>
<td>1 1 1 0</td>
<td>N R R N</td>
<td>- 1.96 2.14</td>
</tr>
<tr>
<td>0 0 0 1</td>
<td>0 0 0 1</td>
<td>F F F R</td>
<td>1.95 2.08 2.02</td>
</tr>
<tr>
<td>1 0 0 1</td>
<td>1 0 0 1</td>
<td>R N N N</td>
<td>1.98 - - -</td>
</tr>
<tr>
<td>1 1 1 0</td>
<td>1 1 1 0</td>
<td>N R R F</td>
<td>- 2.04 2.05 1.87</td>
</tr>
<tr>
<td>1 0 0 1</td>
<td>1 0 0 1</td>
<td>N F F R</td>
<td>- 1.82 1.93 1.96</td>
</tr>
<tr>
<td>1 1 1 1</td>
<td>1 1 1 1</td>
<td>N R R N</td>
<td>- 1.88 2.10 -</td>
</tr>
<tr>
<td>0 1 1 1</td>
<td>0 1 1 1</td>
<td>F N N N</td>
<td>2.03 - - -</td>
</tr>
</tbody>
</table>

Figure 9: lane skew chart

System C Model Calculates Cable delay in picoseconds during simulation as per description in section 7.4

Cable delay = 50000 ps

System C model calculates Return loss in dB during simulation as per description in section 7.3.

Return loss = 19.0849 dB

SystemC model calculates jitter in ps during simulation as per description in section 7.1. Following table 3 represents simulation result for jitter calculation

Lane 0; L1= lane 1; L2= lane 2; L3= lane 3; R= Rise time;F= Fall Time;N= No Transition;

Following line chart shown in figure 10 and figure 11 represents variation of jitter on lane0 and lane 1 respectively based on the lane value during simulation.
Following line chart shown in figure 12 and figure 13 represents variation of jitter on lane 2 and lane 3 respectively based on the lane value during simulation.

System C model calculates total delay for each lane based on following formula and updates transaction.

Total delay = Rise Time or Fall Time + Jitter + Skew + Cable delay

Following Table represents the total delay for each lane as per above mentioned formula.

<table>
<thead>
<tr>
<th>Lane 3</th>
<th>Lane 2</th>
<th>Lane 1</th>
<th>Lane 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>50254.4</td>
<td>50639.1</td>
<td>51548.8</td>
<td>76569.7</td>
</tr>
<tr>
<td>50250.3</td>
<td>50637.9</td>
<td>51548.2</td>
<td>76569.7</td>
</tr>
<tr>
<td>50250.3</td>
<td>50639</td>
<td>51549.1</td>
<td>76569.7</td>
</tr>
<tr>
<td>50253.2</td>
<td>50638.1</td>
<td>51548</td>
<td>76573.8</td>
</tr>
<tr>
<td>50254.3</td>
<td>50635.1</td>
<td>51545</td>
<td>76569.7</td>
</tr>
<tr>
<td>50250.3</td>
<td>50639.1</td>
<td>51549</td>
<td>76572.5</td>
</tr>
<tr>
<td>50250.3</td>
<td>50637.9</td>
<td>51547.9</td>
<td>76573.6</td>
</tr>
<tr>
<td>50250.3</td>
<td>50638.9</td>
<td>51549.1</td>
<td>76569.7</td>
</tr>
<tr>
<td>50253.3</td>
<td>50635.1</td>
<td>51545</td>
<td>76569.7</td>
</tr>
</tbody>
</table>

Table 3: Calculated total delay
9. Conclusion

Transaction level interaction between SystemC and UVM (SystemVerilog) is achieved using UVM Connect library. This interaction is verified by implementing UVM generator model to generate Ethernet L2 packet as transaction and System C printer model to receive same transaction. Ability to perform cross language communication leverages language strengths and expands Verification IP inventory. Also allow SystemC architectural models to be used in UVM verification environment. Generalized noise channel model has been implemented at transaction level using System C and verified for noise parameters like jitter, lane to lane skew, transition time, cable delay which will be useful for performance analysis of communication protocol at higher abstraction level.

10. References