Interoperability Validation Without Direct Integration

N. Nuti, S. Jambulingam
Intel Corporation
1900 Prairie City Rd
Folsom, CA 95630

***Abstract*-Simulation of multiple HDL IPs (Hardware Description Language Intellectual Property) can be a cumbersome endeavor that may delay SoC (System on Chip) validation. The proposed verification method uses an IPC (Inter-Process Communication) framework and DPI (Direct Programming Interface) to indirectly integrate HDL IPs into a simulation environment to validate interoperability. Without waiting for integration to happen, we can validate interoperability ahead of time to reduce surprises at the SoC level.**

I. Introduction

Contemporary methods of validating SoC IP interoperability tend to be arduous and time-consuming because they lack a standardized functional validation methodology. An SoC is made of many interconnected IPs that can be developed and verified in isolated conditions. Lack of initial collaborative effort between IP teams can lead to increased validation complexity because IPs have varying simulation environment requirements and preexisting work must be modified to directly integrate other IPs. Direct IP integration in interoperability validation will require extended support from other teams during setup because simulation environments must be merged. Combining IP modules and testbenches can be difficult because projects may have incompatible prerequisites. Further, a lack of a standardized and seamless integration method increases project duration and can cause validation teams to find bugs late in the overall development process.

This paper discusses an unobtrusive and accessible method of multi-IP simulation with the aim of optimizing and simplifying the process of interoperability validation.

II. Background

Interoperability validation commonly has issues that can disturb or delay results. Some obstacles that can obstruct interoperability validation include convoluted SoC and IP simulation setups, differences between simulation setups, and the challenge of directly integrating multiple external IPs into a single simulation environment. Dealing with these issues while maintaining the validity of integration testing can be a formidable assignment. We must change the multi-system validation status quo. This section outlines the project motivation, problems associated with current industry practices, and research and preexisting work done to develop the indirect integration method.

Motivation

Motivation for the development of an indirect integration validation platform is driven by the need to decrease interoperability validation complexity, normalize a common validation methodology, and optimize turnaround time of vital testing results. Through normalization of interoperability validation practices, users can consistently use the same validation platform for different projects and link simulations of external IPs seamlessly all while requiring minimal amounts of changes to preexisting IP simulation environments. Creating a standard and separated IP interoperability validation framework greatly reduces complexity because users do not need to conform to software or simulation setup requirements that are otherwise essential when users directly merge IP modules and testbenches. Our validation method focuses on the preceding information to simplify interoperability validation as a whole and reduce the time necessary to produce IP conformance results. Traditional IP interoperability validation methods are error and issue prone; our validation method ventures to solve these issues with structure and minimization of changes in simulation environments.

Industry Practices

Typical industry practices for IP interoperability compliance involve validation teams being tasked with directly merging IP modules and testbenches into one project. Engineers working on interoperability validation receive IP simulation projects from other teams (possibly internal or external to the company) and incorporate these external projects into their own simulation environment. Assimilation of external simulation setups into preexisting validation environments can lead to setbacks during simulation setup, debug, and maintenance.

1. Interoperability Validation Setup

Validation of multiple IPs communicating with each other involves consolidation of those IPs into a single simulation environment. Validation engineers adopt the external IP simulations into their own simulation domain testbench. Instantiation of external IP modules into non-native simulation environments requires considerable brute-force and can result in a plethora of issues. Forcing two simulations together can cause messy setups that lack common procedures and have overcomplications due to unnecessarily rigid requirements.

1.A Lack of a Common Process

Validation setups that are absent of a common build process risk having complications from obscure, unexpected, or unknown requirements. Engineers do not commonly have adaptable multi-IP simulation systems that can run interoperability validation and quickly produce results. Therefore, the cost of multi-IP simulation setup can vary dramatically between projects and this lack of consistency results in sporadic approaches that increase project duration.

1.B Overcomplicated Simulation Environments

Validation testbench complexity elevates when disparate IP modules and testbenches are merged together. Simulation environments can contain requirements that involve specific simulation software, simulation software revisions, language compilers, include-files, libraries, etc. When integrating an external IP simulation environment into another simulation environment, the environment accepting the external IP environment must adopt its requirements for proper operation. Assimilating those requisites into the environment of operation may cause conflicts or other issues that must be overcome by engineers. To combat these issues, engineers must overcomplicate their simulation setups to accommodate requirement differences between the original simulation environment and the merged external IP simulation environment. Excessively complex interoperability validation simulation environments lead to convoluted systems and delays in IP interoperability compliance results.

2. Debug of Multi-IP Simulations

Multi-IP interoperability simulation environments that utilize direct integration run the risk of requiring extended setup debugging and having complicated debug sessions with external teams. Complexity of a simulation platform containing two separate IPs increases as requirements vary. These simulation setups become difficult to debug for engineers inheriting new IPs to validate and for engineers that own the IP that is being inherited, because requirements of both IPs need to be juggled and the validation environment needs to be habituated for multiple IPs. In addition, future debug sessions become difficult as engineers from the team inheriting an IP and the engineers from the team that developed the IP both have to look at unfamiliar code. When integrating external IPs into other simulation environments, teams might need to continually debug more than just IP functionality.

3. Simulation Environment Maintenance

Requirements for interoperability validation may change for every new iteration of IP, therefore multi-IP interoperability simulation testbenches might have significant changes for every new IP iteration. If IPs are continuously changing in a multi-IP consolidated simulation environment, then the interoperability validation platform becomes difficult to maintain. Significant IP modifications may result in changes in simulation software, libraries, etc., which may spawn new conflicts with requirements established by another IP in a multi-IP simulation environment.

Research and Prior Work

Research for the indirect interoperability validation project revolved around tactics for enhancing manageability of multi-IP simulation environments, specifically concerning the previously mentioned issues with practices in the validation industry. We explored topics such as: DPI, C++ shared memory, and SystemVerilog simulation tactics.

We began with a documentation review to enhance and broaden our knowledge on various tools and techniques. We referenced SystemVerilog simulation tactics and DPI information and syntax from the *SystemVerilog Language Reference Manual (LRM)* [1]. Using the SystemVerilog LRM, we found that “An imported task has the same semantics as a native SystemVerilog task: it never returns a value, and it can consume simulation time”

(SystemVerilog Section 35.2.1). Using this idea and a single-threaded simulation enforced by usage of SystemVerilog tasks, we found that we could mimic synchronous behavior between two separate simulation instances. This is further discussed in the methodology section. The other research topic scrutinized was C++ shared memory. Before knowing about shared memory, we needed to figure out how to share data between applications. We already had the IP port-level connections available for multiple IPs in C++ via DPI, so we needed a way to transfer the port-level data between the C++ programs attached to the IP simulation environments. We searched around for possible libraries to use and came across the notion of shared memory. An in-depth description of shared memory and the shared memory API can be found in the *POSIX Standard* [2]. With shared memory and memory mapping, from the memory management API [2], we found that we could allocate and organize memory resources and map those memory resources to the input/output ports of our C++ applications interfaced with our simulation testbenches.

Further groundwork for the research involved in this project was established by prior efforts on our IP simulations. The mentioned simulations involved DPI and shared memory to export simulation data to companion C++ programs. Relevant simulation environment work completed prior acted as the catalyst for starting the project, but significant changes in structure and data handling were necessary to attain the current design model.

II. Methodology

In the proposed validation method, we implemented a straightforward approach to simplify and standardize the setup and runtime of simulations used to verify the behaviors between HDL IPs. Our technique involves the idea of indirectly implementing IPs in simulation environments using IPC and DPI ports[1].

Inter-Process Communication

An IPC interface was employed to link two entirely separate yet active simulations. IPC is a technique that grants process synchronization via a bus like shared memory. More specifically, IPC can synchronously link multiple independent simulation environments indirectly through common memory locations stored in kernel space. To apply IPC to our simulation efforts, we had to associate important IP ports with DPI ports, import functions used for transferring packetized data between simulations and memory, and add arguments to our simulation environment compilation script. Therefore, using IPC to verify behavior between IPs simplifies validation efforts because simulation setups require minimal changes to use IPC and are run independently so setup and runtime requirements become less stringent than what is required when merging IP simulations.

SystemVerilog Direct Programming Interface

DPI ports were created in our system so that SystemVerilog testbenches could interact with the C++ IPC system. Using DPI granted us the ability to standardize how our separate simulation environments send and receive task data and minimize the testbench modifications necessary to simulate IP communications. Verifying behavior between IPs using our method requires that independent IP simulation environments linked through IPC agree on port direction and match the number and order of DPI ports between the IPs. Application of DPI in our simulation ecosystem has turned the setup process into a simple addition of a block of code so that interoperability can be tested quickly before SoC-level direct integration.

Indirect IP Integration in Simulation

Our idea of indirect IP integration stems from the idea of having two separate HDL simulations access shared kernel space memory. When simulating transmissions between two IPs indirectly, one simulation environment is responsible for writing to memory at address A and reading from memory at address B while the other simulation environment is responsible for reading from memory at address A and writing to memory at address B. Data leaves and enters testbenches through DPI ports and is written to and read from system memory; so, the active simulations only require each other to be present at runtime. Using our method, IP interoperability can be quickly tested with minimal effort. Now, working IP simulation environments can be slightly modified and independently compiled and run to validate interoperability before merging simulation testbenches directly. Even having IP simulation environments that operate with different software is not a problem, the software will handle the DPI port interactions the same.

Data Handling

Data is handled in a producer and consumer format with individual data lines for each data transmission direction (initiator sends to receiver and receiver sends to initiator). The producer is the IP that is sending a packet,

[1] DPI ports are module ports connected to DPI tasks and functions.

and the consumer is the IP that is receiving a packet. During simulations, one IP must be the producer and the other IP must be the consumer at a given point in time. After the consumer has received a packet, the producer switches to the consumer and the consumer switches to the producer. Both IPs must have context switching to send and receive packets every clock cycle of the simulation. Our implementation uses this method of data handling to produce simulated synchronous communication between IPs (Figure 1).

Our methodology gave us the ability to validate interoperability between IPs seamlessly. We were able to link separate IP simulations in parallel and have these simulations synchronously exchange port data.

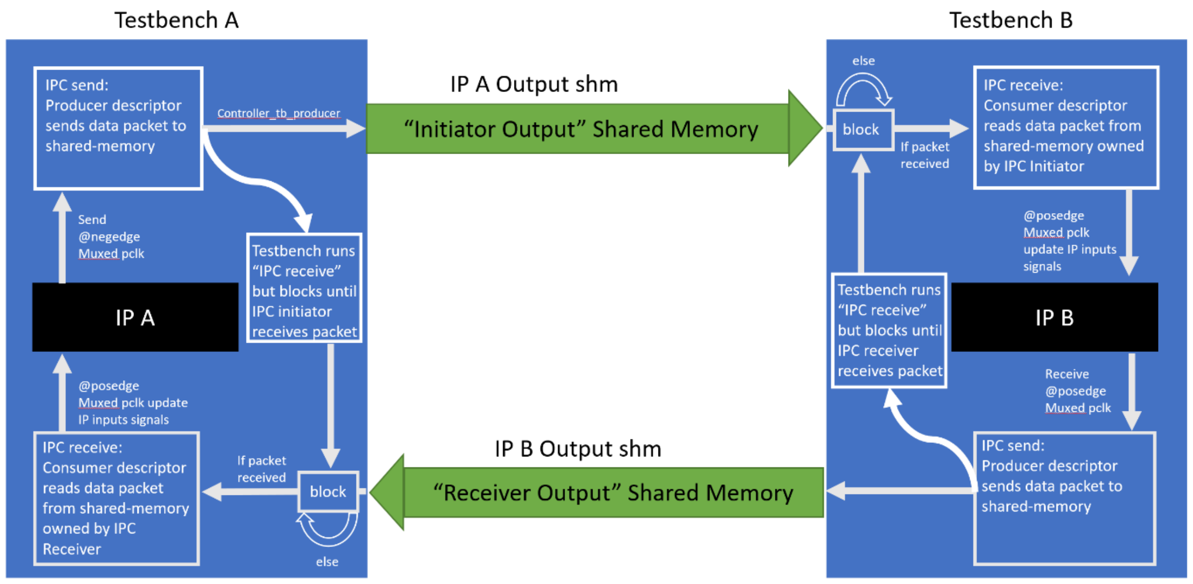
Testbench A
Testbench B
IPC send: Producer descriptor sends data packet to shared-memory
Controller_tb_producer
IP A Output shm
"Initiator Output" Shared Memory
else
block
If packet received
IPC receive: Consumer descriptor reads data packet from shared-memory owned by IPC Initiator
Send @negedge Muxed pclk
Testbench runs "IPC receive" but blocks until IPC initiator receives packet
IP A
@posedge Muxed pclk update IP inputs signals
Testbench runs "IPC receive" but blocks until IPC receiver receives packet
IP B
@posedge Muxed pclk update IP inputs signals
@posedge Muxed pclk update IP inputs signals
Receive @posedge Muxed pclk
IPC receive: Consumer descriptor reads data packet from shared-memory owned by IPC Receiver
If packet received
block
else
IP B Output shm
"Receiver Output" Shared Memory
IPC send: Producer descriptor sends data packet to shared-memory

Figure 1: Parallel Simulations with IPC Flow

Simulation Setup Process

Implementation of indirect integration validation via IPC requires some upfront effort due to its distinct nature. The following outlines an abbreviated set of steps used to incorporate IPC into RTL simulation environments. Please note that, due to space constraints, it is not possible to provide code examples beyond what is presented.

1. Address Testbench Requirements

The goal of indirect integration is to have separate simulation environments communicate with each other at the port-level. To achieve this kind of communication between testbenches, relevant IP ports need to be able to write to and read from an integrated IPC system via DPI. Engineers desiring to validate IPs via indirect integration must consider the following:

- Understand which IP ports need to be connected to DPI and what logic is driven by external IPs.
- Code that is driving signals that should otherwise be driven by another IP must be removed.
- If a clock is supposed to be sent from one IP to another, then another method of clock transfer must be devised. For example, validators can send the value of the clock period via DPI to the other testbench; from there the other testbench generates the clock based on the received clock period.
- Asynchronous signals transferred between IPs must be clocked synchronously through the DPI ports; our method does not allow for asynchronous transfer.
- Simulation data must be sent out through DPI ports on the edge of the clock that is opposite to that with which it is normally associated.
- The current IPC interface is only capable of operating in a single clock domain.
- IP simulation environments require the following DPI tasks imported: a C++ function to setup shared memory interfaces, a C++ function to send data to shared memory, a C++ function to read data from shared memory, and a C++ function for relinquishing the shared memory.

- IP simulation environments require the following DPI tasks exported: a SystemVerilog task for transferring buffered data from shared memory into an active simulation, and a SystemVerilog task for running “$finish” to end the RTL simulation.
- Commands to build the simulation environment need to include C++ files.

2. Address IPC Code Requirements

C++ code requirements of the indirect validation method are simple in idea but can be difficult to implement. The following must be adhered to in order to produce a successful interaction between IP simulation environments using IPC:

- IP simulation environments require their own respective C++ files.
- The following DPI tasks should be “extern” instantiated in the C++ code: SystemVerilog task for reading DPI port data into an active IP simulation, and a SystemVerilog task for running “$finish” to end the RTL simulation.
- The shared memory interfaces must be constructed at the beginning of each IP simulation and destructed at the end of each IP simulation.
- Shared memory must have a sender pipe and receiver pipe for each IP simulation environment.
- One IP must be sending while the other is receiving; this produces the pseudo synchronicity.

3. Create SystemVerilog DPI Send and Receive Tasks

Simulation ecosystems that use IPC will require at least three DPI tasks: a task to send data to shared memory, a task to read and buffer data from shared memory, and a task to pipe the buffered read data into the active RTL simulation. RTL simulation environments are capable of telling the IPC system to forward DPI port information to shared memory for another testbench to digest; the DPI task signature for sending data from the RTL simulation to the IPC shared memory is shown in Figure 2.

```
// Instruct IPC to read DPI ports and send data to shared memory
task send_to_shm({signal_out_tb_1, signal_out_tb_0})
```

Figure 2: DPI Task for Sending Simulation Data to Shared Memory via IPC

Having a testbench read data from another testbench is quite different from writing. The testbench initiating the read must instruct the IPC system to read shared memory (Figure 3) and ultimately send the read data to the DPI ports of the testbench that initiated the read (Figure 4).

```
// Instruct IPC to read shared memory and buffer the read data
void read_from_shm();
```

Figure 3: C++ Function Imported to SystemVerilog Simulation via DPI to Read Shared Memory

```
// Pipe the buffered read data into the RTL simulation
task receive_in_tb(bit [1:0] data);
      signal_in_tb_0 = data[0];
      signal_in_tb_1 = data[1];
endtask
```

Figure 4: DPI Task Used as a C++ Callback for Forwarding Buffered Read Data into Simulation DPI Ports

4. Add DPI Task Imports and Exports to Testbenches

DPI will reference external C++ functions that will send data to and receive data from other RTL simulation environments. These RTL testbenches will need to import these C++ functions in the following manner shown in Figure 5. Note that C++ functions called by SystemVerilog are imported and SystemVerilog tasks and functions called by C++ are exported. Note also that IP testbenches should have their own unique C++ functions.

```
import “DPI-C” context task send_to_shm (bit [0:1] data);
import “DPI-C” context task read_from_shm ();
export “DPI-C” task receive_in_tb ();
```

Figure 5: DPI Task Imports and Exports

5. Create C++ Send and Receive Functions

C++ code that is used in indirect integration needs the ability to write to and read from shared memory interfaces while also being able to interface with DPI ports. We can use the DPI tasks already created in “3. Create SystemVerilog DPI Send and Receive Tasks” because imported DPI tasks must match the names of preexisting C++ functions. Therefore, we need a function to send data from DPI ports to shared memory (Figure 6) and a function to read data from shared memory into an IP simulation (Figure 7). For further information, please view Figure 1 and reference section “35. Direct Programming Interface” of the *SystemVerilog Language Reference Manual (LRM)* [1].

```
// Send RTL Sim DPI Data to Shared Memory
extern “C” void send_to_shm(svBitVecVal *data) {
       while(!shm_buff_ready); // Wait for existing packet to be read
       shm_buff = data;
       packetvalid = true;
}
```

Figure 6: C++ Pseudo Code for Sending DPI Port Data to Shared Memory

```
// Receive Shared Memory Data and send to RTL Sim DPI Ports
extern “C” void read_from_shm () {
       While(!packetvalid); // Wait for new packet to be sent
       readdata = shm_buff;
       receive_in_tb (readdata);
       shm_buff_ready = true;
       packetvalid = false;
}
```

Figure 7: C++ Pseudo Code for Reading Shared Memory into DPI Ports

6. Execution

The full simulation runtime consists of three major modes: setup, active IP communication, and simulation end. Flow diagrams of each simulation mode operation are shown in Figure 8, Figure 9, and Figure 10. The flow diagrams detail the behavior of a simulation environment containing two IPs integrated indirectly and an IPC system with shared memory lines. Also, the flow diagrams detail our project and how the contents of this paper helped us to complete it. Please note that the boxes in Figure 8 that contain “send packet” and “receive packet” are explained in further detail through Figure 1, and the functions established below are from previous subsections within the “Simulation Setup Process” section. This subsection discusses the inner workings of our project execution to serve as a thought-provoking example.

The setup portion of our simulation ecosystem requires that the controller IP simulation is started before the external IP simulation. The controller IP simulation generates two shared memory lines and the external IP latches itself to the shared memory lines. One shared memory line is for the controller to write to and the external IP to read from and the other line is for the opposite. After both active IP simulations are connected to the shared memory lines, the controller sends a packet to be digested by the external IP simulation and waits for an acknowledgement. The external IP simulation reads the packet from shared memory, sends an acknowledgement back, and waits to read another packet from the controller IP simulation. Now that the IPC interface is tested, the IPs can perform their typical operations and send/receive packets every clock cycle.

The IPC active section of the simulation process is similar to how usual simulation flows start: clocks start switching and data transfers between IPs. Specifically, the controller will use the “send_to_shm()” SystemVerilog DPI task to send signal data to the linked C++ function where the data will be stored in shared memory. The controller testbench will change context and run the “read_from_shm()” SystemVerilog DPI task which calls the imported C++ function. The C++ function “read_from_shm()” accesses data from shared memory and uses “receive_in_tb()” SystemVerilog DPI task to transfer data from C++ to the RTL simulation environment. While running “read_from_shm()”, the controller IP testbench will stop all operations until it reads a packet from the shared memory that the external IP writes to. Switching focus to the external IP, when the controller IP writes to its shared memory the external IP testbench will already be waiting for a packet because it ran “read_from_shm()”. After receiving a packet, the external IP testbench will turn around and write to its shared memory using “send_to_shm()” and then immediately run “read_from_shm()” and halt. The controller testbench will then receive the packet the external IP wrote to shared memory, and the controller testbench will continue operating until it runs “read_from_shm()” again. This process of IPs alternating reads and writes happens for the majority of the simulation runtime until the ending phase starts.

The ending phase is triggered by the controller IP sending a packet for signaling the end of the simulation to shared memory. Next the controller IP simulation waits for an acknowledgement from the external IP simulation. When the external IP simulation initiates its next shared memory read, the external IP reads the termination packet and sends an acknowledgement back to the controller IP. The external IP simulation runs “$finish” and the controller IP simulation gathers the termination packet acknowledgement. Lastly, the controller IP simulation destructs the shared memory interfaces and runs “$finish” to end the rest of the running processes.

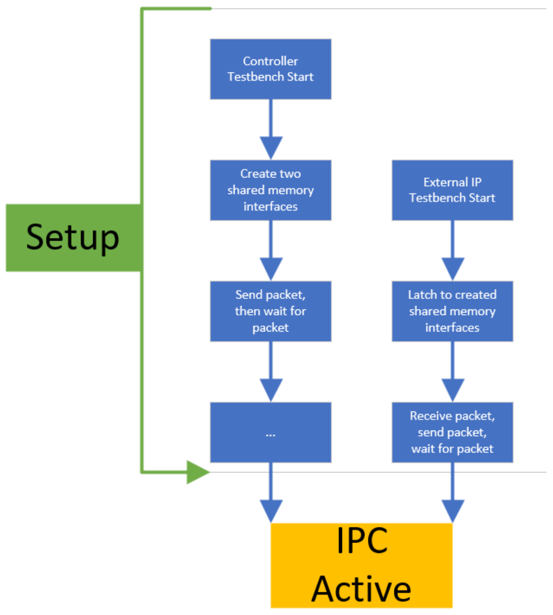
Setup
Controller Testbench Start
Create two shared memory interfaces
Send packet, then wait for packet
...
External IP Testbench Start
Latch to created shared memory interfaces
Receive packet, send packet, wait for packet
IPC Active

Figure 8. Indirect Validation System—Setup Mode

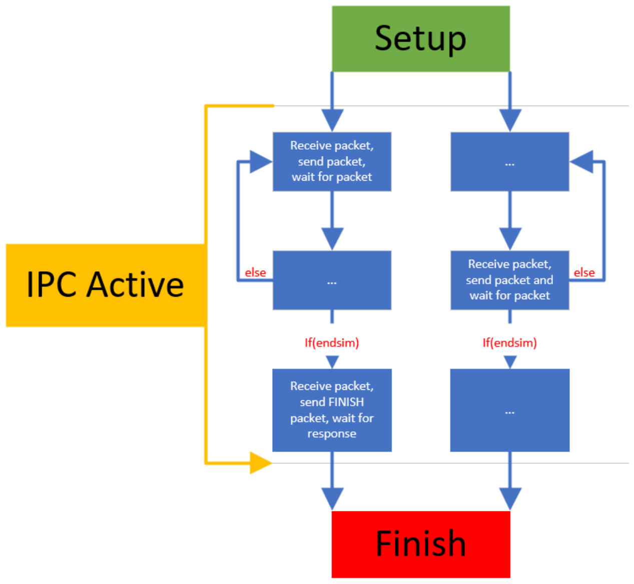
Setup
IPC Active
Receive packet, send packet, wait for packet
...
else
...
Receive packet, send packet and wait for packet
else
If(endsim)
If(endsim)
Receive packet, send FINISH packet, wait for response
...
Finish

Figure 9. Indirect Validation System—IPC Active Mode

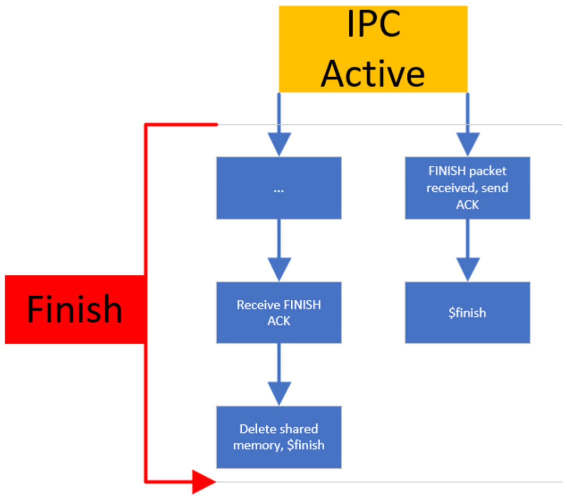
IPC Active
Finish
...
FINISH packet received, send ACK
Receive FINISH ACK
$finish
Delete shared memory, $finish

Figure 10. Indirect Validation System—Finish Mode

III. Results

The validation results involve terse details about the project that used indirect IP integration and how we achieved our goals via this method. Our project involved two simulation environments, our IP environment and an external IP environment. The main task was to have our IP communicate with the external IP by sending setup commands, sending data, and reading statuses. Initially, we merged our IP simulation testbench with the external IP testbench; this method proved to be difficult to implement and maintain. To further optimize our interoperability validation between our IP and the external IP, we implemented the indirect IP integration into our validation environment. Application of the indirect integration method took us less than a week to implement for our interoperability validation ecosystem. In short, indirect IP integration aided us in setting up our IP and the external IP simulation environments quickly and consequently led us to find bugs and oddities. Indirect IP integration helped us separate independent IP changes in simulation environments, loosen compilation requirements, and supply the external IP team with our simulation environment with almost no setup required. The standardization of ports and implementation of IPC has even made it easy for us to interact with other IP and for other teams to setup and simulate with our IP. Overall, IPC and DPI have proven to be effective tools and integration of them into our IP validation environment has made simulation environment configuration and execution painless.

IV. Challenges and Limitations

While this paper discusses the major benefits involved with using indirect IP validation, there are some drawbacks that can be solved through future developments but exist in the current state of the method. This section details some of the challenges and limitations associated with the indirect integration validation method.

1. Asynchronous Signals:

Signals that are transmitted between IPs that are asynchronous must be clocked and therefore made synchronous when using IPC. Asynchronous signals do not inherently work in the current implementation because the SystemVerilog tasks used for interacting with shared memory halt the simulation. Halting the simulation is how we achieve pseudo synchronicity because simulations are not allowed to progress until buffered data is read from shared memory (on the negative edge of the associated clock). Signals that do not adhere to a clock can be implemented but the complexity rises because the simulation or the C++ programs will need to become multithreaded and involve a separate shared memory space just for asynchronous signals.

2. More Than Two IPs Communicating:

Simulation environments that require more than two IPs constantly communicating with each other have not been tested with the indirect integration validation method. There are multiple ways to create an indirect integration simulation method for validating the interoperability of more than two IPs. An example would be to have a "round robin" chain of shared memory where communications are in a constant sequential chain. In summation, simulating with more than two IPs indirectly integrated was out of scope for our implementation and therefore has not been verified.

3. Multiple Clock Domains:

Sometimes IPs need to send signals between each other that are associated with differing clock domains. The concept of having signals of multiple clock domains transmitting across the IPC system has not been tested but is entirely possible. For example, if a user had signals in two clock domains and the clock domains were multiples of each other, then the IPC could operate in the faster clock domain. Even though signals of the slower clock domain would be sent to shared memory at the rate of the faster clock domain, the receiving testbench should still clock the signals in using the slower clock domain because changes to the signal sent to shared memory will occur at the speed of the slower clock domain. A further limitation would be clock domains that aren't multiples of each other; this would require two shared memory interfaces and parallelism that is out of scope for this paper.

V. Conclusion

Our method of interoperability validation and an application of it have proven to be successful. The results from employing the method to our IP and an external IP showed that indirect integration is simple to use and can reduce time and effort required when setting up and running multi-IP simulation environments. Indirect integration of IPs in validation is more efficient than merging IP modules and testbenches and has the ability to refine IP interoperability validation before direct SoC interoperability validation by making simulation setups easier to prepare and use.

In the future, we plan on implementing asynchronous signals across the IPC domain and understanding the simulation statistics of validation with direct versus indirect IP integration. Altogether, this paper examines an efficient method of interoperability validation that can expedite validation of multi-IP simulation environments.

ACKNOWLEDGEMENTS

We would like to thank Vijayakrishnan Rousseau for his technical guidance and valuable knowledge in the fields of validation and verification.

References

[1] "IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language," in *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)* , vol., no., pp.1-1315, 22 Feb. 2018, doi: 10.1109/IEEESTD.2018.8299595.

[2] The Open Group, "POSIX - Base Specifications, Issue 7," IEEE Std 1003.1-2017, The Open Group, 2018. [Online]. Available: https://pubs.opengroup.org/onlinepubs/9699919799/