Bus Trace System: Automating Bus Traffic Debugging in IP-XACT Based SoC Beyond Traditional Debugging Methods

Wonyeong So[1], Yonghyun Yang[2], Sun-il Roe[3], Moonki Jang[4], Youngsik Kim[5], Seonil Brian Choi[6]
Samsung Electronics Co. Ltd., Seoul, Korea
([1]wonyeong.so; [2]yh1597.yang; [3]sunil.roe; [4]moonki.jang; [5]ys31.kim; [6]seonilb.choi@samsung.com)

***Abstract*-The Bus Trace System (BTS) is a novel debugging approach for System-on-Chip (SoC) verifications based on IP-XACT. BTS aims to automatically identify and debug bus hang errors and error responses using only simulation logs, without the need for waveform information. BTS works by utilizing SoC bus navigation information that is generated through an in-house bus connection searching algorithm, which searches IP-XACT to find the full bus connection for each instance. By analyzing the interface logs connected to Design Under Test (DUT), BTS can identify the scope that needs to be debugged. By using BTS, the debug time and verification Turn-Around-Time (TAT) can be reduced, as it automatically finds the debug scope related to bus hang and error response in SoC verifications so that it can help to decrease the verification time spend during overall SoC development. BTS is a powerful debugging methodology that can effectively reduce the debugging time in the trend of increasingly complex SoC architectures.**

I. Introduction

Over the past few decades, transistor densities per single chip have increased exponentially due to continuous advancements in process technology. While in the past only 1-5 million transistors were integrated, this number has increased to over 500 million, and more recently, over 20 billion transistors have been integrated [1] [2]. In addition, with the emergence of new trends such as coherent design in multi-core systems, 3D integration and multi-chip integration, SoC architecture is becoming increasingly complex. This means that verification engineers are spending more time debugging during the SoC development. Statistically, the mean peak number of verification engineers in SoC projects has continued to increase up to 11.6 in 2016 from 4.8 in 2007. At the same time, only 47% of time per SoC project was spent on verification in 2014, but this increased up to 54% in 2018 and this trend continues [2]. Debugging time is that increasing and becoming a critical factor in development of SoC.

In the SoC architecture, communication among Intellectual Properties (IPs) takes place through the internal bus. Due to advancements in process technology, both the number of IP interfaces and the count of internal bus node instances have increased, resulting in a heightened complexity of the SoC structure. This has particularly resulted in more debugging time, especially in error cases such as bus hang and error responses. Because this is due to the fact that the conventional debugging approach involves tracing all relevant buses to identify the initial point of error occurrence. If the verification engineers have a limited understanding of the overall SoC architecture, the time required to debug these errors can become even longer. For example, in a SoC, when a master accesses a slave through 10 bus nodes and a bus hang occurs where the master does not receive a response, verification engineers must check all bus nodes to identify which one is causing the problem. In this case, if it takes T time to check one bus node, then in the worst case it takes $10T$ to check all bus nodes. Particularly, in the process of checking all bus nodes, those that operate as expected do not require verification. If problematic bus nodes are identified in advance, unnecessary debugging time could be eliminated. However, the challenge lies in quickly identifying which bus node has a problem at a glance. Even engineers with extensive verification experience find it difficult to identify the problem node at first glance. Identifying problematic bus nodes promptly can significantly reduce debugging time for errors such as bus hang and error response.

Bus Trace System (BTS) aims to effectively reduce debugging time by automatically identifying problematic bus nodes and reporting them to the engineers. BTS generates bus connection information, known as SoC bus navigation, from the master instance to the slave instance using an in-house connection searching algorithm to automatically identify the problematic bus node. Once the navigation information is created, BTS periodically analyzes logs during simulation to determine the precise scope of debugging required in case of the error occurrence. When BTS detects abnormal behavior such as bus hang and error response in the log, it identifies the debug scope and reports it to the engineers.

Our proposed debug methodology utilizes BTS to automatically identify the debug scope for bus hang and error response, thus providing a fast and efficient debugging process to verification engineers. Figure. 1 illustrates the overall operation of BTS, which is divided into Phase 1 and Phase 2. During Phase 1, SoC bus navigation is created using an in-house bus searching algorithm from IP-XACT based SoC integration information. Phase 2 involves

connecting system Verilog interfaces to the DUT to obtain bus traffic logs during simulation. The simulation log is then periodically monitored, and debugging is performed when an abnormality is detected. Ultimately, the problematic bus node is reported.

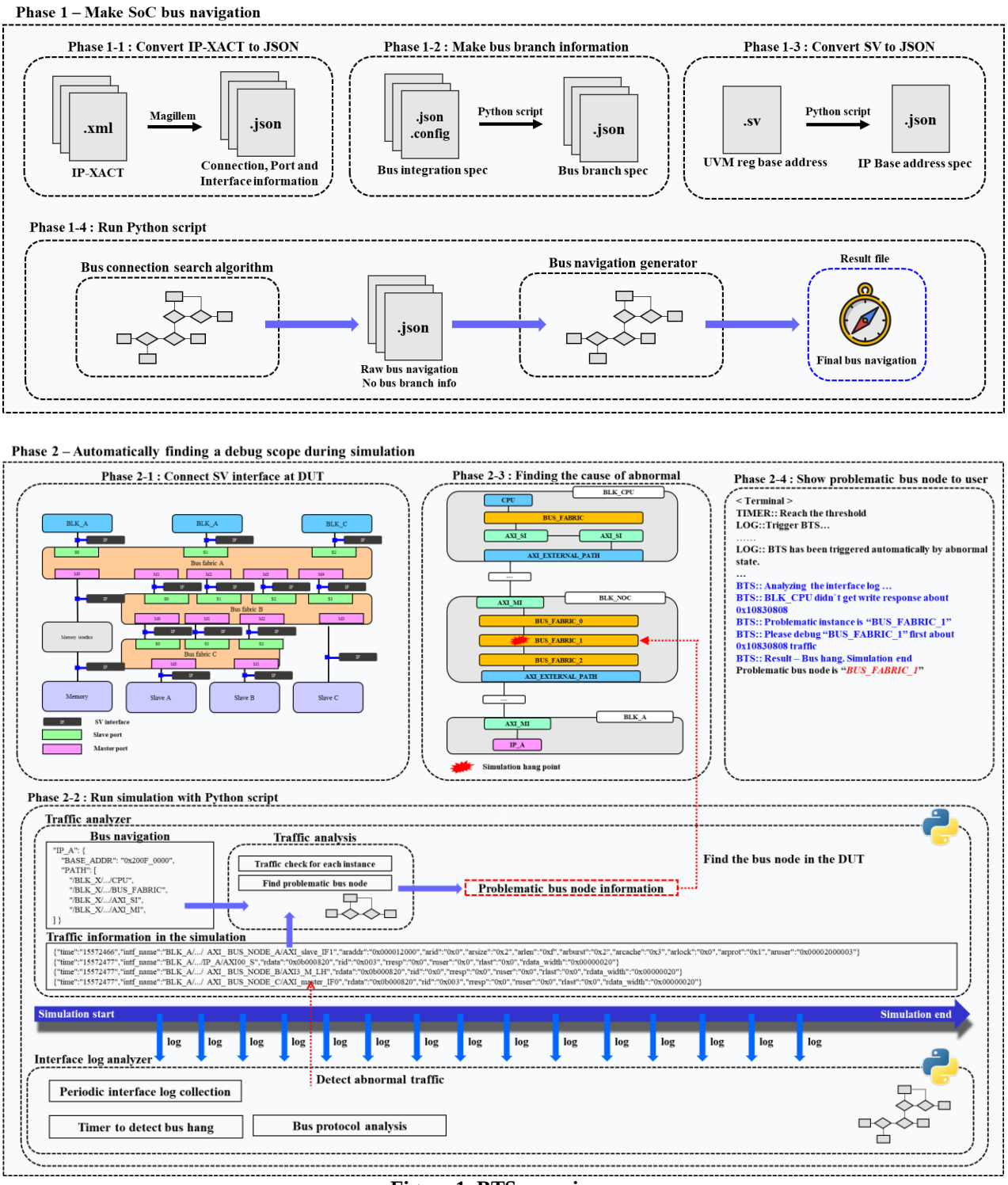


Figure 1. BTS overview

II. BTS PROCESSING

Phase1 – Generate SoC bus navigation information

To automatically obtain bus connection information from master to slave, we create the SoC bus architecture in a format that can be handled by a Python script. First, we convert IP-XACT based SoC integration information to a JSON file via the Magillem tool's export utility [3]. The resulting JSON file contains information about bus connections, ports, and interfaces. To determine the instance connected to each interface of the bus routing logic, which has many interfaces, we used the SoC bus integration specification file as input to execute a Python script, resulting in the creation of a JSON formatted bus routing specification file. Additionally, a python script read our SoC-level UVM register base address SystemVerilog file to generate a spearate base address JSON file for each IP. Once these steps were completed, we ran a Python script that takes the three files as input. The script includes a *bus connection searching algorithm* and *bus navigation generator* to create SoC bus navigation as JSON format. Additionally, in this phase BTS generates a set of key interface information for each bus protocol, including AXI, ACE, CHI, and custom protocols. This key set of interface information is used when BTS detects connectivity information between interfaces.

A. Bus connection searching algorithm

To begin the bus connection search, the starting instance must be determined. In this paper, we chose the host CPU instance as the starting instance and used the UVM register modeling SystemVerilog specification file to generate a list of final destination instances. This algorithm references the instance interface, iteratively searching for instances adjacent to the interfaces of each instance from the starting instance to the target instance. Once the starting instance is determined, the bus protocol type of the instance is checked first. To search all possible bus connections, the bus routing information is not utilized in this process. Instead, bus route information is used when conducting actual searches for valid bus connections. The algorithm then retrieves the interface information from the JSON file extracted during Phase 1-1, refer to Figure. 1, using each node's instance name. It identifies the instance's key interface and bus protocol type. If the key interface is found, the algorithm uses it to identify its counter instance, the target instance connected to the key instance.

An example of this searching algorithm is illustrated in Figure 2. For AXI slave instances, we defined the MISC_M interface as the key interface for the AXI protocol. Therefore, the interface connected to it is identified as the *counter interface*. The instance that has the counter interface is, thus, determined as the *counter instance* in our algorithm. The counter instance then becomes the starting instances and the algorithm recurses. The process ends when the current counter instance exists in the list of final destination instances. If the counter instance is in the list, the search is terminated, and a raw bus navigation file is created. This file is called "raw" because the SoC bus routing specification was not considered in this step. As such, the raw bus navigation file may contain incorrect information that must be rectified before use in simulation.

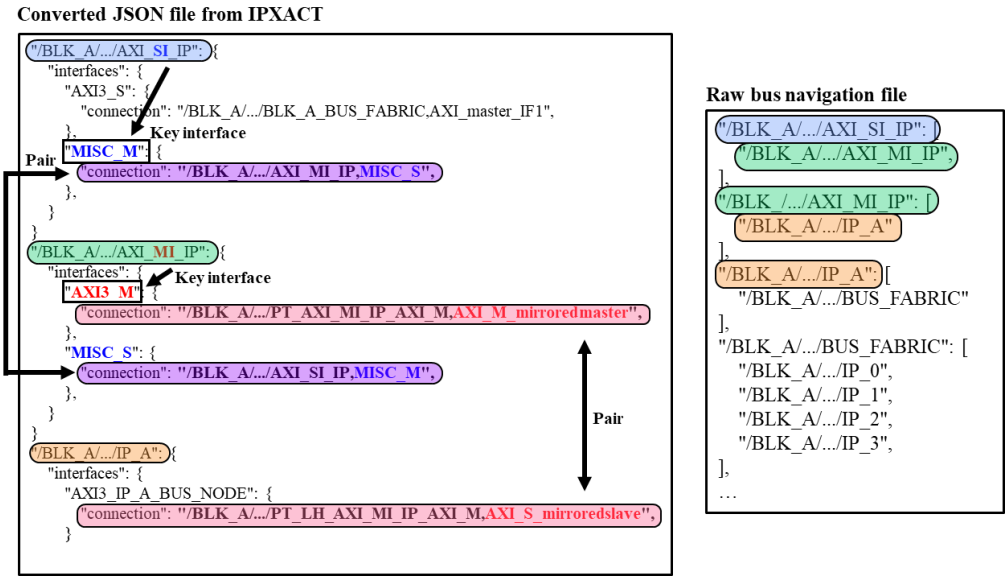


Figure 2. The process of creating a raw bus navigation file.

B. Bus navigation generator

To extract only valid information from the raw navigation file, the bus routing specification file must be utilized. The bus routing specification file contains the bus routing specification for each interface of an instance. However, even if the Python script knows the bus routing information for each interface, it still needs to know which base address to route to. Therefore, in this step, the script retrieves the connection information from the raw navigation file and initiates a new search that includes bus routing information for each final destination instance's base address.

By following this process, the exact counter instance can be identified. Similar to the process of creating the raw navigation file, the counter instance becomes the starting instance again, and the process is repeated recursively. If the counter instance is in the final instances list, the re-search is terminated, and a final bus navigation file is created.

Figure. 3 illustrates an example of determining the counter instance using routing information from an instance that supports bus routing features. In this example, the Python script first specifies the target IP base address. Next, it initiates a new search based on the raw bus navigation file from the starting instance. If the current instance is a bus routing instance, the script checks the routing address range for each interface of the instance and determines the counter interface and instance.

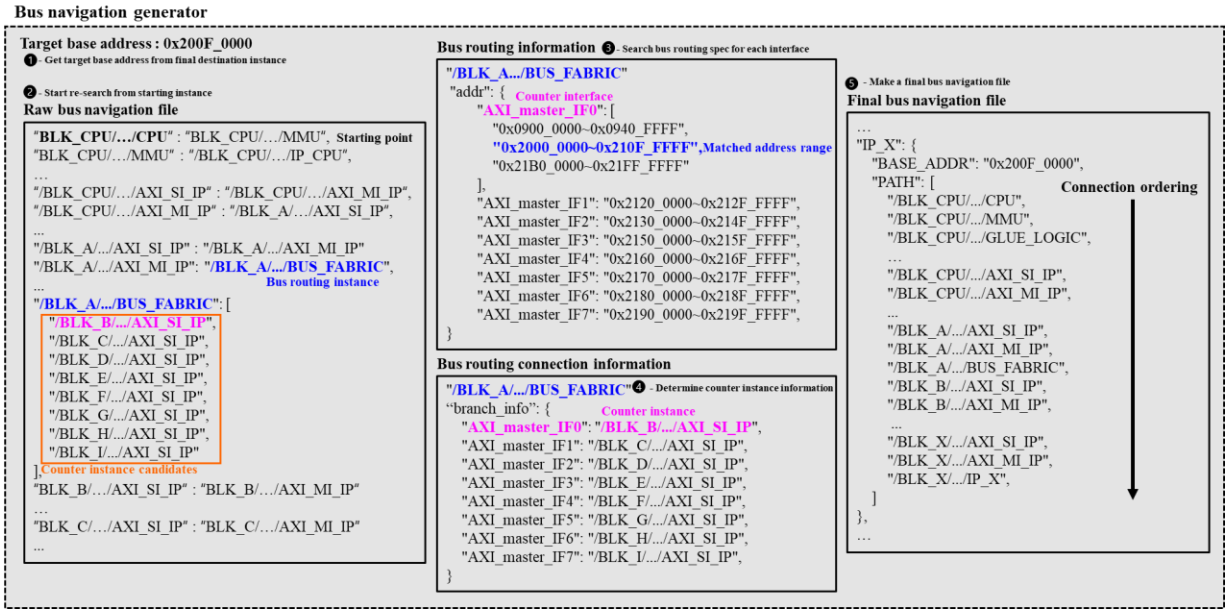


Figure 3. An example of determining the counter instance at the bus routing instance.

Phase2 – Automatically finding a debug scope during simulation

To detect abnormal states in the simulation, such as bus hang and error response, we bound testbench-specific SystemVerilog interfaces to all DUT internal interfaces during simulations and exported observed data as a log file [4]. To generate the testbench interfaces, we used the JSON file extracted from Phase 1-1, Figure. 1, and found all masters and interface information in the file. Next, we defined representative interfaces for each bus protocol, such as CHI, AXI, ACE, and custom bus protocols and created the interface instance for all the masters. Within each protocol-specific interface, there are bus protocol inspectors internally. These inspectors have been developed by modeling each bus protocol in SystemVerilog. The inspectors include a bus protocol handshake detector that detects only valid transactions, a traffic burst length inspector, and include timer logic to detect bus hang. The inspectors output logs periodically that meet the specifications of each bus protocol during the simulation. Afterwards, we bound all generated interface instances to all masters before simulation.

When the simulation starts, bus protocol interface logs are output from all connected interfaces. If a log file is created for each master interface, a large number of files will be generated, and we have determined that this approach is highly inefficient. Instead, we have generated individual interface log files for each bus protocol, such as AXI, ACE, CHI, and so on. Subsequently, only valid traffic detected by the interfaces is outputted to the respective log files in chronological order. Log files are generated individually for each bus protocol such as AXI.json, CHI.json and ACE.json. Plus, these interfaces also have timer logic to detect bus hang. Figure 4 illustrates the overall process of extracting valid bus protocol logs during the simulation.

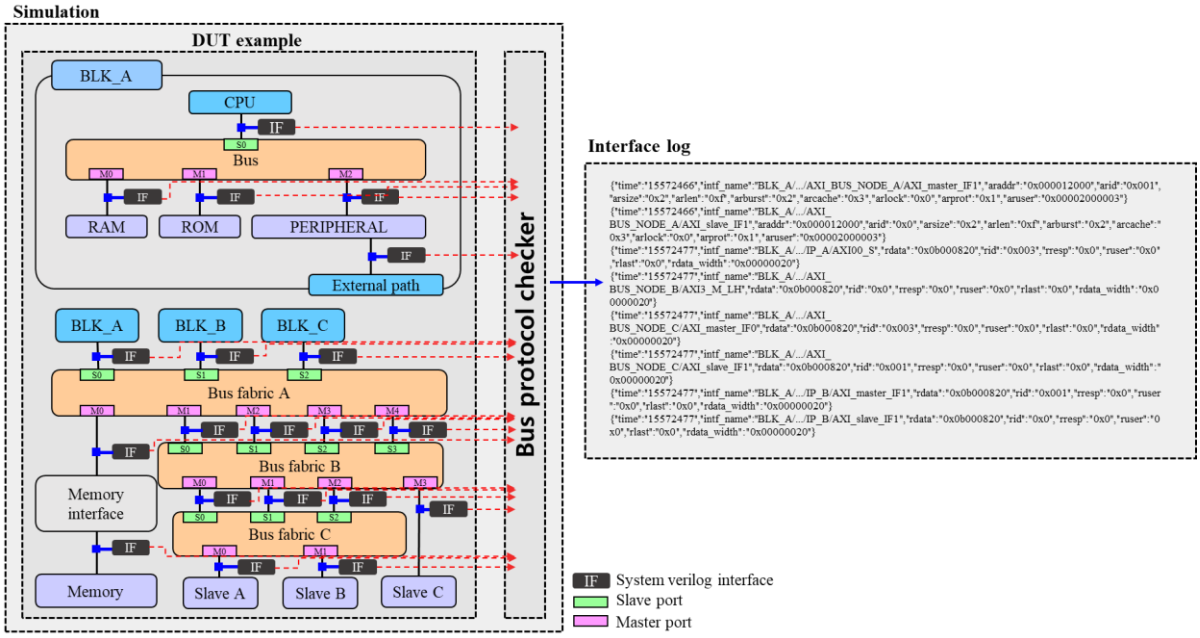


Figure 4. The process of extracting a valid bus protocol simulation log

To analyze the log concurrently with the simulation, a separate Python script is executed with the start of the simulation. This script activates in the event of a bus error response in the log or if the log stops printing while the timer reaches a certain threshold. Its purpose is to identify the problematic area of the system and generate a debug scope. First, the Python script retrieves bus connection information from the bus navigation file based on the traffic's address information. Next, it tracks the traffic trace in the log, starting from the end and working backwards. The reason this approach is more efficient than starting from the beginning is that it excludes traffic that has already been processed. If traffic were to be traced sequentially, it would be inefficient as it would review traffic that has been processed normally. On the other hand, by tracking the flow of traffic in reverse, it quickly identifies the most recent problem and avoids unnecessary traffic review. As a result of this tracking, the Python script reports the problematic instance to the verification engineer. Figure. 5 illustrates an example of using BTS in a SoC to identify the instance causing bus hang when the master issues write traffic but receives no response. If the BLK_CPU does not receive a write response from IP_A for an extended period, despite issuing a write transaction on 0x1083_0808, and the timer exceeds a threshold or the log stops printing, BTS is triggered to debug this abnormal situation.

BTS first calculates the IP base address of the problematic traffic. This is because the created bus navigation for each IP is based on the IP's base address. In this example, the calculated base address is 0x1083_0000. Next, it retrieves the bus connection information corresponding to 0x1083_0000 from the navigation file. Then, BTS tracks the write traffic associated with 0x1083_0808 in the interface log, starting from the last instance in the bus connection information list, as the write response will be issued from the last instance to the start instance. The last write response log in the interface log was printed at BUS_FABRIC_2, suggesting that BUS_FABRIC_1 did not send the response to BLK_CPU. Therefore, the bus hang point is identified as BUS_FABRIC_1. Consequently, BTS reports the bus hang instance name to the verification engineer via the terminal. Figure. 6 shows the debugging result of BTS for the bus hang related to the traffic associated with the 0x1083_0808 address, as printed on the terminal.

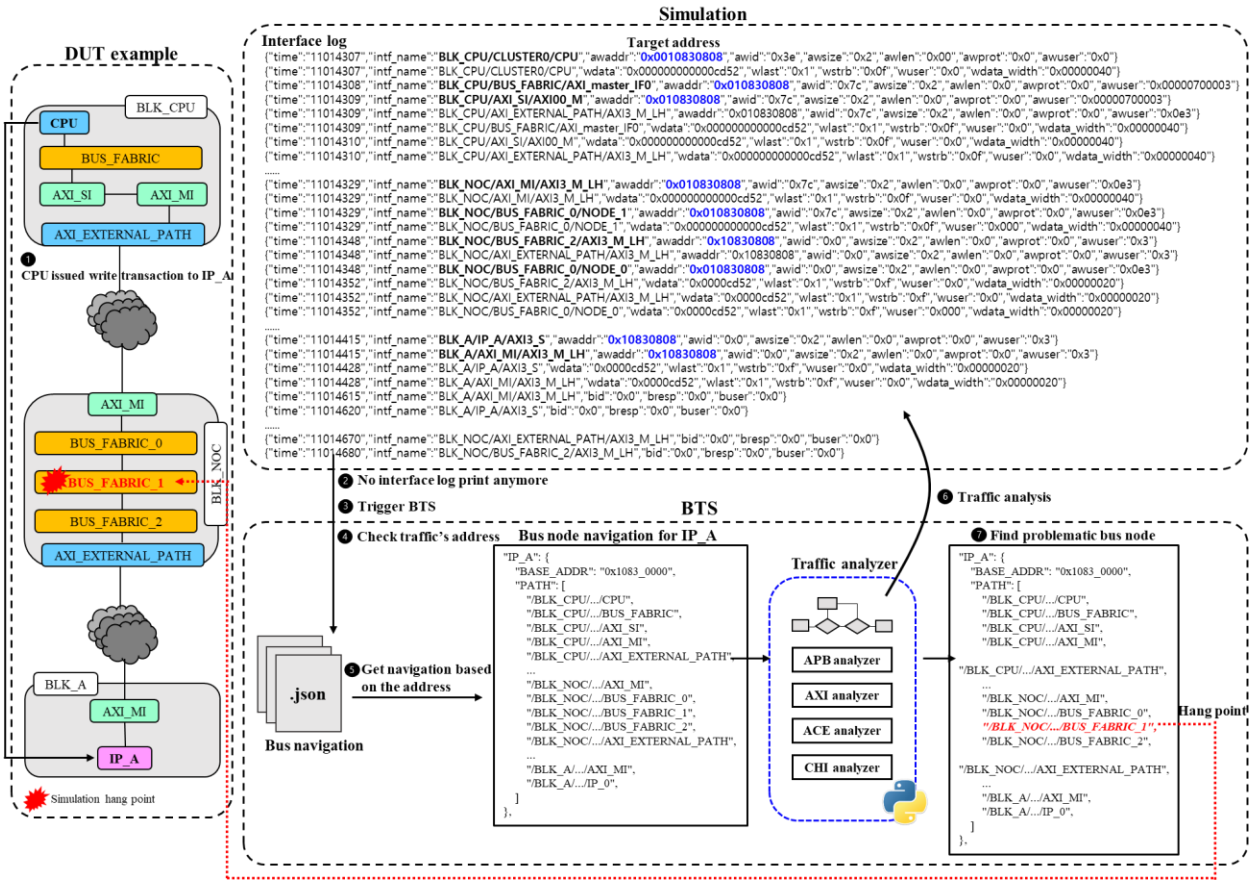


Figure 5. Workflow for discovering the simulation hang point.

User view(Terminal)

LOG:: Simulation is started…
……
LOG:: Basic booting is done
LOG:: Main scenario is started
……
LOG::BLK_CPU::CPU:: Write transaction is issued. addr : 0x10830808, wdata : 0xCD52
……
……
……
TIMER:: Reach the threshold
LOG::Trigger BTS…
……
LOG:: BTS has been triggered automatically by abnormal state.
…
BTS:: Analyzing the interface log …
BTS:: BLK_CPU didn`t get write response about 0x10830808
BTS:: Problematic instance is "BUS_FABRIC_1"
BTS:: Please debug "BUS_FABRIC_1" first about 0x10830808 traffic
BTS:: Result – Bus hang. Simulation end

Figure 6. Terminal view seen by the user when BTS is executed

III. Experimental Result

We have identified bugs such as bus hang and error response that can be easily fixed using BTS in our recent three SoC projects. Table 1 shows the number and ratio of errors corresponding to bus hang and error response among all reported bus cases in our three SoC projects. Although the error ratio varies for each of the three projects,

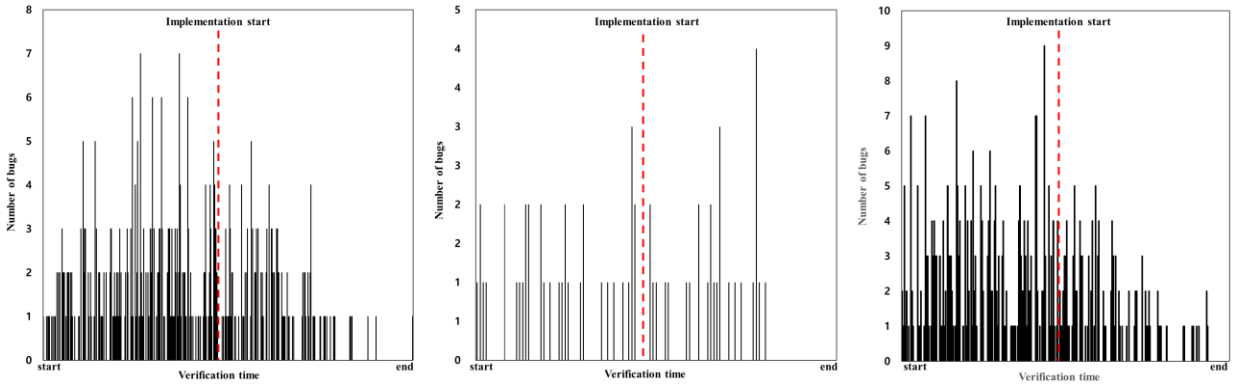
it's noteworthy that these errors account for approximately 10-12% of the total errors. This suggests that around 10-12% of all errors can be identified with BTS. We also analyzed the frequency of these errors throughout the entire SoC project period. Figure 7 shows the cumulative occurrence rate of bus hang and error response bugs for each of the three SoC projects during their respective verification periods. This is calculated by converting the error counts mentioned in Table 1 into percentages. For all three projects, at least 50% of the total error cases occurred before the milestone of implementation start during the overall SoC development period. This generally indicates that such bugs were primarily concentrated and occurred frequently in the early stages of the project schedule. Even when viewed based on the start time of synthesis, a bug occurrence rate of more than 50% can still be observed.

Detecting as many bugs as possible before the start of synthesis could potentially shorten the SoC development schedule. This is because the synthesis stage holds significant importance in the overall SoC development schedule. Bugs discovered after this stage can affect the synthesis process and may even require re-synthesis, leading to potential delays in the layout schedule.

Applying BTS in SoC verification is beneficial as it allows for not only the accurate and easy detection of bugs but also a reduction in debug time. Moreover, achieving quick stabilization of bus integration before the start of synthesis can ultimately help to shorten the SoC project schedule. Also, the bus hang bug can create bottleneck, which may delay the identification of other bugs such as IP malfunctions, security violation and etc. However, through rapid bus stabilization, these bugs can also be identified quickly.

Table 1. Number of bugs that can be easily identified using BTS in three SoC projects

Project	Bus hang(#)	Error response(#)	Total bugs(#)	Ratio(Bus hang + Error response)
A	320	146	3,964	0.1176
B	76	36	1,057	0.106
C	411	92	4,199	0.12



(a) Project A (b) Project B (c) Project C

Figure 7. The occurrence rate of errors such as bus hang and error response during the entire projects period and the bugs occurring rate up to the start of implementation in three SoC projects

IV. CONCLUSION

BTS is an effective and user-friendly system for debugging bus hang errors and error responses, which frequently occur before synthesis begins due to the unstable bus integration of the SoC. Verification engineers often spend a considerable amount of time debugging these bugs because they have to check all bus nodes to find the node where the problem first occurred. Engineers with less verification experience may require even more time to debug these bugs. However, BTS can quickly identify and debug these bugs. Therefore, using BTS during SoC development can significantly reduce the time required for verification, quickly stabilize the SoC bus architecture, and it can lead to a shortened the SoC development schedule. As a result, BTS can be a fast debug methodology in the increasingly complex SoC architecture trend.

REFERENCE

[1] Foster, Harry D. "Trends in functional verification: A 2014 industry study." *Proceedings of the 52nd Annual Design Automation Conference*. 2015

[2] Foster, Harry D. "2020 Wilson Research Group Functional Verification Study", Oct 2020

[3] Magillem SEC 5.2021.1.p3 Release Notes

[4] Yonghyun Yang. “A Fast SOC Verification Methodology Using Traffic Monitor and IP-XACT Based Integration Automation” *57th DAC. 2020*