Enhanced Dynamic Hybrid Simulation Framework for Hardware-Software Verification

Victor Besyakov
Untether AI
Toronto, Ontario, Canada
https://www.untether.ai

***Abstract*- The hardware-software co-verification feasibility constitutes a significant challenge to the System-on-Chip (SoC) development. A comprehensive co-simulation can take months, if not years of CPU time. To solve this problem, a hybrid approach was introduced several years ago. The motivation to use it derives from its ability to address the contradiction between hardware and software development requirements. On the one hand, the software is more attractive to accelerate the simulation and does not require high accuracy all the time. On the other hand, the hardware development relies mainly on a time-aware simulation and can only sacrifice precision occasionally. This paper presents an enhanced dynamic hybrid framework that can satisfy both requirements and may be used for hardware-software co-verification.**

I. INTRODUCTION

A. Background

At best, a hybrid simulation can be described as a combination of “two or more different simulation techniques into one simulation environment in order to benefit from the different simulation characteristics, e.g., different levels of accuracy and simulation speed” [1]. Conceptually, it can be categorized into two groups: static or spatial and dynamic or temporal [1].

One of the most common examples of the static hybrid approach is the hardware-assisted RTL verification. Typically, the synthesizable design runs on the hardware accelerator, while the non-synthesizable testbench remains within the software simulator realm [4,5]. The downside of this approach lies in the performance. The effectiveness of a static hybrid platform is always limited by the slowest simulator.

Compared to the static platform, the dynamic hybrid solution offers an additional option that allows the user to switch between various simulators (i.e. levels of abstraction) during runtime. This technique is frequently utilized to explore CPU unit microarchitecture, performance modeling, and software power consumption [2,3,6,7]. One of the main challenges of the dynamic hybrid simulation is to maintain synchronization among several simulators. It is a very complex and application-dependent task since the entire state of the simulated design must be coherent for all engines [1]. As a result, the dynamic hybrid simulation solution is not widely used in the hardware-software co-verification and there are no commercially successful tools in the marketplace. The proposed enhanced dynamic hybrid simulation framework aims to address these challenges.

B. Enhanced Hybrid platform architecture

Fig. 1 shows the architecture for this framework. It consists of two main subsystems, called host and client, which communicate with each other via a data router and the Inter-Process Communication (IPC) channels (dotted lines).

The implementation of the IPC may vary depending on system requirements. The most conventional approach is to use a standard TCP/IP socket API.

In a typical hybrid system, the Host subsystem is the fastest part of the simulation. The server can be either an Instruction Set Simulator (ISS) or even an actual processor unit.

The Client tends to represent a more timing-accurate part. This may include a conventional RTL simulation, SystemC cycle-accurate model, hardware-assisted simulation, or FPGA prototyping (ref. Client Fig. 2).

The reconfigurable data router (ref. to Router in Fig. 1 and Fig. 3) governs data flow between Host and Client. It also implements an adaptive layer that maps associated interfaces to the IPC protocol. In the case of static simulation, the direction of the data stream always remains the same. For dynamic simulation, data flows can be altered based on the Switch Simulator request events.

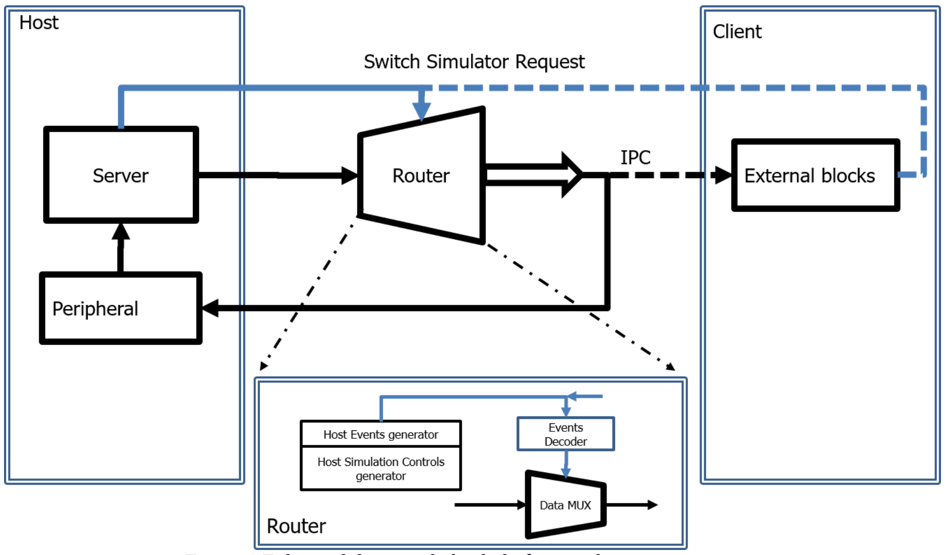


Figure 1 Enhanced dynamic hybrid platform architecture

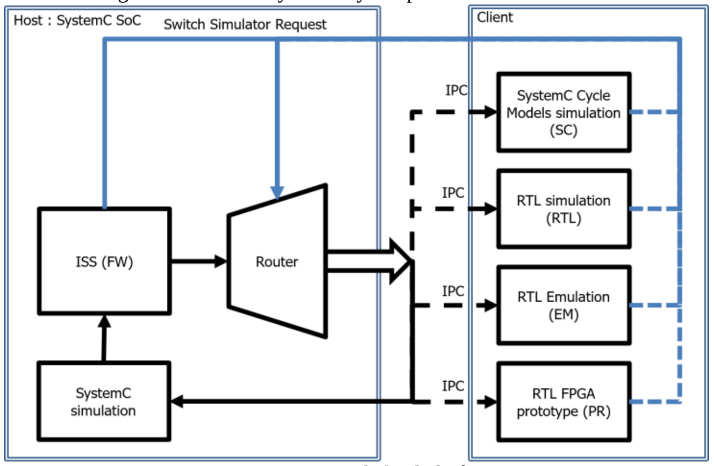


Figure 2 ISS centric hybrid platform

The following subsections cover IPC data interchange, event switching protocols, synchronization algorithms, and hybrid system limitations. This will be illustrated by examples drawn from the ISS-centric simulation platform (see Fig. 2).

II. HYBRID SIMULATION FRAMEWORK ARCHITECTURAL ASPECTS

A. Data Router

The data router manages data traffic between three types of input-output ports (see Fig. 3).

The Client Control ports are dedicated to the exchange of control information between the Client and Host subsystems. This Control flow consists of simulation controls (start the simulation, reboot, freeze clock, and e.t.c) and routing events. Each external port pair interacts independently with the data router and has its own TCP/IP socket.

The second pair is a SystemC simulation data path (Host SoC ports). They enable a connection between the Fast CPU model and the rest of the SoC Host system.

In operational terms, data routing can be static (default mode) or dynamic. When the simulation is launched, the data router obtains the default routing table configuration in the initialization step. In the case of a static hybrid simulation, the routing table remains unchanged during the entire simulation. With dynamic routing, Data MUX

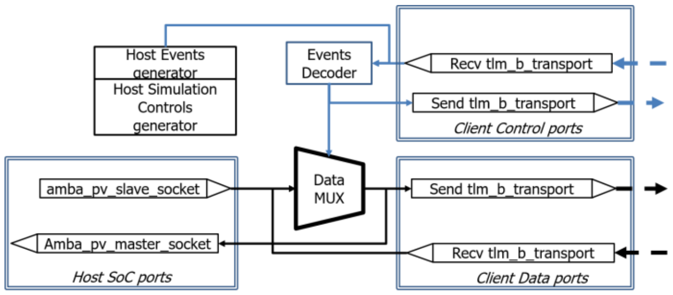


Figure 3 Router block diagram (one Client example)

updates the routing table on a regular basis based on information from internal or external switching events. The event receiver is responsible for processing all events from internal sources and external client control ports.

The Data router is implemented as an extension of the amba_pv_to_tlm_bridge class [14]. In addition to the traffic directing function, the router is responsible for establishing the TCP/IP socket-based communication. It also records and stores inbound and outbound transactions in the traffic database, and manages (accepts or rejects) DMI requests from System C.

The external data ports (Client Data Ports) and the operation of the data ports will be described in paragraph B below.

B. Host-Client data flow

An example of client-host integration is shown in Fig. 4, where the host subsystem is represented by the ARM Fast Processor model [14]. ISS interacts with a hybrid system's peripherals through the ARM Programmer View (PV) ports (see pvbus_m and pvbus_s in Fig. 4). The PV Bus provides functionally accurate communication between bus masters and slaves, but doesn’t have any notion of the data exchange protocol [14]. The PVBus2AMBAPV and AMBAPV2PVBus components implement the required protocol conversion, such as PV to AMBA and vice versa.

The system data flow is illustrated with a read transaction as follows. The outgoing Host-to-Client read request (follow the red lines in Fig. 4) is sent from the processor model output to the ama_pv_m port [14] via the PVBus2AMBAPV protocol converter. Next, the router fetches the AXI bus transaction pair (amba_pv_transaction, amba_pv_extension) and forwards it to the AMBAPV2Shunt adapter. The egress adapter converts read request from the amba_pv to the Shunt TLM structures (cs_tlm_generic_payload_header, cs_tlm_axi3_extension_payload_header [11]) and sent it over TCP/IP to the corresponding client.

On the Client side, the Shunt2UVM adapter gets an ingress transaction, converts it to the regular uvm_tlm_generic_payload, and passes it to AXI BFM. If BFM is compatible with uvm_tlm_generic_payload (example Synopsys AMBA AXI3 BFM [13]), the AXI read request can be converted directly into a bus transaction, with no further transformation. Otherwise, another level of mapping must occur. Then BFM returns the results of the read transaction (follow the purple lines in Fig. 4). The following components of the data flow diagram will be involved: UVM2Shunt adapter, TCP/IP packet, Shunt2AmbaPV adapter, amba_pv_s port, AMBAPV2PVBus, and finally the PVbus input port of the processor.

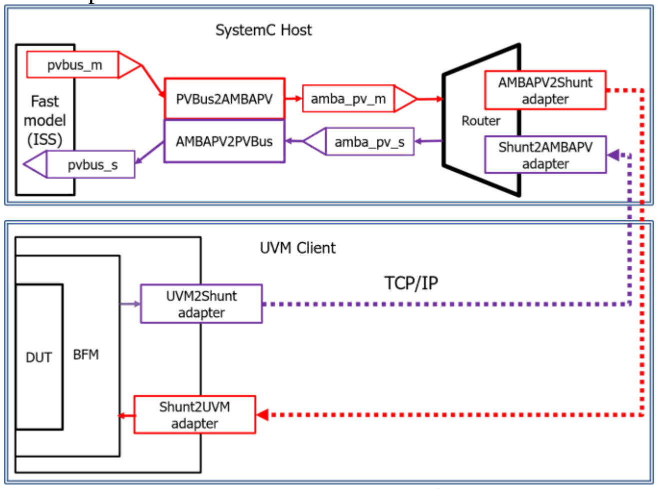


Figure 4 Host-Client data flow

Non-bus transactions such as interrupts or sideband signals are not present in the Fig. 4 example. They need a dedicated set of router ports and converters (for instance: AMBAPVSignal [13] protocol to shunt structures converter). Nevertheless, the principle of operation is still the same. The host processor retrieves the Client's signals via the chain of the Client to the Shunt structures adapter, TCP/IP packet communication, and the Shunt to the AMBAPVSignal converters.

C. Inter-Process Communication

The primary objective of the IPC is to provide a unified reusable data exchange protocol for all potential hybrid platform clients. In practice, the selection of the communication methods is a compromise between simulators interoperability, performance, and data transfer overhead, just to name a few.

The main reason for selecting a TCP/IP protocol is that the hybrid platform requires communication between remote applications.

For the framework, the choice of the TLM 2.0 generic payload [9] over the TCP/IP socket communication is quite natural. The Host and the SC client are TLM 2.0 compliant by definition as SystemC components (ref. to Fig. 2). The UVM-based RTL client can get TLM 2.0 capabilities from the uvm_tlm_generic_payload [8] class.

However, two clients require special consideration. The emulation-based client (see Fig. 2) is linked to external applications via the standard SCE-MI co-emulation interface [10] like most hardware accelerators. Regarding the FPGA client (see Fig 2), there is no standard that can describe how to connect proprietary prototypes to the external world. This leads to the conclusion of creating a TLM 2.0 adapter for these two components instead of backup two or more different protocols.

The open-source communication library "The Shunt" [11,12] is selected as the TCP/IP application layer. It covers all SystemC, System Verilog, and C clients and facilities to establish TLM 2.0 connections.

III. SWITCHING EVENT MANAGEMENT

The hybrid simulation works with events originating from the host and external clients. The router event decoder receives the first set of events directly from the Host Events generator (Fig. 3). The second series of events pass via a TCP/IP connection (Client Control ports Fig. 3). Only three types of events are listed in Fig. 5: temporal, address, and register. However, hybrid systems may have events originating from another source. For example, a software-centered system may have C/C++ debugger or exception handler events.

The “Switching on” and “Switching off” (Fig. 5) requests are generated, if some switching Client simulator criterion is satisfied (see TABLE I). The event decoder (Fig. 4) updates the data routing table according to the event type, associated destination, and event data routing information. It inserts a new routing table entry upon “Switching on” request arrival and deletes an expired routing when the “Switching off” event. Typically, the “Switching on" event enables the destination of the external clients and the “Switching off” request cause to redirect the data back to the host SoC simulation.

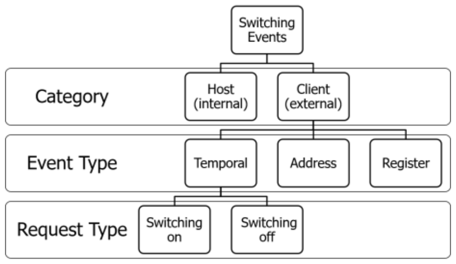


Figure 5 Events Hierarchy

TABLE I
TYPES OF SWITCHING EVENTS

N	Event Name	An additional Attribute	Request type	Source (Fig. 2 abbreviation)	Event Trigger
1	Temporal	N/A	Switching ON and Switching off	SC,RTL,EM,PR	If the simulated time is within the specified interval.
2	Address	Read and Write Transaction	Switching ON and Switching off	Host,SC,RTL,EM,PR	If the transaction address falls within the specified address range.
3	Register	Read and Write Transaction	Switching ON and Switching off	Host,SC,RTL,EM,PR	If the address and the value of the present transaction are equal to the corresponding values in the reference table.

IV. SYNCHRONIZATION ALGORITHM

As mentioned above, the synchronization of data and simulation time between the different simulation agents is a complicated task. It depends on the type of firmware, the implementation of the client/host simulator and on the simulation strategy itself. Indeed, different hybrid system scenarios may require a different sync.

Ideally, the dynamic hybrid simulation platform user should be able to select the most appropriate synchronization algorithm at any time during the simulation. But in practice, it's very difficult to accomplish. A more pragmatic approach might be to choose the synchronization algorithm before the simulation starts and retain it during the entire simulation. Also, when it is possible to select a simulation strategy that can reduce synchronization overhead or even requires no synchronization. One option is to adopt a semi-dynamic approach that requires a one-time swap between host and client [2]. This strategy needs no more than one-time forward synchronization per simulation.

A. "Forward/Backward" synchronization

The "Forward/Backward" synchronization is the most simple and straightforward technique. Briefly, while executing the simulation, the router records data transactions and stores them into the traffic database. Then, when the "Switching on" event occurs, the router transmits all accumulated transactions to the external Client. The Client sends the updated memory context back to the Host during the "Switching off" event.

This kind of synchronization can be optimized by filtering out the exchange traffic. For example, a decision could be made to sync only with the nonzero entry transaction at the most recent address.

The Forward/Backward synchronization may require many synchronization sessions, especially when the host starts looking for information about the client (interrupts, state or data registry updates, memory consistency, etc.). If this occurs too frequently, it can significantly slow down the entire simulation.

B. "Freeze-unfreeze clock" synchronization

The "Freeze-unfreeze clock" is a gating clock synchronization method. It is convenient for distributed simulation when each client runs on their own simulation engine or when the host or client operates in an interactive debug mode. For each data update cycle, the clock is gated for the data exchange period and activated when the exchange is finished.

An optimized algorithm version may take a more adaptive approach by varying the number of clocks for each simulation step. A more sophisticated arbitration system can also be implemented when the client itself can request the service from the host.

C. "By Restart" synchronization

The synchronization "by Restart" is useful, when the fast host runs the "real" firmware, but the client is placed over the slow RTL simulation. For synchronization with the slow simulator, the host has to be restarted at the beginning of each client simulation cycle.

Back in the 90s, this type of synchronization was used in the Hardware modeler implementation [16]. As it was noticed in [15], the hardware modeler had only modest performance and a long simulation problem since it is necessary to reapply the entire stimulus history in order to obtain the next vector. The same drawback also applies to hybrid simulations. However, for non-intrusive black box simulations, this technique is irreplaceable.

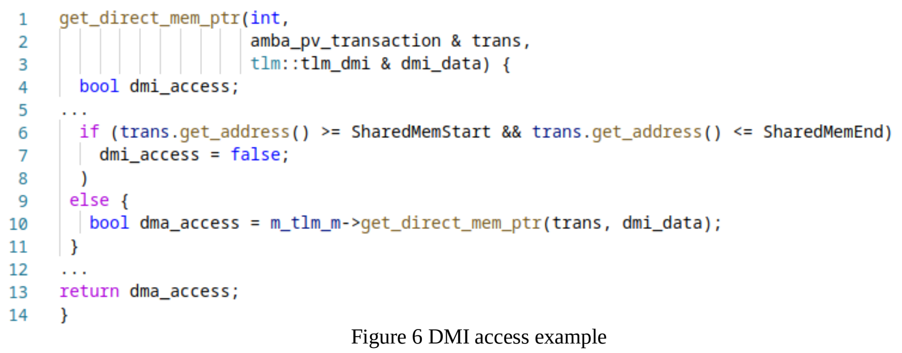
V. IMPLEMENTATION NOTE

A. ISS optimization and Direct Memory Interface (MDI)

The TLM DMI provides methods for bypassing the SoC bus infrastructure (b_transport [9]) and accessing the SoC memory directly. This functionality is critical to simulation performance. According to the study presented in [19], the boot time of Linux is 20 seconds with DMI and over 2 hours without DMI. But from a data router perspective, all DMI activity is hidden, which is a problem. Once the ISS gets the DMI pointer, it forwards all transactions straight to the SoC memory.

Ideally, to solve this issue, the user should have complete control over the data flow. In reality, this is not feasible, as often the ISS is a proprietary black-box model.

The more practical solution is to cut off all attempts to obtain DMI access to the shared host/client memory. This may be accomplished by modifying get_direct_mem_ptr(tlm_transaction & tx ,tlm_dmi& dmi) function inherited from amba_pv_to_tlm_bridge [14,17] (see code snapshot in Fig. 6).



```
get_direct_mem_ptr(int,
                   amba_pv_transaction & trans,
                   tlm::tlm_dmi & dmi_data) {
  bool dmi_access;
...
  if (trans.get_address() >= SharedMemStart && trans.get_address() <= SharedMemEnd)
    dmi_access = false;
  )
 else {
   bool dma_access = m_tlm_m->get_direct_mem_ptr(trans, dmi_data);
 }
...
return dma_access;
}
```

Figure 6 DMI access example

B. Software/Hardware instrumentation

Conventional software instrumentation (SI) is a well-known technique that focuses on modeling, runtime, power consumption, and statistical profiling [22]. In the case of hybrid simulation, SI creates switching events based on specific program behavior. For instance, to redirect traffic upon the Linux prompt or make a breakpoint for debugging purposes. It can also play a role in optimizing synchronization overhead by analyzing a shared memory, creating a database of transaction history, and avoiding time-consuming MDI blocking (see paragraph A above).

The RTL code instrumentation (RI) is an integral part of the System Verilog standard or System Verilog simulators (assertions, code coverage, power-aware verification, UCLI, “$ system” task, etc.). Switching events produced by the integrated RI are the most preferred. It guarantees the same simulated behavior and does not intervene with the original RTL. The same applies to the use of synthesizable assertions and probes in FPGA-based prototypes.

The non-standard RI has no support by any commercially available RTL simulators. Users need to code proprietary methods and ensure design consistency. The use of an open-source tool may facilitate this. For example, Verilator has a built-in code pipeline filter that can "modify inputs without changing primary sources" [20].

C. Distributed simulation challenges

The Host-Clients synchronization algorithm could be significantly simplified if it can be assumed that there is no direct interaction between external clients. It means there is no sharing of resources (memory, buses, sideband signals, etc.) among clients. All client-to-client communications should be conducted via the host system. Another assumption that could be made is that at any given simulation time slot there should be only one client/host data exchange. It will ensure the order of the synchronization transactions. At first glance, such limitations could constitute an obstacle to the hybrid platform adaptation. However, focusing hardware/software co-simulation on the "one client per test" strategy can significantly reduce the negative impact of the synchronization trade-off. For the Host architecture, this means that the Host SoC model has to contain all external client models, all application connections and can function as a complete standalone simulation.

VI. Result and Conclusion

The Dynamic Hybrid Simulation Framework was developed to evaluate the main advantages and disadvantages in terms of the applicability for ASIC software development and functional verification processes. It aims to enable early HW/SW integration by establishing a common platform across different simulator engines. This framework is still in the experimental phase, but even today it could provide significant benefits in software-driven verification.

In static mode, the performance results are comparable to all other static hybrid simulation platforms [18]. One of the framework variants was used to develop a real production software driver.

The dynamic mode was exercised as a proof of concept. In a semi-dynamic configuration (similar to [2]), it shows good performance for the U-BOOT boot loader [21]. It took under five minutes to get a prompt, compared to 4 hours of pure RTL simulation.

The advantages of a dynamic hybrid platform include:

- *Enables shift left Perform SW development and debug on SoC RTL rather than on first silicon and/or prototyping*

- *Provides a more realistic stimulus for RTL verification*
- *Enables HW and SW early integration*
- *Allows finding/fixing system-level bugs in HW, rather than as SW*
- *Helps to identify bottlenecks in performance*
- *Reduces simulation time*
- *Creates a common platform for HW, SW development, and prototyping (emulation)*
- *Improves Verification coverage and RTL quality*

Besides the many benefits and opportunities of dynamic hybrid simulation, this approach faces a number of challenges related to the heterogeneous nature of the hybrid system, such as:

- *Absence of a common simulators API*
- *No debugging environment for the distribution simulations.*
- *No requirement to VIP vendors to be TLM 2.0 compliant*
- *Lack of interoperability across various RTL simulators.*
- *No standard SW/RTL instrumentation*

VII. ACKNOWLEDGMENT

Special thanks to Mr. Igor Melamed for his in-depth review, numerous suggestions, and remarks.

REFERENCES

[1] Kraemer, S. Design and Analysis of Efficient MPSoC Simulation Techniques. Ph. D. Dissertation, RWTH Aachen Univeristy 2011
[2] B. Neifert, Rob Kaye, “High Performance or Cycle Accuracy? You can have both”, ARM White paper, ARM Ltd. 2012.
[3] Fast Models System Creator Version 9.6 User Guide, 2018 Arm. 100997_0906_00 (ID011118)
[4] M. Monton, A. Portero, M. Moreno, B. Martinez, and J. Carrabina, “Mixed SW/SystemC SoC emulation framework,” in Proceedings of the IEEE International Symposium on Industrial Electronics (ISIE ’07), pp. 2338–2341, June 2007.
[5] Arief Wicaksana, Amir Charif, Caaliph Andriamisaina, Nicolas Ventroux. Hybrid Prototyping Methodology for Rapid System Validation in HW/SW Co-Design. The Conference on Design and Architectures for Signal and Image Processing 2019 (DASIP 2019), Oct 2019, Montreal, Canada. cea-02494007
[6] W. Lee, K. Patel, and M. Pedram, "B2Sim - A Fast Microarchitecture Simulator Based on Basic Block Characterization." 4th Int. Conf. Hardware/Software Codesign & Syst. Synthesis (CODES+ISSS '06) Seoul, Korea, 199–204
[7] A. Muttreja, A. Raghunathan, S. Ravi, and N. K. Jha, “Hybrid simulation for embedded software energy estimation,” in Proceedings of the Design Automation Conference (DAC). New York, NY,USA: ACM, June 2005, pp. 23–26
[8] UVM 2020 v1. 1 Library Code for IEEE 1800.2 (UVM). Available: https://www.accellera.org/downloads/standards/uvm
[9] SystemC 2.3.3 (Includes TLM). Available: https://www.accellera.org/downloads/standards/systemc
[10] SCE-MI 2.4 Standard Co-Emulation Modeling Interface Available: https://www.accellera.org/downloads/standards/sce-mi
[11] SystemVerilog/SystemC "TCP/IP Shunt" Open Source Client/Server TCP/IP socket based communication library. Available: https://github.com/xver/Shunt
[12] Victor Besyakov “TCP/IP Socket Based Communication for SystemVerilog Simulation” 2018 SNUG Boston
[13] VC Verification IP for Arm AMBA Protocol. Available: https://www.synopsys.com/verification/verification-ip/amba.html
[14] Fast Models Version 11.15 Reference Manual APM 2014–2021 100964_1115_00_en
[15] James A. Rowson Hardware/Software Co-Simulation 1994 DESIGN AUTOMATION CONFERENCE Redwood Design Automation Inc., San Jose, CA
[16] S. Lee A Hardware-Software Co-simulation Environment. Ph. D. Dissertation UNIVERSITY of CALIFORNIA at BERKELEY 1993.
[17] Doulos. Tutorial 2 - Response Status, DMI, and Debug Transport. From https://www.doulos.com/knowhow/systemc/tlm-20/tutorial-2-response-status-dmi-and-debug-transport/
[18] Claude Helmstetter, Vania Joloboff. SimSoC: A SystemC TLM integrated ISS for full system simulation. APCCAS - IEEE Asia-Pacific Conference on Circuits and Systems – 2008
[19] Jason Andrews, SystemC TLM-2.0 Virtual Platform Direct Memory Interface (DMI) Performance Impact. Cadence Available: https://community.cadence.com/cadence_blogs_8/b/sd/posts/systemc-tlm-2-0-virtual-platform-direct-memory-interface-performance-impact
[20] Wilson Snyder, Ten Creative Uses for Verilator (Slides), 2019-11 CHIPS Alliance Available: https://veripool.org/papers/Verilator_Ten_Creative_CHIPSTools2020.pdf
[21] Das U-Boot – Wikipedia https://en.wikipedia.org/wiki/Das_U-Boot
[22] Kumar, J. Cano, A. Brankovic, D. Pavlou, K. Stavrou, E. Gibert, A. Martínez,and A. González, “Hw/sw co-designed processors: Challenges, design choices and a simulation infrastructure for evaluation,” in2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2017, pp. 185–194.