Current Status and Challenges of SoC Verification for Embedded Systems Market

Wooseung Yang, Moo-Kyeong Chung and Chong-Min Kyung,

Dept. of EECS, KAIST Daejeon, Korea, 305-701

Abstract—SoC has become an indispensable solution in the embedded systems market. This tutorial introduces today's main issues of SoC design with a focus on the verification solutions proposed by EDA vendors and SoC developers. After the SoC platform based on several embedded cores is fixed, design efforts are focused on the verification of peripheral IP's and debugging of the software in the context of the platform. For IP verification, formal methods are first used when applicable and suitable to statically remove design bugs and improve coverage and test-bench automation tools are applied to test the verification of IP's and debug in the context of the platform. For IP with realistic test vectors, finally all the IP's are mapped in FPGA in the emulator to be verified in the real operating environment. For integrated system verification the emulation environment is set up as soon as the platform is selected and the block-level partitioning is done. Well-established emulation platform helps progressive refinement of newly added SoC components and early development and verification of the software.

Index terms—SoC design, SoC verification

I. INTRODUCTION

SoC has become an indispensable solution to meet the cost, area and power requirements of the embedded systems in the embedded systems market. In the design and verification point of view, the size and complexity of the SoC presents new opportunities and challenges to us. This tutorial introduces today's main issues of SoC design with a focus on the verification solutions proposed by various EDA vendors and SoC developers.

Main differences of the SoC design from the earlier ASIC design lie in the emphasis on the IP re-use, design platform and embedded software issues. As the SoC platforms based on several embedded cores are settled, the design efforts are focused on the verification of IP's and the debugging of software in the context of the platform.

To enhance the IP verification productivity the use of formal methods such as model checking, assertion or dynamic formal verification[1] are encouraged and the test-bench automation languages or tools[2] are used to model the test environment. Recently, low-cost emulation solutions based on a small number of multi-million FPGA devices are rising as a flexible SoC IP verification tool[3]. They provide a good visibility for the internal signals of the design mapped in the FPGA. Another approach allows debugging in RTL while running gate-level design in the FPGA [4].

For the debugging of software in the platform, co-simulation tools with enhanced communication techniques [5][6] are commonly used albeit they are very slow. Many simulation / fast-prototyping systems support co-verification with the instruction set simulator running in the host computer.

All the verification techniques have their merits and demerits. Considering that, it is a must that well-established verification flow is set up before the design activity begins. This paper will show that an intensive block-wise verification followed by an in-system verification using a realistic test environment is a very practical method.

This paper is organized as follows. In Section II, current IP verification tools and methodologies are surveyed and in Section III, the verification methods for the integrated systems are explained. A procedure for setting up an emulation platform for a JPEG system starting from a legacy C code for the JPEG standard will be demonstrated in Section IV and conclusions are made in Section V.

II. SOC IP VERIFICATION TOOLS AND METHODS

This section summarizes the trends in IP verification considering the SoC. In addition to the conventional simulation methods, the following verification methodologies are used.

A. Formal Methods

As an alternative and complementary solution to the simulation-based verification methods, several formal methods are commercialized and settled as good verification solutions; equivalence checkers and model checkers. Common benefits of these formal methods over the simulation approaches are that they do not require test-vectors and that the result of the verification is complete in the sense that the correctness of the model is proven mathematically [1][7][8].

Equivalence checkers compare two versions of a circuit to guarantee that they are functionally identical. This is helpful when verifying the behaviors of IP's before and after the logic synthesis, scan-insertion, etc. Nearly all major EDA vendors have equivalence checking tools in their verification tool chains.

Model checkers, on the other hand, compare the behavior of an IP with a specification described in temporal logic such as Computation Tree Logic (CTL) or Linear-time Temporal Logic (LTL). Thus, one can apply the model checking before the HDL description is completed. This can greatly reduce the design verification time since early detection of design error is critical in verification.

Another major focus in recent formal research is on mixing the formal techniques with the simulation-based approaches. Since formal methods generally require a large memory space,
semi-formal analysis can be chosen as a trade-off. Typical example is assertion-based verification. Basic idea is to include assertion statements (or components) in the HDL code, which is not a very new concept. Various contributions are made to enhance the ways to specify the assertions [9][10][11]. Open Verification Library initiative (OVL) [9] defined structural components to specify assertions. Superlog Design Assertion Subset (DAS) [10] proposed by Co-Design Automation and Real Intent Co. provides procedural way to assert properties. Sugar specification from IBM uses temporal logic [11]. These assertion methods are being unified with the initiative of Accellera.

B. Test-bench Automation

Although formal methods are more confident than simulation, the latter is still important when the design size is very large. And also, in many cases, it is much easier to create large amount of test vectors than to describe the complete specification for the design. To work with HDL simulator, conventional test-bench is described in HDL or C language using suitable interfaces such as PLL, VPI, FLI, VHPI, and so on. However, HDL test-bench makes the simulation speed slow, and it is also difficult to describe hardware architecture and signals using C language. As these languages do not have any specific features to generate the test-bench, the test-bench generation time consumes a large portion of the whole SoC design flow. To overcome the limit of the conventional simulation approaches, many test-bench automation tools are developed and made available.

Regarding the test-bench automation tools following observations are made. Firstly, the automatic random generation of stimulus is effective in the finding corner cases. Constrained random generation function can be useful. Secondly, interoperability with major simulation tools and major hardware description languages is strongly desirable. Also connection with C/C++ is important for early application of functional verification. Thirdly, not only the signal level but also the functional level verification is required. Transaction-based verification feature is essential for the functional verification. In addition, coverage analysis feature is needed to measure the completeness of verification.

OpenVera[12] is a functional verification language which provides automatic stimulus generation and coverage analysis feature. TestBuilder [13], which is an open source C++ library, has classes to describe hardware and constrained random generation, etc. It focuses on transaction-based verification. e-language used by Specman Elite readily provides the feature of test generation, data checking and coverage analysis block.

C. IP-wise Emulation

As the complexity of IP blocks increases and corresponding test-bench automation tools accelerate the procedure of generating large amount of test vectors, IP-wise emulation is rising as another key trend. Emulation not only gives confidence on working design, but also provides faster verification speed. But in general, long set-up time, difficulty of debugging and extremely high cost make its usage quite limited, i.e., only as a go/no go test in the final design stage.

Recently developed low-cost emulation solutions using state-of-the-art FPGA's attack these weak points. Since they use only one or a few large capacity FPGA's, classical design partitioning overhead in compile-time can be significantly reduced and the communication overhead among FPGA's in run-time can be also minimized, thus resulting in faster operating speed. In addition, recently announced advanced features of FPGA's can be utilized, e.g., clock buffers, PLL, memory modules, special I/O pads, etc.

There are various ways of using the emulation equipment in IP verification, which can be classified according to how the verification environment is connected to the emulation equipment. One way is to run the emulator with the test-bench model running in the host computer. With this scheme, the design under test (DUT) mapped in the emulator communicates with the test-bench in clock cycle accurate mode or transaction mode. The former is as accurate as conventional cycle-level simulators but the operating speed is slower than the latter due to the synchronization overhead. The latter is 100 ~ 1000 times faster than the former but requires transactors that translate the corresponding bus cycles of an IP to each transaction and vice versa. Another way is to connect the emulator with the actual prototype board. This enables the verification of the IP in real operating environment.

Major interest in user's point of view is how to get easier debugging environment with lower investment. Also, integration with existing HDL design tools and flexible API structure are required [14]. Efforts are being made to create a standard for accessing various emulation equipments with unified API function sets from PC or workstation platform [15].

III. SOC INTEGRATION AND VERIFICATION

The previous section covered the tools and verification methods for SoC IP. Once all the IP's in the SoC design are proven to be functionally correct, the next step is to integrate them with the SoC platform which is typically composed of a processor, pre-defined bus system, and pre-verified bus components. This section summarizes the verification solutions for integrated SoC design.

A. Processor Model Selection

Since the bus systems nowadays are almost processor-centric, the focus is naturally centered on the processor model. As the processor model is now widely used for the co-simulation and the co-emulation rather than the architecture exploration as in the past, it has become important to balance the performance with the flexibility, accuracy and observability. The type of processor models can be classified as follows.

1) Native code

The target software is compiled for the host machine and executed on the host. Although running the native code is the fastest method for the functional verification, it is inaccurate and not suitable for measuring the target performance. Moreover, it cannot support the assembly code. To simulate I/O accesses, either API codes are inserted or exception handlers are used.
2) Instruction set simulator (ISS)

Interpretive ISS executes the simulation loop which has Fetch, Decode, Dispatch and Execution sequence, while keeping all the status of the target processor. Since it is easy to implement, flexible to use and accurate enough, almost all commercially available simulators use this model. However, it suffers from performance overhead, because the instruction fetch and decoding procedure is repeated for every instruction.

To enhance the performance of the ISS, compiled code approaches translate the target instructions to the host instructions statically or dynamically. Static compiled-code method limits the flexibility, but can yield higher performance. Dynamic compiled-code method can reform flexibility [16].

3) Actual processor model

Actual processor executes the target code under the control of the host machine utilizing its debug agent. It is fast and accurate, but additional hardware and interface layer is required. Nowadays commercially available configurable processors that combine processor and FPGA can be used for this purpose.

B. Hardware/Software Co-simulation

The term 'co-simulation' is widely used in various research areas as models belong to and representing different domains are simulated together. In this paper, 'co-simulation' is limited to the simultaneous simulation of hardware model and software model. For the software model, one of the processor models described in the previous subsection can be used. For the hardware modeling, many approaches based on high-level languages are proposed as a complementary or extension to existing languages in academia, with Verilog and VHDL still dominant in the industry.

Co-simulation in this context requires connection between processor models and HDL language models. The most common way to link them is using inter-process communication channels as shown in Fig. 1. The co-simulation stub in the center not only provides communication channel, but also keeps memory coherency between ISS and HDL simulator.

![Fig. 1. Typical co-simulation environment; ISS simulates software code and HDL simulator runs hardware part. Debugger controls the execution of ISS, e.g. run, stop, single-stepping.](image)

The biggest problem in most co-simulation tools is the slow simulation speed. One way to improve the simulation speed is to use more abstract processor model such as compiled code ISS or native code with a trade-off in accuracy and flexibility. Some co-simulation tools provide a trade-off between performance and accuracy by excluding less important memory transactions in the co-simulation stub. But in most cases, the bottleneck is in the performance of HDL simulators. The alternative way to improve the speed is to utilize the emulator instead of HDL simulators.

C. Co-emulation

In comparison with the traditional emulation, co-emulation means collaboration of emulator with the software part running in the host computer. The software part may be an HDL simulator, an ISS or even higher abstraction level models. Most of the emulation equipments have features to link the design in the emulation box with the HDL models but do not support direct linkage with the higher level models. As the emulator runs very fast, the running speed of ISS and memory access along with synchronization overhead can become new bottlenecks. Large expensive emulators provide the solution shown as in Fig. 2 (b), in which software part is also emulated in real processor module.

![Fig. 2. Co-emulation topology; (a) runs software part in host computer and hardware part in emulation box synchronizing at every cycle or every instruction. In (b), software part is also emulated in real processor module.](image)

The difficulty of debugging is another important drawback of the emulation approach. Recent advances in debugging features include the followings.

1) Built-in logic analyzer

Some emulators have embedded logic analyzer feature inside the reconfigurable devices in the emulator, thus eliminating the need for complex wiring to the external logic analyzer. Trade-off is possible among the number of probing ports, sampling depth, and sampling frequency.

2) RTL debugging

Since emulation is based on the gate-level design generated automatically by the synthesis software, matching the symbol names in gate level with those of the RTL design is very difficult, making the debugging very difficult. To solve this problem some products automatically resolves gate level symbol names into RTL symbol names.

3) Restart

This feature enables hot-swapping from emulation mode to simulation mode at a specific time slot. It is possible to run emulation at high speed until a specific time before a design bug appears, and then run simulation from that time.

IV. METHODOLOGY

Each verification technique has its merits and demerits. It is necessary that well-established verification flow is set up before the design activity begins. Conventional functional verification flow typically focuses on simulation-based approach while emulation is optionally applied near the end of the tape-out for design confirmation or for software development. In platform-based SoC design, it is quite desirable that intensive IP-wise verification is followed by early emulation/fast-prototyping system.

In this context, it is required to hasten the emulation set-up before the RTL design is matured as in Fig. 3. In Fig. 3, blocks in dark rectangles are algorithmic or behavioral-level model. First we start with a C code representing the target
software algorithm. We can download the software code and documentation, analyze them, and modify the code to meet the required system specification (Fig. 3(a)). Next, the code is partitioned into several processes or programs, each of them representing the software parts or hardware IP’s (Fig. 3 (b)). In this level, a rough estimation of the architectural aspects considering the communication among each module can be made assuming implementation style of each module. Now before getting down further to the implementation of each module, the emulation environment can be set up using SoC platform and suitable transactors (Fig. 3 (c)). Transactors connect models of various abstraction levels and heterogeneous design languages with the platform-specific bus system. EDA vendors will provide several different types of transactors, for example, a transactor between AHB and C model and a transactor between AHB and cycle-level HDL models. It is also possible to connect the transactors with the test-bench automation tools such as TestBuilder or Vera.

![Algorithm](image)

Fig. 3. Gradual refinement from (a) the C algorithm to (b) functional block model, and finally to (c) platform-level model using emulation environment composed of available IP's, transactors and target bus system.

Once the emulation platform is set up, each block is refined to implementation model as shown in Fig. 4. The C model for hardware part in Fig. 4 (a) is refined to HDL model manually or automatically. Since the interface of HDL modules generated from the C code is not matched with the bus interface, the transistor is replaced with suitable type (If the interface of HDL module is compatible with the bus, the transistor may be removed.) The C model for software part is refined to be compiled for the target processor and simulated in ISS, or if possible, executed in actual processor module.

![C to HDL](image)

Fig. 4. A process for the refinement of (a) C model to (b) HDL/ISS model and to (c) EDIF/real processor for hardware and software part respectively.

V. CONCLUSIONS

In this paper, we have surveyed verification tools and methodologies for SoC, focusing on planned reuse of IP and pre-verified platforms along with the co-development of hardware and software. Since well-designed SoC platforms can be reused and gradually enhanced, platform-based SoC design activity can be divided into the SoC IP design and integrated system verification.

For IP verification, formal methods are first used when applicable and suitable to statically remove the design bugs and improve the coverage, and test-bench automation tools are applied to test the IP with realistic test vectors. Finally all the IP's are mapped in FPGA in the emulator to be verified in the real operating environment. For integrated system verification the emulation environment is set up as soon as the platform is selected and the block-level partitioning is done. Well-established emulation platform helps progressive refinement of newly added SoC components and early development and verification of the software. Powerful debugging features handling both hardware part and software part are also required.

REFERENCES

[4] Axis Systems, "In-Circuit Verification Unifying Simulation and Emulation"

216