ARM SoC Verification Matrix Improves HW/SW Co-Verification

Jason Andrews

Verisity Design, Inc., Mountain View, USA

Abstract

Verification efficiency is the latest topic being discussed among engineers and EDA vendors. In order to work smarter, engineers can make improvements to the overall verification process, by automating best practices rather than focusing on incremental speed improvements in individual point tools.
This article describes how engineers doing ARM SoC verification can be more efficient through automated process solutions based on a single, reconfigurable verification system, applications, and a unified system verification methodology to allow engineers to execute hardware and software tests with a flexible mix of performance and debugging.

The State of Verification

Verification efficiency is the latest topic being discussed among engineers and EDA vendors. Engineers are wondering how to leverage all of the point tools that have been developed to solve specific issues to create a single, cohesive system verification methodology for hardware and software verification.

In order to work smarter, engineers can make improvements to the overall verification process, by automating best practices rather than focusing on incremental speed improvements in individual point tools. In addition, engineers can improve in one of the three areas that take up their time during the verification process:

Verification environment creation is the time spent to construct the environment, including testbenches, testcases, models, etc. This process is mostly manual with some automation in the area of testbench generation.

Execution is the time spent to run the test scenarios. Increasing raw performance is the primary way to run the test scenarios in a shorter period of time.

Interpreting results and debugging is the time spent to decide if test scenarios are working and how to find and fix problems for scenarios that are not working. This is also a mostly manual process with some automation in the area of functional coverage.

This article describes how engineers doing ARM SoC verification can be more efficient through automated process solutions based on a single, reconfigurable verification system, applications, and a unified system verification methodology to allow engineers to execute hardware and software tests with a flexible mix of performance and debugging.

The Three Components of HW/SW Co-Verification

A Verification Process Automation (VPA) solution for SoC verification integrates pre-proven best practices, automation and analyses to simplify verification. VPA spans all stages of the SoC verification process, from the block/unit level, to the chip/system level and to the project level. Specifically for HW/SW co-verification, a unified system verification methodology (SVM) must provide not only best-in-class point tools in each area, but also complete interoperability between them. Three components of HW/SW co-verification are:

1. Verification platform
2. Hardware verification tools and techniques
3. Embedded system software testing and debugging tools and techniques

Verification Platform

The verification platform is the method used to execute a description of the hardware design. It has other common names, such as execution engine or virtual prototype. The hardware design process consists of describing the hardware using one of the two common hardware description languages, Verilog or VHDL, or a high-level modeling language, such as SystemC. This HDL representation of the hardware design can be executed using any number of platforms or execution engines.

Four distinct methods used for the execution of the hardware design that have been identified and are commonly used in SoC design:

- Logic Simulation
- Simulation Acceleration
- In-Circuit Emulation

- Hardware Prototyping

Each hardware execution method has specific debugging techniques associated with it – each with its own set of benefits and limitations. The methods range from the slowest execution method, with the most debugging, to the fastest, with less debugging.

Throughout this paper, the following definitions are used:

Software Simulation refers to an event-based logic simulator that operates by propagating input changes through a design until a steady state condition is reached. Software simulators run on workstations and use Verilog or VHDL as a simulation language to describe the design and the testbench. Some portions of the hardware may be more abstractly modeled using a high-level language, such as SystemC.

Simulation Acceleration refers to the process of mapping the synthesizable portion of the design into a hardware platform specifically designed to increase performance by evaluating the HDL constructs in parallel. The remaining portions of the simulation are not mapped into hardware, but run in a software simulator. The software simulator works in conjunction with the hardware platform to exchange simulation data. Removing most of the simulation events from the software simulator and evaluating them in parallel using dedicated hardware increases performance.

Emulation refers to the process of mapping an entire design into a hardware platform that performs parallel processing to increase performance. There is no constant connection to the workstation during execution, and the hardware platform receives no input from the workstation. By eliminating the connection to the workstation, the hardware platform now runs at its full speed and does not need to wait for any communication.

In-Circuit refers to the use of external hardware coupled to a hardware platform for the purpose of providing a more realistic environment for the design being simulated. This hardware commonly takes the form of circuit boards, sometimes called **target boards** or a **target system**, and test equipment cabled into the hardware platform. Emulation without the use of any target system is defined as **targetless emulation**.

Hardware Prototype refers to the construction of custom hardware or the use of reusable hardware (breadboard) to construct a hardware representation of the system. A prototype is a representation of the final system that can be constructed faster and is available sooner than the actual product. This is achieved by making tradeoffs in product requirements, such as performance and packaging. A common path to a prototype is to save time by substituting programmable logic for ASICs.

Hardware Verification

Hardware verification describes the tools and techniques used to decide if a hardware design is operating correctly. An entire industry has emerged to provide products that augment the verification platform to help engineers develop test scenarios and interpret the results. Special verification languages have been designed to improve efficiency and verification quality. Other commonly used tools are code coverage, lint tools, and debugging tools to visualize results.

To rapidly compose the system verification environment, a successful methodology enables reuse of the components. In addition, a successful methodology supports a variety of software connection techniques and switching between them as well as the ability to apply it to front-end transaction-level analysis and post-silicon validation.

Embedded System Software

The final test for the hardware design is to correctly run the embedded system software. Even if the hardware can successfully run all of the embedded software, there is no guarantee it is bug free, but it does indicate a fairly healthy design. HW/SW co-verification is the best way to operate more efficiently by making sure all of the software works with the hardware before the hardware design is fabricated. Co-verification provides two primary benefits:

1. Software engineers have much earlier access to the hardware design. This allows software designers to develop code and test it concurrently with hardware design and verification. Performing these activities in parallel shortens the project schedule, compared with the serial method of waiting for the prototype to begin software testing. Moreover, the early involvement of the software team results in a much better understanding of the underlying hardware operation.
2. Co-verification provides additional stimulus for the hardware design. In fact, it can provide the true stimulus that will occur in the embedded system. This improves hardware verification when compared to using a contrived testbench that may or may not represent real system conditions.

By performing HW/SW co-verification, a wide range of problems can be found and fixed prior to silicon, such as register map discrepancies, problems in the boot code, errors in DMA controller programming, RTOS boot and configuration errors, bus pipelining problems, and cache coherency mishaps. Some of the errors will be software problems and some hardware-related. Addressing these issues must be done using a logical and well-conceived co-verification strategy.

Co-verification requires that accurate microprocessor models and software debugging tools be available to software engineers as early as possible. It also requires that the verification platform provide the best mix of performance and debugging for software engineers to work effectively with hardware engineers. The system verification methodology supports the easy transition to co-verification and between the various hardware modeling techniques, and adapts to the growing scope of system verification, including application software.

Five distinct types of embedded system software have been identified; the software content (that is, lines of code) increases with each successive step:

- System initialization software and hardware abstraction layer (HAL)
- Hardware diagnostic test suite
- Real-time operating system (RTOS)
- RTOS device drivers
- Application software

Matching the Software with the Platform

One of the main sources of confusion for projects is how to match the type of software being developed with the correct platform or execution engine. Figure 1 presents a diagram of this confusion. Given five types of software and three or four verification platforms, where should the connections occur between them? Which type of software should be run on each type of platform? Are all hardware platforms required?

A complete methodology must define for a specific project which combinations of software should be run on each type of platform to provide the highest quality verification in the shortest time.

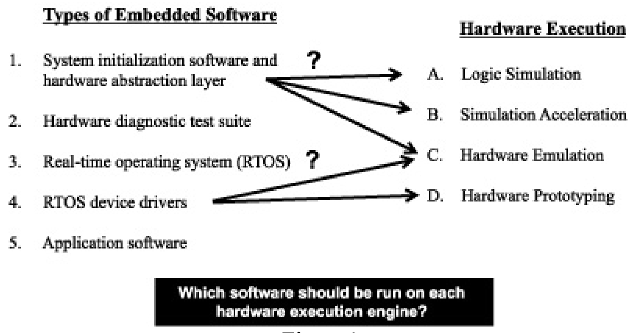
Types of Embedded Software
1. System initialization software and hardware abstraction layer
2. Hardware diagnostic test suite
3. Real-time operating system (RTOS)
4. RTOS device drivers
5. Application software
?
?
Hardware Execution
A. Logic Simulation
B. Simulation Acceleration
C. Hardware Emulation
D. Hardware Prototyping
Which software should be run on each hardware execution engine?

Figure 1

Different Perspectives of Software and Hardware Engineers

Before discussing the specifics of methodology and tools, it is important to recall that software engineers view the world very differently from hardware engineers. Here is a brief review of the different perspectives of software and hardware engineers.

Software Engineer's View of the World

The programming model of the embedded system is the most important information for the software engineer. The programming model for a microprocessor consists of the key attributes of the CPU that are necessary to abstract the processor for the purpose of software development. As an example of a programming model, consider the ARM9E-S CPU.

The ARM9E-S implements the ARM v5TE instruction set that includes the 32-bit ARM instruction set and the 16-bit thumb instruction set. The details of the instruction set are an important part of the programming model. Also covered by the programming model are details related to the operating modes of the CPU, memory format, data types, general purpose register set, status registers, and interrupts and exceptions. All of these microprocessor details are important to the software engineer.

Beyond the microprocessor, software engineers are interested in the memory map for the embedded system. For a 32-bit address space, 4GB of physical memory addresses can be accessed. All embedded systems use only a subset of this physical address space, and the memory map defines the location in the address space of various types of memory and other hardware control registers. The memory map may also define what happens if addresses are accessed where no physical memory exists.

Commonly found types of memory in an embedded system are ROM to hold the initial software to run on the CPU, flash memory, DRAM, SRAM for fast data storage, and memory mapped peripherals. Peripherals can be any dedicated hardware that is programmable from software. These can range from small functions such as a UART or timer to more complex hardware, such as a JPEG encoder/decoder.

The combination of the microprocessor programming model, the memory map, and the individual hardware control registers form the software engineer's view of the embedded system. This information becomes the ultimate authority for all software development and is available in the form of technical manuals on the microprocessor, combined with the system specific memory map supplied by the hardware engineers.

Hardware Engineer's View of the World

Hardware engineers have a different view of the embedded system. Although the internal operation of the microprocessor is important to software engineers, the internal workings of the CPU are much less important to hardware engineers, and the bus interface is most significant. For the hardware design to work correctly, the logic connected to the microprocessor must obey all of the rules of the bus protocol. If the rules of the bus protocol are obeyed, the details of the software tasks are not important.

All microprocessors use some type of protocol to read and write memory. To the hardware engineer, the microprocessor is viewed as a series of memory reads and writes. These reads and writes are used for fetching instructions, accessing peripherals, doing DMA transfers, and many other things, but in the end, they are nothing more than a sequence of reads and writes on the bus. For years, hardware engineers have used a bus functional model (BFM) to abstract the microprocessor into a model of its bus. More recently, this has been described as transaction-based verification since it views the microprocessor as a bus transaction generator.

Co-Verification Methodology

The solution to implement a co-verification methodology for ARM SoC verification and to reconcile the different views of hardware and software engineers is to combine a single platform that provides logic simulation, simulation acceleration, and in-circuit emulation with application-specific solutions for co-verification and transaction-based verification. Consider as an example an SoC that includes an ARM microprocessor. As described in the previous section, hardware engineers are interested in bus transactions of the CPU. This requires a transaction-based interface that works well with the verification platform for use during logic simulation, acceleration, and emulation modes. Since it needs to be used with logic simulation and later with acceleration and emulation, it cannot be constructed such that it will be a bottleneck to overall acceleration and emulation performance. Software engineers require good CPU models and debugging tools. Additionally, the system verification methodology must accommodate these various views of the system for verifying the software itself, and the interaction of the software and the hardware.

For each of the five different types of software, they will prefer either a software model of the ARM CPU or a hardware model of the ARM CPU. The three primary verification platform execution methods combined with the three representations of the ARM microprocessor form the matrix of nine modes of operation shown in Figure 2.

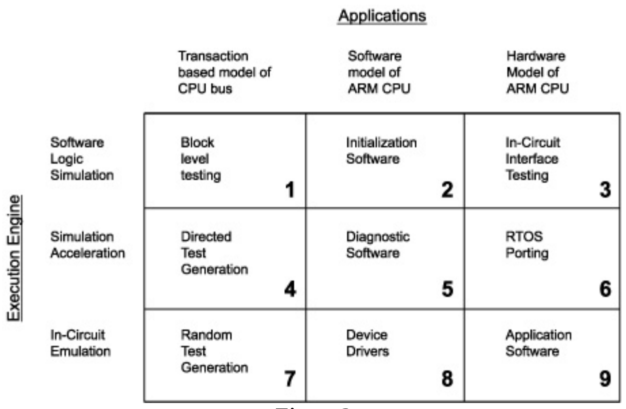
Applications
Transaction based model of CPU bus
Software model of ARM CPU
Hardware Model of ARM CPU
Execution Engine
Software Logic Simulation
Simulation Acceleration
In-Circuit Emulation
Block level testing 1
Initialization Software 2
In-Circuit Interface Testing 3
Directed Test Generation 4
Diagnostic Software 5
RTOS Porting 6
Random Test Generation 7
Device Drivers 8
Application Software 9

Figure 2

The next sections describe how each type of software can choose to be executed by either a software or hardware model of the ARM CPU using one or more of the platform's execution modes.

System Initialization and HAL Development

Many complex SoC projects use nothing more than a full-functional model of the microprocessor core in a logic simulator to write and debug this code. Software debugging with waveforms requires a true guru who understands hardware and software and can disassemble instructions in his head using instruction fetches on the data bus.

For the ARM SoC example, the ideal debugging solution for early development of system initialization and HAL code is one based on a cycle-accurate instruction set simulation model tightly coupled to a logic simulator containing the SoC hardware design. This provides interactive, graphical software debugging for the software engineer to single step through the code and verify register and memory contents with excellent flexibility and control. Simulation performance is less important because the code must be verified line-by-line, and the number of lines of code is relatively small. This situation is labeled as box 2 in the matrix in Figure 2.

Diagnostics
During this development of diagnostic tests, the logic simulator becomes the bottleneck of the verification environment. As tests run longer and the number of tests increases, it becomes more difficult both to verify the entire hardware design and to continue to run old tests as hardware and software errors are fixed. This phase is also the most crucial since it is where most hardware bugs are found. Debugging tools for both software and hardware at this stage are very important.
The best solution uses simulation acceleration to increase the simulation performance over what is possible using an ordinary software simulator. A simulation environment running at 10 to 100 Hz is not fast enough for engineers to run and test. Moreover, the memory optimization techniques commonly used by co-verification tools are not useful because the main purpose of the diagnostics is hardware verification. A simulation acceleration system that runs at speeds of 1 to 10 kHz is the ideal platform for simulation performance and debugging. The use of simulation acceleration with the software model of the ARM is labeled as box 5 in the matrix in Figure 2.

RTOS and Device Drivers
The initial RTOS port is a good place to take advantage of memory optimizations commonly used in co-verification, such as software memory models. These memory optimizations retrieve instructions at a much faster rate than using logic simulation. The result is less simulation detail on how the ARM SoC would work, but increased performance. Since the instruction fetch path is well verified, using the memory optimizations makes sense, rather than going back in the diagnostic test suite as a workaround for a low logic simulator. The initial RTOS boot requires box 5 on the matrix in Figure 2.
Once the RTOS is booted and stable with the selected device drivers, as shown in box 8, future work can be done using a faster execution method, such as in-circuit emulation. The number of hardware bugs is very small, so the increased performance is well worth any tradeoff in hardware debugging. This shifts the focus of the software engineers from box 5 to boxes 6 and 9.

Application Software
Application software requires the highest performance and possible stimulus from other sources, such as graphics, I/O interfaces like USB, or networking. This is an ideal fit for in-circuit emulation. Initial bring up for In-Circuit Emulation (ICE) is done using In-Circuit Simulation (ICS). ICS connects the software simulator with the target board by using the emulator as a pass-through connection to the target system. The necessary target boards, interfaces and test equipment are assembled in the lab for ICE. This represents a shift from box 3 to box 9 on the matrix in Figure 2.

Testbench Development
Hardware engineers are focused on making sure the bus interface logic connected to the microprocessor works correctly. The bus functional model (BFM) allows this to be done efficiently without requiring the overhead of a full functional model (FFM) and software to run on the CPU. There are many different kinds of BFMs available from IP companies, EDA vendors, and microprocessor suppliers. Unfortunately, all of them have been created using C/C++, verification languages or behavioral Verilog or VHDL. These languages are suitable for logic simulation, but are not efficient for simulation acceleration and emulation. The co-verification methodology requires a BFM that runs well for all phases of verification, from the start of a project, as shown in box 1 in Figure 2, moving to acceleration and emulation for directed and random testing, as shown in boxes 4 and 7. The methodology must also incorporate a hardware and software view of the set of status registers and an easily adaptable model of the system's physical address space. The hardware verification environment must increasingly consider the need for easy maintenance and reuse in subsequent system adaptations.

To achieve this flexibility, a transaction-based interface to synthesizable BFM for the CPU bus is ideal. By operating at the transaction level, the communication is minimized between the testbench and the verification platform. Using a synthesizable BFM and a transaction-based interface to the verification platform optimizes performance, while simultaneously allowing for the use of C/C++ or verification languages to create testbenches. A BFM that works the same way from simulation to emulation and provides the required performance, while simultaneously following the industry trend toward verification automation, is an important part of a unified verification methodology.

Matrix Coverage Is Not Enough

At first glance it may seem possible to build a good methodology to integrate hardware and software if all nine boxes on the matrix are covered. This coverage could be obtained by finding different tools for logic simulation, acceleration and emulation. Point tools can also be assembled for co-verification, bus models, and bus protocol checking. Of course, the cost to purchase all these tools and the time to evaluate all of them separately and deal with multiple vendors would be difficult. Unfortunately, the methodology would not be as strong as it could be because the tools do not work together. More than matrix coverage, what is required is interoperability of the boxes of the matrix. The ability to easily switch between boxes without changing platforms or totally rebuilding the environment is a key timesaver. Increasingly important is the flexibility to quickly change the register definitions to optimize the hardware/software architecture, and to move the physical address space to optimize system performance and complete the system verification.

Communication Gap

In addition to having a single system to perform all levels of verification, one of the major barriers that prevents a design team from performing co-verification is the lack of a common communication medium when a problem exists. Debugging a co-verification issue may at times take longer than running the test, because problem isolation involves multiple teams with different expertise working in a lab environment and viewing verification data in different formats.

Compounding the problem, software and hardware teams debug using different techniques and view the problem from different perspectives. The software team works with software models and debugs using software source-level tracing and memory and register viewing. The hardware team works with hardware design languages and debugs by viewing waveforms with history values associated with simulation times of read and write operations. As a result, when the software team detects a potential hardware problem, it cannot be described in hardware terms (time and signal value), nor is it easy to transfer an independent test case to the hardware engineers for further review. Much of this confusion can be eliminated by using an integrated set of debugging tools and models on the common verification platform following a unified system verification methodology.

Conclusion

The combination of a single platform that provides logic simulation, simulation acceleration, and in-circuit emulation with application-specific solutions for co-verification and transaction-based verification is a major improvement over the loose integration of many point tools being used by engineers today. Furthermore, combining many point tools to define a methodology is not a good solution, because the tools will not work well together. Interoperability between the platform and the application of transaction-based verification and HW/SW co-verification is essential to work smarter, not harder.