# Hardware/Software Co-verification & Co-Simulation

**Dr. Danny Rittman**
**danny@tayden.com**

**December 2004**

## Abstract

This paper presents hardware/software co-verification and co-simulation concepts and methodologies. Due to the facts that time-to-market challenge has increased the need for shortening the development process, new techniques and methodologies are introduced. Through the development of ASIC we adopted hardware/software co verification as part of our design process. A fashionable technique is to use co-simulation environments in the verification phase of a design process. As this technique is being adopted, and the fact that companies from a wide industry area are using it, real-time issues are becoming relevant. The market goal is to provide the software designers earlier access to an executable specification of the hardware for early low-level driver debugging. One of the major benefits of using co-verification is the collaboration of the hardware and software designers at an earlier phase of the design cycle.

## Introduction

As designs become more complex, standard co-verification techniques can not provide fast enough solutions for co-design issues. At the same time, many complex SOC (system-on-chip) designs are still using nothing more than a full-functional simulation model of a microprocessor and waveforms to debug complex hardware and software. A full-functional model of a microprocessor fetching and executing code in a logic simulator is not co-verification if the only means of debugging software are waveforms and assembly-language traces.

The main goal of co-verification is to get the entire system—hardware and software—working before the prototyping stage by providing better visibility into its behavior. Co-verification achieves its goals by providing two primary benefits. First, software engineers have much earlier access to the hardware design, which allows software designers to develop code and test it concurrently with hardware design and verification. Performing these activities in parallel save more time from the project schedule than 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. Second, co-verification provides additional testing for the hardware design. In fact, it can provide the true testing that will occur in the embedded system, providing better hardware verification than a contrived testbench that may not represent real system conditions. The increased confidence in the hardware design is invaluable.

There are very many traditional barriers to effective co-design and co-verification such as organizational structures and old fashioned paradigms of other companies in the same market or concepts developed in the past and worked well back then. Suppliers often lack an integrated view of the design process, too. What we need are tools which better estimate the constraints between the boundaries, before iterating through a difficult flow.

Using simulation models enable us to find conflicts between top-down constraints, which come from design requirements and bottom-up constraints, which come from physical data. Bottom-up constraints for software can only be realized in a hardware context because the abstraction-level of software is higher than that of hardware on which it is executed.

It is often the case that hardware is available (which is 'physical data'), so this can't be changed by software/hardware co-design. Only the software can be changed, and it should be fitted to this physical data. Therefore a certain modeling strategy is necessary to cover the existing hardware. This modeling isn't easy and it will never be perfect because the reality is too complex to find a perfect model. As to that it seems easier to design both hardware and software, because it is often easier to design two things that have to work together, than design one thing, and fit it around another. But if both hardware and software have to designed, powerful verification is essential because you have to design two different 'products' that interact with each other and nothing is 'physical' on both 'products'. Of course different techniques have been developed to verify combined hardware-software systems, but each of them has its own limitations. It's possible to run code on models of hardware emulated through dedicated programmable hardware, offering near real-time speed for code execution. Unfortunately, sometimes real-time interaction with other hardware and external environments is required, so full speed code execution isn't supported.

Hardware-software co-design exists for several decades. Ensuring system capabilities, designers had to face the realities of combining digital computing with software algorithms. Verifying interaction between these two prototypes, hardware had to be built. In the '90s this won't be sufficient because co-design is turning from a good idea into an economic necessity. Predictions for the future point to greater embedded software content in hardware systems than ever before. Changes had to be done in order to speed up and improve traditional software-hardware co-design. The developments had to focus in two major areas. One, top-down system level co-design and co-synthesis work at universities. Two, major advances made by EDA vendors in high speed emulation systems. Co-design focuses on the areas of system specification, architectural design, hardware-software partitioning and iteration between hardware and software as design progresses. Finally, co-design is complimented by hardware-software integration and tested. Also, design re-use is being applied more often. Previous and current generation IC's are finding their way into new designs as embedded cores in a mix-and-match fashion. This requires greater convergence of methodologies for co-design and co-verification and high demands on system-on-a-chip (SoC) density. That's why this concept was an avoided for many years, until recently. In the future the need for tools to estimate the impact of design changes earlier in the design process will increase.

To get a hold of elusive design errors, quickly applying the right modeling strategy at the right time is essential. It is often necessary to consider multiple models, but how can multiple approaches be fit into a very tight design process? This depends on the goals and constraints of the design project as well as the computational environment and the end-use. To find the right approach, iteration is the only way out.

Due to the fact that there is no widely accepted methodology or tool available to help designers create a functional specification, mostly unplanned manners are used, heavily relying on informal and manual techniques and exploring only few possibilities. There should be developed a hierarchical modeling methodology to improve this situation. The main concern in such a methodology is precisely specifying the system's functionality and exploring system-level implementations. Creating a system-level design requires some steps to be taken:

A. *Specification capture:* Decomposing functionality into pieces by creating a conceptual model of the system. The result is a functional specification, which lacks any implementation detail.
B. *Specification:* Detailed specifications to ensure complete coverage.
C. *Software and hardware:* The implementation of testing together as a complete system. Software and hardware apparatus is required, using software and hardware design techniques.
D. *Design Exploration:* Design Exploration of alternatives and estimating their quality to find the best result.
E. *Physical design:* Manufacturing data is generated for each component.
F. *QA* – Creating an entire QA mechanism for complete testing to ensure required results.

When successfully run over the steps above, design methodology from product conceptualization to manufacturing is roughly defined. This hierarchical modeling methodology enables high productivity, preserving consistency through all levels and thus avoiding unnecessary iteration, which makes the process more efficient and faster.

Describing a system's functionality, its functionality should first be decomposed and relationships between the pieces should be described. There are many models for describing a system's functionality. The next are the most typical models.

A. *Finite-State Machine (FSM).* This model represented as a set of states and a set of arcs that indicate transition of the system from one state to another as a result of certain occurring events.
B. *Data-flow graph.* A data-flow graph decomposes functionality into data-transforming activities and the dataflow between these activities.
C. *Communicating Sequential Processes (CSP).* This model decomposes the system into a set of concurrently executing processes, processes that execute program instructions sequentially.
D. *Program-State Machine (PSM).* This model combines FSM and CSP by permitting each state of a concurrent FSM to contain actions, described by program instructions.

It is important to note that no model is perfect for all classes of systems. The best one should be chosen, matching closely as possible the characteristics of the system into the models. This procedure should be done very accurately due to the fact that the choice of a model is the most important influence on the ability to understand and define system functionality during system specification.

Specifying functionality, several languages are commonly used by designers. VHDL and Verilog are very popular standards because of the easy description of a CSP model through their process and sequential-statement constructs. Other languages are used as well but none of them directly supports state transitions. Just like some models are better suitable for specific systems, some languages are better suitable for specific models than others.

Hardware/Software co-verification tools permit software to be executed on a hardware design, while the hardware design is being simulated in a HDL simulator. These tools are available from the major EDA vendors. On the hardware side, the HDL simulator is used to run, control, and debug the hardware design; on the software side an embedded debugger is used to display, and control the execution of the software. Using a co-verification tool also requires a model of the processor in the simulated design. These processor models are generally available from the EDA vendor, but in several cases are now available from the silicon vendors. All co-verification tools achieve reasonable levels of performance, with respect to the software, by hiding bus transactions from the logic simulator. Bus cycles modeled in the logic simulator run at logic simulation speeds, about 10 cycles per second while hidden cycles can be processed at around 100,000 cycles per second. The “hidden” bus cycles are generally uninteresting activity that does not impact the operation of the hardware, such as instruction fetches and software data space references. When looking into co-verification tools, the technology is still evolving.

## Concepts and Methodologies

EDA vendors are in constant race to improve the tools capabilities and performance to match a wide variety of designs. One of the most important features of a co verification tool is the ability of the tool to handle complex designs. Another concern is handling properly operation with “hidden” bus cycles. Without being able to take advantage of this cycle “hiding” the performance of the simulations would be too slow to be useful.  Hidden transactions are processed against a memory array that is not contained in memory instances the logic simulator. For example the tools V-CPU from Summit and Eagle-I from Synopsys effectively approaching this problem by partitioning the memory space of the system into software and hardware regions. Accesses into the “software” memory regions are “hidden” and accesses into the hardware regions are run in the logic simulator. Mentor’s tool, Seamless-CVE has a similar concept of hardware and software memory, but stores the data in what they call a “memory image server” for both hardware and software regions. The obvious benefit of Mentor’s approach is that the hardware and software partition can be changed while the simulation is running. There are a couple of not so obvious benefits that result from this difference as well. The main benefit is debug visibility. With V-CPU and Eagle-I the memory partitioning limits debug visibility. The hardware simulator can only see, and therefore can only debug, the hardware partition of the memory.

Likewise, the software debugger only has access to the software partition. With Seamless CVE, both the logic simulator and software debugger have visibility into the entire memory space of the system being simulated. The other benefit of Mentor’s technology is the ability to turn off the cycle “hiding.” The architecture of the memory image server allows the partitioning of memory regions between hardware and software to be changed. In fact, it can be changed during a simulation run. By changing this partition so that all memory is defined as hardware memory, you can turn off the cycle hiding aspect of the simulation. This means that all bus cycles will be driven through the logic simulator. Effectively, the co-verification processor model is turned into a full functional processor model for a portion of the simulation. This is critical during hardware operations where the presentation of events from the processor is sensitive to timing. Mentor’s technology has its share of drawbacks. Seamless CVE is a bit more complex to setup, when compared with its competitors. This extra setup is required by the memory image server. Another annoying limitation of Seamless CVE is its inability to run across the network; that is run the software debugger on a Pentium based PC while the logic simulation runs on a workstation, with communication across a network.

Software programmers have a few tools and methodologies to develop and debug embedded software. A standalone ISS (Instruction Set Simulation) can be used to run compiled code locally on a host workstation or PC. Device drivers and other routines that interact with the hardware must be stubbed out, or the

hardware must be emulated within a debugger macro language. Two disadvantages of this approach are the limitations of the macro language, and the accuracy of the implementation of the macros. An evaluation board that contains the target CPU is often used, and has the advantage of real time performance. Its disadvantage is that its hardware resources are general purpose and bear no or little resemblance to the final product. An FPGA prototype can be created to mimic the hardware to be deployed, but this is a complex undertaking, especially for designs that consume multiple FPGA's.
One solution to accurate hardware/software verification is to use the ISS of the target CPU and "connect" it to the hardware simulator being used by the hardware design group. One obvious disadvantage of this is that the software execution is limited to the speed of the hardware simulator. The Seamless. Co-Verification package from Mentor Graphics increases the speed of the ISS-Hardware Simulator "connection" by allowing most of the ISS instruction cycles to run decoupled from the hardware simulator. This patented technology termed "optimizations" has been used to generate successful Silicon on Chip (SoC) tape-outs, as well as CPU based board designs.

Another interesting tool is an RTOS (Real Time Operating System) simulator. An RTOS simulator does not emulate the instruction set of a CPU; instead it models the resources of the RTOS itself. This allows the programmer to develop and debug task level operations. The RTOS simulator is a higher level of abstraction than an ISS. It is CPU independent and does not require (or allow) assembly code. It is possible to "connect" an RTOS simulator to the hardware simulator through Seamless. At this level of abstraction, it is possible to observe the threads of execution, and how they interact with the hardware. The effect is the appearance that thousands of software cycles have run in conjunction with the hardware in essentially zero time. In other words, the RTOS can be initialized, application tasks started, and the software ready to interact with the hardware before the hardware simulator has advanced. Once in this state, the hardware will be initialized by the RTOS application, and hardware interaction begins. The software can now perform system level transactions with the hardware. This test environment is not concerned with CPU instructions, it will be used to exercise high-level operations in hardware and software; its performance will be bounded by the amount of hardware simulator time needed to perform a given software or testbench request.

The verification process occupies an increasing part of the total development time especially for real-time systems and today it is often the bottleneck in the development process. The increase in time for the validation stage is mainly dependent on four parameters:
A. Increasing of software complexity
B. Increasing of hardware complexity
C. Complex and high frequency of interaction between hardware and software,
D. The desire to produce a "correct at first time" design.

Shortening the development process it is a key demand to decrease the verification time. The new EDA tools particularly for developing application-specific circuits (ASIC) have drastically reduced the design time. Today the verification time is the bottle neck in the development process for ASICs. Both in the software and hardware design processes the verification time is over 60-80%. In today's design processes the verification phase has become the major part, not only because it is time consuming but also due to the increasing complexity in both hardware and software. The partitioning of functionality in both hardware and software also increases the frequency of interaction between them. These parameters among others make co-verification an important factor in the design process. Today there are different methods for co-verification, but the goal, which is to verify software and hardware execution respectively, remains the same.

Without target hardware, verification of software code is today mostly done using cross development tools (compilers, debuggers, simulators, etc.). Whenever a peripheral hardware component is to be accessed, a piece of code simulating the component is executed instead. Thus, at least verification of the functional behavior of the software can be achieved. Another method for software verification is to use the more realistic approach which uses existing hardware, typically implemented as a prototype. This method also enables a more timing accurate verification. The major disadvantage is that software is verified late in the design process. On the hardware side typically one is interested in verification of the interaction (accesses, handshaking, interrupts, signals, etc.) between software and ASICs and other system components. (Figure 1) ASICs are typically designed at register transfer level (RT). This level represents a complete functional model of the ASIC. The model must be verified in detail to demonstrate correct functioning together with the surrounding components and the software. One approach to achieve this is to use testbenches. In a testbench model the ASIC to be verified is instanced as a component. Using models of the surrounding components (e.g. CPUs, RAM, I/O, etc.) can provide input, thus enabling verification of the responses according to specification. Typically this is simulated on a workstation, often at a slow simulation speed if the designs are large. Consequently, simulation of software execution is a slow process which makes it difficult to simulate a complete program. The end result is fairly long time consuming procedure. Although we are evidencing an improvement in EDA tools co-simulation and co verification capabilities, a major improvement is needed.

After verification at the RT-level has been performed, the verification process typically continues on a fast prototype. Today a fast prototype is implemented in either a FPGA (Field Programmable Gate Array) or in hardware emulator (typically uses FPGAs). Enhancing speed of the hardware simulation is typically done on a workstation, enabling faster execution of software and complete programs can be verified. The major advantage when using FPGAs is the ability to make changes to a design very quickly compared to the traditional ASIC fabrication. While the emulator preserves the visibility into a design, the use of FPGAs only has limited visibility. Viewing internal signal states in a FPGA can be easily done via routing the signals out to external pins. One disadvantage in the FPGA/emulator technique is that the timing is much slower in comparison with that in the ASIC. Also, both emulation and FPGA are relatively expensive to use. Co-simulation for verification has recently been introduced as an alternative to testbenches and in some cases to fast prototyping. In fact, the idea of co simulation was derived from using testbenches with processor models. The idea of the new method is to have real software execution as the event driver in a testbench and also to reduce the impact of software simulation time in the traditional testbench. An engine models the CPU which is instanced by a testbench (typically using VHDL or Verilog).

There are different methods used to run the CPU engine, but the overall technique in common is to conceal it from the processor's interfacing to the hardware. Figure 2 illustrates a schematic overview of the connection of the hardware with the software through a controlling unit, a so-called co-simulation kernel.

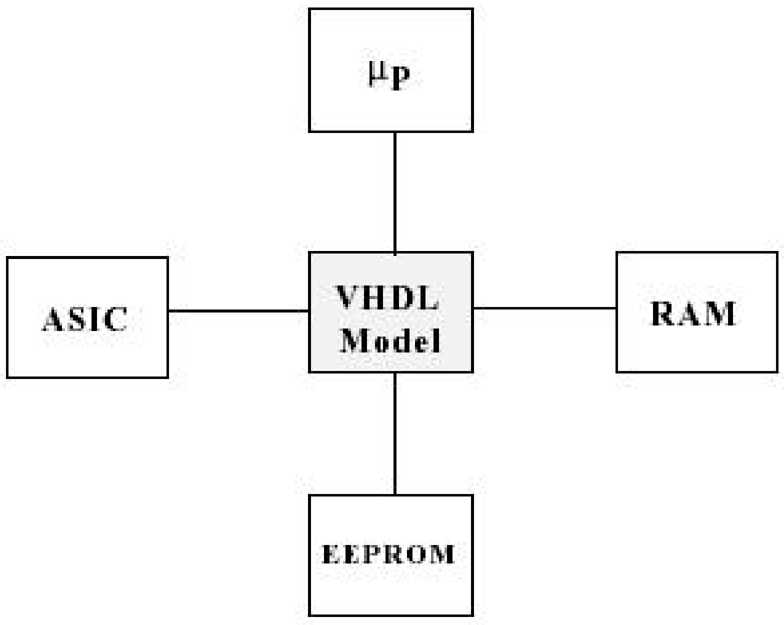

Figure 1: Testbench for a System Simulation

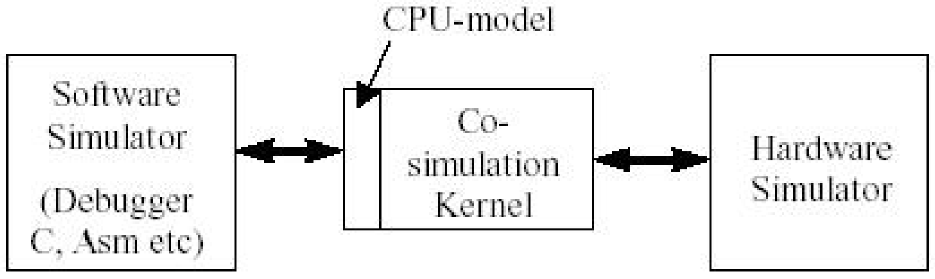

Figure 2: Co-Simulation Environment

Today there are two major commercial tools available, EagleI from Synopsys (Originally from ViewLogic), and Seamless from Mentor Graphics. They are very similar but they use different techniques. An overview of the techniques used in these tools is presented below.

**Seamless Co-Verification Environment**

The Seamless environment enables software and hardware development to be parallel activities, removing the software from the critical path, and reducing the risk of hardware prototype iterations resulting from integration errors. In Seamless, the processor's functionality is separated from its interface. A Bus Interface Model (BIM) simulates the input/output pin behavior for the hardware portion of the simulation. The software portion executes as a separate process, allowing much faster execution, either on an Instruction Set Simulator (ISS) or as Native Compiled Software (NCS). The ISS executes machine code produced by cross-compilers for specific processors. NCS is software compiled for execution on the host machine. Communication between SW and HW is controlled by the co-simulation Kernel (CSK). Figure 3 shows the architectural structure in which Seamless operates. Supported ISSs and BIMs, respectively, are developed for the most popular processors on the market. Examples on processors include the x86 family, 68k, and the PowerPCs. These processors are not always fully modelled and there are some limitations. Some of these limitations are the lack of (or reduced functionality in) caches and memory management units (MMUs). Apart from supported processor models, there are also different types of memory models available. These memory models have a particular connection to

Seamless which enables optimization of bus-cycles (generated by the BIM) for instruction fetches and data access. Supported types include SRAMs, DRAMs, FIFOs and register elements.

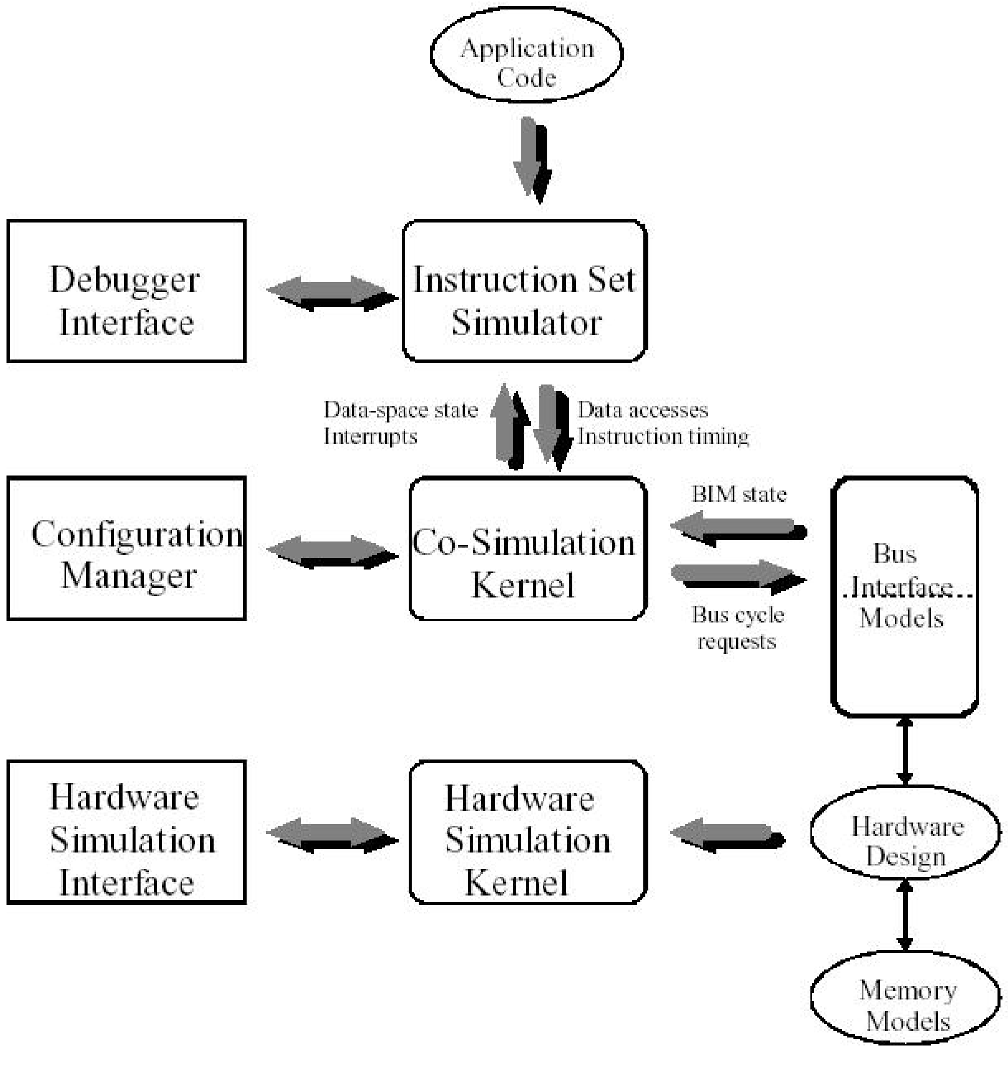

Figure 3: Seamless' Architecture
Image Source: Mentor Graphics

**EagleI**

Similar to the CSK in Seamless, EagleI uses the VSP (Virtual Software Processor) to control the co simulation. EagleI supports three different models, VSP/Link, VSP/Sim and VSP/Tap, each suitable at different stages in a design process. The VSP/Link model uses a technique similar to the NCS execution (see below) which also is the fastest model. Also here, the VSP/Sim model is like an ISS with a true cycle behavior. Not represented in Seamless' environment, is the VSP/Tap model which is a VSP that includes hardware in the form of an In-Circuit-Emulator (ICE). This technique is similar to the ISS but with the extension of a hardware accelerator. Using an ICE the visibility needed is kept, thus it's possible to investigate internal registers and memory.

**Native Compiled Software**

Simulation using NCS is the fastest method when compared with the ISS approach because software is run directly on the workstation. NCS is easily produced by compiling software coded in any high-level language. Thus debugging of NCS can be done using a standard workstation debugger (e.g. dbx). The connection to the hardware process is done by placing calls to the VSP/BIM through an Application Program Interface (API). This interaction only drives the VSP/BIM pins to their defined values and cycles. The modeled processors' internal registers and cache memory is not available in this approach.

**Instruction Set Simulator**

An ISS is a software application that models the functional behavior of a processor's instruction set. It runs much faster than a hardware simulation because it needs not to cope with a processor's internal signal transitions. Since it is machine code for the target processor that is executed, Users can use any language supported by the cross-development tools. The ISS reports the number of clock cycles required for a given instruction to the VSP/CSK. Notification of external events (e.g. interrupts, resets) from the VSP/BIM is reported to the ISS by way of the VSP/CSK.

## Conclusions

Co-Verification and co-simulation have significantly increased system's visibility. Verifying and Simulating software execution using an ISS on a cycle accurate model, timing information can easily be retrieved and be used as feedback when determining task execution flow. Co-simulation has shown to be suitable for verification of task switching, IRQ response time, and software access of hardware components. Due to the slow speed of hardware simulation it is difficult to verify large applications and complete systems. Thus, co-simulation will yet not verify all possible interference's between hardware and software. In many cases it surely will reduce design time, but as verification tool for ASIC design in large systems it will still not exclude the need of FPGAs for fast prototyping. Lack of functionality like caches and MMUs in processor models (Seamless') are still a problem. A model has to be as equivalent to the real hardware as possible, especially when it comes in use in embedded real-time systems. Today, more focus is on system-level design, viewing the total project as a complete system instead of the individual parts—chips, boards, chassis, and software. Corporations are looking for ways to shorten the project schedule and increase confidence in the design. At the verification level, this method translates into selecting the best tools to build a productive environment for both hardware and software engineers. A single integrated system for logic simulation, simulation acceleration, and system emulation, along with the concurrent debugging capabilities of both hardware and software, provides the best platform to get the project done in the shortest amount of time. The ability to easily transition between these models and modes of operation provides the greatest flexibility and control over the entire verification process.

## References

[1] R. Klein, “Miami, a Hardware/Software Co-verification System,” in Proc 7th IEEE Rapid Systems Prototyping Workshop, 1996, p. 173-177

[2] J. Wilson, “Hardware/Software Selected Cycles Solution,” in Proceedings of the 3rd International Workshop on Hardware/ Software Codesign 1995, p.190 194

[3] M. Stanbro, “Getting to the Bottom of HW/SW Coverification Performance Claims,” Computer Design, Vol 37 No 12, December 1998, p65-67 "Reproduction by permission of the International HDL Conference"

[4] J. A. Stankovic and K. Ramamritham, Tutorial ”Hard Real-Time Systems”, IEEE Computer Society Press, ISBN 0-8186-0819-6

[5] Electronic Engineer, Electronic Design Automation, September 1999

[6] Mentor Graphics, ”Seamless Co-Verification Environment, User’s Reference Manual”, 1998

[7] Motorola, ”PowerQUICC, MPC860 User’s Manual”, 2002.

[8] Mentor Graphics (Microtec), ”Introduction to XRAY Pro”, 1996