A Methodology for Virtual Hardware/Software Integration

by Serge Leef
Mentor Graphics Corporation

Abstract
Majority of the systems being designed today are embedded systems that consist of standard and custom hardware as well as standard and custom software. As the EDA vendors focused on tools and methodologies for automation of hardware design, the effects of the software as a component of the system have been largely ignored. While it is possible to execute microscopic amounts of software on the simulated hardware, the performance and usability of current generation of tools is grossly inadequate for true hardware/software co-verification. This paper examines common analyzes hardware/software co-verification problem, proposes a solution and explores methodology implications.

Introduction
Embedded systems are task-specific computing devices that consist of standard and custom hardware and software. Standard hardware is typically made up of a commercial microprocessor or microcontroller, memory and a small number of standard parts. Custom hardware is implemented as Application Specific Integrated Circuits. Standard software frequently consists of a Real Time Operating System (RTOS), and configurable device drivers. Custom software is the embedded application.

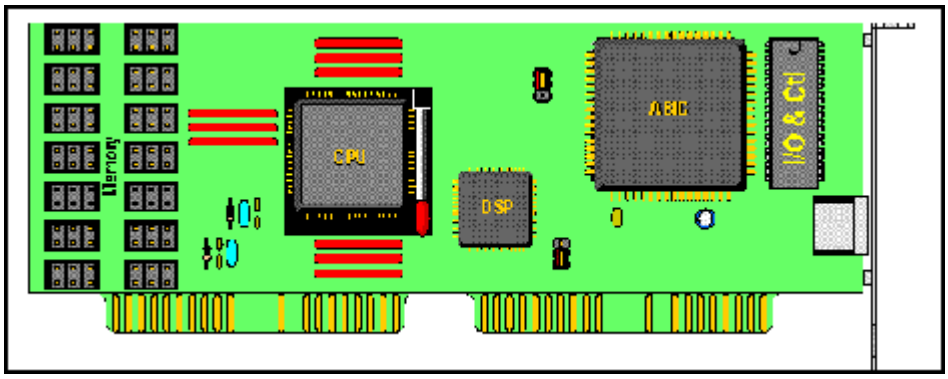

Figure 1. Structure of a typical Embedded System

The complexity of these systems varies widely from low to high end depending on the targeted market segment and product goals. Embedded systems can be found in almost everything that we encounter in our daily lives, such as communication systems ranging from the telephone, to the large switching centers, automobiles, consumer electronics, avionics, etc. In fact some of today's luxury cars contain more than 40 independent embedded systems, controlling such things as the suspension, engine, anti-lock brakes and user consoles [1].

System type	CPU	Memory	ASIC	RTOS	Embedded Application
Basic Consumer Electronics Device	Intel 8051	256KB	none	Simple Scheduler	10000 lines of assembly code
Automotive Control Sub-System	Motorola 68300	512KB - 1MB	20K gate FPGA	Custom microkernel	10000 to 50000 lines of assembly and C code
Telecom Switching Module	Multiple Pentiums and DSP processors	2MB+	Multiple ASICs at 30K to 100K gates each	Commercial RTOS	upto 2M lines of C and C++ code

Table 1. Range of embedded system complexity

Embedded System Types

Embedded systems can be software dominant. That is, standard or previously designed hardware is utilized while the software makes up all or most of the original content that is being developed in the course of a design project. These, typically cost sensitive systems, are quite common in the consumer electronics products. Software dominant systems do not present a hardware/software co-verification challenge: system can be validated by executing software on the existing hardware.

Hardware dominant systems are usually found in the applications where performance is a critical success factor. In these systems, much of the design team's efforts are focused on implementing the functionality in hardware (i.e. boards, ASICs). Hardware dominant systems are difficult to validate with software during the design cycle because the actual hardware does not yet exists and the simulation tools are too slow for the job.

Hardware/Software Co-simulation is a much needed capability for designers of the systems with significant original hardware content. The need for co-verification is frequently proportional to the size of the original hardware being designed.

Hardware/Software Integration

The time spent on the typical electronic system development project is typically partitioned into three basic phases: System, Hardware/Software Design, Integration and Test. In interviews with 18 customers, a striking consistency was observed in the relative duration of each phase: they were all about equal in duration, each representing a third of the length of the project.

Figure 2. Embedded system design process overview

- **System phase** -- during this phase the entity being designed is viewed as an overall system, rather than as distinct hardware and software components. This phase, typically completed by system engineers, results in specifications for how the system will behave. Other deliverables include the architecture and functional specification of the system. Additionally, functional requirements and budgets for both hardware and software components of the system are created. These include constraining costs, size, performance, and physical attributes. At the conclusion of this phase the system is partitioned into software and hardware sub-systems.

- **Hardware/Software Design phase** -- during this phase separate organizations (except in the case of small projects) address their respective problems. The Hardware and Software design and implementation efforts typically start at the same time and end at the same time. The work, however, proceeds independently between start and end. Bridging the gap between the two design teams are the firmware engineers. Firmware engineers normally develop low level software which interfaces to the hardware, thus providing a software foundation for the higher level software. The higher level, or application, software is where the unique functionality of the product is usually implemented (e.g., call-forwarding, engine control decisions.)

- **Integration and Test phase** -- In theory, Integration and Test is the final series of checks prior to the shipment of the system. In practice, it is the first time that the "completed" hardware and independently developed software come together as a system. At this time numerous issues surface, namely: the effects of misinterpretations of interface definition, out-of-date specifications, poorly communicated changes, and ineffective performance modeling, etc. Consequently, one third or more of the total development time is spent in this phase.

Faced with cost and schedule deadlines, developers are forced to redesign and/or lower product objectives. Given the long fabrication lead times and costs associated with redesigning ASICs, the rework is frequently performed in software, which is not always the best solution for the end product. Integration and Test becomes Redesign and Re-implementation, and takes about the same amount of time to complete as the original design and implementation.

Software change costs also tend to be less visible than the cost of an ASIC turn and so the eventual product may be compromised. Also in some cases, the first product release will not contain all of the intended software functionality because it has not been possible to start the integration effort earlier in the design. Two things are necessary before virtual integration and test can be accomplished. The first is the ability to simulate the hardware at speeds sufficient to make software execution a reality. In most cases, this means that the hardware simulation performance must be increased by a factor of at least 1000 over current execution speeds. The second is the need to bring the debug and development environments of the hardware and software closer together. No software engineer is going to be happy looking at waveforms when his development occurred in a high level language

Driving Forces

There are three primary driving forces affecting the market for hardware-software development tools: need to reduce time-to-market, increased software content, and increased design complexity. These are discussed below.

Need to Reduce Time-to-Market

There is a common belief among the electronic systems designers that early detection of design errors will dramatically reduce the amount of redesign/rework needed during Integration and Test. It has also been observed that the cost of testing during the development is small relative to the cost of rework in latter stages in the process. It seems to be generally true that investing energy in the front-end activities reduces downstream costs and results in a better overall product. This approach has worked in other instances, and is one of the foundations of the quality movement (do it right the first time, correct by construction, etc.)

Developers and management both view reducing the length of the Integration and Test phase as the most immediate way to achieve their improved time-to-market objectives. Methodology changes during Design/Implementation phase are widely seen as the most likely source for overall time-to-market improvements.

Increased Software Content in Electronic Systems

As the microprocessors and microcontrollers have become an integral part of embedded designs, the percentage of electronic manufacturers' R&D budgets allocated to software engineering has increased radically. It is estimated that the ratio of hardware to software engineers at most companies has reversed over the past 11 years, such that there are now about 2-3 software engineers for every hardware engineer. That means hardware is decreasingly the dominant factor in the time-to-market equation.

Design Complexity Renders Current Techniques Impractical

Hardware verification has been reasonably addressed by the tools available today. Since typical hardware verification calls for micro to millisecond simulation time frames, the necessary performance level matches up well with today's simulation technology. Current generation of simulation tools is approximately 7 orders of magnitude slower than real time, allowing only small portions of real-time activities to be simulated.

Embedded software requires a different level of performance. Verification of functionality of embedded software frequently calls for execution of seconds to minutes of real time. This means that software developers would have to wait up to 14 days to execute 1 second of real time at gate level simulation speeds.

Between hardware and software components of the system, the firmware is inserted. The firmware is typically made up of device drivers, kernels, diagnostics and boot code. The performance requirements for verification fall somewhere between the two extremes described above. It is clear, however, that current tools are about three orders of magnitude too slow for adequate firmware verification.

It is because of this gap between the performance needed and the performance currently available that the Hardware/Software Co-simulation market has not yet evolved.

Hardware/Software Co-simulation Tool

Based on a close examination of wide range of methodologies used in embedded system design, it appears that a right Hardware/Software Co-simulation tool could have a profound impact on a variety of critical success factors. Such tool would have to provide an infrastructure for virtual integration by supporting a broad range of modeling techniques. Additionally, dramatic performance gains over traditional hardware simulation tools would have to be delivered.

Observations

Numerous organizations have attempted to execute embedded software on simulated hardware, however, few have achieved satisfactory results. While it is possible to achieve a full system simulation, the resulting performance is usually grossly unsatisfactory for meaningful validation of hardware/software interactions. The presence of a microprocessor in the simulation session results in production of huge amounts of simulation events required to service interactions between it and the memory subsystem. This typically leads to performance levels of below 10 instructions/second.

When analyzing the operation of embedded systems, one can readily observe that most of the time is consumed by the CPU operation. CPU retrieves instructions from memory and manipulates contents of registers and memory locations. In most applications, the CPU spends upwards of 90% of its time on internal operations as interactions with external hardware are highly infrequent. It is not uncommon for the CPU to process 200 instructions before needing to exchange data or control information with external hardware. That means the external hardware is effectively idle most of the time.

If, on the other hand, one considers a simulation session that attempts to model an embedded system, a very different observation can be made about time consumption. The load on event-driven hardware simulation tools can be measured in terms of serviced events. When simulating an embedded system, we note that the majority of events produced and serviced by the simulator is directly related to the interaction between the CPU and the memory subsystem. Typically, over 90% of the time is spent on servicing events that perform CPU instruction fetches, reads, and writes to memory.

In other words, the precious simulation time is largely wasted on repetitive re-simulation of fundamentally the same operation. Thus, one can observe that if CPU to/from memory interactions could be serviced via higher performance methods, significant savings could be achieved in the simulation session. This would, in turn, enable the simulation tools to focus most of their energies on simulation of all-important interactions between CPU and external hardware (including standard parts and ASICs).

High Level Overview

Today, the hardware designer has a hardware simulator based on VHDL or Verilog modeling language, that can emulate the behavior of a given target design. The design is entered in a hardware description language, or through a schematic entry tool. The design consists of a set of component models and their connectivity. Often, the microprocessor or controller is modeled by using what is called a bus functional model. The bus functional model does not model the complete behavior of the microprocessor, only the different bus cycles that the processor can execute. The model is controlled by a script that directs it to drive a given set of bus cycles into the design enabling the hardware designer to construct a test that would, for example, write to and then read from each of the memory components in the design.

To execute software on a simulated design, the hardware designer would need a fully functional model of the processor (a complete behavior model). However, writing a program that completely emulates the behavior of a complex processor (such as Pentium) is an extremely complex task. To obtain a full functional model of a processor, a device called a hardware modeler is often used. A hardware modeler is a machine that contains much of the circuitry of a semiconductor tester and is interfaced to a hardware simulator. The hardware simulator passes to the hardware modeler the values on the input pins of the processor, the hardware modeler then drives these values onto the input pins of the actual chip plugged into a socket on the hardware modeler. The hardware modeler samples the output pins of the actual processor and returns these values to the hardware simulator. Modeling the processor in this manner usually results in speeds of 1 to 10 instructions per second on the simulated design. This method is commonly used in today's Hardware/Software Co-simulation efforts.

The software designer utilizes a compiler and a debugger on a general-purpose computer for software design and algorithm development. Often, an instruction set simulator will be used for running assembly and machine code and for gross estimation of software performance. These instruction set simulators often have facilities for handling I/O data streams to simulate to a limited degree the external hardware of the target design. Instruction set simulators run at speeds of ten thousand to several hundred thousand instructions per second, based on their level of detail and the performance of the host computer that they are being run on [2].

The hardware simulator and the instruction set simulator appear to provide an interesting opportunity for integration. A proposed architecture for a Hardware/Software Co-simulation tool is shown below.

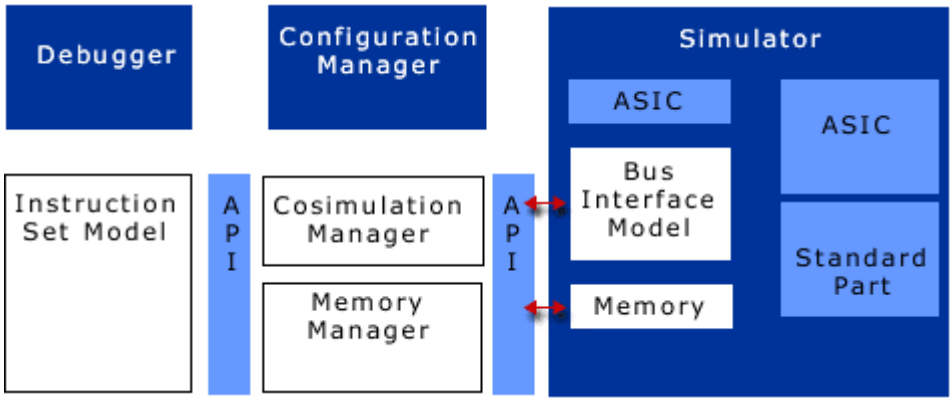

Figure 3. Proposed architecture for a Hardware/Software Cosimulation system.

The Co-simulation Manager coordinates the execution of embedded software on the Instruction Set Model with control of a Bus Interface Model instanciated in a traditional hardware simulator. The Memory Manager encapsulates a coherent view of the address space, contents of which are served on demand to all other modules in the environment. The Configuration Manager allows the user configure the environment for execution and to fine-tune run-time optimization and synchronization mechanisms.

Performance Improvement Opportunities

Performance is a critical factor for any potential solution to this problem. Today's simulation tools are grossly inadequate in this area. A simulation session that executes a power-up sequence for a mobile phone (about 3 seconds of real time) currently consumes 18 days of computer time. Several orders of magnitude of performance improvement need to be delivered in order for a Hardware/Software Co-simulation tool to have reasonable utility. That is, the 18 day run should be brought into the range of minutes of computer time.

In this section we examine how the performance improvements on software execution and hardware simulation sides of the environment can effect the overall throughput. The unaccelerated method for execution of software on simulated hardware is to load the object code into an array declared in an HDL (Hardware Description Language) module and to allow the models (including the model of the CPU) to execute on their own, fully simulating all the I/O activities. Usually the system is modeled at the Register Transfer Level (RTL) of abstraction and the resulting performance is assumed here to be the base or reference performance level (1X).

Software Execution

The techniques described below are distinct in the way they represent the design components and in the performance levels that they deliver.

Model Control Language

In this approach a subset of C language is used to direct a Bus Interface Model (a limited model of a CPU capable of translating processor states into I/O events on the physical pins) to execute one or more bus cycles. The directives expressed in C subset are translated into a series of events emitted by the Bus Interface Model over time. While the language subset allows conditional statements and loops, it is not an efficient vehicle for describing any high level processor behaviors.

The commercial implementation of this technique also suffers from lack of real-time bi-directional relationship with the simulation session. That is, the program is typically translated into some proprietary data file which is subsequently loaded into simulation. The program is incapable of responding to transactions that originate from outside the processor, making this technique suitable strictly for generation of stimulus for the surrounding hardware.

This approach does not result in measurable performance gains since true execution of software is not achieved.

Compiled Software Model

A mild improvement on the above method is known as Compiled Software Modeling (CSM). Instead of C subset, the entire language is supported. Here the embedded software is compiled for a host other than the intended CPU. As the software executes on the host the I/O transactions are trapped and selectively translated into bus cycles (as above) that excite the surrounding hardware. The presence of the Bus Interface Model is still required.

While this technique allows real-time, bi-directional interactions between hardware and software to be modeled, numerous problems persist. Since this approach takes advantage of the host's native compiler, the target processor's assembly language can not be supported. Transaction trapping frequently requires unwanted modifications to be made in the embedded software. And finally, a complete lack of timing makes this approach inappropriate for anything but basic protocol verification and stimulus generation.

The performance of software execution when utilizing this method can be quite impressive. Gains of 10,000X to 100,000X and beyond are possible depending on the type of host and complexity of the target.

Instruction Set Model

A more substantial investment in model development leads to a far more profound improvement in the utility of the co-simulation solution. In Instruction Set Model (ISM) the processor is described in terms of its instruction set. That is, a collection of instructions is modeled where each instruction defines a relationship between constructs

that are internal (registers, on-chip memory) or external (on-board memories) to the processor. Additionally, cycle level timing estimates are included. The accuracy of these estimates is directly proportional to the degree of model detail.

A performance gain of upwards of 10,000X can be achieved making this technique suitable for full-scale hardware/software interface validation and embedded software execution.

Hardware Simulation

While event-driven simulation tools have achieved relatively high levels of performance, they all face the same performance barrier: management of the event queue to service all transitions introduces a theoretical limitation on overall performance. That is, non-event-driven techniques must be examined to achieve significant performance gains.

Cycle-based Simulation A method exists that can achieve higher performance rates at the cost of reduced accuracy. Cycle simulation allows the simulator to abstract away the timing details for all transactions that do not occur on a cycle boundary. This eliminates a vast amount of computation and can lead to 10X to 100X performance gains over traditional simulation.

These tools have not been adopted widely. Cycle simulation puts significant strain on the design methodology without delivering performance level sufficient for execution of meaningful amounts of software.

Emulation Another technique that has been widely explored for this application is Hardware Emulation. In this approach, the design that would ultimately be implemented as one or more ASICs is mapped onto programmable hardware. The programmable hardware assumes the personality of the loaded design and realizes its execution at 1MHz to 10 MHz.

While the execution speed of Hardware Emulation is more than sufficient for embedded software debug (about 1,000,000X over traditional simulation), introducing companion elements of pre-existing hardware (such as CPU) can be cumbersome.

Hardware/Software Co-simulation

It is useful to consider the embedded system in terms of the following components:

- **Software Dominated Partition (SDP)** -- includes a COTS (Commercial Off-The-Shelf) processor, memory system, and all the software including RTOS (Real Time Operating System), device drivers, and the embedded application(s).

- **Hardware Dominated Partition (HDP)** -- includes the original hardware content of the design usually implemented as one or more ASICs.

The proposed solution allows the user to combine a chosen SDP executor with a suitable HDP simulation tool/model, while giving him/her the knobs to control performance/detail tradeoffs during the analysis session.

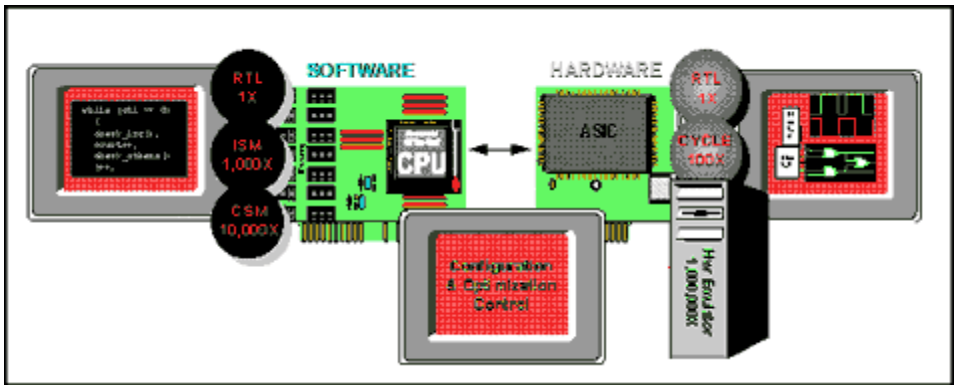

Figure 4. Hardware and software partitions with corresponding execution options

Most of the methods discussed above are supported in the proposed Hardware/Software Co-simulation tool. The selection of techniques needed for Software Dominated Partition would be based on availability and type of software development tools, access to Instruction Set Models, and validation objectives (interface, code, protocol, etc.) The selection of tools needed for Hardware Dominated Partition would be based on access to hardware emulation hardware, types of available simulation tools and validation objectives (ASIC, board, interface).

The figure below summarizes the expected performance levels that would result in different combinations of the discussed techniques.

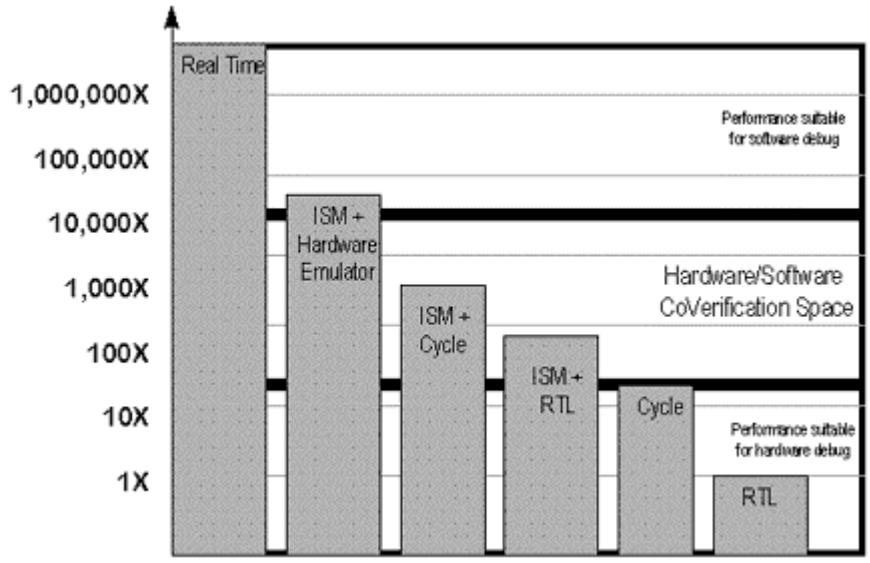

Figure 5. Combined performance levels

Managing the Tradeoffs

Accelerating major portions of the simulation environment is a good start, but is not sufficient by itself. Several key issues need to be considered as well.

Key Issues

Performance The techniques described above all offer different levels of stand-alone performance. It is, however, the combined performance that is important. For instance, it is irrelevant how fast the software side of the system executes if the hardware simulation is left unaccelerated. Even if the processor, memory and embedded code execute infinitely fast, the

overall performance of systems with high original hardware content (greater than 40K gates) can remain at unaccelerated level. When dealing with performance, it is crucial to address the acceleration of the entire system rather than that of components.

Accuracy While it is practical to abstract significant amount detail out of the analysis session, a certain degree of accuracy is important and must be maintained. Most interactions between hardware and software are sensitive to timing constraints. Without estimation of instruction timings, it is not possible to truly validate the H/S interface.

Usability It is pivotal to have accurate machine-specific view of the software execution in addition to the views traditionally provided by the hardware simulation tools. A common debug environment must offer either hardware simulation view of the system, software debug view of the system, or both depending on the user needs. Visibility of the processor states and ability to create composite breakpoints are important elements of effective hybrid system debug environment. One can easily envision the extension of the usage model to include virtual instrumentation as well as bridges to other analysis domains.

Model Availability All of the above approaches require some level of processor model. At least a Bus Interface Model is required for CSM method. An ISM is needed for meaningful H/S Co-simulation. Neither of these model types is trivial to create, thus it is critical to have access to a sizable portfolio of CPUs commonly used in embedded system design. Furthermore, performance of the interface between such models and the rest of the system must highly optimized.

Cross-Domain Optimization Aside from the obvious structural elements required to enable the co-simulation, number of other components on both SDP and HDP sides need to be modified to achieve performance sufficient for execution of industrial quantities of embedded software.

Source level access to the portfolio of ISMs is needed in order to achieve the following:

- **I/O notifications** -- the ISM needs to be able to efficiently notify the other side of co-simulation about read and write activities detected in executing software.

- **Asynchronous write-backs** -- the components residing in the Hardware Dominated Partition must be able to asynchronously modify the memory model being used by an ISM.

- **Non-blocking user interface** -- since we want to allow the user to control the session from both analysis domains, the debugger accompanying the ISMs should not block while waiting for code execution.

The tools servicing HDP also need to be revised.

- **Memory interface** -- a view of the memory needs to be provided to the simulation system that can be modified by the events occurring in the course of simulation. These modifications must subsequently be synchronized with the view of the memory available to SDP to ensure memory coherency.

- **Emulation interface** -- if a hardware emulator is used to accelerate simulation of HDP, a more efficient interface needs to be added. A mechanism for rapid import/export of stimulus and results as well as logic for fine-grained simulation control need to be added.

Detail vs. Performance

An obvious area where tradeoffs can be exploited is in varying the degree of detail presented to the user. When the amount of detail that the simulation tools need to worry about is reduced, the performance will improve.

Memory Controls As was stated earlier, processor to memory transactions typically consume most of the simulation cycles. The degree of detail required in simulating these interactions depends heavily on validation objectives. That is, when the system is first brought up, we want to accurately simulate the operations that allow the processor to fetch instructions from memory. When we observe that the processor can successfully fetch 20 - 30 instructions, the need for detailed and costly simulation of this type of transaction greatly diminishes. The proposed tool allows the user to turn off the detailed simulation of the instruction fetches and to start servicing them via much more efficient and rapid methods that do not require simulation.

Frequently, the embedded software encapsulates complex, memory-bound algorithms. These algorithms require the processor to frequently read and write data values to memory. Under normal circumstances these I/O transactions are fully simulated at a huge cost. That is, each I/O transaction results in creation and service of thousands of simulation events. We are paying a high computational price, but are learning nothing new about our system since all the operations are inherently the same with the only variants being direction and address. The proposed tool allows the user to effectively block entire regions of the address space from simulation. That is, when an I/O transaction affects an address within a blocked space, it is serviced via rapid software-based mechanisms that do not require simulation.

One of the key validation objectives early in the design cycle is verification of basic interactions between the ASIC(s) and the control software that is executing on the CPU. Under normal circumstances, it can take days of simulation just to bring the system to the point where interesting exchanges between the processor and the ASIC are about to commence. With the proposed system, the user can block the entire address space from simulation with the exception of memory mapped addresses that represent pre-defined communication channels between the CPU and the ASIC under test. This allows the session to progress rapidly by executing software that does not effect any hardware other than the memory sub-system, until the non-blocked addresses are hit. When this happens the I/O operations are fully simulated enabling the user to observe detailed interaction between the processor and the ASIC.

Mechanisms for manipulation of memory access are key in allowing the user to tightly focus the simulation session on validation objectives that are of greatest interest without incurring unnecessary computational penalties.

Event Suppression There are other methods that can be employed to further improve the simulation performance by allowing the user to more precisely correlate validation objectives with available run-time controls. As was stated earlier, the amount of events produced in the course of the simulation session is directly related to the consumed computer time.

The designer possesses the knowledge of the system under test and is aware of which part of his design has significant impact on his current validation objectives. Thus, the ability to easily exclude parts of the design from costly simulation without impacting the validation objectives is highly desirable. The proposed system allows easy activation and de-activation of design components and sub-systems.

Furthermore, this level of control over the simulation activity can be extended to individual signals. Clocks, for example, are highly active signals that may be deactivated for extended periods of time without any adverse effect on given validation objectives. The proposed system supports a variety of event suppression mechanisms giving the user ever greater control over detail vs. performance tradeoffs.

Summary

Trends in the embedded systems market are time to market pressures, increasing software content, and increasing design complexity. These trends are driving designers to consider new ways of validating their systems. Clearly, it is extremely expensive to fix significant hardware/software integration problems late in a project. Virtual prototyping offers the promise of finding integration problems earlier in the design cycle. This, in turn, reduces the duration of the Integration and Test phase.

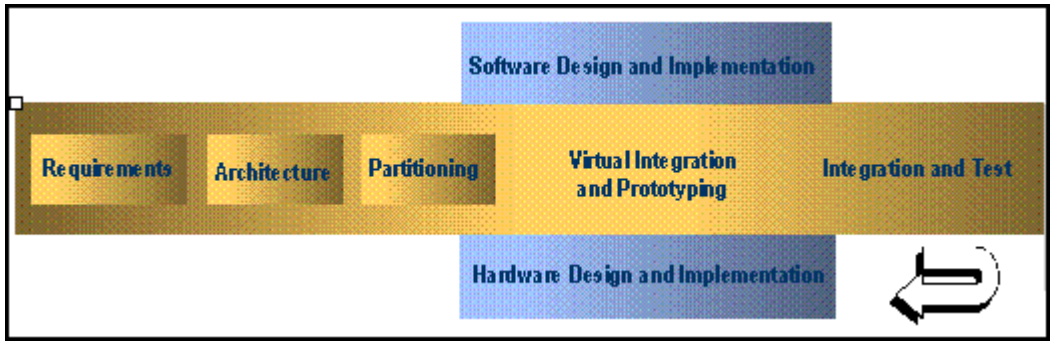

Figure 6. Impact of virtual integration on embedded system design process

Integration of hardware and software design tools is the enabling technology for virtual prototyping. Our experiments have shown that a simple integration between a hardware simulator and an instruction set simulation is inadequate in terms of software simulation performance. The proposed system gives the designer the ability to make intelligent tradeoffs between detail and performance at different times during the design cycle and presents an opportunity for performing virtual prototyping.

The proposed system contains the infrastructure that allows utilization of different techniques for both hardware and software simulation. While on the hardware side, the relative merits of cycle-based simulation and hardware emulation are straightforward to assess, the choice of tools on the software simulation side is less obvious. The table below compares different techniques with respect to four validation objectives.

	ASIC Stimulus	Protocol Verification	H/S Interface Verification and Firmware Debug	System Debug (RTOS + Apps)
Model Control Language (MCL)	**V**			
Compiled Software Model (CSM)	**V**	**V**		
Instruction Set Model (ISM)	**V**	**V**	**V**	
ISM + Hardware Emulator (HE)	**V**	**V**	**V**	**V**

Table 2. Simulation techniques vs. validation objectives

The system described above has been developed and is now available to embedded systems designers world-wide (this will be true @ EuroDAC timeframe).

References

1. *Bailey B., Klein R., Leef S., Hardware/Software Cosimulation Strategies for the Future. Publication pending.*
2. *Klein R., An Industrial Hardware/Software Cosimulation Solution. Publication pending.*