DesignCon 2002

System-on-Chip and IP Design Conference

Co-Verification: From Tool to Methodology

Brian Bailey

Mentor Graphics Corporation
Mentor Consulting Division

Abstract

While many design organizations are using HW/SW co-verification as a tool in their verification flow today, few have yet to make it an integral part of their complete methodology. With software becoming a more significant part of end designs in terms of its size, complexity, and ability to competitively differentiate a hardware platform, a verification strategy that includes both the hardware and software components is becoming essential. Additionally, verification environments are incorporating a wider range and combination of execution environments -- such as event-based simulation, cycle-based simulation, emulation, co-simulation, etc. – including accommodation of their varying performance levels. This paper will discuss some of the tradeoffs made when setting up such environments and how these environments have been used in industrial cases.

Author Biography

Brian Bailey is the chief technologist for the System Verification Development Office for Mentor Consulting, the professional services organization of Mentor Graphics. In this role, Bailey is responsible for integrating and managing new advancements in functional system verification. Previously, Bailey was involved in the creation of the Mentor Graphics® Seamless® Co-Verification Environment™ (CVE™) for hardware and software co-simulation. Prior to his work with the Seamless product, Bailey worked with a variety of simulation technologies, including simulation acceleration, emulation, mixed-signal and multi-level simulators. Bailey began his career in the aircraft design business before leaving to work on the emergence of RTL simulation. Bailey graduated from Brunel University in England in 1981 with first class honors in Electrical and Electronic Engineering.

Actively involved with the Virtual Socket Industry Alliance (VSIA) since its inception, Bailey has also been a key contributor to the System Level Design working group and recently assumed the working group Chair. He also serves as Chair of the Accellera High-Level Language Semantics working group, having successfully taken the group from its formation in June 2000 to drafting a standard that is currently undergoing the approval process.

Introduction

The embedded systems of today are task-specific computing devices consisting of both standard and custom, hardware and software components. The standard hardware components include one or more of a fairly small set of microprocessors, memories, digital signal processors (DSP) and possibly, some standard parts obtained by internal reuse or from third-party vendors. Custom hardware is implemented using the traditional ASIC design process or, in some cases, is realized in field programmable gate arrays (FPGAs). The hardware architecture binds these resources together and provides the framework from which the software processes execute. The standard software components may include real time operating systems (RTOS), configurable drivers, libraries of routines from previous designs, plus custom software for this design. The software architecture defines how these components communicate.

In the past few years, more complexity has been introduced in the software components to the extent where many systems rely on software for their major differentiation. The platforms upon which this software executes range from a single processor to complex platforms containing a number of heterogeneous processors. All of these factors have increased design complexity, but their effect on verification has been even more dramatic. As a general rule, verification complexity grows at more than the square of design complexity and most people do not even include the software when they consider system size and complexity. Thus, verification is fast becoming the bottleneck between idea conception and a successful product launch.

Also, the term “verification” covers a lot of different operations that deserve our attention, from an evaluation of the initial specification, through interpretation, implementation and integration. The goal of verification is to lower the risk of product failure, but this must be done at reasonable cost. There is little value in having a perfect product if it is late to market.

This paper will elaborate on one particular aspect of verification, that of hardware/software co-verification. While tools in this area have been widely adopted at a fairly low level of design abstraction, they have not lived up to early expectations. This paper will discuss the reasons behind these shortcomings and how to extend the usefulness of these tools.

Basics of Co-Verification

For a long time, the only way possible to run software within a hardware simulation environment was by employing full functional models for processors. These models consumed large amounts of simulation time, if they were available at all. Some manufacturers were reluctant to make them available for fear of divulging their intellectual property. Even with these models, it was only possible to run a handful of instructions at a time -- hardly enough for any form of software verification. To overcome these problems, companies have invested in a number of different techniques over time, all of which rely on extracting the processor from the logic simulator and replacing it by a model with a higher level of abstraction.

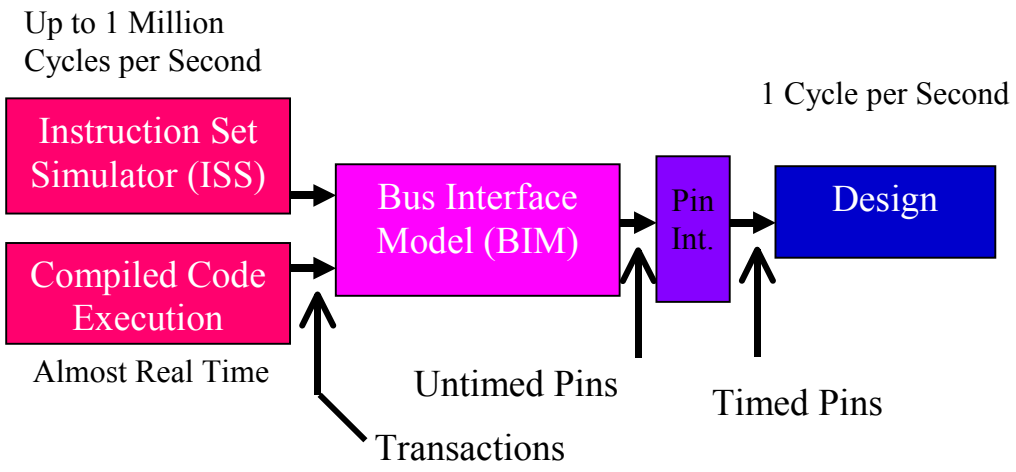

Figure 1. Components of a Process Model

The typical co-verification model contains a number of components, as shown in Figure 1. The processor model is connected to the design at the pin level, and a configuration file within the pin interface model handles timing in and out of the model. The pin interface is driven by a Bus Interface Model (BIM). An early form of the BIM was the Bus Functional Model (BFM). The BFM did not contain any of the internal processor functionality associated with the instruction set. Instead, it modeled only the interface circuitry and was driven by a command language. Today's BIM supports higher levels of abstraction and also provides a transaction interface to which a number of execution vehicles can be attached.

The most popular execution vehicle to drive a BIM is an Instruction Set Simulator (ISS). An ISS provides a functional model of the operations of the processor, including the important state storage elements of the device, but not for the data path that connects them. It is modeled either at a very high level of abstraction or not at all. Each instruction is modeled as the set of state changes it would cause in the internal registers, other state storage devices, and in the memory system to which it is connected. Modeling instructions at the state change level allows the ISS to execute at speeds of up to, and sometimes exceeding, 1 million instructions per second on today's high-end workstation. These models exist at two levels of accuracy: an instruction accurate model and a cycle accurate model.

Instruction Accurate

An instruction accurate model is accurate at instruction boundaries only. While the model does guarantee that the necessary bus cycles will be executed in the right order and that the correct number of clocks will be executed by the end of the instruction cycle, it does not guarantee these bus cycles will be performed at exactly the right time. This model trades off maintenance of the internal state at each clock boundary to obtain higher execution speeds. In most cases, this model is adequate since it closely approximates code performance on real hardware. There may be other cases, such as when critical bus contention policies are being used or when clock accurate verification is required, when the inaccuracies it produces may not be tolerable.

Cycle Accurate

A cycle accurate model, as its name suggests, guarantees the exact bus behavior of an implemented system. While considerably slower than the instruction accurate model, this model executes significantly faster than a full functional model since it models the behavior of the internal activity rather than the internals producing that behavior.

Another way of driving the BIM is through compiled code. In this scenario, embedded software is compiled for the host system rather than for the target, its I/O operations are trapped and translated into the necessary bus cycles, and most of the target's memory subsystems are replaced by those in the host. With these modifications in place, this model runs close to real time and, in some cases, runs faster than the real device.

The user must consider the tradeoffs associated with the performance gains afforded by this model, as clearly, nothing is for free. First of all, the model does not represent the internal registers of the processor, which makes it impossible to perform low level debug. While most operations on the host and target will execute essentially the same, there are a few exceptions – mainly those associated with floating point operations and with word length differences, if they exist, between the host and the target.

However, the biggest difference is that all cycles associated with an instruction fetch or other data operations will be hidden from the hardware since the memory associated with them is replaced by host memory. This is not always a problem. If the hardware and software systems have been written to be self-synchronizing, then this model can be successfully applied to functional verification. This same mechanism also makes it possible to integrate abstract software models such as those described in Specification and Description Language (SDL)[1]. SDL is a highly abstract standard language for describing software that is both textual and graphical, and it allows description of complex, real-time systems.

Performance Fundamentals

So why isn't everyone using co-verification? Wouldn't this enable verification orders of magnitude faster than before? To find these answers, we need to understand Amdahl's Law[2]. Amdahl's Law addresses the overall speedup in a system based on the enhancement of one of its components, taking into account the percentage of time spent in the components that is being enhanced.

$$\text{Ex Time}_{new} = \text{Ex Time}_{old} \times \left[(1 - \text{Percent Enhanced}) + \frac{\text{Percent Enhanced}}{\text{Speedup Enhanced}} \right]$$

$$\text{Speedup} = \frac{\text{Ex Time}_{old}}{\text{Ex Time}_{new}} = \frac{1}{(1 - \text{Percent Enhanced}) + \frac{\text{Percent Enhanced}}{\text{Speedup Enhanced}}}$$

Suppose we have a system with an identified area of inefficiency, work on that piece, and speed it up by a factor of 2. If that piece represents 10% of the total system activity, we would discover that, after plugging the numbers into this equation, the enhancement achieves an overall speedup of just over 5%. If the improved portion represents 20% of the activity, then the overall speedup would be a little over 11%, and so on. The message here is that performance increases affecting only a small part of the overall execution wind up having only a very small effect on overall performance. From the perspective of co-simulation, unless the processor represents a large portion of the execution time of the logic

simulator, and we were to speed up that process by a factor of 100,000, it will not provide much upside performance gain. This is what happens if we were to replace the full functional model with an ISS model. Figures will vary widely based on the design characteristics and how simplified the estimates are made when using this equation, since this calculation ignores other performance diminishing factors, such as communications costs.

While this performance gain seems small, we need to put it in perspective. For an 8-hour run, we would save over 1.5 hours using the assumption of 20% processor activity. If we further assume that verification is performed over the course of two months, with three people utilizing the tool 60% of the time, it would have taken a total of 864 hours. However, by using co-verification, we would save 172 hours -- which represents a time-to-market savings of almost 12 days!

This calculation ignores many of the other benefits that come from co-simulation, such as gaining an integrated tool chain, fast loading of the executable into the simulated systems memory, and a full debugger that can substantially reduce time to diagnose and fix problems. Many users contend that even if there were no time savings associated with co-simulation, they would still use the tool for its ability to provide greater visibility into the design and a better perspective of the complete system from their separate vantage points for hardware and software engineers alike. The hardware engineer can look at waveforms, while the software engineer can see his or her code advancing line-by-line or instruction-by-instruction.

Hardware / Software Relationship

Moving beyond early co-simulation solutions, the second generation of tools takes into account an additional understanding of the relationship between hardware and software. All software operations involve access to memory via a bus, such as instruction fetches, operand fetches, data reads and writes, and I/O operations of memory-mapped peripherals. Almost all of these operations use hardware (the bus, the memory, the arbitration mechanism, etc.), yet they do not affect the state or the operation of the rest of the system. These hardware operations support the software process, and therefore, are a target for optimization as well. When looking at the different types of software, you will notice there is a wide range in the frequency in which the software affects the hardware[3].

Software Type	HW factor
Initialization	27%
Diagnostics	5-15%
Driver Code	5-10%
Application Code	1-5%
RTOS	< 1%

Table 1. HW utilization for software type

With driver code, for instance, at least 90% of the activity associated with software memory accesses could be eliminated without affecting the state of the hardware. To realize this, a co-simulator kernel must be able to dynamically manage the logical placement of memory in one of two places based on the type of software operation in progress. When memory is part of the hardware, it is slow but allows for comprehensive verification. When memory is modeled in the software process, the memory is able to respond as fast as the ISS. This is depicted in Figure 2. If this effect were added to Amdahl's Law calculation and produced a 35% speedup to total system activity, then time-to-market would be reduced

by three weeks from the two months originally set aside for verification. It is important to make sure that for any co-verification solution considered, the logical placement of memory is not fixed, but is alterable dynamically. Without the kernel's capability to dynamically alter the logical placement of memory, it becomes impossible to avoid compromising visibility, resolution, or performance when running simulations containing all software types.

Figure 2. Co-Verification Memory Tradeoff

Of course, the ability to dynamically alter the logical placement of memory adds its own costs in terms of requiring a mechanism in place to ensure coherence of the system memory. For instance, if software has just performed an accelerated write to a memory location that is read in the very next cycle by hardware, the read must be assured of obtaining the updated value. Likewise, operations in hardware that affect memory must be reflected in the software view of that memory. The overhead associated with this will be different for various tool implementations.

While further optimizations in the software side are possible, the key to achieving significant speedup gains in co-verification is by working with the hardware. Before discussing that in detail, the next section will discuss where co-verification fits into the design flow.

Conceptual Design Flow

Referencing a traditional simplification of the design flow, shown in Figure 3, the process starts with some form of system design, either on paper or as an executable specification. Next, the system design is partitioned into its hardware and software components, and the architecture of these components is defined. Also considered are the ways in which the two components will interact with each other. These high-level design processes are highly iterative, and once they have been firmed up, the software and hardware teams start on their separate implementations. Before the physical design can start, the hardware and software designs will be integrated together. Figure 3 below ignores the iteration loops through this flow to keep it simple.

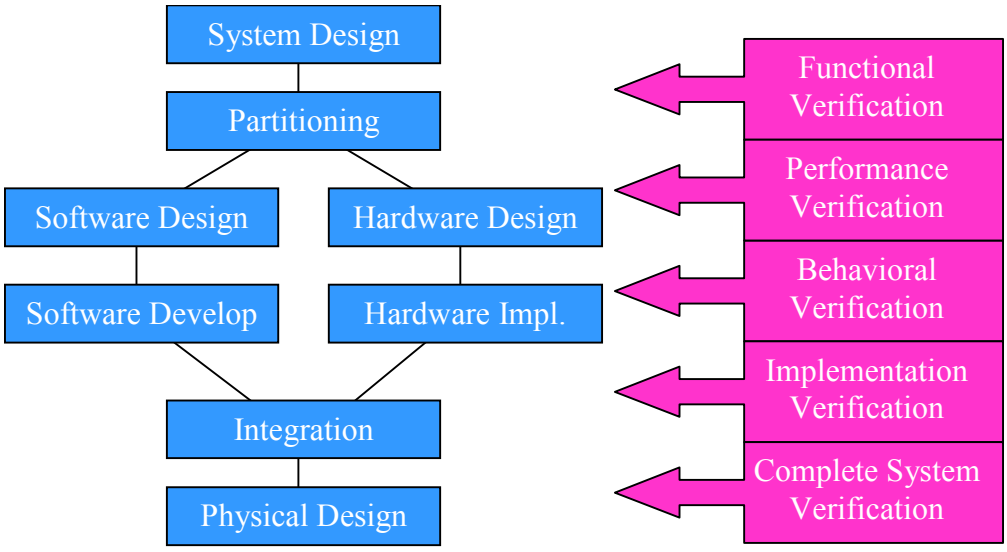

Figure 3. Design and Verification Flow

A verification flow parallels each phase of the design flow. At the executable specification level, the focus is on functional verification. After partitioning the system design into high-level hardware and software specifications, attention moves to evaluating the resultant architectures and then to the performance they provide. Once the partition is set and the hardware and software architectures are determined, the focus moves to verifying their behaviors. When the hardware and software designs have been implemented, they are verified separately and then again on the system level after they are integrated together.

While companies may not have a formalized design flow that covers all of the blocks shown above, they must ascertain that all aspects of verification are covered – the earlier in the flow, the better. Functional verification, for example, cannot be omitted simply because the system is not modeled at a high level. Leaving functional verification until later in the design flow is less efficient since the abstraction level used for modeling will be at higher levels of accuracy than is required if done at the optimal time. With methodologies in place at most companies today, hardware/software co-verification is used only for the integration of hardware and software blocks. Below that level, co-verification has been faulted on speed because it is not possible to run enough software on the RTL model of the hardware to verify system functionality sufficiently. Above that level in the design flow, there are organizational problems and a lack of suitable models. However, these barriers are changing.

Co-Verification in the Design Flow

Co-verification has a very important role to play in all areas of verification except functional verification, even though today's tools may not yet support all of the necessary capabilities. The emergence of new, high-level modeling techniques, such as Cynlib[4] or SystemC[5], ensure abstract models will become available shortly. Once partitioning choices are made and the architecture between the hardware and software is established, high-level co-verification will enable rapid evaluation of system performance. This will illuminate system bottlenecks, allow conceptual solutions to be tested, and greatly improve the level of reliability in the selected partition. An early indication of this role for co-verification is the integration of the Cadence Cierto™ virtual component co-design (VCC) environment (which focuses on the issue of high-level modeling and partitioning), with the Mentor Graphics

Seamless Co-Verification Environment (CVE) tool (which allows for detailed analysis of a partitioning choice).

When the system is partitioned, three pieces are created: the hardware functionality, the software functionality, and the interface between them. The software functionality will undergo further refinement[6] and will include its architecture and a high-level model of the RTOS. The hardware partition will also undergo refinement, and the interface between the two partitions will become the environment supplied by the co-verification tool. This is shown in Figure 4.

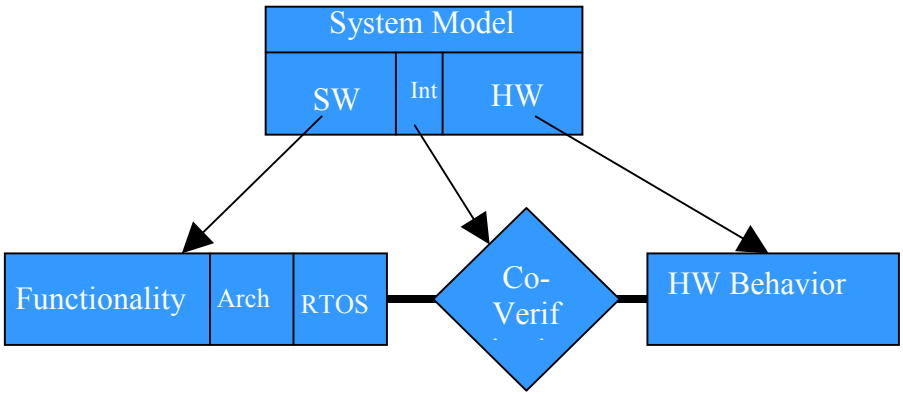

Figure 4. HW / SW Partitioning

With no processor model available at this stage, the interface between the hardware and software partitions is modeled in terms of abstract buses, high-level arbitration mechanisms, high-level memory systems, and symbolic memory maps. As functionality gets implemented on the software side, verification may start by using the compiled code execution engine, which enables the hardware model to act as accurate 'stub' code. Later, parts of the code -- particularly the low-level routines and drivers -- can be executed on the ISS with detailed hardware for the parts of the system that the driver is targeting and abstract hardware for the rest of the system. The ability to mix hardware abstraction models enables much higher levels of performance than previously attainable.

Hardware developers can utilize co-verification in two ways. The first is as a low-level directed test mechanism[7]. By using a BIM and custom software, similar to that used for compiled code execution, hardware engineers can create stimuli and check for expected system behavior. However, co-verification is not limited to processor busses. Models can be created for any hardware interface and then driven by an appropriate language at the transaction level.

The second use of co-verification by the hardware developer is to use actual driver code to provide realistic bus traffic into the hardware. This is a relatively easy way to functionally verify modules and the interactions among modules.

The ability to hasten the simulation performance of the hardware, as referenced earlier, will be the enabler for moving co-verification from a point tool to an integral part of the verification methodology. Hardware simulation performance can be improved in a number of ways: by increasing the abstraction, off-loading it into a dedicated environment, or matching the information resolution with the verification objective. Each of these techniques will now be considered.

Increase Abstraction

As mentioned previously, emerging languages allow hardware systems to be modeled at a higher level of abstraction and may enable future design tools to facilitate automatic refinement of the hardware. These modeling systems utilize the framework of the C++ language and add new class libraries that enable modeling capabilities for hardware specific notions and provide the required degree of concurrency for the hardware. These languages also hold promise for bridging the divide between the hardware and software communities since they allow usage of the same syntax for their descriptions. While the syntax may be the same, the semantics of applying these languages to create the separate software and hardware descriptions will differ. Efforts are underway to define these hardware semantics[8].

The most important criterion for co-verification is ascertaining the interface models between the class library and the co-verification environment match. To fully utilize the power of this combination, communications between hardware and software should be enabled at a number of abstraction levels. Six levels of communications define the range between the most abstract level and the least abstract level (fully implemented pin level communications) as shown in Table 2. Not all of these are likely to exist or be used in any design flow.

Level	Architecture	Description
1	Point-to-Point	All transactions between any hardware and software block use a dedicated connection.
2	Point-to-Point using single resource	Transactions are arbitrated so that only a single communication can occur at any time.
3	bus transaction	Transactions are broadcast with bus attributes such as address, access type, access method.
4	bus message	Transactions are broken down into individual bus operations such as read, burst cycle…
5	bus protocol	The actual bus protocol is run but still uses data abstraction such as an integer for address.
6	pin level	The complete bus protocol at the pin level.

Table 2: Levels of Communications

At the highest level of abstraction, referred to as Level 1, there is no concept of a bus. Blocks communicate with each other through dedicated channels and with arbitrary payloads. These payloads may contain control information and data, and they may occur concurrently (given the limitations of the modeling tool). The next level down, at Level 2, communications is aligned to abstract notions of a channel. Data payloads are not altered, but only one transfer can occur at a time. Time may be consumed in performing the transaction and this is affected by factors such as the transfer type, payload size, etc. At Level 3, the concrete notion of a bus is added, which requires some form of protocol overhead, such as providing a destination address. The payload will be maintained as a single transfer, but characteristics such as bus size, or supported operations such as burst, may be taken into account when estimating transfer times. By Level 4, transactions have been broken down into distinct bus cycles, although it is not until Level 5 when the actual transfer protocol, such as write pulse and other signal waveforms, becomes apparent. Finally, at Level 6, the actual pins are defined and waveforms are applied to them.

Improve Performance

Moore's Law defines the increase in design complexity and also explains the similar effect it has on the performance of logic simulators. Since verification complexity increases more rapidly, the power of using the logic simulator loses ground in a relative sense. The last section covered some of the implications of increased design complexity. However, at some point, most people want to see the design running at the RTL implementation level. When this becomes necessary, emulation remains the only viable option. A typical, high-end emulator runs at 1 MHz, and it provides a complete debug environment and full visibility into the design. This may appear to be Nirvana since both the hardware and the software are running at around 1MHz. Unfortunately, a new bottleneck emerges between the communications with the software environment and the emulator. For a typical bus, there are quite a few transitions within every clock cycle, and this can very quickly degrade overall performance. Using a synchronous bus communications protocol, such as bus level 4 in Table 2, can improve communications performance. But for accurate co-verification, it becomes difficult to move much higher than this point.

Dedicated Environments

All designs need a testbench, which is the principal means by which a stimulus is injected into a design and monitored for correct operation. Today, most designers use the same languages to describe the testbench as they use for the design, such as VHDL or Verilog. Since these languages were never designed to build the complex testbenches that are fast becoming a requirement today, they tend to be inefficient is this regard. Migrating to a specialized testbench language, such as *e*™ from Verisity, can significantly reduce the time taken by the testbench. By using the right level of abstraction for the verification environment and removing the corresponding low-level portions from the logic simulator, users can achieve significant increases in verification performance. In addition, when coupled with emulation, they can overcome the similar limitations that were described above for co-verification. A typical testbench does not need to synchronize with every clock of the design. Transactions can be defined and passed into the emulator, which will then be injected into the design over substantial periods of time. This is how highly efficient emulation - verification environments are created.

Verification Objectives

Before diving into verification, it is advised to take a step back and consider what you are trying to verify. If you are planning on doing a clock accurate simulation, you will need a full RTL model and must synchronize the hardware and software on every clock; these actions will enable you to verify the timing and sequencing of the system. With this resolution and timing, there are limits on achievable performance. However, if you are trying to verify functionality at a more abstract level, such as at the behavioral level, then the synchronization criteria can be relaxed. No longer must the hardware advance at every step in the software, and vice versa. In many designs, there are large blocks of time when it is known the software is waiting for an event. The act of simulating this 'waiting' takes a lot of time. Instead, it is more efficient to shut down the whole software system until the required event occurs or a special event, such as an interrupt, happens. Similarly, if the hardware is starved of data and is waiting for the conclusion of a software process, then it is more efficient not to advance the hardware. These kinds of optimizations have been incorporated into today's co-verification tools that, when coupled with some of the other techniques described above, can produce very high performance verification.

Partitioning

After deciding on the abstraction level to use for each of the blocks in the design and putting the correct interface abstractions between them, the focus switches to how to map them onto the verification tools. This problem is illustrated in Figure 5 below. The design is made up of one or more processors and a number of blocks that are at different levels of abstraction. While we have indicated how to reach some decisions about the components of the verification system, such as the use of dedicated testbench tools, this does not in any way affect the strength of the arguments, which applies equally if the testbench were modeled in a language such as VHDL.

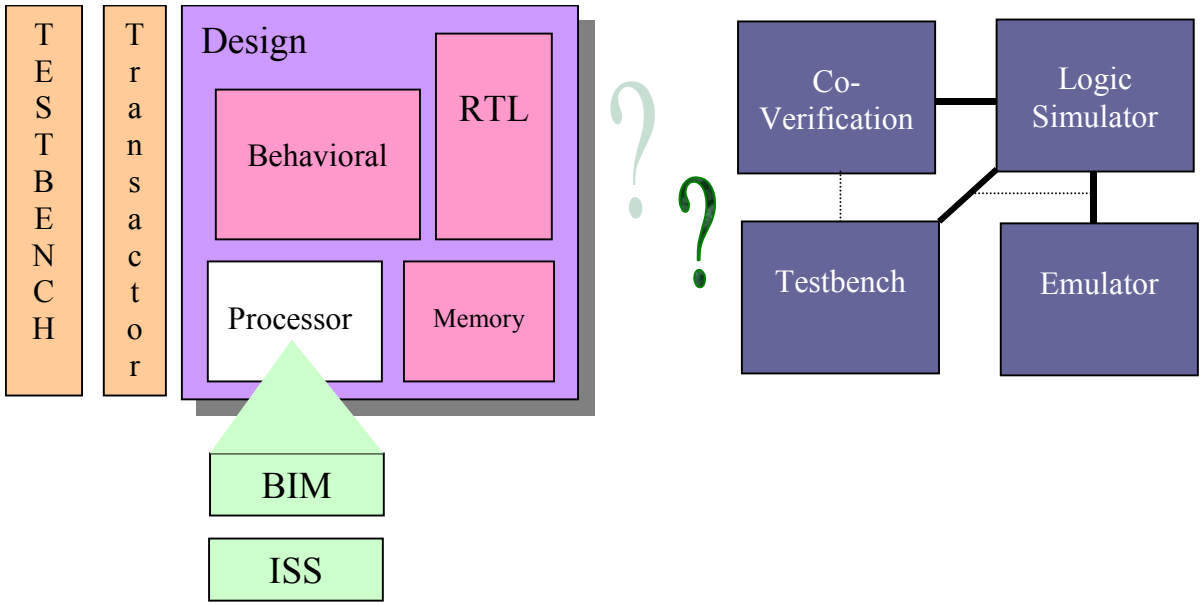

Figure 5. Design and Tools Mapping

A number of different tools are required to execute this complete system at maximum performance and provide a full debug environment. Prior to this section of the paper, all concepts introduced were explained independent of the choice in tools. Now, specific tools will be discussed so that more detail can be given about how to create the verification solution. The specific tools used for this discussion are Seamless for co-verification, the ModelSim® logic simulator from Model Technology, the Mentor Graphics Celaro™ emulator, and Specman Elite™ from Verisity for the testbench. Fortunately, most of these solution creation concepts will apply equally to tools from other vendors. To minimize any bias, we use normalized data in place of absolute data. For the context of this discussion, the term "behavioral" is used to refer to the coding style used for a model that cannot be synthesized by synthesis tools currently available and verification of a behavioral model is limited to a traditional simulator. This situation will change as technologies advance, but also it is likely that some capabilities will remain beyond the scope of synthesis tools. Are there any easy choices? Probably the only easy choice is for the testbench if it is written in the *e* language. Since ModelSim or Celaro does not directly support this language, its use will remain within the Specman Elite environment.

A testbench consists of a number of components which includes stimulus injectors, results capture, result predictors, pattern generators, and many other possible blocks that either tie these

components together or direct their activities. A transactor is a testbench model that is similar to a BIM for co-verification. They both transform a high-level request into a number of pin-level operations, or they take pin-level operations and convert them into higher level transactions. All stimulus injection and results capture are likely to have transactor models. So where should the transactors be placed? From a tool perspective, the obvious answer would be in Specman so that the transactors can be modeled in *e* as well. This would provide a pin-level interface between the testbench and the design. If the points of contact between the testbench and the design are all in the logic simulator, then the location of the transactors doesn't factor. However, if some of the points of contact are in the emulator, then their location matters greatly.

The performance of a pin-level interface into an emulator is dependent on a number of factors, including software processing time, physical transfer time, time needed for accessing the emulator hardware, and possibly some overhead. Emulation products from some vendors have to 'stop' the emulator in order to be able to use the co-emulation interface, and this can be quite expensive. For Celaro, there is zero cost associated with this, and the co-emulation interface can always be accessed. For all of the other time-based components, there is a heavy dependence on pin count and activity densities. If we start by assuming that the pin-level signals change every clock and that the number of pins is fairly small -- as would be the case for most standard interfaces such as busses, USB, PCI, etc.-- then we can assign this case the baseline performance of one. All others will be measured against this baseline. If we move the transactor onto the emulator, there will be a small net saving in the software execution time since time will be saved in the simulator and no additional time will be needed in the emulator. However, this savings is probably insignificant since the time spent between transfers is what is more important. For a processor BIM, with one clock per bus transfer and a parallel data operation, there would also be a negligible speedup. If we consider the serial protocol, USB, we can see some pretty dramatic speedups. In a highly simplistic look at USB, we can have transactions as small as one byte for a payload with a 9-byte protocol overhead, all the way up to 1023 bytes of payload with a similar overhead. This translates into 80 to 8256 bits of information transferred. If we assume that 30% of the adjacent bits actually change (this is probably low), then there would be between 32 and 2477 bit changes at the pin level. This approximates the speedup in communications. In the pin level case, we would have 32 to 2477 pin changes over 80 to 8256 clocks. For the transaction level, there would be one transfer of 10 to 1032 bytes, which would take a little longer than the transfer time for one bit, but not significantly so.

For software execution, the fundamental decision to make is whether the processor is modeled in software or in hardware. This paper has already discussed the construction of software solutions. There are two ways to place the processor into the emulator. The first is to insert a piece of physical silicon into the emulator. Emulation vendors provide plug-in cards for the most popular processors, and these can run at full emulator speeds. Most challenging for the designers of these plug-in cards is to provide the ability to both slow down and stop the processor. Slowing down the processor enables working in environments that have slow clocks. Turning off the processor clock enables interactive debugging of the design. The second option -- especially for people who have licensed the core from an external provider or if the core was developed in house -- is to map the core directly into the emulator. While this allows full visibility into the design, it can take up valuable emulator space. This remains a viable option if the physical silicon is not yet available.

The biggest issue with both of these approaches is ensuring the software can be debugged with a standard debugger, such as Mentor's XRAY® debugger tool. For the case where a plug-in card is used, a JTAG port is required. For the case where the core is mapped into the emulator's programmable

elements, this requires physical I/O pins within the emulator to emulate the JTAG interface. If the physical silicon for a core without RTL is not yet ready/available, then the only way to perform early verification is via software execution with tools such as Seamless CVE. Designers who race to include the latest cores often before the design of the cores is completed frequently take this latter path. Besides providing a very stable model of cores, the ISS for these cores is often available months – sometimes up to a full year -- before completion of the design.

Recently, this was the case for a 3G wireless application. The developer selected a DSP core well ahead of the completion of its design. Therefore, a Seamless model of the core was used to start HW/SW testing. Once the physical version of the core was available, an emulation card was constructed and it was used for verifying the longer software runs. Since there was a limited number of physical cores available, Seamless continued to be used for the smaller software runs.

When using a co-verification solution such as Seamless, one should give careful consideration to placement of the memories since communications between the processor and the rest of the world is through the bus. If that bus is the principal interface to the emulator, then we need to reduce its traffic as much as possible. Seamless provides the ability to optimize access to memory. If some portion of the design's memory is dedicated only to the software process and its does not affect the state of the hardware modeled in the emulator, then this memory should either be placed in the logic simulator or modeled as software-only memory. Then, accesses to this memory would create little to no bus activity. If the memory is kept in the logic simulator, then Seamless can perform optimized reads and writes to them, and the emulator would similarly perform a bus cycle when the hardware needs to gain access. This is not a problem unless the memory is multi-ported. For multi-port memory, additional signals would need to be created for the interface, and this would slow down all cycles. For more complex memories, it is advised to place them in the emulator and have Seamless perform the bus cycles in order to read from and write to them

In contrast, if memory does not need to be accessed by both the hardware and the software, then the processor and the hardware do not need to be kept synchronized on the full clock level. In this case, Seamless can inhibit the advancement of the clock during these accesses, completely hiding these cycles and preventing activity from crossing the boundary. For the behavioral parts of the design, the same arguments exist when looking at the boundary between the behavioral and RTL portions of the code. Clearly, there is no choice for the behavioral portion but the most efficient partition may not be at the obvious places.

In another 3G application, the maker of a component for a base station design wanted to create a verification environment for its customers in advance of the silicon being available. Within a traditional logic simulator, there would be no software debug environment and the simulation of a design of this size and complexity would execute at just a few clocks per second. While it would have been possible to map the entire design onto an emulator, it would have consumed a sizeable portion of the largest emulator available, making it a prohibitively expensive, but high-performance, solution. With the help of Mentor Consulting, the design was divided into three components and spread across Seamless, ModelSim and Celaro. The processor cores were mapped into Seamless and the bulk of the peripherals were mapped into Celaro. With this arrangement, the performance was considerably higher (500X) and the cost much lower (1/20X). Much of the system memory was kept in ModelSim because the bulk of the traffic was between the processor cores and the memory sub-system. Only when a packet load or unload occurred did the rest of the hardware need to access the memory.

Since the majority of the tests were to verify the transformation of just a few packets, the emulator/memory access times were not significant. This scenario also allows for Seamless to make optimized transactions to the memory for functional verification, especially when the rest of the chip was basically idle – such as when waiting for the completion of packet processing or for the arrival of a new packet. While it was possible to also map the peripheral registers into Seamless, additional signals between ModelSim and Celaro would be required. Doing so would have optimized access to Seamless but also would have placed a heavy penalty on all other transfers. Only when access to these registers is especially frequent does it pay to incur the added engineering cost to implement this. Most of the time, only a few registers will yield an improvement in access time when implemented in this way.

Conclusions

In this paper, we have looked at both the current "state of the art" in hardware/software co-verification and how it can be applied to various kinds of designs and tool environments. Performance obtained from these systems can vary widely depending on such factors as the modeling paradigms used, accuracy of the time resolution in the simulation, and how the design is partitioned across these tools. We have shown that in order to achieve the highest performance, careful consideration must be given to the communications among the various tools, particularly on how minimizing the amount and frequency of these communications has significant impact on performance.

References

[1] www.telelogic.com/products/tau/languages/sdl.cfm
[2] Amdahl, G.M. Validity of the single-processor approach to achieving large scale computing capabilities. In AFIPS Conference Proceedings vol. 30 (Atlantic City, N.J., Apr. 18-20). AFIPS Press, Reston, Va., 1967, pp. 483-485.
[3] M. Stanbro. "Getting to the bottom of HW/SW co-verification performance claims." Computer Design Dec 1998
[4] http://www.ForteDS.com/products/cynlib.html
[5] http://www.systemc.org
[6] L. Thompson. "SoC parts expand RTOS design options." EE Times Nov 8th 1999.
[7] J. Kenney. "Minimizing risks with co-verification." EE Times June 15th 1999.
[8] Accellera CWG semantics working group. http://www.eda.org/alc-cwg