

# Hardware/Software Co-Development: Differentiating between Co-Design and Co-Debug

Todd P. Carpenter, Sanjaya Kumar, Fred Rose  
Honeywell Technology Center (HTC)  
3660 Technology Dr  
MPLS MN 55418-1006  
<carpenter\_todd@htc.honeywell.com>

## ABSTRACT

This paper explores issues relating to cooperative software and hardware development, focusing on different stages in a design cycle supported by VHDL-based design and development techniques. Since most commercial tools focus on implementation, integration, and debug, rather than early life-cycle stages such as requirements capture and design, the application of these tools requires different techniques, goals, and expectations. The experiments and trade-offs appropriate for the different stages in the design cycle are evaluated, as well as the performance of the tools and models.

## 1. Introduction

VHDL is a convenient medium for representing portions of an electronic system. This representation, or VHDL model, is used in various ways, including:

- as a specification for subsequent, more detailed levels of design (e.g., a model for synthesis)
- as a test bench for verifying that the model accurately implements the system requirements (debug & validation)
- as a test bed for exploring architectural alternatives (design analysis & evaluation)

Depending on the design cycle, models represent widely varying abstraction levels. Some design processes do not employ multiple abstraction levels, but start with textual requirements and directly implement a register-transfer level (RTL) model in VHDL for subsequent synthesis into a gate-level representation. This paper focuses on design cycles for large scale systems requiring broader system-level design capabilities, in particular those where software's execution characteristics are of prime importance. Examples of such systems include avionics, communications, and signal processing, where the majority of the system's functionality is implemented with software.

A current focus in the development of such systems is *hardware - software co-design*, an often overloaded phrase, fraught with diverse meaning and content. This paper intends to differentiate between two very distinct stages in the development cycle, namely the early stages of requirements analysis and architectural design and the latter stages of implementation, verification, and debug. While many features of the system occur in models at both stages, the model content, purposes, and capabilities vary greatly, as well as requirements levied on the supporting tools. The primary focus of

the great Electronic Design Automation (EDA) houses is currently on those latter stages.

This detailed design stage focus is good business sense from the perspective of the EDA companies and the host of engineers and consultants, since for every system designer there are tens of implementors, and hence tools. Moreover, the computational complexity of those latter stages is currently orders of magnitude higher than traditionally required by system level designers.

However, these traditional design processes have their flaws, as many have experienced when integrating subcomponents. Engineers then have to patch interfaces, or worse yet, fix and redesign subcomponents. In the pure hardware arena, we can now point to clear cases where chip and board-level simulation has reduced the number of "chip passes," that is, the number of times a particular ASIC design has been fixed and resubmitted to the chip foundry [1,2,3]. The number of chip passes is a quantifiable metric, highly correlated to cost and schedule impact. A foundry charging \$100k per turn is common. There are the additional costs and time to integrate the  $n$ th pass chip and test it, and these costs are also quantifiable.

For some time, the EDA focus was supporting sufficient simulation and design rigor to ensure that first-pass silicon yielded working devices. This process is now state-of-the-practice within industry. This requires intensive computational capabilities, since modern ASICs frequently reach the tens to hundreds of thousands of gates.

Next, focus shifted to board-level digital system simulation where individual components are brought together to resolve interface issues. This practice reduces design flaws beyond what is capable with just unit testing. Such board level simulations are common, though the computational complexities of combining several large devices in one simulation is still a cost limiting factor.

Now, we are bringing together multiple domains, such as hardware and software, as well as analog and digital, and verifying their interaction before we attempt to fabricate the devices. This capability is state-of-the-art, rather than current practice. Part of the hesitation is the lack of enabling technology, as well as limited understanding of the issues involved. Vague terminology also hinders progress, such as when one user or design manager thinks "co-design" is one thing and a vendor supplies tools supporting an entirely different definition of "co-design."

The remainder of the paper explores two examples from different ends of the VHDL design spectrum. Each section briefly describes relevant characteristics of typical systems, the goals of the modeling activity, and some of the technological challenges relating to the actual modeling and simulation process. At the end, a comparison is made, and some conclusions are drawn.

## 2. Hardware-Software Co-Design

According to Webster's [4], design is "*to conceive and plan out; to create, plan, or calculate for serving a predetermined end.*" This matches our expectations regarding the early stages of the system life-cycle, where engineers conceive of solutions to meet product requirements. As our system complexities continue to increase, so does our reliance on repeatable and analytical evaluation of alternative architectures. Hardware/software co-design, while not yet defined in my version of Webster's dictionary, is the cooperative design of the hardware and software components of a system, allowing the tradeoff of options in one domain to affect design decisions in the other.

## Example Systems

Since the late 1980's, Honeywell and others have successfully applied VHDL-based, hardware/software co-design methods to several embedded system designs [5,6,7]. These systems are all multiprocessor architectures with a significant portion of the system's behavior captured in software. The number of processing elements varies, ranging from 4 to 128 elements and consisting of both homogeneous and heterogeneous networks of general purpose processors and digital signal processors (DSPs). The communications topologies have included hypercubes, various meshes, and multi-drop busses.

The applications cover avionics, signal and image processing, controls, and communications. The algorithm characteristics include both data dependent and independent operations, with periodic, real-time requirements as well as acyclic, event-driven requirements. Some applications have statically determinable dependencies, whereas others require dynamic mode changes. The number of software tasks ranges from tens to hundreds, and messages between tasks, as well as system input/output messages number from the hundreds to tens of thousands, to hundreds of thousands in some instances.

## Goals

For the systems to which we applied our hardware/software co-design methodology during the early stages of system design, several significant facets of the system design needed resolution. These VHDL performance models were constructed to supply the information necessary for trading off likely alternatives, and for validating earlier decisions. Moreover, the models were to establish budgets for subsequent levels of design, such as amount of time allowed for operating system overhead or communication latency.

## Challenges

Co-design and performance modeling are still relatively new and state-of-the-art technologies, especially compared to the rest of the EDA industry which is quite comfortable with the concepts and processes of RTL design. This immaturity is inhibiting its widespread acceptance, similar to the early days of RTL design and simulation. In particular, libraries are scarce, tools are not terribly supportive, and there is a lack of appropriate standards facilitating interoperability [8].

Part of the education process is establishing reasonable expectations of what information can be garnered from system level simulations. These simulations and analyses are intended to provide guidance and help establish budgets. It is not worth getting these high level, early models to 1% accuracy when other factors, such as algorithm or compiler efficiency, can have effects several orders of magnitude larger. The following lists some major contributors to overall performance for systems with significant portions of the functionality captured in software:

- **Resource capabilities** can be driven by non-functional influences. Power, die size, packaging, and other limitations, as well as model year upgrades and product availability might invalidate earlier decisions about the hardware architecture. System level hardware models must be flexible and capable of rapidly accommodating these changing capabilities.
- **Algorithm efficiency** is a major factor in overall system performance. If the algorithm undergoing evaluation is prototypical, only limited conclusions are possible regarding its computational and communication requirements.
- **Tool efficiency (compiler, assembler, etc.)** can drastically affect system performance, from

memory access patterns to processor efficiency and utilization. One point we continue to stress is that accurate models are only possible when the compilers, application code, and hardware are at the same level of maturity. Instruction-set architecture (ISA) processor models provide limited value when support tools are immature.

- **OS/RTS overhead** is an often overlooked contributor to system behavior. The OS or runtime consumes its own resources, as well as influencing or resolving resource contention issues for other tasks. Some recent projects demonstrated that a more enlightened approach to scheduling and algorithm partitioning resulted in significant increase in throughput and decrease in latency, without necessitating changes in the hardware design [9].

## Results

A typical performance-level design cycle considers the macroscopic effects of interactions between multiple claimants contending for limited resources in a particular design scenario. The model simulations are designed to provide performance estimates and are typically tuned to be within 10% of the actual value. More precise refinement is possible, but such detail tends to be swamped by variations in the compilers, algorithm changes, hardware upgrades, etc. Information gleaned from the simulation includes various types of latencies, utilization, efficiency, and throughput numbers for software tasks, operating system overhead, communications, and storage elements.

Each simulation is designed to take between 5 and 30 minutes wall-clock time on moderate to high-end workstations. With current simulation technology, faster simulations require significant code artifacts, which detract from the model flexibility. Flexibility and maintainability are of prime importance, since the quality of the eventual design can correlate to the breadth and quality of the design space exploration.

The effort applied to performance modeling activities is of course situation dependent, but we have found that several weeks of focussed activity can result in a comprehensive model capable of significant tradeoff studies such as topology, number and type of processors, system sizing studies, resource allocation, and routing. Detailed studies about communications protocols, buffer allocation, software dependencies, and scheduling are also possible, many of which require only a few additional hours to construct.

## 3. Hardware-Software Co-Debug

According to Webster's [4], **debug** is "*to detect and eliminate errors in or malfunctions of...*" which is precisely our task when we have an HDL model ready for synthesis and are verifying that it performs as expected against some specification.

As our systems become more complicated, we find that even the giant chip manufacturers release flawed products. Consider, for example, microprocessors. Time-to-market concerns can force fielding of products not fully verified. However, those who build such systems realize that the meaning of "100% tested" remains open for debate, especially considering the current state of requirements trace-ability and test coverage tools, and the system complexities. For instance, did the specification require that *all* possible combinations of floating point operands yield products accurate to the same precision?

One technique to provide better functional coverage is to simulate significant portions of the application

software executing on hardware models before the hardware is fabricated or released. In some notable cases, vendors simulate launching and running an operating system such as Unix on the hardware simulation, and execute some test applications in software. This has a significant impact on traditional design cycles, since it requires that the runtime and application software is complete, as well as compilers, linkers, and loaders. For model year upgrades, this may not be quite so serious, but for new, high-performance embedded products, postponing hardware fabrication until the software is complete can be a significant change from traditional processes.

## **Example Systems**

Honeywell has been actively using VHDL for RTL and other design abstractions for ASICs and FPGAs since VHDL first became commercially available. The chip gate-counts range from the low thousands to tens-of-thousands of gates and include processors, memories, and communications devices. Board-level integration varies from one ASIC to several tens of ASICs per board.

The implementation of one recent communication and control system has at its core an ASIC consisting of a fully custom processor core surrounded by several communications peripherals. This particular device is for harsh and demanding environments, requiring high performance (speed) as well as ultra-low power consumption. Area is not a primary driving factor. Since current synthesis approaches do not cleanly favor low power development at an abstract RTL level, significant manual effort was devoted to customize the design for power conservation.

This device was previously prototyped in a different technology and incorporated much of the functionality, though not all the performance nor low power capabilities. This prototype device has been used for some time for developing software written in C. These software layers incorporate much of what can be considered a real-time executive and communications protocol stack.

## **Goals**

Given the above ASIC design, our goal was to build a board-level model, "boot" the system software, and bring up a representative application which would exercise the various communications interfaces and on-chip peripherals. The intent was to demonstrate system functionality before the designs were cast in silicon. We also intended to provide feedback to the hardware architects and designers to change design features as necessary to enhance usability.

## **Challenges**

This effort faced several typical, yet daunting challenges:

- The schedule was extremely aggressive. Early results were necessary to facilitate design changes before the design passed on to fabrication. However, there was only a small window between when the hardware was sufficiently complete so that a test of software running on it would be meaningful, and when the design had to be released.
- This verification activity was concurrent with ongoing hardware design. The designers were focusing on optimization and intended to make non-functional changes from the high level users' perspective. Not surprisingly, this didn't always happen. Many changes were forced by nasty synthesis limitations. As a result, some modeling artifacts were inserted to placate synthesis and test, which then rippled through to affect the software.

Debugging those ripple effects was time consuming, since a previously function element might not normally be activated for several hundred thousand clock cycles.

- The system was both large and represented at a low level, resulting in many VHDL design objects. This severely impacted simulation performance. Compilation was fairly inconsequential, since we made extensive use of makefiles to limit the impact of design changes. Simulation time was another thing, and the average ratio of wall-clock time to simulated time was about 30000:1. This was exacerbated when debugging the software behavior, since that generally required significant state, such as register values, to be dumped to trace files for post analysis. When discrepancies were encountered, a previous checkpoint was reloaded, the number of saved signals was increased, and simulation recommenced until the fault could be uncovered.

While this was a fairly vanilla debugging process, the file size caused consternation, especially since the vendor-supplied trace tools were at best cumbersome for examining traces larger than 60MB. We ended up writing a fair number of Perl data reduction routines.

- The design and verification team was spread across the United States, two different companies, two simulation vendors, and at least three different simulation platforms and operating systems. The design itself was undergoing several major changes in different sections, so configuration management was a major problem, possibly *the* largest problem facing the whole development team. Only one simulation tool was even reasonable for running on different platforms. No useful configuration management tool was found that could cope with this disparity.
- The software consisted of several thousands of lines of compiled C software which had its own ongoing development process. Some of the identified errors were in the software, primarily in the initialization portions. The ongoing hardware optimizations made that worse, since many of the hardware changes involved chip initialization.
- Another recurring problem was the ability to save simulation state. This particular system is itself a peripheral device, controlled and configured by a general purpose processor. The test bench consisted of the board level design with its own program store and a stimulus engine which read external scripts for stimulus. Simulators can be finicky about using external test scripts via standard VHDL file I/O, especially when changes are made to those scripts and tests are resumed after some checkpoint, rather than rerun from the beginning. This is especially important when such runs take 8 hours or more.

## Results

This effort met with mixed success. Design defects were uncovered, but each error took many labor hours to uncover. Moreover, the design itself is so flexible that software patches would have been sufficient to remedy the majority of the problems. In the end, the primary class of problems this effort identified were in documentation and initialization, both of which could have been fixed after the chip went to fabrication, without placing the software functional verification effort in the critical path. It is not clear whether the time spent bringing the software and hardware together would have been more effectively spent expanding the existing test suites, a view supported by the fact that several of the identified errors had already been discovered by the hardware design team, and had not yet been propagated to the software testers.

Furthermore, the ability to affect significant hardware changes to enhance usability tended to be suppressed at that late stage of the development cycle. By that time, the effort is primarily concerned with identifying and eliminating defects, which is why this phase is considered "co-debug" rather than "co-design." While software debuggers seamlessly integrated with VHDL simulation environments are seriously cool and merit strong consideration, the cost and schedule tradeoffs remain unclear.

One clear benefit is that this co-debug activity reduces downstream (after chip release) hardware/software integration time, which can then shorten time-to-market, if the following criteria are met:

- The hardware, software, compilers, linkers, and loaders are ready to go before or at the same time the chip is released for fabrication.
- Sufficient documentation and configuration control is in place to accommodate any ongoing development in a coherent fashion.
- Personnel, computing, and storage resources are available to do this integration which occurs at a much slower pace than with the actual hardware. Custom accelerators can help reduce this performance deficit, but they come with their own acquisition, training, integration, and maintenance costs.

## 4. Noting the Differences

The following table provides a direct comparison of the performance-level co-design efforts to the RTL-level co-debug activities. The lengths of times of course vary depending on the test. The times listed are typical for the types of tests we regularly perform. The storage requirements are also typical, and it's not an error that the performance models consume significantly more run-time memory. This is primarily due to the static requirements for VHDL, and the performance models trade memory utilization for flexibility.

Co-design vs Co-debug		
Feature	Codesign	Codebug
Simulated Time	.5-3 sec	.05-1 sec
Simulation Wall-Time	5-30 min	10min-8 hours
Run-time Memory	100-150MB	10-20MB
Results Storage	10MB	10MB-100MB
Typical Accuracy (goal)	10%	clock cycle

Hardware/software co-development offer two obvious advantages for the different levels of abstraction :

- **Design cycle or non recurring costs** were reduced with smarter up-front system engineering, which includes performance modeling.
- **Time-to-market:** hardware/software co-debug offered significant time-to-market advantages primarily when the hardware simulation was used as an integration platform before the chips and boards were available.

## 5. Conclusion

The distinction between co-design and co-debug is primarily one of expectations and capabilities. The co-design tools are used early in the product development cycle to aid the exploration and evaluation of alternatives, and can result in major design modifications. The co-debug tools are used immediately near chip the time of fabrication, and help compress integration schedules. The types of errors uncovered during debug generally do not result in major design perturbations.

For systems on which we have brought the hardware and software together for co-debug before chip fabrication, we have always identified defects which were not yet apparent through other testing efforts. It must be reiterated that the co-debug activities help compress integration schedules. However, it is not obvious whether the additional effort expended in co-debug could instead be applied to other hardware test techniques which provide similar or better error coverage. A valuable exercise would be to measure the number and types of defects identified per labor hour by two parallel efforts, one with the co-debug approach, the other with a more formal hardware verification approach.

As for tools issues, we found that while tools are excellent for detailed hardware simulations, they are still limited when combining domains, especially the hardware/software domain. Configuration management, domain representations, and manipulation of enormous results databases were prohibitive factors. Currently, EDA support seems focussed on the latter stages of the design cycle.

## References

1. VHDL System Design Methodology A 16/30 Success Story, M. Hajj, VIUF, San Jose CA, October 1993.
2. Mitigating System Design Risk with VHDL M. Lindsay, VIUF, San Jose CA, October 1993.
3. Statistics and Heuristics Developed from Systems Engineering with VHDL Models, C. Curtis, VIUF, San Jose CA, October 1993.
4. Webster's Third New International Dictionary (Unabridged), G&C Merriam Company, MS, USA.
5. Evaluating Distributed Multiprocessor Designs, T. Steeves, et. al, RASSP II, Arlington VA, July 1995.
6. VHDL-Based Performance Modeling and Virtual Prototyping, C. Hein and D. Nasoff, RASSP II, Arlington VA, July 1995.
7. RASSP Benchmark 1: Virtual Prototyping of a Synthetic Aperature Radar Processor, E. Rundquist, RASSP II, Arlington VA, July 1995.
8. IEEE DASC codesign study group <<http://vhdl.org/vi/codesign/Welcome.html>>
9. Performance Modeling and Virtual Prototyping Multi-Processor Systems with VHDL, Carl Hein, VIUF, Boston MA, October 1995.

### Further Reading:

10. RASSP Technology Insertion into the Synthetic Aperature Radar Image Processor Application, J. Pridgen, et. al, RASSP II, Arlington VA, July 1995.
11. RASSP Benchmark-1 and -2: A Preliminary Assessment, A. Anderson, et. al, RASSP II, Arlington VA, July 1995.
12. VHDL Performance Modeling, F. Rose, et. al, RASSP I, Arlington VA, August 1994.
13. Advanced Multiprocessor System Modeling, J. Shackleton, T. Steeves, VIUF, Boston MA, October 1995.

## Acknowledgments

I'd like to thank Scott Calhoun from Mississippi State for making the initial observation distinguishing co-design and co-debug.