Hardware/Software Co-verification of CDMA ASIC Designs

Siddhartha Ray Chaudhuri – Qualcomm
Russell Klein – Mentor Graphics Corporation

Abstract

Through the development of three generations of code-division multiple-access (CDMA) IC families, we adopted hardware/software co-verification as part of our design process. Our initial goal was to provide the software team earlier access to an executable specification of the hardware for early low-level driver debugging. After using co-verification through the first generation of our chips we realized its benefits in the hardware design process as well. Another interesting side effect of using co-verification was the collaboration of the hardware and software teams at an earlier phase of the design cycle.

In this paper we will describe our experiences with hardware/software co-verification as we applied them to our design process. We will also detail what we believe to be important criteria when selecting and implementing a co-verification strategy, as well as the limitations of this technology and our view of needed future enhancements.

CDMA Handset ASIC

Hardware/software co-verification was used in the development of several generations of our CDMA handset ASICs, known as Mobile Station Modem (MSM). The MSM is a complete, single chip solution for CDMA and FM digital base-band processing for dual-mode (CDMA/FM) cellular telephones. The MSM incorporates the functionality previously provided by several individual ICs. The MSM interfaces directly with the Analog Base-band Processor (BBA) to provide the core functionality of an IS-95 subscriber unit.

The MSM3100 is a 6th-generation single-chip base-band processor with higher data rates and advanced power control. The high data rates are supported by IS-95B compliant infrastructure. This supports Internet applications with fast packet-switched access to databases, e-mail and web browsing. Target applications for the MSM3100 include subscriber units for multi-mode cellular and PCS handsets, and data communications devices. The MSM3100 includes proprietary CDMA building blocks, the ARM7TDMI processor core and several DSP cores integrated onto a single chip.

Hardware/Software Co-verification

Hardware/Software co-verification tools permit software to be executed on a hardware design, while the hardware design is being simulated in a HDL simulator. These tools are available from the major EDA vendors. On the hardware side, the HDL simulator is used to run, control, and debug the hardware design; on the software side an embedded debugger is used to display, and control the execution of the software. Using a co-verification tool also requires a model of the processor in the simulated design. These processor models are generally available from the EDA vendor, but in several cases are now available from the silicon vendors.

All co-verification tools achieve reasonable levels of performance, with respect to the software, by hiding bus transactions from the logic simulator. Bus cycles modeled in the logic simulator run at logic simulation speeds, about 10 cycles per second. While hidden cycles can be processed at around 100,000 cycles per second. The “hidden” bus cycles are generally uninteresting activity that does not impact the operation of the hardware, such as instruction fetches and software data space references. [1][2]

When we began to look into co-verification tools, they were not yet a proven technology. EDA vendors were making bold claims about the capabilities of their tools. We were skeptical about the performance of these tools and very concerned about the side effects of removing the “hidden” bus cycles from the logic simulation. During simulation of our CDMA design the timing of the presentation of events from the processor is critical to proper operation.

The Drive for Co-verification

During the development of the MSM2x series of products, the level of complexity in both the hardware and software design was increasing significantly. At this time, the hardware and software designs were developed and verified independently. The processor in the MSM2x was an Intel 80186. Beginning with the MSM3000 series, we switched to the ARM7TDMI core, and started to integrate more functionality into a single chip. In earlier projects, the software team could run at least some of their software on a prior design, since the processor and much of the hardware were similar. They could get some software debug activities completed in this manner. The final software debug phase would have had to wait for the chips to be taped out and a prototype design built.

Changing the processor meant that all of the software would be changing. Even high-level application code written in C would be compiled through a new compiler tool chain. Low-level drivers would need to be mostly re-written. With all of these changes to the software, we wanted to give the software team as much of a head start on debugging as possible. Co-verification was chosen as a method of allowing the software team access to a functioning and usable hardware design before prototypes ever could have been available.

Evaluating Co-verification Tools

When considering a co-verification tool, our biggest concern was the ability of the tool to handle real world designs as complex as we were building. EDA vendors would demo their products on very simple circuits and software. We were also concerned about proper operation with "hidden" bus cycles. Without being able to take advantage of this cycle "hiding" the performance of the simulations would be too slow to be useful. Even with cycle "hiding" performance was a big concern.

To address our concerns about the viability of co-verification we decided to have the EDA vendors "test drive" their tools on our design. Although this meant setting up and learning to use several co-verification tools, a very time and labor consuming process, it was well worth the effort.

The co-verification tools from the various vendors appear, at first glance, to be quite comparable. However, running the tools on our design gave us an appreciation of the subtle differences between the tools. As previously mentioned, co-verification tools gain performance by "hiding" bus transactions from the logic simulator. These hidden transactions are processed against a memory array that is not contained in memory instances the logic simulator. Both V-CPU from Summit and Eagle-I from Synopsys approach this problem by partitioning the memory space of the system into software and hardware regions. Accesses into the "software" memory regions are "hidden" and accesses into the hardware regions are run in the logic simulator. Seamless-CVE from Mentor has a similar concept of hardware and software memory, but stores the data in what they call a "memory image server" for both hardware and software regions.

The obvious benefit of Mentor's approach is that the hardware and software partition can be changed while the simulation is running. There are a couple of not so obvious benefits that result from this difference as well. The main benefit is debug visibility. With V-CPU and Eagle-I the memory partitioning limits debug visibility. The hardware simulator can only see, and therefore can only debug, the hardware partition of the memory. Likewise, the software debugger only has access to the software partition. With Seamless CVE, both the logic simulator and software debugger have visibility into the entire memory space of the system being simulated.

The other benefit Mentor has is the ability to turn off the cycle "hiding." The architecture of the memory image server allows the partitioning of memory regions between hardware and software to be changed. In fact, it can be changed during a simulation run. By changing this partition so that all memory is defined as hardware memory, you can turn off the cycle hiding aspect of the simulation. This means that all bus cycles will be driven through the logic simulator. Effectively, the co-verification processor model is turned into a full functional processor model for a portion of the simulation. This is critical during hardware operations where the presentation of events from the processor is sensitive to timing.

Mentor's approach has its share of drawbacks. Seamless CVE is a bit more complex to setup, when compared with its competitors. This extra setup is required by the memory image server. Another annoying limitation of Seamless CVE is its inability to run across the network; that is run the software debugger on a PC while the logic simulation runs on a workstation, with communication across a network. Most of our software development is done on PCs, and our hardware design is done on workstations. So this would be a far more natural, and desirable, configuration. We concluded that the benefits of Mentor's approach outweighed the limitations.

MSM3000

Once we had selected a co-verification tool, we set about applying co-verification to our design process. At the time, co-verification tools were very new technology. Our test drives of these tools showed us that these products were immature. We were concerned about having our design process rely on tools that weren't quite ready. We chose to apply co-verification first to a hardware design that was already known to be good. Using this stable platform, we could explore the capabilities of the tool, without worrying about impacting the project. This also gave us confidence that when a problem was encountered that it would not be the hardware design that was generating the problem.

With a known good hardware design, we modified the design to work with the co-verification tool. This involved three steps; first replacing the processor model, replacing the memory models, and creating a memory map. To replace the processor model, we simply created a new architecture for the ARM7TDMI entity. Since our full functional model of the ARM processor and the co-verification model were both created by ARM, the pin names on the existing entity and the new architecture matched, making the job straightforward. The same approach was used to replace the memory models. Creation of the memory map was a simple process using the co-verification tool'' GUI.

Co-simulation

With the design properly configured for co-verification, we were ready to begin co-simulation. On the MSM3000, our goal was to see what was really possible to accomplish with co-verification on designs as complex as we were building. We started with small snippets of code, focusing on low level driver code that has a high degree of interaction with the hardware.

With no hidden cycles the performance was similar to logic simulation. Which we found generally not useful for the execution of software. With hidden cycles we were achieving execution rates of approximately 1 microsecond of simulated time per second. This was measured using ModelSim as the logic simulator. Using the IKOS accelerator, we measured performance between 10 and 20 microseconds of simulated time/second. While the performance is excellent when compared with logic simulation speeds, it is quite slow when compared with the execution rates that our software team is used to running.

The performance of co-verification was not an issue when applied to low level driver code. These were generally small, short tests that exercised a small subset of the system. As we included more code into the tests, performance became more of an issue. While it would be nice to run application code in a co-verification environment, at this time it is not practical. Some people may also argue that that would be an improper use of co-verification. The performance of the system in the co-verification environment is just not sufficient to load an application and run. However, we found that through a couple of techniques, we were able to test limited higher-level functionality using co-verification.

First, the overall performance of the system running in co-verification is highly dependent upon the ratio of “hidden” bus cycles to bus cycles run in the logic simulator [3]. By constructing tests that maximize the number of hidden cycles, performance can be enhanced significantly. We found that a number of hardware operations that we were running, due to the complexity of our CDMA design, would not function properly with hidden bus cycles. Because of this, we needed to change the partition between hardware and software memory during the simulations to get maximum performance and correct operation. Using breakpoints and automated scripts in the logic simulator, we were able to automate the repartitioning of the memory.

Another technique that allowed us to start looking at higher-level functionality was some simple pruning of the higher-level code. For example, let us consider the case where the actual product will power up and run a ROM based bootstrap loader that copies the application from ROM to RAM and then launches the application. By simply loading the application into RAM directly at time 0, we skipped this time consuming step in the simulation. Further examples would delve into proprietary information, but the concept should be clear. Simple changes to the software can result in significant performance gains without impacting your verification.

Pilot Acquisition Achieved

On the MSM3000 we were able to run a significant amount of application level functionality. We are unable to provide specific details due to the proprietary nature of this data, but we were able to simulate the cellular phone, with the MSM3000, powering up and "acquiring" a pilot. Initially, we ran a "cheated" acquisition, where the digital base station model emanates data at the optimum time when the simulated phone can acquire them. Ultimately, we were able to run a full blind search and acquisition in less than 8 hours of simulation. This was far more functionality than we ever anticipated being able to simulate.

MSM3100

Our work on the MSM3000 chip showed us that co-verification was a viable technology for use on designs as complex as our CDMA chips. In fact, we realized our goal of getting the software team earlier access to the design. In our second generation of this family, we used co-verification while the hardware was being designed. The hardware team would get to a point on the design where it was functional. A “snapshot” of the design would be taken and given to the software team. The software team would run their code against the hardware design. This would allow them to begin the debug process on the software and to report problems found in the hardware to the hardware team. As hardware problems were uncovered, the hardware team would fix the problems and "snapshot" a new version of the hardware. The process of incremental hardware releases kept the hardware and software teams progressing in parallel.

Hardware Problems Found

The software team did uncover problems in the hardware design, and the hardware and software teams began to work in a more collaborative manner. One of the hardware problems found by the software team was in the CDMA demodulator block. The CDMA demodulator block is a part of the hardware design that can be run in a variety of the modes. The hardware team would have eventually tested all combinations of operating modes, but the software team, while running their software uncovered a combination of modes that was not working.

Another problem uncovered by executing software in the co-verification tool was a problem in a clock divider. The MSM3100 implemented a new clock generation and division circuitry that was more efficient than the MSM3000. The clock divider, due to a logic error was dividing down a clock to an incorrect frequency. While the problem could be seen in the logic simulator, the software execution made the problem obvious. The hardware was just not responding to software at appropriate times. These are just a couple of examples of the types of problems that we discovered by running co-verification

These problems, and others like them, were fixed by the hardware team and delivered to the software team in incremental hardware "builds." In our experience, one of the most powerful results of using co-verification is that it makes early collaboration of the hardware and software teams possible. In traditional development cycles, the hardware and software teams get together only after the hardware is in the lab, *i.e.*, after a prototype is built. Once the prototype is done, making any changes to the hardware is expensive and time consuming. As a result, in many companies, the team members sometimes end up taking on an adversarial role. With co-verification, the teams can work together when a problem is found, and engineer a solution – which is what they are good at – instead of avoiding blame. By creating the hardware and software in parallel, the hardware team got feedback from the software team. And the software team was able to evaluate the hardware when there was still a possibility of changing the hardware to accommodate any required changes. With co-verification, the hardware can go through many more design iterations than would have been possible with a physical prototype methodology. It is easy to envision a time when these iterations will be used by the design team to explore the design space to engineer the optimal solution.

Future Co-verification

Through our use of co-verification over the past few generations of the MSM3x family we have realized significant benefits. Its stability, maturity, and functionality have improved noticeably since its introduction. However, we still see room for improvement. The performance of co-verification is not sufficient to simply load and run large applications, or even meaningful regression tests on low-level drivers. However, since higher-level code is largely isolated from the hardware, this is not a glaring limitation. We were able to test certain limited higher-level functionality. The next level of performance improvement should result in the ability to comfortably run meaningful regression tests as well as large applications.

The impracticality of regression tests showed up in our inability to exercise our Viterbi decoder in co-verification. A Viterbi decoder is an algorithm for eliminating background noise from signals. It is used to improve the quality of the received signal. It processes data on a packet basis. Processing one packet takes 20 milliseconds of real time. Even with the co-verification optimizations we were able to run 1 packet in about 6 hours. To fully test all combinations and corner cases, we need to run the algorithm on about 1000 packets, which is impractical on today's co-verification tools.

Conclusions

Our initial investigations of co-verification on the MSM3000 proved that it is viable technology for low-level device driver interfaces. We then decided to use co-verification on the development of the MSM3100. As a result, we estimate that we were able to save about a month on our handset software delivery schedule. Initially, we realized the benefit of giving the software team earlier access to the hardware design. On our second design, we uncovered hardware design errors as well as accelerated our software debug process. Also of significant value to the design process is the collaboration of the hardware and software design teams, as they work together for the first time engineering real solutions to hardware/software integration problems. In past projects we would have patched or worked around these issues in a reactive manner.

Co-verification tools are costly, but their price needs to be considered in the context of the benefits that they deliver. For projects as complex as ours and with significant time to market pressures, the price is worth it. If co-verification enjoyed further speed improvement, it will certainly become more attractive to a wider range of verification interests.

Looking forward to new projects, we will continue to use co-verification. We are also developing other verification methods to address verification tasks where higher levels of performance are required. But we believe that the verification of our low-level timing-critical driver code and its interaction with hardware requires the debug visibility and accuracy that are afforded by co-verification.

References

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

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

[3] M. Stanbro, “Getting to the Bottom of HW/SW Co-verification Performance Claims,” Computer Design, Vol 37 No 12, December 1998, p65-67