Hardware/Software Co-Simulation

by Russ Klein
Mentor Graphics Corporation

Abstract

Increasing time to market pressures, greater design complexity and increasing software content are driving systems designers to look towards new ways of performing system validation. Hardware and software design teams are using separate tools to develop different components that will work closely together in a system. Combining these design environments holds the promise of finding problems earlier in the design cycle. The key to a successful integration of these technologies is to provide sufficient performance to run reasonable amounts of software.

This paper describes a system which integrates an event driven hardware simulator with an instruction set simulator. This system allows dynamic partitioning of the code and data space between the hardware and software simulators. This capability provides performance sufficient for running software without sacrificing the accuracy required by the hardware simulator.

Introduction

Systems designers are facing two trends in the electronics industry: increasing design complexity and increasing software content in those designs. These trends are compounding the problems related to the integration of the hardware and software. Today, the hardware and software that make up a complex embedded system need to be validated as a system. Hardware simulators allow the hardware to be validated, but they do not have sufficient performance to allow software to be validated. Running software on a design in a hardware simulator will typically run at about 10 instructions per second. The consequence of this is that most designs are performed in a serial fashion, that is, the hardware is designed, validated, and prototyped. Then the software is debugged and integrated on that prototype. The problem with this approach is that once the hardware is prototyped any changes are expensive and time consuming to implement.

A hardware software co-simulation environment would alleviate this problem. There are a number of systems that achieve higher performance by moving some or all of the code and data space of the program into the software domain, partitioning the design. In this manner, the memory references related to the reading and writing of data in the software domain are not simulated in the hardware simulator. Removing this work from the hardware simulator can greatly improve the performance of the system. The problem is that the effect of those memory references cannot be seen in the hardware simulation. While most of the time the presumption that code and data references are uninteresting is correct, there are times when all bus activity is significant from the perspective of the hardware simulation.

We have developed a system, which we have internally called "Miami", that maintains comparable levels of performance but provides greater simulation accuracy. It does this by allowing the designer to dynamically repartition the memory. When memory references and their effect is uninteresting, the designer can map those memory references in the software domain. At a later time in the same simulation, those memory references can be re-mapped into the hardware domain. This allows the designer to make trade-offs between the level of performance and the level of detail in the hardware simulation. This paper chronicles our experiments in combining hardware and software simulators as we worked towards developing this system.

Hardware/Software Integration

The typical electronic system development project can be divided into three major phases: system level design, design and implementation, and system integration. During the system level design the overall architecture of the system is determined, and the system is partitioned out into hardware and software components.

During the design and implementation phase of the project, separate organizations address the hardware and software components of the design. The hardware and software design and implementation efforts typically start at the same time and ideally end at the same time. In reality, the software team is limited in what it can complete without a physical prototype of the target design. They can complete the high level design of the software and do some of the algorithm development and validation. Certainly, some of the software will be independent of the underlying hardware, and can be completed and tested outside of the context of the target system. But the much of the debug and integration work cannot even begin until a hardware prototype is available.

In theory, the integration phase is a 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 definitions, out-of-date specifications, and poorly communicated changes.

As problems are uncovered in the integration phase, developers look for the fastest and most inexpensive ways to fix those problems. Given time and cost required to change the hardware, problems are often fixed by changing the software. Unfortunately, only so much can be done in software and the functionality or performance goals of the product may be compromised with a software patch. Occasionally, there may be programmable logic elements in the circuit that can be reprogrammed to work around hardware problems, but again, this is working around the problem, not fixing it.

To fix these problems without incurring the time and expense of changing the hardware, these problems must be discovered before hardware prototypes are created. To do this the software must be run on the hardware while it is still in simulation, that is, a virtual prototype. Two things are necessary before virtual prototyping can be accomplished. The first is the ability to simulate the hardware at speeds sufficient to make software execution feasible. In most cases, this means that the overall simulation performance must be increased by a factor of at least 1000 over the current execution speeds of hardware oriented simulation products. The second is the need to bring the debug and development environments for the hardware and software closer together. Hardware simulator waveform traces do not provide a natural way for a software engineer to debug high level code. The original source form for both the software and hardware must be maintained within a single unified debugging environment.

Current Solutions

Today, the hardware designer has a hardware simulator, such as VHDL or Verilog, 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, but just models 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 against the design being simulated. In this way the hardware designer can construct a test that would, for example, write to and then read from each of the memory components in the design.

To run software on the simulated design, the hardware designer would need a fully functional model of the processor. However, writing a program that completely emulates the behavior of, say, a Pentium is a nearly impossible task. To obtain a full functional model of a processor, a device called a hardware modeler is most often used. A hardware modeler is a device 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 being executed on the simulated design. This method is being described here because it will be used as a point of comparison for some of our results later in this paper.

The software designer has a compiler and a debugger running on a general purpose computer for performing software design and algorithm development. Often, an instruction set simulator will be used for running assembly and machine code and for determining software performance at a gross level. These instruction set simulators often have facilities for generating interrupts and I/O data streams to simulate the effects of 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.

The hardware simulator and the instruction set simulator appear to provide an interesting opportunity for integration. The remainder of this paper summarizes our efforts in performing such an integration that meets the requirements of virtual prototyping.

The Reference Design

We set out to determine the feasibility of constructing a system that combined hardware and software simulation capabilities, while providing a reasonable tradeoff between performance and accuracy. We decided to build a prototype system and characterize its performance and functionality over a variety of conditions. To accurately assess the performance, we would need a target design that could be modeled both using this new approach and using traditional modeling techniques. Ideally, we would also have a working physical implementation that could be used as a reference for functionality and performance.

In addition, the target design needed to meet a number of criteria: we needed to be able to obtain or create a bus functional model, a hardware model, and an instruction set simulator (or as it is referred to through much of this paper, an instruction set model). We also needed access to a reasonable set of realistic software for the target design. Given the requirement that realistic code be available for the target system, it was obvious that a general purpose computer would best meet our needs. With our resource constraints, we decided to model a Z80 CP/M computer. Creation of an instruction set model able to be integrated into a hardware simulator could be completed fairly quickly and easily. The team had access to a hardware model for the Z80 for performing traditional event driven simulations. The design was simple enough to be easily understood and implemented in simulation. Also, one of team members had a Z80 based CP/M machine still in working order, with significant amounts of commercial software.

The Instruction Set Simulator

The first task was to create an instruction set model that could be integrated into a hardware simulation environment. The primary motivation for writing this on our own was that in the context of the hardware simulation the instruction set model would need to be advanced by one half clock cycle at a time, returning the thread of control to the hardware simulator between clock events. Writing and validating the instruction set model took about 6 weeks. The instruction set model contained rudimentary debug capabilities, displaying ranges of memory and contents of registers, etc. Running on an HP-735/99 workstation it ran at about 300,000 instructions per second. The actual CP/M computer runs at about 40,000 instructions per second.

The next task was to get CP/M up and running in the instruction set model. For this particular computer the operating system was broken up into 4 sections. One that was contained in a 16 K ROM in the computer, the others were stored on the system tracks of the boot floppy disc. The operating system executable images were extracted off the CP/M machine and loaded onto the instruction set model that we had created. The instruction set model was modified so that it could properly handle keyboard input and screen output. We placed traps at the operating system entry points for these functions and redirected them through the standard UNIX input and output streams. In this manner we were able to "run" the virtual CP/M machine using the keyboard and screen of the HP workstation.

A cross compiler would have made life easier at this point. We created a number of simple assembler test programs that made calls into the operating system. The goal was to validate that the instruction set simulator was correctly running the operating system. The programs were assembled and run on the CP/M machine and then run in the instruction set model on the HP workstation and the results compared. After about 2 weeks of debugging and testing we were satisfied that the operating system was being run in the instruction set simulator correctly. Finally, we added the Microsoft BASIC language interpreter to the simulation. This gave us the ability to quickly generate and run a wide variety of programs on the virtual target. Being an interpreted language, it would not require us to go through the compile, link, and load steps that a compiler or assembler would require.

Performance Results

At this point we integrated the instruction set model into a bus interface model for the Z80. The bus interface model is the interface between the event driven simulator and the instruction set model. It translates bus cycles and other information from the instruction set model into pins changes in the hardware simulator. It also captures interrupts and other input pin stimulus from the circuit in the hardware simulator and passes them to the instruction set model. It has much of the same functionality of the bus functional model described earlier, but it does not get the bus cycles to be generated from a script, instead it gets them from the instruction set model. In doing this we moved the memory image for the target system out of the instruction set model and into the memory instances of the circuit in the hardware simulator. All memory and I/O accesses for the instruction set model were actually run against the virtual prototype in the hardware simulator.

To evaluate the performance of the system we ran the initial 10,452 instructions of the Microsoft BASIC interpreter. It is difficult to say exactly what the software is doing during this start-up period, as we did not have the source code for this program. We do know that it makes several hundred operating system calls, initializes some data structures and prints the copyright banner for the product. The copyright banner consists of 106 characters, and consumes about 5000 instructions. The reason that we limited our performance test to the initial 10,452 instructions is that the hardware modeler in our reference design limits our total simulation to 256K evaluations of the processor hardware model. Without going into too much detail, a hardware modeler is limited by physical pattern memory as to how many model evaluations can be performed in a given simulation. These 10,452 instructions consume approximately 120,000 clock cycles, the processor model is evaluated on each clock edge, giving us 240,000 processor model evaluations in our simulation.

We measured the performance of the integrated hardware simulator and instruction set model to be 454 instructions per second. (10,452 instructions in 23 seconds). Keeping the software configuration exactly the same and replacing the instruction set model/bus interface model with a hardware model for the Z80, we re-ran the simulation. This configuration is typical of what engineers are currently using the run software on simulated hardware. In this configuration we measured the performance to be 1.3 instructions per second (10,452 instructions in 8040 seconds). This is consistent with the performance that is seen by most hardware engineers using this technique to run software on hardware designs. Typically, CISC processor designs run at 1 to 3 instructions per second and RISC processor designs may run up to 10 or 15 instructions per second. It is the clocks per second in the hardware simulation that remains somewhat proportional to design size and complexity.

These performance numbers are interesting, but somewhat academic, as the hardware design is approximately 15 years old. We increased the hardware content of the design artificially by adding approximately 140,000 ASIC library instances to the design. This is comparable to adding about 500,000 gates of custom hardware to the design. This brought the design to a level of complexity comparable to the high end of what is being designed today. When we did this, the performance of the system using the instruction set model was reduced to 32.5 instructions per second. The performance of the hardware model system was reduced to 1.28 instructions per second. These results were completely expected. From this experiment we validated our assumption that a simple integration of an instruction set simulator and a hardware simulator is not sufficient for addressing the performance requirements of virtual prototyping.

Even with a more efficient method of modeling the processor, the hardware simulator remains the bottleneck. Event driven hardware simulator performance is determined by the number of events that need to be processed in the simulation multiplied by the number of events per second that the simulator can process. This is, of course, an oversimplification, but to have a significant impact on the overall simulation performance either the number of events needs to be reduced, or the speed at which the events are processed must be increased.

A Fixed Partition System

Our next experiment was to move all of the code and data space for the target design into the instruction set model. We had read about a number of systems that have taken this approach to increasing the performance for this type of integration. At the start of the simulation the designer defines the addresses of memory space that will be modeled in the hardware simulator and the addresses that be modeled in the instruction set model. If a memory access is in the space defined to be in the instruction set model, then the access is completed, without advancing the hardware simulator. The hardware simulator is only advanced only when a bus cycle is used to access some portion of the design simulated in hardware. This approach effectively reduces the number events that the hardware simulator must process for a given simulation. In this configuration our simulation of the virtual CP/M computer ran in excess of 10,000 instructions per second.

Although the software performance in this case was impressive, the effect on the hardware simulation was somewhat disconcerting. Instruction fetches and most data references were completely eliminated from the hardware simulation. Although code fetches are often uninteresting, during certain critical points of simulation they can be extremely important. This type of optimization works well if the microprocessor is in control of the entire system or the synchronization of the system is accomplished through polling mechanisms initiated by the processor. The example of the CP/M microcomputer actually worked quite well, however had the keyboard been serviced by hardware interrupts, the system may not have worked at all. Further, removing some of the memory accesses from the hardware simulation would significantly distort the operation of the system through performance critical code segments, such as a device driver. Fundamentally, the problems that are most interesting to observe with a tool like this are the ones that are distorted the most by this type of optimization.

We realized that most of the time, memory accesses are uninteresting. That is, exercising them on the virtual prototype provides no additional insight into the correctness of the design. However, at certain times in the execution of the virtual prototype, the effect of any bus cycle from the processor can be vitally important. A successful tool will allow the designer the dynamically define what bus cycles are executed against the design and what bus cycles are optimized.

Miami Prototype

Our next experiment was to build a memory image server. This program was designed to hold the memory image for the design and serve it to either the hardware simulator or the instruction set model. Keeping the data for the target design in one place ensures that there will not be out of date memory contents in either the hardware simulator or the instruction set model. In the bus interface model we defined 3 methods for the access of a bus cycle. The first is to access the memory image server. This access method makes the memory reference look like it is contained internally to the instruction set processor, no hardware simulation takes place on this type of access. The second is to access the data from the memory image server, and drive a null bus cycle. In this case, the data transfer for the bus cycle is performed quickly, but the hardware and software remain synchronized. The null bus cycle is a cycle that executes the pin changes to acquire the bus, but does not transfer data across it. The null cycle holds the address and data buses for amount of time that it would have taken to complete the data transfer. The third access type is to run the bus cycle completely in the hardware simulator on the virtual prototype. The bus interface model was enhanced to allow the designer to run these different access methods on different address spaces. For example, memory access from 0x0000-0x00FF and 0xC000 to 0xFFFF could be run as complete bus cycles, as this is where the device drivers and the operating system are located, and addresses from 0x0100 to 0xA000 could be run in the most optimal fashion, as this is the application code. This "access map" can be dynamically redefined as the simulation proceeds. As critical sections of the simulation are reached, the map can be redefined to drive all bus cycles and pin changes against the hardware simulation.

The performance of the CP/M machine over our standard section of code ranged from 452 instructions per second for a configuration of running all of the bus cycles through the hardware simulator to 15,867 instructions per second executing only the I/O cycles to the screen and keyboard. While running null bus cycles against the target design to keep the hardware and software synchronized, but accessing memories using the memory image server we attained 1,534 instructions per second. Even with 500,000 gates of ASIC modeled as part of the hardware simulation, we were still able to reach speeds of 1,307 instructions per second. An RTL representation of the ASIC would have resulted in even greater performance. In other words, for today's large designs this represents an increase in performance of over 1000 times faster than current hardware simulation technology.

External Interfaces

Our next step was to generalize this solution, to allow it to be tested against a wider variety of designs. One of the critical problems that we faced was the creation of instruction set models. It seemed foolish to recreate these for each microprocessor or controller that we were interested in modeling, when they are readily available commercially. The crux of the problem was that we needed the instruction set models to advance by one half clock cycle and then return the control thread to the hardware simulator. Most instruction set models are written to advance at least one instruction at a time, and to keep the thread of control. Existing APIs did not provide enough information or control to allow the type of integration that we required.

One of the ideas that we had for solving this problem was to run the instruction set model as a separate process, using an IPC communication channel to pass bus cycles and data between the instruction set simulator and the bus interface model. In doing this, we were able to separate the clock cycle accurate requirements of the hardware simulation from the typically bus cycle accurate instruction set models. Interrupts processed on bus cycle boundaries are handled rather naturally, as a return status from the bus cycle as it is completed. Additional mechanisms were put in place to handle those interrupts that need to be serviced during the processing of a bus cycle. The development of this API was central to the continuation of the project. Interestingly, it is very different from any API that we had seen in any instruction set model. This API may be the topic of a future paper, and clearly represents an opportunity for standardization.

The concepts of this API were validated by taking the gnu Z-8000 instruction set model and modifying it to run with in our virtual prototyping tool. These modifications were successful and we had a simple Z-8000 design up and running as a virtual prototype in a few weeks. This integration did not require a wholesale rewrite of the instruction set simulator, but required some simple code changes in several key areas. Our Z80 example was also rewritten to use this interface. The performance using this API was comparable to the performance numbers reported earlier.

The latest and most ambitious development in this project was to incorporate a commercial instruction set simulator and symbolic debugger into our system. Since the API requires modification of the instruction set simulator, we would need the cooperation of a software tools development company. Our idea was presented to several embedded software tools companies. Through a business alliance, we ultimately performed this integration with the X-RAY simulator and debugger from Microtec Research Incorporated. The integration proved to be fairly easy and is currently being tested.

Our next step in the project is to run this tool on several commercial designs. We are working with various partners to acquire these designs. The results obtained from the work we have done so far has been very encouraging. But the proof will be running this tool on state of the art designs in a commercial environment. By the time this paper is presented, we expect to able to discuss some of our results in this area

Conclusion

Trends in the embedded systems market: time to market pressures, increasing software content, and increasing design complexity are driving designers to look at new ways of validating their systems. We all know about the studies that show how much more expensive it is to fix a problem later in a project. Virtual prototyping offers the promise of finding integration problems much sooner in the design cycle. Integrating hardware and software design tools will provide this functionality. 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 obvious performance optimization, removing code space from the hardware simulation, allows functional verification of some types of systems, but is quite limited in terms of hardware debug and analysis. Our system, that gives the designer the ability to make the trade-offs between detail and performance at different times during the simulation, offers an opportunity for performing virtual prototyping.