Hierarchical Approach to Simulation in a Vertical System for the TriCore Microcontroller

Chris Salzmann, Eric Chesters, Paul Coelho, Yeh-Chen Fu, Jithendra Madala,
Mulka Reddy, Frank Wang
Siemens Microelectronics Inc.

Abstract

This paper describes a hierarchical functional verification system for a core-based system design which minimizes complexity in testbenches while maximizing flexibility in terms of number of clocks and system interfaces and reducing time-to-market. The verification system also provides a defined methodology for early operating system and application software debug.

I. The Problem

The problem faced by digital design groups is how to shorten the time it takes to design and debug new processor designs. The problem of verifying a new machine built upon a new architecture is in many ways significantly more complex than the verification of an architectural increment of an existing machine. Building a verification system for a new machine architecture while the architecture is being developed is a fine example of concurrent engineering.

In the case of building a verification system for a series of system designs based on a new architecture one needs to consider more than the potential for architectural and design change. The verification system must be configurable to permit the modeling of various types of systems. The configurability provided enables customers to use a core-based methodology in the design of systems. In any case, the verification problem is the same, find the bugs, find the bugs, find the bugs, and do so in as short a time as possible. The consensus today is that the most important feature of a machine design is time-to-market.

Design defects fall into several categories. The first category includes the defects due to an incomplete architectural specification. The specification may have been left incomplete for any number of reasons, among them being future expansion of a new architecture. This case is very dangerous since the implications of this for a new machine may not be fully understood for a particular implementation. The second category includes defects due to a specification error. It is possible for the specification of a new architecture to specify conflicting functions, which affect separate sections of the implementation. Hopefully, verification tests or formal techniques later detect these implementation defects. The specification could require a feature, which is either difficult or impossible to implement. In this case, the implementer may easily make costly mistakes.

In any large machine design with many engineers involved in the design process, communication is critical to success, only more so on a new architecture than on an old one. The classical problem here is properly connecting disparate function blocks in the processor. In a new machine, this is more complex since there is no well-trodden path to follow. Errors of omission are more likely than are errors of commission. In a processor, there is also the problem of verification of computational blocks. It is not possible, in a processor design to verify every possible computation. In control logic, there is the problem of dealing with asynchronous events. Every machine must have a memory. Some memories will be dynamic and have refresh, for example. Logic designers often say such things as, "That happens only infrequently, don't worry about that." It is in the corner cases of computations and in less often used features where design defects are most often found. On an asynchronous system chip, there is the problem of synchronizing data streams arriving or departing at various data rates.

In short, there are many potential sources of error in a machine design. These defect sources need to be found as quickly as possible.

In addition to these problems we cannot underestimate the problem caused by architectural changes. The classical approach to verification means that a cycle-accurate model of the architecture must be constructed. Imagine writing a program whose specification continues to change!

II. Recent Methods

Most functional verification systems depend heavily on modeling and simulation. Principles of software reliability are applied to the hardware design process in the sense of being able to compare two separate implementations of the same specification. Typically, a verification group will build two architectural models of the machine. One model suited for use by software developers and which is instruction-cycle accurate. The other model usually constructed is intended for use in hardware verification and for customer use as the definitive cycle-accurate model of the machine. The instruction-accurate model can be built because there is no need to model asynchronous system behavior accurately in a time sense from the perspective of the operating system. The model itself is simpler and a properly designed instruction-accurate model should run 5 or more times faster than a cycle-accurate model. The cycle-accurate model is required for hardware development since this model is the "golden" model of the machine.

There are variations in use of these models as applied to a verification system. The instruction-accurate model is typically a freestanding tool that is supplied to software developers. The variations in use regarding the role played by a cycle-accurate model in a verification system are much more interesting. Fundamentally, there are two main ways to integrate a cycle-accurate model into a verification system. The first way is to simply generate an output file from the instruction-accurate model that contains time, network inputs, network outputs, and relevant state information for use by a testbench written in the preferred HDL. This method has some definite advantages. First among these advantages is the ability to run the directed functional verification test programs against the architectural model only once and save the results on disk drives.

The file generated by the instruction-accurate model may also be used to supply test and compare vectors to a hardware accelerator. It is typically very inefficient to use a hardware accelerator of any kind in an interactive fashion. Doing this de-rates the hardware accelerator's performance to little better than the performance of the attached software-based simulator.

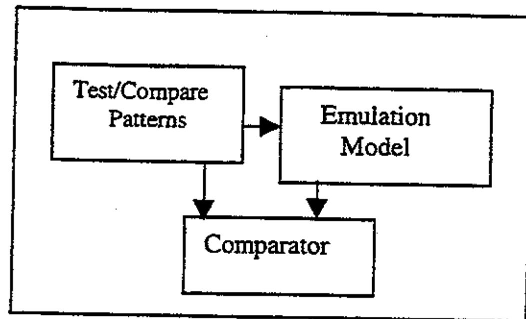

Fig. 1 Emulator testbench

The most efficient way to use the hardware accelerator is to create a file, as described above, with which to load the accelerator's memory model. The accelerator contains a testbench with the system chip on one side of a comparator and the test/compare memory on the other side of the comparator.

The second method used for instruction-accurate model integration in a functional verification system is to attach the instruction-accurate model to the HDL-based testbench through a language interface. For some, this is the preferred implementation method. Such verification systems could be called design-centered. This is due to the fact that all external interfaces to the system design being verified are made through the relevant HDL's language interface. This interface may be an extremely inefficient way to pipe data in and out of the HDL simulator, but in some shops the mantra is "this is what we've always done". A requirement of efficient verification system design is to minimize use of the HDL's language interface as much as possible.

Another problem that arises in the use of the HDL simulator's language interface is that the VHDL language, for example, does not provide a standard C/C++ language interface. Some VHDL simulators do not integrate well with C++ programs. Each VHDL simulator requires a separate interface to be developed. Thus the cycle-accurate model's HDL interfaces in a truly general-purpose, core-based verification system can be unnecessarily complex programs to write. On the Verilog side, there is only one C Language interface, on the VHDL side, 14 or more. Products exist which claim to obviate the need to create many different C interfaces.

One problem in both of these approaches is that most implementations of architectural model integration with a HDL simulator are not easily

portable across levels of design hierarchy. This means that a completely different testbench needs to be created for each block of the CPU, for each peripheral block, for the CPU core, and finally, for the system. The HDL testbenches must also use the C interface methodology provided to communicate with a myriad of different stimulus generators, a generator for each different external input interface of each block, core module, and system implementation. Thus we see the advent of testbench generators.

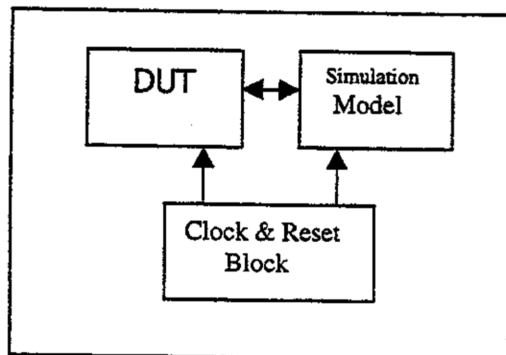

Fig. 1 Testbench Structure

While a particular simulation approach may be very appropriate for an existing architecture used as a core, it may make little or no sense as an efficient solution for a modern core-centric verification system. This is especially true if the simulation approach is intended for use by many different design groups within a large corporation and potentially by the company's customers. The key to success in a core-based methodology is for flexible re-configuration of the architectural model. Most, if not all, systems do provide this ability. The typical solution by vendors of existing architectures is to present the architectural model of their core to a customer and let the customer solve the system problems.

Formal verification methods are now being adopted by various organizations. They fall into two categories: intelligent netlist comparison programs, and model checkers. While they have a place in an overall verification system, it is not the purpose of this paper to discuss formal verification integration issues.

III. TIMING ACCURATE VERIFICATION SYSTEM FEATURES

The new approach to simulation use in a verification system for a core-based methodology proposed in this paper has been implemented on a core-based design project. It is intended to solve the problems of multiple, inefficient C language interfaces in complex testbenches, at the same it introduces a new method of operation in verification simulation methodology tailored to a hardware/software co-verification environment capable of shortening the overall design process, and thus reducing time-to-market.
The approach taken in the design and integration of the cycle-accurate architectural simulation model is the key to the success of the new method. The key concept for the functional verification system for this new machine architecture was to confront and to solve system problems from the beginning.

Solution of system problems began with the selection of the implementation language of the simulator. The language chosen was C++ for the cycle-accurate model while C was chosen for the instruction accurate model. C was chosen for the instruction-accurate model since it would enable a faster implementation to be built in less memory, a factor to consider for PC ports. This is especially true since extensive software simulation typically requires millions of cycles of simulation. C++ was chosen for the cycle-accurate "golden" model since a C++-based architectural model can provide a solution to the integration of arbitrary collections of models and to the implementation of various configurations. Thus, it is possible not only to describe the core with any collection of peripherals; it is also possible to describe the core with any collection of peripherals connected to any arbitrary system simulation. The verification group using this approach need not connect to an external system model initially, since it may be quite difficult using a "real" system model to model timing-based corner cases which may exist between the various peripheral ports of a given system chip design. The hardware verification group would design and connect Peripheral Transaction Generators (PTG) to the "golden" model to cover the corner cases or reuse those supplied by a vendor in the case of standard peripherals.

The software design group faces a different problem. They must implement the appropriate control program for a particular application. For example, the CPU core may be used to implement an automotive engine controller. For example, the system chip design to meet the requirements of an automotive engine controller may have a certain

complement of peripherals. The peripheral behavior, once defined is sufficient for the software group to proceed to selection of an RTOS, compiler, and software system design.

With the "golden" model they are able to verify that the program they are developing is, in fact, code that will properly execute on the core, but one link is still missing. If they had in their possession a computational model of a new engine that could be integrated with the "golden" model described here, it would be possible to create an overall system simulation of the engine. This simulation can enable code debug even before a new engine is ready.

The above proposal solves system-level hardware/software co-design problems; it is also a step in the direction of solving some lower level problems described above. The inefficient language interface problem may now be addressed. The design of this "golden" model also addresses the complexity issue in a HDL testbench. Both goals are simply accomplished by removing the complex RTL descriptions for test pattern generators for the peripheral ports from the HDL and replacing them with the PTGs easily configured and easily integrated C++ modules. This permits easy application of recorded real system data to the peripheral ports of the system chip being modeled. Additionally it reduces the testbench to essentially a large comparator. The application of the vectors to the testbench has been made more efficient by the elimination of different peripheral port calls across the HDL simulator's C interface to only one call to a system model.

We have been able to improve efficiency of the simulation system and to permit software development to proceed earlier in a design cycle. This also allows debug of new peripherals for a new system to proceed earlier since a defined system structure ready to be filled in already exists. A by-product of this is the speed up of the development of the core itself. This is accomplished by applying the high-level system concepts within the core itself. The core is itself, a system of blocks, which communicate via a defined interface to the rest of a system chip. The core has a bus interface, a memory interface, and an interface to an interrupt controller. Thus it is possible to build a bus controller, a bus transaction generator, an interrupt generator, and a memory in C++ for connection to the core model. The core is built of a collection of execution units which themselves have defined interfaces. Assuming the cycle-accurate model was a detailed model of the pipeline and how the various block interface signals changed during instruction execution, it could create test vectors and compare patterns for each core block independent of the others while it, itself was executing a program running at the system level. This has been accomplished.

The cycle-accurate model permits each block of the core to be fully regressed against a system-level test suite before the entire core even exists. Therefore, when each block is integrated to build the core, it has already been system tested.

IV. Functionally Accurate Verification System Features

A practical problem early in design implementation is that the RTL code developed may not exactly follow the machine's pipeline specification. This is a problem because the clock-relative timing of what appears on the machine's busses does not agree with what the pipeline specification or the architectural specification predicts. This means that early in the design cycle the cycle-accurate model will not match the hardware, not because of functional mismatches but because of mismatches caused by bubbles in the pipeline, for example. During this phase of design, the cycle-accurate model is useless.

The fastest way to full functionality and pipeline and timing accuracy is through stages. The first design stage being the creation of a fully functional RTL machine description. During this phase of design it is necessary to determine test pass/fail through comparison of states on selected busses regardless of timing. Thus the cycle-accurate model must have a second operational mode incorporated.

V. Summary and Results

Preliminary results regarding verification functionality are very encouraging. Using the cycle-accurate model and the instruction-accurate model operating systems have been ported. Industry standard benchmarks have been successfully run producing results, which created certain architecture changes. Verification of

TriCore has progressed starting at the base stage of functional comparison. Essentially the new verification system is applying the principles of software reliability. Software reliability states that defects in a software system will be best detected when two disparate implementations of a system specification are compared with each other. Using this methodology, we project a reduction in core design cycle of 3 months.