Software Development in a Hardware Simulation Environment

Benny Schnaider
MayaLee Consulting
Santa Clara, CA 95050
bennys@netcom.com

Einat Yogev
Cisco Systems
San Jose, CA 95134
eyogev@cisco.com

Abstract

Concurrent verification of hardware and software as part of the development process can shorten the time to market of complex systems. The objectives of the Virtual CPU approach is to provide a solution for code development in a simulation environment before the system prototype is ready. The VCPU is an ideal solution for processor-based systems that include multiple new ASICs and boards, in which the hardware, diagnostics and software drivers must be developed concurrently.

1 Introduction

Typical hardware development is based on a Hardware Description Language (HDL) which is used for modeling the hardware. The hardware model is simulated on a Hardware Simulation Engine (HSE), for example, Verilog XL, VCS or PureSpeed. A hardware modeling technique provides a very accurate representation of the hardware, as this is the hardware design entry point from which the ASICs are synthesized and the board layout is derived. The system model is usually available months before the system prototype exists.

The motivation is to provide for the software developers a fast and convenient interface to the Hardware Simulation Engine. The software code, written in ‘C’, can be compiled and run on any local or remote workstation which acts as a Virtual processor. During execution, when a reference is made to a memory location modeled by the HSE, the environment invokes the HSE to execute the reference. In addition, the code accepts responses for read transactions as well as interrupts from the HSE.

The interface between the software program and the HSE is abstracted by a transaction model so it is not limited to the processor bus. This abstraction enables software access to any bus within the system. In hardware development, where some ASICs or subsystems may be ready sooner than others, this becomes important, since simulations can be run even before the processor subsystem is ready for simulation. Another advantage of such an abstraction is that it makes it possible to co-simulate software with smaller configurations of the hardware, thus increasing the simulation speed of the HSE. The software program should be allowed to run only as long as the code accesses the address space within the hardware configuration.

The fact that the software is running on a workstation allows the programmer to use familiar software development and debugging tools, and to debug the hardware model at the same time. In some cases companies start system development even before the processor hardware is available, which enables the concurrent development of the system and the processor itself.

This paper describes the interface between the verification environment and “real code” which is designed to run on the “real machine”. This interface allows for hardware and software co-verification.

2 The System Verification Environment

Logically, the verification environment is broken into two portions as described in figure 1:

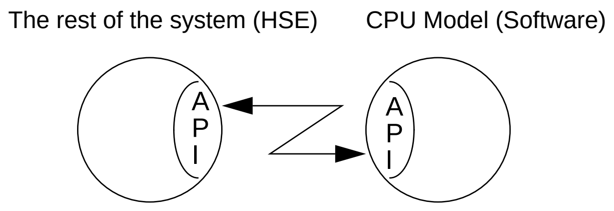


Figure 1: the programmer interface model

The link between the two portions is implemented by a special layer called the API. As shown, the API is distributed between both portions of the verification environment.

The challenge is to be able to co-simulate diagnostics and application programs in conjunction with the system model and verification environment. Although, the verification environment supports several configurations of the system model, all configurations share the same API. The purpose of the API is to provide a transparent interface to the underlying environment. In other words, the user's program requires almost no modifications when migrating between environments (in most cases just re-linking is required, in others re-compilation might be needed.)

Several simulation configurations are supported and each environment requires a different implementation of the API layer. The decision of which specific environment to use varies according to trade-offs between simulation throughput, simulation accuracy and convenience. A system verification environment supports two classes of configurations: a Cycle-accurate CPU model and a *Virtual CPU* model.

2.1 Cycle-accurate CPU model

In this class of configurations, the user programs are cross-compiled into target machine objects. The objects are loaded directly into the CPU memory image, as part of the initialization process of the simulation. Thereafter, the programs are executed by the CPU just like they would in the "real machine". Hence, the simulation is cycle-accurate and represents an accurate relation between the progress of the user program and the rest of the system.

There are several approaches for "Cycle-accurate" CPU models for example: Instruction Set Simulator and Hardware Modeler. In our environment, we are using an hardware modeler as our "Cycle-accurate" environment. In this solution (see figure 2), a real CPU is plugged into a Hardware Modeler while the rest of the CPU board is simulated in a verilog HDL environment. This is an accurate simulation model since it provides cycle-accurate simulation and precise synchronization between the processor and the rest of the model. It requires cross-compilation of source code for the target CPU. The biggest disadvantage of this approach is that it is very slow and complicated, from a software development perspective.

2.2 Virtual CPU

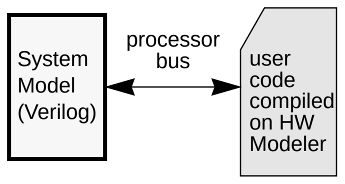
The approach taken in this environment is that the software runs on a Virtual CPU, for example a workstation, which provides an execution environment for the programs. User programs are written in a high level language (C) and are loaded on the target Virtual CPU. The mechanism built to interface between the Virtual CPU and the design model helps verify the algorithmic part of the user programs and the software-hardware interaction.

One of the challenges of this approach is to be able to synchronize between the CPU and the rest of the design model. Note that speed (measured in Cycles Per Second) of the Virtual CPU is much faster than the rest of the simulation model. Several solutions for the synchronization issues are discussed below.

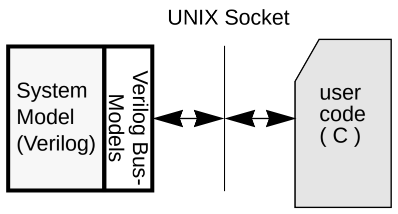
As seen in figure 2, the Virtual CPU implementation is based on a message protocol. The underlying transport layer is UNIX socket Inter-Process Communication (IPC). In this model, the CPU simulation model is separate from the rest of the simulation environment. The simulation environment is comprised of two distinct processes. The link between the two is accomplished by means of socket communication. The peers of the socket connection are special PLIs on the Verilog simulation side and the C-API layer on the other side.

On the HSE side, the interface to the design model is done through a bus model, and for this purpose any bus model written with a read/write/interrupt task interface can be used. This provides a way to apply the "real code" to various configurations of the design model, sometimes before the entire system design is complete, and not necessarily on the processor bus itself.

On the programmer's side of the socket interface, the code interfaces with an API layer which is used in the simulation environment. This layer may be replaced by different sets of service routines when running in the real system.



Using the HW modeler, the real processor co-exists with the HDL simulation model



Virtual CPU runs as a separate process, and interfaces to the same bus model used by the Verilog tests

Figure 2:

System model interface to tests and programs

3 API

The purpose of the API is to provide a transparent interface between user programs and the underlying simulation environment. The API provides the following benefits:

- Smaller development effort
- Use of a single source code for both production and verification
- Hiding of the "low-level" interface details from the programmer

A special API library is implemented for software running on the design model.

The API is comprised of the following interfaces. Some examples are given in table 1.

3.1 Memory Access

All CPU accesses to any registers or memory locations within the design model are based on the same interface. Each access consists of address, data, size and special options.

3.2 High Level APIs

These APIs provide high level operations on the model. Although all the high level operations can be implemented as a combination of register and memory accesses, it can be useful to have higher level APIs. The main reason is the fact that these APIs can be implemented more efficiently, depending on the environment. For example, during initialization, it might be useful to copy the memory image of the internal tables (e.g. buffer memory, routing tables, etc.) instead of using a sequence of individual accesses.

3.3 Interrupt Interface

The interrupt interface allows the HSE to trigger interrupts to the Virtual CPU. The API detects interrupts either synchronously by polling, or asynchronously by using OOB (Out Of Band) socket signaling.

The VCPUinterrupt call instructs the API to call a callback function (callback is a pointer to a function). It is recommended that the callback function point (potentially indirectly) to the "real" ISR (Interrupt Service Routine). Note that the interface for the callback function might be different for the "real-hardware", as some of the ISR is written in assembly language and directly interfaces to the "real CPU" registers.

3.4 Link to Verilog Code

In some cases, the high level program needs to call Verilog tasks/functions. Typically, these cases involve triggering and monitoring internal events in the simulated model. The VCPUcallVerilog API, improves the controllability and observability of the verified design, and saves significant amounts of simulation time. This back-door interface is used primarily for debugging and should consequently be removed from the code running on the real machine.

In order to facilitate this requirement, the API provides a mechanism for the program to call verilog tasks/functions directly. For simplicity, the linkage to the Verilog task is done by mapping numbers into tasks/function calls.

TaskNumber is an integer which maps into a Verilog task/function list. The rest of the parameters are passed directly to the corresponding task/function. Note that parameters are passed by reference, which enables parameters to be used for both input and output. The semantics of input/output depends on the TaskNumber. The list of task numbers and their semantics is expected to grow based on ad-hoc requirements.

3.5 Synchronization

The user program running in the Virtual CPU environment runs much faster than the Verilog process. Synchronization is required to keep the VCPU and the HSE in sync and always occurs on machine cycle boundaries. The API supports several synchronization modes:

Free running: In this mode, the Virtual CPU and the HSE are free running processes and interact in their interface points (e.g., read, write and interrupt transactions) only. The C-API blocks on every access until the operation is completed on the HSE side. For example, in case of write access, the Virtual CPU clocks at the beginning of the write operation and resumes only after the write cycle is completed. This is the default synchronization mode.

Lock Step: The Virtual CPU specifies the number of cycles that need to be run until the next synchronization point. The amount of synchronization time required is passed implicitly with every read/write operation or explicitly when the program asks for the delay.

None: No synchronization. The Virtual CPU gets control back as soon as the call is sent to the "other" simulation environment.

3.6 Processor specific interfaces

This set of interfaces is used for referencing CPU resources such as registers and caches. Typically, these commands are emulation of target processor assembly instructions and require change in the user code.

Table 1: API examples

API Type	Examples
Memory Access commands	VCPUread(addr, data, size, option)
	VCPUwrite(addr, data, size, option)
Interrupt interface	VCPUinterrupt (callback, intrNum, param)
Link to verilog code	VCPUcallVerilog(TaskNumber, param1,...)
Synchroni-zation	VCPUsetSimMode (simMode)
Processor commands	MIPScp0Write(adrs, data)
	MIPScp0Read(adrs, data)

4 HSE Interface Implementation

The implementation of the Virtual CPU interface in the HSE is divided into three layers: the PLI (Programmer Language Interface) layer, the bus-independent layer and the bus-dependent layer as illustrated in figure 3.

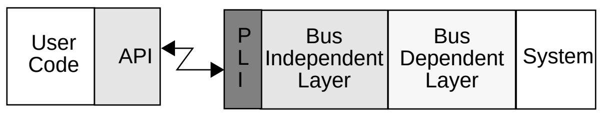


Figure3: HSE Interface

PLI layer:

The PLI layer provides the socket interface to the Virtual CPU on one side and the interface to the HSE on the other side. It is written in "C" and implemented as a PLI code which is linked into the verilog image.

Bus-independent layer:

This layer is activated whenever the VCPU needs to interact with the HSE. It is implemented as a Verilog stub which is instantiated as part of the top level of the design. This is a generic layer which relies on a bus model (the bus-dependent layer) to execute the VCPU requests. The separation of bus-dependent and bus-independent layers adds the flexibility of connecting the VCPU interface to several places in the design as illustrated in figures 4 and 5. The decision of where to connect the interface depends on the availability of the hardware models and the specific requirements from the environment.

Bus-dependent layer:

This layer is basically a bus model that implements the read/write accesses to the hardware. In most cases, this bus model is required for other purposes (e.g., Hardware design verification). The interface between the bus-dependent and bus-independent layer is based on a predefined task interface. The bus dependent layer enables easy migration between different buses without modifying the bus-dependent layer

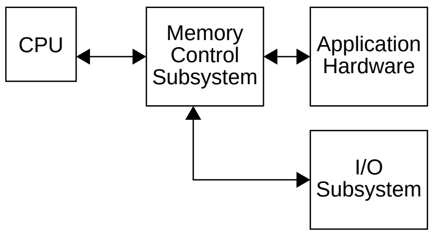


Figure 4: Block diagram of a typical system

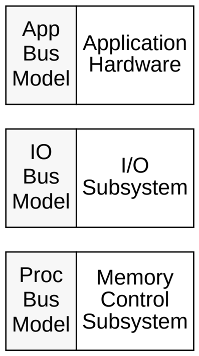


Figure 5: Subsystems with related bus models

5 Performance

For the most part, the simulated program is executed on the Virtual Processor. However, some instructions are emulated and are executed remotely on the HSE. The Virtual Processor speed is around 100 million instructions per second where the HSE is running in the range of 1-100 instructions per second. Hence, the more instructions that are executed on the Virtual processor, the faster the overall performance will be.

This implies that the actual performance gain depends on the ratio of the total number of instructions versus the number of instructions that need to be emulated on the HSE. Moreover, when data cache is modeled on the Virtual Processor, the ratio of local versus emulated instructions increases even more due to cache hits.

To evaluate this, we ran some performance analysis on some of our test cases. Table 2 summarizes the results as measured by the profiling tool.

Notes:

- The acceleration ratio varies substantially and greatly depends on the underlying test.
- I/O is comprised of operations like printf and file read/ write.
- In the case of T3 we discounted the time used for User Interface interactions.

Table 2: Instruction ratio

Name	Type	Total Instructions	IO-Non Cache	Cache hit	Ratio
L1	A loop involving 1 pointer and two counters.	20	1	0	20
T1	Short diag. w/o I/O	4112	77	73	53
T2	Short diag. with I/O	2693326	77	73	34978
T3	Medium diag. w/o I/O	1159890	815	0	1423

6 Pros and Cons

In this section we compare the Virtual CPU approach to the cycle-accurate approaches (e.g., HW modeler).

Speed - As discussed above, the overall speed of the VCPU solution is 20 to 35,000 faster then HW modeler depending on the test case. This is definitely an advantage for the Virtual CPU since the relatively high speed enables some applications that would otherwise be impossible implement (e.g., User Interface driven diagnostics).

Programming environment - The Virtual CPU development environment is very similar to the traditional software development environment. With the hardware modeler, the programmer can not use tools like debuggers and requires substantial knowledge of the HSE environment.

Code modifications - Currently, the Virtual CPU requires modification of the code for each reference to the HSE. This requirement creates a problem for existing code and is not convenient for programmers. In our future work we intend to eliminate this limitation.

Accuracy - There are two elements associated with accuracy: inter- and intra-cycle timing. The intra-cycle timing accuracy is dictated by the quality of bus model and is typically very accurate. On the other hand, the inter-cycle timing accuracy is limited since the program is running on a Virtual processor which is far faster (100 Million to 1) than the HSE. Our experience has been that this is not an issue as we are using other design verification methodologies for testing intra-cycle timing. The most noticeable methodology is transaction interaction tests.

Assembly code - The Virtual CPU does not support simulation of the code written in machine language of the target processor. Although the VCPU API provides interface for specific assembly instructions, users still need to rewrite their code containing assembly instructions.

Processor resources - Modern processors are comprised of several resources (e.g., caches, TLBS). The current implementation of Virtual CPU support for these resources is limited, but it is our intention to add more support for these (see future work).

Virtual machine differences - Some of the processor operations may differ between the Virtual and the target processor. For examples: arithmetic operations, byte order and pointer sizes. In our environment, these differences have not been an issue.

Because of some of the cons described above, we are still using the Hardware modeler for “final” testing and for the tasks that are hard to model on the Virtual CPU.

7 Further Work

In a co-simulation environment of hardware and software, as described above, the overall performance is always limited by the HSE speed. The code running on a workstation slows down to the HSE speed any time there is an interaction with the hardware simulator. Simulation performance that may be acceptable for diagnostics development may not be sufficient for developing the entire application code. It is likely that in the near future a VCPU interface to hardware accelerator or emulator will be implemented, increasing the HSE speed by several orders of magnitude.

For systems that consist of multiple processors, the dual process environment for HSE and VCPU can be scaled further to allow multiple VCPU processes running concurrently with one hardware simulation model.

We are currently working on interfacing the API to the target processor instruction set simulator (ISS). With such a configuration, the code will be cross compiled to run on the ISS, and the ISS will directly run the machine code with a similar API interfacing the HSE. This will provide a more accurate representation of the system and will also enable us to run assembly code and provide a solution for microcode development.

In addition, we are improving our modeling of internal CPU resources such as TLBS.

8 Conclusions

The Virtual CPU approach provides a framework for concurrent engineering. Software can be developed and tested early on in the design process. Typically, this environment is used for the platform-dependent, low-level software development. The software developed in this environment is also used to validate the hardware design. The simulation performance (measured by simulation cycles per second) can be increased by a factor of 4-10, since the simulated design on the HSE is smaller. For software development, this environment is convenient, since traditional tools (for example, compilers and debuggers) can be used for development. The Virtual CPU environment is also faster by a factor of 20 to 50,000 relative to the current co-simulation methodologies (e.g. HW modeler).

References

[1] Stevens UNIX Network Programming *PTR Prentice HAll Englewood Cliffs, New Jersey 07632, 1990*

[2] Berkeley 1986b, UNIX Programmer’s Reference Manual (PRM) 4.3 Berkeley Software Distribution, Computer System Research Group, Computer Science Division, Univ. of California, Berkeley, Calif., Apr. 1986.

[3] Verilog Hardware Description Language Reference Manual (LRM). Draft, IEEE 1364 November, 1994.