Hardware/Software Co-Verification Using the SystemVerilog DPI

Arthur Freitas
Hyperstone GmbH
Konstanz, Germany
afreitas@hyperstone.com

Abstract
During the design and verification of the Hyperstone S5 flash memory controller, we developed a highly effective way to use the SystemVerilog direct programming interface (DPI) to integrate an instruction set simulator (ISS) and a software debugger in logic simulation. The processor simulation was performed by the ISS, while all other hardware components were simulated in the logic simulator. The ISS integration allowed us to filter many of the bus accesses out of the logic simulation, accelerating runtime drastically. The software debugger integration freed both hardware and software engineers to work in their chosen development environments. Other benefits of this approach include testing and integrating code earlier in the design cycle and more easily reproducing, in simulation, problems found in FPGA prototypes.

1. Introduction

The Hyperstone S5 is a powerful single-chip controller for SD/MMC flash memory cards. It includes the Hyperstone E1-32X microprocessor core and SD/MMC interface logic. A substantial part of the system is implemented in firmware, which makes hardware/software co-design and co-verification very advantageous.

Hardware/software co-verification provides our software engineers early access to the hardware design and, through simulation models, to not yet existent flash memories. It also supplies additional stimulus for hardware verification, complementing tests developed by verification engineers with true stimuli that will occur in the final product. Examples include a boot simulation and pre-formatting of the flash controller.

Due to highly competitive design requirements, a hard macro version of the Hyperstone E1-32X processor's 32-bit RISC architecture was used in the system, necessitating a gate-level simulation. Not surprisingly, this resulted in simulation performance well below that required for hardware/software co-verification. In order to increase the performance of the simulation environment, we needed to find a way to accelerate the microprocessor.

The most cost effective solution was to integrate an ISS model of the existing microprocessor. Due to its high abstraction level, the ISS led to the boost in simulation performance we needed. Because the E1-32X microprocessor is well established through usage in previous successful designs, and, therefore, assumed to be bug-free, the verification still addressed the entire DUT.

We also had to find a way to integrate our high-level software debugger with the HDL simulator so our designers could debug their C/assembly source code, rather than viewing events only in a waveform display.

Our software debugging environment communicates with the development board through a UART interface. In order to interface it to the HDL simulator, we wrote a behavioral model of the UART interface in C. In this way, the same development environment already used by the software team could be used to control and examine the hardware simulation.

The next chapters will describe the steps taken to integrate the ISS and the software debugger interface into the HDL-simulation using a SystemVerilog DPI. The productivity improvements we achieved using this method will also be presented.

2. Testbench evolution

2.1 The Original Testbench

The original verification environment for our SD/MMC family of flash controllers was based on a processor-driven testbench (PDT). We wrote verification programs in C and assembly language and cross-compiled them to the Hyperstone microprocessor. We employed the same tools (i.e., compiler, assembler, and linker) used by our software developers to generate an object file of the desired test. Then with the help of additional scripts, we converted the object file into a loadable memory image for the logic simulation. At the start of the simulation, the memory image was loaded by a Verilog system task (i.e., $readmemh) into the memory model array. We had a simple verilog testbench to replicate a card interface driver. This consisted of a register interface, which was memory mapped to the microprocessor, and a finite state machine, which converted I/O accesses from the microprocessor into actual SD/MMC bus protocols. Figure 1 depicts the original verification environment.

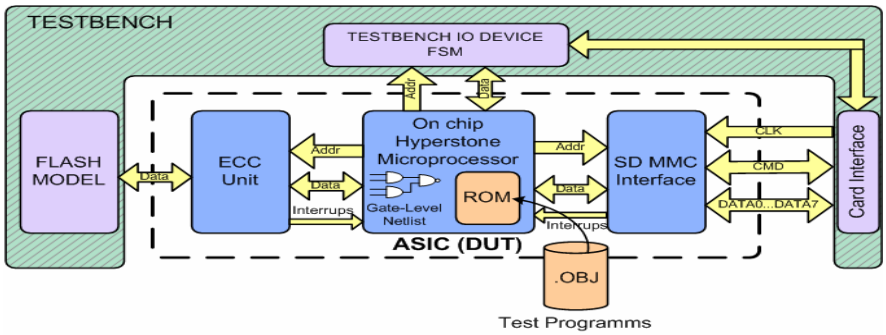

Figure 1 – Original processor driven testbench

Our verification programs performed the following main tasks:

- Handle the interrupts and configure the system.
- Replicate a SD/MMC host, generating proper stimuli in the card bus (i.e., send commands and data).
- Analyze the output, verify the responses, and immediately stop the simulation upon errors.

An entire library of functions to test the system was already available (e.g., functions to send commands, to initialize the card, perform data communication, etc.).
By using these libraries we could program tests at a very high level of abstraction, so the task of writing the regression test suite was quite convenient. Although this approach had the potential to be very productive, it was rather slow because of the gate-level abstraction of the microprocessor. Tests which really stressed the microprocessor's surrounding hardware would require extremely long run times. To achieve a high level of confidence in code and functional coverage, we would have to spend several weeks identifying the coverage holes and writing the appropriate tests. Single simulations could take hours, and a whole regression would take days to run. To write and debug tests we had to wait a long time to get to the point where we wanted to test, and typically we had to do this repeatedly.
In a typical test, the processor comes out of reset, retrieves the reset vector, executes an initialization routine, and then branches to the memory location where the test code was linked to begin its execution. All the bus cycles required to fetch code from memory are of minimal value for functional verification but consume a significant part of runtime.

2.2 The Virtual Prototype Testbench

In order to speed up simulation and build a regression test suite that gave us the desired level of functional coverage, we decided to replace the gate-level representation of the microprocessor with its ISS written in C language (which was already available in-house).

Because the ISS comprises all the memories of the system, the processor cycles used to fetch the code and, read and write static variables were no longer simulated by the logic simulator. Code and data cycles were simply filtered out from the logic simulator, and since they were processed in zero simulation time, the simulation advanced through the test code at a much faster speed. Only accesses to the memory mapped registers of the hardware or to the flash memory had to be simulated by the logic simulator. This drastically reduced the turn-around on regression test suites, allowing many more tests to be run in a given time frame.
To accomplish this, we needed to find a mechanism for Verilog code to call functions written in C. The first idea was to make use of a Verilog PLI, but the Verilog PLI is a complex interface and does not usually support high simulation performance. Conversely, the SystemVerilog DPI allows SystemVerilog code to call C functions as if they were native Verilog functions without the complexity of defining a system task and the associated callft routine, both of which are required by the Verilog PLI.
By using the ISS in simulation, we achieved a boost in simulation performance that provided us with opportunities to further enhance the testbench environment. In the original PDT, the embedded processor had to perform both host and card functionality and, therefore was not free to run the firmware exclusively. However, the speed improvements obtained permitted us to add an additional processor to the testbench. This additional processor could be used as an SD/MMC host to run our verification library, whilst the on-chip microprocessor was dedicated solely to running the firmware. Figure 2 illustrates the new testbench which matched the real application and enabled hardware/firmware co-verification.

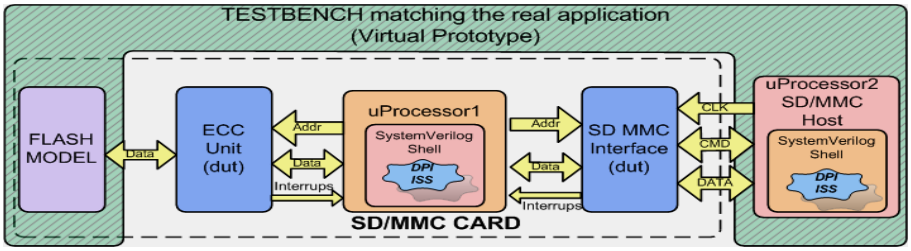

Figure 2 – Testbench matching the real application

3. Using the Systemverilog DPI

The SystemVerilog DPI allows SystemVerilog code to directly call functions written in C++, or standard C and vice versa. Data can be passed between the two domains through function arguments and results. This is accomplished with virtually no overhead.
On both sides the calls look and behave the same as native functions. SystemVerilog can therefore call a C imported function as it were a SystemVerilog task or SystemVerilog function. In the same way, C can call an exported SystemVerilog task or SystemVerilog function as if it were a native C function.
By calling SystemVerilog functions or tasks, C code can access simulation events and consume simulation time. This transfer of control between C and SystemVerilog is actually the only means of synchronization, since all functions are assumed to complete their execution in zero simulation time.
SystemVerilog tasks may consume simulation time (by using #, <=, @, and, wait constructs) and may call child tasks or functions. SystemVerilog functions may call only child functions and are not allowed to consume simulation time.
C functions that are intended to be called by SystemVerilog are referred to as imported functions or tasks and must be declared as follows:

```
import “DPI-C” [pure|context] function [return-type]
function_identifier (port_list);
import “DPI-C” [context] task task_identifier (port_list);
```

The import statement defines that the function uses the DPI interface and will be executed in C, it contains a prototype of the function name and its arguments. For example:

```
import “DPI-C” pure function int mulacc (input int op1, input int op2, output longint res);
```

An import declaration can occur where ever a Verilog task or function definition is allowed. An imported function can have input, output, and inout arguments, and it returns a value. An imported task does not return a value. C functions can be imported as pure or context. The results of a pure function must depend only on its arguments. A context function can use global and static variables and can call other C functions.
It is not legal to call an exported task from within an imported function, but it is legal to call an exported task from within an imported task, if this task is declared with the context property. Due to this rule, we used only tasks in the ISS integration.
A Verilog task or function can be exported to C code using a simple export statement:

```
export “DPI-C” task task_identifier;
```

The arguments of the Verilog task or function are not listed as part of the DPI export declaration, only the name of the task or function must be specified. A task or function can be exported only from the same scope where it is defined.
The following SystemVerilog types are allowed for formal arguments of imported and exported tasks or functions: void, byte, shortint, int, longint, real, shortreal, chandle, and string. The user is responsible for specifying the correct match between parameters in C and SystemVerilog in the DPI declarations. The user is also responsible for the compilation and linking of the C source code into a shared library (i.e., windows .DLL, solaris .so, or hp-ux .sl), which is then loaded by the simulator.

4. Integrating the Instruction Set Simulator

The instruction set simulator is a non-cycle accurate simulation model of the Hyperstone E1-32X microprocessor written in C language. It simulates not only the full instruction set architecture (ISA) but also memories and peripheral circuits, such as timers and interrupt controllers. The simulator executes programs written and compiled for the Hyperstone E1-32X. After every instruction is executed, the entire register stack of the microprocessor is saved in a set of variables. Programs are run in a sequential manner, neither instruction pipelining nor any timing of the microprocessor at the hardware level is modeled.
By using the ISS in HDL simulation, the logic simulator only has to simulate cycles originated by the SD/MMC interface and the ECC unit. Cycles that originally were being simulated to perform the pipeline execution of the microprocessor are now simulated in zero simulation time in C. The bus traffic is also substantially reduced because the ISS presents only the bus cycles addressed to the external hardware and the testbench to the logic simulator. The overhead traffic originally caused by instruction and data fetch is filtered out of the bus.
To integrate the ISS, its memory interface had to be adapted to call the logic simulator to satisfy accesses to the surrounding hardware. Fortunately the ISS memory model was not completely flat: accesses to some specific memory ranges were already calling different C functions (hardware stubs). So in most cases we simply extended these C functions to turn memory transactions into calls to the logic simulator. Also reset, interrupts, and some pins were reported to the ISS.
Another aspect that had to be taken into consideration was the synchronization between the ISS execution and the logic simulator, as there is a trade-off between performance and accuracy. For the hardware verification, the software execution can be thought of as overhead because the processor is there only to generate bus transactions that will stimulate the hardware under test. If we were solely interested in verifying the hardware, we could actually replace the processor and the software with a bus functional model written in C or SystemVerilog. Yet, we wanted a platform where we could verify the firmware and reuse the verification library already available. Therefore, we needed a fully functional model of the microprocessor (i.e., the ISS).

The method we chose for the ISS integration consists of a near cycle-accurate system where the ISS is a slave of the logic simulator. It gets called every clock cycle to take over the control of the simulation and execute one single instruction. After finishing the execution, it gives the control back to the logic simulator. Interrupts are reported on every call. When the ISS has to simulate the execution of `LOAD/STORE` instructions addressed to the external hardware, it passes the control back to the logic simulator. Since instructions executed by the ISS do not consume simulation time, our system is not cycle equivalent to the real system: multi-cycle instructions (e.g., `DIV`, `MUL`, etc) are executed in a single call.
To interface the ISS with the logic simulation we created three new components:

- A SystemVerilog shell
- A C function to interface the actual ISS
- Verilog tasks to perform the bus transactions for the ISS.

As a wrapper for our ISS, the SystemVerilog shell replaced the microprocessor netlist in simulation. The shell has exactly the same I/Os as the actual Verilog. Internally, instead of the gate-level description of the logic, it contains the tasks needed to interface the C code.
We wrote the interfacing function in C to hold and transmit the required parameters to the actual ISS. We imported this C function to the SystemVerilog shell with the following statement:

```
import "DPI-C" context task ProcessorCall (input int reset, input int intrpt1, … , input int pin1,…);
```

This was the only imported task in the SystemVerilog shell. We imported this C function as a context task because it calls our ISS, which in turn calls the SystemVerilog exported tasks. The function has no output parameters and its return value is not used. The input parameters are reset, the interrupts, and some of input pins of the chip. This function is called from SystemVerilog every clock cycle so the ISS can execute instructions of the test program. One can actually configure the number of instructions that the ISS is allowed to execute after each call. The parameters passed are assigned to global variables inside the ISS before its main function is called to resume the program execution. Before the function returns, the ISS saves its state in a set of global variables. Figure 3 depicts this system.

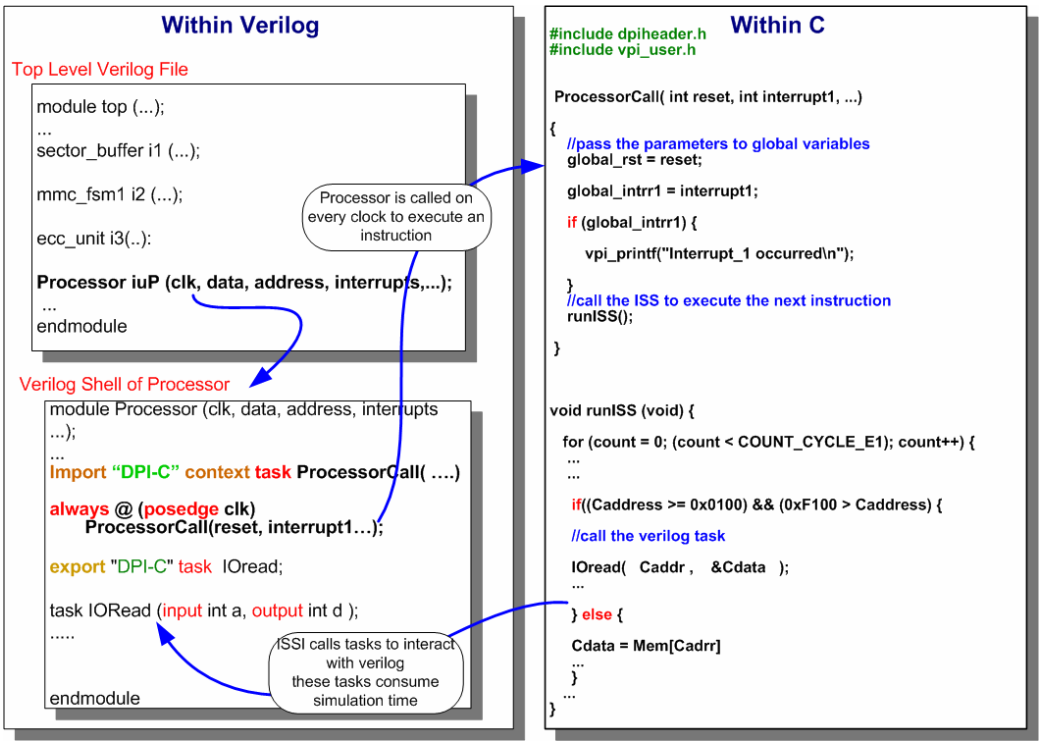

Figure 3 – Instruction Set Simulator (ISS) integration

In simulation, when the reset signal is released, the imported task is called on the next rising edge of the clock. Then the ISS executes the reset routine, initializing some of the control

registers and branching to the address where the test program resides. On every new clock this function is called again, and one more instruction of the test program is executed. Yet, as previously mentioned, the simulation is not cycle accurate with the original microprocessor. Multi-cycle instructions present results immediately since the ISS does not model the exact timing of the microprocessor.
When the ISS needs to access the external hardware it makes use of SystemVerilog tasks. As stated above, the ISS had to be slightly adjusted to redirect memory mapped accesses to the external hardware to the SystemVerilog tasks.
We described these tasks in the microprocessor SystemVerilog shell and exported them so they could be called from the ISS. Whereas the instructions executed by the ISS do not consume simulation time, a SystemVerilog task can consume simulation time, providing a way for the ISS to synchronize with the simulator.
Figure 4 exemplifies a Verilog task that generates an I/O read bus protocol for the ISS. When a `LOAD` instruction addressed to the external hardware must be executed, the ISS calls the exported task to generate the waveform depicted in figure 5. The required address must be gated to the output ports of the Verilog shell that represent the address bus.
After the address set up time (i.e., one clock cycle), the `IORD` signal is gated to its respective output port, and, finally, after the access time(i.e., 2 clock cycles) the data can be latched from the I/O ports of the Verilog data bus. The execution of the ISS is suspended until the Verilog task is completed, and the simulation advances in time, based on the task's time controls.
Note that the task waits for the positive edge of the system clock before it starts executing the procedural assignments also the wait statements (e.g., `# period * `bus_hold`) are waiting for an integer number of clocks. In this way, we synchronize all assignments with the positive edge of the system clock.

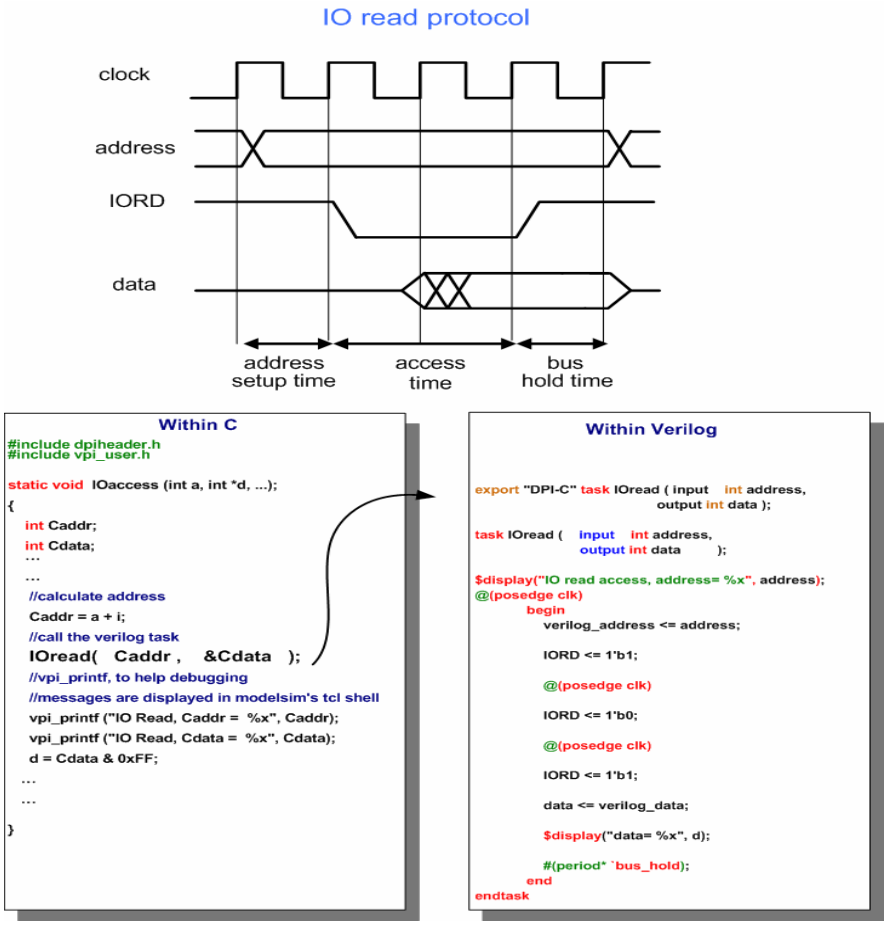

Figure 4 – ISS access to the external hardware

5. Integrating the Software Debugger

Now that we had a virtual prototype, the software developers became interested in simulating pieces of their code in the logic simulator.

Logic simulators are very good for source level debugging of Verilog or VHDL, as well as tracing signals, but you have virtually no visibility into the assembly or C firmware driving the simulation. You can use the compiler and linker to generate map and listing files, which help to find the line of firmware that was being executed for a particular program counter (PC) value. You can also set conditional breakpoints on the PC to stop the simulation before a specific instruction is executed. Yet these methods are not the most productive ways to control the firmware simulation. Also, most firmware developers are not used to working with a logic simulator.

We realized that we could leverage the SystemVerilog DPI to integrate the software debugger into the verification environment, allowing hardware and software developers to use the same tools that they are currently using in the design process.

In order to integrate the software debugger in simulation we wrote a behavioral model of a pseudo-UART in C and SystemVerilog. We had to model only the data communication protocol between the microprocessor and the software debugger.

The SystemVerilog part of the UART has the same interface as the original RTL. Internally it has an address decoder and DPI imported functions to perform the actual byte communication. Processor accesses addressed to the configuration registers of the UART have been simply ignored. Only the accesses meant to transmit and receive data have been processed. The C part of the UART performed the task of interfacing the simulator with the serial port of the PC.

In simulation, after the reset the simulator calls a DPI imported function, which uses an API (i.e., `CreateFile`) to open the serial port connected to the software debugger. When the microprocessor needs to send a byte to the software debugger, it issues a write access to the UART's TX register. This access is decoded in the UART SystemVerilog shell, which calls a DPI imported function that in turn uses an API (i.e., `WriteFile`) to send the byte to the software debugger through the PC's serial interface. After sending a byte the microprocessor polls the UART status register to see if a byte has been received. This will again trigger a DPI call, which in turn uses an API (i.e., `ReadFile`) to check if there's a valid byte waiting. If the status register signals that a byte has been received, the microprocessor reads it from the RX register in the following cycle.

Figure 5 illustrates our final hardware/software co-verification environment.

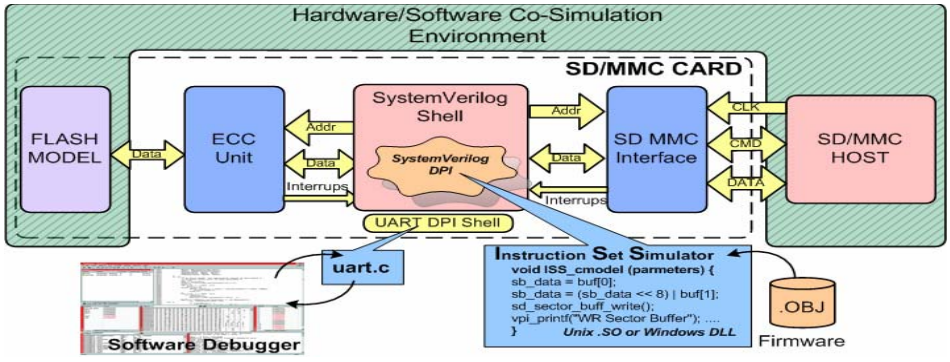

Figure 5 - Hardware/software co-verification environment

6. Benchmarking

For the functional verification regression suite, our benchmark results revealed that by simulating the ISS, we gained runtime improvements of more than 200 times over the actual netlist simulation. Designs compiled with the optimization options of the simulator also showed interesting speed improvements.

We also compared the speed of communication between the software debugger and the flash controller when using our development board (the actual hardware) versus two simulations. In hardware, the bottleneck is clearly our debugging protocol, which can only transmit 4K bytes/second. In the accelerated simulation, using the ISS, the communication between the debugger and the controller was only nine times slower than the development board. This is more than fast enough for our software developers to write firmware in this simulation environment. The normal netlist simulation was 53 times slower than the accelerated simulation. This is still adequate when we need to debug a timing dependent problem that requires a cycle accurate simulation.

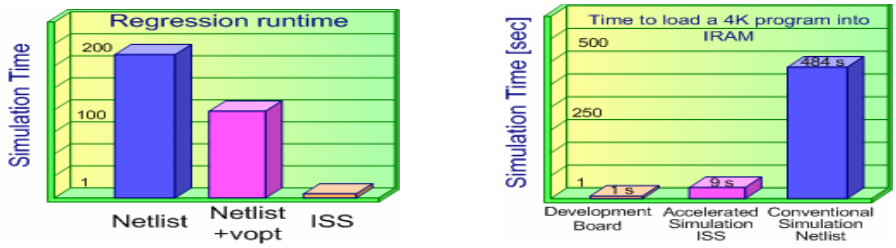

Figure 7 – Benchmark: Regression runtime & debugger/target data communication speed

7. Conclusions

The speed improvements gained by using the SystemVerilog DPI, allowed our processor-driven testbench to evolve to a hardware/firmware co-verification environment. We now have fast functional virtual prototype, which enables us to perform functional verification and rapidly debug problems found in the FPGA prototype, as long as these problems are not dependent on timing relations. For timing dependent problems we can still perform a slower, cycle accurate simulation.

We no longer have to spend time reproducing test cases in the simulation testbench. We can now simply hook up the software debugger to the logic simulator to simulate the same piece of software that is causing the problem in the FPGA or ASIC sample. The firmware is also true stimulus that complements the hardware regression test suite, without additional effort for the verification engineer.

Logic designers are also profiting from the speed improvements, as they work in the same environment and can now more rapidly perform the logic changes and updates.

References

[1] “SystemVerilog 3.1a language reference manual”, Accellera Organization, Inc. Napa, CA

[2] Arthur Freitas “accelerating system verification using c model integration and systemverilog dpi,” Mentor U2U 2006

[3] Stuart Sutherland “Integrating systemc models into verilog using the systemverilog dpi,” SNUG Europe 2004.

[4] Stuart Sutherland “The verilog pli is dead (maybe) -- long live the the systemverilog dpi!” SNUG 2004.

[5] Jim Kenney, "Using a processor-driven testbench for functional verification of embedded SoCs," Embedded.com 2006.

[6] Russell Klein, “Hardware/software co-verification” Embedded Systems Conference 2003.

[7] Jason R. Andrews, “Co-Verification of Hardware and Software for ARM SoC Design” 2005, Elsevier Inc, ISBN 0-7506-7730-9.

[8] Arthur Freitas, *"Hardware/Firmware Co-Verification Using ISS Integration and a SystemVerilog DPI"*, DVCon, 2007

[9] Arthur Freitas, “Abgedeckt und Integriert”, Design&Elektronik Magazine, Heft 11 - November 2006.