Efficient Testbench Code Synthesis for a Hardware Emulator System

I. Mavroidis, I. Papaefstathiou
Microprocessor and Hardware Lab (MHL)
Technical University of Crete (TUC)
Kounoupidiana, Crete, GG73100, Greece
{jacob, ygp}@mhl.tuc.gr

Abstract: **-** *The rising complexity of modern embedded systems is causing a significant increase in the verification effort required by hardware designers and software developers, leading to the "design verification crisis", as it is known among engineers. Today's verification challenges require powerful testbenches and high-performance simulation solutions such as Hardware Simulation Accelerators and Hardware Emulators that have been in use in hardware and electronic system design centers for approximately the last decade. In particular, in order to accelerate functional simulation, hardware emulation is used so as to offload calculation-intensive tasks from the software simulator. However, the communication overhead between the software simulator and hardware emulator is becoming a new critical bottleneck. We tackle this problem by partitioning the code running on the software simulator into two sections: the testbench HDL (Hardware Description Language) code that communicates directly with the Design Under Test (DUT) and the rest C-like testbench code. The former section is transformed into synthesizable code while the latter runs in a general purpose CPU. Our experiments demonstrate that the proposed method reduces the communication overhead by a factor of about 5 compared to a conventional hardware emulated simulation.*

1 Introduction

It has been a common practice for hardware engineers to build prototype boards in order to test their designs and to provide the software engineers a platform on which they could develop their code. Such boards take time to build and maintain, which significantly impacts project schedules and budgets. In addition, semiconductor vendors have always spent a significant amount of resources developing special-purpose software to accelerate the development phase of each microprocessor, microcontroller, or application-specific system.

Moreover, in the last decade, hardware accelerators**,** designed primarily to speed-up front-end simulation, have been available to large design centers with large budgets and design tool support. Hardware emulators on the other hand have also been available as a moderate-cost solution, mainly satisfying the needs of back-end verification.

The hardware accelerator is based on using circuit boards populated with multiple special-purpose ASICs, each of which contains a number of specialized processors and lots of local memory (typically 80% to 90% of these devices are memory). In this case, the HDL representation of the design is compiled into machine code, which is subsequently distributed amongst the various processors. The alternative, the hardware emulator, is to use circuit boards populated with FPGAs, in which case the HDL design is typically synthesized into a gate-level equivalent, which is partitioned across, and loaded into, the various FPGAs. In this case a co-processor executes the non-synthesizable code such as the testbench.

These approaches lighten the burden of hardware design verification by using custom hardware to aid the verification process. However both approaches suffer from the demanding communication between the software and the hardware sections of the system.

2 Related Work

Hardware simulation accelerators and hardware emulators have been in use in hardware and electronic system design centers for approximately the last decade. Nowadays, with the rising design complexity, there is an increased interest in such technologies. Speeding up simulation and verification of complex embedded systems can save design teams a lot of money and effort. Therefore, more and more companies build systems for hardware emulation. On the other hand, the market and relevant product offerings for simulation acceleration systems is still quite limited, due to their high cost.

Among the very limited available simulation acceleration and emulator systems, we note the following:

- The *Palladium-II* system from Cadence [1], which supports hardware acceleration and in-circuit emulation, is speeding up verification 100 to 10000 times when compared with software-based RTL simulation. Palladium is described as an array of "massively parallel Boolean compute engines".
- The *Hammer* accelerator system from Tharas Systems [2], which contains up to 128 specialized processors connected through a proprietary backplane.
- The *Vstation Pro* from Mentor Graphics [3], which provides an environment for verifying complex designs from 1.6 to 120 million gates. The system supports RTL and gate-level verification at a speed of up to 1MHz within a simulation-like debug environment that allows 100 percent signal visibility into the design.
- The *Zebu-XL* system emulator from EVE [4], which can handle designs from 3M to 50M ASIC gates, and is aimed primarily at large-scale chip and system emulation applications. It is offered in a modular, 19

978-3-9810801-2-4/DATE07 © 2007 EDAA

inch rack-mountable configuration, which accepts up to 64 Xilinx Virtex-II XC2V8000 FPGAs.

- The *Riviera-IPT* system from Aldec [5], an FPGA-based PCI board, that is tightly coupled to Aldec's own software simulator. A single board has a capacity of up to 12M system gates. Multiple boards can be connected to the same PCI bus for larger capacity.

The hardware-software communication overhead of these systems has been addressed in the past. Verisity has developed eCelerator [6] which reduces this overhead by using innovative synthesis technology to transform the most frequently executed sections of e-testbenches in hardware. By shifting the computationally most expensive parts onto hardware, the tool achieves significant performance gains in the verification process.

In [7] the authors propose a methodology to reduce the communication overhead by exploiting burst data transfer and parallelism, which are obtained by splitting the testbench and moving a part of it into a hardware accelerator.

Moreover, [8] presents a synthesizable testbench architecture addressing the same problem, which is based on a defined instruction for standalone mode verification. A set of instructions describes transitions of a signal.

3 Communication Bottleneck

Hardware emulators allow designers to implement a circuit using FPGA devices instead of an ASIC, thereby running simulations of the circuit at a much higher throughput than a software simulator can provide. When emulators first became available, all of the circuit had to reside in FPGAs, but today's emulators can communicate with a software simulator and allow designers to use all the models that the software simulator supports.

Although ISS models, TLMs, and pure C or C++ models all provide system designers with the means to evaluate basic system architectures, they can not be synthesized and implemented on an FPGA. In practice, such testbench code runs in a software simulation environment which is usually a general purpose CPU that communicates with the synthesizable DUT. This leads to a communication overhead between the testbench and the synthesizable DUT. Using a software testbench in a hardware-assisted environment is likely to create a major communication bottleneck. Engineers using a testbench specifically designed for performance are likely to find that even though their testbench consumes as little as 10% of the total simulation time, they are still limited to, at most, 10x improvement in the emulated environment.

In this paper, we reduce the communication overhead by synthesizing the portion of the testbench code that directly communicates with the DUT and involves most of the transactions. In particular, the proposed process involves the following steps:

1. Partitioning of the testbench code into the Testbench HDL code that directly interfaces to the DUT, and the C-like behavioral models that interface to the Testbench HDL code.
2. Transform the Testbench HDL code part into synthesizable code.

The transformed testbench code is synthesized along with a library that it is provided by a hardware simulator. In this way, the demanding communication path between the testbench and the DUT is transformed in hardware and therefore it is performed in a much faster way.

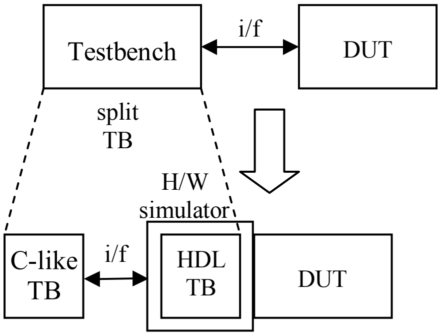

Figure 1. Splitting of the Testbench.

4 System Architecture

A high-performance verification system should incorporate both processors and FPGAs. A processor-only or FPGA-only solution is limited in terms of performance or flexibility in simulating various types of models.

First, in terms of the performance achieved, the maximum clock frequency of FPGAs lags behind that of processors implemented in contemporary ASIC. Therefore, processors with higher clock frequency execute behavioral models faster than FPGAs. On the other hand, FPGAs are more appropriate for executing simultaneous events and computation-intensive processes in parallel. Moreover, testbenches are commonly created using HDL such as Verilog or VHDL, sometimes including C-like programming language linked to an HDL simulator through e.g. the Programming Language Interface (PLI). This technique is used when the testbench needs to simulate more complex and more abstract functions. FPGAs are not capable of simulating models created in C-like languages and/or behavioral HDL that is not synthesizable. Therefore, processors and FPGAs have mutually complementary natures for high-performance verification systems. Modern large FPGAs incorporate general purpose CPUs which facilitates the FPGA-CPU communication path.

In the proposed architecture, shown in Figure 2, a built-in CPU (hardcore) located on the FPGA runs C-like behavioral testbench code, executes testbench floating point expressions, holds testbench large arrays, and accesses external files. Optionally, a memory controller and a FPU can be used to offload the tasks of this CPU.

The hardware simulator generates a simulation clock that coordinates the flow of the simulation. The time resolution of the simulator is defined by that of the testbench. The transformed HDL testbench block can pause the whole simulation environment in order to send requests such as

PLI calls, memory references, file accesses or floating point execution. The HDL testbench block provides all the input signals including the clock signals to the DUT.

The *server block* is responsible to serve the requests from the HDL testbench block. This block can access a CPU in order to execute PLI calls. The large arrays of the testbench code are stored in an external memory.

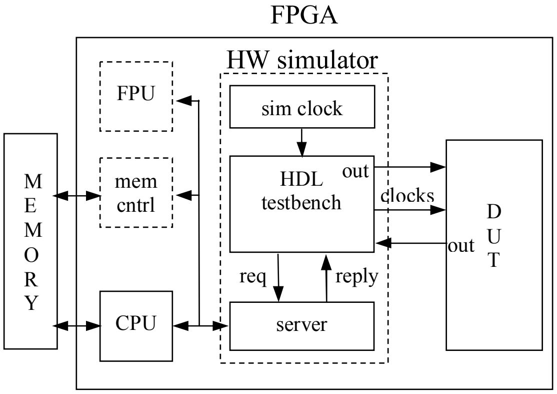

Figure 2. FPGA Emulator Architecture

In the proposed architecture the communication bottleneck between the software part and the hardware part of the simulation is pushed in the server-CPU interface. The accesses on this interface are infrequent. Moreover, this is a fixed interface, independent of the emulated DUT.

4.1 Testbench Transformed Structure

The original VHDL testbench is transformed into synthesizable code that can run in the environment provided by the hardware simulator. The tool we developed transforms a testbench written in VHDL language. However the same concepts can be applied to a Verilog testbench. The *process* body of a VHDL testbench includes various code sections that are not synthesizable. Such portions are mainly timing statements such as the VHDL *wait* statement, large arrays that are impractical or even impossible to be mapped onto FPGA registers, floating point calculations and file handling. The large arrays and the files are stored in the external memory that is accessed by the CPU and the memory controller. A VHDL process of the transformed VHDL testbench running in the hardware simulator can access the CPU and the external memory by sending requests to the server block of Figure 2.

We have enhanced the functionality of the VHDL processes in the transformed testbench in such a way that they can pause the simulation time of the hardware simulator in order to transfer requests to the server block. In every clock cycle the hardware simulator serves all the pending requests before advancing the simulation time counter. The processes in a VHDL language form a tree structure. We use this tree structure to transfer the requests from the body of the process to the server block. A code that receives requests from the process body and forwards them to a scheduler block is attached to the process. In every VHDL *module* a scheduler block is responsible to gather the requests from all its processes and advance them a layer higher in the VHDL hierarchy. This scheduler block can serve the requests in any order since the simulation is paused when any request is pending. This process is illustrated in Figure 3.

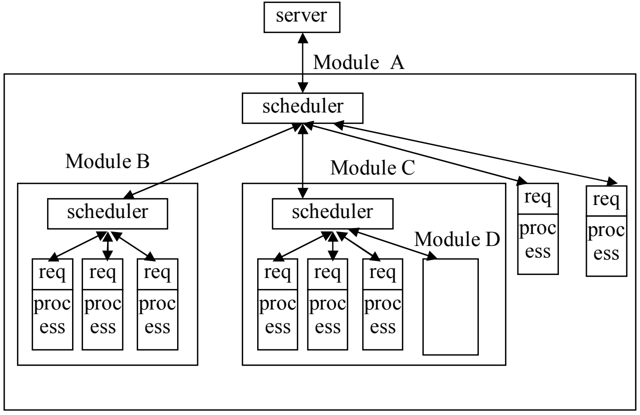

Figure 3. Tree-like Scheduling of Requests.

4.2 Simulation Clock

The hardware simulator provides the simulation clock that coordinates the functions of the simulation. A hardware counter keeps track of the simulation time. All timing references in the testbench code are translated to simulation clock cycles.

The simulation clock depends on the clock of the synthesized VHDL testbench. In particular, every simulation cycle is divided into four simulation ticks, and each tick is equal to one clock cycle of the synthesized testbench. These four tick time intervals are essential for the operations performed by a transformed VHDL process during a simulation cycle as the next section clearly demonstrates. Upon a request from a process the simulation stalls and the simulation cycle starts over.

4.3 Testbench Simulation Flow

The transformation of the testbench code is process-based. During a simulation tick a VHDL process either (a) executes a code segment or (b) waits for the transition of a signal or (c) waits for some time interval or (d) sends a request to the server block. In order to achieve the aforementioned functionality every VHDL process is transformed according to the FSM shown in Figure 4.

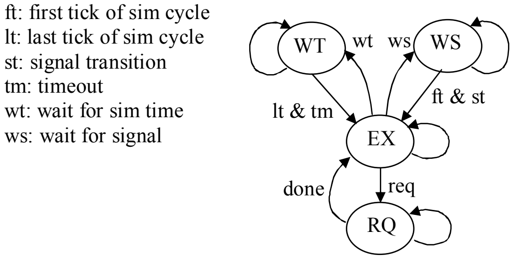

Figure 4. Process State Transition Diagram

In the *EX* state the process executes the synthesizable code of the testbench. A process stays in the *EX* state for 1 or 2 cycles depending on the original code. On a timing statement, such as the VHDL *wait* instruction, the process jumps in the *WT* or *WS* state. Finally the process enters the *RQ* state in order to send a request to the server block.

An example timing diagram that shows three processes and their state transitions is shown in Figure 5.

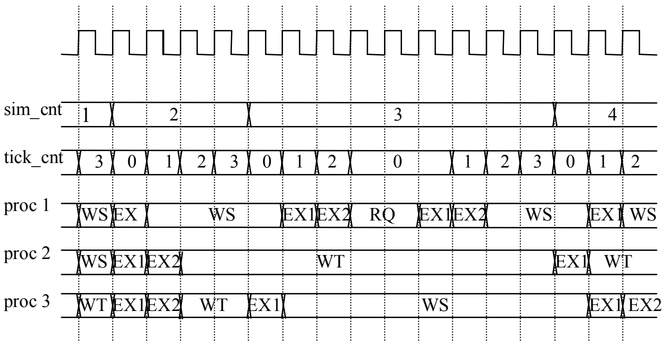

Figure 5. Process Timing Diagram.

The clock signals generated from the transformed testbench are fed to clock buffers of the FPGA that drive in their turn the clock trees of the DUT. The transition from the *WT* state to the *EX* state can happen in the last tick of a simulation cycle while the transition from the *WS* state to the *EX* state can happen in the first tick of a simulation cycle, as shown in Figure 4. In this way, the synchronous signals of the transformed testbench change their values one simulation tick after the clock signals change their values and thus we prevent setup and hold time violations of the signals sent from the testbench to the DUT. This is depicted in Figure 6. Assuming that the *clk* and *val* signals are sent to the DUT, the *val* signal will arrive one tick after the *clk* signal which is certainly the correct behavior.

Original Testbench Code

```
process proc1                      process proc2
  begin                              begin
  clk <= '0'; wait for 10 ns;        val <= '0';
  clk <= '1'; wait for 10 ns;        wait until clk'event and clk='1'
end process;                         val <= not val;
                                   end process;
```

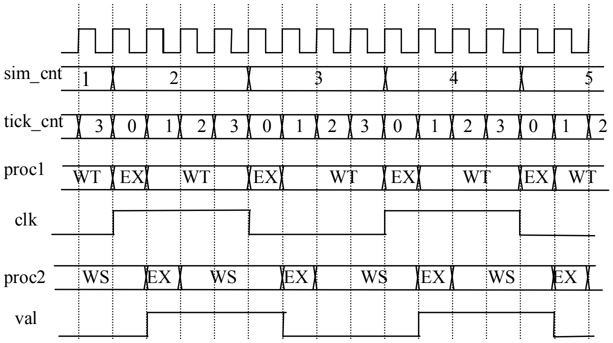

Figure 6. Setup and Hold Time Violations Prevention.

4.4 Pause and Resume Process State

Whenever a VHDL process executes a wait instruction or sends a request to the server block the process must stall, pause its state and resume at some time later. In order to add this functionality to a process all the statements in its body are transformed to conditional statements. Any point in the process body can become an exit point by setting an exit condition at that point. Similarly the last exit point can become an entry point using conditional instructions.

Take for instance the code segment below and its transformation. The wait instruction becomes the exit point when it is first executed and the entry point after 10 simulation cycles, assuming that the simulation cycle is 1 ns.

```
Original code:
  If clk = '1' then
     val <= '1';
     wait 10 ns;
     val <= '0';
  end if;

Transformed code
  If reset = '1' then
     exit_point := 0;
  else
     If (exit_point = 0 or exit_point = 1) and clk = '1' then
        if exit_point = 0 then
           val <= '1';
        end if;
        if exit_point = 0 then
           proc_state <= WT; -- enter WT state
           wait_time <= 10;    -- stay in WT for 10 sim cycles
           exit_point := 1;      -- exit point
        elsif exit_point = '1' then
              exit_point <= 0; -- entry point
        end if;
        if exit_point = 0 then
              val <= '0';
        end if;
     end if;
  end if;
```

If a process can pause its state at any instruction and resume it at some time later then non-blocking assignments may become blocking assignments by mistake. In order to avoid this erroneous behavior we transformed all the non-blocking assignments of the original code to blocking assignments by using extra variables. Every extra variable corresponds to a variable assigned in a non-blocking assignment. The extra variable holds the value of its corresponding variable in the last simulation cycle. A VHDL process in the transformed code assigns the values of all the extra variables in the last tick of every simulation cycle.

Consider the code segment below and its transformation. All the non-blocking assignments of the original code are transformed to blocking assignment in order to avoid having a non-blocking assignment after the wait instruction.

```
Original code:
  process
  a <= '1';
  wait 10 ns;
  b <= a;
  …
  end process;

Transformed code:
  process  begin -- fix non-blocking assignments
     if last_tick = '1' then -- last tick of simulation cycle
        a_last <= a;   -- all variables in non-blocking
        …                    -- assignments
     end if;
  end process;
```

```
process begin
If exit_point = 0 then
   a = ’1’;
end if;
if exit_point = 0 then
   proc_state  <= WT; wait_time <= 10; exit_point <= 1;
elsif exit_point = 1 then exit_point <= 0;
end if;
if exit_point = 0 then
    b <= a_last; -- use the old value of a
end if;
…
end process;
```

4.5 Testbench code Transformations

Several other functions are performed by the tool we developed, so as to be able to reduce the communication overhead in a hardware emulator environment. Briefly, we mention the following code transformations:

- Timing references are transformed to simulation cycles.
- Large multi-dimensional arrays and their references are transformed into one-dimensional arrays in order to simplify their mapping to the external memory.
- *VHDL assertion* statements are sent to the external CPU.
- VHDL a*fter* statements are transformed into VHDL processes that are triggered when the after statements are executed.
- VHDL *select* statements are transformed into *if/else* statements.
- Processes that describe combinational logic which sends requests to the server block are transformed into sequential logic that is clocked with the simulation clock.

5 System Evaluation Environment

In order to evaluate the tool and quantify the proposed methodology we created a typical hardware emulator system in which we applied the proposed framework.

In particular, we assumed that the hardware emulator uses a Xilinx Virtex-2P FPGA which is a widely used state-of-the-art FPGA. We built an example DUT along with its testbench in order to measure the performance of the emulator. The whole system with the transformed testbench and the DUT was synthesized using the Xilinx ISE 7.1 synthesis tool.

The DUT is a simple VHDL code that accesses a parameterized number of SRAM chips. The number of memory chips is defined whenever a new system is built for evaluation. The original testbench includes the VHDL model of a 32-bit ZBT SRAM chip from Micron Technology, Inc [9].

The tool transforms the testbench along with the memory model to synthesizable code. The large memory arrays of the model are stored in the external memory of the emulator that is accessed by the PowerPC. The transformed model makes memory requests to the server block which in turn forwards the requests to the PowerPC.

5.1 Measurements and Comparison

The Xilinx ISE tool reports that the FPGA system which consists of the DUT, the testbench runs at 125 MHz. Therefore the simulation tick time is 8 ns and the simulation cycle is 32 ns. The PowerPC can also run at 125 MHz. The critical path is in the SRAM model as expected. An SRAM memory request takes 15 cycles in average. The DUT performs one SRAM request to every SRAM chip every 400 cycles. Taking all the aforementioned parameters in account we can measure the simulation time and frequency. The results are depicted in Figure 7.

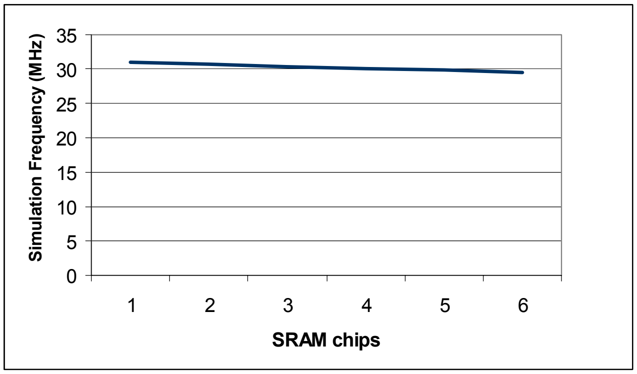

Figure 7. Simulation Frequency.

We notice that the number of SRAM chips slightly affects the overall simulation time. This is because the number of the SRAM devices does affect the number of signals between the DUT and the testbench, but this is the portion of the testbench that is transformed to synthesizable code. As this figure clearly demonstrates we can execute the testbench of 6 SRAM commercial controllers at a speed of 30MHz. This is certainly a significant improvement over the execution of the approximately 300 lines of HDL code of each testbench in a software simulator general-purpose CPU.

In a conventional emulator the DUT communicates to the testbench code through a fast off-chip link which connects the FPGA running the DUT with a stand-alone CPU, such as the PCI Express, that today has a raw bandwidth of at most 4Gbytes/sec, or some other fast communication link. Over this link all the signals of the interface between the testbench and the DUT should be sent at a rate derived by the DUT clock speed (i.e. all the signals should be sent every clock cycle). The number of the interface signals in the current real-world design we used is *sram_chips* * 90, where *sram_chips* is the number of SRAM chips of the DUT (since each SRAM chip has about 90 usable I/O pins). Therefore, in the conventional approach the number of the SRAM chips heavily affects the communication overhead and therefore the overall performance of the system, since the number of the interface signals that should be sent over a clock cycle is proportional to the number of SRAM devices. Moreover, on top of those signals there exists some communication protocol overhead between the DUT and the testbench that can further limit the performance. Since we have synthesized the testbench, we have significantly reduced the communication cost between the

CPU and the DUT; the FPGA which now implements both the DUT and the synthesized testbench communicates with the CPU only when a PLI call (or a memory access request) is issued and this is done very infrequently. We measured the simulation frequency of the aforementioned system in order to compare it against our proposed system. In particular, we measured the communication bandwidth required by the proposed framework and compared it with that of a conventional emulator, assuming that in the standard emulator case the communication protocol overhead between the DUT and the CPU is negligible. Figure 8 shows the simulation speed we derived using our methodology.

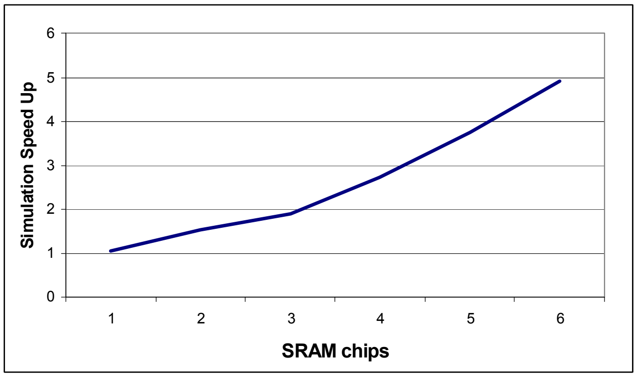

Figure 8. Comparison of the proposed architecture.

So assuming our typical DUT with 600 pins, the proposed architecture can speed-up the simulation by a factor of about 5 compared to a conventional hardware emulated simulation, at the cost of only running our simple script and then synthesize the resulting code using a conventional FPGA EDA flow.

These results are *in favour of the conventional emulator* since it was hard to estimate its performance and therefore several delays such as the time consumed by the PCI driver or the Testbench-DUT communication protocol overhead were assumed to be zero.

6 Future Work

So far we have built a tool that can transform testbench code with simple VHDL instructions. We plan to extend it in such way that it can transform any VHDL code. Apart from the SRAM model it has transformed successfully a 128 Mb SDRAM DDR model from Micron. However, the tool does not support all VHDL constructs yet.

Moreover, we plan to use more than one embedded CPUs in the system. The server block can send many requests in parallel to many CPUs. If the requests sent to each CPU are independent (they access different memory areas) the CPUs can work independently.

7 Conclusion

Hardware emulators and FPGA prototypes have long provided the highest performance when compared with all the verification approaches in the industry, but they have also suffered from a number of severe drawbacks. One of the most important problems is that complex emulator systems demand high communication throughput between the testbench and the synthesizable DUT which can eventually limit the performance of the simulation. To address the above shortcoming, we proposed to split the testbench into two sections and transform the portion of the testbench that communicates very frequently with the DUT to synthesizable code. We built a tool that provides a way to synthesize a behavioral VHDL code in a hardware simulation environment. We claim that we can overcome the testbench-DUT communication bottleneck and therefore increase the capabilities of today's hardware emulators by up to 500% when applied to real-world systems.

References

[1] Cadence, *Palladium Accelerator/Emulator*, http://www.cadence.com/products/functional_ver/palladium/

[2] Tharas Systems, *Hammer SX and MX hardware accelerators*, http://www.tharas.com/products/

[3] Mentor Graphics, *VStationPro,* http://www.mentor.com/products/fv/emulation/vstation_pro/

[4] EVE, *Zebu hardware emulator*, http://www.eve-team.com/products.html

[5] ALDEC, *Riviera*, http://www.aldec.com/products/riviera/

[6] Verisity, *eCeleretor Testbench Acceleration*, http://www.verisity.com/products/ecelerator.html

[7] Young-Il Kim, Wooseung Yang, Young-Su Kwon, Chong-Min Kyung, *"Communication-Efficient Hardware Acceleration for Fast Functional Simulation"*, pp. 293-298, Design Automation Conference, 41st Conference on (DAC'04), 2004.

[8] Ho-seok Choi, Seung-beom Lee, Sin-chong Park , *"Instruction Based Synthesizable Testbench Architecture",* IEICE TRANSACTIONS on Electronics Vol.E89C No.5 pp.653-657, 2006

[9] Micron Technology, http://www.micron.com/