Easy Steps Towards Virtual Prototyping using the SystemVerilog DPI

Dave Rich
Mentor Graphics, Inc.
Fremont, CA
dave_rich@mentor.com

***Abstract*— The hardware and software worlds have been drifting apart ever since John W. Tukey coined the terms "software" and "bit" back in 1958. Tukey introduced these terms as computers were evolving from electromechanical to electronic components. *Hardware* had long meant something you can touch and typically assemble from parts into a larger system. *Software* has come to mean that which you can't touch, yet that which you can change without touching the hardware. In the early age of software development, programmers required extensive knowledge of proprietary hardware architectures in order to write the programs that executed on them. Today, software programming has evolved to standardized languages, like C/++ and Java so programmers can write code independent of the hardware architecture the programs are running on. Finding engineers experienced in both disciplines is becoming very difficult making communication between software and hardware engineers daunting to say the least. This is further complicated because of different modeling languages used by each discipline and different abstraction levels required during each phase of a project.**

Virtual Prototyping is an evolving methodology for the verification of software and hardware in a single environment that is designed to catch these communication breakdowns. Performance of this virtual prototype is critical to the successful completion of this verification task. Execution of software on simulated hardware models can be many orders of magnitude slower than the software executing on the real target hardware, so a virtual prototyping methodology partitions the execution of software and hardware into the proper abstraction level to achieve the desired performance versus accuracy trade-off.

This paper discusses various virtual prototyping methodologies available along with the verification and performance goals each is optimized to address. It will explain the trade-offs considering the different perspectives that hardware and software engineers are able to understand. In particular, this paper will demonstrate a virtual prototype using the modeling interface provided by the SystemVerilog "DPI-C" construct that bridges the C software world with Verilog Hardware Description Language (HDL) world.

Additionally, this paper will explain mechanisms for transaction-level communication between hardware and software using a UVM testbench. It will demonstrate software transactions on the C side that are converted into sequences of bus cycles represented by calls to the UVM register abstraction layer. This makes the hardware verification environment considerably reusable with the virtual platform environment.

I. Introduction

By definition, a System-on-Chip (SoC) device is the blending of software and hardware domains. The architecture of an SoC is a compromise of trade-offs between implementation of tasks in hardware or software based on a number of performance goals. Verification of an SoC is usually divided independently into their respective domains before full system-level verification begins on the actual platform, or a representative prototype such as a hardware FPGA or emulation system implementation. Unfortunately, the availability of a full system usually comes too late in the project cycle to get the desired amount of verification completed in time. Virtual prototyping is a methodology that addresses this problem by getting earlier access to a full system using a variety of software emulation and simulation techniques.

If you look at the typical components that make up an SoC, you may see what is shown in Figure 1 [1].

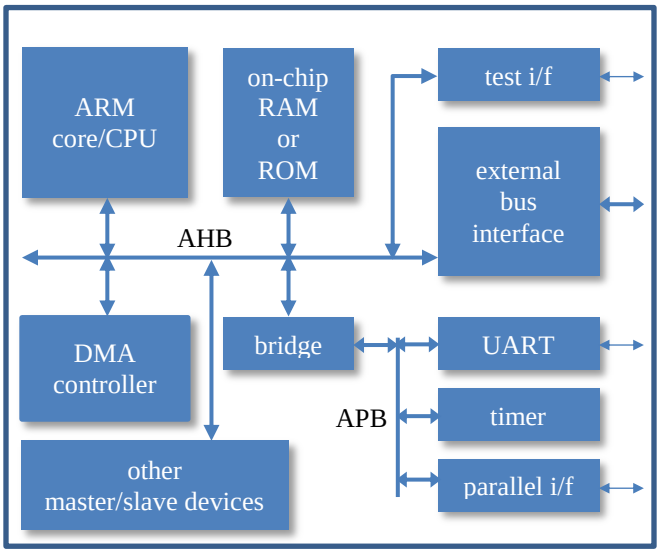

Figure 1 Typical SoC Block Diagram

1. An embedded processing unit, or units, and its sub-system for executing instructions, the core.
2. A bus or fabric that connects all the processing units and devices together; AHB and APB, as in this example.
3. A set of instructions in software/firmware that manage the entire SoC, executing each processing unit, represented by data in the RAM/ROM
4. A set of hardware devices that communicate through standard protocol interfaces with devices external to the SoC.
5. A set of hardware devices that manipulate data external to the processing units; other master or slave bus devices, a DMA controller as in this example.

In most cases, the processing units as well as some of the hardware devices are acquired Intellectual Property (IP) and are considered pre-verified by the organization that developed them. Some of the software is also acquired, such as the embedded OS and the drivers that go along with the acquired hardware devices. That leaves the critical task of verifying the application software, the application hardware, and all the communication mechanisms that connect the SoC together.

II. Verification Architecture

The methodologies for verifying hardware and software separately are fairly well understood. Each methodology has a set of practices that require some intimate knowledge of their respective domains. Since an SoC uses a fairly standard protocol for communication across the system, it is relatively easy to verify each individual component in isolation. However, the task of verifying the communication between those domains becomes a struggle without having the complete platform available.

A virtual prototype addresses a portion of this problem by providing high performance models for the missing hardware components. The software components may be cross compiled onto the same host processor running the virtual prototype, which is typically referred to as *native code* or there may be a software emulation of the target processer running on the host.

On the actual platform, an embedded processor communicates with other components of the SoC using address mapped memory references. Taking advantage of standard protocols, a virtual prototype represents communication between components in terms of an application-specific procedural interface (API). The memory references generated by the embedded processor need to be converted into API calls in simpler prototypes, or intercepted by the virtual prototyping system, and routed by a software backplane, in Figure 2, to call the appropriate software model.

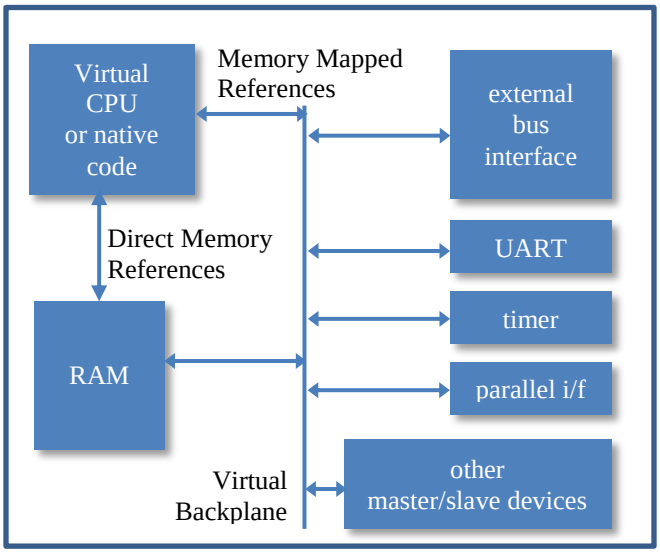

Figure 2 Virtual Prototype Backplane

The virtual prototype achieves high performance by having a host processor running nearly as fast as the target processor, and bus cycles abstracted into simple, transactional function calls. Performance of a virtual platform may be within one or two orders of magnitude of the actual system, which may be enough to run the full embedded OS. Understandably, introducing hardware-level description models into this virtual platform will have a tremendous impact on performance, limiting simulations to smaller portions of the OS or stripped down segments of code.

Verification of hardware can usually begin for each component individually without waiting for a significant amount of the system to be in place. One or more components can be connected to a physical bus for sub-system simulation. Traffic on each bus may be generated by drivers provided by Verification IP (VIP) in place of the actual target processor.

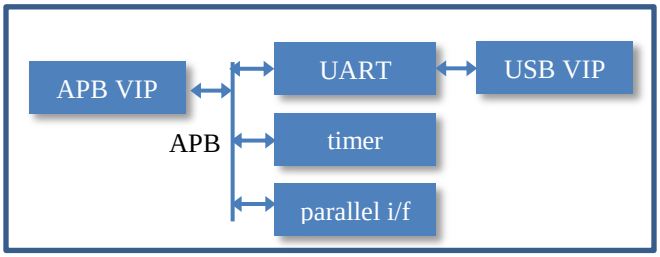

Figure 3 Block Level Verification Environment

Performance of this hardware verification environment is reasonable because the processor has been replaced by a simplified bus functional equivalent, and only the necessary hardware is in place. There is no need to verify the complete hardware design as a full system if certain components have been verified individually, such as the processor itself. Ideally, only the communication between components need be verified at the hardware level.

Correspondingly, there is no need to verify the hardware by running software on the processor. Doing so would be prohibitive in terms of performance because of the extreme number of redundant hardware bus cycles. The only remaining need is to verify that the software can properly access the hardware components.

Traditionally, the most efficient way of executing software that accesses hardware models has been to drive the physical bus with a high-level processor model, like an instruction set simulator (ISS). However, the ISS model is burdened with the task of driving bus signals with pin-level or cycle level accuracy.

If we can combine the most efficient methodologies from the virtual platform environment with the most efficient methodologies from the hardware verification environment, we can achieve our desired performance target.

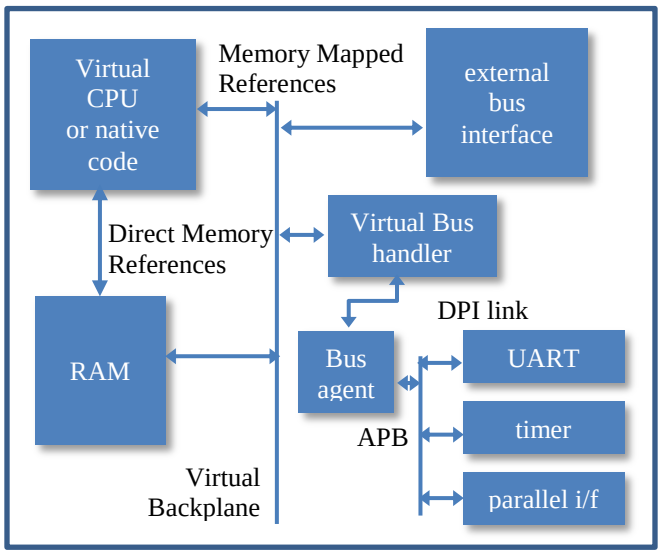

Figure 4 Mixed Software/Hardware Simulation

The SystemVerilog Direct Programming Interface DPI was designed to integrate the C language with the hardware description language at the software level of procedural calls. In contrast, the programming language interface (PLI) was designed for pin-level transactions and other tool interfaces.[2] This DPI link allows us to keep the communication between the virtual bus handler and the bus agent at the software abstraction level of procedure call transactions. The fact that the virtual platform can be run at one level of abstraction while the hardware environment is using another level of abstraction has a significant impact on performance as well.

III. Modelling Abstraction Levels

The choice of abstractions levels is very dependent on the particular application. A key point to remember is that the goal of creating this mixed verification platform is to verify communication between software and hardware, not all the software and hardware in total.

In the simplest terms, levels of abstraction are characterized by timing accuracy. The OSCI[1] SystemC TLM 2.0 standard[4] introduces terminology for coding styles that we can refer to here:

- Untimed (UT) – limited or unspecified timing accuracy. At this level, only ordering of operations matters and there may be no bookkeeping of elapsed simulated time.
- Loosely-timed (LT) – time is broken into slices or some quantum unit. An SoC virtual platform is likely to choose the execution of an instruction as its quantum time unit.
- Approximately-timed (AT) – Quantum units are broken down into phases and the tracking of elapsed simulated time is enough to gather relative performance statistics.
- Cycle-accurate/cycle-callable (CC) Timing is accurate enough to run in lock-step to match the hardware models at a pin-level, clock or bus-cycle boundaries.

On the software side any one of these abstraction levels may be attained. When you cross-compile code written for your target processor to run natively on your current host, you may not care about the timing, only the functionality of the code. There might not even be a way to represent the target processor's timing. If you compile your code to run on an ISS model of your target processor, that model may have statistics for the instruction execution times for each instruction and simply provide a summation of the results. That would be loosely timed. To model the effects of a pipelined system may require that you use an approximately-timed model.

The hardware modeling side has fewer choices. Most synthesizable designs require cycle-accurate or lower level abstractions than what has been described here. However, the testbench verification modeling can be at a much higher level.

Methodologies such as the UVM let you treat the verification architecture more like software. The UVM's register abstraction layer is designed so that you can write hardware verification tests by providing memory address maps that either generate or intercept memory references from simple procedural calls written in your testbench.

IV. Synchronization between Abstraction Levels

At the simplest level, let us think of our software as native C/C++ code running on a host processor, where our host processor is also where the simulation of hardware in SystemVerilog is running. We can use the SystemVerilog DPI to blend the two languages by having the software call a function that represents a bus transaction. In more sophisticated virtual prototyping systems, memory references are intercepted and routed via a table to the appropriate handler, but the end result is the same: a memory reference becomes a procedural call.

The key feature of the SystemVerilog DPI is that it lets a SystemVerilog function or task appear as a simple routine on the C side, and a C routine appears as a task or function on the SystemVerilog side. A subset of compatible argument types may be accessed transparently without any knowledge of the language used to call the routine. Compatible types are those with the same semantic meaning and representation in memory without any need of conversion.

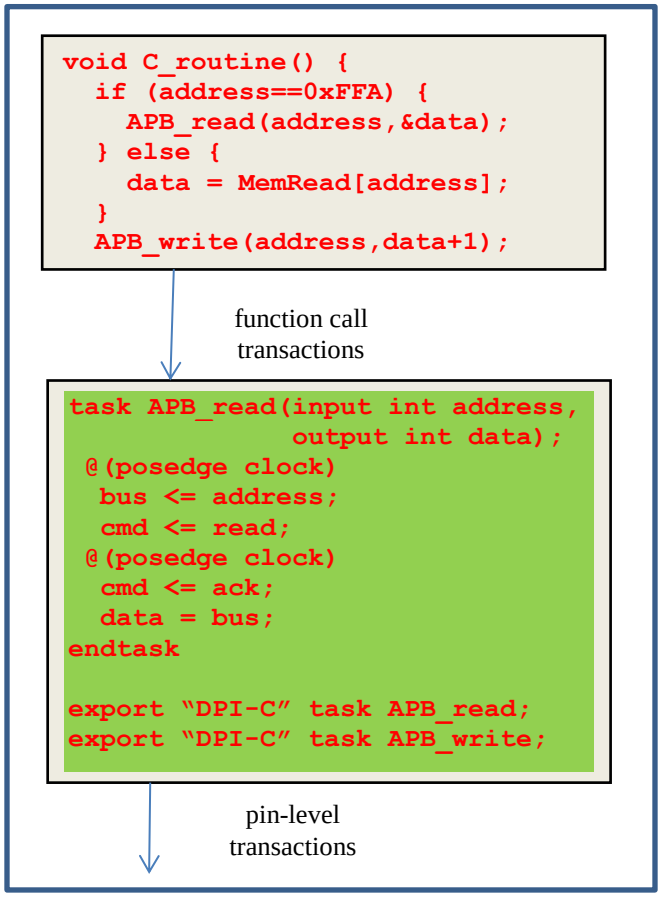

Figure 5 Software to Hardware DPI link

In the code fragment shown in Figure 5, the C code has no notion of timing except when it calls the DPI exported tasks which are both cycle accurate on the SystemVerilog side. The C_routine calls and is blocked until the APB_read and APB_write tasks have completed. While the C code is blocked, simulated hardware time advances on the SystemVerilog side waiting for each clock cycle. This is explained further in Figure 11 Inter-process Communication.

[1] OSCI is now the Accellera Systems Initiative

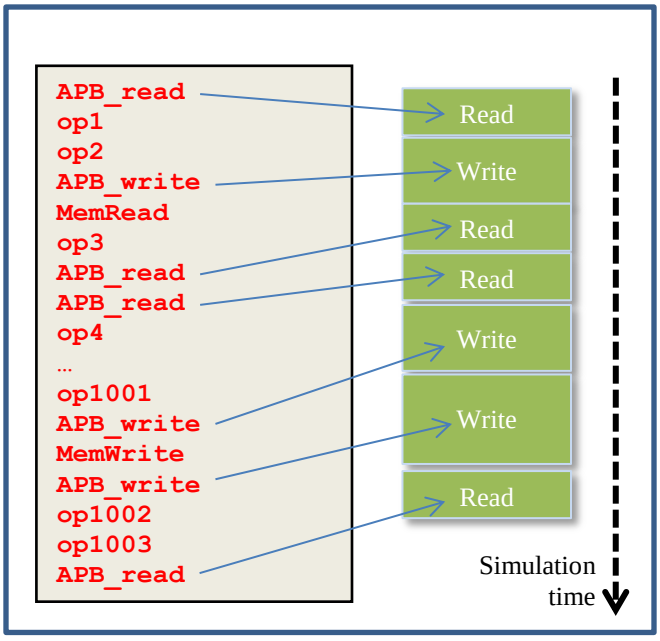

Figure 6 Compressed Hardware Simulation Timing

In the situation shown in Figure 6, hardware simulation time only advances during the bus cycles chosen by the virtual prototype to model in hardware. There may be thousands of consecutive idle hardware bus cycles between active bus cycles. The hardware design might require additional simulation time during idle bus cycles to execute other activity accurately. The virtual prototype may also have its own concept of time by counting the number of instructions executed. We can provide an idle task that advances time to gain some extra level of timing accuracy that represents some percentage of idle bus activity. For optimal performance, there is no need to account for every idle bus cycle.

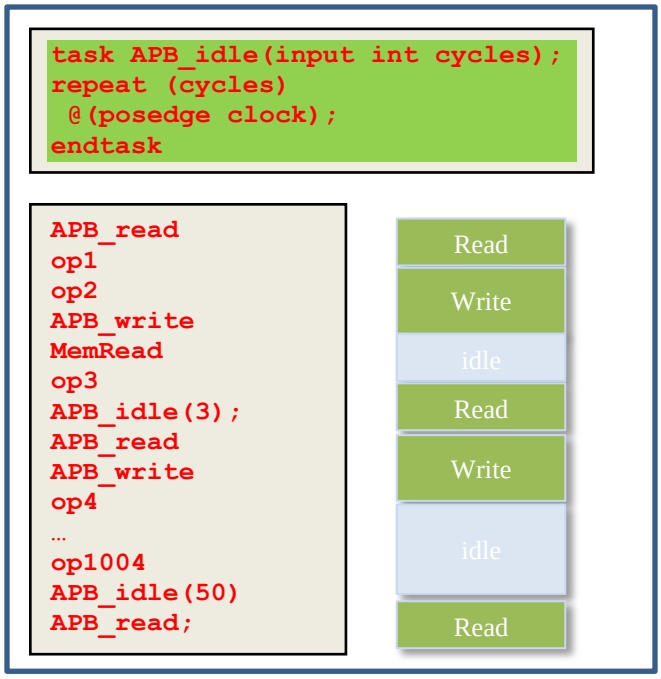

Figure 7 Approximated Hardware Simulation Timing

Instead of injecting a separate idle bus operation, you could add the instruction count to each bus operation and have SystemVerilog inject some constrained random number of bus cycles.

Some designs have interrupts that need to be accounted for. The idle task (as well as any bus cycle task) may be modified to report back that an interrupt was requested.

```
task APB_idle(
input int requestedCycles,
output int iRequested,
output int actualCycles);
int i;
fork
  for(i=0;i<requestedCycles;i++)
              @(posedge clock);
  @(IRQ!=0);
join_any
disable fork;
actualCycles = i;
iRequested = IRQ;
endtask
```

Figure 8 Interrupt Monitor

A completion model between the hardware and software prototypes needs to be agreed upon. Depending on the application, the hardware side can either just report that the interrupt was requested during the bus operation just

completed, or abort the operation and return immediately stating the operation was incomplete.

V. FLOW OF TRANSACTIONS

For easier maintenance and reusability, define a standard transaction instead of creating individual routines and various numbers of argument lists. A transaction is simply a structure of data that that represents a common set of arguments that are typically passed to a method. They can be packed and unpacked into a simple array of bytes making data transfers across the inter-language boundary much simpler. The design of a transaction is show in Figure 9:

Field	*Type*	*Description*
Operation	Enum	Read, write,idle,burst
Address	32-bit	Physical starting address
ReqStart	Time	Time of Request
ReqDuration	Time	Time allocated for operation
TrStart	Time	Actual start time
TrEnd	Time	Actual end time
InterruptMode	Enum	Ignore, Complete, Abort
InterruptReq	Enum	None,Requested
InterruptTime	Time	Time of Interrupt
Length	Int	Size of data
Payload	nBytes	Transaction data

Figure 9 Transaction Specification

Depending on the operation, only some of these fields may be populated at the start of the transaction, others will be filled in at the time of completion.

One important consideration in a mixed language environment is that each side of the language boundary is responsible for allocation and management of the memory used by a transaction. In the earlier examples, we just passed a compatible data type, like an *int*, as an argument that was copied as the call was made. This is fine as long as the size of the argument is a known fixed size to both languages; passing a pointer to a block of memory of unknown size is not allowed. Memory for transactions with dynamic sizes must be allocated on both sides before making the call. Some methods for dealing with this issue are:

- Allocate a fixed maximum size for your transaction and use the Length field to trim the Payload.
- Send the Payload in fixed sized blocks over multiple operations. So if your block size was 2K and you need to send a 5K payload, you would need to send the payload over 3 transactions.
- Split the transaction into two calls. The first call would specify the length of the payload to allocate, and the second call would use the allocated space from the first call.

VI. THE NEED TO BE THE MASTER

So far, we have not considered how to get the communication between the virtual prototype and the hardware simulation started. Software running on a processor runs in a limited number of threads; whereas hardware is by its nature represented by a prolific number of threads. In practice, hardware simulators do not represent threads the same way that software does, so there will need to be some kind of mapping.

Going back to the simplest environment where the software is just C code natively compiled on a simulation host and hardware, the C code can be started from the SystemVerilog side by calling a DPI-C routine that has been imported as a task. In this case, the SystemVerilog side is the *master* thread.

In the following example, the routine c_code has been imported as a task as opposed to a function which allows it to consume time. The way c_code consumes time is by calling an exported task, v_code.

```
int c_code() { /* C task */
  while(1) { /* C thread */
    v_code(args);
    ...
  }
}
```

```
module top;
  import "DPI-C" task c_code();
  initial
    c_code;  // start C thread
  bit clk; always #10 clk++;
  export "DPI-C" task v_code;
  task v_code(args);
    addr <= args;
    @(posedge clk);
    args = result;
  endtask
endmodule
```

Figure 10 Starting a C thread

However, most realistic virtual prototyping systems want to be the master starting their own threads and just have the API calls act as a slave to their threads. The most common way to make this happen is to use an inter-process communication socket (IPC)[6]. An IPC socket is a host OS mechanism for message passing. A server thread sends a message to a client and receives a response from a client thread. Both threads can be started independently of each

other as a master thread, but will block waiting to receive the message or response.

The only significant difference between the master and slave is the master process initiates the socket and the slave process connects to the socket. After that negotiation, the two processes can be completely independent of each other, using the IPC messages to synchronize with each other.

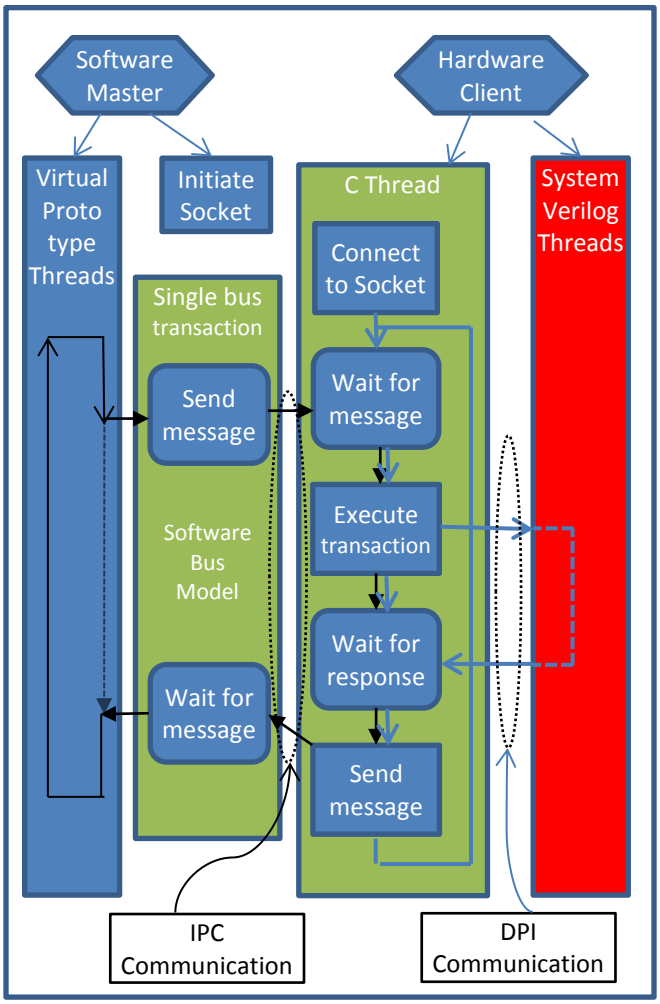

Figure 11 Inter-process Communication

The two loops in Figure 11 represent two independent processes. The software virtual prototype may have many processes, one of which the virtual backplane intercepts and calls a software bus model. The software model sends an IPC message to the C thread started by the hardware simulation client, and waits for a response message. The C thread is the other process, waiting to receive an IPC message, and then executes the transaction by calling the SystemVerilog bus functional models exported through the DPI.

VII. Integration with UVM

Reuse of the existing testbench from the hardware-only based verification environment is always encouraged. The sequence mechanism in the UVM provides a convenient mechanism to convert the bus transaction into a ready set of pin wiggles to the DUT for any particular hardware protocol. The SystemVerilog DPI enables the C code to start sequences to send transactions through the testbench that eventually reaches the interface to the DUT. The UVM's register package provides an additional layer that directs the appropriate memory addresses to the appropriate interface use a register map.

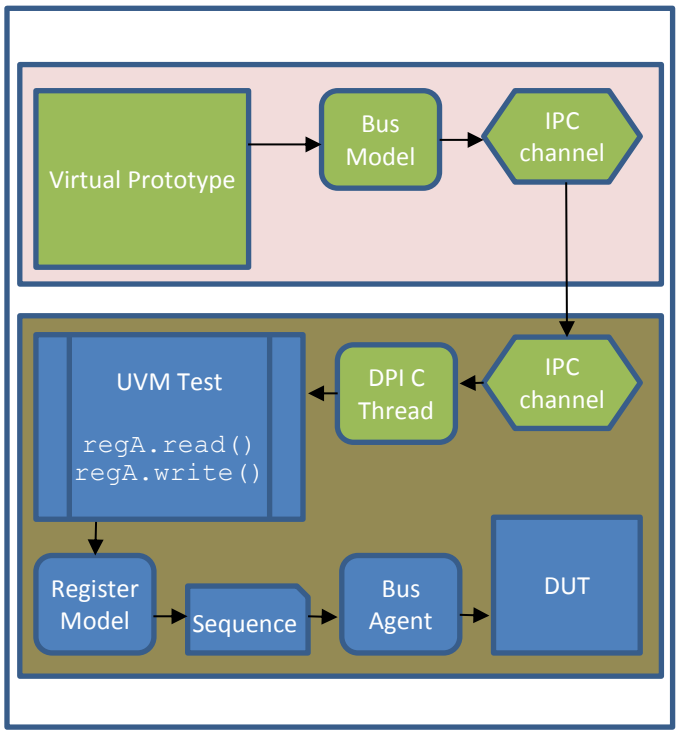

Figure 12 UVM Testbench Re-Use

In the environment shown above, UVM sequences are generated by either a pure UVM test, the virtual prototype process, or a combination of both.

VIII. Summary

The techniques shown in this paper have been deployed on several projects where the virtual prototype has been modeled in a variety of environments from internally developed platforms, open source Python based stimulus generators, and commercially available systems such as Wind River Simics.

There are a number of other issues that time did not permit going into detail in this paper, but have been address in some of these projects. These issues include dealing with multiple interface threads and controlling the interactive debugging environments of two processes communicating through IPC sockets

IX. REFERENCES

[1] Furber, Stephen B. *ARM System-on-chip Architecture*. Harlow, England: Addison-Wesley, 2000**.**

[2] "IEEE Standard for SystemVerilog- Unified Hardware Design, Specification, and Verification Language," IEEE Std 1800-2009, 2009.

[3] "Accellera UVM Reference Manual"

[4] TLM-2.0 Standard." *SystemC TLM (Transaction-level Modeling)*. <http://www.accellera.org/downloads/standards/systemc/tlm>.

[5] Peryer, Mark. *C Based Stimulus for UVM*. Mentor Graphics, n.d. Web. 06 Feb. 2013. <http://www.mentor.com/products/fv/events/c-based-stimulus-for-uvm>.

[6] Spear, Chris. *SystemVerilog for Verification: A Guide to Learning the Testbench Language Features*. New York, NY: Springer, 2008. 229+

[7] Edelman, Rich. *Using SystemVerilog Now with DPI*. Proc. of DVCon 2005, San Jose

[8] Intel,. *Full System Simulation with Wind River Simics*. Web. 06 Feb. 2013. <http://www.windriver.com/products/simics/>.