bluespec™

Delivering Synthesizable Verification IP for Test Benches

Case Study: SystemVerilog VMM vs. BSV for an Ethernet MAC test bench

Executive Overview

Summary

High-level verification languages and environments such as e/Specman, Vera and now SystemVerilog, as used in VMM or OVM, may be the state-of-the-art for writing test bench IP, but they are useless for developing models, transactors and test benches to run in FPGAs for emulation and prototyping. None of these languages are synthesizable. So engineers wishing to move verification assets onto FPGAs have been designing with RTL, the same old slow, resource-intensive and error-prone way.

But now, with the introduction of modern high-level languages for synthesizable verification IP, engineers can design test benches, models and transactors at a high level of abstraction and with extreme reuse, but *they can also synthesize them onto FPGAs* – and they can do this as easily as they do today in simulation-only verification environments. Imagine running your test benches, models and transactors at tens of MHz.

This White Paper outlines important attributes of, and the applications for, modern high-level synthesizable verification environments. Using the example of a test bench for an Ethernet MAC, the paper compares the implementation of a synthesizable test bench done with Bluespec's BSV with a non-synthesizable reference test bench done with SystemVerilog VMM – and it demonstrates that a synthesizable test bench can be implemented with fewer lines of code than using state-of-the-art SystemVerilog.

Outline:

Introduction
Features of an Ideal High-Level Synthesizable Verification Environment
Applications for High-Level Synthesizable Verification
Case Study: SystemVerilog VMM vs. BSV for an Ethernet MAC test bench
Conclusion

Key Features of a Synthesizable Verification Language

- Modern programming language constructs such as polymorphic types, user-defined overloading, structs and enums, tagged unions and pattern-matching, nested types and interfaces, and higher-order functions
- Transaction Level Modeling (TLM) mechanisms and constructs
- High-level concurrency abstraction
- Automatic Finite State Machine (FSM) generation
- Powerful parameterization
- Assertions
- Strong static verification so that verification designs are comprehensively checked at compile time
- Libraries and utilities
- Integration with co-emulation infrastructure for easily connecting from third-party FPGA boards/systems to host

Key Applications

- FPGA-based test benches to test DUTs at high-speeds
- Architectural and cycle-accurate models to run in SystemC or FPGAs
- Golden reference designs that can be run in emulation
- FPGA instrumentation for monitoring and gathering statistics
- Transactorizing RTL to run in SystemC/C/C++ environments or low-cost FPGA platforms
- Simplifying the modeling of architectural and cycle-accurate models
- Rapidly repartitioning test bench architectures for optimal co-emulation performance
- Extending Transaction Level Modeling (TLM) into emulation/prototyping for synthesizable models, test benches, and transactors

©2009 Bluespec, Inc. All rights reserved.

Would you still use RTL for your synthesizable test bench needs if there were a better way? What if you could synthesize from a modern, high-level verification language?

Introduced over ten years ago, high-level verification languages and environments such as *e*/Specman, Vera and now SystemVerilog, as used in VMM or OVM, provide tremendous advantages over the previous state-of-the-art: writing test benches with Verilog and VHDL RTL. As powerful as these verification languages are, they are useless for developing models, transactors and test benches to run in FPGAs for emulation and prototyping. None of these languages are synthesizable. So engineers wishing to move verification assets onto FPGAs have been designing with RTL, the same old slow, resource-intensive and error-prone way. That's no longer necessary. With the introduction of modern high-level languages for synthesizable verification IP, engineers can design test benches, models and transactors at a high level of abstraction and with extreme reuse, but they can also synthesize them onto FPGAs – and they can do this as easily as they do today in simulation-only verification environments.

This White Paper outlines important attributes of, and the applications for, modern high-level synthesizable verification environments. Using the example of a test bench for an Ethernet MAC to illustrate the power of a modern high-level synthesizable verification environment, the paper compares the implementation of a synthesizable test bench done with Bluespec's BSV with a non-synthesizable reference test bench done with SystemVerilog VMM – and it demonstrates that a synthesizable test bench can be implemented with fewer lines of code than using state-of-the-art SystemVerilog.

Features of an Ideal High-Level Synthesizable Verification Environment

Simulation based languages like e, Vera and SystemVerilog, have core attributes which promote rapid development of re-usable verification infrastructure and libraries with fewer test bench bugs, while keeping the description succinct and easy-to-understand. Because FPGAs have fixed resources, there are obviously limits to what can be supported in hardware. For example, while many types of assertions can be synthesized into hardware, there are assertions that cannot be, because they would require unacceptable, and sometimes indeterminate, amounts of hardware. Mindful of these exceptions, an ideal synthesizable verification language should support the same core attributes and features. Further, with the exception of those features which require too much hardware, all of them should be synthesizable, without subset. Essential features include the following:

- Modern programming language constructs such as polymorphic types, user-defined overloading, structs and enums, tagged unions and pattern-matching, nested types and interfaces, and higher-order functions
- Transaction Level Modeling (TLM) mechanisms and constructs
- High-level concurrency abstraction
- Automatic Finite State Machine (FSM) generation
- Powerful parameterization
- Assertions
- Strong static verification so that verification designs are comprehensively checked at compile time
- Libraries and utilities – more important than just providing off-the-shelf libraries such as for standard interfaces or random number generation would be the ability to create powerful, flexible and highly reusable libraries
- Integration with co-emulation infrastructure for easily connecting from third-party FPGA boards/systems to host workstations

Applications for High-Level Synthesizable Verification

A high-level synthesizable verification environment is not necessarily intended to be a replacement for simulation-based test bench languages and toolsets, such as *e*, SystemVerilog or Vera. A high-level synthesizable verification environment can easily complement these existing solutions – for targeting FPGAs, emulation and/or prototyping, which are verification uses where RTL, such as Verilog or VHDL, is the only alternative approach today. A modern, high-level synthesizable verification language offers many benefits over RTL, such as faster development and bringup, significantly fewer verification bugs, and better reuse. The following list summarizes the ways that high-level synthesizable verification is already being used today:

- Building FPGA-resident test benches to test DUTs at very high speeds in FPGAs.
- Rapidly developing architectural- and cycle-accurate models that can be run in SystemC or FPGAs. For example, these models could be used to stub in missing system IP components until RTL is available. With a base platform, where a small number of IP blocks are new, one can quickly design and synthesize a model for an IP block so that emulation can be brought up early in the design cycle.
- Developing golden reference designs that can be used in emulation at much higher speeds.
- Quickly instrumenting FPGAs for monitoring, better observability and statistics for analysis and debug.
- Transactorizing existing, 100% accurate RTL to run low-cost FPGA platforms, and optionally with SystemC/C/C++ environments.
- Transactorizing new RTL to keep pace with evolving standards & specifications before models are available. Often, the first available design is RTL-based, especially for new and evolving standards. By quickly building transactors, these RTL models can be tied into software virtual platforms.
- Simplifying the modeling of architectural and cycle-accurate models.
- Repartitioning test bench architectures for co-emulation environments. Without the right architecture, the host to FPGA board interface can be a critical bottleneck, but moving test bench functionality into FPGAs can be daunting if RTL is the means. See inset, **Test Bench Partitioning Can Kill Performance and Productivity**.
- Extending Transaction Level Modeling (TLM) concepts into emulation or prototypes for models, test benches, and transactors.

Test Bench Partitioning Can Kill Performance and Productivity

When software-based simulation or modeling performance becomes an unacceptable bottleneck, FPGA prototyping and emulation offer compelling alternatives with their execution speeds, which can be many orders of magnitude faster than simulation, and the scalability enabled by parallel hardware, so that performance does not degrade with design size. FPGAs can make feasible those activities where simulation would be unacceptably or frustratingly slow. FPGAs offer the scalability and speeds to tackle many of these types of applications, such as:

- SoC exploration, validation, verification and software development
- Multi-core exploration and verification
- Communications protocol and interface interoperability, where connecting to real equipment is an imperative

In addition to speed and scalability, individual FPGAs, with more than 2.5 million ASIC gates per device, are now available with the density of small SoCs. Ganged together, FPGAs can prototype today's largest designs. But, there are several key challenges to overcome in effectively using them: ease of use, cost and test bench development. Our whitepaper, entitled "Emulation: Enabling It On Every Desktop" focuses on addressing the ease of use and cost issues.

Despite such promise, the performance of FPGA platforms and emulation systems can be severely limited by software test benches and co-emulation links. The maximum speedup over simulation depends critically on how much of the system (and which parts) can be synthesized onto hardware. Even assuming that system components accounting for 99% of the simulation time can be moved into FPGAs, Amdahl's law says that the maximum possible speedup cannot exceed 100X – and that's even before looking at latencies and how tightly coupled the simulation and hardware pieces are.

As they enable flexible re-partitioning of the system in order to avoid bottlenecks, synthesizable test benches, models and transactors are required to reach the 1,000X to 10,000X and more performance gains offered by emulation.

Case Study: SystemVerilog VMM vs. BSV for an Ethernet MAC test bench

Overview:

This case study compares two test bench implementations for an Ethernet MAC, one done with SystemVerilog VMM (the source for this test bench is publicly available under VMM examples in VMM Version: 1.1.1, Release Date: August 3, 2009 at http://www.vmmcentral.org) and the other using synthesizable BSV. There are two main goals for this case study: 1) to create a synthesizable verification environment for an Ethernet MAC, so that the entire test bench and Design Under Test (DUT) could be tested rapidly on a low-cost FPGA board; and 2) to compare a synthesizable solution against a reference test bench done with SystemVerilog VMM to understand relative advantanges and disadvantages.

The DUT for this case study is the OpenCores Ethernet 10/100 MAC/PHY, which provides an MII interface to an Ethernet PHY chip and a Wishbone Master/Slave interface on the system side to a Wishbone bus, which is an OpenCores standard.

Although the BSV test bench version was 100% synthesizable, it demonstrated that you could develop a synthesizable test bench at a very high level of abstraction. ***In fact, for comparable test bench functionality, the synthesizable BSV-based test bench required about 40% fewer lines of code than the SystemVerilog VMM reference test bench.***

Highlights of the BSV Test Bench Implementation:

The BSV-based test bench is architected similarly to the SystemVerilog VMM one. As illustrated in Figure 1, the BSV test bench comprises five main components:

- An Ethernet frame source that is used twice, once on the MII side and once on the system bus side
- A block emulating a software interface for transmitting frames on the system bus side. It uses a master Wishbone interface on the system side of the DUT
- A memory model for receiving frames on the system bus side. It sits off a slave Wishbone interface on the system side of the DUT
- A PHY model for sending and receiving frames to/from the MII interface side of the DUT
- And a MAC model connected to this PHY block

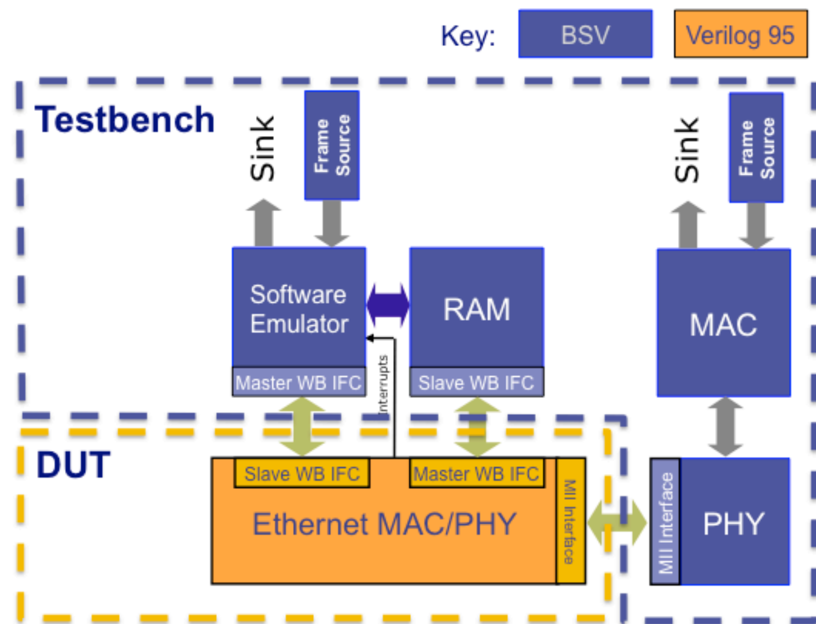

Figure 1 - Test Bench/DUT architecture block diagram

Simplifying Concurrency Management in Test Benches

Because test benches must respond to and generate concurrent events and stimulus, one of the test bench design challenges is managing concurrency. BSV provides a very powerful abstraction, atomic transactions, for managing flow control and resource contention – and, a powerful communication abstraction that builds on this. What do atomic transactions and their associated communications abstraction do for you? They simplify the implementation of test bench code involving shared resource contention – while delivering

synthesizability, more readability and much better reuse. And, they are much more powerful than what SystemVerilog and SystemC provide for expressing and managing concurrency in test benches. Also, they make it easier to express asynchronous, reactive computations, e.g. handing an arriving packet.

With the Ethernet test bench, there are multiple shared resource contentions that need to be managed. For example, the memory block is accessed by both the transmit and receive processes in the DUT and by the software emulator for handling test bench tasks. Another example is that the configuration registers in the DUT are accessed through the Wishbone master bus interfaces by three separate processes in the software emulator. These can be quite complicated to express and get right.

In SystemVerilog and SystemC, concurrent activities are manually managed with traditional threads, which are notorious for being inadequate for complex concurrency and reactivity (e.g., "The Problem with Threads", Professor Ed Lee, UC Berkeley, www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.pdf). The BSV test bench uses atomic transaction *rules* to simplify expression of these concurrent activities. *rules* enable the engineer to think about the circumstances around each individual access of a resource in isolation, one test bench operation at a time, without necessarily having to consider all the other potential accesses of the same resource. This ends up being much easier than with SystemVerilog, and a lot more succinct. Figure 2 shows the detailed BSV code for the slave transactor, which uses *rules*. Figure 3 illustrates the succinctness of this code with a side-by-side comparison for comparable slave transactor behavior in the test bench behavior as expressed with BSV (on the left) and SystemVerilog (on the right) – succinctness in the BSV comes primarily from language features such as *rules* but also somewhat from the modularity delivered through standard libraries.

Figure 2 - Example BSV test bench code for Wishbone slave transactor

BSV – Slave Transactor Code

```
rule monitor_bus (monitoring && buffer.clientIFC.stb.get());

   let tagged_addr = buffer.clientIFC.adr.get();
   let addr = tagged_addr.data;

   if ((self.min_addr.data <= addr) &&
       (addr <= self.max_addr.data))
      begin
         monitoring <= False;
         let wbone_op =
                 createWBoneOp(buffer.clientIFC);
          out_fifo.enq(wbone_op);
      end
endrule

rule respond (!monitoring);
   let wbone_op = in_fifo.first();
   in_fifo.deq();
   monitoring <= True;
   send_response(buffer.clientIFC, wbone_op);
Endrule

rule hold_bus (!monitoring && !in_fifo.notEmpty());
   buffer.clientIFC.ack.drive(False);
   buffer.clientIFC.err.drive(False);
   buffer.clientIFC.rty.drive(False);
endrule
```

Figure 3 - Side-by-side comparisons between slave transactor code in BSV (left) and SystemVerilog (right)

BSV

```
rule monitor_bus (monitoring && buffer.clientIFC.stb.get());

   let tagged_addr = buffer.clientIFC.adr.get();
   let addr = tagged_addr.data;

   if ((self.min_addr.data <= addr) &&
       (addr <= self.max_addr.data))
      begin
         monitoring <= False;
         let wbone_op =
                 createWBoneOp(buffer.clientIFC);
          out_fifo.enq(wbone_op);
      end
endrule

rule respond (!monitoring);
   let wbone_op = in_fifo.first();
   in_fifo.deq();
   monitoring <= True;
   send_response(buffer.clientIFC, wbone_op);
Endrule

rule hold_bus (!monitoring && !in_fifo.notEmpty());
   buffer.clientIFC.ack.drive(False);
   buffer.clientIFC.err.drive(False);
   buffer.clientIFC.rty.drive(False);
endrule
```

SystemVerilog

```
forever begin
   wb_cycle req;

   do begin
     if (this.sigs.cyc !== 1'b1 ||
         this.sigs.stb !== 1'b1) begin
       this.sigs.rdat <= 64'bz;
       this.sigs.rtgd <= 16'bz;
       this.sigs.ack  <=  1'bz;
       this.sigs.rty  <=  1'bz;
       this.sigs.err  <=  1'bz;
     end

     @(this.sigs.cyc or
       this.sigs.stb or
       this.sigs.adr or
       this.sigs.sel or
       this.sigs.we  or
       this.sigs.tga or
       this.sigs.tgc);
   end while (this.sigs.cyc !== 1'b1 ||
          this.sigs.stb !== 1'b1);

   $cast(req, this.req_factory.allocate());
   req.stream_id = this.stream_id;
   req.data_id   = this.data_id++;

   req.addr = this.sigs.adr;

   if (!(this.cfg.min_addr <= req.addr && req.addr <=
     this.cfg.max_addr)) continue;

   req.tga  = this.sigs.tga;
   if (this.sigs.we) begin
     req.kind = wb_cycle::WRITE ;
     req.data = this.sigs.wdat;
     req.tgd  = this.sigs.wtgd;
   end
   else begin
     req.kind = wb_cycle::READ ;
   end
   req.sel  = this.sigs.sel;
   req.tgc  = this.sigs.tgc;

   this.sigs.ack <= 1'b0;
   this.sigs.rty <= 1'b0;
   this.sigs.err <= 1'b0;

   begin
     integer n_wss = 0;
     wb_cycle_resp resp;

     fork
       forever begin
         @ (this.sigs.sck);
         n_wss++;
       end
     join_none

     resp = null;
     `vmm_callback(wb_slave_callbacks,
              pre_cycle(this, req, resp));

     if (resp == null && this.req_chan != null && this.resp_chan !=
      null) begin
       this.req_chan.put(req);
       this.resp_chan.get(resp);
     end
     if (resp != null) begin
       if (resp.req == null) resp.req = req;
       if (resp.cfg == null) resp.cfg = cfg;
       if (!resp.randomize()) begin
         disable fork;
         break;
       end
     end else begin
       this.randomized_resp.req = req;
       this.randomized_resp.cfg = cfg;
       if (!this.randomized_resp.randomize()) begin
         disable fork;
         break;
       end
       $cast(resp, this.randomized_resp.copy());
     end

     if (resp.req != req) begin
       if (resp.req.kind != req.kind ||
           resp.req.addr != req.addr) begin
            break;
       end
     end

     `vmm_callback(wb_slave_callbacks,
              pre_response(this, resp));

     disable fork;

     if (resp.status == wb_cycle_resp::NO_RESP ) break;

     if (n_wss > resp.n_wss) begin
       `vmm_warning(this.log, $psprintf("Number of requested wait
      state in response (%0d) already exhausted due to delay in
      responding (%0d)",
                                 resp.n_wss, n_wss));
     end
     else begin
       repeat (resp.n_wss - n_wss) begin
         @ (this.sigs.sck);
       end
     end

     if (req.kind == wb_cycle::READ ) begin
       this.sigs.rdat <= resp.data;
       this.sigs.rtgd <= resp.tag;
     end

     case (resp.status)
       wb_cycle_resp::ACK : this.sigs.ack <= 1'b1;
       wb_cycle_resp::RTY : this.sigs.rty <= 1'b1;
       wb_cycle_resp::ERR : this.sigs.err <= 1'b1;
     endcase

     if (req.kind == wb_cycle::WRITE ) begin
       bit [63:0] dat;
       bit [15:0] tag;

       @ (this.sigs.sck);

       dat = this.sigs.sck.wdat;
       if (dat !== req.data)
       tag = this.sigs.sck.wtgd;
       if (tag !== req.tgd)
         `vmm_error(this.log, $psprintf("TGD_I not stable during
      write cycle: %h vs %h", this.sigs.sck.wtgd, req.tgd));
     end

     `vmm_callback(wb_slave_callbacks,
              post_cycle(this, resp));

   end
end
```

Creating and Automatically Generating Composable, Parallel, Nested FSMs

BSV includes a powerful sub-language, called StmtFSM, for automatically generating composable, parallel, nested Finite State Machines (FSMs). Since they are built on the same underlying BSV atomic transaction semantics, they leverage the same management of shared resources – so there are none of the surprises inherent in the traditional manual design and coordination of parallel FSMs. And, leveraging the other high-level capabilities of BSV, you can even write your own FSM generators.

StmtFSM specifications can be synthesized automatically to FSMs that can run in RTL or SystemC. StmtFSM is terrific for describing sequential protocols and for populating memories. For example, in the case study, it was used extensively for applications like transmit and receive protocols and initialization sequences.

The following is an example FSM for transmitting frames from the MAC test bench block to the PHY test bench block. As they are based on atomic transactions, individual FSM sequence steps will implicitly flow control without having to complicate the FSM expression with lots of control code. There is an example of this in the code excerpt below. Between the *action…endaction* block (which describes a set of actions that should all happen in parallel) where FSM grabs the next frame with "*let frame = mac_rx_first()*" (commented with "*// %%%*" below), the FSM will automatically wait at this step until there is a frame available. No code needs to be written to support all the checks associated with this, as would be required in SystemVerilog. Similarly, in a call such as "*scoreboard.sent_from_phy_side(frame);*" (commented with "*// ^^^*" below), if there were other concurrent activities updating the scoreboard, the contention would be managed automatically.

```
Stmt tx_driver_seq =
 seq
   while (True)
       seq
         action
           let frame = mac_rx.first();
           ignore <= frame;
           n_attempts <= 0;
           log_backoff <= 1;
           success <= False;
           displayFrame(frame);
         endaction
         while ((n_attempts < self.attempt_limit) && !success)
           seq
                 if (n_attempts > 0)
                   action
                                 let value <- get_backoff;
                                 let max <- get_backoff_max;
                                 backoff <= value;
                   endaction
                 if (n_attempts > 0) delay(backoff * self.slot_time);
                 await(!deferring);
                 action
                   let frame = mac_rx.first();  // %%%
                   phy_tx.enq(frame);
                   success <= True;
                   transmitting <= True;
                   log_backoff <= min(2, (log_backoff + 1));
                 endaction
                 await(!phy_tx.notEmpty); // wait until frame has been transmitted (or a collision has occured)
                 action
                   transmitting <= False;
                   n_attempts <= n_attempts + 1;
                 endaction
           endseq
         action
           let frame = mac_rx.first();
           scoreboard.sent_from_phy_side(frame);  // ^^^
           mac_rx.deq();
         endaction
       endseq
 endseq;

 let tx_driver_fsm <- mkFSM(tx_driver_seq);
```

Random Number Generation

Included in the current toolset for BSV, there are library elements for generating random numbers and a typeclass called Randomizable for generating random application-specific data. In this test bench, a specific instance of the Randomizable typeclass was developed to deliver random Ethernet frames as illustrated with the following code, which is synthesizable. Typeclasses in BSV provide similar power to object-oriented inheritance in C++ and SystemVerilog, and are fully statically typechecked, and fully synthesizable.

```
instance Randomizeable#(FrameData);
        module mkRandomizer (Randomize#(FrameData));

              let max = fromInteger(valueOf(MaxDataLength));
               Randomize#(DataSize) size_gen <- mkConstrainedRandomizer(0, max - 1);
               Randomize#(DataVector) data_gen <- mkGenericRandomizer;

               interface Control cntrl;
                      method Action init();
                          size_gen.cntrl.init();
                         data_gen.cntrl.init();
                      endmethod
               endinterface

               method ActionValue#(FrameData) next ();
                   let size <- size_gen.next();
                   let data <- data_gen.next();
                   let out = FrameData {size: size, data: data};
                   return out;
               endmethod
        endmodule
endinstance
```

Test Bench Comparison: BSV versus SystemVerilog VMM

Only including lines of code for comparable functionality in both test benches (some capabilities were not included in the BSV test bench such as assertions and some capabilities were more limited, such as scoreboarding and the scope of random Frame generation), the BSV-based test bench was implemented in approximately 2600 lines of code whereas the SystemVerilog VMM-based one was about 4400 LOC, making it almost 70% larger. Despite being much more succinct, the BSV-based test bench is 100% synthesizable into Verilog RTL for use in FPGAs, for prototyping and emulation. The results are summarized in Figure 4.

	SystemVerilog VMM	BSV
Lines of Code	~4400	~2600
Synthesizable	✖	✔

Note: Lines of code (LOC) based on comparable functionality. SystemVerilog VMM testbench contained SVA assertions and more sophisticated random frame generation, e.g., which are not included in both LOC counts.

Figure 4 - Test Bench Comparison Summary

Leveraging Reuse to Extend the Test Bench:

In addition to the succinctness of high-level constructs, being able to quickly accommodate changes in specifications and test bench component reuse are important qualities in a verification language as well. As a

test of these additional qualities, an extra exercise was done with the BSV-based test bench. After the basic Ethernet MAC test bench was complete, the BSV test bench components were re-used to develop a test bench for a router/switch design based on multiple Ethernet MACs interconnected through an arbiter/switch.

Because BSV enables rapid changes, promotes reuse and supports a high-degree of parameterization, this effort was completed very quickly and with a high degree of code reuse. The new test bench was implemented in only 15% more lines than the basic one, in less than 3000 lines. Both the BSV MAC and BSV router test benches took less than 9 man-weeks to develop in total, including the time to understand both the DUT design and SystemVerilog test bench implementation.

Conclusion

By providing modern high-level programming language capabilities with features appropriate for verification IP, high-level verification languages such as *e*, SystemVerilog and Vera have replaced Verilog and VHDL as the languages of choice for building test benches for simulation environments. But, these languages were not designed to be both high-level and synthesizable – so, you can't take advantage of their huge advantages over RTL to build models, test benches and transactors for FPGA-based emulation and prototyping.

With the introduction of high-level synthesizable verification, it is now possible to modernize the creation of FPGA-targeted verification IP, bringing the synthesizable domain to the level of productivity and reuse typically experienced now with simulation-based verification. This white paper outlined the features that would be useful in an ideal high-level synthesizable verification environment. Bluespec's BSV supports all of these capabilities – while also allowing all of it to be synthesizable, at any abstraction level and with any design type such as control and datapath. BSV can be used in simulation in addition to emulation – and it's easy to move the simulation/emulation boundary through repartitioning. BSV, due to its expressive power, provides a viable route to developing a single-language verification environment for simulation and emulation. The Ethernet MAC test bench illustrates how synthesizability can be delivered while still leveraging high-level language capabilities and without sacrificing succinctness or reusability compared to state-of-the-art SystemVerilog.

With modern, high-level synthesizable verification, verification IP such as models, test benches and transactors destined for emulation or prototyping platforms no longer have to be done with RTL.

Bluespec's Solutions

Proven in world-class systems and semiconductor companies, Bluespec's solutions enable significantly faster-time-to-solutions for architectural validation, exploration, SW development & IP/system design:

- Bluespec High-Level Modeling Toolkit: a complete toolset and modeling library for developing and running synthesizable high-level models, test benches and transactors. Includes Bluespec's virtual emulation infrastructure for running the models, as well as existing RTL, on third-party FPGA boards and systems.
- Bluespec Virtual Emulation: emulation infrastructure that makes FPGA emulation & prototyping of system-on-chips (SoCs) more affordable and easier-to-use. Includes hardware portable infrastructure for managing and delivering connectivity between simulation-based models and test benches and FPGA-based emulation. Can be used with low-cost FPGA boards, developed in-house or by third-parties.
- BSV: a best-in-class high-level synthesis language, toolset and set of IP libraries that enable powerful parameterization, reusability, and composability for modeling, verification and implementation. Bluespec provides the only general-purpose, high-level synthesis toolset for any use model (models, test benches, production IP) and design type (datapath, control, interconnect).

With Bluespec, models and test benches can be synthesized along with legacy IP to leverage emulation much earlier in the development cycle. Users get better chips to market sooner by developing software in parallel and validating architectures well before freezing a design, e.g. tapeout. Bluespec is the only synthesis tool built on atomic transactions, the only proven technology to manage and simplify large-scale hardware concurrency.

To learn more, please contact Bluespec at sales@bluespec.com.

Bluespec, Inc.
14 Spring Street
Waltham MA 02451
www.bluespec.com
781-250-2200