Verification with Bluespec SystemVerilog™

Abstract

Bluespec SystemVerilog™ (BSV) is an advanced high-level Hardware Description Language that can increase design productivity by 2x or more because of its powerful support for abstraction, static elaboration and static checking. In this document we describe how verification is performed with BSV designs, both standalone and in the context of existing blocks and verification IP that may exist in Verilog, SystemVerilog, VHDL, *e* or SystemC. Topics include: execution options for BSV, why BSV improves Verification, executing BSV along with non-BSV blocks, interfacing non-BSV to BSV in the testbench and in the DUT, using BSV in testbenches, observing execution and debugging BSV blocks, using assertions with BSV blocks.

Abbreviations and terminology:

BSV	Bluespec SystemVerilog
V/SV	Verilog/ SystemVerilog
RTL	Register Transfer Level (used here to refer to Verilog, SystemVerilog or VHDL code)
DUT	Device Under Test
Tb	Testbench
e	The *e* language in the Cadence/Verisity Specman tool set
HDL	Hardware Description Language
OSCI	Open SystemC Initiative (which makes available a free SystemC simulator)
bsc	The Bluespec compiler (also referred to as the Bluespec Synthesis Tool)

References (all part of the standard distribution):

Bluespec SystemVerilog Reference Guide: Language syntax and semantics, library descriptions.

Bluespec SystemVerilog User Guide: Guide to tools (synthesis/compilation, compiler flags, compiler messages, etc.)

Table of Contents

---

1 Background on Bluespec SystemVerilog

Bluespec SystemVerilog™ (BSV) is an advanced hardware description language (HDL) intended as a high-level replacement for legacy HDLs such as Verilog and VHDL, for ASIC and

FPGA design. Although primarily focused on design, BSV's advanced features also improve specification (executable specs, architecture exploration), reuse, and verification, shortening the entire process of design, from spec to verified netlist, by 2x or more. BSV's features include:

- High level, declarative, reactive specification of complex concurrent behavior: BSV *Rules*, in place of RTL/SystemC's "processes and events".
- High level, concurrent-behavior-aware specification of modularity: BSV *Interfaces*, in place of port lists.
- Nested, parameterizable interfaces, allowing easy construction of complex interfaces.
- Strength in specifying resources and complex concurrent access to shared resources.
- Automatic synthesis of the control logic to manage complex concurrency (the most error-prone part of RTL design)
- High level constructs for types, with very flexible type parameterization (polymorphism, overloading, representation flexibility) and strong static type-checking.
- Powerful support for multiple clock domains, with static checking of clock discipline
- Powerful static elaboration, allowing succinct, parameterized expression of repetitive structures, automatic state machine generation, etc.
- Temporal assertions using OVL library or direct SystemVerilog Assertions.

BSV's facilities for types, overloading, parameterization, static elaboration, and abstraction make it comparable to C++ and SystemC in expressive power for specifying structure. Its facilities for complex concurrency with Rules and Rule-based Interface Methods make it much more expressive than both RTL and SystemC for specifying behavior.

---

2 Why BSV improves Verification

One often hears the following statement about high-level HDLs: "The majority of my flow is spent in Verification, not in Design, so even if your new high-level HDL reduced my Design time to zero, it would not significantly impact my overall schedule". The fallacy in this statement is the assumption that verification complexity is independent of the HDL. In this section we outline all the ways in which features of the HDL (in this case BSV) can dramatically reduce verification time.

2.1 Rule and Interface semantics

A central problem in today's designs is the correct management of complex concurrency. Errors of this kind manifest themselves as race conditions, glitches and inconsistent states. These kinds of errors are the hardest to find and to diagnose. BSV Rules have the semantics of *atomic transactions* and thus, by definition, completely eliminate many such errors.

In RTL, interfaces are specified as port lists, but the *protocol* on these interfaces are only specified separately, e.g., in timing diagrams or in accompanying documentation. It is very common for these interface protocol specs to be informal, incomplete, or plain wrong. In BSV,

on the other hand, interfaces are specified with Interface Methods, and the logic for the interaction protocol is generated by the compiler. Thus, a whole class of interface errors never occur in BSV programs (e.g., the ENABLE signal was driven at the wrong clock relative to valid data).

As a more complex example, imagine a module which either accepts a request or yields a response on any cycle. Suppose we replace this with an improved module that can accept the next request simultaneously with yielding the previous response. In BSV, this difference in behavior is formally part of Rule/Method semantics. The compiler is aware of and tracks the scheduling differences, and generates the right client logic in each case.

In summary, all such hardware to manage complex concurrency is *control logic* and this logic is automatically generated by the BSV compiler, eliminating a whole class of RTL errors.

2.2 Abstract types, type-checking, and representation independence

BSV has very strong type checking to ensure that one does not accidentally misuse a type. For example, if there are two state machines, defining each state as a separate type, and using BSV's strongly-typed registers to hold states ensures that one does not accidentally store a state from the first FSM into the state register for the second FSM.

The ability to define structs and unions allows one to express ideas that are closer to the specification, and eliminates common errors, e.g., by using field selection instead of extracting a slice out of a bit vector. If a struct definition changes (e.g., adding a field, changing the order of fields), then field selection is robust to such changes, whereas slice extraction is not.

BSV has a powerful mechanism that separates out the logical view of a data type from its bit representation. For example, bit representations of processor instructions are typically very non-orthogonal, i.e., the contents of one field may determine the layout of other fields. In RTL code, managing these representations issues pervades the code, and changes are hard to track; all this is very error-prone. In BSV, one can define types with simple orthogonal representations, and separately define how the non-orthogonal representation needed for wires or storage elements that carry such types.

2.3 Clock abstract types and statically enforced clock discipline

In BSV, clocks are abstract types, and can only be manipulated with trusted primitives—correctness is guaranteed by type-checking. Further, static checking ensures that it is impossible to cross a clock domain boundary without a synchronizer. BSV clocks are also *gated* clocks, for power management, and static checking ensures that clocks that differ merely in gating conditions can be used safely together. Static checking ensures that it is impossible to send values to, or read garbage values from modules that are currently gated off.

All these facilities, together, eliminate many clocking errors.

2.4 Parameterization and Powerful Static Elaboration

BSV has extremely powerful parameterization and static elaboration. Essentially all BSV constructs (modules, interfaces, rules, functions) can be parameters to other constructs and to generation constructs, and the generation facilities are Turing-complete.

In RTL, lack of such facilities often results in a lot of code replication and cut-and-paste coding, which is very error-prone.

Further, parameterization can also improve verification. For some structures, it is adequate to verify a small instance of the design (which can be done quickly), together with a formal or informal inductive proof that the design will then work at any size.

2.5 Robustness to change

All of the verification issues discussed above become magnified when the design must be changed. Designs change because

- specifications change late in the game (an unavoidable fact of life),
- because microarchitecture must change to fix bugs,
- because microarchitecture must change to meet performance goals,
- because microarchitecture must change to reach timing closure,
- and because we wish to *reuse* designs, or produce *derivative* designs in the future

In each of these cases, if the HDL is brittle and fragile, the smallest change reopens a huge re-verification effort. However, BSV's high-level rule and interface semantics, automatic regeneration of control logic, type-checking, clock-checking, re-instantiation with different parameters and generation, all contribute to the ability to make significant changes without introducing new bugs.

2.6 Summary

(Some of the above points are also explored in a little more detail in Section 6.)

Of course, raising the level of abstraction will never completely eliminate bugs in a design—fundamental complexities in behavioral specification remain—and therefore verification is always necessary. What is achieved by raising the level of abstraction is the elimination of a whole class of bugs that were purely an artifact of the original low-level coding of the design. BSV's abstraction mechanisms are the most powerful and semantically sound of any synthesizable HDL (more even than many unsynthesizable modelling languages), and dramatically improve the ability to produce correct-by-construction designs, thereby significantly improving verification time.

3 Execution options for BSV DUTs and Testbenches

We first discuss the execution options for DUTs and Testbenches that are written entirely in BSV (we will address mixed execution—BSV and non-BSV—in the next section). Figure 1 shows the standard BSV tools and tool flow.

A BSV source file can be executed in one of two ways:

- Compile using the Bluespec Synthesis tool (*bsc*) into Verilog, which can then be run on any RTL simulator
- Compile and execute using *Bluesim*, Bluespec's simulator

The first method (compiling to Verilog) is useful for the following reasons:

- Runs in your existing simulation environment
- Automatically runs concurrently with anything else that runs in your simulation environment (Verilog, VHDL, SystemVerilog, SystemC, *e*)
- The generated Verilog is the same Verilog that you will take through 3rd party synthesis and physical design tools ultimately to tapeout, i.e., you will be verifying the Verilog that becomes hardware (thus, even if you use Bluesim for debugging and early verification, you will also want to perform final checks and regressions using Verilog simulation)

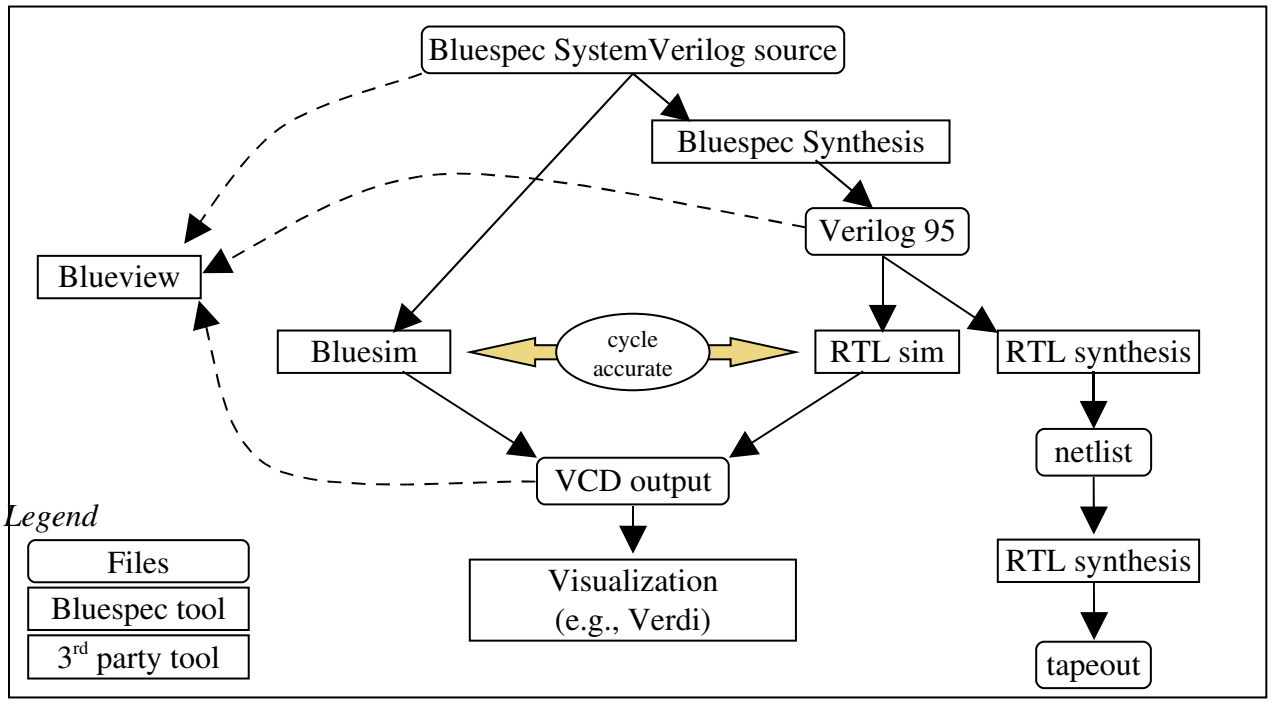

Figure 1 Bluespec tools and tool flow

The second method (Bluesim) is useful for the following reasons:

- (Current)
  - Can handle single-clock-domain BSV designs
  - 2x-3x improvement in simulation speed

- (Future, expected)
    - Even larger speed improvements
    - Better source-level debugging
    - Will handle multiple clock domains
    - Cosimulation with RTL, SystemC

---

4 Execution options for mixed DUTs and Testbenches (BSV and non-BSV)

Currently, the principal way to execute a mixed design (testbench or DUT) is to follow the Verilog route, i.e., use the Bluespec Synthesis tool to compile the BSV source to Verilog, and run it in an RTL simulator, where it can concurrently execute with anything else supported by your RTL simulator, e.g., Verilog, VHDL, SystemVerilog, *e*, and SystemC. If you are using the OSCI SystemC simulator, you would co-execute it with your RTL simulator in the standard way by which you co-execute SystemC with Verilog or VHDL.

In the future (expected in 2006), you will be able to run your BSV design on Bluesim, and co-execute (co-simulate) with

- Verilog/ SystemVerilog/ VHDL/ *e* running on your standard RTL simulator
- SystemC running on your standard SystemC simulator (OSCI, or 3rd party)

---

5 Interfacing non-BSV contexts to BSV modules (Tb-DUT, in Tb, or in DUT)

When designers first start using BSV, they typically design a few blocks using BSV, which are to be inserted into a larger design that is mostly written in traditional RTL (new or existing RTL IP). Also, when testing a BSV module, one may plug it into a new or existing Testbench written using RTL, *e*, or SystemC. In all these situations, the BSV module is plugged into a non-BSV context.

This kind of interfacing is very easy to set up, because the interfaces of BSV modules have a direct and simple mapping into ordinary Verilog port lists. For example, consider the following BSV interface definition:

```
typedef Bit#(24) Addr;
typedef Bit#(32) Data;

interface IfcDUT;
   method Action request  (Addr a);
   method Data   response ();
endinterface
```

When a DUT providing this interface is synthesized by the Bluespec Synthesis tool, it produces a Verilog module with a port list that looks like this:

```
module mkDut(CLK,
             RST_N,
             request_a,
             EN_request,
             RDY_request,
             response,
             RDY_response);
  input  CLK;
  input  RST_N;

  // action method request
  input  [23 : 0] request_a;
  input  EN_request;
  output RDY_request;

  // value method response
  output [31 : 0] response;
  output RDY_response;
  ...
endmodule
```

Method arguments (such as 'a') become input ports ('request_a'). Method results become output ports ('response'). Action methods ('request') have an input "enable" wire ('EN_request'). All methods have "ready" output wires ('RDY_request', 'RDY_response'). There is a simple protocol embodied in these signals. No method may be used until its RDY wire is true. At that time, method arguments can be asserted from outside, Action methods can be used by asserting their ENA wires from outside, and value method results are valid.

The full details of how BSV interfaces are mapped to wires are described in the Bluespec documentation, but the above example illustrates how easy it is interface an external context to the Verilog generated for a BSV module. Whether the context is Verilog, SystemVerilog, VHDL, *e* or SystemC, *interfacing to a BSV module is just like interfacing to an RTL module*.

In the future, you will be able to run the BSV module in Bluesim and interface to a Verilog, SystemVerilog, *e* or SystemC simulation, but the basic principle will be the same, i.e., it will be just like interfacing to an RTL module.

---

6 Using (and reusing) BSV in Testbenches

[This section is relevant for verification engineers who are interested in improving productivity in testbench creation by exploiting BSV features, and can otherwise be skipped.]

BSV has a number of features that facilitate quick and correct testbench creation. These are described in the following subsections.

6.1 Expressing complex concurrency

Testbenches today must express complex concurrency. Examples:

- *Simultaneously* generate packets into two DUT input ports, but only if both ports are not currently flow-controlled
- Generate pipelined requests into a DUT input port, consume pipelined responses from a DUT output port
- Handle out-of-order responses from the DUT
- React to an "interrupt" output from the DUT
- Model the concurrency of a DUT's environment. E.g., if the DUT is a processor, Tb must model the concurrency of parallel, out-or-order memory responses.

In summary, testbenches today need the same power to express complex concurrency that DUTs need.

This is precisely BSV's strength. BSV's *Rules* and *Rule-based Interface Methods* provide powerful facilities for expressing these kinds of behaviors succinctly, and correctly.

6.2 Transactional interfaces and interface abstractions

All BSV modules have transactional interfaces, i.e., even when designing high-performance hardware, BSV interfaces are described in terms of *interface methods*, each of which represents a transaction between the environment and the module.

Further, because of nested interfaces and type parameterization (polymorphism) it is possible to abstract interfaces to a very high level, while still remaining perfectly synthesizable. For example, the BSV library provides the following interfaces (which are written in BSV, i.e., they are not built-in):

```
interface Put #(type t1);
   method Action put  (t1 x);
endinterface

interface Get #(type t2);
   method ActionValue #(t2) get ();
endinterface
```

The Put#() interface contains a single method, put(), and represents a transaction of *putting* an item x of type t1 into a module. The Get#() interface contains a single method, get(), and represents a transaction of *getting* an item of type t2 from a module. In a sense, Get#() and Put#() are *duals* of each other.

From these interfaces, it is easy to define interfaces at a higher level abstraction that are also duals of each other:

```
interface Client #(type req_t, type resp_t);
   interface Get#(req_t)  request;
   interface Put#(resp_t) response;
endinterface
```

```
interface Server #(type req_t, type resp_t);
   method Put #(req_t)  request;
   method Get #(resp_t) response;
endinterface
```

The Client#() interface now contains a get() method yielding requests, and a put() method accepting responses. The Server#() interface contains a put() method accepting requests, and a get() method yielding responses.

Now consider the interface of a DMA block on a bus:

```
interface DMAIfc #(type bus_req_t, type bus_resp_t);
   interface Server#(bus_req_t, bus_resp_t) config;
   interface Client#(bus_req_t, bus_resp_t) read_stream;
   interface Client#(bus_req_t, bus_resp_t) write_stream
endinterface
```

A DMA block is both a Server and a Client. When it is being configured by the CPU, it acts as a Server, i.e., it receives requests from the CPU and sends responses to the CPU. Once it has been configured, the DMA engine acts as two Clients, i.e., a read_stream that sends bus read requests to a source and receives data responses, and a write_stream that sends bus write requests to a sink and receives responses.

These examples illustrate how easy it is to systematically build up complex interfaces in BSV. And all of these interfaces are completely synthesizable.

6.3 Connection abstractions

In BSV, it is possible to write generic *connection abstractions* that capture all the logic and control necessary to interface certain "dual" interface types. In particular, there is an *overloaded module* mkConnection that can be instantiated on two interfaces of type ifc1_t and ifc2_t, expressing the logic necessary to connect them. For example:

```
module mkTop (…);
   Get#(int) ifc_g();                    // Line 1
   mkModuleA  modA (ifcA);               // Line 2
   Put#(int) ifc_p();                    // Line 3
   mkModuleB  modB (ifcB);               // Line 4
   mkConnection (ifcA, ifcB);            // Line 5
endmodule
```

In Lines 1 and 2, we instantiate a module modA, which has a Get#() interface from which it yields values of type int. In Lines 3 and 4, we instantiate a module modB, which has a Put#() interface accepting values of type int. In Line 5, we use mkConnection() to instantiate a module to connect these two interfaces together, i.e., it completely encapsulates the transfer of integer values from modA to modB.

Similarly, once can apply mkConnection to two interfaces of type Client#(t1,t2) and Server#(t1,t2). The module it instantiates will recursively apply mkConnection to the Get#(t1) interface of the client and the Put#(t1) interface of the server, and it will recursively apply mkConnection to the Put#(t2) interface of the client and the Get#(t2) interface of the server.

mkConnection is overloaded in the sense that the particular connection module it instantiates depends on the *types* of the two interfaces to which it is applied. The BSV library pre-defines some overloadings of mkConnection (such as for Get/Put, Client/Server), but the overloading is systematically user-extensible, i.e., the user can extend mkConnection to *any* desired pairs of interface types.

Thus, complex configurations can be expressed in just a few, succinct lines of code (fully synthesizable!). Example:

```
module mkSoC (…);
   Client#(bus_req_t, bus_resp_t) cpu … instantiate …
   Server#(bus_req_t, bus_resp_t) mem … instantiate …
   DMAIfc#(bus_req_t, bus_resp_t) dma … instantiate …
   BusIfc#(3, 2, bus_req_t, bus_resp_t) bus … instantiate …
   mkConnection (cpu, bus.initiators[0]);
   mkConnection (dma.read_stream, bus.initiators[1]);
   mkConnection (dma.write_stream, bus.initiators[2]);
   mkConnection (mem, bus.targets[0]);
   mkConnection (dma.config, bus.targets[1]);
endmodule
```

6.4 Connection refinement across abstraction levels

Because BSV module interfaces are transactional, whether at a high level or at a detailed signalling level, it naturally supports refinement of interfaces across abstraction levels. For example, we might start with two modules that communicate Ethernet packets.

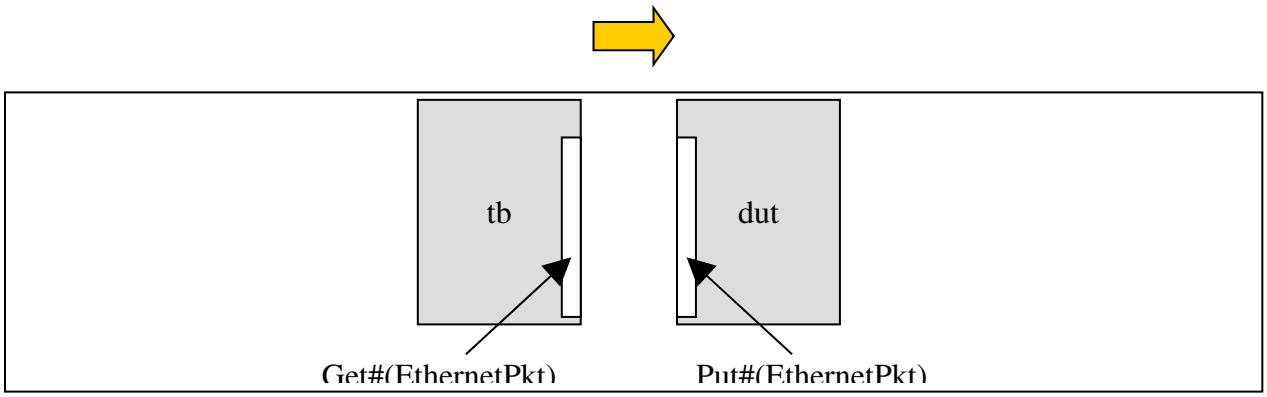

Figure 2 Testbench and DUT with abstract interfaces

i.e., the code might look like this:

```
module mkTop (…);
   Get#(EthernetPkt) tb … instantiate …
```

```
   Put#(EthernetPkt) dut … instantiate …
   mkConnection (tb, dut);
endmodule
```

Later, this can be refined to communicate at a bus signalling level:

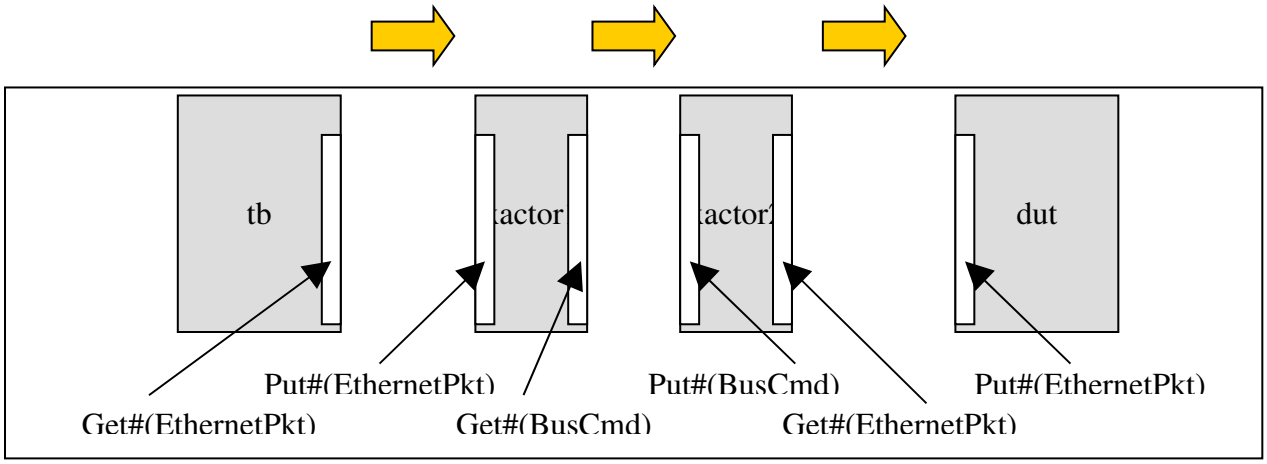

Figure 3 Testbench and DUT with transactors having bus-signalling interfaces

The code for this may look like this:

```
module mkTop (…);
   Get#(EthernetPkt)             tb … instantiate …
   Put#(EthernetPkt)             dut … instantiate …
   Server#(EthernetPkt, BusCmd)  xactor1 … instantiate …
   Server#(BusCmd, EthernetPkt)  xactor2 … instantiate …
   mkConnection (tb, xactor1.request);
   mkConnection (xactor1.response, xactor2.request);
   mkConnection (xactor2.response, dut);
endmodule
```

The tb and dut remain the same; we have simply inserted transactors transparently to lower the level of abstraction of communication to the actual hardware bus signals (such transactors are also sometimes called Bus Functional Models, or BFMs). When written in BSV, the entire structure forms an automatically flow-controlled pipeline from testbench through transactor 1, transactor 2 and into the DUT.

6.5 Reuse due to Rule-based Interface Behaviors

BSV interfaces are more than RTL's port lists. Since BSV interface methods are fragments of Rules, they follow Rule semantics. When the Bluespec Synthesis tool compiles a BSV DUT, it knows about *inter-transaction scheduling constraints.*

For example, suppose there are two interface methods m1 and m2, and their internal logics access a shared resource, then there is a constraint that the environment should not simultaneously enable m1 and m2. In RTL, these constraints may at best be informally documented and easily missed by the verification engineer, so that the testbench may not

correctly follow the interface protocol. In BSV, these constraints are formally analyzed by the Bluespec Synthesis tool and recorded in the compiled code. When the testbench is compiled, these constraints are known, and the control logic necessary to respect the protocol is automatically synthesized.

Thus, unlike RTL, interfaces between BSV testbenches and DUTs are robust, not fragile. Correct interconnection includes correct respect of interface protocols, not merely correct connection of port lists. Interfaces are more reusable because the protocol control logic is resynthesized when interface protocols change, even though the port list may not change at all.

6.6 Reuse due to Parameterization

BSV has very powerful parameterization. The above examples show type parameterization (polymorphism). In addition, in BSV functions and modules can be parameterized by other functions and modules, by interfaces, by Rules, and so on. This makes BSV testbenches and DUTs highly reusable.

6.7 Complex data types

Whereas in RTL one works primarily with bits, BSV has a powerful type definition mechanism that allows user-defined symbolic enumerations, structures (records), tagged unions (type-safe unions), and arrays. All of these can be polymorphic (type parameterized). Strong type checking reduces the possibility of bugs wherein a value of type t1 (e.g., an IP address) is mistakenly used as a value of type t2 (e.g., a Program Counter value), just because they have the same bit representation (e.g., 32 bits).

For static elaboration, the BSV programmer can also use lists, trees and other recursive data structures.

6.8 Data representation flexibility

Another dimension of abstraction is the separation of the logical view of a data type from its bit representation. For example, the 'int' data type can be represented in little-endian or big-endian format. A processor instruction can be logically viewed as a structure containing fields such as opcode, source registers and destination register, but the actual packing of these fields into a 32-bit instruction value may be non-trivial.

BSV addresses this in a systematic way. Data types are always viewed according to their logical structure. Separately, the programmer can define pack() and unpack() functions that convert from the logical view to the bit-representation view; the conversions can be arbitrary. The Bluespec compiler automatically uses these functions wherever necessary, so that the programmer's code need not be cluttered with representation artifacts.

6.9 Random number generation

The BSV library contains an arbitrary-width random number generator, making it easy to generate random test vectors.

6.10 Holding "expected results" and scoreboarding

The BSV library contains a number of "Register Files" and other models which can be used by the testbench as memories to hold temporary data.

The BSV library contains a family of FIFOs that are fully paramterized according to depth and the data type of the content items. These can be used in the testbench to hold "pending responses", i.e., after generating a request into the DUT, the expected response can be placed into the FIFO to be checked against the DUT's response which may appear after some delay.

For DUTs that generate responses out-of-order with respect to the requests, the BSV library contains a "Completion Buffer" abstraction that

- tags each request with a unique, ordered tag
- accepts responses with tags, in any order, and holds them
- yields responses in tag order (which can then be checked against the "expected responses" FIFO).

6.11 State machine generation

Testbenches frequently must carefully orchestrate a sequence of actions. Transactors typically contain state machines to generate or accept a sequence of low-level signals for each logical transaction.

BSV contains a powerful notation and construction mechanism for expressing state machines. Importantly, the generated state machines have standard BSV Rule semantics. For example:

```
Stmt  test_seq =
   seq
      for (i <= 0; i < NI; i <= i + 1)
         for (j <= 0; j < NJ; j <= j + 1) begin
            let pkt <- gen_packet ();
            send_packet (i, j, pkt);
         end
      par
         send_packet (0, 1, pkt0);
         send_packet (1, 1, pkt1);
      endpar
   endseq

mkAutoFSM (test_seq);
```

The first statement specifies a state machine, and the second statement instantiates the FSM (generating all the logic necessary to implement it). The seq/endseq block contains two components that run sequentially—the for-loops followed by the par/endpar component. The for-loops just express a sequence of actions. Here, the loops execute a sequence of send_packet() calls, sending a packet from each input i to each output j of a DUT.

Because the FSM follows rule semantics, the FSM will *automatically stall*, for example, if the DUT is unable to accept a packet when the testbench is ready to send_packet().

The par/endpar section sends two packets precisely in parallel, into inputs 0 and 1, both destined for output 1. Again, if either input port is unable to accept a packet, neither packet gets sent, because of Rule semantics.

In summary, BSV provides a powerful way to express complex FSMs, and these FSMs gain all the benefits of Rule semantics.

6.12 Clock discipline

BSV has very powerful language support for multiple clock domains and gated clocks, with rigorous static checking of clock disciplines, synchronizer presence, etc. This facilitates

- correct expression of clock generation in the Tb for the DUT
- correct connection of the Tb to the DUT

6.13 Synthesizable testbenches

BSV has no "synthesizable subset"—all of BSV is synthesizable. Thus, it is possible to synthesize BSV testbenches into RTL, e.g., for fast execution on an FPGA-based test platform.

---

7 Observing Execution and Debugging BSV Blocks

When verifying a block, in addition to providing stimulus and collecting results (which have been discussed in previous sections), one must also observe internal signals. This is particularly true when debugging, i.e., when diagnosing the cause of errors reported by the testbench.

7.1 Structure of Verilog generated by Bluespec compiler

When observing and debugging Bluespec-generated code in a Verilog simulator, it is useful to understand the structure of the Verilog code. With this structure in mind, it becomes easy to correlate the Verilog code and its waveforms with the BSV source.

7.1.1 State Elements, Module Hierarchy, and Interface Port Lists

All the state elements in the generated Verilog are *exactly* as specified in the BSV source. In BSV compilation there is no "inference" of state elements—what you specify is what you get.

Modules and module instantiation are specified in BSV source in the same way as in Verilog. Every module in the generated Verilog corresponds exactly to some module in the BSV source. However, the converse is not true, i.e., not all BSV source modules become Verilog modules. The BSV programmer has the option of placing the `(* synthesize *)` attribute on a BSV module; each such module becomes a Verilog module. All other modules are *inlined* by the BSV compiler and are not identifiable as separate modules in the Verilog.

As described in Section 5 ("Interfacing non-BSV contexts to BSV modules (Tb-DUT, in Tb, or in DUT)"), there is a very straightforward mapping between the BSV interface to a module and the corresponding port list in the Verilog module.

The BSV compiler is designed to retain names of state elements and modules into the generated Verilog, as far as possible. In general the exact name cannot be retained in the presence of multiple instantiation of modules/state elements, and in the presence of "generated" structures. Even though multiple instances need unique names, the final names will contain the original source name.

7.1.2 Internal Combinational Logic, CAN_FIRE and WILL_FIRE signals

Beyond state elements, the module hierarchy and port signals, what remains is the internal combinational logic in each module. In order to understand this code in the Verilog, it is important to understand the structure of the logic necessary to import BSV Rules. Let us use the following example. The rules represent Euclid's algorithm for computing the GCD (Greatest Common Divisor) of two numbers. If the registers x and y are initialized with the two input numbers, then the rules execute until y contains 0, at which point x contains the GCD of the original two numbers.

```
rule decr ((x <= y) && (y != 0));
   y <= y - x;
endrule

rule swap ((x > y) && (y != 0));
   x <= y;
   y <= x;
endrule
```

First, let us look at the structure of the Verilog to implement the *decr* rule in isolation (Figure 4).

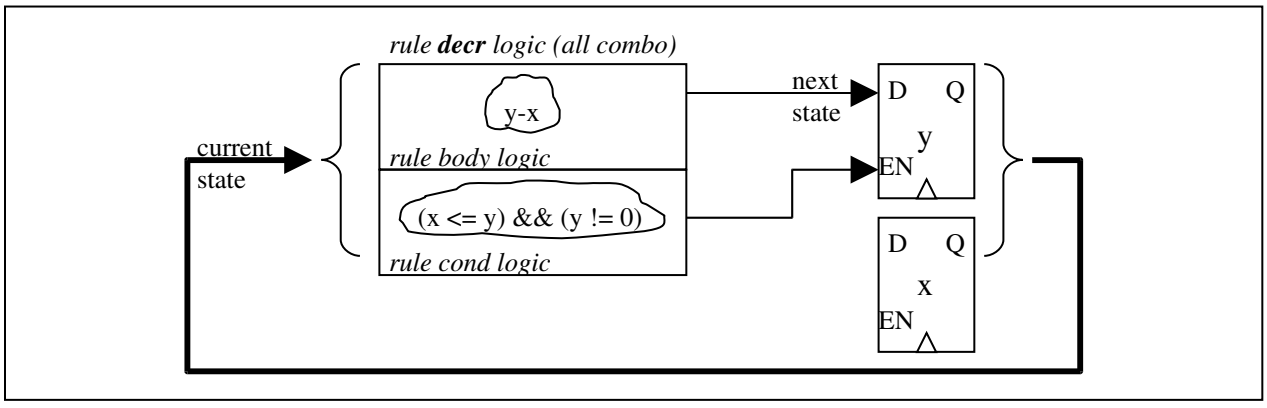

Figure 4 Logic to implement a single rule in isolation

The rule condition is always a pure combinational circuit computing a single boolean value, the condition under which the rule can fire. The rule body is always just a combinational circuit that computes the next state value(s) that would ensue if the rule fires. Next, Figure 5 shows the

structure of the generated Verilog when we implement both rules, which both read and write some common state (in this case, both rules update register *y*).

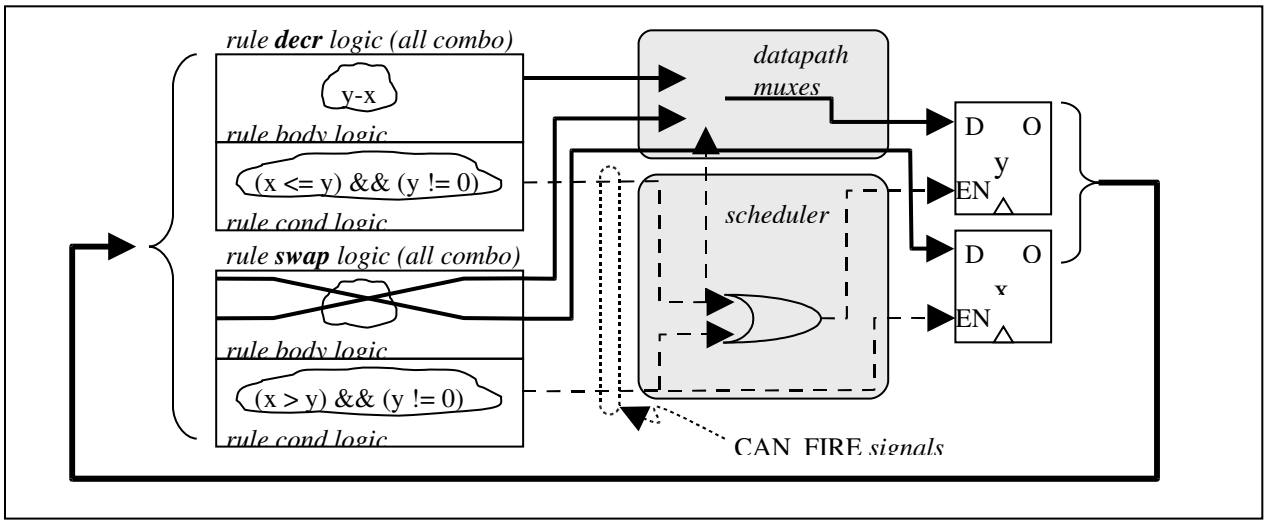

Figure 5 Logic to implement both rules (scheduling and datapath muxing)

We can see that, in addition to the state elements and the rule condition and body logic, the BSV compiler inserts *scheduler* logic and *datapath muxes* (all completely combinational). The datapath muxes merge next-state outputs of rules that update the same state element. The scheduler logic takes the outputs of the rule conditions (also known as CAN_FIRE signals) and controls the datapath muxes and state EN signals so as to effect a subset of rule firings.

(Please do not be concerned that logic for *y != 0* seems replicated. The figure is illustrative only. In fact, the BSV compiler performs very powerful common subexpression optimization that will ensure that logic will be shared.)

In this simple example, the rule conditions were mutually exclusive (rule *decr* fires only if x <=y and rule *swap* fires only if x > y), but in general in BSV rules need not have mutually exclusive conditions, even if they are updating the same state. In this case, the scheduler logic also includes *arbitration* logic that stalls some rules if firing them would lead to an inconsistent state. Since a rule may be stalled even if its CAN_FIRE signal is true, we refer to that scheduled version of the signal as the WILL_FIRE signal for the rule.

Finally, Figure 6 shows the general case, for an arbitrary BSV program.

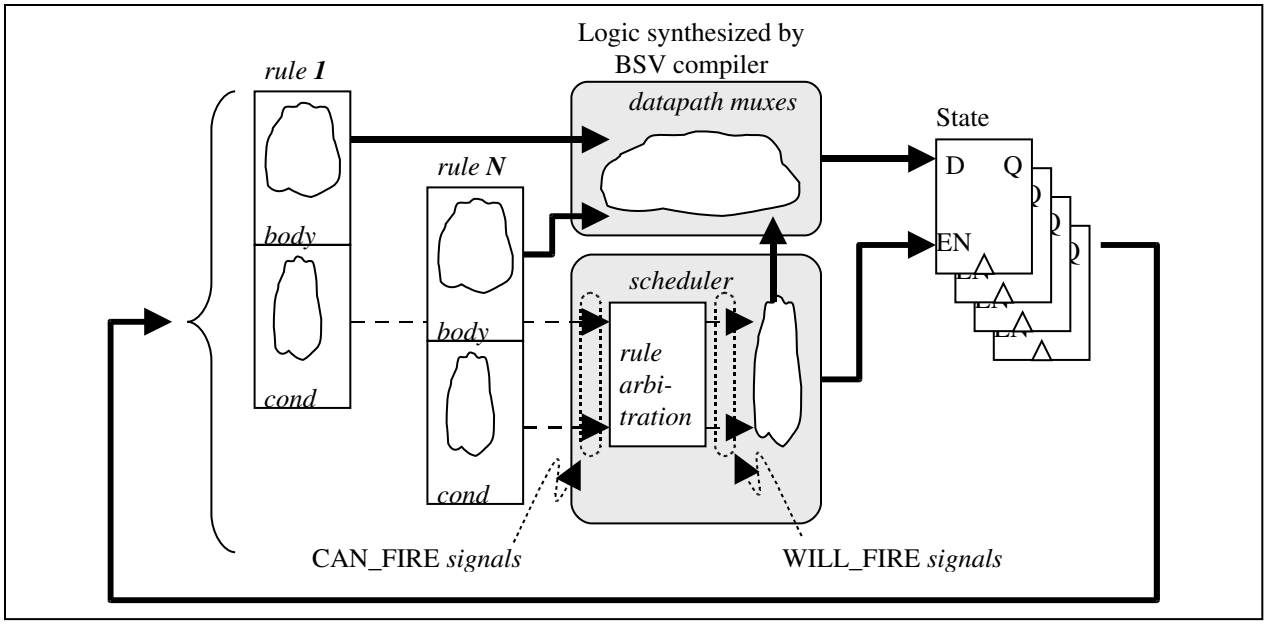

Figure 6 The general structure of all Verilog generated by the BSV compiler

In summary, what this means to you, when you examine the generated Verilog or view waveforms, is:

- The module hierarchy is directly recognizable from the BSV source.
- Module interface ports are directly recognizable from the BSV source.
- All state elements are directly recognizable from the BSV source.
- Inside each module, the *cond* and *body* logics are directly derived from the BSV rule source (taking into account boolean and logic optimization, of course).
- The only "freshly" synthesized logics are the scheduler and the datapath muxes. Even here, the CAN_FIRE and WILL_FIRE signals are named according to the rule names in the source, and are instantly recognizable.

Thus, it is quite easy to do *source level debugging* with the generated Verilog.

7.2 Observing runtime values

The BSV programmer can observe runtime values using the same methods as the Verilog programmer. First, you can insert $display and $write statements in BSV programs, as usual. Second, you can dump VCDs and observe waveforms in the usual way. You can use the usual Verilog $dumpvars, $dumpon and $dumpoff system tasks in BSV code.

7.2.1 Ensuring that certain signals remain visible despite optimization

The Bluespec compiler performs aggressive optimizations to improve code quality. Due to such optimizations, some source code signals may disappear entirely from the generated Verilog. This, of course, can complicate debugging.

The compiler flag `-keep-fires` instructs the compiler not to optimize away any of the rules' CAN_FIRE and WILL_FIRE signals. As discussed later, observing these signals is useful to diagnose which rules are not firing, and why. The User Guide describes this flag in more detail.

The "Probe" facility allows you to produce a named signal in the output Verilog for any value of your choice. For example:

```
module …
   Probe #(int) foo <- mkProbe;
   …
   rule r (… rule condition …)
      …
      foo <= … some expression of type int …;
      …
   endrule
   …
endmodule
```

This ensures that there is a signal in the generated Verilog whose name contains the source name `foo.` Thus, a VCD waveform can be produced for any arbitrary intermediate value.

7.3 Debugging BSV Execution

Observing BSV execution is just the first step. If we observe a problem, the next step is to debug it, i.e., to diagnose why we observed the problem, and then to fix it.

There are essentially two kinds of problems that we might observe in a BSV execution:

- A wrong value is computed somewhere
- A value is not being updated when we expect it to be updated.

These may be related—a wrong value at time t_1 may result in some other value not being updated at time t_2, and vice versa. I.e., a wrong value may put the system in a state that prevents some future update which, in turn, can result in other wrong values. In both cases, we want to trace backwards from the observed problem to the cause of the problem. We want to trace backwards both in time as well as in space (i.e., the logic cone affecting the observed problem).

7.3.1 Debugging wrong values: Standard tracing of cones of logic

Referring to Figure 6 we can see that all the actual functional logic of the design is explicit in the source code, modulo compiler optimizations such as boolean optimizations and common-subexpression sharing. In other words, all state elements are explicit, and all functional combinational logic is explicit in the conditions and bodies of Rules and Interface Methods.

Thus, for example, if we wrote `+` instead of `-`, or forgot to increment a value, the method of locating this bug is standard: we use standard backward cone-of-logic tracing from a problem value to the operator that produced it, to the inputs of that operator, and so on, until we find the

source of the problem. Such functional errors can be directly related to a bug (perhaps a typo) in the BSV source code.

If the value being written into a register is wrong or, more generally, if the value being sent as an argument into an interface method is wrong, then one identifies all the possible rules in the BSV source that generate that value. This set of possible writers is always statically known, i.e., a simple text search in the source can identify them, or a backward trace in a waveform viewer will quickly identify them. For example, if there are three rules that write to register x, then each rule will contain next-state logic that compute the new value of x, and then these three values are muxed into the D input of x. The same principle holds for multiple rules that use the same interface method on a shared sub-module. Thus, modulo the muxing, it is possible to trace back to the next level, in order to find the source of the bad value.

7.3.2 Debugging rule firings

The other kind of bug in a BSV program is that a value does not get updated when expected, because a Rule or Interface Method did not execute when expected. This may manifest itself in various ways, for example

- we may observe that a register or FIFO has a wrong value, because the Rule/Method that enqueues into the FIFO did not fire;
- we may observe that a FIFO remains empty, because the Rule/Method that enqueues into the FIFO never fires
- we may observe that the whole system becomes quiescent (no rules fire)

but all of these boil down to: a Rule did not fire when expected.

Referring to Figure 6 again, a Rule will not fire if either its CAN_FIRE signal is false, or its WILL_FIRE signal is false. Every Rule's CAN_FIRE and WILL_FIRE signals are generally clearly visible in the generated Verilog. The compiler flag `–keep-fires` will ensure that such signals are not optimized away.

The first step is to observe whether its CAN_FIRE signal is unexpectedly false. This case boils down to standard cone-of-logic tracing, i.e., the CAN_FIRE signal is simply a conjunction of the Rule's explicit condition, together with the conditions of all Interface Methods used by the Rule, all Interface Methods used by those Methods, etc. All of these conditions are manifest in the source code. Standard cone-of-logic tracing can identify why the condition is unexpectedly false: either there is a wrong operator in the source code, or the value in some state element is wrong, etc.

If the CAN_FIRE signal is true as expected, but the WILL_FIRE signal is unexpectedly false, then the Bluespec-generated scheduler logic (please refer to Figure 6 again) is preventing the rule from firing, because the rule contends with some other rule for some shared resource, and the scheduler has given priority to the other rule. In this case, rather than attempt to trace through the scheduler logic, it is much easier to examine the schedule produced by the compiler. When compiling the source, the flag `–show-schedule` will cause the compiler to write out a

scheduling report. This report describes all the contentions between rules on shared resources, and describes the chosen prioritization. This information will tell you why this rule did not fire even though its CAN_FIRE signal was true.

If you have identified a scheduling reason for a rule not firing, you will have identified a genuine resource conflict in your BSV program. Resolving a resource conflict is done in the standard ways, e.g.,

- Rescheduling by changing priorities. E.g., use the `descending_urgency` attribute in the source to change the priority of the rules contending for the shared resource.
- Rescheduling by retiming. E.g., fix the state machine containing the rule(s) so that they access the resource at different times or under different conditions
- Replication. E.g., replicate a register, so that each rule accesses a different one, or increase the number of ports to a module, so that each rule accesses a different port.
- Microarchitectural changes. E.g., fix a counter to allow simultaneous increment/decrement, or fix a FIFO to allow simultaneous enq/deq.

You can also place certain scheduling assertions in the source code indicating your expectations about scheduling of certain rules and methods (please see the Reference Guide for details)

- Interface method attributes `always_ready and always_enabled` : assert that a method's condition will be true on every clock, and that an Action or ActionValue method will be enabled on every clock.
- Rule attribute `fire_when_enabled and no_implicit_conditions` : assert that a rule's WILL_FIRE is equal to its CAN_FIRE (i.e., cannot be stalled by another rule), and that a rule's CAN_FIRE is equal to the rule's explicit condition (i.e., all of the methods it uses will always be ready).
- Rule attribute `descending_urgency` : informs the compiler how to prioritize rules when they contend for a shared resource.

All these scheduling assertions are checked *statically* by the compiler, which will complain if it is impossible to schedule the rules/methods as asserted.

Another common pitfall is blindly to read a value from an RWire (using its wget method) without testing its Valid bit, i.e., making the assumption that the RWire's wset method is being simultaneously called in some other rule. If the Valid bit is false, the value carried on the RWire can be garbage. If you observe that this is happening, you can

- Test the Valid bit and handle the invalid case correctly
- Fix the rule schedules, as described earlier, so that the wset executes when expected (and prove to yourself that it will execute).

7.3.3 Coverage

Referring again to Figure 6 and our earlier discussion of module hierarchy, you can see that some coverage measurements are the same as in RTL: access to state elements, activity on module ports, etc.

Behavioral coverage can be measured as follows:

- Rule WILL_FIRE signals can indicate how often a rule fired in an execution.
- Method ENABLE signals can indicate how often a module's method was executed

8 Assertion-based Verification of BSV blocks

The popularity of assertion-based verification has recently increased, and this is likely to continue. An assertion is a statement expressed a logic-based formalism, and expresses some property that must hold in the design. Assertions may be *immediate*, i.e., must hold all the time, or *temporal*, i.e., it expresses some property of values that may span time. An example of an immediate assertion is: "The value in register x must be in the range 0 to 5". An example of a temporal assertion is: "If state becomes BUSY, then state becomes IDLE within 50 cycles". Temporal assertions are particularly useful to express properties of *protocols*.

In principle, one could write RTL or BSV source code to check any property of interest, but because assertions are expressed in a pure, logic-based formalism, they are declarative, very high-level, and represent an *independent* specification of correctness properties (i.e., independent of the implementation). Assertions are also a basis for future formal verification, wherein the property expressed can be checked *statically* using, for example, theorem-proving. For these reasons, assertions have recently become an increasingly prominent verification technique.
The Accellera Standards body provides OVL ("Open Verification Library") which provides a set of 31 pre-packaged assertions written in Verilog Each assertion is a parameterized Verilog module. SystemVerilog has a sub-language called SystemVerilog Assertions (SVA) to express such properties.

The BSV programmer has two routes to using assertions. First, Bluespec has "wrapped" the OVL library so that the OVL modules are available in BSV as BSV modules. Thus, the standard OVL methodology is available to the BSV programmer within the BSV environment.

Second, you can directly write SVA assertions in BSV source code. This aspect of BSV is currently in pre-production status.

9 Summary

BSV's abstraction mechanisms are the most powerful and semantically sound of any synthesizable HDL (more even than many unsynthesizable modelling languages), and dramatically improve the ability to produce correct-by-construction designs, thereby significantly improving verification time. This document has explored these ideas in a little more detail, and has also explored some operational details of how one goes about using BSV in Verification, both debugging BSV designs and using BSV to construct testbenches.