SystemVerilog for VHDL Users

Tom Fitzpatrick
Synopsys, Inc.

Abstract

SystemVerilog was developed to provide an evolutionary path from existing hardware description languages (HDLs) to next-generation design and verification methodologies necessary to support the development of the increasingly complex SoC designs of today and tomorrow. Although its roots are firmly planted in Verilog, many of the features of SystemVerilog were targeted to address capabilities that VHDL users have had for years.

This tutorial will provide an overview of SystemVerilog, focusing on those language features that enable the adoption of SystemVerilog by VHDL designers, such as complex and user-defined data types, multi-dimensional arrays, and the concept of strong data type checking. In addition, we will show how VHDL and Verilog users can take advantage of distinct SystemVerilog features to improve their productivity with advanced coding capability and built-in verification.

1 SystemVerilog Overview

SystemVerilog is the result of an industry-wide effort, through the Accellera standards organization, to create a single language that addresses the design and verification needs of today's (and tomorrow's) users. Accellera was formed in 2000 through the unification of Open Verilog International (OVI) and VHDL International (VIUF) to focus on identifying new standards, develop standards and formats, and foster the adoption of new methodologies. It is therefore no accident that SystemVerilog includes features of both Verilog and VHDL, as well as numerous extensions, to address these needs.

The impetus for the development of SystemVerilog really came from two sources. The first was an attempt to add to the Verilog language [2] many of the design features that VHDL users have had for years. Some of these features, such as multi-dimensional arrays and the *generate* statement were added in Verilog-2001 [3], but as we will discuss in this paper, a substantial amount of additional functionality beyond Verilog-2001 is required. The second factor was the realization that neither Verilog nor VHDL sufficiently support the advanced verification features and methodologies that have led in recent years to the development of proprietary Hardware Verification Languages (HVLs), such as Vera and *e*, and myriad proprietary assertion languages.

This has led to isolated environments, requiring design and verification engineers to become specialists in a particular aspect of the methodology and its accompanying language, at the expense of efficiency and communication between teams.

One of the founding principles of SystemVerilog is the unification of testbench, design and assertion capabilities into a single language. This allows the introduction of new concepts in an incremental fashion, without requiring a complete overhaul of familiar RTL environments. Having a single language allows for common syntax, constructs and idioms to be shared between, for example, the testbench and the design, instead of requiring two completely different languages to express the same basic concept.

Another key to the power and usability of SystemVerilog is that the language *semantics* have been extended to allow the various components of the language to work together seamlessly. For example, extensions to the simulation scheduling provide a clear mapping between event-driven and cycle-based semantics, ensuring consistent results across simulation, synthesis and formal verification tools. Formal property checkers can therefore easily be incorporated into a simulation-based verification methodology.

2 SystemVerilog Design Features

SystemVerilog includes a number of synthesizable design features that have long been available in VHDL. This section discusses these feature and shows how they have been brought into SystemVerilog. In many cases, adding these features to SystemVerilog afforded the opportunity to enhance them to address certain shortcomings of their VHDL implementation as well. In section 2.3, we also discuss SystemVerilog *interfaces*, which provide capabilities similar to, and more powerful than, the VHDL entity-architecture pairing that separates a component's port list from its functional description.

2.1 High-Level Data Types

SystemVerilog adds new data types beyond the scalar types in Verilog. As in VHDL, SystemVerilog includes an enumerated type that is particularly useful for specifying abstract symbolic values for state machine encoding. However, unlike VHDL, SystemVerilog

allows for explicit values to be optionally assigned and for explicit types to be specified to hold the enum value.

VHDL:
```
type MYST is (RESET,IDLE,
              ACK);
```

SystemVerilog:
```
typedef enum bit[1:0]
   {RESET, IDLE, ACK} MYST;
```

The preceding example also demonstrates that SystemVerilog supports user-defined types as well as explicit 2-state ("bit") types.

SystemVerilog also supports complex aggregate types, similar to the VHDL "record" type. However, SystemVerilog uses more of the C-style syntax for specifying aggregates:

VHDL:
```
type PKT is record
  PARITY: std_ulogic;
  ADDR: std_ulogic_vector
          (3 downto 0);
  DEST: std_ulogic_vector
          (3 downto 0);
end record;
```

SystemVerilog:
```
typedef struct {
  logic PARITY;
  logic[3:0] ADDR;
  logic[3:0] DEST;
} pkt_t;
```

Also note that SystemVerilog introduces the "logic" unresolved type, equivalent to "std_ulogic."

Verilog-2001 introduced multi-dimensional arrays to Verilog, and SystemVerilog further extends them to add built-in system functions to query the structure of arrays, such as $left and $right, similar to the 'left and 'right attributes of VHDL. In addition, SystemVerilog also allows arrays of complex types.

In addition to **struct**, SystemVerilog also includes, as does C, the **union** construct, which specifies multiple formats for the same storage. A useful extension in SystemVerilog is that both structs and unions may be *packed* to indicate convenient names for accessing fields of a vector.

SystemVerilog:
```
typedef struct packed {
  logic [15:0] source_port;
  logic [15:0] dest_port;
  logic [31:0] sequence;
} tcp_t;

typedef struct packed {
  logic [15:0] source_port;
  logic [15:0] dest_port;
  logic [15:0] length;
  logic [15:0] checksum
} udp_t;

union packed {
  tcp_t tcp_h;
  udp_t udp_h;
} ip;
```

The packed struct declarations define bit vector types, `tcp_t` and `udp_t`, each of which contains 64 bits, laid out according to a particular pattern. The `ip` union declaration defines a single vector that can be accessed according to either pattern

2.2 Strong Data Typing

Since Verilog only supports scalar data types, it is possible to perform (most of the time) the correct type conversions automatically, albeit with a potential loss of accuracy or precision. For example, integer, bit and real values are correctly and automatically converted when needed. Incorrect conversion occurs when large values are truncated or when the actual value does not have a numerical interpretation – such as ASCII characters.

SystemVerilog maintains the ability to convert automatically between scalar and some aggregate types. Packed *struct* and *union* aggregate types have a well-defined translation to and from scalar bit vectors and can thus be automatically converted. SystemVerilog does offer strong data typing with the higher-level data types.

2.3 Interfaces

SystemVerilog introduces the *interface*, a construct that encapsulates the communication between blocks in a design. At it's most basic, an interface is a bundle of signals (simple_bus) that gets instantiated in a design (top.sb_intf) and can be passed through a module port as a single item (memMod.a), with the specific nets or signals referenced where needed via a hierarchical reference to the port list item (a.req in memMod).

SystemVerilog:
```
interface simple_bus;
  logic req,gnt;
  logic [7:0] addr,data;
  logic [1:0] mode;
  logic start,rdy;
endinterface: simple_bus
```

```
module memMod(interface a,
              input bit clk);
  logic avail;
  always @(posedge clk)
    a.gnt <= a.req & avail;
endmodule

module top;
  bit clk = 0;
  simple_bus sb_intf;

  memMod mem(sb_intf, clk);
endmodule
```

The real power of the interface is its ability to encapsulate the communication protocol as well as the connectivity. By defining the protocol as tasks and/or functions in the interface, the target modules can communicate across the interface by simply calling these tasks in the interface. The interface can also include parameters, variables and local signals, as well as define the connectivity direction for signals using *modports*.

SystemVerilog:

```
interface simple_bus(input bit
clk);
  logic req,gnt;
  logic [7:0] addr,data;
  logic [1:0] mode;
  logic start,rdy;
  modport slave(input req,
       addr,mode,start,clk,
     output gnt,rdy,
     inout data);
...
endinterface: simple_bus
```

This encapsulation mechanism provides two significant advantages. The first is that, by extending the concept of block-based design to the communication between blocks, it is now that case that "somebody owns the bus!" Using interfaces means that instead of multiple engineers defining port connections at opposite ends of a bus, one engineer will encapsulate the bus connectivity and functionality into an interface and allow other team members to connect the interface to their piece of the design. By including assertions in the interface as well, the encapsulated interface automatically becomes self-checking, ensuring that consumers of the interface will adhere to the protocol defined therein.

The other advantage of interfaces is that they can isolate modules from changes in abstraction of other modules in the design. Think of the case where two RTL modules are connected with an interface. With the "send" module calling a task in the interface, the "receive" module could be modeled in RTL, or at the transaction- or gate-level. Because the interface takes care of the abstraction conversion, the send module doesn't have to change to accommodate these changes. This capability is critical in facilitating a methodology where the design starts at the transaction level, making it straightforward to incorporate RTL versions of specific components that connect to transaction-level models of the rest of the system through the interfaces.

2.4 Logic-Specific Processes

VHDL has the *process* statement, Verilog has the equivalent *always* block, and SystemVerilog adds a few nice logic-specific variations of the *always* block that will permit better linting and checking of desired functionality by simulators, synthesis tools and formal tools.

The *always_comb* block conveys the designer's intent to model combinational logic without the need to expand the required combinational sensitivity list. In the following example a simulation or synthesis tool may warn: "Combinational logic requested but latch was inferred."

SystemVerilog:

```
module ao1b (
  output bit q,
  input  bit en, d);

  always_comb
    if (en) q <= d;
endmodule
```

Similarly, SystemVerilog also includes *always_latch* and *always_ff* blocks to specify explicit latch and flip-flop behavior, respectively.

3 SystemVerilog Verification Features

Design features, by their very nature, describe static elements that correspond to actual hardware. When writing testbenches, the coding paradigm is typically much more "software-like" in which dynamic elements are quite useful.

3.1 Dynamic Memory

VHDL includes such constructs as the *access* type to enable specification of dynamic arrays, fifos, queues, etc. All of these constructs have been added explicitly to SystemVerilog, with much of the memory management (in particular, the deallocation of memory) built into the language semantics. The dynamic nature of the array is specified in the declaration, and after that the array is treated as any other array, with the exception that it can be explicitly deleted by calling the "delete" method for the array.

```
int dynarr[] = new[10];
struct1 assocarr[struct2];
struct3 queue[0:$];
```

In the preceding example, dynarr is a dynamic array, declared with an empty range specifier. The allocation of storage is specified by the "new" operator which specifies how many elements are to be allocated to the array. The size of the array may be manually increased or decreased by additional calls to "new." The associative array, assocarr, takes a type name as the range specifier. In this example, each unique value of type struct2 will create a new element of the array. The array itself can be navigated using built-in *next, prev, first* and *last* methods. The queue acts as a variable-length array and uses concatenation and slicing operations to manipulate the array, as:

```
queue = {queue,p};// append p
queue = {r,queue};// prepend r
queue = queue[0:$-1]; // pop p
queue = queue[1:$]; // pop r
```

These convenient coding constructs build on syntax with which users are already familiar, and add the kinds of functionality needed without having to rely on separate VHDL packages of data types and/or functions. User feedback on these features is universally positive, particularly in the area of reading and sharing code between teams.

3.2 Object-Oriented Programming

HVLs, such as Vera, have demonstrated the need to build layered verification environments that facilitate reuse. SystemVerilog's *class* (of which there is no VHDL equivalent) provides an object-oriented programming model in which users may encapsulate data and *methods* (tasks and/or functions) that operate on the data. SystemVerilog supports single-inheritance, allowing new classes to be declared that build on existing classes (called *base classes*) and can add to or overload the data or methods from the base class. As with other OO languages, SystemVerilog also supports virtual methods, static data members, constructors and polymorphism. Classes are instantiated dynamically as *objects*, and can be deleted, assigned and accessed via object *handles*, which behave like safe pointers.

```
class Myclass;
  int a;
  bit b;
  function foo ... endfunction
endclass

class Myclass2 extends Myclass;
  logic [3:0] c;
  function foo ... endfunction
endclass

initial begin
  Myclass mc = new;
  Myclass2 mc2 = new;
end
```

In the preceding example, the base class *Myclass* defines data members *a* and *b*, and the function method *foo*. The extended class *Myclass2* inherits the data members from the base class and adds the data member *c*. The declaration of method function *foo* in *Myclass2* overrides the base class method, although the base class method can be referenced explicitly from *Myclass2* as *super.foo()*.

Class inheritance lets you develop test scenarios in layers. For example, you can start with a base class that is responsible for generating a packet to send into your design. Once you verify that the packet is correct and the design handles it correctly, you can extend the packet generator class to inject errors into the packet. The subclass can still reuse the methods of the base class to send the packet to the design, but now the test verifies that the design handles the error packet correctly. Inheritance allows the subclass to make use of the methods of the base class without having to rewrite them and risk introducing bugs in the testbench.

3.3 Constrained-Random Data Generation

Another verification requirement highlighted by HVLs is the need for the ability to randomly generate coherent and interesting input stimuli. "Randomly generate" is the easy part, which can be done using $random in Verilog or by using one of the pre-defined random generation packages in VHDL. "Coherent and interesting" is the hard part. It requires that the randomly generated values be subjected to constraints to ensure that the set of random values creates valid stimulus. It also requires the ability to cross-constrain multiple instances of those same variable sets to create interesting scenarios. Furthermore, it is also necessary to be able to modify or add to those constraints to create corner cases or inject errors.

SystemVerilog has a powerful constraint specification and control mechanism. Built on top of the object-oriented framework, constraints are declarative and can be overloaded in user or test-specific class extensions. Constraint blocks can also be defined out-of-file or turned off. All of these offer much better control than the simple semantic of soft constraints and are more flexible because no constraint is ever truly hard. Virtual methods can also be overloaded to insert directed procedural data generation statements before or after the randomization process.

The constraint solving mechanism or technology is not specified in the language. It is left to the implementers of the language. But directives, such as *solve before*, can be used to help the solver provide better distribution of solutions for a given variable.

3.4 Dynamic Processes

The fork-join statement allows for the spawning of multiple processes and optionally waiting for all of the processes to complete before continuing execution of other processes and code. VHDL cannot dynamically spawn multiple processes.

SystemVerilog adds even more capability to the original Verilog *fork-join* statement by adding two new join options: join_any and join_none. The fork-join_any combination spawns multiple processes but only waits for the first process to complete before continuing execution of following sequential code. The fork-join_none combination spawns multiple processes but does not wait for any of the processes to complete before continuing execution of the following sequential code. With multiple process thus spawned, SystemVerilog also includes *wait* and *disable fork* commands for synchronizing and terminating spawned processes. The built-in class types *semaphore* and *mailbox* allow for additional synchronization and communication between independent processes.

3.5 Assertions

Many of the SystemVerilog design features mentioned previously, such as the specialized logic processes and other features of the language afford designers the ability to express their *intent* while describing designs, providing additional information to improve the quality of verification. The **Design for Verification** (DFV) methodology thus enabled involves designers more directly in the verification process without requiring a significant learning curve. This DFV methodology is made even more powerful by the introduction of assertions to the language.

SystemVerilog assertions were developed to provide design and verification engineers the means to describe complex behaviors about their designs in a clear and concise manner, building on concepts with which users are already familiar.

With assertions unified syntactically with the rest of SystemVerilog, the user is able to embed assertions directly in-line with the design and other verification code, allowing the tools to infer a great deal of information from the context of the surrounding code. This reduces, in many cases substantially, the amount of code the user must write to specify the behavior, and simplifies the usage model since this information does not have to be duplicated, as it would with a separate assertion language.

The semantics of SVA are defined such that the evaluation of the assertions is guaranteed to be equivalent between simulation, which is event-based, and formal verification, which is cycle-based. This equivalence ensures that multiple tools will all interpret the behaviors specified in SVA in the same way. Moreover, the unification of assertions with design and verification code streamlines interaction to augment the power of assertions. In particular, SystemVerilog allows assertions to communicate information to the testbench and allows the testbench to react to the status of assertions without requiring a separate application programming interface (API) of any kind.

An assertion can apply not only to a Boolean expression, as in VHDL, but also to a sequential property specifying behavior over time. The following example specifies the behavior that when req is high, that *implies* that the ack signal will go high within 4 clock ticks:

```
assert property (@(posedge clk)
   req |=> !ack[*0:3] ##1 ack)
  else call_failed();
```

When the req signal is low, the assertion is ignored, and when the assertion fails, the function *call_failed()* will be called. SystemVerilog also includes the ability to call C functions directly (with needing PLI), so this function could either be a SystemVerilog function or a C function.

4 Adopting SystemVerilog in a VHDL Flow

SystemVerilog adds many of the features with which VHDL designers have become familiar. As such, the barrier for VHDL users to adopt SystemVerilog is much lower than for Verilog. However, with the reality of legacy VHDL code in use, the question of interoperability is critical. Fortunately, many of the issues have been solved for linking Verilog and VHDL components together already.

From the design perspective, the addition of higher-level datatypes in SystemVerilog makes it easier to incorporate a SystemVerilog component into a VHDL design. A synthesizable mapping of VHDL records and multi-dimensional arrays exists to SystemVerilog structures and arrays, so the communication interface between languages can now be expressed at the higher level of abstraction, instead of having to map VHDL constructs to bit vectors as with Verilog.

From the testbench perspective, VHDL users now have an industry standard language that can be used to incorporate proven object-oriented and constrained-random data generation techniques into their methodologies. In fact, the same language interface issues discussed above allow for these capabilities to be layered on top of existing VHDL testbenches, providing a scenario where SystemVerilog class objects could be used to generate random data streams that could easily be passed to VHDL procedures in a bus-functional model, for example.

The use of SystemVerilog assertions is also straightforward with VHDL. Many common assertion checkers are provided as a component library in SystemVerilog that can easily be instantiated in a VHDL design to monitor complex sequential behaviors. Custom

user-defined properties may be written inside SystemVerilog modules that can also be instantiated and connected to a bus, for example, in the VHDL design to validate that the protocol is adhered to. The only feature of SystemVerilog assertions that is not available in VHDL is the embedding of assertions directly in procedural VHDL code and the attendant savings that comes from the automatic inference of contextual information.

5 Conclusion

Although SystemVerilog is built on the Verilog language, many of the features of SystemVerilog were derived from proven VHDL features, and in many cases the features were extended to be even more powerful. As such, VHDL users no longer have to give up familiar features to take advantage of Verilog-based tools and flows. Because mixed-language environments are a reality, SystemVerilog extends and improves the interface between languages, making it even easier to adopt SystemVerilog incrementally in what is today primarily a VHDL environment.

The proliferation of proprietary HVLs and assertion languages in recent years has highlighted the shortcomings of both Verilog and VHDL when it comes to verification. VHDL has been considered superior to Verilog for describing designs at a higher level of abstraction, but when coupled with separate languages and tools for verification, neither language truly fulfills users' requirements.

SystemVerilog, by design, addresses these concerns. For designers, it includes the modeling constructs that VHDL has proven effective, as well as additional powerful constructs like interfaces and assertions to enable a Design for Verification methodology. Designers now have the ability to contribute significantly to the verification process, without requiring a steep learning curve.

For verification, SystemVerilog includes the language features demonstrated effective in HVLs, allowing the most powerful constrained-random and object-oriented methodologies to be implemented. And by extending the language infrastructure to support efficient interaction between the testbench, design and assertions, SystemVerilog eliminates the bottlenecks, both in terms of runtime performance and user productivity, between what have up until now been disjoint and disparate aspects of design and verification. Without this infrastructure, which can only be supported by building it into the language semantics, this unification and improved productivity derived from it cannot be achieved.

The application of SystemVerilog to real design problems has demonstrated a substantial productivity gain due to the reduced amount of code needed to get the same quality of results from synthesis. Less code means not only less time for entry and updating, but also fewer bugs. In addition to a reduced amount of design code, less verification code is needed for the same functionality, which means that more thorough verification is possible at an early stage of the design. Furthermore, the ability to use the same language for design and testbench eases use and debug, and hence improves productivity.

References

[1] Clifford E. Cummings, "SystemVerilog: Is This the Merging of Verilog and VHDL?" SNUG (Synopsys Users Group Boston) Proceedings, September 2003

[2] IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language, IEEE Computer Society, IEEE, New York, NY, IEEE Std 1364-1995

[3] IEEE Standard Verilog Hardware Description Language, IEEE Computer Society, IEEE, New York, NY, IEEE Std 1364-2001

[4] *SystemVerilog 3.1 Accellera's Extensions to Verilog*, Accellera, 2003 freely downloadable from: www.systemverilog.org