2015 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

Designing the Future with Efficiency

Guidance to Adopting SystemVerilog for Design!

Axel Scherer, Cadence Design Systems, Chelmsford, MA, USA (axels@cadence.com)

Junette Tan, PMC Sierra, Burnaby, BC, Canada (Junette.Tan@pmcs.com)

***Abstract*— This year marks the 10th anniversary of the ratification of SystemVerilog IEEE 1800. Still, to date, most IP is developed using classic Verilog. Can we use this antiquated language for designing the future? Do we leave design productivity opportunities on the table? In verification, SystemVerilog has been adopted quickly, since only a few tools in the front-end had to support it. For design models, however, a much larger set of tools need to provide language support. This paper shows which subset of SystemVerilog is supported broadly and is ready for adoption.**

Design, Reuse, Debug, Abstraction, Parameterization, Selfchecking, IEEE 1800, SystemVerilog

I. INTRODUCTION

Both VHDL (IEEE 1076) and Verilog (IEEE 1364) were developed in the 1980s. The primary intent was the ability to model digital hardware in code. Hence, the term Hardware Description Language (HDL) was coined. HDL was a huge leap for electronic design and, further on, enabled massive productivity gains with the introduction of synthesis. With these advances, a designer no longer had to model individual gates but, rather, could focus on a more abstract way of describing functionality. A single person was now able to generate tens of thousands, and later hundreds of thousands, of gates very quickly. It was a watershed moment.

By the mid 1990s design productivity created a verification problem. The ability to release chips that worked on first tape-out was reduced. People wrote simple tests in an HDL to make sure the model was behaving as expected. The methods applied were rather rudimentary. Later, concepts like code coverage came into play, and finally the sophistication level rose with functional coverage, constrained random test generation, verification planning, and so forth.

The core problem remained: An HDL is an HDL. It was not designed for verification. As neither VHDL nor Verilog gained sufficient verification support fast enough, verification languages such as *e* (IEEE 1647), Vera, Superlog, amongst others, emerged. Verilog was eventually superseded by SystemVerilog (IEEE 1800). Its extensions over standard Verilog were primarily targeting verification. The level of expansion cannot be underestimated. The new language contained the entire legacy Verilog HDL, which is complex by itself, and it gained object oriented programming, constrained randomization, functional coverage, assertions, and more.

The adoption of SystemVerilog for verification was not trivial. As John Aynsley, CTO of Doulos, pointed out at DVCon San Jose 2012: “Further evidence of SystemVerilog’s size and

complexity is the length of its BNF formal syntax definition, which covers 43 pages of the standard language reference manual, and is 70-80% larger than that of VHDL."

This is an important statement. In particular, as the complexity of a language relative to the size of its BNF does not scale in a linear fashion. Instead, complexity explodes quickly. John further pointed out that C++ 1998 only has an 18 page BNF!

Nevertheless, the adoption of SystemVerilog for verification still occurred fairly quickly because people were able to cope with its complexity. Additionally, the number of verification tools required to process this language was rather small, making adoption from a tool perspective easier—The primary tool for a testbench is a simulator after all.

In the design space, the world is very different as the hurdles to adoption are very different. The RTL hardware model has to be processed by many more tool categories than does the testbench, let alone accounting for various tool implementations.

SystemVerilog for design requires broad support and optimization from:

> **Linting, Simulation, Emulation/Acceleration, Synthesis, Test insertion (DFT), Low Power, FPGA prototyping, Equivalence Checking, Property checking**

The combination of tool categories combined with a multi-vendor support ecosystem makes adoption for design a much harder problem than for verification.

However, the benefits of SystemVerilog are worth overcoming these challenges.

II. MOTIVATION

Why would anyone ever attempt something like the adoption of SystemVerilog for design? Quite simply, because the gains can be quite significant. SystemVerilog combines some of the benefits of VHDL and introduces additional concepts. SystemVerilog models can avoid whole classes of design bugs up front, by making the code more concise, easier to read and comprehend, and more adaptable to change in current or future projects. It offers the following improvements:

- Increased abstraction with new and user defined data types, and clustering constructs such as structs and interfaces.
- Much better ways of parameterization, which make the code much more suitable for IP development, and more robust to handle engineering change orders (ECOs)
- Preemptive bug avoidance during coding stage.
    - Common mistakes will no longer pass the parser.
    - Other types of mistakes will be checked and flagged by the tools.

These benefits of bug preemption alone can be worth the cost of adoption.

As SystemVerilog and the associated tools have matured, the time is ripe to take advantage of the new features. After 10 years, we have reached an inflection point where the support across the industry makes SystemVerilog for design adoption tangible.

Great strides have been made by tool vendors in the last few years to improve support of SystemVerilog for design. With increased adoption, community pressure will grow and the remaining support gaps will close even faster.

This paper will show what works using examples of basic constructs that everyone should adopt, and it also covers constructs that are very beneficial, but take on more risk.

III. Design and Verification Code Sharing

Fundamentally, code sharing between design and verification teams can results in repeating certain mistakes twice. Design and verification should be derived from a comprehensive specification and that derivation should occur separately. However, aspects that are explicitly stated in the specification documentation do not have to be implemented by both teams. In such cases it makes sense to share code.

The SystemVerilog *package* is useful in that it enables this code sharing. To share code the *package* should contain register addresses and bit fields—both for design connectivity, as well as for verification hook up to the DUT. Further on, there are sets of *type definitions* that also make sense to share, such as certain *enumerated types*.

IV. Basic Construct – always_ff, always_comb, always_latch

One of the least intrusive constructs you can adopt is the *always_ff*, *always_comb* and *always_latch* constructs. Singular *always* blocks tend to infer ambiguity. Hence, if you move to procedural block constructs, the intent of the block is clearly stated in the use of the construct. Additionally, the tools will actually check if the logic in such a block is indeed sequential or combinatorial. Restrictions in coding will make the code cleaner and lead to fewer bugs upfront.

V. Basic Construct – Enumerated Type

The enumerated type is a construct that everyone using SystemVerilog for design should adopt. It is widely supported, does not introduce complex code changes, and brings clear benefits. The reward/risk ratio is very high.

In classic Verilog `` `define `` and `localparam` were widely adopted for various purposes. The problem with `` `define `` is that it is global in nature. In addition, the problem with either option is that the individual definitions are not connected in any form or shape. They might be combined by locality during the definition stage; however, there is no actual grouping in the language. If we use `` `define `` or `localparam` for branching control or FSM implementation, for example, we cannot take advantage of benefits that come with grouped values.

In the following basic definition of an enumerated type for the AHB transaction type, we can see the grouping of values.

```
typedef enum  {IDLE = 4, BUSY = 5, NONSEQUENTIAL = 6, SEQUENTIAL = 7 } htrans_e;
```

This grouping and the associated encoding can be leveraged right away, as follows:

```
htrans_e htrans;

  initial begin : initialization

    htrans = htrans.first();

  end : initialization
```

For example, you can use the built-in function `first()` of the enumerated type to model initialization. The benefit of this approach is that the enumeration definition can change and put a different constant first but the initialization code does not have to change with the definition. This is especially useful as state machines evolve in the design and states are renamed.

Enumerated types bring additional benefits—they prevent common mistakes! For example, it is not allowed to define the same value twice. By default, this does not occur as implicit values are defined in order, from 0 on, by increments of 1. However, if you need to set the values explicitly, you will never be able to introduce a bug caught by dual use of the same value, as this will be caught at compile time. Consider the following example:

```
typedef enum {IDLE = 4, BUSY = 4, NONSEQUENTIAL = 6, SEQUENTIAL = 7} htrans_e;
```

The definition shown above is not legal and causes a compile error. However, if you were using `` `define `` or `localparam` you would not catch this problem at an early stage, as it is perfectly legal, and the individual names are completely independent and have no relationship to one another. The grouping of enumerated values in a type definition is therefore a great benefit, in particular when the enumerated type contains a lot of values, which would increase the likelihood of mistakes.

There is another benefit. Assume you want to enforce the values of the enumerated type to be compliant with a particular data type, as follows:

```
typedef enum logic[3:0] {IDLE = 4, BUSY = 5, NONSEQUENTIAL = 6, SEQUENTIAL = 17} htrans_e;
```

This can be beneficial in order to keep the value set limited and, therefore, also prevent potential bugs. In the example above, `SEQUENTIAL` could not be assigned to `17`. (The compiler will flag it as a violation to `logic[3:0]`, which only allows `4'b0000` to `4'b1111` (decimal `0` to `15`).

Another great benefit of enumerated types is that they are strongly typed and therefore implicitly prevent all kinds of problems in the logic, as shown below.

```
typedef enum logic[3:0] {IDLE = 4, BUSY = 5, NONSEQUENTIAL = 6, SEQUENTIAL = 7} htrans_e;

htrans_e htrans = 8;
```

In the example, you cannot actually assign `8` to `htrans`. Tools can flag that you are actually not using a proper value.

Additionally, the enumerated type definition could be shared between the design and verification processes as part of a *package*. Although this is typically frowned upon, in many

cases it can make sense as these types stem directly from the specification, and in some cases could even be automatically generated from the specification.

Finally, the explicit naming of values through enumerated types makes debug easier. For example, in waveform displays, code value browsing, and other viewers and browsers, it is no longer necessary to work on tool specific *tricks* and custom scripts to get a named value. You will actually *see* the named value everywhere and, since it is part of the language, the tool vendors can collect the names from the internal model representation. Through the introduction of the enumerated type, the name value has become native in the language and the supporting tools.

VI. BASIC CONCEPT: TYPEDEF

Abstraction is almost always a good concept to leverage, in particular when it comes to design productivity—it drove the idea behind HDLs in the first place! With the introduction of type definitions in SystemVerilog, the designer gains additional freedom from the ability to define new types for data or control structures that are customized and meaningful for the application at hand.

Through the abstraction of defined types, the required code length will also be reduced, which will lead to further productivity. Finally, the naming of the types themselves will also lead to better and more maintainable code.

Using the APB example even the simplest type definition can illustrate its usefulness:

```
typedef logic [31:0] apb_data_t;
typedef logic [31:0] apb_addr_t;
```

These two type definitions can now be used in various places.

For example, in the interface definition below, `pwdata` and `prdata` now have a common type. Consequently, changes to the type definition in one place (data width) will impact two places in the interfaces, creating more modularity and lowering the risk of design bugs.

```
interface apb_if (input logic pclk, input logic presetn);

  logic       psel;
  apb_addr_t  paddr;
  logic       penable;
  logic       pwrite;
  apb_data_t  pwdata;
  logic       pready;
  apb_data_t  prdata;

        /* SNAP */
endinterface : apb_if
```

VII. BASIC CONCEPT—USEFUL FUNCTIONS: $BITS, $SIZE

SystemVerilog brings a significant set of new built-in functions to the language. Some of them are extremely beneficial for making your code cleaner and more compact as shown in a

simple example below which illustrates how you can determine the sizing of a generation loop.

```
for (genvar i = 0; i < $size(receivers); i++) //SNAP
```

These new functions complement existing Verilog functions such as `$clogb2` which is often useful in parameter declarations.

VIII. INTERFACES

One of the most important features in SystemVerilog for design is *interfaces*. Interfaces can lead to massive amounts of code reduction, which improves readability, bug rates, and the ability for reuse. However, interfaces are more complex from both a coding and also from a tool perspective. Therefore, tool support across the industry is more challenging and care should be taken to check tool limitations before using the advanced features of interfaces. Current tools support the examples shown in this section.

Interfaces are more than a bundle of wires. They are containers for various data types to connect blocks in an efficient manner. In addition, interfaces offer the opportunity to contain auxiliary logic as well as assertions. Hence, it makes sense that assertions used to check protocol compliance are implemented right in the interface. This helps support modularity, code sharing, and reuse.

In the example below, we define an interface for APB.

```
interface apb_if (input logic pclk, input logic presetn);

  logic        psel;
  apb_addr_t   paddr;
  logic        penable;
  logic        pwrite;
  apb_data_t   pwdata;
  logic        pready;
  apb_data_t   prdata;

  modport master(output psel, paddr, penable, pwrite, pwdata,
                 input  pready, prdata);

  modport slave (input  psel, paddr, penable, pwrite, pwdata,
                 output pready, prdata);

  // ASSERTIONS …

 /* SNAP */
endinterface : apb_if
```

One of the attractive aspects of interfaces is *modports*. In the interface example above, we get directionality from a master and a slave perspective. And, when we use this interface, we can choose which *modport* is appropriate for the intended usage.

Now let's see how such an interface can make module connectivity significantly more compact, adaptable and elegant.

```
    // master
    apb_master  master (.master(host), .slave);

    // interface instances
    apb_if      slave[apb_pkg::SLAVE_NUM] (.*);
```

```
    // slaves
    apb_gpu     gpu  (.apb(slave[apb_pkg::SLAVE_GPU]));
    apb_flash   flash(.apb(slave[apb_pkg::SLAVE_FLASH]);
    apb_usb     usb  (.apb(slave[apb_pkg::SLAVE_USB]));
```

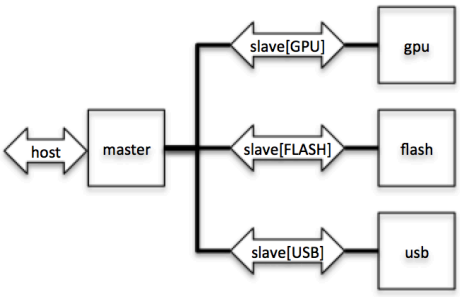

Figure 1: Master-Slave connectivity

In the example above, you can see that we could connect an APB master and three slaves using just five lines of code. The example instantiates three slave interfaces in just one line of code. Then, the example goes on to connect the master to these three slaves. The three slave module instances hook up to their respective interfaces. Additionally the master is also hooked up to a *host* interface. If you write this out in regular Verilog you will wind up with a code explosion of one order of magnitude!

Just for the master alone, the connections to the three slaves would require massive signal hook-up for every signal connection.

```
apb_master master (
   .master_addr  (host_paddr),
// SNAP to shorten code example
   .slave0_addr  (slave_paddr  [SLAVE_GPU]),
   .slave0_wdata (slave_pwdata [SLAVE_GPU]),
   .slave0_write (slave_pwrite [SLAVE_GPU]),
   .slave0_enable(slave_penable[SLAVE_GPU]),
   .slave0_ready (slave_pready [SLAVE_GPU]),
   .slave0_rdata (slave_prdata [SLAVE_GPU]),
// SNAP to shorten code example
   .slave2_rdata (slave_rdata  [SLAVE_USB])
   );
```

The APB example is actually understating the magnitude of code compression, as the APB protocol is very simple and deals with less than ten signals per interface. Imagine you are using AXI4, for example, which contains 40 signals (including *reset* and *clock)*. In this case, you would see much higher code compression.

IX. STRUCTS

Structs are a great way to introduce even more abstraction into a design. Similar to the enumerated type, they form groups of related items. Unlike enumerated types, however, they are not dealing with just a single type definition but they also provide a container for aggregation.

In our APB example we can use this feature to make our code even more elegant, by defining the set of items the master transfers to the slaves, and conversely.

```
    // APB request (master to slave).
```

```
    typedef struct packed {apb_addr_t addr; logic enable; logic write; apb_data_t wdata;} apb_req_s;
    // APB response (slave to master).
    typedef struct packed {logic ready; apb_data_t rdata;} apb_resp_s;
```

Now we can use the struct to simplify our interface definition, as follows:

```
interface apb_if (input logic pclk, input logic presetn);
  import apb_pkg::*;

  logic      psel;
  apb_req_s  req;
  apb_resp_s resp;

  modport master(output req, sel, input  resp);
  modport slave (input  req, sel, output resp);

 /* SNAP */
endinterface : apb_if
```

X. NESTING CONSTRUCTS

Most of the constructs shown in this paper have wide industry support. We also have shown how enumerated types and *structs* can be combined to form better interfaces. The bottom-up approach is working. However, there are other areas in the language that can make adoption challenging. The more levels of nesting you add to the mix, the more likely you will deal with tool issues and support problems. Almost inevitably, at least one tool will not be able to handle something properly. In the example below, we are picking a particular interface using an enumerated type variable. Then, we use a struct inside of it, which also contains an enumerated type variable. Finally, we use a function on that enumerated type variable. With the current level of tool support, this is not going to work across the flow.

```
  assign next_phase = slave[slave_select].req.phase.next;

  // next_phase is of type phase_e, slave is an array of interfaces, slave_select is of type slave_e
  // req is a struct,  phase is of type phase_e
```

Increased adoption of the community should reduce problems with nesting.

XI. SUMMARY

The discussion in this paper has shown which features in SystemVerilog are ready for adoption by the design community today. By taking advantage of these constructs, productivity can increase, with the end result being robust code that is more compact, maintainable, and less prone to bugs. With more companies moving to adopt SystemVerilog for design, it is time to benefit from the features of the language and seize this opportunity to stay ahead of the competition.

REFERENCES

[1] J. Aynsley, "Easier SystemVerilog with UVM: Taming the Beast" DVCon, San Jose, February 2012

[2] J. Tan, M. Schaffstein, S. Sutherland, "SystemVerilog Design: User Experience Defines Multi-Tool, Multi-Vendor Language Working Set " DVCon, San Jose, March 2015