SystemVerilog

SystemVerilog, standardized as **IEEE 1800**, is a hardware description and hardware verification language used to model, design, simulate, test and implement electronic systems. SystemVerilog is based on Verilog and some extensions, and since 2008 Verilog is now part of the same IEEE standard. It is commonly used in the semiconductor and electronic design industry as an evolution of Verilog.

SystemVerilog

SystemVerilog logo

Paradigm	Structured (design) Object-oriented (verification)
Designed by	Synopsys, later IEEE
First appeared	2002
Stable release	IEEE 1800-2017 / February 22, 2018
Typing discipline	Static, weak
Filename extensions	.sv, .svh
Influenced by	
Design: Verilog, VHDL, C++, Verification: OpenVera, Java	

Contents

History

SystemVerilog started with the donation of the Superlog language to Accellera in 2002 by the startup company Co-Design Automation.[1] The bulk of the verification functionality is based on the OpenVera language donated by Synopsys. In 2005, SystemVerilog was adopted as IEEE Standard 1800-2005.[2] In 2009, the standard was merged with the base Verilog (IEEE 1364-2005) standard, creating IEEE Standard 1800-2009. The current version is IEEE standard 1800-2017.[3]

The feature-set of SystemVerilog can be divided into two distinct roles:

1. SystemVerilog for register-transfer level (RTL) design is an extension of Verilog-2005; all features of that language are available in SystemVerilog. Therefore, Verilog is a subset of SystemVerilog.
2. SystemVerilog for verification uses extensive object-oriented programming techniques and is more closely related to Java than Verilog. These constructs are generally not synthesizable.

The remainder of this article discusses the features of SystemVerilog not present in Verilog-2005.

Design features

Data lifetime

There are two types of data lifetime specified in SystemVerilog: static and automatic. Automatic variables are created the moment program execution comes to the scope of the variable. Static variables are created at the start of the program's execution and keep the same value during the entire program's lifespan, unless assigned a new value during execution.

Any variable that is declared inside a task or function without specifying type will be considered automatic. To specify that a variable is static place the "`static`" keyword in the declaration before the type, e.g., "`static int x;`". The "`automatic`" keyword is used in the same way.

New data types

Enhanced variable types add new capability to Verilog's "reg" type:

```
logic [31:0] my_var;
```

Verilog-1995 and -2001 limit reg variables to behavioral statements such as RTL code. SystemVerilog extends the reg type so it can be driven by a single driver such as gate or module. SystemVerilog names this type "logic" to remind users that it has this extra capability and is not a hardware register. The names "logic" and "reg" are interchangeable. A signal with more than one driver (such as a tri-state buffer for general-purpose input/output) needs to be declared a net type such as "wire" so SystemVerilog can resolve the final value.

Multidimensional packed arrays unify and extend Verilog's notion of "registers" and "memories":

```
logic [1:0][2:0] my_pack[32];
```

Classical Verilog permitted only one dimension to be declared to the left of the variable name. SystemVerilog permits any number of such "packed" dimensions. A variable of packed array type maps 1:1 onto an integer arithmetic quantity. In the example above, each element of `my_pack` may be used in expressions as a six-bit integer. The dimensions to the right of the name (32 in this case) are referred to as "unpacked" dimensions. As in Verilog-2001, any number of unpacked dimensions is permitted.

Enumerated data types (`enums`) allow numeric quantities to be assigned meaningful names. Variables declared to be of enumerated type cannot be assigned to variables of a different enumerated type without casting. This is not true of parameters, which were the preferred implementation technique for enumerated quantities in Verilog-2005:

```
typedef enum logic [2:0] {
   RED, GREEN, BLUE, CYAN, MAGENTA, YELLOW
} color_t;

color_t   my_color = GREEN;
initial $display("The color is %s", my_color.name());
```

As shown above, the designer can specify an underlying arithmetic type (`logic [2:0]` in this case) which is used to represent the enumeration value. The meta-values X and Z can be used here, possibly to represent illegal states. The built-in function `name()` returns an ASCII string for the current enumerated value, which is useful in validation and testing.

New integer types: SystemVerilog defines `byte`, `shortint`, `int` and `longint` as two-state signed integral types having 8, 16, 32, and 64 bits respectively. A `bit` type is a variable-width two-state type that works much like `logic`. Two-state types lack the X and Z metavalues of classical Verilog; working with these types may result in faster simulation.

Structures and **unions** work much like they do in the C programming language. SystemVerilog enhancements include the **packed** attribute and the **tagged** attribute. The `tagged` attribute allows runtime tracking of which member(s) of a union are currently in use. The `packed` attribute causes the structure or union to be mapped 1:1 onto a packed array of bits. The contents of `struct` data types occupy a continuous block of memory with no gaps, similar to bitfields in C and C++:

```
typedef struct packed {
    bit [10:0]  expo;
    bit         sign;
    bit [51:0]  mant;
} FP;

FP     zero = 64'b0;
```

As shown in this example, SystemVerilog also supports typedefs, as in C and C++.

Procedural blocks

SystemVerilog introduces three new procedural blocks intended to model hardware: `always_comb` (to model combinational logic), `always_ff` (for flip-flops), and `always_latch` (for latches). Whereas Verilog used a single, general-purpose `always` block to model different types of hardware structures, each of SystemVerilog's new blocks is intended to model a specific type of hardware, by imposing semantic restrictions to ensure that hardware described by the blocks matches the intended usage of the model. An HDL compiler or verification program can take extra steps to ensure that only the intended type of behavior occurs.

An `always_comb` block models combinational logic. The simulator infers the sensitivity list to be all variables from the contained statements:

```
always_comb begin
    tmp = b * b - 4 * a * c;
    no_root = (tmp < 0);
end
```

An `always_latch` block is meant to infer a level-sensitive latch. Again, the sensitivity list is inferred from the code:

```
always_latch
    if (en) q <= d;
```

An `always_ff` block is meant to model synchronous logic (especially edge-sensitive sequential logic):

```
always_ff @(posedge clk)
    count <= count + 1;
```

Electronic design automation (EDA) tools can verify the design's intent by checking that the hardware model does not violate any block usage semantics. For example, the new blocks restrict assignment to a variable by allowing only one source, whereas Verilog's `always` block permitted assignment from multiple procedural sources.

Interfaces

For small designs, the Verilog *port* compactly describes a module's connectivity with the surrounding environment. But major blocks within a large design hierarchy typically possess port counts in the thousands. SystemVerilog introduces concept of interfaces to both reduce the redundancy of port-name declarations between connected modules, as well as group and abstract related signals into a user-declared bundle. Additional concept is modport, that shows direction of logic connections.

Example:

```
interface intf;
  logic a;
  logic b;
  modport in (input a, output b);
  modport out (input b, output a);
endinterface

module top;
  intf i ();
  u_a m1 (.i1(i.in));
  u_b m2 (.i2(i.out));
endmodule

module u_a (intf.in i1);
endmodule

module u_b (intf.out i2);
endmodule
```

Verification features

The following verification features are typically not synthesizable, meaning they cannot be implemented in hardware based on HDL code. Instead, they assist in the creation of extensible, flexible test benches.

New data types

The `string` data type represents a variable-length text string. For example:

```
string s1 = "Hello";
string s2 = "world";
string p = ".?!";
string s3 = {s1, ", ", s2, p[2]}; // string concatenation
$display("[%d] %s", s3.len(), s3); // simulation will print: "[13] Hello, world!"
```

In addition to the static array used in design, SystemVerilog offers dynamic arrays, associative arrays and queues:

```
int cmdline_elements; // # elements for dynamic array
int da[];             // dynamic array
int ai[int];          // associative array, indexed by int
int as[string];       // associative array, indexed by string
int qa[$];            // queue, indexed as an array, or by built-in methods

initial begin
    cmdline_elements = 16;
    da = new[ cmdline_elements ]; // Allocate array with 16 elements
end
```

A dynamic array works much like an unpacked array, but offers the advantage of being dynamically allocated at runtime (as shown above.) Whereas a packed array's size must be known at compile time (from a constant or expression of constants), the dynamic array size can be initialized from another runtime variable, allowing the array to be sized and resize arbitrarily as needed.

An associative array can be thought of as a binary search tree with a user-specified key type and data type. The key implies an ordering; the elements of an associative array can be read out in lexicographic order. Finally, a queue provides much of the functionality of the C++ STL deque type: elements can be added and removed from either end efficiently. These primitives allow the creation of complex data structures required for scoreboarding a large design.

Classes

SystemVerilog provides an object-oriented programming model.

In SystemVerilog, classes support a single-inheritance model, but may implement functionality similar to multiple-inheritance through the use of so-called "interface classes" (identical in concept to the `interface` feature of Java). Classes can be parameterized by type, providing the basic function of C++ templates. However, template specialization and function templates are not supported.

SystemVerilog's polymorphism features are similar to those of C++: the programmer may specifically write a `virtual` function to have a derived class gain control of the function. See virtual function for further info.

Encapsulation and data hiding is accomplished using the `local` and `protected` keywords, which must be applied to any item that is to be hidden. By default, all class properties are public.

Class instances are dynamically created with the `new` keyword. A constructor denoted by `function new` can be defined. SystemVerilog has automatic garbage collection, so there is no language facility to explicitly destroy instances created by the new operator.

Example:

```
virtual class Memory;
    virtual function bit [31:0] read(bit [31:0] addr); endfunction
    virtual function void write(bit [31:0] addr, bit [31:0] data); endfunction
endclass

class SRAM #(parameter AWIDTH=10) extends Memory;
    bit [31:0] mem [1<<AWIDTH];

    virtual function bit [31:0] read(bit [31:0] addr);
        return mem[addr];
    endfunction

    virtual function void write(bit [31:0] addr, bit [31:0] data);
```

```
        mem[addr] = data;
    endfunction
endclass
```

Constrained random generation

Integer quantities, defined either in a class definition or as stand-alone variables in some lexical scope, can be assigned random values based on a set of constraints. This feature is useful for creating randomized scenarios for verification.

Within class definitions, the `rand` and `randc` modifiers signal variables that are to undergo randomization. `randc` specifies permutation-based randomization, where a variable will take on all possible values once before any value is repeated. Variables without modifiers are not randomized.

```
class eth_frame;
    rand bit [47:0] dest;
    rand bit [47:0] src;
    rand bit [15:0] f_type;
    rand byte       payload[];
    bit [31:0]      fcs;
    rand bit [31:0] fcs_corrupt;

    constraint basic {
        payload.size inside {[46:1500]};
    }

    constraint good_fr {
        fcs_corrupt == 0;
    }
endclass
```

In this example, the `fcs` field is not randomized; in practice it will be computed with a CRC generator, and the `fcs_corrupt` field used to corrupt it to inject FCS errors. The two constraints shown are applicable to conforming Ethernet frames. Constraints may be selectively enabled; this feature would be required in the example above to generate corrupt frames. Constraints may be arbitrarily complex, involving interrelationships among variables, implications, and iteration. The SystemVerilog constraint solver is required to find a solution if one exists, but makes no guarantees as to the time it will require to do so as this is in general an NP-hard problem (boolean satisfiability).

Randomization methods

In each SystemVerilog class there are 3 predefined methods for randomization: pre_randomize, randomize and post_randomize. The randomize method is called by the user for randomization of the class variables. The pre_randomize method is called by the randomize method before the randomization and the post_randomize method is called by the randomize method after randomization.

```
class eth_frame;
    rand bit [47:0] dest;
    rand bit [47:0] src;
    rand bit [15:0] f_type;
    rand byte       payload[];
    bit [31:0]      fcs;
    rand bit        corrupted_frame;

    constraint basic {
        payload.size inside {[46:1500]};
    }

    function void post_randomize()
      this.calculate_fcs(); // update the fcs field according to the randomized frame
      if (corrupted_frame)  // if this frame should be corrupted
        this.corrupt_fcs(); // corrupt the fcs
```

```
    endfunction
endclass
```

Controlling constraints

The constraint_mode() and the random_mode() methods are used to control the randomization. constraint_mode() is used to turn a specific constraint on and off and the random_mode is used to turn a randomization of a specific variable on or off. The below code describes and procedurally tests an Ethernet frame:

```
class eth_frame;
    rand bit [47:0] dest;
    rand bit [47:0] src;
    rand bit [15:0] f_type;
    rand byte       payload[];
    bit [31:0]      fcs;
    rand bit        corrupted_frame;

    constraint basic {
        payload.size inside {[46:1500]};
    }

    constraint one_src_cst {
        src == 48'h1f00
    }

    constraint dist_to_fcs {
        fcs dist {0:/30,[1:2500]:/50};  // 30, and 50 are the weights (30/80 or  50/80, in this
example)
    }

endclass
.
.
.
eth_frame my_frame;

my_frame.one_src_cst.constraint_mode(0); // the constraint one_src_cst will not be taken into
account
my_frame.f_type.random_mode(0);        // the f_type variable will not be randomized for this
frame instance.
my_frame.randomize();
```

Assertions

Assertions are useful for verifying properties of a design that manifest themselves after a specific condition or state is reached. SystemVerilog has its own assertion specification language, similar to Property Specification Language. The subset of SystemVerilog language constructs that serves assertion is commonly called SystemVerilog Assertion or SVA.[4]

SystemVerilog assertions are built from **sequences** and **properties**. Properties are a superset of sequences; any sequence may be used as if it were a property, although this is not typically useful.

Sequences consist of boolean expressions augmented with temporal operators. The simplest temporal operator is the ## operator which performs a concatenation:

```
sequence S1;
    @(posedge clk) req ##1 gnt;
endsequence
```

This sequence matches if the `gnt` signal goes high one clock cycle after `req` goes high. Note that all sequence operations are synchronous to a clock.

Other sequential operators include repetition operators, as well as various conjunctions. These operators allow the designer to express complex relationships among design components.

An assertion works by continually attempting to evaluate a sequence or property. An assertion fails if the property fails. The sequence above will fail whenever `req` is low. To accurately express the requirement that `gnt` follow `req` a property is required:

```
property req_gnt;
    @(posedge clk) req |=> gnt;
endproperty

assert_req_gnt: assert property (req_gnt) else $error("req not followed by gnt.");
```

This example shows an **implication** operator `|=>`. The clause to the left of the implication is called the **antecedent** and the clause to the right is called the **consequent**. Evaluation of an implication starts through repeated attempts to evaluate the antecedent. When the antecedent succeeds, the consequent is attempted, and the success of the assertion depends on the success of the consequent. In this example, the consequent won't be attempted until `req` goes high, after which the property will fail if `gnt` is not high on the following clock.

In addition to assertions, SystemVerilog supports assumptions and coverage of properties. An assumption establishes a condition that a formal logic proving tool must assume to be true. An assertion specifies a property that must be proven true. In simulation, both assertions and assumptions are verified against test stimuli. Property coverage allows the verification engineer to verify that assertions are accurately monitoring the design.

Coverage

Coverage as applied to hardware verification languages refers to the collection of statistics based on sampling events within the simulation. Coverage is used to determine when the device under test (DUT) has been exposed to a sufficient variety of stimuli that there is a high confidence that the DUT is functioning correctly. Note that this differs from code coverage which instruments the design code to ensure that all lines of code in the design have been executed. Functional coverage ensures that all desired corner and edge cases in the design space have been explored.

A SystemVerilog coverage group creates a database of "bins" that store a histogram of values of an associated variable. Cross-coverage can also be defined, which creates a histogram representing the Cartesian product of multiple variables.

A sampling event controls when a sample is taken. The sampling event can be a Verilog event, the entry or exit of a block of code, or a call to the `sample` method of the coverage group. Care is required to ensure that data are sampled only when meaningful.

For example:

```
class eth_frame;
   // Definitions as above
   covergroup cov;
      coverpoint dest {
          bins bcast[1] = {48'hFFFFFFFFFFFF};
          bins ucast[1] = default;
      }
      coverpoint f_type {
          bins length[16] = { [0:1535] };
          bins typed[16] = { [1536:32767] };
          bins other[1] = default;
      }
      psize: coverpoint payload.size {
```

```
            bins size[] = { 46, [47:63], 64, [65:511], [512:1023], [1024:1499], 1500 };
        }

        sz_x_t: cross f_type, psize;
    endgroup
endclass
```

In this example, the verification engineer is interested in the distribution of broadcast and unicast frames, the size/f_type field and the payload size. The ranges in the payload size coverpoint reflect the interesting corner cases, including minimum and maximum size frames.

Synchronization

A complex test environment consists of reusable verification components that must communicate with one another. Verilog's 'event' primitive allowed different blocks of procedural statements to trigger each other, but enforcing thread synchronization was up to the programmer's (clever) usage. SystemVerilog offers two primitives specifically for interthread synchronization: *mailbox* and *semaphore*. The mailbox is modeled as a FIFO message queue. Optionally, the FIFO can be type-parameterized so that only objects of the specified type may be passed through it. Typically, objects are class instances representing *transactions*: elementary operations (for example, sending a frame) that are executed by the verification components. The semaphore is modeled as a counting semaphore.

General improvements to classical Verilog

In addition to the new features above, SystemVerilog enhances the usability of Verilog's existing language features. The following are some of these enhancements:

- The procedural assignment operators (<=, =) can now operate directly on arrays.
- Port (inout, input, output) definitions are now expanded to support a wider variety of data types: struct, enum, real, and multi-dimensional types are supported.
- The for loop construct now allows automatic variable declaration inside the for statement. Loop flow control is improved by the *continue* and *break* statements.
- SystemVerilog adds a *do*/while loop to the *while* loop construct.
- Constant variables, i.e. those designated as non-changing during runtime, can be designated by use of *const*.
- Variable initialization can now operate on arrays.
- Increment and decrement operators (`x++`, `++x`, `x--`, `--x`) are supported in SystemVerilog, as are other compound assignment operators (`x += a`, `x -= a`, `x *= a`, `x /= a`, `x %= a`, `x <<= a`, `x >>= a`, `x &= a`, `x ^= a`, `x |= a`) as in C and descendants.
- The preprocessor has improved `define macro-substitution capabilities, specifically substitution within literal-strings (""), as well as concatenation of multiple macro-tokens into a single word.
- The fork/join construct has been expanded with *join_none* and *join_any*.
- Additions to the `timescale directive allow simulation timescale to be controlled more predictably in a large simulation environment, with each source file using a local timescale.
- Task ports can now be declared *ref*. A reference gives the task body direct access to the source arguments in the caller's scope, known as "pass by reference" in computer programming. Since it is operating on the original variable itself, rather than a copy of the argument's value, the task/function can modify variables (but not nets) in the caller's scope in real time. The inout/output port declarations pass variables by value, and defer updating the caller-scope variable until the moment the task exits.
- Functions can now be declared *void*, which means it returns no value.

- Parameters can be declared any type, including user-defined *typedef*s.

Besides this, SystemVerilog allows convenient interface to foreign languages (like C/C++), by SystemVerilog DPI (Direct Programming Interface).

Verification and synthesis software

In the design verification role, SystemVerilog is widely used in the chip-design industry. The three largest EDA vendors (Cadence Design Systems, Mentor Graphics, Synopsys) have incorporated SystemVerilog into their mixed-language HDL simulators. Although no simulator can yet claim support for the entire SystemVerilog LRM, making testbench interoperability a challenge, efforts to promote cross-vendor compatibility are underway. In 2008, Cadence and Mentor released the Open Verification Methodology, an open-source class-library and usage-framework to facilitate the development of re-usable testbenches and canned verification-IP. Synopsys, which had been the first to publish a SystemVerilog class-library (VMM), subsequently responded by opening its proprietary VMM to the general public. Many third-party providers have announced or already released SystemVerilog verification IP.

In the design synthesis role (transformation of a hardware-design description into a gate-netlist), SystemVerilog adoption has been slow. Many design teams use design flows which involve multiple tools from different vendors. Most design teams cannot migrate to SystemVerilog RTL-design until their entire front-end tool suite (linters, formal verification and automated test structure generators) support a common language subset.

See also

- List of SystemVerilog Simulators (Search for SV2005)
- Verilog-AMS
- e (verification language)
- SpecC
- Accellera
- SystemC
- SystemRDL

References

1. Rich, D. “The evolution of SystemVerilog” IEEE Design and Test of Computers, July/August 2003
2. IEEE approves SystemVerilog, revision of Verilog (http://www.eetimes.com/news/design/showArticle.jhtml;?articleID=173601060)
3. 1800-2017 - IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language (http://standards.ieee.org/findstds/standard/1800-2017.html)
4. SystemVerilog Assertion: Introduction (http://www.project-veripage.com/sva_1.php)

- *1800-2005 — IEEE Standard for System Verilog—Unified Hardware Design, Specification, and Verification Language*. 2005. doi:10.1109/IEEESTD.2005.97972 (https://doi.org/10.1109%2FIEEESTD.2005.97972). ISBN 978-0-7381-4810-6.
- *1800-2009 — IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language*. 2009. doi:10.1109/IEEESTD.2009.5354441 (https://doi.org/10.1109%2FIEEESTD.2009.5354441). ISBN 978-0-7381-6130-3.
- *1800-2012 — IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language*. 2013. doi:10.1109/IEEESTD.2013.6469140 (https://doi.org/10.1109%2F

IEEESTD.2013.6469140). ISBN 978-0-7381-8110-3.
- *1800-2017 — IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language*. 2017. doi:10.1109/IEEESTD.2018.8299595 (https://doi.org/10.1109%2FIEEESTD.2018.8299595). ISBN 978-1-5044-4509-2.
- McGrath, Dylan (2005-11-09). "IEEE approves SystemVerilog, revision of Verilog" (http://www.eetimes.com/news/design/showArticle.jhtml;?articleID=173601060). EE Times. Retrieved 2007-01-31.
- Puneet Kumar (2005-11-09). "System Verilog Tutorial" (http://asicguru.com/System-Verilog-Tutorial/1/3).
- Gopi Krishna (2005-11-09). "SystemVerilog ,SVA,SV DPI Tutorials" (http://www.testbench.in).
- HDVL. "More SystemVerilog Weblinks" (http://hdvl.wordpress.com/category/systemverilog/).
- Spear, Chris, "SystemVerilog for Verification" (https://www.amazon.com/SystemVerilog-Verification-Learning-Testbench-Language/dp/0387765298/ref=sr_1_1?ie=UTF8&s=books&qid=1247578512&sr=8-1) Springer, New York City, NY. ISBN 0-387-76529-8
- Stuart Sutherland, Simon Davidmann, Peter Flake, "SystemVerilog for Design Second Edition: A Guide to Using SystemVerilog for Hardware Design and Modeling" (https://www.amazon.com/SystemVerilog-Design-Second-Hardware-Modeling/dp/0387333991/ref=sr_1_4?ie=UTF8&s=books&qid=1247578512&sr=8-4) Springer, New York City, NY. ISBN 0-387-33399-1
- Ben Cohen, Srinivasan Venkataramanan, Ajeetha Kumari and Lisa Piper [1] (http://SystemVerilog.us) SystemVerilog Assertions Handbook, 4th Edition, 2016- http://SystemVerilog.us
- Ben Cohen Srinivasan Venkataramanan and Ajeetha Kumari [2] (http://SystemVerilog.us) A Pragmatic Approach to VMM Adoption, - http://SystemVerilog.us
- Erik Seligman and Tom Schubert [3] (https://www.amazon.com/Formal-Verification-Essential-Toolkit-Modern-ebook/dp/B012VX1MW8/ref=sr_1_1?ie=UTF8&qid=1451183481&sr=8-1&keywords=erik+seligman+formal+verification) Formal Verification: An Essential Toolkit for Modern VLSI Design, Jul 24, 2015,

External links

IEEE Standard Reference

The most recent SystemVerilog standard documents are available at no cost from IEEExplore (https://ieeexplore.ieee.org/browse/standards/get-program/page/series?id=80).

- 1800-2017 - IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language (https://ieeexplore.ieee.org/document/8299595)

Tutorials

- SystemVerilog Tutorial (http://www.asic-world.com/systemverilog/tutorial.html)
- SystemVerilog Tutorial for Beginners (http://www.project-veripage.com/sv_front.php)

Standards Development

- IEEE P1800 (http://www.eda.org/sv-ieee1800/) – Working group for SystemVerilog
- Sites used before IEEE 1800-2005
  - SystemVerilog official website (http://www.systemverilog.org/)
  - SystemVerilog Technical Committees (http://www.vhdl.org/sv/)

Language Extensions

- Verilog AUTOs (http://www.veripool.org/verilog-mode) – An open source meta-comment system to simplify maintaining Verilog code

Online Tools

- EDA Playground (http://www.edaplayground.com) – Run SystemVerilog from a web browser (free online IDE)
- SVeN (http://sven.xtreme-eda.com) – A SystemVerilog BNF Navigator (current to IEEE 1800-2012)

Other Tools

- SVUnit (http://sourceforge.net/projects/svunit/) – unit test framework for developers writing code in SystemVerilog. Verify SystemVerilog modules, classes and interfaces in isolation.
- sv2v (https://github.com/zachjs/sv2v/) - open-source converter from SystemVerilog to Verilog

---