WIKIPEDIA

Verilog

Verilog, standardized as **IEEE 1364**, is a hardware description language (HDL) used to model electronic systems. It is most commonly used in the design and verification of digital circuits at the register-transfer level of abstraction. It is also used in the verification of analog circuits and mixed-signal circuits, as well as in the design of genetic circuits.[1] In 2009, the Verilog standard (IEEE 1364-2005) was merged into the SystemVerilog standard, creating IEEE Standard 1800-2009. Since then, Verilog is officially part of the SystemVerilog language. The current version is IEEE standard 1800-2017.[2]

Verilog

Paradigm	Structured
First appeared	1984
Stable release	IEEE 1364-2005 / 9 November 2005
Typing discipline	Static, weak
Filename extensions	.v, .vh
Dialects	
Verilog-AMS	
Influenced by	
C, Fortran	
Influenced	
SystemVerilog	
Programmable Logic/Verilog at Wikibooks	

Contents

Overview

Hardware description languages such as Verilog are similar to software programming languages because they include ways of describing the propagation time and signal strengths (sensitivity). There are two types of assignment operators; a blocking assignment (=), and a non-blocking (<=) assignment. The non-blocking assignment allows designers to describe a state-machine update without needing to declare and use temporary storage variables. Since these concepts are part of Verilog's language semantics, designers could quickly write descriptions of large circuits in a relatively compact and concise form. At the time of Verilog's introduction (1984), Verilog represented a tremendous productivity improvement for circuit designers who were already using graphical schematic capture software and specially written software programs to document and simulate electronic circuits.

The designers of Verilog wanted a language with syntax similar to the C programming language, which was already widely used in engineering software development. Like C, Verilog is case-sensitive and has a basic preprocessor (though less sophisticated than that of ANSI C/C++). Its control flow keywords (if/else, for, while, case, etc.) are equivalent, and its operator precedence is compatible with C. Syntactic differences include: required bit-widths for variable declarations, demarcation of procedural blocks (Verilog uses begin/end instead of curly braces {}), and many other minor differences. Verilog requires that variables be given a definite size. In C these sizes are assumed from the 'type' of the variable (for instance an integer type may be 8 bits).

A Verilog design consists of a hierarchy of modules. Modules encapsulate *design hierarchy*, and communicate with other modules through a set of declared input, output, and bidirectional ports. Internally, a module can contain any combination of the following: net/variable declarations (wire, reg, integer, etc.), concurrent and sequential statement blocks, and instances of other modules (sub-hierarchies). Sequential statements are placed inside a begin/end block and executed in sequential order within the block. However, the blocks themselves are executed concurrently, making Verilog a dataflow language.

Verilog's concept of 'wire' consists of both signal values (4-state: "1, 0, floating, undefined") and signal strengths (strong, weak, etc.). This system allows abstract modeling of shared signal lines, where multiple sources drive a common net. When a wire has multiple drivers, the wire's (readable) value is resolved by a function of the source drivers and their strengths.

A subset of statements in the Verilog language are synthesizable. Verilog modules that conform to a synthesizable coding style, known as RTL (register-transfer level), can be physically realized by synthesis software. Synthesis software algorithmically transforms the (abstract) Verilog source into a netlist, a logically equivalent description consisting only of elementary logic primitives (AND, OR, NOT, flip-flops, etc.) that are available in a specific FPGA or VLSI technology. Further manipulations to the netlist ultimately lead to a circuit fabrication blueprint (such as a photo mask set for an ASIC or a bitstream file for an FPGA).

History

Beginning

Verilog was created by Prabhu Goel, Phil Moorby and Chi-Lai Huang and Douglas Warmke between late 1983 and early 1984.[3] Chi-Lai Huang had earlier worked on a hardware description LALSD, a language developed by Professor S.Y.H. Su, for his PhD work.[4] The rights holder for this process, at the time proprietary, was "Automated Integrated Design Systems" (later renamed to Gateway Design Automation in 1985). Gateway Design Automation was purchased by Cadence Design Systems in 1990. Cadence now has full proprietary rights to Gateway's Verilog and the Verilog-XL, the HDL-simulator that would become the de facto standard (of Verilog logic simulators) for the next decade. Originally, Verilog was only intended to describe and allow simulation; the automated synthesis of subsets of the language to physically realizable structures (gates etc.) was developed after the language had achieved widespread usage.

Verilog is a portmanteau of the words "verification" and "logic".[5]

Verilog-95

With the increasing success of VHDL at the time, Cadence decided to make the language available for open standardization. Cadence transferred Verilog into the public domain under the Open Verilog International (http://www.ovi.org/) (OVI) (now known as Accellera) organization. Verilog was later submitted to IEEE and became IEEE Standard 1364-1995, commonly referred to as Verilog-95.

In the same time frame Cadence initiated the creation of Verilog-A to put standards support behind its analog simulator Spectre. Verilog-A was never intended to be a standalone language and is a subset of Verilog-AMS which encompassed Verilog-95.

Verilog 2001

Extensions to Verilog-95 were submitted back to IEEE to cover the deficiencies that users had found in the original Verilog standard. These extensions became IEEE Standard 1364-2001 known as Verilog-2001.

Verilog-2001 is a significant upgrade from Verilog-95. First, it adds explicit support for (2's complement) signed nets and variables. Previously, code authors had to perform signed operations using awkward bit-level manipulations (for example, the carry-out bit of a simple 8-bit addition required an explicit description of the Boolean algebra to determine its correct value). The same function under Verilog-2001 can be more succinctly described by one of the built-in operators: +, -, /, *, >>>. A generate/endgenerate construct (similar to VHDL's generate/endgenerate) allows Verilog-2001 to control instance and statement instantiation through normal decision operators (case/if/else). Using generate/endgenerate, Verilog-2001 can instantiate an array of instances, with control over the connectivity of the individual instances. File I/O has been improved by several new system tasks. And finally, a few syntax additions were introduced to improve code readability (e.g. always, @*, named parameter override, C-style function/task/module header declaration).

Verilog-2001 is the version of Verilog supported by the majority of commercial EDA software packages.

Verilog 2005

Not to be confused with SystemVerilog, *Verilog 2005* (IEEE Standard 1364-2005) consists of minor corrections, spec clarifications, and a few new language features (such as the uwire keyword).

A separate part of the Verilog standard, Verilog-AMS, attempts to integrate analog and mixed signal modeling with traditional Verilog.

SystemVerilog

The advent of hardware verification languages such as OpenVera, and Verisity's e language encouraged the development of Superlog by Co-Design Automation Inc (acquired by Synopsys). The foundations of Superlog and Vera were donated to Accellera, which later became the IEEE standard P1800-2005: SystemVerilog.

SystemVerilog is a superset of Verilog-2005, with many new features and capabilities to aid design verification and design modeling. As of 2009, the SystemVerilog and Verilog language standards were merged into SystemVerilog 2009 (IEEE Standard 1800-2009). The current version is IEEE standard 1800-2017.[6]

Example

A simple example of two flip-flops follows:

```
module toplevel(clock,reset);
  input clock;
  input reset;

  reg flop1;
  reg flop2;

  always @ (posedge reset or posedge clock)
    if (reset)
      begin
        flop1 <= 0;
        flop2 <= 1;
      end
    else
      begin
        flop1 <= flop2;
        flop2 <= flop1;
      end
endmodule
```

The <= operator in Verilog is another aspect of its being a hardware description language as opposed to a normal procedural language. This is known as a "non-blocking" assignment. Its action does not register until after the always block has executed. This means that the order of the assignments is irrelevant and will produce the same result: flop1 and flop2 will swap values every clock.

The other assignment operator = is referred to as a blocking assignment. When = assignment is used, for the purposes of logic, the target variable is updated immediately. In the above example, had the statements used the = blocking operator instead of <=, flop1 and flop2 would not have been swapped. Instead, as in traditional programming, the compiler would understand to simply set flop1 equal to flop2 (and subsequently ignore the redundant logic to set flop2 equal to flop1).

An example counter circuit follows:

```
module Div20x (rst, clk, cet, cep, count, tc);
// TITLE 'Divide-by-20 Counter with enables'
// enable CEP is a clock enable only
// enable CET is a clock enable and
// enables the TC output
// a counter using the Verilog language

parameter size = 5;
parameter length = 20;

input rst; // These inputs/outputs represent
input clk; // connections to the module.
input cet;
input cep;

output [size-1:0] count;
output tc;

reg [size-1:0] count; // Signals assigned
                      // within an always
                      // (or initial)block
                      // must be of type reg

wire tc; // Other signals are of type wire

// The always statement below is a parallel
// execution statement that
// executes any time the signals
// rst or clk transition from low to high

always @ (posedge clk or posedge rst)
  if (rst) // This causes reset of the cntr
    count <= {size{1'b0}};
  else
  if (cet && cep) // Enables both  true
    begin
      if (count == length-1)
        count <= {size{1'b0}};
      else
        count <= count + 1'b1;
    end

// the value of tc is continuously assigned
// the value of the expression
assign tc = (cet && (count == length-1));

endmodule
```

An example of delays:

```
...
reg a, b, c, d;
wire e;
...
always @(b or e)
  begin
    a = b & e;
    b = a | b;
    #5 c = b;
    d = #6 c ^ e;
  end
```

The **always** clause above illustrates the other type of method of use, i.e. it executes whenever any of the entities in the list (the **b** or **e**) changes. When one of these changes, **a** is immediately assigned a new value, and due to the blocking assignment, *b* is assigned a new value afterward (taking into account the new value of **a**). After a delay of 5 time units, **c** is assigned the value of **b** and the value of **c** ^ **e** is tucked away in an invisible store. Then after 6 more time units, **d** is assigned the value that was tucked away.

Signals that are driven from within a process (an initial or always block) must be of type **reg**. Signals that are driven from outside a process must be of type **wire**. The keyword **reg** does not necessarily imply a hardware register.

Definition of constants

The definition of constants in Verilog supports the addition of a width parameter. The basic syntax is:

<Width in bits>'<base letter><number>

Examples:

- 12'h123 — Hexadecimal 123 (using 12 bits)
- 20'd44 — Decimal 44 (using 20 bits — 0 extension is automatic)
- 4'b1010 — Binary 1010 (using 4 bits)
- 6'o77 — Octal 77 (using 6 bits)

Synthesizable constructs

There are several statements in Verilog that have no analog in real hardware, e.g. $display. Consequently, much of the language can not be used to describe hardware. The examples presented here are the classic subset of the language that has a direct mapping to real gates.

```
// Mux examples — Three ways to do the same thing.

// The first example uses continuous assignment
wire out;
assign out = sel ? a : b;

// the second example uses a procedure
// to accomplish the same thing.

reg out;
always @(a or b or sel)
  begin
    case(sel)
      1'b0: out = b;
      1'b1: out = a;
    endcase
  end

// Finally — you can use if/else in a
// procedural structure.
reg out;
always @(a or b or sel)
  if (sel)
    out = a;
  else
    out = b;
```

The next interesting structure is a transparent latch; it will pass the input to the output when the gate signal is set for "pass-through", and captures the input and stores it upon transition of the gate signal to "hold". The output will remain stable regardless of the input signal while the gate is set to "hold". In the example below the "pass-through" level of the gate would be when the value of the if clause is true, i.e. gate = 1. This is read "if gate is true, the din is fed to latch_out continuously." Once the if clause is false, the last value at latch_out will remain and is independent of the value of din.

```
// Transparent latch example

reg latch_out;
always @(gate or din)
 if(gate)
   latch_out = din; // Pass through state
   // Note that the else isn't required here. The variable
   // latch_out will follow the value of din while gate is
   // high. When gate goes low, latch_out will remain constant.
```

The flip-flop is the next significant template; in Verilog, the D-flop is the simplest, and it can be modeled as:

```
reg q;
always @(posedge clk)
  q <= d;
```

The significant thing to notice in the example is the use of the non-blocking assignment. A basic rule of thumb is to use **<=** when there is a **posedge** or **negedge** statement within the always clause.

A variant of the D-flop is one with an asynchronous reset; there is a convention that the reset state will be the first if clause within the statement.

```
reg q;
always @(posedge clk or posedge reset)
  if(reset)
    q <= 0;
  else
    q <= d;
```

The next variant is including both an asynchronous reset and asynchronous set condition; again the convention comes into play, i.e. the reset term is followed by the set term.

```
reg q;
always @(posedge clk or posedge reset or posedge set)
 if(reset)
   q <= 0;
 else
 if(set)
   q <= 1;
 else
   q <= d;
```

Note: If this model is used to model a Set/Reset flip flop then simulation errors can result. Consider the following test sequence of events. 1) reset goes high 2) clk goes high 3) set goes high 4) clk goes high again 5) reset goes low followed by 6) set going low. Assume no setup and hold violations.

In this example the always @ statement would first execute when the rising edge of reset occurs which would place q to a value of 0. The next time the always block executes would be the rising edge of clk which again would keep q at a value of 0. The always block then executes when set goes high which because reset is high forces q to remain at 0. This condition may or may not be correct depending on the actual flip flop. However, this is not the main problem with this model. Notice that when reset goes low, that set is still high. In a real flip flop this will cause the output to go to a 1. However, in this model it will not occur because the always block is triggered by rising edges of set and reset — not levels. A different approach may be necessary for set/reset flip flops.

The final basic variant is one that implements a D-flop with a mux feeding its input. The mux has a d-input and feedback from the flop itself. This allows a gated load function.

```
// Basic structure with an EXPLICIT feedback path
always @(posedge clk)
  if(gate)
    q <= d;
  else
    q <= q; // explicit feedback path

// The more common structure ASSUMES the feedback is present
// This is a safe assumption since this is how the
// hardware compiler will interpret it. This structure
// looks much like a latch. The differences are the
// '''@(posedge clk)''' and the non-blocking '''<='''
//
always @(posedge clk)
  if(gate)
    q <= d; // the "else" mux is "implied"
```

Note that there are no "initial" blocks mentioned in this description. There is a split between FPGA and ASIC synthesis tools on this structure. FPGA tools allow initial blocks where reg values are established instead of using a "reset" signal. ASIC synthesis tools don't support such a statement. The reason is that an FPGA's initial state is something that is downloaded into the memory tables of the FPGA. An ASIC is an actual hardware implementation.

Initial and always

There are two separate ways of declaring a Verilog process. These are the **always** and the **initial** keywords. The **always** keyword indicates a free-running process. The **initial** keyword indicates a process executes exactly once. Both constructs begin execution at simulator time 0, and both execute until the end of the block. Once an **always** block has reached its end, it is rescheduled (again). It is a common misconception to believe that an initial block will execute before an always block. In fact, it is better to think of the **initial**-block as a special-case of the **always**-block, one which terminates after it completes for the first time.

```
//Examples:
initial
  begin
    a = 1; // Assign a value to reg a at time 0
    #1; // Wait 1 time unit
    b = a; // Assign the value of reg a to reg b
  end

always @(a or b) // Any time a or b CHANGE, run the process
begin
  if (a)
    c = b;
  else
    d = ~b;
end // Done with this block, now return to the top (i.e. the @ event-control)

always @(posedge a)// Run whenever reg a has a low to high change
  a <= b;
```

These are the classic uses for these two keywords, but there are two significant additional uses. The most common of these is an **always** keyword without the **@(...)** sensitivity list. It is possible to use always as shown below:

```
always
 begin // Always begins executing at time 0 and NEVER stops
   clk = 0; // Set clk to 0
   #1; // Wait for 1 time unit
   clk = 1; // Set clk to 1
   #1; // Wait 1 time unit
 end // Keeps executing — so continue back at the top of the begin
```

The **always** keyword acts similar to the C language construct **while(1) {..}** in the sense that it will execute forever.

The other interesting exception is the use of the **initial** keyword with the addition of the **forever** keyword.

The example below is functionally identical to the **always** example above.

```
initial forever // Start at time 0 and repeat the begin/end forever
 begin
   clk = 0; // Set clk to 0
   #1; // Wait for 1 time unit
   clk = 1; // Set clk to 1
   #1; // Wait 1 time unit
 end
```

Fork/join

The **fork/join** pair are used by Verilog to create parallel processes. All statements (or blocks) between a fork/join pair begin execution simultaneously upon execution flow hitting the **fork**. Execution continues after the **join** upon completion of the longest running statement or block between the **fork** and **join**.

```
initial
 fork
```

```
    $write("A"); // Print Char A
    $write("B"); // Print Char B
    begin
      #1; // Wait 1 time unit
      $write("C");// Print Char C
    end
  join
```

The way the above is written, it is possible to have either the sequences "ABC" or "BAC" print out. The order of simulation between the first $write and the second $write depends on the simulator implementation, and may purposefully be randomized by the simulator. This allows the simulation to contain both accidental race conditions as well as intentional non-deterministic behavior.

Notice that VHDL cannot dynamically spawn multiple processes like Verilog.[7]

Race conditions

The order of execution isn't always guaranteed within Verilog. This can best be illustrated by a classic example. Consider the code snippet below:

```
initial
  a = 0;

initial
  b = a;

initial
  begin
    #1;
    $display("Value a=%d Value of b=%d",a,b);
  end
```

What will be printed out for the values of a and b? Depending on the order of execution of the initial blocks, it could be zero and zero, or alternately zero and some other arbitrary uninitialized value. The $display statement will always execute after both assignment blocks have completed, due to the #1 delay.

Operators

Note: These operators are *not* shown in order of precedence.

Operator type	Operator symbols	Operation performed
Bitwise	~	Bitwise NOT (1's complement)
	&	Bitwise AND
	|	Bitwise OR
	^	Bitwise XOR
	~^ or ^~	Bitwise XNOR
Logical	!	NOT
	&&	AND
	||	OR
Reduction	&	Reduction AND
	~&	Reduction NAND
	|	Reduction OR
	~|	Reduction NOR
	^	Reduction XOR
	~^ or ^~	Reduction XNOR
Arithmetic	+	Addition
	-	Subtraction
	-	2's complement
	*	Multiplication
	/	Division
	**	Exponentiation (*Verilog-2001)
Relational	>	Greater than
	<	Less than
	>=	Greater than or equal to
	<=	Less than or equal to
	==	Logical equality (bit-value 1'bX is removed from comparison)
	!=	Logical inequality (bit-value 1'bX is removed from comparison)
	===	4-state logical equality (bit-value 1'bX is taken as literal)
	!==	4-state logical inequality (bit-value 1'bX is taken as literal)
Shift	>>	Logical right shift
	<<	Logical left shift
	>>>	Arithmetic right shift (*Verilog-2001)
	<<<	Arithmetic left shift (*Verilog-2001)
Concatenation	{, }	Concatenation
Replication	{n{m}}	Replicate value m for n times
Conditional	? :	Conditional

Four-valued logic

The IEEE 1364 standard defines a four-valued logic with four states: 0, 1, Z (high impedance), and X (unknown logic value). For the competing VHDL, a dedicated standard for multi-valued logic exists as IEEE 1164 with nine levels.[8]

System tasks

System tasks are available to handle simple I/O and various design measurement functions during simulation. All system tasks are prefixed with **$** to distinguish them from user tasks and functions. This section presents a short list of the most frequently used tasks. It is by no means a comprehensive list.

- $display — Print to screen a line followed by an automatic newline.
- $write — Print to screen a line without the newline.
- $swrite — Print to variable a line without the newline.
- $sscanf — Read from variable a format-specified string. (*Verilog-2001)
- $fopen — Open a handle to a file (read or write)
- $fdisplay — Print a line from a file followed by an automatic newline.
- $fwrite — Print to file a line without the newline.
- $fscanf — Read from file a format-specified string. (*Verilog-2001)
- $fclose — Close and release an open file handle.
- $readmemh — Read hex file content into a memory array.
- $readmemb — Read binary file content into a memory array.
- $monitor — Print out all the listed variables when any change value.
- $time — Value of current simulation time.
- $dumpfile — Declare the VCD (Value Change Dump) format output file name.
- $dumpvars — Turn on and dump the variables.
- $dumpports — Turn on and dump the variables in Extended-VCD format.
- $random — Return a random value.

Program Language Interface (PLI)

The PLI provides a programmer with a mechanism to transfer control from Verilog to a program function written in C language. It is officially deprecated by IEEE Std 1364-2005 in favor of the newer Verilog Procedural Interface, which completely replaces the PLI.

The PLI (now VPI) enables Verilog to cooperate with other programs written in the C language such as test harnesses, instruction set simulators of a microcontroller, debuggers, and so on. For example, it provides the C functions `tf_putlongp()` and `tf_getlongp()` which are used to write and read the argument of the current Verilog task or function, respectively.

Simulation software

For information on Verilog simulators, see the list of Verilog simulators.

See also

Additional material

- List of Verilog simulators
- Waveform viewer
- SystemVerilog Direct Programming Interface (DPI)
- Verilog Procedural Interface (VPI)

Similar languages

- VHDL
- SystemC — C++ library providing HDL event-driven semantics
- SystemVerilog
- OpenVera
- e (verification language)
- Property Specification Language
- Chisel, an open-source language built on top of Scala

Verilog generators

References

1. Nielsen AA, Der BS, Shin J, Vaidyanathan P, Paralanov V, Strychalski EA, Ross D, Densmore D, Voigt CA (2016). "Genetic circuit design automation" (https://doi.org/10.1126/science.aac7341). *Science*. **352** (6281): aac7341. doi:10.1126/science.aac7341 (https://doi.org/10.1126%2Fscience.aac7341). PMID 27034378 (https://pubmed.ncbi.nlm.nih.gov/27034378).
2. 1800-2vhhu017 - IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language (http://standards.ieee.org/findstds/standard/1800-2017.html)
3. "Verilog's inventor nabs EDA's Kaufman award" (http://www.eetimes.com/document.asp?doc_id=1157349). *EE Times*. 7 November 2005.
4. Huang, Chi-Lai; Su, S.Y.H. "Approaches for Computer-Aided Logic System Design Using Hardware Description Language". *Proceedings of International Computer Symposium 1980, Taipei, Taiwan, December 1980*. pp. 772–79O. OCLC 696254754 (https://www.worldcat.org/oclc/696254754).
5. "Oral History of Philip Raymond "Phil" Moorby" (http://archive.computerhistory.org/resources/access/text/2013/11/102746653-05-01-acc.pdf) (PDF). Computer History Museum. 22 April 2013. pp. 23–25.
6. 1800-2017 - IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language (http://standards.ieee.org/findstds/standard/1800-2017.html)
7. Cummings, Clifford E. (2003). "SystemVerilog — Is This The Merging of Verilog & VHDL?" (http://www.sunburst-design.com/papers/CummingsSNUG2003Boston_SystemVerilog_VHDL.pdf) (PDF). SNUG Boston 2003.
8. Miller, D. Michael; Thornton, Mitchell A. (2008). *Multiple valued logic: concepts and representations*. Synthesis Lectures on Digital Circuits and Systems. **12**. Morgan & Claypool. ISBN 978-1-59829-190-2.

Notes

- *1364-2005 — IEEE Standard for Verilog Hardware Description Language*. 2006. doi:10.1109/IEEESTD.2006.99495 (https://doi.org/10.1109%2FIEEESTD.2006.99495). ISBN 0-7381-4850-4.

- *1364-2001 — IEEE Standard Verilog Hardware Description Language*. 2001. doi:10.1109/IEEESTD.2001.93352 (https://doi.org/10.1109%2FIEEESTD.2001.93352). ISBN 0-7381-2826-0.
- *61691-4-2004 — IEC/IEEE Behavioural Languages — Part 4: Verilog Hardware Description Language (Adoption of IEEE Std 1364-2001)*. 2004. doi:10.1109/IEEESTD.2004.95753 (https://doi.org/10.1109%2FIEEESTD.2004.95753). ISBN 2-8318-7675-3.
- *1364-1995 — IEEE Standard Hardware Description Language Based on the Verilog(R) Hardware Description Language*. 1996. doi:10.1109/IEEESTD.1996.81542 (https://doi.org/10.1109%2FIEEESTD.1996.81542). ISBN 978-0-7381-3065-1.
- Thomas, Donald E.; Moorby, Phillip R. (2013). *The Verilog® Hardware Description Language* (https://books.google.com/books?id=kGDaBwAAQBAJ&pg=PR2) (3rd ed.). Springer. ISBN 978-1475724646.
- [1] (http://instruct1.cit.cornell.edu/Courses/ece576/Verilog/coding_and_synthesis_with_verilog.pdf) Cornell ECE576 Course illustrating synthesis constructs
- Bergeron, Janick (2012). *Writing Testbenches: Functional Verification of HDL Models* (https://books.google.com/books?id=Zi_jBwAAQBAJ&pg=PR1) (2nd ed.). Springer. ISBN 978-1-4615-0302-6. (The HDL Testbench Bible)

External links

Standards development

- IEEE Std 1364-2005 (http://ieeexplore.ieee.org/servlet/opac?punumber=10779) – The official standard for Verilog 2005 (not free).
- IEEE P1364 (http://www.verilog.com/IEEEVerilog.html) – Working group for Verilog (inactive).
- IEEE P1800 (http://www.eda.org/sv-ieee1800/) – Working group for SystemVerilog (replaces above).
- Verilog syntax (http://www.verilog.com/VerilogBNF.html) – A description of the syntax in Backus-Naur form. This predates the IEEE-1364 standard.
- Verilog-AMS (http://www.verilog.org/verilog-ams) – Accellera mixed signal extensions to Verilog

Language extensions

- Verilog AUTOs (http://www.veripool.org/verilog-mode) — An open-source meta-comment used by industry IP to simplify maintaining Verilog code.