**CHAPTER-5**

**VERILOG**

# 5. VERILOG HDL

**5.1 INTRODUCTION**

In the semiconductor and electronic outline industry, Verilog is an equipment portrayal language (HDL) used to show electronic frameworks. Verilog HDL, not to be mistaken for VHDL (a contending dialect), is most generally utilized as a part of the outline, confirmation, and usage of digital rationale chips at the register-exchange level of reflection. It is likewise utilized as a part of the confirmation of analog and blended sign circuits.

**5.2 Overview**

Equipment portrayal dialects, for example, Verilog contrast from programming dialects on the grounds that they incorporate methods for depicting the proliferation of time and flag conditions (affectability). There are two task administrators, a blocking task (=), and a non-blocking (<=) task. The non-blocking task permits planners to portray a state-machine upgrade without expecting to pronounce and utilize transitory capacity variables (in any broad programming dialect we have to characterize some provisional storage rooms for the operands to be worked on along these lines; those are impermanent capacity variables). Since these ideas are a piece of Verilog's dialect semantics, architects could rapidly compose portrayals of expansive circuits in a generally minimal and succinct structure. At the season of Verilog's presentation (1984), Verilog spoke to a huge efficiency change for circuit originators who were at that point utilizing graphical schematic capture software and uniquely composed programming projects to report and mimic electronic circuits.

The originators of Verilog needed a dialect with grammar like the C programming dialect, which was at that point generally utilized as a part of designing programming advancement. Verilog is case-delicate, has a fundamental preprocessor (however less advanced than that of ANSI C/C++), and proportional control stream watchwords (if/else, for, while, case, and so on.), and perfect administrator priority. Syntactic contrasts incorporate variable affirmation (Verilog requires bit-widths on net/register types [clarification needed]), boundary of procedural squares (start/end rather than wavy props {}), and numerous other minor contrasts.

A Verilog configuration comprises of a chain of importance of modules. Modules epitomize plan order, and speak with different modules through an arrangement of pronounced data, yield, and bidirectional ports. Inside, a module can contain any mix of the accompanying: net/variable affirmations (wire, register, number, and so forth.), simultaneous and successive proclamation squares, and occurrences of different modules (sub-chains of importance). Successive articulations are put inside a start/end piece and executed in consecutive request inside of the square. Be that as it may, the pieces themselves are executed simultaneously, qualifying Verilog as a dataflow dialect.

Verilog's idea of "wire" comprises of both sign qualities (4-state: "1, 0, coasting, indistinct") and qualities (solid, feeble, and so on.). This framework permits conceptual demonstrating of shared sign lines, where numerous sources drive a typical net. At the point when a wire has various drivers, the wire's (lucid) worth is determined by a component of the source drivers and their qualities.

A subset of articulations in the Verilog dialect is synthesizable. Verilog modules that fit in with a synthesizable coding style, known as RTL (register-exchange level), can be physically acknowledged by combination programming. Combination programming algorithmically changes the (unique) Verilog source into a net rundown, a consistently proportionate depiction comprising just of rudimentary rationale primitives (AND, OR, NOT, flip-flops, and so on.) that are accessible in a particular FPGA or VLSI innovation. Further controls to the net rundown at last prompt a circuit manufacture plan, (for example, a photograph cover set for an ASIC or a bit stream record for a FPGA).

**5.3 History**

**Beginning**

Verilog was the first cutting edge equipment portrayal dialect to be developed. It was made by Phil Moorby and Prabhu Goel amid the winter of 1983/1984. The wording for this procedure was "Robotized Integrated Design Systems" (later renamed to Gateway Design Automation in 1985) as an equipment demonstrating dialect. Passage Design Automation was bought by Cadence Design Systems in 1990. Rhythm now has full restrictive rights to Gateway's Verilog and the Verilog-XL, the HDL-test system that would turn into the accepted standard (of Verilog rationale test systems) for the following decade. Initially, Verilog was proposed to portray and permit reenactment; just a short time later was backing for blend included.

**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  (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 dominant flavor of Verilog supported by the majority of commercial EDA software packages.

**Verilog 2005**

Not to be mistaken for System Verilog, Verilog 2005 (IEEE Standard 1364-2005) comprises of minor rectifications, spec illuminations, and a couple of new dialect elements, (for example, the wire watchword).

A different piece of the Verilog standard, Verilog-AMS, endeavors to incorporate simple and blended sign displaying with conventional Verilog.

**5.4 System Verilog**

System Verilog is a superset of Verilog-2005, with numerous new components and capacities to help outline confirmation and configuration demonstrating. Starting 2009, the System Verilog and Verilog dialect models were converged into System Verilog 2009 (IEEE Standard 1800-2009).

The appearance of equipment confirmation dialects, for example, Open Vera, and Verisity's dialect empowered the improvement of Super log by Co-Design Automation Inc. Co-Design Automation Inc was later bought by Synopsys. The establishments of Super log and Vera were given to Accellera, which later turned into the IEEE standard P1800-2005: System Verilog.

In the late 1990s, the Verilog Hardware Description Language (HDL) turned into the most generally utilized dialect for portraying equipment for recreation and combination. On the other hand, the initial two forms institutionalized by the IEEE (1364-1995 and 1364-2001) had just straightforward develops for making tests. As configuration sizes exceeded the check capacities of the dialect, business Hardware Verification Languages (HVL, for example, Open Vera and e were made. Organizations that would not have liked to pay for these devices rather burned through several man-years making their own particular custom devices. This profitability emergency (alongside a comparative one on the configuration side) prompted the making of Accellera, a consortium of EDA organizations and clients who needed to make the up and coming era of Verilog. The gift of the Open-Vera dialect framed the premise for the HVL components of System Verilog. Accellera's objective was met in November 2005 with the selection of the IEEE standard P1800-2005 for System Verilog, IEEE (2005).

The most profitable advantage of System Verilog is that it permits the client to build dependable, repeatable check situations, in a predictable linguistic structure, that can be utilized over various undertakings

A percentage of the run of the mill components of a HVL that recognize it from a Hardware Description Language, for example, Verilog or VHDL are

* Constrained-random stimulus generation
* Functional coverage
* Higher-level structures, especially Object Oriented Programming
* Multi-threading and inter process communication
* Support for HDL types such as Verilog’s 4-state values
* Tight integration with event-simulator for control of the design

There are many other useful features, but these allow you to create test-benches at a higher level of abstraction than you are able to achieve with an HDL or a programming language such as C.

System Verilog provides the best framework to achieve coverage-driven verification (CDV). CDV combines automatic test generation, self-checking test-benches, and coverage metrics to significantly reduce the time spent verifying a design. The purpose of CDV is to:

* Eliminate the effort and time spent creating hundreds of tests.
* Ensure thorough verification using up-front goal setting.
* Receive early error notifications and deploy run-time checking and error analysis to simplify debugging.

**Examples**

**Ex1**: A hello world program looks like this:

module main;

initial

begin

$display("Hello world!");

$finish;

end

endmodule

**Ex2**: 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 "<=" administrator in Verilog is another part of its being an equipment depiction dialect rather than a typical procedural dialect. This is known as a "non-blocking" task. Its activity doesn't enlist until the following clock cycle. This implies the request of the assignments are immaterial and will deliver the same result: flop1 and flop2 will swap values each clock.

The other task administrator, "=", is alluded to as a blocking task. Whenever "=" task is utilized, for the reasons of rationale, the objective variable is upgraded promptly. In the above illustration, had the announcements utilized the "=" blocking administrator rather than "<=", flop1 and flop2 would not have been swapped. Rather, as in customary programming, the compiler would comprehend to just set flop1 equivalent to flop2 (and along these lines overlook the repetitive rationale to set flop2 equivalent to flop1.)

**Ex3**: 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

**Ex4**: 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 dependably statement above represents the other sort of system for use, i.e. the dependably statement executes at whatever time any of the substances in the rundown change, i.e. the b or e change. At the point when one of these progressions, promptly an is doled out another quality, and because of the blocking task b is alloted another esteem a while later (considering the new estimation of an.) After a deferral of 5 time units, c is relegated the estimation of b and the estimation of c ^ e is concealed in an undetectable store. At that point after 6 additional time units, d is appointed the worth that was concealed.

Signals that are driven from inside of a procedure (a beginning or dependably square) must be of sort reg. Signals that are driven from outside a procedure must be of sort wire. The catchphrase register does not as a matter of course suggest an equipment register

**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)

**5.5 Synthesizable Constructs**

There are several statements in Verilog that have no analog in real hardware, e.g. $display. Consequently, much of the language cannot 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.

**Ex5:** // Transparent latch example

reg out;

always @(gate or din)

if(gate)

out = din; // Pass through state

// Note that the else isn't required here. The variable

// out will follow the value of din while gate is high.

// When gate goes low, 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 utilized to display a Set/Reset flip tumble then recreation blunders can come about. Consider the accompanying test grouping of occasions. 1) reset goes high 2) clk goes high 3) set goes high 4) clk goes high again 5) reset goes low took after by 6) set going low. Expect no setup and hold infringement.

In this sample the dependably @ explanation would first execute when the rising edge of reset happens which would put q to an estimation of 0. Whenever the dependably piece executes would be the rising edge of clk which again would keep q at an estimation of 0. The dependably piece then executes when set goes high which in light of the fact that reset is high powers q to stay at 0. This condition could possibly be right contingent upon the real flip lemon. Be that as it may, this is not the fundamental issue with this model. Notice that when reset goes low, that set is still high. In a genuine flip tumble this will make the yield go to a 1. Nonetheless, in this model it won't happen on the grounds that the dependably piece is activated by rising edges of set and reset - not levels. An alternate methodology may be vital for set/reset flip lemon.

Note that there are no "beginning" pieces said in this portrayal. There is a split in the middle of FPGA and ASIC union apparatuses on this structure. FPGA devices permit introductory pieces where reg qualities are built up as opposed to utilizing a "reset" signal. ASIC combination devices don't backing such an announcement. The reason is that a FPGA's starting state is something that is downloaded into the memory tables of the FPGA. An ASIC is a genuine equipment usage.

**5.6 Initial Vs Always:**

There are two separate methods for proclaiming a Verilog process. These are the dependably and the beginning watchwords. The dependably watchword demonstrates a free-running procedure. The beginning watchword demonstrates a procedure executes precisely once. Both develops start execution at test system time 0, and both execute until the end of the piece. Once a dependably piece has come to its end, it is rescheduled (once more). It is a typical misguided judgment to trust that a beginning piece will execute before a dependably square. Truth be told, it is ideal to think about the beginning piece as an exceptional instance of the dependably square, one which ends after it finishes surprisingly.

//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 alwayskeyword 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" 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.

**5.7 Race Condition**

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=%b Value of b=%b",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.

**5.8 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 |

**5.9 System Tasks**

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

* $display - Print to screen a line followed by an automatic newline.
* $write - Write 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 - Write to file a line followed by an automatic newline.
* $fwrite - Write 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