Preface

This is a brief summary of the syntax and semantics of the Verilog Hardware Description Language. The summary is not intended at being an exhaustive list of all the constructs and is not meant to be complete. This reference guide also lists constructs that can be synthesized. For any clarifications and to resolve ambiguities please refer to the Verilog Language Reference Manual, Copyright © 1993 by Open Verilog International, Inc. and synthesis vendors Verilog HDL Reference Manuals.

In addition to the OVI Language Reference Manual, for further examples and explanation of the Verilog HDL, the following textbook is recommended: Digital Design and Synthesis With Verilog HDL, Eli Sternheim, Rajvir Singh, Rajeev Madhavan and Yatin Trivedi, Copyright © 1993 by Automata Publishing Company.

Rajeev Madhavan

Use and Copyright

Copyright (c) 1994, 1995 Rajeev Madhavan
Copyright (c) 1994, 1995 Automata Publishing Company

Permission to use, copy and distribute this book for any purpose is hereby granted without fee, provided that

(i) the above copyright notices and this permission notice appear in all copies, and

(ii) the names of Rajeev Madhavan, Automata Publishing and AMBIT Design Systems may not be used in any advertising or publicity relating to this book without the specific, prior written permission of Rajeev Madhavan, Automata Publishing and AMBIT Design Systems.

THE BOOK IS PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EXPRESS, IMPLIED OR OTHERWISE, INCLUDING WITHOUT LIMITATION, ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

IN NO EVENT SHALL RAJEEV MADHAVAN OR AUTOMATA PUBLISHING OR AMBIT DESIGN SYSTEMS BE LIABLE FOR ANY SPECIAL, INCIDENTAL, INDIRECT OR CONSEQUENTIAL DAMAGES OF ANY KIND, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, PROFITS, WHETHER OR NOT ADVISED OF THE POSSIBILITY OF DAMAGE, AND ON ANY THEORY OF LIABILITY, ARISING OUT OF OR IN CONNECTION WITH THE USE OF THIS BOOK.
1.0 Lexical Elements

The language is case sensitive and all the keywords are lower case. White space, namely, spaces, tabs and new-lines are ignored. Verilog has two types of comments:

1. One line comments start with // and end at the end of the line
2. Multi-line comments start with /* and end with */

Variable names have to start with an alphabetic character or underscore followed by alphanumeric or underscore characters. The only exception to this are the system tasks and functions which start with a dollar sign. Escaped identifiers (identifier whose first characters is a backslash (\)) permit non alphanumeric characters in Verilog name. The escaped name includes all the characters following the backslash until the first white space character.

1.1 Integer Literals

<table>
<thead>
<tr>
<th>Binary literal</th>
<th>2'b1Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>Octal literal</td>
<td>2'017</td>
</tr>
<tr>
<td>Decimal literal</td>
<td>9 or 'd9</td>
</tr>
<tr>
<td>Hexadecimal literal</td>
<td>3'h189</td>
</tr>
</tbody>
</table>

Integer literals can have underscores embedded in them for improved readability. For example,

Decimal literal 24_000

1.2 Data Types

The values z and Z stand for high impedance, and x and X stand for uninitialized variables or nets with conflicting drivers. String symbols are enclosed within double quotes ("string") and cannot span multiple lines. Real number literals can be either in fixed notation or in scientific notation.

**Real and Integer Variables example**

real a, b, c ; // a,b,c to be real

integer j, k ; // integer variable
integer i[1:32] ; // array of integer variables
Time, registers and variable usage

A register stores its value from one assignment to the next and is used to model data storage elements.

Nets correspond to physical wires that connect instances. The default range of a wire or reg is one bit. Nets do not store values and have to be continuously driven. If a net has multiple drivers (for example two gate outputs are tied together), then the net value is resolved according to its type.

**Net types**

<table>
<thead>
<tr>
<th>wire</th>
<th>tri</th>
</tr>
</thead>
<tbody>
<tr>
<td>wand</td>
<td>triand</td>
</tr>
<tr>
<td>wor</td>
<td>trior</td>
</tr>
<tr>
<td>tri0</td>
<td>tril</td>
</tr>
<tr>
<td>supply0</td>
<td>supply1</td>
</tr>
<tr>
<td>trireg</td>
<td></td>
</tr>
</tbody>
</table>

For a wire, if all the drivers have the same value then the wire resolves to this value. If all the drivers except one have a value of z then the wire resolves to the non z value. If two or more non z drivers have different drive strength, then the wire resolves to the stronger driver. If two drivers of equal strength have different values, then the
wire resolves to $x$. A `trireg` net behaves like a `wire` except that when all the drivers of the net are in high impedance ($z$) state, then the net retains its last driven value. `trireg`'s are used to model capacitive networks.

```vhdl
wire net1;
/* wire and tri have same functionality. tri is used for multiple drive internal wire */
trireg (medium) capacitor;
/* small, medium, weak are used for charge strength modeling */
```

A `wand` net or `triand` net operates as a wired `and` (wand), and a `wor` net or `trior` net operates as a wired `or` (wor). `tri0` and `tri1` nets model nets with resistive pulldown or pullup devices on them. When a `tri0` net is not driven, then its value is 0. When a `tri1` net is not driven, then its value is 1. `supply0` and `supply1` model nets that are connected to the ground or power supply.

```vhdl
wand net2 ; // wired-and
wor net3 ; // wired-or
triand [4:0] net4 ; // multiple drive wand
trior net5 ; // multiple drive wor
tri0 net6 ;
tri1 net7 ;
supply0 gnd ; // logic 0 supply wire
supply1 vcc ; // logic 1 supply wire
```

Memories are declared using register statements with the address range specified as in the following example,

```vhdl
reg [15:0] mem16X512 [0:511];
// 16-bit by 512 word memory
// mem16X512[4] addresses word 4
// the order lsb:msb or msb:lsb is not important
```

The keyword `scalared` allows access to bits and parts of a bus and `vectored` allows the vector to be modified only collectively.

```vhdl
wire vectored [5:0] neta;
/* a 6-bit vectored net */
tri1 vectored [5:0] netb;
/* a 6-bit vectored tri1 */
```

### 3.0 Compiler Directives
Verilog has compiler directives which affect the processing of the input.
files. The directives start with a grave accent (` '') followed by some keyword. A directive takes effect from the point that it appears in the file until either the end of all the files, or until another directive that cancels the effect of the first one is encountered. For example,

```
'define OPCODEADD 00010
```

This defines a macro named OPCODEADD. When the text `OPCODEADD` appears in the text, then it is replaced by 00010. Verilog macros are simple text substitutions and do not permit arguments.

```
`ifdef SYNTH <Verilog code> 'endif
```

If “SYNTH” is a defined macro, then the Verilog code until `endif is inserted for the next processing phase. If “SYNTH” is not defined macro then the code is discarded.

```
`include <Verilog file>
```

The code in `<Verilog file>` is inserted for the next processing phase. Other standard compiler directives are listed below:

```
'resetall - resets all compiler directives to default values
'define - text-macro substitution
'timescale 1ns / 10ps - specifies time unit/precision
'ifdef, 'else, 'endif - conditional compilation
'include - file inclusion
'signed, 'unsigned - operator selection (OVI 2.0 only)
'celldefine, 'endcelldefine - library modules
'default_nettype wire - default net types
'unconnected_drive pull0|pull1, 'nounconnected_drive - pullup or down unconnected ports
'protect and 'endprotect - encryption capability
'protected and 'endprotected - encryption capability
'expand_vectornets, 'noexpand_vectornets, 'autoexpand_vectornets - vector expansion options
'remove_gatename, 'noremove_gatenames - remove gate names for more than one instance
'remove_netname, 'noremove_netnames - remove net names for more than one instance
```

4.0 System Tasks and Functions

System tasks are tool specific tasks and functions.

```
$display( "Example of using function" );
/* display to screen */
$monitor($time, "a=$b, clk = $b, add=$h", a, clk, add); // monitor signals
$setuphold( posedge clk, datain, setup, hold); // setup and hold checks
```
A list of standard system tasks and functions are listed below:

|$display$, $write$ - utility to display information
|$fdisplay$, $fwrite$ - write to file
|$strobe$, $fstrobe$ - display/write simulation data
|$monitor$, $fmonitor$ - monitor, display/write information to file
|$time$, $realtime$ - current simulation time
|$finish$ - exit the simulator
|$stop$ - stop the simulator
|$setup$ - setup timing check
|$hold$, $width$ - hold/width timing check
|$setuphold$ - combines hold and setup
|$readmemb$/, $readmemh$ - read stimulus patterns into memory
|$readmemb$/, $rreadmemb$ - load data into memory
|$getpattern$ - fast processing of stimulus patterns
|$history$ - print command history
|$save$, $restart$, $incsave$ - saving, restarting, incremental saving
|$scale$ - scaling timeunits from another module
|$scope$ - descend to a particular hierarchy level
|$showscopes$ - complete list of named blocks, tasks, modules...
|$showvars$ - show variables at scope

5.0 Reserved Keywords

The following lists the reserved words of Verilog hardware description language, as of OVLRML 2.0.

<table>
<thead>
<tr>
<th>and</th>
<th>always</th>
<th>assign</th>
<th>attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin</td>
<td>buf</td>
<td>bufif0</td>
<td>bufif1</td>
</tr>
<tr>
<td>case</td>
<td>cmos</td>
<td>deassign</td>
<td>default</td>
</tr>
<tr>
<td>defparam</td>
<td>disable</td>
<td>else</td>
<td>endattribute</td>
</tr>
<tr>
<td>end</td>
<td>endcase</td>
<td>endfunction</td>
<td>endprimitive</td>
</tr>
<tr>
<td>endmodule</td>
<td>endtable</td>
<td>endtask</td>
<td>event</td>
</tr>
<tr>
<td>for</td>
<td>force</td>
<td>forever</td>
<td>fork</td>
</tr>
<tr>
<td>function</td>
<td>highz0</td>
<td>highz1</td>
<td>if</td>
</tr>
<tr>
<td>initial</td>
<td>inout</td>
<td>input</td>
<td>integer</td>
</tr>
<tr>
<td>join</td>
<td>large</td>
<td>medium</td>
<td>module</td>
</tr>
<tr>
<td>nodf0</td>
<td>negedge</td>
<td>nor</td>
<td>not</td>
</tr>
<tr>
<td>notif1</td>
<td>notif1</td>
<td>nmos</td>
<td>or</td>
</tr>
<tr>
<td>output</td>
<td>parameter</td>
<td>pmos</td>
<td>posedge</td>
</tr>
<tr>
<td>primitive</td>
<td>pulldown</td>
<td>pullup</td>
<td>pull0</td>
</tr>
<tr>
<td>pull1</td>
<td>rcmos</td>
<td>reg</td>
<td>release</td>
</tr>
<tr>
<td>repeat</td>
<td>rcmos</td>
<td>rmos</td>
<td>rtran</td>
</tr>
<tr>
<td>rrtranif0</td>
<td>rrtranif1</td>
<td>scalared</td>
<td>small</td>
</tr>
<tr>
<td>specify</td>
<td>specparam</td>
<td>strong0</td>
<td>strong1</td>
</tr>
<tr>
<td>supply0</td>
<td>supply1</td>
<td>table</td>
<td>task</td>
</tr>
<tr>
<td>tran</td>
<td>tranif0</td>
<td>tranif1</td>
<td>time</td>
</tr>
<tr>
<td>tri</td>
<td>triand</td>
<td>trior</td>
<td>trireg</td>
</tr>
<tr>
<td>tri0</td>
<td>tril</td>
<td>vectored</td>
<td>wait</td>
</tr>
<tr>
<td>wand</td>
<td>weak0</td>
<td>weak1</td>
<td>while</td>
</tr>
<tr>
<td>wire</td>
<td>wor</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
6.0 Structures and Hierarchy

Hierarchical HDL structures are achieved by defining modules and instantiating modules. Nested module definitions (i.e. one module definition within another) are not permitted.

6.1 Module Declarations

The module name must be unique and no other module or primitive can have the same name. The port list is optional. A module without a port list or with an empty port list is typically a top level module. A macro-module is a module with a flattened hierarchy and is used by some simulators for efficiency.

module definition example

```verilog
module dff (q,qb,clk,d,rst);
    input clk,d,rst ; // input signals
    output q,qb ; // output definition

    //inout for bidirectionals
    // Net type declarations
    wire dl,dbl ;

    // parameter value assignment
    parameter delay1 = 3,
        delay2 = delay1 + 1; // delay2
        // shows parameter dependance

    /* Hierarchy primitive instantiation, port connection in this section is by
    ordered list */

    nand #delay1 n1(cf,dl,cbf),
        n2(cbf,clk,cf,rst);
    nand #delay2 n3(dl,d,dbl,rst),
        n4(dbl,dl,clk,cbf),
        n5(q,cbf,qb),
        n6(qb,dbl,q,rst);

    /***** for debugging model initial begin
    #500 force dff_lab.rst = 1 ;
    #550 release dff_lab.rst;
    // upward path referencing
    end ******/

endmodule
```
6.2 User Defined Primitive (UDP) Declarations

The UDP’s are used to augment the gate primitives and are defined by truth tables. Instances of UDP’s can be used in the same way as gate primitives. There are 2 types of primitives:

1. Sequential UDP’s permit initialization of output terminals, which are declared to be of reg type and they store values. Level-sensitive entries take precedence over edge-sensitive declarations. An input logic state Z is interpreted as an X. Similarly, only 0, 1, X or - (unchanged) logic values are permitted on the output.

2. Combinational UDP’s do not store values and cannot be initialized.

The following additional abbreviations are permitted in UDP declarations.

```verilog
module dff_lab;
    reg data, rst;
    // Connecting ports by name (map)
    dff d1 (.qb(outb), .q(out),
            .clk(clk), .d(data), .rst(rst));
    // overriding module parameters
    defparam
dff_lab.dff.n1.delay1 = 5,
dff_lab.dff.n2.delay2 = 6;
    // full-path referencing is used
    // over-riding by using #(8,9) delay1=8..

dff d2 #(8,9) (outc, outd, clk, outb, rst);
    // clock generator
    always clk = #10 ~clk;
    // stimulus ... contd

initial begin: stimuli // named block stimulus
    clk = 1; data = 1; rst = 0;
    #20 rst = 1;
    #20 data = 0;
    #600 $finish;
end

initial // hierarchy: downward path referencing
begin
    #100 force dff.n2.rst = 0;
    #200 release dff.n2.rst;
end
endmodule
```
<table>
<thead>
<tr>
<th>Logic/state Representation/transition</th>
<th>Abbreviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>don’t care (0, 1 or X)</td>
<td>?</td>
</tr>
<tr>
<td>Transitions from logic x to logic y (xy).</td>
<td>(xy)</td>
</tr>
<tr>
<td>(01), (10), (0x), (1x), (x1), (x0) (??)</td>
<td></td>
</tr>
<tr>
<td>Transition from (01)</td>
<td>R or r</td>
</tr>
<tr>
<td>Transition from (10)</td>
<td>F or f</td>
</tr>
<tr>
<td>(01), (0X), (X1): positive transition</td>
<td>P or p</td>
</tr>
<tr>
<td>(10), (1x), (x0): negative transition</td>
<td>N or n</td>
</tr>
<tr>
<td>Any transition</td>
<td>* or (??)</td>
</tr>
<tr>
<td>binary don’t care (0, l)</td>
<td>B or b</td>
</tr>
</tbody>
</table>

**Combinational UDP's example**

```verilog
// 3 to 1 multiplexor with 2 select
primitive mux32 (Y, in1, in2, in3, s1, s2);
input in1, in2, in3, s1, s2;
output Y;

table
//in1 in2 in3 s1 s2 Y
0 ? ? 0 0 : 0 ;
1 ? ? 0 0 : 1 ;
? 0 ? 1 0 : 0 ;
? 1 ? 1 0 : 1 ;
? ? 0 ? 1 : 0 ;
? ? 1 ? 1 : 1 ;
0 0 ? ? 0 : 0 ;
1 1 ? ? 0 : 1 ;
0 ? 0 0 ? : 0 ;
1 ? 1 0 ? : 1 ;
? 0 0 1 ? : 0 ;
? 1 1 1 ? : 1 ;
endtable
endprimitive
```
Sequential Level Sensitive UDP’s example

// latch with async reset
primitive latch (q, clock, reset, data);
  input clock, reset, data;
  output q;
  reg q;

  initial q = 'b1; // initialization

  table
    // clock reset data q q+
    0 0 0 : ? : 0 ;
    1 0 ? : ? : - ;
    0 0 1 : ? : 1 ;

  endtable
endprimitive

Sequential Edge Sensitive UDP’s example

// edge triggered D Flip Flop with active high, // async set and reset
primitive dff (QN, D, CP, R, S);
  output QN;
  input D, CP, R, S;
  reg QN;

  table
    // D CP R S : Qtn : Qtn+1
    1 (01) 0 0 : ? : 0 ;
    1 (01) 0 x : ? : 0 ;
    ? ? 0 x : 0 : 0 ;
    0 (01) 0 0 : ? : 1 ; // clocked data
    0 (01) x 0 : ? : 1 ; // pessimism
    ? ? x 0 : 1 : 1 ; // pessimism
    1 (x1) 0 0 : 0 : 0 ;
    0 (x1) 0 0 : 1 : 1 ;
    1 (0x) 0 0 : 0 : 0 ;
    0 (0x) 0 0 : 1 : 1 ;
    ? ? 0 1 : ? : 0 ; // asynchronous set
    ? n 0 0 : ? : - ;

  endtable
endprimitive

9
7.0 Expressions and Operators

Arithmetic and logical operators are used to build expressions. Expressions perform operation on one or more operands, the operands being vectored or scalared nets, registers, bit-selects, part selects, function calls or concatenations thereof.

- **Unary Expression**
  \[ \text{<operator> <operand>} \]
  \[ a = !b; \]

- **Binary and Other Expressions**
  \[ \text{<operand> <operator> <operand>} \]
  \[ \text{if (a < b ) // if (<expression>)} \]
  \[ \{c,d\} = a + b ; \]
  \[ // \text{concatenate and add operator} \]

- Parentheses can be used to change the precedence of operators. For example, \((a+b) * c)\)

**Operator precedence**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>+, -, !, ~ (unary)</td>
<td>Highest</td>
</tr>
<tr>
<td>*, /</td>
<td></td>
</tr>
<tr>
<td>+, - (binary)</td>
<td></td>
</tr>
<tr>
<td>&lt;&lt;, &gt;&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;, &lt;=, &gt;, &gt;=</td>
<td></td>
</tr>
<tr>
<td>=, ==, !=</td>
<td></td>
</tr>
<tr>
<td>===, !==</td>
<td></td>
</tr>
<tr>
<td>&amp;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td></td>
</tr>
<tr>
<td>?(:</td>
<td>Lowest</td>
</tr>
</tbody>
</table>

10
• All operators associate left to right, except for the ternary operator “?:” which associates from right to left.

### Relational Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;</code></td>
<td>a &lt; b // is a less than b? // return 1-bit true/false</td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td>a &gt; b // is a greater than b?</td>
</tr>
<tr>
<td><code>&gt;=</code></td>
<td>a &gt;= b // is a greater than or // equal to b</td>
</tr>
<tr>
<td><code>&lt;=</code></td>
<td>a &lt;= b // is a less than or // equal to b</td>
</tr>
</tbody>
</table>

### Arithmetic Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>*</code></td>
<td>c = a * b ; // multiply a with b</td>
</tr>
<tr>
<td><code>/</code></td>
<td>c = a / b ; // int divide a by b</td>
</tr>
<tr>
<td><code>+</code></td>
<td>sum = a + b ; // add a and b</td>
</tr>
<tr>
<td><code>-</code></td>
<td>diff = a - b ; // subtract b // from a</td>
</tr>
<tr>
<td><code>%</code></td>
<td>amodb = a % b ; // a mod(b)</td>
</tr>
</tbody>
</table>

### Logical Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&amp;&amp;</code></td>
<td>a &amp;&amp; b ; // is a and b true? // returns 1-bit true/false</td>
</tr>
<tr>
<td>`</td>
<td></td>
</tr>
<tr>
<td><code>!</code></td>
<td>if (a) ; // if a is not true c = b ; // assign b to c</td>
</tr>
</tbody>
</table>
### Equality and Identity Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>c = a ; // assign a to c</td>
</tr>
<tr>
<td>==</td>
<td>c == a ; /* is c equal to a returns 1-bit true/false applies for 1 or 0, logic equality, using X or Z operands returns always false 'hx == 'h5 returns 0 */</td>
</tr>
<tr>
<td>!=</td>
<td>c != a ; // is c not equal to // a, retuns 1-bit true/ // false logic equality</td>
</tr>
<tr>
<td>===</td>
<td>a === b ; // is a identical to // b {includes 0, 1, x, z} // 'hx === 'h5 returns 0</td>
</tr>
<tr>
<td>!==</td>
<td>a !== b ; // is a not identical to b returns 1- // bit true/false</td>
</tr>
</tbody>
</table>

### Unary, Bitwise and Reduction Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Unary plus &amp; arithmetic(binary) addition</td>
</tr>
<tr>
<td>-</td>
<td>Unary negation &amp; arithmetic (binary) subtraction</td>
</tr>
<tr>
<td>&amp;</td>
<td>b = &amp;a ; // AND all bits of a</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>b = ^a ; // Exclusive or all bits of a</td>
</tr>
<tr>
<td>~&amp;, ~</td>
<td>, ~^</td>
</tr>
<tr>
<td>~&amp;, ~</td>
<td>, ~^</td>
</tr>
<tr>
<td>~^</td>
<td>bit-wise NAND, NOR, EX-NOR</td>
</tr>
</tbody>
</table>

```
Shift Operators and other Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;&lt;</td>
<td>a &lt;&lt; 1 ; // shift left a by 1-bit</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>a &gt;&gt; 1 ; // shift right a by 1</td>
</tr>
<tr>
<td>?:</td>
<td>c = sel ? a : b ; /* if sel is true c = a, else c = b , ?: ternary operator */</td>
</tr>
<tr>
<td>{}</td>
<td>(co, sum) = a + b + ci ; /* add a, b, ci assign the overflow to co and the result to sum: operator is called concatenation */</td>
</tr>
<tr>
<td>{{{}}}</td>
<td>b = {3{a}} /* replicate a 3 times, equivalent to {a, a, a} */</td>
</tr>
</tbody>
</table>

### 7.1 Parallel Expressions

fork ... join are used for concurrent expression assignments.

```verilog
initial
begin: block
fork
    // This waits for the first event a or b to occur
    @a disable block ;
    @b disable block ;

    // reset at absolute time 20
    #20 reset = 1 ;
    // data at absolute time 100
    #100 data = 0 ;
    // data at absolute time 120
    #120 data = 1 ;
join
end
```

### 7.2 Conditional Statements

The most commonly used conditional statement is the if, if ... else ... conditions. The statement occurs if the expressions controlling the if statement evaluates to true.
if .. else ...conditions example

always @(rst)// simple if -else
  if (rst)
    // procedural assignment
    q = 0;
  else // remove the above continuous assign
    deassign q;
always @(WRITE or READ or STATUS)
begin
  // if - else - if
  if (!WRITE) begin
    out = oldvalue;
  end
  else if (!STATUS) begin
    q = newstatus;
    STATUS = hold;
  end
  else if (!READ) begin
    out = newvalue;
  end
end

module d2X8 (select, out); // priority encode
  input [0:2] select;
  output [0:7] out;
  reg [0:7] out;
  always @(select) begin
    out = 0;
    case (select)
      0: out[0] = 1;
      1: out[1] = 1;
      2: out[2] = 1;
      3: out[3] = 1;
      4: out[4] = 1;
      5: out[5] = 1;
      6: out[6] = 1;
      7: out[7] = 1;
    endcase
  end
endmodule

case, casex, casez: case statements are used for switching between multiple selections (if (case1) ... else if (case2) ... else ...). If there are multiple matches only the first is evaluated. casez treats high impedance values as don’t care’s and casex treats both unknown and high-impedance as don’t care’s.
7.3 Looping Statements

forever, for, while and repeat loops example

```verilog
casex (state)
// treats both x and z as don't care
// during comparison : 3'b01z, 3'b01x, 3'b011
// ... match case 3'b01x
3'b01x: fsm = 0;
3'b0xx: fsm = 1;
default: begin
// default matches all other occurrences
   fsm = 1;
   next_state = 3'b011;
end
endcase

casez (state)
// treats z as don't care during comparison :
// 3'b11z, 3'b1zz, ... match 3'b1??
3'b1??: fsm = 0; // if MSB is 1, matches 3?b1?
3'b01?: fsm = 1;
default: $display("wrong state")
endcase

forever
// should be used with disable or timing control
 @(posedge clock) {co, sum} = a + b + ci;

for (i = 0 ; i < 7 ; i=i+1)
   memory[i] = 0; // initialize to 0

for (i = 0 ; i <= bit-width ; i=i+1)
// multiplier using shift left and add
   if (a[i]) out = out + ( b << (i-1) );

repeat(bit-width) begin
   if (a[0]) out = b + out;
   b = b << 1; // multiplier using
   a = a << 1; // shift left and add
end

while(delay) begin @(posedge clk);
   ldlang = oldldlang;
   delay = delay - 1;
end
```
8.0 Named Blocks, Disabling Blocks

Named blocks are used to create hierarchy within modules and can be used to group a collection of assignments or expressions. The `disable` statement is used to disable or de-activate any named block, tasks or modules. Named blocks, tasks can be accessed by full or reference hierarchy paths (example `dff_lab.stimuli`). Named blocks can have local variables.

*Named blocks and disable statement example*

```verilog
initial forever @ (posedge reset)
    disable MAIN ; // disable named block
    // tasks, modules can also be disabled

always begin: MAIN // defining named blocks
    if (!qfull) begin
        #30 recv(new, newdata) ; // call task
        if (new) begin
            q[head] = newdata ;
            head = head + 1 ; // queue
        end
    end
    else
        disable recv ;
end // MAIN
```

9.0 Tasks and Functions

Tasks and functions permit the grouping of common procedures and then executing these procedures from different places. Arguments are passed in the form of input/inout values and all calls to functions and tasks share variables. The differences between tasks and functions are:

<table>
<thead>
<tr>
<th></th>
<th>Tasks</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Permits time control</td>
<td>Execute in one simulation time</td>
<td></td>
</tr>
<tr>
<td>Can have zero or more arguments</td>
<td></td>
<td>Require at least one input</td>
</tr>
<tr>
<td>Does not return value, assigns value to outputs</td>
<td>Execute a single value, no special output declarations required</td>
<td></td>
</tr>
<tr>
<td>Can have output arguments, permits #, @, -&gt;, wait, task calls.</td>
<td>Execute a single value, no special output declarations required</td>
<td>Does not permit outputs, #, @, -&gt;, wait, task calls.</td>
</tr>
</tbody>
</table>
// task Example

task recv;
output valid;
output [9:0] data;
begin
valid = inreg;
if (valid) begin
ackin = 1;
data = qin;
wait(inreg);
ackin = 0;
end
end

// task instantiation
always begin: MAIN //named definition
if (!qfull) begin
recv(new, newdata); // call task
if (new) begin
q[head] = newdata;
head = head + 1;
end
end else disable recv;
end // MAIN

module foo2 (cs, in1, in2, ns);
input [1:0] cs;
input in1, in2;
output [1:0] ns;
function [1:0] generate_next_state;
input[1:0] current_state;
input input1, input2;
reg [1:0] next_state;
// input1 causes 0->1 transition
// input2 causes 1->2 transition
// 2->0 illegal and unknown states go to 0
begin
  case (current_state)
    2'h0 : next_state = input1 ? 2'h1 : 2'h0;
    2'h1 : next_state = input2 ? 2'h2 : 2'h1;
    2'h2 : next_state = 2'h0;
    default: next_state = 2'h0;
  endcase
generate_next_state = next_state;
end
endfunction // generate_next_state
assign ns = generate_next_state(cs, in1, in2);
endmodule
10.0 Continuous Assignments

Continuous assignments imply that whenever any change on the RHS of the assignment occurs, it is evaluated and assigned to the LHS. These assignments thus drive both vector and scalar values onto nets. Continuous assignments always implement combinational logic (possibly with delays). The driving strengths of a continuous assignment can be specified by the user on the net types.

- Continuous assignment on declaration

```verbatim
/* since only one net15 declaration exists in a given module only one such declarative continuous assignment per signal is allowd */
wire #10 (atrong1, pull0) net15 = enable ;
/* delay of 10 for continous assignment with strengths of logic 1 as strong1 and logic 0 as pull0 */
assign #10 net15 = enable ;
assign (weak1, strong0) {s,c} = a + b ;
```

- Continuous assignment on already declared nets

```verbatim
module dff (q,qb,clk,d,rst);
output q, qb;
input d, rst, clk;
reg q, qb, temp;
always
#1 qb = ~q ; // procedural assignment
always @(rst)
// procedural assignment with triggers
if (rst) assign q = temp;
else deassign q;
always @(posedge clk)
    temp = d;
endmodule
```

11.0 Procedural Assignments

Assignments to register data types may occur within always, initial, task and functions. These expressions are controlled by triggers which cause the assignments to evaluate. The variables to which the expressions are assigned must be made of bit-select or part-select or whole element of a reg, integer, real or time. These triggers can be controlled by loops, if, else... constructs. assign and deassign are used for procedural assignments and to remove the continuous assignments.

```verbatim
module dff (q,qb,clk,d,rst);
    output q, qb;
    input d, rst, clk;
    reg q, qb, temp;
    always
        #1 qb = ~q ; // procedural assignment
    always @(rst)
        // procedural assignment with triggers
        if (rst) assign q = temp;
        else deassign q;
    always @(posedge clk)
        temp = d;
endmodule
```

/* since only one net15 declaration exists in a given module only one such declarative continuous assignment per signal is allowed */
wire #10 (atrong1, pull0) net15 = enable ;
/* delay of 10 for continous assignment with strengths of logic 1 as strong1 and logic 0 as pull0 */
assign #10 net15 = enable ;
assign (weak1, strong0) {s,c} = a + b ;
force and release are also procedural assignments. However, they can force or release values on net data types and registers.

11.1 Blocking Assignment

```verilog
module adder {a, b, ci, co, sum, clk) ;
   input a, b, ci, clk ;
   output co, sum ;
   reg co, sum ;
   always @(posedge clk) // edge control
   // assign co, sum with previous value of a,b,ci
   {co,sum} = #10 a + b + ci ;
endmodule
```

11.2 Non-Blocking Assignment

Allows scheduling of assignments without blocking the procedural flow. Blocking assignments allow timing control which are delays, whereas, non-blocking assignments permit timing control which can be delays or event control. The non-blocking assignment is used to avoid race conditions and can model RTL assignments.

```verilog
/* assume a = 10, b= 20 c = 30 d = 40 at start of block */
always @(posedge clk)
   begin:block
      a <= #10 b ;
      b <= #10 c ;
      c <= #10 d ;
   end
/* at end of block + 10 time units, a = 20, b = 30, c = 40 */
```

12.0 Gate Types, MOS and Bidirectional Switches

Gate declarations permit the user to instantiate different gate-types and assign drive-strengths to the logic values and also any delays

```verilog
<gate-declaration> ::= <component>
   <drive_strength>? <delay>? <gate_instance>
   ,?<gate_instance..>> ;
```
<table>
<thead>
<tr>
<th>Gate Types</th>
<th>Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gates</td>
<td>Allows strengths</td>
</tr>
<tr>
<td></td>
<td>and, nand, or, nor, xor, xnor</td>
</tr>
<tr>
<td></td>
<td>buf, not</td>
</tr>
<tr>
<td>Three State Drivers</td>
<td>Allows strengths</td>
</tr>
<tr>
<td></td>
<td>buif0, buif1</td>
</tr>
<tr>
<td></td>
<td>notif0, notif1</td>
</tr>
<tr>
<td>MOS Switches</td>
<td>No strengths</td>
</tr>
<tr>
<td></td>
<td>nmos, pmos, cmos, rnmos, rpmos, rcmos</td>
</tr>
<tr>
<td>Bi-directional switches</td>
<td>No strengths, non resistive</td>
</tr>
<tr>
<td></td>
<td>tran, tranif0, tranif1</td>
</tr>
<tr>
<td></td>
<td>rtran, rtranif0, rtranif1</td>
</tr>
<tr>
<td></td>
<td>Allows strengths</td>
</tr>
<tr>
<td></td>
<td>pulup</td>
</tr>
<tr>
<td></td>
<td>pulldown</td>
</tr>
</tbody>
</table>

**Gates, switch types, and their instantiations**

```verilog
// Gate level instantiations
nor (highz1, strong0) #(2:3:5) (out, in1, in2);
// instantiates a nor gate with out
// strength of highz1 (for 1) and
// strong0 for 0 #(2:3:5) is the
// min:typ:max delay
pullup1 (strong1) net1;
// instantiates a logic high pullup
mos (out, data, ncontrol, pcontrol);
// MOS devices
```

**Gate level instantiation example**

```verilog
// Gate level instantiations
nor (highz1, strong0) #(2:3:5) (out, in1, in2);
// instantiates a nor gate with out
// strength of highz1 (for 1) and
// strong0 for 0 #(2:3:5) is the
// min:typ:max delay
pullup1 (strong1) net1;
// instantiates a logic high pullup
mos (out, data, ncontrol, pcontrol);
// MOS devices
```
The following strength definitions exist:

- 4 drive strengths (supply, strong, pull, weak)
- 3 capacitor strengths (large, medium, small)
- 1 high impedance state highz

The drive strengths for each of the output signals are:

- Strength of an output signal with logic value 1
  supply1, strong1, pull1, large1, weak1, highz1
- Strength of an output signal with logic value 0
  supply0, strong0, pull0, large0, weak0, highz0

<table>
<thead>
<tr>
<th>Logic 0</th>
<th>Logic 1</th>
<th>Strength</th>
</tr>
</thead>
<tbody>
<tr>
<td>supply0</td>
<td>supply1</td>
<td>Su1</td>
</tr>
<tr>
<td>strong0</td>
<td>strong1</td>
<td>St1</td>
</tr>
<tr>
<td>pull0</td>
<td>pull1</td>
<td>Pul</td>
</tr>
<tr>
<td>large</td>
<td>large</td>
<td>La1</td>
</tr>
<tr>
<td>weak0</td>
<td>weak1</td>
<td>We1</td>
</tr>
<tr>
<td>medium</td>
<td>medium</td>
<td>Me1</td>
</tr>
<tr>
<td>small</td>
<td>small</td>
<td>Sm1</td>
</tr>
<tr>
<td>highz0</td>
<td>highz1</td>
<td>HiZ0</td>
</tr>
</tbody>
</table>

12.1 Gate Delays

The delays allow the modeling of rise time, fall time and turn-off delays for the gates. Each of these delay types may be in the min:typ:max format. The order of the delays are #(trise, tfall, tturn-off). For example,

```
    (out, a, b);
```
For `trireg`, the decay of the capacitive network is modeled using the rise-time delay, fall-time delay and charge-decay. For example,

```
trireg (large) #(0,1,9) capacitor
  // charge strength is large
  // decay with tr=0, tf=1, tdecay=9
```

### 13.0 Specify Blocks

A `specify` block is used to specify timing information for the module in which the specify block is used. Specparams are used to declare delay constants, much like regular parameters inside a module, but unlike module parameters they cannot be overridden. Paths are used to declare time delays between inputs and outputs.

**Delay Model**

<table>
<thead>
<tr>
<th>#(delay)</th>
<th>min:typ:max delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>#(delay, delay)</td>
<td>rise-time delay, fall-time delay, each delay can be with min:typ:max</td>
</tr>
<tr>
<td>#(delay, delay, delay)</td>
<td>rise-time delay, fall-time delay and turn-off delay, each min:typ:max</td>
</tr>
</tbody>
</table>

```
specify // similar to defparam, used for timing
  specparam delay1 = 25.0, delay2 = 24.0;
  // edge sensitive delays -- some simulators
  // do not support this
  (posedge clock) => (out1 ;; in1) = (delay1, delay2) ;
  // conditional delays
  if (OPCODE == 3'h4) (in1, in2 *> out1) = (delay1, delay2) ;
  // +: implies edge-sensitive +ve polarity
  // -: implies edge sensitive -ve polarity
  // *> implies multiple paths
  // level sensitive delays
  if (clock) (in1, in2 *> out1, out2) = 30 ;
  // setuphold
  $setuphold(posedge clock &&& reset,
    in1 &&& reset, 3:5:6, 2:3:6);
  (reset *> out1, out2) = (2:3:5,3:4:5);
endspecify
```
Verilog

Synthesis Constructs

The following is a set of Verilog constructs that are supported by most synthesis tools at the time of this writing. To prevent variations in supported synthesis constructs from tool to tool, this is the least common denominator of supported constructs. Tool reference guides cover specific constructs.

14.0 Verilog Synthesis Constructs

Since it is very difficult for the synthesis tool to find hardware with exact delays, all absolute and relative time declarations are ignored by the tools. Also, all signals are assumed to be of maximum strength (strength 7). Boolean operations on X and Z are not permitted. The constructs are classified as

- Fully supported constructs — Constructs that are supported as defined in the Verilog Language Reference Manual
- Partially supported — Constructs supported with restrictions on them
- Ignored constructs — Constructs that are ignored by the synthesis tool
- Unsupported constructs — Constructs which if used, may cause the synthesis tool to not accept the Verilog input or may cause different results between synthesis and simulation.

14.1 Fully Supported Constructs

```
<module instantiation, with named and positional notations>
<integer data types, with all bases>
<identifiers>
<subranges and slices on right-hand side of assignment>
<continuous assignments>
>>, <<, ?, :, {}
assign (procedural and declarative), begin, end case, casex, casez, endcase
default
```
disable
function, endfunction
if, else, else if
input, output, inout
wire, wand, wor, tri
integer, reg
macromodule, module
parameter
supply0, supply1
task, endtask

14.2 Partially Supported Constructs

<table>
<thead>
<tr>
<th>Construct</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>* , / , %</td>
<td>when both operands constants, or 2nd operand power of 2.</td>
</tr>
<tr>
<td>always</td>
<td>only edge-triggered events.</td>
</tr>
<tr>
<td>for</td>
<td>bounded by static variables: only use “+” or “-” to index.</td>
</tr>
<tr>
<td>posedge, negedge</td>
<td>only with always @ .</td>
</tr>
<tr>
<td>primitive, endprimitive, table, endtable</td>
<td>Combinational and edge-sensitive user defined primitives are often supported.</td>
</tr>
<tr>
<td>&lt;=</td>
<td>limitations on usage with blocking assignment.</td>
</tr>
<tr>
<td>and, nand, or, nor, xor, xnor, buf, not, bufif0, bufif1, notif0, notif1</td>
<td>gate types supported without X or Z constructs</td>
</tr>
<tr>
<td>!, &amp;&amp;,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
14.3 Ignored Constructs

- `<intra-assignment timing controls>`
- `<delay specifications>`
- `scalared`, `vectored`
- `small`, `large`, `medium`
- `specify`
- `time` (some tools treat these as integers)
- `weak1`, `weak0`, `highz0`, `highz1`, `pull0`, `pull1`
- `$keyword` (some tools use these to set synthesis constraints)
- `wait` (some tools support `wait` with a bounded condition)

14.4 Unsupported Constructs

- `<assignment with variable used as bit select on LHS of assignment>`
- `<global variables>`
- `===`, `!==`
- `cmos`, `nmos`, `rcmos`, `rnmos`, `pmos`, `rpmos`
- `deassign`
- `defparam`
- `event`
- `force`
- `fork`, `join`
- `forever`, `while`
- `initial`
- `pullup`, `pulldown`
- `release`
- `repeat`
- `rtran`, `tran`, `tranif0`, `tranif1`, `rtranif0`, `rtranif1`
- `table`, `endtable`, `primitive`, `endprimitive`
## Symbols

- `$display`, `$write` 5
- `$fdisplay`, `$fwrite` 5
- `$finish` 5
- `$getpattern` 5
- `$history` 5
- `$hold`, `$width` 5
- `$monitor`, `$monitor` 5
- `$readmemb`, `$readmemh` 5
- `$save`, `$restart`, `$incsave` 5
- `$scale` 5
- `$scope`, `$showscopes` 5
- `$setup`, `$setuphold` 5
- `$showvars` 5
- `$sreadmemb/$sreadmemh` 5
- `$stop` 5
- `$strobe`, `$fstrobe` 5
- `$time`, `$realtime` 5
- `/* */` 1
- `//` 1
- `'autoexpand_vectornets` 4
- `'celldefine`, `'endcelldefine` 4
- `'default_nettype` 4
- `'define` 4
- `'expand_vectornets` 4
- `'noexpand_vectornets` 4
- `'ifdef`, `'else`, `'endif` 4
- `'include` 4
- `'nounconnected_drive` 4
- `'protect`, `'endprotect` 4
- `'protected`, `'endprotected` 4
- `'remove_gatename` 4
- `'noremove_gatename` 4
- `'remove_netname` 4
- `'noremove_netname` 4
- `'resetall` 4
- `'signed`, `'unsigned` 4
- `'timescale` 4
- `'unconnected_drive` 4

## C

- case 14
- casex 14
- casez 14
- compiler directives 3
- continuous assignments 18

## D

- delays 21
- disable 16

## E

- Equality Operators 12
- Escaped identifiers 1
- Expressions 10

## F

- for 15
- forever 15
- fork ... join 13
- Fully Supported Synthesis Constructs 23
- function 16

## G

- Gate declaration 19
- gate-types 19

## I

- if, if ... else 13
- Integer literals 1
- Identity Operators 12

## L

- Logical Operators 11

## M

- Memories 3
- module 6

## N

- Named blocks 16
- Nets 2
- non-blocking assignments 19

---

**Quick Reference for Verilog HDL**
<table>
<thead>
<tr>
<th>O</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operator precedence 10</td>
<td>vectored 3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partially Supported Synthesis Constructs 24</td>
<td>wait 16</td>
</tr>
<tr>
<td>procedural assignments 18</td>
<td>wand 3</td>
</tr>
<tr>
<td>pulldown 3</td>
<td>while 15</td>
</tr>
<tr>
<td>pullup 3</td>
<td>wire 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>reg, register 2</td>
<td>x, X 1</td>
</tr>
<tr>
<td>Relational Operators 11</td>
<td></td>
</tr>
<tr>
<td>repeat 15</td>
<td></td>
</tr>
<tr>
<td>reserved words 5</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>scaled 3</td>
<td>z, Z 1</td>
</tr>
<tr>
<td>Sequential edge sensitive UDP 9</td>
<td></td>
</tr>
<tr>
<td>Sequential level sensitive UDP 9</td>
<td></td>
</tr>
<tr>
<td>Shift, other Operators 13</td>
<td></td>
</tr>
<tr>
<td>specify block 22</td>
<td></td>
</tr>
<tr>
<td>specparam 22</td>
<td></td>
</tr>
<tr>
<td>String symbols 1</td>
<td></td>
</tr>
<tr>
<td>supply0 3</td>
<td></td>
</tr>
<tr>
<td>supply1 3</td>
<td></td>
</tr>
<tr>
<td>switch types 20</td>
<td></td>
</tr>
<tr>
<td>Synthesis Constructs 23</td>
<td></td>
</tr>
<tr>
<td>Synthesis Ignored Constructs 25</td>
<td></td>
</tr>
<tr>
<td>Synthesis Unsupported Constructs 25</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>task 16</td>
<td>UDP 7</td>
</tr>
<tr>
<td>tri0 3</td>
<td>Unary Expression 10</td>
</tr>
<tr>
<td>tri1 3</td>
<td>Unary, Bitwise and Reduction Operators 12</td>
</tr>
<tr>
<td>triand 3</td>
<td></td>
</tr>
<tr>
<td>trior 3</td>
<td></td>
</tr>
<tr>
<td>trireg 3</td>
<td></td>
</tr>
</tbody>
</table>
Verilog HDL Publications Order Form
Automata Publishing Company
1072 S. Saratoga Sunnyvale Rd., Bldg. A107, Ste 325,
San Jose CA-95129. U.S.A
Phone: 408-255-0705 Fax: 408-253-7916

Verilog Publications:
Publication 1. Digital Design and Synthesis with Verilog HDL
Publication 2. Digital Design and Synthesis with Verilog HDL+
Source diskette + Quick Reference for Verilog HDL

| Name: __________________________ | Title: __________________________ |
| Company: __________________________ | Address: __________________________ |
| City: __________________________ | Zip: __________________________ |
| State: __________________________ | Ph: __________________________ |
| P.O Number if any: __________________________ |
| Charge my Visa/MC/AmExp. # __________________________ |
| Expires on: __________________________ |

<table>
<thead>
<tr>
<th>Publication</th>
<th>1 (US$)</th>
<th>2 (US$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quantity</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Price per book (see below)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shipping (see below)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Salex Tax (CA residents only, @current rate)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total amount due</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For large volume discounts contact Automata Publishing Company

<table>
<thead>
<tr>
<th>Publication Qty-Price/copy</th>
<th>1 (US$)</th>
<th>2 (US$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-4</td>
<td>59.95</td>
<td>65.95</td>
</tr>
<tr>
<td>5-9</td>
<td>54.95</td>
<td>60.95</td>
</tr>
<tr>
<td>10-19</td>
<td>49.95</td>
<td>54.95</td>
</tr>
<tr>
<td>20-44</td>
<td>44.95</td>
<td>49.95</td>
</tr>
<tr>
<td>45-99</td>
<td>39.95</td>
<td>44.45</td>
</tr>
<tr>
<td>100-500</td>
<td>34.95</td>
<td>39.00</td>
</tr>
<tr>
<td>Shipping/copy</td>
<td>3.00</td>
<td>3.00</td>
</tr>
</tbody>
</table>
Quick Reference
for
Verilog® HDL

Rajeev Madhavan

This is a brief summary of the syntax and semantics of the Verilog Hardware Description Language. The reference guide describes all the Verilog HDL constructs and also lists the Register-Transfer Level subset of the Verilog HDL which is used by the existing synthesis tools. Examples are used to illustrate constructs in the Verilog HDL.