Summary of Key Verilog Features (IEEE 1364) *

Module
Encapsulates functionality; may be nested to any depth

```
module module_name (list of ports);
    // Declarations
    Port modes: input, output, inout identifier;
    Nets (e.g., wire A[3:0];)
    Register variable (e.g., reg B[31:0];)
    Constants: (e.g., parameter size=8;)
    Named events
    Continuous assignments (e.g., assign sum = A + B;)
    Behaviors always (cyclic), initial (single-pass)
    specify ... endspecify
    function ... endfunction
    task ... endtask
    // Instantiations
    primitives
    modules
endmodule
```

Multi-Input Primitives

```
and (out, in1, in2, ..., inN);
nand (out, in1, in2, ..., inN);
or (out, in1, in2, ..., inN);
nor (out, in1, in2, ..., inN);
xor (out, in1, in2, ..., inN);
xnor (out, in1, in2, ..., inN);
```

Three-State Multioutput Primitives

```
buf (out1, out2, ..., outN, in); // buffer
not (out1, out2, ..., outN, in); // inverter
```

Three-State Multioutput Primitives

```
bufif0 (out, in, control);
bufif1 (out, in, control);
notif0 (out, in, control);
notif1 (out, in, control);
```

Pullups and Pulldowns

```
pullup (out_y);
pulldown (out_y);
```

Propagation Delays

```
Single delay:
and #3 G1 (y, a, b, c);
Rise/fall:
and #(3,6) G2 (y, a, b, c);
Rise/fall/turnoff:
bufif0 #(3,6,5) (y, x_in, en);
Min:typ:Max:
bufif1 #(3:4:5, 4:5:6, 7:8:9) (y, x_in, en);
```

Command line options for single delay value simulation: +maxdelays, +typdelays, +mindelays

Concurrent Behavioral Statements

May execute a level-sensitive assignment of value to a net (keyword: assign), or
may execute the statements of a cyclic (keyword: always) or single-pass (keyword: initial) behavior. The statements execute sequentially, subject to level-sensitive or edge-sensitive event control expressions.

```
Syntax
assign net_name = [expression];
always begin [procedural statements] end
initial begin [procedural statements] end
```

Cyclic(always) and single-pass (initial) behaviors may be level sensitive and/or
edge sensitive

```
Edge sensitive
always @ (posedge clock)
q <= data;
```
Level sensitive
always @ (enable or data) 
q <= data;

Data Types: Nets and Registers
Nets: Establish structural connectivity between instantiated primitives and/or modules; may be target of a continuous assignment; e.g., wire, tri, wand, wor. Value is determined during simulation by the driver of the net; e.g., a primitive or a continuous assignment.
Example:
wire Y = A + B;

Registers: Store information and retain value until reassigned. Value is determined by an assignment made by a procedural statement. Value is retained until a new assignment is made; e.g., reg, integer, real, realtime, time.
Example:
always @ (posedge clock)
  if (reset) q_out <= 0;
  else q_out <= data_in;

Procedural Statements
Describe logic abstractly; statements execute sequentially to assign value to variables.
if (expression_is_true) statement_1; else statement_2;
case (case_expression)
  case_item_0: statement_0;
  case_item_1: begin
    statement_1_0;
    statement_1_1;
  end
  default: statement;
endcase
for (conditions) statement;
repeat constant_expression statement;
while (expression_is_true) statement;
forever statement;
fork statements join //execute in parallel

Assignments
Continuous: Continuously assigns the value of an expression to a net.
Procedural (Blocking): Uses the = operator; executes statements sequentially; a statement cannot execute until the preceding statement completes execution. Value is assigned immediately.
Procedural (Nonblocking): Uses the != operator; executes statements concurrently, independent of the order in which they are listed. Values are assigned concurrently.
Procedural (Continuous): assign ... deassign overrides procedural assignments to a net.
force ... release overrides all other assignments to a net or a register

Operators
[] {} {{}} concatenation | bitwise or
+-* / arithmetic ` ` bitwise exclusive-or
% modulus & or bitwise equivalence
>=<= relational && reduction and
! logical negation & reduction and
&& logical and | reduction or
I logical or ~ reduction nor
== logical equality ~ or reduction exclusive-or
=== case equality || reduction or
!= logical inequality_reduction exclusive-nor
!?: conditional
& bitwise and ?? logical and
!|| reduction or
& bitwise and ~ bitwise negation
&& reduction nor
~ bitwise negation ?: conditional
| bitwise or
|| reduction or
| bitwise or
^ bitwise exclusive-or
^ bitwise exclusive-nor
^ bitwise equivalence
?? right shift
^ reduction exclusive-or
^ reduction exclusive-nor
!?: conditional
<< left shift
?: conditional
>> right shift

Specify Block
Example: Module Path Delays
specify // specparam declarations (min:typ:max)
specparam t_r = 3:4:5, t_f = 4:5:6;
((A,B) => Y) = (t_r, t_f); // full
(Bus_1 => Bus_1) = (t_r, t_f); // parallel
if (state == S0) (a,b => y) = 2; // state dep
(posedge clk => (y :: d_in)) = (3,4); // edge
endspecify

Example: Timing Checks
specify specparam t_setup = 3:4:5, t_hold = 4:5:6;
$setup (data, posedge clock, t_setup);
$hold (posedge clock, data, t_hold);
endspecify
Memory
Declares an array of words.

Example: Memory declaration and readout

```verilog
module memory_read_display();
    reg[31:0] mem_array[1:1024];
    integer k;
    initial begin
        // read contents of mem_array from
        // a file in hex format
        $readmemh("mem_contents.dat", mem_array);
        // display contents of mem_array
        for (k = 1; k <= 1024; k = k + 1)
            $display("word[%d]=%h", k, mem_array[k]);
end
endmodule
```

Concurrency and Race Conditions
Indeterminate outcomes result from race conditions when multiple behaviors use the procedural assignment operator (=) to reference (read) and assign value to the same variable at the same time.

Example (with race):
```verilog
always @(posedge clock) a = b;
always @(posedge clock) b = a;
```
Use nonblocking assignment operator (<=) to eliminate race conditions; assignments are independent of the order of the behaviors and the order of the statements.

Example:
```verilog
always @(posedge clock) a <= b;
always @(posedge clock) b <= a;
```
Use the procedural assignment operator (=) in level-sensitive behaviors and the nonblocking assignment operator (<=) in edge-sensitive behaviors to avoid race conditions in sequential machines.

Example (without race):
```verilog
always @(posedge clock)
    state <= next_state;
always @(state or inputs)
    case(state)
        state_0: begin
            next_state = state_5;
        endcase
```

Functions
May invoke another function; may not invoke a task. Executes with zero delay time.
May not contain delay control (#), event control (@), and wait.
Must have at least one input argument.
May not have output or input arguments.
Function name serves as a placeholder for a single returned value.

Example:
```verilog
function [4:0] adder;
    input [3:0] a, b;
    adder = a + b;
endfunction
```

Tasks
May invoke other tasks and other functions.
May contain delay control (#), event control (@), and wait.
May have zero or more arguments having mode input, output, inout.
Passes values by its arguments.

Example:
```verilog
adder (sum, a, b);
... task adder;
    output [4:0] sum;
    input [3:0] a, b;
    sum = a + b;
endtask
```

Selected Compiler Directives
```verilog
'image width = 16;
'image include .../project_header.v
'image timescale = 100ns/1ns // time_units / precision
'image ifdef ... 'else ... 'endif
```
Example:

```verilog
module or_model (y, a, b);
output y;
input a, b;
'def cont_assign
  // uses continuous assignment
  assign y = a | b;
'else
  or G1 (y, a, b); // uses primitive
'endif
endmodule
```

Simulation Output

```verilog
$display("string of info %d", variable);
integer K;
initial K = $open("output_file");
always @ (event.control expression) // dump data
begin
  $fdisplay (K, "%h", data[7:0]);
  ...
  $fclose ("output_file");
  $monitor ($time, "out_1=%b, out_2=%b", out_1, out_2);
  $monitor (K, "some value=%h", address[15:0]);
  $monitoron;
  $monitoroff;
end
```

Simulation Data Control

```verilog
initial begin
  // dump simulation data into my_data.dump:
  $dumpfile("my_data.dump");
  // dump all signals:
  $dumpvars;
  // dump variables in module top:
  $dumpvars(1, top);
  // dump variables 2 levels below top.mod1:
  $dumpvars(2, top.mod1);
  // stop dump after 1000 time units:
  #1000 $dumpoff;
  // start/restart dump after 500 time units:
  #500 $dumpon;
  // suspend simulation:
  $stop;
end
```

// terminate simulation after 1000 time units:
#1000 $finish;
end

Parameter Redefinition

In-line (instance-based)

```verilog
module Something();
  parameter size = 4;
  parameter width = 8;
  . . .
endmodule
```

Indirect (hierarchical dereferencing):

```verilog
module Annotate();
  . . .
  defparam . Something . width = 16;
  . . .
endmodule
```

Constructs to Avoid in Synthesis

- time, real, realtime variables
- named event
- triand, trior, tri0, tri1 nets
- vector ranges for integers
- single-pass (initial) behavior
- assign ... deassign procedural continuous assignment
- force ... release procedural continuous assignment
- fork ... join block (parallel activity flow)
- defparam parameter substitution
- Operators: Modulus(%) and Division(/), except for division by 2
- ==, !=, ===, !===
- Primitives: pullup, pulldown, tranif0, tranif1, rtran, rtranif0, rtranif1
- Hierarchical pathnames
- Compiler directives

This material is used while creating the current document:

2. "Advanced Digital Design with the Verilog HDL" by Michael D. Ciletti