Verilog® Quick Reference Card

1. Module

```
module module_name (list of ports);
      input / output / inout declarations
      net / reg declarations
      integer declarations
      parameter declarations

      gate / switch instatnces
      hierarchical instances
      parallel statements
endmodule
```

2. Parallel Statements

Following statements start executing simultaneously inside module

```
initial begin
      {sequential statements}
end
always begin
      {sequential statements}
end
assign wire_name = [expression]};
```

3. Basic Data Types

a. Nets

e.g. **wire, wand, tri, wor**

- Continuously driven
- Gets new value when driver changes
- LHS of continuous assignment

```
tri [15:0] data;
        // unconditional
assign data[15:0] = data_in;
        // conditional
assign data[15:0] = enable ? data_in : 16'bz;
```

b. Registers

e.g. **reg**

- Represents storage
- Always stores last assigned value
- LHS of an assignment in procedural block.

```
reg signal;
@(posedge clock) signal = 1'b1;
        // positive edge
@(reset) signal = 1'b0;  // event (both edges)
```

4. Sequential Statements

Given below are some examples instead of BNF type of definitions.

```
■  if (reset == 0) begin
              data = 8'b00;
   end
■  case (operator)
              2'd0 : z = x + y;
              2'd1 : z = x – y;
              2'd2 : z = x * y;
              default : $display ("Invalid Operation");
   endcase
■  initial begin          // 50 MHz clock
              clock = 0;
              forever #10 clock = ~clock;
   end                    // precission 1 ns
■  repeat (2) @(posedge clk) data;
■  bus <= repeat (5) @ (posedge clk) data
              // evaluate data when the assignment is
              // encountered and assign to bus after 5
              // clocks.
■  repeat (flag) begin // looping
              .... action ....
■  while (i < 10) begin
              .... action ....
   end
■  for (i = 0; i < 9; i = i + 1) begin
              .... action ....
   end
■  wait (!oe) #5 data = d_in;
■  @(negedge clock) q = d;
■  begin                  // finishes at time #25
              #10 x = y;
              #15 a = b;
   end
■  fork
              #10 x = y;
              #15 a = b;
   join
```

5. Gate Primitives

and	(out, in_1, ..., in_n);	**nand**	(out, in_1, ..., in_n);
or	(out, in_1, ..., in_n);	**nor**	(out, in_1, ..., in_n);
xor	(out, in_1, ..., in_n);	**xnor**	(out, in_1, ..., in_n);
buf	(out_1, ..., out_n, in);	**not**	(out_1, ..., out_n, in);
bufif0	(out, in, control);	**bufif1**	(out, in, control);
notif0	(out, in, control);	**notif0**	(out, in, control);
pullup	(out);	**pulldown**	(out);

6. Delays

Single delay	: and #5 my_and (...);
Rise/ Fall	: and #(5, 7) my_and (...);
Rise/ Fall / Transport	: bufif1 #(10, 15, 5) my_buf (...);
All delays as min:typ:max	: or #(4:5:6, 6:7:8) my_or (...);

Compiler options for delays

+maxdelays, +typdelays(default), +mindelays

e.g. **verilog** +maxdelays test.v

7. Declarations

{}, {{}}	concatenation
+ - * /	arithmetic
%	modulus
> >= < <=	relational
!	logical negation
&&	logical and
||	logical or
==	logical equality
!=	logical inequality
===	case equality
!==	case inequality
~	bit-wise negation
&	bit-wise and
|	bit-wise inclusive or
^	bit-wise exclusive or
^~ or ~^	bit-wise equivalence
&	reduction and
~&	reduction nand
|	reduction or
~|	reduction nor
^	reduction xor
~^ or ^~	reduction xnor
<<	left shift
>>	right shift
? :	condition
or	event or

8. Attributes

```
specify
            // specparam declarations (min:typ:max)
    specparam t_setup = 8:9:10, t_hold = 11:12:13;
            // timing constraints checks
    $setup (data, posedge clock, t_rise);
    $hold (posedge clear, data, t_hold);
            // simple pin to pin path delay
    (a => out) = 9; // => means parallel connection
            // edge sensitive pin to pin path delay
    (posedge clock => (out +: in)) = (10, 8);
            // state dependent pin to pin path delay
    if (state_a == 2'b01) (a, b *> out) = 15;
            // *> means full connection
endspecify
```

9. Memory Instantiation

```
module mem_test;
reg [7: 0] memory [0: 10];          // memory declaration
integer i;
initial begin
    // reading the memory content file
    $readmemh ("contents.dat", memory);
    // display contents of initialized memory
   for (i = 0; i < 9, i = i + 1)
        $display ("Memory [%d] = %h", i, memory[i]);
end
endmodule
```

"contents.dat" contains

```
@02         ab        da
@06         00        01
```

- This simple memory model can be used for feeding input data values to simulation environment.
- $readmemb can be used for feeding binary values from contents file.

10. Blocking and Non-blocking Statements

```
// These blocking statements exhibit race condition.
always @(posedge clock)
    a = b;
always @(posedge clock)
    b = a;
// This Non-blocking statement removes above race
// condition and gives true swapping operation
always @(posedge clock)
    a <= b;
always @(posedge clock)
    b <= a;
```

11. Functions and Tasks

Function

- A function can enable another function but not another task.
- Functions always execute in 0 simulation time.
- Functions must not contain any delay, event, or timing control statements.
- Functions must have at least one input argument. They can have more than one input.
- Functions always return a single value. They cannot have output or inout argument.

```
e.g.
....
parity = calc_parity (addr);
....
function calc_parity;
input [31: 0] address;
begin
    calc_parity = ^address;
end
endfunction
```

Task

- A task can enable other tasks and functions.
- Tasks may execute in non-zero simulation time.
- Tasks may contain delay, event, or timing control statements.
- Tasks may have zero or more arguments of type input, output, or inout.
- Tasks do not return with a value, but can pass multiple values through output and inout arguments.

```
....
Cycle_read (read_in, oe_in, data, addr);
....
task Cycle_read;
input           read, oe;  // notice the order
output [7: 0]   data;
input [15: 0]   address;
begin
    #10 read_pin = read;
    #05 oe_pin = oe;
    data = some_funtion (address);
end
endtask
```

12. Commonly Used Compiler Directives

```
'define word_size 32
'include ../header.v
'timescale 100ns/1ns     // ref_time_unit / precision
'ifdef, 'else, 'endif
e.g.
module and_op (a, b, c);
    output a;
    input b, c;
    'ifdef behavioral
            wire a = b & c;
    'else
            and (a, b, c);
    'endif
endmodule
```

13. Observing Outputs

```
$display ("Value of variable is %d", var);
integer flag;
initial flag = $fopen ("out_file");
always @(....);  // dump data in text file
    $fdisplay (flag, "%h", data [7: 0]);
    ........
    $fclose ("out_file");
end
$monitor ($time, "a = %b and b = %b", clock, reset);
$fmonitor (flag, "value = %h", add [15: 0]);
$monitoron;
$monitoroff;
```

14. Simulation Control

```
initial begin
    $dumpfile ("my.dump");        // dump in this file
    $dumpvars;                    // dump all signals
    $dumpvars (1, top);
            // dump variables in module instance top
    $dumpvars (2, top.m1);  // dump 2 levels below top.m1
    #1000 dumpoff;                // stop dump
    #500 dumpon;                  // start / restart dump
     $stop;               // stop for interaction
     #1000 $finish;       // come out of simulation
 end
```

15. Language Constructs Not Supported By Most Synthesis Tools

Declarations and Definitions

- **time** declaration
- **event** declaration
- **triand, trior, tri1, tri0,** and **trireg** net types
- Ranges and arrays for integers
- **primitive** definition

Statements

- **initial** statement
- delay control
- event control
- **wait** statement
- **repeat** statement
- **fork** statement
- **deassign** statement
- **force** statement
- **release** statement
- **defparam** statement

Operators

- Division and modulus operators for variables
- Case equality and inequality operators (=== and !==)

Gate-Level Constructs

- **pullup, pulldown,**
- **tranif0, tranif1, rtran, rtranif0, rtranif1**

Miscellaneous Constructs

- Compiler directives like **'ifdef, 'endif,** and **'else**
- Hierarchical names within a module