Verilog RTL Reference

Module type definition:

```
module ⁝module name⁝(
    input signed [msb:lsb] ⁝port name⁝,
    output reg signed [msb:lsb] ⁝port name⁝,
    inout reg signed [msb:lsb] ⁝port name⁝
);
// Suggested organization:
// declare parameter names and default values
// declare reg signals
// declare wire signals
// assign statements
// submodule instances
// initial block
// always blocks
// tasks and functions
endmodule
```

Synthesizable signal declarations:

```
    // reg signals used in always
    // or initial blocks:
    reg   signed [msb:lsb] ⁝name⁝ ;

    // wire signals used for submodule outputs
    // or assign statements:
    wire  signed [msb:lsb] ⁝name⁝ ;

    // Continuous assignment for wire signals:
    assign     ⁝name⁝= ⁝expression⁝ ;
```

Submodule instances:

```
// "par" is a parameter name
// "input1" and "output1" are port names
// port connections are comma-separated
⁝mod type⁝   ⁝name⁝   #(.⁝par⁝(⁝value⁝)) (
      .input1 (signal1),  // signal1 is reg or wire
      .output1(signal2)   // signal2 is a wire
);
// defparam is alternate way to set parameters:
defparam    ⁝name⁝.⁝par⁝ =  ⁝value⁝ ;
```

Combinational reg logic:

```
    always @( ⁝sensitivity list⁝) begin
        // Blocking assignments,
        // evaluated sequentially:
        ⁝reg signal 1⁝ =  ⁝expression 1⁝  ;
        ⁝reg signal 2⁝ =  ⁝expression 2⁝  ;
    end
```

Combinational sensitivity list:

(a,b,. . .)	signal list
(*)	wildcard

Sequential reg logic:

```
    always @( ⁝sensitivity list⁝) begin
        // Non-blocking assignments,
        // evaluated concurrently:
        ⁝reg signal 1⁝  <=  ⁝expression 1⁝ ;
        ⁝reg signal 2⁝  <=  ⁝expression 2⁝ ;
    end
```

Sequential sensitivity list:

(**posedge** ⁝signal⁝)	apply on rising edge
(**negedge** ⁝signal⁝)	apply on falling edge

Binary Bitwise Operators:

a & b	AND all a[i]&b[i]
a | b	OR all a[i]|b[i]
a ^ b	XOR all a[i]^b[i]

Unary Bitwise Operators:

&a	AND over all a[i]
|a	OR over all a[i]
^a	XOR over all a[i]
~a	NOT all a[i]

Concatenation and Replication:

{a,b}	concatenate vectors a,b
{N{{a}}	replicate a N times
{a,{N{b}}}	concatenate a with N copies of b

Synthesizable Arithmetic Operators:

a+b	addition
a-b	subtraction
a*b	multiplication
-a	negation
a>>b	right-shift (divide by 2^b)
a<<b	left-shift (multiply by 2^b)

Non-synthesizable Arithmetic Operators:

a/b	division
a%b	modulo

Logical Operators (return true/false):

a && b	a is true AND b is true
a || b	a is true OR b is true
a == b	equality (may return x)
a === b	4-value equality (compares x,z)
a != b	inequality (may return x)
a !== b	4-value inequality (compares x,z)
a > b	integer greater-than
a < b	integer less-than
a >= b	integer geq
a <= b	integer leq

Looping (within **always** or **initial** block):

```
integer    i;
always @( ⁝sensitivity list⁝) begin
    for(i=0; i<N; i=i+1) begin
        // looped statements
    end
end
```

Conditionals (within **always** or **initial** block):

```
if( ⁝logical condition⁝) begin
    // statements for first condition
end
else if ( ⁝logical condition⁝) begin
    // statements for second condition
end
else  begin
    // default statements
end

case (⁝signal⁝)
    ⁝val1⁝   : begin
               // Statements for val1
               end
    ⁝val2⁝   : begin
               // Statements for val2
               end
    default : begin
               // default statements
               end
endcase
```

Generate loops (structural):

```
genvar    i;
generate
    for (i=0; i<N; i=i+1) begin
        // looped statements
        // can be assign, module instances
        // or always blocks
    end
endgenerate
```

Task definition (within module):

```
task automatic ⁝task name⁝ ;
    input   ⁝input name⁝ ;
    output ⁝output name⁝ ;
    inout   ⁝inout name⁝ ;
    // Declare internal variables
    begin
        ⁝output name⁝= ⁝expression⁝ ;
    end
endtask
```

Task call (within **always** or **initial** block):

```
⁝task name⁝(⁝signal list⁝);
```

Function definition (within module):

```
function ⁝function name⁝ ;
    input   ⁝input name⁝ ;
    // Declare internal variables
    begin
        ⁝function name⁝  =  ⁝expression⁝ ;
    end
endfunction
```

Function call (within **always** or **initial** block):

```
⁝reg name⁝  =  ⁝function name⁝(⁝inputs⁝);
```

Console output:

$display(format,args)	write formatted text to console with newline
$write(format,args)	write formatted text to console, no newline
$strobe(format,args)	complete assignments, then write to console

File input/output:

integer fid; // file identifier must be declared

fid = $fopen(fname,mode)	open file, mode is **r**, **w** or **a**
$fwrite(fid,format,args)	write to file
$fstrobe(fid,format,args)	strobe to file
$fscanf(fid,format,reg)	read formatted data from file
$fclose(fid)	close file

Format strings:

%d	decimal
%x	hexadecimal
%o	octal
%b	binary
%f	real
%c	ASCII character
%Nd	print decimal with fixed width
%0Nd	print decimal with fixed width and leading zeros
%M	print signal name with hierarchy
\t	insert tab
\n	insert newline

Simulation control:

$finish	end the simulation
$stop	break (pause) the simulation

Testbench template:

```
module testbench();
    reg          clk;
    reg          rst;
    integer      count;
    // declare DUT input signals as reg
    // declare DUT output signals as wire

    // place submodule instance here

    initial begin
          clk     = 0;
          rst     = 0;
          count   = 0;
          forever #10  clk = ~clk;
    end
    always @(posedge clk) begin
          if (count==1)
                rst <= 1;
          else if (clk ⁝1) begin
                // Do test things
          end
    end
endmodule
```

State machine template:

```
module state_machine(
    // declare clk, rst, inputs and outputs
);
    reg [SIZE-1:0]     state;
    // declare other signals here
    // place any submodule instance here

    initial begin
        state= 0;
        // Do other reset things here
    end
    always @(posedge clk, negedge rst) begin
        if (~rst) begin
            // Do reset things here
        else begin
            case (state)
            //——— STATE 0 ————————//
            0:      begin
                        if (⁝branch condition 0⁝) begin
                            state <= ⁝next state⁝     ;
                            // Make branch assignments
                        end
                        else begin
                            // Do state 0 actions here.
                        end
                    end
            //——— STATE 1 ————————//
            1:      begin
                        if (⁝branch condition 1⁝) begin
                            state <= ⁝next state⁝     ;
                            // Make branch assignments
                        end
                        else begin
                            // Do state 1 actions here.
                        end
                    end
            // and so on for the other states...
            default:  state <= 0; // catch state errors
        endcase
    end
endmodule
```