Verilog

- System Tasks/Functions and Compiler Directives -

May 2014

Ando KI

Contents

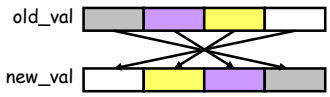
Task and function (1/2)

- Task
  - Declared within a module
  - Referenced only by a behavioral within the module
    - Can be referenced only from within a cyclic (always) or single-pass behavior (initial).
  - Parameters passed to task as inputs and inouts and from task a outputs or inputs
  - Local variables can be declared
  - Recursion not supported although nesting permitted
- Function
  - Implement combinational behavior
  - No timing control
    - '#' or '@' or 'wait' is not permitted
  - May call other functions with no recursion
  - Reference in an expression, e.g., RHS
  - No 'output' or 'inout' allowed
  - No non-blocking assignment
  - The purpose of a *function* is to respond to an input value by returning a single value.
    - A function can be used as an operand in an expression.
    - The value of that operand is the value returned by the function.

Task and function (2/2)

- Task
  - A task can contain time-controlling statements.
  - A task can enable other tasks and functions.
  - A task can have zero or more arguments of any type.
  - A task shall not return a value, since it creates a hierarchical organization of the procedural statements within a Verilog behavior.
- Function
  - A function shall execute in one simulation time unit.
  - A function cannot enable a task.
  - A function shall have at least one **input** type argument and shall not have an **output** or **inout** type argument.
  - A function shall return a single value, which can be scalar (single-bit) or vector (multi-bit).

Task and function usage example




```
// task is not allowed in continuous
// assignment

initial
  switch_byte_task(old_val, new_val);

always
  switch_byte_task(old_val, new_val);
```

```
// function can be used in continuous assignment
wire [31:0] new_val = switch_byte_function(old_val);


 initial
       new_val = switch_byte_function(old_val);

 always
       new_val <= switch_byte_function(old_val);
```

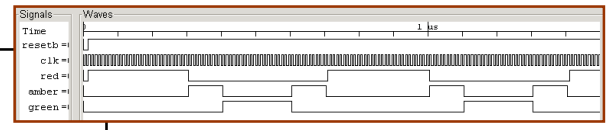

Form of task

```
module xxx (…);
…

task task_name;
   [input_output_inout_arguments;]
   statement;
endtask
…
endmodule
```

```
module xxx (…);
…

task task_name (in_out_inout_arguments);
   statement;
endtask
…
endmodule
```

Task example

'codes/verilog/task_function/traffic'




```
// traffic.v
module traffic_lights(clk, resetb);
  input clk, resetb;
  reg   red, amber, green;
  parameter on = 1, off = 0, red_tics = 30,
            amber_tics = 10, green_tics = 20;
  // initialize colors.
  initial red   = off;
  initial amber = off;
  initial green = off;
  always @ (clk or resetb) begin
     if (resetb==1'b1) begin
        red = on; // turn red light on
        light (red, red_tics); // and wait.
        amber = on; // turn amber light on
        light (amber, amber_tics); // and wait.
        green = on; // turn green light on
        light (green, green_tics); // and wait.
        amber = on; // turn amber light on
        light (amber, amber_tics); // and wait.
     end
  end
```

```
  // task to wait for 'tics' positive edge clocks
  // before turning 'color' light off.
  task light;
     output color;
     input [31:0] tics;
     begin
        repeat (tics) @ (posedge clk);
        color = off; // turn light off.
     end
  endtask
endmodule
```

DIY

Form of function

```
module xxx (...);
...
function function_name;
   input input_arg;
   [input_arguments;]
   begin
     [necessary_statements;]
     function_naem = expression;
   end
endfunction
...
endmodule
```

Default 1-bit output.

At least one input argument

A function definition shall include an assignment of the function result value to the internal variable that has the same name as the function name.

```
module xxx (...);
...

function [m:n] function_name;
   input input_arg;
   [input_arguments;]
   [necessary_statements;]
   function_name = expression;
endfunction
...
endmodule
```

Function example

- 'codes/verilog/task_function/factorial'

```
module top;

 function automatic integer factorial_auto;
    input [31:0] operand;
    integer i;
    if (operand >= 2) factorial_auto = factorial_auto(operand - 1) * operand;
    else factorial_auto = 1;
 endfunction

function integer factorial;
    input [31:0] operand;
    integer i;
    if (operand >= 2) factorial = factorial(operand - 1) * operand;
    else factorial = 1;
 endfunction

 integer result, n;
 initial begin
    for(n = 0; n <= 7; n = n+1) begin
        result = factorial_auto(n);
        $display("%0d factorial_auto=%0d", n, result);
        result = factorial(n);
        $display("%0d factorial=%0d", n, result);
    end
  end
endmodule
```

All items declared inside automatic functions are allocated dynamically for each invocation.
It is required to be enabled for recursive call.

DIY 

```
# 0 factorial_auto=1
# 0 factorial=1
# 1 factorial_auto=1
# 1 factorial=1
# 2 factorial_auto=2
# 2 factorial=1
# 3 factorial_auto=6
# 3 factorial=1
# 4 factorial_auto=24
# 4 factorial=1
# 5 factorial_auto=120
# 5 factorial=1
# 6 factorial_auto=720
# 6 factorial=1
# 7 factorial_auto=5040
# 7 factorial=1
```


System tasks and system functions

- System tasks and system functions are commands executed by a Verilog simulator, not hardware modeling constructs.
  - While VPI routines are functions to be used in C program.
- System tasks and system functions begin with $.
- System task -- a sub-routine procedure.
- System function -- a function returning a return value.
- Built-in system tasks and system functions
  - These are part of Verilog language and built into Verilog simulator.
  - Standard system tasks
    - $display, $stop, $finish
  - Standard system functions
    - $time, $random
- User-defined system tasks and system functions
  - Its name must start with $.

Verilog system task

- System tasks begin with $.
- System tasks are procedural programming statements in the Verilog HDL.
  - Called from the followings, but not from a continuous assignment statement.
    - 'initial' procedure
    - 'always' procedure
    - 'task'
  - It cannot return values, but its argument can be written.
  - Example:

```
always @ (posedge clock) begin
        $readmemh("vectors.dat", mem);
end
```


Verilog system function

- System functions begin with $.
- System functions are expression in the Verilog HDL.
  - Called from the followings,
    - 'initial' procedure
    - 'always' procedure
    - 'task'
    - 'function'
    - `assign' continuous statement
    - Operand in a compound expression
  - It can return value.
    - A value can only be written to the return from a calltf routine through 'vpi_put_vlaue()' with vpiNoDelay.
  - Example

```
always @ (posedge clock) if (chip_out!==$pow(base, exp)) ...
initial $monitor("output=%f", $pow(base, exp));
assign temp = i + $pow(base, exp);
```

Verilog system tasks/functions (basic)

- $stop
  - Suspend simulation when it is encountered.
- $finish [(n)]
  - Finish simulation when it is encountered.
- $time
  - A system function
- $random
  - A system function
- $readmemh
- $display(format, arg1, arg2, …);
  - The same as $write(), but newline is always added at the end of formatted string.
- $monitor(format, arg1, arg2, …);
  - Display formatted string each time any of ar1, arg2, .. changes.
- $write(format, arg1, arg2, …);
  - Display formatted string in standard output when it is encountered→ similar to printf() of C.

Simulation control tasks

- The $finish system task simply makes the simulator exit and pass control back to the host operating system.
- The $stop system task causes simulation to be suspended.
  - Thus, it can be resumed through HDL simulator dependent command.

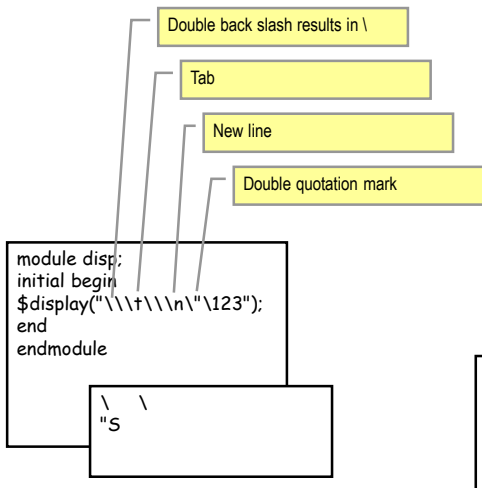
$finish [(n)] ;	$stop [(n)] ;
Parameter value: •0: print nothing •1: print simulation time and location •2: print simulation time, location, and statistics about the memory and CPU time used in simulation	Diagnostic message will be printed when the optional argument is given. Larger argument (0, 1, 2) will print more.

Displaying information (1/3)

- $display, $write, and $monitor display their arguments.
  - $display automatically adds a newline, while $write does not.
    - The $display task, when invoked without arguments, simply prints a newline character. A $write task supplied without parameters prints nothing at all.
  - $monitor automatically displays its argument whenever any value of its argument changes.
    - Exception of the $time.
- The special character \ indicates that the character to follow is a literal or non-printable character.
- The special character % indicates that the next character should be interpreted as a format specification that establishes the display format for a subsequent expression argument.
  - %% will print %.

```
display_tasks ::= display_task_name ( list_of_arguments );
display_task_name ::= $display | $write | $monitor
list_of_arguments ::= format [| format, variable_or_expressions ]
```


Displaying information (2/3)




```
module disp;
reg [31:0] rval;
pulldown (pd);
initial begin
rval = 101;
$display("rval = %h hex %d decimal",rval,rval);
$display("rval = %o octal\nrval = %b bin",rval,rval);
$display("rval has %c ascii character value",rval);
$display("pd strength value is %v",pd);
$display("current scope is %m");
$display("%s is ascii value for 101",101);
$display("simulation time is %t", $time);
end
endmodule
```

```
rval = 00000065 hex 101 decimal
rval = 00000000145 octal
rval = 00000000000000000000000001100101 bin
rval has e ascii character value
pd strength value is StX
current scope is disp
e is ascii value for 101
```

Displaying information (3/3)

Table 66—Escape sequences for printing special characters

Argument	Description
\n	The newline character
\t	The tab character
\\	The \ character
\"	The " character
\ddd	A character specified by 1 to 3 octal digits
%%	The % character

'%x' is also possible.

Table 67—Escape sequences for format specifications

Argument	Description
%h or %H	Display in hexadecimal format
%d or %D	Display in decimal format
%o or %O	Display in octal format
%b or %B	Display in binary format
%c or %C	Display in ASCII character format
%l or %L	Display library binding information
%v or %V	Display net signal strength
%m or %M	Display hierarchical name
%s or %S	Display as a string
%t or %T	Display in current time format
%u or %U	Unformatted 2 value data
%z or %Z	Unformatted 4 value data
%e or %E	Display 'real' in an exponential format
%f or %F	Display 'real' in a decimal format
%g or %G	Display 'real' in exponential or decimal format, which-ever format results in the shorter printed output

Simulation time system tasks

- $time
  - Returns an integer that is a 64-bit time.
  - Fractional part will be round.
- $stime
  - Returns an unsigned integer that is a 32-bit time.
- $realtime
  - Returns a real number time.

- 'codes/verilog/task_function/time'

```
`timescale 1 ns / 1 ps
module top;
 initial begin
  #10; timex;   #20.1; timex;   #0.01; timex;
 end
 task timex;
  begin
  $display("time:      %d", $time);
  $display("stime:     %d", $stime);
  $display("realtime: %f", $realtime);
  end
 endtask
endmodule
```

```
# vsim -do {run -all; quit} -c work_mti.top
# Loading work_mti.top
# run -all; quit
# time:                  10
# stime:          10
# realtime: 10.000000
# time:                  30
# stime:          30
# realtime: 30.100000
# time:                  30
# stime:          30
# realtime: 30.110000
```

Random number generator

- The system function $random provides a mechanism for generating random numbers. It returns a new 32-bit random number each time it is called. The number is a signed integer, so that it can be positive or negative.

Where b is greater than 0, the expression below gives a number in the following range.
[(-b+1)-(b-1)].

```
$random % b;
```

```
// random number: -59 ~ 59

reg [23:0] var_reg;
var_reg = $random % 60;
```

```
// random number: 00, 01, 10, 11

reg [1:0] var_reg;
var_reg = $random & 2'h3;
```

```
// random number of 8-bits.

reg [7:0] var_reg;
var_reg = $random & 8'hFF;
```


Loading memory data from a file (1/2)

- Two system tasks ($readmemb and $readmemh) read and load data from a specified text file into a specified memory.
- It can be called at any time during simulation.
- The text file shall contain only the following:
  - White space (space, new line, tab, form-feed)
  - Comments (// or /* */)
  - Binary for $readmemb and hexadecimal for $readmemh
  - Upper and lower cases are allowed.
  - X, x, Z, z, and _ can be used.
  - Do not use the base format, such as 4'h, 8'b, and so on.
  - Address can be specified by a hexadecimal number following @.
    - @hh…h

```
Load_memory_tasks ::=
  $readmemb("file_name", mem_name [, start_addr [, finish_addr]]);
 |$readmemh("file_name", mem_name [, start_addr [, finish_addr]]);
```

Loading memory data from a file (2/2)

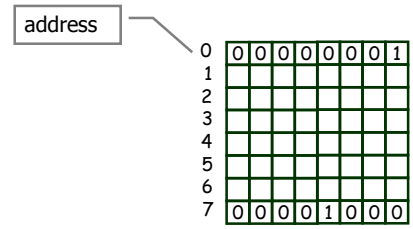
```
reg [7:0] mem[1:128];
… …
initial $readmemh("mem.dat", mem); // case A
initial $readmemh("mem.dat", mem, 16); // case B
initial $readmemh("mem.dat", mem, 128, 1); // case C
```

- Case A:
  - Load up the memory at simulation time 0 starting at the memory address 1.
- Case B:
  - Load up the memory at simulation time 0 starting at the memory address 16.
- Case C:
  - Load up the memory at simulation time 0 starting at the memory address 128 down to 1.

Loading memory example

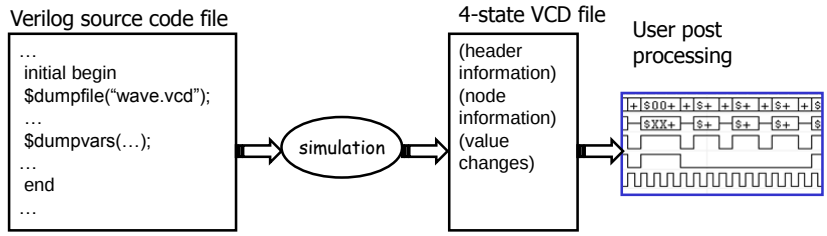
```
module rom ( cs, addr, as, oe, data );
 parameter WIDTH_ADDR=3;
 input              cs;
 input  [WIDTH_ADDR-1:0] addr;
 input              as;
 input              oe;
 output [7:0]       data;
 //---------------------------------------
 wire   [7:0]        data;
 reg    [7:0]        cont;
 //---------------------------------------
 parameter MEMORY_DEPTH=(1<<WIDTH_ADDR);
 reg [7:0] memory[0:MEMORY_DEPTH-1];
//---------------------------------------
 … …
 initial begin
      $readmemh("memory_data.txt", memory);
 end
endmodule
```

```
// memory_data.txt
01
02
03
04
05
06
07
08
```



VCD

- Value Change Dump file
  - Containing information about value changes on selected variables in the design.
  - It is an ASCII file.
- Two types of VCD
  - Four state
    - 0, 1, X, Z
  - Extended
    - All state and strength



VCD handling system tasks (1/2)

- $dumpfile specifies the name of the dump file.
- '$dumpfile([*file_name*]);'
  - Where 'file_name' is optional and it will be 'dump.vcd' if not given.
  - It Specifies the name of the VCD file.

```
initial $dumpfile("wave.vcd");

initial $dumpvars(1, UmodA, UmodB.x);

initial $dumpvars(0, UmodC);
```

- $dumpvars specifies the variables to be dumped.
- '$dumpvars(level, var [, var […]]);
  - 'level'
    - How many levels of the hierarchy below each specified module instance to dump to the VCD file.
    - If level is 0, it causes a dump of all variables in the specified module and in all module instances below the specified module.
  - Subsequent arguments specify which scopes of the model to dump to the VCD file.
    - These argument can be entire module or individual variables within a module.

VCD handling system tasks (2/2)

- Stopping and resuming the dump
  - $dumpon
  - $dumpoff
- Limiting the size of dump file
  - $dumplimit(file_size_in_byte);
  - The 'file_size_in_byte' specifies the maximum size of the VCD file in bytes.
  - If VCD file reaches the number of bytes, the dumping stops.

```
initial $dumpfile("wave.vcd");
initial $dumplimit(1000000);

initial $dumpvars(1, UmodA, UmodB.x);

initial $dumpvars(0, UmodC);

initial begin
  #10   $dumpvars(1, UmodA.y);
  #100 $dumpoff;
  #200 $dumpon;
  #300 $dumpoff;
end
```


Compiler directives

- `define
- `undef
- `include
- `ifdef, `elsif, `else, and `endif
- `ifndef, `elsif, `else, and `endif
- `timescale

‘define and ‘undef

- ‘define
  - The directive `define creates a macro for text substitution.
  - Once a text macro name is defined, it can be used anywhere in a source description.
  - Redefinition of a text macro is allowed and the latest definition of a particular macro read by the compiler prevails when the macro name is encountered in the source text.
- `undef
  - The directive `undef is used to undefine a previously defined text macro.

```
`define SEL_A     4'b0000
`define SEL_B     4'b0001
`define FILE_X    "my_file.txt"
```

Use back-quotation, not single quotation mark

```
`undef SEL_A
`undef SEL_B
`undef FILE_X
```


`include

- `include
  - The `include compiler directive allows one to include the contents of a source file in another file during compilation.
  - The file name in the `include directive can be a full or relative path name.
  - A file included in the source using the `include compiler directive may contain other `include compiler directives.
  - The `include construct cannot be used recursively.

Use back-quotation, not single quotation mark

```
`include "my_file.v"
```

`ifdef, `ifndef, `else, `elsif and `endif

- `ifdef - `elsif - `else - `endif
  - The `ifdef compiler directive checks for the definition of a macro.
  - If the macro is defined, then the line following the `ifdef directive are included.
  - If the macro is not defined and an `else directive exists, then the source is compiled.

```
`ifdef  CERTAIN_MACRO
        `include "my_first.v"
`elsif ANOTHER_MACRO
        // bla bla ..
`else
        `include "my_second.v"
`endif
```

- `ifndef - `elsif - `else - `endif
  - The `ifndef compiler directive checks for the definition of a macro.
  - If the macro is not defined, then the line following the `ifndef directive are included.
  - If the macro is defined and an `else directive exists, then the source is compiled.

```
`ifndef  CERTAIN_MACRO
        `include "my_first.v"
`elsif ANOTHER_MACRO
        // bla bla ..
`else
        `include "my_second.v"
`endif
```


`timescale

- It specifies the time unit and time precision of the modules.
  - It specifies the unit of measurement for time and delay values.
  - It specifies the degree of accuracy for delays.
    - Any time calculations would be internally rounded to the nearest the time precision.
- Its effect spans to all modules that follow this directive until another `timescale directive is read.

```
timescale_compiler_directive ::=
    `timescale time_unit / time_precision
```

Use back-quotation mark.

```
`timescale 1 ns / 1 ps
module xx (yy, zz);
   ......
endmodule
```

Char. string	unit	remarks
s	Second	-
ms	10^{-3} second	Milli
us	10^{-6} second	Micro
ns	10^{-9} second	Nano
ps	10^{-12} second	Pico
fs	10^{-15} second	Femto

Timescale examples

```
`timescale 1 ns / 1 ps
module top;
 initial begin
  #10; timex;   #20.1; timex;   #0.01; timex;
 end
 task timex;
  begin
  $display("time:       %d", $time);
  $display("stime:      %d", $stime);
  $display("realtime: %f", $realtime);
  end
 endtask
endmodule
```

```
# vsim -do {run -all; quit} -c work_mti.top
# Loading work_mti.top
# run -all; quit
# time:                    10
# stime:           10
# realtime: 10.000000
# time:                    30
# stime:           30
# realtime: 30.100000
# time:                    30
# stime:           30
# realtime: 30.110000
```

```
`timescale 1 ns / 1 ns
module top;
 initial begin
  #10; timex;   #20.1; timex;   #0.01; timex;
 end
 task timex;
  begin
  $display("time:       %d", $time);
  $display("stime:      %d", $stime);
  $display("realtime: %f", $realtime);
  end
 endtask
endmodule
```

Note the resolution

```
# vsim -do {run -all; quit} -c work_mti.top
# Loading work_mti.top
# run -all; quit
# time:                    10
# stime:           10
# realtime: 10.000000
# time:                    30
# stime:           30
# realtime: 30.000000
# time:                    30
# stime:           30
# realtime: 30.000000
```

Note the result

Reading

- IEEE Std. 1364-2001, IEEE Standard Verilog Hardware Description Language. (Chapter 9. Behavioral Modeling; Chapter 17. System Tasks and Functions; Chapter 18. Value Change Dump File)