VERILOG

SYSTEM TASKS AND PRODECDURAL ASSIGNMENTS

2015 © Centre for Development of Advanced Computing

ROHIT KHANNA

System Tasks

- Verilog provides standard system tasks for certain operation like
  - Displaying and monitoring data
  - Controlling Simulation
  - Initializing Memory
  - Timing Checks
  - Data Conversion
- Name of System tasks begins with $ followed by name of the task.
- System Tasks are ignored by synthesis tools.

$display

- This system task is used to display value of variables, strings or expressions.
- It is very similar to printf in C.
- $display inserts a newline by default.

Examples:

```
$display("Hello world");
$display("value of count is %d", counter);
$display(a, b, c);
$display($time);
$display($time," a=%d  and b=%h ", a , b);
```

Format Specifiers

Format	Display
%d or %D	Decimal
%b or %B	Binary
%h or %H	Hexadecimal
%o or %O	Octal
%c or %C	ASCII character
%s or %S	String
%v or %V	Strength
%m or %M	Hierarchical Name (no argument)
%e or %E	Real number in scientific format
%f or %F	Real number in decimal format

$write

- $write works same as $display except it does not automatically add a new line.

Example:

```
module write_test;
reg [3:0] count;
initial
begin
count=4'b1010;
$write("Hello Verilog");
$write("count value=%b", count);
end
endmodule
```

```
module display_test;
reg [3:0] count;
initial
begin
count=4'b1010;
$display("Hello Verilog");
$display("count value=%b", count);
end
endmodule
```

Output

```
Hello Verilogcount value=1010
```

```
Hello Verilog
count value=1010
```

$monitor

- $monitor continuously monitors list of variables and executes whenever any of the arguments changes it value.
- $monitor automatically adds a new line.
- Only one $monitor can be active at a time.
- If there are multiple $monitor in the code, the new $monitor will override any previous $monitor in effect.
- $monitoron and $monitoroff are used to enable and disable monitoring respectively. Monitoring is ON by default.

Example : $monitor("Values are a=%d and b=%d", a, b);

$monitor

```
module tb;
   reg  a, b, sel;
   wire out;

    mux m0 (.out(out), .ip1(a), .ip2(b), .select(sel));  //Module Instance

    initial          //initial block for monitoring
    begin
    $monitor("out=%b  ip1=%b  ip2=%b  select=%b", out, a, b, sel);
    end
                  // continues in next slide
endmodule
```

$monitor

```
initial          //Providing Inputs
begin
a=1'b1; b=1'b0; sel=1'b1;
#10 sel=1'b0;
#10 a=1'b0;
#10 b=1'b1;
#10 sel=1'b1;
end
```

Output

```
out=0  ip1=1  ip2=0  select=1
out=1  ip1=1  ip2=0  select=0
out=0  ip1=0  ip2=0  select=0
out=0  ip1=0  ip2=1  select=0
out=1  ip1=0  ip2=1  select=1
```

$strobe

- $strobe is similar to $display
- $strobe automatically adds a new line.
- $strobe displays data at the end of current simulation time.

Example:

```
module tb;
   reg  [2:0] a=3'b000, b=3'b000;

// Continued in next slide

endmodule
```

$strobe

```
initial
begin
a=3'b101;
$display("display1 a=%b  b=%b", a, b);
$strobe("display2 a=%b  b=%b", a, b);
b=3'b011;
#10
$strobe ("display2 a=%b  b=%b", a, b);
b=3'b100;
$display("display1 a=%b  b=%b", a, b);
a=3'b111;
end
```

Output

```
display1 a=101  b=000
display2 a=101  b=011

display1 a=101  b=100
display2 a=111  b=100
```

Other Display

- Decimal is the default format if display format is not specified in $display, $write, $monitor and $strobe.
- Verilog provides several system tasks which has default display format other than decimal.

$displayb	$writeb	$monitorb	$strobeb
$displayh	$writeh	$monitorh	$strobeh
$displayo	$writeo	$monitoro	$strobeo

Time System Functions

- Verilog offers following system functions to access current simulation time.

  $time $stime $realtime

- $time returns an integer that is 64-bit current simulation time in terms of timescale unit.

- $stime returns an unsigned integer that is 32-bit current simulation time in terms of timescale unit.

- $realtime returns a real number that is current simulation time in terms of timescale unit.

  Example $display("Current simulation time ", $time);

Gate Delays

- Verilog allows user to specify delays in logic circuits
- Gate delays can be classified as following

  Rise Delay Fall Delay Turn-off Delay

- Rise Delay is the delay encountered when output changes from 0, X, Z to 1.
- Fall Delay is the delay encountered when output changes from 1, X, Z to 0.
- Turn-off Delay is the delay encountered when output changes from 0, 1, X to Z.

Gate Delays

- Minimum out of all these delays is the delay encountered when output changes from 0, 1, Z to X.
- If only one delay is specified, this value is used for all transitions.
- If two are specified, first refers to rise delay and second refers to fall delay. Turn-off Delay is minimum of the two.
- If three delays are specified, first refers to rise delay, second refers to fall delay and third refers to Turn-off Delay.
- If no delay is specified then delay value is 0.

Gate Delays

Examples

```
and #(2) u0 (out, ip1, ip2);
// Rise Delay=2     Fall Delay=2     Typical Delay=2

and #(2, 3) u1 (out, ip1, ip2);
//Rise Delay=2     Fall Delay=3     Typical Delay=2

bufif0 #(2, 3, 4) u0 (out, ip1, ip2);
//Rise Delay=2     Fall Delay=3     Typical Delay=4

assign #(3) out1= a | b;
//Rise Delay=3     Fall Delay=3     Typical Delay=3

assign #(2, 1) out2 = a ^ b;
//Rise Delay=2     Fall Delay=1     Typical Delay=1
```

Min/Typ/Max Values

- Minimum(min), Typical(typ) and Maximum(max) delays can be specified for each type of delay: rise, fall, and turn-off.
- Any one value (min, typ and max) can be chosen at the start of the simulation.
- If no option is specified during simulation, typical delay values are used for Simulation.

Min/Typ/Max Values

Examples

```
and #(2:3:4) u0 (out, ip1, ip2);
// For all transitions
// Min Delay=2      Typ Delay=3      Max Delay=4

and #(2:3:4, 3:5:7) u1 (out, ip1, ip2);
//Rise Delay:       Min Delay=2      Typ Delay=3      Max Delay=4
//Fall Delay:       Min Delay=3      Typ Delay=5      Max Delay=7
//Turn-off Delay:   Min Delay=2      Typ Delay=3      Max Delay=4

bufif0 #(2:3:4, 3:4:6, 4:5:7) u0 (out, ip1, ctrl);
//Rise Delay:       Min Delay=2      Typ Delay=3      Max Delay=4
// Fall Delay:      Min Delay=3      Typ Delay=4      Max Delay=6
//Turn-off Delay:   Min Delay=4      Typ Delay=5      Max Delay=7
```

Procedure Statements

- Verilog offers two types of procedure statements

  initial always

- Procedure statements represent that section of Verilog where statements are executed in a sequential manner.
- There can be multiple procedure statements, each statement starts at 0 simulation time.
- Nesting of procedure statements is not allowed.
- All statements inside initial statement and always statement constitute initial and always block respectively.

Initial Block

- An initial block executes exactly once during a simulation.
- If there are multiple initial blocks, each block starts to execute concurrently at 0 simulation time.
- If there are multiple statements within an initial block, they should be grouped using begin and end keyword.
- Initial blocks are typically used for initialization, monitoring, providing test inputs and other operations which must be executed only once during the entire simulation.

Initial Block

Syntax:

```
initial  [@ (trigger_control)]
begin
[timing control] procedural assignments;
end
```

- Initial block can be triggered based on an event but is generally used without trigger control.
- begin and end required only if there are multiple statements.
- Timing control is used to delay the execution of a procedural assignment by specified time. # followed by time value is used to provide delay.

Initial Block - Example

```
module tb;
   reg b, c, out;
   reg a=1;
//reg can be initialized during declaration

    initial        //initial block 1
      begin
       a=0; b=0;
       #5 b=1;
       #13 a=1;
       end

initial //initial block 2
c=0;  //begin-end not required

initial    //initial block 3
    begin
    out=1;
    #10 out=0;
    #8 out=1; c=1;
    end
endmodule
```

Initial Block - Result

Simulation Time	Statements Executed
0	a=0; b=0; c=0; out=1;
5	b=1;
10	out=0;
18	a=1; out=1; c=1;

Simulation Control

- Verilog provides two simulation control system tasks

  $stop[(N)] $finish[(N)]

- $stop is used to suspend simulation.
- $finish is used to exit simulation.
- $reset is used to reset simulation to time 0.

N (Option)	Description
0	Prints Nothing
1	Prints Simulation Time and Location
2	Prints Simulation Time, Location, Memory and CPU statics used during simulation

Always Block

- The statements in always executes continuously in a looping fashion.
- This statement is used to model block of digital circuit that executes continuously.
- If there multiple always blocks, then all blocks start at 0 time and executes simultaneously.
- If there are multiple statements within an initial block, they should be grouped using begin and end keyword.

Always Block

Syntax:

```
always  [@ (senstivity_list)]
begin
[timing control] procedural assignments;
end
```

- Execution of always block can be controlled with help of senstivity_list. always block is executed whenever event occurs on any of the variables present in senstivity_list.
- begin and end required only if there are multiple statements.
- Timing control is used to delay the execution of a procedural assignment by specified time.

Always Block- Example

Example : clock generation

```
module clk_gen;
reg clk;

initial
clk=1'b0;          //Initialize clk to 0

always             //begin-end not required
#5 clk=~clk;       //continuously perform not of clk
                   //after every 5 unit time.

endmodule
```

Always Block- Example

Example : Xor Gate

```
module xor_gate (out, a, b);
input a, b;
output reg out;

always @ (a, b)     //Executes whenever a or b changes
out= a ^ b;         //out should to be of type reg

endmodule
```

Always Block- Example

Example : D Flip - Flop

```
module dff (q , d, clk);
input d, clk;
output reg q;

always @ (posedge clk)    // Execute whenever positive edge is
                          // encountered on clk
q= d;                     // q should to be of type reg

endmodule
```

Procedural Assignments

- Procedural assignments are used to update values of reg, integer, real, or time variable.

- The value assigned to the variable does not change until another procedural assignment updates the value.

Syntax :

register_type_variable = or<= [delay or event control] expression

- There are two types of procedural assignments

Blocking Non Blocking

Blocking Statements

- Blocking statements blocks the execution of other statements till the current blocking statements in not executed.
- In case of blocking statements assignment occurs at the same time the statement is encountered.
- Verilog recommends to used blocking statements to model a combinational circuits.
- "=" sign is used to perform blocking procedural assignment.

Syntax : register_type_variable = expression;

Blocking Statements - Example

```
module block_test ;
reg [1:0] a=2'b01, b=2'b10 , c=2'b11;

    initial
    begin
    a=2'b00;
    b=a;
    c=b;
    end

endmodule
```

Result:

a=2'b00
b=2'b00
c=2'b00

Non-Blocking Statements

- Non-Blocking statements does not blocks the execution of other statements.
- In case of non-blocking statements assignment is scheduled to occurs at the end of current simulation time or at the end of procedure block.
- Verilog recommends to used non-blocking statements to model a sequential circuits.
- "<=" sign is used to perform non-blocking procedural assignment.

  Syntax : register_type_variable <= expression;

Non-Blocking Statements - Example

```
module non_block_test ;
reg [1:0] a=2'b01, b=2'b10 , c=2'b11;

    initial
    begin
    a<=2'b00;
    b<=a;
    c<=b;
    end

endmodule
```

Result:

a=2'b00
b=2'b01
c=2'b10

Examples

```
module example1;
  integer a;

initial
begin
#5 a=6;
$strobe ($time, "strobe ", a);
$display($time, "display ", a);
a=12;
end
endmodule
```

Result:

```
5    display  6
5    strobe  12
```

Examples

```
module example2;
  integer a;

initial
begin
#5 a<=6;
$strobe ($time, "strobe", a);
$display($time, "display", a);
a<=12;
end
endmodule
```

Result:

```
5    display  X
5    strobe  12
```

Examples

```
module example3;
  integer a;

initial
begin
#5 a=6;
$strobe ($time, "strobe", a);
$display($time, "display", a);
a<=12;
end
endmodule
```

Result:

```
5    display  6
5    strobe  12
```

Examples

```
module example4;
  integer a;

initial
begin
#5 a<=6;
$strobe ($time, "strobe", a);
$display($time, "display", a);
a=12;
end
endmodule
```

Result:

```
5    display  X
5    strobe   6
```

Race Condition

```
module race_around1 ;
reg [1:0] a=2'b01, b=2'b10;

initial
a=b;

initial
b=a;
endmodule
```

Output

Definitely both a and b will be same

a=2'b01 and b=2'b01

or

a=2'b10 and b=2'b10

Depends upon which initial block executes first

that depends upon the simulator

Race Condition

```
module race_around2 ;
reg [1:0] a=2'b01, b=2'b10;

initial
#0 a=b;

initial
b=a;
endmodule
```

Output

a=2'b01 and b=2'b01

#0 (Zero Delay) assures that statement is executed at last after all statements in current simulation time are executed

Race Condition

```
module race_around3 ;
reg [1:0] a=2'b01, b=2'b10;

initial
a<=b;

initial
b<=a;
endmodule
```

Output

a=2'b10 and b=2'b01

Conditional Statement

- Conditional statements are used to perform some task based on certain condition.
- if and else keywords are used for conditional statements.

  Syntax1: if (<Expression>) begin true_statements; end

- Expression is considered as true if it evaluates to a non-zero value.
- Expression is considered as false if it evaluates to a zero or ambiguous(X) value.

Conditional Statement

Syntax2: True or False

```
if (<Expression>)
   begin
   true_statements;
   end
else
   begin
   false_statements;
   end
```

begin and end are optional in there is only one statement.

If expression evaluates to true, true_statements are executed else false_statements are executed

Conditional Statement

Syntax3: Nested if-else

```
if (<Expression1>) begin true_statements1;  end
else if (<Expression2>) begin true_statements2; end
else if (<Expression3>) begin true_statements3; end
else begin default_statements end;
```

Conditional Statement

Example: Xor Gate

```
always @ (a, b)
begin
if (a==b)
c=1;
else
c=0;
end
```

Multiway Branching

- Verilog provides case statement which tests whether expression matches one of the multiple alternatives.

```
Syntax:  case (<expression>)
         alternative1 : begin statements1; end
         alternative2 : begin statements2; end
         alternative3 : begin statements3; end
         ……
         default : begin default statements; end
         endcase
```

- The expression is compared to the alternatives in the order they are written.

Multiway Branching

- Statements corresponding to first alternative that matches the expression are executed.
- Incase no match is found, default statement is executed.
- default statement is optional. Default statement can written in any where, it is always the last alternative expression is compared to.
- Nesting of case statements is allowed.
- begin and end keyword is optional if there is only one statement to be executed for given alternative.

Multiway Branching

Example1: Multiplexer

```
always @ (*)  //all RHS variable covered in sensitivity list
case(sel)
2'b00: out=a;
2'b01: out=b;
2'b10: out=c;
2'b11: out=d;
default: $display("Invalid selection");
endcase
```

Multiway Branching

Example2

```
always @ (*)  //all RHS variable covered in sensitivity list
case(sel)
2'b00: out=a;
2'b01: out=b;
default: $display("Invalid selection");
2'b01: out=c;
2'b11: out=d;
endcase
```

Multiway Branching

Example3

```
always @ (*)
case(sel)
2'b00: out=a;  2'b01: out=b;
2'b10: out=c;   2'b11: out=d;
2'b0x, 2'bx0, 2'b1x, 2'bx1, 2'bzx, 2'bxz, 2'bxx:
begin  out=0; $display("sel contains X"); end
2'b0z, 2'bz0, 2'b1z, 2'bz1, 2'bzz:
begin  out=0; $display("sel contains Z"); end
default: $display("unspecified input");
endcase
```

case, casez, casex

- case: In a normal case statement, valid inputs are 0, 1, X and Z. The expression and alternatives are compared bit wise bit including X and Z.

- casez: In casez, valid inputs are 0, 1 and X. It treats Z or ? present in expression or alternatives as don't cares.

- casex: In casex, valid inputs are 0 and 1. It treats X, Z or ? present in expression or alternatives as don't cares.

casez

Example: casez

```
always @ (*)  //all RHS variable covered in sensitivity list
casez(sel)
2'b0?: out=a;
2'b0x: out=b;
2'b10: out=c;
2'b11: out=d;
default: out=e;
endcase
```

casex

Example: casex

```
always @ (*)  //all RHS variable covered in sensitivity list
casex(sel)
2'bx0: out=a;
2'b00: out=b;
2'b1?: out=c;
2'b11: out=d;
default: out=e;
endcase
```