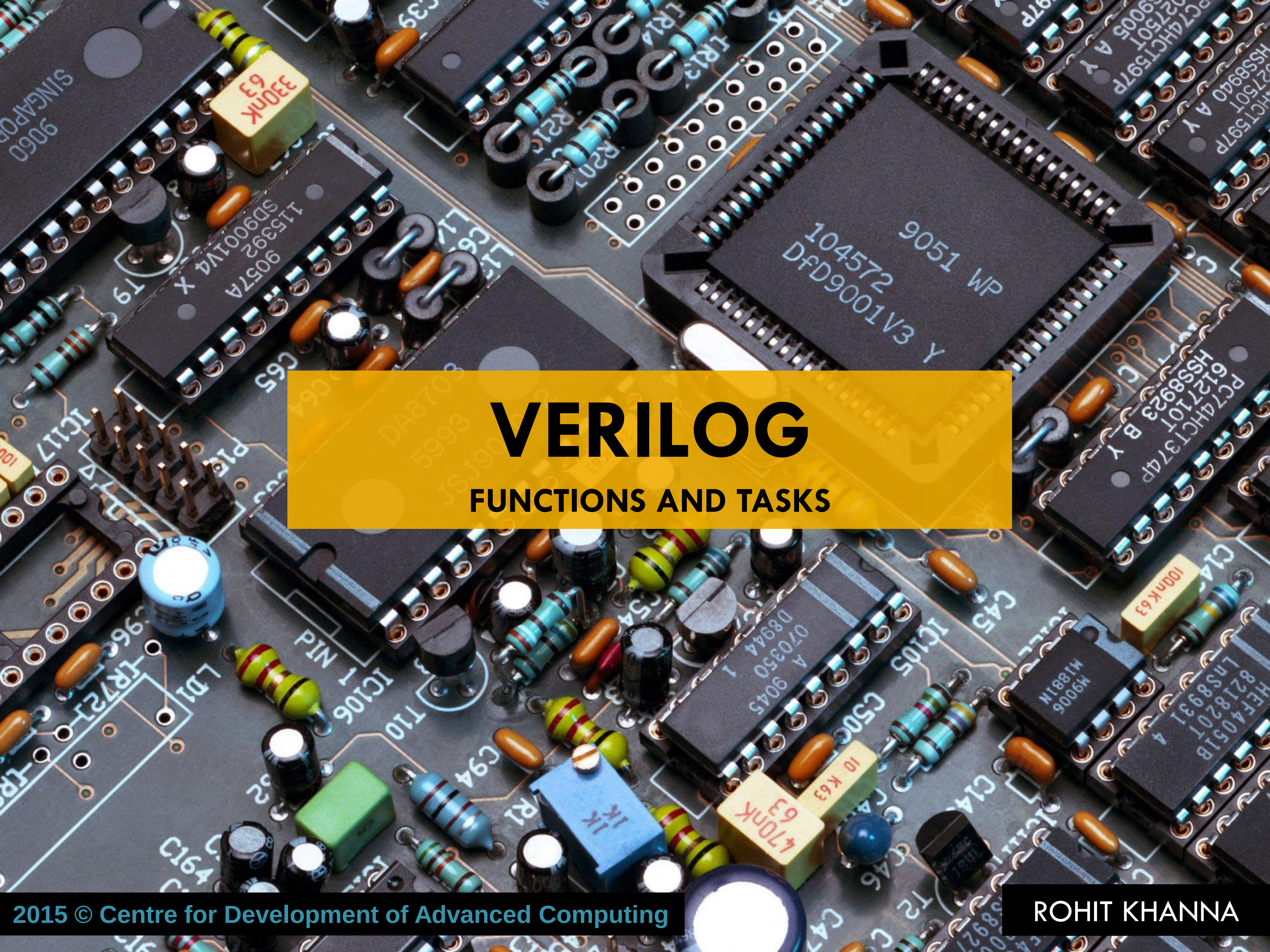


# VERILOG

## FUNCTIONS AND TASKS

ROHIT KHANNA

## Function and Tasks

- In some situations same functionality is required at many places in a program.
- These commonly used codes (functionality) should be abstracted into routines and they should be invoked instead of repeating code.
- Verilog provides functions and tasks to break up large behavior designs into smaller codes.
- Task or functions can only contain sequential statements.

## Functions

- Functions are declared with keyword function and endfunction.
- Functions are used to return a single value. They cannot have output and inout ports.
- A function must have at least one input argument. It may contain more than one input.
- Delay, Event and Timing Control statements are not allowed inside function.
- Functions always executes in zero simulation time.

## Functions

- When a function is declared, a register with the same name is implicitly(by default) declared in verilog.
- A value is returned back by assigning appropriate value to the implicit register (function name).
- The function is invoked by specifying function name followed by input arguments.
- A function can be called from a function, task, port declaration, continuous assignment and procedural assignment.
- A function can invoke other functions but it cannot invoke other tasks.

# Functions

Declaration : Style1

```
function [return_type] function_name;
//input declarations;
//local registers declaration;
  begin       //begin-end if multiple statements
    sequential statements;
  end
endfunction
```

- Return type and all inputs are 1-bit reg by default.
- Return variable name is same as function_name.
- Only Blocking sequential statements are allowed.

## Functions

Declaration : Style2

```
function [return_type] function_name (input_declarations);
//local registers declaration;
  begin
    sequential statements;
  end
endfunction
```

Usage:

```
assign net_name = function_name (input_list);
reg_name1  = function_name (input_list);
reg_name2 <= function_name (input_list);
```

## Functions-Examples

Problem Statement : Write a function which accepts 8 bit input din and returns parity. Use for loop instead of reduction operator.

```
function parity (input [7:0] din);
integer i;
reg temp;
begin
    temp=0;
       for (i=0; i<=7; i= i + 1)
       temp= temp ^ din[i];
    parity=temp;
  end
endfunction
```

```
module parity_cal (din, dout);
input [7:0] din;
output dout;

//function declaration

assign dout=parity(din);

endmodule
```

## Functions-Examples

Problem Statement : Write a function to return maximum out of three integers a, b and c.

```
function integer maximum (input signed [31:0] a, b, c) ;
  begin
    if (a > b && a> c)
      maximum=a;
    else if (b>c)
      maximum=b;
    else
      maximum=c;
   end
endfunction
```

```
module max (a, b, c, result) ;
input signed [31:0] a, b, c;
output integer result;
//function declaration

always @ (*)
result=maximum(a, b, c);

endmodule
```

## Automatic function

- A function call allocates static memory to variables used in its operation.
- If the same function is called again, same set of allocated memories are used for its operation.
- In case of recursive or re-entrant functions (function called within a function) using static memories may give wrong result.
- automatic keyword is added after function keyword to target a recursive function.
- An automatic function allocates a new set of memory to the variables every time a function is called.

## Automatic function

Problem Statement : Write a function which accepts positive integer and returns factorial of that number.

```
function [63:0] fact (input [31:0] din);
  begin
  if (din<=1)
  fact=1;
  else
  fact= fact(din -1) * din ;
  end
endfunction
```

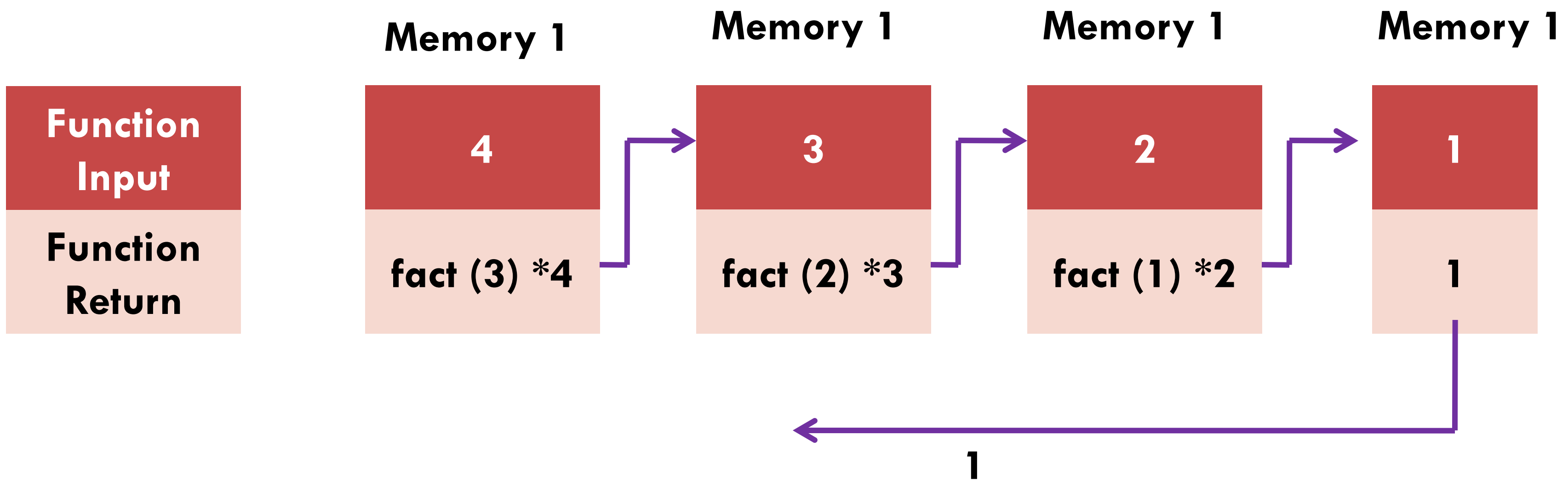
```
module fact_test (din, dout);
input [31:0] din;
output reg [63:0] dout;

//function declaration

always @ (*)
dout=fact(din);

endmodule
```

# Automatic functions

Assume that user gave 4 as an input to the fact_test function. What will be the value of the dout.

Expected: 24

Outcome: 1



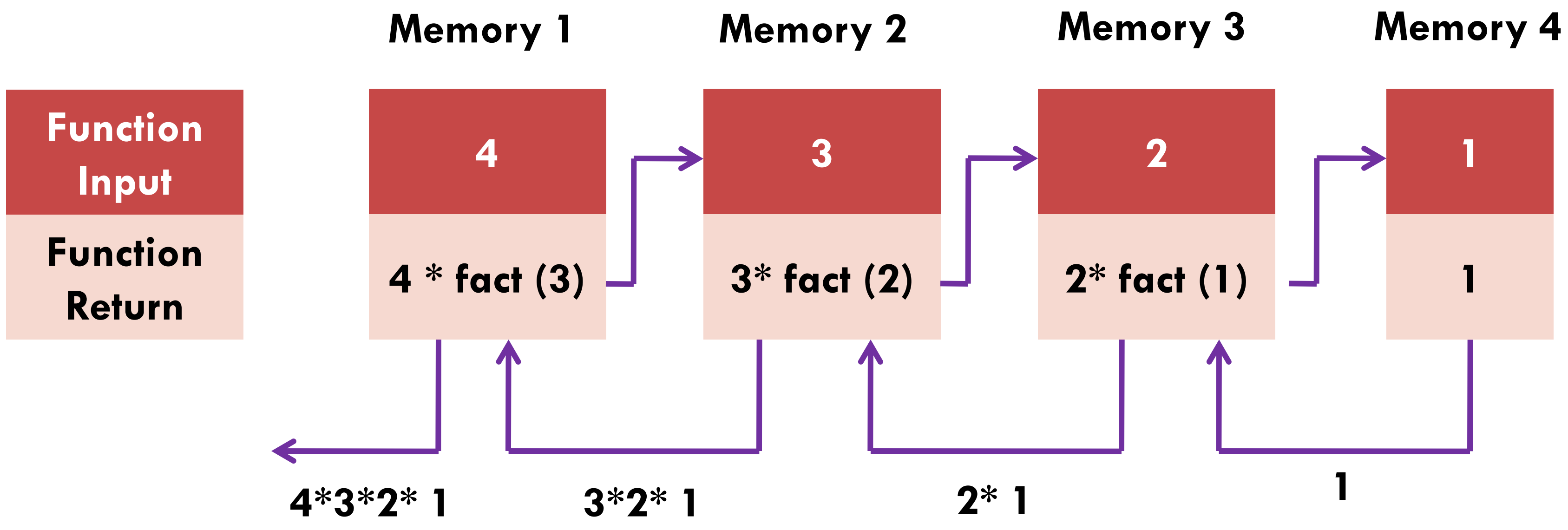
## Automatic functions

To solve this issue we will use add automatic keyword because we intend to design a recursive function.

```
function automatic [63:0] fact (input [31:0] din);
  begin
  if (din<=1)
  fact=1;
  else
  fact= din * fact(din -1);
  end
endfunction
```

# Automatic functions

Since automatic keyword is added to the function, for every function call a new set of memory is created and the result is returned to location where the function call occurs.



## Signed Function

A signed function is one which returns signed values. This can be done by adding signed keyword before the function return size.

Problem Statement : Write a function which accepts 16-bit unsigned number and returns 16-bit signed number.

```
function signed [15:0] sign (input [15:0] a);
  begin
   sign=a;
  end
endfunction
```

```
module sample (a, b, c ) ;
input [15:0] a;
output [15:0] b, c;
//function declaration

assign b=(sign(a))>>>3;
assign c=a>>>3;

endmodule
```

## Functions

Functions have access to variables declared outside its scope.

```
module test (input [31:0] din, output integer b) ;

integer c=10, d=4;

function [31:0] adder (input [31:0] a);
begin d=7; adder= a + c; end
endfunction

always @ (*)
b=adder(din);

endmodule
```

## Constant function

- Constant function are functions which are used during elaboration time.
- A constant function can not modify global variables.
- A constant function can only call a constant function.
- Input argument for a constant function should either be a parameter or a constant.

# Constant function

```
function [31: 0] log2 (input [31: 0] data);
reg [31: 0] temp;
  begin
  temp=0;
while (data>1)
       begin
       data=data>>1;
       temp=temp + 1;
       end
  log2=temp;
end
endfunction
```

```
module encoder (en, din, dout);

parameter in_size=8;
input [in_size-1 : 0] din;
input en;
output [(log2(in_size)) -1: 0] dout;

//function declaration
//module functionality

endmodule
```

# Tasks

- Tasks are declared with keyword task and endtask.
- Tasks can pass multiple values through output and inout ports.
- A task may have zero or more arguments of type input, output or inout.
- Delay, Event and Timing Control statements are allowed inside tasks.
- Tasks may execute in non-zero simulation time.

# Tasks

- A task can be called from a task and procedural assignment.
- A task can invoke other tasks and functions.
- A task is invoked by specifying task name followed by input, output or inout arguments.
- The argument should be specified in the same order they are declared.

## Tasks

Declaration : Style1

```
task task_name;
//input, output or inout declarations;
//local registers declaration;
  begin        //begin-end if multiple statements
    [timing_control] sequential statements;
  end
endfunction
```

- By default the size of ports and local registers is 1-bit reg.
- Both Blocking and Non-Blocking sequential statements are allowed.

## Tasks

Declaration : Style2

```
task task_name (input, output or inout declarations);
// local registers declaration;
  begin
 [timing_control] sequential statements;
  end
endtask
```

Usage:

```
task_name (input, output or inout list);
```

# Tasks-Examples

Problem Statement : Write a task which accepts 8-bit input a, b and provides 8-bit outputs ab_and, ab_xor, ab_nand

```
task task_exam1;
input [7:0] a, b;
output [7:0] ab_and, ab_xor,
 ab_nand;
  begin
    ab_and= a & b;
    ab_xor= a ^ b;
    ab_nand= ~(a & b);
  end
endtask
```

```
module task_test (a, b, c, d, e);
input [7:0] a, b;
output reg [7:0] c, d, e;

//Task declaration

always @ (*)
task_exam1(a, b, c, d, e);

endmodule
```

## Tasks-Examples

Example: inout task

```
task increment;
inout integer a;
a=a + 1;
endtask
```

Result:

```
x=1
x=7
```

```
module task_inout;
integer x=0;
//Task declaration
initial
begin
increment(x);
$display("x=%d", x );
x=6;
increment(x);
$display("x=%d", x );
end
endmodule
```

## Tasks-Examples

Example: task with no argument

```
task count;
integer total;
total=total + 1;
endtask
```

Result:
6

```
module task_test;
//Task declaration
initial
begin
count.total=3;
count;
count;
count;
$display(count.total );
end
endmodule
```

# Tasks-Examples

Example: Multiple Calls

```
task increment;
inout integer x ;
#10 x=x + 1;
endtask
```

Result:

```
0   a=4     b=8
10 a=9      b=8
12 a=9      b=10
```

```
module task_test;
integer a=4, b=8;

//Task declaration

initial
increment(a);

initial
#2 increment(b);

initial
$monitor($time, "a= %d, b=%d", a , b);

endmodule
```

# Automatic Tasks

Example: Multiple Calls

```
task automatic increment;
inout x ;
#10 x=x + 1;
endtask
```

Result:

```
0   a=4     b=8
10 a=5      b=8
12 a=5      b=9
```

```
module task_test;
integer a=4, b=8;

//Task declaration

initial
increment(a);

initial
#2 increment(b);

initial
$monitor($time, “a= %d, b=%d”, a , b);

endmodule
```

## Functions-Tasks Difference

| Functions | Tasks |
|---|---|
| Functions are used to return single value | Tasks can be provide multiple values |
| Function musts have at least one input argument | Tasks may have zero or more arguments of type input, output or inout |
| Timing control statements are not allowed | Timing control statements are allowed |
| Always executes in zero simulation time | May not execute in zero simulation time |
| Only Blocking statements are allowed | Both Blocking and Non Blocking statements are allowed |
| Functions can invoke other functions only | Tasks can invoke other functions and tasks |

## Parameters

- Parameters are used to declare constants in Verilog.
- These constants can be overridden during compilation time.
- defparam keyword is used to override a parameter.
- Parameter can also be overridden using #() after the module name in a module instance.
- If both defparam and #() are used, the value of defparam will be used for overriding.

## Parameter Declaration

```
module custom_adder (a, b, cin, sum, carry);

parameter size=0;
input [size-1: 0] a, b;
input cin;
output [size-1: 0] sum;
output carry;

assign {carry, sum}= a + b + cin;

endmodule
```

## Parameter - ANSI C Style

```
module custom_and
              #(parameter size=0)
              (input [size-1: 0] a, b, output [size-1: 0] op_and);


assign op_and= a & b;

endmodule
```

## Overriding Syntax

```
defparam instance_label.parameter_name1=value1;
defparam instance_label.parameter_name2=value2;


module_name #(value1, value2) label (port mapping);

module_name #(.parameter_name1(value1),
             .parameter_name1(value2))
            label (port mapping);
```

Mixture of defparam and #() can be used in a single code.

## Overriding Parameter

```
module adder_test;

defparam m0.size=4;
reg cin;  reg [m0.size-1: 0] a, b;
wire [m0.size-1: 0] sum; wire carry;

custom_adder m0 (a, b, cin, sum, carry);

always
begin
a=$random; b=$random;  c=$random; #10;
end

endmodule
```

## Overriding Parameter

```
module and_test ;

reg [m0.size – 1 : 0] a, b;
wire [m0.size -1 : 0] op_and;

custom_and #(6) m0 (a, b, op_and);

always
begin
a=$random; b=$random; #10;
end

endmodule
```