Chapter 1

BASIC VERILOG

INTRODUCTION

This chapter addresses frequently asked questions on the basics of the Verilog hardware description language. This chapter deals with FAQs on Verilog assignments, tasks, functions, parameters, and ports. These constructs form a large section of the Verilog code and interconnection in designs.

1.1 Assignments

The following section discusses the different kinds of assignments that are possible in Verilog, and what their features are.

1.1.1 What are the differences between continuous and procedural assignments?

The following table captures the differences between continuous and procedural assignments:

Table 1-1. Differences between continuous and procedural assignments

Continuous assignment	Procedural assignment
Assigns values primarily to **nets**	Assigns values primarily to **reg** variables
Variables and nets **continuously** drive values onto ports	Results of calculations involving variables and nets can be **stored** into variables

Continuous assignment	Procedural assignment
Used to infer combinatorial logic	Used to infer both storage elements like Flip-flops and latches and also combinatorial logic
Assignment occurs whenever the value on the RHS of the expression changes as a continuous process	The value of the previous assignment is held until another assignment is made to the variable
Occurs in assignments to ***wire***, ***port***, ***and net*** type	Occurs in constructs like ***always***, ***initial***, ***task***, ***function***

For example,
`wire out1 = in1 & in2;`
or
`assign out1 = in1 & in2;`
	For example,
	`always @(posedge clk)`
	`  reg1 <= in1;`
	`always @(a or b or s)`
	`  y = (s == 1) ? a : b;`

1.1.2 What are the differences between assignments in *initial* and *always constructs?*

While both ***initial*** and ***always*** constructs are procedural assignments, they differ in the following ways:

Table 1-2. Differences between initial and always blocks

initial	always
Assignments in an initial block begin to execute from time 0 in simulation, and proceed in the specified sequence.	Assignments in an always block also begin from time 0, and repeat forever as a function of the changes on the blocks sensitivity list
Execution of statements in an initial begin-end block stops when the end of the block is reached, i.e., executed only once during simulation	Execution continuously repeats from the begin to the end of the process unless held by a ***wait*** construct throughout the simulation session
Non-synthesizable construct	Synthesizable construct

For example,
`reg [1:0] out1, out2;`
`initial begin`
`  out1 = 2'b10;`
`  #5 out2 = 2'b01;`
`end`
	For example,
	`reg [1:0] out1, out2;`
	`always @(posedge clk)`
	`begin`
	`  out1 <= in1;`
	`  out2 <= out1 & in2;`
	`end`

1.1.3 What are the differences between blocking and nonblocking assignments?

While both blocking and nonblocking assignments are procedural assignments, they differ in behaviour with respect to simulation and logic synthesis as follows:

Table 1-3. Differences between blocking and nonblocking assignments

Blocking assignments	Nonblocking assignments
In a blocking assignment, the evaluation of the expression on the RHS is updated to the LHS variable autonomously based on the delay value (either 0 if no delay specified, or scheduled as a future event if a non-0 value is specified)	Nonblocking assignment to LHS is *scheduled* to occur when the next evaluation cycle occurs in simulation and not immediately. Updates are not available immediately within the same time unit
When multiple blocking assignments are present in a process, the trailing assignments are blocked from occurring until the current assignment is completed	Multiple nonblocking assignments can be scheduled to occur concurrently on the next evaluation cycle in simulation
There is a possibility of race conditions on the variables of blocking assignments if assignments happen to it from two processes concurrently	The race conditions are avoided as the updated value is assigned after evaluation
Recommended to use within combinatorial **always** blocks	Recommended to use within the sequential **always** blocks
Can be used in procedural assignments like ***initial***, ***always*** and continuous assignments to nets like ***assign*** statements	Can be used only in the procedural blocks like ***initial*** and ***always***; Continuous assignment to nets like the ***assign*** statement is not permitted
Represented by "=" operator sign between LHS and RHS	Represented by "<=" operator sign between LHS and RHS

Blocking assignments	Nonblocking assignments

For example,
 	
`initial begin`
`  reg1 = #10 2'b10;`
`  reg2 = #5 2'b01;`
`end`
 	
Starting from time 0, `reg1` will be assigned 2'b10 at time 10 units and `reg2` assigned 2'b01 at time 15 unit. Assignment to `reg2` happens after the assignment of `reg1`
	For example,
	
	`initial begin`
	`  reg1 <= #10 2'b10;`
	`  reg2 <= #5 2'b01;`
	`end`
	
	Starting from time 0, `reg2` will be assigned 2'b01 at time 5 units and `reg1` will be assigned 2'b10 at time 10 unit. Assignment to `reg2` happens earlier than `reg1`

1.1.4 How can I model a bi-directional net with assignments influencing both source and destination?

The ***assign*** statement constitutes a continuous assignment. The changes on the RHS of the statement immediately reflect on the LHS net. However, any changes on the LHS don't get reflected on the RHS. For example, in the following statement, changes to the `rhs` net will update the `lhs` net, but not vice versa.

```
wire rhs, lhs;
assign lhs = rhs;
```

System Verilog has introduced a keyword ***alias,*** which can be used only on nets to have a two-way assignment. For example, in the following code, any changes to the `rhs` is reflected to the `lhs`, and vice versa.

```
module test_alias;

wire [3:0] lhs, rhs;

alias lhs = rhs; // two way assignment

initial begin
  force rhs = 4'h2;
  $display ("lhs = %0h, rhs = %0h", lhs, rhs);
  release rhs;

  force lhs = 4'hc;
```

```
  $display ("lhs = %0h, rhs = %0h", lhs, rhs);
  release lhs;
end

endmodule // test_alias
```

Had the above ***alias*** command been ***assign***, the outputs of the above display outputs would be as follows:

lhs = 2, rhs = 2
lhs = c, rhs = z

However, with the ***alias*** command as it is, the outputs are as follows:

lhs = 2, rhs = 2
lhs = c, rhs = c

In the above example, any change to either side of the net gets reflected on the other side.

1.2 Tasks and Functions

This section discusses the different FAQs on ***task*** and ***function*** in Verilog. The section also discusses a few advancements on these constructs in System Verilog.

1.2.1 What are the differences between a *task* and a *function*?

Both ***tasks*** and ***functions*** in Verilog help in executing common procedures from different places in a module. They help in writing cleaner and maintainable code, by avoiding replication at different places in a module. Essentially, ***functions*** and ***tasks*** provide a "subroutine" mechanism of reusing the same section of code at different places in a module. This allows for easier maintenance of the code.

However, the ***tasks*** and ***functions*** differ in the following aspects:

Table 1-4. Differences between tasks and functions

task	***function***
Can contain time control statements like @(***posedge*** .), delay operator (#)	Executes in zero simulation time

task	*function*
Can call any number of function's or tasks within itself	Can call any number of ***function***'s within itself
Cannot return any value when called; instead the ***task*** can have output arguments	Returns a single value when called. In SytemVerilog the return value can be optionally voided
For example, `gt_result` is an output of a ***task*** to calculate the result of the greater of two input arguments `arg1` and `arg2` `greater_val(arg1, arg2, gt_result);`	For example, `gt_result` is assigned the return of a ***function*** call to calculate the result of the greater of two input arguments `arg1` and `arg2` `gt_result = greater_val(arg1, arg2)`

1.2.2 Are *tasks* and *functions* re-entrant, and how are they different from static task and function calls? Illustrate with an example.

In Verilog-95, tasks and functions were not re-entrant. From Verilog version 2001 onwards, the ***tasks*** and ***functions*** are reentrant. The reentrant ***tasks*** have a keyword ***automatic*** between the keyword ***task*** and the name of the ***task.*** The presence of the keyword ***automatic*** replicates and allocates the variables within a ***task*** dynamically for each task entry during concurrent ***task*** calls, i.e., the values don't get overwritten for each ***task*** call. Without the keyword, the variables are allocated statically, which means these variables are shared across different ***task*** calls, and can hence get overwritten by each ***task*** call.

The following example illustrates the effect of the keyword ***automatic*** for re-entrant tasks. This is a non-synthesizable code for the purpose of illustration only.

```
module modify_taskval;

integer out_val;

task automatic modify_value;
  input [1:0] in_value;
  output [3:0] out_value;
  reg [1:0] my_value;
begin
// syntax error to use nonblocking assignment with
```

```
// automatic variables
  my_value = in_value; // blocking assignment
#5
  $display("my_value = \t%0d, t = %0d",
           my_value, $time);
  out_value = my_value + 2;
end
endtask

initial begin
  fork
    begin // First parallel call
      #1
      $display("in1= \t\t%0d, t = %0d",2, $time);
      modify_value(2, out_val);
    end
    begin // Second parallel call
      #2
      $display("in2 = \t\t%0d, t = %0d",3, $time);
      modify_value(3, out_val);
    end
  join
end

endmodule
```

In the above example, `my_value` is a local variable in the ***task*** `modify_value`. Whenever this ***task*** is called, the input `in_value` is assigned to the local variable after 5 simulation timeunits. Within the ***initial-begin***, there is a ***fork-join***, which launches two parallel processes. One starts after simulation timeunit #1, and other after #2. The first process assigns a value of 2 to the output of the task, and the second one assigns a value of 3 to the output of the task. Running the simulation with the above code, but without the ***automatic*** keyword, provides the following display:

```
in1 =        2, t = 1  // passed value is 2
in2 =        3, t = 2
my_value =   3, t = 6  // retained value is 3
my_value =   3, t = 7
```

The sequence of events without the keyword ***automatic*** is as follows:

1. The launch of the two processes from the ***fork-join*** happens from time 0.
2. The first process calls `modify_value` after #1, and the local variable `my_value` is assigned the value 2. This happens at t=1.
3. The second process calls `modify_value` after #2 and the local variable `my_value` is assigned the value 3. This happens at t=2. Note that the earlier value of 2 assigned to the local variable `my_value` is now overwritten with the value 3.
4. After 4 more time units i.e., at t = 1+5=6, the display of the first ***task*** call becomes active. Since the latest value is now "3", based on the previous step, the value of "3" is displayed for `my_value`, instead of what was passed as "2".
5. Similarly, for the second process i.e., 2+5=7, the display of the second ***task*** call becomes active. Since the latest value is still "3", the value of "3" is displayed for `my_value` here too.

The critical replacement happened in step 3 above, wherein the launch of the 2nd process actually overwrote the value of the first process *before* its turn to display. This occurred because without the ***automatic*** keyword, the variables within the task were ***static***, and shared by all calls to the ***task.***

Now, with the keyword ***automatic*** between the ***task*** and ***task*** name, the following is the output:

```
in1 =        2, t = 1 //passed value is 2
in2 =        3, t = 2
my_value =   2, t = 6 //passed value 2 preserved
my_value =   3, t = 7
```

Following the same steps as above, this time, due to the presence of the keyword ***automatic,*** the unique values of the variables are preserved in each call, and not overwritten by the subsequent ***task*** calls before the variable is being used.

The same explanation holds true for recursive ***function*** calls where a function calls itsef, with the placement of keyword ***automatic*** between ***function*** and the function name.

Note that the keyword ***automatic*** has influence only within the current hierarchy of the concurrent ***task*** calls. The same ***task*** called within separate module hierarchy doesn't overlap, and hence the need for ***automatic*** construct doesn't exist for that scenario.

The following table summarizes the differences between a reentrant ***task*** from a static ***task*** call:

Table 1-5. Differences between reentrant and static tasks

Reentrant task	Static task
Has the keyword ***automatic*** between the ***task*** keyword and identifier	*Doesn't* have the keyword ***automatic*** between the ***task*** keyword and the identifier
Variables declared within the task are allocated dynamically for each concurrent task call	Variable declarations within the task are allocated statically
All variables will be replicated in each concurrent call to store state specific to that invocation	Each concurrent call to the task will OVERWRITE the statically allocated local variables of the task from all other concurrent calls to the task
Variables declared are de-allocated at the end of task invocation	Variables retain their values between invocations
Task items cannot be accessed by hierarchical inferences	Task items can be accessed by hierarchical inferences
Task items shall be allocated new across all uses of the task executing concurrently	Task items can be shared across all uses of the task executing concurrently

1.2.3 How can I override variables in an automatic task?

By default, all variables in a ***module*** are static, i.e., these variables will be replicated for all instances of a module. However, in the case of ***task*** and ***function,*** either the ***task/function*** itself or the variables within them can be defined as ***static*** or ***automatic.*** The following explains the inferences through different combinations of the ***task/function*** and/or its variables, declared either as ***static*** or ***automatic***:

1. No ***automatic*** definition of ***task/function*** or its variables

This is the Verilog-1995 format, wherein the ***task/function*** and its variables were implicitly ***static.*** The variables are allocated only once. Without the mention of the ***automatic*** keyword, multiple calls to ***task/function*** will override their variables.

2. ***static task/function*** definition

System Verilog introduced the keyword ***static***. When a ***task/function*** is explicitly defined as ***static***, then its variables are allocated only once, and can be overridden. This scenario is exactly the same scenario as before.

3. ***automatic task/function*** definition

From Verilog-2001 onwards, and included within SystemVerilog, when the ***task/function*** is declared as ***automatic***, its variables are also implicitly ***automatic***. Hence, during multiple calls of the ***task/function***, the variables are allocated each time and replicated without any overwrites.

4. ***static task/function*** and ***automatic*** variables

SystemVerilog also allows the use of ***automatic*** variables in a ***static task/function***. Those without any changes to ***automatic*** variables will remain implicitly ***static***. This will be useful in scenarios wherein the implicit static variables need to be initialised before the ***task*** call, and the ***automatic*** variables can be allocated each time.

5. ***automatic task/function*** and ***static*** variables

SystemVerilog also allows the use of ***static*** variables in an ***automatic task/function***. Those without any changes to ***static*** variables will remain implicitly ***automatic***. This will be useful in scenarios wherein the static variables need to be updated for each call, whereas the rest can be allocated each time.

1.2.4 What are the restrictions of using *automatic* tasks?

The following are the restrictions of using ***automatic*** tasks:

- Only blocking assignments can be used on ***automatic*** variables. Refer to the earlier FAQ 1.2.2 for an example on this.

- The variables in an ***automatic task*** shall not be referenced by procedural continuous assignments or procedural force statements. In the following code, the variable `my_value` in the ***task*** cannot be referenced by an ***assign*** statement.

```
task automatic modify_value;
  input [1:0] in_value;
  reg [1:0] my_value;
begin
  my_value = in_value;
end
endtask

initial begin
  force modify_value.my_value = 1; // not allowed
  $monitor (modify_value.my_value); //not allowed
end
```

- They shall not be traced by system calls like ***$monitor*** and ***$dumpvars*** as illustrated in the above example.

1.2.5 How can I call a function like a task, that is, not have a return value assigned to a variable?

Until Verilog 2001, any ***function*** call must return a value to the type ***reg***, ***integer***, ***real***, ***time or realtime*** and the code calling the ***function*** must receive the return value. For example, the following is a syntax error:

```
function my_funct;
. . .
endfunction

initial begin
  my_funct(..); // MUST have a destination
end
```

The line in the above example is a syntax error, since the call of `my_funct` does not have a destination. Only a ***task*** can be called without a destination value.

SystemVerilog has introduced a construct ***void*** to facilitate a voided ***function*** call, that is, there is no destination for the ***function*** call. This would make a ***function*** call similar to a ***task*** call. With System Verilog, functions can also have output and inout arguments. The following example illustrates a voided ***function*** call:

```
module func_1bit;

reg [31:0] int_result;     // Global variable
```

```
function void my_func;
  input [31:0] in1;
  input [31:0] in2;
  output [31:0] out1;
// no need to assign the function
// my_func = in1 + in2;
  int_result = in1 + in2;
endfunction

initial begin
  my_func(3,4,int_result);//no destination required
  $display("int_result = %0d",int_result);
end

endmodule
```

The above example displays the result of `int_result = 7`. Some key observations in the above example are:

- The assignment to the ***function*** `my_func` was not required, since its return value is ***void***.
- The 32 bit return range between the keyword ***function*** and `my_func` was also not required, since it is now a ***void*** return.
- The call of the ***function*** `my_func` within the ***initial-begin-end*** does not require a destination, since the return has been voided.
- Some other intermediate variable like `int_result` declared in the above example at the scope of that ***module*** can still be modified within the voided ***function***.
- SystemVerilog also allows functions with a return to be called as a task by casting the function call to void. For example:

```
initial
  void (my_func(...));
```

1.2.6 What are the rules governing usage of a Verilog *function*?

The following rules govern the usage of a Verilog ***function*** construct:

- A ***function*** cannot advance simulation-time, using constructs like #, @. etc.
- A ***function*** shall not have nonblocking assignments.
- A ***function*** without a range defaults to a one bit ***reg*** for the return value.

- It is illegal to declare another object with the same name as the ***function*** in the scope where the function is declared.

1.3 Parameters

The following section discusses a few questions about the usage of parameters, pros and cons of the different approaches and what's new in System Verilog regarding parameters.

1.3.1 How can I override a module's *parameter* values during instantiation?

If a Verilog module uses parameters, there are two ways to override its values. Note that only parameters can be overridden. The localparam and specparam parameters cannot be overridden.

1.3.1.1 During instantiation

In this method, the new values are assigned inline during module instantiation. There are two ways to override during instantiation.

1.3.1.1.1 Assignment by ordered list

In this method, the order in which the parameters are assigned follow the order in which they are declared within the module. For example, the module `parameter_list` contains two parameters, that is, `width` and `depth`, that have been assigned default values within the module. It is instantiated in the following module, `example_parameter_list`, with examples of these parameters overridden with different values in different instantiations.

```
module parameter_list (addr, data); //1995 format
parameter width = 32;
parameter depth = 64;
parameter num_buses = 44;
input  [width-1 : 0] addr;
input  [depth-1 : 0] data;
...
endmodule
```

The same example above can be represented in the Verilog 2001 in the following format, in which the ***parameter*** declarations between the module and ***input/output*** declaration are now declared before the ***module*** port list.

```
module parameter_list
  # (parameter width = 32, // 2001 format
     parameter depth = 64,
     parameter num_buses = 4)
    (addr, data);
input  [width-1 : 0] addr;
input  [depth-1 : 0] data;
...
endmodule

module example_ordered_list;
reg [127 : 0] a;
reg [255 : 0] b;
reg [ 63 : 0] c;
reg [31 : 0] d;

// Instantiating parameter_list module and
// overriding width only
parameter_list #(128) U0 (a, c);

// Instantiating parameter_list module and
// overriding width and depth only
parameter_list #(128, 256) U1 (a, b);

// Instantiating parameter_list module and
// overriding num_buses only
parameter_list #(32, 256, 8) U2 (d, b);

endmodule
```

The restriction of using the above method is:

- The parameter override values have to be contiguous, that is, any ***parameter*** cannot be skipped during override. For example, in the above code with `U2` instantiation, the ***parameter*** `width` and `depth` cannot be skipped while trying to override `width` and `num_buses` only.

Two methods to overcome this restriction are:

- Precede the order of declaring the parameters within the module with the ones that will change, placing the subset that doesn't change later in the order. For example, in the above code with `U0` and `U1` instantiations, the `num_buses` was not required to be changed, and was last in the priority. The default value of 4, assigned to it within the module, will hold true in these two instantiations.

- Assign values to ALL the parameters, including the ones that don't need to be changed. In instantiation `U2`, although only the `num_buses` ***parameter*** needed to be changed, but the `width` and `depth` ***parameter***'s still required to be assigned with the same default value as in the module definition.

1.3.1.1.2 Assignment by name

This is a new feature, available from Verilog-2001 onwards. This is a better approach of overriding the module ***parameter*** by which the parameters are overridden by explicitly specifying the ***parameter*** name and its overriding value. This way, the ***parameter*** value is linked to its name, and not position of declaration.

Using the same module `parameter_list` as defined above, the following example shows the same ***parameter*** overriding, this time specifying by name.

```
module example_by_name;
reg [127 : 0] a;
reg [255 : 0] b;
reg [63 : 0]  c;
reg [31 : 0]  d;

// Instantiating parameter_list module and
// overriding width only
parameter_list #(.width(128)) U0 (a, c);

// Instantiating parameter_list module and
// overriding width and depth
parameter_list #(.width(128), .depth(256))
U1 (a, b);
```

```
// Instantiating parameter_list module and
// overriding depth only
parameter_list #(.depth(256)) U2 (d, b);

endmodule
```

Note that explicit ***parameter*** names were followed by their overriding values in the parenthesis. In the case of U2, just specifying the depth was sufficient, without having to specify anything for width parameter.

1.3.1.2 Using *defparam*

In this method, the ***parameter*** within a ***module*** is accessed by its hierarchical name from anywhere within the scope of the hierarchy. In the following example, the lower level module parameter_list gets instantiated in the example_defparam module. But the values of width and depth are overridden using the ***defparam*** construct.

```
module example_defparam;
reg [127 : 0] a;
reg [255 : 0] b;
reg [63  : 0] c;
reg [31  : 0] d;

// Instantiating parameter_list module and
// overriding width only
parameter_list U0 (a, c);
defparam U0.width = 128;

// Instantiating parameter_list module and
// overriding width and depth
parameter_list U1 (a, b);
defparam U1.width = 128;
defparam U1.depth = 256;

// Instantiating parameter_list module and
// overriding depth only
parameter_list U2 (d, b);
defparam U2.depth = 256;

endmodule
```

The following bullet items summarize the advantages of using the ***defparam*** approach:

- The ordered sequence need not be maintained in overriding the ***parameter*** values.
- A specific ***parameter*** can be overridden rather than re-specifying all the parameters prior to the one that's being overridden.
- Can help with code maintenance by grouping all the ***defparam***'s collectively in a single place, which can be compiled with the rest of the code.

Parameter redefinition at instantiation is the recommended style by most expert Verilog users. There are several reasons to avoid using ***defparam*** for parameter redefinition. Some of the reasons are:

1. The ***defparam*** statements if not collectively present in one place, can be buried in any module, anywhere in the design hierarchy, making code difficult to maintain or reuse (a form of spaghetti code, which should always be avoided).
2. Since the ***defparam*** statements can be buried anywhere in the hierarchy, they can prevent the Verilog language compilers from being able to do true independent compilation of the modules.
3. Since multiple ***defparam*** statements can be made to the same parameter instance, the final value of the parameter in this situation can (and probably will be) different with different tools.
4. The ***defparam*** statements are not supported in the official IEEE 1364.1-2002 synthesis subset for Verilog
5. The IEEE 1364 standards committee is considering a proposal to deprecate ***defparam*** in the next version of the Verilog standard, making the ***defparam*** an obsolete construct.

1.3.2 What are the rules governing *parameter* assignments?

The rules governing the ***parameter*** assignments are as follows:

- The ***parameter*** override at instantiation can be done either by specifying an ordered list or by name, but not a mix of both. For example, the following is an incorrect way of specifying both `width` and `depth`.

```
parameter_list (128, .depth(256)) U_wrong (a,b);
```

- While assigning the ***parameter*** during instantiation, once a ***parameter*** has been assigned a value, there cannot be another assignment to the same ***parameter***. For example, specifying the `width` ***parameter*** twice within the same instantiation is illegal.

```
parameter width = 64;
parameter width = 128;
// Specifying the same parameter more than once
   // is an Error
```

- If a ***parameter*** is assigned both by a ***defparam*** and in the module's instantiation, the ***defparam***'s assignment takes precedence. In the following example, the `width` ***parameter*** is instantiated with value 128, but a ***defparam*** to the same ***parameter*** with the value 64 also follows it, then the ***defparam*** gets precedence, and `width` will finally have the value 64.

```
parameter_list #(128) U1 (a, b);
defparam U1.width = 64; // This statement "wins"
```

1.3.3 How do I prevent selected *parameters* of a module from being overridden during instantiation?

If a particular ***parameter*** within a module should be prevented from being overridden, then it should be declared using the ***localparam*** construct, rather than the ***parameter*** construct. The ***localparam*** construct has been introduced from Verilog-2001. Note that a ***localparam*** variable is fully identical to being defined as a ***parameter,*** too. In the following example, the ***localparam*** construct is used to specify `num_bits`, and hence trying to override it directly gives an error message.

```
module localparam_list (addr, data);
parameter width = 32;
parameter depth = 64;
localparam num_bits = width * depth;
input  [width-1 : 0] addr;
input  [depth-1 : 0] data;
...
endmodule
```

Note, however, that, since the `width` and `depth` are specified using the ***parameter*** construct, they can be overridden during instantiation or using ***defparam***, and hence **will indirectly override** the `num_bits` values.

In general, ***localparam*** constructs are useful in defining new and localized identifiers whose values are *derived* from regular ***parameter***s.

1.3.4 What are the differences between using `*define*, and using either *parameter* or *defparam* for specifying variables?

Both `***define*** and ***parameter*** constructs can be used to specify constants in the design. For example, the `width` ***parameter*** can be specified either as a `***define*** or ***parameter***, as:

```
`define width 64
if (`width == 64) ...
or
parameter width 64;
if (width == 64) ...
```

However, the following are a few differences in using the two constructs:

Table 1-6. Differences between `define and parameter/defparam

`***define***	***parameter/defparam***
`***define*** is basically a text substitution macro	***Parameter*** is used to specify constants in a design
Multiple `***defines*** to the same variable name are not allowed, the final value of the macro is determined by source code order	Although multiple ***parameter*** definitions to the same variable are not allowed within a module, multiple ***defparam***'s to the same variable are allowed, however the final value of the parameter is indeterminate
Cannot be overridden in any mechanism	***Parameter*** can be overridden
Only one constant with the given name can exist in the full scope	Multiple modules can have the same ***parameter*** name, as it is limited to that scope only

1.3.5 What are the pros and cons of specifying the parameters using the *defparam* construct vs. specifying during instantiation?

The advantages of specifying parameters during instantiation method are:

- All the values to all the parameters don't need to be specified. Only those parameters that are assigned the new values need to be specified. The unspecified parameters will retain their default values specified within its module definition.
- The order of specifying the ***parameter*** is not relevant anymore, since the parameters are directly specified and linked by their name.

The disadvantage of specifying ***parameter*** during instantiation are:

- This has a lower precedence when compared to assigning using ***defparam***.

The advantages of specifying ***parameter*** assignments using ***defparam*** are:

- This method always has precedence over specifying parameters during instantiation.
- All the ***parameter*** value override assignments can be grouped inside one module and together in one place, typically in the top-level testbench itself.
- When multiple ***defparams*** for a single ***parameter*** are specified, the ***parameter*** takes the value of the last ***defparam*** statement encountered in the source if, and only if, the multiple ***defparam***'s are in the same file. If there are ***defparam***'s in different files that override the same parameter, the final value of the parameter is indeterminate.

The disadvantages of specifying ***parameter*** assignments using ***defparam*** are:

- The ***parameter*** is typically specified by the scope of the hierarchies underneath which it exists. If a particular module gets ungrouped in its hierarchy, [sometimes necessary during synthesis], then the scope to specify the ***parameter*** is lost, and is unspecified.

For example, if a module is instantiated in a simulation testbench, and its internal parameters are then overridden using hierarchical ***defparam*** constructs (For example, ***defparam*** `U1.U_fifo.width = 32;`). Later, when this module is synthesized, the internal hierarchy within `U1` may no longer exist in the gate-level netlist, depending upon the synthesis strategy chosen. Therefore post-synthesis simulation will fail on the hierarchical ***defparam*** override.

See the earlier FAQ 1.3.1.2 for additional disadvantages of ***defparam*** and why this construct should not be used.

1.3.6 What is the difference between the *specparam* and *parameter* constructs?

The ***specparam*** is a special kind of ***parameter*** that is intended to specify only timing and the delay values. The key differences in using the ***specparam*** and the ***parameter*** constructs are:

Table 1-7. Differences between specparam and parameter

specparam	***parameter***
Can be defined within **both** module and specify block	**Must** be defined **outside** the specify block and within module
A ***specparam*** can be assigned using another ***specparam*** or ***parameter*** or a combination of both	***Parameter*** cannot be assigned the value of a ***specparam***
Value is overridden using SDF annotation	Can be overridden during instantiation or using ***defparam***

1.3.7 What are derived parameters? When are derived parameters useful, and what are their limitations?

When one or more parameters are used to define another parameter, then the result is a derived parameter. The derived parameter can be either of the type ***parameter*** or ***localparam.*** In the following example, two parameters, `width` and `depth`, can be used to define a third parameter, `num_bits`. In this case, the `num_bits` takes a value of 32.

```
module derived_param;
parameter width = 4;
parameter depth = 8;
// num_bits is a derived parameter
```

```
localparam num_bits = width *depth;
endmodule
```

The advantages of using derived parameters are:

- Makes the RTL code reusable
- Enables use of the shorter name of `num_bits` instead of completely specifying (`width * depth`)

The consequence of using derived parameters is that derived parameters can be *indirectly* overridden by overriding their dependent parameters through ***defparam*** constructs. So, ***localparam*** constructs should be used with care when defining derived parameters.

1.4 Ports

The following section discusses a few questions about the usage of ports, pros and cons of the different approaches of port connections, and what's new in SystemVerilog regarding ports.

1.4.1 What are the different approaches of connecting ports in a hierarchical design? What are the pros and cons of each?

While instantiating the sub-modules in a given hierarchy, the port connections to those modules can be done in one of five ways:

1.4.1.1 Ordered port connection

In this method, the port expressions listed for module instance shall be in the same order as the ports listed in the ***module*** declaration, that is, the first element in the list is connected to the first port declared, the second element to the second port and so on. For example, in the code below, the `upper` module instantiates a `lower` module, and the ports are implicitly connected, that is, the connection is based on order **and** position.

```
module lower (addr, data);
input  [width-1 : 0] addr;
inout  [depth-1 : 0] data;
endmodule // lower

module upper (in1, out1);
input  [width-1 : 0] in1;
```

```
output  [depth-1 : 0] out1;

lower U0 (in1, out1); // implicit connection of
// in1 to addr and out1 to data ports

endmodule // upper
```

1.4.1.2 Named port connection

In this method, the connection between the ports can be done explicitly by linking the two names for each side of the connection, that is, the port declaration name from the module declaration can be linked to the name used in the instantiating module. The same example as above would be connected using the named port connection as follows. Note that the order of port connection is changed. However, it is recommended to keep the same order for reusability and readability.

```
lower U1 (
  .data(out1), // Order is changed,
               // connection is by name
  .addr (in1)  // only and not position
);
```

The two main advantages of this method are:

- It improves readability of the connections without having to refer to the port list of the instantiated module as the names from both sides are explicitly specified.
- The order of port connections is not relevant anymore since they are explicitly connected.

Note that the two types of module port connections *cannot* be mixed,, that is, all the connections to the ports of a particular module instance shall be either by order or by name. For example, the following is incorrect:

```
// gives a syntax error
lower U_wrong (in1, .addr(out1));
```

1.4.1.3 Implicit .* port connection

This is a feature available from SystemVerilog only. A new construct of specifying ".*" during module instantiation implicitly connects the ports of the instantiated module with the wires in the instantiating module. The precondition being the fact that the names and sizes need to be matched exactly. For example, in the following code, the upper module instantiates two lower modules. U1 and U2 The ".*" is equivalent to specifying three connections of in1, in2, and in3 between the lower and upper modules.

```
module lower (in1, in2, in3, out1, out2);

input  [7:0] in1, in2, in3;
output [7:0] out1, out2;

assign out1 = in1 & in2;
assign out2 = in1 | in3;

endmodule // lower

module upper (in1, in2, in3, u_out11, u_out12,
              u_out21, u_out22);
input  [7:0] in1, in2, in3;
output [7:0] u_out11, u_out12;
output [7:0] u_out21, u_out22;

wire [7:0] u_out11, u_out12;
wire [7:0] u_out21, u_out22;

// Instantiating lower
lower U1 (
   .*, // .* does .in1(in1), .in2(in2), .in3(in3)
  .out1 (u_out11),
  .out2 (u_out12)
);

// Instantiating lower
lower U2 (
   .*, // .* does .in1(in1), .in2(in2), .in3(in3)
  .out1 (u_out21),
  .out2 (u_out22)
```

```
);

endmodule // upper
```

The synthesized logic of the above code instantiates the two lower modules, U1 and U2. The correct port connections are also established for the ports in1, in2, and in3.

The advantage of the above method is that there is less chance of errors during instantiation, and it avoids repetition of names that implicitly match. Wherever exceptions and deviations exist, it needs to be explicitly specified. In the above, the connection to u_out11, u_out12, u_out21, and u_out22 were made explicit.

The issue in the above method is that the user will not be able to physically "see" the connections.

1.4.1.4 Implicit .name port connection

This is a feature available from SystemVerilog only. A new construct of specifying the port name only once with the ".name" convention, where the "name" is the port name. This avoids specifying the port name twice when the port name and signal name are the same. The instance port name and size should match the connecting variable port name and size during module instantiation. In the following example, the ports in1, in2 and in3 of both the instances of lower module don't have any connecting variable port name.

```
module lower (in1, in2, in3, out1, out2);

input  [7:0] in1, in2, in3;
output [7:0] out1, out2;

assign out1 = in1 & in2;
assign out2 = in1 | in3;

endmodule

module upper (in1, in2, in3, u_out11, u_out12, u_out21,
u_out22);
input  [7:0] in1, in2, in3;
output [7:0] u_out11, u_out12;
output [7:0] u_out21, u_out22;

wire [7:0] u_out11, u_out12;
```

```
wire [7:0] u_out21, u_out22;

// Instantiating lower with out2 floating
lower U1 (
          .in1 , // no variable port name to these
          .in2 , // instance port names
          .in3 ,
          .out1 (u_out11) ,
          .out2 (u_out12)
         );

// Instantiating lower with out2 floating
lower U2 (
          .in1 , // no variable port names to
          .in2 , // these instance port names
          .in3 ,
          .out1 (u_out21) ,
          .out2 (u_out22)
         );

endmodule
```

The synthesized logic of the above code instantiates the two lower modules, `U1` and `U2`. The correct port connections are also established for the ports `in1`, `in2`, and `in3`.

The advantage of the above method is that there is less chance of errors during instantiation, and it avoids repetition of names that implicitly match. Wherever exceptions and deviations exist, it needs to be explicitly specified. In the above, the connection to `u_out11`, `u_out12`, `u_out21`, and `u_out22` were made explicit.

1.4.1.5 Interface port connection

SystemVerilog has introduced a construct ***interface,*** which basically encapsulates a bundle of nets and variables into one group. When there are numerous ports that need to be connected to each other, it is easier to make the connections through the ***interface*** construct. This helps create less verbose and more maintainable code by grouping all common connections in just one place. Any future changes to the interfaces can be modified in the ***interface*** definition, and this will propagate to all the instances where this is being used. The above example is illustrated using the ***interface*** construct as follows:

```
interface basic_con;
wire [7:0] in1, in2, in3; // bi-dir wires
endinterface : basic_con

module lower (basic_con all_ins, // all inputs
              output [7:0] out1, out2);

assign out1 = all_ins.in1 & all_ins.in2;
assign out2 = all_ins.in1 | all_ins.in3;

endmodule

module upper (in1, in2, in3, u_out11, u_out12,
              u_out21, u_out22);
input  [7:0] in1, in2, in3;
output [7:0] u_out11, u_out12;
output [7:0] u_out21, u_out22;

wire [7:0] u_out11, u_out12;
wire [7:0] u_out21, u_out22;

basic_con top_ins();

assign top_ins.in1 = in1;
assign top_ins.in2 = in2;
assign top_ins.in3 = in3;

// Instantiating lower
lower U1 (
// top_ins does .in1(in1), .in2(in2), .in3(in3)
  top_ins,
  u_out11,
  u_out12
);

// Instantiating lower
lower U2 (
// top_ins does .in1(in1), .in2(in2), .in3(in3)
  top_ins,
  u_out21,
  u_out22
);
```

```
endmodule
```

In the above example, the `top_ins` is the interface instantiation that sufficed the purpose of specifying the port connection of the in1 to in3 ports. Some of the salient points of the above example are:

- The above example could also be extended to connect inter-module connections, that is, connections to-from `U1` and `U2`.
- The ***interface*** specification and the explicit port connections could be mixed during one instantiation itself.

1.4.2 Can there be full or partial no-connects to a multi-bit port of a module during its instantiation?

No. There cannot be full or partial no-connects to a multi-bit port of a module during instantiation. For example, the following instantiation with an intermediate bit left to float is illegal, and gives a syntax error:

```
// Instantiating lower with some port bits
// unconnected
lower U1 (
  .in1(u_in1),

 // bit 6 for in2 is floating in 8 bit in2
  .in2({u_in2[7], ,u_in2[5:0]}), // Error

// bits [5:3] for out1 are unconnected in 8 bit
// out1
    .out1({u_out1[7:6], , u_out1[2:0]}), // Error
    .out2(u_out2)
);
```

In the case where there is a genuine situation to not connect a particular output, then it must be connected to an unused ***wire,*** and continue the concatenation with the appropriate bits to be connected. For example, in the above situation, the following two additional declarations, and the connections shown following it is a legal syntax:

```
wire unused1;
wire [2:0] unused2;
```

```
// Instantiating lower with out2 floating
lower U1 (
  .in1(u_in1),
  .in2({u_in2[7],unused1,u_in2[5:0]}),
  .out1({u_out1[7:5],unused2[2:0],u_out1[1:0]}),
  .out2(u_out2)
);
```

Note that a floating input or an unused wire on an output will cause a "z" propagation into the logic. The outputs will drive values onto the unused wires, but these wires do not fanout to other logic, and will be optimized away by synthesis tools.

1.4.3 What happens to the logic after synthesis, that is driving an unconnected output port that is left open (, that is, no-connect) during its module instantiation?

An unconnected output port in simulation will drive a value, but this value does not propagate to any other logic. In synthesis, the cone of any combinatorial logic that drives the unconnected output will get optimized away during boundary optimisation, that is, optimization by synthesis tools across hierarchical boundaries.

In the module `lower1` is instantiated into an `upper1` module, and the same pins are connected all the way to the top level. When this code is synthesized, it will produce the logic as shown in figure 1-1.

```
module lower1 (in1, in2, out1, out2);
input  in1, in2;
output out1, out2;
assign out1 = in1 & in2 ;
assign out2 = in1 | in2 ;
endmodule

module upper1 (u_in1, u_in2,
               u_out1, u_out2);
input  u_in1, u_in2;
output u_out1, u_out2;
lower1 U1 (
  .in1 (u_in1),
  .in2 (u_in2),
  .out1 (u_out1),
```

```
   .out2 (u_out2)
);
endmodule
```

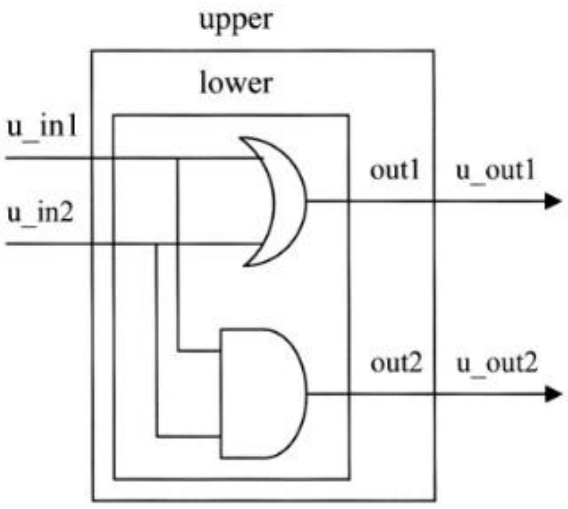

Figure 1-1. No unconnected ports

When `out1` is left unconnected during the instantiation of the lower module, (this port is not going all the way to the top level of `u_out 1`) as shown in this figure, then the logic gets optimized with only the AND gate remaining, and the OR gate getting optimized away.

Similarly, when `out2` is left unconnected during the instantiation of the lower module, the OR gate remains driving `out1` all the way to the top level, and the AND gate gets optimized away.

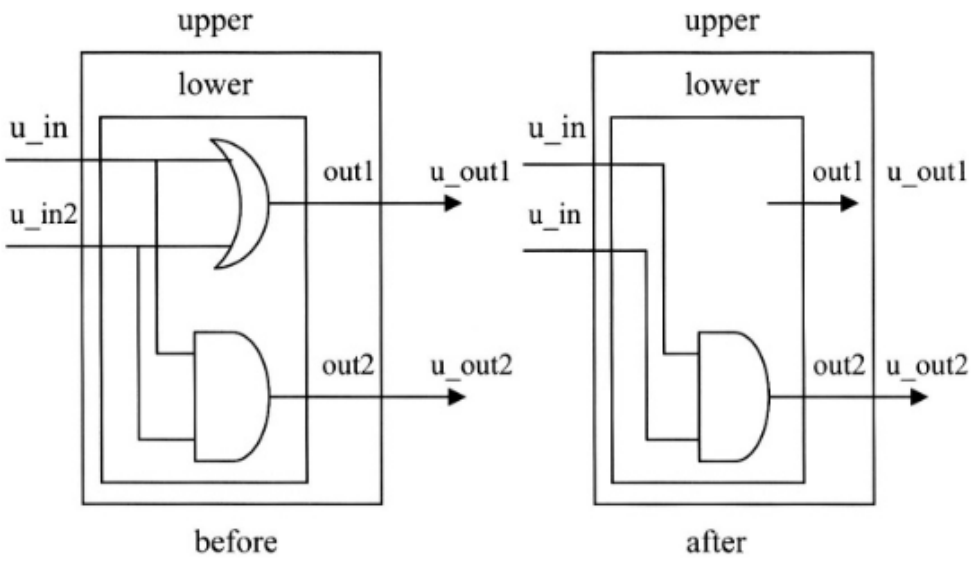

Figure 1-2. Gate eating behind an unconnected output port

1.4.4 What value is sampled by the logic from an input port that is left open (that is, no-connect) during its module instantiation?

By default, an unconnected input port is a floating port, and hence shows "z" during simulation. The logic following it will also propagate the "z", until gated off by an AND gate. The following figure shows the `in1` floating in lower instantiation.

Since `in1` was used as logic input to both the gates, and is no more driving both of them, the logic gets optimized and simplified into a simple wire connection between `in2` and `out2`. This connection still maintains the AND'ing logic required between these two ports, as per its design.

During synthesis, it is recommended to remove the unconnected ports using the synthesis tool commands, as it could potentially be undesirable during back-end processing.

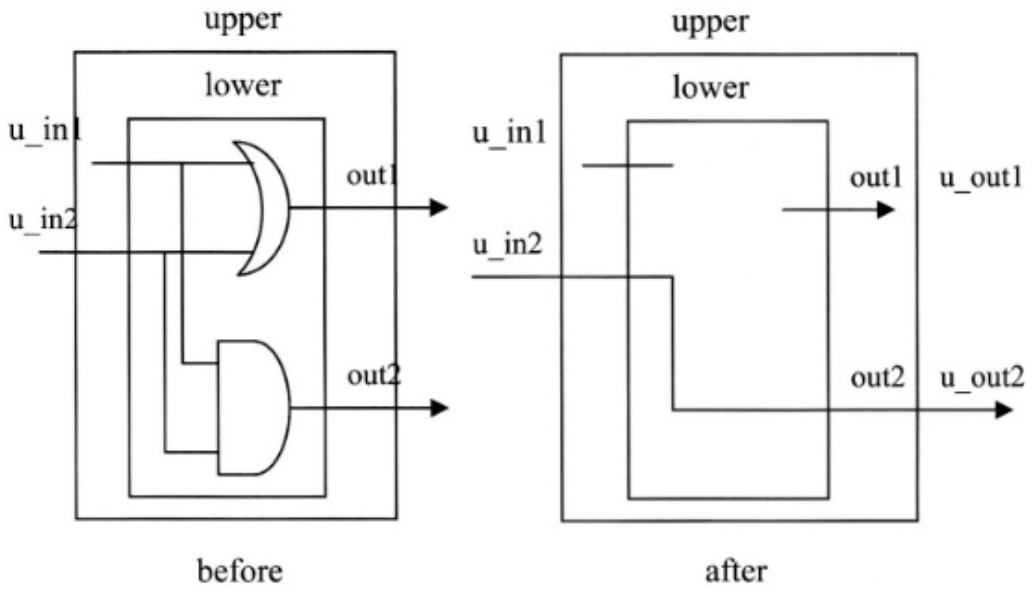

Figure 1-3. When one of the inputs is floating

The default value of z for unconnected input ports can be changed using the compiler directives:

```
`unconnected_drive pull0
and
`unconnected_drive pull1
```

The first directive causes all unconnected input ports to be pulled down to a logic 0. The second directive causes all unconnected input ports to be pulled up to logic 1. The effect of the `unconnected_drive directives can be turned off with the compiler directive `unconnected_drive. For example:

```
`unconnected_drive pull1
module ttl_74ls74 (clk, d, rst, pre, q, qb);
input clk, d, rst, pre; // will pull up if left
                        // unconnected
output q, qb;
...
endmodule
`nounconnected_drive // for the rest of the code
```

1.4.5 How is the connectivity established in Verilog when connecting wires of different widths?

When connecting wires or ports of different widths, the connections are right-justified, that is, the rightmost bit on the RHS gets connected to the rightmost bit of the LHS and so on, until the MSB of either of the net is reached. For example,

```
wire [7:0] net1;
wire [3:0] net2;

assign net1 = net2;
// implicitly net1[3:0] are connected to
// net2[3:0] and net1[7:4] is left floating

assign net2 = net1;
// The wires net1[3:0] are still connected to
// net2[3:0]
```

Note, however, that some simulation and synthesis tools will give a Warning when connecting nets or ports of dissimilar widths.

1.4.6 Can I use a Verilog *function* to define the width of a multi-bit port, *wire*, or *reg* type?

The width elements of ports, ***wire*** or ***reg*** declarations require a constant in both MSB and LSB. Before Verilog 2001, it is a syntax error to specify a ***function*** call to evaluate the value of these widths. For example, the following code is erroneous before Verilog 2001 version.

```
reg [get_high(val1, val2) : get_low(val3, val4)] reg1;
```

In the above example, `get_high` and `get_low` are both ***function*** calls of evaluating a constant result for MSB and LSB respectively.

However, Verilog-2001 allows the use of a ***function*** call to evaluate the MSB or LSB of a width declaration.