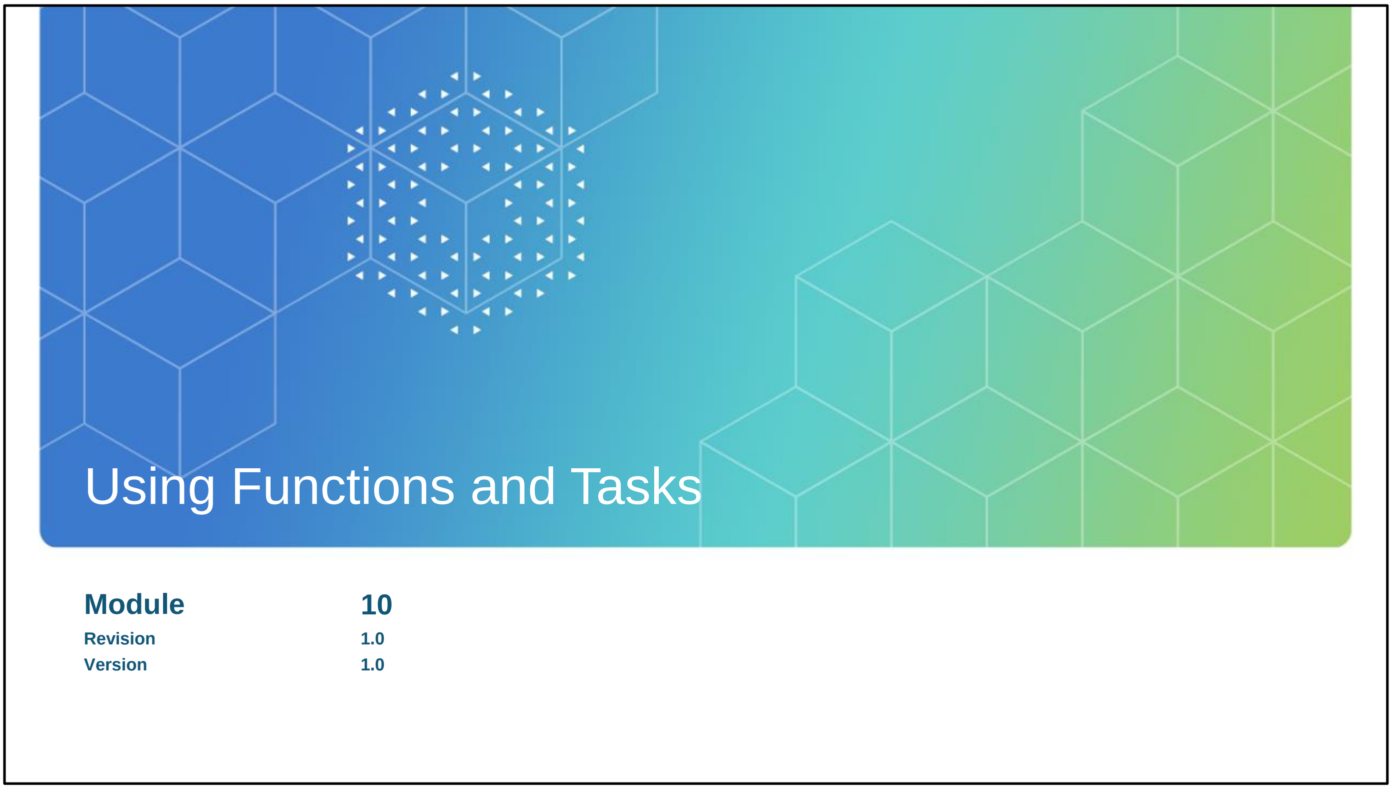
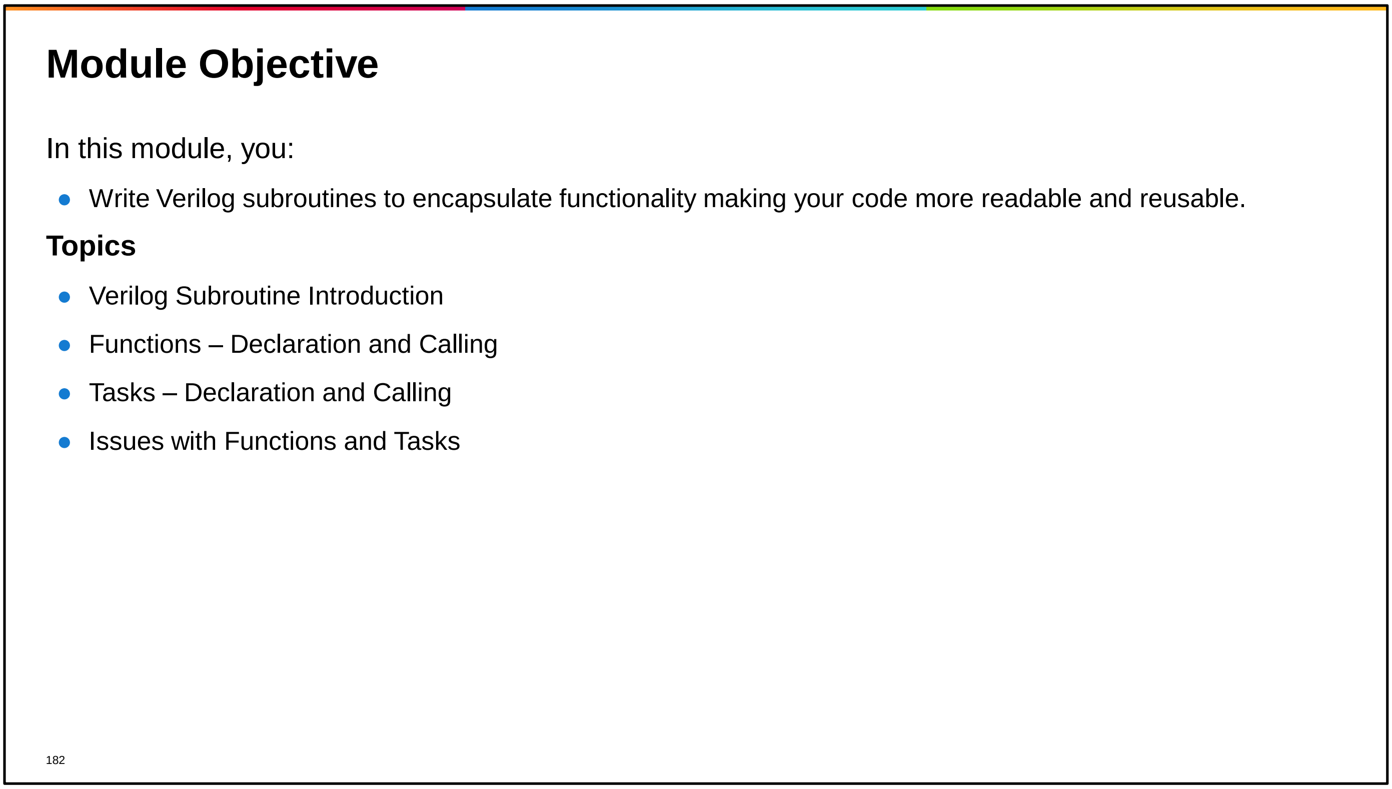


Using Functions and Tasks

Module	**10**
Revision	1.0
Version	1.0

This module describes Verilog subroutines, known as functions and tasks. It explains where and how to define them, how to invoke them, and some issues involved with using them.



Module Objective

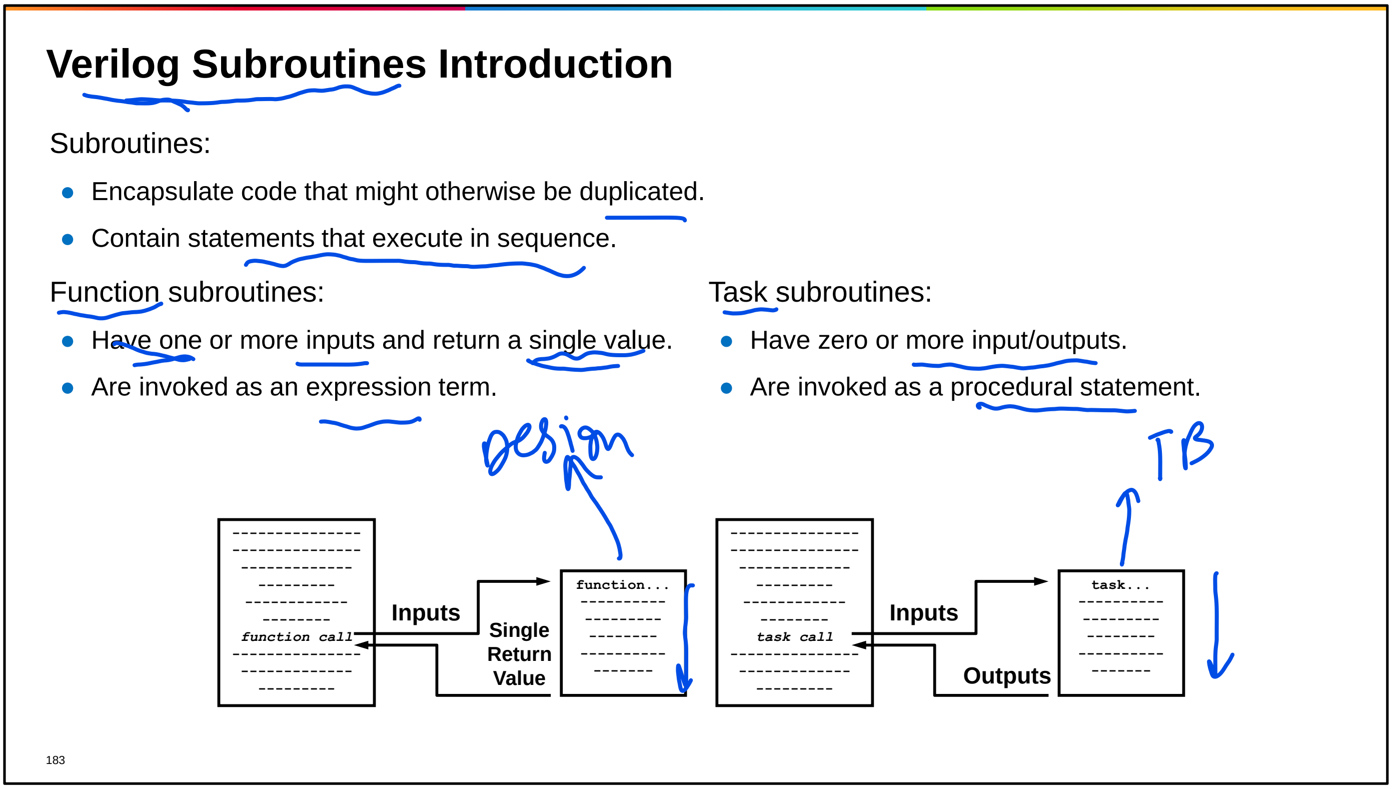
In this module, you:

- Write Verilog subroutines to encapsulate functionality making your code more readable and reusable.

Topics

- Verilog Subroutine Introduction
- Functions – Declaration and Calling
- Tasks – Declaration and Calling
- Issues with Functions and Tasks

Your objective is to effectively use subroutines to encapsulate functionality and thus to make your code more readable and reusable. To do that, you need to know what subroutines are available and how to use them.



Verilog Subroutines Introduction

Subroutines:

- Encapsulate code that might otherwise be duplicated.
- Contain statements that execute in sequence.

Function subroutines:

- Have one or more inputs and return a single value.
- Are invoked as an expression term.

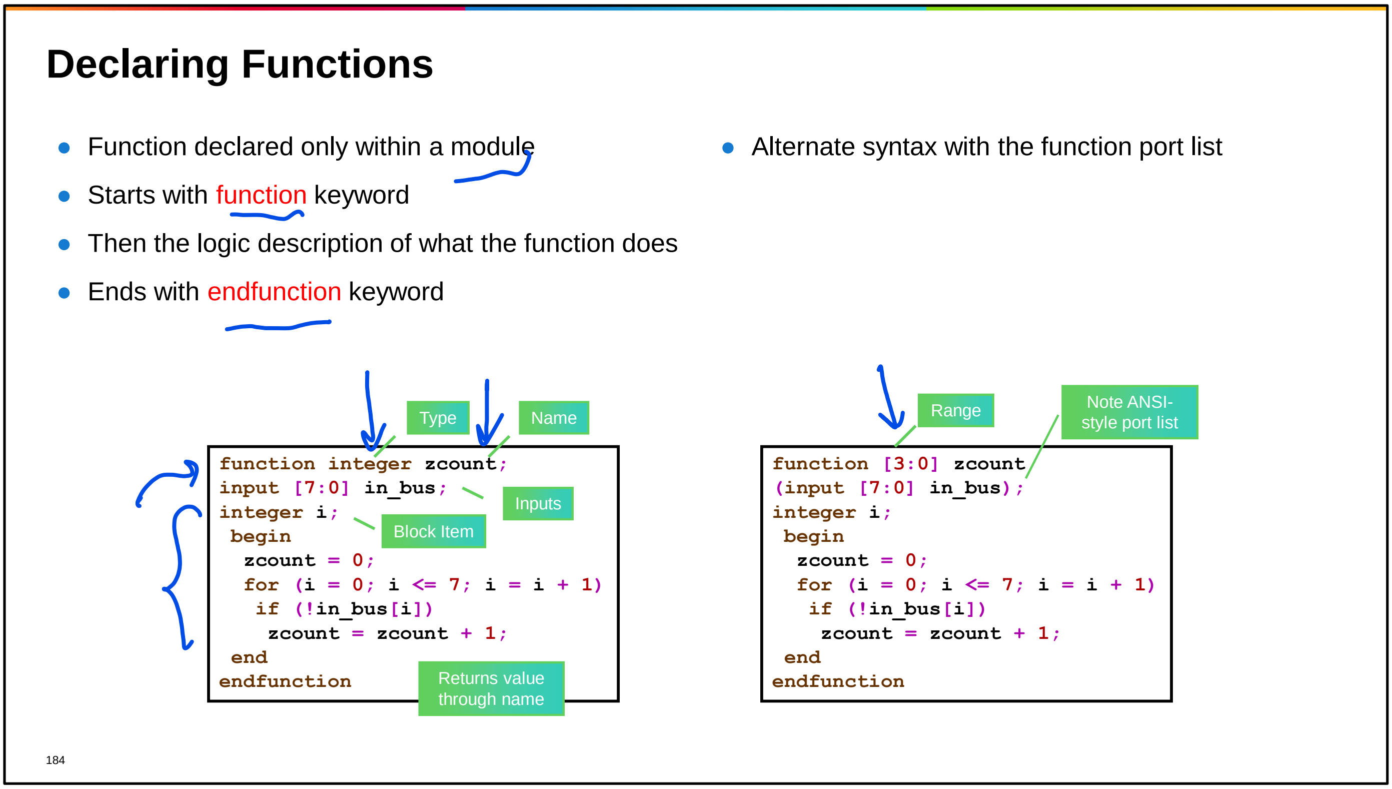
Task subroutines:

- Have zero or more input/outputs.
- Are invoked as a procedural statement.

design
TB
function call
Inputs
Single
Return
Value
function...
task call
Inputs
Outputs
task...

Let’s first examine the purpose of subroutines:

- You can think of a subroutine as a level of hierarchy, or structure, in sequential code.
- You use subroutines to encapsulate some frequently used code in one place where you can more easily manage it. You can provide this code with inputs, to which you assign values as you invoke the subroutine. These input values can flexibly direct the subroutine to perform slightly differently for each invocation.
- Encapsulating sequential code in subroutines is similar to encapsulating design units in modules.



Declaring Functions

- Function declared only within a module
- Starts with function keyword
- Then the logic description of what the function does
- Ends with endfunction keyword

- Alternate syntax with the function port list

Type

Name

```
function integer zcount;
input [7:0] in_bus;
integer i;
 begin
  zcount = 0;
  for (i = 0; i <= 7; i = i + 1)
   if (!in_bus[i])
    zcount = zcount + 1;
 end
endfunction
```

Inputs

Block Item

Returns value through name

Range

Note ANSI-style port list

```
function [3:0] zcount
(input [7:0] in_bus);
integer i;
 begin
  zcount = 0;
  for (i = 0; i <= 7; i = i + 1)
   if (!in_bus[i])
    zcount = zcount + 1;
 end
endfunction
```

Syntax

```
function [automatic] [signed] [range_or_type] function_identifier;
  tf_input_declaration;
 {tf_input_declaration;}
 {block_item_declaration}
 function_statement
endfunction
```

Syntax Alternative

```
function [automatic] [signed] [range_or_type] function_identifier
(function_port_list);
 {block_item_declaration}
 function_statement
endfunction
```

You declare a function only within a module. The declaration starts with the function keyword.

Functions are by default static. That means that exactly one copy of the function variables exists. All calls to the function utilize that one set of function variables, thus calls to static functions are generally not recursive. External code can access static function variables using hierarchical references. Verilog 2001 added the automatic declaration. Automatic functions create a new temporary set of their variables for each call. Automatic functions can be recursive. External code cannot access automatic function variables.

The return type defaults to a single bit. You can alternatively specify integer, real, realtime, or time, or a vector range. A vector is by default not signed. You can declare the vector signed. The signed declaration is a Verilog 2001 feature.

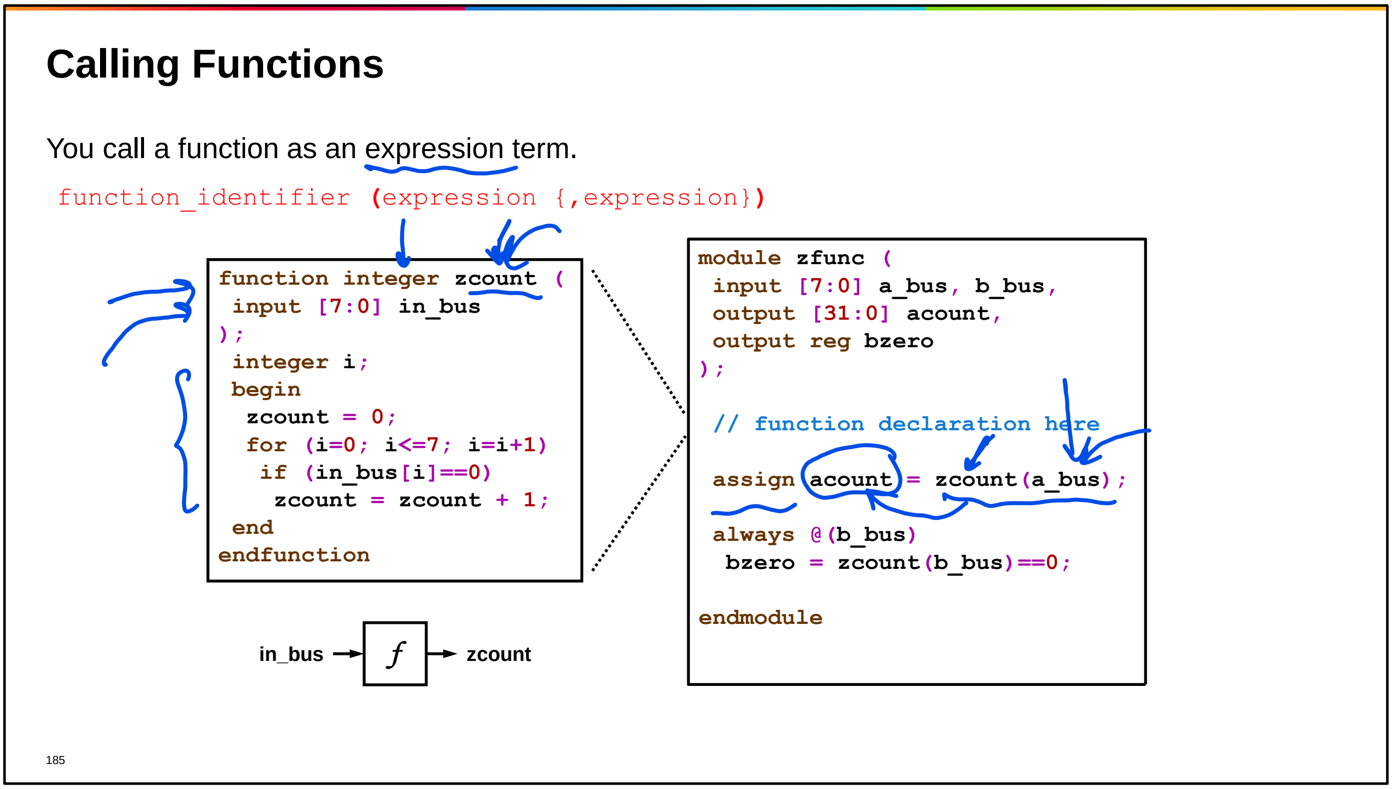
A function must have at least one input port and must have no output ports and no inout ports. You can declare the input ports using either syntax. With the function port list syntax, you can prefix port attributes. The input port type defaults to a single bit. You can alternatively specify integer, real, realtime, or time, or a vector range. A vector is by default not signed. You can declare the vector signed. The signed declaration is a Verilog 2001 feature.

You can declare additional block items. You cannot declare module items. For example, you can declare variables but not nets.

A function declaration contains a statement. The statement may be a statement block, for example grouped between the begin and end keywords. Functions cannot invoke the scheduler. That means function assignments are always blocking and functions always execute completely and instantly.

Functions can have side-effects, for example, a function can assign to a module variable but not to a module net. For a function to have side-effects is undesirable programming practice. Instead, declare the function return vector sufficiently wide to hold all the data you need to return, then deconstruct the vector value as needed upon return.

A function declaration ends with the endfunction keyword.



Calling Functions

You call a function as an expression term.

```
function_identifier (expression {,expression})
```

```
function integer zcount (
 input [7:0] in_bus
);
 integer i;
 begin
  zcount = 0;
  for (i=0; i<=7; i=i+1)
   if (in_bus[i]==0)
    zcount = zcount + 1;
 end
endfunction
```

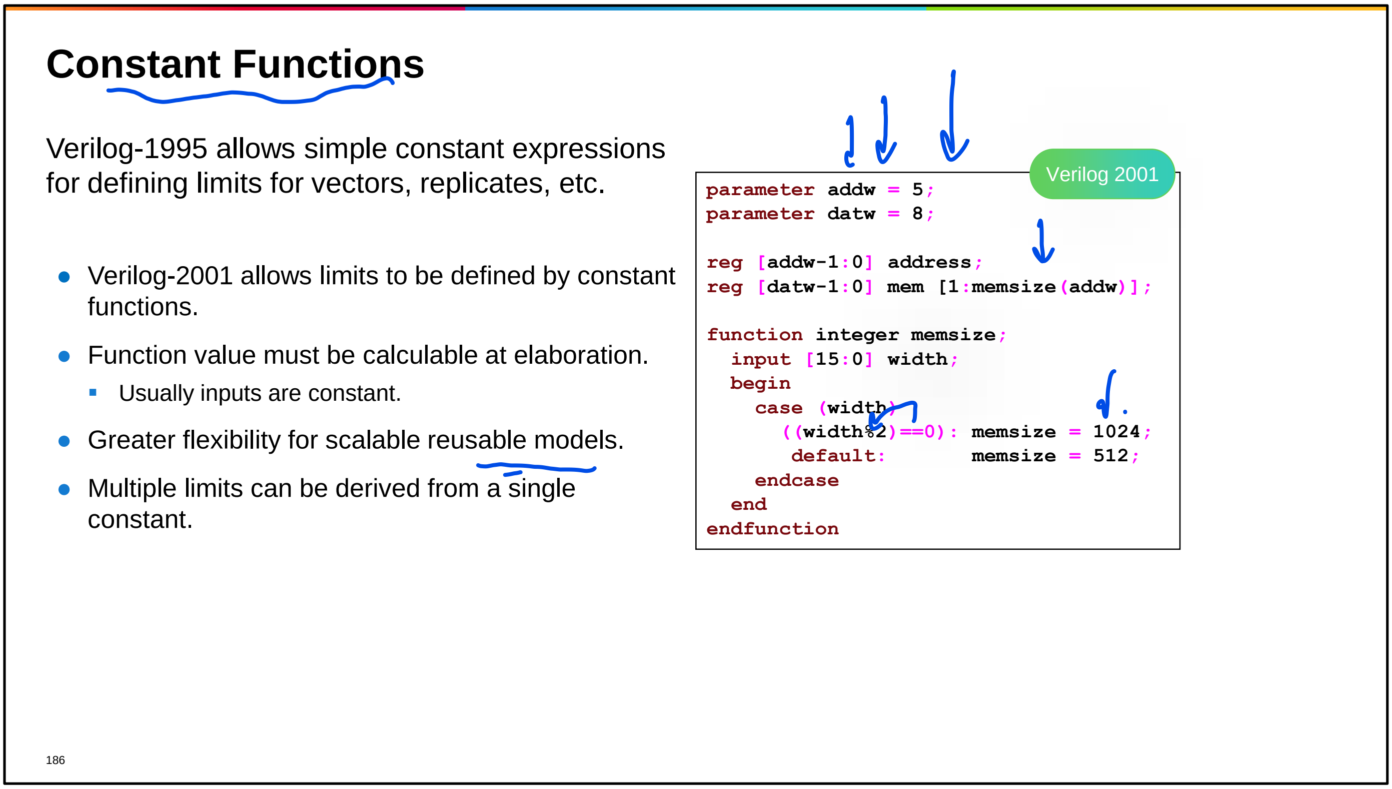
```
module zfunc (
 input [7:0] a_bus, b_bus,
 output [31:0] acount,
 output reg bzero
);

 // function declaration here

 assign acount = zcount(a_bus);

 always @(b_bus)
  bzero = zcount(b_bus)==0;

endmodule
```

in_bus → *f* → zcount

You call a function as an expression term. The simulator assigns the values of the argument expressions to the input ports in the order in which they appear in the call.

When the function completes, the simulator uses the value of the function name variable where the function call appears in the calling expression. Functions return a single value that you use as an operand in an rvalue expression. An rvalue expression is one that may appear only on the right side of an assignment – you cannot assign a value to an rvalue expression. If the function never completes, for example, it executes a loop that loops forever, it never returns and the simulation “hangs”.

In this example, the zcount function returns an integer value representing the number of bits of the in_bus port that are zero. It returns a value between 0 and 8. The continuous assignment call assigns the returned value to the “acount” module output port. The procedural assignment call uses the function return value as an operand in a comparison expression.



Constant Functions

Verilog-1995 allows simple constant expressions for defining limits for vectors, replicates, etc.

- Verilog-2001 allows limits to be defined by constant functions.
- Function value must be calculable at elaboration.
  - Usually inputs are constant.
- Greater flexibility for scalable reusable models.
- Multiple limits can be derived from a single constant.

Verilog 2001

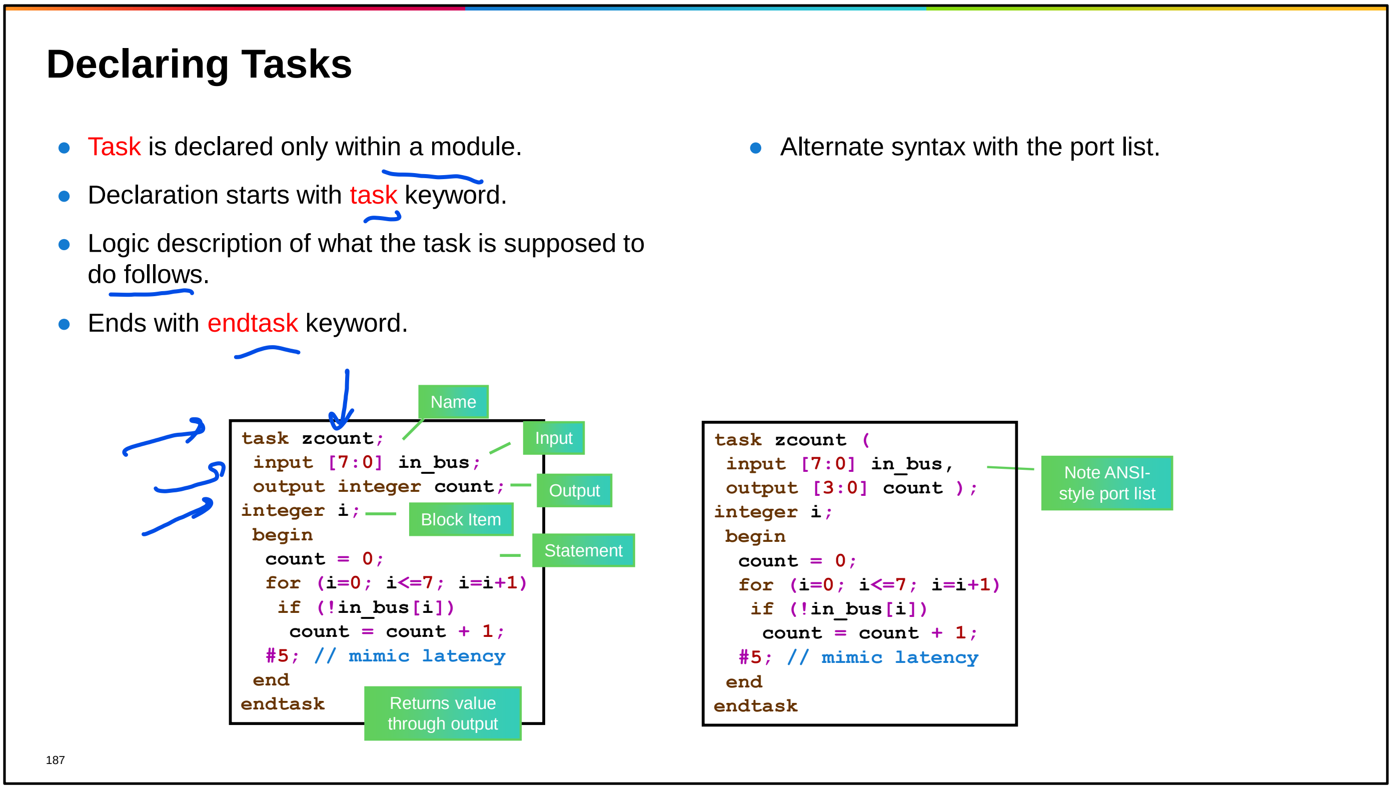
```
parameter addw = 5;
parameter datw = 8;

reg [addw-1:0] address;
reg [datw-1:0] mem [1:memsize(addw)];

function integer memsize;
  input [15:0] width;
  begin
    case (width)
      ((width%2)==0): memsize = 1024;
       default:       memsize = 512;
    endcase
  end
endfunction
```

Verilog 1995 module parameter values must be constant expressions, which are limited to operations on literals and previously declared module parameters.

Verilog 2001 permits you to use a constant function call anywhere you are required to use a constant expression.

The standard lists restrictions upon constant function calls:

- Cannot be placed within any generate scope.
- Cannot contain hierarchical references.
- Cannot contain system function calls.
- Ignores system task calls (except will execute $display in simulator but not elaborator).
- Cannot themselves make constant function calls in any context requiring constant expressions.
  - Can otherwise make constant function calls to functions local to containing module.
- Can access only functions and module parameters and nothing else declared outside the function definition.
- Module parameters must be previously assigned use of defparam can produce undefined results.



Declaring Tasks

- Task is declared only within a module.
- Declaration starts with task keyword.
- Logic description of what the task is supposed to do follows.
- Ends with endtask keyword.

```
task zcount;
 input [7:0] in_bus;
 output integer count;
integer i;
 begin
  count = 0;
  for (i=0; i<=7; i=i+1)
   if (!in_bus[i])
    count = count + 1;
  #5; // mimic latency
 end
endtask
```

Name
Input
Output
Block Item
Statement
Returns value through output

- Alternate syntax with the port list.

```
task zcount (
 input [7:0] in_bus,
 output [3:0] count );
integer i;
 begin
  count = 0;
  for (i=0; i<=7; i=i+1)
   if (!in_bus[i])
    count = count + 1;
  #5; // mimic latency
 end
endtask
```

187

You declare a task only within a module. The declaration starts with the task keyword.

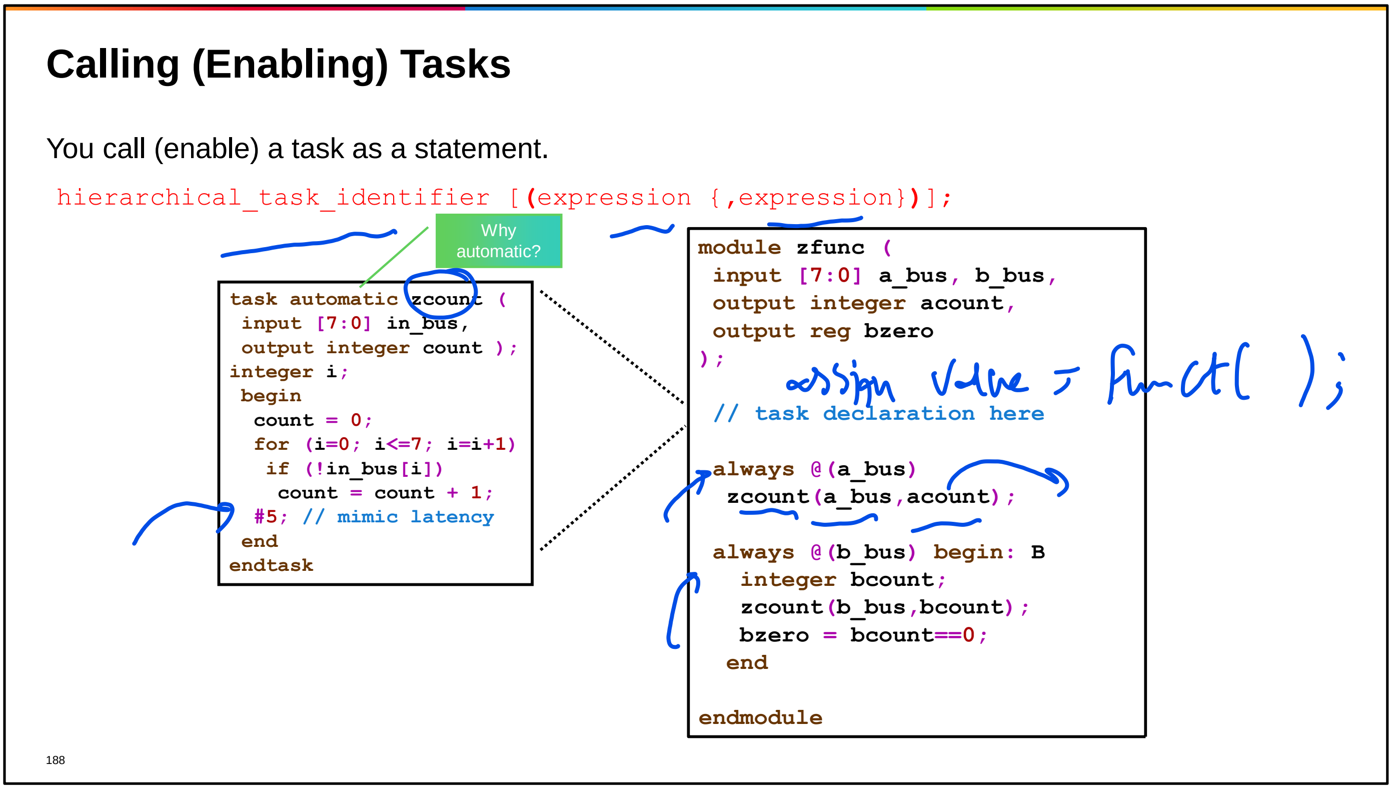
Tasks are by default static. That means that exactly one copy of the task variables exists. All calls to the task utilize that one set of task variables, thus calls to static tasks are generally not re-enterable. External code can access static task variables using hierarchical references. Verilog 2001 added the automatic declaration. Automatic tasks create a new temporary set of their variables for each call. Automatic tasks are re-enterable. External code cannot access automatic task variables.

A task can have any number of input, output or inout ports. You can declare the ports using either syntax. Either syntax accepts port attributes. The port type defaults to a single bit. You can alternatively specify integer, real, realtime, or time, or a vector range. A vector is by default not signed. You can declare the vector signed. The signed declaration is a Verilog 2001 feature.

You can declare additional block items. You cannot declare module items. For example, you can declare variables but not nets.

A task declaration contains a statement. The statement may be a statement block, for example grouped between the begin and end keywords. Tasks can invoke the scheduler. That means task assignments can be blocking or nonblocking and tasks can delay their completion for any amount of time. You can rewrite as a function any task that does not invoke the scheduler.

Tasks can have side effects, for example, a task can assign to a module variable but not to a module net. For a task to have side effects is a very useful feature and common programming practice. A task declaration ends with the endtask keyword.



Calling (Enabling) Tasks

You call (enable) a task as a statement.

```
hierarchical_task_identifier [(expression {,expression})];
```

Why automatic?

```
task automatic zcount (
 input [7:0] in_bus,
 output integer count );
integer i;
 begin
  count = 0;
  for (i=0; i<=7; i=i+1)
   if (!in_bus[i])
    count = count + 1;
  #5; // mimic latency
 end
endtask
```

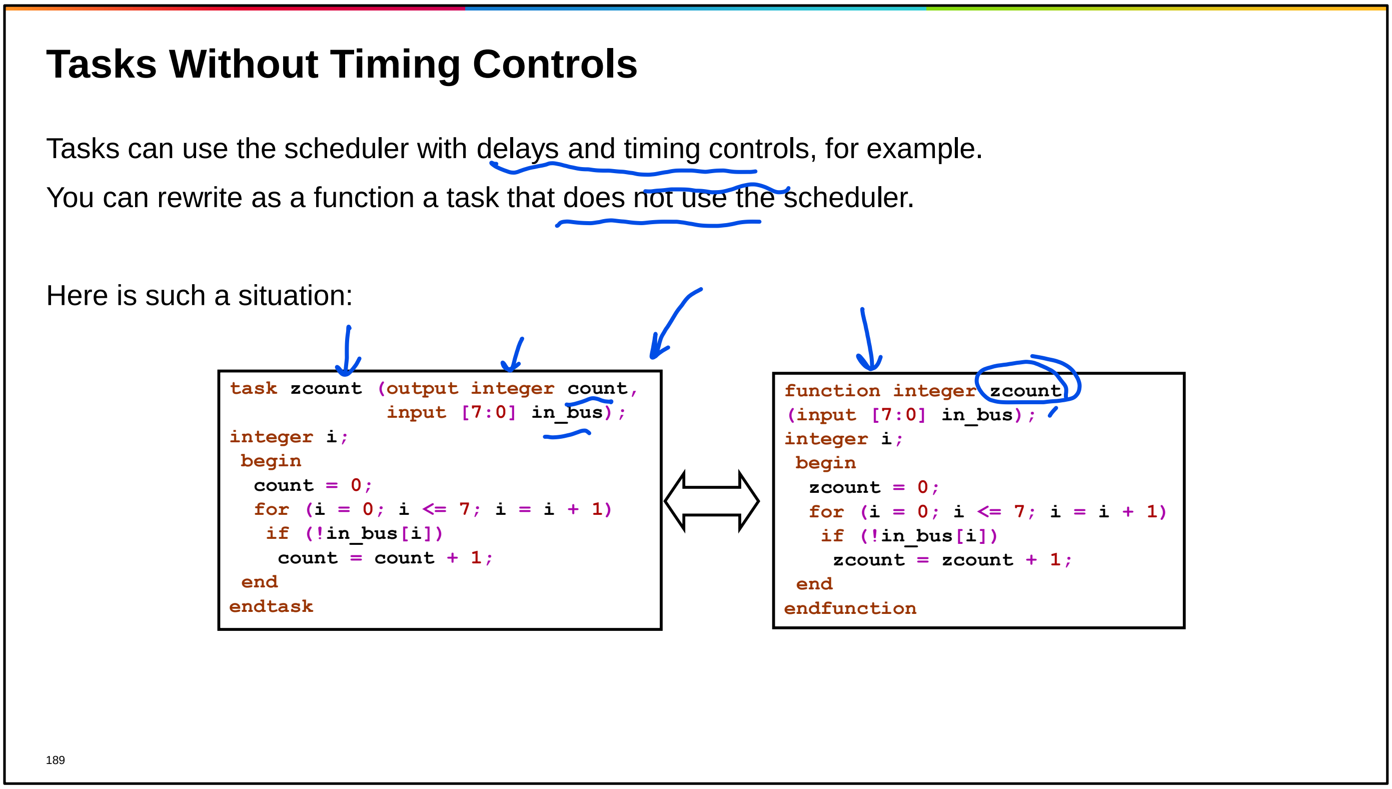
```
module zfunc (
 input [7:0] a_bus, b_bus,
 output integer acount,
 output reg bzero
);

 // task declaration here

 always @(a_bus)
  zcount(a_bus,acount);

 always @(b_bus) begin: B
   integer bcount;
   zcount(b_bus,bcount);
   bzero = bcount==0;
  end

endmodule
```

assign value = funct( );

You call a task as a procedural statement. The simulator assigns the values of the input and inout argument expressions to their ports. The standard refers to these “calls” as enables to remind you that a task can consume simulation time and thus can execute concurrently with other procedures.

When the task completes, the simulator assigns the values of the output and inout ports to their lvalue expression arguments. An lvalue expression may appear on the left side of an assignment – you can assign a value to an lvalue expression. If the task never completes, for example, in a loop that loops forever, it never returns and its calling procedure cannot continue. However, if the task “consumes” time, procedures other than the calling procedure can execute.

In this example, the zcount task returns an integer value representing the number of bits of the in_bus port that are zero. It returns a value between 0 and 8. Both task enables are procedural statements. As both procedures can concurrently have the task enabled, the task variables must be automatic variables so that each procedure gets their own copy.



Tasks Without Timing Controls

Tasks can use the scheduler with delays and timing controls, for example.

You can rewrite as a function a task that does not use the scheduler.

Here is such a situation:

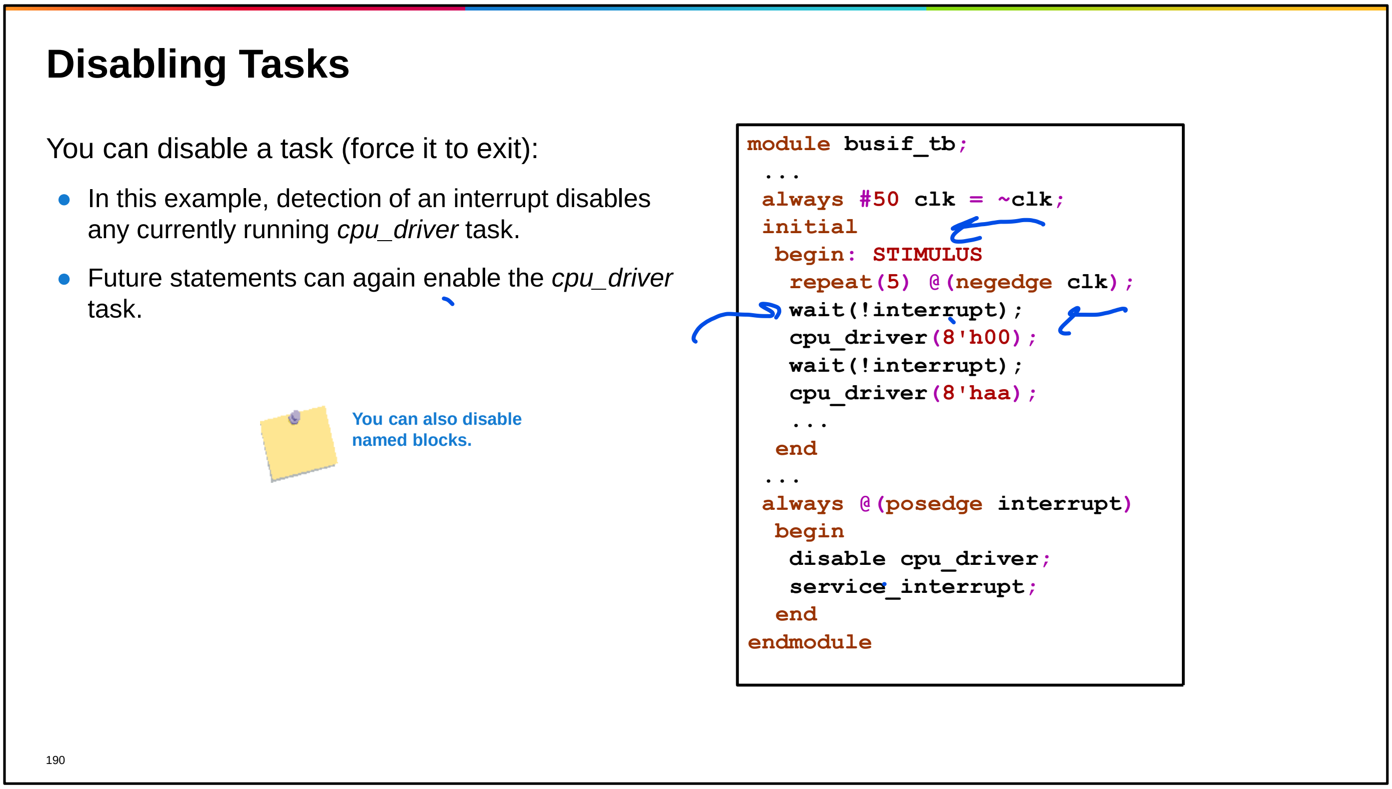
```
task zcount (output integer count,
             input [7:0] in_bus);
integer i;
 begin
  count = 0;
  for (i = 0; i <= 7; i = i + 1)
   if (!in_bus[i])
    count = count + 1;
 end
endtask
```

⇔

```
function integer zcount
(input [7:0] in_bus);
integer i;
 begin
  zcount = 0;
  for (i = 0; i <= 7; i = i + 1)
   if (!in_bus[i])
    zcount = zcount + 1;
 end
endfunction
```

189

Tasks can invoke the scheduler. That means task assignments can be blocking or nonblocking and tasks can delay their completion for any amount of time. A task that does not invoke the scheduler operates very much like a function. You can rewrite as a function any task that does not invoke the scheduler.



Disabling Tasks

You can disable a task (force it to exit):

- In this example, detection of an interrupt disables any currently running *cpu_driver* task.
- Future statements can again enable the *cpu_driver* task.

You can also disable named blocks.

```
module busif_tb;
 ...
 always #50 clk = ~clk;
 initial
  begin: STIMULUS
   repeat(5) @(negedge clk);
   wait(!interrupt);
   cpu_driver(8'h00);
   wait(!interrupt);
   cpu_driver(8'haa);
   ...
  end
 ...
 always @(posedge interrupt)
  begin
   disable cpu_driver;
   service_interrupt;
  end
endmodule
```

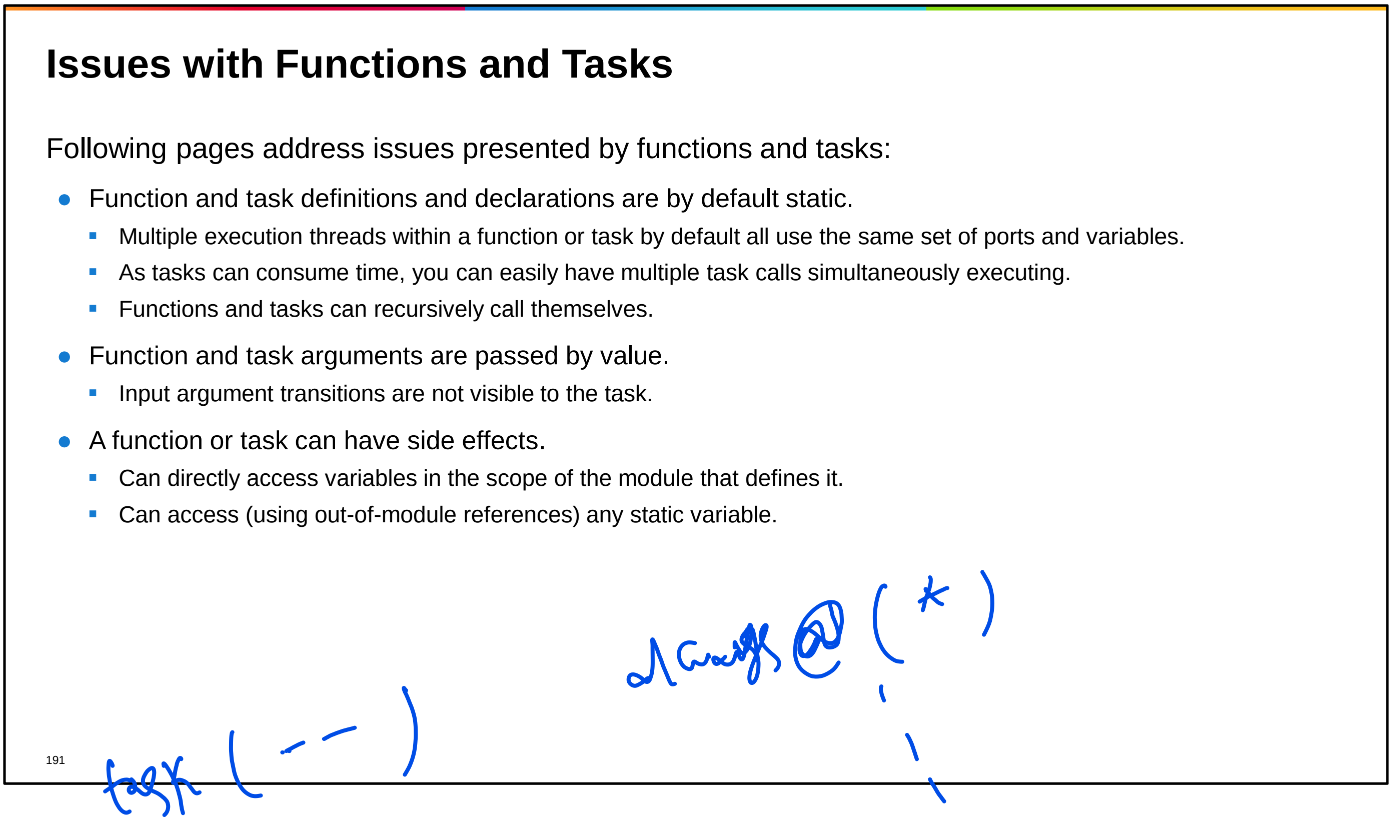
190

The disable statement terminates activity associated with the task or named block. You can use it to work around the lack of C-like break and continue statements. You can use it to behaviorally model asynchronous activity such as interrupts.

The simulator discontinues any activity associated with the task or named block. If executing the task or named block, it resumes execution with the statement after the task call or after the named block. Implementations can choose whether to remove scheduled nonblocking assignments and whether to discontinue procedural continuous assignments, so this can be a source of differences between simulators.

Disabling a task or named block has no persistence. Subsequent statement execution can immediately again call the task and execution can immediately re-enter the named block, for example, if it is located in an always construct or loop construct.

This example repeatedly calls the cpu_driver task to apply stimulus. Upon arrival of an interrupt, this example terminates the current execution of the cpu driver to service the interrupt.



Issues with Functions and Tasks

Following pages address issues presented by functions and tasks:

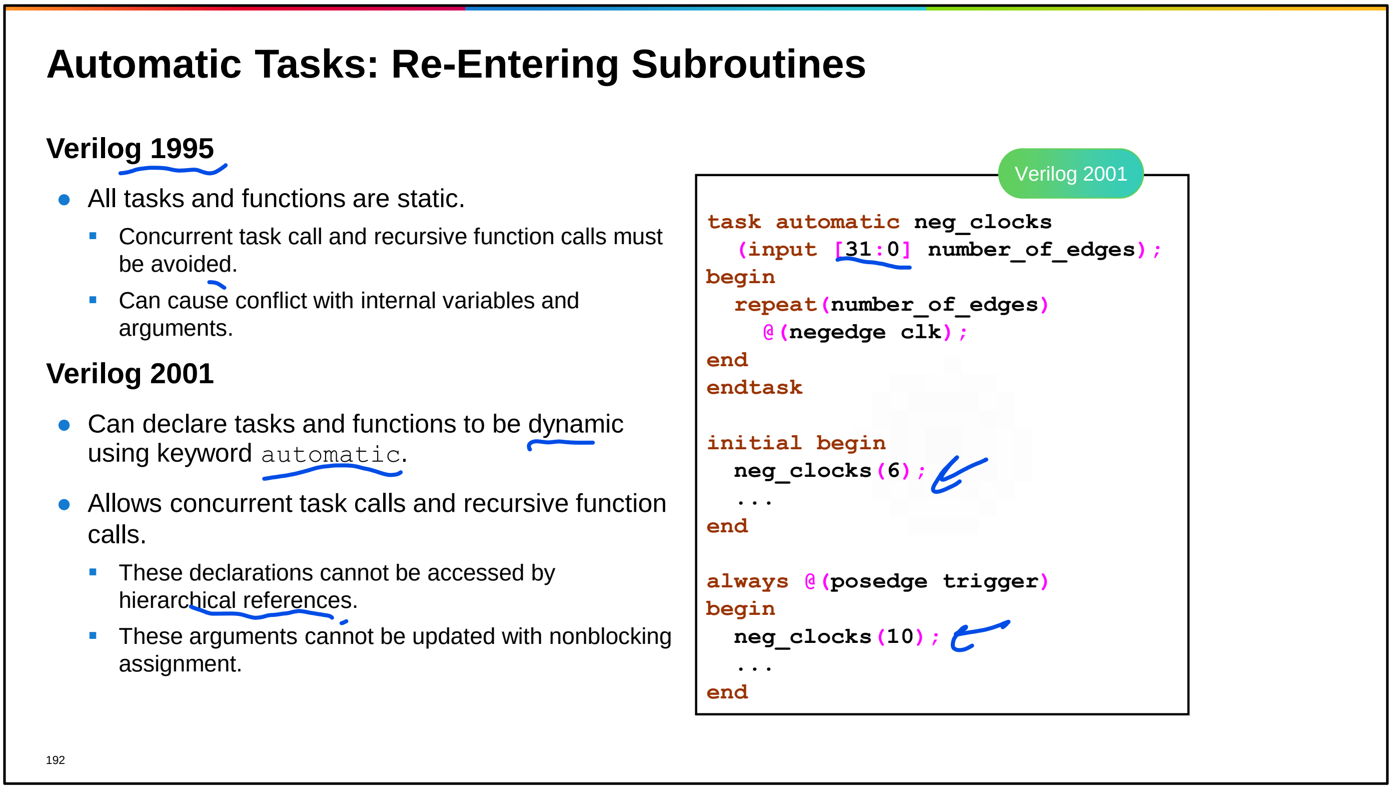
- Function and task definitions and declarations are by default static.
  - Multiple execution threads within a function or task by default all use the same set of ports and variables.
  - As tasks can consume time, you can easily have multiple task calls simultaneously executing.
  - Functions and tasks can recursively call themselves.
- Function and task arguments are passed by value.
  - Input argument transitions are not visible to the task.
- A function or task can have side effects.
  - Can directly access variables in the scope of the module that defines it.
  - Can access (using out-of-module references) any static variable.

always @ (*)

task (- -)

The following pages explore three characteristics of functions and tasks.

- The first issue is that by default there is only one copy of a subroutine’s variables, so that multiple outstanding calls to the same subroutine clobber each other.
- The second issue is that arguments are passed and returned by value. The subroutines maintain local variables that get the input values upon invocation. A task cannot observe future transitions on the net or variable arguments from which the passed value was derived.
- The third issue is that subroutines can have side-effects, they can directly access the defining module’s nets and variables, and can through out-of-module references access any nets and variables in the simulation. This is not an issue as much as it is a benefit, as it can greatly ease your testbench development effort.



Automatic Tasks: Re-Entering Subroutines

Verilog 1995

- All tasks and functions are static.
  - Concurrent task call and recursive function calls must be avoided.
  - Can cause conflict with internal variables and arguments.

Verilog 2001

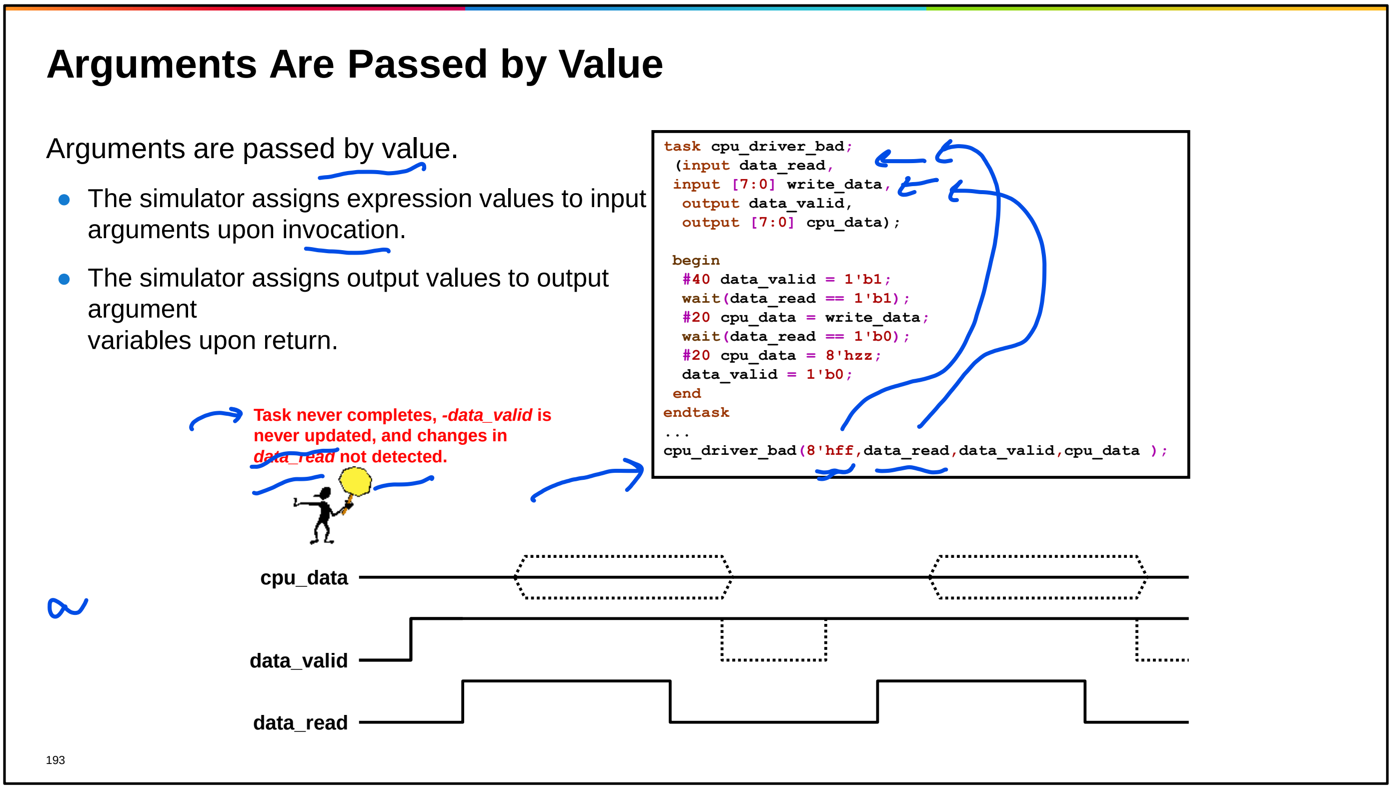
- Can declare tasks and functions to be dynamic using keyword `automatic`.
- Allows concurrent task calls and recursive function calls.
  - These declarations cannot be accessed by hierarchical references.
  - These arguments cannot be updated with nonblocking assignment.

Verilog 2001

```
task automatic neg_clocks
  (input [31:0] number_of_edges);
begin
  repeat(number_of_edges)
    @(negedge clk);
end
endtask

initial begin
  neg_clocks(6);
  ...
end

always @(posedge trigger)
begin
  neg_clocks(10);
  ...
end
```

For Verilog 1995, all subroutines are static. Only one copy of a subroutine arguments and variables exists. Multiple concurrent task calls and recursive function calls all use the same copy of inputs, local variables, and outputs.

For Verilog 2001, you can declare an automatic subroutine. A separate copy of the subroutine arguments and variables exists for each invocation. Multiple concurrent task calls and recursive function calls all use their own copy of inputs, local variables, and outputs. As the automatic arguments and variables exist only for the duration of the subroutine execution, you cannot reference them hierarchically from outside the subroutine, cannot assign to them with nonblocking assignments, and cannot use $monitor or $strobe with them.



Arguments Are Passed by Value

Arguments are passed by value.

- The simulator assigns expression values to input arguments upon invocation.
- The simulator assigns output values to output argument
variables upon return.

```
task cpu_driver_bad;
 (input data_read,
 input [7:0] write_data,
  output data_valid,
  output [7:0] cpu_data);

 begin
  #40 data_valid = 1'b1;
  wait(data_read == 1'b1);
  #20 cpu_data = write_data;
  wait(data_read == 1'b0);
  #20 cpu_data = 8'hzz;
  data_valid = 1'b0;
 end
endtask
...
cpu_driver_bad(8'hff,data_read,data_valid,cpu_data );
```

Task never completes, *-data_valid* is never updated, and changes in *data_read* not detected.

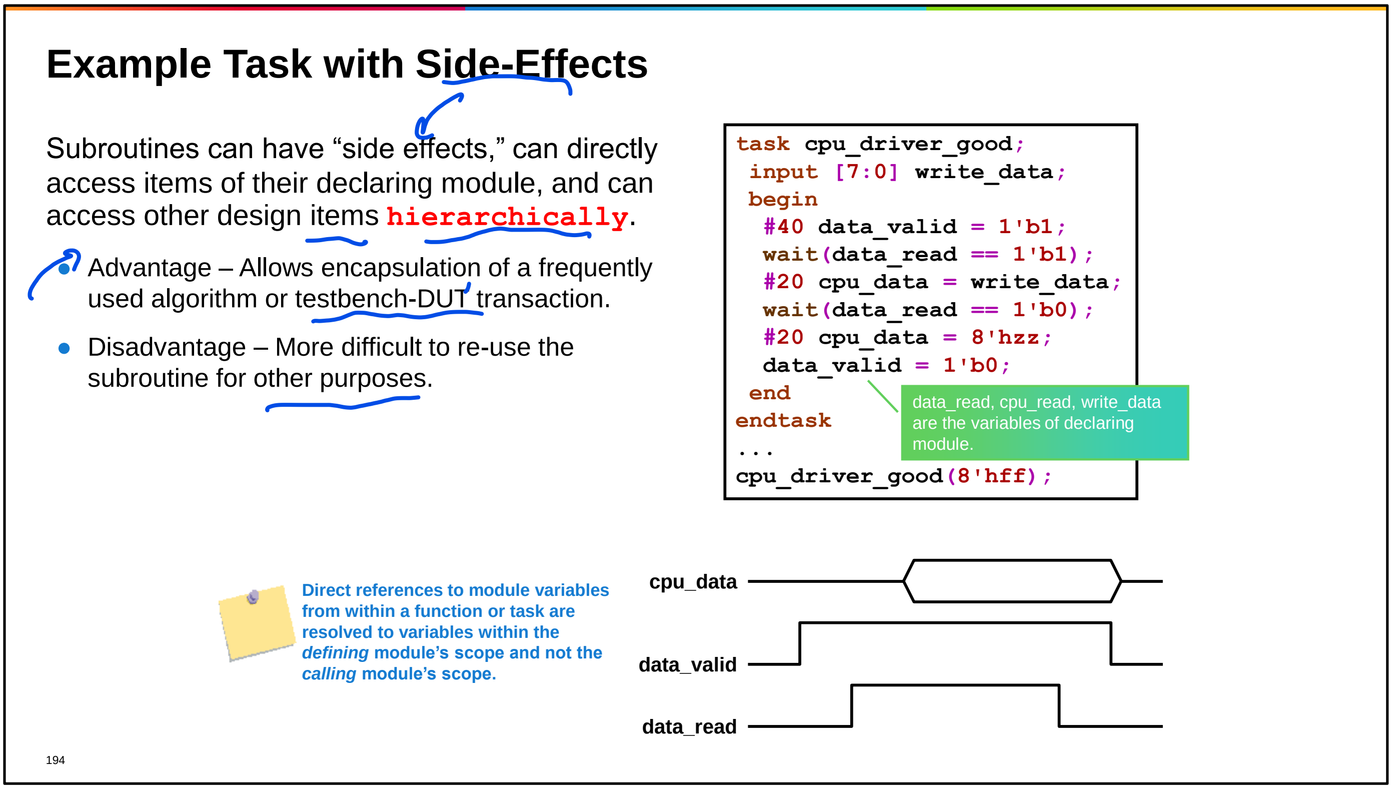
193

Subroutine arguments are passed by value.

- The simulator assigns actual argument values to formal input arguments upon invocation.
- The simulator assigns output values to output argument variables upon return.

The problem with the cpu_driver_bad task is that the input argument data_read is sampled when the task is called and used throughout the task execution. External changes in data_read are not seen within the task during the lifetime of the task execution. Therefore the task hangs at the first wait statement.

Also the assignment to the output argument data_valid is not made – this will only be updated at the end of the task execution, but because the task never completes, data_read is never updated.



Example Task with Side-Effects

Subroutines can have “side effects,” can directly access items of their declaring module, and can access other design items **hierarchically**.

- Advantage – Allows encapsulation of a frequently used algorithm or testbench-DUT transaction.
- Disadvantage – More difficult to re-use the subroutine for other purposes.

```
task cpu_driver_good;
 input [7:0] write_data;
 begin
  #40 data_valid = 1'b1;
  wait(data_read == 1'b1);
  #20 cpu_data = write_data;
  wait(data_read == 1'b0);
  #20 cpu_data = 8'hzz;
  data_valid = 1'b0;
 end
endtask
...
cpu_driver_good(8'hff);
```

data_read, cpu_read, write_data are the variables of declaring module.

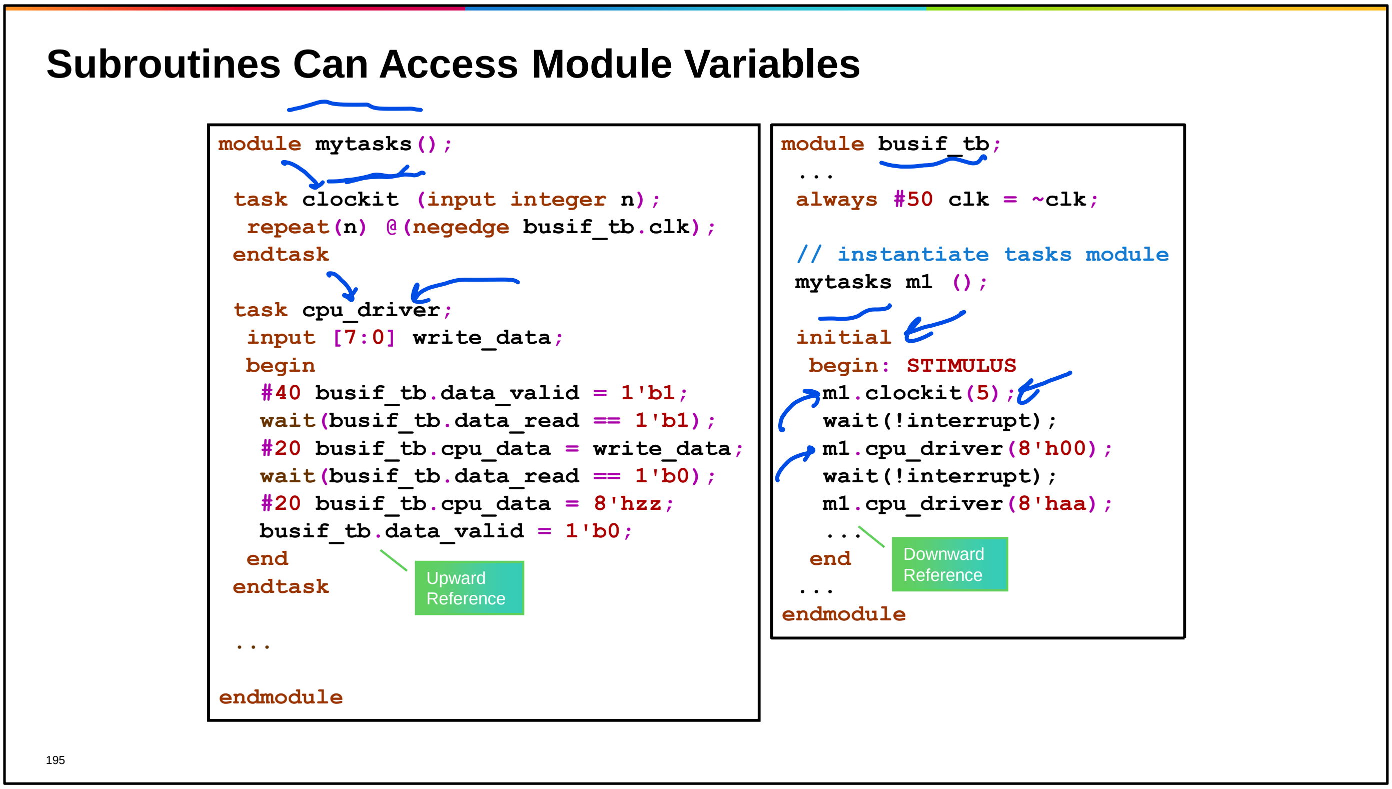
Direct references to module variables from within a function or task are resolved to variables within the ***defining*** **module’s scope and not the** ***calling*** **module’s scope.**

Subroutines can bypass their ports to read and write directly the variables of the defining module, and other design items by hierarchical reference.

This feature greatly simplifies your testbench.

This modified task directly references the data_valid and data_read module variables instead of having their values passed as argument. The wait statements inside the task can now detect transitions of the data_read variable and the task will operate correctly.

Keep in mind that direct subroutine references to module variables are to those within the defining module’s scope and not the calling module’s scope.



Subroutines Can Access Module Variables

```
module mytasks();

 task clockit (input integer n);
  repeat(n) @(negedge busif_tb.clk);
 endtask

 task cpu_driver;
  input [7:0] write_data;
  begin
   #40 busif_tb.data_valid = 1'b1;
   wait(busif_tb.data_read == 1'b1);
   #20 busif_tb.cpu_data = write_data;
   wait(busif_tb.data_read == 1'b0);
   #20 busif_tb.cpu_data = 8'hzz;
   busif_tb.data_valid = 1'b0;
  end
 endtask

 ...

endmodule
```

Upward Reference

```
module busif_tb;
 ...
 always #50 clk = ~clk;

 // instantiate tasks module
 mytasks m1 ();

 initial
  begin: STIMULUS
   m1.clockit(5);
   wait(!interrupt);
   m1.cpu_driver(8'h00);
   wait(!interrupt);
   m1.cpu_driver(8'haa);
   ...
  end
 ...
endmodule
```

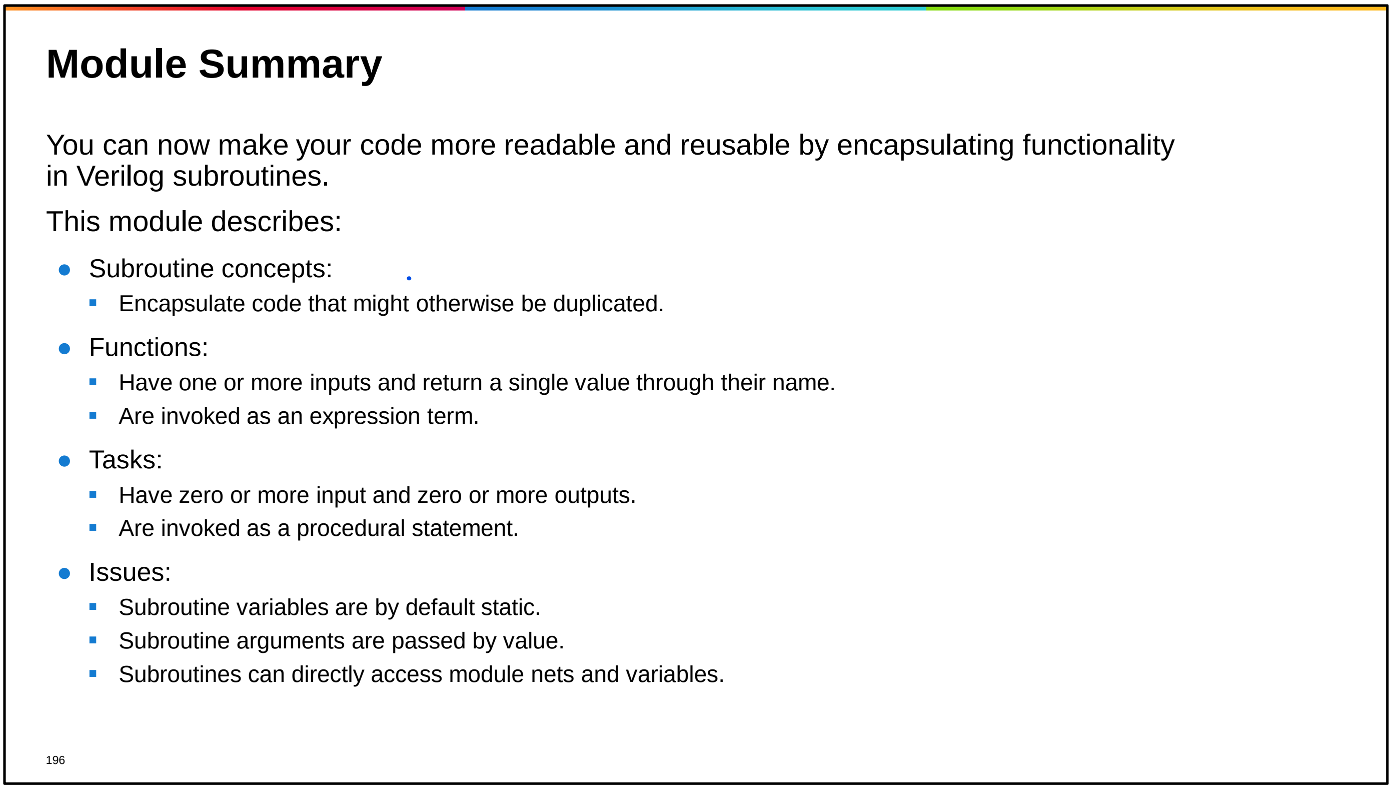
Downward Reference

195

At some point, you can find it convenient to declare a subroutine in one module and call it from another module.

Every Verilog object has a unique path name that starts at a top-level instance, and using the dot (.) character hierarchy separator, traverses down through instance names to the name of the object. You can use these hierarchical names for any Verilog 1995 object. You are not permitted to use the hierarchical name of a Verilog 2001 automatic subroutine variable.

- Hierarchical references can be absolute, starting at a top-level instance and traversing a downward path.
- Hierarchical references can be relative, starting at the scope of the reference and traversing a downward path.
- Hierarchical references can also be upward, but such references can have only a single instance or module name, which resolves to the nearest upward instance or module with that name.



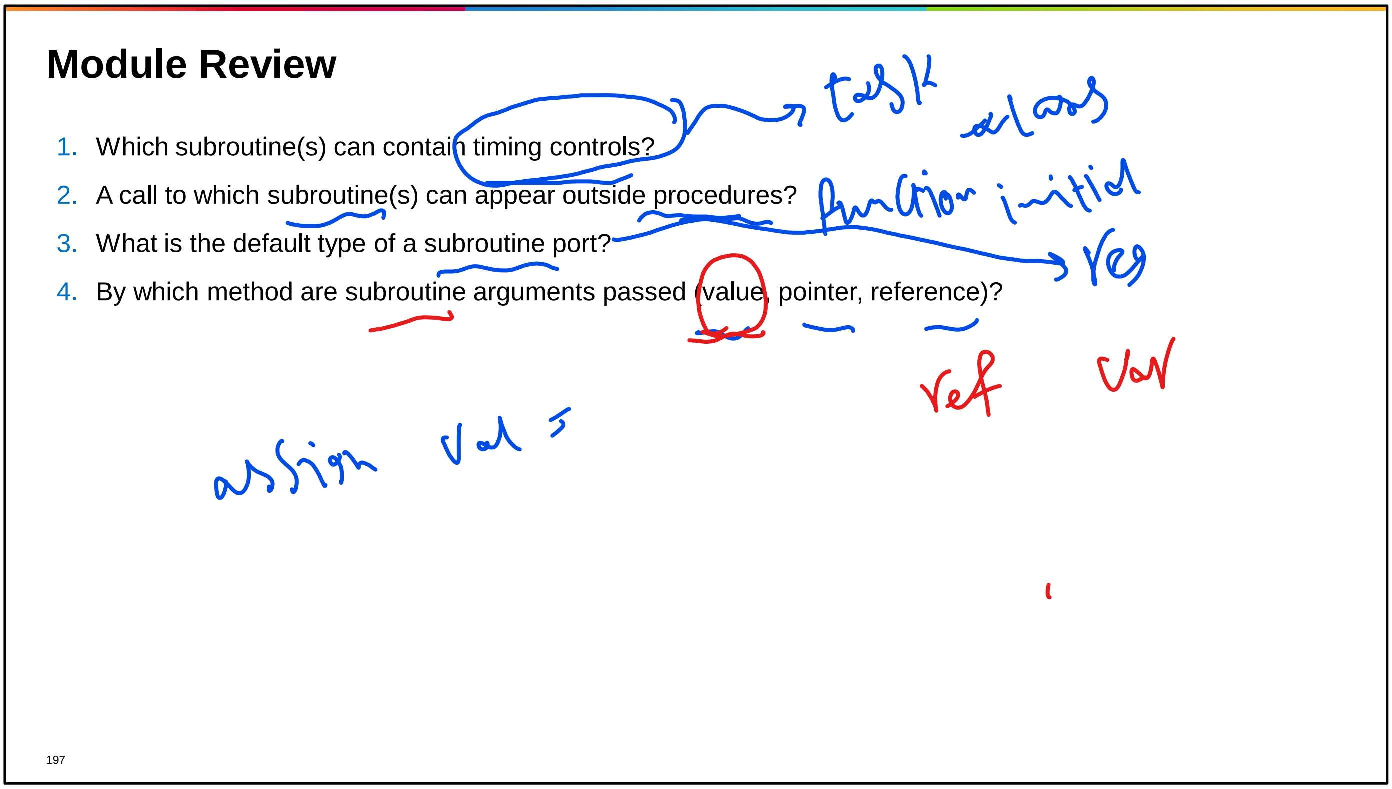
Module Summary

You can now make your code more readable and reusable by encapsulating functionality in Verilog subroutines.

This module describes:

- Subroutine concepts:
  - Encapsulate code that might otherwise be duplicated.
- Functions:
  - Have one or more inputs and return a single value through their name.
  - Are invoked as an expression term.
- Tasks:
  - Have zero or more input and zero or more outputs.
  - Are invoked as a procedural statement.
- Issues:
  - Subroutine variables are by default static.
  - Subroutine arguments are passed by value.
  - Subroutines can directly access module nets and variables.

After completing this module, you can effectively use subroutines to encapsulate functionality and make your code more readable and reusable. This module described Verilog subroutines known as functions and tasks. It explained where and how to define them, how to invoke them, and some issues involved with using them.



Module Review

1. Which subroutine(s) can contain timing controls?
2. A call to which subroutine(s) can appear outside procedures?
3. What is the default type of a subroutine port?
4. By which method are subroutine arguments passed ([✓ value], pointer, reference)?

task

class

function

initial

reg

ref

var

assign val =

197

This page does not contain notes.

Module Review Solutions

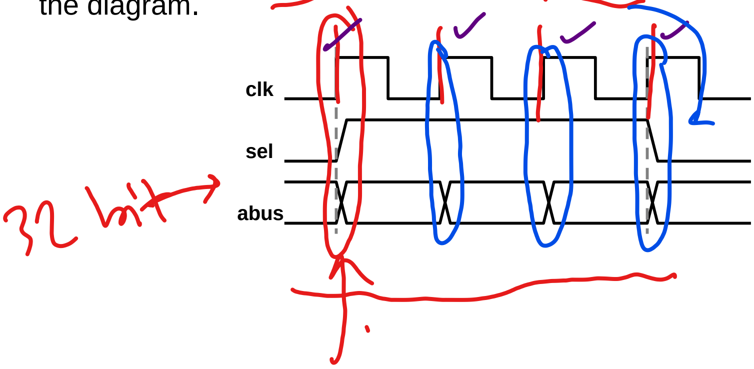
TB Design

1. Which subroutine(s) can contain timing controls?
   - Only a task can contain timing controls. A function cannot invoke the scheduler in any way.
2. A call to which subroutine(s) can appear outside procedures?
   - You use a function call as a term of an expression, so a function call can appear on the right side of a continuous assignment.
3. What is the default type of a subroutine port?
   - A subroutine port is by default a single-bit reg. You can declare it as any variable type. Subroutines cannot declare nets.
4. By which method are subroutine arguments passed (value, pointer, reference)?
   - Subroutine arguments are passed by value.

This page does not contain notes.

Module Exercise

Write a task to accept a 32-bit input argument address and place that address on the “abus” and increment the address through a total of four cycles. The task also controls the “sel” signal. Refer to the diagram.



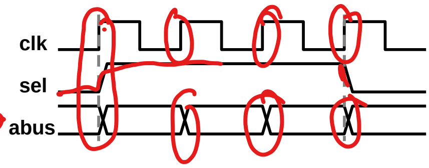
```
Task t5 ;
  input [31;0] address;
  begin
   @(posedge clk)
      sel <= 1;
      abus <= address;
   repeat(3)
     @(posedge clk)
      abus <= abus #1 ;
      sel <= 0;
  end
endtask
```

This page does not contain notes.

Module Exercise Solution

Write a task to accept a 32-bit input argument address and place that address on the “abus” and increment the address through a total of four cycles. The task also controls the “sel” signal. Refer to the diagram.

add



task4 (address)

Solution:

```
task task4;
 input [31:0] addr;
 begin
  @(posedge clk)
  sel <= 1;
  abus <= addr;
  repeat(3)
   @(posedge clk)
    abus <= abus + 1;
  sel <= 0;
 end
endtask
```

This page does not contain notes.

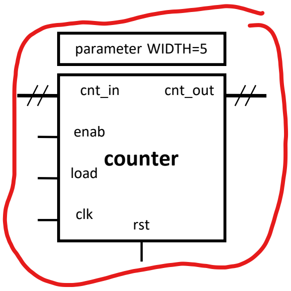
Labs

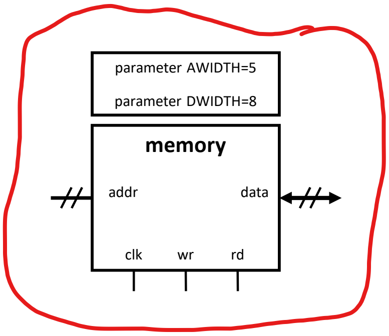
Lab 10-1 Modeling the Counter Using Functions

- Encapsulate counter design combinational behaviors in a function.

Lab 10-2 Modeling the Memory Test Block Using Tasks

- Encapsulate memory test procedural behaviors in tasks.





Your objective is to encapsulate functionality within Verilog subroutines.

For this lab, you:

- Encapsulate counter design combinational behaviors in a function; and
- Encapsulate memory test procedural behaviors in tasks.