Chapter 3
Procedural Statements and Routines

As you verify your design, you need to write a great deal of code, most of which is in tasks and functions. SystemVerilog introduces many incremental improvements to make this easier by making the language look more like C, especially around argument passing. If you have a background in software engineering, these additions should be very familiar.

3.1 Procedural Statements

SystemVerilog adopts many operators and statements from C and C++. You can declare a loop variable inside a `for` loop that then restricts the scope of the loop variable and can prevent some coding bugs. The new auto-increment `++` and auto-decrement `--` operators are available in both pre- and post-forms. The compound assignments, `+=`, `-=`, `^=`, and many more make your code tighter. If you have a label on a `begin` or `fork` statement, you can put the same label on the matching `end` or `join` statement. This makes it easier to match the start and finish of a block. You can also put a label on other SystemVerilog end statements such as `endmodule`, `endtask`, `endfunction`, and others that you will learn in this book. Sample 3.1 demonstrates some of the new constructs.

Sample 3.1 New procedural statements and operators

```
initial
  begin : example
  integer array[10], sum, j;

  // Declare i in for statement
  for (int i=0; i<10; i++)       // Increment i
    array[i] = i;

  // Add up values in the array
  sum = array[9];
  j=8;
  do                            // do...while loop
    sum += array[j];            // Compound assignment
  while (j--);                  // Test if j=0
  $display("Sum=%4d", sum);     // %4d - specify width
end : example                   // End label
```

Two new statements help with loops. First, if you are in a loop, but want to skip over the rest of the statements and do the next iteration, use `continue`. If you want to leave the loop immediately, use `break`.

The compound assignment in Sample 3.1 is equivalent to `sum = sum + array[j];` The loop in Sample 3.2 reads commands from a file using the file I/O system tasks that are part of Verilog-2001. If the command is just a blank line, the code does a `continue`, skipping any further processing of the command. If the command is "`done`," the code does a `break` to terminate the loop.

Sample 3.2 Using `break` and `continue` while reading a file

```
initial begin
  bit [127:0] cmd;
  int file, c;

  file = $fopen("commands.txt", "r");
  while (!$feof(file)) begin
    c = $fscanf(file, "%s", cmd);
    case (cmd)
      "":     continue;     // Blank line - skip to loop end
      "done": break;        // Done - leave loop
      ...                   // Process other commands here
    endcase // case(cmd)
    end
  $fclose(file);
end
```

SystemVerilog expands the `case` statement so that you no longer have to give every possible value, but can instead give a range values as shown in Sample 3.3. This is a version of the `inside` operator shown more in more detail in Section 6.4.5.

Sample 3.3 Case-inside statement with ranges

```
case (graduation_year) inside // <<< Note "inside" keyword
  [1950:1959]: $display("Do you like bobby sox?");
  [1960:1969]: $display("Did you go to Woodstock?");
  [1970:1979]: $display("Did you dance to disco?");
endcase
```

3.2 Tasks, Functions, and Void Functions

Verilog makes a very clear differentiation between tasks and functions. The most important difference is that a task can consume time whereas a function cannot. A function cannot have a delay, `#100`, a blocking statement such as `@(posedge clock)` or `wait (ready)`, or call a task. Additionally, a Verilog function must return a value and the value must be used, as in an assignment statement.

SystemVerilog relaxes this rule a little in that a function can call a task, but only in a thread spawned with the `fork… join_none` statement, which is described in Section 7.1.

If you have a SystemVerilog task that does not consume time, you should make it a `void function`, which is a function that does not return a value. Now it can be called from any task or function. For maximum flexibility, any debug routine should be a void function rather than a task so that it can be called from any task or function. Sample 3.4 prints values from a state machine.

Sample 3.4 Void function for debug

```
function void print_state();
  $display("@%0t: state = %s", $time, cur_state.name());
endfunction
```

In SystemVerilog, if you want to call a function and ignore its return value, cast the result to `void`, as shown in Sample 3.5. Some simulators, such as VCS, allow you to ignore the return value without using the `void` syntax. The LRM says this should be a warning.

Sample 3.5 Ignoring a function's return value

```
void'($fscanf(file, "%d", i));
```

3.3 Task and Function Overview

SystemVerilog makes several small improvements to tasks and functions to make them look more like C or C++ routines. In general, a routine definition or call with no arguments does not need the empty parentheses (). This book includes them for added clarity.

3.3.1 Routine `Begin...End` Removed

The first improvement you may notice in SystemVerilog routines is that `begin...end` blocks are optional, while Verilog-1995 required them on all but single-line routines. The `task` / `endtask` and `function` / `endfunction` keywords are enough to define the routine boundaries, as shown in Sample 3.6.

Sample 3.6 Simple task without `begin...end`

```
task multiple_lines();
  $display("First line");
  $display("Second line");
endtask : multiple_lines
```

3.4 Routine Arguments

Many of the SystemVerilog improvements for routines make it easier to declare arguments and expand the ways you can pass values to and from a routine.

3.4.1 C-style Routine Arguments

SystemVerilog and Verilog-2001 allow you to declare task and function arguments more cleanly and with less repetition. The following Verilog task requires you to declare some arguments twice: once for the direction, and once for the type, as shown in Sample 3.7.

Sample 3.7 Verilog-1995 routine arguments

```
task mytask1;
  output [31:0] x;
  reg    [31:0] x;
  input         y;
  ...
endtask
```

With SystemVerilog, you can use the less verbose C-style, shown in Sample 3.8. Note that you should use the universal input type of `logic`.

Sample 3.8 C-style routine arguments

```
task mytask2 (output logic [31:0] x,
              input  logic y);
...
endtask
```

3.4.2 *Argument Direction*

You can take even more shortcuts with declaring routine arguments. The direction and type default to "input logic" and are sticky, so you don't have to repeat these for similar arguments. Sample 3.9 shows a routine header written using the Verilog-1995 style and SystemVerilog data types.

Sample 3.9 Verbose Verilog-style routine arguments

```
task t3;
  input a, b;
  logic a, b;
  output [15:0] u, v;
  bit [15:0] u, v;
  ...
endtask
```

You could rewrite this as shown in Sample 3.10.

Sample 3.10 Routine arguments with sticky types

```
task t3(a, b, output bit [15:0] u, v);  // Lazy declarations
  ...
endtask
```

The arguments `a` and `b` are input logic, 1-bit wide. The arguments `u` and `v` are 16-bit output bit types. Now that you know this, don't depend on the defaults, as your code will be infested with subtle and hard to find bugs, as explained in Section 3.4.6. Always declare the type and direction for every routine argument.

3.4.3 *Advanced Argument Types*

Verilog had a simple way to handle arguments: an `input` or `inout` was copied to a local variable at the start of the routine, whereas an `output` or `inout` was copied when the routine exited. No memories could be passed into a Verilog routine, only scalars.

In SystemVerilog, you can specify that an argument is passed by reference, rather than copying its value. This argument type, `ref`, has several benefits over `input`, `output`, and `inout`. First, you can now pass an array into a routine, here one that prints the checksum.

Sample 3.11 Passing arrays using `ref` and `const`

```
function automatic void print_csm11 (const ref bit [31:0] a[]);
  bit [31:0] checksum = 0;
  for (int i=0; i<a.size(); i++)
    checksum ^= a[i];
  $display("The array checksum is %h", checksum);
endfunction
```

The = compound assignment in Sample 3.11 is a shorthand way of writing the statement: `checksum = checksum ^ a[i];`

SystemVerilog allows you to pass array arguments without the `ref` direction, but the array is copied onto the stack, an expensive operation for all but the smallest arrays.

The SystemVerilog LRM states that `ref` arguments can only be used in routines with automatic storage. If you specify the `automatic` attribute for programs and module, all the routines inside are automatic. See Section 3.6 for more details on storage.

Sample 3.11 also shows the `const` modifier. As a result, the array `a` points to the array in the routine call, but the contents of the array cannot be modified. If you try to change the contents, the compiler prints an error.

Always use `ref` when passing arrays to a routine for best performance. If you don't want the routine to change the array values, use the `const ref` type, which causes the compiler to check that your routine does not modify the array.

The second benefit of `ref` arguments is that a task can modify a variable and is instantly seen by the calling function. This is useful when you have several threads executing concurrently and want a simple way to pass information. See Chapter 7 for more details on using `fork-join`.

In Sample 3.12, the `thread2` block in the initial block can access the data from memory as soon as `enable` is asserted, even though the `bus_read` task does not return until the bus transaction completes, which could be several cycles later.

Sample 3.12 Using `ref` across threads

```
task automatic bus_read(input logic [31:0] addr,
                        ref   logic [31:0] data);

  // Request bus and drive address
  bus_request <= 1'b1;
  @(posedge bus_grant) bus_addr <= addr;

  // Wait for data from memory
  @(posedge bus_enable) data <= bus_data;

  // Release bus and wait for grant
  bus_request <= 1'b0;
  @(negedge bus_grant);
endtask

logic [31:0] addr, data;

initial
  fork
    bus_read(addr, data);
    begin : thread2
      @data;  // Trigger on data change
      $display("Read %h from bus", data);
    end
  join
```

The `data` argument is passed as `ref`, and as a result, the `@data` statement triggers as soon as `data` changes in the task. If you had declared `data` as `output`, the `@data` statement would not trigger until the end of the bus transaction.

3.4.4 Default Value for an Argument

As your testbench grows in sophistication, you may want to add additional controls to your code but not break existing code. For the function in Sample 3.11, you might want to print a checksum of just the middle values of the array. However, you don't want to go back and rewrite every call to add extra arguments. In SystemVerilog you can specify a default value that is used if you leave out an argument in the call. Sample 3.13 adds `low` and `high` arguments to the `print_csm` function so you can print a checksum of a range of values.

Sample 3.13 Function with default argument values

```
function automatic void print_csm (const ref bit [31:0] a[],
                                    input bit [31:0] low = 0,
                                    input int high = -1);
  bit [31:0] checksum = 0;

  if (high == -1 || high >= a.size())
    high = a.size()-1;

  for (int i=low; i<=high; i++)
    checksum ^= a[i];
  $display("The array checksum is %h", checksum);
endfunction
```

You can call this function in the following ways, as shown in Sample 3.14. Note that the first call is compatible with both versions of the `print_csm` routine.

Sample 3.14 Using default argument values

```
print_csm(a);       // Checksum a[0:size()-1] - default
print_csm(a, 2, 4); // Checksum a[2:4]
print_csm(a, 1);    // Start at 1
print_csm(a,, 2);   // Checksum a[0:2]
print_csm();        // Compile error: a has no default
```

Using a default value of −1 (or any out-of-range value) is a good way to see if the call specified a value.

A Verilog `for` loop always executes the initialization (`int i=low`), and test (`i<=high`) before starting the loop. Thus, if you accidently passed a `low` value that was larger than `high` or the array size, the `for` loop would never execute the body.

3.4.5 Passing Arguments by Name

You may have noticed in the SystemVerilog LRM that the arguments to a task or function are sometimes called “ports,” just like the connections for a module. If you have a task or function with many arguments, some with default values, and you only want to set a few of those arguments, you can specify a subset by specifying the name of the routine argument with a port-like syntax, as shown in Sample 3.15.

Sample 3.15 Binding arguments by name

```
task many (input int a=1, b=2, c=3, d=4);
  $display("%0d %0d %0d %0d", a, b, c, d);
endtask

initial begin          // a  b  c  d
  many(6, 7, 8, 9);    // 6  7  8  9  Specify all values
  many();              // 1  2  3  4  Use defaults
  many(.c(5));         // 1  2  5  4  Only specify c
  many(, 6, .d(8));    // 1  6  3  8  Mix styles
end
```

3.4.6 Common Coding Errors

The most common coding mistake that you are likely to make with a routine is forgetting that the argument type is sticky with respect to the previous argument, and that the default type for the first argument is a single-bit input. Start with the simple task header in Sample 3.16.

Sample 3.16 Original task header

```
task sticky(int a, b);
```

The two arguments are input integers. As you are writing the task, you realize that you need access to an array, so you add a new array argument, and use the `ref` type so it does not have to be copied. Your routine header now looks like Sample 3.17.

Sample 3.17 Task header with additional array argument

```
task automatic sticky(ref int array[50],
                      int a, b);  // What direction are these?
```

What argument types are `a` and `b`? They take the direction of the previous argument that is a `ref`. Using `ref` for a simple variable such as an `int` is not usually needed, but you would not get even a warning from the compiler, and thus would not realize that you were using the wrong direction.

If any argument to your routine is something other than the default input type, specify the direction for all arguments as shown in Sample 3.18.

Sample 3.18 Task header with additional array argument

```
task automatic sticky(ref   int array[50],
                      input int a, b);  // Be explicit
```

3.5 Returning from a Routine

Verilog had a primitive way to end a routine; after you executed the last statement in a routine, it returned to the calling code. In addition, a function returned a value by assigning that value to a variable with the same name as the function.

3.5.1 The Return Statement

SystemVerilog adds the `return` statement to make it easier for you to control the flow in your routines. The task in Sample 3.19 needs to return early because of error checking. Otherwise, it would have to put the rest of the task in an `else` clause, which would cause more indentation and be more difficult to read.

Sample 3.19 Return in a task

```
task automatic load_array(input int len, ref int array[]);
  if (len <= 0) begin
    $display("Bad len");
    return;
  end

  // Code for the rest of the task
  ...
endtask
```

The `return` statement in Sample 3.20 can simplify your functions.

Sample 3.20 Return in a function

```
function bit transmit(input bit [31:0] data);
  // Send transaction
  ...
  return status; // Return status: 0=error
endfunction
```

3.5.2 Returning an Array from a Function

Verilog routines could only return a simple value such as a bit, integer, or vector. If you wanted to compute and return an array, there was no simple way. In System Verilog, a function can return an array, using several techniques.

The first way is to define a type for the array, and then use that in the function declaration. Sample 3.21 uses the array type from Sample 2.40, and creates an function to initialize the array.

Sample 3.21 Returning an array from a function with a typedef

```
typedef int fixed_array5_t[5];
fixed_array5_t f5;

function fixed_array5_t init(input int start);
  foreach (init[i])
    init[i] = i + start;
endfunction

initial begin
  f5 = init(5);
  foreach (f5[i])
    $display("f5[%0d] = %0d", i, f5[i]);
end
```

One problem with the preceding code is that the function `init` creates an array, which is copied into the array `f5`. If the array was large, this could be a large performance problem.

The alternative is to pass the routine by reference. The easiest way is to pass the array into the function as a `ref` argument, as shown in Sample 3.22.

Sample 3.22 Passing an array to a function as a ref argument

```
function automatic void init(ref int f[5], input int start);
  foreach (f[i])
    f[i] = i + start;
endfunction

int fa[5];
initial begin
  init(fa, 5);
  foreach (fa[i])
    $display("fa[%0d] = %0d", i, fa[i]);
end
```

The last way for a function to return an array is to wrap the array inside of a class, and return a handle to an object. Chapter 5 describes classes, objects, and handles.

3.6 Local Data Storage

When Verilog was created in the 1980s, it was tightly tied to describing hardware. As a result, all objects in the language were statically allocated. In particular, routine arguments and local variables were stored in a fixed location, rather than pushing them on a stack like other programming languages. Why try to model dynamic code such as a recursive routine when there is no way to build this in silicon? However, software engineers verifying the designs, who were used to the behavior of stack-based languages such as C, were bitten by these subtle bugs, and were thus limited in their ability to create complex testbenches with libraries of routines.

3.6.1 Automatic Storage

In Verilog-1995, if you tried to call a task from multiple places in your testbench, the local variables shared common, static storage, and so the different threads stepped on each other's values. In Verilog-2001 you can specify that tasks, functions, and modules use automatic storage, which causes the simulator to use the stack for local variables.

In SystemVerilog, routines still use static storage by default, for both modules and program blocks. You should always make program blocks (and their routines) use automatic storage by putting the `automatic` keyword in the program statement. In Chapter 4 you will learn about `program` blocks that hold the testbench code. Section 7.1.6 shows how automatic storage helps when you are creating multiple threads.

Sample 3.23 shows a task to monitor when data are written into memory.

Sample 3.23 Specifying automatic storage in program blocks

```
program automatic test();
  task wait_for_bus(input logic [31:0] addr, expect_data,
                    output logic success);
    while (bus_addr !== addr)
      @(bus_addr);
    success = (bus_data == expect_data);
  endtask
...
endprogram
```

You can call this task multiple times concurrently, as the `addr` and `expect_data` arguments are stored separately for each call. Without the `automatic` modifier, if you called `wait_f or_bus` a second time while the first was still waiting, the second call would overwrite the two arguments.

3.6.2 Variable Initialization

A similar problem occurs when you try to initialize a local variable in a declaration, as it is actually initialized before the start of simulation. The general solution is to avoid initializing a variable in a declaration to anything other than a constant. Use a separate assignment statement to give you better control over when initialization is done.

The task in Sample 3.24 looks at the bus after five cycles and then creates a local variable and attempts to initialize it to the current value of the address bus.

Sample 3.24 Static initialization bug

```
program initialization; // Buggy version

  task check_bus();
    repeat (5) @(posedge clock);
    if (bus_cmd === READ) begin
      // When is local_addr initialized?
      logic [7:0] local_addr = addr<<2;  // Bug
      $display("Local Addr = %h", local_addr);
    end
  endtask

endprogram
```

The bug is that the variable `local_addr` is statically allocated, so it is actually initialized at the start of simulation, not when the `begin`...`end` block is entered. Once again, the solution is to declare the program as `automatic` as shown in Sample 3.25.

Sample 3.25 Static initialization fix: use `automatic`

```
program automatic initialization; // Bug solved
...
endprogram
```

Additionally, you can avoid this by never initializing a variable in the declaration, but this is harder to remember, especially for C programmers. Sample 3.26 show the recommended style of separating the declaration and initialization.

Sample 3.26 Static initialization fix: break apart declaration and initialization

```
logic [7:0] local_addr;
local_addr = addr << 2;  // Bug solved
```

3.7 Time Values

SystemVerilog has several new constructs to allow you to unambiguously specify time values in your system.

3.7.1 Time Units and Precision

When you rely on the `` `timescale`` compiler directive, you must compile the files in the proper order to be sure all the delays use the proper scale and precision. One way avoiding this compilation ordering problem is to require that every file that starts with a `` `timescale `` compiler directive should end with one that resets it back to a company-specific default such as 1ns/1ns.

The `timeunit` and `timeprecision` declarations eliminate this ambiguity by precisely specifying the values for every module. Sample 3.27 shows these declarations. Note that if you use these instead of `` `timescale ``, you must put them in every module that has a delay. See the LRM for more on these declarations.

3.7.2 *Time Literals*

SystemVerilog allows you to unambiguously specify a time value plus units. Your code can use delays such as `0.1ns` or `20ps`. Just remember to use `timeunit` and `timeprecision` or `` `timescale ``. You can make your code even more time aware by using the classic Verilog `$timeformat()`, `$time`, and `$realtime` system tasks. The four arguments to $timeformat are the scaling factor (−9 for nanoseconds, −12 for picoseconds), the number of digits to the right of the decimal point, a string to print after the time value, and the minimum field width.

Sample 3.27 shows various delays and the result from printing the time when it is formatted by `$timeformat()` and the `%t` specifier.

Sample 3.27 Time literals and `$timeformat`

```
module timing;
  timeunit 1ns;
  timeprecision 1ps;
  initial begin
    $timeformat(-9, 3, "ns", 8);
    #1     $display("%t", $realtime); // 1.000ns
    #2ns   $display("%t", $realtime); // 3.000ns
    #0.1ns $display("%t", $realtime); // 3.100ns
    #41ps  $display("%t", $realtime); // 3.141ns
  end
endmodule
```

3.7.3 *Time and Variables*

You can store time values in variables and use them in calculations and delays. The values are scaled and rounded according to the current time scale and precision. Variables of type `time` cannot hold fractional delays as they are just 64-bit integers, so delays will be rounded. You should use `realtime` variables if this is a problem.

Sample 3.28 shows how `realtime` variables are rounded when used as a delay.

Sample 3.28 Time variables and rounding

```
`timescale 1ns/100ps

module ps;

  initial begin
    realtime rtdelay = 800ps;   // Stored as 0.8 (800ps)
    time     tdelay  = 800ps;   // Rounded to 1

    $timeformat(-12, 0, "ps", 5);
    #rtdelay;                   // Delay of 800ps
    $display("%t", rtdelay);    // "800ps"
    #tdelay;                    // Delay another 1ns
    $display("%t", tdelay);     // "1000ps"
   end

endmodule
`timescale 1ns/1ns              // Reset to default
```

3.7.4 $time vs. $realtime

The system task `$time` returns an integer scaled to the time unit of the current module, but missing any fractional units, while `$realtime` returns a real number with the complete time value, including fractions. This book uses `$time` in the examples for brevity, but your testbenches may need to use `$realtime`.

3.8 Conclusion

The new SystemVerilog procedural constructs and task/function features make it easier for you to create testbenches by making the language look more like other programming languages such as C/C++. As a bonus, SystemVerilog has additional HDL constructs such as timing controls, simple thread control, and 4-state logic.

3.9 Exercises

1. Create the SystemVerilog code with the following requirements:
   a. Create a 512 element integer array
   b. Create a 9-bit address variable to index into the array
   c. Initialize the last location in the array to 5
   d. Call a task, `my_task()`, and pass the array and the address

e. Create `my_task()` that takes two inputs: a constant 512-element integer array passed by reference, and a 9-bit address. The task calls a function, `print_int()`, and passes the array element indexed by the address, pre-decrementing the address.
f. Create `print_int()` that prints out the simulation time and the value of the input. The function has no return value.

2. For the following SystemVerilog code, what is displayed if the task `my_task2()` is automatic?

```
int new_address1, new_address2;
bit clk;
initial begin
  fork
     my_task2(21, new_address1);
     my_task2(20, new_address2);
  join
  $display("new_address1 = %0d", new_address1);
  $display("new_address2 = %0d", new_address2);
end

initial
  forever #50 clk = !clk;

task my_task2(input int address, output int
new_address);
  @(clk);
  new_address = address;
endtask
```

3. For the same SystemVerilog code in Exercise 2, what is displayed if the task `my_task2()` is not automatic?
4. Create the SystemVerilog code to specify that the time should be printed in ps (picoseconds), display 2 digits to the right of the decimal point, and use as few characters as possible

5. Using the formatting system task from Exercise 4, what is displayed by the following code?

```
timeunit 1ns;
timeprecision 1ps;
parameter real t_real = 5.5;
parameter time t_time = 5ns;

initial begin
  #t_time $display("1 %t", $realtime);
  #t_real $display("1 %t", $realtime);
  #t_time $display("1 %t", $realtime);
  #t_real $display("1 %t", $realtime);
end

initial begin
  #t_time $display("2 %t", $time);
  #t_real $display("2 %t", $time);
  #t_time $display("2 %t", $time);
  #t_real $display("2 %t", $time);
end
```