Chapter 2

Data Types

SystemVerilog offers many improved data structures compared with Verilog. Some of these were created for designers but are also useful for testbenches. In this chapter, you will learn about the data structures most useful for verification.

SystemVerilog introduces new data types with the following benefits.

- Two-state: better performance, reduced memory usage
- Queues, dynamic and associative arrays: reduced memory usage, built-in support for searching and sorting
- Classes and structures: support for abstract data structures
- Unions and packed structures: allow multiple views of the same data
- Strings: built-in string support
- Enumerated types: code is easier to write and understand

2.1 Built-In Data Types

Verilog-1995 has two basic data types: variables and nets, both which hold 4-state values: 0, 1, Z, and X. RTL code uses variables to store combinational and sequential values. Variables can be unsigned single or multi-bit (`reg [7:0] m`), signed 32-bit variables (`integer`), unsigned 64-bit variables (`time`), and floating point numbers (`real`). Variables can be grouped together into arrays that have a fixed size. All storage is static, meaning that all variables are alive for the entire simulation and routines cannot use a stack to hold arguments and local values. A net is used to connect parts

of a design such as gate primitives and module instances. Nets come in many flavors, but most designers use scalar and vector wires to connect together the ports of design blocks.

SystemVerilog adds many new data types to help both hardware designers and verification engineers.

2.1.1 The Logic Type

The one thing in Verilog that always leaves new users scratching their heads is the difference between a `reg` and a `wire`. When driving a port, which should you use? How about when you are connecting blocks? SystemVerilog improves the classic `reg` data type so that it can be driven by continuous assignments, gates, and modules, in addition to being a variable. It is given the synonym `logic` so that it does not look like a register declaration. A `logic` signal can be used anywhere a net is used, except that a `logic` variable cannot be driven by multiple structural drivers, such as when you are modeling a bidirectional bus. In this case, the variable needs to be a net-type such as `wire` so that SystemVerilog can resolve the multiple values to determine the final value.

Sample 2.1 shows the SystemVerilog `logic` type.

Sample 2.1 Using the logic type

```
module logic_data_type(input logic rst_h);
  parameter CYCLE = 20;
  logic q, q_l, d, clk, rst_l;
  initial begin
    clk = 0;                    // Procedural assignment
    forever #(CYCLE/2) clk = ~clk;
  end

  assign rst_l = ~rst_h;        // Continuous assignment
  not n1(q_l, q);               // q_l is driven by gate
  my_dff d1(q, d, clk, rst_l); // q is driven by module

endmodule
```

You can use the `logic` type to find netlist bugs as this type can only have a single driver. Rather than trying to choose between `reg` and `wire`, declare all your signals as `logic`, and you'll get a compilation error if it has multiple drivers. Of course, any signal that you do want to have multiple drivers, such as a bidirectional bus, should be declared with a net type such as `wire`.

2.1.2 2-State Data Types

SystemVerilog introduces several 2-state data types to improve simulator performance and reduce memory usage, compared with variables declared as 4-state types. The simplest type is the `bit`, which is always unsigned. There are four signed 2-state types: `byte`, `shortint`, `int`, and `longint` as shown in Sample 2.2.

Sample 2.2 Signed data types

```
bit b;           // 2-state, single-bit
bit [31:0] b32;  // 2-state, 32-bit unsigned integer
int unsigned ui; // 2-state, 32-bit unsigned integer
int i;           // 2-state, 32-bit signed integer
byte b8;         // 2-state, 8-bit signed integer
shortint s;      // 2-state, 16-bit signed integer
longint l;       // 2-state, 64-bit signed integer
integer i4;      // 4-state, 32-bit signed integer
time t;          // 4-state, 64-bit unsigned integer
real r;          // 2-state, double precision floating pt
```

You might be tempted to use types such as `byte` to replace more verbose declarations such as `logic [7:0]`. Hardware designers should be careful as these new types are signed variables, and so a `byte` variable can only count up to 127, not the 255 you may expect. (It has the range –128 to +127.) You could use `byte unsigned`, but that is more verbose than just `bit [7:0]`. Signed variables can also cause unexpected results with randomization, as discussed in Chap. 6.

Be careful connecting 2-state variables to the design under test, especially its outputs. If the hardware tries to drive an X or Z, these values are converted to a 2-state value, and your testbench code may never know. Don't try to remember if they are converted to 0 or 1; instead, always check for propagation of unknown values. Use the `$isunknown()` operator that returns 1 if any bit of the expression is X or Z, as shown in Sample 2.3.

Sample 2.3 Checking for 4-state values

```
if ($isunknown(iport) == 1)
  $display("@%0t: 4-state value detected on iport %b",
           $time, iport);
```

The format `%0t` and the argument `$time` print the current simulation time, formatted as specified with the `$timeformat`() routine. Time values are explored in more detail in Section 3.7.

2.2 Fixed-Size Arrays

SystemVerilog offers several flavors of arrays beyond the single-dimension, fixed-size Verilog-1995 arrays. Many enhancements have been made to these classic arrays.

2.2.1 Declaring and Initializing Fixed-Size Arrays

Verilog requires that the low and high array limits must be given in the declaration. Since almost all arrays use a low index of 0, SystemVerilog lets you use the shortcut of just giving the array size, which is similar to C's style.

Sample 2.4 Declaring fixed-size arrays

```
int lo_hi[0:15];        // 16 ints [0]..[15]
int c_style[16];        // 16 ints [0]..[15]
```

You can create multidimensional fixed-size arrays by specifying the dimensions after the variable name. Sample 2.5 creates several two-dimensional arrays of integers, 8 entries by 4, and sets the last entry to 1. Multidimensional arrays were introduced in Verilog-2001, but the compact declaration style is new.

Sample 2.5 Declaring and using multidimensional arrays

```
int array2 [0:7][0:3];  // Verbose declaration
int array3 [8][4];      // Compact declaration
array2[7][3] = 1;       // Set last array element
```

If your code accidently tries to read from an out-of-bounds address, SystemVerilog will return the default value for the array element type. That just means that an array of 4-state types, such as `logic`, will return X's, whereas an array of 2-state types, such as `int` or `bit`, will return 0. This applies for all array types – fixed, dynamic, associative, or queue, and also if your address has an X or Z. An undriven net is Z.

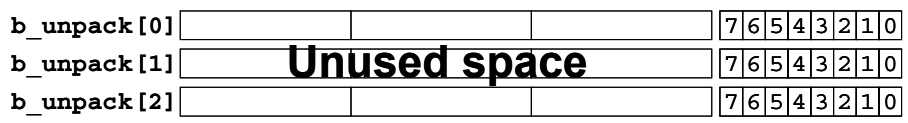
Many SystemVerilog simulators store each element on a 32-bit word boundary. So a `byte`, `shortint`, and `int` are all stored in a single word, whereas a `longint` is stored in two words.

An unpacked array, such as the one shown in Sample 2.6, stores the values in the lower portion of the word, whereas the upper bits are unused. The array of bytes, `b_unpack`, is stored in three words, as shown in Figure 2-1.

Sample 2.6 Unpacked array declarations

```
bit [7:0] b_unpack[3];   // Unpacked
```

Figure 2-1 Unpacked array storage



Packed arrays are explained in Section 2.2.6.

Simulators generally store 4-state types such as `logic` and `integer` in two or more consecutive words, using twice the storage as 2-state variables.

2.2.2 The Array Literal

Sample 2.7 shows how to initialize an array using an array literal, which is an apostrophe followed by the values in curly braces. (This is not the accent grave used for compiler directives and macros.) You can set some or all elements at once. You are able to replicate values by putting a count before the curly braces. Lastly, you might specify a default value for any element that does not have an explicit value.

Sample 2.7 Initializing an array

```
int ascend[4] = '{0,1,2,3}; // Initialize 4 elements
int descend[5];

descend = '{4,3,2,1,0};         // Set 5 elements
descend[0:2] = '{5,6,7};        // Set first 3 elements
ascend = '{4{8}};               // Four values of 8
descend = '{9, 8, default:-1}; // {9, 8, -1, -1, -1}
```

2.2.3 Basic Array Operations – `for` and `foreach`

The most common way to manipulate an array is with a `for`- or `foreach`-loop. In Sample 2.8, the variable `i` is declared local to the `for`-loop. The SystemVerilog function `$size` returns the size of the array. In the `foreach`-loop, you specify the array name and an index in square brackets, and SystemVerilog automatically steps through all the elements of the array. The index variable is automatically declared for you and is local to the loop.

Sample 2.8 Using arrays with `for-` and `foreach`-loops

```
initial begin
  bit [31:0] src[5], dst[5];
  for (int i=0; i<$size(src); i++)
    src[i] = i;
  foreach (dst[j])
    dst[j] = src[j] * 2;  // dst doubles src values
end
```

Note that in Sample 2.9, the syntax of the `foreach`-loop for multidimensional arrays may not be what you expected! Instead of listing each subscript in separate square brackets – `[i][j]` – they are combined with a comma – `[i,j]`.

Sample 2.9 Initialize and step through a multidimensional array

```
int md[2][3] = '{'{0,1,2}, '{3,4,5}};
initial begin
  $display("Initial value:");
  foreach (md[i,j])   // Yes, this is the right syntax
    $display("md[%0d][%0d] = %0d", i, j, md[i][j]);

  $display("New value:");
  // Replicate last 3 values of 5
  md = '{'{9, 8, 7}, '{3{'5}}};
  foreach (md[i,j])   // Yes, this is the right syntax
    $display("md[%0d][%0d] = %0d", i, j, md[i][j]);
end
```

The output from Sample 2.9 is shown in Sample 2.10.

Sample 2.10 Output from printing multidimensional array values

```
Initial value:
md[0][0] = 0
md[0][1] = 1
md[0][2] = 2
md[1][0] = 3
md[1][1] = 4
md[1][2] = 5
New value:
md[0][0] = 9
md[0][1] = 8
md[0][2] = 7
md[1][0] = 5
md[1][1] = 5
md[1][2] = 5
```

You can omit some dimensions in the `foreach`-loop if you don't need to step through all of them. Sample 2.11 prints a two-dimensional array in a rectangle. It

steps through the first dimension in the outer loop, and then through the second dimension in the inner loop.

Sample 2.11 Printing a multidimensional array

```
initial begin
  byte twoD[4][6];
  foreach(twoD[i,j])
    twoD[i][j] = i*10+j;

  foreach (twoD[i]) begin    // Step through first dim.
    $write("%2d:", i);
    foreach(twoD[,j])        // Step through second
      $write("%3d", twoD[i][j]);
    $display;
  end
end
```

Sample 2.11 produces the following output.

Sample 2.12 Output from printing multidimensional array values

```
 0:  0  1  2  3  4  5
 1: 10 11 12 13 14 15
 2: 20 21 22 23 24 25
 3: 30 31 32 33 34 35
```

Lastly, a `foreach`-loop iterates using the ranges in the original declaration. The array `f[5]` is equivalent to `f[0:4]`, and a `foreach(f[i])` is equivalent to `for(int i=0;i<=4; i++)`. With the array `rev[6:2]`, the statement `foreach(rev[i])` is equivalent to `for(int i=6; i>=2; i--)`.

2.2.4 Basic Array Operations – Copy and Compare

You can perform aggregate compare and copy of arrays without loops. (An aggregate operation works on the entire array as opposed to working on just an individual element.) Comparisons are limited to just equality and inequality. Sample 2.13 shows several examples of compares. The `? :` conditional operator is a mini `if`-statement. In Sample 2.13, it is used to choose between two strings. The final compare uses an array slice, `src[1:4]`, which creates a temporary array with four elements.

Sample 2.13 Array copy and compare operations

```
initial begin
  bit [31:0] src[5] = '{0,1,2,3,4},
             dst[5] = '{5,4,3,2,1};

  // Aggregate compare the two arrays
  if (src==dst)
    $display("src == dst");
  else
    $display("src != dst");

  // Aggregate copy all src values to dst
  dst = src;

  // Change just one element
  src[0] = 5;

  // Are all values equal (no!)
  $display("src %s dst", (src == dst) ? "==" : "!=");

  // Use array slice to compare elements 1-4
  // $display("src[1:4] %s dst [1:4]",
               src[1:4] == dst[1:4]) ? "==" : "!=");
end
```

You cannot perform aggregate arithmetic operations such as addition on arrays. Instead, you can use loops. For logical operations such as `xor`, you have to either use a loop or use packed arrays as described in Section 2.2.6.

2.2.5 Bit and Array Subscripts, Together at Last

A common annoyance in Verilog-1995 is that you cannot use array and bit subscripts together. Verilog-2001 removes this restriction for fixed-size arrays. Sample 2.14 prints the first array element (binary 101), its lowest bit (1), and the next two higher bits (binary 10).

Sample 2.14 Using word and bit subscripts together

```
initial begin
  bit [31:0] src[5] = '{5{5}};
  $displayb(src[0],,          // 'b101 or 'd5
            src[0][0],,       // 'b1
            src[0][2:1]);     // 'b10
end
```

Although this change is not new to SystemVerilog, many users may not know about this useful improvement in Verilog-2001.

2.2.6 Packed Arrays

For some data types, you may want both to access the entire value and also to divide it into smaller elements. For example, you may have a 32-bit register that sometimes you want to treat as four 8-bit values and at other times as a single, unsigned value. A SystemVerilog packed array is treated as both an array and a single value. It is stored as a contiguous set of bits with no unused space, unlike an unpacked array.

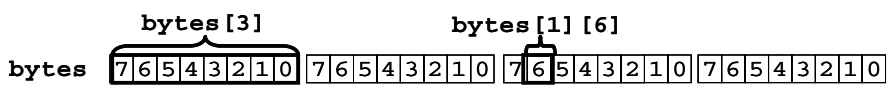
2.2.7 Packed Array Examples

The packed bit and array dimensions are specified as part of the type, before the variable name. These dimensions must be specified in the `[msb:lsb]` format, not `[size]`. Sample 2.15 shows the variable `bytes`, a packed array of four bytes that are stored in a single 32-bit word as shown in Figure 2-2.

Sample 2.15 Packed array declaration and usage

```
bit [3:0] [7:0] bytes;  // 4 bytes packed into 32-bits
bytes = 32'hCafe_Dada;
$displayh(bytes,,        // Show all 32-bits
          bytes[3],,     // Most significant byte "CA"
          bytes[3][7]);  // Most significant bit  "1"
```

Figure 2-2 Packed array layout



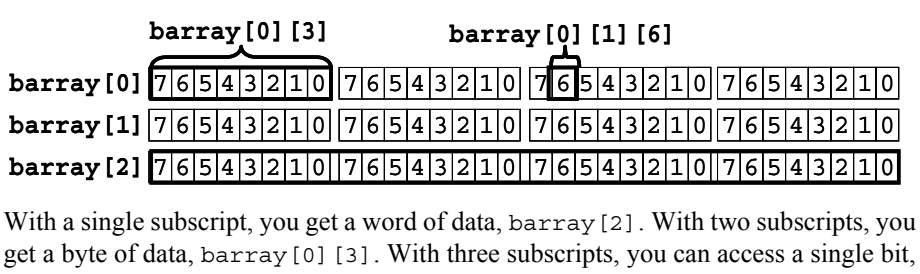
You can mix packed and unpacked dimensions. You may want to make an array that represents a memory that can be accessed as bits, bytes, or longwords. In Sample 2.16, `barray` is an unpacked array of three packed elements.

Sample 2.16 Declaration for a mixed packed/unpacked array

```
bit [3:0] [7:0] barray [3];    // Packed: 3x32-bit
bit [31:0] lw = 32'h0123_4567; // Word
bit [7:0] [3:0] nibbles;       // Packed array of nibbles
barray[0] = lw;
barray[0][3] = 8'h01;
barray[0][1][6] = 1'b1;
nibbles = barray[0];           // Copy packed values
```

The variable `bytes` in Sample 2.15 is a packed array of four bytes that are stored in a single word. `barray` is an array of three of these elements, which are stored in memory as shown in Figure 2-3.

Figure 2-3 Packed array bit layout



With a single subscript, you get a word of data, `barray[2]`. With two subscripts, you get a byte of data, `barray[0][3]`. With three subscripts, you can access a single bit, `barray[0][1][6]`. Note that because one dimension is specified after the name, `barray[3]`, that dimension is unpacked, and so you always need to use at least one subscript.

The last line of Sample 2.16 copies between two packed arrays. Since the underlying values are just bits, you can copy even if the arrays have different dimensions.

2.2.8 Choosing Between Packed and Unpacked Arrays

Which should you choose – a packed or an unpacked array? A packed array is handy if you need to convert to and from scalars. For example, you might need to reference a memory as a byte or as a word. The `barray` in Figure 2-3 can handle this requirement. Any array type can be packed, including dynamic arrays, queues, and associative arrays, which are explained in Sections 2.3–2.5.

If you need to wait for a change in an array, you have to use a packed array. Perhaps your testbench might need to wake up when a memory changes value, and so you want to use the `@` operator. This is however only legal with scalar values and packed arrays. In Sample 2.16 you can block on the variable `lw`, and `barray[0]`, but not the entire array `barray` unless you expand it: `@(barray[0] or barray[1] or barray[2])`.

2.3 Dynamic Arrays

The basic Verilog array type shown so far is known as a fixed-size array, as its size is set at compile time. What if you do not know the size of the array until run-time? For example, you may want to generate a random number of transactions at the start of simulation. If you stored the transactions in an fixed-size array, it would have to be large enough to hold the maximum number of transactions, but would typically hold far fewer, thus wasting memory. SystemVerilog provides a dynamic array that can be allocated and resized during simulation and so your simulation consumes a minimal amount of memory.

A dynamic array is declared with empty word subscripts `[]`. This means that you do not specify the array size at compile time; instead, you give it at run-time. The array is

initially empty, and so you must call the `new[]` constructor to allocate space, passing in the number of entries in the square brackets. If you pass the name of an array to the `new[]` constructor, the values are copied into the new elements, as shown in Sample 2.17.

Sample 2.17 Using dynamic arrays

```
int dyn[], d2[];                  // Declare dynamic arrays

initial begin
  dyn = new[5];                   // A: Allocate 5 elements
  foreach (dyn[j]) dyn[j] = j;  // B: Initialize the elements
  d2 = dyn;                       // C: Copy a dynamic array
  d2[0] = 5;                      // D: Modify the copy
  $display(dyn[0],d2[0]);         // E: See both values (0 & 5)
  dyn = new[20](dyn);             // F: Allocate 20 ints & copy
  dyn = new[100];                 // G: Allocate 100 new ints
                                  //    Old values are lost
  dyn.delete();                   // H: Delete all elements
end
```

In Sample 2.17, Line A calls `new[5]` to allocate 5 array elements. The dynamic array `dyn` now holds 5 `int`'s. B sets the value of each element of the array to its index value. Line C allocates another array and copies the contents of `dyn` into it. Lines D and E show that the arrays `dyn` and `d2` are separate. Line E allocates 20 new elements, and copies the existing 5 elements of `dyn` to the beginning of the array. Then the old 5-element array is deallocated. The result is that `dyn` points to a 20-element array. The last call to `new[]` allocates 100 elements, but the existing values are not copied. The old 20-element array is deallocated. Finally, line H deletes the `dyn` array.

The `$size` function returns the size of an array. Dynamic arrays have several built-in routines, such as `delete` and `size`.

If you want to declare a constant array of values but do not want to bother counting the number of elements, use a dynamic array with an array literal. In Sample 2.18, there are 9 masks for 8 bits, but you should let SystemVerilog count them, rather than making a fixed-size array and accidently choosing the wrong size of 8.

Sample 2.18 Using a dynamic array for an uncounted list

```
bit [7:0] mask[] = '{8'b0000_0000, 8'b0000_0001,
                     8'b0000_0011, 8'b0000_0111,
                     8'b0000_1111, 8'b0001_1111,
                     8'b0011_1111, 8'b0111_1111,
                     8'b1111_1111};
```

You can make assignments between fixed-size and dynamic arrays as long as they have the same base type such as `int`. You can assign a dynamic array to a fixed array as long as they have the same number of elements.

When you copy a fixed-size array to a dynamic array, SystemVerilog calls the `new[]` constructor to allocate space, and then copies the values.

2.4 Queues

SystemVerilog introduces a new data type, the queue, which combines the best of an linked list and array. Like a linked list, you can add or remove elements anywhere in a queue, without the performance hit of a dynamic array that has to allocate a new array and copy the entire contents. Like an array, you can directly access any element with an index, without linked list's overhead of stepping through the preceding elements.

A queue is declared with word subscripts containing a dollar sign: `[$]`. The elements of a queue are numbered from 0 to $. Sample 2.19 shows how you can add and remove values from a queue using methods. Note that queue literals only have curly braces, and are missing the initial apostrophe of array literals.

The SystemVerilog queue is similar to the Standard Template Library's deque data type. You create a queue by adding elements. SystemVerilog typically allocates extra space so that you can quickly insert additional elements. If you add enough elements that the queue runs out of that extra space, SystemVerilog automatically allocates more. As a result, you can grow and shrink a queue without the performance penalty of a dynamic array, and SystemVerilog keeps track of the free space for you. Note that you never call the `new[]` constructor for a queue.

Sample 2.19 Queue operations

```
int j = 1,
    q2[$] = {3,4},          // Queue literals do not use '
    q[$] = {0,2,5};         // {0,2,5}

initial begin
  q.insert(1, j);           // {0,1,2,5}     Insert 1 before 2
  q.insert(3, q2);          // {0,1,2,3,4,5} Insert queue in q[1]
  q.delete(1);              // {0,2,3,4,5}   Delete elem. #1

  // These operations are fast
  q.push_front(6);          // {6,0,2,3,4,5} Insert at front
  j = q.pop_back;           // {6,0,2,3,4}   j = 5
  q.push_back(8);           // {6,0,2,3,4,8} Insert at back
  j = q.pop_front;          // {0,2,3,4,8}   j = 6
  foreach (q[i])
    $display(q[i]);         //               Print entire queue
  q.delete();               // {}            Delete entire queue
end
```

You can use word subscripts and concatenation instead of methods. As a shortcut, if you put a $ on the left side of a range, such as `[$:2]`, the `$` stands for the minimum value, `[0:2]`. A `$` on the right side, as in `[1:$]`, stands for the maximum value, `[1:2]`, in first line of the initial block of Sample 2.20.

Sample 2.20 Queue operations

```
int j = 1,
    q2[$] = {3,4},          // Queue literals do not use '
    q[$] = {0,2,5};         // {0,2,5}

initial begin               // Result
  q = {q[0], j, q[1:$]};    // {0,1,2,5}     Insert 1 before 2
  q = {q[0:2], q2, q[3:$]}; // {0,1,2,3,4,5} Insert queue in q
  q = {q[0], q[2:$]};       // {0,2,3,4,5}   Delete elem. #1

  // These operations are fast
  q = {6, q};               // {6,0,2,3,4,5} Insert at front
  j = q[$];                 // j = 5         Equivalent of
  q = q[0:$-1];             // {6,0,2,3,4}      pop_back
  q = {q, 8};               // {6,0,2,3,4,8} Insert at back
  j = q[0];                 // j = 6         Equivalent of
  q = q[1:$];               // {0,2,3,4,8}      pop_front

  q = {};                   // {}            Delete entire queue
end
```

[1]Not all SystemVerilog simulators support inserting a queue with the insert() method.

The queue elements are stored in contiguous locations, and so it is efficient to push and pop elements from the front and back. This takes a fixed amount of time no matter how large the queue. Adding and deleting elements in the middle of a queue requires shifting the existing data to make room. The time to do this grows linearly with the size of the queue.

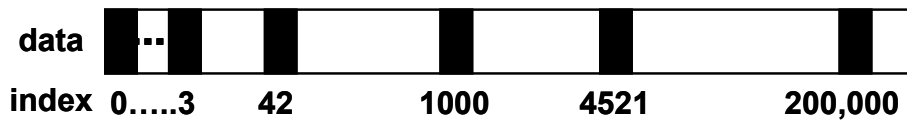
You can copy the contents of a fixed or dynamic array into a queue.

2.5 Associative Arrays

Dynamic arrays are good if you want to occasionally create a big array, but what if you want something really large? Perhaps you are modeling a processor that has a multi-gigabyte address range. During a typical test, the processor may only touch a few hundred or thousand memory locations containing executable code and data, so allocating and initializing gigabytes of storage is wasteful.

SystemVerilog offers associative arrays that store entries in a sparse matrix. This means that while you can address a very large address space, SystemVerilog only allocates memory for an element when you write to it. In the following picture, the associative array holds the values 0:3, 42, 1,000, 4,521, and 200,000. The memory used to store these is far less than would be needed to store a fixed or dynamic array with 200,000 entries.

Figure 2-4 Associative array



An associative array can be stored by the simulator as a tree or hash table. This additional overhead is acceptable when you need to store arrays with widely separated index values, such as packets indexed with 32-bit addresses or 64-bit data values.

An associative array is declared with a data type[2] in square brackets, such as `[int]` or `[Packet]`. Sample 2.21 shows declaring, initializing, and stepping through an associative array.

[2]You can also declare an associative array with wildcard subscripts, as in wild[*]. However, this style is not recommended as you are allowing subscripts of any data type. One of the many problems is with foreach-loops – what type is the variable j in foreach(wild[j])?

Sample 2.21 Declaring, initializing, and using associative arrays

```
initial begin
  bit [63:0] assoc[int], idx = 1;

  // Initialize widely scattered values
  repeat (64) begin
    assoc[idx] = idx;
    idx = idx << 1;
  end

  // Step through all index values with foreach
  foreach (assoc[i])
    $display("assoc[%h] = %h", i, assoc[i]);

  // Step through all index values with functions
  if (assoc.first(idx))
    begin                    // Get first index
    do
      $display("assoc[%h]=%h", idx, assoc[idx]);
    while (assoc.next(idx));  // Get next index
  end

  // Find and delete the first element
  assoc.first(idx);
  assoc.delete(idx);
  $display("The array now has %0d elements", assoc.num);
end
```

Sample 2.21 has the associative array, `assoc`, with very scattered elements: 1, 2, 4, 8, 16, etc. A simple `for`-loop cannot step through them; you need to use a `foreach`-loop. If you want finer control, you can use the `first` and `next` functions in a `do...while` loop. These functions modify the index argument, and return 0 or 1 depending on whether any elements are left in the array.

Associative arrays can also be addressed with a string index, similar to Perl's hash arrays. Sample 2.22 reads a file with strings and builds the associative array `switch` so that you can quickly map from a string value to a number. Strings are explained in more detail in Section 2.14. You can use the function `exists()` to check if an element exists, as shown in Sample 2.22. If you try to read an element that has not been written, SystemVerilog returns the default value for the array type, such as 0 for 2-state types, or X for 4-state types.

Sample 2.22 Using an associative array with a string index

```
/*
Input file contains:
  42   min_address
  1492 max_address
*/
int switch[string], min_address, max_address;
initial begin
  int i, r, file;
  string s;
  file = $fopen("switch.txt", "r");
  while (! $feof(file)) begin
    r = $fscanf(file, "%d %s", i, s);
    switch[s] = i;
  end
  $fclose(file);

  // Get the min address, default is 0
  min_address = switch["min_address"];

  // Get the max address, default = 1000
  if (switch.exists("max_address"))
    max_address = switch["max_address"];
  else
    max_address = 1000;

  // Print all switches
  foreach (switch[s])
    $display("switch['%s']=%0d", s, switch[s]);
end
```

2.6 Linked Lists

SystemVerilog provides a linked list data-structure that is analogous to the STL (Standard Template Library) List container. The container is defined as a parameterized class, meaning that it can be customized to hold data of any type.

Now that you know there is a linked list in SystemVerilog, avoid using it. C++ programmers might be familiar with the STL version, but SystemVerilog's queues are more efficient and easier to use.

2.7 Array Methods

There are many array methods that you can use on any unpacked array types: fixed, dynamic, queue, and associative. These routines can be as simple as giving the current array size or as complex as sorting the elements. The parentheses are optional if there are no arguments.

2.7.1 Array Reduction Methods

A basic array reduction method takes an array and reduces it to a single value, as shown in Sample 2.23. The most common reduction method is `sum`, which adds together all the values in an array. Be careful of SystemVerilog's rules for handling the width of operations. By default, if you add the values of a single-bit array, the result is a single bit. However, if you use it in a 32-bit expression, store the result in a 32-bit variable, compare it to a 32-bit variable, or use the proper `with` expression. SystemVerilog uses 32-bits when adding up the values. The `with` expression is described in Section 2.7.2.

Sample 2.23 Creating the sum of an array

```
bit on[10];  // Array of single bits
int total;

initial begin
  foreach (on[i])
    on[i] = i;   // on[i] gets 0 or 1

  // Print the single-bit sum
  $display("on.sum = %0d", on.sum); // on.sum = 1

  // Print the sum using 32-bit total
  $display("on.sum = %0d", on.sum + 32'd0); // on.sum = 5

  // Sum the values using 32-bits as total is 32-bits
  total = on.sum;
  $display("total = %0d", total);     // total = 5

  // Compare the sum to a 32-bit value
  if (on.sum >=32'd5)               // True
    $display("sum has 5 or more 1's");

  // Compute with 32-bit signed arithimetic
  $display("int sum=%0d", on.sum with (int'(item)));
  end
```

Other array reduction methods are `product`, `and`, `or`, and `xor`.

SystemVerilog does not have a method specifically for choosing a random element from an array, and so use the index `$urandom_range(array.size()-1)` for queues and dynamic arrays, and `$urandom_range($size(array)-1)` for fixed arrays, queues, dynamic, and associative arrays. See Section 6.10 for more information on `$urandom_range`.

If you need to choose a random element from an associative array, you need to step through the elements one by one as there is no direct way to access the *N*th element. Sample 2.24 shows how to pick a random element from an associative array indexed by integers. If the array was indexed by a string, just change the type of `idx` to `string`.

Sample 2.24 Picking a random element from an associative array

```
int aa[int], rand_idx, element, count;

element = $urandom_range(aa.size()-1);
foreach(aa[i])
  if (count++ == element) begin
    rand_idx = i;   // Save the associative array index
    break;          //    and quit
  end

$display("%0d element aa[%0d] = %0d",
         element, rand_idx, aa[rand_idx]);
```

2.7.2 Array Locator Methods

What is the largest value in an array? Does an array contain a certain value? The array locator methods find data in an unpacked array. These methods always return a queue.

Sample 2.25 uses a fixed-size array, `f[6]`, a dynamic array, `d[]`, and a queue, `q[$]`. The `min` and `max` functions find the smallest and largest elements in an array. Note that they return a queue, not a scalar as you might expect. These methods also work for associative arrays. The `unique` method returns a queue of the unique values from the array – duplicate values are not included.

Sample 2.25 Array locator methods: `min`, `max`, `unique`

```
int f[6] = '{1,6,2,6,8,6};
int d[]  = '{2,4,6,8,10};
int q[$] = {1,3,5,7}, tq[$];

tq = q.min();                          // {1}
tq = d.max();                          // {10}
tq = f.unique();                       // {1,6,2,8}
```

You could search through an array using a `foreach`-loop, but SystemVerilog can do this in one operation with a locator method. The `with` expression tells SystemVerilog how to perform the search, as shown in Sample 2.26.

Sample 2.26 Array locator methods: `find`

```
int d[] = '{9,1,8,3,4,4}, tq[$];

// Find all elements greater than 3
tq = d.find with (item > 3);           // {9,8,4,4}
// Equivalent code
tq.delete();
foreach (d[i])
  if (d[i] > 3)
    tq.push_back(d[i]);

tq = d.find_index with (item > 3);     // {0,2,4,5}
tq = d.find_first with (item > 99);    // {} - none found
tq = d.find_first_index with (item==8); // {2} d[2]=8
tq = d.find_last with (item==4);       // {4}
tq = d.find_last_index with (item==4); // {5} d[5]=4
```

In a `with` clause, the name `item` is called the iterator argument and represents a single element of the array. You can specify your own name by putting it in the argument list of the array method as shown in Sample 2.27.

Sample 2.27 Declaring the iterator argument

```
tq = d.find_first with (item==4);      // These
tq = d.find_first() with (item==4);    // are
tq = d.find_first(item) with (item==4); // all
tq = d.find_first(x) with (x==4);      // equivalent
```

Sample 2.28 shows various ways to total up a subset of the values in the array. The first total compares the item with 7. This relational returns a 1 (true) or 0 (false) and multiplies this with the array. So the sum of {9,0,8,0,0,0} is 17. The second total is computed using the ? : conditional operator.

Sample 2.28 Array locator methods

```
int count, total, d[] = '{9,1,8,3,4,4};

count = d.sum with (item >  7);  // 2: {9, 8}
total = d.sum with ((item > 7) * item); // 17= 9+8
count = d.sum with (item <  8);  // 4: {1, 3, 4, 4}
total = d.sum with (item < 8 ? item : 0); // 12=1+3+4+4
count = d.sum with (item == 4);  // 2: {4, 4}
```

When you combine an array reduction such as `sum` using the `with` clause, the results may surprise you. In Sample 2.28, the `sum` operator totals the number of times that the expression is true. For the first statement in Sample 2.28, there are two array elements that are greater than 7 (9 and 8) and so `count` is set to 2.

The array locator methods that return an index, such as **`find_index`**, return a queue of type **`int`**, not **`integer`**. Your code may not compile if you use the wrong queue type with these statements.

2.7.3 Array Sorting and Ordering

SystemVerilog has several methods for changing the order of elements in an array. You can sort the elements, reverse their order, or shuffle the order as shown in Sample 2.29. Notice that these change the original array, unlike the array locator methods in Section 2.7.2, which create a queue to hold the results.

Sample 2.29 Sorting an array

```
int d[] =         '{9,1,8,3,4,4};
d.reverse();  // '{4,4,3,8,1,9}
d.sort();     // '{1,3,4,4,8,9}
d.rsort();    // '{9,8,4,4,3,1}
d.shuffle();  // '{9,4,3,8,1,4}
```

The `reverse` and `shuffle` methods have no `with`-clause, and so they work on the entire array. Sample 2.30 shows how to sort a structure by sub-fields. Structures and packed structures are explained in Section 2.10.

Sample 2.30 Sorting an array of structures

```
struct packed { byte red, green, blue; } c[];
initial begin
  c = new[100];          // Allocate 100 pixels
  foreach(c[i])
    c[i] = $urandom;     // Fill with random values
  c.sort with (item.red); // sort using red only

  // sort by green value then blue
  c.sort(x) with ({x.green, x.blue});
end
```

2.7.4 Building a Scoreboard with Array Locator Methods

The array locator methods can be used to build a scoreboard. Sample 2.31 defines the `Packet` structure, then creates a scoreboard made from a queue of these structures. Section 2.9 describes how to create structures with `typedef`.

Sample 2.31 A scoreboard with array methods

```
typedef struct packed
  {bit [7:0] addr;
   bit [7:0] pr;
   bit [15:0] data; } Packet;

Packet scb[$];

function void check_addr(bit [7:0] addr);
  int intq[$];

  intq = scb.find_index() with (item.addr == addr);
  case (intq.size())
  0: $display("Addr %h not found in scoreboard", addr);
  1: scb.delete(intq[0]);
  default:
    $display("ERROR: Multiple hits for addr %h", addr);
  endcase
endfunction : check_addr
```

The `check_addr()` function in Sample 2.31 looks up an address in the scoreboard. The `find_index()` method returns an `int` queue. If the queue is empty (size==0), no match was found. If the queue has one member (size==1), a single match was found, which the `check_addr()` function deletes. If the queue has multiple members (size > 1), there are multiple packets in the scoreboard whose address matches the requested one.

A better choice for storing packet information is a class, which is described in Chap. 5. You can read more about structures in Section 2.10.

2.8 Choosing a Storage Type

Here are some guidelines for choosing the right storage type based on flexibility, memory usage, speed, and sorting. These are just rules of thumb, and results may vary between simulators.

2.8.1 Flexibility

Use a fixed-size or dynamic array if it is accessed with consecutive positive integer indices: 0, 1, 2, 3…. Choose a fixed-size array if the array size is known at compile time, or choose a dynamic array if the size is not known until run-time. For example, variable-size packets can easily be stored in a dynamic array. If you are writing routines to manipulate arrays, consider using just dynamic arrays, as one routine can work with any size dynamic array as long as the element type (`int`, `string`, etc.) matches. Likewise, you can pass a queue of any size into a routine as long as the element type matches the queue argument. Associative arrays can also be passed regardless of size. However, a routine with a fixed-size array argument only accepts arrays of the specified length.

Choose associative arrays for nonstandard indices such as widely separated values because of random values or addresses. Associative arrays can also be used to model content-addressable memories.

Queues are a good way to store values when the number of elements grows and shrinks a lot during simulation, such as a scoreboard that holds expected values.

2.8.2 Memory Usage

If you want to reduce the simulation memory usage, use 2-state elements. You should choose data sizes that are multiples of 32 bits to avoid wasted space. Simulators usually store anything smaller in a 32-bit word. For example, an array of 1,024 bytes wastes ¾ of the memory if the simulator puts each element in a 32-bit word. Packed arrays can also help conserve memory.

For arrays that hold up to a thousand elements, the type of array that you choose does not make a big difference in memory usage (unless there are many instances of these arrays). For arrays with a thousand to a million active elements, fixed-size and dynamic arrays are the most memory efficient. You may want to reconsider your algorithms if you need arrays with more than a million active elements.

Queues are slightly less efficient to access than fixed-size or dynamic arrays because of additional pointers. However, if your data set grows and shrinks often, and you

store it in a dynamic memory, you will have to manually call `new[]` to allocate memory and copy. This is an expensive operation and would wipe out any gains from using a dynamic memory.

Modeling memories larger than a few megabytes should be done with an associative array. Note that each element in an associative array can take several times more memory than a fixed-size or dynamic memory because of pointer overhead.

2.8.3 Speed

Choose your array type based on how many times it is accessed per clock cycle. For only a few reads and writes, you could use any type, as the overhead is minor compared with the DUT. As you use an array more often, its size and type matters.

Fixed-size and dynamic arrays are stored in contiguous memory, and so any element can be found in the same amount of time, regardless of array size.

Queues have almost the same access time as a fixed-size or dynamic array for reads and writes. The first and last elements can be pushed and popped with almost no overhead. Inserting or removing elements in the middle requires many elements to be shifted up or down to make room. If you need to insert new elements into a large queue, your testbench may slow down, and so consider changing how you store new elements.

When reading and writing associative arrays, the simulator must search for the element in memory. The LRM does not specify how this is done, but popular ways are hash tables and trees. These require more computation than other arrays, and therefore associative arrays are the slowest.

2.8.4 Sorting

Since SystemVerilog can sort any single-dimension array (fixed-size, dynamic, and associative arrays plus queues), you should pick based on how often the values are added to the array. If the values are received all at once, choose a fixed-size or dynamic array so that you only have to allocate the array once. If the data slowly dribbles in, choose a queue, as adding new elements to the head or tail is very efficient.

If you have unique and noncontiguous values, such as `'{1, 10, 11, 50}`, you can store them in an associative array by using them as an index. Using the routines `first`, `next`, and `prev`, you can search an associative array for a value and find successive values. Lists are doubly linked, and so you can find values both larger and smaller than the current value. Both of these support removing a value. However, the associative array is much faster in accessing any given element, given an index.

For example, you can use an associative array of bits to hold expected 32-bit values. When the value is created, write to that location. When you need to see if a given

value has been written, use the `exists` function. When done with an element, use `delete` to remove it from the associative array.

2.8.5 Choosing the Best Data Structure

Here are some suggestions on choosing a data structure.

- *Network packets*. Properties: fixed size, accessed sequentially. Use a fixed-size or dynamic array for fixed- or variable-size packets.
- *Scoreboard of expected values*. Properties: size not known until run-time, accessed by value, and a constantly changing size. In general, use a queue, as you are continually adding and deleting elements during simulation. If you can give every transaction a fixed id, such as 1, 2, 3, …, you could use this as an index into the queue. If your transaction is filled with random values, you can just push them into a queue and search for unique values. If the scoreboard may have hundreds of elements, and you are often inserting and deleting them from the middle, an associative array may be faster.
- *Sorted structures*. Use a queue if the data comes out in a predictable order, or an associative array if the order is unspecified. If the scoreboard never needs to be searched, just store the expected values in a mailbox, as shown in Section 7.6.
- *Modeling very large memories, greater than a million entries*. If you do not need every location, use an associative array as a sparse memory. If you do need every location, try a different approach where you do not need so much live data. Still stuck? Be sure to use 2-state values packed into 32-bits.
- *Command names or opcodes from a file*. Property: translate a string to a fixed value. Read string from a file, and then look up the commands or opcodes in an associative array using the command as a string index.

You can create an array of handles that point to objects, as shown in Chap. 5 on Basic OOP.

2.9 Creating New Types with `typedef`

You can create new types using the `typedef` statement. For example, you may have an ALU that can be configured at compile-time to use 8, 16, 24, or 32-bit operands. In Verilog you would define a macro for the operand width and another for the type as shown in Sample 2.32.

Sample 2.32 User-defined type-macro in Verilog

```
// Old Verilog style
`define OPSIZE 8
`define OPREG reg [`OPSIZE-1:0]

`OPREG op_a, op_b;
```

You are not really creating a new type; you are just performing text substitution. In SystemVerilog you create a new type with the following code. This book uses the convention that user-defined types use the suffix "`_t`."

Sample 2.33 User-defined type in SystemVerilog

```
// New SystemVerilog style
parameter OPSIZE = 8;
typedef reg [OPSIZE-1:0] opreg_t;

opreg_t op_a, op_b;
```

In general, SystemVerilog lets you copy between these basic types with no warning, either extending or truncating values if there is a width mismatch.

Note that `parameter` and `typedef` statements can be put in a package so that they can be shared across the design and testbench, as shown in Section 4.6.

One of the most useful types you can create is an unsigned, 2-state, 32-bit integer. Most values in a testbench are positive integers such as field length or number of transactions received, and so having a signed integer can cause problems. Put the definition of `uint` in a package of common definitions so that it can be used anywhere in your simulation.

Sample 2.34 Definition of `uint`

```
typedef bit [31:0] uint;     // 32-bit unsigned 2-state
typedef int unsigned uint;   // Equivalent definition
```

The syntax for defining a new array type is not obvious. You need to put the array subscripts on the new name. Sample 2.35 creates a new type, `fixed_array5`, which is a fixed array with 5 elements. It then declares an array of this type and initializes it.

Sample 2.35 User-defined array type

```
typedef int fixed_array5[5];
fixed_array5 f5;

initial begin
  foreach (f5[i])
    f5[i] = i;
end
```

2.10 Creating User-Defined Structures

One of the biggest limitations of Verilog is the lack of data structures. In SystemVerilog you can create a structure using the `struct` statement, similar to what is available in C. However, a `struct` has just a subset of the functionality of a class, and so use a class instead for your testbenches, as shown in Chap. 5. Just as a Verilog module combines both data (signals) and code (always/initial blocks plus routines), a class combines data and routines to make an entity that can be easily debugged and reused. A `struct` just groups data fields together. Without the code that manipulates the data, you are only creating half of the solution.

Since a `struct` is just a collection of data, it can be synthesized. If you want to model a complex data type, such as a pixel, in your design code, put it in a `struct`. This can also be passed through module ports. Eventually, when you want to generate constrained random data, look to classes.

2.10.1 Creating a `struct` and a New Type

You can combine several variables into a structure. Sample 2.36 creates a structure called `pixel` that has three unsigned bytes for red, green, and blue.

Sample 2.36 Creating a single pixel type

```
struct {bit [7:0] r, g, b;} pixel;
```

The problem with the preceding declaration is that it creates a single pixel of this type. To be able to share pixels using ports and routines, you should create a new type instead, as shown in Sample 2.37.

Sample 2.37 The pixel `struct`

```
typedef struct {bit [7:0] r, g, b;} pixel_s;
pixel_s my_pixel;
```

Use the suffix "`_s`" when declaring a `struct`. This makes it easier to spot user-defined types, simplifying the process of sharing and reusing code.

2.10.2 Initializing a Structure

You can assign multiple values to a struct just like an array, either in the declaration or in a procedural assignment. Just surround the values with an apostrophe and braces, as shown in Sample 2.38.

Sample 2.38 Initializing a struct

```
initial begin
  typedef struct {int a;
                  byte b;
                  shortint c;
                  int d;} my_struct_s;
  my_struct_s st = '{32'haaaa_aaaad,
                     8'hbb,
                     16'hcccc,
                     32'hdddd_dddd};

  $display("str = %x %x %x %x ", st.a, st.b, st.c, st.d);
end
```

2.10.3 Making a Union of Several Types

In hardware, the interpretation of a set of bits in a register may depend on the value of other bits. For example, a processor instruction may have many layouts based on the opcode. Immediate-mode operands might store a literal value in the operand field. This value may be decoded differently for integer instructions than for floating point instructions. Sample 2.39 stores both the integer `i` and the real `f` in the same location.

Sample 2.39 Using `typedef` to create a union

```
typedef union { int i; real f; } num_u;
num_u un;
un.f = 0.0; // set value in floating point format
```

Use the suffix "`_u`" when declaring a union.

Unions are useful when you frequently need to read and write a register in several different formats. However, don't go overboard, especially just to save memory. Unions may help squeeze a few bytes out of a structure, but at the expense of having to create and maintain a more complicated data structure. Instead, make a flat class with a discriminant variable, as shown in Section 8.4.4. This "kind" variable indicates which type of transaction you have, and thus which fields to read, write, and randomize. If you just need an array of values, plus all the bits, use a packed array as described in Section 2.2.6

2.10.4 Packed Structures

SystemVerilog allows you more control in how bits are laid out in memory by using packed structures. A packed structure is stored as a contiguous set of bits with no unused space. The `struct` for a pixel in Sample 2.37 used three values, and so it is stored in three longwords, even though it only needs three bytes. You can specify that it should be packed into the smallest possible space.

Sample 2.40 Packed structure

```
typedef struct packed {bit [7:0] r, g, b;} pixel_p_s;
pixel_p_s my_pixel;
```

Packed structures are used when the underlying bits represent a numerical value, or when you are trying to reduce memory usage. For example, you could pack together several bit-fields to make a single register. Or you might pack together the opcode and operand fields to make a value that contains an entire processor instruction.

2.10.5 Choosing Between Packed and Unpacked Structures

When you are trying to choose between packed and unpacked structures, consider how the structure is most commonly used, and the alignment of the elements. If you plan on making aggregate operations on the structure, such as copying the entire structure, a packed structure is more efficient. However, if your code accesses the individual members more than the entire structure, use an unpacked structure. The difference in performance is greater if the elements are not aligned on byte boundaries, have sizes that don't match the typical byte, or have word instructions used by processors. Reading and writing elements with odd sizes in a packed structure requires expensive shift and mask operations.

2.11 Type Conversion

The proliferation of data types in SystemVerilog means that you will need to convert between them. If the layout of the bits between the source and destination variables are the same, such as an integer and enumerated type, cast between the two values. If the bit layouts differ, such as an array of bytes and words, use the streaming operators to rearrange the bits.

2.11.1 The Static Cast

The static cast operation converts between two types with no checking of values. You specify the destination type, an apostrophe, and the expression to be converted as shown in Sample 2.41. Note that Verilog has always implicitly converted between types such as integer and real, and also between different width vectors.

Sample 2.41 Converting between int and real with static cast

```
int i;
real r;

i = int '(10.0 - 0.1); // cast is optional
r = real'(42);         // cast is optional
```

2.11.2 The Dynamic Cast

The dynamic cast, `$cast`, allows you to check for out-of-bounds values. See Section 2.12.3 for an explanation and example with enumerated types.

2.11.3 Streaming Operators

When used on the right side of an assignment, the streaming operators << and >> take an expression, structure, or array, and packs it into a stream of bits. The >> operator streams data from left to right while << streams from right to left, as shown in Sample 2.42. You can also give a slice size, which is used to break up the source before being streamed. You can not assign the bit stream result directly to an unpacked array. Instead, use the streaming operators on the left side of an assignment to unpack the bit stream into an unpacked array.

Sample 2.42 Basic streaming operator

```
initial begin
  int h;
  bit [7:0] b, g[4], j[4] = '{8'ha, 8'hb, 8'hc, 8'hd};
  bit [7:0] q, r, s, t;

  h = { >> {j}};         // 0a0b0c0d - pack array into int
  h = { << {j}};         // b030d050 reverse bits
  h = { << byte {j}};    // 0d0c0b0a reverse bytes
  g = { << byte {j}};    // 0d, 0c, 0b, 0a unpack into array
  b = { << {8'b0011_0101}};   // 1010_1100 reverse bits
  b = { << 4 {8'b0011_0101}}; // 0101_0011 reverse nibble
  {>> {q, r, s, t}} = j; // Scatter j into bytes
  h = {>>{t, s, r, q}};  // Gather bytes into h
end
```

You could do the same operations with many concatenation operators, {}, but the streaming operators are more compact and easier to read.

If you need to pack or unpack arrays, use the streaming operator to convert between arrays of different element sizes. For instance, you can convert an array of bytes to an array of words. You can use fixed size arrays, dynamic arrays, and queues. Sample 2.43 converts between queues, but would also work with dynamic arrays. Array elements are automatically allocated as needed.

Sample 2.43 Converting between queues with streaming operator

```
initial begin
  bit [15:0] wq[$] = {16'h1234, 16'h5678};
  bit [7:0]  bq[$];

  // Convert word array to byte
  bq = { >> {wq}};  // 12 34 56 78
  // Convert byte array to words
  bq = {8'h98, 8'h76, 8'h54, 8'h32};
  wq = { >> {bq}};  // 9876 5432
end
```

A common mistake when streaming between arrays is mismatched array subscripts. The word subscript `[256]` in an array declaration is equivalent to `[0:255]`, not `[255:0]`. Since many arrays are declared with the word subscripts `[high:low]`, streaming them to an array with the subscript `[size]` would result in the elements ending up in reverse order. Likewise, streaming an unpacked array declared as `bit [7:0] src[255:0]` to the packed array declared as `bit [7:0] [255:0] dst` will scramble the order of values. The correct declaration for a packed array of bytes is `bit [255:0] [7:0] dst`.

You can also use the streaming operator to pack and unpack structures, such as an ATM cell, into an array of bytes. In Sample 2.44, a structure is streamed into a dynamic array of bytes, and then the byte array is streamed back into the structure.

Sample 2.44 Converting between a structure and array with streaming operators

```
initial begin
  typedef struct {int a;
                  byte b;
                  shortint c;
                  int d;} my_struct_s;
  my_struct_s st = '{32'haaaa_aaaa,
                     8'hbb,
                     16'hcccc,
                     32'hdddd_dddd};
  byte b[];

  // Covert from struct to byte array
  b = { >> {st}};      // {aa aa aa aa bb cc cc dd dd dd dd}

  // Convert from byte array to a struct
  b = '{8'h11, 8'h22, 8'h33, 8'h44, 8'h55, 8'h66, 8'h77,
        8'h88, 8'h99, 8'haa, 8'hbb};
  st = { >> {b}};      // st = 11223344, 55, 6677, 8899aabb
end
```

2.12 Enumerated Types

Before enumerated types, you have to use text macros. Their global scope is too broad, and in most cases are visible in the debugger. An enumeration creates a strong variable type that is limited to a set of specified names, such as the instruction opcodes or state machine values. For example, the names ADD, MOVE, or ROTW make your code easier to write and maintain than if you had used literals such as `8'h01` or macros. Another alternative for defining constants is a parameter. These are fine for individual values, but an enumerated type automatically gives a unique value to every name in the list.

The simplest enumerated type declaration contains a list of constant names and one or more variables as shown in Sample 2.45. This creates an anonymous enumerated type, but it cannot be used for any other variables than the ones in this declaration.

Sample 2.45 A simple enumerated type

```
enum {RED, BLUE, GREEN} color;
```

In general you want to create a named enumerated type to easily declare multiple variables, especially if these are used as routine arguments or module ports. You first create the enumerated type, and then the variables of this type. You can get the string representation of an enumerated variable with the built-in function `name`(), as shown in Sample 2.46.

Sample 2.46 Enumerated types

```
// Create data type for values 0, 1, 2
typedef enum {INIT, DECODE, IDLE} fsmstate_e;
fsmstate_e pstate, nstate;   // declare typed variables

initial begin
  case (pstate)
    IDLE:    nstate = INIT;    // data assignment
    INIT:    nstate = DECODE;
    default: nstate = IDLE;
  endcase
  $display("Next state is %s",
           nstate.name());     // Display symbolic state name
end
```

Use the suffix "_e" when declaring an enumerated type.

2.12.1 Defining Enumerated Values

The actual values default to integers starting at 0 and then increase. You can choose your own enumerated values. The code in Sample 2.47 uses the default value of 0 for `INIT`, then 2 for `DECODE`, and 3 for `IDLE`.

Sample 2.47 Specifying enumerated values

```
typedef enum {INIT, DECODE=2, IDLE} fsmtype_e;
```

Enumerated constants, such as `INIT` in Sample 2.47, follow the same scoping rules as variables. Consequently, if you use the same name in several enumerated types (such as `INIT` in different state machines), they have to be declared in different scopes such as modules, program blocks, packages, routines, or classes.

An enumerated type is stored as `int` unless you specify otherwise. Be careful when assigning values to enumerated constants, as the default value of an `int` is 0. In Sample 2.48, `position` is initialized to 0, which is not a legal `ordinal_e` variable. This behavior is not a tool bug – it is how the language is specified. So always specify an enumerated constant with the value of 0, as shown in Sample 2.49, just to catch the testbench error.

Sample 2.48 Incorrectly specifying enumerated values

```
typedef enum {FIRST=1, SECOND, THIRD} ordinal_e;
ordinal_e position;
```

Sample 2.49 Correctly specifying enumerated values

```
typedef enum {BAD_O=0, FIRST=1, SECOND, THIRD} ordinal_e;
ordinal_e position;
```

2.12.2 Routines for Enumerated Types

SystemVerilog provides several functions for stepping through enumerated types.

- `first()` returns the first member of the enumeration.
- `last`() returns the last member of the enumeration.
- `next`() returns the next element of the enumeration.
- `next(N)` returns the *N*th next element.
- `prev()` returns the previous element of the enumeration.
- `prev(N)` returns the *N*th previous element.

The functions `next` and `prev` wrap around when they reach the beginning or end of the enumeration.

Note that there is no easy way to write a `for`-loop that steps through all members of an enumerated type if you use an enumerated loop variable. You get the starting member with `first` and the next member with `next`. The problem is creating a comparison for the final iteration through the loop. If you use the test `current!=current.last`, the loop ends before using the last value. If you use `current<=current.last`, you get an infinite loop, as `next` never gives you a value that is greater than the final value. This is similar to trying to make a `for`-loop that steps through the values 0.3 with index declared as `bit [1:0]`. The loop will never exit!

You can use a `do...while` loop to step through all the values, as shown in Sample 2.50.

Sample 2.50 Stepping through all enumerated members

```
typedef enum {RED, BLUE, GREEN} color_e;
color_e color;
color = color.first;
do
  begin
  $display("Color = %0d/%s", color, color.name);
  color = color.next;
  end
while (color != color.first);  // Done at wrap-around
```

2.12.3 Converting to and from Enumerated Types

The default type for an enumerated type is `int` (2-state). You can take the value of an enumerated variable and assign it to a nonenumerated variable such as an `int` with a simple assignment. SystemVerilog does not, however, let you store an integer value in an `enum` without explicitly changing the type. Instead, it requires you to explicitly cast the value to make you realize that you could be writing an out-of-bounds value.

Sample 2.51 Assignments between integers and enumerated types

```
typedef enum {RED, BLUE, GREEN} COLOR_E;
COLOR_E color, c2;
int c;

initial begin
  color = BLUE;          // Set to known good value
  c = color;             // Convert from enum to int (1)
  c++;                   // Increment int (2)
  if (!$cast(color, c)) // Cast int back to enum
    $display("Cast failed for c=%0d", c);
  $display("Color is %0d / %s", color, color.name);
  c++;                   // 3 is out-of-bounds for enum
  c2 = COLOR_E'(c);      // No type checking
  $display("Color is %0d / %s", color, color.name);
  end
```

When called as a function as shown in Sample 2.51, `$cast`() tried to assign the right value to the left variable. If the assignment succeeds, `$cast()` returns 1. If the assignment fails because of an out-of-bounds value, no assignment is made and the function returns 0. If you use `$cast`() as a task and the operation fails, SystemVerilog prints an error.

You can also cast the value using the `type'(val)` as shown in the example, but this does not do any type checking, and so the result may be out-of-bounds. For example,

after the static cast in Sample 2.51, `c2` has an out-of-bounds value. You should avoid this style.

2.13 Constants

There are several types of constants in SystemVerilog. The classic Verilog way to create a constant is with a text macro. On the plus side, macros have global scope and can be used for bit field definitions and type definitions. On the negative side, macros are global, so that they can cause conflicts if you just need a local constant. Lastly, a macro requires the ` character so that it is recognized and expanded by the compiler.

In SystemVerilog, parameters can be declared in a package and so they can be used across multiple modules. This approach can replace many Verilog macros that were just being used as constants. You can use a `typedef` to replace those clunky macros. The next choice is a `parameter`. A Verilog `parameter` was loosely typed and was limited in scope to a single module. Verilog-2001 added typed parameters, but the limited scope kept parameters from being widely used.

SystemVerilog also supports the `const` modifier that allows you to make a variable that can be initialized in the declaration but not written by procedural code.

Sample 2.52 Declaring a `const` variable

```
initial begin
  const byte colon = ":";
  ...
end
```

In Sample 2.52, the value of `colon` is initialized when the `initial` block is entered. In the next chapter, Sample 3.10 shows a `const` routine argument.

2.14 Strings

If you have ever tried to use a Verilog `reg` variable to hold a string of characters, your suffering is over. The SystemVerilog `string` type holds variable-length strings. An individual character is of type `byte`. The elements of a string of length *N* are numbered 0 to *N*-1. Note that, unlike C, there is no null character at the end of a string, and any attempt to use the character "\0" is ignored. Memory for strings is dynamically allocated, so you do not have to worry about running out of space to store the string.

Sample 2.53 shows various string operations. The function `getc(`*N*`)` returns the byte at location *N*, while `toupper` returns an upper-case copy of the string and `tolower` returns a lowercase copy. The curly braces {} are used for concatenation. The task `putc(`*M*`, `*C*`)` writes a byte *C* into a string at location *M*, which must be between 0 and

the length as given by `len`. The `substr(start,end)` function extracts characters from location `start` to `end`.

Sample 2.53 String methods

```
string s;
initial begin
  s = "IEEE ";
  $display(s.getc(0));      // Display: 73 ('I')
  $display(s.tolower());    // Display: ieee

  s.putc(s.len()-1, "-");   // change ' '-> '-'
  s = {s, "P1800"};         // "IEEE-P1800"

  $display(s.substr(2, 5)); // Display: EE-P

  // Create temporary string, note format
  my_log_rtn($psprintf("%s %5d", s, 42));
end

task my_log(string message);
  // Print a message to a log
  $display("@%0t: %s", $time, message);
endtask
```

Note how useful dynamic strings can be. In other languages such as C, you have to keep making temporary strings to hold the result from a function. In Sample 2.53, the `$psprintf()` function is used instead of `$sformat()`, from Verilog-2001. This new function returns a formatted temporary string that, as shown above, can be passed directly to another routine. This saves you from having to declare a temporary string and passing it between the formatting statement and the routine call.

2.15 Expression Width

A prime source for unexpected behavior in Verilog has been the width of expressions. Sample 2.54 adds 1+1 using four different styles. Addition `A` uses two 1-bit variables, and so with this precision 1+1=0. Addition `B` uses 8-bit precision because there is an 8-bit variable on the left side of the assignment. In this case, 1+1=2. Addition `C` uses a dummy constant to force SystemVerilog to use 2-bit precision. Lastly, in addition `D`, the first value is cast to be a 2-bit value with the cast operator, and so 1+1=2.

Sample 2.54 Expression width depends on context

```
bit [7:0] b8;
bit one = 1'b1;                 // Single bit
$displayb(one + one);           // A: 1+1 = 0

b8 = one + one;                 // B: 1+1 = 2
$displayb(b8);

$displayb(one + one + 2'b0);  // C: 1+1 = 2 with constant

$displayb(2'(one) + one);     // D: 1+1 = 2 with cast
```

There are several tricks you can use to avoid this problem. First, avoid situations where the overflow is lost, as in addition `A`. Use a temporary, such as `b8`, with the desired width. Or, you can add another value to force the minimum precision, such as `2'b0`. Lastly, in SystemVerilog, you can cast one of the variables to the desired precision.

2.16 Conclusion

SystemVerilog provides many new data types and structures so that you can create high-level testbenches without having to worry about the bit-level representation. Queues work well for creating scoreboards for which you constantly need to add and remove data. Dynamic arrays allow you to choose the array size at run-time for maximum testbench flexibility. Associative arrays are used for sparse memories and some scoreboards with a single index. Enumerated types make your code easier to read and write by creating groups of named constants.

Don't go off and create a procedural testbench with just these constructs. Explore the OOP capabilities of SystemVerilog in Chap. 5 to learn how to design code at an even higher level of abstraction, thus creating robust and reusable code.

http://www.springer.com/978-0-387-76529-7

SystemVerilog for Verification
A Guide to Learning the Testbench Language Features
Spear, C.
2008, XXXVI, 429 p. 5 illus., Hardcover
ISBN: 978-0-387-76529-7