

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.

System Verilog 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. 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. Lastly, 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. Verilog-2001 allows you to switch between static and dynamic storage, such as stacks.

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` as some people new to Verilog thought that `reg` declared a digital register, and not a signal. 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` or `tri`.

2.1.2 2-State Data Types

SystemVerilog introduces several 2-state data types to improve simulator performance and reduce memory usage, compared to 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 point

```



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, 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 Chapter 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. Additionally, many new features have been added to support these data types.

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, as shown in Sample 2.4.

Sample 2.4 Declaring fixed-size arrays

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

How can you compute the number of bits needed to address a given array size? SystemVerilog has the `$clog2()` function that calculates the ceiling of log base 2, as shown in Sample 2.5.

Sample 2.5 Calculating the address width for a memory

```
parameter int MEM_SIZE = 256;
parameter int ADDR_WIDTH = $clog2(MEM_SIZE); // $clog2(256) = 8
bit [15:0] mem[MEM_SIZE];
bit [ADDR_WIDTH-1:0] addr;           // [7:0]
```

You can create multi-dimensional fixed-size arrays by specifying the dimensions after the variable name. Sample 2.6 creates several two-dimensional arrays of integers, 8 entries by 4, and sets the last entry to 1. Multi-dimensional arrays were introduced in Verilog-2001, but the compact declaration style is new.

Sample 2.6 Declaring and using multi-dimensional arrays

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

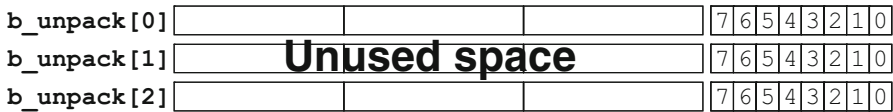
If your code accidentally tries to read from an out-of-bounds address, SystemVerilog will return the default value for the array element type. That just means that an array of 4-state types, such as `logic`, will return X's, whereas an array of 2-state types, such as `int` or `bit`, will return 0. This 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 2.7, 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 Fig. 2.1.

Sample 2.7 Unpacked array declarations

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

**Fig. 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 2.8 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 can replicate values by putting a count before the curly braces.

Sample 2.8 Initializing an array

```
initial begin
    static 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] = '{7,6,5};          // Set just first 3 elements
    ascend = '{4{8}};                 // Four values of 8
    ascend = '{default:42};           // All elements are set to 42
end
```

Notice that in Sample 2.8, the declaration of the array `ascend` includes an initial value. The 2009 LRM states that these variables must be declared either in a static block, or have the `static` keyword. Since this book recommends always declaring your test modules and programs as `automatic`, you need to add the `static` keyword to a declaration plus initialization when it is inside an `initial` block.

A great new feature in the 2009 LRM is printing with the `%p` format specifier. This prints an assignment pattern that is equivalent to the data object's value. You can print any data type in SystemVerilog including arrays, structures, classes, and more. Sample 2.9 shows how to print an array with the `%p` format specifier.

Sample 2.9 Printing with %p print specifier

```

initial begin
    ascend = '{0,1,2,3};
    $display("%p", ascend);      // '{0, 1, 2, 3}
    ascend = '{4{8}};
    $display("%p", ascend);      // '{8, 8, 8, 8}
end

```

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 2.10, 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 2.10 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;          // Initialize src array
    foreach (dst[j])
        dst[j] = src[j] * 2; // Set dst array to 2 * src
end

```

Note that in Sample 2.11, the syntax of the `foreach` loop for multi-dimensional 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 2.11 Initialize and step through a multi-dimensional 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 2.11 is shown in Sample 2.12.

Sample 2.12 Output from printing multi-dimensional 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 2.13 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 2.13 Printing a multi-dimensional 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 2.13 produces the output shown in Sample 2.14.

Sample 2.14 Output from printing multi-dimensional 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 2.15 shows several examples of compares. The `? :` conditional operator is a mini `if-else` statement. In Sample 2.15, it is used to choose between two strings. The final compare uses an array slice, `src[1:4]`, which creates a temporary array with 4 elements.

Sample 2.15 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 (they are equal)
    $display("src[1:4] %s dst[1:4]",
             (src[1:4] == dst[1:4]) ? "==" : "!=");
end
```

A copy between fixed arrays of different sizes causes a compile error. You can not perform aggregate arithmetic such as addition or subtraction on arrays, for example, `a = b + c`. Instead, use `foreach` 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 2.16 prints the first array element (binary 101), its lowest bit (1), and the next two higher bits (binary 10).

Sample 2.16 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. FYI - a double comma in a `$display` statement inserts a space.

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 2.17 shows the variable `bytes`, a packed array of four bytes that are stored in a single 32-bit word as shown in Fig. 2.2.

Sample 2.17 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" of "CA"

```

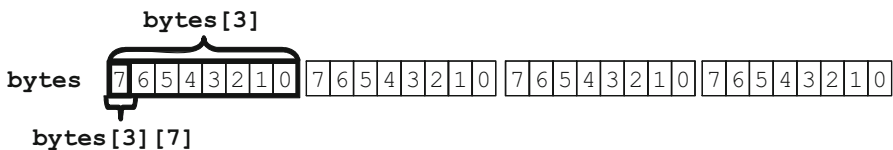


Fig. 2.2 Packed array layout

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. Sample 2.18 shows `barray`, an unpacked array of five packed elements, each four bytes wide, which are stored in memory as shown in Fig. 2.3.

Sample 2.18 Declaration for a mixed packed/unpacked array

```
bit [3:0] [7:0] barray [5];    // 5 elements: packed 4-bytes
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[2];          // Copy packed values
```

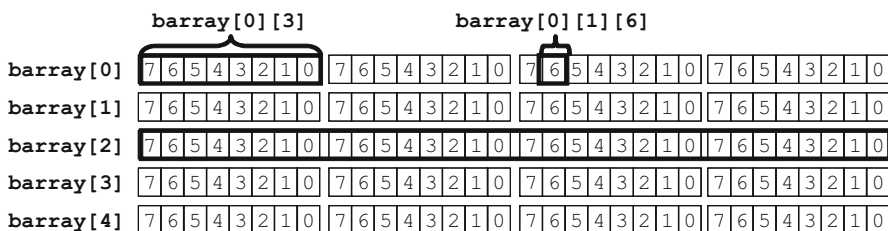


Fig. 2.3 Packed array bit layout

With a single subscript, you get a word of data, `barray[0]`. 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]`. Because one dimension is specified after the name, `barray[5]`, that dimension is unpacked, so you must always give at least one subscript.

The last line of Sample 2.18 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 Fig. 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, 2.4, and 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, so you want to use the `@` operator. This is only legal with scalar values and packed arrays. In Sample 2.18 you can block on the variables `lw` or `barray[0]`, but not the entire array `barray` unless you expand it: `@(barray[0] or barray[1] or barray[2] or barray[3] or barray[4])`.

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 generate a random number of transactions at the start of simulation. If you stored the transactions in a 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 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, give it at run time. The array is initially empty, so you must call the `new[]` constructor to allocate space, passing in the number of entries in the square brackets. If you pass an array name to the `new[]` constructor, the values are copied into the new elements, as shown in Sample 2.19.

Sample 2.19 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 2.19, Line A calls `new[5]` to allocate 5 array elements. The dynamic array `dyn` now holds 5 `int`'s. Line 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 F allocates 20 new elements, and copies the existing 5 elements of `dyn` to the beginning of the array. Then the old 5-element `dyn` 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 a fixed or dynamic 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 2.20 there are 9 mask elements of 8-bits each. You should let SystemVerilog count them, rather than making a fixed-size array and accidentally choosing the wrong array size.

Sample 2.20 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.

You can have multi-dimensional dynamic arrays, so long as you are careful when constructing the sub-arrays. Remember, a multi-dimensional array in SystemVerilog can be thought of as an array of other arrays. First you need to construct the left-most dimension. Then construct the sub-arrays. In Sample 2.21, each sub-array has a different size.

Sample 2.21 Multi-dimensional dynamic array

```
// A dynamic array of dynamic arrays
int d[] [];

initial begin
    // Construct the first or left-most dimension
    d = new[4];

    // Construct the 2nd dimension, each array a different size
    foreach(d[i])
        d[i] = new[i+1];

    // Initialize the elements. d[4][2] = 42;
    foreach(d[i,j])
        d[i][j] = i*10 + j;
end
```

2.4 Queues

SystemVerilog introduces a new data type, the queue, which combines the best of a 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 2.22 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 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 2.22 Queue methods

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

initial begin
    q.insert(1, j);           // {0,1,2,3}  Insert j before ele #1
    q.delete(1);              // {0,2,3}    Delete element #1

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

The LRM does not allow inserting a queue in another queue using the above methods, though some simulators permit this.

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 2.23.

Sample 2.23 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[$];               // j = 5          pop_back
    q = q[0:$-1];           // {6,0,2,3,4}      equivalent

    q = {q, 8};             // {6,0,2,3,4,8} Insert at back

    j = q[0];               // j = 6          pop_front
    q = q[1:$];             // {0,2,3,4,8}      equivalent

    q = {};                 // {}           Delete contents
end

```

The queue elements are stored in contiguous locations, 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, 1000, 4521, 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, as shown in Figure 2.4.

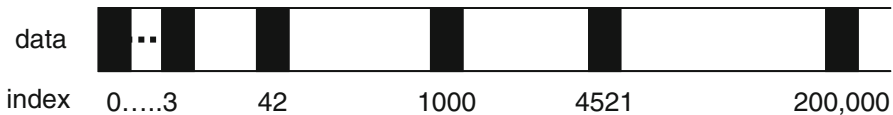


Fig. 2.4 Associative array

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 in square brackets, such as `[int]`, or `[Packet]`. Sample 2.24 shows declaring, initializing, printing, and stepping through an associative array.

Sample 2.24 Declaring, initializing, and using associative arrays

```
byte assoc[byte], idx = 1;
initial begin
    // Initialize widely scattered values
    do begin
        assoc[idx] = idx;
        idx = idx << 1;
    end while (idx != 0);

    // 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)) // Get first index
        do
            $display("assoc[%h]=%h", idx, assoc[idx]);
            while (assoc.next(idx)); // Get next index

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

Sample 2.24 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. You can find the number of elements in an associative array with the `num` or `size` functions.

Associative arrays can also be addressed with a string index, similar to Perl's hash arrays. Sample 2.25 reads a file with strings and builds the associative array

`switch` so you can quickly map from a string value to a number. Strings are explained in more detail in Section 2.15.

If you try to read an element of an associative array that has not been written, SystemVerilog returns the default value for the array base type, such as 0 for 2-state types such as `bit` or `int`, or X for 4-state types such as `logic`. The simulator may also give a warning message. You can use the function `exists()` to check if an element has been allocated, as shown in Sample 2.25.

Sample 2.25 Using an associative array with a string index

```
/* Input file contains:
   42   min_address
  1492 max_address
*/

int switch[string], min_address, max_address, i, file;
initial begin
    string s;
    file = $fopen("switch.txt", "r");
    while (! $feof(file)) begin
        $fscanf(file, "%d %s", i, s);
        switch[s] = i;
    end
    $fclose(file);

    // Get the min address
    // If string not found, use default value of 0 for int array
    min_address = switch["min_address"];

    // Get the max address.
    // Use 1000 if max_address does not exist
    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
```

You can initialize an associative array with the array literal with `index:element` pairs as shown in Sample 2.26. When you print the array with `%p`, the elements are displayed in the same format.

Sample 2.26 Initializing and printing associative arrays

```

int power_of_2[int] = '{0:1, 1:2, 2:4}';
initial begin
    for (int i=3; i<5; i++)
        power_of_2[i] = 1 << i;
    $display("%p", power_of_2); // '{0:1, 1:2, 2:4, 3:8, 4:16}'
end

```

You can also declare an associative array with wildcard subscripts, as in `wild[*]`. However, this style is not recommended as you are allowing an index of almost any data type. One of the many resulting problems is with `foreach`-loops: what type is the variable `j` in `foreach(wild[j])`? Integer, string, bit, or logic?

2.6 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.

2.6.1 Array Reduction Methods

A basic array reduction method takes an array and reduces it to a single value, as shown in Sample 2.27. You can calculate the sum, product, or perform a logical operation on all the elements.

Sample 2.27 Array reduction operations

```

byte b[$] = {2, 3, 4, 5};
int w;
w = b.sum(); // 14 = 2 + 3 + 4 + 5
w = b.product(); // 120 = 2 * 3 * 4 * 5
w = b.and(); // 0000_0000 = 2 & 3 & 4 & 5

```

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

SystemVerilog does not have a method specifically for choosing a random element from an array, 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 one-line way to access the *N*th element. Sample 2.28 shows how to choose a random element from an associative array

indexed by integers by first picking a random number, then stepping through the array. If the array was indexed by a string, just change the type of `idx` to `string`.

Sample 2.28 Picking a random element from an associative array

```
// Declare and initialize associative array with 7 elements
int aa[int] = '{0:1, 5:2, 10:4, 15:8, 20:16, 25:32, 30:64};
int idx, element, count;

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

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

2.6.2 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. At first you may wonder why these return a queue of values. After all, there is only one maximum value in an array. However, SystemVerilog needs a queue for the case when you ask for a value from an empty queue or dynamic array.

Sample 2.29 shows the array locator methods: `min` and `max` functions find the smallest and largest elements in an array. 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 2.29 Array locator methods: `min`, `max`, `unique`

```
int f[6] = '{1,6,2,6,8,6};    // Fixed-size array
int d[]  = '{2,4,6,8,10};    // Dynamic array
int q[$] = {1,3,5,7},        // Queue
      tq[$];                 // Temporary queue for result

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 2.30. These methods return an empty queue if the value you are searching for does not exist in the array.

Sample 2.30 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 2.31.

Sample 2.31 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 2.32 shows various ways to total up a subset of the values in the array. The first line compares the item with 7. This relational returns a 1 (true) or 0 (false) so the calculation is a sum of the array {1,0,1,0,0,0}. The second multiplies the boolean result with the array element being tested. So the total is the sum of {9,0,8,0,0,0}, which is 17. The third calculates the total of elements less than 8. The fourth total is computed using the `? :` conditional operator. The last counts the number of 4's.

Sample 2.32 Array locator methods

```

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

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

```

When you combine an array reduction such as `sum` using the `with` clause, the results may surprise you. In Sample 2.32, the `sum` operator totals the number of

times that the expression is true. For the first statement in Sample 2.32, there are two array elements that are greater than 7 (9 and 8) 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.



Be careful of SystemVerilog's rules for the width of operations. Normally, if you were to add a set of single bit values, SystemVerilog would make the calculations with enough precision not to lose any bits. But the `sum` method uses the width of the array. So, if you add the values of a single-bit array, the result is a single bit, which is probably not what you expected. The solution is to use a `with` expression as shown in Sample 2.33.

Sample 2.33 Creating the sum of an array of single bits

```
bit one[6]; // Array of single bits
int total;

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

    // Compute the single-bit sum
    total = one.sum(); // total = 1 = (0+1+0+1+0+1) & 1

    // Compute with 32-bit signed arithmetic
    total = one.sum() with (int'(item)); // total = 3
end
```

2.6.3 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 2.34. Notice that these change the original array, unlike the array locator methods in Section 2.6.2, which create a queue to hold the results.

Sample 2.34 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, so they work on the entire array. Sample 2.35 shows how to sort a structure by sub-fields. Structures and packed structures are explained in Section 2.9.

Sample 2.35 Sorting an array of structures

```
struct packed { bit [7:0] r, g, b; } c[];
c = '{r:7, g:4, b:9}, {r:3, g:2, b:9}, {r:5, g:2, b:1}};

c.sort with (item.r); // sort using r only
// '{r:3, g:2, b:9}, {r:5, g:2, b:1}, {r:7, g:4, b:9}}

c.sort(x) with ({x.g, x.b}); // Sort g first, then b
// '{r:5, g:2, b:1}, {r:3, g:2, b:9}, {r:7, g:4, b:9}}

```

Only fixed and dynamic arrays, plus queues can be sorted, reversed, or shuffled. Associative arrays can not be reordered.

2.6.4 Building a Scoreboard with Array Locator Methods

The array locator methods can be used to build a scoreboard. Sample 2.36 defines the `Packet` structure, then creates a scoreboard made from a queue of these structures. Section 2.8 describes how to create structures with `typedef`.

Sample 2.36 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 2.36 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 matching the requested one.

A better choice for storing packet information is a class, which is described in Chapter 5. You can read more about structures in Section 2.9.

2.7 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.

2.7.1 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 types match: `int`, `string`, etc. 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.

2.7.2 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 1024 bytes wastes $\frac{3}{4}$ 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.

2.7.3 *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, 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, 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.

2.7.4 *Data Access*

Since SystemVerilog can sort any single-dimension array (fixed-size, dynamic, and associative arrays plus queues), you should pick the array type based on how often the values are added to it. 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, 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.

2.7.5 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: array 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. If you model your transactions as objects, the scoreboard can be a queue of handles. See Chapter 5 for more information of classes.
- *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 plan on accessing every location, try a different approach where you do not need so much live data. Be sure to use 2-state values packed into 32-bits to conserve simulation memory.
- *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.

2.8 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 2.37.

Sample 2.37 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 as shown in Sample 2.38. This book uses the convention that user-defined types use the suffix “_t” except for the basic `uint`.

Sample 2.38 User-defined type in SystemVerilog

```
// New SystemVerilog style
parameter OPSIZE = 8;
typedef logic [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 they can be shared across the design and testbench, as shown in Section 2.10.



One of the most useful types you can create is an unsigned, 2-state, 32-bit integer as shown in Sample 2.39. 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 it can be used anywhere.

Sample 2.39 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 2.40 creates a new type, `fixed_array5_t`, a fixed array with 5 elements. It then declares an array of this type and initializes it.

Sample 2.40 User-defined array type

```
typedef int fixed_array5_t[5];
fixed_array5_t f5;    // Equivalent to "int f5[5]"

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

A good use for a user defined type is an associative array, which must be declared with an index that is a simple type. You could change Sample 2.24 to use 64 bit values by changing the first line as shown in Sample 2.41.

Sample 2.41 User-defined associative array index

```
typedef bit[63:0] bit64_t;
bit64_t assoc[bit64_t], idx = 1;
```

2.9 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, so use a class instead for your testbenches, as shown in Chapter 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.

2.9.1 Creating a Struct and a New Type

You can combine several variables into a structure. Sample 2.42 creates a structure called `pixel` that has three unsigned bytes for red, green, and blue.

Sample 2.42 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 2.43.

Sample 2.43 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.

2.9.2 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 2.44.

Sample 2.44 Initializing a struct

```
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};

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

2.9.3 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 2.45 stores both the unsigned bit vector *b* and the integer *i* in the same location.

Sample 2.45 Using typedef to create a union

```
typedef union { bit [31:0] b; int i; } num_u;
num_u un;
un.i = -1; // set value using signed integer
```

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 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 Section in [2.2.6](#)

2.9.4 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 2.43](#) has three values, 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 with the `packed` keyword, as shown in [Sample 2.46](#).

Sample 2.46 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.

2.9.5 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.

2.10 Packages

At the start of a project, you need to create new types and parameters. For example, if your processor communicates with your company's ABC bus, your testbench needs to define ABC data types, and parameters to specify the bus width and timing. Another project may want to use these types, plus those for the XYZ bus.

You could create separate files for each bus and use the `'include` statement to bring in the files during compilation. But then every name associated with each bus must be unique, even those that are internal variables, never intended to be visible. How can you organize these types to avoid name conflicts?

The SystemVerilog package allows you to share declarations among modules, packages, plus programs and interface, which are described in Chapter 4. Sample 2.47 shows the package for the ABC bus.

Sample 2.47 Package for ABC bus

```
package ABC;
    parameter int abc_data_width = 32;
    typedef logic [abc_data_width-1:0] abc_data_t;
    parameter time timeout = 100ns;
    string message = "ABC done";
endpackage // ABC
```

You import symbols from a package with the `import` statement. The compiler only looks in imported packages when a symbol is not defined in the usual search path. In Sample 2.48, the first `import` statement makes the symbols `abc_data_width`, `abc_data_t`, and `timeout` visible if there is no local variable with the same name. The variable `message` in ABC is hidden by the one in the module.

Sample 2.48 Importing packages

```
module test;
    import ABC::*;                                // Search ABC for symbols

    abc_data_t data;                               // From package ABC
    string message = "Test timed out"; // Hides message in ABC

    initial begin
        #(timeout);                                // From package ABC
        $display("Timeout - %s", message);
        $finish;
    end
endmodule
```

If you really want to see the `message` variable in `ABC`, use `ABC::message`.

You can import specific symbols from a package with the scope operator, `::`. Sample 2.49 imports all the symbols from `ABC`, plus just the `timeout` variable from `XYZ`.

Sample 2.49 Importing selected symbols from a package

```
module test;
  import ABC::*;                // Search ABC for symbols
  import XYZ::timeout;          // Just import timeout
  string message = "Test timed out"; // Hides message in ABC

  initial begin
    #(timeout);                 // From package XYZ
    $display("Timeout - %s", message);
    $finish;
  end
endmodule
```

Packages can only see symbols defined inside themselves, or packages that they import. You can not have hierarchical references to symbols such as signals, routines, or modules from outside the package. Think of a package as being completely standalone, able to plug in where needed, with no outside dependencies.

A package can contain routines, plus classes, as shown in Section 5.4.

2.11 Type Conversion

SystemVerilog has several rules to ensure that expressions are evaluated with little or no loss of accuracy. For example, if you add two 8-bit values, the addition is done with 9-bit precision to avoid overflow. Multiply two 8-bit values, and SystemVerilog calculates a 16-bit result.

The proliferation of data types in SystemVerilog means that you may 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 as described in Section 2.12.

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 2.50. Note that Verilog has always implicitly converted between types such as integer and real, and also between different width vectors.

Sample 2.50 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 2.13.3 for an explanation and example with enumerated types.



Use a static cast when you want SystemVerilog to use a type with more precision, like when using the `sum` method for a single bit array. Use the dynamic cast when converting from a type with a larger number of values than the destination, such as `int` to an enumerated variable.

2.12 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 2.51. You can also give a slice size, 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 2.51 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 2.52 converts between queues, but would also work with dynamic arrays. Array elements are automatically allocated as needed.

Sample 2.52 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 2.53 a structure is streamed into a dynamic array of bytes, then the byte array is streamed back into the structure.

Sample 2.53 Converting between a structure and an 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'hddd_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
```

2.13 Enumerated Types

An enumerated type allows you to create a set of related but unique constants such as states in a state machine or opcodes. In classic Verilog, you had to use text macros. Their global scope is too broad, and their value might not be visible in the debugger. An enumeration creates a strongly typed variable that is limited to a set of specified names. 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. A weaker 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 2.54. This creates an anonymous enumerated type, but it cannot be used for any other variables than the ones in this declaration.

Sample 2.54 A simple enumerated type, not recommended

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

It is recommended to create a named enumerated type so you can declare multiple variables of the same type, especially if these are used as routine arguments or module ports. You first create the enumerated type, and then the variables of this type, as shown in Sample 2.55. You can get the string representation of an enumerated variable with the built-in function `name()`.

Sample 2.55 Enumerated types, recommended style

```
// 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 name.

2.13.1 Defining Enumerated Values

The actual values default to `int` starting at 0 and then increase. You can choose your own enumerated values. The code in Sample 2.56 uses the default value of 0 for `INIT`, then 2 for `DECODE`, and 3 for `IDLE`.

Sample 2.56 Specifying enumerated values

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

Enumerated constants, such as `INIT` in Sample 2.56, 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 2.57, `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 2.58, just to catch the testbench error.

Sample 2.57 Incorrectly specifying enumerated values

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

Sample 2.58 Correctly specifying enumerated values

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

2.13.2 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 clean 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` function and the next member with `next`. A `for` loop ends when the loop variable is outside the defined bounds, but the `next` function always returns a value inside the enumeration. 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 an index declared as `bit [1:0]`. The loop never exits! You can get around this limitation by either using an integer variable in the loop, or incrementing the enumerated variable, but both of these solutions can give illegal values if your enumerated values are not contiguous, such as 1, 2, 3, 5, 8.

You can use a `do...while` loop to step through all the values, checking when the value wraps around, as shown in Sample 2.59.

Sample 2.59 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
```

2.13.3 *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 non-enumerated 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 2.60 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("c2 is %0d / '%s'", c2, c2.name());
end
```

When called as a function as shown in Sample 2.60, `$cast()` tried to assign the right value to the left variable. If the assignment succeeds, `$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, so the result may be out-of-bounds. For example,

after the static cast in Sample 2.60, `c2` has an out-of-bounds value. You should avoid this style of casting with enumerated types.

2.14 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.

A Verilog `parameter` was loosely typed and was limited in scope to a single module. Verilog-2001 added typed parameters, but their limited scope kept parameters from being widely used. In SystemVerilog, parameters can be declared in a package so they can be used across multiple modules. This approach can replace most Verilog macros that were just being used as constants.

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 2.61 Declaring a `const` variable

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

In Sample 2.61, the value of `colon` is initialized at run time, when the `initial` block is entered. In the next chapter, Sample 3.11 shows a `const` routine argument.

2.15 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 2.62 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 , that must be between 0

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

Sample 2.62 String methods

```
string s;

initial begin
    s = "IEEE ";
    $display(s.getc(0));      // Display: 73, ASCII value of 'I'
    $display(s.toLowerCase()); // Display: 'ieee '

    s.putc(s.len()-1, "-");   // change ' '-> '-'
    s = {s, "1800"};         // "IEEE-1800"

    $display(s.substr(2, 5)); // Display: EE-1

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

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

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 2.62, the `$sformatf` function is used instead of `$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. The undocumented function `$psprintf` has the same functionality as `$sformatf`, but is not in the LRM, even though most vendors support this non-standard system function.



There are two ways to compare strings, but they behave differently. The equality operator, `s1==s2`, returns 1 if the strings are identical, and 0 if they are not. The string comparison function, `s1.compare(s2)`, returns 1 if `s1` is greater than `s2`, 0 if they are equal, and -1 if `s1` is less than `s2`. While this matches the ANSI C `strcmp()` behavior, it may not be what you expect.

2.16 Expression Width

A prime source for unexpected behavior in Verilog has been the width of expressions. Sample 2.63 adds `1+1` using four different styles. Addition A uses two 1-bit variables, 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, so $1+1=2$.

Sample 2.63 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.

2.17 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 Chapter 5 to learn how to design code at an even higher level of abstraction, thus creating robust and reusable code.

2.18 Exercises

1. Given the following code sample:

```
byte my_byte;
integer my_integer;
int my_int;
bit [15:0] my_bit;
shortint my_short_int1;
shortint my_short_int2;

my_integer = 32'b000_1111_xxxx_zzzz;
my_int = my_integer;
my_bit = 16'h8000;
my_short_int1 = my_bit;
my_short_int2 = my_short_int1-1;
```

- What is the range of values `my_byte` can take?
 - What is the value of `my_int` in hex?
 - What is the value of `my_bit` in decimal?
 - What is the value of `my_short_int1` in decimal?
 - What is the value of `my_short_int2` in decimal?
2. Given the following code sample:

```
bit [7:0] my_mem[3];
logic [3:0] my_logicmem[4];
logic [3:0] my_logic;

my_mem = '{default:8'hA5};
my_logicmem = '{0,1,2,3};
my_logic = 4'hF;
```

Evaluate the following statements in the given order and give the result for each assignment

- `my_mem[2] = my_logicmem[4];`
- `my_logic = my_logicmem[4];`
- `my_logicmem[3] = my_mem[3];`
- `my_mem[3] = my_logic;`
- `my_logic = my_logicmem[1];`
- `my_logic = my_mem[1];`
- `my_logic = my_logicmem[my_logicmem[41];`

3. Write the SystemVerilog code to:

- a. Declare a 2-state array, `my_array`, that holds four 12-bit values
- b. Initialize `my_array` so that:

```
* my_array[0] = 12'h012
* my_array[1] = 12'h345
* my_array[2] = 12'h678
* my_array[3] = 12'h9AB
```

- c. Traverse `my_array` and print out bits [5:4] of each 12-bit element

```
* With a for loop
* With a foreach loop
```

4. Declare a 5 by 31 multi-dimensional unpacked array, `my_array1`. Each element of the unpacked array holds a 4-state value.

- a. Which of the following assignment statements are legal and not out of bounds?

```
* my_array1[4][30] = 1'b1;
* my_array1[29][4] = 1'b1;
* my_array1[4] = 32'b1;
```

- b. Draw `my_array1` after the legal assignments complete.

5. Declare a 5 by 31 multi-dimensional packed array, `my_array2`. Each element of the packed array holds a 2-state value.

- a. Which of the following assignment statements are legal and not out of bounds?

```
* my_array2[4][30] = 1'b1;
* my_array2[29][4] = 1'b1;
* my_array2[3] = 32'b1;
```

- b. Draw `my_array2` after the assignment statements complete.

6. Given the following code, determine what will be displayed.

```
module test;
    string street[$];

    initial begin
        street = {"Tejon", "Bijou", "Boulder"};
        $display("Street[0] = %s", street[0]);
        street.insert(2, "Platte");
        $display("Street[2] = %s", street[2]);
        street.push_front("St. Vrain");
        $display("Street[2] = %s", street[2]);
        $display("pop_back = %s", street.pop_back());
        $display("street.size = %d", street.size);
    end
endmodule // test
```

7. Write code for the following problems.

- a. Create memory using an associative array for a processor with a word width of 24 bits and an address space of 2^{20} words. Assume the PC starts at address 0 at reset. Program space starts at 0×400 . The ISR is at the maximum address.
- b. Fill the memory with the following instructions:

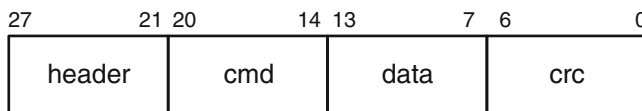
```
* 24'hA50400; // Jump to location 0x400 for the main code
* 24'h123456; // Instruction 1 located at location 0x400
* 24'h789ABC; // Instruction 2 located at location 0x401
* 24'h0F1E2D; // ISR = Return from interrupt
```

- c. Print out the elements and the number of elements in the array.

8. Create the SystemVerilog code for the following requirements

- a. Create a 3-byte queue and initialize it with 2, -1, and 127
- b. Print out the sum of the queue in the decimal radix
- c. Print out the min and max values in the queue
- d. Sort all values in the queue and print out the resulting queue
- e. Print out the index of any negative values in the queue
- f. Print out the positive values in the queue
- g. Reverse sort all values in the queue and print out the resulting queue

9. Define a user defined 7-bit type and encapsulate the fields of the following packet in a structure using your new type. Lastly, assign the header to 7'h5A.



10. Create the SystemVerilog code for the following requirements

- a. Create a user-defined type, nibble, of 4 bits
- b. Create a real variable, r, and initialize it to 4.33
- c. Create a short int variable, i_pack
- d. Create an unpacked array, k, containing 4 elements of your user defined type nibble and initialize it to 4'h0, 4'hF, 4'hE, and 4'hD
- e. Print out k
- f. Stream k into i_pack right to left on a bit basis and print it out
- g. Stream k into i_pack right to left on a nibble basis and print it out
- h. Type convert real r into a nibble, assign it to k[0], and print out k

11. An ALU has the opcodes shown in Table 2.1.

Table 2.1 ALU Opcodes

<i>Opcode</i>	<i>Encoding</i>
Add: $A + B$	2'b00
Sub: $A - B$	2'b01
Bit-wise invert: A	2'b10
Reduction Or: B	2'b11

Write a testbench that performs the following tasks.

- a. Create an enumerated type of the opcodes: `opcode_e`
- b. Create a variable, `opcode`, of type `opcode_e`
- c. Loop through all the values of variable `opcode` every 10ns
- d. Instantiate an ALU with one 2-bit input opcode