Insert as section 4.14 – Before "Arrays assignment"

4.14 Queues

A queue is a variable-size, ordered collection of homogeneous elements. A queue supports constant time access to all its elements as well as constant time insertion and removal at the beginning or the end of the queue. Each element in a queue is identified by an ordinal number that represents its position within the queue, with 0 representing the first, and $ representing the last. A queue is analogous to a one-dimensional unpacked array that grows and shrinks automatically. Thus, like arrays, queues can be manipulated using the indexing, concatenation, slicing operator syntax, and equality operators.

Queues are declared using the same syntax as unpacked arrays, but specifying $ as the array size.

For example:

```
byte q1[$];                      // A queue of bytes
string names[$] = { "Bob" };     // A queue of strings with one element
integer Q[$] = { 3, 2, 7 };      // An initialized queue of integers
```

The empty array literal {} is used to denote an empty queue. If an initial value is not provided in the declaration, the queue variable is initialized to the empty queue.

4.14.1 Queue Operators

Queues support the same operations that can be performed on unpacked arrays, and using the same operators and rules except as defined below:

```
int q[$] = { 2, 4, 8 };
int p[$];
int e, pos;

e = q[0];           // read the first (leftmost) item
e = q[$];           // read the last (rightmost) item
q[0] = e;           // write the first item
p = q;              // read and write entire queue (copy)

q = { q, 6 };       // insert '6' at the end (append 6)
q = { e, q };       // insert 'e' at the beginning (prepend e)

q = q[1:$];         // delete the first (leftmost) item
q = q[0:$-1];       // delete the last (rightmost) item
q = q[1:$-1];       // delete the first and last items

q = {};             // clear the queue (delete all items)

q = { q[0:pos-1], e, q[pos,$] };    // insert 'e' at position pos
q = { q[0:pos], e, q[pos+1,$] };    // insert 'e' after position pos
```

Unlike arrays, the empty queue, {}, is a valid queue and the result of some queue operations. The following rules govern queue operators:

- Q[a : b] yields a queue with b – a +1 elements.
  - If a > b then Q[a:b] yields the empty queue {}.
  - Q[n : n] yields a queue with one item, the one at position n. Thus, Q[n : n] === { Q[n] }.
  - If n lies outside Q's range (n < 0 or n > $) then Q[n:n] yields the empty queue {}.
  - If either a or b are 4-state expressions containing X or Z values, it yields the empty queue {}.

— Q[a : b] where a < 0 is the same as Q[0 : b].

— Q[a : b] where b > $ is the same as Q[a : $].

— An invalid index value (i.e., a 4-state expression with X's or Z's, or a value that lies outside 0..$) shall cause a read operation (e = Q[n]) to return the default initial value for the type of queue item (as described in Table 4-1).

— An invalid index (i.e., a 4-state expression with X's or Z's, or a value that lies outside 0..$+1) shall cause a write operation to be ignored and a run-time warning to be issued. Note that writing to Q[$+1] is legal.

NOTE: Queues and dynamic arrays have the same assignment and argument passing semantics.

4.14.2 Queue methods

In addition to the array operators, queues provide several built-in methods.

4.14.3 size()

The prototype for the size() method is:

```
function int size();
```

The size() method returns the number of items in the queue. If the queue is empty, it returns 0.

```
for ( int j = 0; j < q.size; j++ ) $display( q[j] );
```

4.14.4 insert()

The prototype of the insert() method is:

```
function void insert(int index, queue_type item);
```

The insert() method inserts the given item at the specified index position.
— `Q.insert(i, e)` is equivalent to: `Q = {Q[0:i-1], e, Q[i,$]}`

4.14.5 delete()

The prototype of the delete() method is:

```
function void delete(int index);
```

The insert() method inserts the item at the specified index position.
— `Q.delete(i)` is equivalent to: `Q = {Q[0:i-1], Q[i+1,$]}`

4.14.5 pop_front()

The prototype of the pop_front() method is:

```
function queue_type pop_front();
```

The pop_front () method removes and returns the first element of the queue.
— `e = Q.pop_front()` is equivalent to: `e = Q[0]; Q = Q[1,$]`

4.14.6 pop_back()

The prototype of the pop_back() method is:

```
function queue_type pop_back();
```

The pop_back() method removes and returns the last element of the queue.
— `e = Q.pop_back()` is equivalent to: `e = Q[$]; Q = Q[0,$-1]`

4.14.7 push_front()

The prototype of the push_front() method is:

```
function void push_front(queue_type item);
```

The push_front () method inserts the given element at the front of the queue.
— `Q.push_front(e)` is equivalent to: `Q = {e, Q}`

4.14.8 push_back()

The prototype of the push_back() method is:

```
function void push_back(queue_type item);
```

The push_back() method inserts the given element at the end of the queue.
— `Q.push_back(e)` is equivalent to: `Q = {Q, e}`

4.15 Array Manipulation Methods

SystemVerilog provides several built-in methods to facilitate array searching, ordering, and reduction.

The general syntax to call these array methods is:

array_method_call ::= *// from Annex A.8.2*
array_identifier **.** method_identifier [**(**[iterator_identifier] **)**] [**with (** expression **)**]

The optional **with** clause accepts an expression enclosed in parenthesis. In contrast, the **with** clause used by the randomize method (see Section 12.6) accept a set of constraints enclosed in braces.

4.15.1 Array Locator Methods

Array locator methods operate on any unpacked array, including queues, but their return type is a queue. These locator methods allow searching an array for elements (or their indexes) that satisfy a given expression. Array locator methods traverse the array in an unspecified order. The optional with expression should not include any side effects; if it does, the results are unpredictable.

The prototype of these methods is:

function array_type [$] locator_method (array_type iterator = `item`)**;** // same type as the array
or
function int_or_index_type [$] index_locator_method (array_type iterator = `item`)**;** // index type

Index locator methods return a queue of **int** for all arrays except associative arrays, which return a queue of the same type as the associative index type.

If no elements satisfy the given expression or the array is empty (in the case of a queue or dynamic array) then an empty queue is returned, otherwise these methods return a queue containing all items that satisfy the ex

pression. Index locator methods return a queue with the indexes of all items that satisfy the expression. The optional expression specified by the **with** clause must evaluate to a boolean value.

Locator methods iterate over the array elements, which are then used to evaluate the expression specified by the **with** clause. The *iterator* argument optionally specifies the name of the variable used by the **with** expression to designate the element of the array at each iteration. If it is not specified, the name **`item`** is used by default. The scope for the iterator name is the **with** expression.

The following locator methods are supported (the **with** clause is mandatory) :

— **`find()`** returns all the elements satisfying the given expression

— **`find_index()`** returns the indexes of all the elements satisfying the given expression

— **`find_first()`** returns the first element satisfying the given expression

— **`find_first_index()`** returns the index of the first element satisfying the given expression

— **`find_last()`** returns the last element satisfying the given expression

— **`find_last_index()`** returns the index of the last element satisfying the given expression

For the following locator methods the **with** clause (and its expression) can be omitted if the relational operators (<, >, ==) are defined for the element type of the given array. If a **with** clause is specified, the relational operators (<, >, ==) must be defined for the type of the expression.

— **`min()`** returns the element with the minimum value or whose expression evaluates to a minimum

— **`max()`** returns the element with the maximum value or whose expression evaluates to a maximum

— **`unique()`** returns all elements with unique values or whose expression is unique

— **`unique_index()`** returns the indexes of all elements with unique values or whose expression is unique

Examples:

```
string SA[10], qs[$];
int IA[*], qi[$];

qi = IA.find( x ) with ( x > 5 );                 // Find all items greater than 5

qi = IA.find_index with ( item == 3 );            // Find indexes of all items equal to 3

qs = SA.find_first with ( item == "Bob" );        // Find first item equal to Bob

qs = SA.find_last( y ) with ( y == "Henry" );     // Find last item equal to Henry

qi = SA.find_last_index( s ) with ( s > "Z" );    // Find index of last item greater than Z

qi = IA.min;                                      // Find smallest item

qs = SA.max with ( item.atoi );                   // Find string with largest numerical value

qs = SA.unique;                                   // Find all unique strings elements

qs = SA.unique( s ) with ( s.tolower );           // Find all unique strings in lower-case
```

4.15.2 Array ordering methods

Array ordering methods can reorder the elements of one-dimensional arrays or queues.

The general prototype for the ordering methods is:

function void ordering_method (array_type iterator = item)

The following ordering methods are supported:

— **`reverse()`** reverses all the elements of the array (packed or unpacked). Specifying a **with** clause shall be a compiler error.

— **`sort()`** sorts the unpacked array in ascending order, optionally using the expression in the **with** clause. The **with** clause (and its expression) is optional when the relational operators are defined for the array element type.

— **`rsort()`** sorts the unpacked array in descending order, optionally using the expression in the **with** clause. The **with** clause (and its expression) is optional when the relational operators are defined for the array element type.

— **`shuffle()`** randomizes the order of the elements in the array. Specifying a **with** clause shall be a compiler error.

Examples:

```
string s[] = { "hello", "sad", "world" };
s.reverse;                  // s becomes { "world", "sad", "hello" };

logic [4:1] b = 4'bXZ01;
b.reverse;                  // b becomes 4'b10ZX

int q[$] = { 4, 5, 3, 1 };
q.sort;                     // q becomes { 1, 3, 4, 5 }

struct { byte red, green, blue } c [512];
c.sort with ( item.red );              // sort c using the red field only
c.sort( x ) with ( x.blue << 8 + x.green );     // sort by blue then green
```

4.15.3 Array reduction methods

Array reduction methods can be applied to any unpacked array to reduce the array to a single value. The expression within the optional **with** clause can be used to specify the item to use in the reduction.

The prototype for these methods is:

function expression_or_array_type reduction_method (array_type iterator = item)

The method returns a single value of the same type as the array element type or, if specified, the type of the expression in the **with** clause. The **with** clause can be omitted if the corresponding arithmetic or boolean reduction operation is defined for the array element type. If a **with** clause is specified, the corresponding arithmetic or boolean reduction operation must be defined for the type of the expression.

The following reduction methods are supported:

— **`sum()`** returns the sum of all the array elements, or if a **with** clause is specified, returns the sum of the values yielded by evaluating the expression for each array element.

— **`product()`** returns the product of all the array elements, or if a **with** clause is specified, returns the product of the values yielded by evaluating the expression for each array element.

— **`and()`** returns the bit-wise AND (`&`) of all the array elements, or if a **with** clause is specified, returns the bit-wise AND of the values yielded by evaluating the expression for each array element

— **`or()`** returns the bit-wise OR (`|`) of all the array elements, or if a **with** clause is specified, returns the bit-wise OR of the values yielded by evaluating the expression for each array element

— **`xor()`** returns the logical XOR (`^`) of all the array elements, or if a **with** clause is specified, returns the XOR of the values yielded by evaluating the expression for each array element

Examples:

```
byte b[] = { 1, 2, 3, 4 };
int y;

y = b.sum ;                      // y becomes 10 => 1 + 2 + 3 + 4
y = b.product ;                  // y becomes 24 => 1 * 2 * 3 * 4
y = b.xor with ( item + 4 );     // y becomes 12 => 5 ^ 6 ^ 7 ^ 8
```

4.15.4 Iterator index querying

The expressions used by array manipulation methods sometimes need the actual array indexes at each iteration, not just the array element. The **index** method of an iterator returns the index value of the specified dimension. The prototype of the index method is:

function int_or_index_type **index** (int dimension = 1)

The array dimensions are numbered as defined in Section 22-2: The slowest varying is dimension 1. Successively faster varying dimensions have sequentially higher dimension numbers. If the dimension is not specified, the first dimension is used by default

The return type of the **index** method is an **int** for all array iterator items except associative arrays, which returns an index of the same type as the associative index type.

For example:

```
int  arr[]
int  mem[9:0][9:0], mem2[9:0][9:0];
int  q[$];
...

    // find all items equal to their position (index)
q = arr.find with ( item == item.index );

    // find all items in mem that are greater than corresponding item in mem2
q = mem.find( x ) with ( x > mem2[x.index(1)][x.index(2)] );
```