Testbench and Verification related enhancements to VHDL

By the TBV Team

Team email: vhdl-200x-tbv@eda.org
Team leader contact: jbhasker@esilicon.com

Rev 0.8, May 19, 2004

Index	Issue	Status
TBV1	Boolean, integer, real vector types	Proposal submitted – To be addressed by the DTA team
TBV2	Associative arrays	Proposal reviewed.
TBV2.1	Associative arrays: Based on generic packages	Proposal submitted.
TBV3	Fork join	Proposal reviewed.
TBV4	Queues/FIFOs	Proposal submitted.
TBV4.1	Queues/FIFOs: Based on generic packages	Proposal submitted.
TBV5	Improved formatted TextIO	To be addressed by FT team.
TBV6	Assigned image values for identifier-based enumeration type values	Need proposal
TBV7	Sync and handshaking (event objects)	Proposal submitted
TBV8	Request action / wait for action	Need proposal
TBV9	Expected value detectors	Need proposal
TBV10	Access to coverage data for reactive TB	Need proposal
TBV11	XMR (Hierarchical signal reference)	To be addressed by FT team
TBV12	Sparse arrays	Need proposal
TBV13	Object orientation	Need proposal
TBV14	Random value generation w/ optional and dynamic weighting	Need proposal
TBV15	Random object initialization	Need proposal
TBV16	Random 2 state value resolution in place of X generation	Need proposal
TBV17	Random choice selection w/ optional and dynamic weighting	Need proposal
TBV18	Loading and dumping memories	Need proposal
TBV19	Lists	Proposal submitted.

TBV1:

Summary: Boolean, integer, real vector types
Related issues:
Relevant LRM section:
Current status: Open

Date submitted: March 6, 2003
Author submission: Robert Ingham
Author email: robert.ingham@candc.co.uk

Enhancement

I would like to see the following additional predefined array types:

```
    type    boolean_vector  is  array (natural range <>)    of boolean;
    type    integer_vector  is  array (natural range <>)    of integer;
    type    real_vector     is  array (natural range <>)    of  real;
```

I have found these useful in developing verification models, and the 'boolean_vector' type useful in developing parameterized modules where the
number of I/O sub-modules is passed in as a generic. I note that the 'boolean' type is now well supported for synthesis. It may be that

```
    type    time_vector     is  array (natural range <>)    of  time;
```

is also of some merit.

Analysis & Resolution

TBV2:

Summary: Associative arrays
Related issues:
Relevant LRM section: **3.2.1**
Current status: Proposal submitted

Date submitted: April 17, 2003
Author submission: J Bhasker
Author email: jbhasker@esilicon.com

Enhancement

An associative array is useful for holding sparse data. The indices are not restricted to a contiguous range. It is allocated storage as and when used.

To define an associative array, use the keyword **associative**~~ in an unconstrained array type declaration~~.

type *type_name* **is** ~~**array**~~
associative(~~index_subtype_definition~~assoc_type {, ~~index_subtype_definition~~assoc_type}) **of**
*element*_subtype_indication;

assoc_type is either an array type or a discrete type.
A range constraint can be specified for a discrete type.
An index constraint can be specified for an array type. These constraints only define a bound for the indices.

Some examples:

type myaaT is ~~array~~ associative (INTEGER~~ range <>~~) of BIT;
type COLOR is ~~{~~(Red, Blue, Green, Yellow, Orange~~};~~);
type my2aaT is ~~array~~ associative (COLOR ~~range <>,~~ COLOR~~ range<>~~) of INTEGER;
type A1 is associative (STRING) of STRING(1 to 20);
type A2 is associative (INTEGER range 0 to 20) of BIT_VECTOR(0 to 3);
type A3 is associative (BIT_VECTOR(7 downto 0)) of STRING(1 to 20);
-- Two associative arrays:
variable mem_aa: myaaT:
signal matrix: my2aaT;

An associative array type declaration implicitly defines the following subprograms.

- function *delete* (arg: type_name; i1: index_subtype {; i2: index_subtype}) return boolean;
- function *exists* (arg: type_name; i1:index_subtype {; i2: index_subtype}) return boolean;
- function *size* (arg: type_name) return NATURAL;

- function *first* (*arg*: type_name; variable *i1*:index_subtype {; variable *i2*: index_subtype}) return boolean;
- function *last* (*arg*: type_name; variable *i1*:index_subtype {; variable *i2*: index_subtype}) return boolean;
- function *next* (*arg*: type_name; variable *i1*:index_subtype {; variable *i2*: index_subtype}) return boolean;
- function *prev* (*arg*: type_name; variable *i1*:index_subtype {; variable *i2*: index_subtype}) return boolean;
- function dump (arg: type_name; file: file_type) return Boolean;
- function load (arg: type_name; file: file_type) retirn Boolean;

The following actions can be performed on an associative array object.

1. Insert an element – creates an element in the associative array by assigning a value.

   Mem_aa(2) := '0';
   Matrix (Blue, Red) <= 5;

2. Read an element – reads the specified indexed element from the associative array. If the specified index in not within the index's range, a range constraint error occurs.

   := mem_aa(5) ….
   := matrix (Blue, Red) …
   <= matrix (blue, green) .. – Error as element not yet assigned.

3. Count of elements – Function size returns the total number of elements in the array. IF array is empty, returns 0.

   If (size(mem_aa) > 5) then …

4. Does element exist – Function exists returns true if the element exists, else it returns false.

   If (exists (matrix, blue, red)) then . . .

5. Get first element – Function first returns the index of the first element in the array in the index variable. Function returns false if array is empty.

   Variable var_q: integer;
   If (first (mem_aa, var_q)) then … -- index of first is returned in var_q.

6. Get last element – Function last returns the index of the last element in the array in the index variable. Function returns false if array is empty.

   Variable var_m, var_n: COLOR;
   If (last (matrix, var_m, var_n)) then … -- index of last is returned in var_q.

7. Get next element – Function next returns the index of the next element in the array based on the value of the index variable. On return, the index variable contains the index of the next variable. Function returns false if array is empty or trying to access beyond last.

If (next (mem_aa, var_q)) then …
-- index of next is returned in var_q based on current value of var_q.

8. Get previous element – Function previous returns the index of the previous element in the array based on the index of the index variable. On return, the index variable contains the index of the previous element. Function returns false if array is empty or trying to access beyond first.
   If (prev (matrix, var_m, var_n)) then … -- index of previous is returned in var_m, var_n based on the current value of var_m, var_n.

9. Dump associative array – The functions dumps the contents of the associative array to the specified file. If no file is specified, it dumps the info to STDOUT. The order of elements dumped is from the smallest index to the largest. Functions returns false if some error occurred during the writing of the information.
   The file if it already exists is deleted of its contents before the new info is added. The format of the dump is one entry per line in pairs (index, value) form.
10. Read associative array – An associative array can be loaded values directly from a file. The file contains pairs of values such as (index, value) which are either comma-separated or space-separated or are on separate lines. A line that starts with – is interpreted as a comment and is ignored. The file should not contain any other form of text.
    The function returns false if some error occurs during the read process.

Ordering of elements
The first, next, prev, last functions are based on the premise that all elements of an associative array are ordered from smallest to the largest based on the index values. The ordering on the elements is based on rightmost dimension to the leftmost dimension. And dimension ordering is based on the comparison operators as defined by the language.

For example, the elements of *matrix* are assumed to be ordered :
(red,red), (red, blue), (red, green) . . . (orange, yellow), (orange, orange)

Assignment
Nothing special here. Assignment between associative arrays of the same type are allowed. Associative arrays can also be passed as parameters to subprograms.

Other alternatives
- One other way to implement would be using attributes. However there is no clean way I could think of to implement the traversal operations.
- Instead of having many implicit functions, another alternative would be to have only one implicit function "assoc_op" that takes in an operation argument in addition to the other args.

Analysis & Resolution

TBV2.1:

Summary: Associative arrays: based on generic packages
Related issues: **TBV2**
Relevant LRM section:
Current status: Open

Date submitted: 20-May-04
Author submission: `Peter Ashenden`
Author email: `peter@ashenden.com.au`

Enhancement

Requirements

Proposal TBV2 from the Test Bench and Verification team identifies a requirement for an associative array data type. Such a data type is essentially a partial mapping from an index type to an element type. The requirements included in TBV2 are

- The number of extant mappings (the "size" of the array) can be determined.
- For a given index value, the mapping to an element may or may not exist. Existence can be queried, and a mapping can be added, changed and deleted.
- The index type is ordered.
- Provision be made for iteration over the extant mappings.
- Provision be made to load an associative array from a file and to dump the contents of an associative array to a file.
- Provision be made for assignment of the contents of one associative array to an object of the same associative array type.
- Provision be made for passing associative arrays as parameters to subprograms.

Alternative Implementations of Associative Arrays

The requirements for associative arrays can be met with a generic package that defines an abstract data type (ADT). The ADT defines a type for associative arrays and a number of operations, provided as subprograms, for working with ADT values.

There are numerous tradeoffs that can be made when choosing and implementation for the ADT, including space vs performance tradeoffs. In particular, the choice of data structure used will affect storage space and runtime performance. Since the number of extant mappings in an associative array is not statically known, a dynamically-allocated data structure could be used. In the context of VHDL, however, a dynamically allocated structure cannot be used as the value of a signal. If that were required, a statically allocated data structure could be used, albeit at the cost of imposing a limit on the number of extant mappings in a given associative array value.

A Binary Tree Implementation

One dynamically allocated data structure that can be used is a binary tree, in which each node stores an index value and the element mapped from that element. The left subtree of a node contains all mappings for lesser indices, and the right subtree contains all mappings for greater indices.

A Binary Tree Package Declaration

A package declaration using the binary tree data structure is

```
package associative_arrays is
  generic ( type index_type;
            type element_type;
            function "<"( L, R : index_type ) return boolean is <> );

  type associative_array;

  -- tree_record and structure of associative_array are private
  type tree_record is record
    index : index_type;
    element : element_type;
    left_subtree, right_subtree : associative_array;
  end record tree_record;
  type associative_array is access tree_record;

  function size ( a : in associative_array ) return natural;

  function exists ( a : in associative_array;
                    i : in index_type ) return boolean;

  function get ( a : in associative_array;
                 i : in index_type ) return element_type;

  procedure set ( a : inout associative_array;
                  i : in index_type;
                  e : in element_type );

  procedure delete ( a : inout associative_array;
                     i : in index_type );

  procedure delete_all ( a : inout associative_array );

  generic ( procedure action ( i : in index_type; e : in element_type ) )
  procedure iterate ( a : in associative_array );

  generic ( procedure action ( i : in index_type; e : in element_type ) )
  procedure iterate_reverse ( a : in associative_array );

  procedure copy ( a1 : in associative_array;
                   a2 : out associative_array );

  generic ( procedure read ( file f : std.textio.text;
                             i : out index_type;
                             e : out element_type ) )
  procedure load ( file f : std.textio.text;
                   a : inout associative_array );

  generic ( procedure write ( file f : std.textio.text;
                              i : in index_type;
                              e : in element_type ) )
  procedure dump ( file f : std.textio.text;
                   a : in associative_array );

end package associative_arrays;
```

The package has three generics. The `index_type` and `element_type` generics are used as the types of indices and elements, respectively. Since the index type is ordered, there should be an ordering predicate that can be expressed as a "less than" operator. That operator function is the third generic of the package. The default value is whatever "<" operator for the index type is visible at the point of instantiating the package.

The type associative_array denotes the ADT defined by the package. The fact that it is implemented as a pointer to a binary tree is private to the package, but VHDL does not provide a mechanism for enforcing that.

The remainder of the package declaration is a collection of operations on ADT values. The iteration operations are expressed as generic procedures that have an action procedure to apply to each element. The iterators can be instantiated with different actions procedures to achieve different effects. The copy operation is needed to perform elementwise copy, as the built-in assignment operation would simply provide pointer aliasing. The load and dump procedures are also expressed as generic procedures, with generic subprograms to read an index and element value from a file and to write an index and element to a file, respectively.

A Binary Tree Package Body

A package body for the binary tree data structure is

```
package body associative_arrays is

  function size ( a : in associative_array ) return natural is
  begin
    if a = null then
      return 0;
    else
      return 1 + size(a.left_subtree) + size(a.right_subtree);
    end if;
  end function size;

  function exists ( a : in associative_array;
                    i : in index_type ) return boolean is
  begin
    if a = null then
      return false;
    elsif i = a.index then
      return true;
    elsif i < a.index then
      exists(a.left_subtree, i);
    else
      exists(a.right_subtree, i);
    end if;
  end function exists;

  function get ( a : in associative_array;
                 i : in index_type ) return element_type is
  begin
    if a = null then
      report "associative_arrays.get: no element at given index"
        severity failure;
    elsif i = a.index then
      return a.element;
    elsif i < a.index then
      return get(a.left_subtree, i);
    else
      return get(a.right_subtree, i);
    end if;
  end function get;

  procedure set ( a : inout associative_array;
                  i : in index_type;
                  e : in element_type ) is
  begin
    if a = null then
      a := new tree_record'(i, e, null, null);
    elsif i = a.index then
      a.element := e;
    elsif i < a.index then
      set(a.left_subtree, i, e);
    else
      set(a.right_subtree, i, e);
    end if;
  end procedure set;

  procedure delete ( a : inout associative_array;
                     i : in index_type ) is
```

```
  variable node_to_delete : associative_array;

  procedure remove_least_node ( a : inout associative_array;
                               n : out associative_array ) is
  begin
    if a.left_subtree = null then
      n := a;
      a := a.right_subtree;
    else
      remove_least_node(a.left_subtree, n);
    end if;
  end procedure remove_least_node;

begin
  if a = null then
    return;
  elsif i = a.index then
    node_to_delete := a;
    if a.right_subtree = null then
      a := a.left_subtree;
    else
      remove_least_node(node_to_delete.right_subtree, a);
      a.left_subtree := node_to_delete.left_subtree;
      a.right_subtree := node_to_delete.right_subtree;
    end if;
    deallocate(node_to_delete);
  elsif i < a.index then
    delete(a.left_subtree, i);
  else
    delete(a.right_subtree, i);
  end if;
end procedure delete;

procedure delete_all ( a : inout associative_array ) is
begin
  if a = null then
    return;
  else
    delete_all(a.left_subtree);
    delete_all(a.right_subtree);
    deallocate(a);
  end if;
end procedure delate_all;

generic ( procedure action ( i : in index_type; e : in element_type ) )
procedure iterate ( a : in associative_array ) is

  procedure recursive_iterate ( a : in associative_array ) is
  begin
    if a = null then
      return;
    else
      recursive_iterate(a.left_subtree);
      action(a.index, a.element);
      recursive_iterate(a.right_subtree);
    end if;
  end procedure recursive_iterate;

begin
  recursive_iterate(a);
end procedure iterate;

generic ( procedure action ( i : in index_type; e : in element_type ) )
procedure iterate_reverse ( a : in associative_array );

  procedure recursive_iterate_reverse ( a : in associative_array ) is
  begin
    if a = null then
      return;
    else
      recursive_iterate_reverse(a.right_subtree);
      action(a.index, a.element);
```

```
        recursive_iterate_reverse(a.left_subtree);
      end if;
    end procedure recursive_iterate;

  begin
    recursive_iterate_reverse(a);
  end procedure iterate_reverse;

  procedure copy ( a1 : in associative_array;
                   a2 : out associative_array ) is
  begin
    if a1 = null then
      a2 := null;
    else
      a2 := new tree_record'(a1.index, a1.element, null, null);
      copy(a1.left_subtree, a2.left_subtree);
      copy(a1.right_subtree, a2.right_subtree);
    end if;
  end procedure copy;

  generic ( procedure read ( file f : std.textio.text;
                             i : out index_type;
                             e : out element_type ) )
  procedure load ( file f : std.textio.text;
                   a : inout associative_array ) is
    variable i : index_type;
    variable e : element_type;
  begin
    delete_all(a);
    while not endfile(f) loop
      read(f, i, e);
      set(a, i, e);
    end loop;
  end procedure load;

  generic ( procedure write ( file f : std.textio.text;
                              i : in index_type;
                              e : in element_type ) )
  procedure dump ( file f : std.textio.text;
                   a : in associative_array ) is

    procedure dump_pair ( i : in index_type; e : in element_type ) is
    begin
      write(f, i, e);
    end procedure dump_pair;

    procedure iterate_dump_pair is new iterate
      generic map ( action => dump_pair );

  begin
    iterate_dump_pair(a);
  end procedure dump;

end package associative_arrays;
```

The implementation of the ADT operations is expressed using recursive subprograms. The size operation, for example, tests whether the tree is empty, and returns 0 if so. Otherwise, it returns the 1 (for the root node) plus the sum of the sizes of the two subtrees.

The exists operation tests whether there is a mapping for a given index value. If the tree is empty, there is no mapping, so the function returns false. If the sought index value is the same as the index value stored at the root node, the function returns true. Otherwise, the sought index value must be either less than or greater than that stored at the root node. The function thus recursively tests for existence in the left or right subtree, depending on the relationship between the sought index and the root-node index. It uses the "<" function provided as a generic subprogram to perform the comparison.

The get operation accesses the element mapped from a given index value. It is similar to the exists operation, but instead of returning true when a match is found, it returns the element value at the matching node. If no match is found, the function reports an assertion violation with severity failure.

The set operation updates the mapping from a given index value if such a mapping exists, or adds the mapping otherwise. If the tree is empty, the mapping does not exists, so it is added, creating a singleton tree. Creation of the tree updates the null pointer passed in as the procedure argument. If the root node index is the argument index, the mapping does exist, and is updated. Otherwise, the mapping is sought and updated in either the left or right subtree, depending on the relationship between the sought index and the root-node index.

The delete operation removes a mapping from a given index if such a mapping exists, otherwise it has no effect. The operation is more complex than those previously described, due to the need to rearrange the tree around the deleted node. The procedure first checks for an empty tree. In that case, there is no mapping from the given index, so the procedure simply returns.

In the case of the given index matching the root node index, the root node is the one that must be deleted. The procedure keeps a pointer to that node in the variable node_to_delete. The plan is then to move the node that is next in order of index up to the place of the deleted node. However, if the right subtree of the node to be deleted is empty, there is no such element within the tree rooted at the node to be deleted. So, instead, the entire left subtree, is moved up in place of the deleted node.

If there is a non-empty right subtree for the deleted node, the node with next greatest index value is found by traversing down the leftmost branch of that subtree until a node is found with no left subtree. This is done by the procedure remove_least_node, applied to the right subtree of the deleted node. The procedure returns a pointer to the least node in the parameter n, and updates the pointer to the least node to point to the least node's right subtree. This has the effect of removing the least node from the tree and splicing its subtree onto the least node's parent node.

The actual parameter to the call to remove_last_node is the pointer to the deleted node in the tree. Since the remove_last_node procedure updates that actual parameter with the pointer to the least node, the effect is to move the least node up to where the deleted node was in the tree. The subsequent two assignments then reattach the deleted node's subtrees to the repositioned node. Once that is done, storage for the deleted node is deallocated.

The final part of the delete procedure deals with the case of the deleted index not matching the root index. The mapping for the given index is deleted from the left or right subtree, as appropriate.

The delete_all operation deletes all mappings. For an empty tree, there is nothing to do. Otherwise, the procedure deletes all mappings in the left and right subtrees of the root node, then deallocates storage for the root node itself.

The iterate operation calls an action procedure for each mapping, in order. Note that the iterate procedure itself is a generic procedure, and so cannot be called recursively. Hence, the iterate procedure declares a local recursive procedure, recursive_iterate, to do the work. For an empty tree, there is nothing to do. Otherwise, the procedure recursively applies the iteration to the left subtree (all of whose nodes have lesser index values), then calls the action procedure for the index and element values in the root node, and finally applies the iteration to the right subtree (all of whose nodes have greater index values).

The iterate_reverse operation is similar to the iterate operation, except that it recursively applies the iteration to the right subtree before the root node, and to the left subtree after the root node. The efect is to iterate over mappings in order of greatest to least index value.

The copy operation is a "deep copy," which creates a new tree identical to an original tree. While it would be possible to implement this with an iteration over the original tree, using an action procedure to set a mapping in the new tree, the effect would be to create a "vine" structured tree, with consequent performance degradation. The approach adopted here is simply to replicate the original tree's structure. If the original tree is empty, the new tree is made empty. Otherwise, the root node is replicated, and recursive calls are made to copy the original left and right subtrees to the new left and right subtrees, respectively.

The load operation reads the contents of a text file to load an associative array. The generic procedure read is supplied as the action to read an index value and an element value from a textfile. The load procedure first deletes all extant mappings from the associative array. Then, so long as the end of the file has not been reached, the procedure reads the next index and element values from the file and adds the mapping to the associative array.

The dump operation writes the contents of an associative array to a file. Similarly to the load procedure, there is a generic procedure write to write an index value and an element value to a text file. The dump procedure makes use of the iterate generic procedure to write mappings in order. To do this, it defines an action procedure, dump_pair, to dump an index/element pair to the file, and instantiates the iterate procedure with this action. The dump procedure simply invokes the instantiated iterate procedure.

A Vector Implementation

A statically allocated data structure that can be used is a vector of index/element pairs. Mappings are stored in order of increasing index, with the size of the vector determining the maximum number of mappings that can be stored. In order to be able to track the number of extant mappings, the vector and a count of mappings is kept in a record data structure that represents an associative array.

A Binary Tree Package Declaration

A package declaration using the vector data structure is

```
package associative_arrays is
  generic ( type index_type;
            type element_type;
            max_size : positive;
            function “<“( L, R : index_type ) return boolean is <> );

  -- map_record and structure of associative_array are private
  type map_record is record
    index : index_type;
    element : element_type;
  end record map_record;
  type map_vector is array ( natural range <> ) of map_record;
  type associative_array is record
    size : natural range 0 to max_size;
    maps : map_vector(1 to max_size);
  end record;

  function size ( a : in associative_array ) return natural;

  function exists ( a : in associative_array;
                    i : in index_type ) return boolean;

  function get ( a : in associative_array;
                 i : in index_type ) return element_type;

  procedure set ( a : inout associative_array;
                  i : in index_type;
                  e : in element_type );

  procedure delete ( a : inout associative_array;
                     i : in index_type );

  procedure delete_all ( a : inout associative_array );

  generic ( procedure action ( i : in index_type; e : in element_type ) )
  procedure iterate ( a : in associative_array );

  generic ( procedure action ( i : in index_type; e : in element_type ) )
  procedure iterate_reverse ( a : in associative_array );

  procedure copy ( a1 : in associative_array;
                   a2 : out associative_array );

  generic ( procedure read ( file f : std.textio.text;
                             i : out index_type;
```

```
                                e : out element_type ) )
    procedure load ( file f : std.textio.text;
                     a : inout associative_array );

    generic ( procedure write ( file f : std.textio.text;
                                i : in index_type;
                                e : in element_type ) )
    procedure dump ( file f : std.textio.text;
                     a : in associative_array );

  end package associative_arrays;
```

The package is almost identical to that for the tree data structure. One difference is the extra generic constant, max_size, that specifies the maximum number of mappings that can be stored in a given associative array value. Should different associative arrays be needed with different capacities, the package could instantiated multiple times with different values for the ma x_size generic.

The other difference is the concrete type used to represent an associative array. It is a record containing an element, size, that indicates how many mappings are extant in the array, and an element, maps, that is a vector of index/ element records. The extant mappings are stored in vector elements starting from 1 and proceeding in order of index up to the value of the size element of the associative array record.

The remainder of the package declaration is the same collection of operations on ADT values that were declared in the binary tree version of the package. A model using associative arrays could change implementation simply by changing package instantiations. Application of operations is independent of the underlying implementation, except in contexts where dynamically allocated data structures are not allowed.

A Vector Package Body

A package body for the vector data structure is

```
  package body associative_arrays is

    function size ( a : in associative_array ) return natural is
    begin
      return a.size;
    end function size;

    function exists ( a : in associative_array;
                      i : in index_type ) return boolean is
    begin
      for j in 1 to a.size loop
        if i = a.maps(j).index then
          return true;
        end if;
      end loop;
      return false;
    end function exists;

    function get ( a : in associative_array;
                   i : in index_type ) return element_type is
    begin
      for j in 1 to a.size loop
        if i = a.maps(j).index then
          return a.maps(j).element;
        end if;
      end loop;
      report "associative_arrays.get: no element at given index"
        severity failure;
    end function get;

    procedure set ( a : inout associative_array;
                    i : in index_type;
                    e : in element_type ) is
      variable j : positive range 1 to max_size+1;
    begin
      j := 1;
      while j <= a.size and a.maps(j).index < i loop
```

```
    j := j + 1;
  end loop;
  if j <= a.size and i = a.maps(j).index then
    a.maps(j).element = e;
  elsif a.size = max_size then
    report “associative_arrays.set: no space to insert new element”
      severity failure;
  else
    a.maps(j+1 to a.size+1) := a.maps(j to a.size);
    a.maps(j).index := i;
    a.maps(j).element := e;
    a.size := a.size + 1;
  end if;
end procedure set;

procedure delete ( a : inout associative_array;
                   i : in index_type ) is
  variable j : positive range 1 to max_size+1;
begin
  j := 1;
  while j <= a.size and a.maps(j).index < i loop
    j := j + 1;
  end loop;
  if j > a.size or a.maps(j) /= i then
    return;
  else
    a.maps(j to a.size-1) := a.maps(j+1 to a.size);
    a.size := a.size - 1;
  end if;
end procedure delete;

procedure delete_all ( a : inout associative_array ) is
begin
  a.size := 0;
end procedure delate_all;

generic ( procedure action ( i : in index_type; e : in element_type ) )
procedure iterate ( a : in associative_array ) is
begin
  for j in 1 to a.size loop
    action(a.maps(j).index, a.maps(j).element);
  end loop;
end procedure iterate;

generic ( procedure action ( i : in index_type; e : in element_type ) )
procedure iterate_reverse ( a : in associative_array );
begin
  for j in a.size downto 1 loop
    action(a.maps(j).index, a.maps(j).element);
  end loop;
end procedure iterate_reverse;

procedure copy ( a1 : in associative_array;
                 a2 : out associative_array ) is
begin
  a2 := a1;
end procedure copy;

generic ( procedure read ( file f : std.textio.text;
                           i : out index_type;
                           e : out element_type ) )
procedure load ( file f : std.textio.text;
                 a : inout associative_array ) is
  variable i : index_type;
  variable e : element_type;
begin
  delete_all(a);
  while not endfile(f) loop
    read(f, i, e);
    set(a, i, e);
  end loop;
end procedure load;
```

```
    generic ( procedure write ( file f : std.textio.text;
                                i : in index_type;
                                e : in element_type ) )
    procedure dump ( file f : std.textio.text;
                     a : in associative_array ) is

      procedure dump_pair ( i : in index_type; e : in element_type ) is
      begin
        write(f, i, e);
      end procedure dump_pair;

      procedure iterate_dump_pair is new iterate
        generic map ( action => dump_pair );

    begin
      iterate_dump_pair(a);
    end procedure dump;

  end package associative_arrays;
```

The implementation of the ADT operations is, in most cases, simpler for the vector data structure than for the binary tree, albeit at the cost of storage space consumed by partially populated vectors. The size operation, for example, simply returns the value of the size element of the associative array record.

The exists operation scans the maps vector starting from element 1 up to the last populated map. If it finds map with index value equal to the sought index, the function returns true. If the loop completes without finding such a map, the function returns false.

The get operation is similar to the exists operation, but instead of returning true when a match is found, it returns the element value from the matching map. If no match is found, the function reports an assertion violation with severity failure.

The set operation is somewhat more complex than its binary-tree counterpart. It starts by scanning the maps vector until it reaches the end or finds a map whose index is greater than or equal to the sought index. The position of the scan in the vector is maintained in the variable j. If, on completion of the scan, j refers to an extant map whose index equals the sought map, that map is simply updated. Otherwise, a new map needs to be inserted into the vector. If the associative array is already at full capacity, the procedure reports an error message. If there is room for a further map, maps from j upwards are moved along one position and the map at position j is updated with the new map index and element. Finally, the size of the associative array is incremented.

The delete operation, on the other hand, is a lot simpler than its binary-tree counterpart. The procedure starts by scanning the maps vector until it reaches the end or finds a map whose index is greater than or equal to the index to be deleted. On completion of the scan, if the position variable, j, is past the end of the extant mappings or the map at the scanned position is not equal to the index to be deleted, there is no mapping with the index to be deleted, so the procedure simply returns. Otherwise, the mapping to be deleted is extant and is at the position given by j. Maps after j in the vector are moved down one position, overwriting the map to be deleted. Finally, the size of the associative array is decremented.

The next four operations are all very simple. The delete_all operation deletes all mappings simply by setting the size of the associative array to 0. The iterate and iterate_reverse operations consist of for loops that scan the extant mappings, calling the action procedure for each one. The copy operation uses the built-in variable assignment operation to perform the copy.

The load and dump procedures are exactly the same as their binary-tree counterparts. This is because they are implemented using the other ADT operations defined in the package. They do not rely on the concrete data type used to implement the associative array type.

Examples of Package Usage

Suppose a model requires an associative array of bit-vector test patterns that use time values as the index type. If we are using the binary-tree implementation, the package may be instantiated as shown below.

Since the predefined function "<"operating on time values is visible at the point of instantiation, it is used as the actual function for the formal function "<".

```
type test_pattern is bit_vector(0 to input_size-1);

package test_patterns is new work.associative_arrays
  generic map ( index_type => time,
                element_type => test_pattern );
```

If we were to use the vector implementation and needed to allow for 1000 test pattens, we would instantiate the package as

```
package test_patterns is new work.associative_arrays
  generic map ( index_type => time,
                element_type => test_pattern;
                max_size => 1000 );
```

We can create a variable in which to store test patterns and a procedure to load test patterns from a file as follows:

```
variable patterns_to_apply : test_patterns.associative_array;

procedure read_pattern ( file f : std.textio.text;
                         t : out delay_length;
                         p : test_pattern ) is
  use std.textio.all;
  variable L : line;
begin
  readline(f, L);
  read(L, t);
  read(L, p);
end procedure read_pattern;

procedure load_patterns is new test_patterns.load
  generic map ( action => read_pattern );
```

We can also instantiate the iterate procedure to apply test patterns to a signal as follows:

```
signal test_input_signal : test_pattern;

procedure apply_pattern ( t : in delay_length;
                          p : in test_pattern ) is
begin
  wait for t - now;
  test_input_signal <= p;
end procedure apply_pattern;

procedure apply_patterns is new test_patterns.iterate
  generic map ( action => apply_pattern );
```

A process in a testbench incorporating these definitions can now call the instantiated procedures to load and apply the test patterns:

```
load_patterns(pattern_file, patterns_to_apply);
apply_patterns(patterns_to_apply);
```

Conclusions

In this white paper, we have illustrated how the proposed genericity extensions for VHDL can be used to describe an abstract data type (ADT) for use in a testbench and verification context. Different implementations of the ADT can be developed to meet various requirements on storage, performance and application context.

The associative array feature is one of several requested in the VHDL-200x Test Bench and Verification area. These requests could be met with a library of packages providing abstract data types with alternative implementations. Such a library would be similar in nature to standard libraries provided with programming languages, such as the C++ Standard Template Library, and the collections classes in the Java class library.

Analysis & Resolution

TBV3:

Summary: Fork join
Related issues:
Relevant LRM section:
Current status: Under review

Date submitted: April 22, 2003
Author submission:
Author email:

Enhancement

Allow a forkjoin_statement as yet another sequential statement. Syntax is:

```
      [Fj_label:] Fork
            [ [sblk_lbl1:] ~~SBLOCK ~~DECLARE -- sequential block
                  <declarations> ]
            begin
                  { sequential statements }
            end [~~sblock~~DECLARE] [sblk_lbl1] ;

            [ [sblk_lbl2:] ~~SBLOCK~~DECLARE
                  <declarations> ]
            begin
                  { sequential_statements }
            end [~~sblock~~DECLARE] [sblk_lbl2] ;
            . . .

      join [all | none | first | [ condition_clause] [timeout_clause] ]
      [fj_label];
```

A forkjoin_stmt can contain 0 or more sequential blocks. A label is optional. A forkjoin_stmt causes all enclosing sequential blocks to be executed in parallel. The “kind” of join – “all/none/first/condition_clause/timeout_clause” - determines how execution continues subsequent to a forkjoin_stmt. The kind value of “all” indicates that all sequential blocks must complete execution before exiting the forkjoin_stmt. The value “none” indicates that execution continues immediately and that you do not wait for any sequential blocks to complete. The value “first” indicates that forkjoin_stmt can exit as soon as one sequential block completes execution. The default kind is “all”. The condition clause specifies the condition that must be true before execution continues following the join (the ‘DONE attribute may be used in the condition clause). The timeout_clause specifies the amount of time to wait before execution continues beyond the join.

A forkjoin_stmt is not allowed in a function body.

A sequential block (similar to a block stmt) groups sequential statements. The label, keyword ~~SBLOCK~~ DECLARE and the declarations are optional. At a minimum, you need only the begin and end keywords. A sequential block may appear outside of a forkjoin as a independent sequential statement. A 'DONE attribute is defined for every sequential block and is applicable to a label of the sequential block. The attribute has the value false while the sequential block is being executed. (Semantics of this need to be sync'ed up with the Modeling and Productivity Group).

The outstanding sequential blocks do not terminate when the join clause is activated. Otherwise the value of "none" would be meaningless.

(JR) Calling fork/join with none from within a subprogram has some tricky side effects. I would assume that standard vhdl visibility exists so what happens if you have code like

```
   process
     PROCEDURE P (...
       VARIABLE x : integer;
       PROCEDURE Q ....
       BEGIN
         FORK
  L1:      BEGIN
             x := x +1;
             WAIT FOR 10 ns;
             x : = x +1;
             WAIT FOR 10 ns;
           END;
           BEGIN
             ...
           END;
         JOIN NONE;
       END;
     BEGIN
       Q(..);
     END;
   BEGIN
     P;
     WAIT FOR 5 ns;
   END PROCESS;
```

The fact that Q and P exit before the sequential block L1 terminates implies that the stack for P and Q must remain around. I can see a number of implementation issues associated with that.

Recommend that forked blocks terminate when a procedure exits.

Accessing global variables could cause problems such as which fork got executed first. Some options are not to allow global variables to be assigned within forked process (communication occurs only via signals), or allow global variables but make them as "shared" variables if more than one forked process intends to update its value (the later is recommended).

Variables cannot be waited upon in a condition clause.

Analysis & Resolution

TBV4:

Summary: FIFOs (mail_boxes)
Related issues:
Relevant LRM section:
Current status: Proposal submitted

Date submitted: June 16, 2003
Author submission: J. Bhasker
Author email: jbhasker@esilicon.com

Enhancement

A fifo is a collection of elements of the same type that can only be accessed by either pushing data into the fifo or by popping the data out of the fifo. If no data is available in the fifo during a pop, the enclosing process may optionally suspend until a value is pushed in into the fifo by another process. A fifo may also be used to model a mailbox.

```
type fifo_type is fifo (<> or size) of any_data_type;
```

Examples of fifo types:

```
Type f_a is fifo (<>) of INTEGER;
Type f_b is fifo (20) of BIT_VECTOR(3 downto 0);
```

When <> is specified, the fifo is of arbitrary length (unconstrained fifo). If an explicit size is specified, it specifies the max number of elements that can be held in the fifo (constrained fifo).

Examples of object declarations:

```
Signal A: f_a;
Variable B: f_b := (“001”, “110”, “111”, “111”);  -- Initializes
the fifo.
Signal C: f_a := (33, 54, 66);
```

The following fifo operations can be performed on a fifo-type object:

1. push an element
2. pop an element
3. check if empty
4. check if full

The following attributes can be applied to fifo objects to perform the intended operation.

Fifo_object’push (value) : pushes the value into the fifo_object. Returns false if fifo is full, true otherwise.

Fifo_object’pop(remove_flag, wait_flag): pops the value at the top of the fifo_object. If remove_flag is true, then the item is also deleted from the fifo (the default behavior). If the value is false, then the item is NOT deleted from the fifo.

If wait_flag is true and fifo is empty, enclosing process suspends (this attribute can only be used in contexts where wait statements are allowed).
Fifo_object'size: Returns number of elements in the fifo.

Analysis & Resolution

TBV4.1:

Summary: FIFOs (mail_boxes)**: based on generic packages**
Related issues: **TBV4, Based on type genericity being supported.**
Relevant LRM section:
Current status: Proposal submitted

Date submitted: May 19, 2004
Author submission: J. Bhasker
Author email: jbhasker@esilicon.com

Enhancement

A fifo is a collection of elements of the same type that can only be accessed by either pushing data into the fifo or by popping the data out of the fifo. If no data is available in the fifo during a pop, the enclosing process may optionally suspend until a value is pushed in into the fifo by another process. A fifo may also be used to model a mailbox.

```
-- Want to store values of type pattern in a fifo:
type pattern is std_logic_vector(7 downto 0);

-- Instantiate the generic fifo package:
package pattern_fifo_pkg is new work.generic_fifo_pkg
     generic map (element_type => pattern, max_size => 20);

-- Unconstrained fifo of integers:
package integer_fifo_pkg is new work.generic_fifo_pkg
     generic map (element_type => INTEGER);
```

Examples of object declarations:

```
variable slvfifo_a: pattern_fifo_pkg.fifo;
variable slvfifo_b: pattern_fifo_pkg.fifo;
variable intfifo: integer_fifo_pkg.fifo;
```

The following fifo operations can be performed on a fifo-type object:

5. push an element
6. pop an element
7. check if empty
8. check if full

The following functions can be applied to fifo objects to perform the intended operation.

Push (Fifo_object, value) : pushes the value into the fifo_object. Returns false if fifo is full, true otherwise.

Pop (Fifo_object, remove_flag): pops the value at the top of the fifo_object, which is returned. If remove_flag is true, then the item is also deleted from the fifo (the default behavior). If the value is false, then the item is NOT deleted from the fifo.

Size (Fifo_object): Returns number of elements in the fifo.

Is_Empty (Fifo_object): Checks if fifo is empty.
Is_Full (Fifo_object): Checks if fifo is full.

Implementation view

The previous section describes the users perspective. This section describes how the fifo is implemented – as a generic package.

```
Package generic_fifo_pkg is
      Generic ( type element_type; max_size: positive := integer'high);
  Type fifo;

  -- Circular linked list: added at top, taken out from bottom.
  Type fifo_record is record
    Element: element_type;
    Prev_element,
    Next_element: fifo;
  End record fifo_record;

  Type fifo is access fifo_record;
  Function size (f: in fifo) return natural;
  Function is_empty (f: in fifo) return Boolean;
  Function is_full (f: in fifo) return Boolean;
  Function push (f: in fifo; e: in element_type) return Boolean;
  Function pop (f: in fifo; ) return element_type;
End package generic_fifo_pkg;

Package body generic_fifo_pkg is
  Function size (f: in fifo) return natural is
    Variable count: natural := 0;
    Variable f_ptr: fifo := f;
  Begin
     If f = null then
       Return 0;

     Loop
        Count := count + 1;
      F_ptr := f_ptr->next_element;
      Exit when f_ptr = f;
    End loop;
    Return (count);
  End function size;

  Function is_empty (f: fifo) return Boolean is
  Begin
  Return (f != null);
  end function is_empty;

<more to be added>

End package body generic_fifo_pkg;
```

Analysis & Resolution

TBV7:

Summary: Sync and handshaking (event objects)
Related issues:
Relevant LRM section:
Current status: Proposal submitted

Date submitted: June 18, 2003
Author submission: J. Bhasker
Author email: jbhasker@esilicon.com

Enhancement

For synchronization and hand-shaking between processes, events are required. VHDL already has the notion of events and can wait for events. What is missing is an easy way to create events.

This proposal first declares a signal to be an “event” kind of signal – specified in the declaration of the signal.

```
signal Check: BIT event;
```

Such an event signal cannot be assigned a value or read from – this is the only restriction for the event signal, otherwise it behaves just like any other signal. It can however be used in event lists (to wait for) and the new attribute ‘CAUSE_EVENT can be applied to it. Event signals can be passed as subprogram parameters, where they are treated just like signals.

The ‘CAUSE_EVENT when applied to an event signal causes an event in the next delta if no delay value is specified. If a delay value is specified, then an event occurs after the specified delay.

```
Check’CAUSE_EVENT();
Check’CAUSE_EVENT(10 ns);
```

The ‘CAUSE_EVENT attribute is a procedure – so it can act either as a sequential procedure or a concurrent procedure. When two events are scheduled in multiple processes at exactly the same time, the events cancel each other out (bus resolution).

Processes waiting for a event to occur on Check will get triggered when an event occurs – could be either a wait statement or could be in the sensitivity list of a process.

Analysis & Resolution

TBV12:

Summary: Sparse arrays
Related issues:
Relevant LRM section:
Current status: Open

Date submitted: xxx
Author submission: xxx
Author email: xxx

Enhancement

Associative arrays could be used to model sparse arrays. However, sparse arrays are more specific. They are used specifically for modeling large memories efficiently when only a small percentage of the memory addresses are used in any given simulation.

BTW, it would also be good to define load and dump operations for associative/sparse arrays.

Analysis & Resolution

TBV19:

Summary: Lists
Related issues:
Relevant LRM section:
Current status: Proposal submitted

Date submitted: 06/27/03
Author submission: **Venkataramanan, Srinivasan**
Author email: **srinivasan.venkataramanan@intel.com**

Enhancement

A list is an ordered collection of elements of the same type. A list is always indexed continguosly either in ascending order or in descending order. A list type is declared using the following declaration:

type list_type **is list**(<> or range) **of** another_type;

```
Example:
      Type list_a_type is list (<>) of INTEGER;
      Type list_b_type is list (0 to 5) of BIT_VECTOR(0 to 3);
      Type list_c_type is list (3 downto 0) of STRING(0 to 5);
```

When a range of <> is specified, the list is of an unspecified maximum length (0 to INTEGER'HIGH-1 is default). When a range is specified, the range specifies the MAX number of elements that can be contained in the list. The range is only a constraint. The head of the list is always the leftmost index.

'LEFT and 'RIGHT attributes can be used on a list type to access its bounds – these attributes when used with a type that has range of <> yield 0 and INTEGER'HIGH-1 respectively.

During list operations, the list is always anchored at the leftmost index and grows towards the right. Indices are adjusted to be always consecutive.

```
Examples of object decls:
      Variable usb_fan: list_I_type;  // empty by default.
      Signal usb_data: list_b_type := (“001”, “000”, “000”); // list
       // has three elements indexed from 0 to 2.
      Signal phy_recd: list_b_type := (”001”, ”000”); // will
        // create a 2 element list, indexed from 0 to 1.
```

The following attributes can be applied to objects of a list type to perform list operations.

List_object'DELETE [(index)] – deletes the element at specified index (and all indices adjusted appropriately). If index is not in within list length, returns false, else returns true. If no index specified, all elements in the list are deleted and a true is returned. Deleting an empty list still returns a true.

List_object’INSERT (value [, index]) – inserts a value at the specified index. All other indices/values to the right are adjusted appropriately. If list size becomes greater than specified size, a value of false is returned and the insert does not take place. A true is returned if the insert is successful. If no index specified, insert occurs at end of list (tail). To add to head of list, use list_type’LEFT as index. The value could be a value of the list type or another list type. If value is another list, the new list is inserted at the specified position (always growing to the right).).

What will happen when b_list of size say 3 is added to a_list whose MAX size is set to 10, and a_list already has 8 elements? Should 2 elements of b_list be added or none will be added to a_list? The answer is that none of the elements will be added and a value of false is returned.

List_object’LENGTH – returns the number of elements in the list. 0 if list is empty.

List_object’SORT – sorts the list based on the values in the list in non-decreasing order based on the relational operator defined for the value type. Returns true if successful (if list changed).

List_object’UNIQUE – deletes any duplicate values of the type in the list. (not necessarily numerical values – so bit_vector “000” is different from bit_vector “0000”). Returns true if successful (if list changed).

List_object’REVERSE – reverses the order of elements in the list. Returns true if successful (if list changed).

List_object’EXISTS(value) – Returns TRUE if the value being passed as argument exists within the list, else FALSE.

List_object’INDEX(value) Returns an INTEGER denoting the index of the element which matches with the value argument. Returns –1 if the value doesn’t exist. The search happens from LEFT to RIGHT, search stops at the first match, so if there are multiple elements matching the value being searched, first index will be returned. To search from RIGHT side, do a REVERSE first and then perform a search.

You can assign a list to another list with an assignment statement. This creates a completely new copy of the list.

```
Usb_data <= phy_recd;
```

You can initialize a list using an aggregate constant as shown in the example earlier.

```
Usb_data <= (“100”, “111”);
```

You can also pass lists as arguments to subprograms (similar to arrays).

To write out values in a list, iterate on the list and write out each value.

Note: You can always build your own lists by using the new operator and access pointers.

Open issues

1. How can we perform extraction?
   List_object’EXTRACT(<expr>) – returns a new list which is a sub list of List_object with its elements matching the <expr>. E.g.

   ```
   Type usb_pkts is list (<>) of usb_packet;

   Old_pkts = usb_pkts’EXTRACT(it.log_time > now);
   ```

   But how do we specify sub-field of the elements of any list? An existing HVL supports a key-word as “it” which refers to each element in a list.

2. How do we support KEYed lists? I.e. say a list is created as USB packet.

   ```
   Type list_usb_pkts is list (<>) of usb_packet;
   ```

   usb_packet is a record data type with elements such as header, payload, uid etc. Now when a packet is received from DUT, user wants to search for the pkt with uid as KEY.

   One possible way is to declare the list as KEYed list and allow key based searching.

   ```
   Type list_usb_pkts is list (<>) of usb_packet (key : uid :
   bit_vector (31 downto 0) );

   Rx_pkt_index = usb_pkts’KEY_INDEX (rx_pkt)
   ```

3. How can we do MIN and MAX?

Analysis & Resolution

TBVx:

Summary: xxx
Related issues:
Relevant LRM section:
Current status: Open

Date submitted: xxx
Author submission: xxx
Author email: xxx

Enhancement

I

Analysis & Resolution
