

# Testbench and Verification related enhancements to VHDL

By the TBV Team

Team email: [vhdl-200x-tbv@eda.org](mailto:vhdl-200x-tbv@eda.org)

Team leader contact: [jbhasker@esilicon.com](mailto:jbhasker@esilicon.com)

Rev 0.7, June 27, 2003

Index	Issue	Status
TBV1	Boolean, integer, real vector types	Proposal submitted – To be addressed by the DTA team
TBV2	Associative arrays	Proposal reviewed.
TBV3	Fork join	Proposal reviewed.
TBV4	Queues/FIFOs	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_definitionassoc_type {,
index_subtype_definitionassoc_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 (INTEGERrange ⇔) of BIT;
type COLOR is {(Red, Blue, Green, Yellow, Orange};);
type my2aaT is array-associative (COLOR range ⇔, COLORrange ⇔) 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) return 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 is 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**

-----

## 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 [sblockDECLARE] [sblk_lbl1] ;

  [ [sblk_lbl2:] SBLOCK_DECLARE
    <declarations> ]
  begin
    { sequential statements }
  end [sblockDECLARE] [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**

-----

## 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 contiguously 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_recld;
```

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**

-----