PROPOSAL: Clarify LRM description of streaming operators

Relates to: svdb-1707, svdb-1384
Applies to: IEEE P1800/D3a, 2007

This is the clean copy of clause 11.4.15 as modified by proposal 1707-jb3, with additional edits as required by 1384. Text in blue is provisionally inferred from Mantis 1384.

Version 2, 5-July-2007:
Minor editorial changes

Version 3, 13-July-2007:
Fixed oversights, and changes discussed at SV-EC meeting of 9 July 2007

11.4.15 Streaming operators (pack/unpack)

The bit-stream casting described in 6.24.3 is most useful when the conversion operation can be easily expressed using only a type cast and the specific ordering of the bit stream is not important. Sometimes, however, a stream that matches a particular machine organization is required. The streaming operators perform packing of bit-stream types (see 6.24.3) into a sequence of bits in a user-specified order. When used in the left-hand side, the streaming operators perform the reverse operation, i.e., unpack a stream of bits into one or more variables.

If the data being packed contains any 4-state types, the result of a pack operation is a 4-state stream; otherwise, the result of a pack is a 2-state stream. In the remainder of this clause (11.4.15) the word *bit*, without other qualification, denotes either a 2-state or a 4-state bit as required by this paragraph.

The syntax of the bit-stream concatenation is as follows:

```
streaming_concatenation ::= { stream_operator [ slice_size ] stream_concatenation }   // from A.8.1
stream_operator ::= >> | <<
slice_size ::= simple_type | constant_expression
stream_concatenation ::= { stream_expression { , stream_expression } }
stream_expression ::= expression [ with [ array_range_expression ] ]
array_range_expression ::=
        expression
        | expression : expression
        | expression +: expression
        | expression -: expression
primary ::=                                                                          // from A.8.4
        ...
        | streaming_concatenation
```

Syntax 11-10—Streaming concatenation syntax (excerpt from Annex A)

A *streaming_concatenation* (as specified in the syntax above) shall be used either as the target of an assignment, or as the source of an assignment, or as the operand of a bit-stream cast, or as a

stream_expression in another *streaming_concatenation*. Use of *streaming_concatenation* as the target of an assignment, and the associated unpack operation, is described in sub-clause 11.4.15.3 below.

It shall be an error to use a *streaming_concatenation* as an operand in an expression without first casting it to a bit-stream type. When a *streaming_concatenation* is used as the source of an assignment, the target of that assignment shall be either a data object of bit-stream type or a *streaming_concatenation*.

If the target is a data object of bit-stream type, the stream created by the source *streaming_concatenation* shall be implicitly cast to the type of the target. If this target represents a fixed-size variable and the stream is larger than the variable, an error will be generated. If the target variable is larger than the stream, the stream is left-aligned and zero-filled on the right. If the target represents a dynamically sized variable, such as a queue or dynamic array, the variable is resized to accommodate the entire stream. If, after resizing, the variable is larger than the stream, the stream is left-aligned and zero-filled on the right.

The pack operation performed by a *streaming_concatenation* is here described in two steps for convenience, but the intermediate result between the two steps is never visible and therefore tools are free to implement it in any way that yields the same overall result. First, all integral data in the *stream_expression*s are concatenated into a single stream of bits, similarly to bit-stream casting (as described in clause 6.24.3) but with fewer restrictions. Second, the resulting stream may be re-ordered in a manner specified by the *stream_operator* and *slice_size*. These two steps are described in more detail in sub-clauses 11.4.15.1 and 11.4.15.2 below.

11.4.15.1 Concatenation of *stream_expression*s

Each *stream_expression* within the *stream_concatenation*, starting with the leftmost and proceeding from left to right through the comma-separated list of *stream_expression*s, is converted to a bit-stream and appended to a packed array (stream) of bits, the *generic stream*, by recursively applying the following procedure:

`if` the expression is a *streaming_concatenation* or it is of any bit-stream type

it shall be cast to a packed array of bit using a bit-stream cast, including casting 2-state to 4-state if necessary, and that packed array shall then be appended to the right-hand end of the *generic stream*;

`else if` the expression is an unpacked array (*i.e.* a queue, dynamic array, associative array or fixed-size unpacked array)

this procedure shall be applied in turn to each element of the array. An associative array is processed in index-sorted order. Other unpacked arrays are processed in the order in which they would be traversed by a foreach loop (*see* 12.7.3) having exactly one index variable;

`else if` the expression is of a struct type

this procedure shall be applied in turn to each element of the struct, in declaration order;

`else if` the expression is of an untagged union type

this procedure shall be applied to the first-declared member of the union;

`else if` the expression is a `null` class handle

the expression shall be skipped (not streamed), and a warning may be issued;

`else if` the expression is a non-`null` class handle

this procedure shall be applied in turn to each data member of the referenced object, and not the handle itself. Class members shall be streamed in declaration order. Extended class members shall be streamed after the members of their superclass. It shall be an error if the object has any local or protected data members that would not otherwise be accessible from the scope in which the *streaming_concatenation* occurs. The result of streaming an object hierarchy that contains cycles shall be undefined, and an error may be issued;

`else`

the expression shall be skipped (not streamed), and an error shall be issued.

In the description above, the phrase *skipped (not streamed)* means that the expression in question is not appended to the stream, and operation of the procedure then proceeds with the next item in turn. Implementations are not required to continue the procedure after issuing an error.

11.4.15.2 Re-ordering of the generic stream

The stream resulting from the operation described in sub-clause 11.4.15.1 is then re-ordered by slicing it into blocks and then re-ordering those blocks.

The *slice_size* determines the size of each block, measured in bits. If a *slice_size* is not specified, the default is 1. If specified, it may be a constant integral expression, or a simple type. If a type is used, the block size shall be the number of bits in that type. If a constant integral expression is used, it shall be an error for the value of the expression to be zero or negative.

The *stream_operator* << or >> determines the order in which blocks of data are streamed: >> causes blocks of data to be streamed in left-to-right order, while << causes blocks of data to be streamed in right-to-left order. Left-to-right streaming using >> shall cause the *slice_size* to be ignored, and no re-ordering performed. Right-to-left streaming using << shall reverse the order of blocks in the stream, preserving the order of bits within each block. For right-to-left streaming using <<, the stream is sliced into blocks with the specified number of bits, starting with the right-most bit. If as a result of slicing the last (left-most) block has fewer bits than the block size, the last block has the size of the remaining bits; there is no padding or truncation.

For example:

```
int j = { "A", "B", "C", "D" };
{ >> {j}}                      // generates stream "A" "B" "C" "D"
{ << byte {j}}                 // generates stream "D" "C" "B" "A" (little endian)
{ << 16 {j}}                   // generates stream "C" "D" "A" "B"
{ << { 8'b0011_0101 }}         // generates stream 'b1010_1100 (bit reverse)
{ << 4 { 6'b11_0101 }}         // generates stream 'b0101_11
{ >> 4 { 6'b11_0101 }}         // generates stream 'b1101_01 (same)
{ << 2 { { << { 4'b1101 }} }} // generates stream 'b1110
```

11.4.15.3 Streaming concatenation as an assignment target (unpack)

When a *streaming_concatenation* appears as the target of an assignment, the streaming operators perform the reverse operation; *i.e.* to unpack a stream of bits into one or more variables. The source expression shall be of bit-stream type, or the result of another *streaming_concatenation*. If the source expression contains more bits than are needed, the appropriate number of bits shall be consumed from its left (most significant) end. However, if more bits are needed than are provided by the source expression, an error shall be generated.

Unpacking a 4-state stream into a 2-state target is done by casting to a 2-state type, and vice versa. Null handles are skipped by both the pack and unpack operations; therefore, the unpack operation shall not create class objects. If a particular object hierarchy is to be reconstructed from a stream, the object hierarchy into which the stream is to be unpacked must be created before the streaming operator is applied. Streaming into a target object of class type does not modify any internal properties of the object (such as, for example, its random number generator seed, or its constraint mode).

For example:

```
logic [10:0] up [3:0];
logic [11:1] p1, p2, p3, p4;
bit [96:1] y = {>>{ a, b, c }};  // OK: pack a, b, c
int j = {>>{ a, b, c }};         // error: j is 32 bits < 96 bits
bit [99:0] d = {>>{ a, b, c }};  // OK: b is padded with 4 bits
{>>{ a, b, c }} = 23'b1;         // error: too few bits in stream
{>>{ a, b, c }} = 96'b1;         // OK: unpack a = 0, b = 0, c = 1
{>>{ a, b, c }} = 100'b1;        // OK: unpack as above (4 bits unread)
{ >> {p1, p2, p3, p4}} = up;     // OK: unpack p1 = up[3], p2 = up[2],
                                 //             p3 = up[1], p4 = up[0]
```

11.4.15.4 Streaming dynamically sized data

Remaining text as per P1800/D3a sub-clause 11.4.15.1

Jonathan Bromley, 13 July 2007