Add the text in blue after the following sentence in section 3.14

Structures can be converted to bits preserving the bit pattern, which means they can be converted back to the same value without any loss of information. When unpacked data is converted to the packed representation, the order of the data in the packed representation is such that the first field in the structure occupies the most significant bits. The effect is the same as a concatenation of the data items (struct fields or array elements) in order. The type of the elements in an unpacked structure or array must be valid for a packed representation in order to be cast to any other type, whether packed or unpacked.

An explicit cast between packed types is not required since they are treated as integral values, but a cast can be used by tools to perform stronger type checking.

Add the text in blue to the last sentence in section 4.2

A packed array cannot be directly assigned to an unpacked array without an explicit cast.

Add as section 3.16 – Last section in "Data Types" after "Dynamic casting"

3.16 Bit-stream casting

Type casting may also be applied to unpacked arrays and structs. It is thus possible to convert freely between *bit-stream* types using explicit casts. Types that may be packed into a stream of bits are called bit-stream types. A bit-stream type is a type consisting of the following:

- Any integral, packed, or **string** type
- Unpacked arrays, structures, or classes of the above types
- Dynamically-sized arrays (dynamic, associative, or queues) of any of the above types

This definition is recursive, so that for example a structure containing a queue of **int** is a bit-stream type.

Assuming A is of bit-stream type *source_t* and B is of bit-stream type *dest_t*, it is legal to convert A into B by an explicit cast:

```
B = dest_t'(A);
```

The conversion from A of type *source_t* to B of type *dest_t* proceeds in two steps:

1. Conversion from *source_t* to a generic packed value containing the same number of bits as *source_t*.
   - If *source_t* contains any 4-state data, the entire packed value is 4-state; otherwise, it is 2-state.

2. Conversion from the generic packed value to *dest_t*.
   - If the generic packed value is a 4-state type and parts of *dest_t* designate 2-state types then those parts in *dest_t* are assigned as if cast to a 2-state.

When a dynamic array, queue, or string is converted to the packed representation, the item at index 0 occupies the most significant bits. When an associative array is converted to the packed representation, items are packed in index-sorted order with the first indexed element occupying the most significant bits.

Both *source_t* and *dest_t* may include one or more dynamically sized data in any position (for example, a structure containing a dynamic array followed by a queue of bytes). If the source type, `source_t`, includes dynamically-sized variables, they are all included in the bit-stream. If the destination type, *dest_t*, includes unbounded dynamically-sized types, the conversion process is greedy: compute the size of the *source_t*, sub

tract the size of the fixed-size data items in the destination, and then adjust the size of the first dynamically sized item in the destination to the remaining size; any remaining dynamically-sized items are left empty.

For the purposes of a bit-stream cast, a string is considered a dynamic array of bytes.

Regardless of whether the destination type contains only fixed-size items or dynamically-sized items, data is extracted into the destination in left-to-right order. It is thus legal to fill a dynamically-sized item with data extracted from the middle of the packed representation.

If both *source_t* and *dest_t* are fixed sized unpacked types of different sizes then a cast generates a compile-time error. If *source_t* or *dest_t* contain dynamically-sized types then a difference in their sizes will generate an error either at compile time or run time, as soon as it is possible to determine the size mismatch. For example:

// Illegal conversion from 24-bit struct to **int** (32 bits) - compile time error

```
struct {bit[7:0] a; shortint b;} a;
int b = int'(a);
```

// Illegal conversion from 20-bit struct to **int** (32 bits) - run time error

```
struct {bit a[$]; shortint b;} a = {{1,2,3,4}, 67};
int b = int'(a);
```

// Illegal conversion from **int** (32 bits) to struct dest_t (25 or 33 bits) - compile time error

```
typedef struct {byte a[$]; bit b;} dest_t;
int a;
dest_t b = dest_t'(a);
```

Bit-stream casting can be used to convert between different aggregate types, such as two structure types, or a structure and an array or queue type. This conversion can be useful to model packet data transmission over serial communication streams. For example, the code below uses bit-stream casting to model a control packet transfer over a data stream:

```
typedef struct {
    shortint address;
    reg [3:0] code;
    byte command [2];
} Control;

typedef bit Bits [36:1];

Control p;
Bits stream[$];

p = ...                                 // initialize control packet
stream = {stream, Bits'(p)}             // append packet to unpacked queue of bits

Control q;
q = Control'(stream[0]);                // convert stream back to a Control packet
stream = stream[1:$];                   // remove packet from stream
```

The following example uses bit-stream casting to model a data packet transfer over a byte stream:

```
typedef struct {
    byte length;
    shortint address;
    byte payload[];
    byte chksum;
} Packet;
```

The above type defines a generic data packet in which the size of the payload field is stored in the length field. Below is a function that randomly initializes the packet and computes the checksum.

```
function Packet genPkt();
    Packet  p;

    void'( randomize( p.address, p.length, p.payload )
        with { p.length > 1 && p.payload.size == p.length } );
    p.chksum = p.payload.xor();
    return p;
endfunction
```

The byte stream is modeled using a queue, and a bit-stream cast is used to send the packet over the stream.

```
byte[$] channel;
channel = {channel, (byte[$])'(genPkt())};
```

And the code to receive the packet:

```
Packet p;
int    size;

size = channel[0] + 4;
p = Packet'( channel[0 : size] );       // convert stream to Packet
channel = channel[ size + 1, $ ];       // remove packet data from stream
```

Add as section 7.16 – Last section of "Operators and Expressions"

7.16 Streaming operators (pack / unpack)

The bit-stream casting described in Section 3.16 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 Section 3.16) into a sequence of bits in a user-specified order. When used in the left-hand-side, the streaming operators perform the reverse operation, 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. Unpacking a 4-state stream into a 2-state type is done by a cast to a 2-state variable, and vice-versa.

The syntax of the bit-stream concatenation is:

```
streaming_expression ::= { stream_operator [ slice_size ] stream_concatenation }        From Annex A.8.1

stream_operator ::= >> | <<

slice_size ::= type_identifier | constant_expression

stream_concatenation  ::= { stream_expression { , stream_expression } }

stream_expression ::=
              expression
            | array_identifier [ with [ array_range_expression ] ]

array_range_expression ::=
              expression
            | expression : expression
            | expression +: expression
            | expression -: expression
```

primary ::= *From Annex A.8.4*
…
| streaming_expression

The stream-operator determines the order in which data is streamed: >> causes data to be streamed in left-to-right order, while << causes data to be streamed in right-to-left order. If a slice-size is specified then the data to be streamed is first broken up into slices with the specified number of bits, and then the slices are streamed in the specified order. If a slice-size is not specified, the default is 1 (or **bit**). If, as a result of slicing, the last slice is less than the slice width then no padding is added.

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
```

The streaming operators operate directly on integral types and streams. When applied to unpacked aggregate types, such as unpacked arrays, unpacked structures, or classes, they recursively traverse the data in depth-first order until reaching an integral type. A multi-dimensional packed array is thus treated as a single integral type, whereas an unpacked array of packed items causes each packed item to be streamed individually. The streaming operators can only process bit-stream types; any other types shall generate an error.

The result of the pack operation can be assigned directly to any bit-stream type variable. If the left-hand side represents a fixed-size variable and the stream is larger than the variable, an error will be generated. If the variable is larger than the stream, the stream is left-justified and zero-filled on the right. If the left-hand side represents a dynamic-size 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-justified and zero-filled on the right. The stream is not an integral value; to participate in an expression, a cast is required.

The unpack operation accepts any bit-stream type on the right-hand side, including a stream. The right-hand side data being unpacked is allowed to have more bits than are consumed by the unpack operation. However, if more bits are needed than are provided by the right-hand side expression, an error is generated.

For example:

```
int a, b, c;
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] b = {>>{ 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)
```

7.16.1 Streaming dynamically-sized data

If the unpack operation includes unbounded dynamically-sized types, the process is greedy (as in a cast): the first dynamically-sized item is resized to accept all the available data (excluding subsequent fixed-sized items) in the stream; any remaining dynamically-sized items are left empty. This mechanism is sufficient to unpack a packet-sized stream that contains only one dynamically-sized data item. However, when the stream contains multiple variable-sized data packets, or each data packet contains more than one variable-sized data item, or the size of the data to be unpacked is stored in the middle of the stream, this mechanism can become cumber

some and error-prone. To overcome these problems, the unpack operation allows a **with** expression to explicitly specify the extent of a variable-sized field within the unpack operation.

The syntax of the **with** expression is:

```
stream_expression ::=
            expression
          | array_identifier [ with [ array_range_expression ] ]

array_range_expression ::=
            expression
          | expression : expression
          | expression +: expression
          | expression -: expression
```

The array range expression within the **with** construct must be of integral type and evaluate to values that lie within the bounds of a fixed-size array, or to positive values for dynamic arrays or queues. The array identifier may be any one-dimensional unpacked array (including a queue). The expression within the **with** is evaluated immediately before its corresponding array is streamed (i.e., packed or unpacked). Thus, the expression may refer to data that is unpacked by the same operator but before the array. If the expression refers to variables that are unpacked after the corresponding array (to the right of the array) then the expression is evaluated using the previous values of the variables.

When used within the context of an unpack operation and the array-identifier designates a variable-sized array, the array shall be resized to accommodate the range expression. If the array-identifier designates a fixed-sized array and the range expression evaluates to a range outside the extent of the array, only the range that lies within the array is unpacked and an error is generated. If the range expression evaluates to a range smaller than the extent of the array (fixed or variable sized), only the specified items are unpacked into the designated array locations; the remainder of the array is unmodified.

When used within the context of a pack (on the right-hand side), it behaves the same as an array slice: The specified number of array items are packed into the stream. If the range expression evaluates to a range smaller than the extent of the array, only the specified array items are streamed. If the range expression evaluates to a range greater than the extent of the array size, the entire array is streamed and the remaining items are generated using the default value (as described in Table 5-1) for the given array.

For example, the code below uses streaming operators to model a packet transfer over a byte stream that uses little-endian encoding:

```
byte stream[$];          // byte stream

class Packet
    rand int header;
    rand int len;
    rand byte payload[];
    int crc;

    constraint G { len > 1; payload.size == len ; }

    function void post_randomize; crc = payload.sum; endfunction
endclass

...
send: begin      // Create random packer and transmit
    byte q[$];
    Packet p = new;
    void'(p.randomize());
    q = {<< byte{p.header, p.len, p.payload, p.crc}};   // pack
```

```
        stream = {stream, q};                                   // append to stream
    end

    ...
    receive: begin      // Receive packet, unpcak, and remove
        byte q[$];
        Packet p = new;
        {<< byte{ p.header, p.len, p.payload with [0 +: p.len], p.crc }} = stream;
        stream = stream[ $bits(p) / 8 : $ ];                    // remove packet
    end
```

In the example above, the pack operation could have been written as either:

```
    q = {<<byte{p.header, p.len, p.payload with [0 +: p.len], p.crc}};
```

or

```
    q = {<<byte{p.header, p.len, p.payload with [0 : p.len-1], p.crc}};
```

or

```
    q = {<<byte{p}};
```

The result in this case would be the same since `p.len` is the size of `p.payload` as specified by the constraint.