Proposal for Unpacked Array and Structure Expressions

Section numbers refer to SystemVerilog 3.0

Add to end of section 2.8 Structure literals

Structure literals can also use member name and value, or data type and default value (*see Expressions*):

```
c = {a:0, b:0.0}; // member name and value for that member
c = {default:0}; // all elements of structure c are set to 0
d = ab'{int:1, shortreal:1.0}; // data type and default value for all members of that type
```

To initialize an array of structures the nested braces should reflect the array and the structure, for example:

```
ab abarr[1:0] = {{1, 1.0}, {2, 2.0}};
```

Add after section 7.9 Concatenation

7.10 Unpacked Array Expressions

Braces are also used for expressions to assign to unpacked arrays. Unlike in C, the expressions must match element for element and the braces must match the array dimensions. The type of each element is matched against the type of the initializer expression according to the same rules as for a scalar. This means that the following do not give size warnings, unlike the similar assignments above:

```
bit unpackedbits [1:0] = {1,1}; // no size warning as bit can be set to 1
int unpackedints [1:0] = {1'b1, 1'b1}; // no size warning as int can be set to 1'b1
```

The syntax of multiple concatenations can be used for unpacked array expressions as well.

. unpackedbits = {2{y}} for {y, y}.

SystemVerilog determines the context of the braces by looking at the left hand side of an assignment. If the left hand side is an unpacked array, the braces represent an unpacked array literal or expression. Outside the context of an assignment on the right hand side, an explicit cast must be used with the braces to distinguish it from a concatenation.

It can sometimes be useful to set array elements to a value without having to keep track of how many members there are. This can be done with the **default** keyword:

```
initial unpackedints = {default:2};  // sets elements to 2
```

For more arrays of structures, it is useful to specify one or more matching types, as illustrated under structure expressions below.

```
Struct {int a; time b;} abkey[1:0];
abkey = {{a:1, b:2ns}, {int:5, time:$time}};
```

The rules for unpacked array matching are as follows:

For type:value: if the element or subarray type of the unpacked array exactly matches this type, then each element or subarray will be set to the value. The value must be castable to the array element or subarray type. Otherwise, if the unpacked array is

multidimensional, then there is a recursive descent into each subarray of the array using the rules in this section and the type and default specifiers. Otherwise, if the unpacked array is an array of structures, there is a recursive descent into each element of the array using the rules for structure expressions and the type and default specifiers.

For **default**:value this specifies the default value to use for each element of an unpacked array that has not been covered by the earlier rules in this section. The value must be castable to the array element type.

7.11 Structure Expressions

A structure expression (packed or unpacked) can be built from member expressions using braces and commas, with the members in declaration order. It can also be built with the names of the members

```
module mod1;

  typedef struct {
    int x;
    int y;
  } st;

  st s1;
  int k = 1;
  initial begin
    #1 s1 = {1, 2+k}; // by position
    #1 $display( s1.x, s1.y);
    #1 s1 = {x:2, y:3+k); // by name
    #1 $display( s1);
    #1 $finish;
  end
endmodule
```

It can sometimes be useful to set structure members to a value without having to keep track of how many members there are, or what the names are. This can be done with the **default** keyword:

```
initial s1 = {default:2};  // sets x and y to 2
```

The {member:value} or {data type:default value} syntax can also be used:

```
ab abkey[1:0] = {{a:1, b:1.0}, {int:2, shortreal:2.0}};
```

Note that the **default** keyword applies to members in nested structures or elements in unpacked arrays in structures. In fact it descends the nesting to a built-in type or a packed array of them.

```
struct {
  int A;
  struct {
    int B, C;
  }BC1, BC2;
```

```
}
ABC = {A:1, BC1:{B:2, C:3}, BC2:{B:4,C:5}};
DEF = {default:10};
```

To deal with the problem of members of different types, a type can be used as the key. This overrides the default for members of that type:

```
typedef struct {
   logic[7:0] a;
   bit b;
   bit [31:0] c;
   string s;
} sa;
sa s2;
initial s2 = {bit[31:0]:1, default:0, string:""}; // set all to 0 except the array of bits to 1 and string to ""
```

Similarly an individual member can be set to override the general default and the type default:

```
initial #10 s1 = {default:'1, s = ""}; // set all to 1 except s to ""
```

SystemVerilog determines the context of the braces by looking at the left hand side of an assignment. If the left hand side is an unpacked structure, the braces represent an unpacked structure literal or expression. Outside the context of an assignment on the right hand side, an explicit cast must be used with the braces to distinguish it from a concatenation.

The matching rules are as follows:

A member:value: specifies an explicit value for a named member of the structure. The named member must be at the top level of the structure - a member with the same name in some level of substructure will NOT be set. The value must be castable to the member type, otherwise an error is generated.

The type:value specifies an explicit value for a field in the structure which exactly matches the type and has not been set by a fieldname specifier above. If the same key type is mentioned more than once, the LAST value is used.

The **default**:value applies to members that are not matched by either member name or type and are not either structures or unpacked arrays. The value must be castable to the member type, otherwise an error is generated. For unmatched structure members, the type and default specifiers are applied recursively according to the rules in this section to each member of the substructure. For unmatched unpacked array members, the type and default specifiers are applied to the array according to the rules for unpacked arrays.

Every member must be covered by one of these rules.

7.12 Aggregate Expressions

Unpacked structure and array variables, literals, and expressions may all be used as aggregate expressions. A multi-element slice of an unpacked array is may also be used as an aggregate expression.

Aggregate expressions may be copied in an assignment, through a port, or as an argument to a task or function. Aggregate expressions may also be compared with equality or inequality operators. To be copied or compared, the type of an aggregate expression must be equivalent.

Unpacked structures types are equivalent by the hierarchical name of its type alone. This means in order to have two equivalent unpacked structures in two different scopes, the type must be defined in one of the following ways:

- Defined in a higher-level scope common to both expressions.
- Passed through type parameter.
- Imported by hierarchical reference.

Unpacked arrays types are equivalent by having equivalent element types and identical shape. Shape is defined as the number of dimensions and the number of elements in each dimension, not the actual range of the dimension.

7.13 Conditional Operator

conditional_expression ::= (From Annex A - A.8.3)
expression1 ? { attribute_instance } expression2 : expression3

As defined in Verilog, if expression1 is true, the operator returns expression2, if false, it returns expression3. If expression1 evaluates to ambiguous value (x or z), then both expression2 and expression3 shall be evaluated and their results shall be combined, bit by bit.
SystemVerilog extends the conditional operator to non bit-level types and aggregate expressions using the following rules:

- If both expression2 and expression3 are bit-level types, or a packed aggregate of bit type, the operation proceeds as defined.
- If expression2 or expression3 is a bit-level type and the opposing expression can be implicitly cast to a bit-level type, the cast is made and proceeds as defined.
- For all other cases, the type of expression2 and expression3 must be equivalent.

If expression1 evaluates to ambiguous value, then both expression2 and expression3 shall be evaluated and their results shall be combined, element-by-element. If the elements match, the element is returned. If they do not match, then the default-uninitialized value for that element's type shall be returned.