Data Type Expressions in SystemVerilog

Background

There is a need for expressions that compute equality/inequality of data types. At the minimum these expressions are useful during elaboration, to enable generate statements that can handle the flexibility of the SystemVerilog data type system.

```
 module multitypemodule #(parameter type thetype=logic)
    (input  thetype datai,
     output thetype datao);
     generate
       if (thetype == typedef1)
         always_comb begin
           datao.field1 = datai.field2;
           datao.field2 = datai.field1;
        end
       else if (thetype == typedef2)
         always_comb begin
           datao[1:0] = datai[3:2];
           datao[3:2] = datai[1:0];
         end
       else
         always_comb
           datao = datai;
    endgenerate
endmodule
```

This capability could be extended to run-time but this feature would be more applicable to general programming or test bench code. Justification for extending this capability to run-time is welcomed but left for interested parties. The rest of this document specifies the capabilities for elaboration only.

Accompanying the need to compare data types is the need for a function that reports the data type of an expression. For example:

```
multitypemodule #($type(a)) imultitypemodule (a,b);
```

The addition of $type makes macros more convenient:

```
`define mymacro(a,b,id) multitypemodule #($type(a)) id (a,b);
```

The $type system function is justified just as the $bits, $left, etc. system function(s) are justified. The $type system function would extend introspective capabilities to SystemVerilog data types just as the other system functions add introspective capabilities to Verilog-2001 and previous data types. If $bits, etc. become methods, $type should follow accordingly.

Proposal

The following is proposed language to add data type equality/inequality expressions and the $type system function to SystemVerilog.

3.? Data Type Equality

Two date types will be defined as equal data types using the following inductive definition. If the two data types are not defined equal using the following definition then they are defined to be not equal.

1. A data type, type1 is equal to itself if type1 is of one of the following types
   - Integer vector type
   - Integer atom type
   - Type declaration identifier
   - Non integer type
   - string
   - event
   - chandle
   - class scope type identifier

2. type1 is equal to type2, if type1 is equal to type3 and type2 is equal to type3

3. type1 is equal to type2, if type2 is equal to type1

4. 'type1 signed' is equal to type1, if the by default type1 is signed (i.e. *byte signed* is equal to *byte*)
5. 'type1 unsigned' is equal to type1, if the by default type1 is unsigned (i.e. *logic unsigned* is equal to *logic*)
6. type1 signed is equal to type1 signed
7. type1 unsigned is equal to type1 unsigned
8. If type1 and type2 are equal then the packed arrays type1[M1:L1] and type1[M2:L2] if the evaluated values of M1 and L1 are respectively equal to M2 and L2 (i.e. *logic[10:0]* is equal to *logic [5+5:0]*)
9. type1 is equal to type2, if type1 is defined to be type2 using a **typedef** definition.
10. type1 is equal to type2, if type1 is a type parameter assigned to be type2.

Note that according to the previous definition:

- Two struct or union types are not equal if they have the same fields.
- Two enumerated types are not equal even if they are both of the same data type and both have the same enumeration values.
- An enumerated data type is not equal to the enumeration's data type (i.e. *enum int {...}* is not equal to *int*)
- **logic** and **reg** types are not equal
- logic[0:0] and logic are not equal
- reg[31:0] and integer are not equal

7.? Data Type Expressions

SystemVerilog includes the ability to compute the equality/inequality of data types using data type expressions. Data type expressions enable the comparison of two data types using the operators equality operator '==' and inequality operator '!='. Data type expressions are evaluated during elaboration.

Example data type expressions include:

```
typedef logic [1:0] typedef1;
      typedef logic [1:0] typedef2;
      typedef logic [3:0] typedef3;

      typedef1 == typedef2
      typedef1 != typedef2

      typedef struct {
        logic a;
      } stype1;
      typedef struct {
        logic a;
      } stype2;
      typedef stype2 stype3;

      stype1 != stype2
      stype2 == stype3
      logic [1:0] == typedef1
```

Data type expressions are generally applicable to generate statements:

```
      generate
        if (thetype == typedef1)
          always_comb begin
            datao.field1 = datai.field2;
            datao.field2 = datai.field1;
          end
        else if (thetype == typedef2)
          always_comb begin
             datao[1:0] = datai[3:2];
             datao[3:2] = datai[1:0];
          end
        else
          always_comb
            datao = datai;
      endgenerate
```

22.? Data Type System Function

type_function ::=   // not in Annex A
                              $type (expression)

The $type system function returns a data type corresponding to the input expression. The data type returned by the $type system function can be used like any other data type, for example:

```
parameter type paramtype = $type(structvar);

$type(somevar) var_somevartype;
```

The data type expression should be compatible with system tasks that make use of the %s format specification.

```
logic [1:0] mda [1:0][2:0];
...
$display("type mda = %s",$type(mda));
```

The value for other format types is undefined.

Annex A

Under A.8.3 Expressions

REPLACE
constant_expression ::=
    constant_primary
  | unary_operator { attribute_instance } constant_primary
  | constant_expression binary_operator { attribute_instance }   constant_expression
    | constant_expression **?** { attribute_instance } constant_expression **:**
          constant_expression
  | string_literal

WITH
constant_expression ::=
    constant_primary
  | unary_operator { attribute_instance } constant_primary
  | constant_expression binary_operator { attribute_instance }   constant_expression
    | constant_expression **?** { attribute_instance } constant_expression **:**
          constant_expression
  | string_literal
  | constant_data_type_expression

ADD
constant_data_type_expression ::=
  data_type data_type_operator data_type

Under A.8.6 Operators
ADD
data_type_operator ::=
  == | **!=**