Subject: PASSED - Enumerations Proposal

For your information, attached is the amended Enumerations Proposal that passed in the April 15th SystemVerilog Conference call.

Cleanup and clarification wording is still permitted but not anticipated at this time.

Please note that two more clarification examples (shown in blue and added as the third and fourth examples in section 3.6) were added after the vote. Please make sure the examples and descriptions are acceptable by all.

To simplify the incorporation of the proposals, the entire section 3.6 should be replaced with this new section 3.6 with all proposals incorporated and shown in their correct locations within the section.

PASSED #1 - Replace section 3.6 of draft 5 with the enumerations proposal

Regards - Cliff

PROPOSAL #1: Replace section 3.6 of draft 5 with the enumerations proposal as shown:

Proposed change to BNF section A.2.2.1 - data_type

```
data_type ::=
      integer_vector_type [ signing ] { packed_dimension } [ range ]
    | integer_atom_type [ signing ] { packed_dimension }
    | type_declaration_identifier
    | non_integer_type
    | struct { { struct_union_member } }
    | union { { struct_union_member } }
    | enum [ [ integer_type ] [ signing ] { packed_dimension } ]
        { enum_identifier [ = constant_expression ]
        { , enum_identifier [= constant_expression ] } }
    | void
```

3.6 Enumerations

Enumerated data types provide the capability to abstractly declare strongly typed variables without either a data type or data value(s) and later add the required data type and value(s) for designs that require more definition. Enumerated data types also can be easily referenced or displayed using the enumerated names as opposed to the enumerated values.

In the absence of a data type declaration~~ and if no values are assigned~~, the default data type shall be **`int`**. ~~If only binary values are assigned to the **`enum`** names, the default data type shall be **`int`**, If no data type is assigned and the enumerated names have assigned values that include **`x`**'s and **`z`**'s, the default data type shall be **`logic`**. This means that an enumeration that defaults to a data type of **`int`** because the **`enum`** names were either unassigned or were originally assigned binary values will default to a data type of **`logic`**, after the design is recompiled, if **`x`** assignments or **`z`** assignments are added to the enumeration when the design file is modified.~~ Any other data type used with enumerated types requires an explicit data type declaration.

An enumerated type has one of a set of named values. In the following example, "light1" and "light2" are defined to be variables of the anonymous (unnamed) enumerated **int** type that includes the three members: "red", "yellow" and "green."

```
enum {red, yellow, green} light1, light2; // ‘anonymous’ int type
```

An **enum** name with **x** or **z** assignments assigned to an **enum** with no explicit data type declaration shall be a syntax error.
~~An unassigned **enum** name that follows an **enum** name with **x** or **z** assignments shall be a syntax error.~~

```
// Syntax error: IDLE=2'b00, XX=2'bx <ERROR>, S1=2'b01~~??~~, S2=2'b10~~??~~
enum {IDLE, XX='x, S1=2'b01, S2=2'b10} state, next;
```

<<CLEANUP - Cliff added two more examples here>>
An **enum** of a 4-state type, such as **integer**, that includes one or more names with **x** or **z** assignments shall be permitted.

```
// Correct: IDLE=2'b00, XX=2'bx, S1=2'b01, S2=2'b10
enum integer {IDLE, XX='x, S1=2'b01, S2=2'b10} state, next;
```

An unassigned **enum** name that follows and **enum** name with **x** or **z** assignments shall be a syntax error.

```
// Syntax error: IDLE=2'b00, XX=2'bx, S1=??, S2=??
enum integer {IDLE, XX='x, S1, S2} state, next;
```

The values can be cast to integer types, and increment from an initial value of 0. This can be over-ridden.

```
enum {bronze=3, silver, gold} medal; // silver=4, gold=5
```

The values can be set for some of the names and not set for other names. A name without a value is automatically assigned an increment of the value of the previous name.

```
// c is automatically assigned the increment-value of 8
enum {a=3, b=7, c}; alphabet;
```

If an automatically incremented value is assigned elsewhere in the same enumeration, this shall be a syntax error.

```
// Syntax error: c and d are both assigned 8
enum {a=0, b=7, c, d=8}; alphabet;
```

If the first name is not assigned a value, it is given the initial value of 0.

```
// a=0, b=7, c=8
enum {a, b=7, c}; alphabet;
```

A sized constant can be used to set the size of the type. All sizes must be the same.

```
// silver=4'h4, gold=4'h5 (all are 4 bits wide)
enum {bronze=4’h3, silver, gold} medal4; ~~// 4 bits wide~~
```

A type name can be given so that the same type can be used in many places.

```
typedef enum {NO, YES} boolean;
boolean myvar; // named type
```

Adding a constant range to the **enum** declaration can be used to set the size of the type. If any of the **enum** members are defined with a different sized constant, this shall be a syntax error.

```
        // Error in the bronze and gold member declarations
        enum [3:0] {bronze=5'h13, silver, gold=3'h5} medal4;

        // Correct declaration - bronze and gold sizes are redundant
        enum [3:0] {bronze=4'h13, silver, gold=4'h5} medal4;
```

The type is checked in assignments, arguments and relational operators (which check the values). Like C, there is no overloading of literals, so medal and medal4 cannot be defined in the same scope, since they contain the same names.