Insert the following sections after 5.7

5.8 Type Compatibility

Some SystemVerilog constructs and operations require a certain level of type compatibility for their operands to be legal. There are four levels of type compatibility, formally defined here: Equivalent, Assignment Compatible, Cast Compatible, and Non-Equivalent.

5.8.1 Equivalent Types

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

1. Any built-in type is equivalent to every other occurrence of itself, in every scope.
2. A simple typedef or type parameter override that renames a built-in or user defined type is equivalent to that built-in or user defined type within the scope of the type identifier.
```
typedef bit node; // 'bit' and 'node' are equivalent types
typedef type1 type2; // 'type1' and 'type2' are equivalent types
```
3. An anonymous **enum**, **struct**, or **union** type is equivalent to itself among variables declared within the same declaration statement and no other types.
```
struct {int A; int B;} AB1, AB2; // AB1, AB2 have equivalent types
struct {int A; int B;} AB3; // AB3 is not type equivalent to AB1
```
4. An user defined type (An typedef for an **enum**, unpacked **struct**, or unpacked **union**, or a **class**) is equivalent to itself and variables declared using that type within the scope of the type identifier.
```
typedef struct {int A; int B;} AB_t;
AB_t AB1; AB_t AB2;  // AB1 and AB2 have equivalent types
typedef struct {int A; int B;} otherAB_t;
otherAB_t AB3;       // AB3 is not type equivalent to AB1 or AB2
```
5. Packed arrays, packed structures, and built-in integral types are equivalent if they contain the same number of total bits, are either all 2-state or all 4-state, and are either all signed or all unsigned. Note that if any bit of a packed structure or union is 4-state, the entire structure or union is considered 4-state.
```
typedef bit signed [7:0]BYTE;   // equivalent to the byte type
typedef struct packed signed {bit[3:0] a,b;} uint8;
                                // equivalent to the byte type
```
6. Unpacked array 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.
```
bit [9:0]  A[0:5];
bit [1:10] B[6];
typedef bit [10:1] uint10;
uint10 C[6:1]; // A, B and C have equivalent types
typedef int anint[0:0]; // anint is not type equivalent to int
```
7. Explicitly adding signed or unsigned modifiers to a type that does not change its default signing, does not create a new type. Otherwise, the signing must match to have equivalence
```
typedef bit unsigned ubit; // type equivalent to bit
```
8. A user defined type declared in a package is always equivalent to itself, regardless of the scope where the type is imported.

The scope of a type identifier includes the hierarchical instance scope. This means that each instance with user defined types declared inside the instance creates a unique type. To have type equivalence among multiple instances of the same module, interface, or program, a type must be declared at higher level in the compilation unit scope than the declaration of the module, interface or program, or imported from a package.

The following example is assumed to be within one compilation unit, although the package declaration need not be in the same unit:

```
package p1;
typedef struct {int A;} t_1;
endpackage
typedef struct {int A;} t_2;
module sub();
  import p1:t_1;
  parameter type t_3 = int;
  parameter type t_4 = int;
  typedef struct {int A;} t_5;
  t_1 v1; t_2 v2; t_3 v3; t_4 v4; t_5 v5;
endmodule
module top();
  typedef struct {int A;} t_6;
  sub #(.t_3(t_6)) s1 ();
  sub #(.t_3(t_6)) s2 ();
  initial begin
    s1.v1 = s2.v1; // legal - both types from package p1 (rule 8)
    s1.v2 = s2.v2; // legal - both types from $unit (rule 4)
    s1.v3 = s2.v3; // legal - both types from top (rule 2)
    s1.v4 = s2.v4; // legal - both types are int (rule 1)
    s1.v5 = s2.v5; // illegal - types from s1 and s2 (rule 4)
  end
endmodule
```

5.8.2 Assignment Compatible

All equivalent types, and all non-equivalent types that have implicit casting rules defined between them are assignment compatible types. For example, all integral types are assignment compatible. Conversion between assignment compatible types may involve loss of data by truncation or rounding.

Compatibility may be in one direction only. For example, an **enum** can be converted to an integral type without a cast, but not in the other way around. Implicit casting rules are defined in Section 3 Data Types and Section 7 Operators and Expressions.

5.8.3 Cast Compatible

All assignment compatible types, plus all non-equivalent types that have defined explicit casting rules are cast compatible types. For example, an integral type requires a cast to be assigned to an **enum**.

Explicit casting rules are defined in Section 3 Data Types.

5.8.4 Type Incompatible

These are all the remaining non-equivalent types that have no defined implicit or explicit casting rules. Class handles and **`chandles`** are type incompatible with all other types.

Add the text in blue to the sentence in section 7.15 Aggregate expressions

To be copied or compared, the type of an aggregate expression must be equivalent. See section 8.5.1 Equivalent Types.

Remove the text at the end of section 7.15 Aggregate expressions.

~~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.~~

Replace the text in section 10.5.2 Pass by reference

~~Arguments passed by reference must match exactly, no promotion, conversion, or auto-casting is possible when passing arguments by reference. In particular, array arguments must match their type and all dimensions exactly. Fixed-size arrays cannot be mixed with dynamic arrays and vice-versa.~~

WITH

Arguments passed by reference must be matched with equivalent data types. No casting shall be permitted. See section 8.5.1 Equivalent Types.

Add the following text in blue in section 18.9.1 Port connection rules for variables

— A **`ref`** port shall be connected to an equivalent variable data type. References to the port variable shall be treated as hierarchal references to the variable it is connected to in its instantiation. This kind of port can not be left unconnected. See section 8.5.1 Equivalent Types.

Insert the following sections between 22.1 and 22.22

22.1 Elaboration-time Typeof Function

```
type_function ::= // not in Annex A
$typeof ( expression )
| $typeof ( type_identifier )
| $typeof ( data_type )
```

Syntax 22-1—typeof function syntax (not in Annex A)

The **$typeof** system function returns a type_identifier derived from its argument. The data type returned by the **$typeof** system function may be used to assign or override a type parameter, or in a comparison with another $typeof, evaluated during elaboration.

When called with an expression as its argument, **$typeof** returns a type identifier that represents the self-determined type result of the expression. The expression's return type is determined during elaboration but never evaluated. The expression shall not contain any hierarchical identifiers or references to elements of dynamic objects. In all contexts, **$typeof** together with its argument can be used in any place an elaboration constant is required.

When used in a comparison, equality (==) or case equality is true if the operands are type equivalent. (See section 5.8 Type Equivalency).

For example:

```
bit [12:0] A_bus, B_bus;
parameter type bus_t = $typeof(A_bus);
generate
   case ($typeof(but_t))
      $typeof(bit[12:0]): addfixed_int #(bus_t) (A_bus,B_bus);
      $typeof(real): add_float #($typeof(A_bus)) (A_bus,B_bus);
   endcase
endgenerate
```

The actual value returned by **$typeof** is not visible to the user and is not defined.

22.2 Typename Function

typename_function ::= // not in Annex A
$typename (expression **)**
| **$typename (** type_identifier **)**
| **$typename (** data_type **)**

Syntax 22-1—type function syntax (not in Annex A)

The **$typename** system function returns a string that represents the resolved type of its argument.

The return string is constructed in the following steps:

1. Simple typedefs are resolved back to built-in or user defined types.
2. The default signing is removed, even if present explicitly in the source
3. System generated names are created for anonymous user defined types (structs/unions/enums).
4. A '$' is used as the placeholder for the name of anonymous unpacked array
5. Actual encoded values are appended with numeration labels.
6. User defined type names are prefixed with their defining package or scope namespace.
7. Array ranges are represented as unsized decimal numbers.
8. Whitespace in the source is removed and a single space is added to separate identifiers and keywords from each other.

This process is similar to the way that type equality is computed, except that array ranges and built-in equivalent types are not normalized in the generated string. Thus **$typename** can be used in string comparisons for stricter type-checking of arrays than **$typeof**.

When called with an expression as its argument, $typename returns a string that represents the self-determined type result of the expression. The expression's return type is determined during elaboration but never evaluated. When used as an eleboration time constant, the expression shall not contain any hierarchical identifiers or references to elements of dynamic objects.

```
// source code            // $typename would return
typedef bit node;         // "bit"
node signed [2:0] X;      // "bit signed[2:0]"
int signed Y;             // "int"
package A;
  enum {A,B,C=99} X;      // "enum{A=32'd0,B=32'd1,C='32bX}A::e$1"
  typedef bit [9:1'b1] word // "A::bit[9:1]"
endpackage : A
import A::*;
module top;
  typedef struct {node A,B;} AB_t;
  AB_t AB[10];            // "struct{bit A;bit B;}top.AB_t$[0:9]"
```

Replace all the text in section 22.2 $bits with

size_function ::= // not in Annex A
$bits (expression **)**
| $bits (type_identifier **)**

Syntax 22-2—Size function syntax (not in Annex A)

The `$bits` system function returns the number of bits required to hold an expression as a bit stream. See section 3.16 bit-stream cast for a definition of legal types. A 4-state value counts as one bit. Given the declaration:

```
logic [31:0] foo;
```

Then `$bits`(foo) shall return 32, even if the implementation uses more than 32-bits of storage to represent the 4-state values. Given the declaration:

```
typedef struct {
logic valid;
  bit [8:1] data;
} MyType;
```

The expression `$bits(MyType)` shall return 9, the number of data bits needed by a variable of type `MyType`. The `$bits` function can be used as an elaboration-time constant when used on fixed sized types; hence, it can be used in the declaration of other types or variables.

```
typedef bit[$bits(MyType):1] MyBits;//same as typedef bit [9:1]
MyBits;
MyBits b;
```

Variable b can be used to hold the bit pattern of a variable of type MyType without loss of information.

The **`$bits`** system function returns logic X when called with a dynamically sized type that is currently empty. It is an error to use the **`$bits`** system function directly with a dynamically sized type identifier.

Replace section 22.4 array query functions with

```
array_query_functions ::= // not in Annex A
    array_dimension_function ( array_identifier [ , dimension_expression ] )
    | array_dimension_function ( type_identifier [ , dimension_expression ] )
    | $dimensions ( array_identifier )
    | $dimensions ( type_identifier )

array_dimension_function ::=
    $left
    | $right
    | $low
    | $high
    | $increment
|    $size
dimension_expression ::= expression
```

Syntax 22-2—Array querying function syntax (not in Annex A)

SystemVerilog provides system functions to return information about a particular dimention of an array variable or type. The default for the optional dimension expression is 1. The array dimension can specify any fixed sized index (packed or unpacked), or any dynamically sized index (dynamic, associative, or queue).

— `$left` shall return the left bound (msb) of the dimension
— `$right` shall return the right bound (lsb) of the dimension
— `$low` shall return the minimum of `$left` and `$right` of the dimension
— `$high` shall return the maximum of `$left` and `$right` of the dimension
— `$increment` shall return 1 if `$left` is greater than or equal to `$right`, and -1 if `$left` is less than `$right`
— `$size` shall return the number of elements in the dimension, which is equivalent to `$high` - `$low` + 1
— `$dimensions` shall return the number of dimensions in the array, or 0 for a singular object
The dimensions of an array shall be numbered as follows: The slowest varying dimension (packed or unpacked) is dimension 1. Successively faster varying dimensions have sequentially higher dimension numbers. Intermediate type definitions are expanded first before numbering the dimensions.

For example:

```
// Dimension numbers
//     3     4        1     2
reg [3:0][2:1] n [1:5][2:8];
typedef reg [3:0][2:1] packed_reg;
packed_reg n[1:5][2:8]; // same dimensions as in the lines above
```

For a fixed sized integer type (integer, shortint, longint, and byte), dimension 1 is pre-defined. For an integer N declared without a range specifier, its bounds are assumed to be `[$bits(N)-1:0]`.

If an out-of-range dimension is specified, these functions shall return a logic X.

When used on a dynamic array or queue dimension, these functions return information about the current state of the array. If the dimention is currently empty, these functions shall return a logic X. It is an error to use these functions directly on a dynamically sized type identifier.
Use on associative array dimentions is restricted to index types with integral values. With integral indexes, these functions shall return:

— `$left` shall return 0
— `$right` shall return the highest possible index value
— `$low` shall return the lowest currently allocated index value
— `$high` shall return the largest currently allocated index value
— `$increment` shall return -1
— `$size` shall return the number of elements currently allocated

If the array identifier is a fixed sized array, these query functions may be used as a constant function and passed as a parameter before elaboration. These query functions may also be used on fixed sized type identifiers in which case it is always treated as a constant function.

Given the declaration below:

```
typedef logic [16:1] Word;
Word Ram[0:9];
```

The following system functions return 16:

```
$size(Word)
$size(Ram,2)
```