6. Data Declarations

6.1 Introduction

NOTES:

There are several forms of data in SystemVerilog: literals (see Clause 3), parameters ~~(see Clause 22 6.3~~), constants, variables, nets, and attributes (see ~~Clause 7~~ 4.17). A data object is a named entity that has a data value associated with it, such as a parameter, a variable, or a net.

Verilog constants are literals, genvars parameters, localparams and specparams. Verilog also has variables and nets. Variables must be written by procedural statements, and nets must be written by continuous assignments or ports.

SystemVerilog extends the functionality of variables by allowing them to either be written by procedural statements or driven by a single continuous assignment, similar to a **`wire`**. Since the keyword **`reg`** no longer describes the users intent in many cases, the keyword **`logic`** is added as a more accurate description that is equivalent to **`reg`**. See 6.9.2 for details on SystemVerilog type equivalence rules. Verilog has already deprecated the use of the term *register* in favor of *variable*.

SystemVerilog follows Verilog by requiring data to be declared before it is used, apart from implicit nets. The rules for implicit nets are the same as in Verilog.

A variable can be static (storage allocated on instantiation and never de-allocated) or automatic (stack storage allocated on entry to a scope (such as a task, function or block) and de-allocated on exit). C has the keywords **`static`** and **`auto`**. SystemVerilog follows Verilog in respect of the static default storage class, with automatic tasks and functions, but allows **`static`** to override a default of **`automatic`** for a particular variable in such tasks and functions.

SystemVerilog extends the set of data types that are available for modeling Verilog storage and transmission elements. In addition to the Verilog data types, new predefined data types and user-defined data types can be used to declare constants, variables, and nets.

6.2 Data declaration syntax

data_declaration[15] ::= *// from A.2.1.3*
  [**const**] [lifetime] ~~variable_declaration~~ data_type_or_implicit list_of_variable_decl_assignments **;**
| type_declaration
| package_import_declaration
| virtual_interface_declaration

net_declaration[14] ::=
  net_type [drive_strength | charge_strength] [**vectored** | **scalared**]
    data_type_or_implicit [delay3] list_of_net_decl_assignments **;**

lifetime ::= **static** | **automatic**

14. A charge strength shall only be used with the ~~**trireg**~~ **trireg** keyword. When the ~~**vectored**~~ **vectored** or ~~**scalared**~~ **scalared** keyword is used, there shall be at least one packed dimension.

15. In a data_declaration that is not within the procedural context, it shall be illegal to use the ~~**automatic**~~ **automatic** keyword. In a data_declaration, it shall be illegal to omit the explicit data_type before a list_of_variable_decl_assignments unless the **var** keyword is used.

Syntax 6-1—Data declaration syntax (excerpt from Annex A)

6.3 Constants

Constants are named data variables that never change. ~~There are three kinds of constants, declared with the keywords **localparam**, **specparam** and **const**, respectively.~~ Verilog provides three constructs for defining ~~compile~~ elaboration-time constants: the **`parameter`**, **`localparam`** and **`specparam`** ~~statements~~ declarations.

All three can be initialized with a literal.

```
localparam byte colon1 = ":" ;
specparam delay = 10 ; // specparams are used for specify blocks
const logic flag = 1 ;
```

~~The language~~ Verilog provides four methods for setting the value of parameter constants in a design. Each parameter must be assigned a default value when declared. The default value of a parameter of an instantiated module can be overridden in each instance of the module using one of the following:

— Implicit in-line parameter redefinition (e.g. `foo #(value, value) u1 (...);`)

— Explicit in-line parameter redefinition (e.g. `foo #(.name(value), .name(value)) u1 (...);`)

— **defparam** statements, using hierarchical path names to redefine each parameter

NOTE—The **defparam** statement might be removed from future versions of the language. See 25.2.

6.3.1 Parameter declaration syntax

```
local_parameter_declaration ::=                                   // from A.2.1.1
        localparam data_type_or_implicit  list_of_param_assignments ;
      | localparam type  list_of_type_assignments ;
parameter_declaration ::=
        parameter data_type_or_implicit  list_of_param_assignments
      | parameter type  list_of_type_assignments
specparam_declaration ::=
        specparam [ packed_dimension ] list_of_specparam_assignments ;
data_type_or_implicit ::=                                         // from A.2.2.1
        data_type
      | [ signing ] { packed_dimension }
type_reference ::=
        type ( expression[28] )
      | type ( data_type )
list_of_param_assignments ::= param_assignment { , param_assignment }   // from A.2.3
list_of_specparam_assignments ::= specparam_assignment { , specparam_assignment }
list_of_type_assignments ::= type_assignment { , type_assignment }
param_assignment ::=                                              // from A.2.4
        parameter_identifier { unpacked_dimension } = constant_param_expression
specparam_assignment ::=
        specparam_identifier = constant_mintypmax_expression
      | pulse_control_specparam
type_assignment ::=
        type_identifier = data_type
      | type_identifier = $typeof ( expression[28] )
      | type_identifier = $typeof ( data_type )
parameter_port_list ::=                                           // from A.1.3
        # ( list_of_param_assignments { , parameter_port_declaration } )
      | # ( parameter_port_declaration { , parameter_port_declaration } )
      | #( )
parameter_port_declaration ::=
        parameter_declaration
      | data_type list_of_param_assignments
      | type list_of_type_assignments
```

28. ~~The~~ An expression that is used as the argument ~~to the $typeof system function shall contain no hierarchical references~~ in a type_reference shall not contain any hierarchical references or references to elements of dynamic objects.

Syntax 6-1—Parameter declaration syntax (excerpt from Annex A)

6.3.2 Value parameters

A module, interface, program or class can have parameters, which are set during elaboration and are constant during simulation. They are defined with data types and default values. ~~With SystemVerilog, if no data type is supplied, parameters default to type `logic` of arbitrary size for Verilog compatibility and interoperability.~~ For compatibility with Verilog, if no data type is supplied, the type is determined when the value is determined.

In an assignment to, or override of, a parameter without an explicit type declaration, the type of the right-hand expression shall be ~~unsized,~~ real or integral. If the expression is real, the parameter is real. If the expression is integral, the parameter is a **logic** vector of the same size with range `[size-1:0]`. In an assignment to, or override of, a parameter with an explicit type declaration, the type of the right-hand expression shall be assignment compatible with the declared type.

Unlike non-local parameters, local parameters can be declared in a generate block, in a package, or in a compilation unit scope. In these contexts, the **parameter** keyword can be used as a synonym for the **localparam** keyword.

6.3.2.1 $ as a parameter value

$** can be assigned to parameters of integer types. A parameter to which **$ is assigned shall only be used wherever **$** can be specified as a literal constant.

For example, **$** represents unbounded range specification, where the upper index can be any integer.

```
parameter r2 = $;
property inq1(r1,r2);
   @(posedge clk) a ##[r1:r2] b ##1 c |=> d;
endproperty
assert inq1(3);
```

To support whether a constant is **$**, a system function is provided to test whether a constant is a **$**. The syntax of the system function is

```
$isunbounded(const_expression);
```

`$isunbounded` returns true if *const_expression* is unbounded. Typically, `$isunbounded` would be used as a condition in the generate statement.

The example below illustrates the benefit of using $ in writing properties concisely where the range is parameterized. The checker in the example ensures that a bus driven by signal en remains 0, i.e, quiet for the specified minimum (`min_quiet`) and maximum (`max_quiet`) quiet time.

~~Note that~~ NOTE—The function `$isunbounded` is used for checking the validity of the actual arguments.

```
interface quiet_time_checker #(parameter min_quiet = 0,
                               parameter max_quiet = 0)
                              (input logic clk, reset_n, logic [1:0]en);

   generate
      if ( max_quiet == 0) begin
         property quiet_time;
            @(posedge clk) reset_n |-> ($countones(en) == 1);
         endproperty
         a1: assert property (quiet_time);
      end
      else begin
         property quiet_time;
            @(posedge clk)
               (reset_n && ($past(en) != 0) && en == 0)
               |->(en == 0)[*min_quiet:max_quiet]
            ##1 ($countones(en) == 1);
         endproperty
         a1: assert property (quiet_time);
      end
      if ((min_quiet == 0) && ($isunbounded(max_quiet))
         $display(warning_msg);
```

```
   endgenerate
endinterface


quiet_time_checker #(0, 0) quiet_never (clk,1,enables);
quiet_time_checker #(2, 4) quiet_in_window (clk,1,enables);
quiet_time_checker #(0, $) quiet_any (clk,1,enables);
```

Another example below illustrates that by testing for **$**, a property can be configured according to the requirements. When parameter `max_cks` is unbounded, it is not required to test for `expr` to become false.

```
interface width_checker #(parameter min_cks = 1, parameter max_cks = 1)
                         (input logic clk, reset_n, expr);

   generate
      if ($isunbounded(max_cks)) begin
         property width;
            @(posedge clk)
               (reset_n && $rose(expr)) |-> (expr [* min_cks]);
         endproperty
         a2: assert property (width);
      end
      else begin
         property assert_width_p;
            @(posedge clk)
               (reset_n && $rose(expr)) |-> (expr[* min_cks:max_cks])
                  ##1 (!expr);
         endproperty
         a2: assert property (width);
      end
   endgenerate
endinterface


width_checker #(3, $) max_width_unspecified (clk,1,enables);
width_checker #(2, 4) width_specified (clk,1,enables);
```

6.3.3 Type parameters

SystemVerilog adds the ability for a parameter to also specify a data type, allowing modules or instances to have data whose type is set for each instance.

```
module ma   #( parameter p1 = 1, parameter type p2 = shortint )
            (input logic [p1:0] i, output logic [p1:0] o);
   p2 j = 0; // type of j is set by a parameter, (shortint unless redefined)
   always @(i) begin
      o = i;
      j++;
   end
endmodule

module mb;
   logic [3:0] i,o;
   ma #(.p1(3), .p2(int)) u1(i,o); //redefines p2 to a type of int
endmodule
```

~~In an assignment to, or override of, a parameter without an explicit type declaration, the type of the right-hand expression shall be unsized, real or integral. If the expression is real, the parameter is real. If the expression is~~

~~integral, the parameter is a **logic** vector of the same size with range [size-1:0]. In an assignment to, or override of, a parameter with an explicit type declaration, the type of the right-hand expression shall be assignment compatible with the declared type.~~ In an assignment to, or override of, a type parameter, the right-hand expression shall represent a data type.

It is an error to override a type parameter with a **defparam** statement.

6.3.4 Parameter port lists

SystemVerilog also adds the ability to omit the **parameter** keyword in a parameter port list.

```
class vector #(size = 1);
   logic [size-1:0] v;
endclass

typedef vector#(16) word;

interface simple_bus #(AWIDTH = 64, type T = word) (input bit clk) ;
endinterface
```

In a list of parameters, a parameter can depend on earlier parameters. In the following declaration, the default value of the second parameter depends on the value of the first parameter. The third parameter is a type, and the fourth parameter is a value of that type.

```
module mc # (int N = 5, M = N*16, type T = int, T x = 0)
  ( ... );
...
endmodule
```

6.3.5 Const constants

SystemVerilog adds another form of a local constant, **const**. A **const** form of constant differs from a **localparam** constant in that the **localparam** must be set during elaboration, whereas a **const** can be set during simulation, such as in an automatic task.

A value parameter ~~or local parameter~~ (**parameter**, **localparam** or **specparam**) can only be set to an expression of literals, value parameters or local parameters, genvars, enumerated names, or a constant function of these. Package references are allowed. Hierarchical names are not allowed. A **specparam** can also be set to an expression containing one or more specparams.

A data-type parameter (**parameter type**) can only be set to a data-type. Package references are allowed. Hierarchical names are not allowed.

~~A specparam can also be set to an expression containing one or more specparams.~~

A static constant declared with the **const** keyword can ~~only~~ be set to an expression of literals, parameters, local parameters, genvars, enumerated names, a constant function of these, or other constants. ~~The parameters, local parameters or constant functions can have hierarchical names because constants declared with the **const** keyword are calculated after elaboration. An automatic constant declared with the **const** keyword can be set to any expression that would be legal without the **const** keyword.~~ Hierarchical names are allowed because constants declared with the **const** keyword are calculated after elaboration.

```
const logic option = a.b.c ;
```

~~A constant expression contains literals and other named constants.~~

An automatic constant declared with the **const** keyword can be set to any expression that would be legal without the **const** keyword.

An instance of a class (an object handle) can also be declared with the **const** keyword.

```
const class_name object = new(5,3);
```

This means that the object acts like a variable that cannot be written. The arguments to the **new** method must be constant expressions. The members of the object can be written (except for those members that are declared const).

~~SystemVerilog enhancements to **parameter** and **localparam** constant declarations are presented in Clause 22. SystemVerilog does not change **specparam** constants declarations. A **const** form of constant differs from a **localparam** constant in that the **localparam** must be set during elaboration, whereas a **const** can be set during simulation, such as in an automatic task.~~

6.4 Variables

One form of variable declaration consists of a data type followed by one or more instances.

```
shortint s1, s2[0:9];
```

Another form of variable declaration begins with the keyword **var**. The data type is optional in this case. If a data type is not specified then the data type **logic** shall be inferred.

```
var byte my_byte;    // equivalent to "byte my_byte;"
var v;               // equivalent to "var logic v;"
var [15:0] vw;       // equivalent to "var logic [15:0] vw;"
var enum bit { clear, error } status;
input var logic data_in;
var reg r;
```

A variable can be declared with an initializer, for example:

```
int i = 0;
```

~~In Verilog, an initialization value specified as part of the declaration is executed as if the assignment were made from an initial block, after simulation has started. Therefore, the initialization can cause an event on that variable at simulation time zero.~~

~~In SystemVerilog, setting the initial value of a static variable as part of the variable declaration (including static class members) shall occur before any **initial** or **always** blocks are started, and so does not generate an event. If an event is needed, an **initial** block should be used to assign the initial values.~~

In Verilog, an initialization value specified as part of the declaration is executed as if the assignment were made from an **initial** block, after simulation has started. In SystemVerilog, setting the initial value of a static variable as part of the variable declaration (including static class members) shall occur before any **initial** or **always** blocks are started.

Initial values in SystemVerilog are not constrained to simple constants; they can include run-time expressions, including dynamic memory allocation. For example, a static class handle or a mailbox can be created and initialized by calling its **new** method (see 14.3.1), or static variables can be initialized to random values by calling the `$urandom` system task. This requires a special pre-initial pass at run-time.

The following table contains the default values for SystemVerilog variables.