SystemVerilog Parameters and Their Datatypes

Dave McEwan

2022-08-25

Contents

This tutorial paper covers some of the more subtle details around SystemVerilog datatypes, in particular how their specifications apply to overridable parameters in synthesisable designs. Firstly, the main arguments are summarised as a set of 6 guidelines. The middle sections explain the necessary details and provide reasoning for the set of guidelines. Finally, some appendices cover relevant material including a clarification of related syntax, a recommended method of sanity-checking parameters, and important extracts from the SystemVerilog Language Reference Manual. An accompanying SystemVerilog file contains examples of all points which is additionally useful to demonstrate differences between simulators.

Guidelines for Synthesisable SystemVerilog

- Each module parameter should have an explicit datatype.
- The datatype of each module parameter should be 2-state, not 4-state.
- Each module parameter should have an explicit default assignment.
- Do not mix 4-state and 2-state members in packed structures.
- Do not use case (in)equality operation with overridable 4-state parameters.
- Do not use wildcard (in)equality operation with overridable 4-state parameters.

Background on Integral Datatypes

To understand the reasoning behind the first (seemingly simple) guideline, it's important to understand some details around SystemVerilog integral datatypes. Integral datatypes are the "usual" types used in digital design code, though other datatypes are commonly used in analog designs and in verification environments, e.g. `real`, `void`, `chandle`, `string`, and `event`. Integral datatypes are summarised in the following table.

Keyword	2/4-state	Value [1]	Vector [2]	Width	Signedness [4]
bit	2	`'0`	yes	1 [3]	unsigned
byte	2	`'0`	no	8	signed
shortint	2	`'0`	no	16	signed
int	2	`'0`	no	32	signed
longint	2	`'0`	no	63	signed
logic	4	`'X`	yes	1 [3]	unsigned
integer	4	`'X`	no	32	signed
time	4	`'X`	no	64	unsigned

1. Values of nets or variables can be overridden using one of the assignment constructs with (blocking, non-blocking, continuous, and procedural). The value of a parameter, i.e. an elaboration-time constant, is set by either a parameter override in a module instance (such as `Mod #(.FOO(5)) u_inst ();`), or a default value in the module declaration (such as `module Mod #(int FOO = 123) ();`.
2. Vector types may have packed dimensions which are specified before the identifier like `bit [5:0] foo`, as well as unpacked dimensions which are specified after the identifier like `bit foo [5]`. Non-vector types may not have packed dimensions.
3. The default width of a vector type is overridden using packed dimensions.
4. The default signedness of any integral type may be overridden using the `signed` or `unsigned` keywords.

SystemVerilog also allows the user to define their own aggregate datatypes consisting of packed/unpacked structures, unions, enumerations, dynamic arrays, associative array, and queues. The aggregate datatypes relevant to this discussion are packed structures. A packed structure is essentially a bit vector with named bit slices, i.e. it can be operated on as a whole, and has the same attributes as a `bit` or `logic`. As such, a packed structure can be either 2-state or 4-state, which is inferred from the datatypes of its members. If any of a packed structure's members are 4-state, then the whole structure is treated as 4-state.

```
typedef struct packed {
  bit b;      // 2-state
  int a;      // 2-state
} s2;
```

```
typedef struct packed {
  logic b;    // 4-state
  integer a;  // 4-state
} s4;

typedef struct packed {
  logic b;    // 4-state
  int a;      // 2-state
} sM;

// 2-state constants, good practice.
localparam s2 FOO_A = {1'b1, 32'd123};
localparam s2 FOO_B = '1;
localparam s2 FOO_C = constantFoo();

// 4-state constants, warrants close inspection.
localparam s4 BAR_A = {1'bZ, 32'b01XZ}; // Used in a wildcard comparison?
localparam s4 BAR_B = 'X;               // Legal, but probably nonsense!

// Accidentally 4-state constants, intended to be 2-state.
localparam sM BAZ_A = {1'b1, 32'd123};  // All bits are 0 or 1, but 4-state.
localparam sM BAZ_B = constantBaz();    // Maybe hidden X/Z in here.
```

NOTE: IEEE1800-2017 is somewhat unclear on how `sM` should be treated within constant functions, so there may be implementation-specific differences. Compare the messages about `sM` in reports from different simulators for full details.

Overriding Module Parameters

Let's use a couple of example modules to demonstrate exactly what datatypes are inferred by different styles of declaration. In the `CI` module (child, implicit types), the type of each parameter is inferred from its default value (`integer`), but will be changed to the type of any override value from a parent module. It is therefore sensible to somehow check within `CI` that the parameters are of the expected type (using `type`) and size (using `$size`), particularly. The default type of all parameters is `logic [MSB:0]` where `MSB` is at least 31 but is implementation-dependent (see IEEE1800-2017 page 121).

```
module CI
  #(parameter FIVE = 5
  , parameter VEC1D = {32'd1, 32'd2, 32'd3}
  ) ();
endmodule
```

In the `CE2` module (child, explicit 2-state types), the type of each parameter is explicitly declared and cannot be overriden by a parent module. This is the approach advocated in this document. No further checks on the type or size of these parameters are required because any override values froma a parent module are implicitly cast to the explicitly declared type.

```
module CE2
  #(parameter int FIVE = 5
  , parameter bit [2:0][31:0] VEC1D = {32'd1, 32'd2, 32'd3}
  ) ();
endmodule
```

The `CE4` module (child, explicit 4-state types) is similar to `CE2` except that the explicitly declared types are 4-state instead of 2-state.

```
module CE4
  #(parameter integer FIVE = 5
  , parameter logic [2:0][31:0] VEC1D = {32'd1, 32'd2, 32'd3}
  ) ();
endmodule
```

Note that for all 3 examples, the syntax `VEC1D = {1, 2, 3}` would be illegal, i.e. unsized literals are not allowed in concatenations or replications.

In a parent module, override values can be created via localparam constants. The use of intermediate `localparam`s is only to highlight their types, but any constant expressions specified in the same way will have equivalent types.

Implicitly typed override values can have types which are "good", i.e. compatible with what the child module expects, or "bad", i.e. incompatible or unexpected types. The prefixes "IG_" and "IB_" are used to clarify implicit-good and implicit-bad types respectively. Neither `IG_FIVE` or `IG_VEC1D` have well-defined sizes because their implict types are `logic` vectors of *at least* 32 bits, but the exact size is implementation-dependent. `IB_FIVE` is clearly something of a different type, i.e. `string` with four 8b ASCII values (2-state, `32'h66_69_76_65`). The size of `IB_VEC1D` is 66 bits, and the value is the concatenation of 7, 8, and 9 left-shifted by specific amounts.

```
localparam IG_FIVE = 5;
localparam IG_VEC1D = {32'd111, 32'd222, 32'd333};
localparam IB_FIVE = "five";
localparam IB_VEC1D = {11'd7, 22'd8, 33'd9};
```

Similarly, explicitly typed override values can have good or bad types. These examples use the prefixes "EG_" and "EB_" to clarify explicit good/bad types (from the child module's perspective).

```
localparam int EG_FIVE = 5;
localparam bit [2:0][31:0] EG_VEC1D = {32'd111, 32'd222, 32'd333};
localparam logic [3:0] EB_FIVE = 4'bXZ01;
localparam logic [2:0][9:0] EB_VEC1D = {10'd111, 10'd222, 10'd333};
```

To see how the parameter types propagate, let's consider the 12 combinations of explict/implicit and good/bad override values with the implict, explict 2-state, and explicit 4-state child modules:

```
CI #(.FIVE (IG_FIVE), .VEC1D (IG_VEC1D)) u_ci_ig ();
CI #(.FIVE (EG_FIVE), .VEC1D (EG_VEC1D)) u_ci_eg ();
CI #(.FIVE (IB_FIVE), .VEC1D (IB_VEC1D)) u_ci_ib ();
CI #(.FIVE (EB_FIVE), .VEC1D (EB_VEC1D)) u_ci_eb ();

CE2 #(.FIVE (IG_FIVE), .VEC1D (IG_VEC1D)) u_ce2_ig ();
CE2 #(.FIVE (EG_FIVE), .VEC1D (EG_VEC1D)) u_ce2_eg ();
CE2 #(.FIVE (IB_FIVE), .VEC1D (IB_VEC1D)) u_ce2_ib ();
CE2 #(.FIVE (EB_FIVE), .VEC1D (EB_VEC1D)) u_ce2_eb ();

CE4 #(.FIVE (IG_FIVE), .VEC1D (IG_VEC1D)) u_ce4_ig ();
CE4 #(.FIVE (EG_FIVE), .VEC1D (EG_VEC1D)) u_ce4_eg ();
CE4 #(.FIVE (IB_FIVE), .VEC1D (IB_VEC1D)) u_ce4_ib ();
CE4 #(.FIVE (EB_FIVE), .VEC1D (EB_VEC1D)) u_ce4_eb ();
```

All instances of `CI` should elaborate successfully and each child module parameter is given the type of its corresponding override value. Logical constructions within `u_ci_ib` and `u_ci_eb` (bad/unexpected types) based on those parameters may have different semantics depending on the overriden types. All instances of the `CE2` and `CE4` modules (which have explicit types) can freely use their parameters in the knowledge that all override values are of the expected type. That is, `FIVE` is always signed, 2-state, 32b, and "the 3nd element of `FIVE`" is always taken to mean an unsigned, 2-state, 1b value. Any overrides where the value cannot be coerced to the child's type will cause an error in elaboration.

The following table compares the type:value semantics of "the 3rd element of FIVE" and "the 2nd element of VEC1D" over the 12 instances.

Instance	FIVE[2]	VEC1D[1]
CI:		
u_ci_ig	logic:1'b1	logic:1'b0
u_ci_eg	bit:1'b1	bit:1'b0
u_ci_ib	logic:1'b1	logic:1'b0
u_ci_eb	logic:1'bZ	bit:1'b0
CE2:		
u_ce2_ig	bit:1'b1	bit [31:0]:32'd222
u_ce2_eg	bit:1'b1	bit [31:0]:32'd222
u_ce2_ib	bit:1'b1 ("e" = 8'h65)	bit [31:0]:32'h0380_0010
u_ce2_eb	bit:1'b0 (XZ01 $\rightarrow$ 0001)	bit [31:0]:32'd0
CE4:		
u_ce4_ig	logic:1'b1	logic [31:0]:32'd222
u_ce4_eg	logic:1'b1	logic [31:0]:32'd222
u_ce4_ib	logic:1'b1 ("e" = 8'h65)	logic [31:0]:32'h0380_0010
u_ce4_eb	logic:1'bZ	logic [31:0]:32'd0

The potential for parameters to be overridden by values of types which the author did not expect is clearly demonstrated. The author of CE2 has the easiest task, as all parameter values are of a fixed type (signedness, 2/4-state, size). The author of CI should take care to handle each supported variation carefully and present an elaboration error when an unsupported variation is detected. In particular, the size of each parameter element should be checked, and care should be taken with the use of operators (e.g. == vs ===), as discussed in the later section on Comparison Operations.

Bottom-Up Type Definitions

One argument for untyped module parameters is that the child (CI) has useful visibility of the type of an override value. This argument is based on the premise that a child module doesn't trust that parent modules will necessarily provide sensible override values. The expected type of each parameter is defined through a mixture of elaboration-time checks, run-time checks, and the exact semantics of *all* parameter uses. At first glance, this may appear desirable, but on further reflection, this presents further problems:

- The authors of all parent and grandparent modules must fully understand the child's expected parameter types, but are forced to infer types manually. This involves the slow and error-prone process of reading all uses of each untyped parameter port on the child module.

- A parent module author expects to write whatever code they see fit, and instance a child module however they deem appropriate. A parent module author does not expect the child module to dictate the form and rigor of override parameters by forbidding explicit types.
- Any type convensions in intermediate parent modules remove the benefit of checks in the child modules. For example, in the hierarchy `grandparent.parent.child` where a parameter is passed from `grandparent` to `child`, any conversions (explicit or implict) in `parent` remove the usefulness of type checks in `child`. That means that *every* intermediate parent should be checked to ensure it does not contain any implicit or explicit type conversions - something not possible to enforce via any SytemVerilog language feature.
- If the meaning or expected type of any parameter is changed in the child module, the checks must be reworked (difficult, error-prone) and corresponding changes are required in all parent modules. Where a check is forgotten or updated incorrectly there is no way to detect this.

The simpler option is for each child module to explicitly define the type of each parameter, giving parent modules a canonical and easily readible account of which types are valid. The child module can, and should, still have elaboration-time checks on the values of each parameter.

Default Values

Just as parameters, like all data objects, have default types defined by the language specification, they also have default values. Where a default type is implied by lack of an explicit type in a parameter's declaration, a default value is implied by lack of an assignment. Fortunately, the rules for integral data objects are simple:

1. All unassigned bits of a 4-state parameter have the implicit value `'X`.
2. All unassigned bits of a 2-state parameter have the implicit value `'0`.

One way for a parameter to accidentally contain default values is through use of a constant function which doesn't assign to every component.

```
function automatic integer f_myConstant ();
  for (int i=0; i < 32; i++) begin
    case (i % 3)
      0: f_myConstant[i] = 1'b0;
      1: f_myConstant[i] = 1'b1;
      // Woops! Missing arm for i=2?
    endcase
  end
endfunction

CE4 #(.FIVE (f_myConstant())) u_ce4_x ();
```

In the above example, the instance of `u_ce4_x` appears to override the value of `FIVE`, but only a full semantic analysis of `f_myConstant` can reveal that some bits are, perhaps unexpectedly, `1'bX`. In real-world usage, the definition of `f_myConstant` may be in a different file, or be selected from alternative definitions in a range of files, i.e. finding the correct definition may be non-trivial. Also in real-world usage, the construction of `f_myConstant` may be much more complex than the simple example above, perhaps through deep conditional logic and spread over many lines. This shows that with parameters declared using implicit or explicit 4-state types it difficult for a human reader to be confident that any override values do not contain Xs.

NOTE: Again, due to the imprecise wording of IEEE1800-2017, the exact override value may be different between simulators. Compare the messages about `u_ce4_x.FIVE` in reports from different simulators for full details.

Default Assignments

An author can clarify to readers that they have considered the default value of a module parameter by using an explict default assignment. This allows an integrator to avoid providing an override value, perhaps because default value is sensible, or perhaps because they forgot. To force an integrator to consider a specific parameter and provide an override value, an author can declare a parameter without a default assignment. In the following example, the parameters have explicit types, but no default assignment.

```
module REQ
  #(parameter ANY
  , parameter bit BIT
  , parameter logic LOGIC
  , parameter int INT
  , parameter integer INTEGER
  ) ();
endmodule
```

Parameter ports without a default assignment must be overriden by parent modules, otherwise an error should be raised at elaboration time, see IEEE1800-2017 Annex A footnote 18. Override parameters provided by a parent module must be of a compatible type because the override value will be coerced to the type declared in the child module. In the above example, the override values (except for `ANY`) will be coerced to the corresponding type, but the child module must accept a value of any type for `ANY`.

NOTE: Omitting default assignments may not be supported by all tools, see the attached `ParameterDatatypes.sv` for further details.

Comparison Operations

So far, we have seen that parameters have implicit or explicit types, and implicit or explict values based on those types. In particular, it has been shown that 4-state parameters may contain Xs, but nothing has been said about why this may cause problems. One source of potential problems with 4-state parameters lies in the use of SystemVerilog comparison operations.

Comparison operation are those which take two operands and return a 1b value which says whether the comparison was successful, unsuccessful, or unknown. The comparison operations in SystemVerilog are:

Operator	Type	Name	Class
`<`	`logic`	less than	partial ordering
`>`	`logic`	greater than	partial ordering
`<=`	`logic`	less than or equal to	partial ordering
`>=`	`logic`	greater than or equal to	partial ordering
`==`	`logic`	logical equality	partial equality
`!=`	`logic`	logical inequality	partial equality
`===`	`bit`	case equality	total equality
`!==`	`bit`	case inequality	total equality
`==?`	`logic`	wildcard equality	partial equality
`!=?`	`logic`	wildcard inequality	partial equality

The difference between partial and total ordering/equality operations is that a total order/equality comparison must return either True (`1'b1`) or False (1'b0), but a partial order/equality comparison may additionally return a result of Unknown (`1'bX`). For all of the partial ordering/equality operations, if the LHS operand contains any Xs, then the result is `1'bX`. This also applies to Xs in RHS operands, except for wildcard (in)equality operations.

Where a signal `c` is compared against a parameter-defined value, it is essential for synthesisable code to propagate any Xs in `c` to the result of the comparison. Without the ability to propagate Xs through the comparison, e.g. `d = (c === 32'd5)`, a simulator will assign `d = 1'b0` when any bits of `c` are unknown. A synthesised circuit does not have the concept of "unknown" values, so use of the case (in)equality operators will cause a simulation/synthesis mismatch.

As suggested by the names "logical equality" and "case equality", the same semantics are applied to conditional if-else statements and case statements respectively. The subtle difference between case and logical equalities introduces a high potential for simulation/synthesis mismatches when used with parameters containing Xs.

```
localparam logic [1:0] OKAY = 2'b00;
localparam logic [1:0] WOOPS = 2'bX1;
```

```
logic [1:0] a;
integer b1, b2;

always_comb
  case (a)
    OKAY:    b1 = 555;
    WOOPS:   b1 = 666;
    default: b1 = 777;
  endcase

always_comb
  if (a == OKAY)
    b2 = 555;
  else if (a == WOOPS)
    b2 = 666;
  else
    b2 = 777;
```

In the above example, two 4-state parameters are defined, but `WOOPS` has one bit accidentally set to `a`. Two synthesisable signals `b1` and `b2` are assigned to by *approximately* equivalent processes. The author is intends that whenever `a` contains bits of unknown value, `b1` and `b2` are also of unknown value, or perhaps `777`. Instead, `b1` has a case where an unknown value in `a` is not propagated the value of `WOOPS` is matched using case equality. Note that the `b2` is not exactly equivalent, and that any Xs in `a`, `OKAY`, or `WOOPS` will be propagated, as intended. While this contrived example is very simple, it should be clear that more complex constructions with 4-state parameters are at increased risk of introducing exact comparison matches to `1'bX`. The risk is particularly high when code is restructured and/or the code does not have a verification environment mandating 100% toggle, expression and condition coverage.

Wildcard (in)equality operations are one usecase where Xs in a parameter is a useful, if not essential, ability. In a wildcard (in)equality operation, Xs and Zs may be used to mask out bits from the LHS operand. Positive matches can be achieved using multiple LHS values, e.g. `4'b0100 ==? 4'b01XZ` → `1'b1` and `4'b0111 ==? 4'b01XZ` → `1'b1`. Negative matches can be also be achieved using multiple LHS values, e.g. `4'b1100 ==? 4'b01XZ` → `1'b0` and `4'b1111 ==? 4'b01XZ` → `1'b0`. However, care must be taken about which side operands are placed on; Although both `1'b0 ==? 1'bX` and `1'b1 ==? 1'bX` result in `1'b1`, changing sides changes the results as both `1'bX ==? 1'b0` and `1'bX ==? 1'b1` result in `1'bX`. Where wildcard (in)equality operations are used with module parameters, it is not possible for a child module's author to know if any Xs are intentional or accidental. Instead of using a wildcard equality operation with Xs or Zs to mask out certain bits, it is recommended that more general masking is used, i.e. bitwise negation, AND, OR, etc. operations.

Conclusion

The semantics and effects of module parameter port datatypes have been described through worked examples to demonstrate features of the SystemVerilog language. Parameters always have a datatype, whether it is implictly defined, explicitly declared, or given by the type of an override value. Similarly, parameters always have a value which can be set implictly, via a default assignment, or provided by a parent module. In typical design code, parameters should be 2-state, although this requires authors to pay due care to their declarations. The use of case statements is shown to be a potential source of simulation/synthesis mismatch problems when used with 4-state parameters, which must be de-risked through either rigourous manual review or application of the 6 guidelines.

Practical demonstrations of all described syntax and semantics can be seen by running the attached SystemVerilog file in a variety of simulators via the attached Makefile. It is particularly interesting to compare the reports produced by different simulators, demonstrating that concerns about mismatches between tools are founded in well-founded.

Appendix: Concatenations vs Array Literals

A frequent point of confusion in SystemVerilog literal arrays is the syntax `'{a, b}` vs `{a, b}` (note the apostrophe), packed vs unpacked, and implicit vs explicit types. Braces (`{` and `}`) are used in SystemVerilog for several purposes:

- Concatenation, e.g. `{4'hA, 4'h5}` → `8'hA5`.
- Streaming concatenation, e.g. `{<< 4 {16'hABCD}}` → `16'hDCBA`
- Replication, e.g. `{3{4'hA}}` → `12'hAAA`
- Set membership, e.g. `3 inside {1, 2, 3}` → `1'b1`
- Array literals, e.g. `bit [2:0][31:0] foo = '{1, 2, 3}`
- Structure literals, e.g. `'{fieldA:1, fieldB:2, default:3}`

NOTE: Although logical reductions like conjunction, disjunction, etc. often use a unary operator (like `&`) together with a concatention, e.g. `&{FOO < 5, 32'd123}` → `1'b0`, the braces still define a concatenation.

The apostrophe character (`'`) is also used for several purposes:

- Width delimiter in literals, e.g. `5'd123`.
- Casting to a type, e.g. `int'(foo)`.
- Casting to a width, e.g. `(12)'(foo)`.
- Assignment patterns, e.g. `a = '{1, 2, default:3}`.

These examples show a variety of valid syntax assigning the numbers 1, 2, and 333 to constant arrays.

```
localparam bit [3:0][31:0] concatenationA = {32'd1, 32'd2, 32'd333};
localparam bit [127:0] concatenationB = {32'd1, 32'd2, 32'd333};

localparam bit [2:0][31:0] literalPackedA = '{1, 2, 333};
localparam bit [0:2][31:0] literalPackedB = '{1, 2, 333};

localparam int literalUnpackedA [3] = '{1, 2, 333};
localparam int literalUnpackedB [0:2] = '{1, 2, 333};
localparam int literalUnpackedC [2:0] = '{1, 2, 333};
```

The following table demonstrates the importance of the type of the assignment target, including the order dimensions.

Name	`[0]`	`[1]`	`[2]`
concatenationA	333	2	1
concatenationB	`1'b1`	`1'b0`	`1'b1`
literalPackedA	333	2	1
literalPackedB	1	2	333
literalUnpackedA	1	2	333
literalUnpackedB	1	2	333
literalUnpackedC	333	2	1

Appendix: Checking Parameter Values

Parameters are constants at elaboration time, therefore they can be sanity checked at elaboration time. SystemVerilog has a set of language features specifically included for this purpose and, as described in IEEE1800-2017 Clause 20.11 (Elaboration system tasks), is comprised of four tasks (`$fatal`, `$error`, `$warning`, and `$info`). Although these task names are also valid at run-time, it is much nicer from user/integrator's perspective if parameters checks are performed during elaboration (with a debug loop of maybe 5 seconds) versus during simulation (with a debug look of maybe 5 minutes). The following examples demonstrate a variety of ways to check that a parameter `FOO` is integral and less than `5`.

Poor Practice

Firstly, the use of an `initial` block declares a process which executes exactly once, at the start of a simulation. This example also clarifies the concept of "positive form" (saying what you want) versus "negative form" (saying what you *don't* want.

```
initial begin
  if (FOO >= 5)         // Negative form.
    $error("FOO is greater than 5.");

  if (!(FOO < 5))       // Positive form.
    $error("FOO is not less than 5.");

  // Rephrased as an immediate assertion.
  assert (!(FOO < 5));
end
```

Terrible Practice

A related style is to schedule the check to be executed multiple times, e.g. on every rising edge of a clock, wasting valuable simulation time. This example also demonstrates another bad practice - using monadic functionality in non-testbench code. On a multi-threaded simulation, the simulator may execute `always` processes in parallel or in any order, so the user has no control over the ordering of `$display`'d characters on STDOUT, or which thread updates `errorCount` first.

```
always @(posedge clk) if (!(FOO < 5)) begin
  $display("FOO is not less than 5.");  // IO function.
  errorCounter++;                       // Global variable update.
end
```

```
// Rephrased as a concurrent assertion.
property prop_fooLt5;
  @(posedge clk) !(FOO < 5)
endproperty
asrt_fooLt5: assert property (prop_fooLt5);
```

Good Practice

To check parameters at elaboration-time, as opposed to run-time, generate statements can be used with elaboration system tasks. Note that this example is almost identical in syntax to the examples in the `initial` block. Simply removing `initial begin` and `end` converts the run-time process into a generate statement which is evaluated in elaboration.

```
if (!(FOO < 5))
  $error("FOO is not less than 5.");
```

A Neat Style (PARAMCHECK)

Parameters can be checked to be one of a choice of values, be in a range of values, or to satisfy any other type of constraint which can be written as a boolean expression. When there are many parameters and many constraints, writing a separate generate statement for each constraint can produce an overwhelming volume of code. A useful style with a lower volume of code is to combine a set of constraints via conjunction. Note that in a conjunction, i.e. "all of these must be true", constraint must be written in positive form. To help readers understand your intent, and avoid confusion around indentation and punctuation, it is also useful to declare a boolean constant. The following example lists all constraints in the conjunction defining `PARAMCHECK_ALLGOOD`, then uses a single conditional generate statement to perform the check in elaboration.

```
localparam bit PARAMCHECK_ALLGOOD =
  &{(0 < WIDTH)
  , (WIDTH < 22)
  , (MIN_DEPTH <= DEPTH)
  , (DEPTH <= MAX_DEPTH)
  , (FOO < 5)
  };
if (!PARAMCHECK_ALLGOOD) begin
  $error("Parameter constraint violation.");
  $info("WIDTH=%0d%", WIDTH);
  $info("DEPTH=%0d%", DEPTH);
  $info("FOO=%b%", FOO);
end
```

Elaboration system tasks are defined to output their file path, line number, and hierarchical scope, so the user can find the list of constraints in the source code. To help debug elaboration errors, the above example uses `$info` to show the values which the constraints are applied to. With the list of constraints (in source code) and the exact input to those constraints (via `$info`), the user is provided with everything they need to identify all of their parameter constraint violations.

It is preferable to use `$error`, rather than `$info`, `$warning` or `$fatal`. Both `$info` and `$warning` will allow simulation to proceed and their messages may be overlooked in long simulation logs. Elaboration is stopped immediately upon hitting `$fatal`, which means that the user is shown only one problem at once. Using `$error` allows elaboration to complete, giving the user all messages in one shot, but prevents simulation from occurring.

For parameters with complex requirements, intermediate constants and constant functions are useful extensions to this style of parameter checking.

```
function automatic bit [N_ITEM-1:0] f_paramcheck_MYARRAY ();
  for (int i=0; i < N_ITEM; i++)
    f_paramcheck_MYARRAY[i] =
      &{(0 <= MYARRAY[i])
      , (MYARRAY[i] < 5)
      , (MYARRAY[i] != 2)
      };
endfunction
localparam bit [N_ITEM-1:0] PARAMCHECKGOOD_MYARRAY = f_paramcheck_MYARRAY();

localparam bit PARAMCHECK_ALLGOOD =
  &{(0 < WIDTH)
  // ... snip ...
  , &PARAMCHECKGOOD_MYARRAY
  // ... snip ...
  };
if (!PARAMCHECK_ALLGOOD) begin
  $error("Parameter constraint violation.");
  // ... snip ...
  $info("PARAMCHECKGOOD_MYARRAY=%b", PARAMCHECKGOOD_MYARRAY);
  // ... snip ...
  for (genvar i=0; i < N_ITEM; i++)
    $info("MYARRAY[%0d]=%0d", i, MYARRAY[i]);
end
```

The above example checks each item in an array against a complex set of constraints. The user is first presented with a bit-vector showing `1` for a passing items and `0` for items violating their constraints, then the exact value of each item in the array.

It is useful to give intermediate constants and constant functions a prefix which is consistent and describes the intent, i.e. “PARAMCHECK”. Without further context, a new reader will able to easily identify that all objects with “PARAMCHECK” (or similar) in their identifier are used for checking parameters. A consistent prefix enables code analysis tools to identify that these objects are used for checking parameter against constraints and may extract relevant expressions or values, e.g. for documentation.

The above examples are compatible with modern versions of SystemVerilog (IEEE1800-2009,2012,2017).

A Backwards-Compatible Approach

Some projects require compatibility with older versions of (System)Verilog. Elaboration system tasks (`$error`, `$info`, etc.) were only added in the 2009 version of SystemVerilog, so another approach was/is required for designs based on earlier versions. One way to perform elaboration-time checks is by coercing a boolean (from a check) to an integer index of a vector. The following examples are compatible with every version of Verilog (IEEE1364-1995,2001,2005) and SystemVerilog (IEEE1800-2005,2009,2012,2017).

```
wire [1:1] dummy1; // Only index 1/true is valid.
wire paramcheck_allgood = dummy1[ // Compact form.
  &{(0 < WIDTH)
  , (WIDTH < 22)
  , (MIN_DEPTH <= DEPTH)
  , (DEPTH <= MAX_DEPTH)
  , (FOO < 5)
  }];
wire paramcheck_FOO = dummy1[FOO < 5]; // Verbose form.

wire dummy0;
wire paramcheck_BAR = dummy0[!(BAR < 5)]; // Negative form.
```

When a tool encounters a violation, the index of a dummy signal is set to an invalid index. The exact error message is implementation dependent, so some users may prefer a compact style, while others may prefer the verbose form, based on how much information their tools report.

Appendix: Key Quotes from the Language Reference Manual (IEEE1800-2017)

These are key pieces of the SystemVerilog specification, reproduced here to provide an overview of where the most relevant information lies. Although care is taken to provide a reasonable amount of context here, there is no substitute for reading the LRM.

Table 6-7 Default variable initial values

Type	Default initial value
4-state integral	`'X`
2-state integral	`'0`
...	...

Table 6-8 Integer data types

Keyword	Attributes
`shortint`	2-state data type, 16-bit signed integer
`int`	2-state data type, 32-bit signed integer
`longint`	2-state data type, 64-bit signed integer
`byte`	2-state data type, 8-bit signed integer or ASCII character
`bit`	2-state data type, user-defined vector size, unsigned
`logic`	4-state data type, user-defined vector size, unsigned
`reg`	4-state data type, user-defined vector size, unsigned
`integer`	4-state data type, 32-bit signed integer
`time`	4-state data type, 64-bit unsigned integer

Clause 6.4 Singular and aggregate types

Data types are categorized as either singular or aggregate. A singular type shall be any data type except an unpacked structure, unpacked union, or unpacked array (see 7.4 on arrays). An aggregate type shall be any unpacked structure, unpacked union, or unpacked array data type. A singular variable or expression represents a single value, symbol, or handle. Aggregate expressions and variables represent a set or collection of singular values. Integral types (see 6.11.1) are always singular even though they can be sliced into multiple singular values. The `string` data type is singular even though a string can be indexed in a similar way to an unpacked array of bytes.

These categories are defined so that operators and functions can simply refer to these data types as a collective group. For example, some functions recursively descend into an aggregate variable until reaching a singular value and then perform an operation on each singular value.

Although a class is a type, there are no variables or expressions of class type directly, only class object handles that are singular. Therefore, classes need not be categorized in this manner (see Clause 8 on classes).

Clause 6.11.2 2-state (two-value) and 4-state (4-value) data types

Types that can have unknown and high-impedance values are called 4-state types. These are `logic`, `reg`, `integer`, and `time`. The other types do not have unknown values and are called 2-state types, for example, `bit` and `int`.

The difference between `int` and `integer` is that `int` is a 2-state type and `integer` is a 4-state type. The 4-state values have additional bits, which encode the `X` and `Z` states. The 2-state data types can simulate faster, take less memory, and are preferred in some design styles.

The keyword `reg` does not always accurately describe user intent, as it could be perceived to imply a hardware register. The keyword `logic` is a more descriptive term. `logic` and `reg` denote the same type.

Automatic type conversions from a smaller number of bits to a larger number of bits involve zero extensions if unsigned or sign extensions if signed. Automatic type conversions from a larger number of bits to a smaller number of bits involve truncations of the most significant bits (MSBs). When a 4-state value is automatically converted to a 2-state value, any unknown or high-impedance bits shall be converted to zeros.

Clause 6.20.1 Parameter declaration syntax (bottom of page 120)

The `parameter` keyword can be omitted in a parameter port list.

Clause 6.20.2 Value parameters (bottom half of page 121)

A parameter constant can have a type specification and a range specification. The type and range of parameters shall be in accordance with the following rules:

- A parameter declaration with no type or range specification shall default to the type and range of the final value assigned to the parameter, after any value overrides have been applied. 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]`.

- A parameter with a range specification, but with no type specification, shall have the range of the parameter declaration and shall be unsigned. The sign and range shall not be affected by value overrides.
- A parameter with a type specification, but with no range specification, shall be of the type specified. A signed parameter shall default to the range of the final value assigned to the parameter, after any value overrides have been applied.
- A parameter with a signed type specification and with a range specification shall be signed and shall have the range of its declaration. The sign and range shall not be affected by value overrides.
- A parameter with no range specification and with either a signed type specification or no type specification shall have an implied range with an lsb equal to 0 and an msb equal to one less than the size of the final value assigned to the parameter.
- A parameter with no range specification, with either a signed type specification or no type specification, and for which the final value assigned to it is unsized shall have an implied range with an lsb equal to 0 and an msb equal to an implementation-dependent value of at least 31.

Clause 7.2.1 Packed structures

A packed structure is a mechanism for subdividing a vector into subfields, which can be conveniently accessed as members. Consequently, a packed structure consists of bit fields, which are packed together in memory without gaps. An unpacked structure has an implementation-dependent packing, normally matching the C compiler. A packed structure differs from an unpacked structure in that, when a packed structure appears as a primary, it shall be treated as a single vector.

A packed structure can also be used as a whole with arithmetic and logical operators, and its behavior is determined by its signedness, with unsigned being the default. The first member specified is the most significant and subsequent members follow in decreasing significance.

```
struct packed signed {
  int a;
  shortint b;
  byte c;
  bit [7:0] d;
} pack1; // signed, 2-state

struct packed unsigned {
  time a;
  integer b;
  logic [31:0] c;
} pack2; // unsigned, 4-state
```

The signing of unpacked structures is not allowed. The following declaration would be considered illegal:

```
typedef struct signed {
  int f1 ;
  logic f2 ;
} sIllegalSignedUnpackedStructType; // illegal declaration
```

If all data types within a packed structure are 2-state, the structure as a whole is treated as a 2-state vector.

If any data type within a packed structure is 4-state, the structure as a whole is treated as a 4-state vector. If there are also 2-state members in the structure, there is an implicit conversion from 4-state to 2-state when reading those members and from 2-state to 4-state when writing them.

One or more bits of a packed structure can be selected as if it were a packed array with the range [n-1:0]: `pack1 [15:8] // c` Only packed data types and the integer data types summarized in Table 6-8 (see 6.11) shall be legal in packed structures.

A packed structure can be used with a typedef.

```
typedef struct packed { // default unsigned
  bit [3:0] GFC;
  bit [7:0] VPI;
  bit [11:0] VCI;
  bit CLP;
  bit [3:0] PT ;
  bit [7:0] HEC;
  bit [47:0] [7:0] Payload;
  bit [2:0] filler;
} s_atmcell;
```

Clause 11.4.12 Concatenation operators (page 271)

Unsized constant numbers shall not be allowed in concatenations. This is because the size of each operand in the concatenation is needed to calculate the complete size of the concatenation.

A concatenation is not the same as a structure literal (see 5.10) or an array literal (see 5.11). Concatenations are enclosed in just braces (`{ }`), whereas structure and array literals are enclosed in braces that begin with an apostrophe (`'{ }`).

Clause 12.5 Case statement (middle of page 305)

Apart from syntax, the case statement differs from the multiway if–else–if construct in two important ways:

1. The conditional expressions in the if–else–if construct are more general than comparing one expression with several others, as in the case statement.
2. The case statement provides a definitive result when there are `x` and `z` values in an expression.

In a *case_expression* comparison, the comparison only succeeds when each bit matches exactly with respect to the values `0`, `1`, `x`, and `z`. As a consequence, care is needed in specifying the expressions in the case statement. The bit length of all the expressions needs to be equal, so that exact bitwise matching can be performed. Therefore, the length of all the *case_item_expressions*, as well as the *case_expression*, shall be made equal to the length of the longest *case_expression* and *case_item_expressions*. If any of these expressions is unsigned, then all of them shall be treated as unsigned. If all of these expressions are signed, then they shall be treated as signed. The reason for providing a *case_expression* comparison that handles the `x` and `z` values is that it provides a mechanism for detecting such values and reducing the pessimism that can be generated by their presence.

Clause 23.10 Overriding module parameters (between pages 731,732)

A value parameter (see 6.20.2) can have a type specification and a range specification. The effect of parameter overrides on a value parameter's type and range shall be in accordance with the following rules:

- A value parameter declaration with no type or range specification shall default to the type and range of the final override value assigned to the parameter.
- A value parameter with a range specification, but with no type specification, shall have the range of the parameter declaration and shall be unsigned. An override value shall be converted to the type and range of the parameter.
- A value parameter with a type specification, but with no range specification, shall be of the type specified. An override value shall be converted to the type of the parameter. A signed parameter shall default to the range of the final override value assigned to the parameter.
- A value parameter with a signed type specification and with a range specification shall be signed and shall have the range of its declaration. An override value shall be converted to the type and range of the parameter.