

16.6 Let construct (Note to the editor: Please shift clause numbering accordingly.)

```
let_declaration ::= // from A.12.2.10
    let let_identifier[ ( [let_port_list] ) ] = expression_or_dist;
let_identifier ::=
    identifier
let_port_list ::=
    let_port_item { , let_port_item }
let_port_item ::=
    { attribute_instance } let_formal_type identifier [= expression_or_dist]
let_formal_type ::=
    data_type_or_implicit
    | context
let_instance ::=
    let_identifier[ ( [let_list_of_arguments] ) ]
expression ::=
    ...
    | let_instance
let_list_of_arguments ::=
    [let_actual_arg] { , [let_actual_arg] } { , . identifier ([let_actual_arg] ) }
    | . identifier ([let_actual_arg] ) { , . identifier ( [let_actual_arg] ) }
let_actual_arg ::=
    expression_or_dist
```

The let statement provides means for defining parameterized expressions that can be used in other let statements and assertions.

Such definitions may be used for customization and may replace the compiler directives in many cases. The let construct is safer because it has a local scope, while the scope of compiler directives is global. Let is also more flexible, since like sequences and properties it allows defining default argument values and argument passing both by order and by name. Including let statements into packages (25) is a natural way to implement a well-structured customization for assertions. For example,

```
package pex_gen9_common_expressions;
    let VALID_ARB(req, vld, arb_override) = (|(req&vld) || arb_override));
    ...
endpackage

module my_checker;
    import pex_gen9_common_expressions::*;
    logic a, b;
    wire [1:0] request;
    wire [1:0] valid;
    reg arb_out, ovr;
    ...
    prop: assert property(@(posedge clk)
```

```

        request |-> VALID_ARB(request,valid, ovr);
    ...
endmodule

```

Like properties and sequences serve as templates for concurrent assertions, **let** may serve this purpose in for immediate assertions. For example,

```

let at_least_two(sig, rst = 1'b0) = rst || ($countones(sig) >= 2);
reg [15:0] sig1; reg [3:0] sig2;
always_comb begin
    q1: assert (at_least_two(sig1));
    q2: assert (at_least_two(~sig2));
end

```

Note that in this case the let statement cannot be substituted by a function since formal arguments of a function need to have a specific type. Thus there would have to be different function definitions for different argument widths. The only alternative to **let** is a compiler directive:

```

`define at_least_two(sig, rst = 1'b0) rst || ($countones(sig) >= 2)

```

However, in that case it has a global scope and the information about the assertion template is not available to the compiler for error reporting and run-time debugging.

Another use of let is in modeling for assertions using some additional variables. Usually RTL level code is used for this purpose. For example,

```

wire type(a + b) c; assign c = a + b;
...
assert property(@(posedge clk) cond |=> c < d);

```

However, a synthesis tool may treat such auxiliary signals *c* as real ones and synthesize them into silicon. Workarounds using ``ifdef` are needed for this purpose. Also, the exact type must be specified or **type** operator must be used for this purpose, but it becomes awkward with long signal or expression name. Moreover, the type operator represents the self-determined width of the result while the user needs the context-determined width. Thus, if both *a* and *b* are two-bit wide then *c* will also two-bit wide, and the result will sometimes be truncated. Using the let statement makes writing more elegant and safe:

```

let c = a + b;
...
assert property(@(posedge clk) cond |=> c < d);

```

The construct thus facilitates the creation of a modeling layer for SystemVerilog Assertions and assertion-based checkers.

The let statement is characterized as follows:

1. The **let** statements can define parameterized expressions that can be instantiated in sequences, properties, verification statements and other **let** definitions.
2. As in sequences and properties, the formal arguments can optionally be typed. To declare a type for a formal argument of a **let** statement, it is required to prefix the argument with a type. A formal argument that is not prefixed by a type will be untyped. A type name can refer to a comma separated list of arguments.
3. There are two ways to achieve implicit typing of arguments. The first is to write the implicitly typed arguments at the beginning of the formal argument list, prior to any typed argument. The second is to use the **context** type. Because a type applies to multiple comma-separated arguments, the **context** type is required if an implicitly typed argument is to be placed after a typed argument in the formal argument list. The **context** type

specifies that the semantics for binding to the argument shall be as if the argument were written at the beginning of the formal argument list, prior to any typed argument.

4. The formal arguments can have optional default values.
5. Scoping rules are such that referenced identifiers / names in the let expression which are not formal arguments bind in the declarative scope of the let statement; this is similar to sequence and property declarations. Such names must be declared before used.
6. In the scope of declaration, let must be defined before used. No cross-module references to **let** definitions are allowed.
7. The **let** expression is inlined in the place of instantiation without any partial evaluation of the expression, the same way as sequences do. Recursive **let** instantiations are not permitted.
8. **let** expressions can contain sampled value function calls (**\$rose**, **\$fell**, **\$past**, **\$stable**, **\$changed**). Their clock if not explicitly specified is inferred in the instantiation context in the same way as if the functions were used directly in sequences, properties or verification statements.
9. The **let** statement identifier must be unique in its namespace.
10. Like functions, let statements can be imported from a package. Binding of identifiers / names that are not formal arguments must be satisfied by the declarations in the package.
11. A reference to a **let** in a context that requires a *constant expression* (see 11.1.1) shall only use the operators defined in Table 11-1 and can refer to constant numbers, strings, parameters, constant bit-selects and part-selects of parameters, constant function calls (see 13.4.4), and constant system function calls only..

Examples:

1) let with arguments and without arguments.

```
module m;
  logic clk, a, b;
  logic p, q, r;
  // with formal arguments and default value on y
  let eq(x, y = b) = x == y;
  // without parameters, binds to a, b above
  let tmp = a && b;
  ...
  a1: assert property @(posedge clk) eq(p,q);
  always_comb begin
    a2: assert (eq(r)); // use default for y
    a3: assert (tmp);
  end
endmodule
```

The effective code after inlining of let definitions:

```
module m;
  bit clk, a, b;
  logic p, q, r;
  // let eq(x, y=b) = x == y;
  // let tmp = a && b;
  ...
  a1: assert property @(posedge clk) p == q;
```

```

    always_comb begin
        a2: assert (r == b); // use default for y
        a3: assert (a && b);
    end
endmodule

```

2) Declarative context binding of **let** arguments.

```

bit x = 1'b1;
let y = !x;
...
always_comb begin
    // redundant definition,
    // y binds to preceding definition
    bit x = 1'b0;
    a1: assert (a || y);
end

```

The effective code after inlining:

```

bit x = 1'b1;
// let y = !x;
...
always_comb begin
    // redundant definition,
    // y binds to preceding definition of x
    bit x = 1'b0;
    a1: assert (a || 1'b0);
end

```

3) Sequences (and properties) with **let** in structural context.

```

begin : top
    logic a, b;
    let x = a || b;
    sequence s;
        @clk x ##1 b;
    endsequence : s
    generate begin : mid
        logic a, b;
        ap: assert property(@clk s |-> a && b);
        ...
    end : mid
    endgenerate
end : top

```

After **let** and **sequence** inlining (full variable path names used to show exact binding):

```

begin : top
    logic a, b;
    // let x = a || b;
    // sequence s;
    // @clk top.a ##1 top.b;
    // endsequence : s
    generate begin : mid
        logic a, b;
        ap: assert property(@clk top.a || top.b |-> top.mid.a ##1 top.mid.b);
        ...
    end : mid
    endgenerate
end : top

```

4) Sequences (and properties) used in procedural context.

```
begin : top
  logic a, b;
  let x = a || b;
  sequence s;
    @clk x ##1 b;
  endsequence : s
  always @clk begin : mid
    logic a, b;
    ap: assert property(s.ended |-> x && b);
    ...
  end : mid
end : top
```

After let and sequence inlining, and clock inference:

```
begin : top
  logic a, b;
  // let x = a || b;
  sequence s;
    @clk (top.a || top.b) ##1 top.b;
  endsequence : s
  always @clk begin : mid
    logic a, b;
    ap: assert property(@clk s.ended |-> (top.a || top.b) && top.mid.b);
    ...
  end : mid
end : top
```

5) let declared in a generate statement.

```
module m(...);
  bit clk, a, b;
  bit [2:0] c;
  for (genvar i = 0; i < 3; i++) begin : L0
    if (i != 1) begin : L1
      let my_let(x) = !x || b && c[i];
      my_assert: assert property (@(posedge clk) my_let(a));
    end : L1
  end : L0
endmodule
```

This will resolve to the following equivalent code:

```
module m(...);
  bit clk, a, b;
  bit [2:0] c;
  begin : L0[0]
    begin : L1
      // let my_let(x) = !x || m.b && m.c[0];
      my_assert: assert property (
        @(posedge clk) !m.a || m.b && m.c[0]);
    end : L1
  end : L0[0]
  begin : L0[2]
    begin : L1
      // let my_let(x) = !x || m.b && m.c[2];
      my_assert: assert property (
        @(posedge clk) !m.a || m.b && m.c[2]);
    end : L1
  end : L0[2]
```

endmodule

Note that the **let** statements may not be referred to hierarchically using the paths `m.L0[0].L1` and `m.L0[2].L1`.

6) Import from a package shall follow the same rules as when importing functions from packages. Notice that the variable `z` in `my_let` refers to the declaration of `z` in the scope of the package. Since the function `my_fn` is also referred to in the imported `let` definition, neither the function nor the variable need to be imported as shown in module `m2`.

```
package pack;
  logic z;
  function bit my_fn(bit x); ...; endfunction
  let my_let(x, y) = x && my_fn(y) && z;
endpackage

module m1 (...);
  import pack::*;
  bit clk, a, b;
  // my_let(a, b) will expand into
  // m1.a && my_fn(m1.b) && z
  my_assert: assert property (@(posedge clk) my_let(a, b));
  ...
endmodule

module m2 (...);
  import pack::my_let;
  bit clk, a, b;
  // my_let(a, b) will expand into
  // m2.a && my_fn(m2.b)
  my_assert_1: assert property (@(posedge clk) my_let(a, b));

  // Illegal use age of my_fn as it is not known in m2
  my_assert_2: assert property (@(posedge clk) my_fn(b));
  ...
endmodule
```

7) Using sampled value functions.

```
begin : top
  logic a, b;
  let x = $past(a);
  sequence s;
    x ##1 $rose(b);
  endsequence : s
  generate
    begin : mid
      logic a, b;
      ap: assert property(@clk a |-> s);
      ...
    end : mid
  endgenerate
end : top
```

After **let** inlining:

```
begin : top
  logic a, b;
  // let x = $past(a);
  sequence s;
    $past(a) ##1 $rose(b); // no clock inferred yet
  endsequence : s
```

```

generate
  begin : mid
    logic a, b;
    ap: assert property(@clk a |-> s);
    ...
  end : mid
endgenerate
end : top

```

After sequence inlining and clock inference:

```

begin : top
  logic a, b;
  // let x = $past(a);
  sequence s;
    $past(a) ##1 $rose(b);
  endsequence : s
  generate
    begin : mid
      logic a, b;
      // clk used in $past and $rose
      ap: assert property(@clk top.mid.a |->
        $past(top.a) ##1 $rose(top.b));
      ...
    end : mid
  endgenerate
end : top

```

8) Typed formal arguments and named actual argument association. Type of *x* and *y* is **bit**, type of *z* is determined from the actual argument.

```

module m;
  logic a, b; int v;

  let eq(bit x, y = b, context z = 3) = (z == v) && (x == y);
  let tmp = a && b;
  ...
  sequence s(x);
    x ##1 b;
  endsequence
  property p;
    bit a;
    // default b used for y, 3 for z
    @(posedge clk) (1'b1, a=b) ##1 eq(.x(a) |=>
      (tmp == 0) ##0 s(a));

  endproperty : p
a1: assert property (p);
endmodule

```

The effective code after inlining:

```

module m;
  bit a, b; int v;
  //let eq(bit x, y = b, context z = 3) =
  //  (z == v) && (x == y);
  // let tmp = a && b;
  ...
  sequence s(x);
    x ##1 b;
  endsequence
  property p;
    bit a;
    @(posedge clk) (1'b1, m.p.a = bit'(m.b)) ##1

```

```

        ((3 == m.v) && (m.p.a == bit'(m.b)) | =>
         (m.a && m.b) == 0) ##0 (m.p.a ##1 m.b);
    endproperty : p
a1: assert property (p);
endmodule

```

9) Conflicting names.

```

module m;
    bit a, b;

    // Illegal: conflicting definition of a
    let a = !b;
    ...
endmodule

```

In Annex A

Modify A.2.10

from

```

concurrent_assertion_item_declaration ::=
    property_declaration
    | sequence_declaration

```

to

```

concurrent_assertion_item_declaration ::=
    property_declaration
    | sequence_declaration
    | let_declaration

```

from

```

sequence_actual_arg ::=
    event_expression

```

to

```

sequence_actual_arg ::=
    event_expression
    | let_instance

```

from


```

let let_identifier[ ( [let_port_list] ) ] = expression_or_dist;

let_identifier ::=
    identifier

let_port_list ::=
    let_port_item { , let_port_item }

let_port_item ::=
    { attribute_instance } let_formal_type identifier [= expression_or_dist]

let_formal_type ::=
    data_type_or_implicit
    | context

let_instance ::=
    let_identifier[ ( [let_list_of_arguments] ) ]

expression ::=
    ...
    | let_instance

let_list_of_arguments ::=
    [let_actual_arg] { , [let_actual_arg] } { , . identifier ([let_actual_arg]) }
    | . identifier ([let_actual_arg] ) { , . identifier ( [let_actual_arg] ) }

let_actual_arg ::=
    expression_or_dist

```

Add to Table B1—Reserved keywords

let

Annex M

Modify M.2

from

```

/* property decl, spec */
#define vpiPropertyDecl 655
#define vpiPropertySpec 656
#define vpiPropertyExpr 657

```

```
#define vpiMulticlockSequenceExpr 658
#define vpiClockedSeq 659
#define vpiPropertyInst 660
#define vpiSequenceDecl 661
#define vpiActualArgExpr 663
#define vpiSequenceInst 664
#define vpiImmediateAssert 665
#define vpiReturn 666
```

to

```
/* property decl, spec */
#define vpiPropertyDecl 655
#define vpiPropertySpec 656
#define vpiPropertyExpr 657
#define vpiMulticlockSequenceExpr 658
#define vpiClockedSeq 659
#define vpiPropertyInst 660
#define vpiSequenceDecl 661
#define vpiActualArgExpr 663
#define vpiSequenceInst 664
#define vpiImmediateAssert 665
#define vpiLetDecl Editor to fill
#define vpiLetInst Editor to fill
#define vpiReturn 666
```