The "let" expression - motivation

Properties and sequences serve as templates for concurrent assertions, but there is no such construct for immediate assertions. Therefore, it is not possible to create packages/libraries of assertion templates for both concurrent and immediate assertions. The proposed **let** construct thus provides such a template as one of its possible uses. For example,

```
// in a package
let at_least_two(sig, rst = 1'b0) = rst || ($countones(sig) >= 2);

// in a design
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`** expression cannot be substituted by a function since formal arguments of a function need to have a specific type. There would have to be different function definitions for different argument widths. The **`let`** expression does not require that the type of the arguments be specified, it is inferred from the context.

The only alternative to **`let`** is a compiler directive. However, 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`** expressions into packages (See Clause 25) is a natural way to implement a well-structured customization for assertions. For example, the above **`let`** construct is roughly equivalent to the following directive:

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

Another use for **`let`** is in modeling for assertions instead of introducing additional nets. 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);
```

A synthesis tool may treat the auxiliary nets such as `c` as real ones and synthesize them into silicon. This may not be desirable. Workarounds using `` `ifdef `` are needed for this purpose. Also, the exact type must be specified or the **`type`** operator must be used for this purpose, but it becomes awkward with long signal or expression names. 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 of bit width two then `c` will also have the bit width two, and the result will sometimes be truncated. Using the **`let`** expression makes writing more elegant and safe:

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

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

14.3 Clocking block declaration

REPLACE in *Syntax 14-1*

clocking_item ::=
default default_skew **;**
| clocking_direction list_of_clocking_decl_assign **;**
| { attribute_instance } concurrent_assertion_item_declaration

WITH

clocking_item ::=
default default_skew **;**
| clocking_direction list_of_clocking_decl_assign **;**
| { attribute_instance } ~~concurrent~~ assertion_item_declaration

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

---

assertion_item_declaration ::= // *from A.2.10*
…
| let_declaration
let_declaration ::=
let let_identifier[**(** [let_port_list] **)**] **=** expression**;**
let_identifier ::=
identifier
let_port_list ::=
let_port_item {**,** let_port_item}
let_port_item ::=
{ attribute_instance } let_formal_type identifier [**=** expression]
let_formal_type ::=
data_type_or_implicit
let_instance ::=
let_identifier[**(** [let_list_of_arguments] **)**]
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
expression ::= // *from A.8.3*
…
| let_instance[40]

Note to the editor: insert the first available reference number. Currently it is 40

---

40) *let_instance* may be used only in the expressions which are part of sequence or property expressions, in *property_spec*, in `let` declarations or in immediate assertions.

Note to the editor: insert the first available reference number. Currently it is 40.

Syntax 16-2—Declaring let syntax

Note to the editor: Shift the numeration of the subsequent tables accordingly.

A **let** declaration defines a template expression (a **let** expression), customized by its ports. A **let** declaration may be instantiated in sequence or property expressions, in *property_spec*, in other **let** declarations or in immediate assertions.

let declarations can be used for customization and can 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 binding by name as well as by position. Furthermore, including **let** declarayions into packages (see Clause 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
```

Similar to the way that properties and sequences serve as templates for concurrent assertions, the **let** construct can serve this purpose 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
```

In this case the let instantiation cannot be replaced by a function call 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. Another alternative to **let** is a compiler directive, but it has a global scope, as described earlier:

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

Another intended use of **let** is in modeling for assertions to provide shortcuts for identifiers or subexpressions. For example,

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

The formal arguments may optionally be typed, following a similar syntactic form as for arguments to sequences and properties. The argument types are restricted the same way as in boolean expressions used in assertions (see 16.5.1). The formal arguments may have optional default values.

Like in sequences and properties, variables used in a **let** that are not formal arguments to the **let** are resolved according to the scoping rules from the scope in which the **let** is declared.

In the scope of declaration, **let** expressions must be defined before used. No hierarchical references to **let** declarations are allowed.

The **let** expression is substituted in the place of instantiation without any partial evaluation of the expression. When a **let** is instantiated, actual arguments can be passed to the **let** expression. The **let** expression gets expanded with the actual arguments by replacing the formal arguments with the actual arguments. Semantic checks are performed to ensure that the expanded expression with the actual arguments is legal. The substituted expression is enclosed in parentheses (...) so as to preserve the priority of evaluation of the **let** expression. Recursive **let** instantiations are not permitted.

let expressions may contain sampled value function calls (see 16.8.3). Their clock, if not explicitly specified, is inferred in the instantiation context in the same way as if the functions were used directly in the instantiation context. It shall be an error, if the clock is required, but cannot be inferred in the instantiation context.

A **let** can be declared in any of the following:

— A module
— An interface
— A program
— A clocking block
— A package
— A compilation-unit scope
— A generate block

Examples:

a) **let** with arguments and without arguments.

```
module m;

    logic clk, a, b;
    logic p, q, r;

    // let 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 : m
```

The effective code after expanding **let** instances:

```
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) m.p == m.q);
```

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

b) Declarative context binding of **`let`** arguments

```
    module top;
      bit x = 1'b1;
      bit a;
      let y = x;
      ...
      always_comb begin
         // y binds to preceding definition of x
         // in the declarative context of let
          bit x = 1'b0;
          a1: assert (a || y);
      end
    endmodule : top
```

The effective code after expanding **`let`** instances:

```
    module top;
      bit x = 1'b1;
      bit a;
      // let y = x;
      ...
      always_comb begin
         // y binds to preceding definition of x
         // in the declarative context of let
          bit x = 1'b0;
          a1: assert (a || (top.x));
      end
    endmodule : top
```

c) Sequences (and properties) with **`let`** in structural context

```
    module top;
        logic a, b;
        let x = a || b;
        sequence s;
            x ##1 b;
        endsequence : s
        …
    endmodule : top
```

The effective code after expanding **`let`** instances:

```
    module top;
        logic a, b;
        // let x = a || b;
        sequence s;
            (top.a || top.b) ##1 b;
        endsequence : s
        …
    endmodule : top
```

d) **`let`** declared in a **`generate`** block

```
    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];
            a1: assert property (@(posedge clk) my_let(a)); //OK
        end : L1
    end : L0
    a2: assert property (@(posedge clk) L0[0].L1.my_let(a)); // Illegal
endmodule : m
```

Assertion `a1` after expansion becomes two assertions `L0[0].L1.a1` and `L0[2].L1.a1`, the first of them being

```
assert property (@(posedge clk) (!m.a || m.b && m.c[0]));
```

and the second one being

```
assert property (@(posedge clk) (!m.a || m.b && m.c[2]));
```

Assertion `a2` is illegal since it references the **`let`** expression hierarchically, while hierarchical references to **`let`** expressions are not allowed.

e) **`let`** with typed arguments

```
module m(bit clock);
    logic [15:0] a, b;
    logic c, d;
    typedef bit [15:0] bits;
    …
    let ones_match(bits x, y) = x == y;
    let same(logic x, y) = x === y;

    always_comb
        a1: assert(ones_match(a, b));

    property toggles(bit x, y)
        same(x, y) |=> !same(x, y);
    endproperty

    a2: assert property (@(posedge clk) toggles(c, d));
endmodule : m
```

In this example assertion the **`let`** expression `ones_match` checks that both arguments have bits set to 1 at the same position. Because of the explicit specification of the formal argument types in the **`let`** declaration, all argument bits having unknown logic value or a high-impedance value become 0, and therefore the comparison captures the match of the bits set to 1. The **`let`** expression `same` tests for the case equality (see 11.4.6) of its operands. When instantiated in the property `toggles` its actual arguments will be of type **`bit`**. The effective code after expanding **`let`** instances:

```
module m(...);
    logic [15:0] a, b;
    bit c, d;
    typedef bit [15:0] bits;
    …
    // let ones_match(bits x, y) = x == y;
    // let same(logic x, y) = x === y;

    always_comb
```

```
          a1:assert(bits'(a) == bits'(b));

      property toggles(bit x, y)
         (logic'(x) === logic'(y)) |=> ! (logic'(x) === logic'(y));
      endproperty

      a2: assert property (@(posedge clk) toggles(c, d));
  endmodule : m
```

f) Sampled value functions in `let`

```
  module m(logic clock);
      logic a;
      let p1(x) = $past(x);
      let p2(x) = $past(x,,,@(posedge clock));
      let s(x) = $sampled(x);
      always_comb begin
          a1: assert(p1(a));
          a2: assert(p2(a));
          a3: assert(s(a));
      end
      always @(posedge clk)
         a4: assert property(p1(a));
      …
  endmodule : m
```

The effective code after expanding `let` instances:

```
  module m(logic clock);
      logic a;
      // let p1(x) = $past(x);
      // let p2(x) = $past(x,,,@(posedge clock));
      // let s(x) = $sampled(x);
      always_comb begin
          a1: assert($past(a)); // Illegal: no clock can be inferred
          a2: assert($past(a,,,@(posedge clock)));
          a3: assert($sampled (a));
      end
      always @(posedge clk)
         a4: assert property($past(a)); // @(posedge clk) is inferred
      …
  endmodule : m
```

Note to the editor: the numeration of clauses and tables in the rest of this document is before shifting.

16.7 Declaring sequences

REPLACE in Syntax 16-4—Declaring sequence syntax

concurrent_assertion_item_declaration ::= *// from A.2.10*

...

| sequence_declaration

WITH

~~concurrent_~~assertion_item_declaration ::= *// from A.2.10*

...

| sequence_declaration

…

16.12 Declaring properties

REPLACE in *Syntax 16-14—Property construct syntax*

concurrent_assertion_item_declaration ::= *// from A.2.10*

property_declaration
...

WITH
~~concurrent_~~assertion_item_declaration ::= *// from A.2.10*

property_declaration
...

25.2 Package declarations

REPLACE in *Syntax 25-1—Package declaration syntax*

package_or_generate_item_declaration ::=
net_declaration
| data_declaration
| task_declaration
| function_declaration
| dpi_import_export
| extern_constraint_declaration
| class_declaration
| class_constructor_declaration
| parameter_declaration **;**
| local_parameter_declaration
| covergroup_declaration
| overload_declaration
| concurrent_assertion_item_declaration
| **;**

WITH
package_or_generate_item_declaration ::=
net_declaration
| data_declaration
| task_declaration
| function_declaration
| dpi_import_export
| extern_constraint_declaration
| class_declaration
| class_constructor_declaration
| parameter_declaration **;**
| local_parameter_declaration
| covergroup_declaration
| overload_declaration
| ~~concurrent_~~assertion_item_declaration
| **;**

A.1.10 Package items

REPLACE

```
package_or_generate_item_declaration ::=
        net_declaration
        | data_declaration
        | task_declaration
        | function_declaration
        | dpi_import_export
        | extern_constraint_declaration
        | class_declaration
        | class_constructor_declaration
        | parameter_declaration ;
        | local_parameter_declaration
        | covergroup_declaration
        | overload_declaration
        | concurrent_assertion_item_declaration
        | ;
```

WITH

```
package_or_generate_item_declaration ::=
        net_declaration
        | data_declaration
        | task_declaration
        | function_declaration
        | dpi_import_export
        | extern_constraint_declaration
        | class_declaration
        | class_constructor_declaration
        | parameter_declaration ;
        | local_parameter_declaration
        | covergroup_declaration
        | overload_declaration
        | ~~concurrent_~~assertion_item_declaration
        | ;
```

A.2.10 Package items

REPLACE

```
concurrent_assertion_item_declaration ::=
        property_declaration
        | sequence_declaration
```

WITH

```
~~concurrent_~~assertion_item_declaration ::=
        property_declaration
        | sequence_declaration
        | let_declaration
```

ADD at the end

let_declaration ::=
    let let_identifier[([let_port_list])] = expression;

let_identifier ::=
    identifier

let_port_list ::=
    let_port_item {, let_port_item}

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

let_formal_type ::=
    data_type_or_implicit

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

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

A.6.11 Clocking block

REPLACE

clocking_item ::=
    default default_skew ;
    | clocking_direction list_of_clocking_decl_assign ;
    | { attribute_instance } concurrent_assertion_item_declaration

WITH

clocking_item ::=
    default default_skew ;
    | clocking_direction list_of_clocking_decl_assign ;
    | { attribute_instance } ~~concurrent~~ assertion_item_declaration

A.2.10 Assertion declarations

ADD at the end

A.2.10 Expressions

REPLACE

expression ::=
primary
| unary_operator { attribute_instance } primary
| inc_or_dec_expression
| (operator_assignment)
| expression binary_operator { attribute_instance } expression
| conditional_expression
| inside_expression
| tagged_union_expression

WITH

expression ::=
primary
| unary_operator { attribute_instance } primary
| inc_or_dec_expression
| (operator_assignment)
| expression binary_operator { attribute_instance } expression
| conditional_expression
| inside_expression
| tagged_union_expression
| let_instance[40]

Note to the editor: insert the first available reference number. Currently it is 40.

A.10 Footnotes (normative)

ADD to the end

40) *let_instance* may be used only in the expressions which are part of sequence or property expressions, in *property_spec*, in **`let`** declarations or in immediate assertions.

Note to the editor: insert the first available reference number. Currently it is 40.

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
```

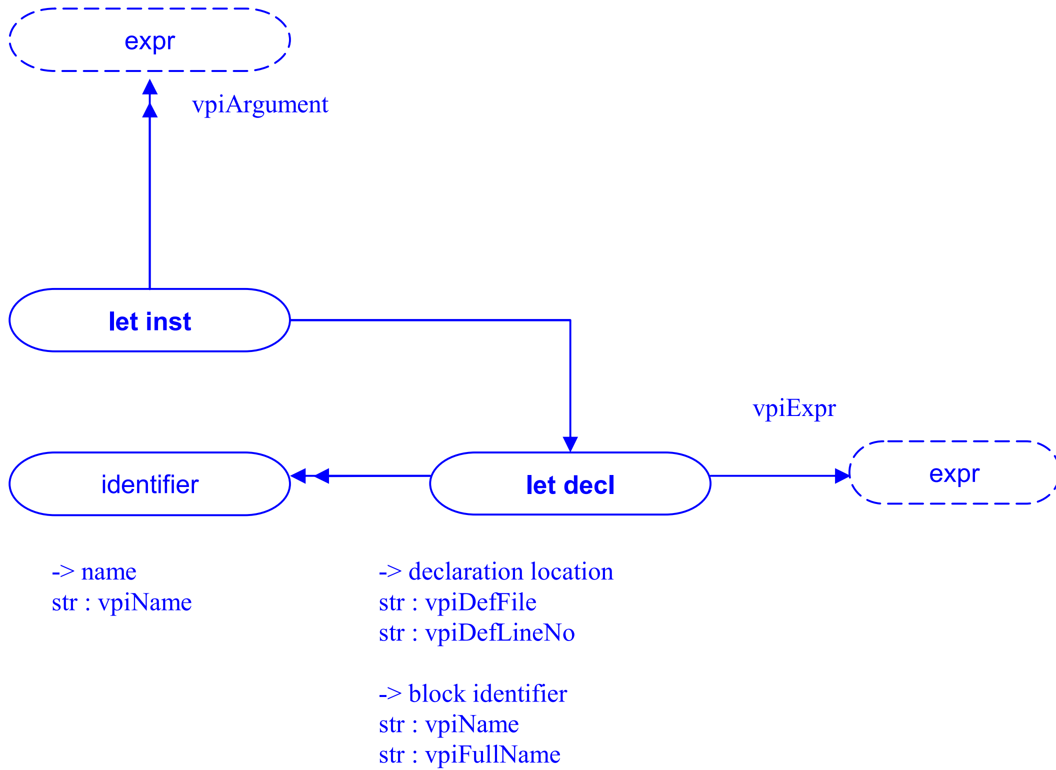
to

```
/* 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
```

Add to Clause 36

36.76 let declaration

Note to Editor: It is best to put this after 36.48 Multi-clock sequence, and renumber all that follows.



Details :

The vpiArgument iterator shall return the let instance arguments in the order that the formals for the let are declared, so that the correspondence between each argument and its respective formal can be made. If a formal has a default value, that value shall appear as the argument should the instantiation not provide a value for that argument.

36.50 Expressions

CHANGE TO