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

let_declaration ::= // *from A.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])]

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

  | let_instance

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. Furthermore, 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
```

As properties and sequences serve as templates for concurrent assertions, the **let** statement may 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
```

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 with the disadvantages as described earlier:

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

Another use or **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);
```

A synthesis tool may treat the auxiliary signals 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 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. **Let** cannot be instantiated in action blocks of verification statements. The operand types are restricted the same way as in Boolean expressions used in assertions (16.5.1).

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 one is to write the implicitly typed arguments at the beginning of the formal argument list, prior to any typed argument. The second one 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 substituted in the place of instantiation without any partial evaluation of the expression, the same way as sequences do. 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.

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 substitution 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 **let** expression substitution:

```
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.

```
module top;
    parameter int p = 0;
    logic a, b;
    let x = a || b;
    sequence s;
        x ##1 b;
    endsequence : s
    generate
       if (p != 0) begin : mid
          logic a, b;
          ap: assert property(@clk s |-> a && b);
          ...
       end : mid
    endgenerate
endmodule : top
```

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

```
module top;
    parameter int p = 0;
    logic a, b;
    // let x = a || b;
    // sequence s;
    //    @clk top.a ##1 top.b;
    // endsequence : s
    generate
        if (p != 0) begin : mid
           logic a, b;
           ap: assert property(@clk ((top.a || top.b) ##1 top.b)
                                              |-> top.mid.a && top.mid.b);
           ...
        end : mid
    endgenerate
```

```
endmodule : top
```

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

```
module 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
endmodule : top
```

After **let** and **sequence** substitution, and clock inference:

```
module 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 top.s.ended |-> (top.a || top.b) && top.mid.b);
            ...
    end : mid
endmodule : 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`. However, if the function or the variable were referred to directly in the code, it would have to be imported explicitly. Therefore, since `m2` does not contain the import of `my_fn` from the package `pack`, the reference to `my_fn` in `my_assert_2` is illegal.

```
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 && pack::my_fn(m1.b) && pack::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 && pack::my_fn(m2.b) && pack::z)
    my_assert_1: assert property (@(posedge clk) my_let(a, b));

    // Illegal usage 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.

```
module top;
    parameter int p = 0;
    logic a, b;
    let x = $past(a);
    sequence s;
        x ##1 $rose(b);
    endsequence : s
    generate
        if (p !=0) begin : mid
            logic a, b;
            ap: assert property(@clk a |-> s);
            ...
        end : mid
    endgenerate
endmodule : top
```

After **let** substitution:

```
module top;
    parameter int p = 0;
    logic a, b;
    // let x = $past(a);
    sequence s;
        ($past(a)) ##1 $rose(b); // no clock inferred yet
    endsequence : s
    generate
        if (p !=0) begin : mid
            logic a, b;
             ap: assert property(@clk a |-> s);
            ...
        end : mid
    endgenerate
endmodule : top
```

After sequence substitution and clock inference:

```
module top;
    parameter int p = 0;
    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,,,@clk)) ##1 $rose(top.b, @clk)));
            ...
        end : mid
    endgenerate
endmodule : top
```

8) Typed formal arguments and named actual argument association. The type of `x` and `y` is **bit**, the 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 substitution:

```
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, a /*local variable in p*/ = bit'(m.b)) ##1
                       ((3 == m.v) && (a == bit'(m.b))
                            |=>
                                 ((m.a && m.b) == 0) ##0
                                 (a /* local variable in p*/ ##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 newly numbered 16.7 Sequences

Change in Syntax 16-2—Sequence syntax

from

sequence_expr ::=
 cycle_delay_range sequence_expr { cycle_delay_range sequence_expr }
 | sequence_expr cycle_delay_range sequence_expr { cycle_delay_range sequence_expr }
 | expression_or_dist [boolean_abbrev]
 | sequence_instance [sequence_abbrev]
 | **(** sequence_expr {**,** sequence_match_item } **)** [sequence_abbrev]
 | sequence_expr **and** sequence_expr
 | sequence_expr **intersect** sequence_expr
 | sequence_expr **or** sequence_expr
 | **first_match (** sequence_expr {**,** sequence_match_item} **)**
 | expression_or_dist **throughout** sequence_expr
 | sequence_expr **within** sequence_expr
 | clocking_event sequence_expr
cycle_delay_range ::=
 ## integral_number
 | **##** identifier
 | **## (** constant_expression **)**
 | **## [** cycle_delay_const_range_expression **]**
sequence_match_item ::=
 operator_assignment
 | inc_or_dec_expression

| subroutine_call
sequence_instance ::=
ps_sequence_identifier [([sequence_list_of_arguments])]
sequence_list_of_arguments ::=
[sequence_actual_arg] { , [sequence_actual_arg] } { , . identifier ([sequence_actual_arg]) }
| . identifier ([sequence_actual_arg]) { , . identifier ([sequence_actual_arg]) }
sequence_actual_arg ::=
event_expression
boolean_abbrev ::=
consecutive_repetition
| non_consecutive_repetition
| goto_repetition
sequence_abbrev ::= consecutive_repetition
consecutive_repetition ::= [* const_or_range_expression]
non_consecutive_repetition ::= [= const_or_range_expression]
goto_repetition ::= [-> const_or_range_expression]
const_or_range_expression ::=
constant_expression
| cycle_delay_const_range_expression
cycle_delay_const_range_expression ::=
constant_expression : constant_expression
| constant_expression : $
expression_or_dist ::= expression [**dist** { dist_list }]
expression_or_dist_or_let ::=
expression_or_dist

to

sequence_expr ::=
cycle_delay_range sequence_expr { cycle_delay_range sequence_expr }
| sequence_expr cycle_delay_range sequence_expr { cycle_delay_range sequence_expr }
| expression_or_dist_or_let [boolean_abbrev]
| sequence_instance [sequence_abbrev]
| (sequence_expr {, sequence_match_item }) [sequence_abbrev]
| sequence_expr **and** sequence_expr
| sequence_expr **intersect** sequence_expr
| sequence_expr **or** sequence_expr
| **first_match** (sequence_expr {, sequence_match_item})
| expression_or_dist_or_let **throughout** sequence_expr
| sequence_expr **within** sequence_expr
| clocking_event sequence_expr
cycle_delay_range ::=
integral_number
| ## identifier
| ## (constant_expression)
| ## [cycle_delay_const_range_expression]
sequence_match_item ::=
operator_assignment
| inc_or_dec_expression
| subroutine_call
sequence_instance ::=
ps_sequence_identifier [([sequence_list_of_arguments])]
sequence_list_of_arguments ::=

[sequence_actual_arg] { , [sequence_actual_arg] } { , . identifier ([sequence_actual_arg]) }
| . identifier ([sequence_actual_arg]) { , . identifier ([sequence_actual_arg]) }

sequence_actual_arg ::=
event_expression
| let_instance

boolean_abbrev ::=
consecutive_repetition
| non_consecutive_repetition
| goto_repetition

sequence_abbrev ::= consecutive_repetition

consecutive_repetition ::= [* const_or_range_expression]

non_consecutive_repetition ::= [= const_or_range_expression]

goto_repetition ::= [-> const_or_range_expression]

const_or_range_expression ::=
constant_expression
| cycle_delay_const_range_expression

cycle_delay_const_range_expression ::=
constant_expression : constant_expression
| constant_expression : $

expression_or_dist ::= expression [**dist** { dist_list }]

expression_or_dist_or_let ::=
expression_or_dist
| let_instance

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

sequence_expr ::=
cycle_delay_range sequence_expr { cycle_delay_range sequence_expr }
| sequence_expr cycle_delay_range sequence_expr { cycle_delay_range sequence_expr }
| expression_or_dist [boolean_abbrev]
| sequence_instance [sequence_abbrev]
| (sequence_expr {, sequence_match_item }) [sequence_abbrev]
| sequence_expr **and** sequence_expr
| sequence_expr **intersect** sequence_expr
| sequence_expr **or** sequence_expr
| **first_match** (sequence_expr {, sequence_match_item})
| expression_or_dist **throughout** sequence_expr
| sequence_expr **within** sequence_expr
| clocking_event sequence_expr
cycle_delay_range ::=
integral_number
| ## identifier
| ## (constant_expression)
| ## [cycle_delay_const_range_expression]
sequence_match_item ::=
operator_assignment
| inc_or_dec_expression
| subroutine_call
sequence_instance ::=
ps_sequence_identifier [([sequence_list_of_arguments])]
sequence_list_of_arguments ::=
[sequence_actual_arg] { , [sequence_actual_arg] } { , . identifier ([sequence_actual_arg]) }
| . identifier ([sequence_actual_arg]) { , . identifier ([sequence_actual_arg]) }
sequence_actual_arg ::=
event_expression
boolean_abbrev ::=
consecutive_repetition
| non_consecutive_repetition
| goto_repetition
sequence_abbrev ::= consecutive_repetition
consecutive_repetition ::= [* const_or_range_expression]
non_consecutive_repetition ::= [= const_or_range_expression]
goto_repetition ::= [-> const_or_range_expression]
const_or_range_expression ::=
constant_expression
| cycle_delay_const_range_expression
cycle_delay_const_range_expression ::=
constant_expression : constant_expression
| constant_expression : $
expression_or_dist ::= expression [**dist** { dist_list }]

to

```
sequence_expr ::=
        cycle_delay_range sequence_expr { cycle_delay_range sequence_expr }
      | sequence_expr cycle_delay_range sequence_expr { cycle_delay_range sequence_expr }
      | expression_or_dist_or_let [ boolean_abbrev ]
      | sequence_instance [ sequence_abbrev ]
      | ( sequence_expr {, sequence_match_item } ) [ sequence_abbrev ]
      | sequence_expr and sequence_expr
      | sequence_expr intersect sequence_expr
      | sequence_expr or sequence_expr
      | first_match ( sequence_expr {, sequence_match_item} )
      | expression_or_dist_or_let throughout sequence_expr
      | sequence_expr within sequence_expr
      | clocking_event sequence_expr
cycle_delay_range ::=
        ## integral_number
      | ## identifier
      | ## ( constant_expression )
      | ## [ cycle_delay_const_range_expression ]
sequence_match_item ::=
        operator_assignment
      | inc_or_dec_expression
      | subroutine_call
sequence_instance ::=
        ps_sequence_identifier [ ( [ sequence_list_of_arguments ] ) ]
sequence_list_of_arguments ::=
        [sequence_actual_arg] { , [sequence_actual_arg] } { , . identifier ( [sequence_actual_arg] ) }
      | . identifier ( [sequence_actual_arg] ) { , . identifier ( [sequence_actual_arg] ) }
sequence_actual_arg ::=
        event_expression
      | let_instance
boolean_abbrev ::=
        consecutive_repetition
      | non_consecutive_repetition
      | goto_repetition
sequence_abbrev ::= consecutive_repetition
consecutive_repetition ::= [* const_or_range_expression ]
non_consecutive_repetition ::= [= const_or_range_expression ]
goto_repetition ::= [-> const_or_range_expression ]
const_or_range_expression ::=
        constant_expression
      | cycle_delay_const_range_expression
cycle_delay_const_range_expression ::=
        constant_expression : constant_expression
      | constant_expression : $
expression_or_dist ::= expression [ dist { dist_list } ]
expression_or_dist_or_let ::=
        expression_or_dist
      | let_instance
```

from

```
sequence_match_item ::=
        operator_assignment
```

| inc_or_dec_expression
| subroutine_call

to

sequence_match_item ::=
~~operator_assignment~~
local_var_assignment
| inc_or_dec_expression
| subroutine_call
local_var_assignment ::=
variable_lvalue assignment_operator expression_or_let
expression_or_let ::=
epxression
| let_instance
expression_or_dist_or_let ::=
expression_or_dist
| let_instance

Add in A.2.10

let_declaration ::=
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])]

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
| let_instance

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
```

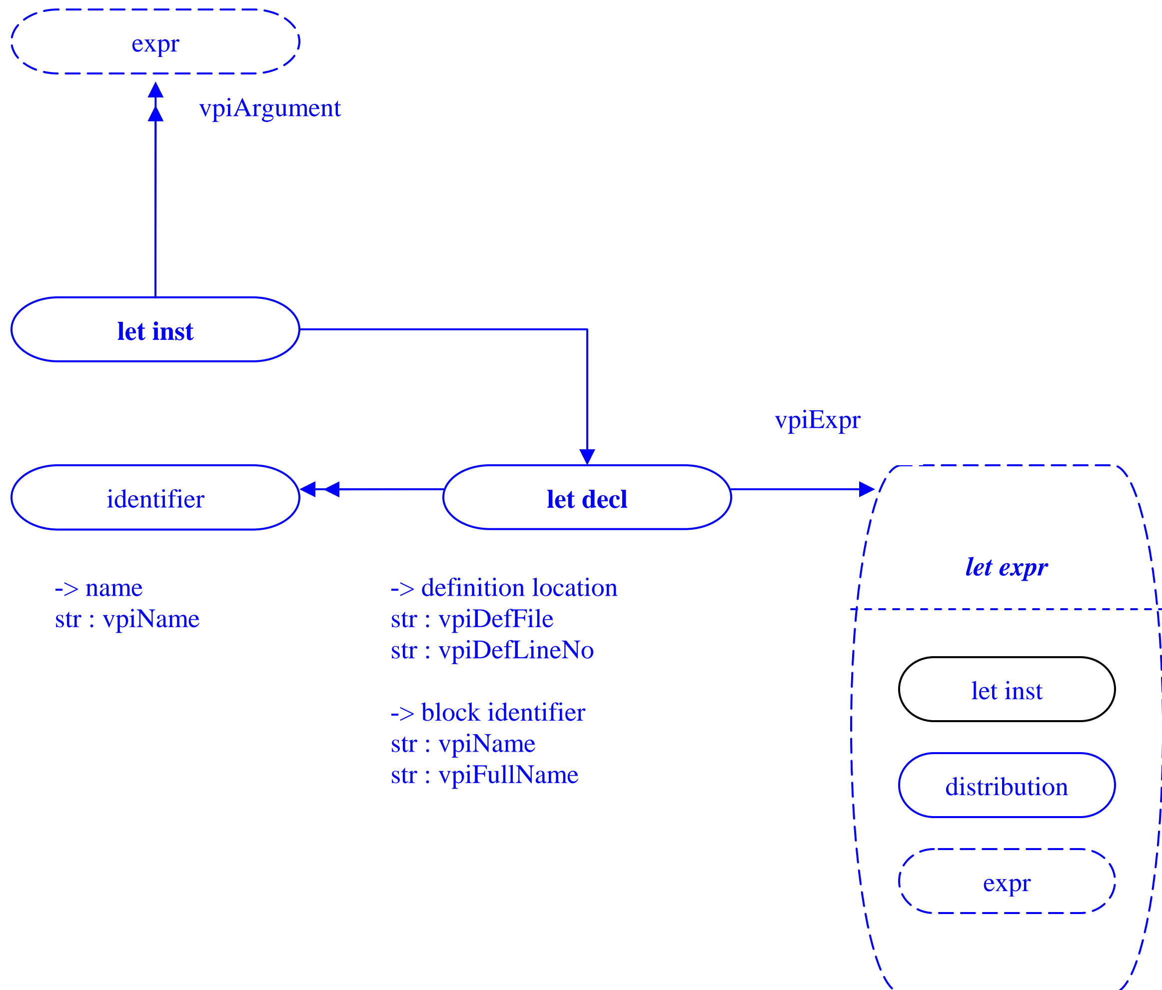
to

```
/* 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

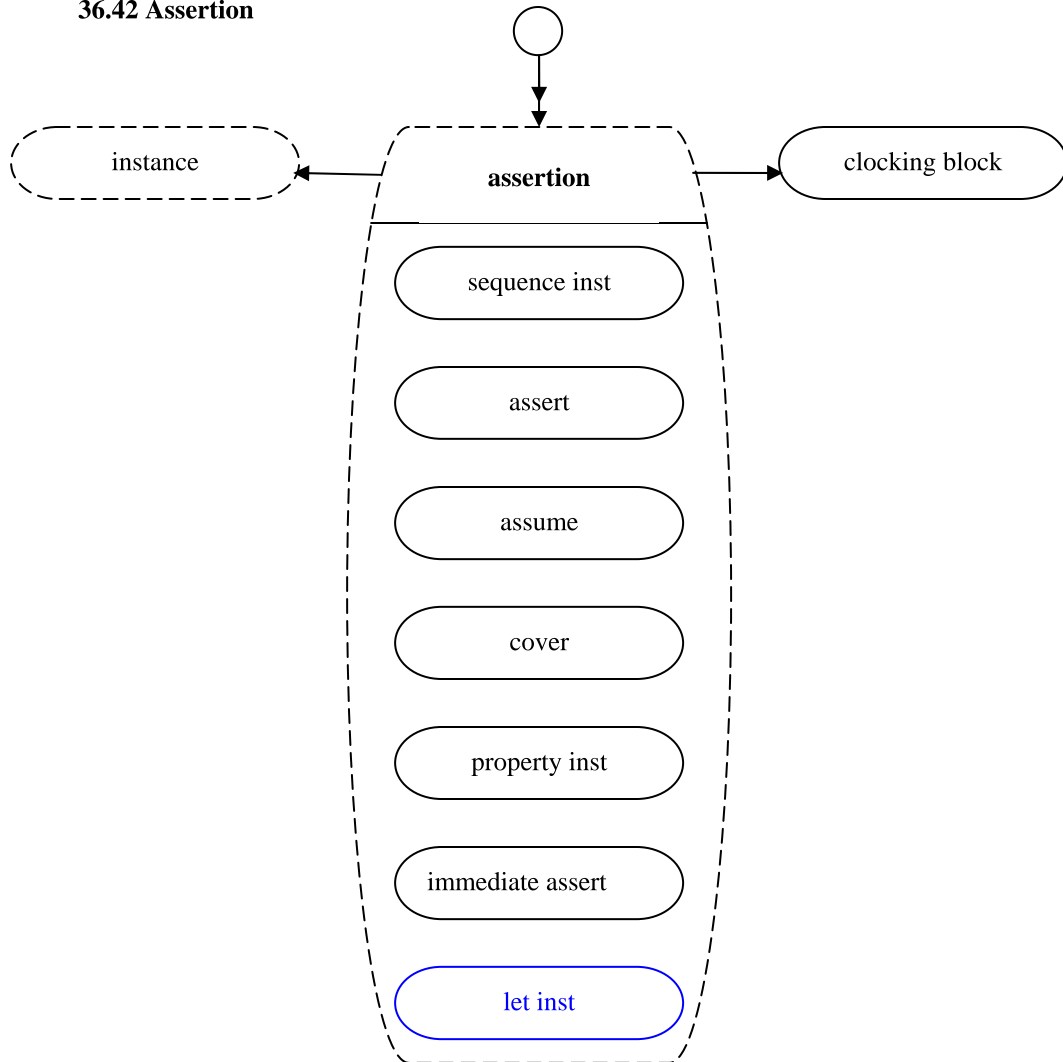
Note to Editor: It is best to put this after 36.48 Multi-clock sequence, and renumber all that follows.



Details :

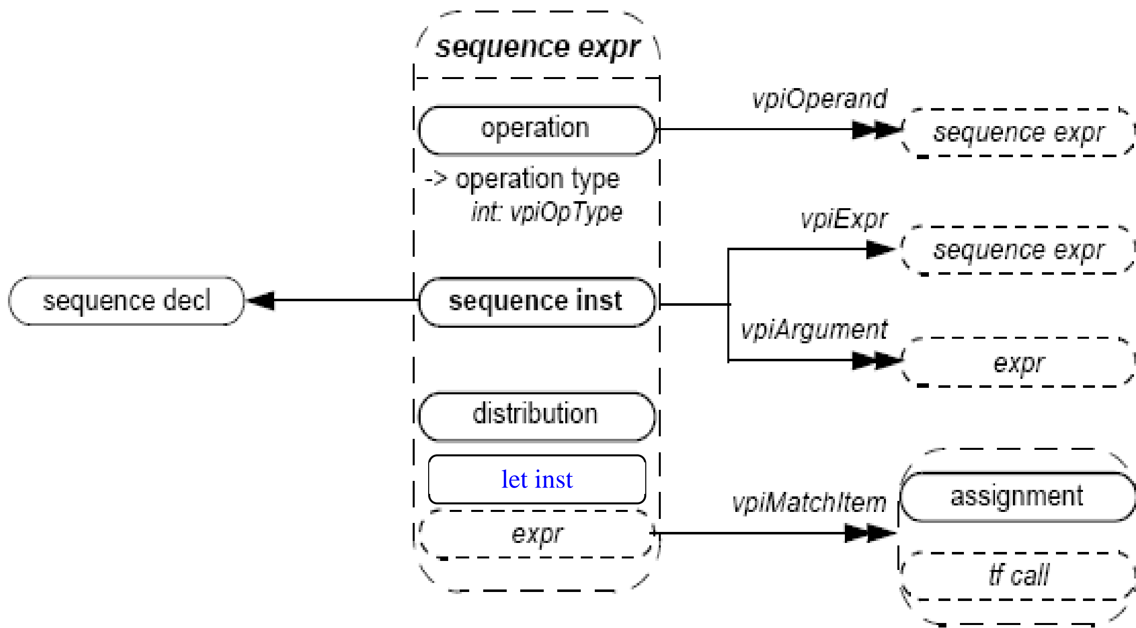
The vpiArgument iterator shall return the let instance arguments in the order that the formals for the let are declared, so that the correspondence between each argument and its respective formal can be made. If a formal has a default value, that value shall appear as the argument should the instantiation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

36.42 Assertion



-> assertion type
    Int : vpiTpe
-> location
    str : vpiFile
    int : vpiStartLine
    int : vpiColumn
    int : vpiEndLine
    int : vpiEndColumn
-> assertion name
    str : vpiName

36.47 Sequence expression


sequence expr
operation
-> operation type
int: vpiOpType
sequence inst
distribution
let inst
expr
vpiOperand
sequence expr
vpiExpr
sequence expr
vpiArgument
expr
sequence decl
vpiMatchItem
assignment
tf call