The "let" construct

Motivation

This proposal introduces a *let* statement: a meta-language construct used to name a parameterized expression.

There are several reasons to introduce this construct:

1. To provide a safer alternative to compiler directives
2. To provide a template for immediate assertions
3. To assign names to expressions for assertion modeling
4. To enable processing argument lists of arbitrary length

Let statement is very similar to *sequence* and *property* statements with the following difference:

- *let* defines an expression, while *sequence* defines a sequence expression and *property* defines a property expression
- *let* instantiation may be used in an arbitrary expression, while *sequence* and *property* instantiations may be used only where sequence and property expressions (correspondingly) are allowed.
-

Alternative to compiler directives

Let may be used for customization, and it may replace the compiler directives in many cases. *Let* construct is safer because it has a local scope, while the scope of compiler directives is global. *Let* is also more flexible, since it allows defining default argument values and argument passing both by order and by name. Including *let* statements into packages is a natural way to implement a well-structured customization. E.g.,

```
package pack;
    let past_and(x, y) = $past(x && y);
    …
endpackage

module m;
    import pack;
    logic a, b;
    …
    p: assert property(@(posedge clk) en |-> past_and
(a, b));
    …
endmodule
```

Template for immediate assertions

Concurrent assertions have templates (properties, sequences), but immediate assertions have none. I.e., the common functionality may be factored out from concurrent assertions:

```
property triggers(ante, cons, clk = proj_clk, rst =
1'b0);
      @(posedge clk) disable iff(rst) ante |-> cons;
endproperty

p1: assert property(triggers(a1, c1));
p2: assert property(triggers(a2, c2));
```

but immediate assertions should be written each time from scratch

```
q1: assert(rst || $onehot(~sig1));
q2: assert(rst || $onehot(~sig2));
```

Let statement solves this problem:

```
let one_zero(sig, rst = 1'b0) = rst || $onehot(~sig);
q1: assert (one_zero(~sig1));
q2: assert (one_zero(~sig2));
```

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 we would have to define different functions for different argument widths. The only alternative to *let* is a compiler directive:

```
`define one_zero(sig, rst) ((rst) || $onehot(~(sig)))
```

which is a worse solution since 1) it has a global scope, and 2) the information about assertion template is not available to the compiler: collecting statistics on assertion usage becomes infeasible, linting is problematic, etc.

Assertion modeling

One needs sometimes to perform auxiliary computations to write a checker. Usually SV RTL level is used for this purpose. E.g.,

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

Such practice is problematic for several reasons. One of them is that a synthesis tool will treat such auxiliary signals (`c`) as real ones and will synthesize them into silicon. Workarounds using `` `ifdef `` are needed for this purpose. Another reason is that an exact type must be specified, which is not user friendly. **`type`** operator may be used for this purpose, but it looks clumsy when the signal names are long. Even worse, the **`type`** operator represents the self-determined type of the result while the user needs the context-

determined type. Thus, if both `a` and `b` are two bit long then `c` will also two-bit long, 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);
```

Note again that using functions may solve a synthesis problem, but not a usability problem (the function requires a type specification and the full context for sampled value functions such as `$past`).

Processing argument lists of arbitrary length

Let statement is useful to process argument lists, since combining *let* statements with generate (or other similar) constructs is a natural way to handle multiple arguments. Multiple argument processing is a subject of another enhancement proposal, therefore we illustrate the idea here for a plain array:

```
int args[50:1];
for (genvar i = 0; i <= 50; i++) begin : b1
     if (i == 0) begin : b2
        let res = '0;
     end
     else begin : b2
     let res = b1[i-1].res + args[i];
     end
 end
 let res = b1[50].b2.res;
```

Proposal

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

The purpose is to allow metalanguage level parameterized definitions of symbols that have the following characteristics:

1. The *let* statements can define parameterized *expressions* that can be instantiated in both assertions and procedural code.[1]

2. As in Mantis items 928 and 1601 for 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

[1] To define property or sequence expressions use existing sequence and property declarations.

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 though 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. In the scope of declaration, *let* must be defined before used.

6. 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.

7. The *let* expression can be referenced by hierarchical name, consistent with the SystemVerilog naming conventions.

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 in sequence or property definitions.

9. *Let* statement name should be unique in its namespace as any other name in the namespace.

10. *Let* statements can be referenced using a hierarchical reference in another scope, but as mentioned above, any names that are not formal arguments will be bound in the declarative scope of the *let* statement.

11. Like *sequence* and *property* declarations, *let* statements can be placed in generated blocks (conditional, loops) and referenced as any other statement.

12. *Let* statements can be declared in interfaces and accessed by a port reference to the interface or modport in a module.

13. *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. An individual *let* statement imported from a package does not automatically make the names of its left hand side visible in the context of its instantiation.

Examples:

1) *Let* with arguments and without.

```
module m;
    bit a, b;

    // 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;

    sequence s(x);
        x ##1 b;
    endsequence

    property p;
       bit a;
       // default value b used for y
       @(posedge clk) (1, a=0) ##0 eq(a) |=>
          (tmp == 0) ##0 s(a);
    endproperty : p

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

The effective code after inlining:

```
module m;
    bit a, b;
    // let eq(x, y=b) = x == y;
    // let tmp = a && b;
    sequence s(x);
        x ##1 b;
    endsequence
    property p;
        bit a;
        @(posedge clk) a == m.b |=>
            (m.a && m.b == 0) ##0 (a ##1 m.b);
    endproperty : p
    a1: assert property (p);
endmodule
```

2) Declarative context binding of *let* arguments.

```
let x = 1'b1;
let y = !x;
```

```
bit z = y;
...
begin
    // redundant definition,
    // y binds to preceding definition
    let x = 1'b0;
    a <= y;
end
```

The effective code after inlining:

```
// let x = 1'b1;
// let y = !x;
bit z = !1'b1;
...
begin
    // redundant definition,
    // y binds to preceding definition
    let x = 1'b0;
    a <= !1'b1;
end
```

3) Sequences (and properties) in structural context.

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

After *let* inlining

```
begin : top
    logic a, b;
    // let x = a;
    sequence s;
        @clk a ##1 b;
    endsequence : s
    generate begin : mid
```

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

After sequence inlining (full variable path names used to show exact binding)

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

5) Sequences (and properties) used in procedural context

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

After let inlining and clock inference

```
    begin : top
        logic a, b;
        // let x = a;
        sequence s;
            @clk a ##1 b;
        endsequence : s
```

```
    always @clk begin : mid
        logic a, b;
        ap: assert property(@clk s.ended |-> x);
         ...
    end : mid
end : top
```

After sequence inlining

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

6) Let declared in a scope and referenced using a hierarchical reference.

```
module m0(...);
    bit a, b;
    let my_let(x) = !x || a && b;
    ...
endmodule
module m(....);
    bit clk;
    reg a, b, c;
    my_assert:
        assert property (@(posedge clk) m0.my_let(c));
    ...
    endmodule
```

After *let* expansion it will become

```
module m0(...);
    bit a, b;
    // let my_let(x) = !x || a && b;
    ...
endmodule
module m(....);
    bit clk;
    reg a, b, c;
```

```
    // the let expression refers to a, b from m0 scope
    my_assert: assert property (
        @(posedge clk) !c || m0.a && m0.b);
    ...
    endmodule
```

7) 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
```

The *let* statements can also be referred to hierarchically using the paths `m.L0[0].L1` and `m.L0[2].L1`. For example, `L0[0].L1.my_let(c)` used inside `m` will expand to `m.c || m.b && m.c`

8) Reference to a let statement in an interface

```
interface itf;
    logic a, b;
    let my_let = a && b; // a and b resolve within itf
    wire c = my_let;
    modport mp1(input a, output b, c);
    modport mp2(output a, input b, c);
endinterface

// cannot access via modport,
// must be the interface type
module m(itf bus);
    bit clk, a, b;
    // bus.my_let will expand into bus.a && bus.b
    my_assert:
        assert property (@(posedge clk) bus.my_let);
    ...
endmodule
```

9) Import from a package should follow the same rules as when importing functions from packages.

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

module m1 (…);
    import pack :: *;
    bit clk, a, b;
    // my_let(a, b) will expand into
    // m1.a && my_fn(m1.b)
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
    // m1.a && my_fn(m1.b)
my_assert_1:
    assert property (@(posedge clk) my_let(a, b));

    // Illegal: my_fn is not known in m2
my_assert_2:
    assert property (@(posedge clk) my_fn(b));
...
```

```
endmodule
```

10) 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;
        // No clock yet, a and b bound
        $past(a) ##1 $rose(b);
    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); // No clock inferred
    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
```

11) 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;
    bit a, b; int v;

    // default on y and z
    let eq(bit x, y = b, context z = 3) =
        (z == v) && (x == y);

    // without parameters, binds to a, b above
    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) 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;

    // default on y and z
    //let eq(bit x, y = b, context z = 3) =
    //    (z == v) && (x == y);

    // without parameters, binds to a, b above
    // let tmp = a && b;
    ...
```

```
        sequence s(x);
            x ##1 b;
        endsequence
        property p;
            bit a;
            @(posedge clk) ((3 == m.v) && (m.a == m.b) |=>
                ((m.a && m.b) ==0) ##0 (a ##1 m.b);
        endproperty : p
    a1: assert property (p);
    endmodule
```

12) Conflicting names

```
    module m;
        bit a, b;

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

Syntax

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

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
| **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
```

Notes

1. The *let* statement is expanded in the instantiation context in a similar way as a sequence declaration is, variables that are not formal arguments are bound in the scope of the let definition (declarative context.)

2. A *let* instance can appear wherever *expression* is allowed.