The "let" construct

02/20/2007

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

The purpose is to allow macro-like parameterized definitions of symbols that have the following characteristics:

1. The *let* statements can define parameterized *expressions* (like macros) 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 separated list of arguments.

3. The supported data types for *let* formal arguments are the types that are allowed for operands in assertion Boolean expressions (see 17.4.1). 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. References to identifiers / names in the declarative scope of the *let* statement are replaced by hierarchical references if different from the instantiation scope. This is to avoid clashes with local declarations. Recursive *let* instantiations are not permitted.

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

7. Let expressions can contain sampled value function calls ($rose, $fell, $past, $stable). 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.

8. *Let* expressions are checked for valid syntax in both the definition and the instantiation context.

9. Individual instances of *let* expressions can be viewed as variables in debug environment provided the instance is not dependent on the instantiation context such as bit width is not self-determined or the *let* statement contains formal arguments.

10. *Let* statements an 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 an 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.

14. *Let* statements are not synthesized by DC, i.e., *let* is not part of design code, but the result of its expansion in the instantiation context is synthesized (provided that the code is synthesizable).

15. If assertions are compiled into RTL for emulation, the result of inlining *let* constructs is compiled as well.

Examples:

1) *let* with arguments and without

```
module m;
   bit a, b;
   let eq(x, y=b) = x == y; // with formal arguments and default value on y
   let tmp = a && b; // without parameters, binds to a, b above
   sequence s(x);;
         x ##1 b;
   endsequence
   property p;
         bit a;
         @(posedge clk) eq(a) |=> (tmp==0) ##0 s(a); // default value b used for y
```

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

This is equivalent to:

```
module m;
    bit a, b;
    // let eq(x, y=b) = x == y;
    // let tmp = a && b;
    var type(a && b) let_tmp_4; assign let_tmp_4 = a && b; // debug var for tmp
    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
  let x = 1'b0; // redundant definition, y binds to preceding definition
  a <= y;
end
```

This is equivalent to:

```
// let x = 1'b1;
var type(1'b1) let_x_1; assign let_x_1 = 1'b1;
// let y = !1'b1;
var type(1'b1) let_y_2; assign let_y_2 = !1'b1;
bit z = ! 1'b1;
...
begin
  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
```

This is equivalent to:

```
begin : top
      logic a, b;
      // let x = a;
      var type(a) let_x_3; assign let_x_3 = a; // debug variable
      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 expansion (full variable path names used to show exact binding)

```
begin : top
      logic a, b;
      // let x = a;
      var type(a) let_x_3; let_x_3 = a;
      sequence s; // form that runs in s.ended
                @clk  top.a ##1 top.b;
      endsequence : s
      generate
        begin : mid
          logic a, b;
          // both s.ended and explicit s use top declarations
```

```
        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 clock inference this is equivalent to:

```
begin : top
    logic a, b;
    // let x = a;
    var type(top.a) let_x_3; assign let_x_3 = top.a;
    sequence s;
      @clk a ##1 b;
    endsequence : s
    always @clk begin : mid
       logic a, b;
       ap: assert property(@clk s.ended |-> top.a);
       ...
    end : mid
end : top
```

This is equivalent to

```
 begin : top
    logic a, b;
    // let x = a;
    var type(a) let_x_3; assign let_x_3 = a;
    sequence s; // form that runs in s.ended
              @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
```

This is equivalent to:

```
    module m0(...);
        bit a, b;
        let my_let(x) = !x || m0.a && m0.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
    end
    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
    end
  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
  module m(itf bus); // cannot access via modport, must be the interface type
    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
```

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 s.ended |-> s);
          ...
        end : mid
      endgenerate
  end : top
```

This is equivalent to:

```
    begin : top
      logic a, b;
      // let x = $past(a);
      sequence s;
        $past(a) ##1 $rose(b); // No clock yet, a and b bound
      endsequence : s
      generate
        begin : mid
          logic a, b;
          ap: assert property(@clk a |-> s);
          ...
        end : mid
      endgenerate
  end : top
```

After clock inference this is equivalent to:

```
 begin : top
    logic a, b;
    // let x = $past(a);
    // There is no debug variable since $past depends on context for its clock
    sequence s;
      $past(a) ##1 $rose(b); // No clock inferred
    endsequence : s
    generate
     begin : mid
      logic a, b;
      ap: assert property(@clk top.mid.a |->
                    $past(top.a) ##1 $rose(top.b)); // clk used in $past and $rose
      ...
     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;
   let eq(bit x, y=b, context z=3) = (z == v) &&(x == y); // default on y and z
   let tmp = a && b; // without parameters, binds to a, b above
   ...
   sequence s(x);;
         x ##1 b;
   endsequence
   property p;
         bit a;
         @(posedge clk) eq(.x(a))
                  |=>
                  (tmp==0) ##0 s(a); // default b used for y , 3 for z
   endproperty : p
   a1: assert property (p);
endmodule
```

This is equivalent to:

```
module m;
   bit a, b;
   // let eq(bit x, y=b, context z=3) = (z == v) &&(x == y);
   // let tmp = a && b;
   var type(a && b) let_tmp_4; assign let_tmp_4 = a && b; // debug var for tmp
```

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

Syntax

```
let_declaration ::=
        let let_identifier[ ( [let_port_list] ) ] = let_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_expr ::=
        expression

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 ::=
        let_instance
    |   expression
```

Notes

1. The *let* statement may be 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.

3. Basic syntax checking is done on the definition itself, that it is a valid expression (e.g., that there is no always statement in *let_expr*.)

4. If there are syntactic errors caused by wrong instantiation then an error message relative to the instantiation context should be issued, containing reference to the corresponding *let* definition.
   For example,

   ```
   let concat(x, y) = {x , y};
   ...
   always concat(b, c); // erroneous instance
   ```

   - A debug variable is not created for *let* statements that do not have self-determined bit width or contain formal arguments.
   - Naming the additional let variables for debugging: It may consist of the following concatenation of strings:
     let_<let_identifier>_<line_number_of_context>
     There is only one variable declaration required for each let statement.
     If there are more than one let statement declared on the same line, a *<occurrence_number>* may be appended to the variable name. The number is the occurrence index of the let statement on the line. The name then has the following form:
     let_<let_identifier>_<line_number_of_context>_<occurrence_number>