1.1 Non-blocking assignment (NBA) for assertions

This proposal addresses the need for allowing a non-blocking assignment that only takes place at the occurrence of a clock tick. In Verilog, one can write

```
always @(posedge clk)
   reg1 <= a & b;
```

However, the expression on the RHS of the assignment cannot contain any temporal functions such ended and $rose. This prevents supplementary modeling that is often required for assertions.

The proposal is to extend the syntax of clocking domains to allow NBA assignments. Since all operations in a clocking domain are with respect to a clock, it causes no difficulty in using any temporal function that is allowed in assertions. Properties and sequences can be defined in a clocking domain. In addition, the effect of an assignment is that the assigned value to the variable is only available at the next clock tick in any property using the variable.

Although it is not included in this proposal, ability to replicate and conditionally replicate NBA assignments, properties and sequences would simplify writing complex and re-usable assertions. Such capability already exists in modules and interfaces with generate statements. The recommendation is to allow generate statements within clocking domain and program.

1.1.1 Syntax

clocking_decl ::= [**default**] **clocking** [clocking_identifier] clocking_event **;** // from Annex A.6.11
{ clocking_item }
endclocking
clocking_event ::=
@ identifier
| **@ (** event_expression **)**
clocking_item :=
default default_skew **;**
| clocking_direction list_of_clocking_decl_assign **;**
| { attribute_instance } concurrent_assertion_item_declaration
| variable_lvalue **<=** expression **;**
default_skew ::=
input clocking_skew
| **output** clocking_skew
| **input** clocking_skew **output** clocking_skew
clocking_direction ::=
input [clocking_skew]
| **output** [clocking_skew]
| **input** [clocking_skew] **output** [clocking_skew]
| **inout**
list_of_clocking_decl_assign ::= clocking_decl_assign { , clocking_decl_assign }
clocking_decl_assign ::= signal_identifier [**=** hierarchical_identifier]
clocking_skew ::=
edge_identifier [delay_control]
| delay_control

edge_identifier ::= **posedge** | **negedge**

delay_control ::=

delay_value // from Annex A.7.4

| # **(** mintypmax_expression **)** // from Annex A.6.5

1.1.2 Semantics

The clocking domain syntax is extended by allowing

variable_lvalue <= expression

The variable must be declared as an output to the clocking domain. However, as in the case of properties, the variables used in the expression need not be declared as ports to the clocking domain.

The expression is the same as the expression used in the properties. As such, it allows

— value change functions ($rose, $fell, and $stable)

— ended method on a sequence

— system functions ($onehot,$onehot0,$inset,$insetz,$isunknown,$past,$countones)

The expression is evaluated after the evaluation of properties, using the sampled values of the expression variables. Since the sequences are evaluated prior to the expression evaluation, the ended method if used in the expression returns its value for the current clock tick.

1.1.3 Examples

The in_progress variable detects a rising edge on a boolean start_expr, and keeps track of when test_expr becomes true. The property then checks that this happens within the proper time range.

```
int in_progress;
clocking clk1 @(posedge clk);
   output in_progress;
   in_progress <= !sample_not_resetting ? 0 :
                     ($rose(start_expr) && (in_progress == 0) &&
                     !(test_expr == 1'b1)? 1 :
                        (test_expr == 1'b1) ? 0 :
                           (in_progress < max_cks) &&
                           (in_progress > 0) ? in_progress+1 : 0;
endclocking

property assert_frame_p;
   @(posedge clk)
      not_resetting |->
           ((((( in_progress >= min_cks) ||
               (test_expr == 1'b0)) && (in_progress < max_cks)) ||
               (in_progress == 0)) &&
              ((test_expr == 1'b0) || (!$rose(start_event))));
endproperty
```

In another example below, sequence s_deferred_deq is used to provide a delayed trigger by delay_latency for a dequeue operation to a series of assertions that verify the behavior of a fifo. In the example s_deferred_deq.ended is used in an assignment to a variable that is the fifo head pointer.

```
clocking clk1 @(posedge clk);
   output head_ptr;
   head_ptr <= (reset) ? 0 :
         (s_deferred_deq.ended && (ova_v_q_size > 0) ) ?
                   ((ova_v_head_ptr < depth-1) ?
                        (ova_v_head_ptr + 16'b1) :16'b0) :
                   ova_v_head_ptr;
endclocking

sequence s_deferred_deq;
 @clk1 deq ##deq_latency `true;
endsequence
```

1.2 Assumptions

The purpose of this enhancement is allow properties to be considered as assumptions for formal as well as for dynamic simulation tools. When a property is assumed, the tools may constraint the environment so that the property holds. Additionally, for random simulation, biasing on the inputs provides a way to make random choices.

This proposal includes two extensions. First, it provides an assumption construct, similar to the assertion and cover construct. Second, it extends the expression syntax to allow biasing specification. The syntax of the biasing is already provided by the constraint feature of System Verilog.

1.2.1 Syntax for assumption

procedural_assertion_item ::= // from Annex A.6.10

assert_property_statement

| cover_property_statement

| assume_property_statement

concurrent_assertion_item ::=

concurrent_assert_statement

| concurrent_cover_statement

| concurrent_assume_statement

concurrent_assert_statement ::= // from Annex A.2.10

[block_identifier**:**] assert_property_statement

concurrent_cover_statement ::=

[block_identifier**:**] cover_property_statement

concurrent_assume_statement ::=

[block_identifier**:**] assume_property_statement

assert_property_statement::=

assert property (property_spec **)** action_block

| **assert property (** property_instance **)** action_block

cover_property_statement::=

cover property (property_spec **)** statement_or_null

| **cover property (** property_instance **)** statement_or_null

cover_property_statement::=

assume property (property_spec **) ;**

| **assume property (** property_instance **) ;**

The procedural_assertion_item and concurrent_assertion_item are extended to include the **assume** construct. Note that **assume** does not provide an action block, as the actions for an assumption serve no purpose.

1.2.2 Semantics for assume

For the procedural usage of **assume**, the rules governing the inference of clocks and enabling conditions are identical to **assert**. Also, the rules for interpretation when embedded in an always or an initial block are identical to **assert**. Namely, when **assume** is embedded in an always block, the property is assumed to hold for every clock tick. When **assume** is embedded in an initial block, the property is assumed to hold for the first clock tick.

1.2.3 Syntax for biasing

assertion_expression ::=

expression

| expression **dist {** dist_list **} ;** // from Annex A.1.9

dist_list ::= dist_item { **,** dist_item }

dist_item ::=

value_range **:=** expression

| value_range **:/** expression

value_range ::=

expression

| **[** expression **:** expression **]**

This syntax introduces a new production named assertion_expression. In the BNF for assertions, the expression production must be replaced by assertion_expression. The operator **dist** and the production dist_list is explained in Section 12.4.4 of the System Verilog LRM.

1.2.4 Semantics for biasing

The biasing feature is only useful when properties are considered as assumptions to drive random simulation. For assertions or coverage, the biasing that is associated with any expression can be safely ignored converted to the set membership function. For example,

```
property proto
   @(posedge clk) (req dist {0:=40, 1:=60}) |-> req[*1:$] ##0 ack;
endproperty
property proto_assertion
   @(posedge clk) (req inside {0, 1}) |-> req[*1:$] ##0 ack;
endproperty
```

In the above example, signal request is specified with distribution in property proto and is converted to an equivalent property proto_assertion that is suitable for assertions.

Also, formal tools may follow the same conversion for assertions as well as for assumptions.

The semantics of biasing construct are explained in Section 12.4.4 of the System Verilog LRM.

It should be noted that the properties that are assumed must hold in the same way with or without biasing. The biasing simply provides a means to select values of free variables, according to the specified weights, when there is a choice of selection at a particular time.

1.2.5 Example of assume

Consider a simple synchronous request - acknowledge protocol, where the variable req can be raised at any time and must stay asserted until ack is asserted. In the next clock cycle both req and ack must be deasserted.

Properties governing req are:

```
property pr1;
   @(posedge clk) !reset_n |-> !req; //when reset_n is asserted (0),keep req 0
endproperty
property pr2;
   @(posedge clk) ack |=> !req; // one cycle after ack, req must be deasserted
end property
property pr3;
   @(posedge clk) req |-> req[*1:$] ##0 ack; // hold req asserted until
// and includingack asserted
endproperty
```

Properties governing ack are:

```
property pa1;
   @(posedge clk) !reset_n || !req |-> !ack;
endproperty
property pa2;
   @(posedge clk) ack |=> !ack;
endproperty
```

When verifying the behavior of a protocol controller which has to respond to requests on req, the assertions assert_req1 and assert_req2 should be proven while assuming that statements assume_ack1, assume_ack2 and assume_ack3 hold at all times.

```
assume_ack1:assume property (pr1);
assume_ack2:assume property (pr2);
assume_ack3:assume property (pr3);

assert_req1:assert property (pa1)
   else $display("\n ack asserted while req is still deassrted");
assert_req2:assert property (pa2)
   else $display("\n ack is extended over more than one cycle");
```

1.3 Using properties in the constraint block

The extensions proposed in Section 1.2 provide an easy mechanism for the application of assumptions. Along with the biasing, this mechanism is adequate for simple cases. For simulation, tools can determine free and state variables of the assumptions, apply randomization to the free variables, and drive inputs to the design such that all assumptions for the design are valid.

Often times, for more complicated cases, users need a better control over what gets randomized and when inputs to the design are driven. This must be carefully considered based on how outputs of the design settle in time relative to the clock. System Verilog provides a feature to specify boolean constraints using the **constraint** construct. The **constraint** construct is supported by random variable declarations and a function to ran-

domize random variables using the boolean constraints. This functionality is encapsulated by the class definition which allows to package multiple constraint definitions, random variables representing the free variables in the constraints, and a randomize function that chooses values for the random variables. Refer to Section 12.4 of the System Verilog LRM for details.

This proposal extends the **constraint** construct by allowing instantiations of properties within the constraint block.

1.3.1 Syntax

constraint_declaration ::=

[**static**] **constraint** constraint_identifier **{** { constraint_block } **}** *// from Annex A.1.9*

constraint_block ::=

solve identifier_list **before** identifier_list **;**

| expression **dist {** dist_list **} ;**

| property_instance **;**

| constraint_expression

constraint_expression ::=

expression **;**

| expression **=>** constraint_set

| **if (** expression **)** constraint_set [**else** constraint_set]

constraint_set ::=

constraint_expression

| **{** { constraint_expression } **}**

constraint_prototype ::= [**static**] **constraint** constraint_identifier

extern_constraint_declaration ::=

[**static**] **constraint** class_identifier **::** constraint_identifier **{** { constraint_block } **}**

identifier_list ::= identifier { **,** identifier }

The syntax of constraint_declaration is extended by including property_instance in constraint_block.

1.3.2 Semantics

When a class containing constraint blocks is instantiated, the properties start like assertions. The expressions of the properties form boolean constraints. These boolean constraints are conjoined with all other boolean constraints specified in the class. Properties may use random variables declared in the class. When the randomize function is invoked, the constraints are solved by choosing the appropriate random variable values. At the next clock tick, properties advance forward according to the values obtained by solving the constraints and a new set of constraints is thus selected for the next call to the randomize method.

While properties act like assertions to ensure that they hold at every clock tick, they provide boolean constraints that must be solved at the time of randomization. Here, the users determine the appropriate time to randomize, using their knowledge of the design.

All other features available with constraint blocks such as constraint_mode ON/OFF apply to constraints inferred from properties.

1.3.3 Example of the constraint block

Using the same example in 1.2.5, the code below illustrates the use of constraint blocks.

```
program p(input bit ack, input bit clk, output reset_n, output bit req);
```



```
property pr1(req, ack);
   @(posedge clk) !reset_n |-> !req;
endproperty
property pr2(req, ack);
   @(posedge clk) ack |=> !req;
end property
property pr3(req, ack);
   @(posedge clk) req |-> req[*1:$] ##0 ack;
endproperty

class req_class;
   rand bit v_req = 0;
   constraint req_constr {
      req_v dist {0:=40, 1:=60};
      pr1(v_req, ack);
      pr2(v_req, ack);
      pr3(v_req, ack);
   }
endclass
initial begin
   reset_n = 0;
   @(posedge clk);
   reset_n = 1;
end

initial begin
   req_class drive_req = new();
   while (not_done) begin
      @(negedge clk);
      drive_req.randomize();
      // drive the output using drive_req
      req = v_req;
   end
… etc.
end

endprogram
```

1.4 Access to sampled values

This proposal addresses the need to access sampled values of variables consistent with the sampling of variables specified in assertions.

The method currently available is to declare a clocking domain with the intended clock, specify required variables in the clocking domain, and access the sampled value by the variable name prefixed with the clocking domain name. For example,

```
clocking_domain gclk @(posedge sysclk);
   input a,b,c
endclocking
property p1;
   @gclk a ##1 b ##1 c;
endproperty;
assert (p1) $display(“success for p1, a = %d;”,gclk.a);
```

To allow users a direct access to sampled variable values without the use of explicit clock domain, a system function $sampled is proposed. This function can be invoked anywhere in the design to access a variable’s sampled value with respect to a clock.

1.4.1 Syntax of $sampled function

sampled_function ::=

$sampled (expression[**,** event_expression] **) ;**

1.4.2 Semantics of $sampled function

expression must comply with the restrictions on the expression usage in assertions. Refer to Section 17.4 for details on the restrictions.

Optionally, clocking for the expression is specified by event_expression. The value of the expression is sampled by event_expression. event_expression may be omitted in two cases:

— $sampled is used in an action block of a singly clocked assertion. The clock of the assertion is inferred for the function.

— $sampled is used in a block for which default clocking is specified. The default clocking is inferred for the function.

If clocking is specified, no inference of clocking is applied.

$sampled may be used anywhere in the code as a function call. The value returned by $sampled is the latest sampled value for the expression.

1.4.3 Example for $sampled

The example presented in Section 1.4 is rewritten using $sampled below:

```
property p1;
   @gclk a ##1 b ##1 c;
endproperty;
assert (p1) $display("success for p1, a = %d;",$sampled(a));
```

In this case, the sampled value of a is with respect to `@gclk`.

```
task out_1;
   out1 = b && $sampled(a, posedge clk); //samples with respect to posedge clk
   -
   -
endtask;
```

1.5 Access to gated clocked variables

This proposal provides for the need to access past values of a variable that is modelled with a gated clock. In the case where an assert uses a normal clock, while the variables are assigned with a gated clock, it requires a significant effort in modelling past gated values of such variables. For example,

```
always @(posedge clk)
    if (enable)  q <= d;


always @(posedge clk)
   assert (done |=> {out == $past(q, 2)) ;
```

In this example, the value of q returned by $past is the value of q at the second clock tick in the past. However, the value of q intended is the value at the second update of q in the past. The user must write additional code to keep track of the enabled updated values of q.

The proposal extends the $past function by providing for an optional argument that specifies the enabling con-

dition.

1.5.1 Syntax of extended $past

past_function ::=

$past (expression1[**,** number_of_ticks] [**,** expression2] **) ;**

expression1 can be any expression allowed in assertions. expression1 refers to the expression whose past value is needed.

number_of_ticks refers to a non-negative integral value.

expression2 refers to the enabling expression.

The default for omitting number_of_ticks is 1. The default for omitting expression2 is true.

Both, number_of_ticks and expression2 are optional. If expression2 is specified, but number_of_ticks is omitted, then a must be specified for the empty argument, such as

```
$past(in1, , enable);
```

In this example, number_of_ticks is 1.

1.5.2 Semantics of $past

When expression2 is not specified, the semantics of $past is unchanged. If expression2 is specified, the sampling of expression is performed based on its clock gated with expression2. For example,

```
always @(posedge clk)
    if (enable)  q <= d;


always @(posedge clk)
   assert (done |=> {out == $past(q, 2,enable)) ;
```

In this example, the sampling of q for evaluating $past is based on the clocking

```
posedge clk iff enable
```

1.6 Passing unbounded range as an argument

This proposal enhances the capability of specifying range that can be fully parameterized,i.e is passed as an argument to properties and sequences. Currently, the range is specified as

cycle_delay_const_range_expression ::=

constant_expression **:** constant_expression

| constant_expression **: $**

The symbol used to specify an unbounded range is $. The first constant_expression specifies the low range limit, while the second constant_expression specifies the high range limit. Since the high limit can be a $, it cannot be passed as an argument to a property. For example,

```
property inq(r1,r2);
   @(posedge clk) a ##[r1:r2] b ##1 c |=> d;
endproperty
assert inq(3, 10);
```

In the example above, r1 and r2 replace the range for the delay. However, in order to make the high range unbounded, the property must be rewritten as,

```
property inq1(r1);
   @(posedge clk) a ##[r1:$] b ##1 c |=> d;
```



```
    endproperty
    assert inq1(3);
```

This inability to make unbounded range as an argument causes difficulty in writing re-usable assertions.

This proposal allows $ as a symbolic constant representing unbounded integer value. It can be passed as a parameter using the parameter assignment mechanism, or as an argument to a property or a sequence. In addition, a compile time system function is provided to test whether a constant is a $. The syntax of the system function is

$isunbounded(const_expression);

This function returns true if the constant_expression is unbounded. $isunbounded can be used to conditionally generate properties using the generate statement.

It is also recommended to define basic arithmetic operations on $ to evaluate expression when $ is used in a constant expression.

In order minimize the impact on the evaluation of general expressions, restrictions must be placed on the use of $. $ may not be assigned to a variable, or passed as an argument to a function or a task.

