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 Alternative Non-blocking assignment (NBA) proposal for assertions

This proposal addresses the need for allowing assignments that only take 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 a temporal function such as $past and $rose. Such supplementary modeling is often required for assertions. Due to the inability of temporal function usage in expressions outside the assertion constructs, the user needs to write code to duplicate the semantics of temporal function in System Verilog in order to accomplish the purpose of supplementary modeling.

This proposal extends the definitions of temporal functions such that the functions can be used in expressions outside of the assertion constructs. Thus, any modeling code can then use the temporal functions and reduce the required modeling effort.

The temporal functions defined in this proposal are value change functions and $past. To use the functionality of **ended** in modeling code, the new proposed functionality (SV-EC EXT-7) of detecting the end of sequences is utilized and illustrated in this proposal.

1.2.1 Value change functions

Three functions are provided to detect changes in values between two adjacent clock ticks: `$rose`, `$fell` and `$stable`.

$rose (expression [, clocking_event])

$fell (expression [, clocking_event])

$stable (expression [, clocking_event])

A value change expression detects the change in value of an expression from the value of that expression at one clock tick prior to the current simulation time unit. Here, the current simulation time unit refers to the simulation time unit in which the function is evaluated. The result of a value change expression is true or false and can be used as a boolean expression. At the first clock tick, as determined by the clocking event, or before the first clock tick, the result of these functions are computed by comparing the current value to ‘x’.

`$rose` returns true if the least significant bit of the expression changed to 1. Otherwise, it returns false.

`$fell` returns true if the least significant bit of the expression changed to 0. Otherwise, it returns false.

`$stable` returns true if the value of the expression did not change. Otherwise, it returns false.

The clocking event argument is optional. If the clocking event is specified, then it overrides the clocking event inferred from the procedural context and the clocking event specified as default. If the clocking event is not specified, the clocking event must be either specified as a default or inferred from the procedural context. Otherwise, an error shall be reported. If a clocking event is specified as a default and also inferred from the procedural context, then the inferred clocking event is applied.

When these functions are used in an assertion, the clocking event argument of the functions, if specified, shall be identical to the clocking event of the expression in the assertion. If the clocking event argument is not specified, then the clocking event used for the assertion is applied. In the case of multi-clock assertion, the appropriate clocking event for the expression where the function is used, is applied.

It should be noted that the clocking event is used to obtain the sampled value of the argument expression at a clock tick prior to the current simulation time unit. This sampled value is compared against the value of the expression determined at the prepone time of the current simulation time unit in which the evaluation of the function takes place.

A typical use would be as shown in the example below.

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

In this example, the clocking event `(posedge clk)` is applied to $rose. $rose is true whenever the value of b changed to 1 from the previous tick of the clocking event.

1.2.2 $past function

The syntax of $past is extended as:

$past (expression [,number_of_ticks] [, clocking_event])

`$past` returns the sampled value of the expression that was present number_of_ticks prior to the current simulation time unit. Here, the current simulation time unit refers to the simulation time unit in which the function is evaluated.

The optional argument number_of_ticks specifies the number of clock ticks in the past. If number_of_ticks is not specified, then it defaults to 1.

If the specified clock tick in the past is before the start of simulation, the returned value from the `$past` function is a value of X.

The other optional argument clocking_event specifies the clock for the function. The rules governing the usage are same as described for the value change function in the previous section.

$past can be used in any System Verilog expression. An example is shown below.

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

In this example, the clocking event `(posedge clk)` is applied to $past. $past is evaluated in the current occurrence of `(posedge clk),` and returns the value of b sampled at the previous occurrence of `(posedge clk).`

1.2.3 Detecting and using the end of a sequence

Extension EXT-7 proposed in the SV-EC committee allows to detect the match of a sequence, and to use the end point of the sequence like an event. For example,

```
sequence s1;
   @(posedge clk) a ##b ##c;
```



```
endsequence

initial
L1: @ (s1) sig1 <= sig2;
```

The statement L1 waits for a match of the sequence s1, and then executes the assignment.

The following example illustrates the use of a sequence match in an expression.

```
L2: sig1 <= sig2 && s1.triggered;
```

The statement L2 uses the level sensitive sequence match using the triggered method. s1.triggered is true in the simulation time unit after the observe region in which the sequence match occurs. s1.triggered gets reset to false in the next simulation time unit.

The above functionality can be used for assertion modeling. The following points, however, should be noted:

— triggered method returns true only after the evaluation of sequence in the observe region

— since program blocks execute after the observe region, triggered returns the result as expected

— the code in a module normally executes prior to the observe region, so the execution must be delayed until after the observe region to get the expected value of triggered

An example below illustrates the use of triggered in a program block, using the prior definition of sequence s.

```
forever
L3: @ (posedge clk) sig3 <= sig4 && s.triggered;
```

In statement L3, s.triggered evaluates to true if there was a match of sequence s in the current clock tick.

An example below illustrates the use of triggered in a module, using the prior definition of sequence s.

```
program p;
   L4: assign s_detect = !s_detect;
endprogram

always @ (posedge clk) begin
   L5: @(s_detect);
   L6:  sig5 <= sig6 && s.triggered;
end
```

Statement L4 is used to create a signal change in the program, so that the code in the always block in a module can be delayed until the program executes. This happens for every simulation time unit, although the delaying is actually used only for the clock tick according to the clocking event `(posedge clk)`.

Statement L5 waits until the sequence evaluation completes and the program statement executes.

Statement L6 uses triggered to obtain the sequence match at the current clock tick.

