

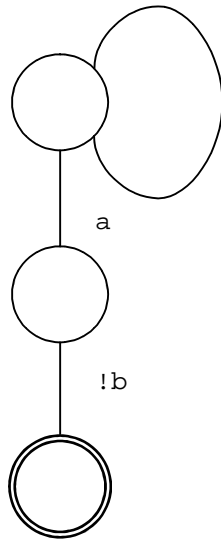
Rationale

The main goal of the global clocking concept proposed here is to provide a definition of the primary clock synchronizing transitions in formal models. Another goal is to introduce a global default clock for concurrent assertions.

DUT for formal verification purposes are usually considered as a set of states and transitions between states. In RTL-level models verification the transitions are synchronous, i.e., at each step one and only one transition (possibly idle) is taken. This sequence of steps defines the primary clock in a natural way, s.t. all system transitions are synchronized by it. The primary clock is fair, i.e., it never stops ticking. In the metalanguage describing the formal semantics of SystemVerilog (see Annex E), the primary clock is denoted as `@1`, but this notation cannot be used in the language itself, since the event `@1` never happens. To be able to specify the primary clock in the language, a special notation is needed, and it is suggested to call it `$global_clock` in this proposal.

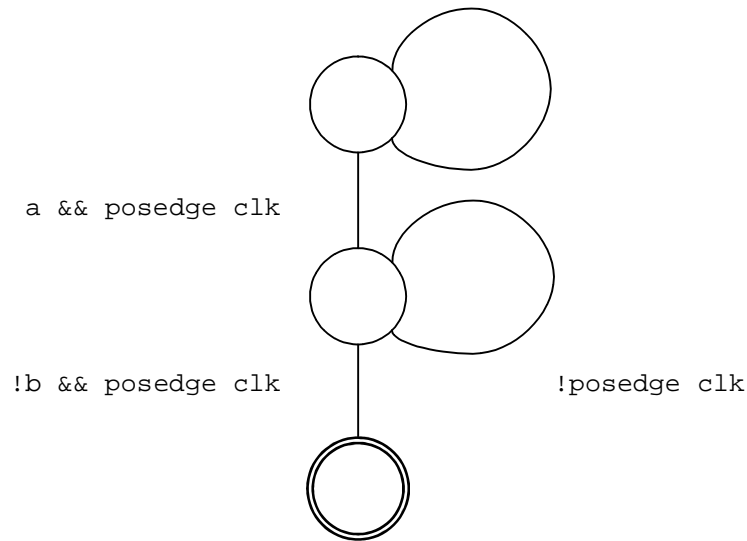
Formal verification of assertions controlled by primary clock is more efficient then the verification of assertions controlled by any other clock. Compare the automata generated for the following two assertions:

always p1: assert property (@\$global_clock a |=> b);



and

always p2: assert property (@(posedge clk) a |=> b);



The first automaton is simpler: it does not have clocking conditions along its edges, and it does not have an extra idle edge for awaiting the clock.

The notion of the primary clock allows specifying the next value of a variable – the value of the variable assigned at the end of a given transition: this value is set at the next tick of the primary clock $\$next_i(v)$ (this function is not part of this proposal and is referenced here for illustration purposes only). Using primary clock and next value functions is very natural for specifying stability properties. Consider the assertion saying that some signal sig should be always stable. The natural way to write this assertion is:

```
always assert property(@$global_clock sig == $next_i(sig));
```

Without the notion of global clock this assertion has to be rewritten as:

```
always assert property(@clk ##1 sig == $past(sig));
```

which has the following drawbacks (I intentionally use $\$past$ and not $\$stable$ here to avoid a discussion about future stability functions):

1. Most users will write the above assertion as `always assert property(@clk sig == $past(sig));` without skipping the initial phase, and thus will check that sig is always 0 (if sig has a two-value type), instead of the intended assertion.
2. The automaton of the second assertion is more expensive than that one of the first assertion since it contains more states.
3. The first assertion is more robust since it doesn't need to be changed when a faster clock is introduced in the design.

To be able to use primary clock in simulation, there is a need to associate a real clocking event with it. Therefore the global clocking construct is suggested. To achieve the consistent assertion behavior in the simulation, it is required that in assertions **referencing** the global clock, all other clocks are synchronized with it. E.g., for the following global clock definition

global clocking @(posedge fclk); endclocking

and assertion

```
assert property (@clk a ##1 @$global_clock b | => c);
```

it is required that all clk changes happen on the rising edge of fclk. global clocking definition is ignored by formal verification tools and \$global_clock is always assumed to be the primary clock.

No special requirements on assertion clocks are imposed for assertions which do not reference \$global_clock. Thus the introduction of global clocking and \$global_clock constructs is backward compatible and introduces no penalty in simulation for assertions that do not reference \$global_clock.

15.2 Clocking block declaration

REPLACE

```
clocking_declaration ::=  
    [ default ] clocking [ clocking_identifier ] clocking_event ;  
    { clocking_item } endclocking [ : clocking_identifier ]
```

WITH

```
clocking_declaration ::=  
    [ default ] clocking [ clocking_identifier ] clocking_event ;  
    { clocking_item } endclocking [ : clocking_identifier ]  
    | global clocking [ clocking_identifier ] clocking_event ;
```

15.10 Cycle delay:

REPLACE

What constitutes a cycle is determined by the default clocking in effect (see 15.11). If no default clocking has been specified for the current module, interface, or program, then the compiler shall issue an error.

Example:

```
    ## 5;          // wait 5 cycles (clocking events) using the default  
clocking  
    ## (j + 1); // wait j+1 cycles (clocking events) using the default  
clocking
```

WITH

What constitutes a cycle is determined by the default clocking in effect (see [Note to editor: Insert a reference to *Default clocking* subclause](#)). ~~If no default clocking has been specified for the current module, interface, or program, then the compiler shall issue an error.~~ If no default clocking has been specified for the current module, interface, or program, but the global clocking has been specified for the design, the cycle is taken relative to the global clocking (see [Note to editor: Reference to *Global clocking* subclause](#)). If neither default nor the global clocking have been specified then the compiler shall issue an error.

Example:

```
## 5;          // wait 5 cycles (clocking events) using the default  
clocking  
## (j + 1); // wait j+1 cycles (clocking events) using the default  
clocking
```

ADD

15.12 Global clocking

Note to editor: Shift the numeration of the following sections accordingly.

One clocking can be specified as the *global clocking* for the design.

The syntax for the global clocking specification statement is as follows:

```
clocking_declaration ::=  
    ...  
    | global clocking [clocking_identifier ] clocking_event ; endclocking
```

Only one global clocking can be specified anywhere in the design in any one compilation unit of the design (i.e., global clocking must have at most one definition on the entire model after elaboration). It is an error if there is more than one such specification of global clocking.

The system function `$global_clock` returns the event expression specified in the global clocking statement. The function has no arguments. It shall be a compiler error to invoke `$global_clock` function when no global clocking has been defined.

The main purpose of the global clocking is to specify which clocking events correspond to the primary clock used in formal verification.

The global clocking also acts as a default clocking in those modules, interfaces and programs where no default clocking has been specified.

Example: Declaring a global clocking

```
module top ;  
    logic clk1, clk2;  
    global clocking sys @(clk1 or clk2); endclocking  
    // ...  
endmodule
```

In this example a global clocking event `sys` is defined as a change of one of two clocks `clk1` and `clk2`. The global clocking name `sys` is optional, since it may be referred to as `$global_clock` from anywhere in the design.

The global clock does not need to be a real signal in the design, but rather an event.

17.3 Concurrent assertions overview

REPLACE

The clock expression that controls evaluation of a sequence can be more complex than just a single signal name. Expressions such as `(clk && gating_signal)` and `(clk iff gating_signal)` can be used to represent a gated clock. Other more complex expressions are possible. However, in order to ensure proper

behavior of the system and conform as closely as possible to truly cycle-based semantics, the signals in a clock expression must be glitch-free and should only transition once at any simulation time.

WITH

The clock expression that controls evaluation of a sequence can be more complex than just a single signal name. Expressions such as `(clk && gating_signal)` and `(clk iff gating_signal)` can be used to represent a gated clock. Other more complex expressions are possible. However, in order to ensure proper behavior of the system and conform as closely as possible to truly cycle-based semantics, the signals in a clock expression must be glitch-free and should only transition once at any simulation time.

If an assertion is controlled by `$global_clock` (see [Note to editor: insert a reference to *Global clocking* subclause here](#)) then in simulation `$global_clock` is substituted by the clocking event defined in the **global clocking** statement. In formal verification the `$global_clock` is considered to be a primary system clock (see E.3.1). Thus, in the following example

```
global clocking @clk; endclocking

...

assert property(@$global_clock a);
```

the assertion states that `a` remains true at each primary phase, which is interpreted in simulation as

```
assert property(@clk a);
```

17.7.3 Sampled value functions

REPLACE

The following rules are used to infer the clocking event:

- If used in an assertion, the appropriate clocking event from the assertion is used.
- If used in an action block of a singly clocked assertion, the clock of the assertion is used.
- If used in a procedural block, the inferred clock, if any, for the procedural code (see 17.13.5) is used.

Otherwise, default clocking (see 15.11) is used.

WITH

The following rules are used to infer the clocking event, in the order shown:

- If used in an assertion, the appropriate clocking event from the assertion is used.
- If used in an action block of a singly clocked assertion, the clock of the assertion is used.
- If used in a procedural block, the inferred clock, if any, for the procedural code (see 17.13.5) is used.
- If default clocking (see [Note to editor: insert a reference to *Default clocking* subclause here](#)) has been specified, it is used.
- If global clocking (see [Note to editor: insert a reference to *Global clocking* subclause here](#)) has been specified, it is used.

17.12 Multiclock support

17.12.2 Multiclocked properties

ADD TO THE END

To assure consistency between simulation and formal verification, in the multiclocked properties referncing the global clock all other clocks should be aligned with the ticks of the global clocks: the property clocks may change only at the same simulation ticks when the global clock is changing (i.e., when the global clocking event happens). E.g., in the properties

```
@($global_clock) sig0 ##1 @(negedge clk1) sig1  
  
@(posedge clk2) s0 | => @$global_clock) s1
```

negedge clk1 and **posedge** clk2 must be aligned with \$global_clock.

17.14 Clock resolution

REPLACE

In general, a clocking event applies throughout its scope except where superseded by an inner clocking event, as with clock flow in multiclocked sequences and properties.

WITH

— Global clock, for example:

```
module top; // Top-level instance  
logic proj_clk ;  
global clocking @proj_clk; endclocking // System clock  
...  
endmodule  
...  
property p5; (a |-> ##2 b); endproperty  
assert property (p5); // assertion uses @proj_clk as its clocking event
```

In general, a clocking event applies throughout its scope except where superseded by an inner clocking event, as with clock flow in multiclocked sequences and properties.

REPLACE

- a) In a module, interface, or program with a default clocking event, a concurrent assertion statement that has no otherwise specified leading clocking event is treated as though the default clocking event had been written explicitly as the leading clocking event. The default clocking event does not apply to a sequence or property declaration except in the case that the declaration appears in a **clocking** block whose clocking event is the default.
- b) The following rules apply within a **clocking** block:
 - 1) No explicit clocking event is allowed in any property or sequence declaration within the **clocking** block. All sequence and property declarations within the **clocking** block are treated as though the clocking event of the **clocking** block had been written explicitly as the leading clocking event.
 - 2) Multiclocked sequences and properties are not allowed within the **clocking** block.
 - 3) If a named sequence or property that is declared outside the **clocking** block is instantiated within the **clocking** block, the instance must be singly clocked and its clocking event must be identical to that of the **clocking** block.

- c) A contextually inferred clocking event from a procedural block supersedes a default clocking event. The contextually inferred clocking event is treated as though it had been written as the leading clocking event of any concurrent assertion statement to which the inferred clock applies. The maximal property of such a concurrent assertion statement must be singly clocked, and the clocking event, if specified otherwise, must be identical to the contextually inferred clocking event.
- d) An explicitly specified leading clocking event in a concurrent assertion statement supersedes a default clocking event.
- e) A multiclocked sequence or property can inherit the default clocking event as its leading clocking event. If a multiclocked property is the maximal property of a concurrent assertion statement, then the property must have a unique semantic leading clock (see 17.14.1).
- f) If a concurrent assertion statement has no explicit leading clocking event, there is no default clocking event, and no contextually inferred clocking event applies to the assertion statement, then the maximal property of the assertion statement must be an instance of a sequence or property for which a unique leading clocking event is determined.

WITH

- a) In a module, interface, or program with a default clocking event, a concurrent assertion statement that has no otherwise specified leading clocking event is treated as though the default clocking event had been written explicitly as the leading clocking event. The default clocking event does not apply to a sequence or property declaration except in the case that the declaration appears in a clocking block whose clocking event is the default. If default clocking is not defined and the design has a global clocking declaration then a concurrent assertion that has no other leading clock inferred will acquire global clocking as the leading clocking event.
- b) The following rules apply within a **clocking** block:
 - 1) No explicit clocking event is allowed in any property or sequence declaration within the **clocking** block. All sequence and property declarations within the **clocking** block are treated as though the clocking event of the **clocking** block had been written explicitly as the leading clocking event.
 - 2) Multiclocked sequences and properties are not allowed within the **clocking** block.
 - 3) If a named sequence or property that is declared outside the **clocking** block is instantiated within the **clocking** block, the instance must be singly clocked and its clocking event must be identical to that of the **clocking** block.
- c) A contextually inferred clocking event from a procedural block supersedes a default/global clocking event. The contextually inferred clocking event is treated as though it had been written as the leading clocking event of any concurrent assertion statement to which the inferred clock applies. The maximal property of such a concurrent assertion statement must be singly clocked, and the clocking event, if specified otherwise, must be identical to the contextually inferred clocking event.
- d) An explicitly specified leading clocking event in a concurrent assertion statement supersedes a default/global clocking event.
- e) A multiclocked sequence or property can inherit the default/global clocking event as its leading clocking event. If a multiclocked property is the maximal property of a concurrent assertion statement, then the property must have a unique semantic leading clock (see 17.14.1).
- f) If a concurrent assertion statement has no explicit leading clocking event, there is no default/global clocking event, and no contextually inferred clocking event applies to the assertion statement, then

the maximal property of the assertion statement must be an instance of a sequence or property for which a unique leading clocking event is determined.

REPLACE

```
module examples_without_default (input logic a, b, c, clk);

property q1 ;
  $rose(a) /-> ##[1:5] b;
endproperty

property q5;
  @(negedge clk) b[*3] /=> !b;
endproperty

property q6;
  q1 and q5;
endproperty

a5 : assert property (q6);
    // illegal: no leading clocking event

a6 : assert property ($fell(c) /=> q6);
    // illegal: no leading clocking event

sequence s2 ;
  $rose(a) ##[1:5] b;
endsequence

c1 : cover property (s2) ;
    // illegal: no leading clocking event

c2 : cover property @( negedge clk) s2) ;
    // legal: explicit leading clocking event, @(negedge clk)

sequence s3;
  @(negedge clk) s2;
endsequence

c3 : cover property (s3);
    // legal: leading clocking event, @(negedge clk),
    // determined from declaration of s3

c4 : cover property (s3 ##1 b);
    // illegal: no default, inferred, or explicit leading
    // clocking event and maximal property is not an instance

endmodule
```

WITH

```
module examples_without_default (input logic a, b, c, clk);

property q1 ;
  $rose(a) /-> ##[1:5] b;
endproperty

property q5;
  @(negedge clk) b[*3] /=> !b;
endproperty
```

```

property q6;
    q1 and q5;
endproperty

a5 : assert property (q6);
    // illegal: no leading clocking event
    // legal if global clocking declared:
    // $global_clock is the semantic leading clock
    // illegal, otherwise: no leading clocking event

a6 : assert property ($fell(c) /=> q6);
    // illegal: no leading clocking event
    // legal if global clocking declared:
    // $global_clock is the semantic leading clock
    // illegal, otherwise: no leading clocking event

a7 : assert property ($rose(clk) ##1 (!$rose(clk)[*1:$]) |-> $stable(sig) );
    // legal if global clocking declared
    // a7 ensures that sig is stable at all system clock ticks
    //     between two design clock clk ticks
    // uses the system clock to sample a design clock and a signal

sequence s2 ;
    $rose(a) ##[1:5] b;
endsequence

c1 : cover property (s2) ;
    // illegal: no leading clocking event
    // legal if global clocking declared:
    // $global_clock is the semantic leading clock
    // illegal, otherwise: no leading clocking event

c2 : cover property @( negedge clk) s2) ;
    // legal : explicit leading clocking event, @(negedge clk)

sequence s3;
    @(negedge clk) s2;
endsequence

c3 : cover property (s3);
    // legal : leading clocking event, @(negedge clk),
    // determined from declaration of s3

c4 : cover property (s3 ##1 b);
    // illegal : no default, inferred, or explicit leading
// clocking event and maximal property is not an instance
    // legal if global clocking declared:
    // $global_clock is the semantic leading clock
    // illegal, otherwise: no leading clocking event

endmodule

```

A.6.11 Clocking block

REPLACE

clocking declaration ::=
 [**default**] **clocking** [clocking identifier] clocking event ;
 {clocking item } **endclocking** [: clocking identifier]

WITH

clocking_declaration ::=
 [**default**] **clocking** [clocking_identifier] clocking_event ;
 { clocking_item } **endclocking** [: clocking_identifier]
 | global clocking [clocking_identifier] clocking_event ;

E.3.1 Rewrite rules for clocks

REPLACE

The semantics of clocked sequences and properties is defined in terms of the semantics of unclocked sequences and properties. The following rewrite rules define the transformation of a clocked sequence or property into an unclocked version that is equivalent for the purposes of defining the satisfaction relation. In this transformation, it is required that the conditions in event controls not be dependent upon any local variables.

— @(*c*) *b* (!*c* [*0:\$] ##1 *c* & *b*) .
 — @(*c*) (1, *v* = *e*) (@(*c*) 1 ##0 (1, *v* = *e*)) .
 — @(*c*) (*P*) (@(*c*) *P*) .
 — @(*c*) (*R*1 ##1 *R*2) (@(*c*) *R*1 ##1 @(*c*) *R*2) .
 — @(*c*) (*R*1 ##0 *R*2) (@(*c*) *R*1 ##0 @(*c*) *R*2) .
 — @(*c*) (*R*1 **or** *R*2) (@(*c*) *R*1 **or** @(*c*) *R*2) .
 — @(*c*) (*R*1 **intersect** *R*2) (@(*c*) *R*1 **intersect** @(*c*) *R*2) .
 — @(*c*) **first_match** (*R*) **first_match** (@(*c*) *R*) .
 — @(*c*) *R* [*0] (@(*c*) *R*) [*0] .
 — @(*c*) *R* [*1:\$] (@(*c*) *R*) [*1:\$] .
 — @(*c*) **disable iff** (*b*) *P* **disable iff** (*b*) @(*c*) *P* .
 — @(*c*) **not** *P* **not** @(*c*) *P* .
 — @(*c*) (*R* |→ *P*) (@(*c*) *R* |→ @(*c*) *P*) .
 — @(*c*) (*P*1 **or** *P*2) (@(*c*) *P*1 **or** @(*c*) *P*2) .
 — @(*c*) (*P*1 **and** *P*2) (@(*c*) *P*1 **and** @(*c*) *P*2) .

WITH

The semantics of clocked sequences and properties is defined in terms of the semantics of unclocked sequences and properties. The following rewrite rules define the transformation of a clocked sequence or property into an unclocked version that is equivalent for the purposes of defining the satisfaction relation. In this transformation, it is required that the conditions in event controls not be dependent upon any local variables.

— @(\$global_clock) *b* (*b*) .
 — @(*c*) *b* (!*c* [*0:\$] ##1 *c* & *b*) .
 — @(*c*) (1, *v* = *e*) (@(*c*) 1 ##0 (1, *v* = *e*)) .
 — @(*c*) (*P*) (@(*c*) *P*) .
 — @(*c*) (*R*1 ##1 *R*2) (@(*c*) *R*1 ##1 @(*c*) *R*2) .
 — @(*c*) (*R*1 ##0 *R*2) (@(*c*) *R*1 ##0 @(*c*) *R*2) .
 — @(*c*) (*R*1 **or** *R*2) (@(*c*) *R*1 **or** @(*c*) *R*2) .
 — @(*c*) (*R*1 **intersect** *R*2) (@(*c*) *R*1 **intersect** @(*c*) *R*2) .
 — @(*c*) **first_match** (*R*) **first_match** (@(*c*) *R*) .
 — @(*c*) *R* [*0] (@(*c*) *R*) [*0] .

- @(*c*) R [*1:\$] (@(*c*) R)[*1:\$] .
- @(*c*) **disable iff** (*b*) P **disable iff** (*b*) @(*c*) P .
- @(*c*) **not** P **not** @(*c*) P .
- @(*c*) (R |→ P) (@(*c*) R |→ @(*c*) P) .
- @(*c*) ($P1$ **or** $P2$) (@(*c*) $P1$ **or** @(*c*) $P2$) .
- @(*c*) ($P1$ **and** $P2$) (@(*c*) $P1$ **and** @(*c*) $P2$) .