

This proposal requires the following modifications:

- Modify Section 17.7.3 as below.
- Modify Section 17.9 as specified below.
- Modify Section 22.7 as shown below.

17.7.3 Value change Sampled value functions

This section describes the system functions available for accessing sampled values of an expression. These functions include the capability to access current sampled value, access sampled value in the past, or detect changes in sampled value. Sampling of an expression is explained in Section 17.3. Following functions are provided.

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

```
$sampled(expression [ , clocking_event])  
$rose(expression [ , clocking_event])  
$fell(expression [ , clocking_event])  
$stable(expression [ , clocking_event])  
$past (expression1 [ ,number_of_ticks][ ,expression2][ ,clocking_event] )
```

~~A value change expression at a clock tick detects the change in value of an expression from the value of that expression at the previous clock tick. The result of a value change expression is true or false and can be used as a boolean expression. At the first clock tick after the assertion is started, the result of these functions are computed by comparing the current value to 'x'.~~

The use of these functions is not limited to assertion features. These functions may be used as expressions in procedural code. The clocking event, although optional as an explicit argument to the functions, is required for their semantics. The clocking event is used to sample the value of the argument expression.

The clocking event must be explicitly specified as an argument, or inferred from the code where it is used. 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 (Section 17.12.2) is used.

Otherwise, default clocking (Section 15.11) is used.

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. In the case of multi-clock assertion, the appropriate clocking event for the expression where the function is used, is applied to the function.

Function `$sampled` returns the sampled value of the expression with respect to the last occurrence of the clocking event. When `$sampled` is invoked prior to the occurrence of the first clocking event, the value of X is returned. The use of `$sampled` in assertions, although allowed, is redundant, as the result of the function is identical to the sampled value of the expression itself used in the assertion.

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

A value change function detects the change in the sampled value of an expression. 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. Here, the current simulation time unit refers to the simulation time unit in which the function is evaluated. This sampled value is compared against the value of the expression determined at the prepon time of the current

simulation time unit. The result of a value change expression is true or false and can be used as a boolean expression.

`$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.

When these functions are used at the first clock tick, determined by the clocking event, or before the first clock tick, the result of these functions are computed by comparing the current sampled value of the expression to X.

Figure 17-3 illustrates two examples of value changes when used in an assertion:

— Value change expression `e1` is defined as `$rose(req)`

— Value change expression `e2` is defined as `$fell(ack)`

INSERT FIGURE 17-3

The clock ticks used for sampling the variables are derived from the clock for the property, which is different from the simulation ticks. Assume, for now, that this clock is defined elsewhere. At clock tick 3, `e1` occurs because the value of `req` at clock tick 2 was low and at clock tick 3, the value is high. Similarly, `e2` occurs at clock tick 6 because the value of `ack` was sampled as high at clock tick 5 and sampled as low at clock tick 6.

The example below illustrates the use of `$rose` in System Verilog code outside assertions.

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

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

In addition to accessing value changes, the past values can be accessed with the `$past` function. Following three optional arguments are provided:

`expression2` is used as a gating expression for the clocking event

`number_of_ticks` specifies the number of clock ticks in the past

`clocking_event` specifies the clocking event for sampling `expression1`

`expression1` and `expression2` can be any expression allowed in assertions.

`number_of_ticks` must be one or greater. If `number_of_ticks` is not specified, then it defaults to 1. `$past` returns the sampled value of the expression that was present `number_of_ticks` prior to the time of evaluation of `$past`. A clock tick is based on `clocking_event`. 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 optional argument `clocking_event` specifies the clock for the function. The rules governing the usage of `clocking_event` are same as described for the value change function.

When intermediate optional arguments between two arguments are not needed, a comma must be placed for each omitted argument. For example,

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

Here, a comma is specified to omit `number_of_ticks`. The default of one is used for the empty `number_of_ticks` argument. Note that a comma for the omitted `clocking_event` argument is not needed, as it does not fall within the specified arguments.

`$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`).

When `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 expression `posedge clk iff enable`

17.9 System functions

Assertions are commonly used to evaluate certain specific characteristics of a design implementation, such as whether a particular signal is “one-hot”. The following system functions are included to facilitate such common assertion functionality:

- `$onehot (<expression>)` returns true if only one bit of the expression is high.
- `$onehot0 (<expression>)` returns true if at most one bit of the expression is high.
- `$inset (<expression>, <expression> {, <expression> })` returns true if the first expression is equal to at least one of the subsequent expression arguments.
- `$insetz (<expression>, <expression> {, <expression> })` returns true if the first expression is equal to at least other expression argument. The comparison is performed using casez semantics, so ‘z’ or ‘?’ bits are treated as don’t-cares.
- `$isunknown (<expression>)` returns true if any bit of the expression is ‘x’. This is equivalent to `^<expression> === 'bx`.

All of the above system functions have a return type of bit. A return value of `1'b1` indicates true, and a return value of `1'b0` indicates false.

~~In addition to accessing values of signals at the time of evaluation of a boolean expression, the past values can be accessed with the `$past` function:~~

```
$past (<expression> [, number_of_ticks])
```

~~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. `$past` returns the sampled value of the expression that was present `number_of_ticks` prior to the time of evaluation of `$past`. 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.~~

Another useful function provided for the boolean expression is `$countones`, to count the number of 1s in a bit vector expression.

```
$countones ( expression )
```

An `x` and `z` value of a bit is not counted towards the number of ones.

22.7 Assertion system functions

INSERT syntax box 22-5

Assertions are commonly used to evaluate certain specific characteristics of a design implementation, such as whether a particular signal is “one-hot”. The following system functions are included to facilitate such common assertion functionality:

- `$onehot` returns true if one and only one bit of expression is high.
- `$onehot0` returns true if at most one bit of expression is high.
- `$inset` returns true if the first expression is equal to at least one of the subsequent expression arguments.
- `$insetz` returns true if the first expression is equal to at least one other expression argument. Comparison is performed using **casez** semantics, so `z` or `?` bits are treated as don’t-cares.
- `$isunknown` returns true if any bit of the expression is `X`. This is equivalent to `^expression === 'bx`.

All of the above system functions shall have a return type of **bit**. A return value of `1'b1` shall indicate true, and a return value of `1'b0` shall indicate false.

A function is provided to return sampled value of an expression.

```
$sampled ( expression [,clocking_event])
```

Three functions are provided for assertions to detect changes in values between two adjacent clock ticks.

```
$rose ( expression [,clocking_event])
$fell ( expression [,clocking_event])
$stable ( expression [,clocking_event])
```

~~These functions are discussed in Section 17.7.3.~~

The past values can be accessed with the `$past` function.

```
$past (expression [,number_of_ticks] [,expression2] [,clocking_event])
```

`$sampled`, `$rose`, `$fell`, `$stable` and `$past` is discussed in Section 17.7.3.

The number of 1s in a bit vector expression can be determined with the `$countones` function.

```
$countones ( expression)
```

~~`$past` and `$countones` are~~ is discussed in Section 17.9.