All references are to SystemVerilog 3.1/ draft 5

Chapter 17

1) Section 17.2, pp. 152

taken when the assert expression is true. The statement associated with **`else`** is called a *fail statement* and is executed if the assertion fails. That is, the expression does not evaluate to a known, non-zero value. The **else** statement can also be omitted. The action block is executed immediately after the evaluation of the assert expression.

2) Section 17.2, pp. 152

Note: ~~The pass and fail statements are executed in the reactive region. This is explained in the scheduling semantics section, Section 14.~~ The assertion control system tasks are described in Section 22.6.

3) Section 17.3, pp. 154
An expression such as `(clk && gating_signal)` **and** `(clk` **iff** `gating_signal)` could be used to represent ~~a~~ gated clocks.

4) Section 17.5 Syntax 17-2, pp. 156 : delete the following production

~~assertion_variable_declaration ::=~~
~~data_type list_of_variable_identifiers;~~

5) Section 17.5 Syntax 17-2, pp. 156: misalignment

| ## [cycle_delay_const_range_expression] ~~sequence_instance ::=~~

sequence_instance ::=
sequence_identifier [(actual_arg_list)]

6) Section 17.5 Syntax 17-3, pp. 157

| ## [cycle_delay_const_range_expression] ~~sequence_instance ::=~~
~~sequence_identifier [(actual_arg_list)]~~

7) Section 17.5, pp. 157 - 158
The following are examples of delay expressions. `‘true` is a boolean expression that always evaluates to true and is used for visual clarity. It can be defined as:

```
‘define true 1

##0 a // means a
##1 a // means ‘true ##1 a
##2 a // means ‘true ##1 ‘true ##1 a
##[0:3]a // ~~means(a)~~ means (a) or (‘true ##1 a) or (‘true ##~~1‘true~~1 ‘true ##1 a) or
(‘true ##1 ‘true ##1 ‘true ##1 a)
a ##2 b // means a ##1 ‘true ##1 b
```

~~Note: ‘true is not provided by the language but is used as a boolean expression macro that always returns true. It can be defined as:~~

```
~~‘define true 1~~
```

8) Section 17.5, pp. 158

This specifies that `req` will be true on the current ~~sample~~ clock tick, and `gnt` will be true on the second subsequent ~~sample~~clock tick, as shown in Figure 17-2.

9) Section 17.5, pp. 158

The following specifies that signal `b` will be true on the Nth ~~sample~~ clock tick after signal `a`:

10) Section 17.5, pp. 158

In the above example, `c` is the endpoint of sequence `seq1`, and `d` is the start of sequence `seq2`. When concatenated with 0 clock tick delay~~sampling~~, `c` and `d` must occur at the same time, resulting in a concatenated sequence equivalent to:

11) Section 17.6, pp. 160

In this example, sequences `s1` and `s2` are evaluated ~~sampled~~ on each successive posedge of `clk`. The sequence `s3` is evaluated ~~sampled~~ on the negedge of `clk`.

12) Section 17.7.2, pp. 162

To specify the *consecutive repetition* ~~repetition~~ of an expression within a sequence,

13) Section 17.7.2, pp. 162

A *consecutive repetition* ~~repetition~~ specifies that the item or expression must occur

14) Section 17.7.2, pp. 162

```
‘true ##3 (a [*3])   // means ‘true ##1 ‘true ##1‘true1 ‘true ##1 a ##1 a ##1 a
(‘true ##2 a) [*3]   // means (‘true ##2 a) ##1 (‘true ## 2 a) ##1
                     // (‘true ##2 a), which in turn means ‘true ##1 ‘true ##1
                     // a ##1‘true1 ‘true ##1 ‘true ##1 a ##1‘true1 ‘true
##1‘true1 ‘true ##1 a
```

15) Section 17.7.2, pp. 164

The ~~It is also possible to specify nonconsecutive exact repetition (goto~~ *goto repetition* (non-consecutive exact repetition) specifies the repetition of a boolean expression such as~~with:~~

16) Section 17.7.2, pp. 164

Adding the range specification to this allows the construction of useful sequences containing a boolean expression that is true for at most `N` occurences~~samples~~:

17) Section 17.7.2, pp. 164

The *non-consecutive* ~~A nonconsecutive count~~ repetition extends the goto repetition by extra clock ticks ~~samples~~ where the boolean expression is not true

18) Section 17.7.3, pp. 164: Replace syntax block 17-6 with

```
$rose ( expression )

$fell ( expression )

$stable ( expression )
```

19) Section 17.7.11, pp. 176

All matches of *antecedent sequence_expr* must satisfy *consequent sequence_expr*. The satisfaction of the *consequent sequence_expr* means that there is at least one match of the *sequence_expr*.

20) Section 17.8, pp. 180

Local variables can be written on repeated sequences and accomplish accumulation of values.

```
sequence rep_v;
```

```
    int x;
    'true,x = 0 ##0
    (!a [* 0:$] ##1 a, x = x+data)[*4] ##1 b ##1 c && (data_out == x);
endsequence
```

The local variables declared in one sequence are not visible in the sequence where it gets instantiated. An example below illustrates an illegal access to local variable `v1` of **sequence** `sub_seq1` in **sequence** `seq1`.

```
sequence sub_seq1;
   int v1;
   a ##1 !a, v1 = data_in ##1 !b*[0:$] ##1 b && (data_out == v1);
endsequence
sequence seq1;
   c ##1 sub_seq1 ##1 (do1 == v1); // error since v1 is not visible
endsequence
```

To access a local variable of a sub-sequence, a local variable must be declared and passed to the instantiated sub-sequence through an argument. An example below illustrates this usage.

```
sequence sub_seq2(lv);
   a ##1 !a, lv = data_in ##1 !b*[0:$] ##1 b && (data_out == lv);
endsequence
sequence seq2;
   int v1;
   c ##1 sub_seq2(v1) ##1 (do1 == v1); // v1 is now bound to lv
endsequence
```

Note that when a local variable is a formal argument of a **sequence** definition, it is illegal to declare the variable, as shown below.

```
sequence sub_seq3(lv);
   int lv; // illegal since lv is a formal argument
   a ##1 !a, lv = data_in ##1 !b*[0:$] ##1 b && (data_out == lv);
endsequence
```

There are special considerations on using local variables in parallel branches using operators **`or`**, **`and`**, and **`intersect`**.

21) Section 17.9, pp. 182

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

22) Section 17.9, pp. 181

— `$onehot0`(<expression>) returns true if at most one bit of the expression is ~~low~~ high.

23) Section 22.7, pp. 236

— `$onehot0`(<expression>) returns true if at most one bit of the expression is ~~low~~ high.

24) Syntax Box 17-11, pp. 173
| ~~sequence_expr~~ expression **throughout** sequence_expr

25) Section 17.7.8, pp.173

expression ~~*sequence_expr* is an expression which~~ must evaluate true at every clock tick during the evaluation of *sequence_expr*. ~~If a sequence for *sequence_expr* starts at time `t1` and ends at time `t2`, then expression must hold true from time `t1` to `t2`. If either the *sequence_expr* does not match or the expression becomes false while the *sequence_expr* is being evaluated, the sequence does not match.~~ If an evaluation of *sequence_expr* starts at time `t1` and ends with a match at time `t2`, then for *sequence_expr* to match, *expression* must hold true from time `t1` to `t2`.

26) Section 17.7.9, pp. 174

The **within** construct

sequence_expr1 ***within*** *sequence_expr2*

is an abbreviation for writing:

*(1[*0:$] ##1 ~~**sequence_expr1**~~ sequence_expr1 ##1 1[*0:$])* ***intersect*** *sequence_expr2*

27) Section 17.7.9, pp. 174

matches on the trace shown in ~~Figure 17-13~~ Figure 17-12.

28) Section 17.7.11, pp. 177

A property written to express this condition is shown below.

```
~~‘define data_end_exp (data_phase && ((irdy==0) && ($fell(trdy) ||~~
~~$fell(stop))))~~
‘define data_end_exp (data_phase && ((irdy==0)&&($fell(trdy)||$fell(stop))))
property data_end_rule1;
   @(posedge mclk)
   ‘data_end_exp |-> ##[1:2] $rose(frame) ##1 $rose(irdy);
endproperty
```

29) Section 17.7.11, pp. 179

```
property p16;
   (write_en & data_valid) ##0
   (write_en && (retire_address[0:4]==addr)) ~~[*1]~~ [*2] |->
   ##[3:8] write_en && !data_valid &&(write_address[0:4]==addr);
endproperty
```

30) Syntax box 17-5, pp. 152

goto_repetition ::= [*-> const_or_range_expression]

31) Syntax box 17-21

procedural_assertion_item ::=

~~concurrent_~~assert_property_statement

| ~~concurrent_~~cover_property_statement

...

concurrent_assert_statement ::=

[block_identifier:] ~~**assert property (** property_spec**)** action_block~~ assert_property_statement

~~| [block_identifier:] **assert property (** property_instance**)** action_block~~

concurrent_cover_statement ::=

| [block_identifier:] ~~**cover property (** property_spec**)** statement_or_null~~cover_property_statement

~~| [block_identifier:] **cover property (** property_instance**)** statement_or_null~~

~~concurrent_assert_statement~~assert_property_statement::=

~~[block_identifier:]~~ **assert property (** property_spec**)** action_block

| ~~[block_identifier:]~~ **assert property (** property_instance**)** action_block

~~concurrent_cover_statement~~ cover_property_statement::=

| ~~[block_identifier:]~~ **cover property (** property_spec**)** statement_or_null

| ~~[block_identifier:]~~ **cover property (** property_instance**)** statement_or_null

~~property_instance ::=~~

~~property_identifier [(actual_arg_list)]~~

concurrent_assertion_item ::=

concurrent_assert_statement

| concurrent_cover_statement

...

32) Section 17.12, pp. 187

The action_block shall not include any concurrent **assert** or **`cover`** statement. The action_block, however, may contain immediate assertion statements.

Note: The pass and fail statements are executed in the reactive region. The regions of execution are explained in the scheduling semantics section, Section 14.

33) Section 17.14, pp. 195:
— `fpu_props` is the name of the program containing properties.
— `fpu_rules_1` is the program instance name.

BNF

1. Section A.2.10 , pp. 289

goto_repetition ::= [*-> const_or_range_expression]

2. Section A.6.10, pp.298

procedural_assertion_item ::=

~~concurrent_~~assert_property_statement

| ~~concurrent_~~cover_property_statement

| immediate_assert_statement

immediate_assert_statement::=

assert (expression **)** action_block

~~concurrent_assert_statement~~ ::=

~~[block_identifier:] **assert property (** property_spec**)** action_block~~

| ~~[block_identifier:] **assert property (** property_instance**)** action_block~~

~~concurrent_cover_statement~~ ::=

| ~~[block_identifier:] **cover property (** property_spec**)** statement_or_null~~

| ~~[block_identifier:] **cover property (** property_instance**)** statement_or_null~~

~~property_instance ::=~~

~~property_identifier [(actual_arg_list)]~~

3. Section A.2.10, pp.288

add

concurrent_assert_statement ::=

[block_identifier:] assert_property_statement

concurrent_cover_statement ::=

| [block_identifier:] cover_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

property_instance ::=

property_identifier [**(** actual_arg_list **)**]