## 12.6 Event

In Verilog, named events are static objects that can be triggered via the -> operator, and processes can wait for an event to be triggered via the @ operator. SystemVerilog events support the same basic operations, but enhance Verilog events in several ways. The most salient enhancement is that the triggered state of Verilog named events has no duration, whereas in SystemVerilog this state persists throughout the time-step on which the event is triggered. Also, SystemVerilog events act as handles to synchronization queues, thus, they can be passed as arguments to tasks, and they can be dynamically allocated and reclaimed.

Existing Verilog event operations (@ and ->) are backward compatible and continue to work the same way when used in the static Verilog context. The additional functionality described below works with all events in either the static or the dynamic context.

A SystemVerilog event provides a handle to an underlying synchronization object. When a process waits for an event to be triggered, the process is put on a queue maintained within the synchronization object. Processes can wait for a SystemVerilog event to be triggered via the @ operator, or using the **wait**() construct to examine their triggered state. Events are always triggered using the -> operator.

The syntax to declare named events is discussed in section 3.9.

### 12.6.1 Triggering an Event

Named events are triggered via the -> operator.

The syntax to trigger an event is:

```
-> event_identifier;
```

Triggering an event unblocks all processes currently waiting on that event. When triggered, named events behave like a one-shot, that is, the trigger state itself is not observable, only its effect. This is similar to the way in which an edge can trigger a flip-flop but the state of the edge can not be ascertained, i.e., if( posedge clock ) is illegal.

### 12.6.2 Waiting for an Event

The basic mechanism to wait for an event to be triggered is via the event control operator: @.

The syntax for this use of the @ operator is:

```
@ event_identifier;
```

The @ operator blocks the calling process until the given event is triggered.

For a trigger to unblock a process waiting on an event, the waiting process must execute the @ statement before the triggering process executes the trigger operator, ->. If the trigger executes first then the waiting process remains blocked.

### 12.6.3 Persistent Trigger: triggered property

SystemVerilog can distinguish the event trigger itself, which is instantaneous, from the event's triggered state, which persists throughout the time-step (i.e., until simulation time advances). The **triggered** event property allows users to examine this state.

The syntax for invoking the triggered property uses a method-like syntax:

```
event_identifier.triggered
```

The triggered event property evaluates to true if the given event has been triggered in the current time-step and false otherwise.

The triggered event property is most useful when used in the context of a **wait** construct:

**wait** ( event_identifier.**triggered** )

Using this mechanism, an event trigger will unblock the waiting process whether the **wait** executes before or at the same simulation time as the trigger operation. The trigger property, thus, helps eliminate a common race condition that occurs when both the trigger and the **wait** happen at the same time. A process that blocks waiting for an event may or may not unblock depending on the execution order of the waiting and triggering processes. However, a process that waits on the triggered state always unblocks, regardless of the order of execution of the **wait** and trigger operations.

Example:

```
event done, blast;            // declare two new events
event done_too = done;        // declare done_too as alias to done

task trigger( event ev );
    -> ev;
endtask
...
fork
    @ done_too;         // wait for done through done_too
    #1 trigger( done ); // trigger done through task trigger
join

fork
    -> blast;
    wait( blast.triggered );
join
```

The first fork in the example shows how two event identifiers done and done_too refer to the same synchronization object, and also how an event can be passed to a generic task that triggers the event. In the example, one process waits for the event via done_too, while the actual triggering is done via the trigger task that is passed done as an argument.

In the second fork, one process may trigger the event `blast` before the other process (if the processes in the **`fork`**…**`join`** execute in source order) has a chance to execute, and wait for the event. Nonetheless, the second process unblocks and the fork terminates. This is because the process waits for the event's triggered state, which remains in its triggered state for the duration of the time-step.

## 12.7 Event sequencing

### 12.7.1 wait_order()

The **wait_order** construct suspends the calling process until all the specified events are triggered in the given order (left to right), or any of the un-triggered events is triggered out of order and causes the construct to fail.

The syntax for the **wait_order** construct is:

```
wait_order( event_identifier[.triggered] {, event_identifier} ) [=> variable]
```

Events are not limited to occur only once, that is, once an event occurs in the prescribed order, it can be triggered again without causing the construct to fail. Only the first event in the list can wait for the triggered state.

The action taken when the construct fails depends on whether the phrase '=> variable' is specified or not. If it is specified then the completion status of the construct is assigned to the given variable: 1 for success, and 0 for failure. If the phrase '=> variable' isn't specified, a failure generates a run-time error.

For example:

```
wait_order( a, b, c );
```

Suspends the current process until events a, b, and c trigger in the order a –> b –> c. If the events trigger out of order, a run-time error is generated.

```
bit success;
wait_order( a, b, c ) => success;
if( ! success )
  $display( "Error: events out of order" );
```

In this example, the completion status is stored in the variable `success`, which is then examined to display a message, but without an error being generated.

## 12.8 Event variables

An event is a unique data type with several important properties. Unlike Verilog, SystemVerilog events can be assigned to one another. When one event is assigned to another the synchronization queue of the source event is shared by both the source and the destination event. In this sense, events act as full-fledged variables and not merely as labels.

### 12.8.1 Disabling Events

If an event variable is assigned the special **null** value, the event is ignored in subsequent event control operations. That is, when the event is set to **null**, no process can wait for the event again.

For example:

```
event E1 = null; subsequent
@ E1;
```

The statement @ E1 does not block because event E1 is no longer blocking.

### 12.8.2 Merging Events

When one event variable is assigned to another, the two become merged. Thus, executing -> on either event variable affects processes waiting on either event variable.

For example:

```
event a, b, c;
a = b;
-> c;
-> a;    // also triggers b
-> b;    // also triggers a
a = c;
b = a;
-> a;    // also triggers b and c
-> b;    // also triggers a and c
-> c;    // also triggers a and b
```

When merging events, the assignment only affects subsequent executions of ->and wait(). If a process is locked waiting for event1 when another event is assigned to event1, the currently waiting process will never unblock. For example:

```
fork
   T1: while(1) @ E2;
   T2: while(1) @ E1;
   T3: begin
         E2 = E1;
         while(1) -> E2;
   end
join
```

This example forks off three concurrent processes. Each process starts at the same time. Thus, at the same time that process T1 and T2 are blocked, process T3 assigns event E1 to E2. This means that process T1 will never unblock, because the event E2 is now E1. To unblock both threads T1 and T2, the merger of E2 and E1 must take place before the fork.