12.6 Event

In Verilog, named events are static objects that can be triggered via the -> operator, and processes can ~~block until~~ wait for an event ~~is~~ to be triggered via the @ operator. SystemVerilog events support the same basic operations, but enhance Verilog events in several ways. The most salient ~~semantic difference~~ enhancement is that the triggered state of Verilog named events ~~do not have a value or~~ has no duration, whereas in SystemVerilog ~~events can have a persisteney that lasts~~ this state persists throughout the time-step on which ~~they are~~ 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 ~~FIFO~~ queue maintained within the synchronization object. Processes can wait for a SystemVerilog event to be triggered ~~either~~ via the @ ~~operator~~ operator, or using the wait() ~~construct~~construct to examine their triggered state. Events are always triggered using the -> operator.
~~Events are always triggered using the -> operator.~~

~~SystemVerilog provides for two different types of events: persistent events and non-persistent events. These~~
~~two are described below.~~
The syntax to declare named events is discussed in section 3.9.

12.6.1 ~~Non-Persistent Events~~Triggering an Event

~~Non-persistent events are the same as named Verilog events. They behave like a one-shot, that is, their triggered state 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: if(posedge clock) is illegal.~~
Named events are triggered via the -> operator.

~~Triggering a non-persistent event causes all processes currently waiting on the event to unblock. For a trigger to unblock a process that is waiting on non-persistent event, that process must execute the wait (or @) before the triggering process executes the trigger operator, ->. A process that executes wait() on a non-persistent event after the event has been triggered will block.~~

The syntax to ~~declare a non-persistent~~ trigger an event is:

```
~~event~~ -> 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 ~~Persistent Events: event bit~~Waiting for an Event

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

~~Persistent events are similar to non-persistent events except that once triggered, the triggered state persists throughout the time-step, that is, until simulation time advances. Thus, a persistent event will unblock all processes that execute the wait() construct either before or at the same simulation time as the trigger operation.~~

~~The persistent trigger behavior helps eliminate a common race condition that occurs when both the trigger and the wait operations happen at the same time. A process that blocks on a regular (non-persistent) event may or may not unblock depending on the execution order of the waiting and triggering processes, while a persistent event always unblock the waiting process, regardless of the order of execution.~~

The syntax ~~to declare a persistent event~~ for this use of the @ operator is:

```
~~event bit~~ @ event_identifier;
```

~~Persistent and non-persistent events support the same set of operators, but they are different types. A persistent event may only be assigned (or passed as an argument) to another persistent event and vice-versa (see Section 11.6.2).~~

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

~~12.6.3 Triggering an Event~~

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.

~~All events regardless of their type (persistent or non-persistent) are triggered via the -> operator.~~

~~The syntax to trigger an event is:~~

```
~~-> event_identifier;~~
```

~~If the event_identifier is a persistent event then the event will remain in the triggered state until the simulation time advances. Otherwise, the persistent state is unobservable.~~

~~Triggering a persistent event more than once at the same simulation time has no effect. However, triggering a non-persistent event more than once, at the same simulation time, results in multiple triggers.~~

12.6.~~4 Waiting for an Event~~3 Persistent Trigger: triggered property

~~There are two mechanisms that can be used to wait for an event. The @ operator and the wait construct.~~

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 this use of the @ operator is:~~

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

```
~~@ event_identifier;~~
event_identifier.triggered
```

~~The @ operator always blocks the calling process until an event is triggered.~~
The triggered property evaluates to true if the given event has been triggered in the current time-step and false otherwise.

~~The syntax for this use of the wait construct is:~~
The triggered event property is most useful when used in the context of a **wait** construct:

wait (event_identifier.**triggered**)

```
wait( event_identifier );
```

~~The wait construct blocks if the given event is a non-persistent event or the persistent event has not been trig-gered at the current simulation time.~~

~~Both mechanisms can be used to wait for either a persistent or a non-persistent event. The wait construct is only meaningful when the event is persistent.~~
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.

~~Examples~~Example:

```
event done;, blast;// declare a two new eventevents
event done_too = done;;// declare done_too as alias to done
event bit blast; // persistent event

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

fork
    -> blast;// trigger blast event
    wait( blast );// wait for blast event
    wait( blast.triggered );
join
```

The first fork in the ~~examples~~ 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 ~~will trigger either~~ triggers the event. In the ~~example~~ 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.

~~When~~ In the second ~~fork executes~~fork, ~~the first~~ one process may ~~triggers~~ trigger the event `blast` before the ~~second~~ other process (~~assuming~~ if the processes in ~~a~~ the **fork**…**join** execute in source order) has a chance to ~~execute~~ execute, and wait for the event. ~~Nonetheless~~ Nonetheless, the second process unblocks

and the fork terminates. This is because ~~blast is a persistent event so it~~ the process waits for the event's triggered state, which remains in its triggered state for the duration of the time-step~~. Note that if blast were declared as a non-persistent event the second process would never unblock~~.

12.7 Event synchronization utilities

12.7.1 $wait_all()

The $wait_all system tasks suspends the calling process until all of the specified events are triggered. This task implicitly waits for the triggered state of each event.

The syntax for the $wait_all task is:

```
$wait_all( event_identifier {, event_identifier } )
```

For example:

```
$wait_all( a, b, c );
```

suspends the current process until the ~~3~~ three events a, b, and c are triggered.

Any of the specified events may be triggered more than once; the only requirement to unblock the calling process is that each event be triggered at least once.

12.7.2 $wait_any()

The $wait_any system tasks suspends the calling process until any of the specified events are triggered. This task implicitly waits for the triggered state of each event.

The syntax for the $wait_any task is:

```
$wait_any( event_identifier {, event_identifier } )
```

For example:

```
$wait_any( a, b, c );
```

suspends the current process until either event a, or event b, or event c is triggered.

12.7.3 $wait_order()

The $wait_order system task suspends the calling process until all the specified events are triggered (similar to $wait_all), but the events must be triggered in the given order (left to right). If an event is received out of order, the process unblocks and generates a run-time error.

The syntax for the $wait_order task is:

```
$wait_order( ~~event_identifier~~ event_identifier[.triggered] {,
event_identifier } )
```

Events are not limited to occur only once, that is, once an event occurs in the prescribed order, it can be triggered again without generating an error.
~~When $wait_order is called, only~~ Note: Only the first event in the list can ~~be in~~ wait for the triggered state.
~~If any other persistent event is in triggered state, it generates a run-time error.~~

For example:

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

suspends the current process until events trigger in the order a –> b –> c.

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 calls to wait(). That is, when the event is set to **null**, no process can wait for the event again.

For example:

```
event E1 = null;
@ 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 blocked waiting for event1 when another event is assigned to event1, the wait 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.