VHDL
Signal Assignment and Resolved Signals

1. Signal Assignment Statement
2. Delay Mechanism
3. Transport Delay
4. Inertial Delay
5. Resolved Signals
6. Guarded Signals

Signal Assignment Statement

The projected output waveform stored in the driver of a signal can be modified by a signal assignment statement.

signal_assignment_statement ::= target <= [transport | reject time_expression] inertial waveform;

waveform ::= waveform_element {, waveform_element}

waveform_element ::= value_expression [after time_expression]

S <= transport 100 after 20 ns, 15 after 35 ns;
S <= 1 after 20 ns, 15 after 35 ns;

• The concrete way a driver is updated as result of a signal assignment, depends on the delay mechanism (transport or inertial).
• The delay mechanism can be explicitly specified as part of the signal assignment; if no mechanism is specified, the default is inertial.

Transport Delay

Transport delay models devices that exhibit nearly infinite frequency response: any pulse is transmitted, no matter how short its duration.

This is typical when modeling transmission lines.

No transaction scheduled to be executed before a new one is affected by a signal assignment with transport delay.

Update rule:
1. All old transactions scheduled to occur at the same time or after the first new transaction are deleted from the projected waveform.
2. The new transactions are appended to the end of the driver.

Examples

Consider the following assignments executed at simulation time 100 ns (the projected waveform, at that moment, consists of a single transaction with value 0):

S <= transport 100 after 20 ns, 15 after 35 ns;
S <= transport 10 after 40 ns;
S <= transport 25 after 38 ns;

Driver for S after first two assignments:

<table>
<thead>
<tr>
<th>Value</th>
<th>0</th>
<th>100</th>
<th>15</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>100 ns</td>
<td>120 ns</td>
<td>135 ns</td>
<td>140 ns</td>
</tr>
</tbody>
</table>

Driver for S after last assignment:

<table>
<thead>
<tr>
<th>Value</th>
<th>0</th>
<th>100</th>
<th>15</th>
<th>25</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>100 ns</td>
<td>120 ns</td>
<td>135 ns</td>
<td>138 ns</td>
</tr>
</tbody>
</table>

• Every change on the input will be processed, regardless of how short the time interval between this change and the next one.
Inertial Delay

Inertial delay models the timing behavior of current switching circuits: an input value must be stable for a certain duration, called pulse rejection limit, before the value propagates to the output.

\[ S \leftarrow \text{reject } 5 \text{ ns inertial } X \text{ after } 10 \text{ ns}; \]

- Additional update rule (after update operations have been performed exactly like for transport delay):

All old transactions scheduled to occur at times between the time of the first new transaction and this time minus the pulse rejection limit are deleted from the projected waveform; excepted are those transactions which are immediately preceding the first new transaction and have the same value with it.

Inertial Delay (cont'd)

- If no pulse rejection limit is specified, it is considered to be equal with the time value in the first waveform element

\[ S \leftarrow X \text{ after } 10 \text{ ns, } 0 \text{ after } 25 \text{ ns}; \]

is equivalent to:

\[ S \leftarrow \text{reject } 10 \text{ ns inertial } X \text{ after } 10 \text{ ns, } 0 \text{ after } 25 \text{ ns}; \]

Examples

Consider the assignments below, executed at simulation time 100 ns, when the driver for signal S has the following contents:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 ns</td>
<td>110 ns</td>
<td>135 ns</td>
<td></td>
</tr>
</tbody>
</table>

\[ S \leftarrow 8 \text{ after } 20 \text{ ns, } 2 \text{ after } 40 \text{ ns, } 5 \text{ after } 65 \text{ ns, } 10 \text{ after } 100 \text{ ns}; \]

\[ S \leftarrow \text{reject } 55 \text{ ns inertial } 5 \text{ after } 90 \text{ ns}; \]

Driver for S after first assignment:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>8</th>
<th>2</th>
<th>5</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 ns</td>
<td>120 ns</td>
<td>140 ns</td>
<td>165 ns</td>
<td>200 ns</td>
<td></td>
</tr>
</tbody>
</table>

Driver for S after second assignment:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>8</th>
<th>5</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 ns</td>
<td>120 ns</td>
<td>165 ns</td>
<td>190 ns</td>
<td></td>
</tr>
</tbody>
</table>
Resolved Signals and Resolution Functions

- **Resolved signal**: a signal for which several drivers exist (several processes assign values to that signal). For each resolved signal the designer has to specify an associated resolution function.

- The resolution function computes the value which is used to update the current signal value, depending on the actual values of the drivers.

- The resolution function is automatically called by the simulation kernel every time the signal value has to be updated.

### Example

A resolved signal, `Line`, which models an interconnection line to which the output of several devices is connected. Each device is modeled by one process.

The resolution function implements a **wired or**.

```plaintext
architecture Example of ... is
  type Bit4 is ('X','0','1','Z');
  type B_Vector is array(Integer range <>) of Bit4;

  function Wired_Or(Input: B_Vector) return Bit4 is
    variable Result: Bit4:='0';
    begin
      for I in Input'Range loop
        if Input(I)='1' then
          Result:='1';
          exit;
        elsif Input(I)='X' then
          Result:='X';
        end if;
      end loop;
      return Result;
    end Wired_or;
end
```

Example

```plaintext
signal Line: Wired_Or Bit4;
begin
  P1: process
  begin
    -- -- -- --
    Line <= '1';
    -- -- -- --
  end process;

  P2: process
  begin
    -- -- -- --
    Line <= '0';
    -- -- -- --
  end process;
end Example.
```

- Each time a resolution function is invoked by the simulation kernel, it is passed an array value, each element of which is determined by a driver of the corresponding resolved signal.
Guarded Signals

- A **guarded signal** is a resolved signal which has been declared to be of class **register** or **bus**.

```vhd
subtype BIT_8 is BIT_VECTOR(7 downto 0);
signal Connect: Bus_resolution BIT_8 bus;
```

- Only resolved signals can be guarded signals

The special thing about guarded signals:

- Guarded signals (and only they) can have drivers disconnected!

Guarded Signals (cont’d)

What means a driver of a guarded signal to be disconnected?

- A disconnected driver does not influence the current value of the signal.

Why do we need something like this?

- Guarded signals are an elegant way to model devices which are driven by several sources some of which can, temporarily, be turned off.

How can we disconnect a driver?

- A driver is disconnected by assigning to the guarded signal the value null in a sequential signal assignment.
- A driver is automatically disconnected as result of a guarded signal assignment (inside a block statement), whenever the guard associated to the block is false (we have not discussed block statement and guarded signal assignments here).

```
architecture Example of ... is
subtype BIT_8 is BIT_VECTOR(7 downto 0);
type B8_Vector is array(Integer range <>) of Bit_8;
function Bus_resolution(Input: B8_Vector)
return Bit_8 is
variable Result: Bit_8:=’00000000’;
begin
  -- if Input is void (Input’Range=0),
  -- because all input drivers are dis-
  -- connected, the for is not executed
  for I in Input’Range loop
      . . . . . .
      Result := .......
      . . . . . .
  end loop;
  return Result;
end Bus_resolution;
```

Guarded Signals (cont’d)

signal Connect: Bus_resolution BIT_8 bus;
begin
P1: process
begin
  -- - - - - -
  Connect <= null after 10ns;
  -- - - - - -
end process;

P2: process
begin
  -- - - - - -
  Connect <= “01010010” after 20ns;
  -- - - - - -
end process;
end Example.
Guarded Signals (cont’d)

At activation of the resolution function, disconnected drivers (whose current value is null) are ignored.

What happens if all drivers are disconnected?

This is the situation where there is a difference between resolved signals of class register and bus.

- **Bus signals**: the resolution function is activated with a void argument; the function returns a value which becomes the new current value of the signal.
- **Register signals**: the resolution function is not activated; the signal keeps its current value.

Summary

- Signal assignment statements can be executed according to two different delay mechanisms: transport and inertial.

- Transport delay, models devices with nearly infinite frequency response. Such are e.g. interconnection lines. With inertial delay, an input value must be stable for a certain duration before it propagates to the output.

- A signal to which several processes assign values is a resolved signal. For such a signal several drivers are created.

- A resolution function has to be defined for each resolved signal. It is automatically activated by the simulation kernel every time the signal value has to be updated.

- Guarded signals are a special category of resolved signal. They can have their drivers disconnected.
  A disconnected driver will be ignored when activating the resolution function

- A guarded signal of class register keeps its current value when all its drivers are disconnected. A signal of class bus, with all drivers disconnected, will get a new value which is determined by the resolution function.