Simulation

- **Simulation** is the process of conducting experiments on a model of a system for the purpose of understanding or verifying the operation of the actual system.

Simulation in the VHDL/PLD methodology:
- Functional verification verifies the functional operation of a description.
- Post-synthesis simulation verifies that the synthesizer accurately translated the description to logic.
- Timing simulation verifies that the synthesized logic, when mapped to the target PLD, will meet the system's requirements.
- Construct semantics are defined in the IEEE Std 1076 Language Reference Manual (LRM) in terms of how an event-driven simulator must execute the constructs.

Time of execution of simulation cycles

![Time of execution of simulation cycles](image)

Steps in an event-driven simulation

The LRM (Language Reference Manual) defines how an event-driven simulator must execute the VHDL. Simulator vendors implement this concept.

An event-driven simulator performs three steps to accomplish a simulation:
1) elaboration
2) initialization
3) repeated execution of simulation cycles

Elaboration

- In VHDL the statement part of an architecture body can contain only concurrent statements.
- The process statement is the basic behavioral statement.
- Every concurrent VHDL statement has an equivalent process representation.
- We use the term simulation process to mean a process that is equivalent to some concurrent statement in a design entity's architecture and is used for the simulation of that statement.
- Elaboration is the creation of a simulation model for a design entity from its VHDL description. This simulation model consists of a net of simulation processes.
- During elaboration, all concurrent statements are converted to equivalent simulation processes.
Flattening a hierarchical structure

- Any VHDL program can ultimately be viewed as a collection of simulation processes communicating through signals, that is, a simulation net. The resulting simulation net is executed to simulate the design entity's behavior.
- During elaboration a design that has a hierarchical structure is flattened until entire design is described by a simulation net.
- In addition, all of the data objects (an object is a named item that has a value of a specified type) declared in the description are created.

A single inverter circuit to be simulated

To illustrate the simulation of hierarchical design, let's consider trivial structure that consists of the instantiation of a single inverter component. The design file contains two design entities.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
entity not_gate is -- not gate entity with name not_gate
port (x : in std_logic ; o : out std_logic) :
end;
architecture dataflow of not_gate is
begin
  o <= not x ;
end dataflow ;
```

A single inverter circuit to be simulated

```vhdl
library ieee ;
use ieee.std_logic_1164.all ;
entity ckt is -- top level entity
port (a : in std_logic ; f : out std_logic) ;
end ;
architecture struct of ckt is -- top level architecture
begin
  u0 : entity not_gate port map (x  =>  a,  o  =>  f ) ;
end struct ;
```

The second design entity, ckt, is the circuit whose operation we wish to verify.

Testbench for inverter circuit

To simulate ckt we use a testbench named testbench. Design entity ckt is the UUT in design entity testbench.

```vhdl
library ieee ;
use ieee.std_logic_1164.all ;
entity testbench is
end testbench ;
architecture behavior of testbench is
signal a  :  std_logic;
signal f  :  std_logic ;
begin
  u0 : entity ckt port map (a => a,  f => f ) ;
end process ;
```

Testbench for inverter circuit

```vhdl
tb : process
constant period : time  :=  20 ns ;
begin
  wait for period ; -- Wait 20 ns after initialization
  a  <=  '1' ;
  wait for period ; -- Wait another 20 ns
  assert ( f  =  '0' ) ;
  a  <=  '0' ;
  report fltest failed@ severity error ;
  wait for period ; -- Wait another 20 ns
  assert ( f  =  '1' ) ;
  report fltest failed@ severity error ;
end process ;
end behavior ;
```

Relationship of the testbench to the circuit

Design entity not_gate has dataflow architecture which is behavioral. Thus, not_gate is a primitive component. The elaboration create a simulation process uut/u0, which represents an instance of not_gate (u0) in an instance of ckt(uut). In the simulation process, the label uut/u0 represents the hierarchical path to u0. A conceptual representation of the simulation process uut/u0 is:
**Signal drivers**

- **Driver** is a source (contributor) to the value of a signal. During elaboration only one driver is created for each signal assigned a value in a simulation process.
- If two (or more) different simulation processes contain assignments to the same signal than this signal is multiply driven (drivers are associated with each simulation process).
- A signal driver is modeled as a queue.
- Each time a signal assignment statement is executed, the simulator posts (schedules) a transaction in the signal's driver. Each transaction is a pair consisting of a value component and a time component.
- Transactions are ordered in a signal's driver queue based on their time component. The ordered set of transactions specifies projected values (projected waveforms).

**Simulator kernel process**

During a simulation, the kernel process coordinates the activities of the simulation processes. The kernel process is responsible for detecting events and causing the appropriate simulation processes to execute in response to those events.

The kernel process maintains the current value variable for each signal in the simulation model. It utilizes the resolution function to determine a multiple driven signal's current value from its drivers' current values.

A simulator program is typically run on a host computer with a single CPU (only one simulation process can actually be executing at a time). So, the simulator must execute simulation processes sequentially, but in a way that models concurrent operation.

**States of a simulation process**

Three states:
- **Suspended**: simulation process is not running or active.
- **Active**: simulation process is in the active processes queue waiting to be executed.
- **Running**: simulator is executing the simulation process.

**Active and waiting processes queues**

All simulation processes that need to be run during a particular simulation cycle are placed in the active processes queue by the kernel. The kernel runs each simulation process from the queue and executes it until it suspends. If a simulation process is suspended because of a wait statement of the form `wait for time_expression`, that simulation process is put in the waiting processes queue in time order. A simulation process in the waiting processes queue is referred to as a scheduled process because it is scheduled to be resumed (be placed in the active processes queue) at a specific future time.

When a simulation process without a sensitivity list or wait statement is simulated, then such a process never suspends (however syntactically it is legal).

**Simulation initialization**

At the beginning of the initialization phase, the current simulation time ($T_c$) is set to 0.

The kernel places all of the simulation processes in the active processes queue. The initial execution of each simulation process ensures that all initial transactions are scheduled, so than the simulation may continue.
Delta delays. Two-dimensional view of time

Testbenches can use signal assignment with after clause. Example: \( x <= a \) and \( b \) after 2 ns; If a signal assignment without an after clause is executed, the associated transaction is assigned a time component \( \delta \) delay greater than the current simulation time. Since a delta delay is infinitesimal, when added to the current simulation time it does not advance the numerical time value. Conceptually, simulation time is viewed as two-dimensional.

Simulation cycles: update phase

After the initialization phase, all simulation processes are in suspended states. The simulation cycle is then executed. A simulation cycle consists of two phases: an update phase and an execution phase. The update phase first determines the next value for the current simulation time. Based on this new simulation time, a determination is made as to whether the simulation is complete. A change in a signal's value can occur only during the update phase. If a signal update results in an event, all simulation processes sensitive to this event are placed in the active processes queue. In addition, all simulation processes that are scheduled to resume at the current simulation time are placed in the active processes queue.

Simulation cycles: execution phase

Each simulation process in the active processes queue is executed until it suspends. Execution of a signal assignment statements causes transactions to be scheduled for specified signal's driver. The order in which the kernel takes simulation processes from the active processes queue and executes them is not important, because these processes are concurrent.

When do signals take new values?

- It is not until simulation process suspends that the transactions resulting from signal assignment in that process are placed in the appropriate signal driver queues.
- It is not until the update phase of the next simulation cycle, after all the simulation processes that were in the active processes queue during the current simulation cycle have been executed and suspended, that any signals assigned values by these processes are updated.
- This means that the assignment of a new value to a signal cannot take effect until the update phase of the next simulation cycle. This is after the process containing the assignment has suspended. Therefore, the simulation process does not see the effect of any signal assignment until the next time it resumes.

Summary of simulation cycle steps

1. The time of the next simulation cycle is determined. If \( T_n = T_c + \delta \), the next cycle is a delta simulation cycle.
2. Simulation is complete if \( T_n = \text{TIME}=\text{HIGH} \) and there are no signal drivers with transactions scheduled for \( T_n \) (active drivers) or simulation processes scheduled to resume at \( T_n \).
3. If the simulation is not complete, the current simulation time is set to \( T_n \).
4. The value of all signals having transactions scheduled for \( T_c \) are updated. Events may occur on some signals as a result of updating their values.
5. All processes that are sensitive to the events in (4) are put in the active processes queue.
6. All processes scheduled to resume at \( T_c \) are also put in the active processes queue.
7. Each simulation process is taken from the active processes queue and executed until it suspends. Execution of simulation processes may cause new transactions to be posted in signal drivers.
8. Return to step 1.
Simulation cycles for inverter example

<table>
<thead>
<tr>
<th>Signal</th>
<th>Value</th>
<th>α</th>
<th>Signal Driver Transaction Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>After elaboration</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>'U'</td>
<td>0</td>
<td>'U' @ 0</td>
</tr>
<tr>
<td>f</td>
<td>'U'</td>
<td>0</td>
<td>'U' @ 0</td>
</tr>
<tr>
<td>After initialization Tc = 0, Tn = 0 + 1 h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>'U'</td>
<td>0</td>
<td>'U' @ 0</td>
</tr>
<tr>
<td>f</td>
<td>'U'</td>
<td>0</td>
<td>'U' @ 0 + 1 h</td>
</tr>
<tr>
<td>After simulation cycle Tc = 0 + 1 h, Tn = 20</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>'U'</td>
<td>0</td>
<td>'U' @ 0</td>
</tr>
<tr>
<td>f</td>
<td>'U'</td>
<td>0</td>
<td>'U' @ 0 + 1 h</td>
</tr>
</tbody>
</table>

Simulation cycles for inverter example

<table>
<thead>
<tr>
<th>After simulation cycle Tc = 20, Tn = 20 + 1 h</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
</tr>
<tr>
<td>f</td>
</tr>
<tr>
<td>After simulation cycle Tc = 20 + 1 h, Tn = 20 + 2 h</td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>f</td>
</tr>
<tr>
<td>After simulation cycle Tc = 20 + 2 h, Tn = 40</td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>f</td>
</tr>
</tbody>
</table>

Simulation cycles for inverter example

<table>
<thead>
<tr>
<th>After simulation cycle Tc = 40 + 1 h, Tn = 40 + 2 h</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
</tr>
<tr>
<td>f</td>
</tr>
<tr>
<td>After simulation cycle Tc = 40 + 2 h, Tn = 60</td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>f</td>
</tr>
<tr>
<td>After simulation cycle Tc = 60, Tn = done</td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>f</td>
</tr>
</tbody>
</table>

Waveform and simulation cycles of inverter

A simulation cycles list only shows deltas where an event occurred.

Signals versus Variables

Signals in an architecture:
- An initial value specified in a signal’s declaration is not synthesizable.
- Must be declared outside of any processes.
- Are visible to all processes (and other concurrent statements) in the architecture in which they are declared.
- Have current and projected (future) values.

Variables in an architecture:
- An initial value specified in a variable’s declaration in a process is not synthesizable.
- Can only be declared within process or subprograms.
- Are not visible outside of the process or subprogram in which they are declared.
- Variables are local to the process.
- Have only a current value.

Signals versus Variables

Signals in an architecture:
- Retain their values between executions of a process.
- A signal assignment schedules a transaction in the signal’s driver queue. If the signal assignment has no after clause, the assigned value takes effect at the update phase of the next simulation cycle (one δ later). The signal’s value is not changed during the current simulation cycle.

Variables in an architecture:
- Retain their values between executions of the process in which they are declared. Do not retain their values between executions of the subprogram in which they are declared.
- When a variable assignment statement is executed, the variable takes its new value immediately. Statements in the process that follow the variable assignment statement see its new value.