In behavioral style of modeling the behavior of the entity is expressed using sequentially executed, procedural type code. The key features of this modeling are -

- The behavioral modeling describes the system by showing how the outputs behave according to the changes in the inputs.
- While describing in the behavioral modeling, it is not necessary to know the logic diagram of the system but it is required to know how the output behaves in response to the change in the input.
- In VHDL, **process** is the main behavioral description statement.
- The statements inside the process are sequential.

Sequential Vs concurrent statements:

Sequential statements are those statements, where the order or sequence of writing the statements is important and defines the step by step execution, followed one after the other.

In concurrent style of modeling the digital circuit, the order of statements is not important. Data flow and structural style modeling follow concurrent statements.

Every entity is represented using an entity declaration and at least one architecture body.

Entity Declaration:

An entity declaration describes the external interface of the entity, that is, it gives the black-box view. It specifies the name of the entity, the names of interface ports, their mode (i.e., direction), and the type of ports. The syntax for an entity declaration is

entity *entity-name* **is**
[**generic** (*list-of-generics-and-their-types)* ;]
[**port** (*list-of-interface-port-names-and-their-types)* ;]
[*entity-item-declarations*]
[**begin**
entity-statements]
end [*entity-name*];

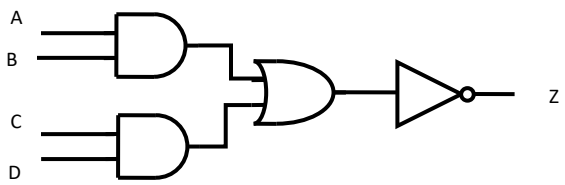
The *entity-name* is the name of the entity and the interface ports are the signals through which the entity passes information to and from its external environment. Each interface port can have one of the following modes:

1. **in:** Unidirectional port, indicating that the signal is an input and data can be written to.
2. **out:** Unidirectional port, indicating that the signal is an output of an entity whose value can be read. The value of an output port can only be updated within the entity model; it cannot be read.
3. **inout:** the value of a bidirectional port can be read and updated within the entity model.

4. **buffer:** the value of a buffer port can be read and updated within the entity model.

Example:

Consider an And-Or-Invert (AOI) circuit is shown in Fig. and its corresponding entity declaration is



entity AOI is

port (A, B, C, D: **in** BIT; Z: **out** BIT);

end AOI;

The entity declaration specifies that the name of the entity is AOI and that it has four input signals of type BIT and one output signal of type BIT.

Architecture Body

An architecture body describes the internal view of an entity. It describes the structure of the entity. Architecture consists of two portions:

- Architecture declaration and
- Architecture body.

The syntax of an architecture body is

architecture *architecture-name* **of** *entity-name* **is**

[*architecture-item-declarations]*

begin

Concurrent-statements; *these are —>*

Process-statement

Block-statement

Concurrent-procedure-call

Concurrent-assertion-statement

Concurrent-signal-assignment-statement

Component-instantiation-statement

generate-statement

end [*architecture-name*] ;

The architecture name is a user defined name of the architecture body. It can be same as entity name or different.

Process Statement

A process statement contains sequential statements that describe the functionality of a portion *of* an entity in sequential terms. The syntax of a process statement is

[*process-label:*] **process** [*(sensitivity-list)*]

[*process-item-declarations*]

begin

sequential-statements; *these are ->*

variable-assignment-statement

signal-assignment-statement

wait-statement

if-statement

case-statement

loop-statement

null-statement

exit-statement

next-statement

assertion-statement

procedure-call-statement

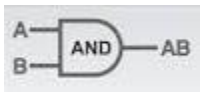
return-statement.

end process [*process-label]*;

Sensitivity list:

Sensitivity list is the set of signals to which the process is sensitive to (responsive). i.e., whenever an event occurs on any one of the signals in the sensitivity list, process comes into execution. The process is suspended only after executing all the statements inside the process in sequence.

For eg:



Consider the behavior model of AND gate. The signals A,B are included in the sensitivity list. So whenever the value of 'A' or 'B' or both changes from '0' to '1' or '1' to '0' , the process will start execution and the output of the gate is updated according to the expression C <= A and B;

Any change in the state of any element of the sensitivity list s treated as an event. The process is activated (initiated) on if an event occurs; otherwise process remains inactive. If the process has no sensitivity list, the process is executed continuously.

Variable Assignment Statement

Variables are the class of VHDL objects allowed only with the sequential style of modeling. Variables are objects that are used for the local storage within a process and subprogram alone. Inside a process local variables can be declared in the declarative part before the keyword **begin** to represent its local temporary values.

The first statement of the process statement is the variable assignment statement that assigns a value to variable **temp.** Variables can be declared and used inside a process statement. A variable is assigned a value using the variable assignment statement that typically has the form

Variable-object : = expression;

Differences between Signals and Variables

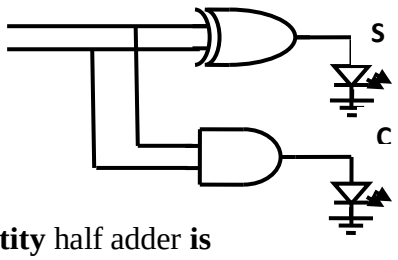
Signals	Variables
1. 1.These are VHDL objects used to represent wires and interconnections	These are temporary storage in VHDL
2. .Values of signals are updated only after default delta delay or specified user delay	Values of variables are updated immediately on the execution of variable assignment statement.
3. Require event scheduling and synchronizing of signal drivers.	No event scheduling and synchronizing is required.
4. Consume more memory space	Consume less memory space
5. Use of signals is allowed in styles of modeling	Variables are used only in behavioral modeling
6. Assignment operator is <=	Assignment operator is **:=**

[Write the VHDL code for the AOI circuit using Behavioral modeling].

```
entity AOI is
        port (A, B, C, D: in BIT; Z: out BIT);
end AOI;

architecture AOI of AOI is
        begin
            process (A, B, C, D)
            variable TEMPI ,TEMP2: BIT;
            begin
                TEMP1 := A and B;
                TEMP2:=C and D;
                TEMP1 := TEMP1 or TEMP2;
                    Z<= not TEMP1;
            end process;
            end AOI_SEQUENTIAL;
```

VHDL Behavioral description of Half adder



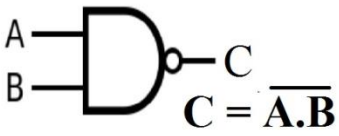

```
entity half adder is
```

```
port ( A : in bit;        B : in bit;
       Sum : out bit      Cout : out bit);
end  half_adder;

architecture adder of half_adder is
begin
      processs (A, B)
     begin
          sum <= A xor  B after 10ns;          -- signal assignment statement 1
          cout < = A and  B after 10ns         -- signal assignment statement 2
                                               -- with 10ns delay
    end process
end  adder;
```

Variable Assignment Statement: Examples

Write the VHDL code for two input nand gate using Behavioral modeling



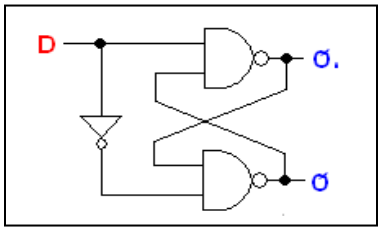

```
Entity nand2  is
Port      ( A : in bit;
            B: in  std-logic ;
            C : out std-logic);
End nand2;
```

```
Architecture nand2 of nand2 is
   Begin
        Process (A,B)

                Begin
                if A='1' and B='1' then
                C<= '0';
                 else
                C <= '1';
                End if;
                End process;
                End behavioral;
```

Write the VHDL code for D-latch using Behavioral modeling

```
entity D_latch is
                Port (D, Clk : in bit;
                      Q , Qbar : out bit);
end D_latch;
```




```
architecture behaviour of D_latch is
begin
```

```
process (D , Clk)
variable temp1 , temp2 : bit;
If Clk = '1' then
temp1 := D;                    --variable assignment statement.
temp2 := not temp1;            --variable assignment statement.
        end if ;
Q <= temp1;                    -- value of temp1 is assigned to Q
     Qbar <= temp2;            -- value of temp2 is assigned to Qbar.
end process;
end  D_latch;
```

Signal Assignment Statement:

A signal assignment statement can appear within a process or outside of a process. If it occurs outside of a process, it is considered to be a concurrent signal assignment statement. When a signal assignment statement appears within a process, it is considered to be a sequential signal assignment statement and is executed in sequence with respect to the other sequential statements that appear within that process.

When a signal assignment statement is executed, the value of the expression is computed and this value is scheduled to be assigned to the signal after the specified delay. If no delay is specified, the delay is assumed to be a default delta delay.

The syntax is

***Signal-object* <= *expression* [after *delay-value*];**

Example:

```
COUNTER <= COUNTER+ "0010";          - Assign after a delta delay.
  PAR <= PAR xor DIN after 12 ns;
  Z <= (A0 and A1) or (B0and B1) or (C0 and C1) after 6 ns;
```

Write the VHDL code for D-latch using Behavioral modeling:

```
entity D_latch is
Port (D, En : in bit;
      Q : buffer bit;
   Qbar : out bit);
end D_latch;
architecture DL of D_latch is
begin
                If En = '1' then
                  Q <= D;
                       Qbar <= not Q;
                end if ;
end process;
end  DL;
```

Delta Delay

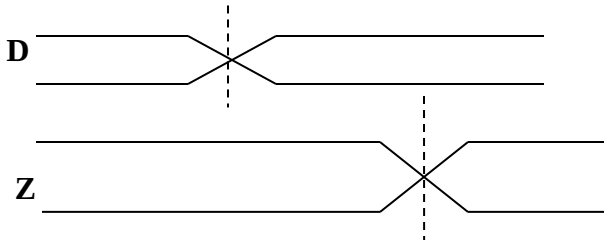
A *delta delay* *is* a very small delay. This delay models hardware where a minimal amount of time is needed for a change to occur. Delta delay allows for ordering of events that occur at the same simulation time during a simulation. Each unit of simulation time can be considered to be composed of an infinite number of delta delays. Therefore, an event always occurs at a real simulation time plus an integral multiple of delta delays.

For example, events can occur at 15 ns, 15 ns+IA, 15 ns+2A, 15 ns+3A,
22 ns, 22 ns+A, 27 ns, 27 ns+A, and so on.

Consider the AOI architecture

```
architecture AOI of AOI is
begin
    process (A, B, C, D)
    variable TEMPI ,TEMP2: BIT;
    begin
        TEMP1 := A and B;              -- statement 1
        TEMP2:= C and D;               --statement 2
        TEMP1 := TEMP1 or TEMP2;       -- statement 3
            Z<= not TEMP1;             --statement 4
    end process;
    end AOI_SEQUENTIAL;
```

Let us assume that an event occurs on input signal D (i.e., there is a change of value on signal D) at simulation time T. Statement I is executed first and TEMPI is assigned a value immediately since it is a variable. Statement 2 is executed next and TEMP2 is assigned a value immediately. Statement 3 is executed next which uses the values of TEMPI and TEMP2 computed in statements I and 2, respectively, to determine the new value for TEMPI. And finally, statement 4 is executed that causes signal Z to get the value of its right-hand-side expression after a delta delay, that is, signal Z gets its value only at time T+A; this is shown in Fig.



Sequential statements:

The sequential statements exist inside the boundaries of a process statement as well as sub-programs. These are-

1. The variable assignment statement
2. the signal assignment statement
3. wait statement
4. if statement
5. Case statement
6. Null statement

7. loop statement
8. Exit statement
9. Next statement
10. Assertion statement
11. Report statement.

Wait Statement:

The wait statement is a statement that causes suspension of a process or a procedure.

WAIT statement exists in three forms as follows.

1. wait on signal_list;

Eg: **wait on** S1, S2;

The process will be suspended on the wait statement and will be resumed when one of the S1 or S2 signals changes its value.

2. wait until condition;

Eg: **wait until** Enable = '1';

The wait statement will resume the process when the Enable signal changes its value to '1'.

3. wait for time;

Eg: **wait for** 50 ns;

A process containing this statement will be suspended for 50 ns.

4. **WAIT FOR 0:**

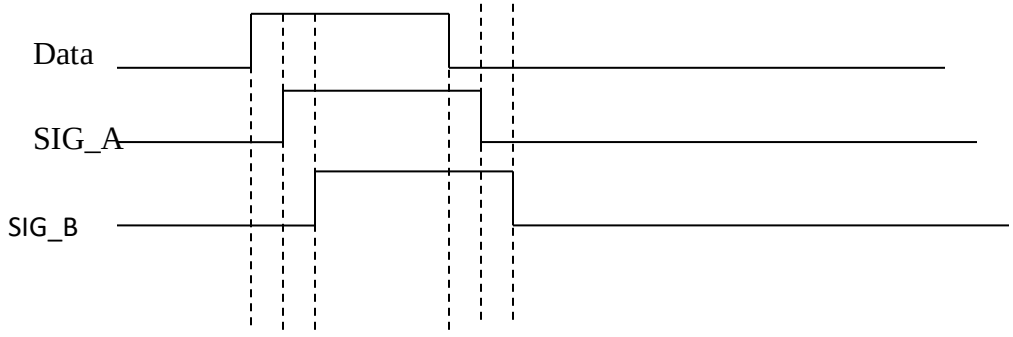
Syntax : Wait for 0ns

It means to wait for one delta cycle. This is a useful statement when the process is to be delayed.

For e.g.:

Wait 0: **process**
Begin
Wait on data
Sig_A <= data;
Wait for 0 ns;
Sig_B <= Sig_A;
End process;

The **Wait for 0ns** causes the process to suspend for 1Δ. SIG_A gets updated with its new value. Process resumes at 10 + 1Δ. SIG_B gets the new value of SIG_A at 10 + 2Δ. This is as shown below.



If Statement:

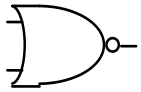
*The **if** statement is a statement that depending on the value of one or more corresponding conditions, selects for execution one or none of the enclosed sequences of statements,* IF statement exists in three forms.

1. **if** *boolean-expression then*
   sequential-statements
   end if;

Example1:

```
if SUM <= 100 then
SUM := SUM+10; end if;
```

Example2: Execution of nor gate:



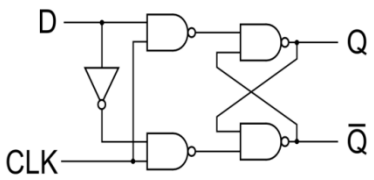
A	B	Y = A+B
0	0	1
0	1	0
1	0	0
1	1	0

```
entity NOR2 is
    port (A, B: in BIT; Z: out BIT);
end NOR2;
architecture behaviour of NOR2 is
    begin
    process (A, B)
        constant RISE_TIME: TIME := 10 ns;
        constant FALL_TIME: TIME := 5 ns:
        variable TEMP: BIT;
            begin
            TEMP := A nor B;
            If (TEMP = '1 ') then
              Z <= TEMP after RISE_TIME;
              else
              Z <= TEMP after FALL_TIME;
                  end if;
        end process;
        end Behaviour;
```

2. **if** *boolean-expression then*
   sequential-statements
   elsif
   boolean-expression **then**
   sequential-statements
   else
   sequential-statements
   end if;

Example: Execution of D flip-flop:

```
Entity  dff is
PORT (d, clk, rst : in-std-logic;
        Q, qbar: out-std-logic );
End dff;
Architecture behavior of dff is
        Begin
                Process(rst, clk)
                        Begin
                        If rst = '0' then
                        Q = '0';
                        elsif clk 'event and clk ='1' then
                        Q = 'd';
                        End if;
End Process;
End behavior;
```



3 **if** *boolean-expression then*
sequential-statements
else
sequential-statement
end if;

Example: Execution of 4 bit up counter

```
Entity counter is
    Port(E,clk,rst : in_std_logic;
    Count: inout std_logic_vector (3 down to 0);
 End counter;

Architecture behavior of counter is
Begin
        Process(rst, clk)
        Begin
        If rst = '0' then
          Count <= "0000";
        elsif clk 'event and clk ='1' then
          Count <= count +1;
         Else
           Count <= count;
        End if;
End Process;
End behavior;
```

The **if** statement is executed by checking each condition sequentially until the first true condition is found; then, the set of sequential statements associated with this condition is executed. The **if** statement can have

zero or more **elsif c**lauses and an optional else clause. An **if** statement is also a sequential statement, and therefore, the previous syntax allows for arbitrary nesting of if statements.

Case Statement:

The syntax is

case *expression* **is**
when *choices* => *sequential-statements*
when *choices* => *sequential-statements*
[**when others** => *sequential-statements*]
.
.
..
end case;

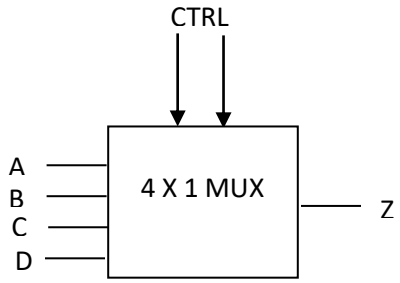
The CASE statement executes the proper statement depending on the value of the input instruction. If the value of the instruction is one of the choices listed in the WHEN clauses then the statement following the when clause is executed.

If the value of the expression is outside the range of the choices given, then the expression following the OTHERS clause is executed.

Examples:

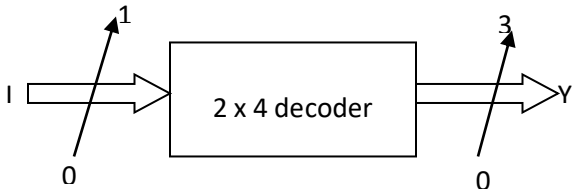
Write the VHDL code for MUX.

```
entity MUX is
        port (A, B, C, D: in BIT;
        CTRL: in BIT_VECTOR(0 to 1);
                Z: out BIT);
        end MUX;
        architecture BEHAVIOR of MUX is
        begin
                process (A, B, C, D, CTRL)
                begin
                case CTRL is
                        when "00" => Z:= A:
                        when "01" => Z := B;
                        when "10" => Z := C;
                        when "11" => Z := D;
                        end case;
                end process
                end BEHAVIOR;
```



Ctrl lines		Inputs			
S_0	S_1	D	C	B	A
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

Write the VHDL code for DECODER (2 x 4 decoder).

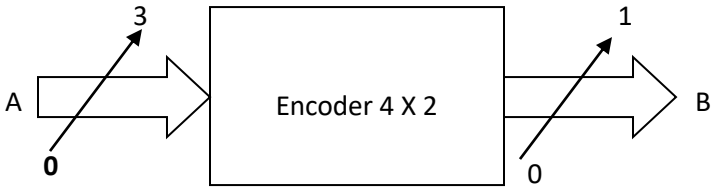



```
Entity decoder is
Port ( I : in std-logic-vector (1 downto 0);
       y : out  std-logic-vector (3 downto 0));
End decoder;
```

Ctrl lines		Inputs			
I_1	I_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

```
Architecture behavioral of decoder is
Begin
Process ( I)
Case I is
When "00" => y := "0001",
When "01" => y := "0010",
When "10" => y := "0100",
When "11" => y := "1000",
Ende case;
End behavioral ;
```

VHDL code for ENCODER (4 x 2 encoder).



INPUTS				OUTPUTS	
A(3)	A(2)	A(1)	A(0)	B(1)	B(0)
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

```
entity encoder is
Port (A: in STD_LOGIC_ VECTOR (3 Down to 0);
      B: out STD_LOGIC_VECTOR (1 Down to 0));
end encoder;

architecture Behavioral of encoder is
begin
        process (A)
                begin
                case A is
                When "0001" =>B <= "00";
                When "0010" =>B <= "01";
                When "0100" =>B <= "10";
                When "1000" =>B <= "11";
                 When others =>B <= "00";
                end case;
        end process;
        end Behavioral;
```

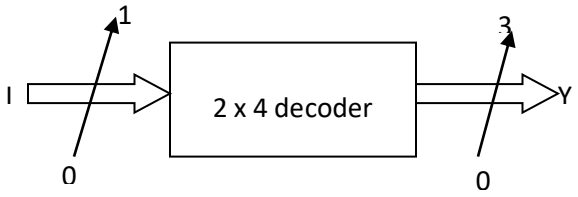
Null Statement

The statement **NULL** is a sequential statement that does not cause any action to take place and execution continues with the next statement.

It can be used to indicate that when some conditions are met, no action is to be performed. Such an application is useful in particular in conjunction with case statements to exclude some conditions.

Example:

Write the VHDL code for DECODER (2 x 4 decoder).




```
Entity  decoder is
Port  ( I : in std-logic-vector (1 downto 0);
        y : out  std-logic-vector (3 downto 0));
End decoder;

Architecture behavioral of decoder is
Begin
Process ( I)
        Case I is
        When "00" => y := "0001",
        When "01" => y := "0010",
        When "10" => y := "0100",
        When "11" => y := "1000",
        when others => null;
        End case;
End Process:
End behavioral ;
```

Ctrl lines		Inputs			
I_1	I_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

Loop Statement:

Loop is a sequential statement. The LOOP statement is used whenever an operation needs to be repeated. Loop statements are used for iteration is needed in a model.

The syntax of a loop statement is

[*loop-label* :] *iteration-scheme* **loop**

sequential-statements

end loop [*loop-label*] ;

There are three types of iteration schemes.

- **for** *iteration scheme.*

  FOR identifier In range LOOP

  Statements;

  END LOOP;

- **while** *iteration scheme.*

  WHILE condition LOOP

  Statements;

  END LOOP;

- The third form no iteration scheme is specified.

  LOOP

  Statements;

  END LOOP;

For Loop:

The **FOR-LOOP** statement is used whenever an operation needs to be repeated.
The **for** loop defines a loop parameter which takes on the type of the range specified.

for *identifier* **in** *range*

Example:

Write the VHDL code for factorial of a number using FOR LOOP:

```
 Entity  fact  is
PORT (clk: in-std-logic;
        Factorial : out-integer );
End fact;

Architecture behavioral of fact is
        Begin
                Process (clk )
                        Begin
                        FACTORIAL := 1;
                        If clk ‘event and clk =’1’ then
                        for NUMBER in 2 to N loop
                                FACTORIAL := FACTORIAL * NUMBER;
                        End loop;
                        End IF;
                        End Process
End behavioral;
```

The body of the for loop is executed (N-1) times, with the loop identifier, NUMBER, being incremented by I at the end of each iteration.

While Loop:

WHILE loop differs from FOR loop as it repeats the sequential statements until a particular condition is met with. The syntax is

while *boolean-expression*

Example:

```
J:=0;SUM:=10;
while J < 20 loop
        SUM := SUM * 2;
        J:=J+3;
end loop;
```

The statements within the body of the loop are executed sequentially and repeatedly as long as the loop condition, J < 20, is true. At this point, execution continues with the statement following the loop statement.

The third and final form of the iteration scheme is one where no iteration scheme is specified. In this form of **loop** statement, all statements in the loop body are repeatedly executed until some other action causes it to exit the loop. These actions can be caused by an **exit** statement, **a next** statement, or a return statement

Example:

```
SUM:=1;J:=0;
        L2: a label loop
        J:=J+21;
        SUM := SUM* 10;
        exit when SUM > 100;
end loop L2;
```

In this example, the exit statement causes the execution to jump out of loop L2 when SUM becomes greater than 100. If the exit statement were not present, the loop would execute indefinitely.

Exit Statement

The **EXIT** statement is a sequential statement that can be used only inside a loop. It is used to jump out of the loop conditionally or unconditionally and terminate the loop. The LOOP label in the EXIT statement identifies the particular loop to be exited

exit [*loop-label]* [**when** *condition]***:**

If no loop label is specified, the innermost loop is exited

Example:

```
SUM := 1; J := 0;
L3: loop
        J:=J+21;
        SUM := SUM* 10;
        if (SUM > 100) then
exit L3;
        end if;
end loop L3;
```

Next Statement

*The **next** statement is used to complete execution of one of the iterations of an enclosing loop statement. The completion is conditional if the statement includes a condition.*

Its syntax is

next [*loop-label]* [**when** *condition]*;

Example:

```
for J in 10 downto 5 loop
    if (SUM < TOTAL_SUM) then
    SUM := SUM +2;
    elsif (SUM = TOTAL_SUM) then
    next;
    else
    null;
    end if;
```

The difference between the Next statement and exit statement is that- the exit statement "exits" the loop entirely, while the next statement skips to the "next" loop iteration

Assertion Statement

Assertion statement checks whether a specified condition is true and reports an error if it is not.

The syntax is

assert *boolean-expression*

[**report** *string-expression*]

[**severity** *expression]*:

T**he** *assertion statement* has three optional fields and usually all three are used.

The condition specified in an ***assertion statement*** **must evaluate to a Boolean value** (true or false). If it is false, it is said that an assertion violation occurred.

The expression specified in the **report** clause **must be of predefined type STRING** and is a message to be reported when assertion violation occurred.

If the **severity** clause is present, it must **specify an expression of predefined type** SEVERITY_LEVEL, which determines the severity level of the assertion violation.

Example:

Functional errors, timing errors can be reported via **assert**:

```
entity RSFF is
    port( R,S, rst, CLK: in std_logic;
    Q,Qbar: out std_logic);
    End RSFF;
```

```
Architecture behavioral of RSFF is
        begin
            process (CLK , R,S)
        begin
            if (CLK‘ event and clock = ‘1’) then
            assert (S =’1’ and R =’1’);
            report “Undefined status ”
            severity Error;
            end if;
            end process;
        end behavioural;
```

Report statement:

A report statement can be used to display a message. It is similar to an assertion statement but without the assertion check. The syntax is

report *string expression*

[**severity** *expression*];

When report statement is executed, it causes the specified string to be printed and the severity level to be reported to the simulator for appropriate action.

Examples:

1. **if** CLR = ‘Z’ **then**
   report *“signal CLR has a high impedance value”;*
   end if;

2. **if** CLK /= ‘0’ **and** CLK /= ‘1’ **then**
   report *“CLK is neither ‘0’ nor ‘1’ ”;*
   severity *ERROR;*
   end if;

More on Signal Assignment Statement:

Delay is the time gap between the giving the value at the input and the time at which the change due to input is reflected at the output. By default, the propagation delay of the circuit is present but VHDL gives the user the flexibility to specify the delay to manage the correct updation of values in case of concurrent statements which are all executed in parallel. There are two types of delay models in VHDL.

- Inertial and
- transport

Inertial Delay Model:

Inertial delay models the delays found in switching circuits. It represents the minimum length of time for which an input value must be stable change at the output. In addition, the value appears at the output after the specified delay. If the input is not stable for the specified time, no output change occurs.

The syntax is

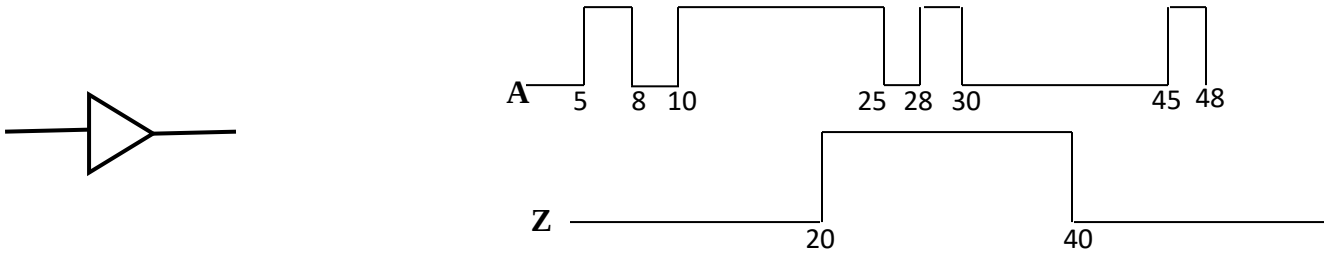
Signal –object<= [[**reject** pulse-rejection-limit] **inertial**] expression **after** inertial-delay-value;

If no pulse rejection limit is specified, the default pulse rejection limit is the inertial delay value itself.

Example:

Consider a non-inverting buffer with an inertial delay of 10 ns.

Ie., Z = **reject** 4 ns **inertial** A **after** 10 ns



Events on signal A that occur at 5 ns and 8 ns are not stable for the inertial delay duration and hence do not propagate to the output. Event on A at 10ns remains stable for more than the inertial delay, and therefore, the value is propagated to the target signal Z after the inertial delay; Z gets the value 1' at 20 ns. Events on signal A at 25ns and 28 ns do not affect the output since they are not stable for the inertial delay duration. Transition 1' to '0' at time 30 ns on signal A remains stable for at least the inertial delay duration, and therefore, a '0' is propagated to signal Z with a delay of 10 ns; Z gets the new value at 40 ns. Other events on A do not affect the target signal Z.

Since inertial delay is most commonly found in digital circuits, it is the default delay model. This delay model is often used to filter out unwanted spikes and transients on signals.

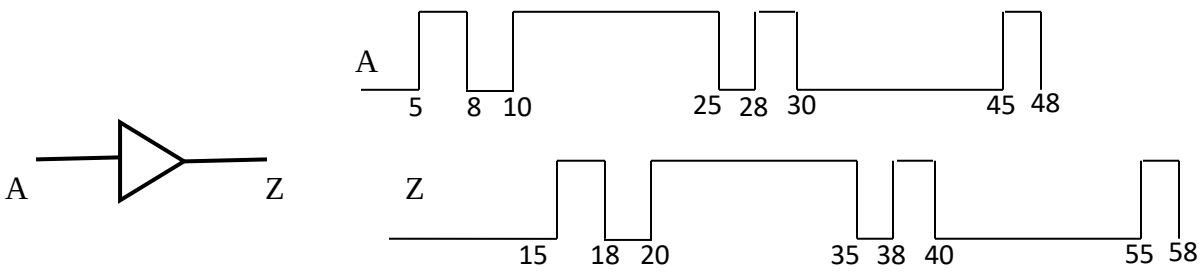
Transport Delay Model

In Transport delay model the change in the input are transported to the output. The only delay that comes into play is the propagation delay and there is pulse rejection limit. The syntax is

Transport expression **after** inertial-delay-value;

Example:

Consider a non-inverting buffer with a transport delay of 10 ns.



In this case, spikes would be propagated through instead of being ignored as in the inertial delay case.

Differences between Inertial delay and Transport delay:

Inertial delay	Transport delay
1. This models propagation delay due to components as well as pulse rejection width.	This models delay due to wires or interconnections.
2. Any pulse whose duration is smaller than the propagation delay is not allowed to reach the output.	This delay allows every change to reach the output.
3. This most commonly used	Not very commonly used. .

Creating Signal Waveforms:

It is possible to assign multiple values to a signal, each with a different delay value.

For example,

PHASE1 <= '0', '1' **after** 8 ns, '0' **after** 13 ns, '1' **after** 50 ns;

When this signal assignment statement is executed, say at time T, it causes four values to be scheduled for signal PHASE l , the value '0' is scheduled to be assigned at time T+A, 1' at T+8 ns, '0' at T+13 ns, and 1' at T+50 ns. Thus, a waveform appears on the signal PHASE l as shown in Fig.

The syntax of signal assignment statement is

Signal –object<= [**transport**| **[reject** pulse-rejection-limit] **inertial**] waveform-element;

Each waveform element has a value part, specified by an expression (called the waveform expression in this text), and a delay part, specified by an after clause that specifies the delay. The delays in the waveform elements must appear in increasing order. A waveform element is of the form

expression **after** *time-expression*

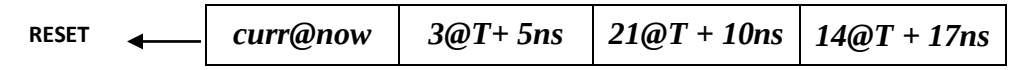
Signal Drivers:

The effective value of a signal, if there is more than one assignment to the same signal within a process, can be obtained by creating the drivers.

Every signal assignment in a process creates a *driver* for that signal. The driver of a signal holds its current value and all its future values as a sequence of one or more transactions, where each transaction identifies the value to appear on the signal along with the time at which the value is to appear.

Example:

```
process
begin
. . .
        RESET <= 3 after 5 ns, 21 after 10 ns, 14 after 17 ns;
end process;
```



All transactions on the driver are ordered in increasing order of time

In the above example, when the signal assignment statement is executed, say at time T, three new transactions are added to the driver for the RESET signal. The first transaction is the current value of the signal.

When simulation time advances to T+5 ns, the first transaction is deleted from the driver and RESET gets the value of 3. When time advances to T+I0 ns, the second transaction is deleted and RESET gets the value of 21. When time advances to T+17 ns, the third transaction is deleted and RESET gets the value of 14.

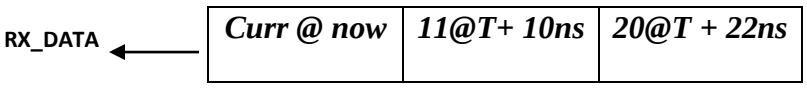
Effect of Transport Delay on Signal Drivers

Consider an example of a process having three signal assignments to the same signal RX_DATA.

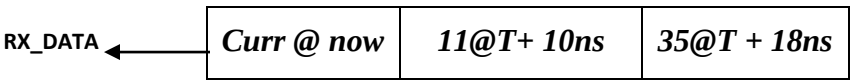
```
signal RX_DATA: NATURAL;
. . .
process
begin
        RX_DATA <= transport 11 after 10 ns:
        RX_DATA <= transport 20 after 22 ns;
        RX_DATA <= transport 35 after 18 ns;
end process;
```

Assume that the statements are executed at time *T*. The transactions on the driver for RX_DATA are created as follows.

When the first signal assignment is executed, the transaction, 11@T+10 ns, is added to the driver. After the second signal assignment is executed, the transaction, 20@T+ 22 ns, is appended to the driver since the delay of this transaction (= 22 ns) is larger than the delay of the pending transactions on the driver. The driver for RX_DATA looks like this



When the third signal assignment statement is executed, the new transaction, 35@T+18 ns, causes the 20@T+22 ns transaction to be deleted and the new transaction is appended to the driver. Because the delay for the new transaction (=18 ns) is less than the delay of the last transaction sitting on the driver (=22 ns). This effect is caused because transport delay is used. In general, a new transaction will delete all transactions sitting on a driver that are to occur at or later than the delay of the new transaction. Therefore, the driver for RX_DATA is changed to



Effect of Inertial Delay on Signal Drivers

When inertial delays are used, both the signal value being assigned and the delay value affect the deletion and addition of transactions. If the delay of the new transaction is earlier than an existing transaction, the latter is deleted and the new one is added at the end of the driver, regardless of the signal values of the two transactions

On the other hand, if the delay of the new transaction is greater than an already existing one, the signal values of the two transactions are compared. If they are the same, the new transaction is simply added at the end of the driver, if not, the existing one is deleted before adding the new transaction. Deletion occurs for every existing transaction with a signal value that is different from the new transaction.

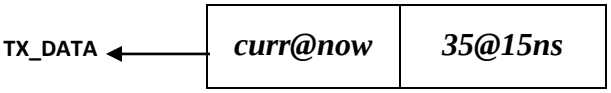
Example:

Consider the following process statement.

```
process
begin
        TX_DATA <= 11 after 10 ns;
        TX_DATA <= 22 after 20 ns;
        TX_DATA <= 33 after 15 ns;
        wait; -- Suspends indefinitely.
end process;
```

The transaction, 11@10 ns, first gets added to the driver. The second transaction, 22@20 ns, causes the 11@10 ns transaction on the driver to be deleted because the signal value, that is, 22, of the new transaction is different from the value of the transaction on the driver, that is, 11. The state of the driver at this point is

The execution of the third signal assignment causes the transaction 22@20 ns to be deleted from the driver, since the delay of the new transaction (=15 ns) is less than the delay of the transaction on the driver (similar to the transport delay case). The final status of the driver is



References :

1. Palnitkar, Samir, *Verilog HDL*. Pearson Education; Second edition (2003).
2. LizyKurien and Charles Roth. *Principles of Digital Systems Design and VHDL*.Cengage Publishing.
3. Fundamentals of HDL –A P Godse and D A Godse , Technical Publications
4. VHDL – Basics to Programming , Gaganpreet Kaur, PEARSON
5. https://technobyte.org/vhdl-behavioral-modeling-style-architecture/
6. https://getmyuni.azureedge.net/assets/main/study-material/notes/computer-science_engineering_digital-logic-design_vhdl_notes.pdf