Course Name: Digital Hardware Design
Course Code: 17B1NEC741

VHDL-3

Dr. Arti Noor
Dean, Academic Affairs
Electronics and Communication Engineering,
Jaypee Institute of Information Technology, Noida

Signal and variable assignments

- Variables are used for sequential execution whereas signals are used for concurrent execution.

The main comparison between them is as:

A. Variable assignment symbol:= and signal assignment <=

B. Variable can only be declared and used inside process command, whereas signal signals can be used inside or outside processes.

C. Variables that are assigned immediately take the value of the assignment.

D. Signals depend on it’s combinational or sequential code to know when the signal takes the value of the assignment. It may be after the delta delay in the default case.

- The difference between the executions of signal and variable:

```
Process (a,b)
Begin
b<= a;
c<= b;
end process;
```

a takes value ‘1’ at T ns

Time: 0 ns, a=’0’ and b=’0’.
Time: T ns, a=’1’ i.e. a changed to 1 and b is kept same. This means the first statement will execute only as its event dependent and a is only changing.
Time: T + Δ, b will get 1, c will get old value of b i.e. 0.
Time: T+ 2.Δ, b will not change its value as a is not changing but since b has just changed, so the second statement will execute and c will get 1.
Now neither b nor a are changing so both the statement will not be executing any more and system will be having value on a as 1, b as 1 and c as 1 finally.

Initial value

- All signals have an initial value when simulation begins. This value can either be user defined or will be default value.
- The simulator will assign these initial values to signals during the simulation.
- User can define the initial value by the following statements

  Signal a: std_logic:=’1’;
- For type bit, the initial value is ‘0’and that for std_logic is U.

Process (a)

Variable b,c:std_logic;

Begin

b: = a;

c: = b;

end process;

The variable assignment takes place immediately. Let’s analyze the following event.

- Time: 0 ns, a=’0’ and b=’0’.
- Time: T ns, a=’1’ i.e. a changed to 1 and b is kept same.
- Time: T + Δ, b will get 1, since b gets its new value immediately so c will get this new value of b immediately i.e. c will become 1.

WHEN TO USE SIGNAL AND VARIABLE

- Determine the suitable conditions to use signal/variable?????
- Ask yourself “will I be using this assigned value in this same simulation tick”
- If yes, obviously you need to use variable, otherwise use signal.
- It is also for the reader to remember that usage of variable is local only to a process.
- If an assigned value is to be shared among different process, a signal is more appropriate.

```
Process(  )
Begin
  ……..
Var : = 5;
  …..
  …..
If( Var=5) then
  …..
  ….
End process;
```

Assignment used in the same simulation tick in the sequential statements

```
P1: Process(  )
Begin
  ……..
Var < = 5;
  …..
  end process p1;
P2: Process( )
Begin

If( Var=5) then
  …..
  ….
End process p2;
```

Process statement

- The process statement contains only sequential statements.
- it is declared after **architecture-begin**. The general format for process command is:

Architecture **name** of **entity name** is
Begin
Process (sensitivity list)
Variables declaration
Begin
Sequential statements

End process;

- Sensitivity list is the name of signal on which the execution of process depends.
- There are basically two types of process statements.

  (a) Combinational process (b) sequential process

Combinational process: whenever combinational system is to be designed, put the entire signal that are in the **in** mode, in the sensitivity list.

```
process(I0, I1,Sel)
begin
Case Sel is
 When ‘0’ => f<=I0;
 When others => f<=I1;
 End case;
End process;
```

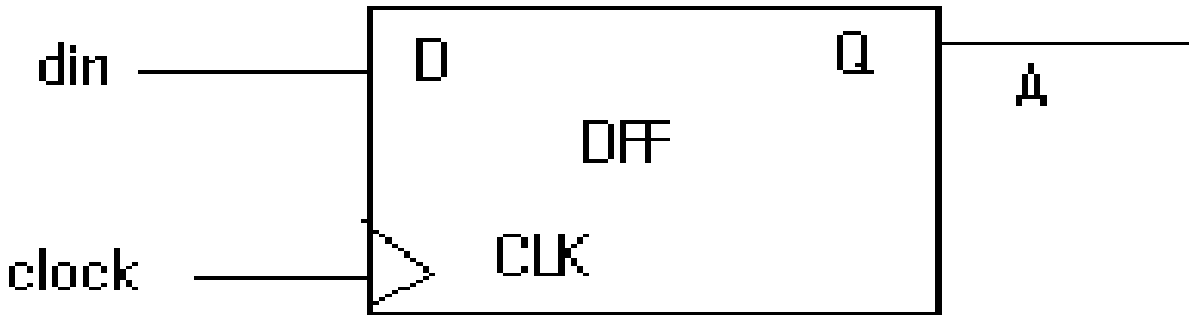
Sequential process:

- This is basically clocked process means process depends on the clock transitions.
- Clocked processes are synchronous and several such processes can be joined with the same clock. It means that no process can start unless clock transitions occur.

```
Process (clock)
    Begin
     A<= din;
     End process;
```

This process will start with clock changes from 0 to 1 or from 1 to 0.

- **At each transition din value will be assigned to output A.**
- **If clock is not changing then output will retain its previous value.**
- **This clocked process will be used during flip flop realization.**



Different clock descriptions

Method 1: process (clock)

Begin

If (clock’ event and clock=’0’) then

Q_{out}<=din;

End if;

End process;

- **Method 2:** process (clock)

  Begin

  If (clock=’1’) then Q_{out}<= din;

  End if;

- **Method 3:** process

  Begin

  Wait until clock=’1’;

  Q_{out}<= din;

  End process;

- **Method 4:** process (clock)

  Begin

  If (clock’ event and clock=’0’and clock ’last_value=’1’) then

  Q_{out}<=din;

  End if;

  End process;

Handling synchronous and asynchronous system modeling

- There are two different ways of resetting a flip flop. They are synchronous reset and asynchronous reset.
- **Asynchronous reset:** With asynchronous reset, the flip flop will be reset as soon as the reset is activated, irrespective of clock event.
- This means reset signal must be included in the sensitivity list along with the clock.

```
Process (clock, reset)
        Begin
        If (reset='1') then output<='0';
        Elsif ( clock' event and clock='0') then
        Output<= input;
        End if;
        End process;
```

- **Synchronous reset**: With synchronous reset the flip flop can only be reset at an active clock edge.

```
Alt2:   Process (clock)
        Begin
        If ( clock' event and clock='0') then
        If (reset='1') then output<='0';
        else
        Output<= input;
       End if;
       End if;
       End process;
```

- **Latches:** As we know latch is level triggered and output of the latch keep on responding according the changes in input.

```
Process (enable, input)
        Begin
        If (enable='1') then
        Output<= input;
        End if;
        End process;
```

Wait statements

There are four different ways of describing wait statements in a process:

1. Process (x, y)
2. Wait until a=’1’;
3. Wait on x, y ;
4. Wait for 10 ns;

The first and the third are identical if wait on x, y is placed at the end of the process.

The following two processes are therefore identical

```
P1: Process(x, y)
Begin
If (x>y) then
 Q<=’1’;
Else
Q<=’0’;
End if;
End process;
```

```
p2: process
    begin
     if (x>y) then
     q<=’1’;
    else
     q<=’0’;
     end if;
     wait on x, y;
    End process;
```

- **Wait until a=’1’;** means that, for the wait condition to be satisfied and execution of the code to continue, it is necessary for signal a to have an event, i.e. change value and the new value to be ‘1’, i.e. a rising edge for signal a.
- **Wait on x, y;** is satisfied when either signal x or y has an event (changes value)
- **Wait for 10 ns:** means that the simulator will wait for 10 ns before continuing execution of the process.

Synthesis tools do not support use of wait for 10 ns type of wait command

Some examples of **wait** statements are

```
wait on A, B, C;                          -- statement 1
wait until (A = B);                       -- statement 2
wait for 10ns;                            -- statement 3
wait on CLOCK for 20ns;                   -- statement 4
wait until (SUM > 100) for 50 ms;         -- statement 5

process                                   -- No sensitivity list.
variable TEMP1 ,TEMP2: BIT;
begin
TEMP1 :=A and B:
TEMP2 := C and D;
TEMP1 := TEMP1 or TEMP2;
Z<= not TEMP1;
wait on A, B, C, D;                  -- Replaces the sensitivity list.
end process;
```

- **Loop statements:** There are two kinds of loop statements in sequential VHDL.

  (a) For loop (b) While loop.

- **For Loop and while Loop:** both are used inside process commands as sequential commands. The format for using the same is given below.

Label: for <identifier> in <range> loop

```
F1:  For i in 0 to 3 loop
       If (a (i) ='1') then
            Q (i) <= b (i);
              Else
            Q (i) <= c(i);
             End if;
             End loop F1;
```

- Each for loop command must have end loop

- **While loop:**

Example:

```
While (q<12) loop
   Statements…..
   q<= q + 1;
End loop;
```

Exit Statement

- The exit statement is a sequential statement that can be used only inside a loop.

- It causes execution to jump out of the innermost loop or the loop whose label is specified. The syntax for an exit statement is

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

```
SUM := 1; J := 0;
L3: loop
J:=J+21;
SUM := SUM* 10;
if (SUM > 100) then
exit L3;        -- "exit;" also would have been sufficient.
end if;
end loop L3;
```

Next Statement

- The next statement is also a sequential statement that can be used only inside a loop.

- The syntax is the same as that for the exit statement except that the keyword next replaces the keyword exit. Its syntax is

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

```
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;
K:=K+1;
end loop;
```

Assertion Statement

- Assertion statements are useful in modeling constraints of an entity.

- For example, you may want to check if a signal value lies within a specified range, or check the setup and hold times for signals arriving at the inputs of an entity.

- If the check fails, an error is reported.
- The syntax of an assertion statement is

assert *boolean-expression*
[**report** *string-expression*]
[**severity** *expression]*:

Example:

assert (DATA <= 255)
report "Data out of range.';