

Advanced VLSI Design
Prof. D. K. Sharma
Department of Electrical Engineering
Indian Institute of Technology- Bombay

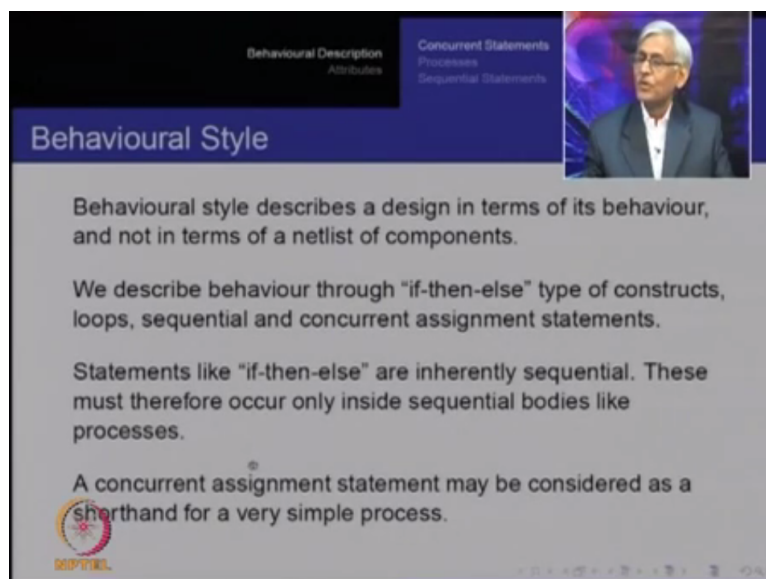
Lecture - 22
Behavioral Description in VHDL

In the previous lectures, we have looked at description in a structural style. The structural style instantiates components and then interconnects them. We also looked at repetition grammar. If you have multiple components of the same type, they can all be instantiated together using the repetition grammar. Today we look at the behavioral description using VHDL.

The behavioral description involves at what happens when kind of descriptions of hardware. That means we are not describing a circuit, we are describing the behavior of the circuit. Now we have had reason to look at this before and we know that such descriptions will require multiple line descriptions and that such things can be done using sequential statements. So therefore rather than concurrent descriptions, which are common to structural descriptions.

So we will look at the aspect of the language today, which is focused on describing a circuit behaviorally.

(Refer Slide Time: 01:45)



The slide is titled "Behavioural Style" and features a video inset of Prof. D. K. Sharma in the top right corner. The main content area has a dark blue header with the title and a sidebar with navigation links. The text on the slide describes the behavioral style of VHDL, emphasizing sequential constructs and the use of processes for sequential statements.

Behavioural Description
Attributes

Concurrent Statements
Processes
Sequential Statements

Behavioural Style

Behavioural style describes a design in terms of its behaviour, and not in terms of a netlist of components.

We describe behaviour through "if-then-else" type of constructs, loops, sequential and concurrent assignment statements.

Statements like "if-then-else" are inherently sequential. These must therefore occur only inside sequential bodies like processes.

A concurrent assignment statement may be considered as a shorthand for a very simple process.

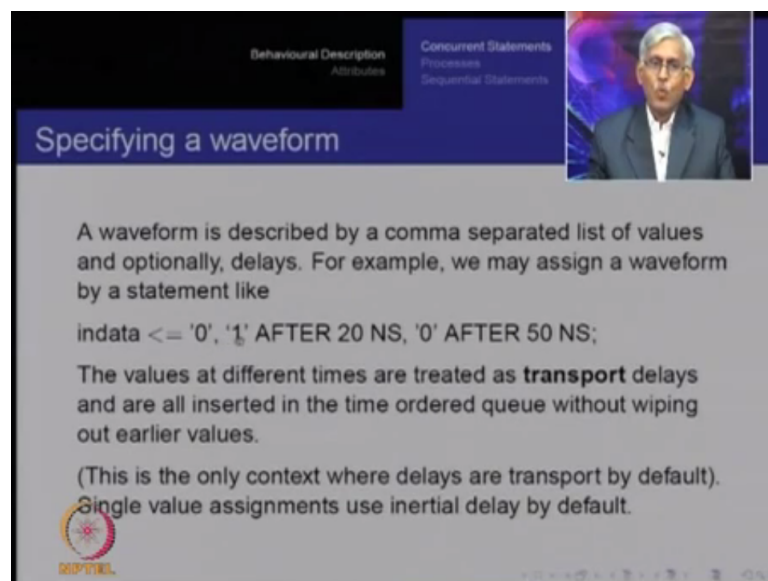
NPTEL

The behavioral style describes a design in terms of its behavior and not in terms of a netlist of components. So therefore the kind of control structures are very similar to programming

languages like if-then-else type of constructs, loops, sequential and concurrent assignment statements and so on. Since these are inherently sequential these must therefore occur only inside sequential bodies, we had discussed this earlier and in VHDL such bodies are called processes.

A concurrent assignment statement may be considered as a shorthand for a very simple one-line process. So essentially all concurrent statements can also be considered as processes. So processes are concurrent outside, sequential inside and if the statement is trivially describable in one line then that statement itself can be considered a concurrent statement.

(Refer Slide Time: 02:54)



The slide is titled "Specifying a waveform" and is part of a presentation on VHDL. It features a navigation menu on the right with "Concurrent Statements" selected. The main content area contains the following text:

A waveform is described by a comma separated list of values and optionally, delays. For example, we may assign a waveform by a statement like

```
indata <= '0', '1' AFTER 20 NS, '0' AFTER 50 NS;
```

The values at different times are treated as **transport** delays and are all inserted in the time ordered queue without wiping out earlier values.

(This is the only context where delays are transport by default).
Single value assignments use inertial delay by default.


The slide also includes a small video inset of a speaker in the top right corner and a logo in the bottom left corner.

There are various parts of describing the signals in this. First of all, we look at specifying a waveform. A waveform is described by a comma separated list of values and optionally delays. For example, we may assign a waveform by a statement like this. We say assign to indata, the value is 0 and then 1 after 20 nanoseconds and 0 after 50 nanoseconds. Now this is the only case where delays are treated by default as transport delays.

(Refer Slide Time: 03:45)

Behavioural Description
Attributes

Concurrent Statements
Processes
Sequential Statements




Concurrent Assignment

A concurrent assignment can be made conditionally by using 'when' clauses.

```

name <= [delay-mechanism]
    waveform when Boolean-expression else
    waveform when Boolean-expression;
  
```

The assignment is made from the first waveform where the Boolean expression evaluates to TRUE.




So if you have a narrow pulse here, it will not vanish. Now a concurrent assignment can be made conditionally by using when clauses. Notice that this is a same concurrent assignment. It is not inside a process statement. So the general context in which such statements are used is you have some signal to which you are making the assignment and the assignment is not just instantaneous we are assigning a whole waveform to this signal.

So we say that this is the signal name assigned to this name with this delay mechanism. This waveform when this Boolean expression is true else this waveform when this Boolean expression. So the assignment is made from the first waveform where the first Boolean expression evaluates to true.

(Refer Slide Time: 04:44)

Behavioural Description
Attributes

Concurrent Statements
Processes
Sequential Statements




Concurrent Assignment

The assignment can also be made on a selective basis, based on the value of some expression:

```

with expression select
name <= [delay-mechanism]
    waveform when choices,
    waveform when choices;
  
```

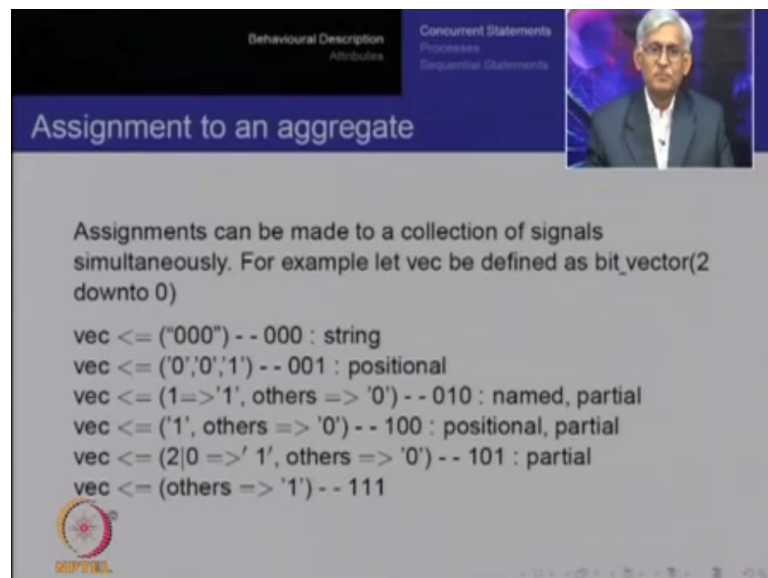
If the expression evaluates to one of the specified choices, the corresponding assignment is made.



The assignment can also be made on a selective basis based on the value of some expression like with expression select. This is somewhat like a case statement. Notice that these are all concurrent assignments and we are bringing in elements of programming into single line concurrent statements. So for these we can say with this expression select and then this is the signal name to which assignments are being made.

So you say assign to this signal with this delay mechanism, this waveform when the expression evaluates to these choices, this waveform when the expression evaluates to these choices and so on. So if the expression evaluates to one of the specified choices, the corresponding assignment is made.

(Refer Slide Time: 05:52)



The slide is titled "Assignment to an aggregate" and features a small inset image of a man in the top right corner. The main content lists several ways to assign values to a bit vector 'vec' defined as 'bit_vector(2 downto 0)'. The assignments are categorized by their format: string, positional, named, and partial.

```
vec <= ("000") -- 000 : string
vec <= ('0','0','1') -- 001 : positional
vec <= (1=>'1', others => '0') -- 010 : named, partial
vec <= ('1', others => '0') -- 100 : positional, partial
vec <= (2|0 => '1', others => '0') -- 101 : partial
vec <= (others => '1') -- 111
```

At the bottom left, there is a logo for "NPTL" (National Professional Training Library) and at the bottom right, there are navigation icons.

The assignments need not be made to a single signal. They can be made to a collection of signals simultaneously. For example, let vec be defined as a bit vector 2 down to 0 that means it is a 3-bit quantity. Look at the format of various kinds of assignments to this. Assign to vec within double coats 000 that means this 000 is to be treated as a string and therefore each element is to be treated as a character, which is like a 0 inside single coats.

And then each bit of vec will be assigned is 0. We can also give a comma separated list. For example, we can say assign to vec 0, 0, 1. In that case, bit vector 2 will get the value 0, bit vector 1 will get the value 0 and bit vector 0 will get the value 1. Remember bit vector is defined as 2 down to 0. So this is a positional assignment. We can also assign by name so for example, we can say assign to vector notice the right side association here, you say 2 index 1 give the value within single coats 1 and to others the value is 0 that means to index 1.

We will give the value 1 and all other elements of this vector will be given 0. This is a named partial assignment. Similarly, we can have a positional partial assignment namely we say assign to vec the value 1 and others 0. Notice the association of this 1 is now by position. So this is the first element of vector and because it is defined as a bit vector 2 down to 0 this is actually vec2.

So then all the others are assigned 0 that means 1 value, which is assigned specifically is because of its position and therefore it is called positional assignment. So you can give a whole number of values, which will be successively and positionally assigned to the bit vector and you may choose to say all others should be assigned 0. You can also make this kind of assignment by giving multiple indices.

For example, you say vector 2 vertical bar 0 associated with 1 and others with 0 that means vector 2 becomes 1, vector 0 becomes 1, and the 1 which is left out namely it could be multiple bits in this case that is only 1. So you say others go to 0. Notice that others need not require at least one assignment otherwise. You can use others directly for example you can say assign to vec others associate with 1.

And in that case all bits of this will be associated with 1. So in short various kinds of assignments are possible. You can assign a whole string to all the bits of the vector. You can assign bit by bit and these bits are assigned positionally. You can assign a few by name for example you are saying that to index 1 assign the value 1 to others assign 0. You can assign some bits positionally.

As in this case the first one like the first 0 here is assigned to vec2 and then instead of detailing each bit value we just together assign all the values 0 saying others to 0 that is positional and partial. Similarly, you can use more than 1 index at a time, which is to be assign the same value by saying vector 2 as well as vector 0 are to be assigned 1 whereas all other bits of this vector are to be assigned 0 and others can be used without even having a positional or named element coming before.

(Refer Slide Time: 10:33)

The slide is titled 'Processes' and is part of a presentation on Behavioral Description Attributes. It includes a video inset of a man in a suit. The main text explains that sequential constructs need to be placed inside a process and provides the following syntax:

```
[ process-label: ] process [(sensitivity-list)] [is]  
[declarations]  
begin  
    [sequential statements]  
end process [process-label];
```

Sequential statements include "if" constructs, case statements, looping constructs, assertions, wait statements etc.

The slide also features a small logo in the bottom left corner and navigation icons in the bottom right corner.

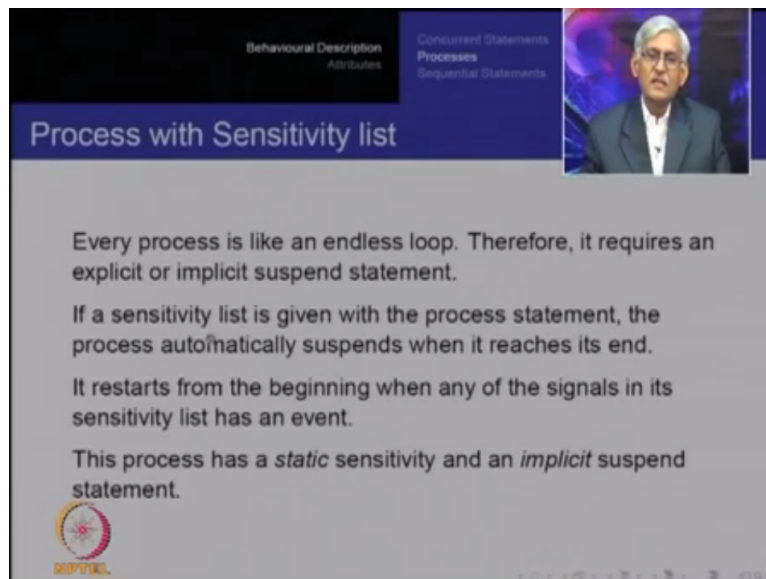
In that case all bits receive that value. This was as far as single statements with some logic thrown in were concerned. Those were concurrent statements; however, you may have complicated assignments, which may require a multi-line description. So such constructs are called sequential constructs. The entire sequence of lines has to be executed and they have to be placed inside what is called a process.

So a process uses the syntax. A process label followed by the keyword process followed by a sensitivity list. Then the keyword is and then begin and end process. Things inside square brackets are optional elements. The actual components of the process have two halves; you can declare some variable, some signals and so on. These are local to this process and then the actual logic is described using sequential statements.

And these sequential statements are enclosed between begin and end process. Now the sequential statements include if constructs case statements, looping constructs, assertions and wait statements etc. All of these are very similar to the equivalent constructs in programming languages. So in short, what you are saying is that when I enter a process I am going to describe the behavior of this part of the hardware as if it was a small piece of software.

Its behavior will be described in a programming language like fashion; however, there are additional things then the behavior described in a software fashion and that simply says that you have to include a sensitivity list, which says that when should this process be actually invoked. In the tutorial that we had done some time ago, we had seen how sensitivity is of various processes are struck.

(Refer Slide Time: 12:55)



Behavioural Description
Attributes

Concurrent Statements
Processes
Sequential Statements


Process with Sensitivity list

Every process is like an endless loop. Therefore, it requires an explicit or implicit suspend statement.

If a sensitivity list is given with the process statement, the process automatically suspends when it reaches its end.

It restarts from the beginning when any of the signals in its sensitivity list has an event.

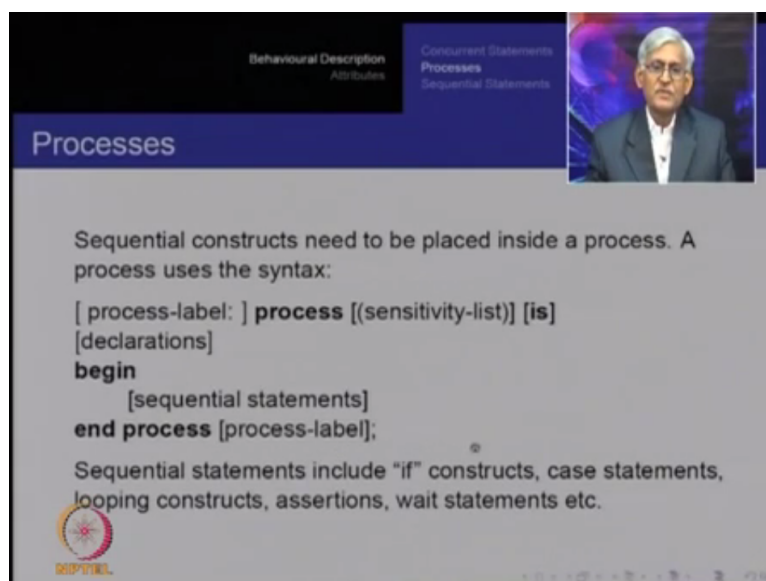
This process has a *static* sensitivity and an *implicit* suspend statement.



In this case, the sensitivity can modulate the way that this particular device is invoked. Now every process is like an endless loop. Therefore, it requires an explicit or implicit suspend statement. We have discussed this earlier. Now if a sensitivity list is given by the way the sensitivity list is optional. You can have dynamic sensitivity, but if a sensitivity list is given with the process statement, the process automatically suspends when it reaches its ends and loops back to the beginning.

It restarts from the beginning when any of the signals in its sensitivity list has an event. This is the case of a static sensitivity list and then implicit suspend statement.

(Refer Slide Time: 13:41)



Behavioural Description
Attributes


Concurrent Statements
Processes
Sequential Statements

Processes

Sequential constructs need to be placed inside a process. A process uses the syntax:

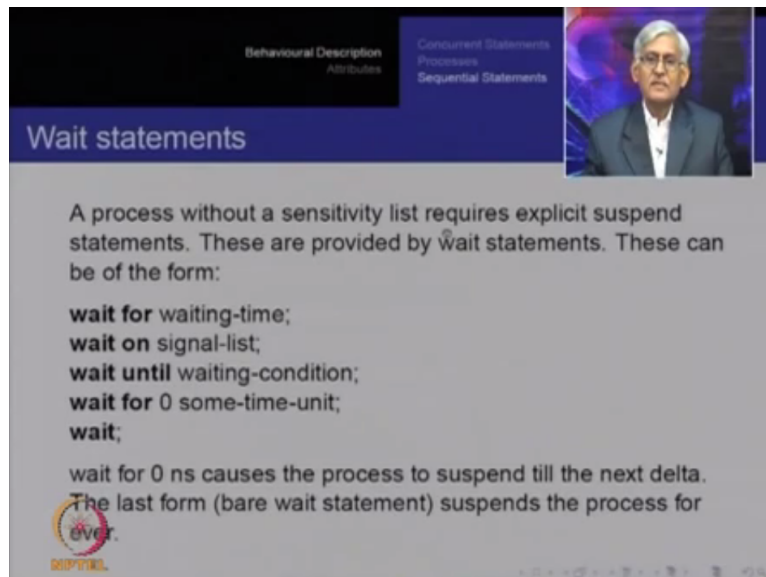
```
[ process-label: ] process [(sensitivity-list)] [is]  
[declarations]  
begin  
    [sequential statements]  
end process [process-label];
```

Sequential statements include "if" constructs, case statements, looping constructs, assertions, wait statements etc.



So essentially if you have a sensitivity list given here then these signals to which this process is sensitive is static it is specified once in for all and then the suspension is implied when it reaches end process, the process loops back to the beginning to the sensitivity list and waits there, it is suspended and whenever any signal in this list has an event, the process starts all over again.

(Refer Slide Time: 14:12)



The slide is titled "Wait statements" and is part of a presentation on Behavioral Description Attributes. It lists several wait constructs: **wait for** waiting-time; **wait on** signal-list; **wait until** waiting-condition; **wait for 0** some-time-unit; and **wait**;. It explains that **wait for 0 ns** causes the process to suspend till the next delta, and the last form (bare wait statement) suspends the process for ever. A small video inset in the top right corner shows a man speaking.

Behavioural Description Attributes

Concurrent Statements
Processes
Sequential Statements

Wait statements

A process without a sensitivity list requires explicit suspend statements. These are provided by wait statements. These can be of the form:

- wait for** waiting-time;
- wait on** signal-list;
- wait until** waiting-condition;
- wait for 0** some-time-unit;
- wait**;

wait for 0 ns causes the process to suspend till the next delta.
The last form (bare wait statement) suspends the process for ever.

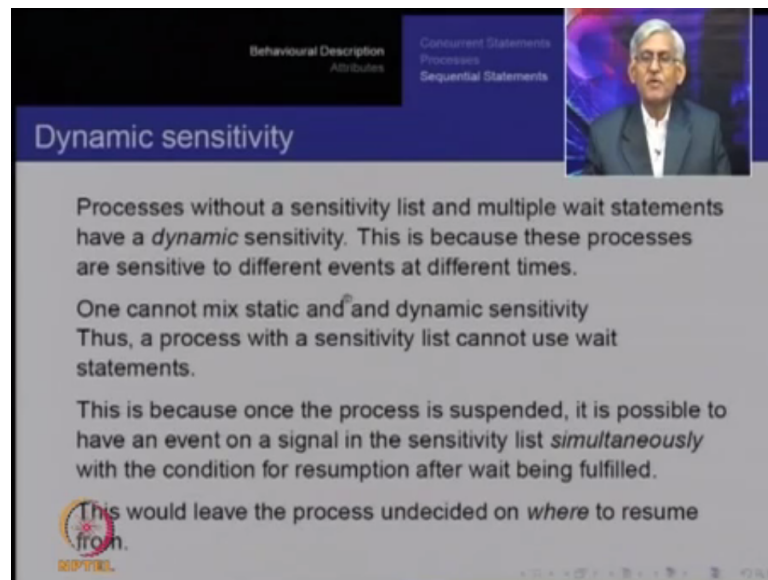
A process without a sensitivity list is also allowed. In this case, the sensitivity is dynamic and such processes require explicit suspend statements, otherwise they will run forever. These suspend statements are provided by special construct called wait statements. Various kinds of wait statements are described. For example, you can say wait for and give an amount of time for which the process will be suspended.

When this time ends the process will be re-invoked. Wait on a signal list that means you wait for an event so the process is suspended immediately at this wait and you wait for an event on any of the signals in this list. For wait until you specify a waiting condition and you wait until that condition becomes true. Similarly, you can say wait for 0 nano seconds or 0 anytime unit. This has a special meaning and we will see this in a while.

And finally a wait without any argument that means this process should be suspended forever. Wait for 0 nanoseconds causes the process to suspend till the next delta. So wait for 0 essentially says that restart this process only in the next delta. All these wait statements essentially cause the restarting of the process not at the beginning, but from the point where the process were suspended.

So in the whole description a program like description there is a wait statement, they could all be conditional like this. The process actually suspends and when an event to restart it takes place then the process restarts from the statement immediately following the wait statement.

(Refer Slide Time: 16:23)



The slide is titled "Dynamic sensitivity" and features a small video inset of a man in the top right corner. The slide content is as follows:

Behavioural Description
Attributes

Concurrent Statements
Processes
Sequential Statements

Dynamic sensitivity

Processes without a sensitivity list and multiple wait statements have a *dynamic* sensitivity. This is because these processes are sensitive to different events at different times.

One cannot mix static and dynamic sensitivity
Thus, a process with a sensitivity list cannot use wait statements.

This is because once the process is suspended, it is possible to have an event on a signal in the sensitivity list *simultaneously* with the condition for resumption after wait being fulfilled.

This would leave the process undecided on *where* to resume from.

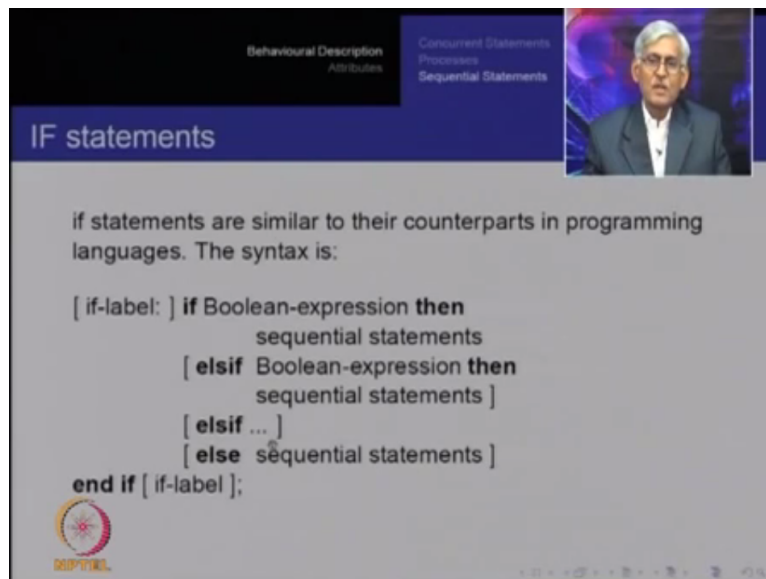
So these for example processes without a sensitivity list and multiple wait statements have a dynamic sensitivity. This is because these processes are sensitive to different events at different times. The starting of this process is waiting for the argument of that wait statement and it is possible therefore to make a process sensitive to different things at different times by inserting multiple wait statements, which specify the restarting of the process dependent on different dynamics conditions.

Notice that one cannot mix static and dynamic sensitivity lists. For example, if you give a sensitivity list right in the beginning then you are not allowed to use a wait statement. On the other hand, if you do not give a sensitivity list, you must use a wait statement. This is because once the process is suspended; it is possible to have an event on a signal in the sensitivity list simultaneously with the condition for resumption after the wait is fulfilled.

So if you allow dynamic and static sensitivity together you would not know where to start this program. If there is an event on a signal, which is in the sensitivity list and also the restarting after wait is fulfilled then you do not know whether to start it after the wait statement or right from the beginning. The sensitivity list requires the process to be started

right from the beginning and the wait statement requires it to be restarted immediately after the wait statement and if you permitted both then there will always be this problem.

(Refer Slide Time: 18:18)



The slide is titled "IF statements" and is part of a presentation on "Behavioural Description Attributes". It includes a navigation menu with "Concurrent Statements", "Processes", and "Sequential Statements". A small video inset shows a man speaking. The main text explains that if statements are similar to programming languages and provides the following syntax:

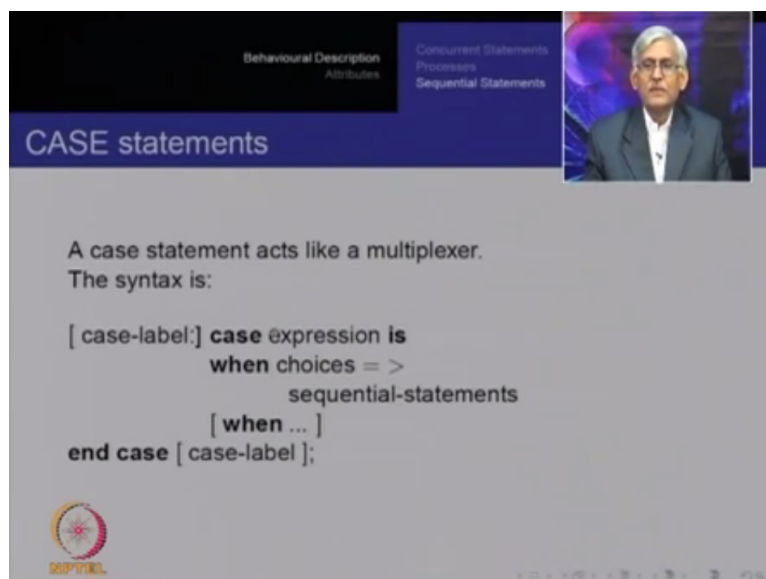
```
[ if-label: ] if Boolean-expression then
    sequential statements
[ elsif Boolean-expression then
    sequential statements ]
[ elsif ... ]
[ else sequential statements ]
end if [ if-label ];
```

The slide also features a logo in the bottom left corner and navigation controls in the bottom right.

So therefore we cannot have a dynamic and a static sensitivity list simultaneously. Now Let us look at many of the constructs, which are possible in processes. If statements are very similar to their counter parts in programing languages, their syntax is you may have an if label. If this Boolean expression, then this group of sequential statements else if this Boolean expression then this group of sequential statements.

A sequence of else if and finally an else, which will be executed if none of these Boolean expression is satisfied.

(Refer Slide Time: 18:55)



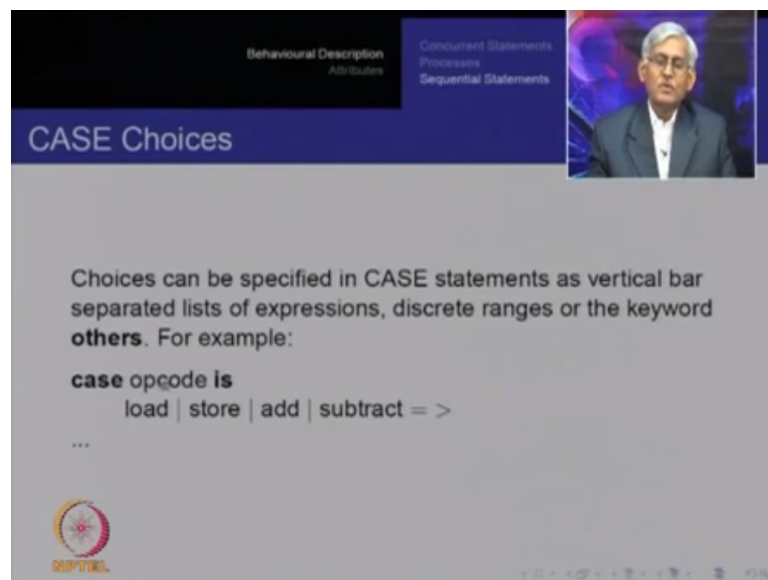
The slide is titled "CASE statements" and is part of a presentation on "Behavioural Description Attributes". It includes a navigation menu with "Concurrent Statements", "Processes", and "Sequential Statements". A small video inset shows a man speaking. The main text explains that a case statement acts like a multiplexer and provides the following syntax:

```
[ case-label:] case expression is
    when choices = >
        sequential-statements
    [ when ... ]
end case [ case-label ];
```

The slide also features a logo in the bottom left corner and navigation controls in the bottom right.

Similarly, a case statement acts like a multiplexer in hardware. The syntax is you can have the specific case label, which is optional and then you say case followed by an expression is and then lots of when clauses. When this expression is evaluated it can evaluate to a discrete set of values. You specify certain choices and when those choices are the values to which this expression evaluates then the corresponding sequential statements are executed.

(Refer Slide Time: 19:32)




Choices can be specified in a case statement as vertical bar separated lists of expressions, discrete ranges or the keyword others. For example, let us say that we have an enumerated variable called opcode. Opcode is a signal and we say case opcode is load store add or subtract. In all these cases, the following sequential statements will be executed. So for example if the opcode is either load or store or add or subtract, then the following sequences will all be executed, but not otherwise.

(Refer Slide Time: 20:18)

Behavioural Description
Attributes

Concurrent Statements
Processes
Sequential Statements




Loop Statements

There are several different forms of the loop statement. The simplest is the endless loop:

```
[ loop-label: ] loop
[ loop-label: ] loop
    sequential statements
end loop [ loop-label ];
```

⊛

This constitutes an endless loop.
It is assumed that it will have an exit statement or a wait statement inside to suspend operation.




So in short, the case need not be one single option, you can give multiple options for the value of the expression. Similarly, you have loop statements, the simplest of these is loop, which is essentially a loop label, loop followed by sequential statements and then end loop. Notice that this constitutes an endless loop and therefore there must be an exit statement in between otherwise this loop will never exit.

(Refer Slide Time: 20:53)

Behavioural Description
Attributes

Concurrent Statements
Processes
Sequential Statements




Exiting a Loop

The exit statement has the syntax:

```
[ label: ] exit [ loop-label ] [ when Boolean expression ]
```

The loop label allows one to exit several levels of nested loops.

We can also skip to the end of a loop by using the **next** statement. This works like "continue" in C.

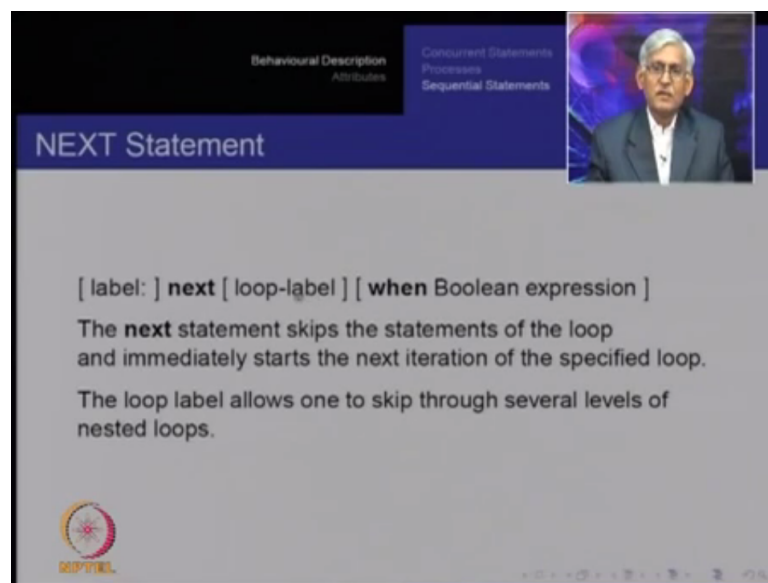


So the exit statement has the syntax. The exit itself can have its own label. Exit followed by the loop label, which loops are being exited. Remember there can be loops inside loops. So you can specify, which loops is being exited when a Boolean expression. So essentially the loop goes on till this Boolean expression becomes true and when this Boolean expression is found to be true, the loop exits.

The loop label allows one to exit several levels of nested loops. So suppose there is a loop within a loop within a loop and then exit becomes true you exit to which level because this inner statement is a part of each of the outer loop and that is why this loop label can be optionally specified. If it is not specified, then only the innermost loop is exited. You can also skip to the end of the loop.

You can skip the statements up to the end of the loop by using the next statement. This works like continue in c that means the loop is not executed, but during this iteration the rest of the instructions in this loop are skipped and you reach right to the end of the loop.

(Refer Slide Time: 22:18)




So the next statement has this syntax next you might specify the loop label again loops might be nested and you want to skip up to the end of which loop can be specified by this loop label if not specified that means you mean the innermost loop and when followed by Boolean expression.

(Refer Slide Time: 22:39)

Behavioural Description
Attributes

Concurrent Statements
Processes
Sequential Statements




WHILE Loops

VHDL also has a **while** loop.

```
[ loop-label: ]
while Boolean-expression loop
    sequential statements
end loop [ loop-label ];
```

The loop continues as long as the Boolean expression is TRUE.




Apart from this simple loop, VHDL also has a while loop. So you say while this Boolean expression is true loop followed by sequential statements end loop. So while this expression remains true you keep looping between here and end loop. So this at every iteration this expression is evaluated and should this expression become true ever then the loop will break. This is very similar to while statements in most programming languages.

(Refer Slide Time: 23:18)

Behavioural Description
Attributes

Concurrent Statements
Processes
Sequential Statements




For Loops

VHDL also provides a **for** loop.

```
[ loop-label: ]
for identifier in discrete-range loop
    sequential statements
end loop [ loop-label ];
```

The discrete range can be of the form
expression **to** | **downto** expression

The identifier is initialized to the left limit of the range and takes on successive values in the discrete range till it exceeds the right limit.



You also have a for loop, for loop is a count kind of loop very much like for loops in many programming languages that you might be familiar with. For identifier in a discrete range loop, that means all the sequential statements till end loop are executed for the identifier acquiring each of the values given in this discrete range. So the discrete range can be of the form 2 or down 2.

The identifier is initialized to the left limit of the range and takes on successive values in the discrete range till it exceeds the right limit.

(Refer Slide Time: 24:09)

The slide is titled "Assertions and Reports" and is part of a presentation on "Behavioural Description Attributes". It includes a video inset of a man speaking. The main text on the slide explains the syntax and behavior of the assert statement.

Behavioural Description Attributes

Concurrent Statements
Processes
Sequential Statements

Assertions and Reports

The assert statement takes the form

```
[ label: ] assert Boolean expression  
    [ report expression ] [ severity expression ];
```

If the Boolean expression is TRUE, no action is taken.
If it is FALSE, an assertion violation is said to have occurred.
The simulators then outputs the **report** expression.
Subsequent operation depends on the severity clause.

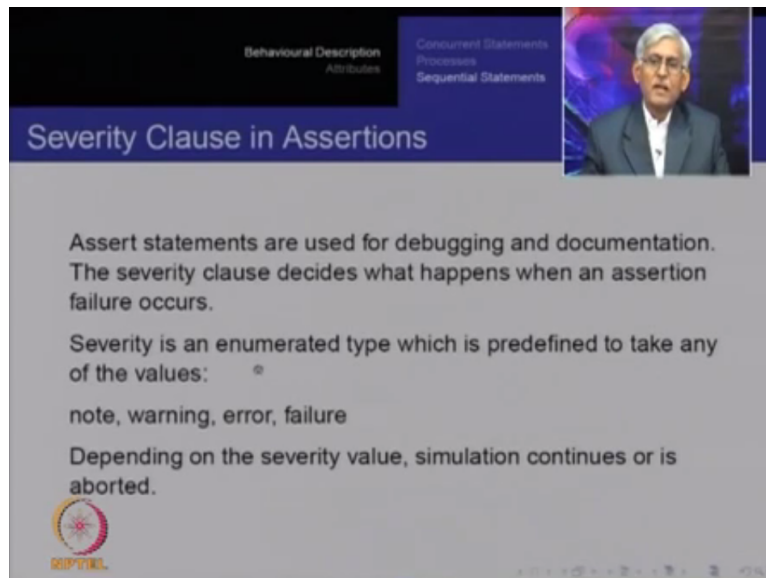
ISPTVIL

This takes care of most of the programming type of instructions. We also have some insertions, which are largely used for debugging and for making sure that up to now the logic has been correct and the major statement in this case and which forms the core of most test benches is the assert statement. The assert statement takes the form an optional label followed by the keyword assert followed by Boolean expression.

The idea is that you normally expect this Boolean expression to be true. Therefore, you are asserting that this condition is true; however, should this expression ever be found to be false then you will print a message given by the report clause and then take an action depending on the severity of this violation of assert.

So normally you expect something to be true, but should that expression not be true how severe is this unexpected thing is specified by severity and the message to be printed when this expression is found to be false is given by this clause. So if the Boolean expression is true, no action is taken, but if it is false and assertion violation is said to have occurred. The simulator then outputs the report expression and this subsequent operation depends on the severity clause.

(Refer Slide Time: 25:53)



Behavioural Description
Attributes

Concurrent Statements
Processes
Sequential Statements


Severity Clause in Assertions

Assert statements are used for debugging and documentation. The severity clause decides what happens when an assertion failure occurs.

Severity is an enumerated type which is predefined to take any of the values: *

note, warning, error, failure

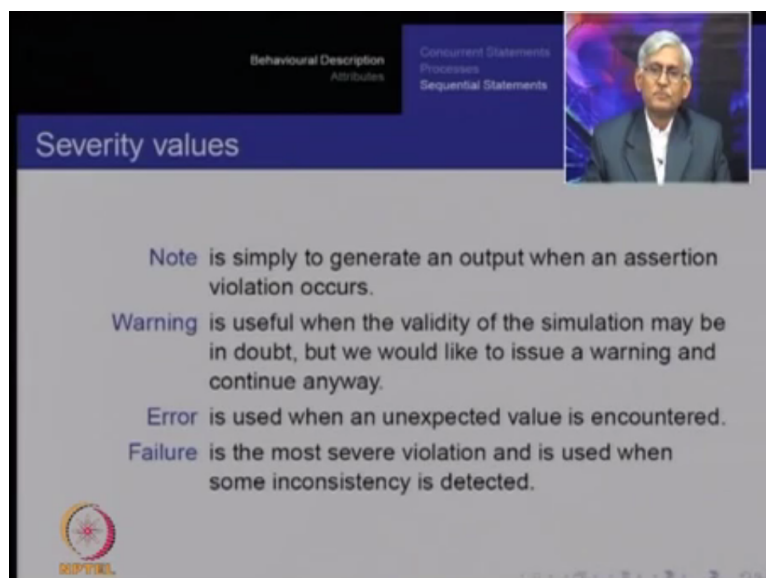
Depending on the severity value, simulation continues or is aborted.



Assert statements are used for debugging and documentation. The severity clause decides what happens when an assertion failure occurs. Severity is an enumerated type, which is predefined to take any of the values. You might recall that we had discussed this when we had looked at the building elements of VHDL and the severity can take one of four enumerated values.

These are note, warning, error and failure. Now depending on the severity value, simulation continues or is aborted.

(Refer Slide Time: 26:35)



Behavioural Description
Attributes

Concurrent Statements
Processes
Sequential Statements


Severity values

Note is simply to generate an output when an assertion violation occurs.

Warning is useful when the validity of the simulation may be in doubt, but we would like to issue a warning and continue anyway.

Error is used when an unexpected value is encountered.

Failure is the most severe violation and is used when some inconsistency is detected.

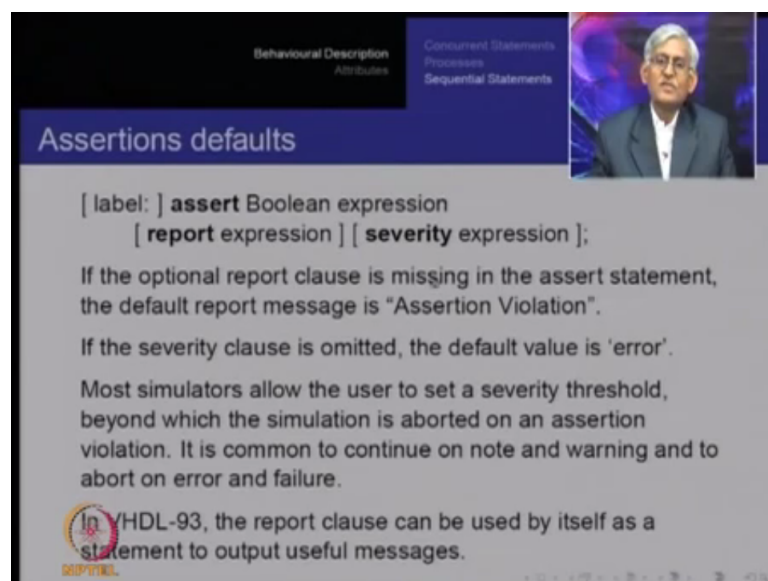


So these are the four possible severity values. Note is simply to generate an output when an assertion violation occurs maybe this assertion violation is not serious at all. You normally expect something to be true and you simply want to be notified when it is not true and then

go on with the simulation. In such a case, use a severity of note. Warning is useful when the validity of the simulation maybe in doubt, but we would like to issue a warning and continue anyway.

Error is used when an unexpected value is encountered and normally you would like to stop the simulation at this point. There is no point going ahead if this assertion has failed and failure is the most severe violation and is used when some inconsistency is detected. For example, you might be taking square root of a negative number.

(Refer Slide Time: 27:30)



The slide is titled "Assertions defaults" and features a small inset image of a man in the top right corner. The slide content is as follows:

Behavioural Description
Attributes

Concurrent Statements
Processes
Sequential Statements

Assertions defaults

```
[ label: ] assert Boolean expression  
[ report expression ] [ severity expression ];
```

If the optional report clause is missing in the assert statement, the default report message is "Assertion Violation".

If the severity clause is omitted, the default value is 'error'.

Most simulators allow the user to set a severity threshold, beyond which the simulation is aborted on an assertion violation. It is common to continue on note and warning and to abort on error and failure.

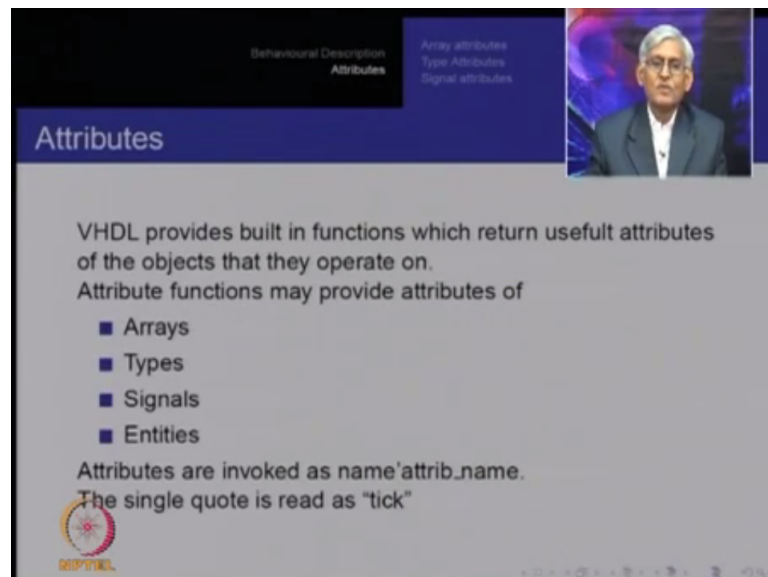
In VHDL-93, the report clause can be used by itself as a statement to output useful messages.

So the assertion defaults are if the optional report clause is missing then there is a default report clause and it simply prints the phrase assertion violation. So you need not give the report clause. In that case, it will be assumed that you are happy with printing assertion violation whenever the Boolean expression is false. If the severity clause is omitted, the default value is error that means whatever actions are associated with an error those will be taken.

Most simulators allow the user to set a severity threshold beyond, which the simulation is aborted on an assertion violation. It is common to continue on note and warning and to abort on error and failure. In VHDL 93 onwards the report clause can be used by itself without an assert as a statement to output useful messages. In the older versions, one on purpose caused an assertion failure with the severity of note in order to printout something.

Since it was commonly done the newer versions of VHDL 93 allow you to use the report clause directly without going through this artificial device of causing an assertion failure and giving it as a note so that the simulation will anyway continue then only use this to report an output. So from VHDL 93 onwards you can use reports.

(Refer Slide Time: 29:23)




Now there are various built in functions defined in VHDL. These functions are called attributes and they have the syntax signal name tick followed by attribute. There are many, many attributes and we will actually not go through an exhaustive list of these, but we want that very similar named attributes are used for different kinds of objects and the attribute functions provide attributes in this list.

The attributes of arrays for example the size, minimum value maximum value, of signal types, of specific signals themselves and of entities. Remember an object may qualify to be classified as more than one of these and the higher of these then prevails. The single code is often read as a tick.

(Refer Slide Time: 30:31)

Behavioural Description
Attributes

Array attributes
Type Attributes
Signal attributes



Array Attributes

Array attributes interrogate the property of arrays. Consider the declaration:
 TYPE regfile IS ARRAY(0 To 3, 7 Downto 0) OF BIT;

Then we can use the following attributes:

'LEFT : regfile'LEFT(2) = 7 'RIGHT: regfile'RIGHT(1) = 3 'HIGH: regfile'HIGH(2) = 7 'LOW: regfile'LOW(1) = 0	'RANGE: regfile'RANGE(1)= 0 TO 3 'REVERSE_RANGE: regfile'REVERSE_RANGE(1) = 3 DOWNT0 0 'LENGTH: regfile'LENGTH(1) = 4 'ASCENDING: regfile'ASCENDING(1) = TRUE
---	--

So for example here are the examples of various array attributes. Array attributes interrogate the property of arrays. Notice they do not interrogate the properties of each array element. So these report the property of arrays. Each attribute has a written value. So consider the declaration type, regfile is array 0 to 3, 7 down to 0 of bit. So that means there are four 8 bit registers.

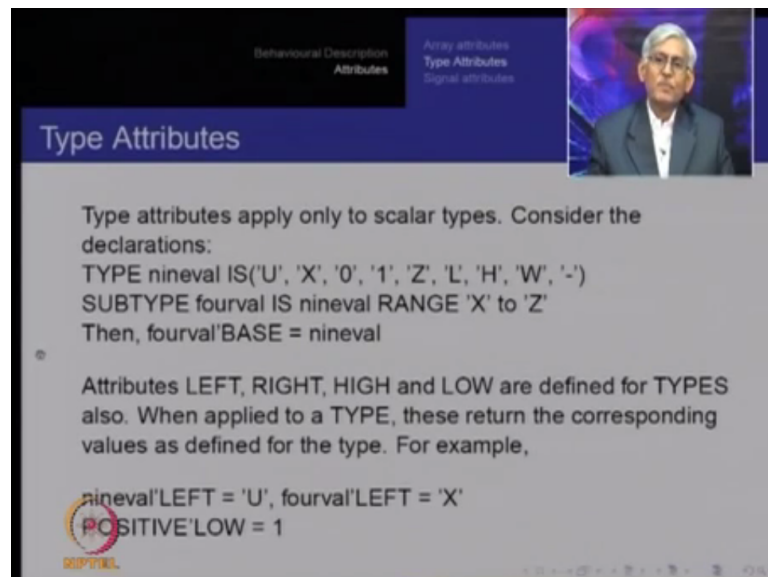
Each register is 7 down to 0 and the register index goes from 0 to 3. Now you can use the following attributes. Tick left for example if you said regfile tick left. Notice regfile is actually a signal being a bit, but because it is an array the array attributes has precedence. So tick left will return regfile tick left 2 will return the left value of the second attribute of the array.

Notice these are the attributes of the array. The first attribute is 0 to 3, the second attribute is 7 down to 0 and tick left 2 will return the left value of this second attribute and therefore 7. You have tick right for example regfile tick right 1 that means you are now talking about attribute number 1 and you want the right value of this attribute not the left value. So therefore the right value is 3.

So this function regfile tick right 1 will return 3. You also have attributes called high and low, which are not positional. For example, regfile tick high 2 will return the higher of these attributes irrespective of the order in which they are placed and in this case that will return 7. Regfile low of 1 will return 0. Similarly, you may request the range specified. You may expect a reverse range that means a range will be picked up and reversed.

For example, 2 will become down 2 and down 2 will become 2 and length will return the size of that attribute. So remember these are arrays and array attributes normally refer to the domain of the indices of this array.

(Refer Slide Time: 33:19)



The slide is titled "Type Attributes" and is part of a presentation. It contains the following text:

Behavioural Description
Attributes

Array attributes
Type Attributes
Signal attributes

Type Attributes

Type attributes apply only to scalar types. Consider the declarations:
TYPE nineval IS('U', 'X', '0', '1', 'Z', 'L', 'H', 'W', '-')
SUBTYPE fourval IS nineval RANGE 'X' to 'Z'
Then, fourval'BASE = nineval

Attributes LEFT, RIGHT, HIGH and LOW are defined for TYPES also. When applied to a TYPE, these return the corresponding values as defined for the type. For example,

nineval'LEFT = 'U', fourval'LEFT = 'X'
POSITIVE'LOW = 1

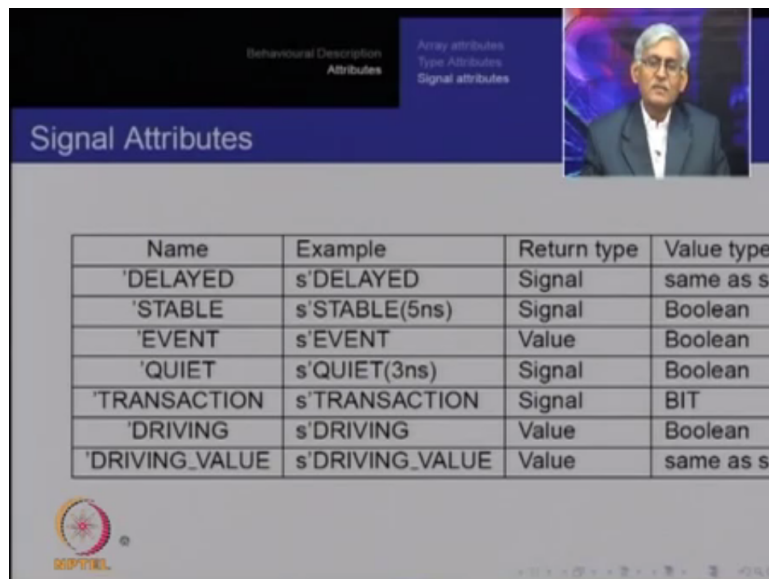
The type attributes apply only to scalar types. Notice if it is not a scalar then the array attribute will take over. So type attributes apply only to scalar types. Consider the declaration type nineval is, this is an enumerated type. I am enumerating the values that this type can take and these values are U, X, 0, 1, Z, L, H, W and -. You may recall that these are the values that std logic signals take.

I may declare a subtype of this saying subtype fourval is nineval range X to Z that means this X, 0, 1, and Z high impedance that is a different type, which is the subtype of this. Then you have an attribute called tick base and fourval tick base will return nineval because fourval is a subtype and its base is the parent type, which is nineval. Attributes left, right, high and low are defined for types also.

That is why I have said that it can be confusing you have to be aware of the kind of object whose attributes are being demanded. The name of the attribute might be the same as in this case tick left. So nineval tick left will return U that is the left most value of this type not an array. Fourval tick left will return X. For example, there is a predefined type called positive, which takes only positive integers greater than 0.

So positive tick low will return 1 because that is the lowest value, which is defined for the type positive.

(Refer Slide Time: 35:16)



Name	Example	Return type	Value type
'DELAYED	s'DELAYED	Signal	same as s
'STABLE	s'STABLE(5ns)	Signal	Boolean
'EVENT	s'EVENT	Value	Boolean
'QUIET	s'QUIET(3ns)	Signal	Boolean
'TRANSACTION	s'TRANSACTION	Signal	BIT
'DRIVING	s'DRIVING	Value	Boolean
'DRIVING_VALUE	s'DRIVING_VALUE	Value	same as s

Similar to types there are signal attributes and there are quite a few of these by the way. I have picked out the more important ones here and you will do well to just browse through the attributes, which are available to you in VHDL. So tick delayed is same as the signal, but it is delayed by a certain amount. For example, tick stable is an assertion it returns Boolean this one returns a signal which is a delayed version of the original signal.


Tick stable is a Boolean value, which is true or false if the signal has not changed during the last 5 nanoseconds. So s tick stable 5 nanoseconds will return true if there has been no new value assign to signal s during the last 5 nanoseconds. Tick event means there has not been a new value an event on this signal. Notice that tick stable returns a signal whose value is Boolean whereas tick event returns a value whose type is Boolean.

Tick quiet is also a signal s tick quiet and essentially it means that there has been no transaction on this. Tick driving means that this signal is assigned to in the present hardware being described and tick driving value says this is the value to which the signal is currently being driven.

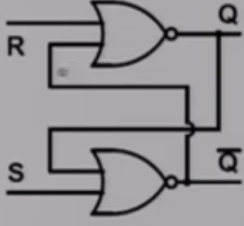
(Refer Slide Time: 37:05)

Behavioural Description
Attributes

Array attributes
Type Attributes
Signal attributes



Case of RS Latch



Entity RS_Latch is
 Port(R,S: IN BIT; Q, Qbar: OUT BIT);
 End Entity RS_Latch;
 Architecture trouble of RS_Latch is
 Begin
 $Q \leq R \text{ NOR } Qbar;$
 $Qbar \leq S \text{ NOR } Q;$
 End Architecture trouble;

This will run into trouble as Q and Qbar are declared to be outputs and cannot be used on the RHS expression of an assignment.


These last two can be understood by a somewhat annoying sequence, which comes up in a piece of hardware, which is quite common. Consider the following latch; this is the standard RS latch. Now in this case remember when we do the port declaration of this entity then we will declare R and S as input ports and Q and Q bar as the output ports. What we would like to say is that the signal Q is the NOR of R and Q bar.

Similarly, Q bar is the NOR of the signal Q and S; however, this will run into trouble. The simple description of hardware will run into trouble because Q and Q bar are declared to be outputs and these cannot be used as inputs and therefore on the right hand side of an assignment expression. So how to get around this problem, we have several choices.

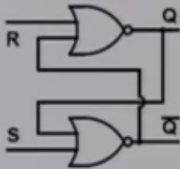
(Refer Slide Time: 38:10)

Behavioural Description
Attributes

Array attributes
Type Attributes
Signal attributes



RS Latch



We have several choices:

Declare Q and Qbar to be inout.
 This is not safe as this will allow outside circuitry to drive Q and Qbar nodes.

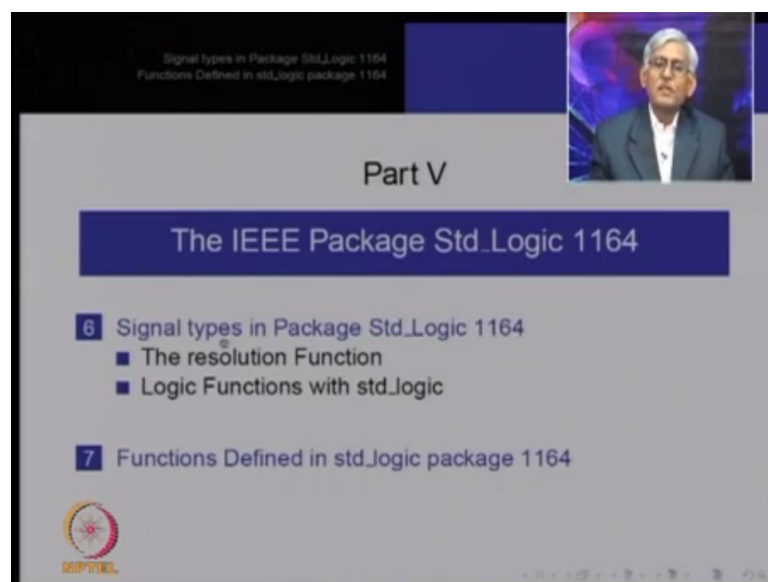
Use structural description and connect nor outputs to internal signals s1 and s2. Later assign s1 and s2 to Q, Qbar.
 Introduces artificial delay in driving of Q and Qbar.

Better choice is to use the driving value attribute

You could declare Q and Q bar of the type inout. Now the syntax error will not take place because after all Q and Q bar could be input could be outputs and you are using it as an input or an output signal; however, this is not safe as this will allow outside circuitry to drive Q and Q bar nodes. That means signals if by mistake some external hardware actually starts sending signals in through this path then that will not be flagged as an error.

Because you have declared that this is the legal operation Q and Q bar R of the type inout. It is much better to use the driving value attribute. In this case essentially you say that Q bar is the NOR of S and the driving value of Q that means whatever value you are driving to Q from inside that value and S when NOR should give you Q bar and similarly for Q. So this attribute allows you to get around these syntactical problems, where in a latch outputs are also inputs.

(Refer Slide Time: 39:26)



I think at this stage there are various special packages available with VHDL and perhaps an important and frequently used package is the IEEE package, which is called the standard logic package 1164 is a standardized package used in VHDL. Indeed, most signals that we use in realistic designs are not of the type bit, which can have only two values and will not flag and show you an erroneous value like X attached to it.

So therefore you use special signal types and to prevent different people from using incompatible signal types there is an IEEE standard package called standard logic 1164. This permits 9 valued logic, which we have actually obliquely referred to before; however, we will

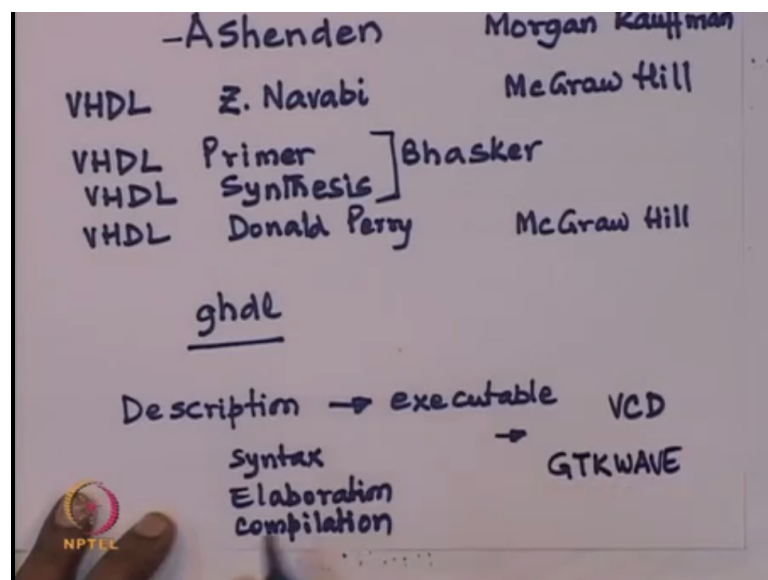
not go through the details of this. This is actually best learned by going through the documentation of this package.

In any case, it is not a part of the language it is like a library being included and this library is very frequently included. If you want to learn VHDL you would do well to read the documentation of this library. It is somewhat like the std library in C. You almost always included, but it is not part of the language similarly the standard IO inclusions in C, they are not part of the language, they are part of a library.

But we will not go through the details of this package in this course. What I would like to do at this stage is to tell you about various resources, which are available for self-learning of VHDL. There is a very large number of text books, which are available for VHDL and I would not like to single out just one or two from those; however, I will give you these examples.

I think there are many other books which are equally good, but just to get you started I would like to give you the details of a few VHDL books.

(Refer Slide Time: 41:52)



You have a book by Peter Ashenden, this book is published by Morgan Kaufman, this is quite a good book. There is a book called VHDL by Z. Navabi, it has some good examples and the multi-bit comparator that I referred to is in fact taken from this book. There is also a book called VHDL Primer by Bhasker, by the way Navabi's book is published by McGraw Hill.

And there is the old favorite on VHDL, which is by Donald Perry it is slightly dated, also published by McGraw Hill and the whole cost of other books, which are published in fact you have a very rich choice of books for VHDL most of which are quite good actually. I would like to however say that Ashenden makes the distinction between different dilates of VHDL quite clear.

So in that sense it is a good book, it gives good examples as well. Navabi has very detailed practical examples of hardware being described using the language. Bhasker also has a separate book on synthesis, which deals with how the descriptions that you use in VHDL are converted to actual hardware. Notice that not all statements are synthesizable. Only a restricted subset can be directly convert into circuits and this subset is called the synthesizable subset.

Bhaskar's book on synthesis using VHDL actually illustrates how various synthesizable constructs are actually translated to pieces of hardware. So these are quite useful resources; however, as I had said earlier the important point about hardware description languages is to practice. Just learning the theory and the syntax will not give you an actual understanding of the hardware description languages.

Now you may have access to professional software for VLSI design. So you might have say Kayden software installed in your lab in that case you are welcome to use it; however, excellent public domain implementations of VHDL are in fact available and if you do not have a lab equipped with a somewhat expensive professional software you can still practice your skills in designing with VHDL.

And I would like in particular to mention a package called GHDL. This is an implementation of VHDL, which is public domain, which is available for Linux kind of platforms and many other platforms as well. What GHDL does is that it converts the description to an executable program and when this executable program is run, the output is what you would expect this similar to output to be.

So it is essentially a multi-step process in fact. You go through a step of first syntax check, next elaboration you elaborate the design, and finally a compilation. The compilations step

then gives you this executable. Now you run this executable and there are scripts, which essentially will do this whole sequence of operation in apparently one step, but actually these are separate steps followed by GHDL and then you run this executable if everything here has been successful.

And the output of this executable will be the output of the VHDL simulation. GHDL is completely public domain indeed even the basic source code of GHDL is available if you are interested. It is actually compiled in Ada and you need not install Ada if you do not intend to compile GHDL for yourself. You can get essentially executable versions of GHDL and these are available for almost all distributions of Linux and directly installed in that case you do not have to install the Ada extensions to GCC the compiler.

If on the other hand you would like to see the source code, then you will have to install the Ada extensions of GCC you do that then you compile GHDL for yourself you will be able to compile newer and newer versions of GHDL as they come out. Now the output of GHDL is then largely text based, but there are some special programs, which allow you to graphically view the waveforms and these programs are also in the public domain.

And again there are multiple programs, which allow you to view what GHDL does is to borrow a format from Verilog and dumps its output in a format called VCD and this format can then be interpreted by plotting programs, which will produce the graphical waves. There are many, many programs available on this, just search on the keyword VCD viewers. There is this GTK wave, which is just one such program, but there are many other programs, which are used for viewing the output graphically.

With this we shall bring to an end this discussion on VHDL. We shall take just one turn on introducing very briefly Verilog and that will finish the hardware description languages of this part of VLSI design.