Advanced VLSI Design

Prof. D. K. Sharma
Department of Electrical Engineering
Indian Institute of Technology – Bombay

Lecture - 19
Introduction to VHDL

In our previous lecture, we have looked at underlying principles of Hardware Description Language, we have seen that these are similar to programming languages in many ways, however there are many important ways in which a hardware description language is very different from a programming language.

In particular, two of the features which are different for hardware description language or that it treats time in a specific way it keeps track of the simulation time the circuit time differently and second it handles concurrency in a different way, so in an hardware many components are active at the same time they must be accounted for and we have seen through a couple of illustrative examples how delay is handled and how this concurrency is managed.

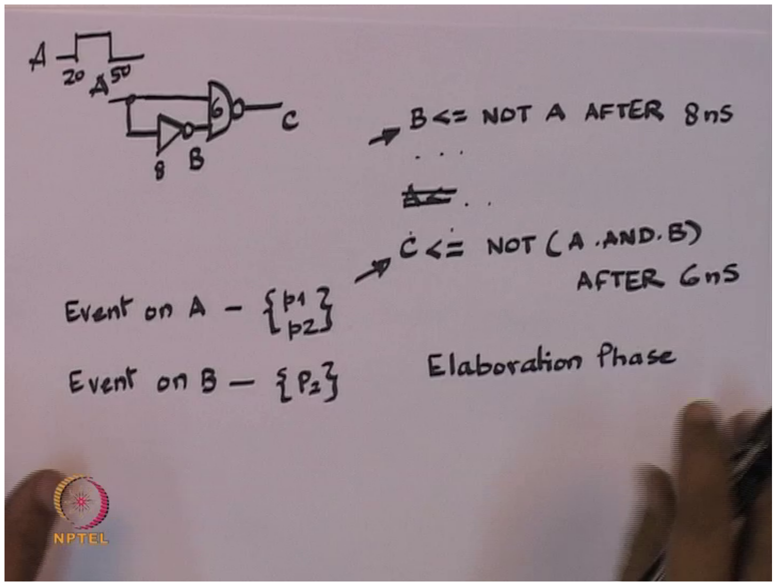
If you recall we had the example in which there was this inverter and a NAND gate and at different times this sensitivities of different components were struck and they are called upon to re-simulate their outputs, now one principle which comes out from this then is that unlike programming languages the place where a particular description is kept is unimportant.

This particular description is run not when it turns comes up sequentially as would happen in a programming language, but when it sensitivity is struck and once the sensitivity is struck the data structures that we construct during the elaboration phase will take us to that line wherever it is kept in our description, so therefore essentially the concurrent part of the description can be anywhere in the description without affecting the final outcome of the simulation.

This is a very important difference from programming languages, this however brings forth in different future which we should examine before we go over to specific programming languages.

So let us look at what happens during simulation, if you recall we had done that example of inverter and a NAND, so let us and do not worry too much about the syntax at this level.

(Refer Slide Time: 03:31)



Let us look at what happened when let us say this sensitivity of the inverter was struck, somewhere we have a description which is the equivalent of the following it says I just read row the circuit that we had simulated, this was node A, this was B and this was C and the behaviour of the inverter and the NAND is described somewhere in our hardware description language, for example it might be that for the inverter we said recall that the delay here was 8 units and this was 6 units.

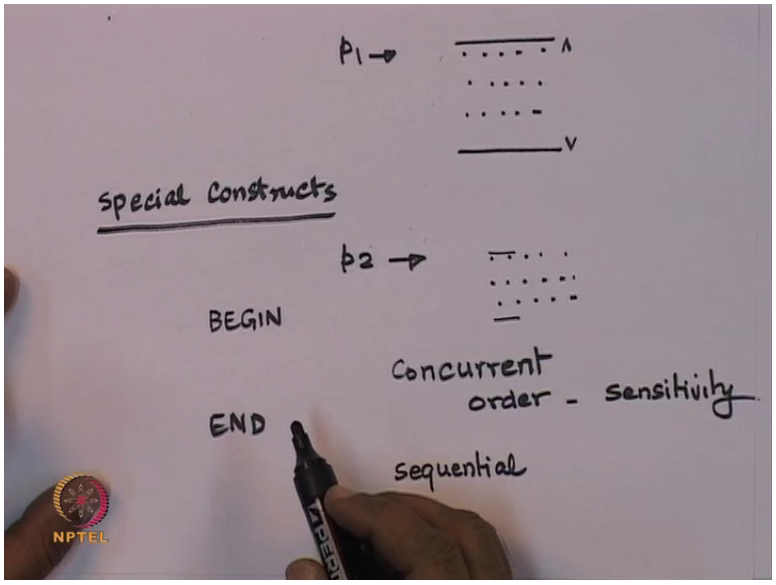
So we had somewhere the equivalent of assign to be the value of NOT A after say after 8 units of time say nanoseconds, similarly somewhere in our description we would have the equivalent of this and again the syntax is not important, assign to C the value of NOT of A and B after say 6 nanoseconds, now these descriptions are there somewhere there might be other lines somewhere in this description.

And we had described this initial waveform where you had them transitions on this was the waveform where you had then transitions on this was the wave form for A and the transitions were 20 and 50, so the point is that whenever there was an event on A our programmer will have, our simulator will have a pointer to these two lines and we will execute the code wherever this pointer takes us.

If on the other hand there is an event on B then we want to execute only this part and not that part, so let us say internally that these two pointer are p1 and p2 them and event on A would have essentially a list which will say execute whatever is being pointed to by p1 as well as what ever been pointed to by p2, on the other hand if there is an event on B then execute the code which is pointed to by p2, this data structure is built during the elaboration phase.

So therefore the simulator goes through the time ordered queue updates the signals and looks for events, for every event there is a list like this and then it goes follows that pointer and executes whatever code that pointer takes you to, this is simple if the hardware being described is very simple, for example the behaviour of the inverter is finished in just one line so is the behaviour of the NAND, but what if each one of this modules was quite complicated and in fact required a longest description.

(Refer Slide Time: 08:25)



In other words, we are looking at a case when we have several modules and let us say that you have p1 pointing to some complicated descriptions and similarly p2 pointing to a much more complicated description, now when a sensitivity is struck which requires you to execute the code at p1 you know where to begin executing but you do not know where to stop, you do not know where the description of this particular hardware ends, you do not know where it starts that is what p1 tells you.

But it does not tell you how many lines of code are to be executed that is to say we do not know how long is the behavioural description of this component, therefore you need constructs, special constructs which will delimit the extent of code which describes one particular piece of hardware very often this might be statements like begin and end, notice that these keywords might be used in other contexts as well.

But we do need to delimit an entire block of statements which will be executed when the sensitivity of this whole block occurs, now we have a somewhat mixed case this blocks of code are concurrent, therefore their order is determined by this sensitivity, but once you start executing them then the entire block must be executed.

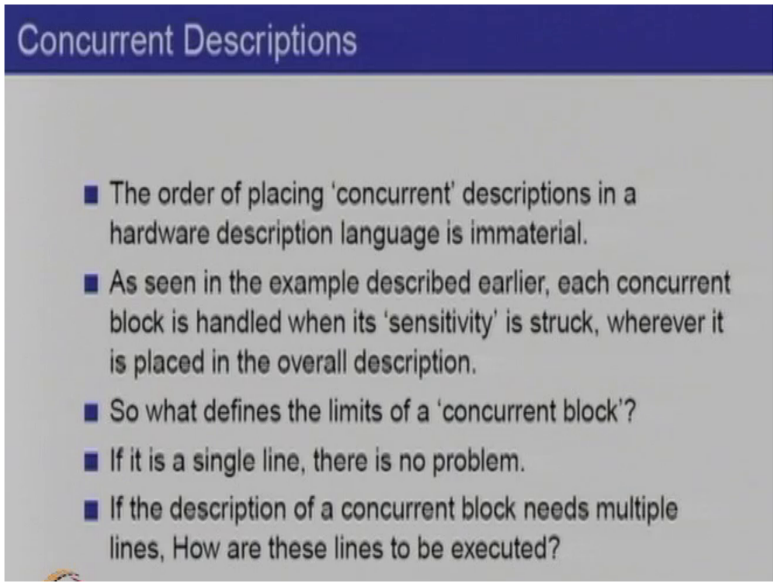
And what that means is that within this block it must be executed as if it is a software program that means it is no more concurrent it is in fact sequential, that means these lines are to be treated as an atomic code all of which is to be executed just like software one line after the other and the entire block is to be executed, because this is what executes this is what delimits the behaviour of the circuit module whose sensitivity has been struck.

Similarly, all modules will be delimited by some statements and when the sensitivity of that particular module is struck then the entire code which simulates that component needs to be executed and then now we have the flow of the simulation going in a mixed way, we have as before the time ordered queue we advanced the time to the earliest time as we had seen during the interview - during the earlier lecture.

And then we execute the code of all the hardware which is sensitive to the events which have occurred and code now we realize need not be single line it could be an entire block of code and then that block must be executed as if it is sequential and program like, so now blocks are executed like concurrent and within the block statements are executed as if they are software programs are sequential.

This mixture is the essence of the way hardware description language is operated and therefore they need to have syntactical devices which tell us which part of the code is concurrent and which part of the code is sequential, eventually the entire description is a collection of concurrent code, however each piece of concurrent code can be sequential this is what we are getting at, so let us look at it from a different point of view.

(Refer Slide Time: 14:05)



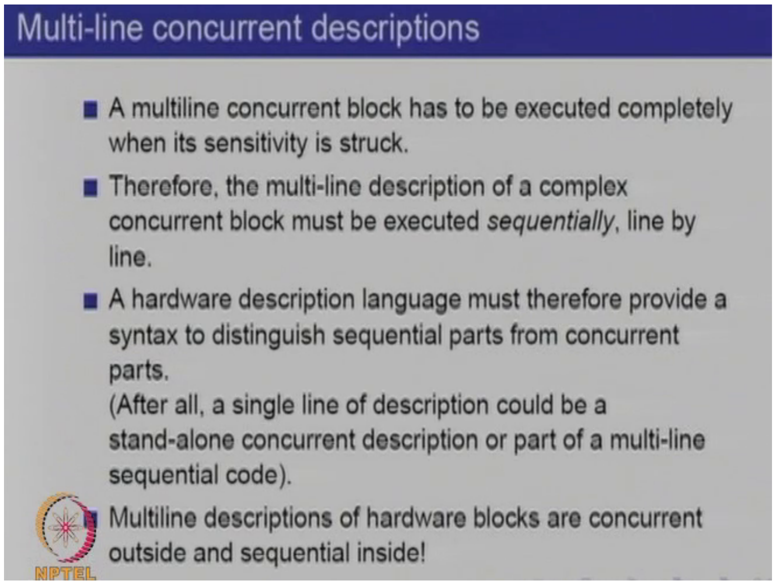
Concurrent Descriptions

- The order of placing 'concurrent' descriptions in a hardware description language is immaterial.
- As seen in the example described earlier, each concurrent block is handled when its 'sensitivity' is struck, wherever it is placed in the overall description.
- So what defines the limits of a 'concurrent block'?
- If it is a single line, there is no problem.
- If the description of a concurrent block needs multiple lines, How are these lines to be executed?

We have seen that the order of placing concurrent descriptions in a hardware description language is immaterial, as we had seen in the example described earlier each concurrent block is handled when it sensitivity is struck wherever it is placed in the overall description, so what defines the limits of a concurrent block if it is a single line there is no problem you just execute that line and you are done with it.

If the description of the concurrent block needs multiple lines how are these lines are to be executed, we need to delimit them.

(Refer Slide Time: 14:41)



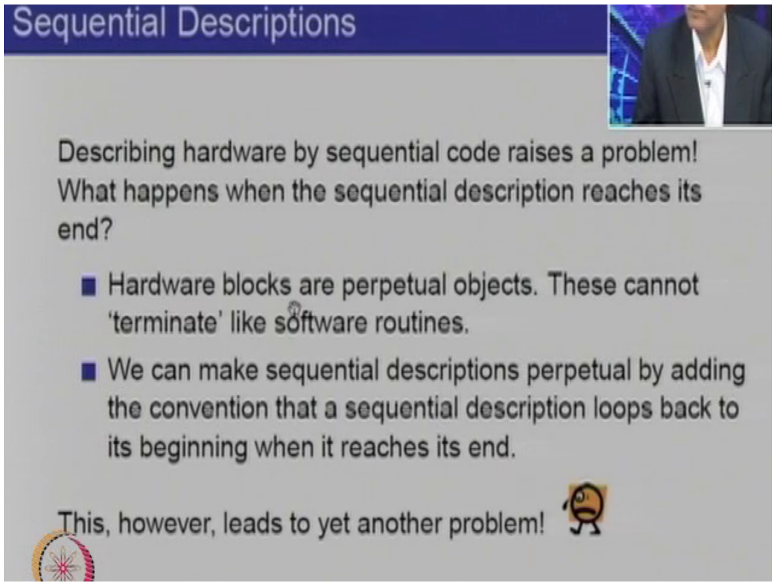
Multi-line concurrent descriptions

- A multiline concurrent block has to be executed completely when its sensitivity is struck.
- Therefore, the multi-line description of a complex concurrent block must be executed *sequentially*, line by line.
- A hardware description language must therefore provide a syntax to distinguish sequential parts from concurrent parts.
  (After all, a single line of description could be a stand-alone concurrent description or part of a multi-line sequential code).
- Multiline descriptions of hardware blocks are concurrent outside and sequential inside!

So a multi-line concurrent blocked has to be executed completely when it sensitivity is struck, therefore the multi-line description of a complex concurrent block must be executed sequentially line by line, a hardware description language must therefore provide a syntax to distinguish sequential parts from concurrent parts. After all, a single line of description could be a standalone concurrent description or part of a multi-line sequential code.

So you do not know whether this single line is complete by itself or is it a part of a multi-line sequential, so we must have syntactical devices which will tell us how this line is to be treated and multi-line descriptions of hardware blocks are concurrent outside and sequential inside.

(Refer Slide Time: 15:36)



Sequential Descriptions

Describing hardware by sequential code raises a problem! What happens when the sequential description reaches its end?

- Hardware blocks are perpetual objects. These cannot 'terminate' like software routines.
- We can make sequential descriptions perpetual by adding the convention that a sequential description loops back to its beginning when it reaches its end.

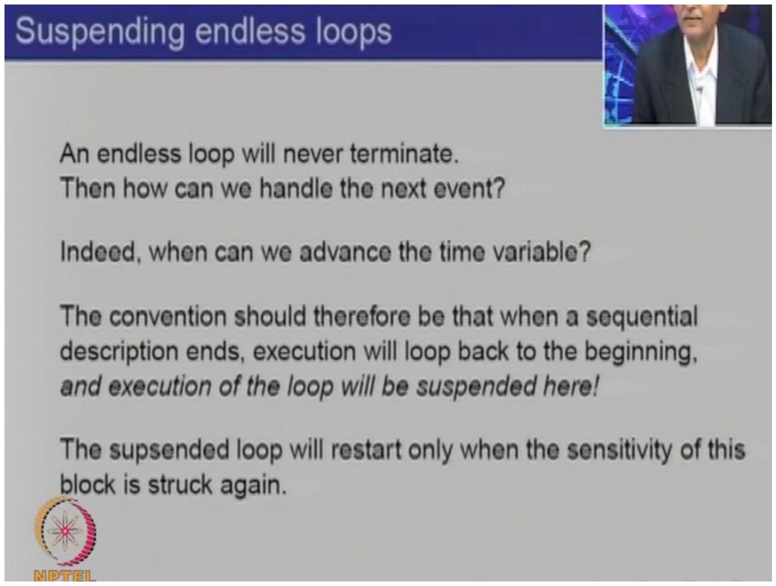
This, however, leads to yet another problem!

How the describing description of hardware by sequential code raises a problem! What happens when the sequential description reaches its end? The hardware blocks are perpetual objects, these cannot terminate like software routines, therefore we must have an assumption about the behaviour of this software block, what happens when you reach the end of this blog?

So we can make sequential descriptions perpetual by adding the convention that a sequential description loops back to the beginning when it reaches its end, in that sense now we have solved the problem this sequential code is the representation of a piece of hardware, the piece of hardware perpetual. The software has a termination and we get rid of the termination by making it perpetual by making it look back to the beginning when it reaches the end.

So now the software block is also perpetual, however this solution immediately causes yet another problem.

(Refer Slide Time: 16:57)



Suspending endless loops

An endless loop will never terminate.
Then how can we handle the next event?

Indeed, when can we advance the time variable?

The convention should therefore be that when a sequential description ends, execution will loop back to the beginning, *and execution of the loop will be suspended here!*

The supsended loop will restart only when the sensitivity of this block is struck again.

An endless loop will never terminate, then how can we handle the next event? the next event is waiting to be handled and the current event is represented by an endless loop, so this loop will go on forever and now we are struck our time cannot advance and we cannot simulate the rest of the hardware, the solution to this is actually easy, we loop back to the beginning and halt there that is all that we need to do.

That means when we reach the end of the description which describes a particular piece of hardware then we do loop back to the beginning but having loop back we stop there and then go over and handle the next event, now we have solved the problem, so the convention should therefore be that when a sequential description ends, execution will loop back to the beginning and the execution of the loop will be suspended here.

Suspended loop will restart only when the sensitivity of this block is struck again, by the way this describes a somewhat simple case of sensitivities which we have seen later we will have reason to see that there can be dynamic sensitivities that means a piece of software may actually suspend in the middle and wait for some different event to take place to restart from where its stopped that is a special case which we will we looking at it later.

But right now it is enough to see how a piece of sequential code can represent the description of a piece of hardware, so now the hardware perpetual, software is not, software terminates and we get rid of this by making the software loop back and stop there. So now the parallel is complete and we can represent hardware by such descriptions.

Having done the background work of understanding how we carry out any hardware description, we now look at an example of a specific hardware description language and in this respect we may shall have a brief look at VHDL as well as Verilog, these are the two leading hardware description language used for VLSI design these days.

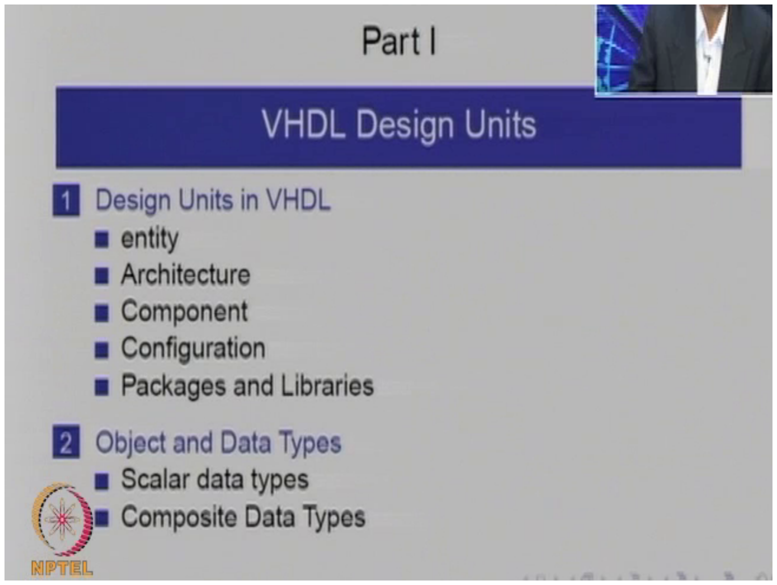
Hardware description languages ideally are learnt by practice you cannot learn VHDL or Verilog through a course of lectures like this one, therefore the endeavor during this lectures is to explain the concept which are important for understanding these languages, we strongly encourage you to actually use hardware description language and Hardware descriptions for simple circuits at first and complicated circuits later and that is the real learning of specific hardware description language.

However, in this series of lectures we have included some discussion on the underlying principles of specific hardware description languages as well, at this point I would like to say that excellent text books are available both for VHDL and Verilog we do not recommend a particular text book at this time many excellent text books are available. Later when we end this module, we will give you a list of text books that you might find interesting it is by no means exhaustive and new excellent books keep appearing for both VHDL as well as Verilog.

I would also like to add at this point that very good public domain implementations of VHDL simulation and Verilog simulation are available. In some of the examples that I take up during this course of lectures I shall be using GHDL for implementation of VHDL and I shall be using Icarus Verilog which is also a public domain program both of these run on Linux but I understand that Windows versions are also becoming available.

However, I shall be using only the Linux versions of these. Generally, the most up-to-date versions are to be found for Linux, it is quite easy to install these if you have a Linux machine available and the specific examples that it takes will in fact run on Linux, that done. Let us now have a look at VHDL.

(Refer Slide Time: 22:06)



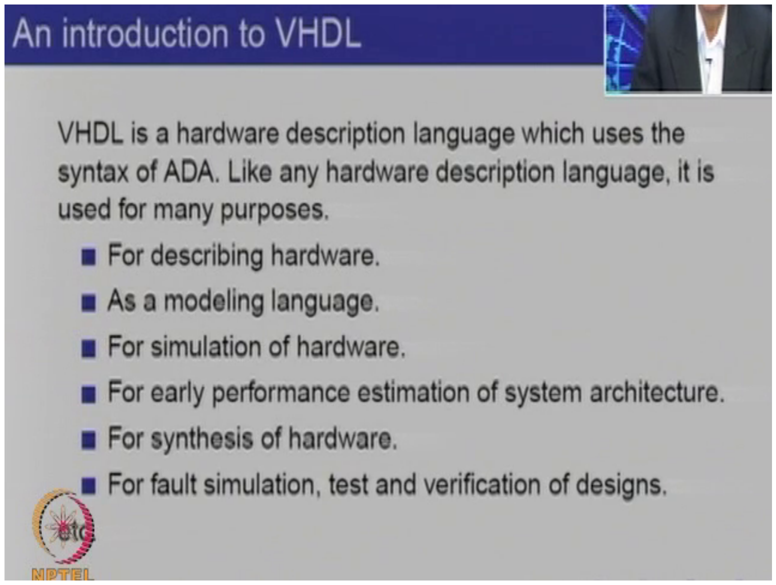
Part I

VHDL Design Units

1. Design Units in VHDL
   - entity
   - Architecture
   - Component
   - Configuration
   - Packages and Libraries
2. Object and Data Types
   - Scalar data types
   - Composite Data Types

Now, we are going to look first at the design units in VHDL and then look at object and data types and how these are handled in VHDL.

(Refer Slide Time: 22:16)


An introduction to VHDL

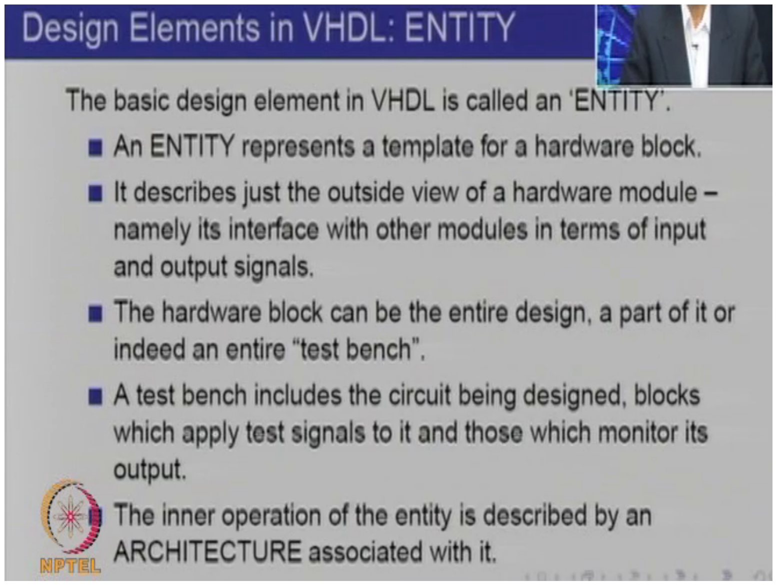
VHDL is a hardware description language which uses the syntax of ADA. Like any hardware description language, it is used for many purposes.

- For describing hardware.
- As a modeling language.
- For simulation of hardware.
- For early performance estimation of system architecture.
- For synthesis of hardware.
- For fault simulation, test and verification of designs.

Now VHDL is a hardware description language which uses the syntax of a programming language called ADA, now ADA is perhaps not very widely used for software but as I had said earlier all hardware description languages take a page out of software and therefore they form the base as some programming language and for VHDL that language is ADA, like any hardware description language VHDL is used for many purposes.

It used for describing hardware, as a modelling language, for simulation of hardware, for early performance estimation of system architecture, for synthesis of hardware, for fault simulation, test and verification of designs etc. these are some of the major uses of any hardware description language in particular VHDL.

(Refer Slide Time: 23:15)



Design Elements in VHDL: ENTITY

The basic design element in VHDL is called an 'ENTITY'.

- An ENTITY represents a template for a hardware block.
- It describes just the outside view of a hardware module – namely its interface with other modules in terms of input and output signals.
- The hardware block can be the entire design, a part of it or indeed an entire "test bench".
- A test bench includes the circuit being designed, blocks which apply test signals to it and those which monitor its output.
- The inner operation of the entity is described by an ARCHITECTURE associated with it.

The basic design element in VHDL is entity, a piece of hardware is called an entity. An entity represents a template for a hardware block, by template I mean the following, in a hardware description language for example you might describe the behaviour of a flip-flop, but later you might use several flip-flops in a design and during simulation each is flip-flop is to be used differently, it is inputs are different, it is sensitivity list is different, it is connected to different signals.

However, the behavioural description for each one of these components is the same, therefore for a type of component we describe the behaviour only once and this code will be invoked with specific inputs and specific output transactions, whenever the sensitivity of a particular instance of this kind of a component is there, so therefore we need a template for a kind of hardware and we can describe this hardware once and for all.

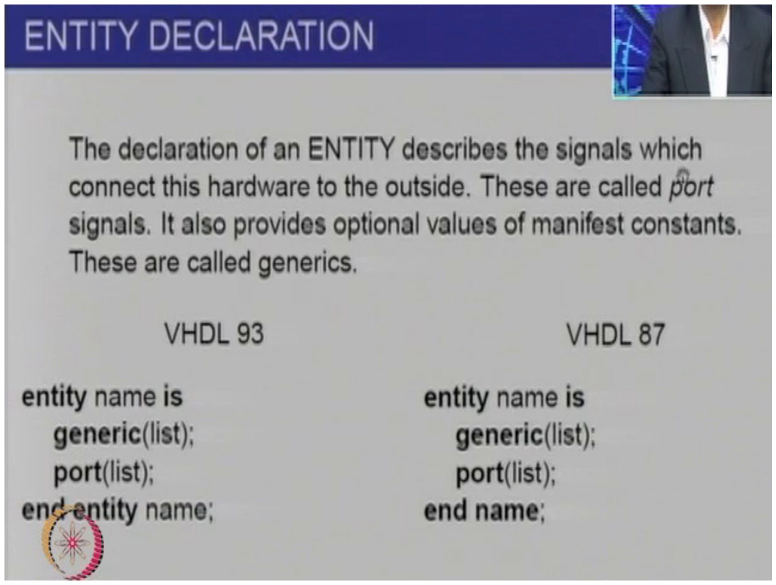
We are quite aware of the fact that several instances which are independent and which will be simulated independently might exist in a hardware description language, those will then be called components, however the entity therefore is a template for a hardware block, it describes just the outside view of a hardware module, it does not say how it works inside, it simply describes its interface with other modules in terms of input and output signals.

The hardware block described by an entity could be the entire design a part of it or indeed an entire test bench which contains the circuit being designed, what is the test bench? It includes the circuit being designed, various blocks which apply test signals to it and those which are connected to its output in order to monitor its behaviour. The inner operation of the entity the external interconnect is described by the entity.

But the inner operation of the entity is described by an object called an Architecture and therefore each entity has one or more architectures associated with it, so therefore a piece of hardware has a template you might have many instances of the same kind in a hardware, however the behavioural description - the structural description of such an object is done using a template the exterior behaviour of this template is described by an entity.

And the inner working of this kind of hardware is described by an architecture, later we instantiate components of this kind whose template is described by this entity architecture and specific instances will be independently simulated when we simulate the entire structure, so this picture should be clear in your mind therefore the entity architecture pair describes the template for a kind of hardware and not one specific instance of this hardware.

(Refer Slide Time: 27:27)



ENTITY DECLARATION

The declaration of an ENTITY describes the signals which connect this hardware to the outside. These are called *port* signals. It also provides optional values of manifest constants. These are called generics.

VHDL 93	VHDL 87
entity name **is**	**entity** name **is**
generic(list);	**generic**(list);
port(list);	**port**(list);
end entity name;	**end name**;

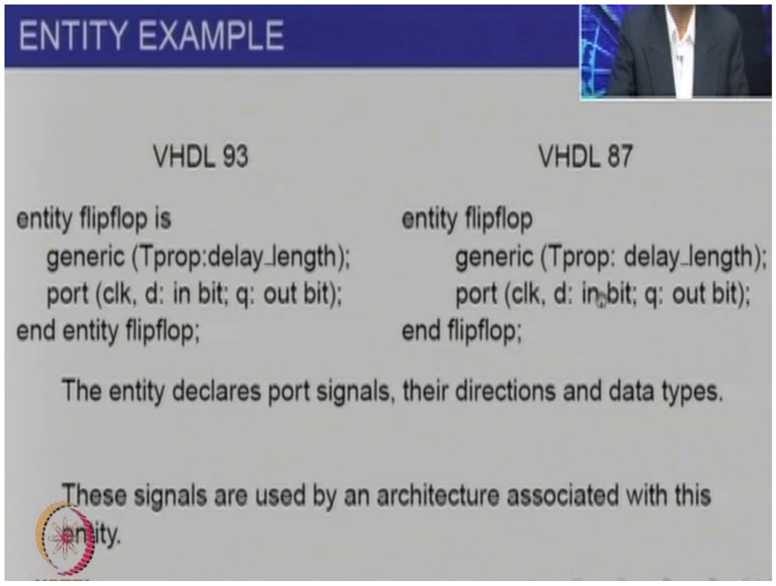
The syntax is also important, the declaration of an entity describes the signals which connect the hardware to be outside and that is all, the rest is done by the architecture. These are called port signals, it also provides optional values of manifest constants which are called generics, for example the delay or whatever the width of a bus, these are called generics. The description of the architecture might be such that it is general.

For example, you might have a buffer and the description could be for an 8-bit buffer or a 16-bit buffer, it does not matter the discretion might be the same, it just needs to know how many bits buffer is being described, such manifest constants are described during the entity apart from the interconnection with the outside world and then they are called generics, so you have a list of generic and a list of signals which form the ports.

Notice that the language has evolved over time there are several versions of this language there is VHDL 83, VHDL 87, VHDL 93 and much later versions also, but two versions are dominate, the more modern version which is VHDL 93 is more consistent in its syntax, the earlier version was not quite consistent sometimes you used like you can see here entity name is but end name, whereas VHDL 93 uses a consistent syntax entity something is and then end that object and the name.

So this kind of syntax is universal in VHDL 93, whereas it uses somewhat variable in VHDL 87 for entities this specific keyword entity was not required for architecture and other things it was required, so such inconsistencies have been removed in a more modern version of VHDL, however most simulators are capable of handling both the 1987 syntax as well as 93 and more modern versions of syntax.

(Refer Slide Time: 30:04)



ENTITY EXAMPLE

```
VHDL 93                                    VHDL 87

entity flipflop is                         entity flipflop
    generic (Tprop:delay_length);              generic (Tprop: delay_length);
    port (clk, d: in bit; q: out bit);         port (clk, d: in bit; q: out bit);
end entity flipflop;                       end flipflop;
```

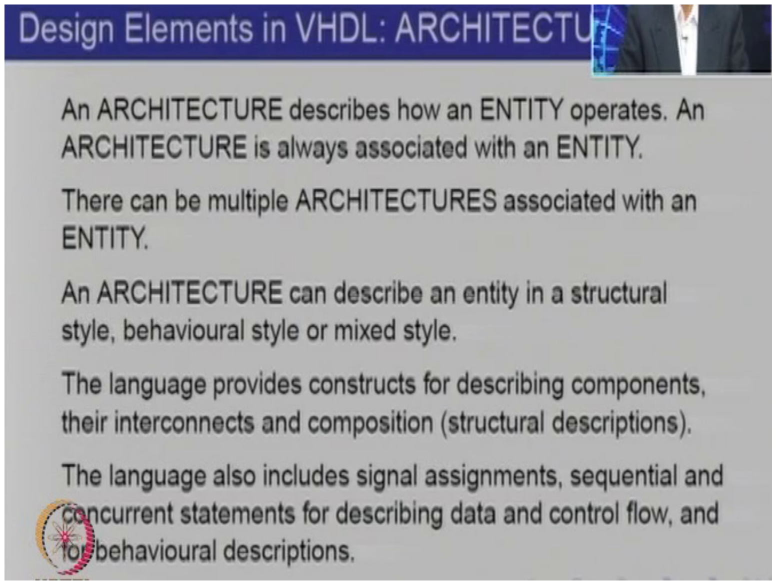
The entity declares port signals, their directions and data types.

These signals are used by an architecture associated with this entity.

Here is an example of how you describe a particular entity we are describing a flip-flop, so you have an entity flip-flop and you have a generic called Tprop which is the delay of this, then you have a port, port list the ports describes clock and d as two signals which are inputs and are of the type bit and q as a signal which is the output and is also of the type bit and end flip-flop that is all the entities supposed to contain.

In the more modern version of VHDL the first part is essentially the same entity flip-flop is, generic and port list and then end entity flip-flop rather than just end flip-flop, however most languages tend to support both kinds of syntax but for all your new descriptions it is better to stick to the more consistent syntax described by the later versions of the language.

(Refer Slide Time: 31:22)



Design Elements in VHDL: ARCHITECTU

An ARCHITECTURE describes how an ENTITY operates. An ARCHITECTURE is always associated with an ENTITY.

There can be multiple ARCHITECTURES associated with an ENTITY.

An ARCHITECTURE can describe an entity in a structural style, behavioural style or mixed style.

The language provides constructs for describing components, their interconnects and composition (structural descriptions).

The language also includes signal assignments, sequential and concurrent statements for describing data and control flow, and for behavioural descriptions.

Now let us go to the Architecture, now an architecture describes how an entity operates. An architecture is always associated with an entity, the architecture name itself is not sufficient to determine which piece of hardware is being described, so while architecture describes the inner working of a piece of hardware it is always to be considered in association with the entity whose architecture is being described.

There can be multiple architectures associated with an entity, for example I might start describing a microprocessor as we had discussed earlier we would like to design it hierarchically, in this hierarchy I might describe it in terms of various units like the bus interface unit, the instruction decoder, the arithmetic and logical unit etc. etc. Each one of these units will then be an entity, however at this point I do not know how to implement these units.

So I will describe them behaviourally, so it will have a behavioural architecture associated with each of the units, later as my design progresses I might come down right down to the logic description of the ALU or what have you in that case the same entity will have a new architecture the old architecture was completely behavioural and the new architectural is structural.

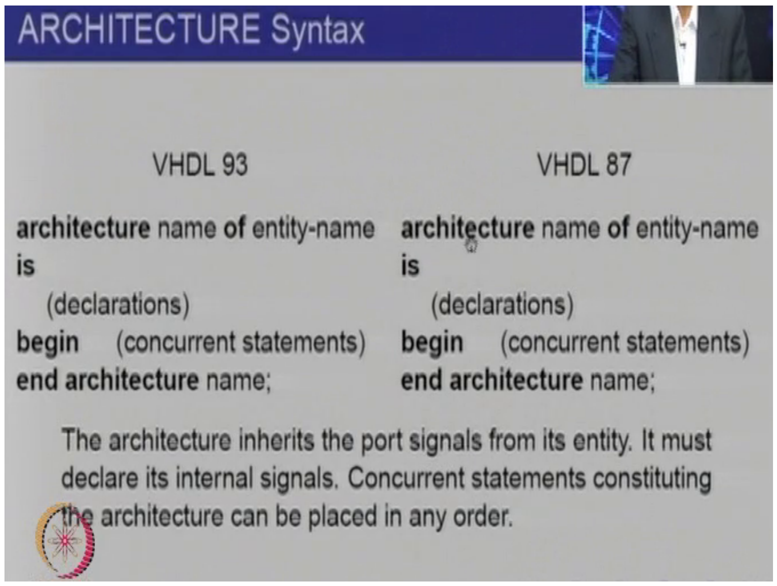
Because I have finished my design and both architectures can exist simultaneously for the same entity, we choose which architecture will be used by a syntactical device called Configuration, when we configure the design then we specify that for this instance of this entity architecture pair use this entity and this architecture.

So the entity could have multiple architectures and during a simulation you may actually choose a particular architecture for simulation and which architecture will be used will be described by a structure called a configuration, what is the architecture do? It describes how the entity actually operates that means the logic of operation of this entity remember entity architectures are template for multiple occurrences of possibly multiple occurrences of a device.

Now an architecture can describe an entity in a structural style, in a behavioural style or indeed in a mixed style that means the architecture could describe this piece of hardware as an interconnection of other components or in a what happens when kind of style or indeed mixed that means part of it is what happens when kind of logic and the rest is a description of interconnection of various pieces of hardware.

The language provides constructs for describing components there interconnects and composition these are called structural descriptions, it also includes signal assignments, sequential and concurrent statements for describing data and control flow, and for behavioural descriptions, it is the combination of all such statements which forms the architecture of an element.

(Refer Slide Time: 35:22)



ARCHITECTURE Syntax

VHDL 93	VHDL 87
architecture name **of** entity-name **is**	**architecture** name **of** entity-name **is**
(declarations)	(declarations)
begin (concurrent statements)	**begin** (concurrent statements)
end architecture name;	**end architecture** name;

The architecture inherits the port signals from its entity. It must declare its internal signals. Concurrent statements constituting the architecture can be placed in any order.

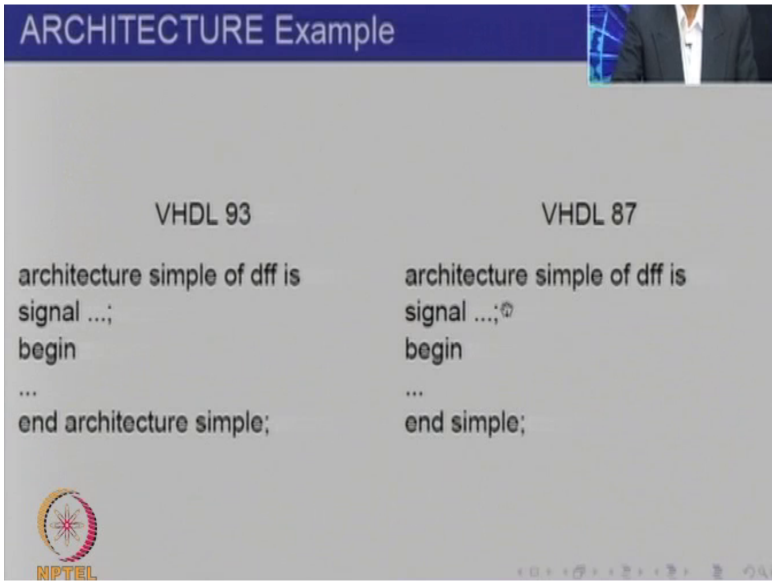
Here is the syntax of an architecture, in the earlier syntax you would say architecture this is the name of this particular architecture remember there can be multiple architectures for the same entity and therefore each architecture must have a name, so architecture this name this is the architecture name of this entity name is this is followed by declarations of internal signals and what have you.

Begin followed by end architecture this block describes the architecture of this particular component and begin and end will contain a list of concurrent statements each statement need not be a single line, these are concurrent statements and concurrent statement could even be a process which is a multi-line description of a concurrent object, the current syntax is as I said earlier is consistent architecture name of entity name is and then end architecture name.

In this particular case there is not a lot of difference between the two syntaxes entity has changed and what the new VHDL 93 has done is to make the syntactical structure of all objects which make up this language consistent. Now noticed that the architecture inherits the port signals from its entities, the port signals are like global signals for the architecture those need not be declared in fact they are not declared.

However, all the internal signals to be architecture must be declared and those are the ones which will come in this declarations component of the architecture, concurrent statements constituting the architecture can be placed as always in any order.

(Refer Slide Time: 37:41)



ARCHITECTURE Example

VHDL 93

```
architecture simple of dff is
signal ...;
begin
...
end architecture simple;
```

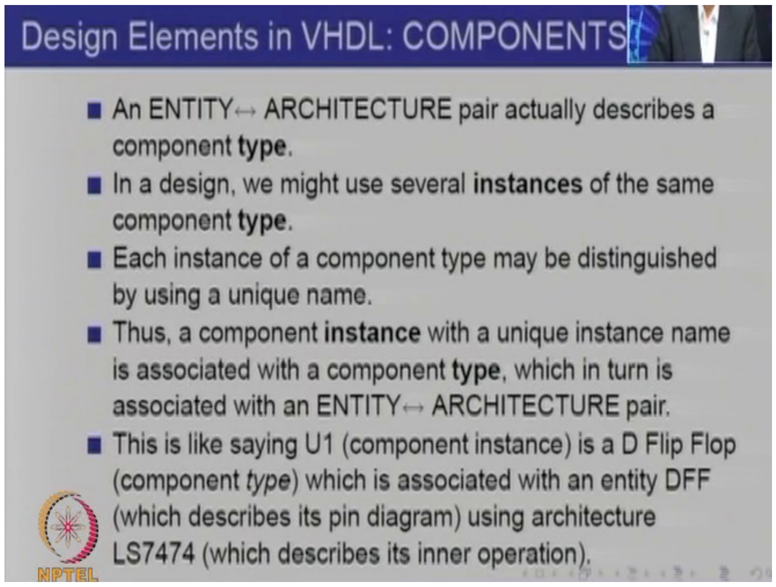
VHDL 87

```
architecture simple of dff is
signal ...;
begin
...
end simple;
```

Here is the specific example, architecture simple this is the name of the architecture, of dff, dff is the name of entity that is the entity name, so architecture simple of dff is then follow declarations like signal this that or the other, then begin and then end simple. In case of VHDL 93 you will have architecture simple of d flip-flop is, again declaration signal etc. begin and then end architecture simple.

In the modern syntax, after end, you specify what kind of object is ending and also the name of the particular type, so end architecture simple where simple was the name of this architecture.

(Refer Slide Time: 38:48)



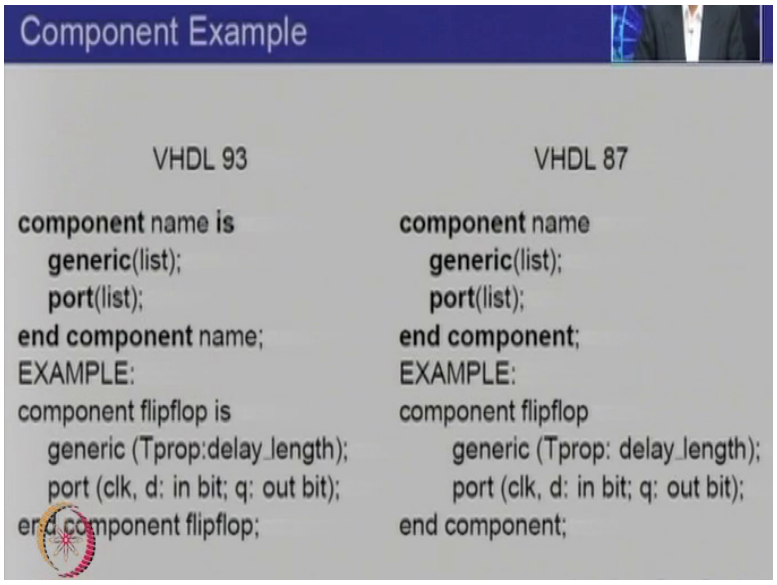
Design Elements in VHDL: COMPONENTS

- An ENTITY↔ ARCHITECTURE pair actually describes a component **type**.
- In a design, we might use several **instances** of the same component **type**.
- Each instance of a component type may be distinguished by using a unique name.
- Thus, a component **instance** with a unique instance name is associated with a component **type**, which in turn is associated with an ENTITY↔ ARCHITECTURE pair.
- This is like saying U1 (component instance) is a D Flip Flop (component *type*) which is associated with an entity DFF (which describes its pin diagram) using architecture LS7474 (which describes its inner operation).

Now let us look at specific design elements in VHDL. we have looked at entity architecture, this entity architecture pair actually describes a template as we had not seen a type, in a design we might use several instances of the same component type, each instance of a component type must be distinguished by using a unique name, thus a component instance whether unique instance name is associated with the component type which in turn is associated with an entity architecture pair.

To give an example of this, it is like saying U1, U1 is a component instance is a D flip-flop which is a component type which is associated with an entity DFF, the entity DFF will describe its pin diagram or this signals that is connected to, using architecture LS7474, so the architecture LS7474 describes its inner operation. So this is a hierarchy of description elements, the entity architecture pair is a template this is then linked to a component which associates it with a type and then specific instances of this component type will be used in an actual hardware description.

(Refer Slide Time: 40:32)



Component Example

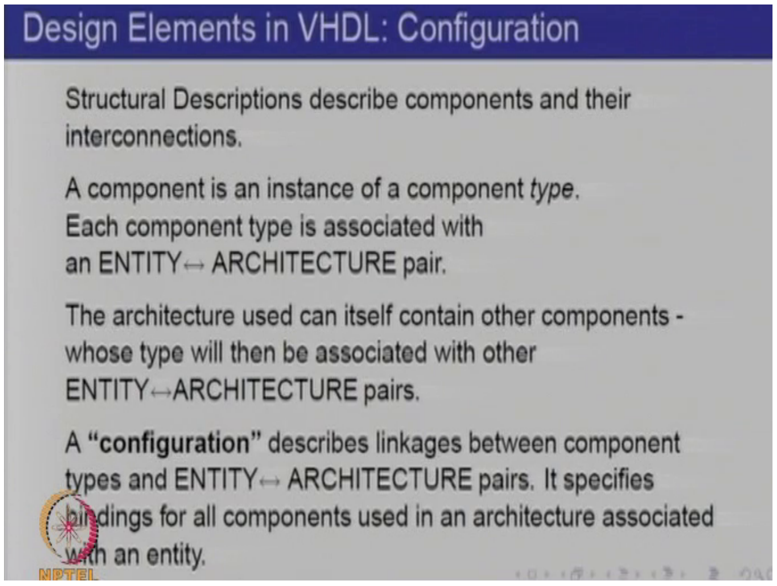
```
VHDL 93                                  VHDL 87

component name is                        component name
    generic(list);                           generic(list);
    port(list);                              port(list);
end component name;                      end component;
EXAMPLE:                                 EXAMPLE:
component flipflop is                    component flipflop
    generic (Tprop:delay_length);            generic (Tprop: delay_length);
    port (clk, d: in bit; q: out bit);       port (clk, d: in bit; q: out bit);
end component flipflop;                  end component;
```

Here is an example, you declare a component notice this is not an instance still you declare a component by this syntax, its syntax is very similar to the syntax of an entity, so you declare generics and a port list for the component, for example you say I have a component flip-flop with generic Tprop which is a delay length with ports clock, d in input of type bit and q a signal which is an output of type bit.

Now this is a component declaration it is simply telling the simulator that I can have multiple instances of this component, later I will associate this component with an entity architecture pair. In VHDL 93 however this somewhat rigid descendancy of hierarchy is relaxed, and in VHDL 93 you might instantiate and an entity architecture pair directly, VHDL 87 did not allow an entity architecture pair to be instantiated directly.

The entity architecture pair was something abstract, a component that declared a type and then instances could only instantiate components, the modern version of VHDL has relaxed it a little bit and in fact that is diluted somewhat the utility of the components, because you can for the sake of convenience at least for simple designs instantiate entity architecture pairs directly.

(Refer Slide Time: 42:36)



Design Elements in VHDL: Configuration

Structural Descriptions describe components and their interconnections.

A component is an instance of a component *type*. Each component type is associated with an ENTITY↔ ARCHITECTURE pair.

The architecture used can itself contain other components - whose type will then be associated with other ENTITY↔ARCHITECTURE pairs.

A **"configuration"** describes linkages between component types and ENTITY↔ ARCHITECTURE pairs. It specifies bindings for all components used in an architecture associated with an entity.

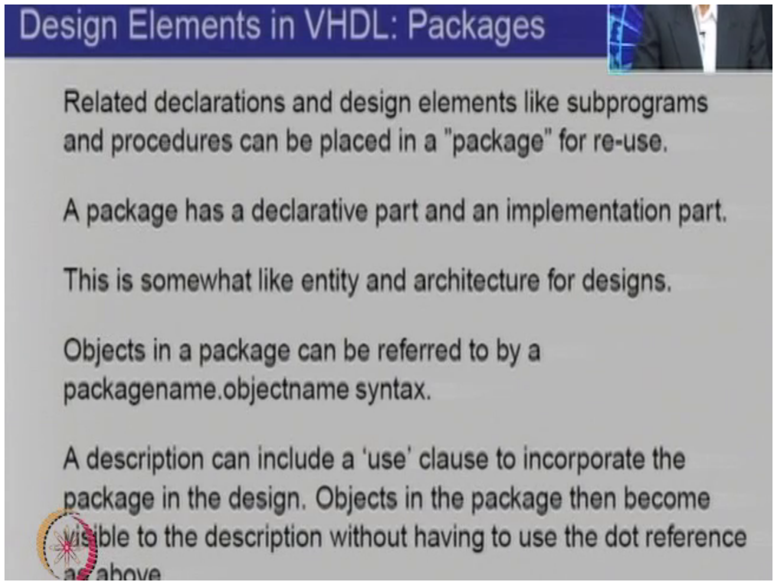
Now which architecture will be used with an entity and which entity architecture pair will be used for a component that must be made clear and for that we need something called a Configuration. Structural Descriptions describe component instances actually component instances and their interconnects, so a component is an instance of a component type each component type is associated with an entity architecture pair.

The architecture used can itself contain other components it could be a structural description, so you have an architecture which is a structural architecture and it describes a component in terms of interconnection of other components, for example you could describe a latch as an interconnection of NAND gates, now this NAND gates are also components and then those must also be associated with entity architecture pairs.

So this connection of a component with an entity architecture pair which is recursive because the architecture itself can contain other components which then are linked with other entity architecture pairs and so on, this connection is described by an object called a configuration, which describes linkages between component types and entity architecture pairs. It specifies bindings for all components used in an architecture associated with an entity.

So now we have looked at various elements which constitute a description in VHDL, we have looked at entity, we have looked at architecture, we have looked at component which actually describes the component type, we have looked at instances of components, and now we have looked at configuration which binds a component instance to a component, a component to an entity architecture pair and recursively because the architecture results might contain other components, so this is the function of a configuration.

(Refer Slide Time: 45:27)



Design Elements in VHDL: Packages

Related declarations and design elements like subprograms and procedures can be placed in a "package" for re-use.

A package has a declarative part and an implementation part.

This is somewhat like entity and architecture for designs.

Objects in a package can be referred to by a packagename.objectname syntax.

A description can include a 'use' clause to incorporate the package in the design. Objects in the package then become visible to the description without having to use the dot reference as above

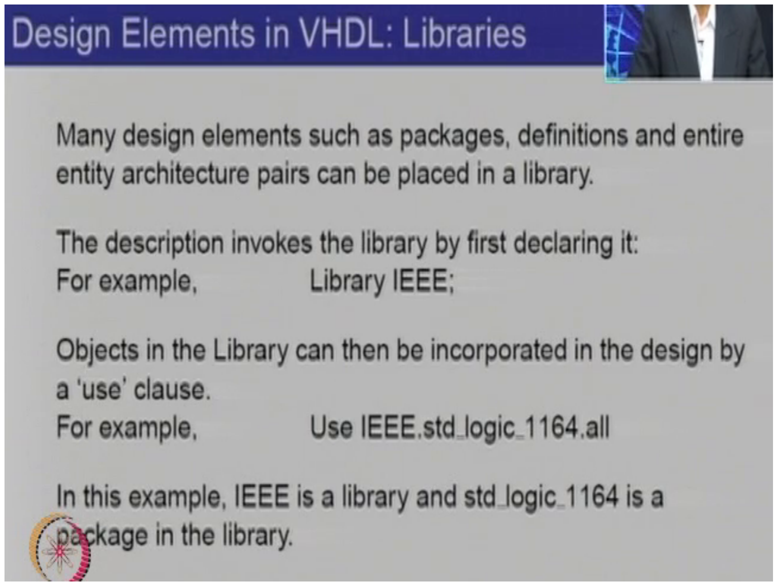
Now let us look at some other things in VHDL other elements and these are called packages related declarations and design elements like subprograms and procedures can be placed in a package for reuse. A package has a declarative part and an implementation part, so VHDL is a segmented language you would have noticed this everywhere, hardware is described separately as entity and architecture.

Similarly, a package has a declarative part which is the external interface and an implementation for which is the innards of the implementation, so this is somewhat like entity and architecture for designs. Objects in a package can be referred to by a packagename dot objectname kind of syntax, so now related objects can be placed in a package, a description can include a use clause to incorporate the package in the design.

This is somewhat like including a library during compilation of software, objects in the package then become visible to the description without having to use the dot reference as above, so when you say use this package essentially it makes that package visible to the designs and now you can use all the objects which are available in that package.

This facilitates reuse of descriptions readymade functions, signal types etc. are available as packages and indeed have been standardized for convenient use and therefore you do not have to reinvent the wheel every time many frequently done jobs are already available as packages and you can just simply include those in your hardware descriptions.

(Refer Slide Time: 47:30)



Design Elements in VHDL: Libraries

Many design elements such as packages, definitions and entire entity architecture pairs can be placed in a library.

The description invokes the library by first declaring it:
For example, Library IEEE;

Objects in the Library can then be incorporated in the design by a 'use' clause.
For example, Use IEEE.std_logic_1164.all

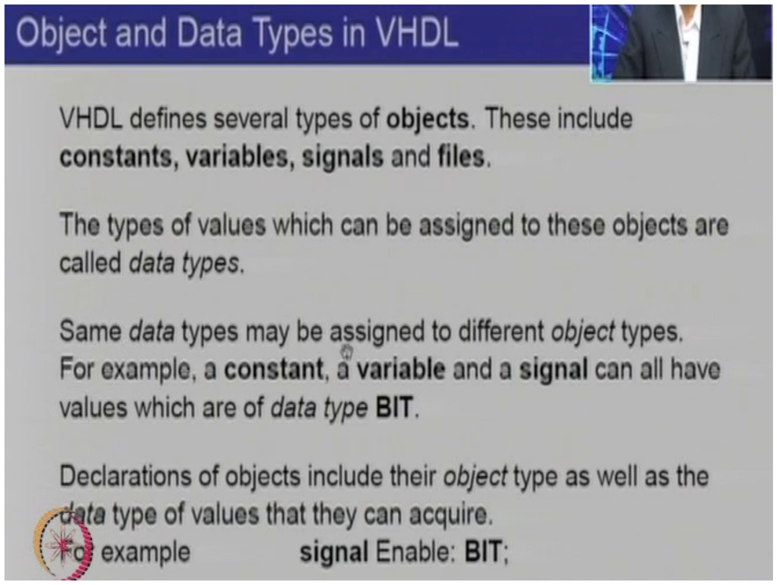
In this example, IEEE is a library and std_logic_1164 is a package in the library.

You can also have libraries, so for example many design elements such as packages, definitions, and entire entity architecture pairs they can all be placed in a library, the description invokes the library by first declaring it, for example you might say Library IEEE all that it does is it wants the software that IEEE is the name of some library that is all it does. Then objects in the library then can be incorporated in the design by a use clause.

For example, you might say use IEEE dot std logic 1164 dot all, what it says is that the library called IEEE has a package called std logic 1164 and everything which is included in that package is to be made visible for my designs, so we have the use clause for a library dot package dot package component kind of thing you could have declared just chosen components of a particular package.

For example, you might like certain elements of a package but may want to rewrite certain other elements of that package, then you have the freedom of using the use clause in which you use only selected packages which do not conflict with what you are describing and describe the rest yourself.

(Refer Slide Time: 49:15)



Object and Data Types in VHDL

VHDL defines several types of **objects**. These include **constants, variables, signals** and **files**.

The types of values which can be assigned to these objects are called *data types*.

Same *data* types may be assigned to different *object* types. For example, a **constant**, a **variable** and a **signal** can all have values which are of *data type* **BIT**.

Declarations of objects include their *object* type as well as the *data* type of values that they can acquire.
For example **signal** Enable: **BIT**;

Once we understand this superstructure of objects in VHDL, we should now look at objects and data types. VHDL defines several types of objects, please be a little patient with this terminology, there are many things that we have introduced today, we have introduced various design units, now I am talking of objects. So there are various kind of objects which are included in VHDL and this includes.

For example, constants, variables, signals and files. now the types of values which can be assigned to these objects are called data types, some data types may be assigned to different object types, so be careful about this classification these classifications are orthogonal, for example you have a data type called a bit, now an object type constant could be assigned a value of a bit.

For example, you might say a constant which we will called true is assign the value of datatype bit whose value is 1, a variable which is not a signal which is only a computational device could also have a data type called bit and finally a signal which is actually the kind of object that we have been talking about in our tutorial and so on. So that signal can also be of a data type bit, so make a distinction between object types and data types.

All object types can be constants, variables, signals and files and then there are a few others and one or more of them can use different data types, when you declare an object you include their object type as well as the data type of values that they can acquire. For example, you would declare signal enable bit, then the object type is signal the name of this particular object is enable and its data type is bit, so the declaration involves both these types.

We shall stop at this point in this discussion and then continue in our next lecture, describing how this object and their types appear in actual descriptions.