

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

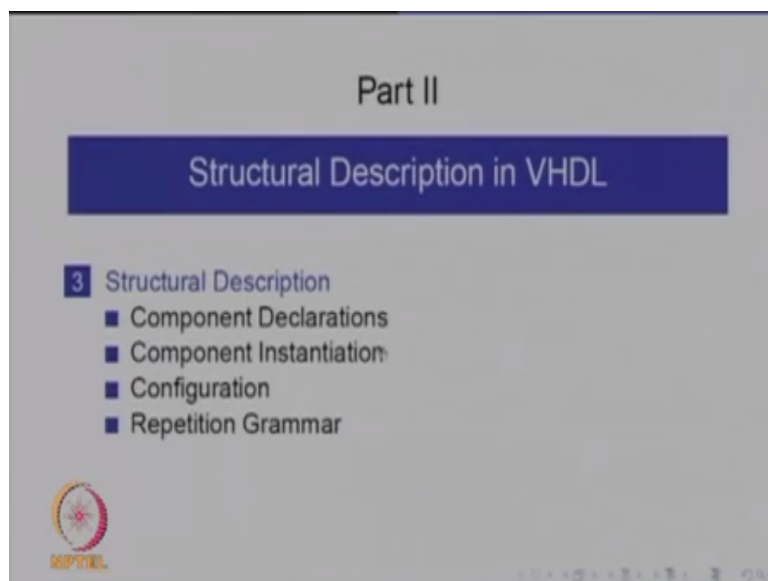
Lecture - 21
Structural Description in VHDL

In the previous lecture we have looked at various components of the VHDL language. We have looked at object types. We have looked at data types. We have seen how physical quantities can be represented with units and then we had seen composites and various user-defined types and subtypes can be used in VHDL. This defines the basic structure of the language. We now want to look at the styles in which we design or describe hardware using VHDL. We first look at structural description in VHDL.

As we have discussed earlier, structural description means that we have to name all the components that we are using in a circuit and give a list of connections from one sub circuit to the other. Therefore, the language must provide the wherewithal for placing these components of hardware, attaching these components with a particular kind of behavior and a list of wires, which will connect pins of this sub circuit to pins of other sub circuits.

All the devices in the language, which cater to this requirement they are described as structural description languages, structural description commands and those are the ones that will look at first.

(Refer Slide Time: 02:04)

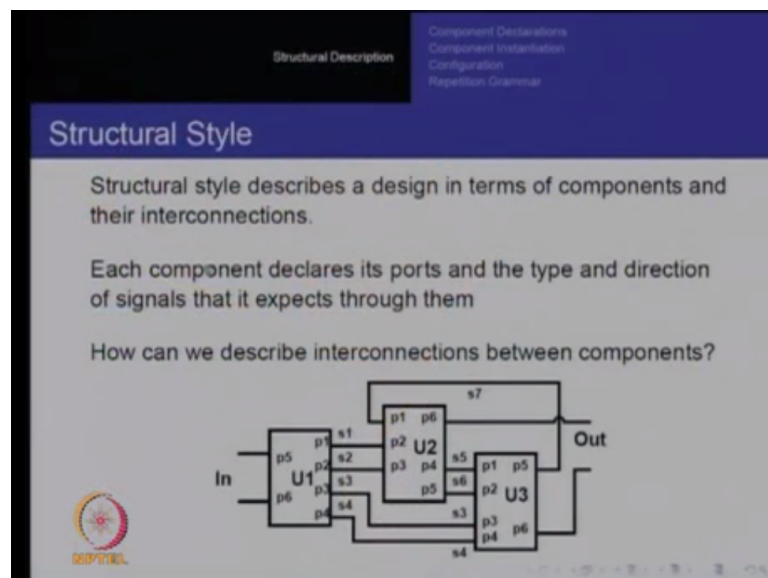


So therefore structural description consists of these parts, you have component declarations in which you declare the properties of each component that you would be using, component instantiation in which you actually place the component at various positions in the circuit, the configuration refers to the choice of binding this component to entities, to architectures and so on and very often we use multiple components of the same time.

And therefore it is convenient to have a repetition grammar to give you an example you might have eight registers in a particular design. The register is declared as a component and described it maps to a particular entity architecture pair; however, you do not need just one register, you need eight and it is painful to go ahead and describe each instantiation of this component type called register.

And therefore it is convenient if you could using repetition grammar place eight instances of this type of component.

(Refer Slide Time: 03:33)



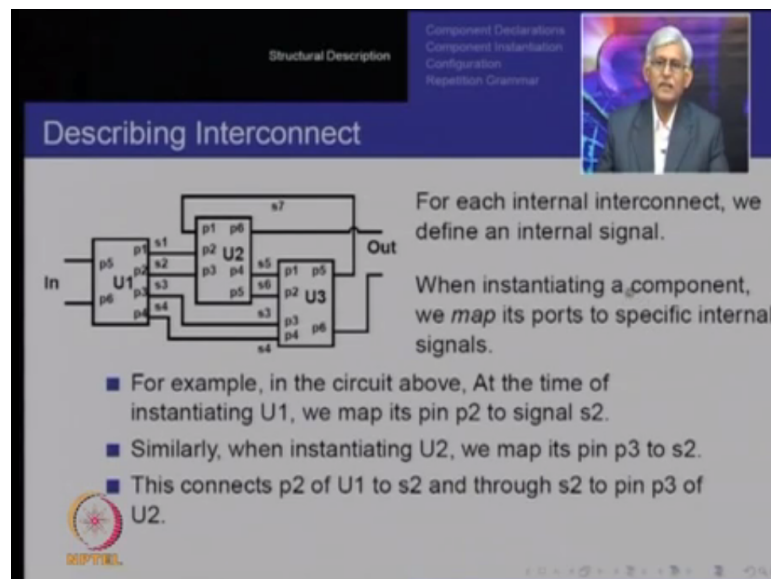
So therefore repetition grammar is an important part of structural description in VHDL. Let us look at some circuit and see how we shall go about describing it structurally. So the structural design describes a design in terms of components and their interconnects. Each component declares its ports and the type and direction of signals that it expects through these ports.

This is very similar to the declaration of an entity. So the next part is how can we describe the interconnects between these components and we take this example you have three

components here, U1, U2, and U3. Then each one has its own declaration of various ports so for example U1 has p1, p2, p3, p4, p5 and p6, U2 similarly and U3 similarly. These might or might not be identical components.

Now once we have instantiated these components the rest depends on how do we tell VHDL that p1 of U2 is in fact connected to p5 of U3 and so on.

(Refer Slide Time: 04:49)



These are done using internal signals. So for each internal interconnect we define an internal signal. When instantiating a component, we map its ports to specific internal signals. For example, in the circuit above, at the time of instantiating U1, we map its pin p2 to signal s2. Notice p2 is connected to the signal s2 so when we instantiate U1 we not only place a kind of a component type here, but we say that p2 of this instance of this component must be connected to the signal called s2.

Similarly, when instantiating U2, we map its pin p3 to s2 so U3 is a component, U2 is a component and U2 is mapped to a component type and when we instantiate that component type, we shall map its pin p3 here to that same signal s2. Therefore, it is known to VHDL that pin p2 of U1 is connected to pin p3 of U2 through the signal s2.

(Refer Slide Time: 06:20)

Structural Description

Component Declarations
Component Instantiation
Configuration
Repetition Grammar

Structural Architecture

A purely structural architecture for an entity will consist of

- 1 Component declarations: to associate component types with their port lists.
- 2 Signal Declarations: to declare the signals used.
- 3 Component Instantiations: to place component instances and to portmap their ports to signals. Signals can be internal or port signals declared by the ENTITY.
- 4 Configurations: to bind component types to ENTITY→ARCHITECTURE pairs.
- 5 Repetition grammar: for describing multiple instances of the same component type – for example, memory cells or bus buffers.

SPTEL

Now a purely structural architecture for an entity will consist of first of all components declarations, what this does is, it associates a component type not individual instance of a component, but a component type with their port lists. It is very similar to entity declarations. You also need signal declarations so that you can then use those signals to connect one instance of a component to another component.

Component instantiations to place component instances and to portmap their ports to signals. The pins can be connected either to internal signals or to the port signals declared by the entity, which is now being described. We also need configurations, these configurations will bind component types to entity architecture pairs and these configurations can be inline that is they could be part of the architecture or they could be standalone units outside the entity architecture description.


In addition to these basic requirements, we require a repetition grammar for convenience. This is for describing multiple instances of the same component type. For example, you have let say 1024 memory cells, all of them are identical. It would be indeed very inconvenient to have to describe each instance of a memory cell individually and it makes a life much simpler if you could just place all 1024 of them using a repetition grammar.

Similarly, a very large bus may have buffers associated with each line of the bus and it is convenient if we have a repetition grammar, which will describe the placing of all these buffers in one go.

(Refer Slide Time: 08:35)

Structural Description

Component Declarations
 Component Instantiation
 Configuration
 Repetition Grammar



Component Declarations

VHDL 93	VHDL 87
<pre> component name is generic(list); port(list); end component name; EXAMPLE: component flipflop is generic (Tprop:delay_length); port (clk, d: in bit; q: out bit); end component flipflop; </pre>	<pre> component name generic(list); port(list); end component; EXAMPLE: component flipflop generic (Tprop: delay_length); port (clk, d: in bit; q: out bit); end component; </pre>

So these are the components that we need for describing hardware structurally and let us look at many of these components of the language as we go along. First is the declaration of a component and these are VHDL 87 and 93 versions of the component declaration. It declares a component type not a specific instance of a component and it is very similar to the entity declaration that we have seen earlier.

I shall not drag you through the differences between VHDL 87 and VHDL 93 every time. It is here for you to see and it is better to get use to the more consistent style required by VHDL 93. So in the discussions we shall mostly look at the VHDL 93 style of declaration. So then you have the declaration says component that is the keyword what appears in bold letters here is the keyword.

So component and this is the name of this component type. So component name is is is also a keyword and then the list of all the generics and the list of all the ports. So remember this list is essentially the same format as a record data structure that we had discussed last time. And then end component name, for example we say component flipflop is that means this declaration is describing it component of the type flipflop.

Generic Tprop which is the delay length which says what is the propagation delay of this flipflop and then a port list saying port clock and d as input and these are of type bit and q as type output and bit and finally end component flipflop. This format is identical to an entity declaration, but be aware that you are not describing an entity here. You are describing a component, which will then be bound to entity architecture.

Because of this similarity in fact in VHDL 93 you need not declare a component type separately if you are always going to use the same entity architecture pair for this component and in such cases you can directly instantiate entity architecture pairs. This option was not available with VHDL 87.

(Refer Slide Time: 11:39)

Structural Description

Component Declarations
Component Instantiation
Configuration
Repetition Grammar

Component Instantiation

VHDL-93: Direct Instantiation

VHDL-93 allows direct instantiation of ENTITY ↔ ARCHITECTURE pairs without having to go through a component *type* declaration first.

Instance-name: **entity** entity-name (architecture-name)
generic map(list)
port map(list);

This form is convenient, but does not have the flexibility of associating alternative ENTITY ↔ ARCHITECTURE pairs with a component.

VHDL-87 does not allow direct instantiation.

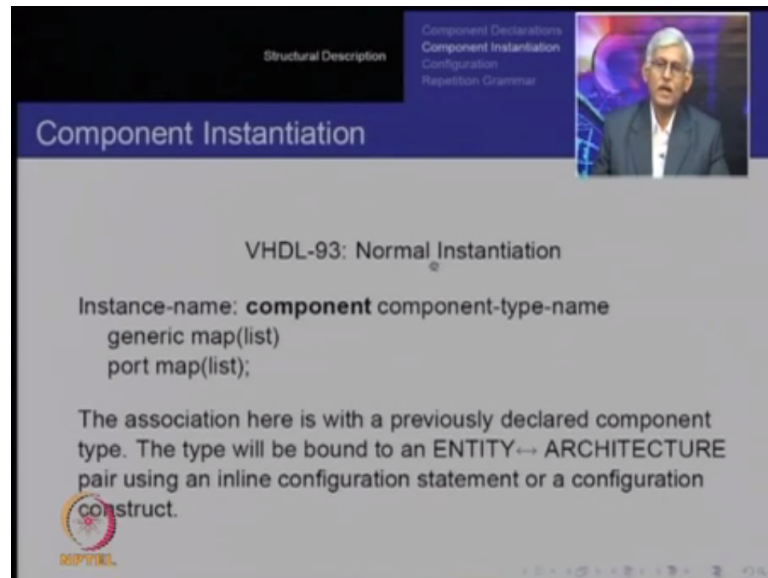
So this is what we have been talking about. In VHDL 93, you can do direct instantiation that means an entity architecture pair can be instantiated to specific pieces of hardware without having to go through a component type declaration first so this is an instance name. Notice now I am not declaring a type of component, I am declaring a specific part of the hardware. For example, this could be our U1, U2, U3 that we had seen.

Those are specific instances so you have the instance name and then the keyword entity which warns the language that I am not using a component type, I am going to instantiate an entity directly. So instance name could be U2 for example and then the keyword entity it will appear entirely like this entity and then the name of the entity and within bracket the architecture, which you want to instantiate.

So this entity architecture pair is then instantiated into a specific instance of hardware here and then you have the generic map and the port map as before. Notice that this port map now will be for this specific instance that means it will bind those pins that you have declared in the entity to specific signals. This form is convenient though it does not have the full flexibility of associating alternate entity architecture pairs with a component.

Because when you instantiate the component this is already bound to an entity architecture pair. VHDL 87 did not allow the direct instantiation and because it permitted only component instantiation, this keyword was not there at all, it was taken for granted that only components can be instantiated whereas in VHDL 93 you have both options. If you directly instantiate an entity you will use a keyword entity otherwise, you must use a keyword component.

(Refer Slide Time: 13:45)



The slide is titled "Component Instantiation" and features a video inset of a speaker in the top right corner. The main content area is titled "VHDL-93: Normal Instantiation" and displays the following syntax:

```
Instance-name: component component-type-name  
generic map(list)  
port map(list);
```

Below the syntax, the text explains: "The association here is with a previously declared component type. The type will be bound to an ENTITY ↔ ARCHITECTURE pair using an inline configuration statement or a configuration construct." A small "construct." logo is visible at the bottom left of the text area.

So this is the normal instantiation when you have declared a component type and then you have instance name. This is the specific instance that you are placing. The keyword component then the component type name now and then the generic map and the port map. The association here is with the previously declared component type. The type will be bound to an entity architecture pair using an inline configuration statement or a separate configuration construct.

(Refer Slide Time: 14:16)

Structural Description

Component Declarations

Component Instantiation

Configuration

Repetition Grammar

Component Instantiation

VHDL-87

The keyword **component** is not used in VHDL-87. This is because direct instantiations are not allowed and therefore the binding is *a/ways* to a component.

```

Instance-name: component-type-name
generic map(list)
port map(list);

```

The association is with a previously declared component type. The type will be bound to an ENTITY→ARCHITECTURE pair during an inline configuration statement or construct.

In VHDL 87, the keyword component is not required to be used because the only option you had was of instantiating component type. So there in that we used to use instance name and no keyword component, directly the name of the component and then the generic map and the port list.

(Refer Slide Time: 14:39)

Structural Description

Component Declarations

Component Instantiation

Configuration

Repetition Grammar

Inline Configuration

The association between component types and ENTITY→ARCHITECTURE pairs can be made inline with a *use* clause.

```

for all: component-name
use entity entity-name(architecture-name);

```

Instead of saying **for all**, we can specify a list of selected instances of this component type to which this binding will apply.

```

instance-name-list: component-name
use entity entity-name(architecture-name);

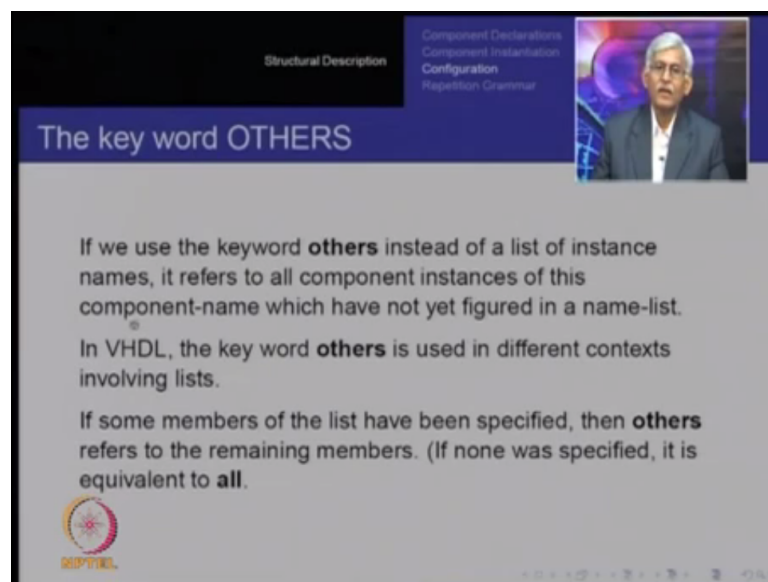
```

Now let us look at configurations and the simplest of these is an inline configuration that means the binding of a component type to an entity architecture pair is made inline with a use clause. So for example we say for all component name that means you are now saying that whenever the component name flipflop is invoked then for all instances of this use the entity with this entity name and with this architecture name.

You also have the option of not saying for all, you could specify specific instances and bind some instances to one entity architecture pair and other instances to other entity architecture pairs. So this is the additional flexibility you will get if in fact you are using inline configuration or indeed a standalone configuration. So in that case instead of for all you will have an instance name list.

You might say for example that U1, U2, U4 and then component name and then you say use the entity these two are keywords and must appear verbatim in your description. And here comes the name of the entity that you will be using and here is the architecture name. So these lines then constitute a configuration. It simply says that when an actual instance is being placed, we know eventually which entity architecture will be evoked when its sensitivity has been hit.

(Refer Slide Time: 16:46)



If we use the keyword others instead of a list of instance name it refers to all component instances of this component name, which have not yet figured in a named list. In VHDL, the keyword others is used in different contexts involving lists and if some members of the list have been specified then others refers to the remaining members. If none were specified, it is equivalent to saying all.

(Refer Slide Time: 17:22)


Structural Description

Component Declarations

Component Instantiation

Configuration

Repetition Grammar




Inline Configuration

The association between component types and ENTITY→ARCHITECTURE pairs can be made inline with a *use* clause.

for all: component-name
use entity entity-name(architecture-name);

Instead of saying **for all**, we can specify a list of selected instances of this component type to which this binding will apply.

instance-name-list: component-name
use entity entity-name(architecture-name);



So recall we had said for all here. If we do not give a list and use others, then that is equivalent to for all. On the other hand, we could even say U1, U2 and use some entity for that and then say others that means all other instances of this kind of component will use the entity name, architecture name given in the other statement.

(Refer Slide Time: 17:53)


Structural Description

Component Declarations

Component Instantiation

Configuration


Repetition Grammar



Hierarchical Configuration

When we associate a *component type* with a previously defined ENTITY→ ARCHITECTURE pair, the chosen architecture could itself contain other components - and these components in turn would be associated with other ENTITY→ ARCHITECTURE pairs.

This hierarchical association can be described by a standalone design unit called a **configuration**.



Now this configuration could be hierarchical. Remember whenever using this, we are using it in an architecture of some higher level component that means we have a higher level component, which consists of a more detailed description using lower level components in the hierarchy. So therefore these configurations and such statements etc are going inside in architecture, which happens to be structural in nature.

So in that architecture we could use an inline configuration or indeed use an external configuration by specifying its name. Now this architecture will then specify a component type remember this component type will then be mapped using the configuration to entity architecture, but that architecture itself could be structural. Therefore, we need to bind the components of that lower level architecture also by a configuration.

And therefore the configuration itself can be hierarchical. For example, you could say in the current architecture, U1 is the component. This U1 is then is some component type, which is mapped to entity e1 and architecture a1, but a1 uses some other components and they must then be bound to entity e13 and architecture 13. So therefore this binding can percolate down the hierarchies and such configurations are called hierarchical configurations.

(Refer Slide Time: 19:55)

The slide is titled "Hierarchical Configuration" and features a navigation menu on the top right with the following items: Structural Description, Component Declarations, Component Instantiation, Configuration, and Repetition Grammar. The main text on the slide reads: "When we associate a *component type* with a previously defined ENTITY → ARCHITECTURE pair, the chosen architecture could itself contain other components - and these components in turn would be associated with other ENTITY → ARCHITECTURE pairs. This hierarchical association can be described by a standalone design unit called a **configuration**." The slide also includes a small circular logo in the bottom left corner and a video inset in the top right corner showing a man speaking.

So this hierarchical association within an architecture you have a component that component is marked to an entity architecture and that architecture has other components and so on. So this hierarchical listing of components can be done once in for all in a single configuration.

(Refer Slide Time: 20:17)

Structural Description

Component Declarations
Component Instantiation
Configuration
Repetition Grammar

Hierarchical Configuration

VHDL contains fairly complex configuration statements. A simplified construct is introduced here:

```

configuration config-name of entity-name is
  for architecture-name
    for component-instance-namelist: component-type-name
      use entity entity-name(architecture-name);
    end for
  end for
end configuration config-name;

```

NPTEL

So here is an example, hierarchical configuration can be fairly complex and VHDL contains fairly complex configuration statements. We now introduce a very simplified construct here, which is an inline configuration. So for example you say configuration, this is the name of that configuration this is actually a standalone configuration in that case. So configuration and this is the name of this particular configuration of this is the entity name is for such and such architecture of this entity.


So remember the configuration is a standalone unit, it has its own name and then it says that I am describing the configuration of this entity whose name is this and then for such and such architecture of this entity and within that architecture for this component instance name list use this component type name, which must use then the entity such and such entity lower level hierarchical entity with such and such architecture.

And notice that you have two for's here and both for's must be ended. This can then be followed for some other architecture of the same entity. Indeed, inside this for architecture you could have for these component instance lists map thusly and for other lists map in a different way. So this whole construct founds a hierarchical configuration by itself.

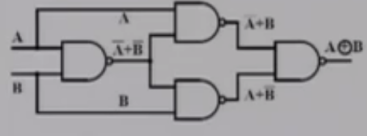
(Refer Slide Time: 22:04)

Structural Description


Component Declarations
 Component Instantiation
Configuration
 Repetition Grammar



Structural description: Example



- Let us choose the xor gate shown on the left as an example for structural description.
- It uses four instances of a single type of component: two input NAND.
- We shall describe the NAND gate first.




Navigation icons

Let us illustrate it by an example. So this is in fact an XOR gate and we are constructing this XOR gate using NANDs. The inputs of this whole thing is an entity and this interconnection is in fact the structural architecture for this entity. This structural architecture uses four instances of the same component type, which happens to be a NAND. So you have a component type called NAND and there are four separate instances of this component type NAND.

(Refer Slide Time: 22:53)


Structural Description

Component Declarations
 Component Instantiation
Configuration
 Repetition Grammar



The work library

- In VHDL, as we describe entities and architectures, these are compiled into a special library called **WORK**.
- This library is always included and does not have to be declared.
- In some sense, the WORK library represent the current state of development of the project for designing something.



Navigation icons

So therefore you need to describe the NAND gate first. So in VHDL you describe entities and architectures and these are when compiled into a special library called work. The work library does not have to be specifically declared. So if you invoke some components, which you have just now described, then you do not have to invoke a library. On the other hand if you

invoke components, which were described elsewhere maybe by somebody else then you have to declare a library and then invoke the use clause to use those things from a specific library.

So in some sense, the work library represents the current state of the development of the project for designing something.

(Refer Slide Time: 23:40)

The slide is titled "Definition of NAND" and is part of a presentation. It features a navigation menu on the right with items: "Structural Description", "Component Declarations", "Component Instantiation", "Configuration", and "Repetition Grammar". A small video inset in the top right corner shows a man speaking. The main content area contains the following VHDL code:

```
Entity nand2 is  
  port (in1, in2: in bit; p: out bit);  
end entity nand2;
```

Below the code, it says: "We do not use any generic for this simple example."

On the right side, the architecture is defined:

```
Architecture trivial of nand2 is  
  p <= not (in1 and in2);  
end Architecture trivial;
```

Below the architecture, it says: "'not' and 'and' are inbuilt logical functions. (Actually so is nand – but we are trying to be cute!)"

At the bottom, it says: "Now that we have this entity-architecture pair, we can use it to build our xor gate."

The slide also features the NPTEL logo in the bottom left corner.

So let us now build upwards from NAND here. We say entity NAND to is. Notice we are declaring now end entity. This is the elemental level entity of a simple to input NAND. This is our port list. We are not using any generics here. So the port in1 and in2 are inputs and are of type bit whereas p is of output direction and type bit and end entity NAND2. That is all there is to the entity NAND2.


Then we describe an architecture we happen to call this architecture trivial then architecture trivial of NAND2 is and then we say assign to p the value of not in1 and in2. So this describes how p is the NAND of in1 and in2. Not and and are built in logic functions. Now that we have this entity architecture pair, we can use these to build our XOR gate. So now our work library contains this description of the entity NAND and this architecture of entity NAND2.

Okay entity is called NAND2. The architecture is called trivial and it is the architecture of NAND2.

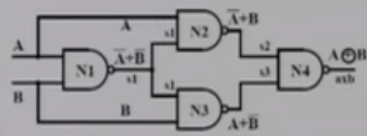
(Refer Slide Time: 25:16)

Structural Description

Component Declarations
 Component Instantiation
 Configuration
 Repetition Grammar



XOR Gate example




```

USE WORK.ALL
Entity xor is
port(a,b: in bit; axb: out bit);
End Entity xor;

Architecture simple of xor is
component NAND2in is port(a,b:
in bit; axb: out bit);
For all NAND2in: use Entity
NAND2(Trivial);
signal s1,s2,s3: bit;

```



Navigation icons


Now we want to build this XOR gate that we had specified. We say `USE WORK.ALL` that means everything described in work is to be recognized as something to be used and now we declare the entity XOR so entity declare is and now we declare the port of XOR now. So port a, b are inputs, so a, b are inputs and of type bit and port axb this is the name, axb is the output and is also of type bit.

And then finally end entity XOR. With this we have declared the entity called XOR. Now we need an architecture for this and we happen to call this architecture, we happen to name it as simple. Architecture simple of XOR is, now remember we have to declare a component. So component NAND2in is and if you recall components are declared the same way as entities are.

(Refer Slide Time: 26:48)

Structural Description

Component Declarations
 Component Instantiation
 Configuration
 Repetition Grammar



Definition of NAND


Entity nand2 is
 port (in1, in2: in bit; p: out bit);
end entity nand2;

We do not use any generic for this simple example.

Architecture trivial of nand2 is
 p <= not (in1 and in2);
end Architecture trivial;

'not' and 'and' are inbuilt logical functions.
 (Actually so is nand – but we are trying to be cute!)

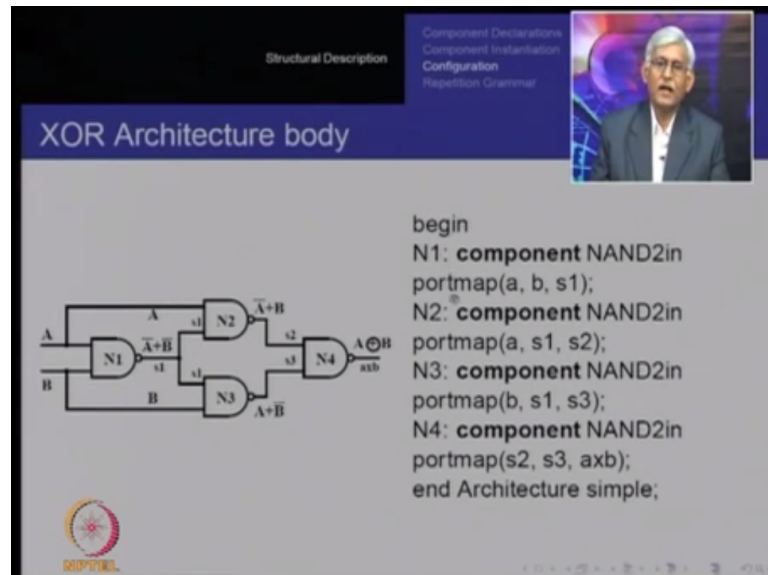
Now that we have this entity-architecture pair, we can use it to build our xor gate.



Navigation icons

So we declare that component NAND2in is and this is the port list, port a, b in bit axb out bit. For all NAND2 in use entity NAND2 with architecture trivial. Recall that NAND2 with architecture trivial was declared earlier in the work library.

(Refer Slide Time: 26:56)



Now the architecture begins and the architecture is simply a list of instantiations because it is structural with all the signals that we should have declared earlier. So signals s1, s2, s3, which are the internal signals must be declared earlier and that is what we have done here signal s1, s2, s3 bit. Now we begin here and this is a, b and then this is instance n1. Notice all instances they are all NAND gates, but they are separate instances.

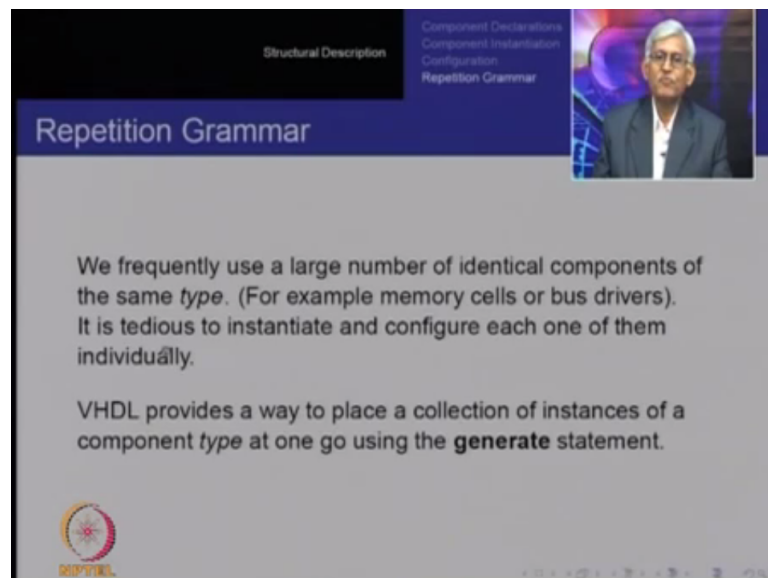
So n1 is an instance and we tell the language that we are instantiating a component we are not directly instantiating an entity architecture pair. So we use the keyword component and NAND2in is the name of the component. Recall that this component was declared here. So we say that instance n1 uses a type called NAND2in and it is to be port map to a, b and s1. That is a say a and b are port signals and the output is to be mapped to an internal signal called s1.

Similarly, we can instantiate n2; n2 is this instance of NAND. So n2 will use the component, which we have declared earlier in a work library called NAND2in. And we shall port map it to a, which is the port signal and to the signal s2, which was the output of NAND N1 here. So a and s1 are then its inputs and then s2 is a new signal and that is its output. Similarly, we instantiate N3; N3 again keyword component, name of the component NAND2in.

And then we port map its ports to b, which is the port signal of XOR, s1 which is the internal signal, which is the output of N1 and then this new signal s3, which is its output. And finally we instantiate N4; N4 is also of the component type NAND2in and we port map this instance to inputs being s2 and s3, here s2 s3 these are the outputs of N2 and N3 as you can see here s2 and s3 and its output goes directly to the port axb of the XOR.

So with this now we have described the entity architecture for an XOR. We can now map a component to the entity architecture pair naming the entity as XOR, here entity XOR with the architecture simple. So now XOR becomes a component and from now onwards we can use XOR as a component type just like we use NAND2 as a component.

(Refer Slide Time: 30:26)



Let us have a look at the repetition grammar. We frequently use a large number of identical components of the same type. We had look at this example before. For example, we could be using a large number of memory cells or bus driver or what have you. Now it is tedious really to instantiate and configure each one of them individually. So recall all this work that we did for N1, N2, N3, N4 would have to be done for let say 1024 components of a 1k memory.

And we can have much larger repetitive components in VLSI. VHDL provides a way to place a collection of instances of a component type at one go using a statement called generate. Notice that generate is interpreted and handled before the simulation actually begins. So this is in that sense a macro of the language so to speak. That means the hierarchy is expanded before the detailed simulation occurs and generate is interpreted as essentially something, which relieves you of the repetitive description process.

From that point onwards the description will use this expanded form, which is internally set up.

(Refer Slide Time: 32:09)

The slide is titled "GENERATE Statement" and is part of a presentation on VHDL. It explains that the generate statement contains a for loop that takes effect during the circuit elaboration step, used to repeat instantiation constructs. An example code snippet is provided, showing a group named 'for index in 0 to width-1 generate' with a 'begin' block containing a 'some-name: component outbuf' instantiation with a 'portmap (...);' and an 'end generate groupname;' statement. A note at the bottom states that the defined index in the 'for' construct has local scope and can be used to pick specific signals from an array in portmap statements.

Structural Description

Component Declarations
Component Instantiation
Configuration
Repetition Grammar

GENERATE Statement

The generate statement contains a for loop which takes effect during the **circuit elaboration** step. This can be used to repeat instantiation constructs. We illustrate this statement with an example:

```
groupname: for index in 0 to width-1 generate
begin
    some-name: component outbuf
        portmap (...);
end generate groupname;
```

The defined index in the "for" construct has local scope and can be used to pick specific signals from an array in portmap statements.

So let us look at the generate statement. The generate statement contains a for loop, which takes effect during the circuit elaboration step. This is what I had said that before the simulation begins there is an elaboration step and during elaborating the circuit we make use of the generate statement. After elaboration has been done, it is a rified, generate statement is not there in your description because the effect of this has already been expanded out in a more detailed circuit.

This statement can be used to repeat instantiation constructs. Indeed, in theory a generate statement can be used to repeat any concurrent statement, but in actual use it is much more common to use it to repeat instantiation. Let us illustrate this with an example. Here you have a name, which is the name of the entire group not a single instantiation or any. This is the name of the entire group, which is then being instantiated.

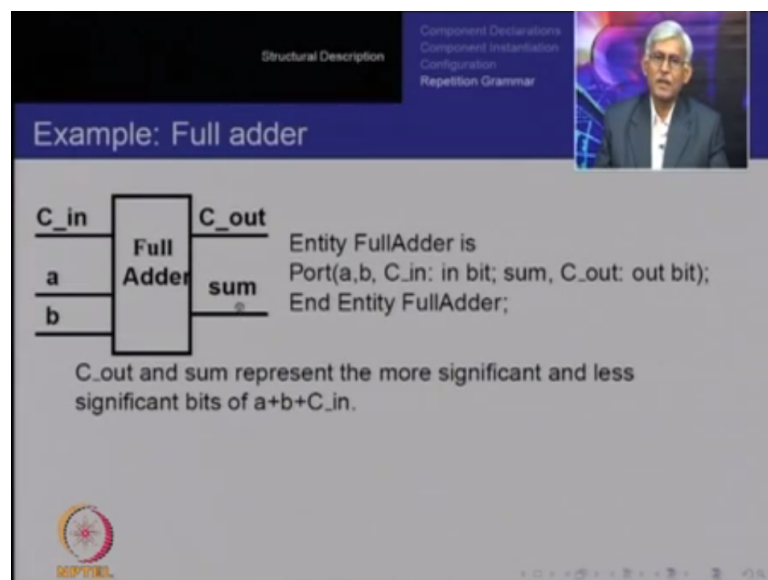
Since this is the repetitive statement, it has syntactic constructs, which are very similar to other repetitive statements and programming language and we use a for here. So you say for any index here, which could be an integer. For index in the range 0 to width - 1 generate. So this is the generate construct. It simply says that what follows here and it happens to be between this begin and end pair has to be repeated for the value of an identifier called index, which will span from 0 to width - 1.

Whatever follows here is to be repeated that many times. Then you have the begin and this is the instantiation, some name followed by the keyword component, this will appear verbatim as component. This is the name of the component say outbuf and then port map. So suppose we are placing many buffers on a bus the bus is for example 32 bits wide. So then we are saying for index in 0 to 31 generate begin. This is the name of the group saying buffer group.

Component which has to be placed repeatedly, the type of that component is outbuf and then you portmap in terms of this index. So each instantiation will use different signals and the names of those signals will be derived from this index. Now the defined index in the for construct has local scope that means it has scope only inside this. You may use the same name outside the generate statement and that will not be this index.

This index has local scope only inside here and can be used to pick specific signals from an array in the portmap statements. Remember this portmap is to be repeated, but the port mapping is different for different instances. So we get around this problem by using an array of interconnect signals and the index in that array comes from this.

(Refer Slide Time: 35:51)



Here is an example, you have a FullAdder. This is an example of structural description. You have a FullAdder, it has three inputs a, b and carry in and it has two outputs sum and carry out. We must as always begin with an entity declaration. Entity FullAdder is, this is followed by a port less thing, port a, b and c in as inputs of type bit and sum and c out as outputs and also of type bit and entity FullAdder, that is all there is to the entity declaration.

Remember an entity is a look at this hardware from the outside and from the outside what we see are a, b and c in as inputs and sum and c out as outputs. A force as is conventional sum and c out represent sum is the less significant and c out is the more significant bit of the two-bit result of adding a, b and c in.

(Refer Slide Time: 37:10)

Structural Description

Component Declarations
Component Instantiation
Configuration
Repetition Grammar

Example: Full adder

C_in

a

b

Full Adder

C_out

sum

Entity FullAdder is
Port(a,b, C_in: in bit; sum, C_out: out bit);
End Entity FullAdder;

C_out and sum represent the more significant and less significant bits of $a+b+C_{in}$.

Suppose this is too difficult for the likes of us to figure out 😊

Now just assume that this is too difficult for us to figure out. In reality you will be using much more complicated circuits, but we just use this example as if designing a FullAdder is too difficult for us and that it must be hierarchically broken down.

(Refer Slide Time: 37:30)

Structural Description

Component Declarations
Component Instantiation
Configuration
Repetition Grammar

Example: Full adder

C_in

a

b

Full Adder

C_out

sum

Entity FullAdder is
Port(a,b, C_in: in bit; sum, C_out: out bit);
End Entity FullAdder;

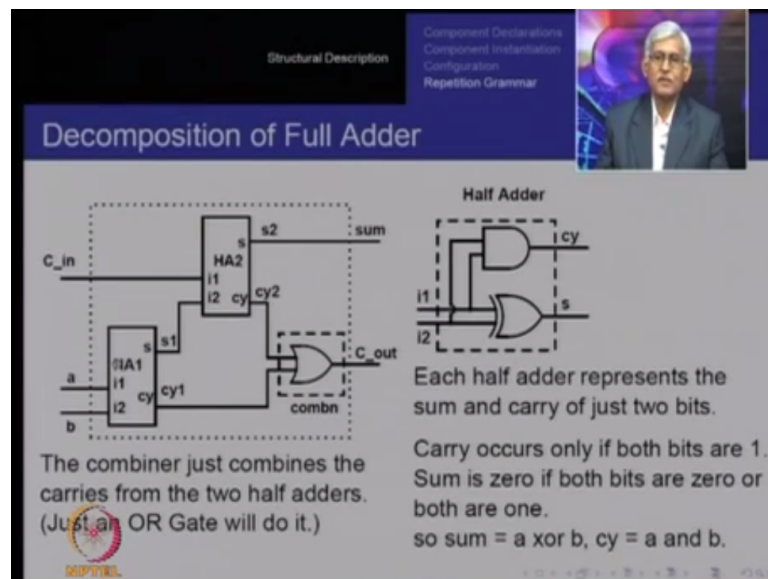
C_out and sum represent the more significant and less significant bits of $a+b+C_{in}$.

Suppose this is too difficult for the likes of us to figure out 😊

We would like to decompose the circuit into blocks which handle two bits at a time.

So we would like to decompose this circuit into blocks, which handle only two bits at a time.

(Refer Slide Time: 37:35)



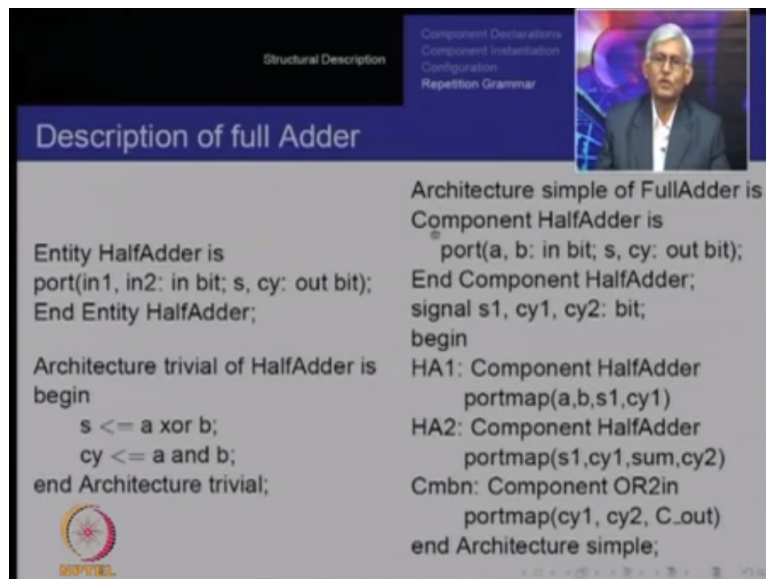
So let say that we cannot design a FullAdder from scratch. We would like to describe it in terms of a simpler circuit called a half adder, which adds only two bits at a time. So then we can decompose the FullAdder into two half adders. The first half adder adds a and b and produces two outputs sum1 and carry1 . This sum1 is then combined using a half adder with the carry in.

So we add carry in to this sum and the output of this again produces two bits, $s2$ is the sum of this half adder and that appears as the sum of the FullAdder and the two carries need to be combined to produce a single carry out and it turns out functionally that the OR of these two carries is the carry out that means if a and b produce a carry in that case the carry out should be 1 or even if they do not if the sum of these produces a carry from c in then also the carry out should be 1.

So therefore functionally the carry out of the FullAdder is the OR of the carries of the two half adders. Each half adder, so now we have described the FullAdder in terms of half adder. We must go hierarchically down and declare how the half adder is going to work. The half adder is easy to design. Each half adder represents a sum and carry of just two bits. Carry occurs only if both bits are 1 and therefore it is the and gate.

And the sum is 0 if both bits are 0 or 1 and 1 if the bits are dissimilar so this is an XOR gate. So now we have described the half adder in terms of known gates. Recall that just some time ago, we had described the XOR gate as a combination of NAND gates. So now we are down to NAND gates for describing everything else.

(Refer Slide Time: 40:00)



The slide is titled "Description of full Adder" and is part of a presentation on VHDL. It contains two VHDL code snippets. The first snippet defines an entity "HalfAdder" with two input ports (in1, in2) and two output ports (s, cy). The second snippet defines an architecture "trivial" for the HalfAdder, showing the logic for sum (s) and carry (cy). The third snippet defines an architecture "simple" for the FullAdder, which uses two HalfAdder components (HA1, HA2) and an OR2in component (Cmbn) to calculate the sum and carry. A small video inset in the top right corner shows a man speaking. The slide also has a navigation bar at the top with options like "Structural Description", "Component Declarations", "Component Instantiation", "Configuration", and "Repetition Grammar".

```
Entity HalfAdder is
port(in1, in2: in bit; s, cy: out bit);
End Entity HalfAdder;

Architecture trivial of HalfAdder is
begin
    s <= a xor b;
    cy <= a and b;
end Architecture trivial;


Architecture simple of FullAdder is
Component HalfAdder is
    port(a, b: in bit; s, cy: out bit);
End Component HalfAdder;
signal s1, cy1, cy2: bit;
begin
    HA1: Component HalfAdder
        portmap(a,b,s1,cy1)
    HA2: Component HalfAdder
        portmap(s1,cy1,sum,cy2)
    Cmbn: Component OR2in
        portmap(cy1, cy2, C_out)
end Architecture simple;
```

Now this is how in VHDL will describe the FullAdder. Entity half adder is; this is what we need for the FullAdder. Port in1, in2 as inputs of type bit and port s and carry these are outputs of type bit and this ends the entity half adder. The architecture trivial of half adder is and you assign to s a xor b and assign to carry a and b and this is the trivial architecture. Now architecture simple of FullAdder is now we declare a component type half adder.

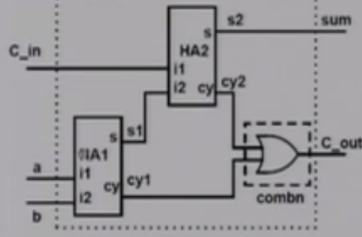
Notice we have an entity half adder, but we are going through the full route not instantiating the entity directly. Declaring a component type, which is half adder so we say component half adder is and this component has port a and b as inputs, which are bits and s and carry s outputs, which are also bits and this ends the component declaration and now we declare signals s1, cy1 and cy2.

(Refer Slide Time: 41:13)

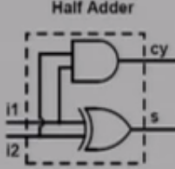
Structural Description
 Component Declarations
 Component Instantiation
 Configuration
 Repetition Grammar



Decomposition of Full Adder



The combiner just combines the carries from the two half adders. (Just an OR Gate will do it.)




Each half adder represents the sum and carry of just two bits.

Carry occurs only if both bits are 1. Sum is zero if both bits are zero or both are one.
 so $sum = a \oplus b$, $cy = a \text{ and } b$.

Notice these are the internal signals of the FullAdder, s1, cy1 and cy2 are the internal signals remember s2 is directly mapped to a port and therefore need not be declared as an internal signal. Similarly a and b are directly available as ports. So it is a carry in, so these do not have to be declared as internal signal. So at the result it is only s1, s2, carry1 and carry2, which are internal to this architecture and that is what we are doing.

(Refer Slide Time: 41:47)

Structural Description
 Component Declarations
 Component Instantiation
 Configuration
 Repetition Grammar



Description of full Adder

Entity HalfAdder is
 port(in1, in2: in bit; s, cy: out bit);
 End Entity HalfAdder;

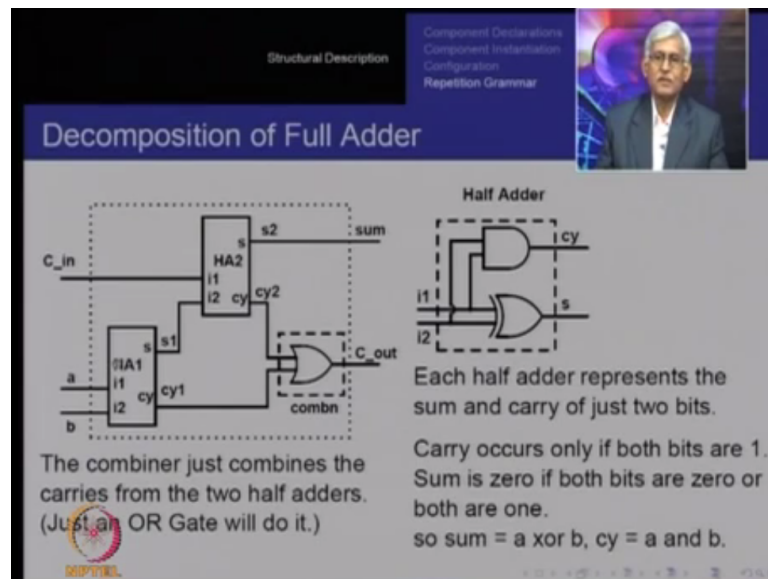
Architecture trivial of HalfAdder is
 begin
 $s \leq a \text{ xor } b$;
 $cy \leq a \text{ and } b$;
 end Architecture trivial;

Architecture simple of FullAdder is
 Component HalfAdder is
 port(a, b: in bit; s, cy: out bit);
 End Component HalfAdder;
 signal s1, cy1, cy2: bit;
 begin
 HA1: Component HalfAdder
 portmap(a,b,s1,cy1)
 HA2: Component HalfAdder
 portmap(s1,cy1,sum,cy2)
 Cmbn: Component OR2in
 portmap(cy1, cy2, C_out)
 end Architecture simple;

We are declaring s1, carry1, carry2 as the internal signals; s2 will be directly mapped to the port sum and then we place these components. There are two instances of the half adder so HA1 component half adder, remember we have declared the component half adder here and portmap these ports that had been declared here to a, b, s1 and carry1 that is this half adder, this half adder inputs are mapped to a and b and the outputs are mapped to s1 and carry1.

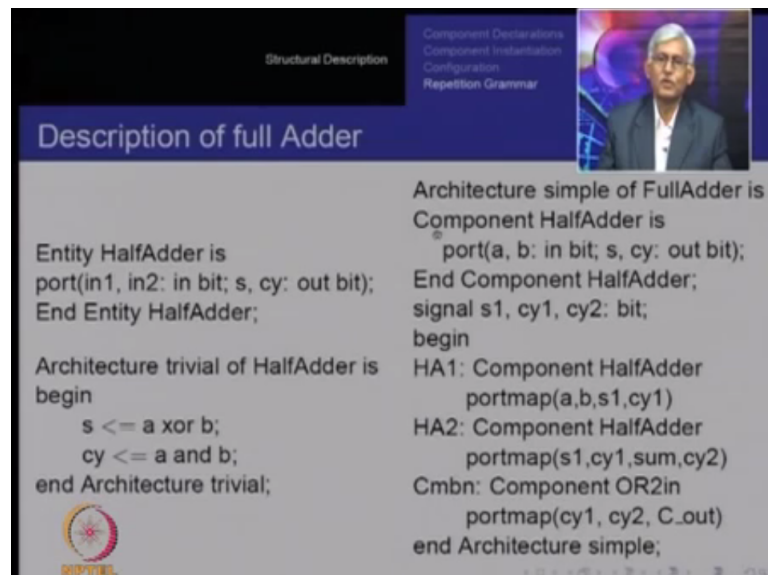
Then HA2 is instantiated this is also the component half adder, but this is port mapped to s1 and carry1 and the outputs are port mapped to sum and carry2.

(Refer Slide Time: 42:47)



Recall this is HA1, the inputs are mapped to a and b. the outputs are s and s1 and cy1 and then s1 and carry in are the inputs to half adder 2 and sum and carry of this are mapped to sum directly here, which is the port signal and cy2 is then mapped to the input of this R gate.

(Refer Slide Time: 43:15)




So that is what we are saying HA1 is component half adder, portmap to a, b as input, outputs are s1 and carry1. HA2 is also the same component half adder and it is port mapped, two inputs are s1 and carry1 now and outputs are the sum of the entity and carry2. Now we need to combine the two carry into a single carry and that instance is called combination. We have a previously declared component OR2.

So we say component OR2 input and portmap the OR gate to carry1 and carry2 and the output appears as C out of the FullAdder and this ends the simple architecture.

(Refer Slide Time: 44:09)

Structural Description
 Component Declarations
 Component Instantiation
 Configuration
 Repetition Grammar

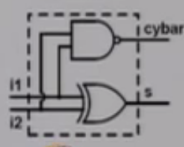


The half adder

Carry from the half adder is an AND gate, and the combiner is an OR.

But Gates without inversion are slow. So we bring out $\overline{\text{carry}}$ rather than carry, using a NAND gate.

Half Adder



```

Entity HalfAdder is
  port(in1, in2: in bit; s, cybar: out bit);
End Entity HalfAdder;
Architecture better of HalfAdder is
begin
  s <= a xor b;
  cybar <= a nand b;
end Architecture better;

```

The combiner should now be an OR of negative true signals.
This is just a NAND.

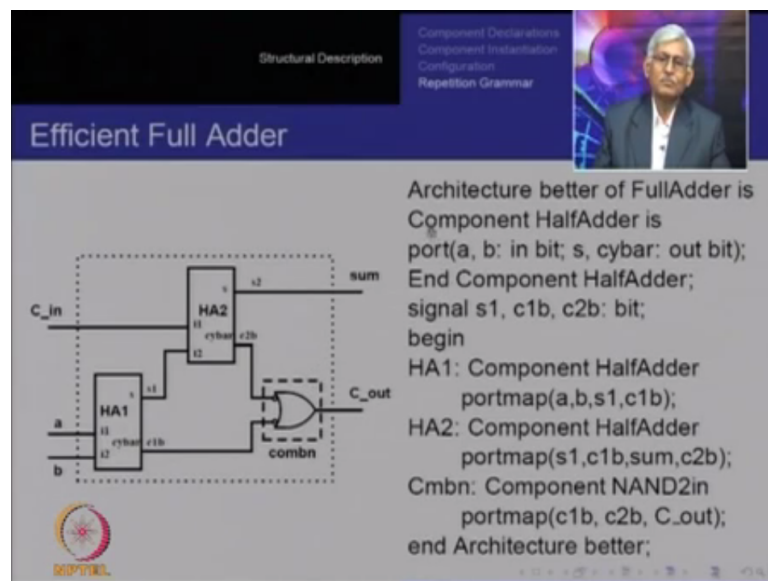
Now let us look at the half adder, the carry from the half adder is an AND gate and the combiner eventually is an OR, but gates without inversions are slow, AND gates and OR gates are slow. So let us improve this design and bring out carry bar rather than carry using a NAND gate. A NAND gate is a natural in CMOS implementation. So then we redefine the half adder to produce sum and carry bar.

Remember the sum is an XOR, which can be constructed from NANDs as we had seen earlier. So rather than carry using a NAND gate we declare a carry bar. So now we declare a new half adder whose output is carry bar not carry. So entity half adder is port in1, in2 as before inputs bits and sum as before, but now cy bar, which is an output of type bit and that ends the entity half adder.

And the architecture better, this is the new architecture of half adder is and you begin assign to some a xor b and assign to carry bar a NAND b. This ends the architecture better of half adder, this is hopefully faster because it uses only inverting gates. The combiner should now remember the carry is now negated, carry is the bar of the carry, cy bar is output. Therefore, the combiner should be an OR of negative true signals.

Whenever the carry is true, the output of half adder will be false. So therefore the combiner should be the OR of false signals and that is just a NAND.

(Refer Slide Time: 45:56)




So now you have HA1 with a better architecture, HA2 with a better architecture both giving carry bars and the combiner now is the OR of negative inputs, which is in other words a NAND. So now we have the architecture better or FullAdder, which uses component half adder so we declare a component half adder with these ports with signal s1, c1b, c2b and we instantiate HA1 and HA2, which is component half adder portmap and so on.

And then the combination is a NAND2 and this ends the architecture better. Then we have the efficient FullAdder, which uses this better half adder and the combination is in fact NAND2. This illustrates essentially how true instantiate components.

(Refer Slide Time: 46:57)

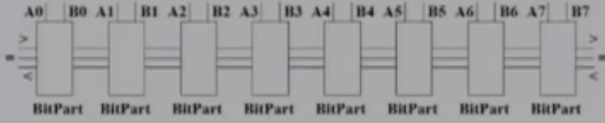
Decomposition of Byte Comparator

Constructing the Byte Comparator
Structural Description of Bit



©

The byte comparator is difficult to design directly.
We can break up the design into bit comparators
with cascading inputs gt_in, eq_in and lt_in;
and cascading outputs gt_out, eq_out and lt_out.

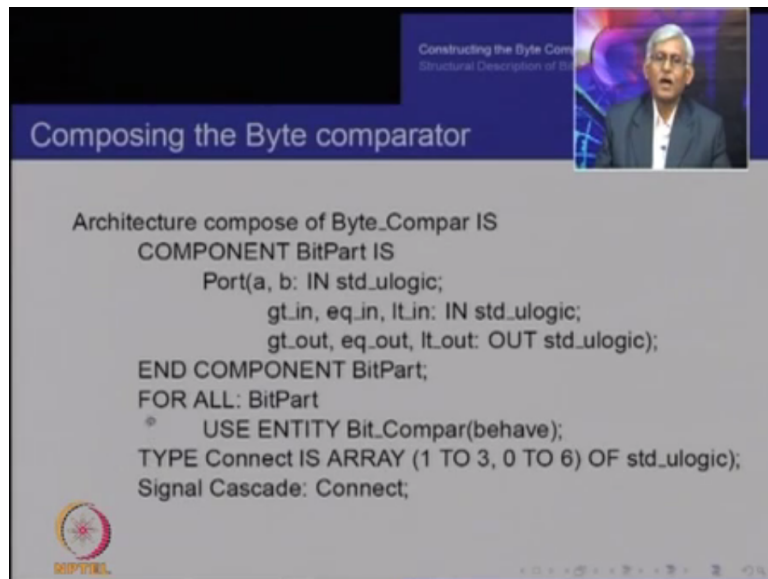


Notice that the most significant bit is compared closest to the output.

We have not instantiated generics and that can be shown by having a byte comparator, which compares two bytes and each internal component is a bit comparator. So essentially we are making a byte comparator to give you an example of a hardware, which uses repetitive hardware components and assume that you have input bit comparators and these bit comparators will then be stacked eight of them.

So that we have to use a generic to describe eight of them and then we have three outputs from each bit comparator. When you compare the two single bits if the result less than equal to or greater than and then we stack these end to end to form a byte comparator. Notice that the most significant bit is compared closest to the output because if the most significant bit produces a greater than or less than result then the output is independent of the rest of the bits and can appear immediately at the output.

(Refer Slide Time: 48:20)



Constructing the Byte Comparator
Structural Description of the Byte Comparator

Composing the Byte comparator

Architecture compose of Byte_Compar IS

```

COMPONENT BitPart IS
    Port(a, b: IN std_ulogic;
          gt_in, eq_in, lt_in: IN std_ulogic;
          gt_out, eq_out, lt_out: OUT std_ulogic);
END COMPONENT BitPart;
FOR ALL: BitPart
    * USE ENTITY Bit_Compar(behave);
TYPE Connect IS ARRAY (1 TO 3, 0 TO 6) OF std_ulogic;
Signal Cascade: Connect;
```

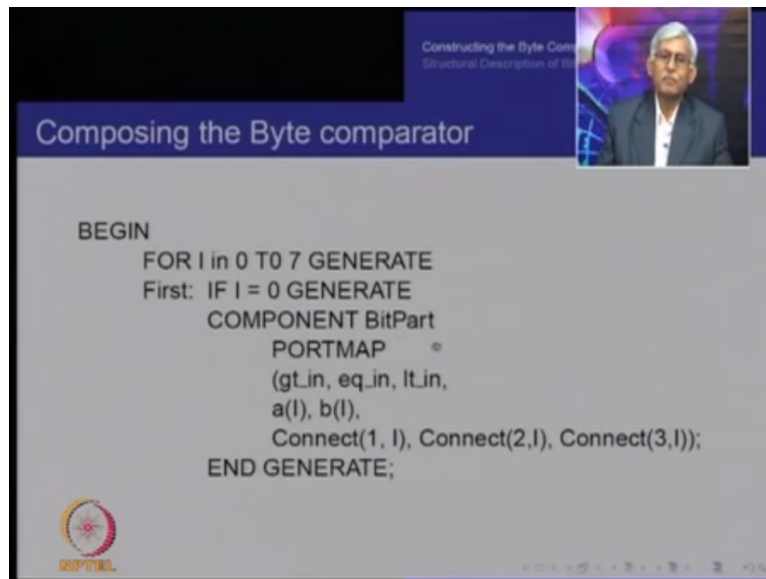
So now you have this architecture compose of byte compar IS and then we declare a component bit part with these ports and these outputs greater than in, equal in and less than in as the inputs because you are concatenating design and greater than out, equal out and less than out as outputs. So each bit comparator has a and b as inputs, but it also has concatenating inputs from the previous comparator and it produces three outputs which is greater than equal or less than as outputs.

So this declares the component and then we say that for all bit part use entity bit compar behavior. So hopefully we have already declared an entity called bit compar, which has an architecture behavior and now we declare a signal called connect, which will connect all these bit comparator one to the other. So essentially these three will occur between each repetitive components.

So we declare connect as a bidimensional array so it is array one to three because of these three signals and 0 to 6 for all these where these group of three is to be repeated of std ulogic, std ulogic is a bit type signal, which we have not yet done, but we could start using it here. This is a type that we have declared. Remember this is a type, so type connect is an two dimensional array and now we say signal cascade is connect.

That means cascade is a special signal which is a two dimensional array of std ulogic, which is like type bit. So it is a two dimensional array of 3 by 7 actually 0 to 6 of std ulogic.

(Refer Slide Time: 50:26)



Constructing the Byte Comparator
Structural Description of BitPart

Composing the Byte comparator

```

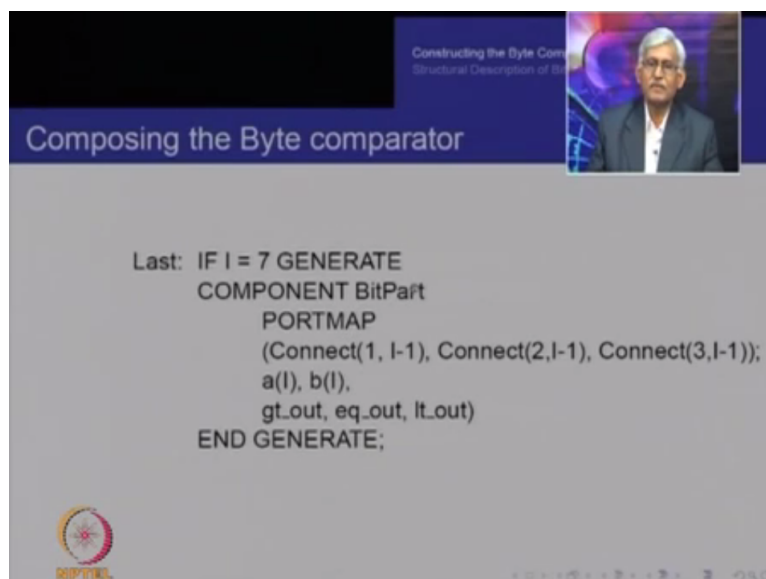
BEGIN
  FOR I in 0 TO 7 GENERATE
    First: IF I = 0 GENERATE
      COMPONENT BitPart
        PORTMAP
          (gt_in, eq_in, lt_in,
           a(I), b(I),
           Connect(1, I), Connect(2,I), Connect(3,I));
      END GENERATE;
  
```

SPTEL

And this is the example that we wanted to show then we will say for I in 0 to 7 this is going for all the bits generate and then we have a special case if I equal to 0 generate component bit part and when we map it in case of I equal to 0 we map it to the port entries, the 0 part maps directly through the entries of the entity and the outputs are aI, bI. These are the other two bit inputs.

And the outputs are Connect 1 I, 2 I and 3 I. I is the bit number, which will go from 0 to 7 end generate. Notice connect is the two dimensional array and here 1, 2 and 3 are being mapped to the corresponding greater than in, equal in and less than in signals and end generate.

(Refer Slide Time: 51:25)



Constructing the Byte Comparator
Structural Description of BitPart

Composing the Byte comparator

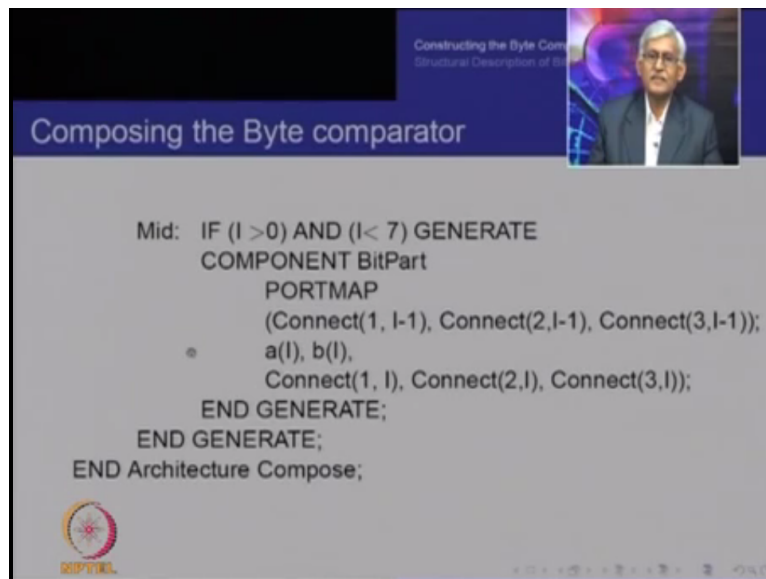
```

Last: IF I = 7 GENERATE
  COMPONENT BitPart
    PORTMAP
      (Connect(1, I-1), Connect(2,I-1), Connect(3,I-1));
      a(I), b(I),
      gt_out, eq_out, lt_out)
  END GENERATE;
  
```

SPTEL

Also the last part is unique if I equal to 7 then generate and it is very similar except that the inputs are now cascaded, but the outputs are the port outputs.

(Refer Slide Time: 51:37)



Constructing the Byte Comparator
Structural Description of BitPart

Composing the Byte comparator

```
Mid: IF (I > 0) AND (I < 7) GENERATE
      COMPONENT BitPart
      PORTMAP
        (Connect(1, I-1), Connect(2, I-1), Connect(3, I-1));
      a(I), b(I),
      Connect(1, I), Connect(2, I), Connect(3, I));
    END GENERATE;
  END GENERATE;
END Architecture Compose;
```

© 2014

And this is the interesting part, this is the middle part and if I is greater than 0 and I less than 7, then no port signals are involved and then we say generate and then we say that the component to be used is the bit part and the port mapping is the array at the input and the array at the output, only aI and bI are the bit specific signals which will come. So by using a generate then we have used all the component parts.

So this gives an illustration of how we can describe hardware structurally using components, using entity architecture pairs directly and with or without repetition statements. We bring this lecture to a close here and then we will wrap up the series on hardware descriptions by describing how behavioral descriptions can be done in VHDL. We stop this lecture here.