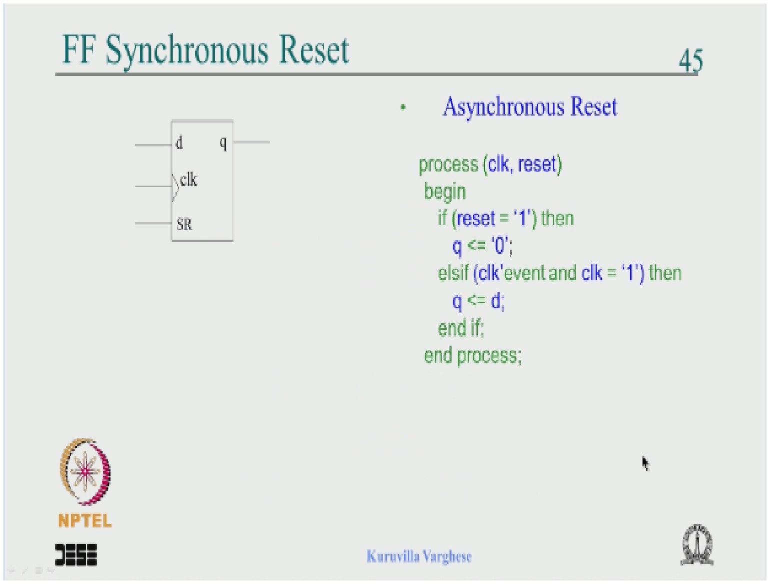
Digital Systems Design with PLDs and FPGAs

Kuruvilla Varghese
Department of Electronic Systems Engineering
Indian Institute of Science-Bangalore

Lecture-18
Libraries and Packages

Welcome to this lecture on VHDL in the course digital system design with PLDs and FPGAs. The last lecture we have seen the VHDL coding of sequential elements along with the combinational circuit. Then we have looked at I mean we started on how to write packages. At least we did a pulmonary kind of work towards it. So, we will quickly run through the slides and get onto today's part.

(Refer Slide Time: 00:58)



FF Synchronous Reset 45

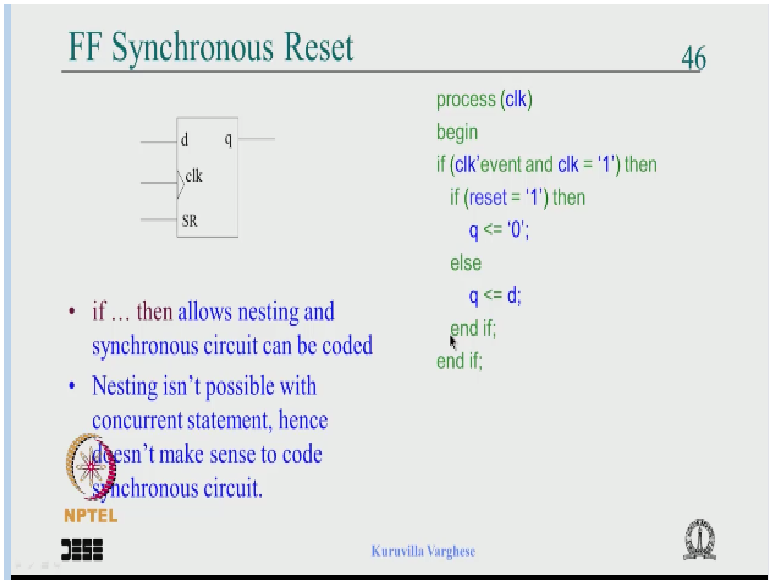
- Asynchronous Reset

```
process (clk, reset)
 begin
   if (reset = '1') then
     q <= '0';
   elsif (clk'event and clk = '1') then
     q <= d;
   end if;
 end process;
```

So, the last class, we have seen about the VHDL coding of synchronous reset in flip-flop. Unlike a synchronous reset the synchronous reset is kind of synchronous with the clock, if it is active. Then on the next clock, the q will be 0. And that is a once the clock comes it has priority over the d. If it is active then irrespective of the d it becomes 0. Then if it is not active, whatever on the next clock edge the d goes to q.

So, naturally in a synchronous reset we have put it before this clock event, clock is equal to 1, so that cannot be done. So, this is synchronous to the clock, so everything happens with respective the clock. So, this reset in the sensitivity less is not required also we write the reset within this clock event, clock is equal to 1. Because it is synchronous, and we give priority to reset, so we write the reset first then the q comes second.

(Refer Slide Time: 02:06)



FF Synchronous Reset

46

d
q
clk
SR

- if … then allows nesting and synchronous circuit can be coded
- Nesting isn't possible with concurrent statement, hence doesn't make sense to code synchronous circuit.

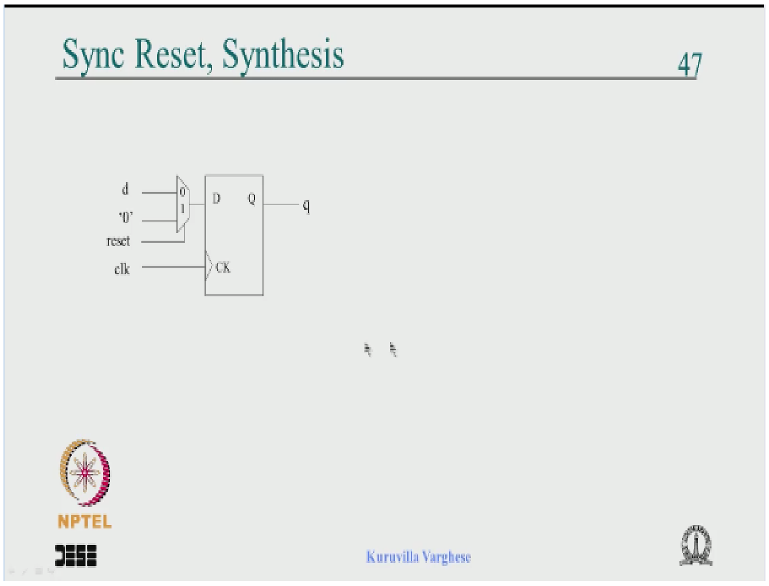
```
process (clk)
begin
if (clk'event and clk = '1') then
  if (reset = '1') then
     q <= '0';
  else
     q <= d;
  end if;
end if;
```

NPTEL
Kuruvilla Varghese

So, that is what is shown here process clock in this we write if reset is 1, q is 0, else q gets d. And this shows a kind of multiplexer structure with reset as a select line. When it is 1, q is getting 0. Otherwise q is getting d. So, the synthesis tool will put that multiplexer. We will see that, and we have also said that in a process, that nestings allows as to kind of behaviourally represent the synchronous reset.

But in a concurrent statement it is difficult. So, it does not make sense, once again may be the synthesis tool will support some kind of syntax which at least a kind of schematically does not sound correct yes. If it is support you can use it. But then I suggest that as for as the flip-flops and registers are concern you go for a process in VHDL.

(Refer Slide Time: 03:11)



Sync Reset, Synthesis

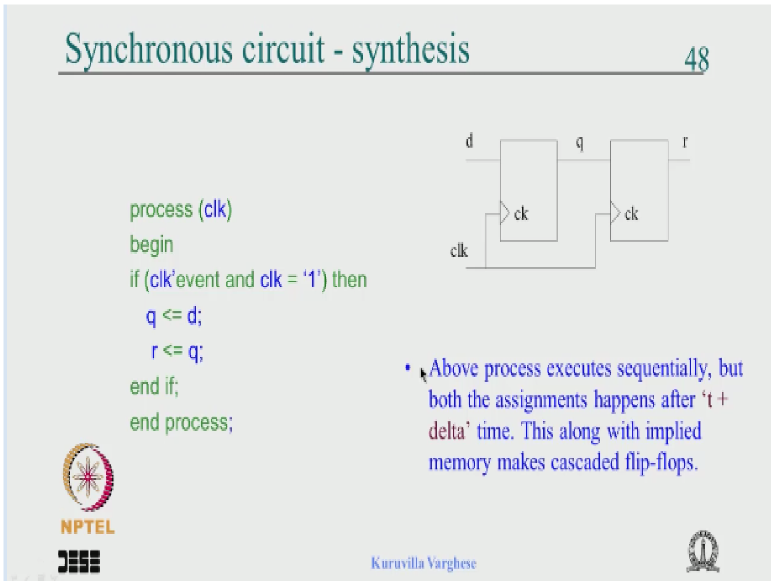
47

NPTEL

Kuruvilla Varghese

And this is what the synthesis tool may caught of that code the clock goes here. Now when reset is a select line of 2 to 1 mugs. When reset is 1, 0 goes here and reset is 0. Then the d goes here. Then you get what you want.

(Refer Slide Time: 03:27)



Synchronous circuit - synthesis

48

```
process (clk)
begin
if (clk'event and clk = '1') then
   q <= d;
   r <= q;
end if;
end process;
```

d
q
r
ck
ck
clk

- Above process executes sequentially, but both the assignments happens after 't + delta' time. This along with implied memory makes cascaded flip-flops.

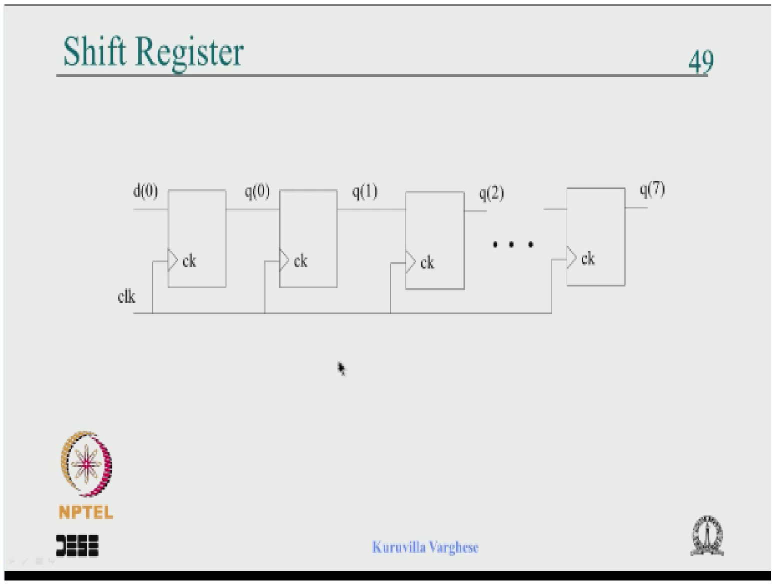
NPTEL

Kuruvilla Varghese

We have looked at this q gets d and r gets q, definitely q and r will get 2 flip-flops that is evident. Because you have an event on the clock, what is the current value of d goes to q at +delta time, current value of q goes to r +delta time. So, you get something like this which if use a variable you will not get it you know that. The variable you could write this with a variable and we will see that.

But if you do an immediate assignment, you will not get this, if you say q colon equal to d or colon equal to q. Then you will just get one r, 1 flip-flop not 2 flip-flops okay.

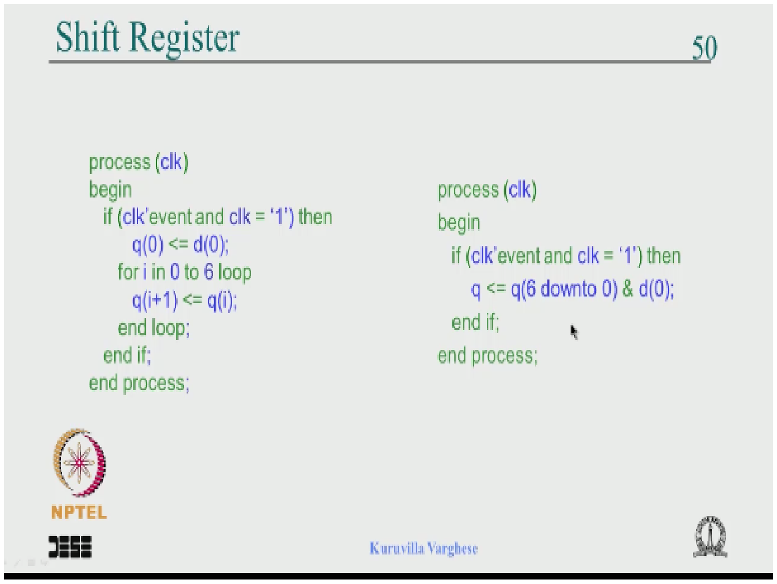
(Refer Slide Time: 04:20)



Shift Register

And we looked at a shift register, how to model a shift register. So, here it is an 8 bit shift register we have made it little awkward d0 goes to q0, q0 to q1 and so on. And you could write a loop within that clock event, clock is equal to 1. That is one way or you can view it as a kind of vector that is q7 down to 0 or q gets q6 down to 0, and concatenated with d0.

(Refer Slide Time: 04:52)



Shift Register

50

```
process (clk)
begin
  if (clk'event and clk = '1') then
      q(0) <= d(0);
    for i in 0 to 6 loop
      q(i+1) <= q(i);
    end loop;
  end if;
end process;
```

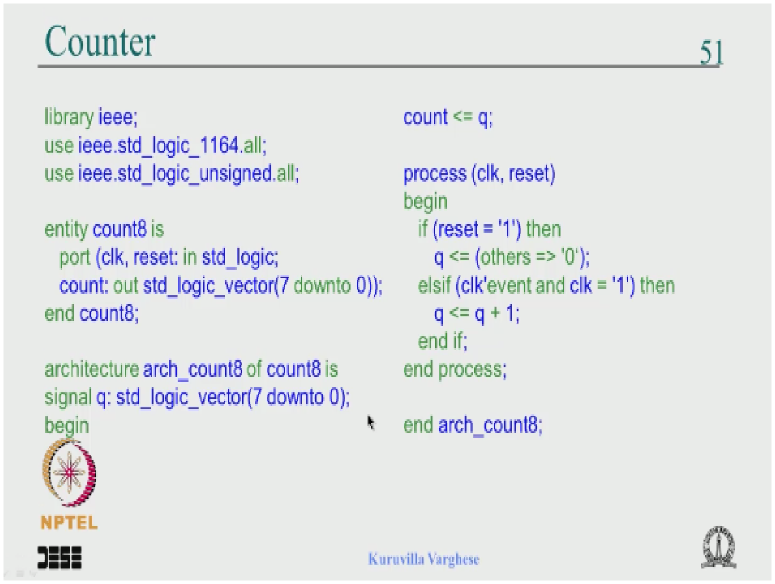
```
process (clk)
begin
  if (clk'event and clk = '1') then
     q <= q(6 downto 0) & d(0);
  end if;
end process;
```

NPTEL
Kuruvilla Varghese

So, both like this is the loop upon the clock q0 get d0, and for i in 0 to 6 loop qi + 1 is qi that means for the zeroth index, you get q1 gets q0, for the sixth syntax you get q7 get q0 end. But you could write like this q which is nothing, but 7 down to 0 is q6 down to 0, and d0 in one sort. You get it and that is also like as, I said you could show as a single flip flop with thick lines and the output is q, and the input is q6 down to 0 and d of 0.

So, that is the 2 ways of looking at it I would prefer this any day then this also kind of at least for a people were not through with the syntax. It will looks as if it is some loop is written something which is consecutively executing which is true for simulator not for synthesis.

(Refer Slide Time: 05:54)



Counter

51

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity count8 is
  port (clk, reset: in std_logic;
  count: out std_logic_vector(7 downto 0));
end count8;

architecture arch_count8 of count8 is
signal q: std_logic_vector(7 downto 0);
begin

count <= q;

process (clk, reset)
begin
  if (reset = '1') then
    q <= (others => '0');
  elsif (clk'event and clk = '1') then
    q <= q + 1;
  end if;
end process;

end arch_count8;
```

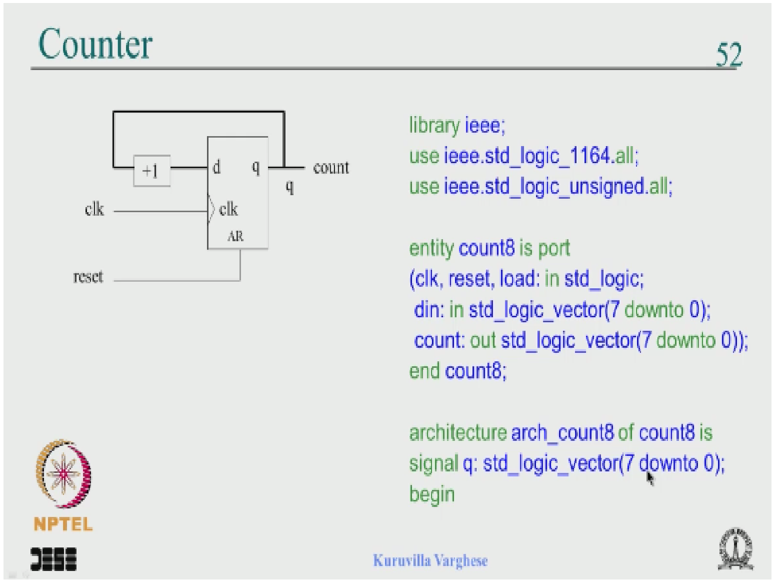
Kuruvilla Varghese

And we have seen counter and this is standard logic unsigned is use so, that we are using a + here. And clock reset is a input 8 bit count, output is the count is an output and since we cannot say count gets count + 1. We define a signal of the same bit q which is of 8 bit vector and in the code, we will in the statement region, we get we say count gets q. And write the code in terms of q, if reset is 1 q gets others 0.

Otherwise upon the clock q gets q + 1 so, and as I said this + the one side is standard logic vector other side is integer. So, this operator which is normally use for integer is use is over loaded in this particular package. That is why this we use this package so, in this code to notice the use of this particular package. And way we use signal to overcome the limitation of the output type of port, because you define something is output.

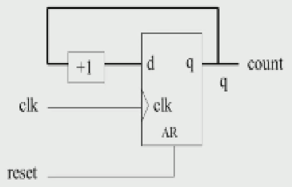
Then it cannot come on the you know the right hand side of an assignment, because then it is treated as an input. But if you declare a signal it does not limitation so, that is kind of we are overcome by this method. And this is something which you should keep in mind.

(Refer Slide Time: 07:29)



Counter

52

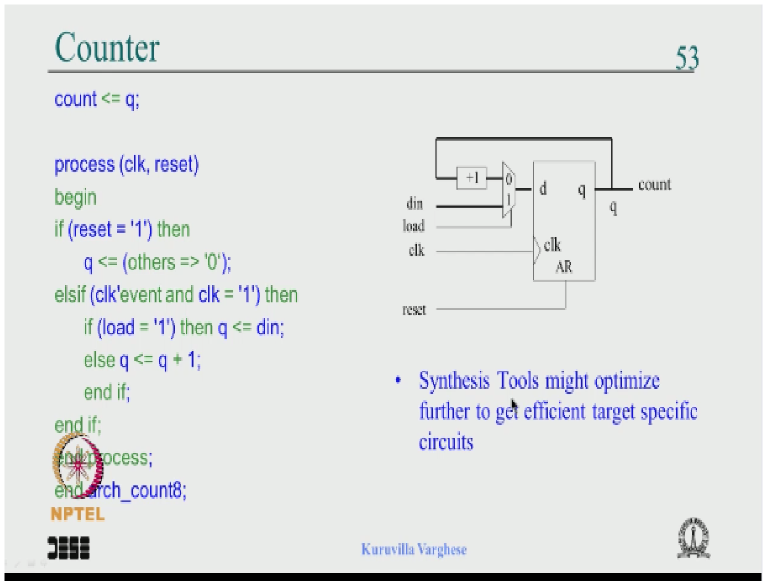


```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity count8 is port
(clk, reset, load: in std_logic;
 din: in std_logic_vector(7 downto 0);
 count: out std_logic_vector(7 downto 0));
end count8;

architecture arch_count8 of count8 is
signal q: std_logic_vector(7 downto 0);
begin
```

Kuruvilla Varghese

And this is how it is synthesise, which say that upon the clock q gets q + 1. So upon the clock q will get q + 1. And that is how it is synthesise, and we have gone 1 level up, we have added an 8 bit input, and a load signal again, we treated as a synchronous load. When the load is high this din is loaded to the output, and load goes low it **it** is starts count from there. It is a synchronous load so, naturally this comes in the clock tic event. And clock is equal to 1 and we give priority to that over the counting and this is a signal declaration standard logic unsigned library.

(Refer Slide Time: 08:16)



Counter

53

```
count <= q;

process (clk, reset)
begin
if (reset = '1') then
    q <= (others => '0');
elsif (clk'event and clk = '1') then
    if (load = '1') then q <= din;
    else q <= q + 1;
    end if;
end if;
end process;
end arch_count8;
```

+1
0
1
d
q
count
q
din
load
clk
clk
AR
reset

- Synthesis Tools might optimize further to get efficient target specific circuits

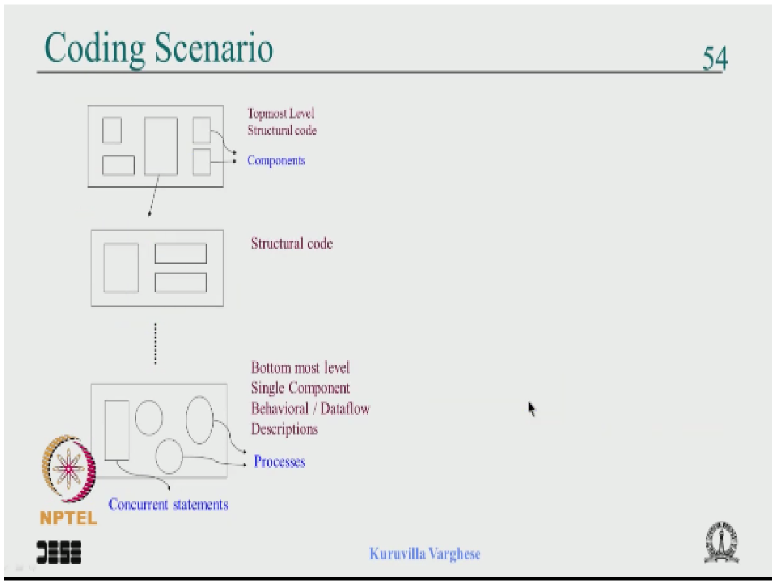
Kuruvilla Varghese

And here the count is assigned with the q process clock reset, if reset is 1, q is initialise is 0 upon the clock. Now this is the game if load is 1, then q gets din else q get q + 1 end if, and this is the end, if for the clock event end process and end architecture. And once again this shows if load is 1 q gets something else, q get q + 1. So this shows a 221 mugs where load is the select line when it is 1 q get the din, when it is 0 gets q + 1 so, that is shown here.

So, to the d because everything is synchronous the moment, you write something underneath then there is synchronous which is connected to d. And this represent a mugs, when load is 1 q gets din, that is this and load is 0, q gets q + 1. You know that is, how it is synthesise and synthesis tool might optimize it further depending on the target like in a FPGA, you may have kind of dedicated results for this +1 and so on.

So, that could be kind of and the this could be part of the logic results within a FPGA. You may not use kind of standard resources within a FPGA that is what I need and when we get to the FPGA and we will understand it better.

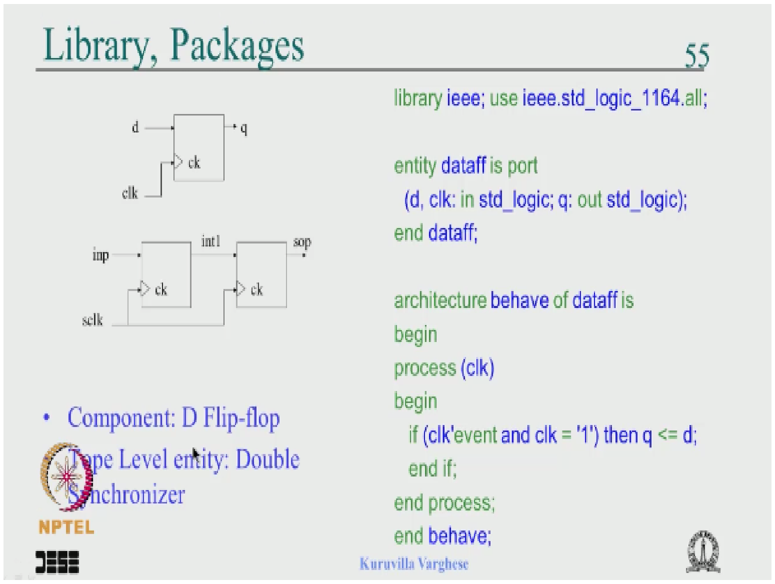
(Refer Slide Time: 09:52)


Coding Scenario

54

And this is a coding scenario top most code will be normally in a complex case will be the structural code made of components and each of the component could be further structural code depending on the complex city. But when it comes to the last level normally level2, level3 this particular component, when it is reduced to a reasonable level will be composed of some concurrent statement processes procedures and things like that. That is how the coding goes.

(Refer Slide Time: 10:26)


Library, Packages

55

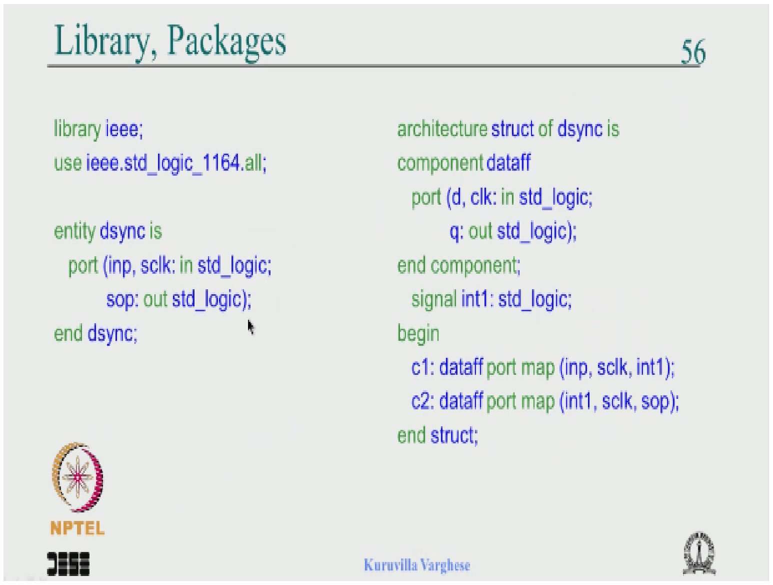
```
library ieee; use ieee.std_logic_1164.all;

entity dataff is port
  (d, clk: in std_logic; q: out std_logic);
end dataff;

architecture behave of dataff is
begin
process (clk)
begin
  if (clk'event and clk = '1') then q <= d;
  end if;
end process;
end behave;
```

- Component: D Flip-flop
- Tope Level entity: Double Synchronizer

And the topic we have started was is how to put a component in a package, and put that in a library okay, that is what we have seen the example. We have taken was a flip flop and the top level entity which use this is a component is a double state synchroniser or a two state shift register, whatever you would like to call. So, it is structure is very simple the clock is called s clock, which is connected together input goes to the d of the first flip flop.

The q is internal signal which is connected to the d of the second flip flop, and the second flip flop output is the sop. So, we have seen how normally, we write it the library the package entity d, and clock are the input. Q is the output, and this is the behavioural code the process clock begin, if clock event clock is equal to 1, q gets d end if end process and so on. Now we are going to look at the code.

(Refer Slide Time: 11:34)



Library, Packages

56

```
library ieee;
use ieee.std_logic_1164.all;

entity dsync is
  port (inp, sclk: in std_logic;
        sop: out std_logic);
end dsync;
```

```
architecture struct of dsync is
component dataff
  port (d, clk: in std_logic;
        q: out std_logic);
end component;
  signal int1: std_logic;
begin
  c1: dataff port map (inp, sclk, int1);
  c2: dataff port map (int1, sclk, sop);
end struct;
```

Kuruvilla Varghese

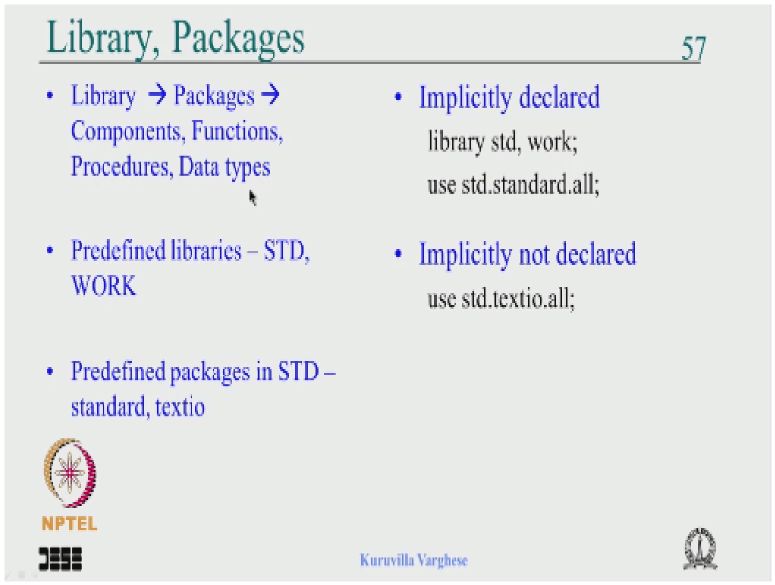
This particular code so, we have the library entity, these are the input as clock is a input sop is a output. So, as here input as clock is a input sop is a output, we need a component called data flip flop. We need an internal signal okay so, that is declared, here in the architecture declaration region, we have the component declaration with the d clock and q. We have a signal declaration, and begin the statement region.

We have instantiated this data flip flop twice with input as clock and internal one, then that is connected to the next input s clock and sop the game is over. Now what we are going to do is we are going to write here we have assume that this particular thing the component and the top level code is in one file or in one project. But we have now, what we have to do is that, we are going to write this in a package, and put it in a library.

And then we are going to use that library and package to do this kind of instantiation okay. So, one thing when we put something in a package this part the component declaration will go along with the package than in the top level entity okay. You would have seen that in a when you write a library you know functions in a library in c code. You have a function prototype which is specified which goes in the header file okay.

So, then you say # include the header file and then that comes at the beginning of the code. Then all the data types is verified something similar happens here okay, we move this component along with the package. And we are going to say use that package and automatically that comes there, and the compiler can check the data type. When you instantiate that is a game so, let us look at the how to write the package, we are going write this component in a package and put it into a library.

(Refer Slide Time: 13:53)



Library, Packages

57

- Library → Packages → Components, Functions, Procedures, Data types
- Predefined libraries – STD, WORK
- Predefined packages in STD – standard, textio

- Implicitly declared

```
library std, work;
use std.standard.all;
```

- Implicitly not declared

```
use std.textio.all;
```

NPTEL

Kuruvilla Varghese

And so, before that a few what about the library and packages the hierarchy of library is like this, you could have a multiple libraries and each library can have multiple packages and each packages can have within it, components, functions, procedures and data types all these can be in a package okay. And there can be multiple package, and mind you there are pre-defined libraries.

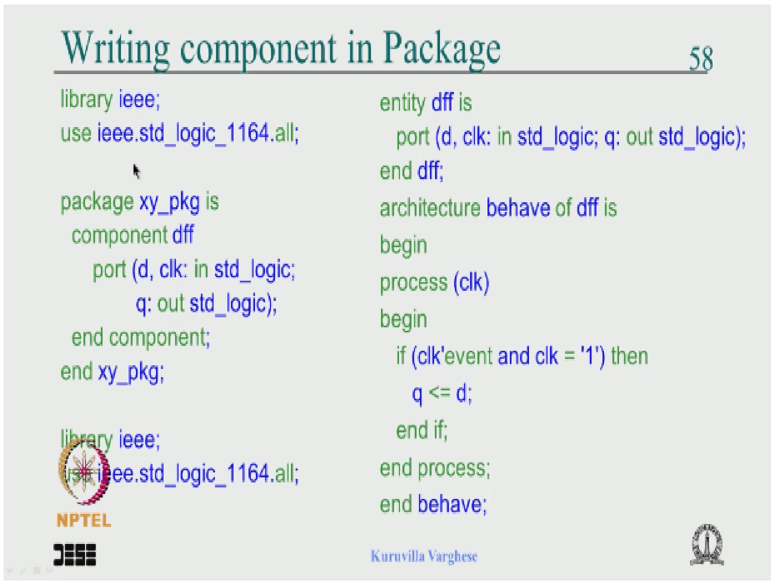
That means we are going to not to declare that that is called STD library and WORK library. STD library is a library which contains all the basic declarations and definitions of the VHDL. That means it contains a packages which declare the bit, the Boolean, the bit vector, the operators like +, - relational operators, logical operators with regard to bit and Boolean.

All that is in a package called standard and that standard package is in the standard library. And the work library is when you compile something, when you work on a design you have written some entity and architecture, and you are trying to synthesise it simulate it. So, during that process that entity and architecture is kind of compiled into this particular library work library. And there are 2 packages in this standard library, one is called standard, and which contains all the standard you know data types, operators, functions and all that.

And it also has another package called textio which basically has the all file declarations, which can be use for test bench and writing something to the screen and things like that. And now mind you this things are implicitly declared, you do not have to declare it when you write a code, you do not have to say library STD comma WORK which is already understood, which is implicit you do not have to say.

Similarly you do not have to say use std.standard.all; because, that is also implicit. But, what is not implicit is this textio package. So, if you want to textio package you have to use like you have to say use std.textio.all; okay. So, that is a brief about the various libraries and packages, standard libraries standard packages what is implicitly declared, what is not implicitly declared and so on. And let us now see how we can write a counter sorry not the counter, the d-flip-flop in a package.

(Refer Slide Time: 17:02)



Writing component in Package

58

```
library ieee;
use ieee.std_logic_1164.all;

package xy_pkg is
  component dff
     port (d, clk: in std_logic;
           q: out std_logic);
  end component;
end xy_pkg;

library ieee;
use ieee.std_logic_1164.all;
```

```
entity dff is
  port (d, clk: in std_logic; q: out std_logic);
end dff;
architecture behave of dff is
begin
process (clk)
begin
  if (clk'event and clk = '1') then
    q <= d;
  end if;
end process;
end behave;
```

Kuruvilla Varghese

So, this is how it is written this is the entity and architecture of the d flip-flop. So, you say library declaration, package declaration, entity which we called now dff the port is din and clock is a input, q is the output and we write the architecture declaration region with the process for the d flip-flop. Now on top of that we write a package body, which say the package some name is end that name.

And within that you have to write the component declaration, which was earlier in the architecture declaration region of the top level entity. Wherever we have use this particular component. But that is brought in here now. And suppose you have multiple component, then you have you can write within this package body. The component dff and component, then component counter, end counter and so on.

And then you have to follow it up the all the entities like if you have dff and a counter as component. Then you put a library declaration, entity for dff, architecture for dff followed with library declaration for counter, entity for counter, architecture for counter and so on okay. Once you do this the package coding is over. Now this particular package has to be compiled into a particular library okay.

Now this is the syntax for writing the package. But how to compile this package into a particular library depends on the tool you have working with okay. So, you have to it depends you know.

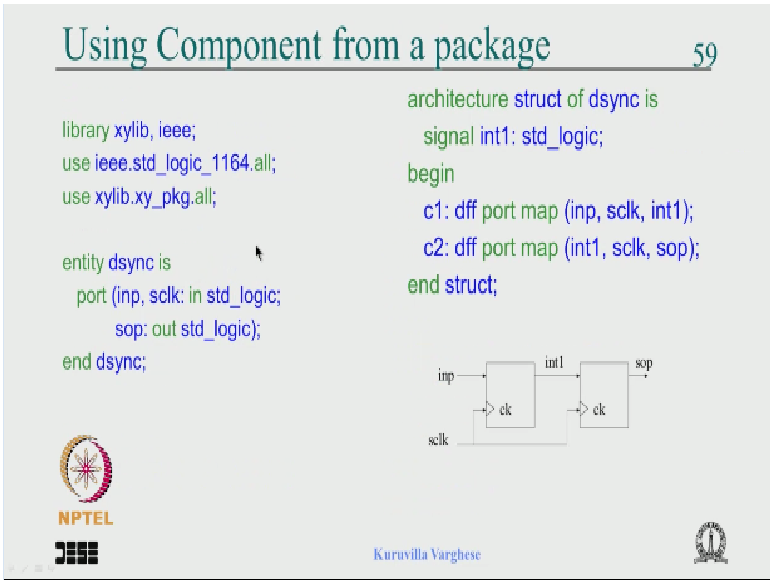
Sometime you work with some simulator like model sim, sometime you work with the vendor tool like the (()) (19:05) quarters tool and so on okay.

So, how to compile a particular package into a library is tool or vendor specific. Many items it means that you have to specify a particular library and invoke the compiler with that particular library as a target okay. So, if it is a you have working with the command line. Then many times you give the compiler the command the file name and some option has to which library it should go to.

But in a kind of something with the GUI you may have to choose particular library and compile you are create project which is specific to a library and compile this file in the project to a particular library and so on. But I am not able to say anything at this moment, maybe when I give a tool demo I will show you whichever tool I am using how that it should be done in a particular tool.

But the essence is that you write the package and compile that into a library. For the time being assume that this particular d flip-flop this d flip-flop is in the package is xy package and let us assume we have compile this package. This whole thing into a library called xy lib, xy _ lib okay. Now we will see how to use that okay.

(Refer Slide Time: 20:48)



Using Component from a package

59

```
library xylib, ieee;
use ieee.std_logic_1164.all;
use xylib.xy_pkg.all;

entity dsync is
  port (inp, sclk: in std_logic;
       sop: out std_logic);
end dsync;
```

```
architecture struct of dsync is
  signal int1: std_logic;
begin
  c1: dff port map (inp, sclk, int1);
  c2: dff port map (int1, sclk, sop);
end struct;
```

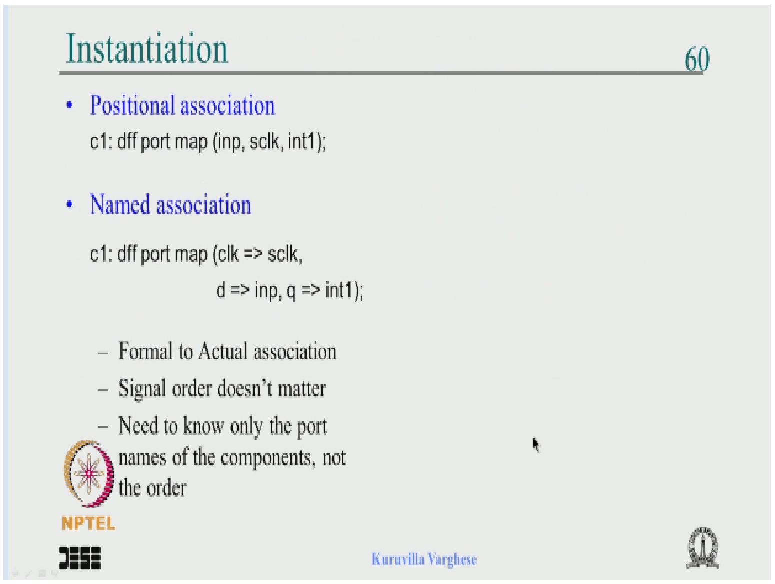
inp
int1
sop
ck
ck
sclk
NPTEL
Kuruvilla Varghese

In the top level component, this is our top level component. So, we write now we are we have used, we have put this flip-flop in a package called xy package and we have put it in a library. So, we are going to say library xy lib ieee, use ieee standard logic 1164.all; and we say use xylib.xy package.all; because that is where this particular component is. And now this the entity of the double synchroniser with input and x clock as input and sop as a output.

Now as I said since we say use xylib xy package.all; and the component declaration is part of the xy package. We do not have to declare the component okay, that is already there in this particular package and we declare the signal, internal signal. Which is signal int1 is standard logic and you just instantiate the component dff. This is the first one port map d input goes to d, clock, s clock goes to clock and int1 is output.

Similarly the second flip-flop int1 is a input, s clock is the clock and sop is output, the game is done. So, that is simple, so the moment you put that in a component, that component in a package and in a library not only you can use it anybody can use this particular package and the library particular component. And they have to just say use xylib.xy package.all; and they use all the components within that particular package. So, that is a meaning of this that is how you write a package.

(Refer Slide Time: 22:36)



Instantiation

60

- Positional association

```
c1: dff port map (inp, sclk, int1);
```

- Named association

```
c1: dff port map (clk => sclk,
                  d => inp, q => int1);
```

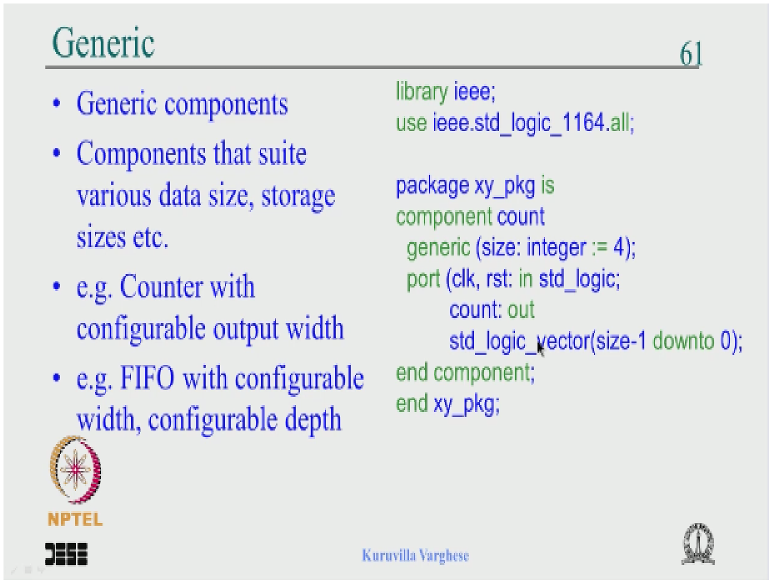
  - Formal to Actual association
  - Signal order doesn't matter
  - Need to know only the port names of the components, not the order

NPTEL

Kuruvilla Varghese

Now when you instantiate the same rule apply we have seen the component instantiation we have done a positional association formal to actual is associated positionally. But you could say you can say here the d can be assigned with a formal to actual. The clock can be assign in particular order, in any order like you can say clock is kind of associated with a s clock, d is associated with inp, q is associated with int1 as for as the first component is concern. So, that is how the instantiation is done.

(Refer Slide Time: 23:21)



Generic

61

- Generic components
- Components that suite various data size, storage sizes etc.
- e.g. Counter with configurable output width
- e.g. FIFO with configurable width, configurable depth

```
library ieee;
use ieee.std_logic_1164.all;

package xy_pkg is
component count
  generic (size: integer := 4);
  port (clk, rst: in std_logic;
        count: out
        std_logic_vector(size-1 downto 0);
end component;
end xy_pkg;
```

NPTEL

Kuruvilla Varghese

So, now let us look at the little more detail when we write a particular component in a package. Say when you write a the code for a counter. Suppose you have written the counter as a 8 bit counter okay. But somebody might need a 16 bit counter okay. So, or somebody might need a 20 bit or a 32 bit. Even if you think of present microprocessor or microcontroller scenario you might end up with an 8 bit counter, 16 bit counter and 32 bit counter and so on .

So, it is very awkward to write various counters and put it to the library. But if you look at the counter code we have written. Everything is kind of like if you have an 8 bit counter to go to 16 bit counter, there is no much change is required. Wherever you have said the counter size the output is say count is standard logic vector say 7 down to 0, if you say 15 down to 0 automatically that becomes a 16 bit counter.

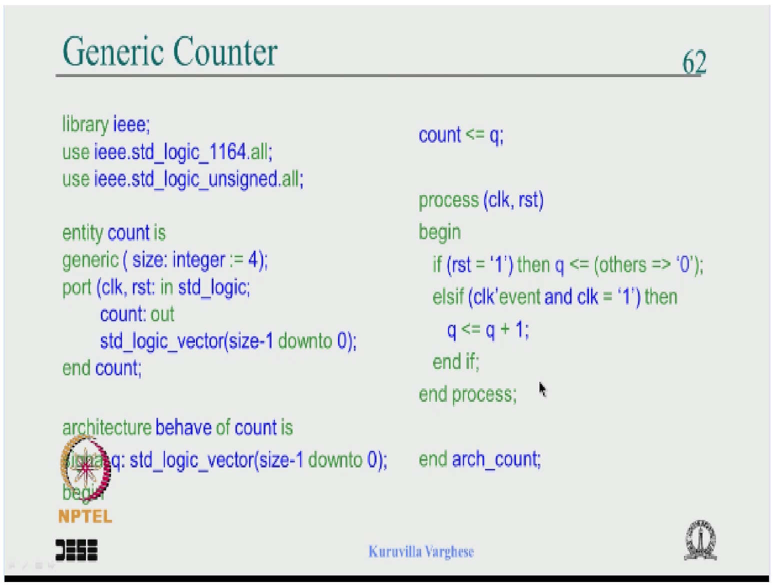
That shows that instead of kind of hot coding the width, if you can say, if you can specify it as a constant and somehow okay. We do not specify the concern value then, but when we instantiate if we can specify what is the width we require, then we will get a generic counter who is whose size or which size is configurable okay. It is size can be configurable that is the basic idea and that depends in the case of counter it is simple.

It may not be very simple to write generic count, generic components with various parameters kind of configurable. You take a FIFO you can have a configurable width may be FIFO as a width of a 8 bit or a 16 bit. And you can also think of a FIFO with the configurable depth like, you can say a 256 entry FIFO or a 1k entry FIFO and so on. So, maybe it is worthwhile to write a configurable width and configurable that depth FIFO.

And similarly you can think about it is say you think about it dual port RAM or a multiple RAM where the width is configurable, the depth is configurable, number of ports are configurable all that. And when it comes to that it may not be that easy to write a generic kind of code. But at least in very simple cases, it is very nice if you can write generic component which are configurable which is configurable sizes that and so on okay.

It is up to your need and imagination that you can write the generic components. So, let us look at an example of a generic counter okay counter which can be configured upon the instantiation that is our idea. So, let us look at the slide so, this is how the generic.

(Refer Slide Time: 26:55)



Generic Counter

62

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity count is
generic ( size: integer := 4);
port (clk, rst: in std_logic;
      count: out
      std_logic_vector(size-1 downto 0);
end count;

architecture behave of count is
signal q: std_logic_vector(size-1 downto 0);
begin

count <= q;

process (clk, rst)
begin
  if (rst = '1') then q <= (others => '0');
  elsif (clk'event and clk = '1') then
    q <= q + 1;
  end if;
end process;

end arch_count;
```

Kuruvilla Varghese

So I will show the code first, then I will show the package before that. So, assume that we have a package declaration followed with a counter declaration. So, the counter you see the library the packages standard logic 1164 and standard logic unsigned. Because we know that we are going to use q gets q+1**.** Here is the game in the entity when we declare the port say the inputs are clock and reset which is kind of in type, count is output.

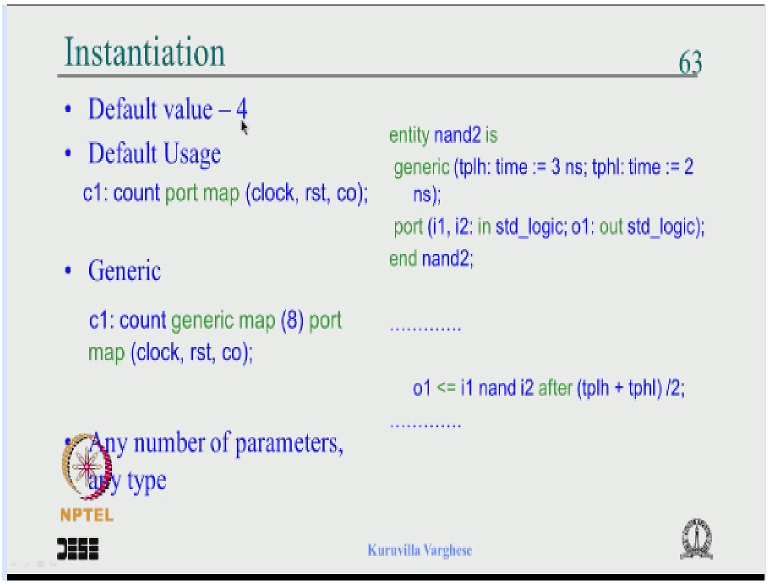
Now which says standard logic vector now instead of saying that 7 down to 0. We define a constant called size okay. We say size -1 down to 0 okay. If it is 8, then it is 8-1 down to 0. If it is 16 then it is 15 down to 0 and so on okay. Now a we define a constant and this nothing but a constant generic is a constant. But the name given in this generic structure is the is this keyword generic.

And we open the bracket and we give the name for that particular generic size it is data type okay size colon data type. And it is default value, this is not a you know the fix value. Suppose at the time of instantiation if you do not specify any width, then it is taken as a 4 bit counter, that is a meaning of this colon equal to 4. Now we need a q signal in the architecture declaration region.

The same thing applies we say instead of particular width, we say q is standard logic vector, size -1 down to 0, count gets q, the code is same clock reset, if reset is 1, q is 0. Otherwise on the clock q gets q +1, end if and process okay. That is a code for a generic counter, since we have putting this into a package, then we write a component package header with the library and the package the 1164 package, use close then we say package xy package is then whatever was in the entity.

Like the generic and the port we write here generic and which is same as the port declaration. Whatever was in the entity declaration, then the port declaration say and component okay generic is required because you see that this data type need to be verify. When we specify when we instantiate the compiler has to verify that this integer, that is why it is written here. And now when you use there are 2 ways of using it, when you exactly instantiate that in a in a top level component.

(Refer Slide Time: 30:08)



Instantiation

63

- Default value – 4
- Default Usage

  c1: count port map (clock, rst, co);
- Generic

  c1: count generic map (8) port map (clock, rst, co);
- Any number of parameters, any type

```
entity nand2 is
 generic (tplh: time := 3 ns; tphl: time := 2
   ns);
 port (i1, i2: in std_logic; o1: out std_logic);
end nand2;

............

   o1 <= i1 nand i2 after (tplh + tphl) /2;
............
```

NPTEL
Kuruvilla Varghese

The default value is 4,so you say say like earlier you can say c1, count port map the clock that means the clock goes to the clock pin, the reset goes to reset pin and the co goes to the output of the counter okay. So, here you will get a counter with the default value which is 4 bit. Now suppose you want this to be an 8 bit counter like port is map from formal to actual.

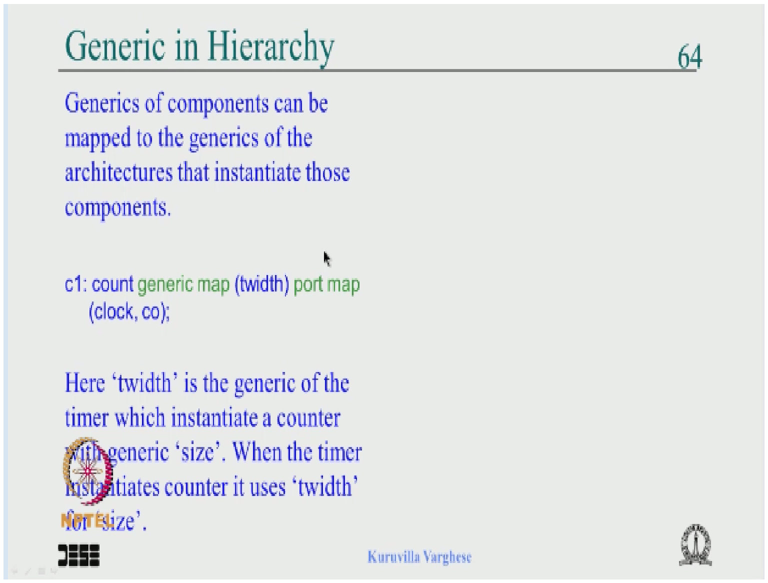
We do a generic map formal was the constant size and the size is map to 8. So, a label, count, generic map 8 and port map, clock reset the co, mind you in the generic you can have any number of parameters. So, you can think of a generic with size, integer equal to 4 or 4 and then you have semi colon, then you say width colon, integer equal to some default value and so and so.

Any number of generic parameters are possible and when you map it you can say say generic map, then the size is **is** map to something with is map to something with a named association you can say that. So, that is shown as an example here say we had a writing this is only for simulation, entity, nand2 that is the entities name and it is a port declaration, i1, i2 is input, o1 is output of standard logic type.

And we have 2 generic, generic tplh that is a propagation delay is of type time, default value is 3 nanosecond, tplh type time 2 nanosecond and somewhere in the code. We are going to write o1 is i1 nand i2 after tplh + tplh divided by 2. Because this is given normally we say after 10 nanosecond after 5 nanosecond. So, now this is given a proper average of the propagation delay.

So, that is how the you know that shows another example of using the generic which shows that the multiple generic and we use more than 1 can be use, you have to map it definitely when you instantiate you have to say nand 2 generic map you can say maybe 5 nanosecond, 7 nanosecond and port map and things like that. So, exactly like this you can say, so that is how you instantiate a generic component.

(Refer Slide Time: 32:59)



Generic in Hierarchy

64

Generics of components can be mapped to the generics of the architectures that instantiate those components.

```
c1: count generic map (twidth) port map
    (clock, co);
```

Here 'twidth' is the generic of the timer which instantiate a counter with generic 'size'. When the timer instatiates counter it uses 'twidth' for 'size'.

Kuruvilla Varghese

Now when you have generic in a hierarchy okay , suppose we have timer which is a generic timer, suppose somebody has written a timer which is generic and the width of the timer is specified as a generic called twidth okay, time width. In that means you write the entity timer here, you say generic twidth is some integer and some width okay. And we assume that after all the timer is nothing but a counter with some decoding.

So, we assume that in the timer code this particular counter is instantiated okay. And counter itself it is generic, you assume that the timer is generic and the counter is generic. So, when you instantiate the counter in the timer, generic timer the size of the counter like we had a size generic, that can be, because if you write that was specific value then there is an issue.

Because if the timer is going to be use instantiated by somebody else with the particular width. So, naturally that has to come down to the counter. So, you can say like say count generic map, twidth which is nothing but the generic of the timer which instantiate this counter. So, essentially the generic can be kind of hierarchically pass down through the component, that is a meaning of it.

I hope that is I have made it clear, because to show an example I have to write so many code. So, that is why this is with this much discussion. I think it is most probably it is clear in your mind. So, that is about the basically the libraries and the packages we have seen, how to put a component in a package. So, there is a particular package body which has to come on top of the entity and architecture.

And then you have to compile that into a particular library, which is tools specific. And once you compile then you just say use that particular library dot the particular package dot all. And you do not have to say component, you do not have to do the component declaration, and the architecture declaration region. Because that is the part of the package now and then you can instantiate the components. Then we have seen how to write generic components.

Component with a kind of generic width, generic depth and so on so, that when you instantiate, you can specify the width, or the depth or the sizes so that you do not kind of write various size components for the library. You have a single generic component which, when you instantiate you specify the configurable parameters then you get what you want, and also we have seen that when you use it hierarchically.

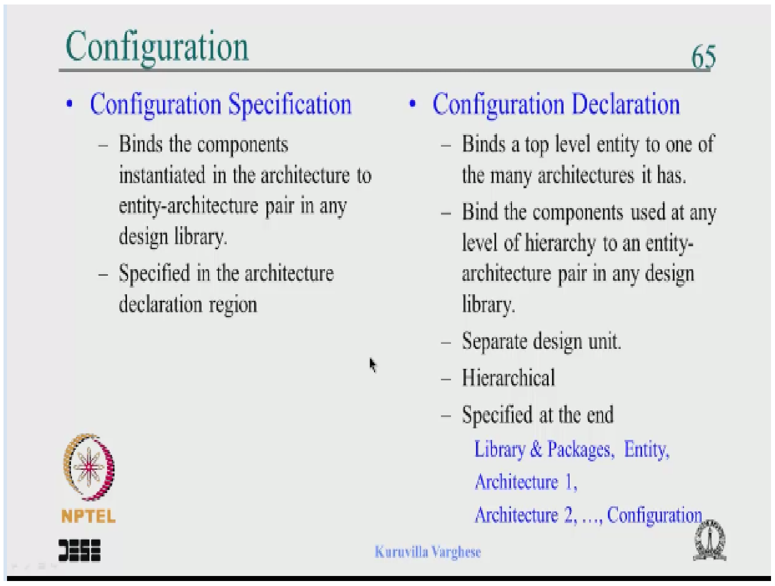
A generic component is used in a generic top level entity the generic of the top level entity can be passed down as generic of the instantiated component, so, that is lot of jugglery with the words. But a I think you can kind of catch the basic idea behind it. So, let us move on so, that kind of completes the packages. But then there is one thing like when you suppose you have a put up at particular say counter.

So, you have called you have particular counter, a counter and your put in a library okay. Now there could be somebody else might write a component called counter again and put it some library. And you use you say at the beginning of the code, library, ieee, xylimb my lib and so on. And then you say use xylib.xy package.all, use mylib.my package.all; assume that there is a component called counter in my lib and xylib.

And somewhere you instantiate that particular counter and depending on which package is kind of specified at the beginning . That will come from that, it may not be the counter that you want okay. So, there has to be some way of telling the tool or the compiler saying that for this particular instantiation. This particular component has to come from a particular library, particular package.

And you can even maybe very specific that maybe that particular component has multiple architectures you can say I want this particular component and a particular architecture. So, the way to specify that is called configuration.

(Refer Slide Time: 38:19)



Configuration

65

- Configuration Specification
  - Binds the components instantiated in the architecture to entity-architecture pair in any design library.
  - Specified in the architecture declaration region
- Configuration Declaration
  - Binds a top level entity to one of the many architectures it has.
  - Bind the components used at any level of hierarchy to an entity-architecture pair in any design library.
  - Separate design unit.
  - Hierarchical
  - Specified at the end
    Library & Packages, Entity,
    Architecture 1,
    Architecture 2, …, Configuration

Kuruvilla Varghese

And there are 2 kind of configuration, configuration specification and configuration declaration okay. So, the configuration specification, basically is specified in the architecture declaration region. And it binds the components instantiated in the architecture statement region to a particular library. That means basically when you say configuration specification; we are going to instantiate lot of components in the architecture kind of statement region.

And those components can be declared in the architecture declaration region. But in the declaration region again we are going to say which library, which package, which is the entity architecture pair in that particular package to be used for this particular instantiation. That is what is configuration specification. But when you say configuration declaration it is more than that, it is a separate unit, design unit.

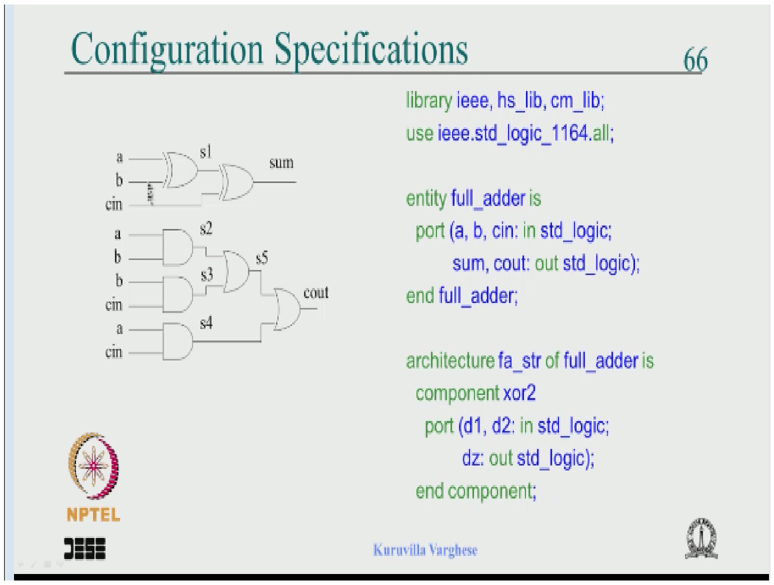
That means you write a entity you write an architecture and you write a configuration for that particular entity okay. It is a separate design unit and it is hierarchical that means that you may write an entity and architecture with the configuration. And a higher level entity might use this particular entity architecture pair along with a configuration and that top level entity can have it is own configuration and so on okay.

So, this configuration declaration is a separate unit, it is hierarchical. It is specified at the end you know. So, you have suppose you have a library and package, you have an entity that entity might have multiple architectures say architecture 1, architecture 2, at the end we specify the configuration, which say at least which of the architecture to be use when we particularly kind of use that in a design Okay, that is a basic idea of configuration.

So, it does 2 things, it will bind the top level entity to one of the many architecture it has. So, this configuration is going to specify which of the architecture is bind to this particular entity okay. Also not only that it will say that suppose this entity uses lot of components. It will say where those components from which library, which package, which architecture it should be used in a where it comes from it **it** is specified.

So, it specify configuration declaration specify 2 things. It is specify which architecture to be used by the top level entity, which are the components where does the component instantiated comes from is specified. So, we will see both okay, configuration specification and configuration declaration .

(Refer Slide Time: 41:39)



Configuration Specifications

66

```
library ieee, hs_lib, cm_lib;
use ieee.std_logic_1164.all;

entity full_adder is
  port (a, b, cin: in std_logic;
        sum, cout: out std_logic);
end full_adder;

architecture fa_str of full_adder is
  component xor2
    port (d1, d2: in std_logic;
          dz: out std_logic);
  end component;
```

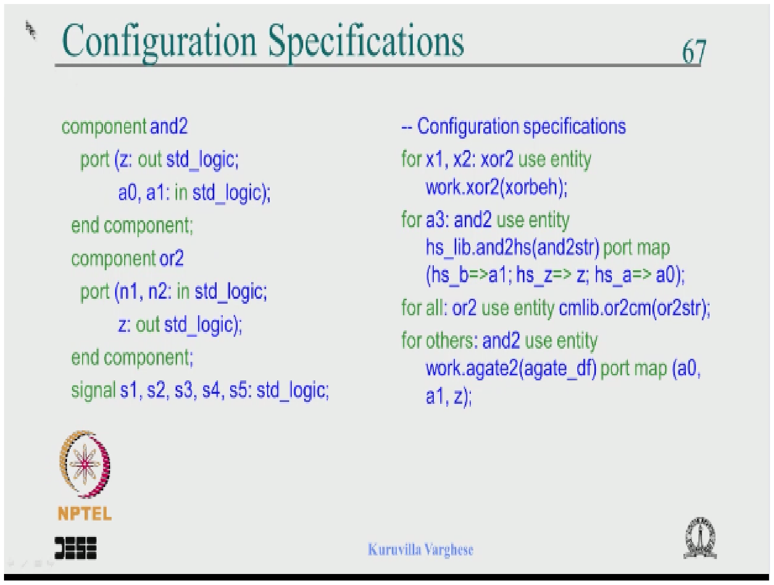
NPTEL

Kuruvilla Varghese

We take this as an example the full order . So, equations are simple a xor b, xor c that is a sum. So we are going to instantiate 2 components, 2 xor gate, ab gives s1, s1 and c in gives the sum. And so here it is the carry out ab, bc, ac, ab gives s2, bc gives s3, ac give s4, then s2, s3 give s5, s5 and s4 gives c out. So, 2 xor gate, 3 AND gate and 2 OR gates okay.

So, let us look at the configuration specification first, so this is the library declaration we are say assuming that there are an addition to ieee, there are hs libs, cm lib where these AND gates and OR gates and xor gates apart. And this is the full order entity, so abc is a input, sum and c out is a the output and this is the declaration region and where we declare the components xor 2, d1, d2 is input, dz is output.

(Refer Slide Time: 42:50)



Configuration Specifications

67

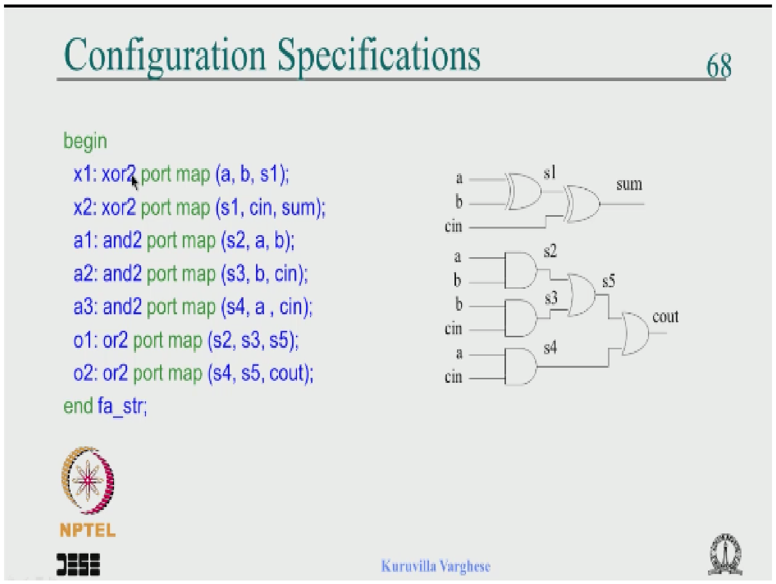
```
component and2
  port (z: out std_logic;
      a0, a1: in std_logic);
 end component;
 component or2
  port (n1, n2: in std_logic;
      z: out std_logic);
 end component;
 signal s1, s2, s3, s4, s5: std_logic;
```

```
-- Configuration specifications
for x1, x2: xor2 use entity
   work.xor2(xorbeh);
for a3: and2 use entity
   hs_lib.and2hs(and2str) port map
   (hs_b=>a1; hs_z=> z; hs_a=> a0);
for all: or2 use entity cmlib.or2cm(or2str);
for others: and2 use entity
   work.agate2(agate_df) port map (a0,
   a1, z);
```

Kuruvilla Varghese

And and2 we specify the output first then the input in the and2 component and then the or2 component where the inputs are first and the output is a second. And we declare the internal signal like s1, s2, s3, s4, s5. So s1, s2,s3, s4, s5 is declared. So, that we can you do the instantiation. Now when it comes to begin this is the instantiation.

(Refer Slide Time: 43:15)



Configuration Specifications

68

```
begin
  x1: xor2 port map (a, b, s1);
  x2: xor2 port map (s1, cin, sum);
  a1: and2 port map (s2, a, b);
  a2: and2 port map (s3, b, cin);
  a3: and2 port map (s4, a , cin);
  o1: or2 port map (s2, s3, s5);
  o2: or2 port map (s4, s5, cout);
end fa_str;
```

NPTEL

Kuruvilla Varghese

You say x1, x2 are the xor gates. You can see it is nicely instantiated abs1 s1cin a sum and2 is kind of reverse so, ab and s2 bc and s3, bc and s3, ac and s4 that is the and2 and or the first or is s2, s3, s5 that is this and s4, s5 cout that is this. Now we are going to say where the x1, x2, a1, a2, a3 o1 or2 is coming from so, we have mind you we have 2 xor gate, 3 and gates and 2 or gates okay, x1, x2 a1 to a3, o1 and o2.

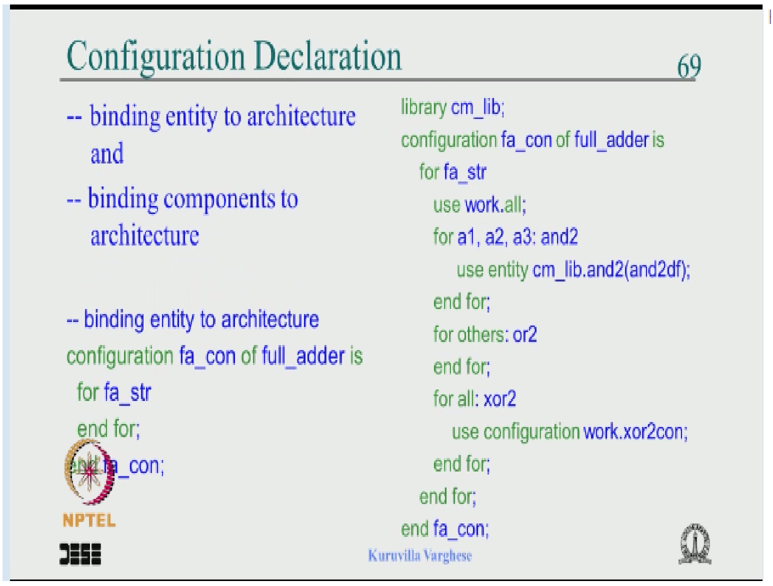
And that is specify before the begin this binding of the component to the library package or the architecture is specified before the begin. So, we go back and you see after the component declaration, signal declaration we are specifying the configuration. So, which say that for x1, x2 these are the instantiated kind of labels colon that is the entity xor2 use entity work which is the work library dot xor2 that means that is the entity name.

We assume that there was we have written an entity called xor2 with multiple architecture which is compiled into while work library and you see in the bracket which is say xor behave or which is the architecture we have use okay. So, we assume xor2 has multiple architecture this is the particular architecture we want to use for x1 and x2. So, the syntax is 4 label the entity use entity library if you are package entity and within the bracket the architecture.

Similarly say for a3 and 2 use entity library here there is no package it is directly compile into library. The entity architecture and you could even change the port mapping by kind of if we have use some other port names in the instantiation. You can change it from the formal to the actual whichever way we have use for the instantiation. So, you can ree do the port mapping and for all or2 to that means for all instantiation of or2, use entity you know this particular library entity and architecture for others.

And do that means we are here we have used only for a3 for all other and2 use the particular library, entity, architecture and the port map is done to change the port mapping. So, that is how the specification configuration specification is done where in the instantiated components are bind to particular library packages, entity and architecture okay.

(Refer Slide Time: 46:45)



Configuration Declaration

69

-- binding entity to architecture and

-- binding components to architecture

```
-- binding entity to architecture
configuration fa_con of full_adder is
  for fa_str
  end for;
end fa_con;
```

```
library cm_lib;
configuration fa_con of full_adder is
  for fa_str
    use work.all;
    for a1, a2, a3: and2
      use entity cm_lib.and2(and2df);
    end for;
    for others: or2
    end for;
    for all: xor2
      use configuration work.xor2con;
    end for;
  end for;
end fa_con;
```

Kuruvilla Varghese

So, let us see the configuration declaration. So, which says 2 purposes which binds an entity to architecture. So, we are assuming that this particular entity for which the configuration is written as multiple architectures and which also bind the components which is used in the architecture to particular library. So, let us look at the very simple example wherein the entity just bind to the architecture.

So, there this is the syntax, configuration which is a keyword a name. So, here we are writing the configuration for full order. So, I have called it fa_con, facon of full adder is and we say end that name, then that is the body of the configuration declaration and inside we say for fa_str. So, this is the architecture which to which this particular entity is binded okay for fa str end for okay.

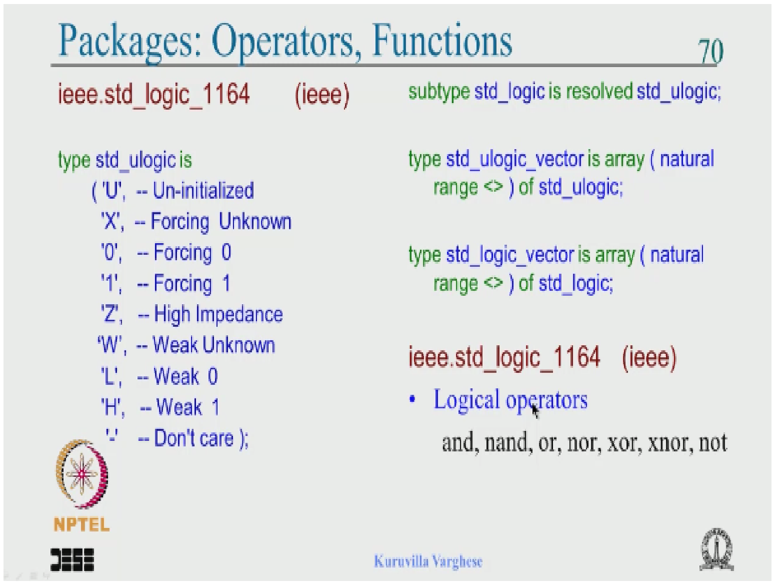
So, very simple thing we bind the entity to architecture, so this particular entity is bound to this particular architecture. But assume that you have lot of components are declared and you want to bind those component to specific library. So, that can be specified within this body of like you say fa str end for within this, you can specify as we have specified in the configuration specification so, that is shown here like configuration.

Name of full adder is and you say for fa_str end4 that is why it is that is how the entity is binded to the architecture. Now within that you can say for a1, a2, a3 and2 use entity, library entity architecture and so on. So, all or how we have done like for others or2 means for all or2. We do not say anything it can come from anywhere, but interestingly we say here for all xor2 that means it now or previous case you add x1, x2 to instantiation.

We say use configuration work dot xor2con that means for this particular entity xor2 there is an architecture may be multiple architecture also in along with the entity. There is a configuration called xor2con which binds the xor2 entity to some architecture which binds if at all some components is use within the xor2 that is bind to a particular library. So, that is how the beginning we said that the configuration is hierarchical.

So, we have a configuration for the full adder and we also have a configuration for the component use within and we just say use that particular configuration. So, that is what how the configuration declaration is done.

(Refer Slide Time: 50:04)



Packages: Operators, Functions

70

ieee.std_logic_1164 (ieee)

```
type std_ulogic is
    ( 'U',  -- Un-initialized
      'X',  -- Forcing  Unknown
      '0',  -- Forcing  0
      '1',  -- Forcing  1
      'Z',  -- High Impedance
      'W',  -- Weak Unknown
      'L',  -- Weak  0
      'H',  -- Weak  1
      '-'   -- Don't care );
```

```
subtype std_logic is resolved std_ulogic;

type std_ulogic_vector is array ( natural
   range <> ) of std_ulogic;

type std_logic_vector is array ( natural
   range <> ) of std_logic;
```

ieee.std_logic_1164 (ieee)

- Logical operators

  and, nand, or, nor, xor, xnor, not

Kuruvilla Varghese

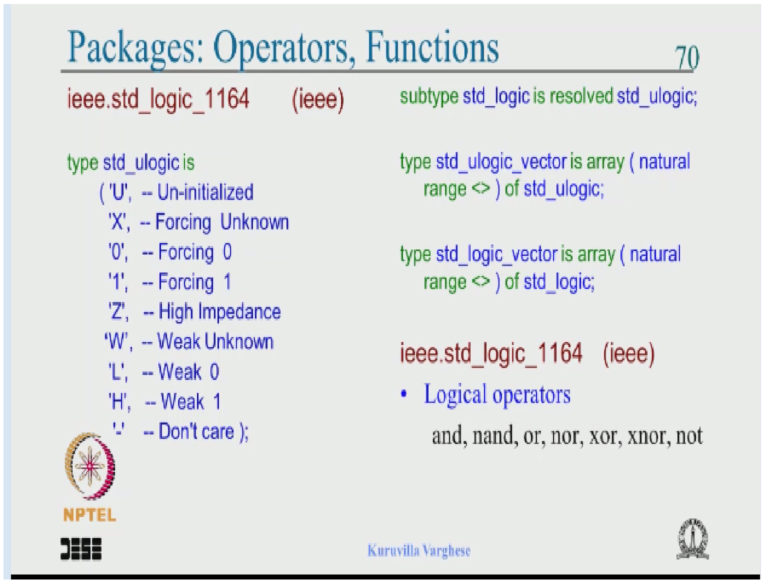
So, now so, the 2 things we have seen with the configuration is that basically configuration specification and configuration declaration, configuration specification is specified in the architecture declaration region. So, whenever you have a some components instantiated in the architecture statement region. This configuration declaration specify where these component labels by labels.

You can specify where it comes from which library which package which entity and which architecture okay. And the configuration specification and bind that particular entity to an architecture also it says if the components are instantiated where it comes from. And the configuration can be hierarchical, so you could write components with configuration and go to a top level entity where just components is used.

And you can say at the top level entity to use that particular configuration for those components instantiated, so that is hierarchical. So, that is kind of complete the library and packages we have seen how to write generic you know library, packages how to put it library, how to instantiate it, how to use generic packages and how to specify where these components should come from by configuration specification, configuration declaration and so on.

So, we have little more portion left in the VHDL for a logical conclusion before we start with the controller in digital **sys** kind of state machine. So, maybe we have a we will have a brief look at it and we will try to complete in the next lecture **.**

(Refer Slide Time: 52:04)



Packages: Operators, Functions

70

ieee.std_logic_1164 (ieee)

```
type std_ulogic is
    ( 'U', -- Un-initialized
      'X', -- Forcing Unknown
      '0', -- Forcing 0
      '1', -- Forcing 1
      'Z', -- High Impedance
      'W', -- Weak Unknown
      'L', -- Weak 0
      'H', -- Weak 1
      '-'  -- Don't care );
```

```
subtype std_logic is resolved std_ulogic;

type std_ulogic_vector is array ( natural
    range <> ) of std_ulogic;

type std_logic_vector is array ( natural
    range <> ) of std_logic;
```

ieee.std_logic_1164 (ieee)

- Logical operators

  and, nand, or, nor, xor, xnor, not

NPTEL

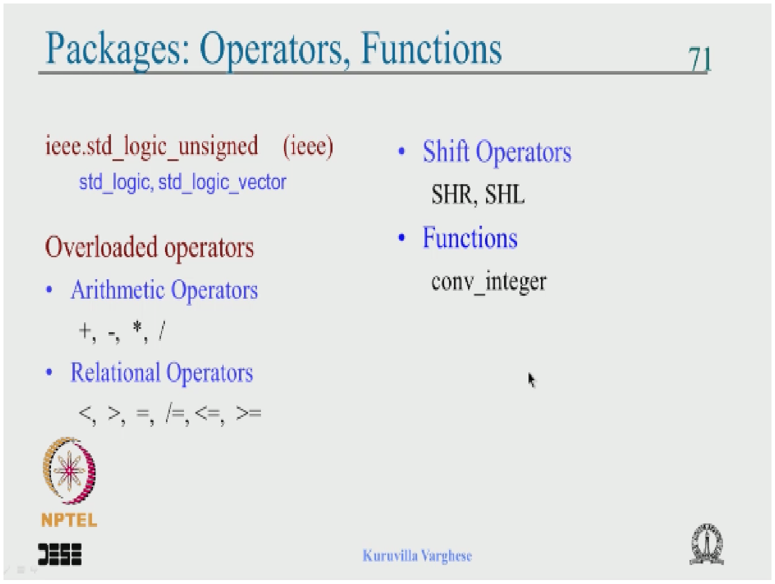
Kuruvilla Varghese

So, I just want to give a various packages **,** operators and function, so that you have familiar you can use it properly. So, we have seen this particular package already like you have a package called standard logic 1164, which is an ieee package in the ieee library. And we know that the standard U logic is define in that, we have seen it takes 9 values U, X, 0, 1, Z, W, L, H and dash which is do not care.

And we have a subtype which is called standard logic which is a resolved version of standard U logic and I have briefly mentioned earlier resolve means when multiple standard U logic drives bus this resolution function tell what is the resultant value that is the meaning of this resolved we will see that as we go along in the VHDL. And we have a standard U logic vector defined as a unconstraint array.

Standard logic vector which is defined as a unconstraint array of standard logic and we have in this standard logic 1164 package all the logical operators like and, nand, or, nor, xor, xnor and not overloaded for this particular standard logic and standard logic vector okay. So, you want to use that, then logical operators with this standard logic. Then use particular library

(Refer Slide Time: 53:39)



Packages: Operators, Functions

71

ieee.std_logic_unsigned (ieee)

std_logic, std_logic_vector

Overloaded operators

- Arithmetic Operators

  +, -, *, /

- Relational Operators

  <, >, =, /=, <=, >=

- Shift Operators

  SHR, SHL

- Functions

  conv_integer

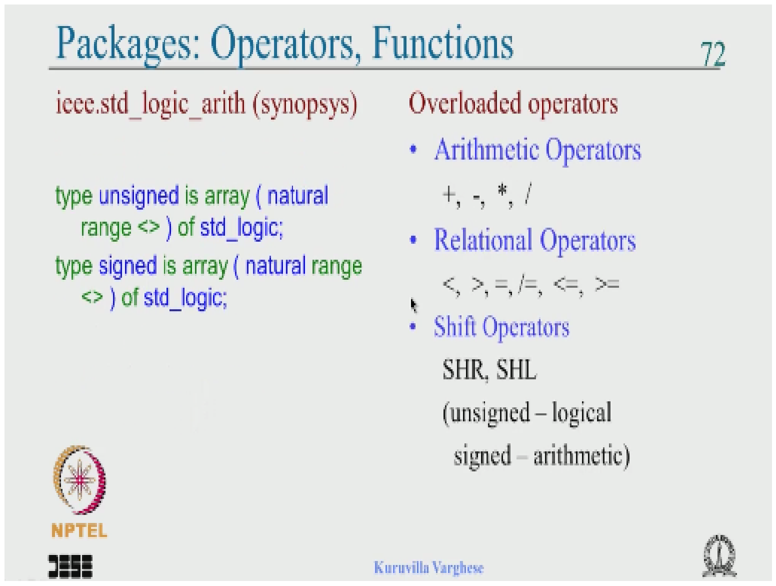
Kuruvilla Varghese

And also it has another because this only specify the logic operators. But **we** we need arithmetic operators, we need relational operators. But that is not found in the 1164 library package and that is specified in this particular package like standard _ logic _ unsigned package it is specified. And basically it is specify the standard logic and standard logic vector and you have these operators' arithmetic operators.

You have +, -, * and /, you have relational operators like <, >, =, /=, <= and not I mean >= you have shift operators now mind you that this is called SHR shift right and SHL which is shift left okay. And you can convert this many a times in the code we need to convert a standard logic vector to an integer say example is that you have some address as standard logic vector.

And you have write trying to write the VHDL code for a memory or a FIFO and this the address location is an array okay. So, you specify address as say for bit 1 010 so, it should go to the tenth location. So, normally that is converted to an integer as a 10 so, where in this is use which is called conv_integer and you specify the standard logic vector here. You get the integer as a it returns an integer which can be use as an integer type.

(Refer Slide Time: 55:40)



Packages: Operators, Functions

72

ieee.std_logic_arith (synopsys)

```
type unsigned is array ( natural
   range <> ) of std_logic;
type signed is array ( natural range
   <> ) of std_logic;
```

Overloaded operators

- Arithmetic Operators
  +, -, *, /
- Relational Operators
  <, >, =, /=, <=, >=
- Shift Operators
  SHR, SHL
  (unsigned – logical
  signed – arithmetic)

NPTEL

Kuruvilla Varghese

So, I think we will see in the next lecture all the packages various packages various operators so, that you can use this packages to quickly design that is the basic idea. And also if you have a idea about various packages, various operators. It is very easy to code and very easy to be kind of keep compatibility. So what is the kind of recommendation of what is current packages.

Because there are some old packages which is kind of super seeded by the new packages so, it is recommended that when you design use the current packages than the old one and things like that. And some packages may be use for a particular only simulation not for synthesis. So, all that we will see in the next lecture please go through the portions revise it try to understand it work out small example. So, that you become through with the syntax and you get idea so, I wish you all the best and thank you.