

Digital Systems Design with PLDs and FPGAs
Kuruvilla Varghese
Department of Electronic Systems Engineering
Indian Institute of Science - Bangalore

Lecture-19
Operators, Delay modelling

So, welcome to this lecture on VHDL in the course digital system design with PLDs and FPGAs. In the last lecture we have looked at basically how to write packages basically to write components in a package and put it in a library.

(Refer Slide Time: 00:41)



And instantiated in a some top level entity also we have looked at the configuration, specification and configuration declaration, which talks essentially about how to say like here instantiating some components from a library. So, the configuration specification bind this instantiation into a specific library, specific package, specific entity and specific architecture that is a basic idea.

And configuration declaration does this thing in a separate design unit. Also it has additional function. Suppose a top level entity has multiple architectures, it tells which architecture to be use for that configuration specified. So, depending on how you write configuration even the top level entity can have multiple architectures, that is the basic idea.

Because when we at the beginning of the lecture we have discussed that the VHDL can have multiple architectures. But we did not say how the tools are going to infer which architecture you require okay. So, this can happen in 2 scenario, one scenario is that, you have a 1 entity and multiple architectures and you want to use at the time of synthesising or simulating or implementing whatever maybe the case that a particular architecture.

Suppose you have an entity with 3 architectures and you are just going to use that entity alone, that component alone. And you want to say out of the 3 which architecture to be used for the current compilation okay. So, that is done by the configuration declaration. In addition suppose in you are entity, you are instantiating components from library and you can specifically say a particular component should come from a particular library.

Otherwise you will at the mercy of the tool. Because there could be similarly named components in various packages and various libraries, and if you just say some library name and use that library, package and various, maybe you will write some 5 use closes. So, the tool will pick whichever comes first to a suppose, you have use a component called counter at the tool is going to look at the first package, second package and so on.

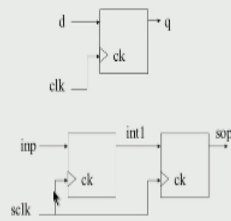
Wherever it finds a name which is matching the instantiation and the arguments of like various ports you have the you know you have the data type. And if the data type matches it is pick up that particular component, probably that not the one you want then in that case you can very specifically say a particular component instantiation you can go to the level of specifying a particular label.

Suppose you have use xor gate 5 times like suppose a labels where x1, x2, x3, x4, x5 these component instantiation labels. And you can say for x1 it has to come from a particular package and for x2 maybe from another package and so on. So, it is very it can very specific very detail. So, that is what we have seen in the last lecture. So, let us quickly look at the slides of the last lecture for a revision.

(Refer Slide Time: 04:47)

Library, Packages

55



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



```
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;
```

Kuruvilla Varghese



So, we have taken an example of a double synchroniser as a top level entity, and a flip-flop as a component, and we have seen that in the normal course you will write the code for the flip-flop so, the library, entity and architecture and in the same file or in the same project.

(Refer Slide Time: 05:08)

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





You will have the double synchroniser circuit the top level entity which has you know the input and the clock and the output. And you declare the component which is just particular data flip-flop declare the signal to interconnect. Because you have declare just declare this internal signal, instantiate it twice and connect all the signals and that is what we have done, declare the component declare the signal, the first flip-flop is instantiated with appropriate mapping. Second one that is it.

(Refer Slide Time: 05:53)

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

But when you write a okay this is a bit about the package and library, the hierarchy is that library, multiple libraries within each library you can have multiple packages and within package a within a package you can have components, functions, procedures and data types. And there are predefined library, one is STD which contains these 2 packages, standard package and textio package. This is what which contains the bit binary the real all those definition, and all the operators related to it.

And the textio is used for file operation, which can be used in test benches. And the work library is the one which your current design whatever you write is compiled into the work library. And normally it is understood that without these 2 no tool can work so, you do not kind of expressed idly declare these like libraries, STD comma work is not required it is implicit it is understood. Similarly, the essential package in the standard library standard.

So you do not have to say use std.standard all this is implicit, this is assume it is there and, but if you have use the textio in the std library that is not implicitly declared. Because it is not used often it is used only in the test benches and so when you want to use it you have to say use std.textio.all.

(Refer Slide Time: 07:43)

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;
```

NPTEL



Kuruvilla Varghese



And this is how we write the package so, the top you write a package header follow that with a entity and architecture. And we are calling our component the same component dff. So, at the bottom you have a library, entity and architecture of d flip-flop. And at the top you have a package header, package some name is end package within that you write the component of this dff exactly same as the component declaration.

In the top level entity which is nothing but this entity whatever is inside the entity is repeated there and that is it. Once you do that this particular component is the package definition is over and this can be put it in a particular library. And that is tool specific, vendor specific how to compile a package into a library, you have to refer to the manual are the user guide of the tools use. But when we go for a demo we can definitely see that at least the package the tool we are going to use how to use that within that tool we will see.

(Refer Slide Time: 09:02)

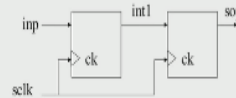
Using Component from a package

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;
```



Kuruvilla Varghese



So, how to use this fact that you have put a component in a package and compile into a library. But we have to see how it has to be used in a top level entity in our case, this double synchroniser. So, first thing is to note that we have put that package in a particular library so, you have to add the library clause library xylib that is where we have assumed. We put this xy package and you have to say use xylib dot xy package dot all.

Then comes the entity but in the architecture declaration region we do not have the component declaration. Because now that is part of the package are part of the component which is in the library. So, we do not have that we just declare the internal signal which is required and then we instantiate as previously this particular component that is all what is required and as I said.

You can have multiple component in a package there is does not matter suppose you have 10 components you can include the component declarations all the component declarations within the package body and follow it up with the entity and architecture of all the components okay, suppose you have 10 components write the entity and architecture of the first component followed with the second component and so on. You can write that in a single file compile it into a library that is simple as it is and definitely.

(Refer Slide Time: 10:44)

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



When you instantiate you can use positional association or named association this is any time better than this particular thing, where you have to keep we have to remember the order of the formal parameters which is sometime difficult to no and more over.

(Refer Slide Time: 11:08)

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



NPTEL

Kuruvilla Varghese



When you write a component one would like it to be generic because if you have a counter then you should be worried about 8 bit counter, 16 bit counter and so on. You should have a counter which is generic in size and when you instantiated you should be able to specify the size we have seen that.

(Refer Slide Time: 11:36)

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



And how we do is that instead of hard coding the size, we define a constant called generic is nothing but a constant say here it is called size, and we say size is 4 we say size – 1 down to 0 so, the syntax is that in the entity the declaration, you include generic and open the parenthesis. And within that you can write any number of generic depending on the requirement say here we require only a size. But in the case of a FIFO we have discussed maybe you have a data width and the size of the FIFO in terms of locations and so on.

So, here size is declared as an integer with the default value of 4, this is the default value, when you instantiate if you do not specify anything, then it is treated as 4 okay. So, and definitely for a 4 bit vector we have 3 down to 0. So, we say size-1 down to 0 and we have a signal. So, wherever required the size is required then you say size-1 down to 0. Then follow it up with the architecture.

And at the top you have a component declaration which is nothing but similar to the entity. So, all these appears at the component side, then you can put it into a library but then we have to see how to use that okay how to specify the size we require when you instantiated.

(Refer Slide Time: 13:13)

Instantiation

63

- Default value – 4
- Default Usage
- Generic
- 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;

c1: count generic map (8) port
  map (clock, rst, co);

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

```


Kurusvillla Varghese


So, that is what is shown here in the normal case if you say count that is a counter we have put port map and the input signal, output signal the width will be 4. But you want us specific width. So, like ports are map you have to say generics are map. So, the count generic map and 8, there is only one generic here. So, that gets a value 8, so everything here is 7 down to 0 now okay. So, you get an 8 bit counter and if there are multiple generics, then you have to say comma like here you can say comma 16 and so on.

And this is nothing but positional association you can have a kind of named association which say size that is what it say here, size and with the forward arrow 8 you can say. Then if there are multiple generic then you put a comma and the next one and so on okay. And this is the normal port map. So you can any number of parameters and we have seen an example of an nand gate with propagation delay, tplh and tphl defined. And we have seen that the behaviour is specified like o1 gets i1 NAND 2 after tplh+tphl by 2 okay.

(Refer Slide Time: 14:42)

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 instantiates counter it uses 'twidth' as 'size'.



Kuruvilla Varghese



And we have also seen that when you instantiate a particular component in a top level entity. The top level entities generic can be pass down to the component which is instantiated okay. Example we have treated was a counter which is a generic counter which is instantiated in a generic timer. So, naturally the width of the timer will be width of the counter. So, when you instantiate the counter in the top level entity of the timer, and top level architecture of the timer.

Then instead of hot coding it we say the generic of the timer twidth. So, when that is ultimately specify you know it is going to be you know instantiated or specified. Then like the timer can be instantiated in a CPU, then that will be specified at that time and that will be pass down to the counter. So, that is the generic in hierarchy.

(Refer Slide Time: 15:48)

• 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



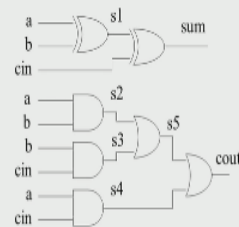
Kuruvilla Varghese



And this is a configuration specification which tells how to bind the instantiated components to entity architecture pair. And this is specified in the architecture declaration region. Configuration declaration has the 2 purposes one is the same purpose bind the components to a particular library. Also it binds the top level entity of this particular top level design to a particular architecture it has okay and this is a separate unit, this is hierarchical okay we will see, what is the meaning of that.

(Refer Slide Time: 16:30)

Configuration Specifications



```
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;
```



Kuruvilla Varghese



And we have seen an example a full order with 2 xor gate, 3 AND gates and 2 OR gates. So, you know the component declaration is xor AND and OR.

(Refer Slide Time: 16:41)

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);

```



NPTEL

Kuruvilla Varghese



(Refer Slide Time: 16:43)

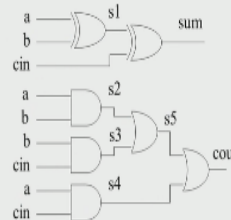
Configuration Specifications

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



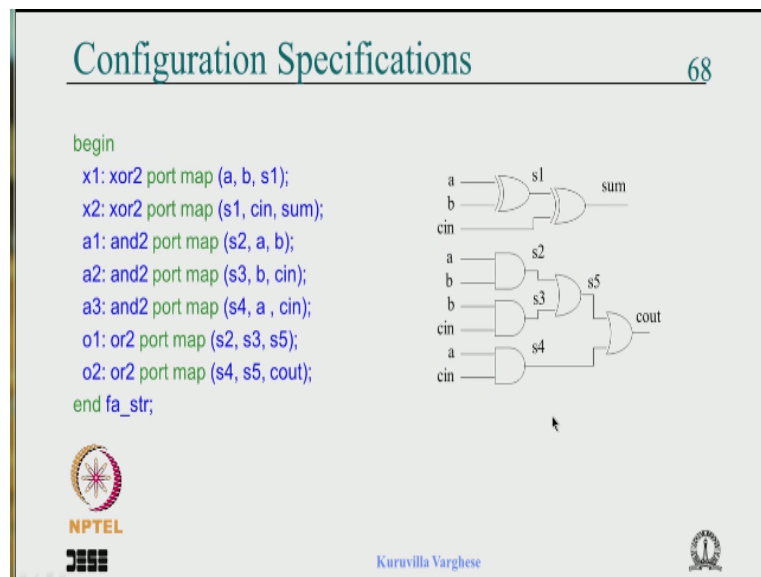
And we have in the architecture statement region we have instantiated this. But in the declaration region you can say for x1, x2 that those are the labels of xor2 instantiation, that it comes from use entity, library dot Package here there is no package. But, because it in the work library, so, the entity and the architecture okay. For a3 that means in the AND gate a3 alone, use entity this library, this entity, this architecture.

And whatever the formal ports which is in the library is map to something called a1, just a name change. In case you have different name here, we have not use we have use a positional association, so probably this does not matter. But suppose you have said suppose instead of hsb.

Suppose here we write say for a3, we say instead of saying hsb map to s4. Suppose you had said kind of a1 maps to s4 then this could be done to change it.

And there are 2 special syntax for all or2, that means for all the instantiation of or2 use a particular entity architecture pair. You say for others that means here you see AND gate instantiation for particular a3 we have used a particular component from a library for all others that means a1 and a2 use something for 1 square. That is a way to specify

(Refer Slide Time: 18:25)



And when it comes to configuration declaration for a single binding of the entity, top level entity to architecture you write a configuration name of the entity you say for a particular architecture name end for, that means this entity is map to this particular thing that is all. But you want to specify the component binding that can be done inside. So, you say for a1, a2, a3 use a particular AND gate say end for; for others or2, end for.

That means we are not specifying anything, for all xor2, use configuration work dot xorcon, that means the xor2 has a configuration as part of it entity architecture which say that which are the where this particular thing should comes from and which is architecture to use all that. So, that shows the hierarchy of the configuration.

(Refer Slide Time: 19:31)

Packages: Operators, Functions

70

```
ieee.std_logic_1164 (ieee) subtype std_logic is resolved std_ulogic;
```

```
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 );
```

```
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 think this is what we have started briefly in the last classes. So, there are different packages and different packages are different operators, functions and it can be little confusing at the beginning which particular operator to use which particular function to use and so on. So, we will look at and it is very difficult maybe it is not given in a textbook sometime.

And one way to know this particular operators definition is by looking into the library okay. That would mean that you go through the source code of the library okay. That is a very compare something. Because you have to open the source in VHDL or very low you know go through all the code and in the process you mix some changes to it. And then if you compile it for a some tool, simulated tool, then it can give errors and all that.

So, I am giving you a brief about the various packages, various operators and functions. So, the primary package we have come across other than the standard package is the standard logic 1164, where this standard u logic is defined with all the 9 values, we have seen that okay. And a standard logic is nothing but a standard u logic. But it is going through a resolution function in the case of multiple drivers.

And we have a standard logic vector, standard u logic vector and standard logic vector which is defined as an a unconstraint array of standard u logic and standard logic respectively. So, it can take any value of to raise 32 okay. And when we declare we constrain it by specifying the size

and standard logic 1164 contains only the logical operators for this particular standard u logic, standard logic, standard u logic vector and standard logic vector.

So, if you want to do some arithmetic with it, then we need to use a different package this contains only the logical operators. So, please keep that in mind.

(Refer Slide Time: 22:06)

The slide is titled "Packages: Operators, Functions" and is numbered 71. It lists the contents of the `ieee.std_logic_unsigned` package, which is part of the IEEE standard. The package includes the following:

- `std_logic, std_logic_vector`
- Overloaded operators**
 - Arithmetic Operators**: `+, -, *, /`
 - Relational Operators**: `<, >, =, /=, <=, >=`
- Shift Operators**: `SHR, SHL`
- Functions**: `conv_integer`

The slide also features the NPTEL logo and the name Kuruvilla Varghese.

So, the next thing we have already seen in some example which is the package standard logic unsigned this is also the an ieee packages. So, it is ieee dot Standard logic unsigned, it has standard logic and standard logic vector as a data type. The operators like `+`, `-`, `*`, `/` is overloaded for it also it has relational operators all `<`, `>`, `=`, `/=`, `<=`, `>=` all that is there. So, you can the moment you say use ieee standard logic unsigned you can use all these.

In addition, it has shift operators which is called SHR which is shift right and SHL which is shift left okay, mind you this is all logical shift there no arithmetic shift which is offered in this particular library. So, if you are working with tools compliment maybe it is little difficult either you will not be use this particular operator you may have to write code for arithmetic shift okay.

And as I said the digital I mean design when you design through the VHDL it enforces strict tight checking. So, suppose you have a standard logic vector which has to be converted to integer, then you have to specifically convert it to integer, it will not be you cannot assign a standard

logic vector to an integer maybe it is 8 bit. but you know that the 8 bit goes from 0, 255 for unsigned.

But, that like you cannot assigned that when integer unless you convert standard logic vector to an integer. So, this is this particular function convert this standard logic vector to integer you can say `conv_integer` open the bracket and whichever is your standard logic vector you know the object you can put here until return an integer you know. That is how this is this particular function to be used.

(Refer Slide Time: 24:38)

Packages: Operators, Functions 72

ieee.std_logic_arith (synopsis)

```
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

And there is a synopsis specified library which is called `std_logic_arith` okay. Now that does not use standard logic vector as a base vector type. It has 2 arrays of standard logic which is unconstrained, one is called unsigned other is called signed as this name suggest that can be used for arithmetic you can have unsigned addition and signed addition it is exactly similar to standard logic vector.

But then the name is unsigned and the name is signed here. So, you have this packages all the operators overloaded for unsigned and signed. So, you have all the arithmetic operators all the relational operators like in the previous case you have like standard logic unsigned you have SHR and SHL. But if the type you are using is unsigned, then it is a logical shift. If it is a signed data type you are using, then this will be an arithmetic shift automatically.

So, you do not have to worry, so if you play with the tools complement number, then this can be very useful. Because you do some kind of computation with the sign extension, then we have to shift it properly with the sign bit, otherwise the value will not be correct, the result will not be correct. So, this is taken care of in this SHR and SHL to provided use a proper data type signed.

(Refer Slide Time: 26:29)


Packages: Operators, Functions 73

ieee.std_logic_arith (synopsys)

Conversion Functions


from: std_logic_vector,
unsigned, signed, integer

conv_integer
conv_unsigned
conv_signed
conv_std_logic_vector

 NPTEL

- **Usage**

```
library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_arith.all ;
use ieee.std_logic_unsigned.all ;
```
- **Recommendations**
 - Use std_logic_arith for numeric operations
 - Use std_logic_unsigned only for counters and testbenches
 - Don't use the package std_logic_signed



Kuruvilla Varghese

So, it also has conversion function, so basically it has conversion function from standard logic vector unsigned, signed and integer. So, there are 4 conversion function so, the convert integer you convert from all the other 3 like standard logic vector signed and unsigned to the integer. So, you just open the bracket and write whatever so, that means this convert integer is overloaded 3 times in this case for standard logic vector unsigned and signed.

Similarly you have convert unsigned which will put from standard logic vector signed and integer, converts signed, convert standard logic vector and all that okay. So, that it means that it convert to standard logic vector convert to integer from these 3 types . Whatever is not mention here is the is from kind of data type. Now if you want to use this libraries, synopsis libraries you say library ieee.

Use ieee standard logic 1164 because the standard logic base type itself is defined here. Then you say standard logic arith where in you can use kind of all the arithmetic relationship. And logical

operators and sorry functions and you say at the end ieee standard logic unsigned. Because the shift operators unsigned has yeah there are arithmetic and shift and all that operators here.

And mind you this is put below this, so that main operators are from the arith. So, these are the recommendation for all the design use standard logic arith and unsigned you can use it for counters and test benches and do not use the package, there is a package called std logic signed do not use it. Now like there is a package from ieee standard like in place of the synopsis packages. So, these 2 are synopsis packages arith and sorry arith is a synopsis package.

(Refer Slide Time: 29:05)

The slide is titled "Packages: Operators, Functions" with a slide number "74" in the top right corner. It lists the package "ieee.numeric_std (ieee)". Below this, it shows two type declarations: "type unsigned is array (natural range <>) of std_logic;" and "type signed is array (natural range <>) of std_logic;". To the right, under the heading "Overloaded operators", there are three bullet points: "Arithmetic Operators" with symbols "+, -, *, /, abs, rem, mod", "Relational Operators" with symbols "<, >, =, /=, <=, >=", and "Logical operators" with symbols "and, nand, or, nor, xor, xnor, not". At the bottom left is the NPTEL logo, and at the bottom right is the name "Kuvilla Varghese" and a small circular logo.

Packages: Operators, Functions 74

ieee.numeric_std (ieee)

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

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

Overloaded operators

- Arithmetic Operators
+, -, *, /, abs, rem, mod
- Relational Operators
<, >, =, /=, <=, >=
- Logical operators
and, nand, or, nor, xor, xnor, not

NPTEL

Kuvilla Varghese

So, there is another similar package from the ieee, so that is called numeric_std or we called numeric standard similar to arith you have unsigned and signed specify as a array of standard logic. And you have arithmetic operators now you not only you have +, -, *, /, you have absolute, rem and mod you have in the numeric standard you have relational operators, you have logical operators.

(Refer Slide Time: 29:40)

Packages: Operators, Functions

75

ieee.numeric_std (ieee)

• Usage

• Shift operators (signed, unsigned)

shift_left, shift_right
rotate_left, rotate_right
sll, srl, rol, ror

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

• Conversion Functions

to_integer
to_unsigned
to_signed



NPTEL

Kuruvilla Varghese



You have shift operations in your shift left, shift right, rotate left, rotate right, sll, srl which is nothing but this rol, ror which is nothing but this. So, exactly same and you have conversion function 2 integer 2 unsigned, signed from like if you take 2 integer it is from unsigned or signed or standard logic vector, and similarly others and how to use this particular package.

So, you say library ieee, library use ieee dot Standard logic 1164 all; that is required because standard logic is defined there. And use ieee dot numeric standard dot all; okay. That is how the ieee numeric standard library is used. In the earlier case we have seen that we have to use 1164 unsigned and arith. But here is only numeric standard need to be use. So, that is the various libraries, various packages operators and functions.

(Refer Slide Time: 30:53)

Type Conversions

76

- Automatic
 - Between base types and subtypes
- Using Conversion Functions
 - e.g. `to_integer`, `conv_integer`
- Type Casting
 - between signed, unsigned, and `std_logic_vector`

```
sl_vect <= std_logic_vector(usg_vect)
sl_vect <= std_logic_vector(sg_vect)
usg_vect <= unsigned(sl_vect)
sg_vect <= signed(sl_vect)

signed("1101")
```



NPTEL



Kuruvilla Varghese



Now when it come to type conversion many a times you have to do, because we use different libraries and so you have to move between sometime standard logic vector to signed and unsigned and integer and so on. So, there are 3 ways you can convert, so it is automatic between base type and subtype okay. So, suppose you have a a subtype then you do not worry you have defined a subtype of something.

Then you do not have to worry it just assigned it will work, it is automatic you do not have to do the type conversion. In some cases you have to use the express it conversion function like `to_integer`, `conv_integer` and so on maybe so, you have to convert from standard logic vector to an integer. So, you can use either of this functions and you know that the signed, unsigned and standard logic vector all are the unconstraint array of standard logic.

So, essentially though name is different the data type is same. It is only the name difference. So, the VHDL allows you to do a type casting as in c. So, when you convert between these 3 between any 2 of them. When you can just use a type casting say suppose you have a standard logic vector called `sl_vect` and you want to assign this unsigned vector `usg_vect` to this.

You do not do a, you do not have call `std_logic_vector` or to standard logic or covert standard logic just say `std_logic_vector(usg_vector)` and get the standard logic vector and similarly suppose you have `usg_unsigned_vector` and you want to assign a standard logic vector,

you just say unsigned then you give this standard logic vector as an argument then unsigned vector will get it.

And particularly suppose you want to use some numerical value for whatever purpose. And say you want to pass this signed number then you just say signed you know in the code you give that numerical value of the standard logic vector it will be automatically converted to the signed data type okay. So, that is about do the type conversion between the various similar data types.

Now you should be careful when we write when we call a function say suppose in some case, we needed a standard logic vector to be converted to an integer okay. So, we will call a function, which convert the standard logic to an integer okay. And if you know that suppose you have a 4 bit binary number to convert to an integer the algorithm is that just simple you know you trape to the binary number bit twice.

Wherever there is 1 you say the you have an accumulator you say that accumulator is nothing but accumulator + to raise to i, i is the current of that bit position okay. So, if you have 1010, so it is basically to raise to 8 + sorry to raise to 3+ to raise to 1 which is 10 okay. And that you go through an iteration like for i in 0 to 3 then if i is 1, then some variable is variable + to raise to i and so on okay.

So, that is how it is computed, but like you should not think that this is doing to be synthesise into a hardware okay the data type checking is enforced by the VHDL as a language okay. So, the fact that we write some code to convert a data type, that should not be synthesised into a circuit okay. So, basically that you should understand and so there are attributes which say like when you write a library function for this type conversion. There are like attributes which is inserted. So that the synthesis tool will not synthesise that part of the code, that you should understand, so, that is what is written here.

(Refer Slide Time: 35:58)

- Type conversion is required when you connect a signal of one data type (e.g. integer) to another (e.g. `std_logic_vector`), as VHDL is a strict type checking language
- But, in a code, where a `std_logic_vector` address is converted to integer, as an index into a memory array, type conversion implies an address decoder
- Type conversion implies no hardware, Hence directives (user defined attributes) are given to synthesis tool, not to synthesize the code.



NPTEL
JNTU

Kuruvilla Varghese



But, having said that sometime when you convert suppose take an example memory okay. And suppose you have an a memory with 8 bit address okay. And so, the number of locations are 256, it will be normally address from 0 to 255 okay. Now like in VHDL code you can specify an array of location index by the address okay. So, you will have a memory array which goes from 0 to 255.

So, when you suppose you are reading a memory location you would put the output of that particular memory location to a data bus, and you need to specify the array index. So, to that extend we will kind of convert the address which is in the standard logic vector to an integer. And supply this as an index to the array okay. Now though the type conversion does not imply any hardware.

But there is a hardware here which is hidden, that means you have converting and index which is in the standard logic vector to an integer which is indexing an array okay. Then that represent a an address decoder. Because you know that in a memory a particular location is access by an address decoder, you specify the address, a particular decoder will go and select a particular location.

And so, that like implicitly sometime this the type conversion not the type conversion alone. But the fact that converted integer is indexed into a an array can represent a address decoder I maybe

what I have written is little kind of misleading . It is not the type conversion which implies an address decoder. The fact that, that converted number is used as an index into array can mean that it is an address decoder. So, that is what I want to convey. So, I hope you are kind of clear about this particular type conversion.

(Refer Slide Time: 38:36)

Arithmetic
78

```

signal a, b, s: unsigned(7 downto 0)
;
signal s9: unsigned(8 downto 0);
signal s7: unsigned(6 downto 0);



-- Simple Addition, no carry out
s <= a + b;

-- Carry Out in result
s9 <= ('0' & a) + ('0' & b);

-- For smaller result, slice input arrays
s7 <= a(6 downto 0) + b(6 downto 0)

signal a, b, s: unsigned(7 downto 0);
signal s9: unsigned(8 downto 0);
signal cin: std_logic;
-- Carry in
s9 <= (a & '1') + (b & cin);
s <= s9(8 downto 1);

```


Kuruvilla Varghese


So, let us see some examples of arithmetic okay maybe some kind of at least at the start this will bring in clarity. Suppose imagine there are a, b these are the input kind of vectors, 8 bit vectors which is defined as a unsigned 7 down to 0. S is we are going to assign some output unsigned 7 down to 0 all these a, b, s are 8 bit and we have an 9 bit s which is called s9 which is unsigned 8 down to 0.

We also have an s7 which is unsigned 6 down to 0 okay. Now very simple addition suppose, we are kind of doing an addition or you are implementing an adder. Then the simple addition is that you say s get a +b okay. Now a is 8 bit, b is 8 bit and s is also 8 bit. and which normally in a digital course that you would have learned that you add 2 8bits and you end up with a 9 bit. but when you work with a operator you add a and b.

And you just end up with in kind of 8 bit result, the carry is ignored when it is synthesise or implemented. But in some case you in like when we suppose you are doing a multiplication we are trying to design a multiplier, then you know that in the multiplier algorithm you need to add

the partial products. And there an AND then you have a shifting. So, there if you add 2 8bit, that will result in a 9 bit and we require the 9 bit okay.

In that case maybe in a simple addition we may not require the carry bit, but there are cases where we require the carry bit. In such a case what we do is that see how we can get that 9 bit. So, s_9 is an 9 bit vector which as assigned then we up end 0 at the as a more significant bit of a. So, you say 0 concatenate with a which is 8 bit + 0 concatenate with b. So, if there is a carry from msp which is 7 a7 or b7 position.

Then that goes to the eighth position and you get a 9 bit result okay. Suppose you want to live with only the 8 bit result, then you can definitely say like sorry a 7 bit result then you can say a6 down to 0 + b6 down to 0. Suppose you want to you have a larger input and you have a smaller output, then you pickup the same size as a output. Then you will get there is no issue.

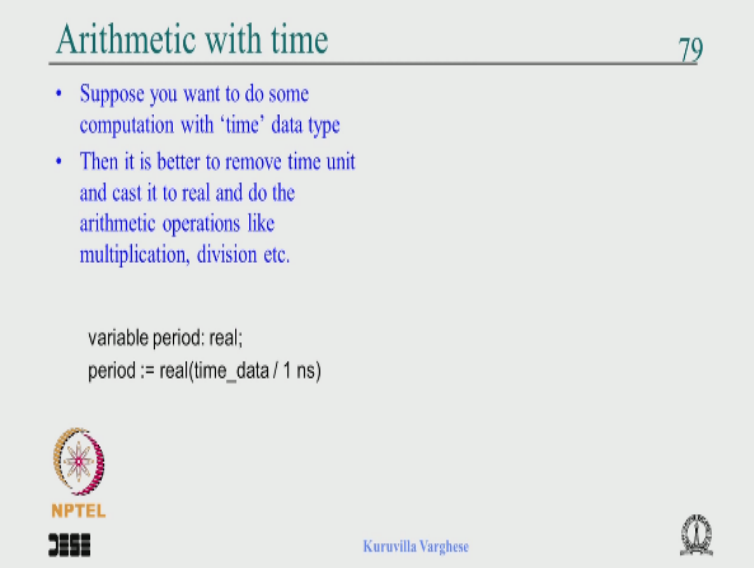
Because this is 7 down to 0 which is 8 bit, but a 7 is only 7, 7 bits y, so you use a6 down to 0 + b6 down to 0. And some cases you may need to supply carry in okay maybe you have split the add of whatever reason into 2 stages the carry out of the previous stage comma as a carry in of the next stage. In such a case we do the opposite of upending 0 at the msp position, what we do is that this is a 9 bit result.

And a and 1 at the lease significant base + b and the carry input okay. So, you get the s_9 okay, 9 bit result. But this part should not be part of the address sorry part of the result okay. Because which say that suppose we are interested in having the some outputs. We want to push some carry if it is generated to the next the first stage. So, here you know you have made it 1. So, if the carry in is 0 then the carry out from the first stage is 0.

But if it is 1, then $1 + 1$ is 0, the carry in to the second stage is 1 so, but this 1 output, the some output itself, as the lease significant bit is not what we required. So, we ignore it, so ultimately you just state the up to the first bit from the 8 bit and cast it to an 8 bit result okay. That is how we use the carry in if required. So, this shows how to use a carry out, this shows how to use the carry in with this standard operators and we have use,

So, you do not have to write a ripple order code or a look ward order code. Normally you can stick with this + in the case of FPGA there are dedicated resources to implement the +. So, which is very kind of high performance and so on. We will see that when we go to the particular FPGA lectures and that is how that is about the arithmetic width kind of addition width the carry out with carry in and so on.

(Refer Slide Time: 44:24)



Arithmetic with time 79

- Suppose you want to do some computation with 'time' data type
- Then it is better to remove time unit and cast it to real and do the arithmetic operations like multiplication, division etc.

```
variable period: real;  
period := real(time_data / 1 ns)
```

NPTEL

Kuruvilla Varghese

So, now we let us look at arithmetic with time that you know that time is specified as kind of integer units. So, you say 1 nanosecond, 2 nanosecond you do not say 1.5 nanosecond. So, but when you do some at least for simulation wise when you do some calculation like you want to calculate the period and you want to multiply the period with something. Then you may end up in a kind of real numbers.

So, it will be good if you can convert the time to a real numbers. So, suppose here I am showing that we declare a variable which is period which is of type real and now we say period get real that is a type casting. And time data which is in nanosecond say 100 nanosecond divided by 1 nanosecond. So, we remove the unit and you get a 100 and that is cast it is real and then you get a period in real.

Then you multiply divide then you get the real number as a result. So, that is how we do the arithmetic with time. So, briefly we have looked at the packages basically the ieee standard logic 1164 package standard logic unsigned which is ieee library, a synopsis package called standard logic arith how to use that various arithmetic operators, relational operators, logical operators, shift operators and conversion function.

And we have seen numeric standard how to use that. We also said the type conversion is not kind of does not represent a hardware it is just for the language type checking. But when we in special cases when we convert and when we use it for indexing and array it some represent some decoder, that should be should not be forgotten. And ultimately we have seen some example of arithmetic like some similar size result as a input when you add.

And when you have a carry input what to do, when you have carry output how to get the result, the wider result with the carry out and so on. And we have ultimately seen the time some computation using the time, during simulation where we convert the time unit into the real numbers. So, that you can multiply where you end with a fractions, then that can be preserve if you convert to the real number.

(Refer Slide Time: 46:59)



Delay Modeling

80

- Inertial delay,
- Transport delay

- Inertial delay
 - Models delay through capacitive networks and through gates with threshold values
 - Pulse rejection value less than the inertial delay with "reject" clause.

- Two parameters:
 - Minimum pulse width required for the device to recognize the level change (i.e. for the input to cross the threshold)
 - Propagation delay of the device



Kuruvilla Varghese

So, let us talk about this particular topic and which is called delay modelling, how the VHDL model the delay. We have already seen the syntax we say that something get after 5 nanosecond.

Then there is a delay for that output to display or to come out with the 5 nanosecond delay. So, there are 2 types of delay specified in the VHDL one is called inertial delay and another is called transport delay.

So, basically inertial delay will model the delay through capacitive networks you know that you have a line with capacitance you apply some binary value. Then that capacitor has to charge up to that particular value. And before that before it getting fully charge if the input applied is removed, then there won't be any effect, so that is what is inertial delay and that works for the gates with threshold.

Because you have an inverter and you apply a one at the input of the inverter. Then naturally there is a capacitance on the line. And the input has to charge up say previously it was 0 above the threshold for the output to start appearing okay. So, there is like at the input there is some certain pulse width then only like minimum pulse width, then only the input will go above the threshold and the output will appear okay.

And also there is a propagation delay. So, inertial delay has 2 parts, one is a minimum a pulse width which is required for the output to come okay. Suppose once again suppose you have a inverter with 5 nanosecond delay. And if you apply a 1 nanosecond pulse at the input of the inverter for sure you can be sure that it will not make any effect at the output.

It will not appear at the output, because the propagation delay was 5 nanosecond then you apply narrow pulse a 1 nanosecond, so it will not appear okay. But we are not in position to say that you appear apply a 4.5 nanosecond pulse whether it will appear at the output okay. That we are not able to say at the gate level. We have no great way to predict this if by simulation are anything like that.

But if you consider the equivalent transistor lay out and do like suppose you have made an and gate or an inverter using say take an inverter with the PMOS transistor and NMOS transistor do the lay out do the place and route of VLSI chip. And do as spy simulation then we you will able

to see the exact effect, but when we model this at the gate level. We do not have such across is okay.

So, it is at a very gross level we are going to model and it has limitation. You should understand that there are certain limitation with regard to this kind of modelling and that can be reflected. There will be side effect if you are careful in simulation; you will see the effect of this in certain cases depending on particular model used to represent the delay in the VHDL output code particularly for simulation timing simulation so on.

So, it has two parameters one is the minimum pulse width required for the output to appear second is the propagation delay itself okay. So, we will see so, that is the inertial delay so, we will see the inertial delay and move on to the transport delay.



(Refer Slide Time: 51:13)

Delay Modeling

81

```
x <= a after 5 ns;  
x <= inertial a after 5 ns;  
y <= reject 3 ns inertial a after 5 ns;  
  
u <= '1' after 5 ns, '0' after 8 ns,  
    '1' after 12 ns;  
  
z <= transport a after 5 ns;
```

- Transport delay
 - Models delay through transmission lines and networks. No pulse rejection.



Kuvuilla Varghese

So, let us look at the inertial delay the syntax for is that say x gets a after 5 nanosecond okay. That means the meaning of it is that between x and a there is a 5 nanosecond delay also it means that anything less than 5 nanosecond suppose you apply to a will not appear at the x okay that is a meaning of it. But exactly same if you say x is inertial a after 5 nanosecond it is same as like a after 5 nanosecond.

Because the default delay is inertial, so whether you say a after 5 nanosecond or inertial a after 5 nanosecond does not matter. But in this case the minimum pulse width is 5 nanosecond but suppose you have a case where the pulse width required for the output appear is 3 nanosecond and propagation delay is 5 nanosecond which probably may not be true like it maybe very close to 5 nanosecond .

But take this case then you can specify you can decouple the minimum pulse width and the propagation delay. And the syntax for that is you say reject 3 nanosecond inertial a after 5 nanosecond okay. So, it means that anything below 3 nanosecond will be rejected and anything above 3 nanosecond will be delayed by 5 nanosecond also you can say that you gets say 1 after see 5 nanosecond, 0 after 8 nanosecond, 1 after 12 nanosecond.

It means that you will get a signal for first 5 nanosecond 1 next 3 nanosecond, because here we have saying the real time okay real so, for first 5 nanosecond will be 1 next 3 nanosecond it will be 0 and the next 4 nanosecond it will be 1. So, this is very useful kind of syntax to generate some reform. And particularly this is useful in test benches. So, that is about the the inertial delay.

And transport delay is the delay through a transmission line okay. Basically no pulse is rejected you know you have a long line of bus okay. And you apply a pulse irrespective of the width of the pulse. It is going to go the other end with the delay okay. But it won't be rejected, because the pulse width is some kind of less than something. And way to specify the transport delay is you say z is transport a after 5 nanosecond okay.

You say instead of inertial you say transport a after 5 nanosecond, then you get the transport delay so, maybe I will show some example the waveform and some kind of cases where the this delay modelling is use for verification. But just for today we will wind up at this part. We have told about delay modelling which is inertial delay and transport delay inertial delay a kind of model.

The delay through capacity networks are delay of a gate with the threshold which essentially means that you need some minimum pulse width at the input for the output appear due to this threshold crossing and the propagation delay. And at the as I said again at the digital gate level, **h** we have kind or gross delay we cannot be accurate. But if you take a VLSI transfer level implementation do as spies you will know the exact delay.

But this is very useful at least where such like from the spy simulation such delays are known. Then we can if knowing the device characteristics you can model using this syntax reasonably correctly, because it allows you to specify the minimum pulse width and the propagation delay. So, we have seen the syntax for that default is inertial, so we have in the simple case. When you say a after 5 nanosecond or inertial delay a after 5 nanosecond.

The propagation delay and the minimum pulse width is same when it is different user reject close and you can generate wave form by specify you know the various values using the syntax and the transport delay model the delay through a transmission line where in there is no reject like there is no minimum pulse width requirement. And the syntax for is that z you know some output get transport a after 5 nanosecond okay.

As I said in the next lecture we will some kind of an example of the syntax with the waveforms and we will also maybe how to use this delay modelling to verify some timing of a flip flop we will see that. Then we will go through some maybe the VHDL code examples in the next lectures so, that your familiar you get some good familiarity with the VHDL language. So, I stop here please revise and do not take it lightly it just because it is a language because it represent the hardware you have to get in to the habit of thinking hardware than just treat it as a language. So, I wish you all the best and thank you.