

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

Lecture-15
Sequential statements and Loops

So welcome to this lecture on VHDL in the course digital system design with PLDs and FPGAs. In the last lecture we have seen the concurrent statement, when else and the equivalent sequential statement if then else. So let us run through the slide before. We come to the today's part today. We will look at the case when sequential statement and the loops.

Both concurrent and the sequential loop, concurrent is called generate and the sequential one is called loop. You know the normal for loop so, let us run through the slide before we get it today's lecture.

(Refer Slide Time: 01:08)

The slide is titled "when - else" and is numbered 9 in the top right corner. It contains the following content:

- Syntax**

```
output_signal <= expa when cond1 else  
                    expb when cond2 else  
                    .....  
                    expx when condx else  
                    expy;
```
- General conditions**
e.g. condition1: (a = b)
condition2: (d > 3)
- Priority**
- Truth Table ?**
- Legend:
output_signal: output(s),
condi: condition in terms of inputs
expi: expression in terms of inputs

At the bottom left is the NPTEL logo, and at the bottom center is the name "Kuruvilla Varghese".

So in the last lecture the first part was the when else the syntax is 31 output signal for you know you have an expression in terms of inputs. When condition some condition again and terms of input else. Else means if this condition with not met another expression and some condition happens and so on. So this shows a priority and when you come here it is not of condition 1 and condition 2.

When you go to the next step it is not of condition 1 not of condition 2 and condition 3 and so on. So it builds up at the end there is an else which captures you know the knot of everything previously specify. And the conditions are you know general in the sense that in the condition 1 could be a equal to b then d grade the 3. The expression itself and have the inputs some expression of inputs obviously accommodates priority. And as I said from the description one might think it is not a kind of truth table. But it is truth table in a very abstract sense.

(Refer Slide Time: 02:24)

when - else 10

```


output_signal <= expa when cond1 else
    expb when cond2 else
    .....
    expx when condx else
    expy;

```


- Equations


```

output_signal =
    expa and cond1 or
    expb and cond2 and not(cond1) or
    expc and cond3 and not(cond2)
    and not(cond1) or
    .....
            
```



Kuruvilla Varghese



And this is all the equation builds up which we have seen

(Refer Slide Time: 02:28)


when - else 11

```


y <= a when (p > q) else
    b when (r = 2) else
    c;

```

p(1)	p(0)	q(1)	q(0)	r(1)	r(0)	y
0	0	0	0	0	0	c
0	0	0	0	0	1	c
0	0	0	0	1	0	b
0	0	0	0	1	1	c
0	1	0	0	x	x	a
0	1	0	1	0	0	c
0	1	0	1	0	1	c
0	1	0	1	1	0	b
0	1	0	1	1	1	c
1	1	1	1	1	1	c



Kuruvilla Varghese



And we have seen that when you write such a statement like `a when p greater than q else b when r equal to 2 else C`. Essentially are specifying a big truth table we discuss it is almost 128 rows of the truth table. So, wherever like you have `a, p, q, r, a, b, c`. As I said `abc` is not shown as separate column, but it has to be then when wherever `p` is greater than `q`. Then you have `a` where ever `p` is less than or equal to `q` and `r` is equal to 2.

Then you have `b` else everywhere else you have see so, that is what it this convey actually. This simulator as I said whenever there is an event on `a, b, c, p, q, r`. This is computed synthesis tool is not going to make a truth table as such it is going to infer the operator greater equal. And that kind of circuit will choose `a` marks where `a b c` let in okay. We will see that later how that is kind of synthesis. But that synthesis tool is going to in for operators and replace with the library codes.

(Refer Slide Time: 03:47)

when - else

12

```

y <= a when (p > q) else
  b when (r = 2) else
  c;
```

- In the code above, first condition translates to all those values of `p` and `q` for which `(p > q)`. i.e. it translates to multiple rows of the truth table. In this case, signal '`r`' is a don't care
- When it comes to second condition, it translates to all those values of `p, q` and `r` for which `p <= q` and `r = 2`. Once again, it means multiple rows of the truth table.

- For the simulator, an event on any of the signals in conditions or expressions will trigger the computation of the output signal.
- Synthesis tool may not use the truth table, if the standard operators/structures could be inferred from the code

```

prio <= "00" when (a = '1') else
  "01" when (b = '1') else
  "10" when (c = '1') else
  "11";
```

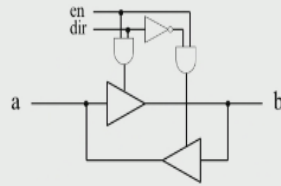
Kuruvilla Varghese

And things like that and that is what is written here and we have seen an example code of priority. I am not written the library entity, the architecture body and all that I suppose you can manage you can do that because `a, b, c` in the port you know the `a, b, c` come as a input as standard logic and `prio` come as a output which is standard logic 1 down to 0 and all that. I suggest you this and simulate in a tool if you know the tool already. Otherwise as we proceed I will show a demonstration of some tools.

(Refer Slide Time: 04:30)

when - else

13



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

```
entity transc is port
(a, b: inout std_logic_vector(7 downto 0);
en, dir: in std_logic);
end transc;
```

NPTEL



Kuruvilla Varghese



architecture dflow of transc is

begin

```
b <= a when (dir = '1' and en = '1') else
(others => 'Z');
```

```
a <= b when (dir = '0' and en = '1') else
(others => 'Z');
```

end dflow;

By which we can try this and we also have seen an example of transceiver with when else. So it uses when else construct and it also uses the it shows the usage of in out and this array assignment so on.

(Refer Slide Time: 04:50)

Sequential Statements

14

- if-then-else
- case-when

a, b, signals of *cond1*: inputs
y: output

- if-then-else – syntax 1

```
if cond1 then
y <= a;
else
y <= b;
end if;
```

- Equation

$y = a \text{ and } \text{cond1} \text{ or } b \text{ and not}(\text{cond1})$

Note: *cond1* here means the Boolean equation of the condition.

Note: Sequential statements are used in process, functions and procedures only



NPTEL



Kuruvilla Varghese



And we looked at if then else which is equivalent when else. So in the simplest form it has it is like this. You know you have if condition is 1 then y gets a else y gets b. So the equation is that a and condition 1 or b and not of condition 1. And this if then can be used only in sequential bodies like the processes function and procedure. It cannot be used in the architecture statement region, where only the concurrent statement works.

(Refer Slide Time: 05:25)

if-then-else

15

- General conditions
- Priority
- Syntax 2

```
if cond1 then
  y <= a;
elsif cond2 then
  y <= b;
elsif cond3 then
  y <= c;
else
  y <= d;
end if;
```



NPTEL

- Equations

$$y = a \text{ and } \text{cond1} \text{ or } b \text{ and } \text{cond2} \text{ and } \text{not}(\text{cond1}) \text{ or } c \text{ and } \text{cond3} \text{ and } \text{not}(\text{cond2}) \text{ and } \text{not}(\text{cond1}) \text{ or } d \text{ and } \text{not}(\text{cond3}) \text{ and } \text{not}(\text{cond2}) \text{ and } \text{not}(\text{cond1})$$

Kuruvilla Varghese



And the most generic form is just you can specify as many conditions you want. And you end with an else which captures all the of all the conditions. That is how the truth table is completed okay. So, this goes like this you know if condition warn then y gets a else if condition to y gets b and so on. So, the equation is the y is a and condition 1 as I said condition 1 itself is composed of some input signal.

When I say a and condition 1 it is symbolic definitely that has to if you write the truth table and work out product term. It expands in multiple mint terms or product terms or all of product terms in so on. So that should be kept in mind. So, this is still lot of kind of Boolean statement is little abstract. So y is a and condition 1 or b and condition 2 and not of condition 1 or c and condition 3 not of condition 1.

And not of condition 2 or d and not of condition 3 and not of condition 2 and of condition 1 that is what shown here. So, as I said like a priority encoder picture like the AN gate with the bubble increases the same thing happens here. The structure is same here also there will be an AND gate similar to that. Where the multiple conditions kind of you know gets inverted or complemented and control that part.

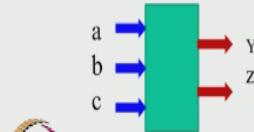
(Refer Slide Time: 07:02)

if-then-else

16

- Equivalent to when-else, but
- Multiple outputs
- Nesting

```
if cond1 then
  y <= a; z <= a and b;
elsif cond2 then
  y <= b; z <= c;
elsif cond3 then
  y <= c; z <= a;
else
  y <= d; z <= b;
end if;
```



Kuruvilla Varghese



And this is equivalent when else but that is the major difference that you can specify multiple outputs you can implement nesting and we have seen an example where there are 3 inputs abc and you have 2 outputs y and z. Then you could write like if condition 1 then y gets a. Z gets something else, else if condition 2 y gets you know you can specify y and z for all these conditions. But in real life it may happen that it is not so need. You know the z may not you cannot assign z for all these condition, may be a subset of conditions only work for z and so on. So in such a case you can use the nesting.

(Refer Slide Time: 07:57)

if-then-else

17

- More complex behaviour/structure can be specified by nesting. E.g. if there are multiple outputs and we may not be able to specify all outputs for same conditions

- Equations

$y = a \text{ and } \text{cond1} \text{ and } \text{cond2} \text{ or } \dots$

```
if cond1 then
  if cond2 then
    y <= a;
  elsif
    .....
  end if;
elsif
  .....
end if;
```



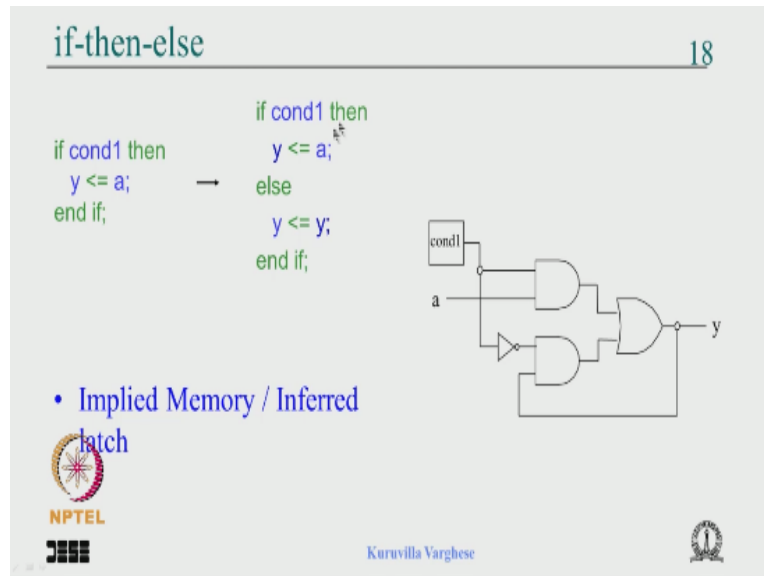
Kuruvilla Varghese



You know you say if condition 1 then you can write this z straight away here that it condition 1 applicable to z. But condition like y has a restricted condition then that can be specified you

know that can be split and specified in condition 2. So, if condition 2 then y gets a else if I gets something else. So as for as y is concern the equation becomes y is a and condition 2 and condition 1. And or say if it is y is equal to b you say b and not of condition 2 and condition 1. So, like that it goes so, you can work it out.

(Refer Slide Time: 08:39)



Then we looked at something which is not fully specify suppose in VHDL if you write like this which is quit valued there is nothing kind of no error with this syntax. If condition 1 then y gets a end if, that means we are not specifying the else condition. Then it means as for as VHDL is concern VHDL treat this statement as like this. You say if condition 1 then y gets a l y okay.

Now, mind you when you write the real code do not write like this okay write like this okay that is, what is these standard. So, if do not never write like this always if you want a latch you write like that. And this as I said is a latch where the if this condition is not MAT. The output is feedback you know you have the output going back as a input. You know that is that is what it achieve is kind of code.

So it is called Implied Memory or Inferred latch Implied Memory because, this code implies memory through feedback or the synthesis tool will Inferred latch from this code that is why called Inferred latch. And the picture is shown here at 2-1 marks the select line is control by the

condition 1. So if it is select line is 1 then a goes to the output, if select line is 0 then it comes back.

(Refer Slide Time: 10:20)

Implied Memory / Inferred latch

19

- Concurrent equivalents
 - with en select
 - y <= a when '1';
 - with en select
 - y <= a when '1',
 - unaffected when others;
 - y <= a when en = '1';
 - y <= a when en = '1' else
 - unaffected;
- Concurrent: unaffected
- Sequential: null

- Implied Memory / Inferred latch is useful in specifying the behaviour of latches and flip-flops or registers
- But, unintentional implied latches can happen
 - e.g. when multiple outputs are specified for each conditions, a missing output can result in implied latch on that output. This is all the more possible when nested loops are not balanced, as it is difficult to detect
- This is one of the common errors that inexperienced designer commit in VHDL coding

Kuruvilla Varghese

That is how the latch is implemented you could have concurrent statement which kind of with this same effect if you miss the else part. So which select you can say with enable select y gets a when 1 and you are not specifying anything else. You are not specifying you know b when 0 or something like that or you have it means y gets a when 1. And y when 0 that is the meaning of it.

And similarly you can say with enable select y gets a when 1 you can say unaffected when others okay for in concurrent statement. When you say unaffected it remembers previous value as regard to the when else. This is the statement which achieves a latch y gets a when enable is 1. And you do not specify else or if you specify else.

You say unaffected like the unaffected in the concurrent statement in sequentially have a null. So it again means that here you could say if condition 1 then y gets a else y get null. You get the same kind of inferred latch by the synthesis tool. So, implied latch is useful when we come to the may be this end of this class or the next lecture. We will handle the sequential circuit or the flip-flops and registers and so on.

There this quite useful because, this kind of coding is used to specify the behaviour of memory you know the behaviour of latches and flip-flops and registers specifically the memory part of it. But the problem with this kind of code is that unintentional latches can happen. That happens when there is a lot of nested loops and if something is missing particularly if there is not symmetric. It is very difficult to work out. And as I said this can you know become an unintended kind of latch. But that trouble with that is that n is one of the most one of the common errors.

(Refer Slide Time: 12:48)

Implied Memory / Inferred latch 20

- It is difficult to expose this error in simulation, as just verifying all conditions would not be sufficient.
- Suppose, one output was missing in condition 3, and the previous condition simulated has the same value expected of this output in condition 3, then output will be correct.
- If the designer has inadvertently missed specifying an output, then working out the condition for exposing such an error would be

NPTEL logo, Kuruvilla Varghese logo, and a small circular logo.

I see people make in VHDL but the important thing to remember is that if you make that error to debug is difficult. Because a testing for all inputs want expose it because, it for a particular condition if it is a latch. You to expose it you it you have to have at the previous condition previous tested condition which should have an opposite output then only that will be a expose here. But if the previous condition as the expected output as.

Now then there is no way to detect this error so, this should be kept in mind and if the designer as kind of made a mistake the here show see will not aware of it. And there no question of working out condition that if not aware of does not arise and, this can be waste lot of times. So I just be very careful when you write multiple output when, you do copy paste when you do nested kind of if nested case and so on.

(Refer Slide Time: 14:01)

- Syntax

```
case sel_signal is
  when value1 =>
    (statements)
  when value2 =>
    (statements)
  .....
  when valuex =>
    (statements)
  when others =>
    (statements)
```



NPTEL

Kuruvilla Varghese



- All mutually exclusive values of sel_signal need to be specified
- No priority, Truth table
- Equivalent to with-select, but
- Multiple Outputs
- Nesting

You have to be careful okay not that issue you should not do copy paste but, there is a danger okay. So let us come to the next sequential statement which is case when, and this is a syntax please have a look at it. So, this is identical to the which select concurrent statement only the syntax is different. So you have case some input signal is now for you say when value 1.

So you can write statements here okay sequential statement assignments here when value 2 assignment, when value last value assignment, when others you write assignment. So these values are the mutually exclusive values of the select signal. So it is exactly like you know which select and you have specifying the truth table. And the statement like n b is normally the output assignment it can be numerical value.

It can be expression y the input is there okay. So it is like when else but, as in if then you know comparison with the when else sorry this is my mistake this is apologise this is equivalent to with-select not when else because, that is what we you know there no priority and we specify the specify the truth table. So, this is equivalent with select so, like the comparison between if then else and when else.

The same thing applies here you can specify multiple output that means in the statement part. You can have x gets something, y gets something, z gets something and all that. And you can do

the nesting you can nest a case within the case that means, you could say case select signal is when value 1 under that may be. You might say case some other signal is then.

You know you could make you know nesting or you could say when value 1 if some condition so, you have the freedom give makes case. And the if and that is one of the kind of useful structure like you have a toy. The outer level case and for each value, you have you know the if statement which very useful structure, we will see that when we go ahead.



(Refer Slide Time: 16:41)

Case-when

22

```
case sel is
  when val1 =>
    x <= a;
    y <= b;
  when val2 =>
    x <= c;
    y <= d;
end case;
```

- Equations
x = a and (decode of sel = val1) or
c and (decode of sel = val2)
y = b and (decode of sel = val1) or
d and (decode of sel = val2)
- Implied memory, if any output is not specified in all choices of selection signal.



Kuvilla Varghese

Maybe we will see an example of the case when now once again I will show only the real code. You could know write all the other parts like entity, architecture body, library and all that. And that signal declaration all that should be done properly so, here I am writing two outputs. You know the x and y using the case so, the syntax is case select is in another input is when value 1. When the select is equal to value 1 x get a, y gets b.

Then you say when value 2 okay now, I should have said, when a others because, I need to specify all the conditions so, here instead of value 2 it should be when others okay which should capture the value 2 also okay. So but assume that there is a when others at the end. So that everything goes well with this kind of code. So, when others you can imagine there is 1, when others here so, when value 2 x gets c, y gets d.

So if you look at the equation like with select you have x is a and decode of value 1 or a and decode of value 1 means you know you have so, select line is 2 bits and it is 0 0 then it is s1 bar

and \bar{s}_0 . And so that is what or c and decode of value 2 know that is it. Similarly you know you can y is b and decode of value 1 or d and decode of value 2 that is what is written here.

So that is how the equation or the circuit is kind of you know Inferred and even in this case if you have a kind of case statement. There are multiple choices okay and if you miss something in 1 please. Then it can create an implied latch it is not only in if then everywhere, whether it is concurrent statement or sequential statement. Implied latch can happen you should be careful once again as I said, when the structure is complex.

(Refer Slide Time: 19:02)

Case-when Nesting 23



- “case ... when ...” can be nested with “if .. then ..” to specify complex structure / behavior.
- Equation

```

case sel is
  when val1 =>
    if cond2 then
      y <= a;
    elsif
      .....
    end if;
  when val2 =>
    ...
end case;

```

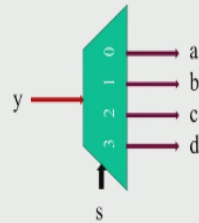
$$y = a \text{ and decode of } (sel = val1) \text{ and } cond2 \text{ or } \dots$$


Kuruvilla Varghese


Then you have to be more careful okay, you could as I said the next the case when with the case when or if then. It can be the the case when within if or if within case when and so on. So I am showing in useful example case select is when value1. If condition 2 then y gets a else you know you have else if which goes on so, if you look at this part only.

Then the equation comes y is a and condition2 or the decode of select is equal to value 1, then you say sorry y is a and condition 2 and at decode of select is value 1 or if there is a y equal to b then if it is else then y equal to b not of condition 1. And decode of select is equal to value 1 and so on it goes so, you can make out you know you can easily work out the equations and the circuit from the structure. Because by now you should be through with what I am saying

(Refer Slide Time: 20:16)



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

entity dmux1t4 is
  port (y: in std_logic_vector(3 downto 0);
        s: in std_logic_vector(1 downto 0);
        a, b, c, d: out std_logic_vector(3
          downto 0));
end dmux1t4;
```

```
architecture arch_dmux1t4 of dmux1t4 is
begin
```



Kuruvilla Varghese



And let us see an example because, you know, which select so, it is identical. So let us look at this case of a 1 to 4 de-multiplexer and each input and output all are 4 bits okay. It could be any number of bits and since you have 4 selection 4 outputs you have 2 bit select line, you know that is what is s is 2 bit, y is 4 bit a b c d are 4 bits. So that is a de-multiplexer so, let us look at the code.

So, this is the library declaration the package and this the entity we dmux 1 to 4 1t4 is then you say port. The y is in standard logic vector 3 down to 0 because 4 bit for 4 outputs, you have two select line. So, s is input again standard logic vector 1 down to 0. And a b c d are output which each of which is standard logic vector 3 down to 0. So that completes a entity the architecture you given name say some name is given of this entity.

Then you do not have anything to declare quite very simple input output 2 inputs 4 outputs that is all. So, you say begin now we are going to write the case statement for this and you know that remember the case can be used only in sequential body. So, we are going to write a process and within which we are going to write the case okay.

Now, the next question is that what should be in the sensitivity list okay. So, we said that when you write when you implement something with the process, you have to write necessarily the input in the sensitivity less. So, because any change in the input should trigger a computation as

well as simulator is concern. So, we write s and y in the sensitivity less and we have 4 outputs so, when we write a case statement for each choice. We have to specify all the outputs you know a b c d as to be assign for each value of you know the s 00, 01, 10 and 11. So that is what we are going to do.



(Refer Slide Time: 22:54)

Demultiplexer

25

```

process (s, y)
begin
  case s is
    when "00" =>
      a <= y; b <= "0000"; c <= "0000";
      d <= "0000";
    when "01" =>
      b <= y; a <= "0000"; c <= "0000";
      d <= "0000";
    when "10" =>
      c <= y; a <= "0000"; b <= "0000";
      d <= "0000";
    when "11" =>
      d <= y; a <= "0000"; b <= "0000";
      c <= "0000";
    when others =>
      a <= "0000"; b <= "0000"; c <= "0000";
      d <= "0000";
  end case;
end process;
end arch_dmux1t4;
  
```


Kurusilla Varghese


So, that is shows the process sensitivity list as 2 inputs s y because any change in any of them should the simulator should you know compute and you say begin and you say case s is the input select input is say when 00 when it is 0. You know that a y will go to a when it is 1 y goes to b when it is 2 y goes to c 3 y goes to d. So that part is written here when 00 a gets y but, all other thing has to be specified otherwise there will be latches.

So, we specify b as 0, c as 0, d 0, when 01, we assign b y rest all is 0. When 10 c gets y rest all is 0, when 11 d gets y rest all is 0, when others everything is 0 really for synthesis it does not matter. But for simulator if some other condition occurs by mistake, then this comes up and as I said that is easy to debug so, this is the code for the de-multiplexer 1 to 4 de-multiplexer using the case statement.

Now, the question is that here your force to write all the outputs in all the you know all the choices of this s value. Now the question is can we say because, if there are too many outputs then that is going to be a problem, and you know if something is miss, then you can create

implied latches. So, is there way out that is what we are going to see and also as I said there is no need to write, when others separately. You could combine with 11 so, here you could write when others and you can avoid is kind of statement it does not matter. So let us see little more concise version of it.



(Refer Slide Time: 25:01)

Demultiplexer

26

```

process (s, y)
begin
  a <= "0000"; b <= "0000";
  c <= "0000"; d <= "0000";
  case s is
    when "00" =>
      a <= y;
    when "01" =>
      b <= y;
    when "10" =>
      c <= y;
    when "11" =>
      d <= y;
    when others =>
      a <= "0000"; b <= "0000";
      c <= "0000"; d <= "0000";
  end case;
end process;
end arch_dmux1t4;
  
```


Kurusivilla Varghese


Here you see that the same process with the s and y in the sensitivity less and bit say begin at the beginning before the case abcd is made 0 okay. Now in the case statement, you say when 00 a is y okay and we are not specifying bcd because, bcd is 0 here okay. Similarly when 01 b gets y rest all is anyway specified 0. Because when the simulator computes it goes you know that in a process statements are executed from the top to bottom once.

You know only once so this works because, you know the when if there is a event on s that means if the select line changes. The simulator starts computing and initially abcd is assigned because, the say simulation time is 100 nanosecond abcd is assigned kind of all 0s. And suppose s is 00 then a gets y and now since it is at the same simulation time we had an assignment 100 + delta a 0. But there will be replace by this a gets y.

So it works properly and mind you all this is we are assuming that this is for the simulator okay. This all this game of delta cycle everything is for simulator but, for synthesis tool definitely it

looks at this all statement. So it looks at the case statement and understand that it is a kind of structure comes out of it. And you know it is basically a multiplex de-multiplexer.

Because depending on the select line the output various different output get the same input. So that is inferred as a de-multiplexer and also it has to take is initialisation part of it okay. So this works perfectly fine but, you should take liberty with the synthesis many a times it is dangerous to assumes something about delta cycle. And write all kinds of code and expect that will happen.

This is a very kind of clear case that it works but, sometime in a complex case. If you assume some delta cycle initialisation things can go wrong. So you have to be very careful with the coding for synthesis tool because, that perfectly okay for simulation. But for synthesis you have to keep these things in mind. So, we have come to like we have completed the sequential statement major.

You know construct the last one we have seen was a case. When identical to which select but, it support multiple output, it support nesting and if any output is missing, it will create implied latch and we have seen an example of case 1 with 1 to 4 de-multiplexer okay. So, what is remaining is the loops the concurrent statement and the sequential statement. So let us look at the loop part of the VHDL. So, let us move on to the slide.

(Refer Slide Time: 28:44)

The slide is titled "Loops" in a green serif font, with the number "27" in the top right corner. It contains a bulleted list in blue text:

- Concurrent: generate
- Sequential: loop
- Generate
 - Equations
 - Component Instantiations

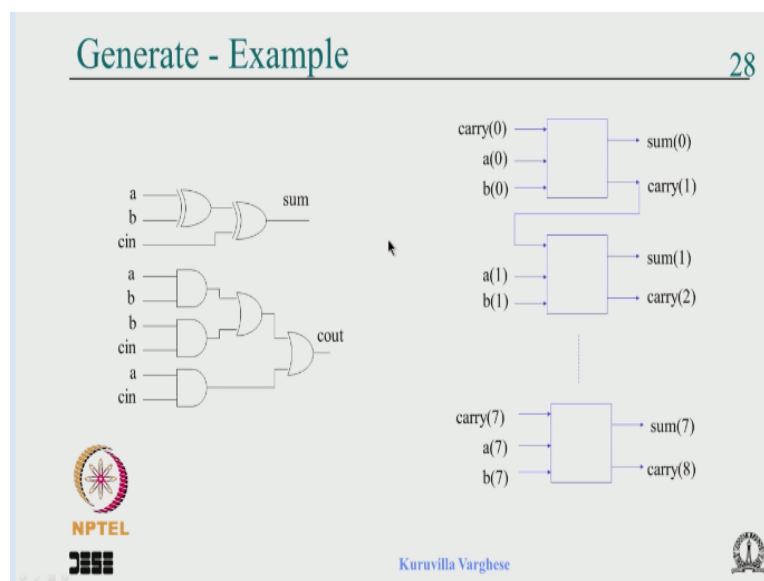
At the bottom left, there is a circular logo with a star-like pattern and the text "NPTEL" below it. At the bottom center, there is a small logo consisting of three squares. At the bottom right, the name "Kuruvilla Varghese" is written in a small blue font, next to a small circular logo.

So, there are 2 loops 1 is concurrent which say it is like it is called generate and the sequential which is called loop itself. So, the syntax will be for some indexing generate and, for sequential again for something indexing like I is equal to 0 to something loop okay and, the generate can work with equation, it can work with component instantiation. That means you can write an equation, Boolean equation and you can put it in loop.

Then you will get kind of repeated pattern of what is return. Of course you have to write it properly, so if you have a regular modular structure interconnected in a decent you know normal symmetric way you can use the generate to you know very concisely express that kind of structure very, efficiently.

Similarly you can even for component instantiation when blocks are interconnected in a symmetrical way in a regular way. You could use the generate loop to come out with this structure. So, let us move on, let us take an example.

(Refer Slide Time: 30:10)



I am taking an example of a ripple adder, so you know that ripple adder is composed a full order. So, we are going to work out a kind of 8 bit ripple adder. So, you have the ripple order is made of full order. So, the full order has 2 part, the sum generation and the carry generation. If you remember the sum equation is $a \oplus b \oplus c$ okay.

That is a sum equation and carry is 1, when more than 2 or more inputs are high okay active. Then the carry is the 1, so it is nothing but ab or bc or ac, that is what is shown here. So, now we take this as a module where the input is a, b and c in carry in. And the output are sum and carry out. Now if you write like you have a 1 full order here, with a0, b0 and carry0 as input.

Then you get some 0 as output, the output carry output is now the output out of the first stage to the input to the second stage, it is called carry 1. And the next stage you have a1b1 as input the real inputs, the carry from the previous stage, carry1 is input. Then it generate some 1 and carry 2 which is use in the next stage and you go all the way up to the sum 7 and carry 8, sometime this is called carry out okay.

(Refer Slide Time: 31:51)

Generate
29

- Equations


```

for i in 0 to 7 generate
  sum(i) <= a(i) xor b(i) xor carry(i);
  carry(i+1) <= (a(i) and b(i)) or ((a(i) or b(i)) and carry(i));
end generate;
          
```
- Component Instantiations


```

for i in 0 to 7 generate
  u1: fulladd port map (carry(i), a(i), b(i),
                        sum(i), carry(i+1));
end generate;
          
```

Kuruvilla Varghese

So, let us see how we can write and generate kind of loop for this and you say the syntax is for i in 0 to 7 generate, i is an indexing variable which is use temporarily for you know to expanding this. So for i in 0 to 7, you could even say 7 down to 0 it does not matter. Only thing is that indexing start from 7, so it should work normally it does not matter whether you write 0 to 7 or 7 down to 0.

So, you look at it, so if the index is 0, then some of 0 is nothing but a of 0 xor b of 0 xor carry 0. So, that is what is shown here, some 0 is a0, b0 and carry0. And you say carry i + 1, that carry 0+1 carry 1 is nothing but a0 and b0, ab0, ac you know a0 and carry 0 and b0 and, carry0 that is

a meaning of it. Now, when we go to the next iteration, when the index become 1. Then you say you get the sum 1 as a_1b_1 , carry 1 xor of that.

When you come to the carry 2 output, you see it is composed of a_1b_1 and the carry1 which comes from the previous output previous stages output. So that is how this is you know this happens so, automatically the carry1 goes the output of the first stage goes as a input the second stage the output of the carry output of the second stage goes as input the second third stage and so on okay.

So that is what is shown here, now you could do this by writing component also that may that means that you write a full order you know the component by writing the library, entity and architecture. The architectures statement you just write 1. You know 1 full adder not 8 of them like this 1 full adder. So which says some is nothing but, $a \text{ xor } b \text{ xor } c$ c in and c out is nothing but $ab \text{ bc } ac$ like that you can write.

Then you write a top level component which is composed of interconnection of all these, you know you have to show the necklace which we have seen then and that can be worked out by this kind of generate loop. So we have a full adder with input carry a b output is some and the carry out so, we write a the full adder port map 0 to 7 generate carry i a_i , b_i . These are input some i and carry $i + 1$ is output so, exactly same thing will happen.

You get a_0 b_0 carry0 as input some 0 come as the output and you see the carry 1 output is used in the next statement. When in the as an input so, it works perfectly. So you could write generate loops either with the equation or with the component instantiation both works perfectly for a once again for a small example like a ripple order. You do not do a component instantiation it is enough if you write like this.

You can write structural coding like this in a complex case. So just for completion sake you know clarifying I have written this component instantiation using the generate loop.

(Refer Slide Time: 35:50)

Conditional Loops

30

- if ... Generate (Concurrent statement, no else /elsif)

```
loop_label: if (condition / expression)
generate
-----
-----
end generate;
```

- If the condition is true, then the generate is done
- Useful to generate irregular structures, i.e. Conditional on loop index (e.g. $i = 2$) different structures can be generated



Kuruvilla Varghese



That is what is shown here. You could in principle have the loops conditionally that means you can here. We are it is a regular structure you know the 8 full order modules are interconnected using this but, it may happen that very regular. You know something happens from 0 to 5 something else happen for 6 and 7 and so on.

(Refer Slide Time: 36:17)

Conditional Loops

30

- if ... Generate (Concurrent statement, no else /elsif)

```
loop_label: if (condition / expression)
generate
-----
-----
end generate;
```

- If the condition is true, then the generate is done
- Useful to generate irregular structures, i.e. Conditional on loop index (e.g. $i = 2$) different structures can be generated



Kuruvilla Varghese



So, in such cases you could write some conditional generate, you say give a label and you say if condition or some expression of the condition then you say generate and you write various statement and you write n generate that means unless this condition is not MAT this will not be generated. So, this can be used to like a generate some kind of asymmetrical regular structure. Because you know where things go kind of does not fit in.

So, you specify that condition then that either do something else, you can say if condition1 then generate you know then only conditionally that will be generated.

(Refer Slide Time: 37:00)

Sequential: For ... Loop

31

```

if (rst = '1') then
  for i in 0 to 7 loop
    fifo(i) <= (others => '0');
  end loop;
else
  .....

```

- Syntax


```

loop_label: while expression loop
  .....
end loop;

```
- Example


```

tbloop: while not endfile(vector_file)
loop
  .....

```

- The code decides whether the loop is synthesizable. Above loop is easily synthesizable.

Kuruvilla Varghese

Similarly when we come to the loop for the sequential loop, the syntax is like this 4 i in some range loop then write some statement n loop okay. So, this is the kind of 4 loop in VHDL look at the code I have written. I have written if reset is 1 then i in 0 to 7 loop FIFO is other 0 n loop. You know it continues okay may be else is there so, many a times people say the loops 4 loops are not synthesizable.

Once again this is a kind of very general statement which is not true many a times, this code is synthesizable you know. It only say that there is a reset input when it is high all the FIFOs reset okay. Now it is very simple you have flip-flops within the FIFO, you tight together and all the reset and you connected to the reset input. Then it works perfectly fine, so, this is synthesizable but when somebody say the loop is not synthesizable one.

Once again as in the case variable it means that you take a algorithm which is specified using a for loop straight away translate it into VHDL. Then if it give it to as synthesis tool you may not you know sensible circuit is, what is meaning when somebody say the 4 loops are not

synthesizable many a times people say this as kind of just as a kind of thumb rule or what they have learnt from somebody else without much thinking.

So, you should not get kind of worried by such a very sweeping general statements okay. So, there is an kind of conditional loop which say that say, you have a label loop label then you say the syntax is while expression loop n loop that while something is true loop is loop again this is useful in generating some structures, where which is not symmetric. You can say while i is greater than 2 then do something okay.

So, very specific okay else you know the other things can be specified before and 1 other the while loop usage is just 1 in test benches. You say while not n file, n file is a kind of procedure and argument is vector file that means that it say whether we have reached the end of the file. So, which say while not n file the vector file, then do the loop okay normally you have test vectors written in a file line by line.



This say if the end of the file is not reached then you do loop and you do the test bench, you know so, this is another useful thing for the while loop. Though you can use it for you know with the normal loops sequential loops this while. You know conditional loop.

(Refer Slide Time: 40:27)

Loop Control

32

<ul style="list-style-type: none">Exit <pre>exit; exit [loop_label]; exit [loop_label] when condition;</pre> <pre>if condition then exit; end if;</pre>	<ul style="list-style-type: none">Next <pre>next; next [loop_label]; next [loop_label] when condition;</pre> <pre>if condition then next; end if;</pre>
---	---



Kuvempu University

You can use that and like in c language you can have some control over the loop by exit and next. You know the exit if you write somewhere you have a big loop and you write a exit it means that you just come to that point and will come out okay. A kind of blankets it is hardly useful because, you have a loop which goes from top to bottom and if you are putting exit write in the middle, the remaining part is not anywhere computed.

So, just writing exit may not be a big idea may not be useful you could say exit loop label whichever kind of here if you say exit somewhere here. Then you say exit to loop so, that it exists because, that is useful when you have nested loops and you want to exit the whole loops you know the main loop you want to exit. Then you can give the proper level then will exit, otherwise if there are nested loop it can confuse.

Whether the immediate loop or the outer loop need to be exited another useful thing is that, you can say exit loop label when some condition is met. This is maybe good to kind of generate kind of irregular structures when you have a loop you say exit loop label when a particular condition happens okay. And that can be rewritten in this form also like you say if condition then exit end if so, where ever you have written exit you know loop label.

When condition 1 you can say if condition 1 then exit end if exit loop label if you want to write you can write. Then next is again the that next is that you skip to the next index. When you are looping some are the middle if you say next then at that point it does not continues with the rest of the loop. It just skip that index at that point and goes back to the beginning and start from the top okay, so that also is useful.

But a blanket and qualified next may not be a great use. So you can say exit next loop label whenever there is nesting, you can clearly specify which loop you are you know exiting or you can say next loop label when some condition is met, again useful. So, this can be rewritten or rephrased thus if condition same thing when instead of when condition if condition. Then do the next that means then you skip it.

So, that shows the loop control exit and the next which can be useful. But then you have to think whether it makes sense as far as circuit is concerned if you do not think about the hardware and you keep writing some blanket statement the proper circuit you may not get that should be kept in mind. So, you should not think which simulation it is okay with synthesis you need to be careful.

And if you can think of a very simple way of synthesising the circuit then the tool might do that, but yourself not sure what kind of circuit this will you know generate or this functionality can be brought in there is a less chance that the synthesis tool will do some magic and give you a circuit with the behaviour that you have written in the VHDL using the exit or next and so on okay that should be kept in mind.

So, that completes the statements, so the last part was the loops generate loop which can be used for equation and component instantiation and we have seen an example of ripple adder which is composed of full adder and we shown it two ways. One way is using the equation and one way is using the component in instantiation as I said for simple a case like this component instantiation is not very attractive.

But definitely the equation looks data not to discourage you if you want to write component instantiation. You can write then we have seen the loop control the exit and the next and we have seen basically 4 loops the sequential loops we had this loop is synthesizable.

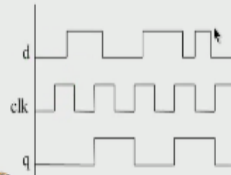
(Refer Slide Time: 45:12)

Sequential Circuits: D Flip Flop

33



- Flip Flops behavior is modeled in VHDL, not their equivalent circuit.
- Behavior: Up on the active edge of the clock the input is transferred to the output and is held (Memory) until the next active clock edge.



Kurusvillla Varghese



So, that is what we have seen the last part of the lecture so, let us so, like we have come as for as see combination circuit is concern. We have kind of completed the concurrent and sequential statement, you could start writing VHDL code for various combination circuit. You know you can I suggest you try write simple things write multiplexer, de-multiplexer. Now when you are adders when you like say I will give some 2, 3 examples. So, you write an multiplexer de-multiplexer.

Then you write a adder then you write a priority encoder so, for multiplexer and de-multiplexer use when else and case when for priority encoder use if then and when else and for adder use the component instantiation and the equations you use generate loop and so on. So, this can be practice so, that you get practice in modelling the combination circuit. So let us look at the sequential circuit or the flip-flops and registers.

How this can be modelled in VHDL or coded in VHDL which will work for synthesis and simulation and so on. So, let us come to take the simplest example which is that D flip-flop. So, first thing note that you know that within a D flip-flop there are 2 latches masses slave latch. And there are different ways of implementing it one popular way of implementing is it is using 2 latches.

One is a master one is a slave if time permits we can have a look at that structure later but, in VHDL we are not going to write the flip-flop in terms of the circuit that implements of flip-flop. That means we are not going to write a structural code where you know the using the gates what is inside, you know that is not on what we are going to straight in VHDL is that.

Somehow the behaviour of the hertz to get D flip-flop is captured in a meaning full VHDL statement there is what how it is model and if you look at the behaviour of the D hertz to get D flip-flop. You see that there is a clock and on the active edge let us take it as a positive edge trigger because, that no bubble here. So, upon the positive edge whatever is at the input is transferred to the output and it is memorise okay.

That is the behaviour of the hertz to get D flip-flop okay. So, that is what is happening here when the clock edge come the 0 is transferred here and it is memorise still the clock edge and the next clock edge it is 1 and it is memorise still the next clock edge. And that is what is seen shown here and here it is 0 again the 0 up till here. Then it is 1 so, 1 comes here but, then it is remember till the next clock edge and again it is 0 so, it becomes 0.

So, that is the behaviour so, let us think how we can write this in VHDL. So, we have to say first of all you see that the input goes to output only on the clock edge. That means the whole processes is sensitive to clock so, that say that you can write a process with clock in the sensitivity less okay. So, whenever there is an event and we are talking about simulator now. Whenever there is an event on the clock that process gets computed okay.

Now the question next question is that fine if you write the clock in the sensitivity less, if there is an event on the clock you know that gets computed but, then event on the clock can be the positive edge or the negative edge. So, how do we specify the positive clock edge okay. So, there is a trick okay what we do is that in the process we next statement after the begin. We write is that if clock is equal to 1 okay. So that gives.

You know the clock is 1 only in the positive edge okay because, there is an active edge then if clock is 1 that means it is the positive clock edge that is how it is written. And we write you know then I will show the code.



(Refer Slide Time: 50:17)

Flip Flop - Simulator

34

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

- 'clk' in the sensitivity list computes the process on both the edges of clocks.
- The clause (clk = '1') selects the positive edge of the clock.
- The Implied memory / Inferred latch takes care of memory



So, this is the code which says that process clock so, whenever there is an event on the clock. This process will be computed, you say begin if clock is equal to 1. So, this should be combine the event on the clock event there is an event on the clock and clock is 1 it is a positive edge. Then q gets d then if not if you want it to memorise are the latch the output. So, you say do not say do not say else. You say end if that means if this condition is not met.

You just memorise it and you say end process so, that is a perfect code for the edge to get flip-flop as far as simulation is concerned okay. That is what is written here even under clock will make the computation happen. And if clock is equal to 1 will make it positive edge detect the positive edge. Q gets d end if and there no else end if so, that will memorise it and that gives you flip-flop VHDL code as far as simulator is concerned.

Now the question is that we have told that the sensitivity list is a business for the simulator because, in real time it you know it simulates the behaviour okay it has simulated so, it has worry all about the event happening concurrency and things like that. But for the synthesis tool it looks

at the code what is written and if you look at the code, what it is written it say that if clock is equal to 1 q gets d.

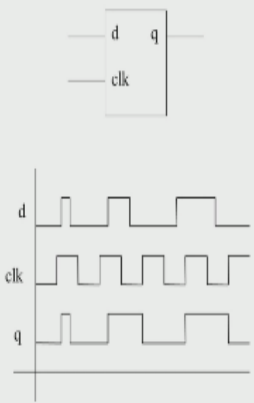
That means as long as clock is remaining high q will reflect the changes in d and when the clock goes low because, of this end if it will remember okay. So, if given to a synthesis tool this particular code it say that as long as clock is 1 q gets d. And when the clock goes low it will remembers and mind you that is nothing but, a transparent latch. So, for if you write such a code given to the simulator it is a flip-flop given to synthesis.

(Refer Slide Time: 52:43)

Flip Flop - Simulator

35

- But synthesis tools ignore sensitivity list, hence the above code would mean a (transparent) latch, as the events on 'clk' is not considered. Whenever 'clk' is high 'q' gets 'd' and on the negative edge the last value of 'q' is held. (Memory)



The diagram shows a block representing a transparent latch with inputs 'd' and 'clk', and output 'q'. Below it is a timing diagram with three waveforms: 'd', 'clk', and 'q'. The 'd' signal is a square wave. The 'clk' signal is a square wave. The 'q' signal follows 'd' whenever 'clk' is high, and holds its previous value when 'clk' is low.

NPTEL

Kuruvilla Varghese

It is a latch and the latch behaviour we know that when and that is what is written here okay a because, synthesis tool ignore the sensitivity less. The above code would mean a transparent latch the event on clock and all is not look by the synthesis tool whenever clock is high you gets d and on the negative edge it is remember. So that is what is shown here as long as clock is high the q reflects the d and at this edge.

Whatever is the value is remember till it is again enable and when it is enable the change in d is reflected and at this edge it is remembered till it is enable. So, that is what is happening here so, the question is that given to it to a synthesis tool it becomes a latch. And simulator treat is flip-flop now the question is that how do we make a proper code for a flip-flop as for as synthesis tool is concern. Now this gives your clue.

Because, here the simulator for the simulator it works properly because, the simulator start when there is an event on the clock okay. And this clock is equal to 1 will make a positive edge to get. So, it is enough as for as the synthesis tool is concern this effect is captured that means that we have to say that if there is an event on the clock and clock is equal to 1. Then q gets d if you can write that then synthesis tool will treat. This as a flip-flop and that is what is exactly done.


(Refer Slide Time: 54:31)

FF – Synthesis, Simulation


36

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

- `clk'event` is a predefined attribute which is true whenever there is an event on the signal 'clk'. The statement `clk'event and clk = '1'`, would mean the positive edge of clock. Statement `clk'event` would be redundant for simulation.
- The statement `clk'event and clk = '1'`, would also be true when `clk` transits from 'U' to '1' (`std_logic`), this could occur at the beginning of simulation.



NPTEL



Kuruvilla Varghese

By this code which say process clock you say begin and you write instead of clock is equal to 1. You write if clock that is a clock tick this is equal to tick event and clock is equal to 1. Then q gets d it this means any event on the clock anything any change on the clock for as this is sufficient. Because we are checking some event and clock is equal to 1, then it is a positive edge may be some other things can happen. I will say in a moment q gets d end if.

But for the simulator you know that whenever there is an event on the clock it is a starts computing when it comes here there is a repeat there is something redundant here. This start whenever there is an event again it is a clock tick given. So, these are redundant statement as for simulator is concern, but does not concern, still it is a reputation. But it does not harm so, that is what is so, this is a code.

The code for flip-flop which works the same for the I mean you know which works as a edge to get flip-flop for simulator and synthesis tool. But mind you there is 1 difference though suppose the beginning of simulation the U like the output was U and somehow you know the output goes from the say the clock was flagged as U. And clock started coming as 1 then there is an transit from U to 1.

And this can fast trigger as for as simulation is concern like when it simulate there is a clock tick event which is U to 1. And clock is 1 and that is true so, a U to 1 transition can trigger a computation as for as simulation is concern. So, there could be some false trigger, when you simulate it not a very good idea actually because, then the simulator behaviour and the synthesis behaviour will be different.

So, at the beginning whatever hardware you really synthesis will work different from if simulated either you should you know this fact and take care in the simulation or you should make somehow the simulation and synthesis you know behaviour same may be will discuss in this in the appropriate time. So, I think we can we will it is a kind of the time is coming to a close.

So, we will stop here we will continue with the VHDL modelling for the sequential circuit like flip-flops, latches registers and so on with the next class. So what we have seen is the that in VHDL the latches flip-flops are model by the behaviour. And we have come out with the kind of reasonable code for describing behaviour, and one very important thing mind you that we have put some sensitive syntax for that behaviour.

But you know that we are human and we are reasoning that, but for a tool this is just a template okay. It means wherever there is a clock tick event clock is equal to 1, q gets d end if that is treated as a as a latch. It is not by understanding the meaning of that so, this is taken as template. So, you should not kind of take too much liberty with it, you know so maybe there is a dum synthesis tool which will work.

If you write tactic event and clock is equal to 1 but, it may not work if you write clock is equal to 1 and clock tick event okay. So, do not take too much liberty like we have put some sensible description for the behaviour that is the only the meaning it is we have looked at what works for simulation that is in work for synthesis tool then we have added some extra redundant statement. So, when you get a code which works.

So, for both simulator and synthesis so, with that I stop here now we have covered quite a bit please revise please look at this sequential modelling try to understand it. We will built on it so, please revise it I wish you all the best and thank you.