Reference Designer

Necessity of parameter

We will directly present an example of a simple adder to understand the necessity of `parameter`

```
1. module adder(a,b,sum);
2.    parameter N = 4;
3.
4.    input [N-1:0] a;
5.    input [N-1:0] b;
6.    output [N-1:0] sum;
7.
8.    assign sum = a + b;
9. endmodule
```

There is nothing fancy about the code above. It is not meant to be real life adder and it does not has carry in or the carry out. However, it explains the concept of the parameter. By changing the line

```
parameter N = 4;
```

to

```
parameter N = 8;
```

We can chage the 4 bit adder to an 8 bit adder. This provides a huge benefit where we do not have to make changes at several places. This enhances the code reusability. Just by changing one line of code you are able to change and reuse the design.

We will now write an actual full adder, one with parameter.

```
1. /*
2. Full Adder Module with Parameter
3. Written by referencedesigner.com
4. */
5. module  fulladder (in1, in2, cin, sum, cout);
6. parameter  N  =  4;
7. input wire  [N-1:0]  in1 ,  in2 ;
8. input wire cin;
9. output wire  [N-1:0] sum;
10. output wire  cout  ;
11.
12. wire  [N:0] tempsum  ;
13.
14. assign  tempsum =  {1'b0, in1} +  {1'b0, in2} + cin ;
15. assign  sum  =  tempsum[N-1:0]  ;
16. assign  cout= tempsum[N] ;
17. endmodule
```

We may change width of adder to 2, 4 or 8 or anything we wish, just by changing the line that says `parameter N = 4 ;` One thing we may wish to note is that, it is not clear which MSB the carry correspond to. So we introduce an intermediate variable that we call tempsum. Let is take a look at the following code.

```
1. wire  [N:0] tempsum  ;
2.  assign  tempsum =  {1'b0, in1} +  {1'b0, in2} + cin ;
```

Finally we get the carry out from the MSB bit of the tempsum.

Parameter and Module instantiation

Changing the one line of code to change the parameter value is not the only benefit of parameter. Another benefit could be instantiating a module, and while doing so, we can change the value of the parameter.

So we can create two instantiation of the a module each with different value of parameter. That is one big advantage of using parameter. Before we do that, let us take a look at the test bench of the above adder, that instantiate the full adder in the test bench

```
1. /*
2. Full Adder Module parameter instantiation
3. */
```

```
4. `timescale 1ns / 100ps
5.
6. module fulladdertb;
7.
8. reg [3:0] input1;
9. reg [3:0] input2;
10. reg carryin;
11.
12. wire [3:0] out;
13. wire carryout;
14.
15.
16. fulladder #(4) uut (
17. .in1(input1),
18. .in2(input2),
19. .cin(carryin),
20. .sum(out),
21. .cout(carryout)
22. );
23.
24. initial
25. begin
26. input1 =0;
27. input2 =0;
28. carryin =0;
29. #20; input1 =1;
30. #20; input2 =2;
31. #20; input1 =0;
32. #20; carryin =1;
33. #20; input2=0;
34. #20; input1=15;
35. #20; input2=15;
36. #40;
37. end
38.
39.
40. initial
41. begin
42. $monitor("time = %2d, CIN =%1b, IN1=%h, IN2=%h, COUT=%1b, OUT=%h", $time,carryin,input2, input1,carryout,out);
43. end
44.
45. endmodule
```

Notice the instantiation of the full adder

```
1. fulladder #(4) uut (
2. .in1(input1),
3. .in2(input2),
4. .cin(carryin),
5. .sum(out),
6. .cout(carryout)
7. );
```

Immediately after the name of the module being istantiated we can pass the parameter, that can change the value of the parameter. So if we had written `fulladder #(8)` , we would have been instantiating an 8 bit adder. Although, the test bench may not be very good place to do it in real life. In the above test bench, since we did not want to change the parameter we could have written

```
1. fulladder uut
```

in place of

```
1. fulladder (#4) uut
```

If we would have liked to check the adder only for two bit width, we could have changed the test bench code as follows

```
1. reg [1:0] input1;
2. reg [1:0] input2;
3. reg carryin;
4.
5. wire [1:0] out;
6. wire carryout;
7. fulladder #(2) uut (
```

Of course, we will also limit the input values between 0 to 2 in that case. To recap :

Paramter is a constant that can be defined inside a module. The constant is local to the module. However, when we create an instance of the module , the value of the parameter can be changed in the instance. Also, we may

have multiple instances of the module, each with different value of the parameter.

Let us consider the adder example once again

```
module  fulladder (in1, in2, cin, sum, cout);
parameter  N  =  4;
input wire  [N-1:0]  in1 ,  in2 ;
input wire cin;
output wire  [N-1:0] sum;
output wire  cout  ;

wire  [N:0] tempsum  ;

assign  tempsum =  {1'b0, in1} +  {1'b0, in2} + cin ;
assign  sum  =  tempsum[N-1:0]  ;
assign  cout= tempsum[N] ;
endmodule

```

In this example, we can create two instances of fulladder as follows

```
module fulladder #(4) adder1 ( .in1(in1), .in2(in2), .cin(cin), .sum(.sum), .cout(cout));
module fulladder #(8) adder2 ( .in1(in1), .in2(in2), .cin(cin), .sum(.sum), .cout(cout));
```

The first instance has a width of 4 and the second instance has a width of 8. If we do not specify the parameter in instantiation, its value stays unchanged as in

```
module fulladder adder1 ( .in1(in1), .in2(in2), .cin(cin), .sum(.sum), .cout(cout));
```

which is same as

```
module fulladder #(4) adder1 ( .in1(in1), .in2(in2), .cin(cin), .sum(.sum), .cout(cout));
```

If we have multiple parameters, and if we need to change the value of one of the parameters, we will have to list all the parameters in the order of their appearance. Consider the following example

```
module regexample (q, d, clk, rst_n);
 parameter Trst = 1,
 Tckq = 1,
 N = 4,
 NOT_USED =1;

 output [N-1:0] q;
 input
[N-1:0] d;
 input
clk, rst_n;
 reg
[N-1:0] q;
 always @(posedge clk or negedge rst_n)
 if (!rst_n) q <= #Trst 0;
 else        q <= #Tckq d;
endmodule
```

Verilog requires that if you wish to change, say third parameter in instantiation,then you must list 1st to 3rd parameter WITH values, even if their values did not change.

If you wish to change only the second parameter, then you must list 1st and secod parameters with their values.

Engineers have used a trick, where, they list the frequently used parameter as the first parameter, so they do not have to list other parameters.

The rule is - "all parameter values up to and including all values that are changed, must be listed in the instantiation"

So if we wish to change the parameter N in the above example in its instantiation, we should do it like

```
regexample #(1,1,8) r1 (.q(q), .d(d),
 .clk(clk), .rst_n(rst_n));
```

Though following is also correct.

```
regexample #(1,1,8,1) r1 (.q(q), .d(d),
 .clk(clk), .rst_n(rst_n));
```

You can not omit the values of the first two - so following will be incorrect

```
regexample #(,,8) r1 (.q(q), .d(d),
 .clk(clk), .rst_n(rst_n));
```

Can we not pass the parameters by name instead ? Well not is Verilog 1995. But this capability was added in Verilog 2001 and you can now use instantiation like

```
regexample #(.N(8)) r1 (.q(q), .d(d),
 .clk(clk), .rst_n(rst_n));
```

This makes life simpler - does it not ?

Another enhancement in Verilog 2001 is the addition of localparam. With localparam, you do not allow it to be changed directly with instantiation.

However, localparam can be expressed in terms of parameter and when the value of the parameter changes on intantiation, the localparam changes.

The example below shows the usage of localparam

```
module adder_localparameter   (
    input wire [3:0] in1, in2,
        output wire cout ,
    output wire [3:0] sum

   );

   // Declaration of local parameter
   localparam N = 4;

   wire [N:0] sum_temp;

   assign sum_temp = {1'b0, in1} + {1'b0, in2};
   assign sum = sum_temp[N-1:0];
   assign cout= sum_temp[N];

endmodule
```

< Previous Next >