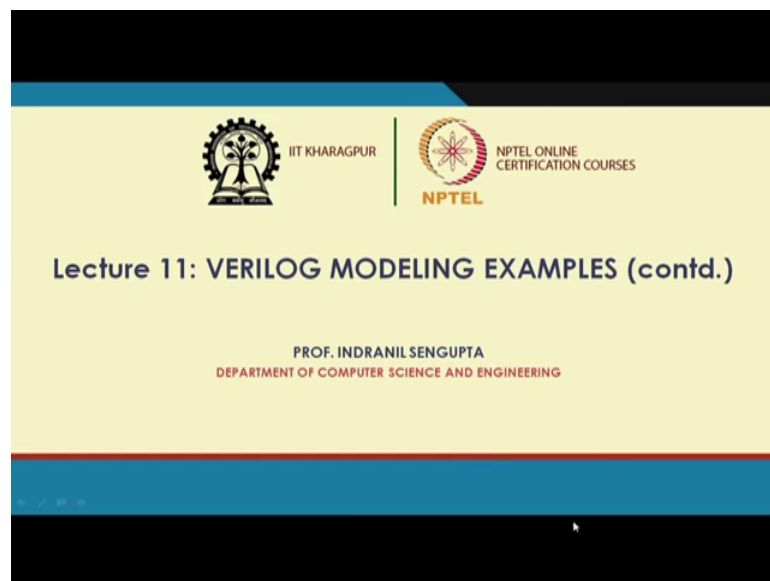


Hardware Modeling using Verilog
Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 11
Verilog Modeling Examples (Contd.)

So, in this lecture, we shall be working out another example. See working with examples is good in two respects one is that it can give you confidence in actually learning about the language and how you can approach the design of a thing; and secondly, you can actually understand the flow the design flow how it is actually done in practice. So, the second example that we will be taking is a slightly more complex example than the multiplexer we took last time. This is the example of an adder and means an adder circuit which is part of an arithmetic logic unit - ALU.

(Refer Slide Time: 01:03)



So, this is our lecture topic.

(Refer Slide Time: 01:06)

Example 2

Version 1: Behavioral description of a 16-bit adder.

- Generation of status flags:
 - Sign : whether the sum is negative or positive
 - Zero : whether the sum is zero
 - Carry : whether there is a carry out of the last stage
 - Parity : whether the number of 1's in the sum is even or odd
 - Overflow : whether the sum cannot fit in 16 bits

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | Hardware Modeling Using Verilog

So, we describe an adder; again we start with the behavioral description of an adder. Now, let us see that what we are trying to do here now.

(Refer Slide Time: 01:26)

© IIT KHARAGPUR

X Y

ALU (ADDITION)

Z

Generation of Status Flags :

- Sign
- Zero
- Carry
- Parity
- Overflow

X 15

X 10

Y 10

Z 01

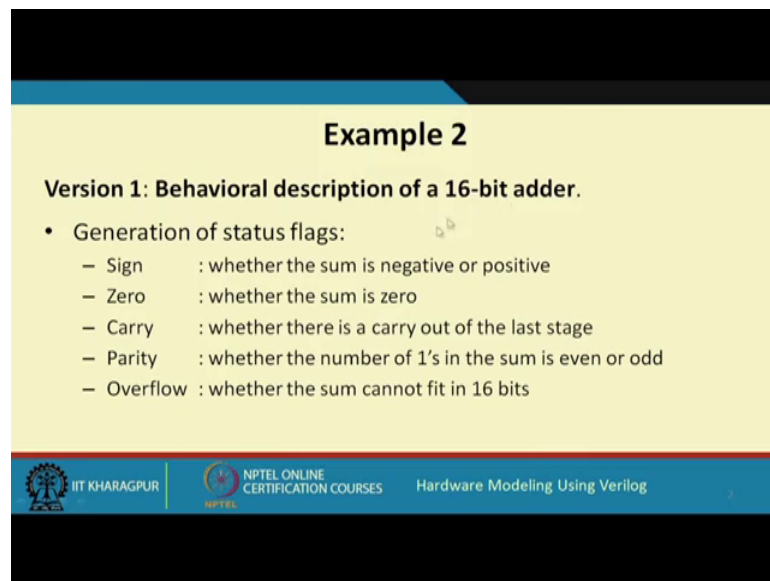
$X[15] \cdot Y[15] \cdot Z[15]'$
 $+ X[15]' \cdot Y[15]' \cdot Z[15]$

This adder is just symbolic as an example we have taken, but in general let us call it these an arithmetic and logic unit, but in the code that I will show will only talk about addition. So, what does the ALU do, the ALU will take two data as input, let us call them X and Y and it will be generating a result let us call them as that Z as output. So, the example that I will be illustrating I shall be only talking about addition. And when this

structure of course will be elaborating an addition we shall be looking at one more thing what we shall be looking at the process of generation of the status flags.

See in many processors, I mean you may be aware of there has some status flags which are automatically set as a result of some arithmetic or logic operations. So, this status flag that we shall be looking are sign, zero, carry, parity and overflow. This status flags actually will give us some idea regarding the addition operation that have been taking place. This sign flag will tell whether the result is negative or positive, this zero flag will tell whether the result is zero or non-zero, the carry flag will tell whether there was a carry out a of the addition. The parity flag will tell whether the number of ones in the result is odd or even; and overflow will tell that well the result means after addition I have done something which is not fitting there is an overflow, so whether there is an overflow or not. So, we shall be looking at the design of an adder with the generation of this five status flags.

(Refer Slide Time: 04:06)



Example 2

Version 1: Behavioral description of a 16-bit adder.

- Generation of status flags:
 - Sign : whether the sum is negative or positive
 - Zero : whether the sum is zero
 - Carry : whether there is a carry out of the last stage
 - Parity : whether the number of 1's in the sum is even or odd
 - Overflow : whether the sum cannot fit in 16 bits



IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | Hardware Modeling Using Verilog

So, we start with the version one of our design which is the behavioral description of the adder. So, this five status flag I have already mentioned, fine.

(Refer Slide Time: 04:19)

```
module ALU (X, Y, Z, Sign, Zero, Carry, Parity, Overflow);
input [15:0] X, Y;
output [15:0] Z;
output Sign, Zero, Carry, Parity, Overflow;

assign {Carry, Z} = X + Y; // 16-bit addition
assign Sign = Z[15];
assign Zero = ~|Z;
assign Parity = ~^Z;
assign Overflow = (X[15] & Y[15] & ~Z[15]) |
                 (~X[15] & ~Y[15] & Z[15]);
endmodule
```



Hardware Modeling Using Verilog

Let us straight away look into the design. This is our behavioral design of our ALU or the adder. You see what are the parameters the name of the module I have given as ALU, X, Y, Z; X and Y are the inputs. Let us assume they are 16-bit numbers 0 to 15, Z is the output. And these are the five flags – sign, zero, carry, parity and overflow, these are all outputs. Now, the addition operation I am doing using a behavioral fashion using the single statement – assign, carry and Z using the concatenation operation, because Z will be the result. And the carry out of the addition that will go in to the carry flag right that is the carry. So, I am generating carry right away like this, carry and Z this will be 16 plus 1 - 17 bits equal to X plus Y.

Now, the other status flags how are they generated. Well, sign well numbers are represented in twos complement form typically. Now, in 2's complement representation the most significant bit of the number indicates the sign; if it is 1, it is negative; if it is 0, it is positive. So, the MSB of the result Z that will go into the sign flag straight away. So, we are straight away assigning Z 15 to sign the most significant bit. Zero flag, what is zero flag? Zero flag will tell whether the result is zero or not which means Z the 16-bit sum it is zero or not, zero means all the bits are zero. If it is zero, this zero flag will be set it will be 1. So, what kind of operation do you require if all the bits of Z are 0, I have to set Z flag to 1; zero flag to 1. So, I need a nor operation I take or of all the zeros, if the inputs are all zero, output of OR will be 1, then a NOT it will be the OR will be zero,


then NOT it will one, so NOR will be 1. So, I need a reduction operator NOR, just I write assign zero equal to reduction NOR on Z that will be 0.

And parity is just exclusive nor if it is even parity it will be 1; if it is odd parity it is 1, it is 0. So, simply write assign parity equal to again reduction operator exclusive NOR of Z. So, generation of sign zero and parity are very simple. Now, overflow there are various ways to detect overflow. The way we are implementing overflow is like this. Let us say we have a number X and Y, this is our bit number 15 or most significant bit, and this is the sum Z. So, here also we have bit number 15. Now, here we are saying that there will be overflow if the condition that has to be followed is this. I am just writing down then I am explaining. This means that either both X 15 and Y 15 were 1 and 1, this is a dot means AND operation, plus means OR operation, X 15, Y 15, Z 15 bar which means this is 1, this is 1, and this is 0.


So, the numbers were negative after addition the number has become positive which is never possible, this is possible only when overflow has taken place or the reverse condition, this was 0 this was 0, but in the sum it has become 1, the numbers are positive, but this sum is showing us negative. So, if we implement this logic that will give you the overflow straight away right. So, here we have done exactly that we have used this same expression to generate the overflow right. So, this is the behavioral description of the adder.

(Refer Slide Time: 09:37)

```
module alutest;
  reg [15:0] X, Y;
  wire [15:0] Z;   wire S, ZR, CY, P, V;
  ALU DUT (X, Y, Z, S, ZR, CY, P, V);
  initial
  begin
    $dumpfile ("alu.vcd"); $dumpvars (0,alutest);
    $monitor ($time," X=%h, Y=%h, Z=%h, S=%b, ZR=%b, CY=%b, P=%b,
      V=%b", X, Y, Z, S, ZR, CY, P, V);
    #5 X = 16'h8fff; Y = 16'h8000;
    #5 X = 16'hfffe; Y = 16'h0002;
    #5 X = 16'hAAAA; Y = 16'h5555;
    #5 $finish;
  end
endmodule
```




IIT KHARAGPUR



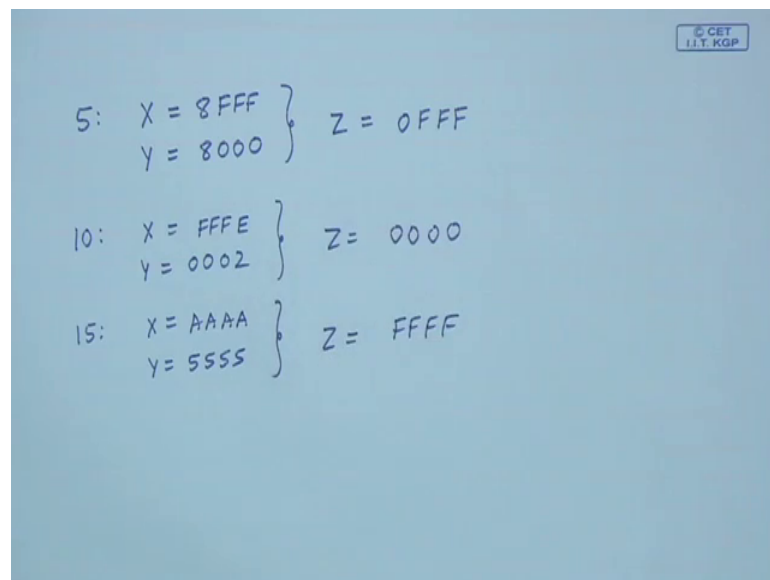
NPTEL ONLINE
CERTIFICATION COURSES

Hardware Modeling Using Verilog



Now, just like in the example we showed earlier we have also written a test bench here. So, we called it as ALU test. So, we have instantiated this module called ALU these the module ALU. So, I have given this name DUT and the parameters are given. So, again in the initial block we have given dumb file dumb variables. We have monitored the time X Y, Z and Z all these status flags. So, here we instantiated the status flag names have given as S, 0, ZR CY, P and V in the same order same order we have given the variable names same order. So, this names you can change same names are not required. And the input data that you applied are you can see with delay of 5 5 5, we have given. Un the first step we have given the first number X equal to 16 bit hexadecimal 8 F F F and 8 0 0 0.

(Refer Slide Time: 10:56)




So, let us just note down this numbers at time 5, we have applied X equal to 8 F F F and Y equal to 8 0 0 0. Then again after a delay of 5, we are applying X equal to F F F E and Y equal to 0 0 0 2. So, after delay of 5 means at time 10, we are applying X equal to F F F E, Y equal to 0 0 0 2. And gain after delay of 5, we are applying X is A A A A and Y is 5 5 5 5, so that time 15 X is A A A A, Y is 5 5 5 and 5 right. So, let us see just by normal addition what will be the sum. If I just add 8 F F F, and X and Y equal to 8 0 0 0 Z will be F and zero is F, F and 0 is F, F and 0 is F, and 8 and 8 will be 0 and there will be a carry out. And here E and 2 if you add it will be 0, there will be a carry. So, 1 and f will be 0 carry 1 and F will be 0 carry 1 F will be 0, it will be all 0, and there will be carry out again. And here A, if you add A and 5 is F, B, C, D, E, F, so F F F F right. So, this should be the sum and the of course, the flags will be set.

(Refer Slide Time: 12:57)

Simulation Output

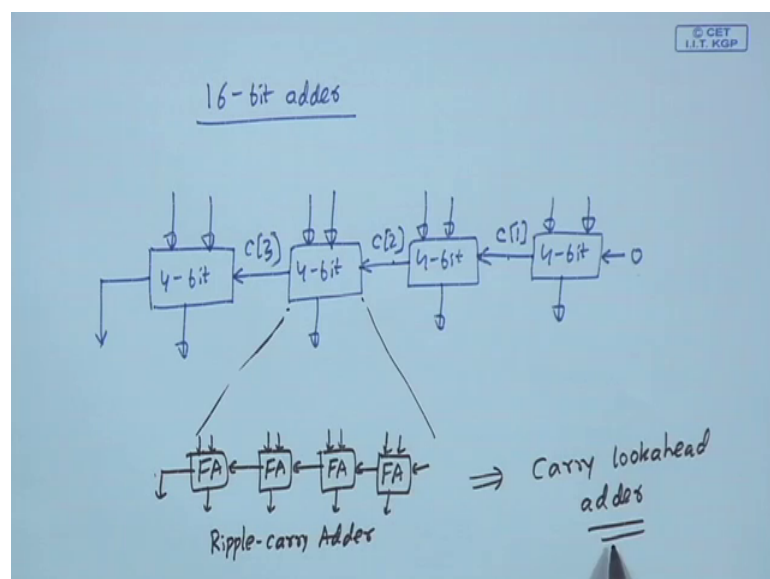
```
0 X=xxxx, Y=xxxx, Z=xxxx, S=x, Z=x, CY=x, P=x, V=x
5 X=8fff, Y=8000, Z=0fff, S=0, Z=0, CY=1, P=1, V=1
10 X=ffff, Y=0002, Z=0000, S=1, Z=1, CY=1, P=1, V=0
15 X=aaaa, Y=5555, Z=ffff, S=1, Z=0, CY=0, P=1, V=0
```

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | Hardware Modeling Using Verilog



Let us see. So, if you just Simulate the simulation output is showing this at time five 0fff, 0000,ffff. So, this is exactly what we got here 0fff, 0000, ffff. And in the all cases you can verify the sign 0fff the sign is positive; 00 this is also, so here you can all check the signs here zero flag, carry flag, parity flag and the overflow flag, you can check all of them, there will be consistent. So, this is our simulation output with respect to the description which you have given here right fine. So, it works correctly.

(Refer Slide Time: 14:19)



Now, what we are trying to do here we have just given a behavioral description of a full adder of an adder now this you want to refine in to more detailed description. So, the first step what we do, the first step what we do is we had started with the design of a 16-bit adder right. So, what you are saying will be using 4-bit adders let us take four 4-bit adders, so they will be all fed with two 4-bit numbers then we generating a sum. So, the carry in for the first stage will be 0, and this carryout will go into the carry in of this, this carryout will go into the carry in of this, this carryout will go into the carry in and this will be the final carryout.

So, let us do a structural design like this. So, now, we are this implementing 16 bit adder using four 4-bit adders, and connecting them like this ripple-carry between the stages right. So, this is our next step. So, well, this what you got this with respect to the simulation output, we have also shown it here. Here I think there is a small typographical error this should be s equal to 0, fine s equal to 0. See over here s is 0 and it goes 1 here; it goes 1 here lasted fine. So, in the waveform also you can see the same thing. So, this is X this is Y sum is Z you can see the value 0fff, you can see the value 0000, ffff, you can see the values right, and you can also see the flag value zero or one whatever it is coming fine.

(Refer Slide Time: 16:30)

Version 2: Structural description of 16-bit adder using 4-bit adder blocks (with ripple carry between blocks).


```

module ALU (X, Y, Z, Sign, Zero, Carry, Parity, Overflow);
input [15:0] X, Y;
output [15:0] Z;
output Sign, Zero, Carry, Parity, Overflow;
wire c[3:1];


assign Sign = Z[15];
assign Zero = ~|Z;
assign Parity = ~^Z;
assign Overflow = (X[15] & Y[15] & ~Z[15]) |
(~X[15] & ~Y[15] & Z[15]);

```

.. Contd.




IIT KHARAGPUR



NPTEL ONLINE CERTIFICATION COURSES

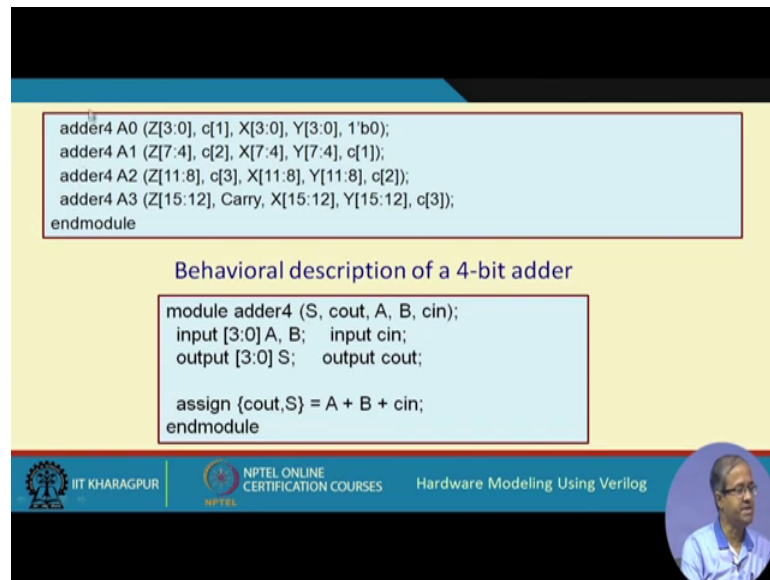
Hardware Modeling Using Verilog



Now, in this second version as a I said, we are using 4-bit adders with ripple-carry between blocks. Now, our let this ALU description module description the first part

remains the same, this was exactly identical with the previous one. What you have done we have just added a wire. So, why you need this wire, we need this wire to just implement this intermediate carries, this we call as c 1, this we call as c 2, this we call as c 3 this intermediate carries. So, we are defining them as wires, wires c 3.

(Refer Slide Time: 17:17)



```
adder4 A0 (Z[3:0], c[1], X[3:0], Y[3:0], 1'b0);
adder4 A1 (Z[7:4], c[2], X[7:4], Y[7:4], c[1]);
adder4 A2 (Z[11:8], c[3], X[11:8], Y[11:8], c[2]);
adder4 A3 (Z[15:12], Carry, X[15:12], Y[15:12], c[3]);
endmodule
```

Behavioral description of a 4-bit adder

```
module adder4 (S, cout, A, B, cin);
input [3:0] A, B;   input cin;
output [3:0] S;    output cout;

assign {cout,S} = A + B + cin;
endmodule
```

The slide also features logos for IIT KHARAGPUR, NPTEL ONLINE CERTIFICATION COURSES, and the text 'Hardware Modeling Using Verilog' along with a small portrait of a man in the bottom right corner.

And we have just instantiated four adders like that. Just see exactly like that or here for the first adder the inputs are X 0 to 3 and Y 0 to 3; for the second adder inputs are X 4 to 7, 4 to 7; next one 8 to 11, and last one 12 to 15. So, for the first adder they carry in is zero second adder carry in c 1 and for the first adder carry out is c 1; for the second adder this c 1 is the carry in. Here the carryout is c 2 for the next adder c 2 is the carry in. c 3 is the carryout for the next one c 3 is the carry in. And the final carryout is carry, this is how the carry is generated. Because here we have not shown the carry, carry will be generated like that sign, zero, parity and overflow are shown. Carries coming here and these are the sum, but the 4-bits adders still will leaving it in a behavioral description like this, this adder four is the 4-bit adder, this was still defining like this. So, this design if you simulate you will see that you are still getting the same simulation result, same simulation output, this you can verify all right fine.


(Refer Slide Time: 18:48)

Version 3: Structural Modeling of Ripple Carry Adder

```
module adder4 (S, cout, A, B, cin);
input [3:0] A, B;   input cin;
output [3:0] S;   output cout;
wire c1,c2,c3;

fulladder FA0 (S[0],c1,A[0],B[0],cin);
fulladder FA1 (S[1],c2,A[1],B[1],c1);
fulladder FA2 (S[2],c3,A[2],B[2],c2);
fulladder FA3 (S[3],cout,A[3],B[3],c3);
endmodule
```

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | Hardware Modeling Using Verilog

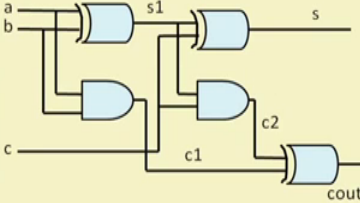


So, the next stage is that we are actually modelling these 4-bit adders as a repeat carry adder.


(Refer Slide Time: 18:58)

```
module fulladder (s, cout, a, b, c);
input a, b, c;
output s, cout;
wire s1,c1,c2;

xor G1 (s1,a,b), G2 (s,s1,c),
G3 (cout,c2,c1);
and G4 (c1,a,b), G5 (c2,s1,c);
endmodule
```



IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | Hardware Modeling Using Verilog



See here in the earlier design we have just use this 4-bit just four bit adder modules right. Now, what I am doing is that you see just look at this diagram. This 4-bit adder modules, so each of this four bit adder modules we are expanding we are using four full adders, we are using four full adders and using four full adders we are using a ripple-carry adder. So, the carry output of one will be going in to the carry input of the next, these will be the

inputs and some output these are standard designs of adder. So, we are actually going for a ripple-carry adder of 4-bits which will be looking like this four full adders in cascade. So, this is exactly what you have done. This adder four module which earlier was looking like this behavioral this we are making it structural.

Now, we are implementing adder four by instantiating full adder four times and interconnecting them in a suitable way. Like the first one is getting $A_0 B_0 c_{in}$ in is the carry in; $c_1 c_1$ is here; c_2, c_2 is here; $c_3 c_3$ is here; and final carrier out is $c_{out} c_{out}$. So, this version is also a very standard way. So, you see if we have this refinement, and if you do a simulation again, you will again see that you will be getting this same simulation result. So, I strongly suggest that you should actually go along with this class not only listening to what I am saying, but also actual implementing and simulating the codes and sink that whether it is working correctly or not. So, if you do this you will see that still you will be getting the same result which means that so far our design is correct so far so good.

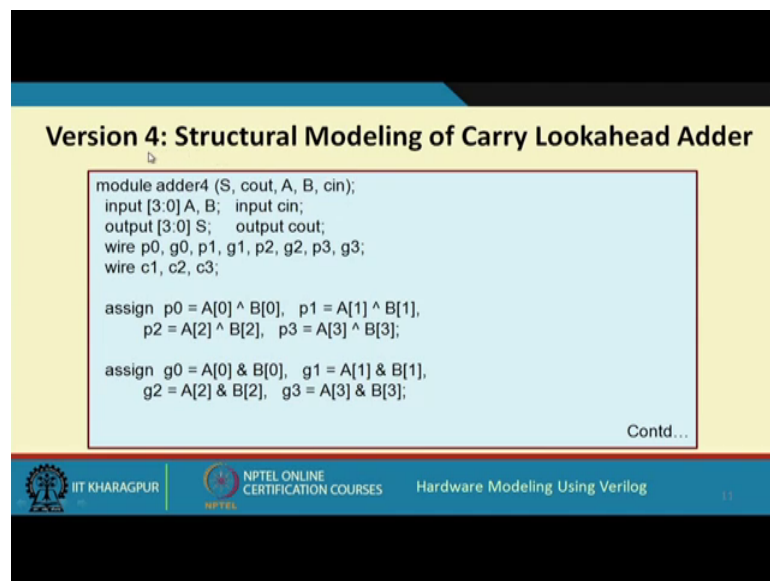
Now, this full adders at the end, this full adders also you can implement using structural way because you see this is one implementation of a full adder which is a compact implementation which requires a total of five gates - three XOR gates, two and gates. There are many implementations of full adder; this is one possible implementation. So, here this structural implementation that I am showing here of full adder uses this netlist. So, you see this $s, cout, a, b, c$ are the input and output lines; a, b, c is the input, s and $cout$ are the output and the intermediate wires are $s_{one} c_1$ and c_2 .

So, they are three XOR gates. So, we are instantiating the gates like this XOR g_1, s_1, a, b , it means this is g_1, s_1 is the output, a, b are the inputs. $g_2 s_1 c_1$ this is $g_2 s_1, c_1$. Last one $g_3, cout, c_2, c_1, cout, g_3, cout, c_2, c_1$ and they are g_4, c_1, a, b this g_4, c_1, a and b . g_5 , this is g_5, c_2 this is s_1 , this is c . So, you see when you instantiate gates in a structural way you can group the similar get types together like the three XOR gates instead of writing XOR three times I have written XOR once and just have separated them by commas this way you can write.

So, you see now we have arrived at a complete structural description of a 16-bit adder where the individual 4-bit adders are also ripple-carry. Now, let us also look at an improved version of this design where this 4-bit adder that I have shown. So, instead of

mapping this in to a ripple-carry adder like this, so we can map it as an alternative way into a carry look-ahead adder. So, carry look-ahead adder is a faster way to implement an adder. See in a ripple-carry adder the carry is rippling through the stages. So, the worst case delay will be the maximum time it takes for the carry to ripple through, but in the carry look ahead adder, we are generating all the carries in parallel, so that the addition becomes very fast.

(Refer Slide Time: 24:39)



Version 4: Structural Modeling of Carry Lookahead Adder

```
module adder4 (S, cout, A, B, cin);
  input [3:0] A, B;  input cin;
  output [3:0] S;   output cout;
  wire p0, g0, p1, g1, p2, g2, p3, g3;
  wire c1, c2, c3;

  assign p0 = A[0] ^ B[0],  p1 = A[1] ^ B[1],
         p2 = A[2] ^ B[2],  p3 = A[3] ^ B[3];

  assign g0 = A[0] & B[0],  g1 = A[1] & B[1],
         g2 = A[2] & B[2],  g3 = A[3] & B[3];
endmodule
```

Contd...

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | Hardware Modeling Using Verilog


So, in the next design here we shall or here we are looking at a design of a 4-bit adder, adder four using carry look ahead principle. Now, here let us skip this.

(Refer Slide Time: 24:55)


```
assign c1 = g0 | (p0 & cin),
       c2 = g1 | (p1 & g0) | (p1 & p0 & cin),
       c3 = g2 | (p2 & g1) | (p2 & p1 & g0) | (p2 & p1 & p0 & cin),
       cout = g3 | (p3 & g2) | (p3 & p2 & g1) | (p3 & p2 & p1 & g0) |
             (p3 & p2 & p1 & p0 & cin);

assign S[0] = p0 ^ cin,
       S[1] = p1 ^ c1,
       S[2] = p2 ^ c2,
       S[3] = p3 ^ c3;

endmodule
```




IIT KHARAGPUR



NPTEL ONLINE
CERTIFICATION COURSES

Hardware Modeling Using Verilog




We shall come back here before the explaining these description.


(Refer Slide Time: 24:57)

How does a Carry Look-ahead Adder work?

- The propagation delay of an n-bit ripple carry order is proportional to n.
 - Due to the rippling effect of carry sequentially from one stage to the next.
- One possible way to speedup the addition.
 - Generate the carry signals for the various stages in parallel.
 - Time complexity reduces from $O(n)$ to $O(1)$.
 - Hardware complexity increases rapidly with n.




IIT KHARAGPUR



NPTEL ONLINE
CERTIFICATION COURSES

Hardware Modeling Using Verilog



Let us first look at how a carry look-ahead adder works, because this will be required to understand how you have come up with these expressions, because these expressions may look a little complicated. So, you see for a ripple-carry adder that means, as you have seen the total propagation delay will be proportional to the number of stages n , because as I had said they carry ripples though from one stage to the other. So, carry look-ahead adder is one way to speed up the addition where the carry signals are

generated in parallel for all the various stages. So, effectively the addition time reduces from order n in ripple-carry adder to order one which is a constant time in carry look-ahead adder, but the flip side is the drawback is the hardware complexity also increases very rapidly with a number of bits n .

(Refer Slide Time: 26:06)

- Consider the i -th stage in the addition process.
- We define the *carry generate* and *carry propagate* functions as:
 - $g_i = A_i \cdot B_i$
 - $p_i = A_i \oplus B_i$
- $g_i = 1$ represents the condition when a carry is generated in stage- i independent of the other stages.
- $p_i = 1$ represents the condition when an input carry C_i will be propagated to the output carry C_{i+1} .

Diagram of a Full Adder (FA) with inputs A_i , B_i , and C_i , and outputs S_i and C_{i+1} .

Equation: $C_{i+1} = g_i + p_i \cdot C_i$

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | Hardware Modeling Using Verilog

Let us see how. Well, if we look at a single full adder. So, in a ripple-carry adder you look at the i th stage full adder where the inputs are A_i , B_i and C_i the output is a sum S_i and a carry $S_i + 1$. Now, we are defining two signals or two you can say two functions called carry generate and carry propagate. Well carry generate specifies the condition where a carry is generated it in the full adder irrespective of the carry in the condition is whenever both the data inputs A_i and B_i are 1, then carry will be generated irrespective of c_i . So, that is g_i , g_i is A_i and B_i and propagate carry propagate represents the condition it says that when the input carry will be propagating to the output. What is that condition? The condition is means among A and B at least one of them means exactly one of them should be one, and the other should be zero. Suppose it is 0 and 1 then if there is a carry coming 0 1 1 that will generate a carry the carry will be propagating. So, this is the condition for carry propagation exactly one of A_i and B_i should be 1.

So, this carry propagation is defined by the exclusive odd function. This is generate and propagate. So, g_i and p_i you can implement by NAND gate or an XOR gate right. Now,


with this you can write that the output carry expression output carry will be given by will either the carry is generated or the propagate condition is true and there is an input carry. So, c_{i+1} you can write down as by an expression g_i or p_i, c_i . Now, this you can use recursively to find out an expression.

(Refer Slide Time: 28:25)


Unrolling the Recurrence

$$\begin{aligned}
 c_{i+1} &= g_i + p_i c_i = g_i + p_i (g_{i-1} + p_{i-1} c_{i-1}) = g_i + p_i g_{i-1} + p_i p_{i-1} c_{i-1} \\
 &= g_i + p_i g_{i-1} + p_i p_{i-1} (g_{i-2} + p_{i-2} c_{i-2}) \\
 &= g_i + p_i g_{i-1} + p_i p_{i-1} g_{i-2} + p_i p_{i-1} p_{i-2} c_{i-2} = \dots
 \end{aligned}$$

$$c_{i+1} = g_i + \sum_{k=0}^{i-1} g_k \prod_{j=k+1}^i p_j + c_0 \prod_{j=0}^i p_j$$



IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | Hardware Modeling Using Verilog



c_{i+1} equal to g_i plus $p_i c_i$. So, you can write this c_i , you can write as g_{i-1} plus $p_{i-1} c_{i-1}$, you can expand it like this. This c_{i-1} again you can write as g_{i-2} plus $p_{i-2} c_{i-2}$, like this you can recursively go on expanding. So, I am not showing you the whole thing. See, basically what it means I am just writing down some simple expressions.

(Refer Slide Time: 29:00)

$$C_{i+1} = g_i + P_i \cdot C_i$$
$$C_1 = g_0 + P_0 C_0$$
$$C_2 = g_1 + P_1 C_1 = g_1 + P_1 g_0 + P_1 P_0 C_0$$
$$C_3 = g_2 + P_2 C_2 = g_2 + P_2 g_1 + P_2 P_1 g_0 + P_2 P_1 P_0 C_0$$

⋮

See, what you got is c_{i+1} equal to $g_i + p_i \cdot c_i$. So, you can generate the carry for the first stage c_1 as you just substitute i equal to 0, $g_0 + p_0 c_0$. So, what will be c_1 this c_1 , c_2 , c_2 will be again you put g put i equal to 2, and i equal to 1, so $g_1 + p_1 c_1$ right. Now, c_1 you have already got this. So, you can substitute c_1 here. So, this becomes $g_1 + p_1$ multiplied by this, so $p_1 g_0 + p_1 p_0 c_0$. You go to c_3 c_3 will be $g_2 + p_2 c_2$. Now, c_2 you have already got this, so this will be $g_2 + p_2$ multiplied by this multiply this $p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0$ and so on like this we will be getting this expression. So, all the carries you can get just by a two level and or kind of expression, there is no question of carries rippling through, but as you go on this expressions become more complex, bigger expressions, more number of gates this is what I was saying.

(Refer Slide Time: 30:44)

Generation of the Carry and Sum bits

$$C_4 = g_3 + g_2p_3 + g_1p_2p_3 + g_0p_1p_2p_3 + c_0p_0p_1p_2p_3$$

$$C_3 = g_2 + g_1p_2 + g_0p_1p_2 + c_0p_0p_1p_2$$

$$C_2 = g_1 + g_0p_1 + c_0p_0p_1$$

$$C_1 = g_0 + c_0p_0$$

4 AND2 gates
3 AND3 gates
2 AND4 gates
1 AND5 gate
1 OR2, 1 OR3, 1 OR4
and 1 OR5 gate

$$S_0 = A_0 \oplus B_0 \oplus c_0 = p_0 \oplus c_0$$

$$S_1 = p_1 \oplus c_1$$

$$S_2 = p_2 \oplus c_2$$

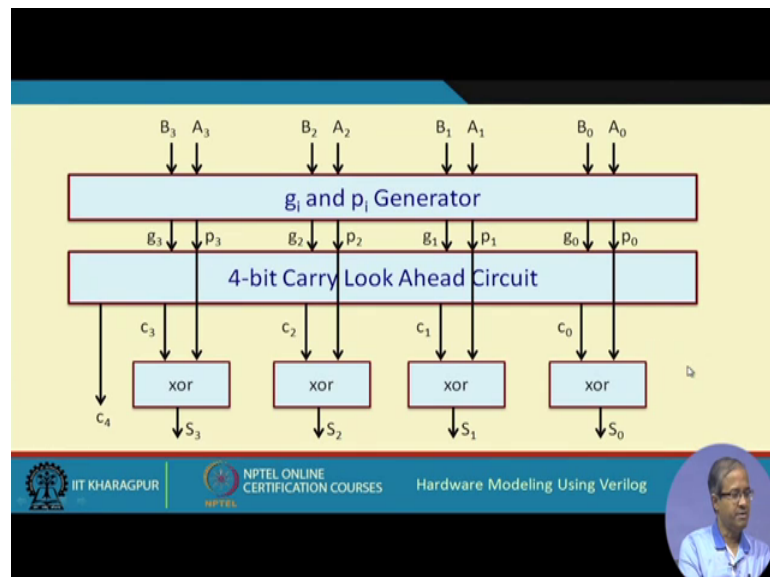
$$S_3 = p_3 \oplus c_3$$

4 XOR2 gates

NPTEL ONLINE CERTIFICATION COURSES
Hardware Modeling Using Verilog

So, actually this is what I have just now showed c 1 is this, c 2 is this, c 3 and c 4. So, you require so many gates. And some you can directly generate, you see p is nothing but A and Bs XOR. So, the sum is the XOR of all the three bits. So, you can simply write p 0 X or c 0. Similarly, S 1 is p 1 XOR c 1 and so on you need 4 XOR gates.

(Refer Slide Time: 31:16)



So, in a carry look-ahead adder in a yeah carry look-ahead you have a circuit first which generates all the g and ps which means XOR and AND gates. Then we have a carry look-ahead circuit which implements these functions c 1, c 2, c 3 c 4 using AND, OR circuits.

So, it generates c_0, c_1, c_2, c_3 and also c_4 . So, now, you can directly generate this sums by using XORs, XOR the carry with the ps. So, the total time is constant irrespective of the number, there is no ripple of the carry. So, this will be much faster.

So, just remember these expression and the sum expressions. So, if you remember these expressions you can just correlate that what you have done is exactly that, here the carry propagate and generate signals I have defined as a wires in between and the intermediate carries also. So, p_0 is $A_0 \text{ XOR } B_0$; p_1 $A_1 \text{ XOR } b_1$ and so on. Similarly, the generate signals and A_0 and B_0 , A_1 and B_1 and so on. So, I generate the propagate signals, I generate the carry generate signals then I generate all the carry signals, c_1 equal to $g_0 \text{ OR } p_0 \text{ cin}$, cin is c_0 . You see this expression c_1 equal to $g_0 \text{ OR } p_0$ this c_0 is cin ; c_2 is $g_1 \text{ OR } p_1 \text{ OR } p_0 \text{ cin}$ exactly we have written down that expression. c_2 equal to $g_1 \text{ OR } p_1 \text{ OR } p_0 \text{ cin}$ c_3 and c_4 is finally, cout . So, as you can see as the number of bits increases these expressions are becoming more complicated and finally, sum is the xors. So, this is a carry look-ahead adder.

So, in the same way as I had said you can replace a module by a better or an improved module. You replace a module by another module like, for example, we replaced a 4 bit ripple-carry adder by a 4 bit carry look-ahead adder we have got a faster adder, but we can do simulation and you can find out that functionally our design is correct. It is working correctly. So, this actually completes a our discussion for today's lecture. See what I try to illustrate over the last two examples we discuss is that when we look at non trivial designs slightly bigger designs we often break the design up in a hierarchical fashion. Whatever something was discussed in a or described in a behavioral fashion at one level, we do a some kind of iterative refinement. We break that behavioral description into structure description, try to be more detailed and we repeat the process until we reach a point where our entire design becomes structural.

So, in a normal digital system design, you will see later that well we do not only have combination circuits, we also have sequential circuits flip-flops, state machines and so on. So, we shall see later that when we have such a system where we have combination circuits, registers, and also lot of finite state machines, there it may not be worthwhile to convert all the modules, all the descriptions into structural. Well, the data path the circuits where the actual calculations or computations have carried out, those of course, we can convert into structural fashion. But the controller the circuit which implements a

finite set machine which generates the control signals often that we do not convert into a structural fashion, because it involves too much of a work on the part of the designer. So, we shall just learn these things slowly over the next lectures.

Thank you.