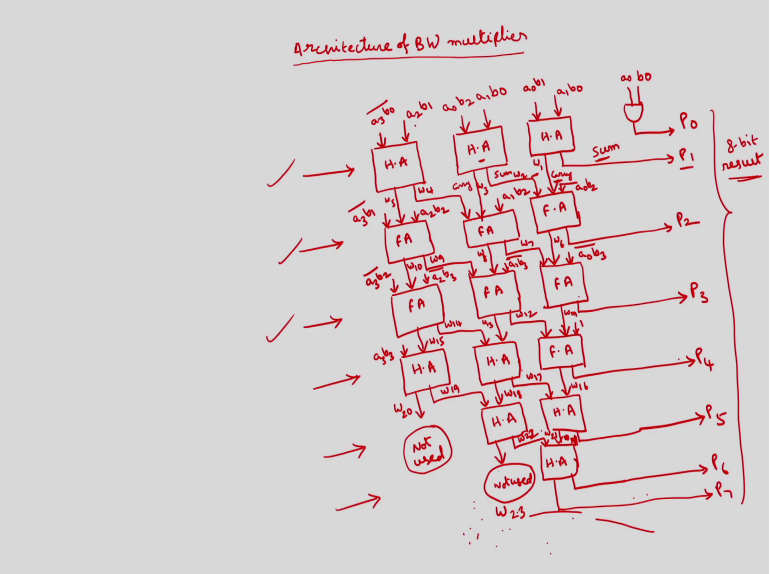
System Design Through VERILOG
Prof. Shaik Rafi Ahmed
Department of Electrical and Electronics Engineering
Indian Institute of Technology, Guwahati

Case Studies
Lecture - 27
IIR filter implementation

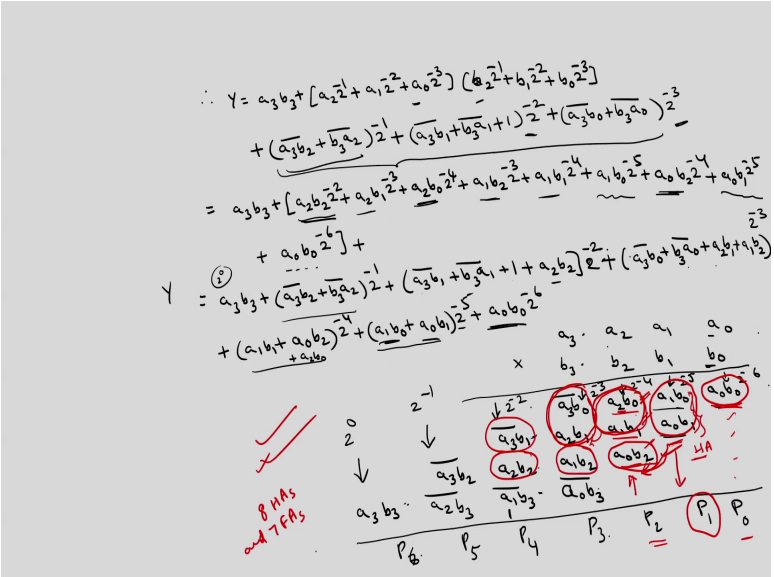
In the last lecture, I have discussed about Baugh Wooley multiplier. So, we have discussed the algorithm of Baugh Wooley multiplier. So, today, we will discuss about the architecture of Baugh Wooley multiplier.

(Refer Slide Time: 00:49)



BW stands for Baugh Wooley. So, basically, we can implement this Baugh Wooley multiplier using half and full adders. You can see the algorithm which we have discussed in the last lecture.

(Refer Slide Time: 01:24)



So, this was the algorithm that we have developed in the last lecture. So, basically, this a 0 b 0 we require AND gate. So, a 1 b 0 plus a 0 b 1 to add this, we require half adder. Now, at this stage, we have already 3-bits and then, carry from this; total 4 bits, we have to add. So, we require one full adder and one half adder. Similarly, here also. So, all these additions of each column can be performed by using full adders and half adders.

So, we are going to draw the architecture of this one. So, we require total 8 half adders and 7 full adders. This is the optimum design. The optimum design requires 8 half adders and 7 full adders. So, we can have the other combinations also. So, initially a 0 b 0 can be generated by using the AND gate basically; a 0 b 0 is generated using AND gate. This is directly your P 0, whose inputs are a 0 b 0.

You can see here a 0 b 0 is directly P 0 because here there are no bits. So, through AND gate we can generate P 0. So, to generate P 1, we require a half adder, whose inputs are a 1 b 0, a 0 b 1. This is a half adder; a 0 b 1, a 1 b 0. Of course, this product terms can be generated by using AND gate again. So, for half adder we have two inputs and two outputs. The outputs are sum and carry. So, this sum bit of this one will acts as P 1, this is sum bit and carry bit, we are going to add in the subsequent stage.

So, here whatever the sum bit that we are going to get is P 1, carry bit we are going to propagate to the next stage. So, in the next stage again to perform the first two, we require the half adder. Then, along with this sum of this one, there is a carry from here on this bit. Total, I have to add 5 bits here. Initially, I will add these 2 bits using one half adder. So, you will get sum along with that sum, carry will be propagated here. So, along with the sum bit of these two product times plus this plus carry from this one.

So, totally, I have to add 5 bits; these 2 bits along with that carry a 0 b 2 and carry of this summation. So, I require one full adder, one half header to get the P 2. So, another half adder here, then we require one more full adder. So, what are the inputs of these half and full adders? In this half adder is first I want to add a 0 b 2, a 1 b 1, a 0 b 2, a 1 b 1.

So, we will get some sum; sum value as well as carry. This is sum of this one, this is carry of this half adder. And this sum bit is the carry bit, we are going to generate apply to this full adder. So, what about the third bit? Third bit will be a 0 b 2. We are performing this addition using half adder along with the sum bit of this half adder, this bit and the carry from the previous stage we are going to apply to the full adder.

So, this bit is nothing but a 0 b 2. So, the sum bit of this one is your P 2 and carry bit you have to propagate to next stage. So, like that if you proceed this, so this will be another half adder here. So, the inputs are a 3 b 0 bar a 2 b 1, a 3 b 0 bar a 2 b 1; then, we will be having one more full adder here, another full adder here. So, we can explain the operation in similar manner. I will draw the final architecture. This is another full adder, another full adder.

So, here this sum bit is applied to the full adder this one and carry bit is applied here. Here we will having three full adders. So, this sum bit is applied, carry bit and the sum bit of this one, carry bit is applied here, sum bit is applied here. This is having some symmetrical structure. the other signals, I will fill from the table.

Now, here half adders are enough. This is full adder again because we have a one extra term 1. Then, we require two half adders and another half adder here. So, this I will give instead of here, I will show the other side. So, this is P 0, this is P 1, this is P 2 and here, we generate P

3. Here, we generate P 4, P 5 and here, we generate both the bits P 6 and P 7; total 8-bit result. This is 8-bit result.

Now, coming back to the connections here, this is having similar structure. So, I have reached all these terms up to this term is over, these two terms are over, these two terms are over, these two terms are over. So, in the other, we have a 0 b 2, a 1 b 2, a 2 b 2. So, a 0 b 2, a 1 b 2, a 0 b 2 is over. Here, a 1 b 2 is required, then we have a 2 b 2, but this require another input, this is full adder. So, a 2 b 2, then a 3 b 1 bar. Similarly in the next row. So, I have covered this also, this also, this also. So, these are left.

Next row, we have a 0 b 3 bar a 1 b 3 bar a 3 b 3 bar. So, here we have full adder. This is a 0 b 3 bar and this is having other input a 1 b 3 bar, this is a 2 b 3 bar and this is a 2 b 3 bar is over; a 3 b 2 bar, a 0 b 3, a 1 b 3, a 2 b 3, a 3 b 2 bar. So, here this half adder requires a 3 b 3.

So, this is the complete architecture of a 4 by 4 Baugh Wooley multiplier. So, whatever the table that we have here this table, I have mapped onto the architecture, using minimum number of full adders and half adders. So, you can see a different types of these mappings, among all the mappings this is the optimized mapping.

So, totally, we have 8 half adders and 7 full adders, you can easily analyze this. So, total seven 8-bit multiplication output is there. So, these actually these are not used; these are extra bits. Now, in order to write the Verilog code, so we have to first write half adder code and then, full adder code, then you have to instantiate the half adder code 8 times, full adder code 7 times to get the Baugh Wooley multiplication.

So, we require some wires like here this is wire we require a wire w 1. So, this is also w 2, w 3, w 4, w 5, w 6, w 7, w 8, w 9, w 10, w 11, w 12, w 13, w 14, w 15, w 16, w 17, w 18, w 19, w 20, w 21. Of course, these two are not used. So, no need of definition of these wires. So, totally we require 21 wires.

(Refer Slide Time: 14:52)

```
module baugh_mult(a, b, p);
input [3:0] a, b;
output [7:0] p;
supply1 one;
wire w1, w2, w3, w4, w5, w6, w7, w8, w9, w10, w11,
w12, w13, w14, w15, w16, w17, w18, w19, w20, w21,
w22, w23;

assign p[0] = a[0]&b[0];

half_adder HA1(a[1]&b[0], a[0]&b[1], p[1], w1);
half_adder HA2(a[1]&b[0], a[0]&b[1], w2, w3);
half_adder HA3(~(a[3]&b[0]), a[2]&b[1], w4, w5);

full_adder FA1(w2, w1, a[0]&b[2], p[2],w6);
full_adder FA2(w4, w3, a[1]&b[2],w7, w8);
full_adder FA3(w5, a[2]&b[2], ~(a[3]&b[1]), w9,w10);

full_adder FA4(w6, w7, ~(a[0]&b[3]), p[3], w11);
full_adder FA5(w8, w9, ~(a[1]&b[3]), w12,w13);
full_adder FA6(w10, ~(a[2]&b[3]), ~(a[3]&b[2]), w14,w15);

full_adder FA7(one, w11, w12, p[4], w16);
half_adder HA4(w13, w14, w17, w18);
half_adder HA5(a[3]&b[3], w15, w19, w20);

half_adder HA6(w16,w17, p[5], w21);
half_adder HA7(w18, w19, w22,w23);

half_adder HA8(w21, w22,p[7], p[6]);

endmodule

module half_adder (x, y, s, cout);
input x, y;
output s, cout;
assign s = x^y^cin;
assign cout = (x&y)|(y&cin)|(x&cin);
endmodule

module full_adder (x, y, cin,s,cout);
input x, y, cin;
output s, cout;
assign s = x^y;
assign cout = x&y;
endmodule
```

So, you can see now the Verilog code. So, this is the Verilog code corresponding to Baugh Wooley multiplier. So, basically, we are going to construct this Baugh Wooley multiplier using half adders and full adders. This is the code for half adder. So, we have taken x, y, as inputs; sum and carry out, c out are the outputs.

So, sum is x exclusive or y exclusive or c in and carry is x y plus y z plus z x. So, this is module for half adder; this is the module for full adder. Here also we have x, y, c in, s, c out. We see the adder in which we have to take while instantiating, the inputs, outputs and carry, sum and out.

So, we have to take this sequence in the same order. So, this is well understood, this we have already explained. Now, coming to this main program of this Baugh Wooley multiplier, so basically we have two inputs a, b; output is p. We have P 0 to P 7 and a and b are 4-bit numbers.

So, a and b are 4-bit numbers; output P is 8-bit number, P 0 to P 7 and we require total 21 wires are enough, actually here I have taken 23 wires because this also we have taken as wires, if we take this also w. So, if I taken the same adder, if I take this as w 20, this is the adder that I have followed in the program 20. This is 21, w 21, w 22 and this is w 23 that is why we have taken total 23 wires.

So, initially a 0 b 0 is P 0. So, this LSB bit of this P can be obtained through this AND operation. Then, we have three half adders. So, I have written the adder in the same manner; in the first row, we have three half adders, then we have three full adders, then we have three full adders, then two half adders, one full adder, then we have two half adders, then we have one half adder. So, I have written in the same manner.

So, the first three sentences we have written for three half adders. So, this three half adder will implement these three half adders. Similarly, these three full adders will implement these full adders. Similarly, so this three full adders will implement these full adders. Similarly, we have half adder, half adder, full adder. This one, then two half adders, then one half adder; so, in the same order.

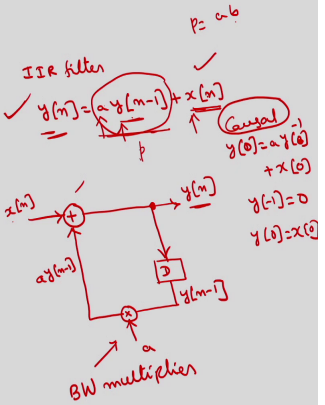
So, you can easily see that for the first half adder the inputs are a 1 b 0, a 0 b 1, a 1 b 0, a 0 b 1, the outputs are sum bit is the P 1 itself, another is wire w 1. So, this wire w 1 P 1. So, this is the order that I am following. This first bit represent sum bit and this is carry bit. So, the sum bit of this half adder is P 1 and carry bit is wire.

So, sum bit is P 1, carry bit is wire w 1. So, similarly, I have written for the other half adders and full adders also. This half adder will be having a 0 b 2, a 1 b 0, a 0 b 2, a 1 b 0 and second half adder will be having inputs are a 1 b 0, a 0 b 2; this is a 1 b 0, a 0 b 2 sorry my mistake and the third one is having a 3 b 0, a 2 b 1 and a 3 b 0 bar; this is a 3 b 0 bar, a 2 b 1, a 2 b 1.

So, similar you can verify this. So, this is the complete code for the Baugh Wooley multiplier. So, our final goal is to implement IIR filter using Baugh Wooley multiplier because using unsigned multiplier, we implemented a 3 type FIR filter; but to implement IIR filter, I am using a signed multiplier. Because in most of the applications, we have to deal with signed data. So, data can be negative also. So, here I am going to implement IIR filter.

(Refer Slide Time: 19:49)

```
module IIRfilter(clk,rst,a,x,y);
input clk,rst;
input [3:0]a,x;
output [3:0]y;
reg [3:0] y_val;//register to store intermediate value of y
wire [7:0] baugh_prod_actual;//BW multiplier product
baugh_mult  bm1(.a(a), .b(y_val), .p(baugh_prod_actual));
always@(posedge clk,rst,x,a)
begin
  if (rst) begin
    y_val <= x;
  end
  else begin
    y_val <= x + booth_prod_actual[3:0];
  end
end
assign y = y_val;
endmodule
```



So, this is the code for IIR filter. So, we have written this IIR filter code by instantiating the Baugh Wooley multiplier. So, here, I have taken a basic IIR filter as infinite impulse response filter is simply y of n is equal to a into y of n minus 1 plus x of n. So, this consisting of one multiplier, one adder.

This is recursive because the output not only depends upon the present input, but also on the previous output; whereas, in case of IIR system, the output only depends upon the present and past inputs without regard to the previous outputs. So, if we take this log diagram of this IIR filter, this is x of n and this is y of n. There will be a loop.

The difference between the FIR and IIR filter is in case of FIR filter, open loop system. FIR filter is open loop system, IIR filter is closed loop system. This is a multiplier, this is a delay. So, this y of n becomes y of n minus 1 after delay. Here we are going to multiply with a and this will be applied as a second input for the AND gate.

This will be a into y of n minus 1 plus x of n, this will be output y of n. So, here basically this multiplication we are going to implement by using Baugh Wooley multiplier. So, this is basically adder. So, here, this is the module for IIR filter which implement this equation. So, we have clock reset a x y. This a is one input, x is another input, y is the output. So, inputs are clock reset.

So, two inputs are a is a 3-bit number; x is an also another 3-bit number and output can be 3-bits or it can be if I multiply two 4-bit numbers, we will get 8-bit result. So, a is here 4-bit number, x is 4-bit number, y is also 4-bit number, here I am taking. If you want, you can take 8-bit result output y also.

Then, we are taking because this y is 4-bit, this y variable also we are taking as 4-bit ok. So, this is to store the intermediate values. Then, I am defining a vector wire with 8 bits. I am calling this as a Baugh product actual. This will give the multiplier product. Then, I have instantiated the Baugh multiplier which we have discussed in the earlier slide.

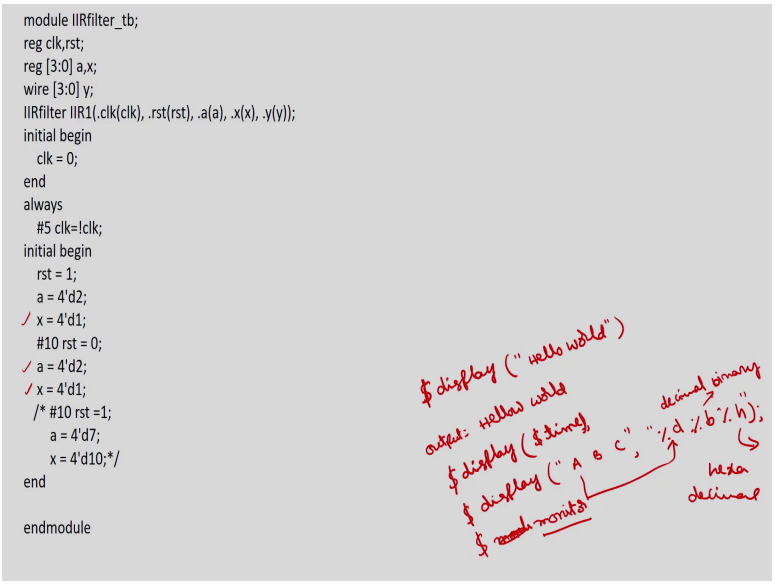
So, this multiplier Baugh multiplier a, b, p. So, Baugh multiplier a, b, p. So, a to a, b to b nothing but y variable because I want a into y n minus 1. So, in Baugh multiplayer, we have taken a into b is equal to p, but here I want a into in place of b, I want y. So, b, I have connected to y. So, P 2 Baugh product actual this we have defined here. So, I am going to store this Baugh product actual to the p.

So, that P this we are calling as p, this P we have to add to x to get the final output y. So, always at positive of edge of clock reset or x or a, begin if reset is equal to begin. So, initial value, I will start with x of n. Because initially I do not know. So, if I assume that the system is causal, if I want to find out y of 1, what is this? a into y of 0 plus x of 0; sorry y of 0 if you want to find out, this is y of minus 1.

So, y of minus 1 normally we will assume that 0 because this system, I am assuming as causal. So, if y of minus 1 is 0, what is y of 0? Simply x of 0. That is why the initial value when reset is there, this y variable we are going to take to x because y of minus 1 becomes 0. Then, otherwise y variable is equal to x, the previous value of this x plus this product which is equal to a into y of n minus 1, we are calling as P and then end.

So, finally, this y variable will be assigned to the y. So, this is how we can implement this IIR filter using Baugh Wooley multiplier.

(Refer Slide Time: 24:58)



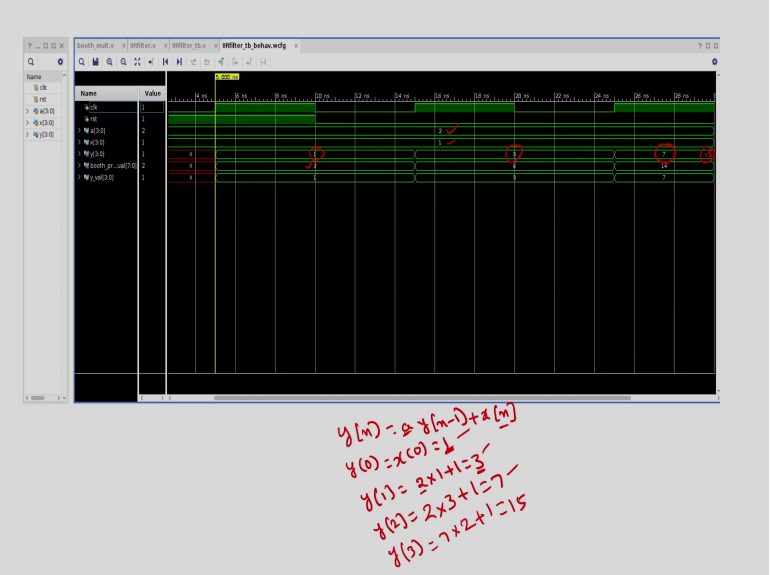
Then, if you take the test bench waveforms, this is test bench. So, the similar program, but we have given the time delays. So, here sometimes, you can use this if you want to display this values y or a or b. So, there are some keywords like display. If you write like display “Hello world”.

So, “Hello World” will be displayed. The output of this one will be Hello world will be displayed on the screen. If you do not write this, normally the you will get wave forms. If you want to display the values, we can is display by using this keyword, display keyword. So, if you want to display time, so this will display the current simulation time.

So, if you want to display some numerical values say for example, display “A, B, C are the three variables comma percentage of d comma percentage of b percentage of h. So, this represent that a is represented in decimal, d stands for decimal; b stands for binary; h stands for hexa decimal. You can have octal. You can have either lower case letters or uppercase letters.

This how you can display, also there is another keyword calls monitor. Monitor is also similar to display. But here in case of monitor, all the parameters which are inside this monitor will be continuously accessed. If any one of these variable changes, then the corresponding values will be displayed on the screen.

(Refer Slide Time: 27:23)



So, this is the test bench correspond to IIR filter and these are the waveforms. So, you can easily see that initial value of this y is x itself 1. So, here basically, we have this y of n is equal to a into y of n minus 1 plus x of n. So, initially y 0 is x 0 which you have said to 1 here. Then, you have taken this x is permanently you have connected to 1; x permanently connected to 1.

So, initial value is 1 and then, a is given as 2, a is given as 2. So, what is y 1? So, the previous value of this one is 1, 2 into 1 plus x is permanently 1 3. That is why the value is 3 here. So, in the second iteration, this one will be the input, y 2 will be equal to this a is permanently 2, a is permanently 2 into this previous value is 3 plus x is permanently 1, 7.

So, like that we will proceed and then, so these are the output y values. So, we will get values as 1, 3, 7; the next value if you want, y 3, it will be 7 into a is 2 only plus 1 is 15. So, like that it will proceed next value here will be 15, we will get somewhere. So, these are about this the way forms corresponding to the IIR filter.

So, with this, I complete the course. So, while writing these Verilog codes, I might have committed some silly mistakes. That you correct yourself because actually I normally prefer to write the Verilog codes instead of using the entire code in the slide. So, in the initial lectures, I have written the codes.

So, while writing the codes, I might have committed some mistakes because here no students to find out those mistakes. I request you to correct those mistakes yourself and for complex problems, anyhow I have used the slides. So, normally I believe that uh; so, explaining the code while writing will make better understanding of the audience. So, and if any other mistakes is there, you can send the mail or you can inform through the discussion forum.

Thank you.