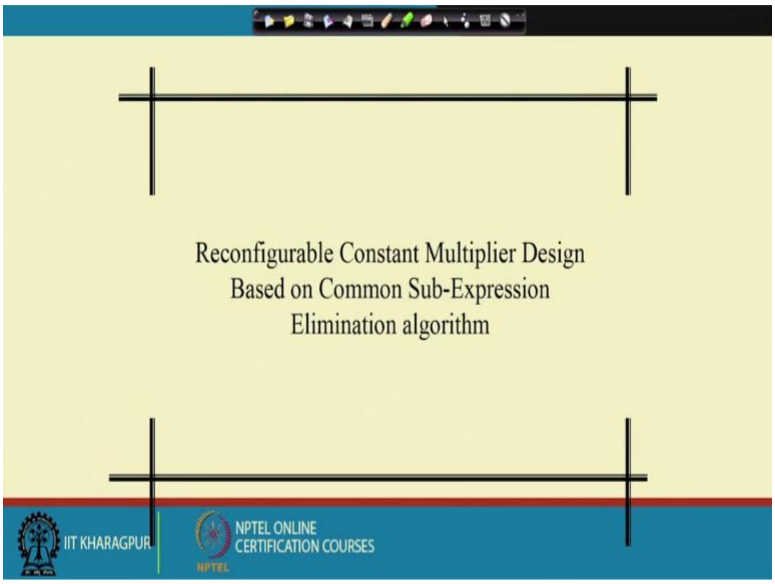
Architectural Design of Digital Integrated Circuits
Prof. Indranil Hatai
School of VLSI Technology
Indian Institute of Engineering Science and Technology, Shibpur, Howrah

Lecture - 38
Multiplier Architecture (Contd.)

So, welcome back to the course on Architectural Design of ICs.

(Refer Slide Time: 00:22)

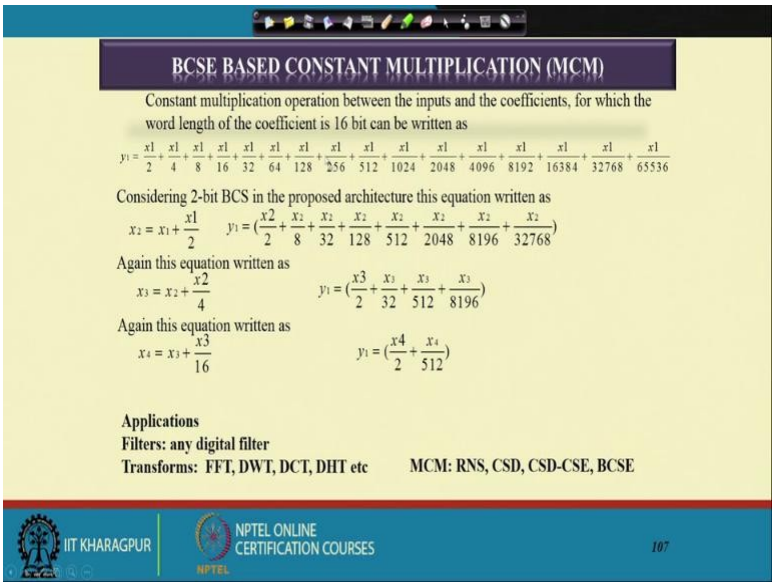


So, in the last class, we have seen what is the meaning of or what is this constant multiplication and then, why I require this reconfigurable architecture for this constant multiplier design, what is the usefulness of constant multiplier design? That means, in compare to the generic multiplier, that means design I can save or I can optimize the multiplier design in a better way whenever we are doing this constant multiplication operation, ok.

So, after that we have considered one 3 TAP fir filter, where we are using 3 constant numbers where as which is let us consider a b and c. So, at that time after that we have seen that ok, it requires this 24 number of anding operation, then 21 number of addition operation to do this whole of, to implement the whole circuit of doing this a x in b x in and c x in, ok.

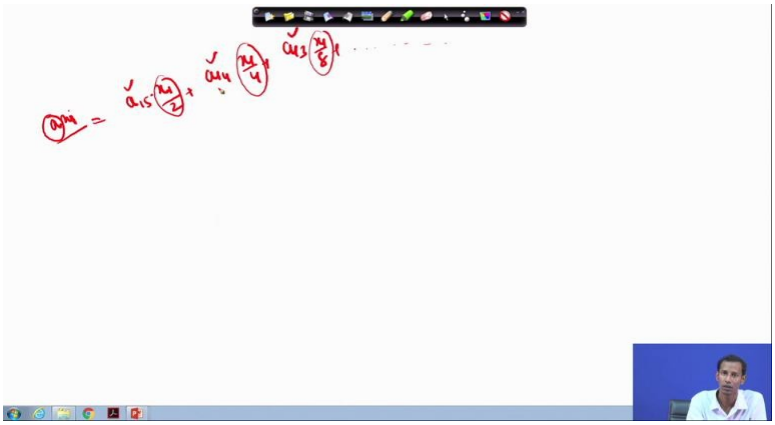
So, now my point is that can I optimize this more? This can I save or can I reduce these numbers 24 and 21 in a better way or in a. That means, can I reduce this number. If I can reduce, then how can I reduce that? So, I told you for that there are different algorithms to do that ok. So, among them this binary common sub expression elimination and algorithm is one of the efficient one which can where by which we can reduce which is basically applicable for reconfigurable this constant multiplier design by which we can reduce this number. So, how we can reduce these number that we will see.

(Refer Slide Time: 02:05)



So, suppose in a constant multiplication operation ok, we are having this input and the coefficient. So, input and the coefficient means x in is the input and coefficient is nothing, but this a b and c. So, each of this if I consider each of these are of 16 bit, ok. So, 16 bit means if these input is x 1 ok, they are of 16 bit and this y 1, y 1 is basically a into x 1, ok that means what this y 1 is basically a into x 1, equals to y 1.

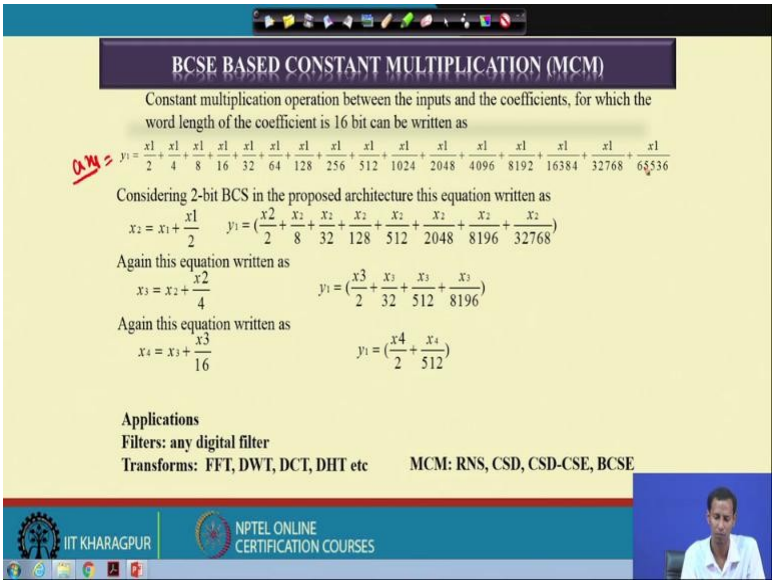
(Refer Slide Time: 02:55)



So, how can I write this y 1? I can write considering the fact that what I said that the worst case condition of this a is when all the big position of a are 1, when all the big position of a are 1, at that time that means each of these product term I have to consider, right. That means what? Means what at that time that means, a 15 into x 1 by 2 plus a 14 into x 1 by 4 a13 into x 1 by 8 something like this. Each of this, if each of this bit is 1, so that means at that time these equations can be written only in terms of x 1 and that is basically written in this form where I have removed the corresponding position considering that a 15 is 1.

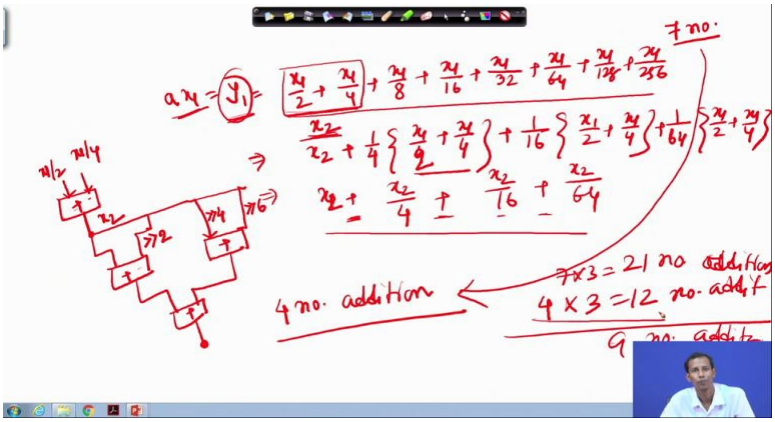
So, I will get x 1 by 2. Here a 14 is 1. So, I will get x 1 by 4, here a 13 is 1. So, I will get x 1 by 8 over here. So, a 12 is 1. So, x 1 by 16 is 1. So, in this manner I will get that means, for doing this a x 1 which is y 1.

(Refer Slide Time: 04:41)



So, I will get each of these terms which I have to add, ok. So, in here what is there? That means we have considered 16 numbers; let us consider 8 numbers, ok.

(Refer Slide Time: 05:05)



So, if I consider 8 numbers, so a into x 1 which is y 1 that can be written as x 1 by 2 plus x 1 by 4 plus x 1 by 8, x 1 by 16 plus x 1 by 32, x 1 by 64, x 1 by 128 plus x 1 by 256, right. Now, if you see in these particular terms each of these basically contains the values of x 1. So, now if I take this as a common term actually this is nothing, but actually this is generic mathematics. If I take this x 1 by 2 plus x 1 by 4 as let us consider these as x 2, ok. So, now how I can represent the same equation?

So, this now this can be represented as x 2. So, if I just take 1 by 4 common over here, so this can also be written as x 1 by, sorry 2 plus x 1 by 4. Then, if I just take common of these, so this also can be written as x 1 by 2 plus x 1 by 4 plus again here if I take common, so x 1 by 2 plus x 1 by 4, right. So, if I just take this x 1 by 2 plus x 1 by 4 as x 2, so how can I write? X 2 plus this is also x 2. So, now it will become x 2 by 4 plus here x 2 by 16 plus x 2 by 64, ok. That means, now what I have to do? Initially only once I have to do this, x 1 by 2 plus x 1 by 4 that means, if I just do it x 1 by 2 and x 1 by 4, this is producing x 2, right. So, then to do this what I require? Then this basically again it will come this is basically right shift by 4 means this is by 2 bit.

So, I need another addition operation, where this is 16 means this is basically left shift by 4, then again I need to do this where this is basically left shift by 6 and then, if I just add, then finally I will get the same thing whatever is that. That means, y 1 a into x 1. The same thing I will get by this corresponding structure. If I take common of this x 1 by 2 plus x 1 by 4, that means now initially if I do not take common, so at that time how many

addition operation I require? That is 7 numbers, right. In this particular case, if I take common as this x 1 by 2 plus x 1 by 4 as x 2, if I take it common the factor has common, then how many require? It is 1 2 3 4.

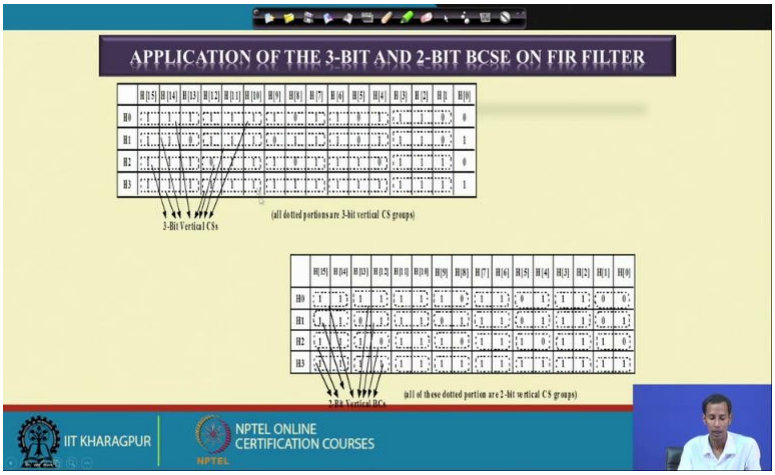
So, that means 4 number of addition operation I require now. So, that means number now becomes down to 7 to 4. So, for one particular multipliers, now this number has come down to 4. So, for 3 how many require? 12 number of addition operation where as initially it was 7 into 3 21 number of addition operation. So, that means I can save 9 number of addition operation, ok. So, these particular things whatever I did that x 1 by 2 plus x 1 by 4 which I have taken it common.

So, this particular things is basically known as binary common sub expression elimination algorithm. So, these are the basically sub expression which is related to this. So, based on that, now I can reduce the number of addition terms in this way, ok. So, this kind of structure is basically useful, very much useful whenever we do this reconfigurable constant multiplier design. So, that means now what is that? For one constant multiplier I am setting 3 number of addition operation. So, now more the number of constant multiplication is required for one filter implementation or for any transfer implementation. So, the more number of constant of this I will use, more the number of savings I will I can guarantee. That means, for 3 numbers that became 3 into 3, 9 numbers savings.

So, if the number is 10, so at that time 10 into 3 is that is 30 number of addition operation. So, the more in the number of that constant multiplier, more in the number of saving, ok. So, this is the fact that using this binary common sub expression, this is the beauty of this binary common sub expression algorithm which is basically used for constant multiplication operation.

So, if you just and here you see application, it can be used in any digital filter, then any transform which is basically FFT, DWT, DCT, DHT and as you see that this constant multiplication that can be represented in terms of that can be done in terms of binary common sub expression, then canonical sign digit based common sub expression, then this residual number system based common sub expression, then canonical sign digit based expression. All these things you can do.

(Refer Slide Time: 12:31)



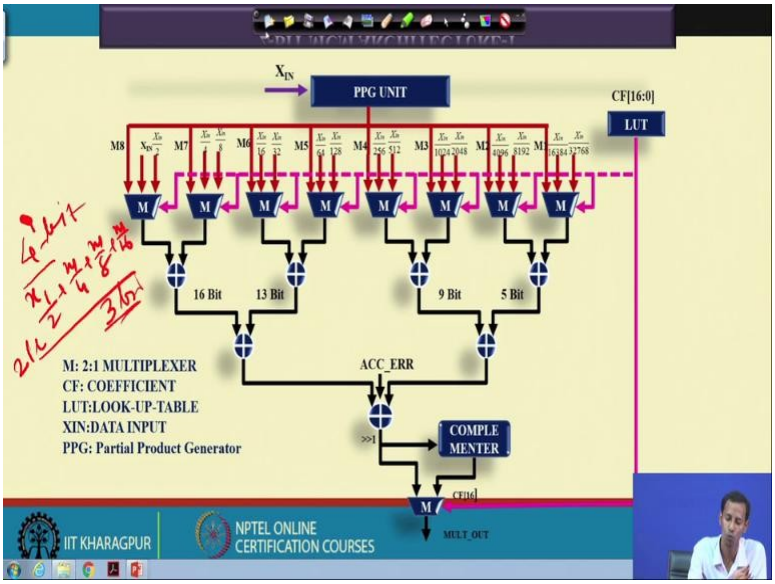
So, here what is this consider? This is basically considering this 2 bit binary common sub expression. So, y 2 bit as this x 1 by 2 plus x 1 by 4 if I take. So, at that time here actually it is taking x 1 a plus x 1 by 2. So, instead that you can also take x 1 by 2 plus x 1 by 4. Nothing is actually that is not a problem. It depends upon which common sub, sub expression you want to basically take it common, ok. So, as we are considering only two numbers over here, so that is why two number means that they are basically differed by their positions. So, this is 2 to the power minus 1 position. This is for 2 to the power minus 2 position, ok. So, that is why we are considering 2 bit binary common sub expression. So, if we take common 3 number of them, that means x 1 by 2 plus x 1 by 4 plus x 1 by 8. So, at time it becames 3 bit binary common sub expressions.

So, if you take it four times common, that means it became 4 bit common sub expression elimination algorithm. So, it depends on how many number of bits you want to take it common for this binary common sub expression, ok. So, it is not that the more the number of, that means common sub expression you will choose if the length of this BCS is more. That means, this number in the later case that will be reduced by divide of that; divide of that means here you see as I am considering 2 bit. So, that means this 16 numbers that has reduced to 8 now, ok. So, if I take it common 4 times over here, so at that time it will become the 4th terms will come over here.

So, that means 4 for doing this 4, I need 3 adder and here to doing this, then again I need another 3 adder. So, total 6 number of addition operation I require at that time. Initially here I require to do this 16 I require 15, but if I take 14 bit common, sorry 4 bit BCS, so at time 6 number of addition operation I require. So, if take 2 bit BCS common, so at that time for this I need 1 and 7. I will get over here. So, total 8 number of additional operation I require. So, that means more the number of bit you consider for BCS, the

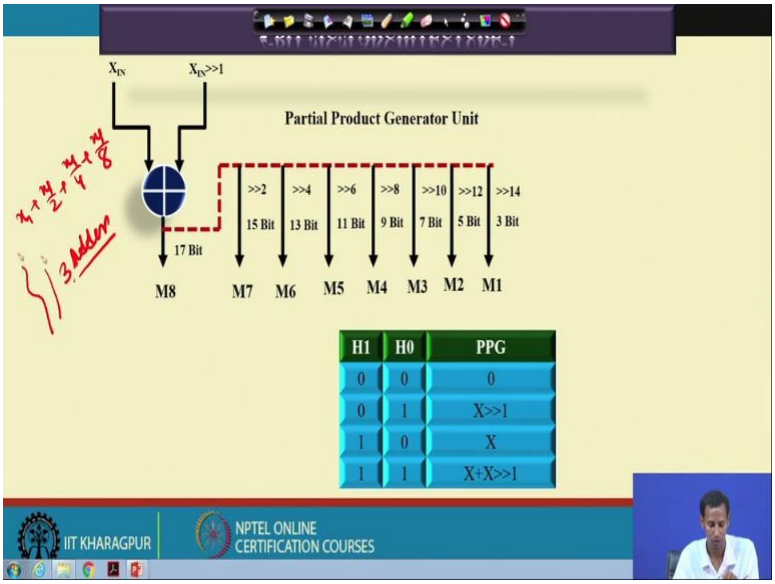
lesser the addition operation you require to do this constant multiplication, but the fact is that at that time the addition operation which I am not considering in this particular fact actually if I just see the corresponding structure of these, ok.

(Refer Slide Time: 15:45)



So, this is basically this for this x 1 plus x, x 1 plus x 1 by 2 that has been generated through this partial product unit and based on that in each of this additional operation is basically doing this.

(Refer Slide Time: 16:11)



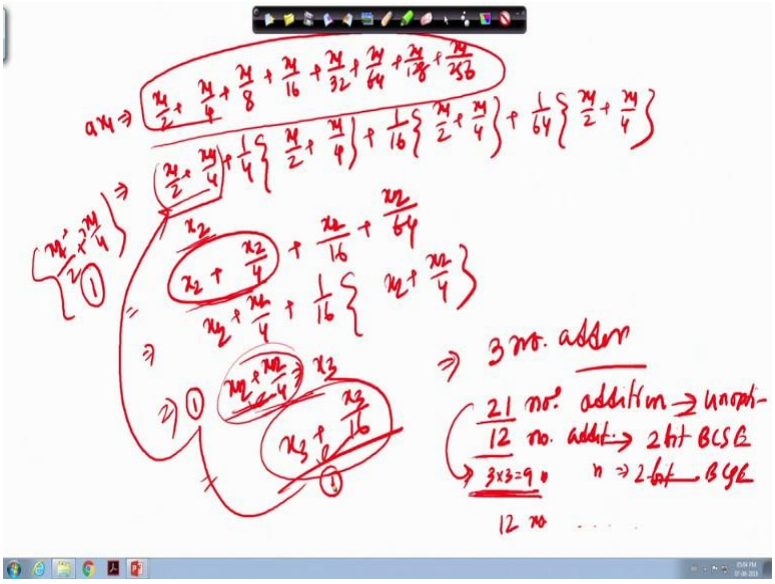
If I take 4 bit common so, at that time doing this x 1 plus x 1 by 2, I need only 1 adder right, but if I take 4 bit common, so at that time how many bits I need to add? So, at that time I need to add x 1 plus x 1 by 2 plus x 1 by 4 plus x 1 by 8. That means, at this I need 3 adders in a chain. So, 3 adder and then, again this particular this will be then adder in a chain. So, then again for this another 4 bit, that means 4 of this addition operation x 1 by 2 plus x 1 by 4 plus x 1 by 8 plus x 1 by 16. This requires another three levels of addition operations. Sorry not three level, this requires basically two level of additional operation. So, that means total four number of additional operation I require.

So, that means it is not the fact that more the number of BCS I will choose and I will get the benefits in terms of the area will be minimized. At that time, the number or addition operation will be minimized, but speed wise the addition operation in a chain that will be maximized at that time, ok. So, that is why it is not that you need to choose this size of BCS in such a way, so that you are getting in terms of this speed as well as the area in both the things if you are getting the benefits so, at that time you choose the BCS according to that.

So, according to that we have seen that this 2 bit BCS or 3 bit BCS. They are the like the max. That means, they are the most useful BCS terms or the length of the BCS is more that means, the advantageous one in terms of speed as well as the power and area for constant multiplication, this reconfigurable constant multiplication design, ok.

So, this is one of the technique used for this constant multiplication. Now, again you see another thing over here. How can I now what is happening? Now the thing is that I require total what I said. That means, for this I require 12 number of addition operation, right. So, can I reduce this number more? Can I reduce it? Yes I can reduce it.

(Refer Slide Time: 19:37)



How can I reduce it? Then again if I just write this particular terms a into x 1, that is x 1 by 2 plus x 1 by 4 plus x 1 by 8 x 1 by 16, ok. So, what I did initially? X 1 by 2 plus x 1 by 4 plus 1 by 4 x 1 by 2 plus x 1 by 4 1 by 16, that is x 1 by 2 plus x 1 by 4, then again 1 by 64 that is x 1 by 2 plus x 1 by 4, right. So, what I did? X 1 by 2 plus x 1 by 4. I write that as a x 2. So, this will become x 2 plus x 2 by 4, then x 2 plus x 2 by 16 plus x 2 by 64.

So, in a next level if you see these corresponding terms x 2 plus x 2 by 4, if you take common, then x 2 plus x 2 by 4, that means what x 2 plus x 2 by 4. If I take that as x 3, so this can be written as x 3 plus x 3 by 16. So, that means to do this x 2 plus x 2 by 4, I require what I require only one adder here. I require one adder and to do this x 2 plus, sorry x 1 by 2 plus x 1 by 4 I require one adder. So, that means total 3 number of adder operation I require to do this whole addition operation.

Here I require how much? I require total four number of, so that means here if I take another level common, so that means at that time I require only 3 number of addition operation. So, that means now initially how many I require? 21 number of addition operation, right that is for unoptimized case. So, after doing this 2 bit BCSC, so that number becomes 21 and then, again if I take two level of common, so at that time it requires 3 into 3 means 9 number of addition operation. So, the same this is 2 bit BCSC, but the common I have applied on two directional which vertical once and horizontal

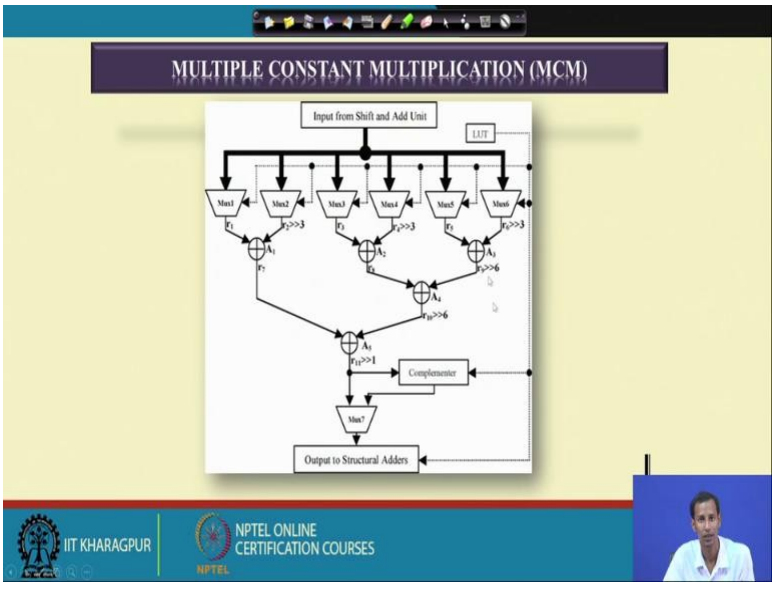
once, ok. So, how vertical once how horizontal once that, I will show you in the, that means presentation, but here the number will become 9 now.

So, initially we have started with 21 number of additional operation. So, after 2 bit BCS, we have found out the number s 12. So, after that we have come down to the number s 9. So, that means total how many number? 12 number of additional operations we are savings for these 3 particular things. So, that means the more the number of more the, that means constant multiplier we use, more the savings, we can guarantee using this binary common sub expression for constant multiplication operation, ok. So, this is the beauty of this particular algorithm and this particular method, ok.

So, here you see this is this is the, that means this two dimensional factor. So, here if I consider this 4 coefficient, so suppose I consider and the length of this is 16 bit. Length of the coefficient if that is 16 bit, so 3 bit will be applied like this as this you see. So, each 15 each, 14 each 13 if this will be applied something this if we have to use this 2 bit BCS.

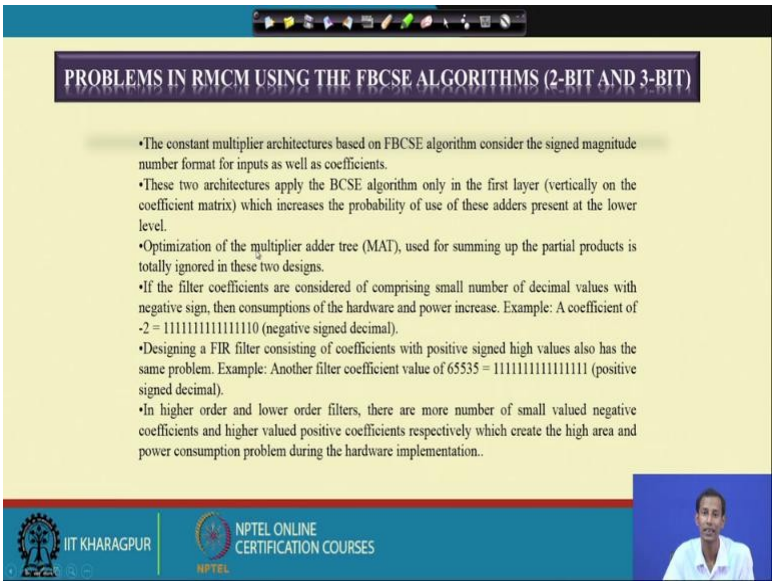
So, at that time it will consider only two consecutive bits. That means which is 15, each 14, 13 is to l, each 11 is 10, each 9 is 8 something like this. So, instead 3 bit, it will consider 3 consecutive bit which is each 15 a h 14 a h 13, ok. So, considering this fact I will get the circuits something like this. The architecture for this is for one single constant multiplication and then, for this is for 3 bit.

(Refer Slide Time: 25:11)



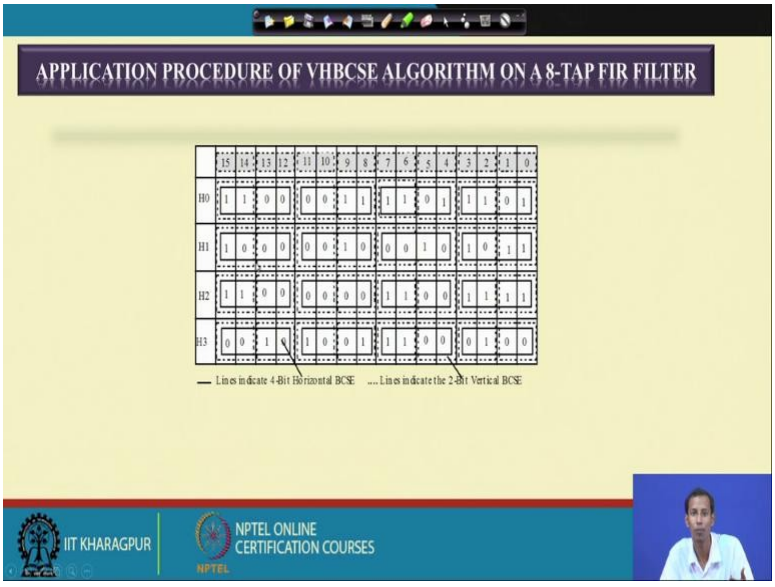
One single constant multiplication.

(Refer Slide Time: 25:13)



Then, as what I said this basically suffers from that means if I consider another level of optimization, so at that time I can reduce the number more.

(Refer Slide Time: 25:25)

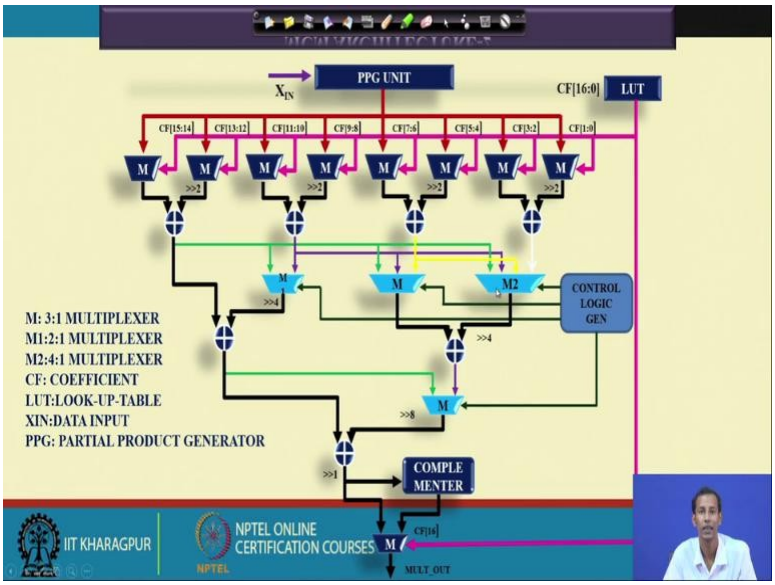


That means, here what we do? That means, initially we did the, that means initially what happens at the very fast level considering this 2 bit, we have applied the common sub expression once, then what we did this among this. That means, here you see which

direction I am basically applying this corresponding this common sub expression that is in the vertical direction.

Now, this among these among these horizontal direction, that means between each of these coefficients if I have to find out another common sub expression, so at that time I can reduce the number more by another one if I consider this 8 bit coefficient, ok. So, that we have already seen in the previous case and based on that we can come down to this another architecture.

(Refer Slide Time: 26:35)



That means, here this will basically skip the multiplication operation. The multiplication operations are there, but if I found that more number of common, so at that time that particular multiplier will be used skipped. It will use the shifted version of the pervious results, means what actually what we see? Here from here, what we have seen that, after doing this a common taking, common of this x 1 we are getting this x 2, then again we are taking common of this x 2 terms to get the terms reduced by only 2, ok.

So, that when we will get the terms common in terms of x 2, then only I will I can come down to these x 3 terms, ok. So, in this way we are basically doing the optimization in the reconfigurable constant multiplier design.

So, this is just to using the binary common sub expression. So, if you can use any other number method instead of binary, you can use this residual number system and then, you

can apply this common sub expression to get the results more or you can use this canonical sign digit based format number representation and then, you can apply this common sub expression to reduce the number in a more way. So, that means this is the way of doing or this is the techniques that means, step by step technique how we can reduce the number? The number of this addition operation in a better way, ok. So, this is the end of this multiplier chapter, ok.

So, from tomorrow we will or from the next class onwards, we will start to see about this different architecture or different that means structure for doing this. That means, now we have build up the basic operation like adder design, we have now we have gone through the multiplier design, we have gone through the basic these algorithms part, we have we have already gone through that. That means, the 2 x, then 2 x plus 1, then 4 x, then log 2 base x. So, that kind of simple blocks structure efficient way to design that we have already done. Operator wise we have already designed or we have learned the fact.

Now, based on these particular operators, now we can design the functions, ok. Now, we will try to see the architecture of the function which are basically used different of these operators. So, that means now what this function level architecture is. Now, I will try to optimize and in this function level basically architecture, they require this operator level architecture. So, operator level architecture optimization, we have already seen. Now, function level optimization we will see. So, that means there will be two level of optimization.

So, from the next class, we will try to see different architecture of function like FFT Fast Fourier Transforms, then this Cordic Architecture, ok. So, what they have used and what are the architectures available for and how you can develop the architecture for that, that we will see from the next class onwards.

So, thank you for today.