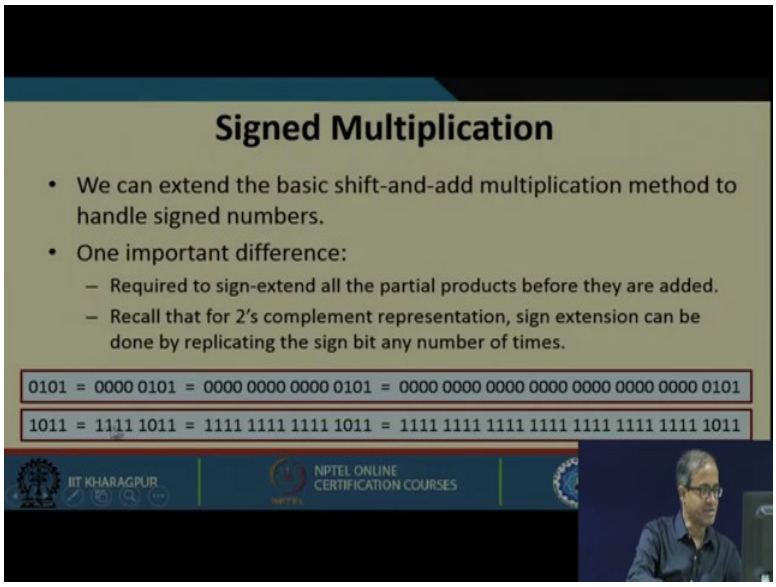
Computer Architecture and Organization
Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 36
Design of Multipliers (Part 2)

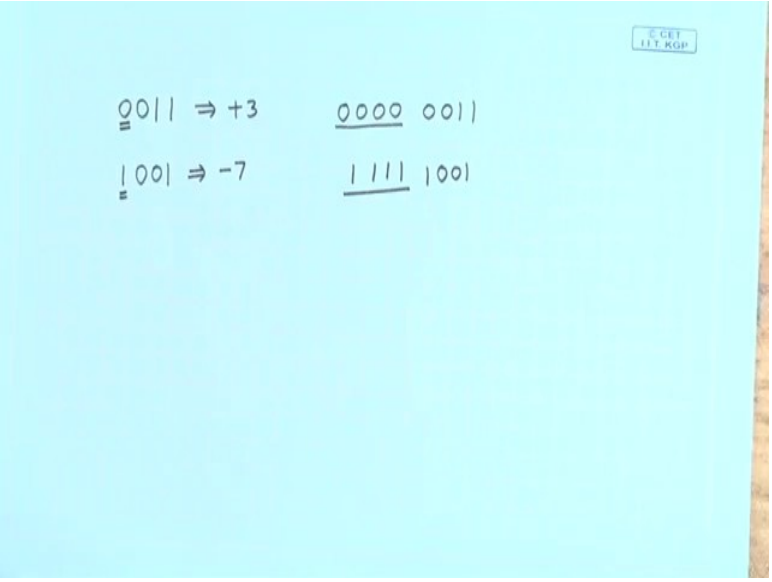
In the last lecture we have seen how to carry out unsigned multiplication. We continue our discussion in the present lecture. We shall see how we can carry out signed multiplication, multiplication of two numbers which can be either positive or negative specifically in 2's compliment form. So, how we can do that?

(Refer Slide Time: 00:45)



So, our topic of discussion today's signed multiplication. The first approach that we talk about is a simple extension of the basic shift and adds multiplication method that we have already seen, but here there is one important difference depending on the sign of the multiplier or the multiplicand. So, you will have to sign extend all the partial products. Now you recall earlier being it was discussed that any signed number can be sign extended to any number of bits.

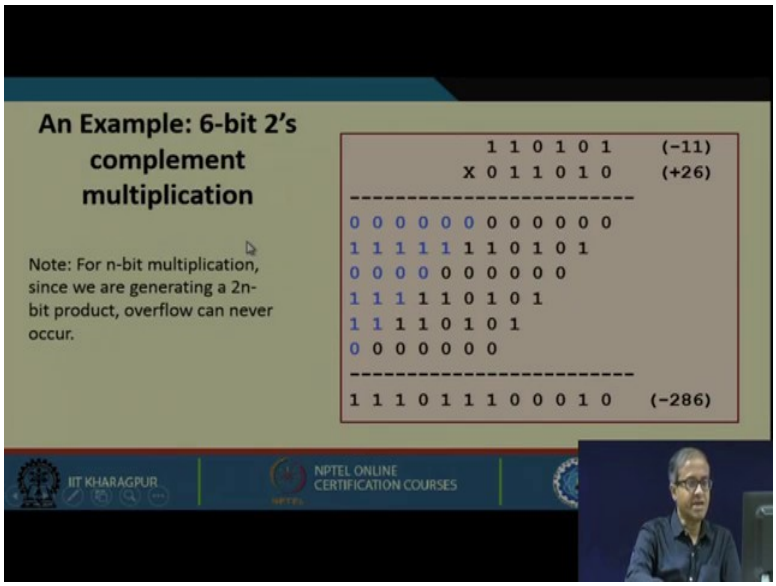
(Refer Slide Time: 01:50)



So, what is the rule for sign extension for those compliment numbers? If the number is positive you can fill the number with as many 0's you want. If the number is negative, you can fill it with any number of 1's you want. Like for example, 0 0 1 1 represents +3 in 4 bits; suppose you require to represent in 8 bits, you simply add four 0's in the beginning; that means, you are simply extending the sign of the number to these additional bits.

Let us see another example; 1 0 0 1 represents -7 in 2's compliment. If you again want to represent it in 8 bits, you look at this sign it is 1, you replicate this sign bit. The value of the number remains same in this process. This is actually what you have to do. The example is shown here; also let us say number 0 1 0 1 --- this is a positive number, you can sign extend it to 8 bits, sign extend it to 16 bits, sign extend to 32 bits like this. Similarly a negative number --- sign extend to 8 bits to 16 bits to 32 bits, just replicate the sign bit as many times you want.

(Refer Slide Time: 03:15)

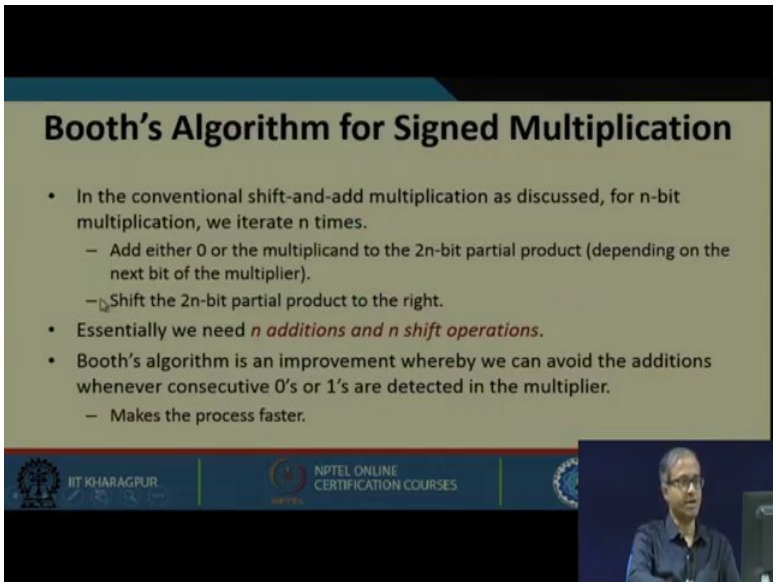


So, in the shift and add extension for signed numbers, we just use this principle. We are multiplying -11 (is the multiplicand) that is a negative number, with +26. We represent them as 6 bit numbers. So, the result is supposed to be 12 bits. So, when the first bit is 0 0, multiplied by this will be 0. So, we are sign extending it by adding 6 0's in the beginning to make it 12 bit partial product.

Next is 1. So, we add this, this is negative that is why we use a sign extension to it by all 1's. Next again 0, 0 sign extension; next is 1, 1 sign extension, and so on. So, if you add these bits, you will see that what you have got; this is indeed -286 that is the product.

The simple shift and add we can extend by using this sign extension concept. So, we will be getting correct result in terms of the sign of the product, and another thing I just mention earlier also. You are also assuming one thing that in case of multiplication since you are assuming that two n bit numbers are being multiplied to generate at 2n bit product. So, overflow can never occur here because the product can be at most 2n bits not more than that.

(Refer Slide Time: 05:25)

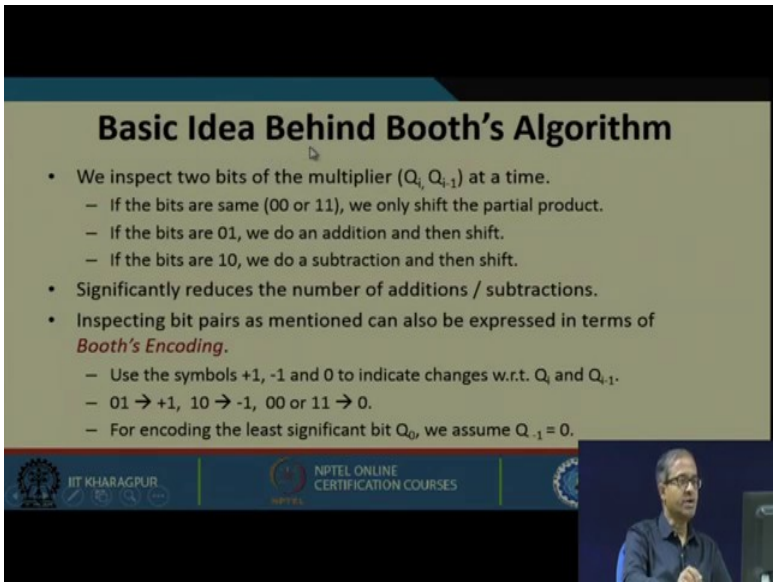


Now, we look at some improvement to this basic algorithm that we have just now seen.

In this shift and add method if you just recall the basic mechanism what you are doing. Our multiplier and the multiplicand are both n bit numbers; we are repeating the process n times, inspecting one bit of the multiplier every time, you are either adding the multiplicand or adding zero. So, we need n number of addition steps, and n number of shifting steps in the basic shift and add method. Now what we discuss here is that essentially this is also a type of a shift and add multiplier, but we are trying to reduce the number of addition steps by using some kind of bit encoding technique.

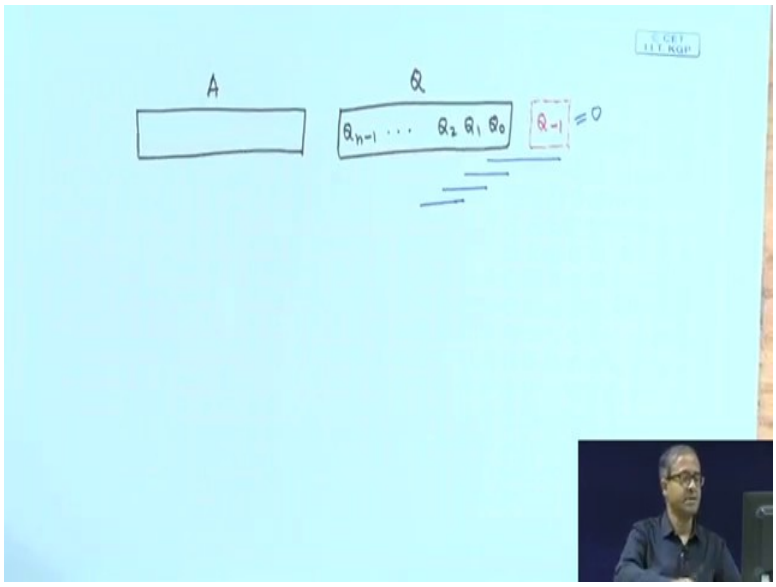
The method we talk about now is called Booth's algorithm; and can be used for signed numbers also. So, as I just now said in conventional shift and add for n bit multiplication, we have to repeat the process for every bit n times, you either add 0 or the multiplicand to the partial product at every iteration and also you shift. So, we need n additions and n shift operations; and as I said in Booth's algorithm we are trying to avoid additions wherever possible. Very specifically whenever there are consecutive 0's or consecutive 1's in the multiplier, we avoid additions; this can make the process faster.

(Refer Slide Time: 07:30)



Let us look at the basic idea.

(Refer Slide Time: 07:36)



We assume we have a temporary register called A; we have a quotient register Q. So, initially the quotient register is loaded with the quotient. So, it will be Qn-1 .. Q2 Q1 Q0. Now here we are adding another single bit register, this we are calling as Q-1. We will see very shortly in Booth's algorithm we do not look at one bit of the multiplier at a time, but rather we look at pairs of bits Qi and Qi-1, for all i. So, you see earlier we are looking

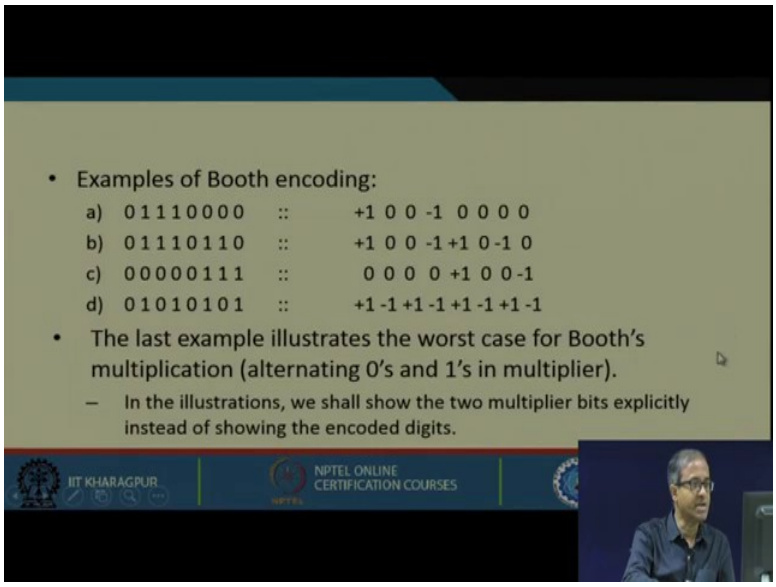
at Q0 first then Q1 then Q2, but now we shall be looking at pairs Q0 Q-1. We need to add another flip flop at the end which is initialized to 0.

Now, the rule of multiplication is very simple, we look at this pairs of bits. So, if the bits are the same, either 0 0 or 1 1, no need to do any addition or subtraction, just only shift the partial product. If the bits are 0 1, then you do A = A + M and then do a right shift. If the bits are 1 0, do A = A - M and then right shift. Here I am not going into the formal proof of the Booth's algorithm, the method looks very simple. So, if you work out you can also verify that whatever you are doing here is actually carrying out multiplication.

By doing this whenever we find 0 0's or 1 1's in the bit pairs, we avoid addition or subtraction. This can significantly reduce the number of additions or subtraction. Now in some textbooks you will see that instead of looking at these bit pairs, they have used an alternate notation called Booth's encoding, they have used the symbols +1, -1 and 0 to indicate the status of these bit pairs; whether they are changing or not changing.

If the bits are 0 and 1 you code it as +1, if it is 1 0 you call it -1. If it is 0 0 or 1 1 it does not change; you call it 0. I have said that for coding the last bit Q-1 that you added here, is initialized to 0. Let us take some examples of Booth encoding technique. Suppose my multiplier was this.

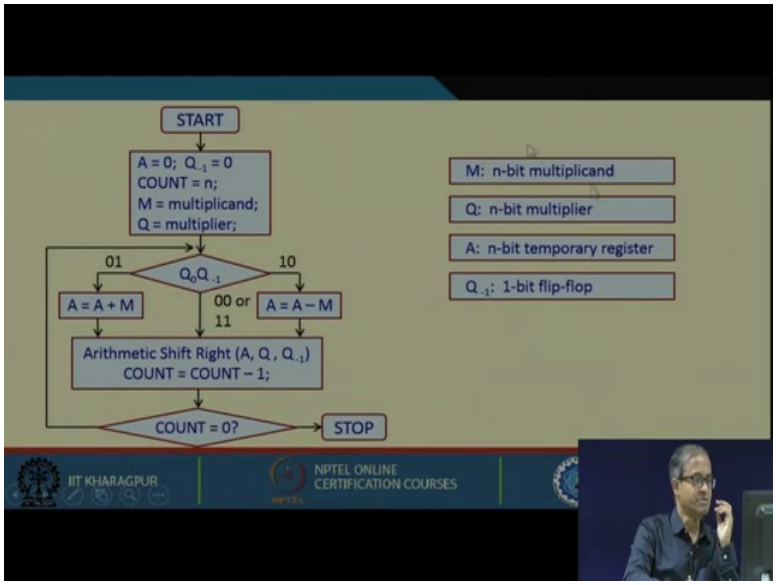
(Refer Slide Time: 11:21)



So, there will be an additional 0 here, that Q-1. So, 0 0 is coded as 0, this 0 0 is coded as 0 this 0 0 0 0 two more zeros, 1 and 0 will be coded as -1, 1 1 will be coded as 0, 1 1 will be coded as 0, 0 1 is coded as +1. So, is a essentially looking from right to left; in this encodings scheme 0 means that we need only shifting no addition or subtraction, -1 means we need a subtraction, +1 means we need an addition. Another exampl; last two bits are 0 0 this is 0, 1 0 is -1, 1 1 0, 0 1 is +1, 1 0 is -1, and so on.

Now, the last example this is the worst case for Booth's algorithm alternating 0's and ones. So, we will find here you have not got any 0's; all are +1 and -1. So, we will have to always do either an addition or subtraction. This is the worst case of Booth's algorithm, but on the average you may have to do less additions.

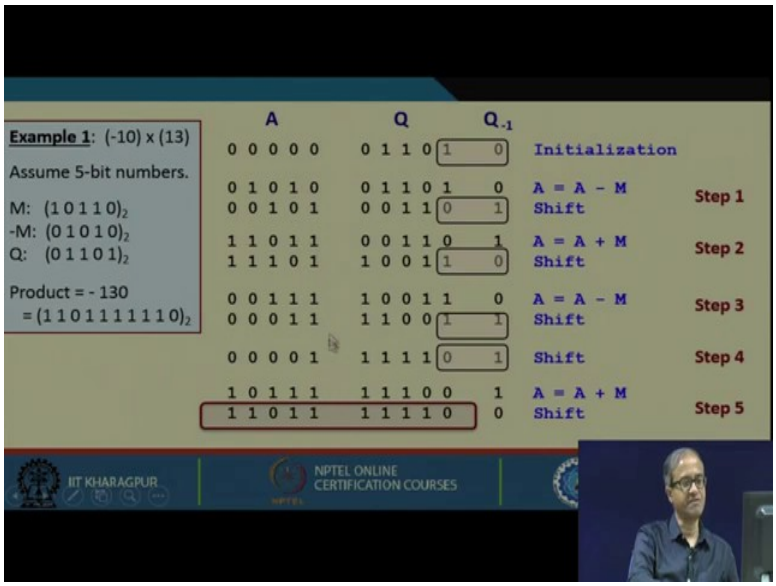
(Refer Slide Time: 12:58)



Booth's algorithm goes like this in flowchart form. I have already shown you that A is a register of n bit, M holds the multiplicand, Q holds the multiplier, and Q-1 is a one bit flip flop initialized to zero. So, you inspect Q0 and Q-1 every time.

So, if it is 0 1 then you have to add, if it is 1 0 you have to subtract, but if it is 0 0 or 1 1 you do not have to add or subtract.

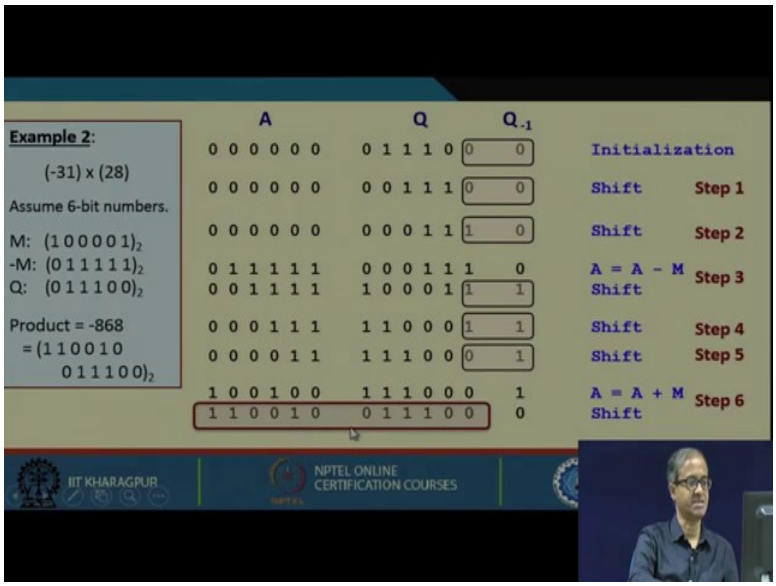
(Refer Slide Time: 14:44)



So, the algorithm is very simple. Let us work out some examples. As I said if there are consecutive 0's and 1's, no addition or subtraction are needed. So, you can skip over those runs of 0's and 1's. Let us take an example. Multiplicand is negative -10. This is 13. This M is 1 0 1 1 0, -M this is 2's complement 0 1 0 1 0. So, whenever I have to subtract I will actually add -M. So, I am showing both M and also -M.

So, here the product will be -130. This is our initial thing you load with the quotient A is 0, Q1 is also 0, you check the bit pair it is 1 0. So, 1 0 means we have to subtract A = A - M. So, you are adding -M to A. Then you do a shift. The process repeats.

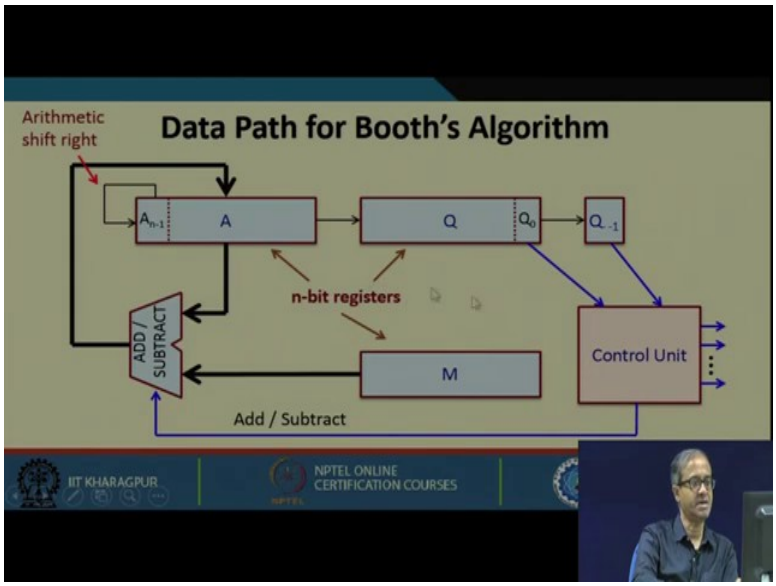
(Refer Slide Time: 17:15)



And sometimes you are skipping addition or subtraction step doing only shift. Let us take another example -31 and 28, this is your quotient and this is your multiplicand, -M is this.

So, the product is supposed to be -868. So, you check the last 2 bits 0 0. So, only shift no addition next two bits are again 0 0. So, only shift, 1 0 means are subtraction. So, we add –M, then shift; again 1 1, so no addition subtraction only shift. Again 1 1, so only shift. Finally, you have 0 1 means addition, and then a shift. So, you are done this will be your final result. So, you see the Booth's algorithm is simple in this example you have seen that you are able to skip 1 2 3 4 times; only twice you needed to do some addition or subtraction.

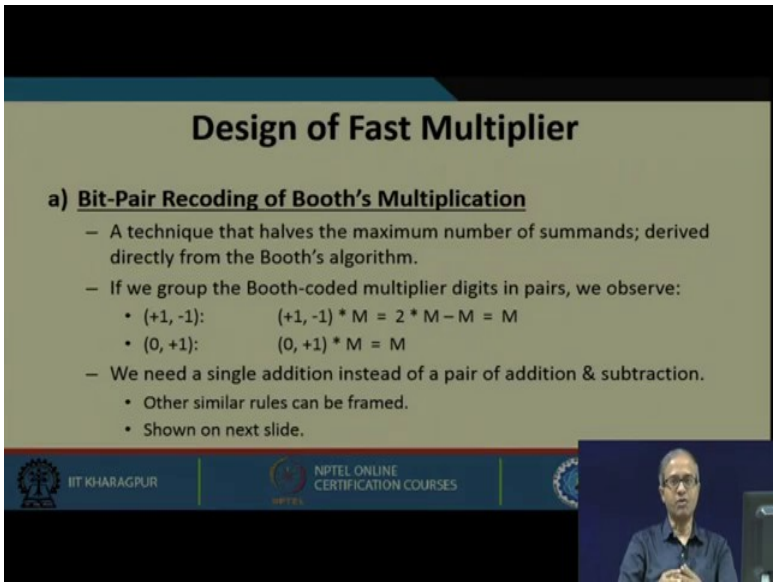
(Refer Slide Time: 18:39)



This method is significantly faster as compared to be previous shift and add multiplication method. In terms of the hardware requirement it is very similar. You need a register a Q, and one additional Q-1 flip flop here. Earlier you needed only an adder, but now we have an adder or subtractor because sometimes you also need to add and here while you are doing a right shift like in the previous example, when you are doing a right shift you are doing an arithmetic right shift. So, the sign bit would be replicated.

So, that sign bit it is coming back when you are right shifting this is for arithmetic right shift, and the control unit will be checking both Q0 and Q-1 together, and we will decide whether to add or subtract, or whether to skip this step. The control signals will be generated accordingly.

(Refer Slide Time: 19:38)



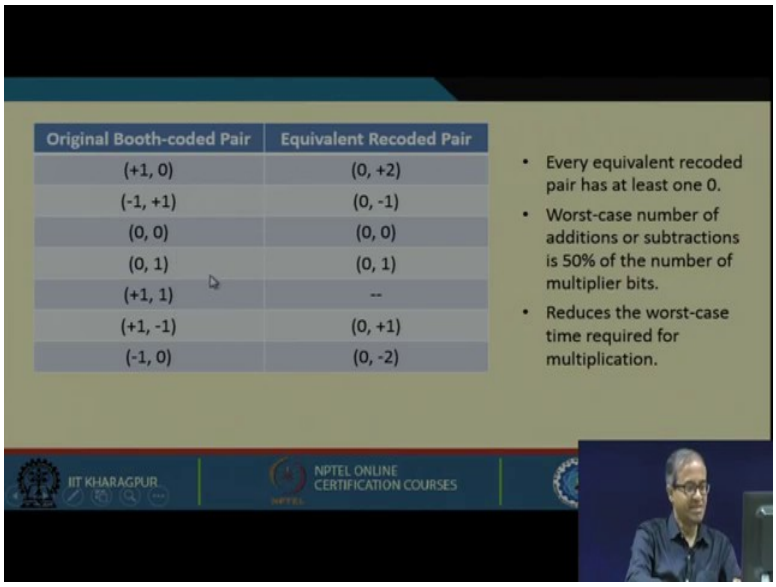
Now, let us look at how we can improve the speed of the multiplier even further. We have already seen how both multiplier can reduce a number of additions or subtractions. Now we look at a modification of Booth's multiplier here, this is called bit pair recoding of Booth's multiplier. So, this method effectively halves the maximum number of addition or subtraction.

In Booth's algorithm we have seen for that 0 1, 0 1, 0 1, 0 1 multipliers scenario, in the worst case you need n addition or subtraction. In this new method, what we are saying is that in the worst case maximum number of addition and subtraction will be only 50% of what was there earlier. So, we are reducing the worst-case complexity of addition subtraction to almost half.

The observations are like this. In the original Booth encoding what we have seen; let us say we have two symbols +1 -1 one after the other; +1 means addition, -1 means subtraction of M. Now +1 coming before means that I have to make one left shift M. Left shift of M means 2 * M, and then we subtract M. So, +1 -1 both operating on M means effectively 2 * M - M; because this +1 will be shift left by 1, effectively it means multiplying by 2, and -1 will be lower significant - M. So, effectively this means M. You also observe if we have a paired 0 +1, this also means M because you multiplied by M in the lower place in a higher place it is 0, you do not do anything it remain same.

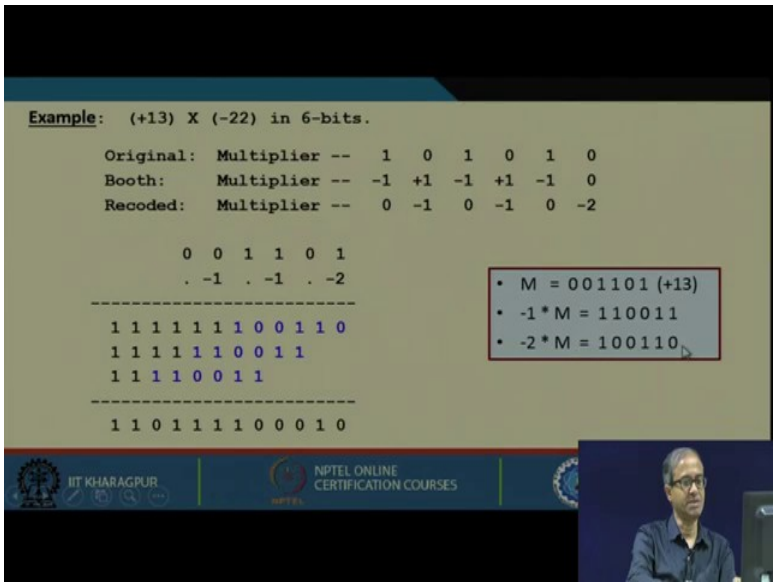
The final result of these two are equivalent. So, wherever you have +1 -1, you can as well replace them by 0 +1, which means you have brought in an additional 0 which means one less additional subtraction. There are other similar rules you can frame.

(Refer Slide Time: 22:16)



We are showing the rules here in the next slide. Just like we have shown here you can similarly try this out. +1 0 can be encoded as 0 +2; means here you are multiplying by 2. So, this new symbol +2 and -2 comes in this new method. +2 means shift by 1 position. In the modified encoding at most 50 percent times will be doing additional subtraction, rest to all times you will be doing shifting.

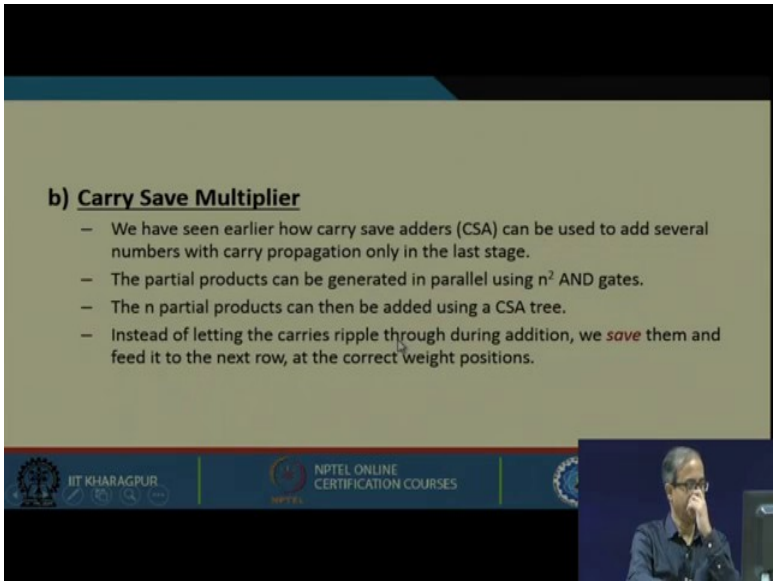
(Refer Slide Time: 23:36)



So, this reduces the worst case time required by the Booth's multiplication algorithm. So, let us take an example +13 x -22. The process is worked out step by step.

So, you see the process of multiplication becomes much simpler here; only 3 steps are required, the other 3 steps you can skip.

(Refer Slide Time: 25:41)

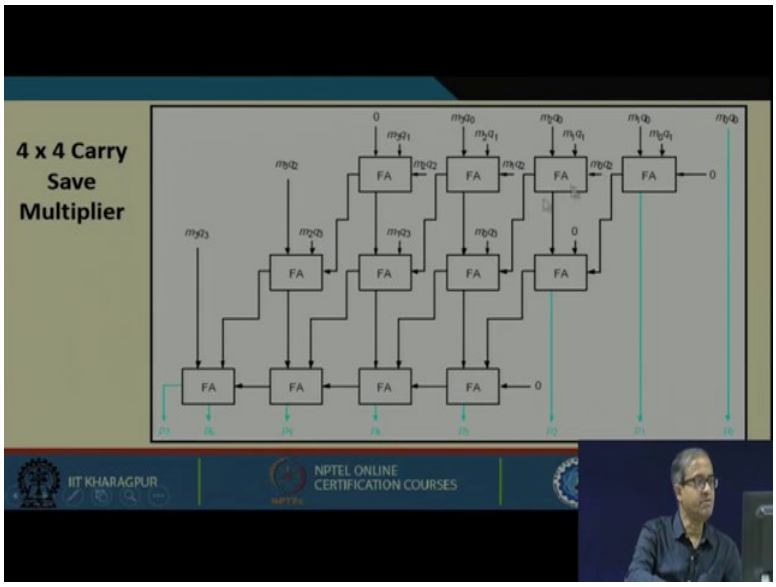


The last kind of multiplier that we talk about is called carry save multiplier. See earlier we looked at carry save adder. We have seen that we can use carry save adder to add several numbers without any carry propagation; we are adding the carry only in the last

stage and also we have seen, in the combinational array multiplier that the partial product generation requires n^2 AND gates.

What we say here is that we can use a tree of carry save adders to multiply the n partial products. To add the n partial products, we need a carry save adder tree. Now see what we saw for carry save adder is that a single carry save adder can add up to 3 numbers, if we want to add more number of numbers like 4 5 6 you will require several carry save adders. Now for multiplication also there will be n partial product you will have to add all those n partial products. So, you can have a similar carry save adder tree where all those n partial products you are adding and finally, at the end you get the final product.

(Refer Slide Time: 27:10)

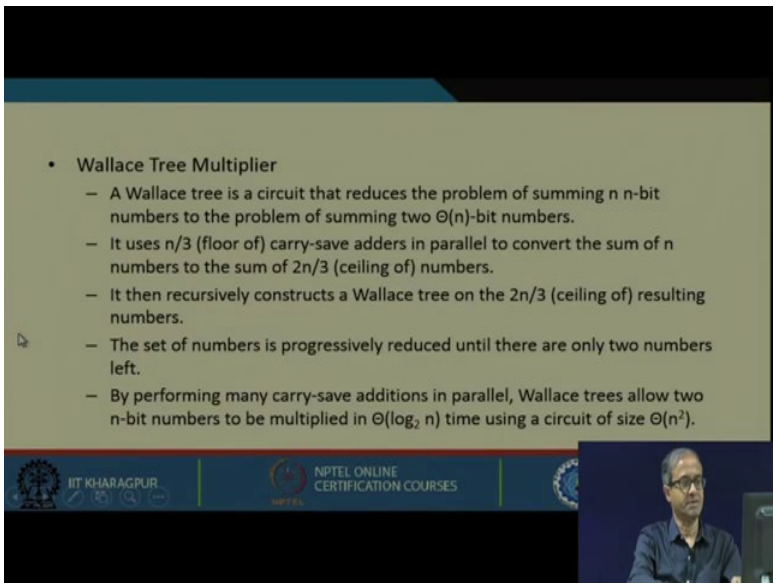


So, here we show an example for a 4 x 4 multiplier. These are independent full adders, the first two stages indicate carry save adders there no connection in between. So, you see that all the partial product generated by the AND gates are fed here this is M1 Q0, M0 Q1, M0 Q2, M1 Q1, M2 Q0.

So, if you can check you will see that exactly the same thing is happening and when the carry output is going to next stage, you are doing a one bit shift and then connecting so that the shifting can be implemented. And some of the product terms are generated directly; for example, M0 Q0 means the LSB of the product. The first full adder can generate P1, the next bit this full adder can generate P2, but in the last stage you will be

needing a regular adder with carry propagation. So, here I have shown a ripple carry adder this can also be a curry look ahead adder.

(Refer Slide Time: 28:34)

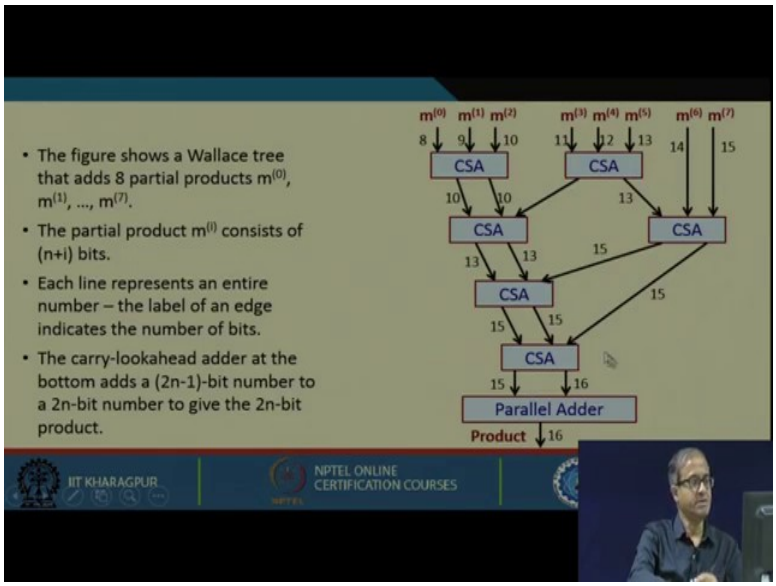


So, this will be generating the remaining 5 bits of the product. So, if we have a circuit like this you can carry out multiplication much faster using carry save adder.

You have an alternate method called Wallace tree. Wallace tree is very similar in concept. It is a tree structure that reduces the problem of summing n numbers to the problem of summing two numbers, which are of size Theta(n). So, this also uses carry save adders; it uses the floor(n/3) many carry save adders to convert the sum of n numbers to the sum of ceiling(2n/3) numbers. We are not going into the detail of this mathematics; we shall just show some example Wallace tree formulation for a particular multiplier.

Using many carry save addition in parallel Wallace tree will allow 2n bit numbers to be multiplied in Theta(log n) time, this is important you are having a logarithmic time multiplier, but the circuit complexity will become Theta(n^2).

(Refer Slide Time: 29:53)



So, this is one Wallace tree example that I am showing, which is adding 8 partial products M0, M1 up to M7. Suppose we are doing 8 x 8 multiplication. So, the final result should be 16 bit. These numbers beside these edges will indicate number of bits that are being generated; here this will also be 13. These are the partial products, these are the carry save adder tree you generate, some of the carry save adders are working in parallel, but these two are sequential, and finally you have a parallel adder.

So, what Wallace tree says --- it is a particular way of arranging the carry save adders, so that maximum amount of parallelism can be exploited and the depth of the tree is reduced. So, how many carry save adders maximum can be used; this is just an example I have shown. So, with this we come to the end of this lecture.

We have so far looked at the design of adders and multipliers. One thing is left among the basic arithmetic operations, namely division. In the next lecture we shall be looking at some algorithms for division and how they can be implemented.

Thank you.