Digital Circuits and Systems
Prof. Shankar Balachandran
Department of Electrical Engineering
Indian Institute of Technology, Bombay
And
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Module - 51
Fast Adder

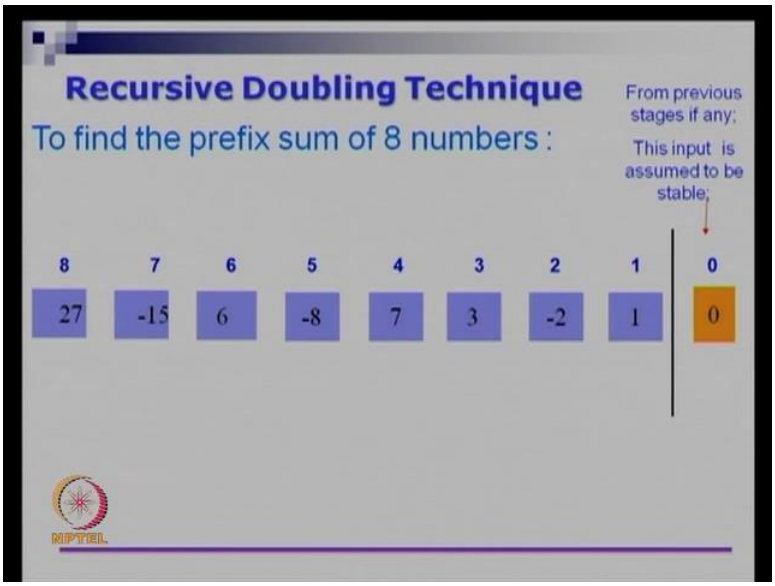
Welcome every one, we are at module 51 of this course of the 9th week and in this module, we are going to look at Fast Adder. So, one thing want you to go and recollect is, when we looked at the ripple carry adder, I said the carry has to propagate all the way from the LSB to the MSB in the first case, in the worst case. So, this may not happen for all the input combinations that you give.

But there do exists input combinations where a carry add; that is coming from the LSB stage, may have to ripple through all the different stages and have to go to the final stage to generate the carry and sum. And if you make a wider and wider adder, this becomes a problem. So, what we will do in this a lecture is, we will go and look at mechanisms by which we can cut down on the delay of this.

So, I am going to talk about one specific fast adder. So, by fast I mean it is not going to be linear, the number of the delay, the delay that will be required by the circuit will not be linear in terms of the size of the addition. So, if I want to do n bit addition, preferably I want something which is logarithm n base and logarithm n is huge reduction. So, if I am going to look at let us say 32 bit addition I will need 5 gate delays, if I have 64 bit additions I will need 6 gate delays and so on.

So, something like that is good, because then you can make wider and wider adders, your clock frequency will not be affected so much. So, I want to talk about one specific technique here and we will see the motivation, we will see an analogy of how to do something and we will see how to use that in adders. So, the first problem I am going to talk about, so I am not going to talk about bits to begin with. So, I am going to talk about a specific problem here.

(Refer Slide Time: 02:00)



So, let us say I have 8 numbers. So, let us say these are the 8 numbers that are given and let us say from in software, let us say we are going to do this, I have an array of size 9 and I have numbers from location A of 8 to A of 1, which has some numbers. And let say I have stored A of 0 to I have 0, let say, this is what I have. I have an array of 9 numbers, so location 0 to 8 and I have specific values in it.

So, what I want to do is the problem definition is to find the prefix sum of 8 numbers. So, this problem may seem unrelated to addition now, but I will go and connect it later. So, let us say I want to find a prefix sum. So, the prefix sum is defined as what I want is, at A of 8, I want the summation of all the values from here to here. So, all the 9 locations, I want the value stored in A of 8.

For A of 7, everything from A of 7 to the right side, I want to store it in A of 7, for A of 6, everything including A of 6 and to it is right side, so I want all of that to be stored and so on. So, this is called the prefix sum problem. So, it is called prefix, because if you are looking at the final values that is required, the sum that should be at A of i should be everything from A of 0 to A of i. So, that is why, it is called the prefix sum. This is the problem and let say, I want to solve this problem.

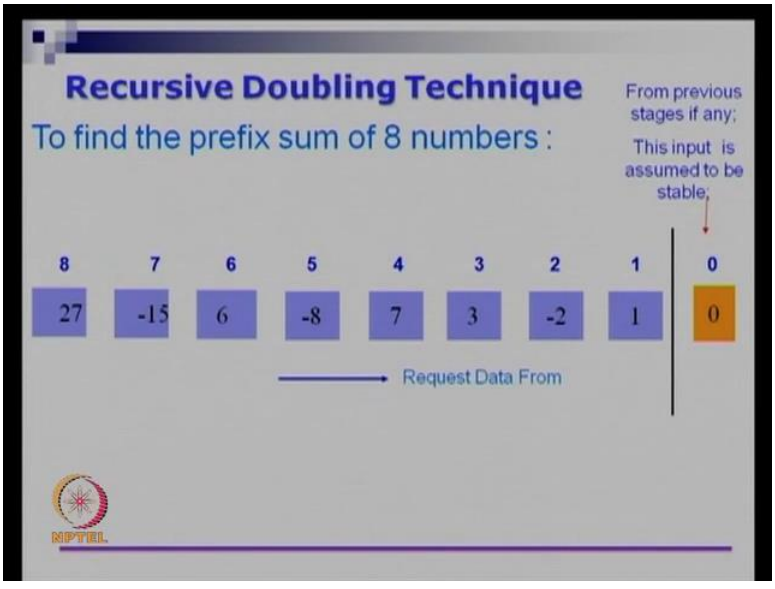
One way to do that is start with here, take this number, add it to here. So, that will make it A of 1 to be 1 and take A of 1, add it here that will make A of 2 equal to minus 1. Then, take that minus 1, add it to 3, add it to A of 3, which is also 3; that will make this do, then take A of 4, add the result from here, so that will make it 9 and so on. So, what I am

going to do is, I am going to start from the right side and if I, so calculate the sum so far and add it to A of i, then that will give me A of i to 0, it would end up in that result.

If I do that, so then to add all these n numbers, I have to start from here, do one addition at a time. Take the sum put it in the array, take that sum and add it to the next element, find out the new sum, put it in that array location, take that new sum and add it and so on, I will need n steps to do this. So, prefix sum, if I do it in the directory, it will require n steps, if I want to have n elements for which I want to calculate the prefix sum, then I will need an array of size.

So, I will have an array of n plus 1 and I will need linear number of steps as there are elements. So, the first thing I am going to do is, I am going to come up with the parallel way of doing this. So, I will assume that A of 0 is always going to be 0 and technically, I am only interested in A of 8 to A of 1. Let us assume that A of 0 is 0 by default always and I am going to add A of 8 should be summation of all the eight elements, A of 7 should be the summation of the 7 elements from 7 to 1 and so on. So, to do this I am going to do something in parallel and I am going to call this the recursive doubling technique. So, let us see why this is called so.

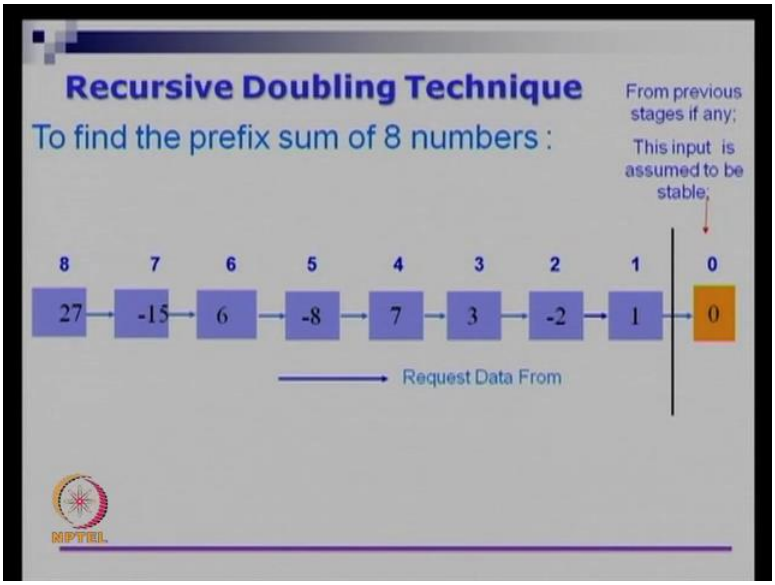
(Refer Slide Time: 05:28)



So, the first thing I am going to do is, I will assume that each element can somehow communicate with neighboring element and can request data from the neighboring element and add it to itself. So, I am going to draw an arrow to say which element is communicating with which, so let us assume that, these are elements that can talk to each

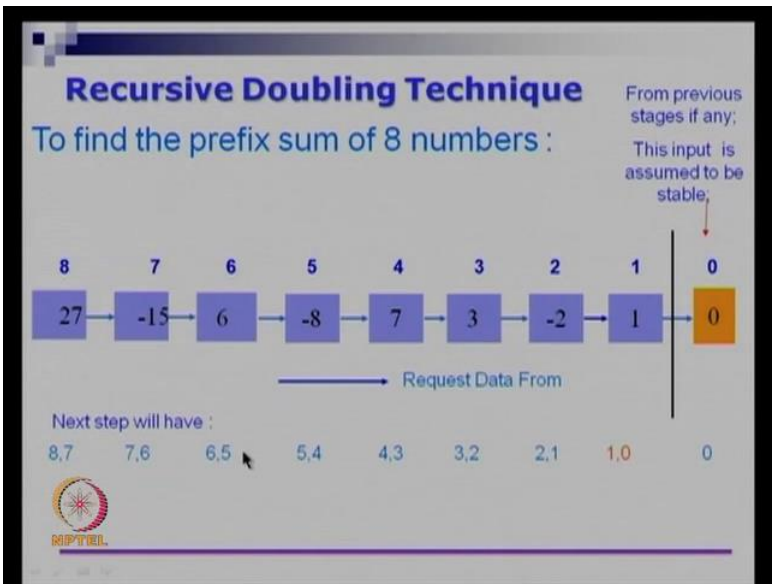
other.

(Refer Slide Time: 05:48)



So, then what I will do is, I will make element A of 8 communicate with the A of 7 and get the value at A of 7's whatever is stored, give it here. Simultaneously, A of 7 will talk to A of 6, take the value and store it here; simultaneously A of 6 will talk to 5 and so on. So, in the 1st round, everyone talks to their neighbor on the right side. So, 8's neighbor is 7, 7's neighbor is 6 and so on, everyone talks to the neighbor on the right side, take whatever value they have and add it to their own value and that is going to be called the 1st round.

(Refer Slide Time: 06:25)

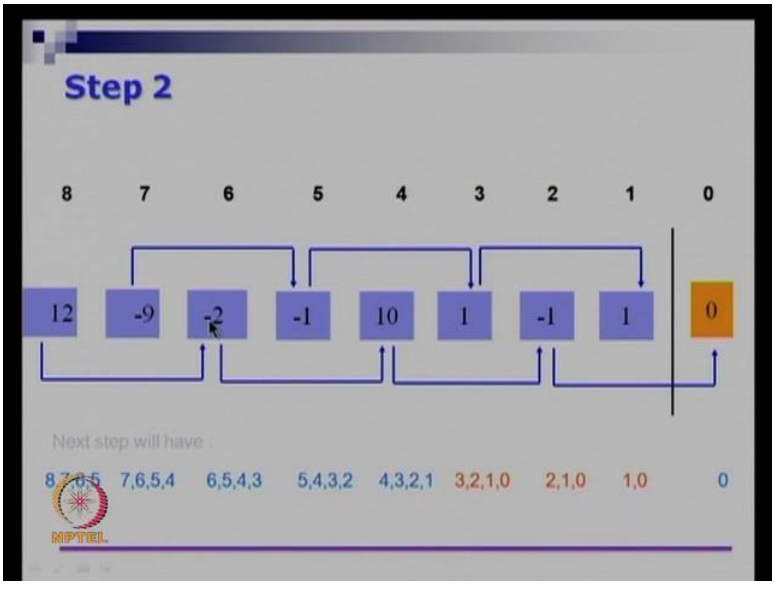


So, once the 1st round is over, then so at the 1st round A of 8 would have ask for A of 7, which is minus 15, let us assume that the end of that round, 27 is added to minus 15 and that result is store back in A of 8. Similarly, minus 15 is added to 6, the result is minus 9, let say that is stored in 7. Then, 6 and minus 8, if you add up with minus 2, let say that is stored here and so on.

At the end of 1st round, what would happen is, A of 8 will have the value stored, the value that will be stored will be the sum of A of 8 and A of 7. At 7, you will have the sum of A of 7 and A of 6, at 6 you will have the sum of A of 6 and A of 5 and so on, at A of 1, you will have the sum of A of 1 and A of 0. So, what we have so far is, every location has talk to the neighbor and got the values and that is added to the current location.

So, clearly this is not prefix sum yet, because this is only having a 8 and 7, it does not have the summation of 6 to 1 yet, this does not have summation of 5 to 1, this does not have summation of 4 to 1 and what not. So, at the end of 1st round, all you have is, at every location you have the summation of two values. One interesting that is happened is A of 1, which is supposed to be A of 1 plus A of 0 is already attained at the end of round 1. So, at the end of round 1, A of 1 is settled, you are not going to change A of 1 here after, so A of 1 is settled.

(Refer Slide Time: 08:03)



Let us look at the next round, the next round what we are going to do is, every element will communicate with elements that is two steps away. So, ((Refer Time: 08:11)) let us go to the previous 1, 27 minus 15 was 12, minus 15 plus 6 is 9, 6 plus minus 8 is 2 and

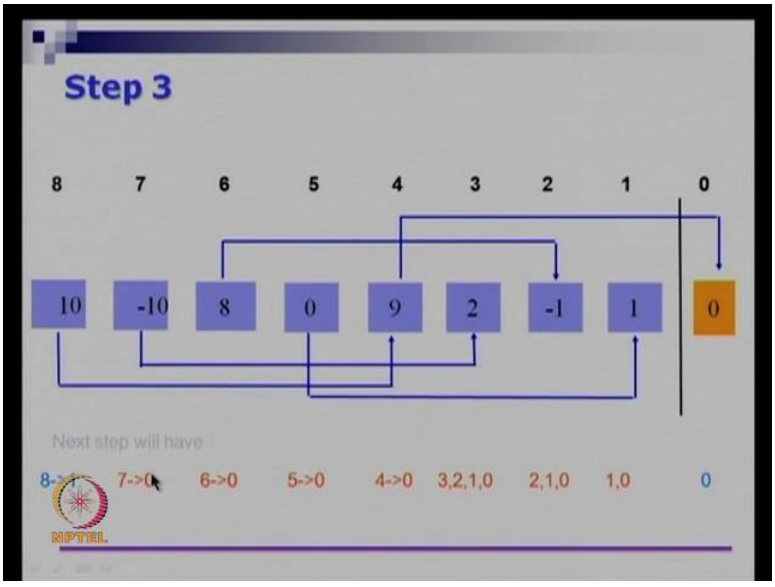
so on is minus 2 and so on, let us say we have those elements there. Now, every element will have to communicate with a neighbor, who is two steps away.

So, A of 8 communicates with 6, 7 communicate with 5 and so on, 2 communicates with 0. However, A of 1, it is already settled; it should not communicate with anything, 0 is also settled, it will not communicate with anything. So, let say we do that, then at the end of it, A of 8 plus A of 6 would be 12 minus 2, which is 10. However, you are not adding A of 8 and A of 6, already A of 8 at added up A of 8 and A of 7 and A of 6 had added up A of 6 and A of 5.

So, at the end of this step, A of 8 will have the summation of A of 8, 7, 6 and 5 from the original array, then A of 7 will have A of 7, 6, 5, 4 from the original array and so on and A of 3 will have 3 to 1, 0, 2 will have 2 plus 1, 0 from this side. So, 2 and 1 was already there at 2, now it is also adding whatever is at A of 0 and 1 comma 0 was settled in the previous round itself and so on.

So, when you communicate with neighbor of distance 2, then you get two more elements added up. So, in the 1st round, two elements were added up, in the 2nd round, four elements get added up and these locations will not change any more, A of 3, A of 2, A of 1 will not change any more, in the 2nd round. So, in the 1st round, one element got settled, in the 2nd round, two element gets settled and all the other once have addition of four consecutive locations.

(Refer Slide Time: 10:06)

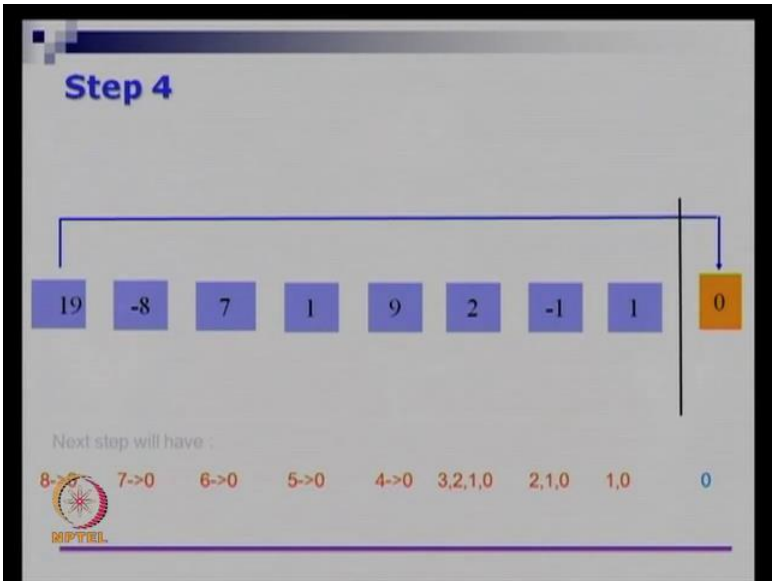


Let us do step 3. So, in step 3, what would happen is, every element will have to talk to a

neighbor, who is at a distance of 4 away. So, 8 will have to talk to 4, 7 will talk to 3, 6 will talk to 2 and 5 will talk to 1 and 4 will talk to 0 and when that happens, 3 2 1 0 have already been settled in the previous rounds. So, at the end of 3rd round, 8 will have the summation of 8 to 1, 7 will have to summation of 7 to 0, 6 will have to summation of 6 to 0 and so on.

So, my problem statement was add up everything up to 0, up to the location 0. So, at this point in 3rd round itself, A of 7 got everything from 7 to 0 and 6 got everything from 6 to 0 and so on. So, in the end of 2nd round, we have settled only three elements, in the 3rd round, we settle four more elements to a total of 7 elements.

(Refer Slide Time: 11:04)



Finally, in the 4th round, every element talks to a neighbor, who is a distance of 8 away, so 7 does not have a neighbor, who is distance of 8 away, 6 does not have a neighbor of distance 8 away and so on. So, A of 8 is the only one, who has a neighbor of distance 8 away, which is A of 0, you add that also and at this round all things are settled down. So, what we have done is, we have taken an array of size n, which is what we have to add with A of 0 implicitly being 0.

And in 4 rounds, we are able to add all the 8 values and put it in A of 8. If I have done this problem on with 16 elements plus 1 additional element at the end, in 4 rounds I could have done this, in 5 rounds I could have done this. So, if in general, if I wanted this to be done for an array of size n, which where n is some power of 2, then I would have needed logarithm of n plus 1 steps. So, it is prefix sum, if you do it in parallel, it can be

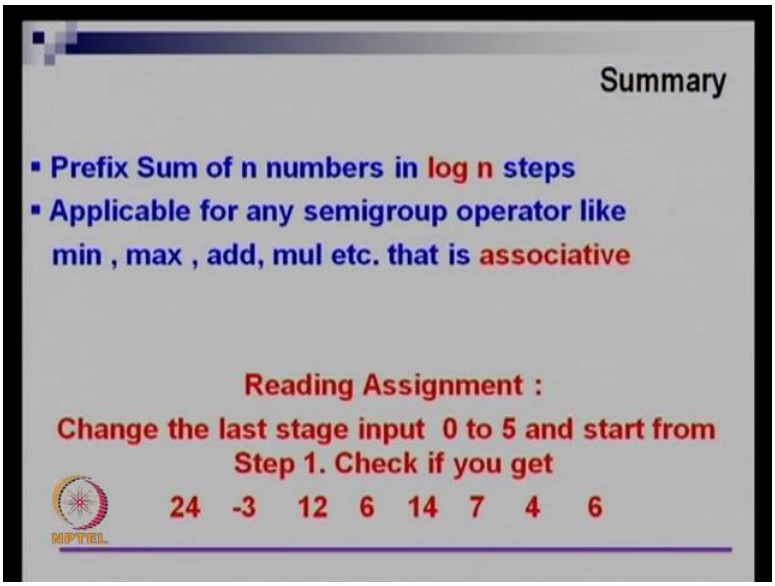
done in logarithm n plus 1 steps.

So, the reason why I call this parallel is, all these additions are happening in parallel. So, let us go to the very 1st round, at the very 1st round, you will need n adders, these n adders must take A of 8, 7. So, one adder will take A of 8 and A of 7 added, another adder will take A of 7 and A of 6 added, another adder will take 6 and 5 and add it and so on. In the 1st round you need n adders, in the 2nd round, you will need n by 2 adders, the 3rd round you will need n by 4 adders and so on. The worst case in a single step you need n adders.

So, we dedicated more hardware, instead of having only one adder, which takes two elements at a time added put the result back and then take two elements at a time, put it back and so on. Instead of doing that, in the 1st round, if you put n adders, then in the next round, we need only n by 2, I can use n by 2 from the previous round and not use the other n by 2.

The 3rd round I will use n by 4 of the original n that I started with and not use the other 3 and by 4 and so on. If you do all of that carefully of course, all of this has to be then carefully, so that all the wiring is matching and so on. Anyway this is now as a software program, I am telling you how to do this, if you allow to do additions and parallel, then you can do this. The key thing is, we can do this in logarithm n steps. So, I am going to take this and see, how to do parallel addition of binary numbers, I will come to that in a little while.

(Refer Slide Time: 13:39)

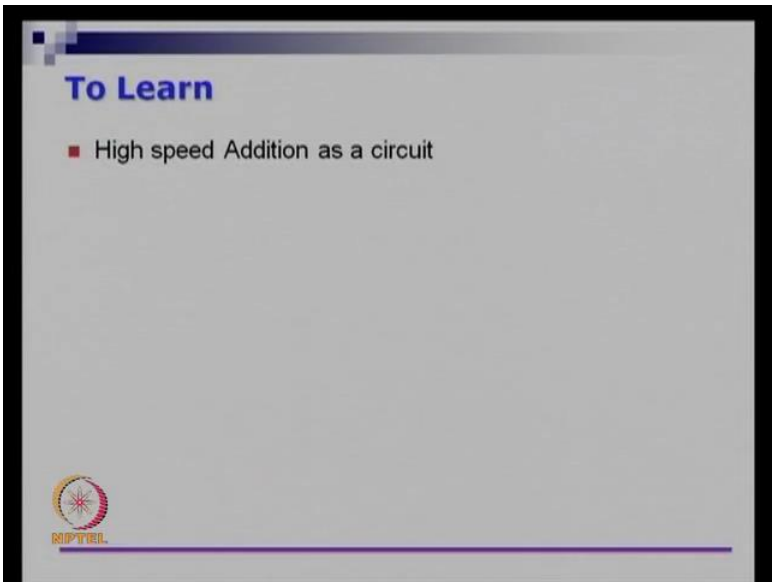


So, the summary is that, the prefix sum of n numbers can be done in logarithm n steps and you can do this for any operator like minimum, maximum, add, multiply and so on as long as the operators are associated. So, instead of asking for prefix sum, I can ask for prefix min, prefix min is at every location, I want the minimum of the smallest element from that location all the way to the right. So, that is called prefix min.

Prefix max is maximum of all the elements, prefix mul is prefix multiplication of all the elements from A of 1, all the way up to A of i for every i. So, such a thing is called semi group operator. So, let us not worry about why it is call that. So, it is called a semi group operator as long as you have such an operator, you can do prefix calculation in logarithm n steps.

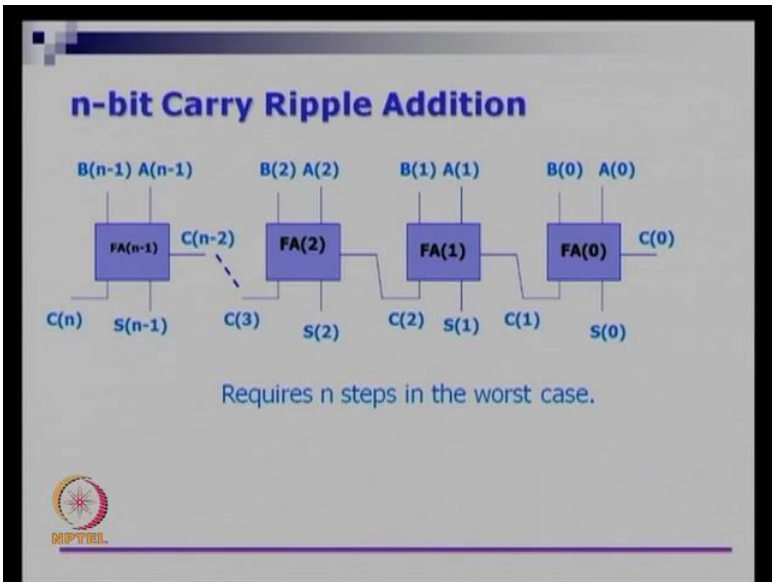
So, you cannot do things like subtraction and so on, because subtraction A minus B minus C is not same as the A minus B minus C. So, this is not for such kind of operators. So, I want you to go and do a reading assignment, where the last stage input at stage 0, you change it to 5. So, I said I assume that, A of 0 is 0, instead change it to 5 and see, whether you get these values, this is a reading assignment for you.

(Refer Slide Time: 15:04)



Once you understand that, let us now look at high speed addition as we do it in a circuit.

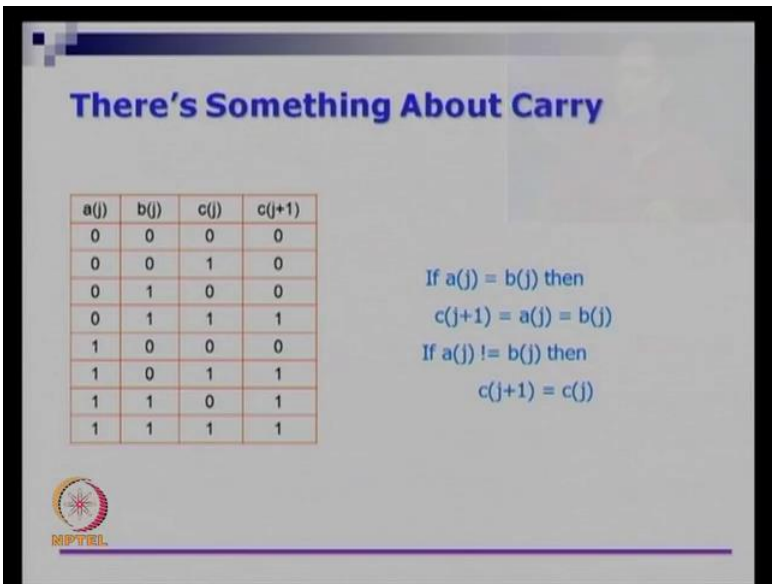
(Refer Slide Time: 15:08)



So, to appreciate what the high speed addition is, let us look at n bit ripple carry addition in this case n. So, for sum n, we start with B of 0 and A of 0, let us assume that, there is some carry in that is coming in to the circuit, it produce sum and carry. Then, we get B of 1, A of 1 that has to be added to carry 1 and that will produce a sum and carry and so on.

So, we know that the worst case this A, B and C, change in that may change carry 1 and that may have to ripple through all of this and may have to change C of n and S of n minus 1. We said that, this requires linear number of steps in the worst case, where it is linear in the size of the width of addition itself.

(Refer Slide Time: 15:53)



So, let us look at a specific location j. So, I am picking one of the adder location, let us call it j and for a of j, I want to look at, what is happening to the carry, because sum is not a problem. So, you give a and b, you get a sum and then it you get a carry. The sum gets decided, but the carry has to move to the left side, it has to move to the next location, sum does not have to move. Sum is decided for location j, but carry of j plus 1 has to move to the other locations greater than j.

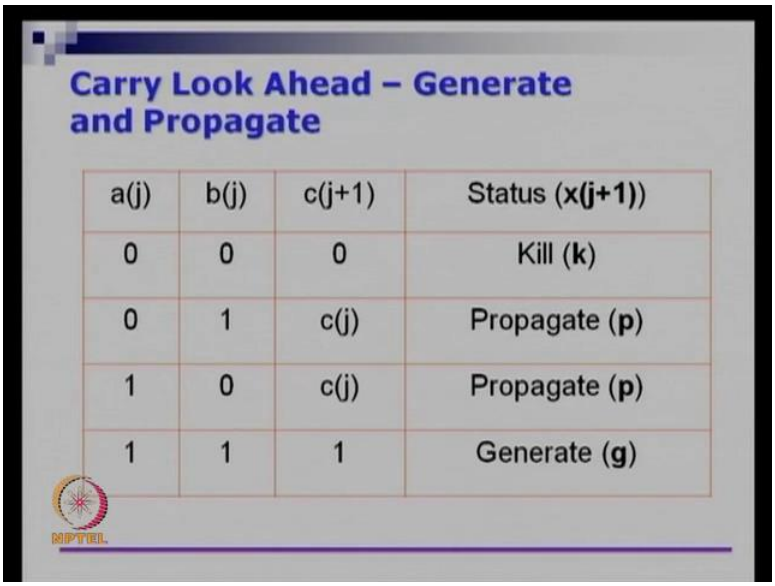
So, I am interested in looking at c of j plus 1. So, let us look at a of j, b of j and c of j and see, what c of j plus 1 is. So, this is just truth table, if a, b, c are all 0's, c of j plus 1 is 0, if a is 0, b of j is 1 and c of j is 0, then c of j plus 1 is 0 and so on. So, if I get an input carry at location j and a and b is something that I am going to give parallel anyway. Once, c of j comes, I can calculate c of j plus 1, but when c of j comes, this is truth table, this essentially the truth table for a full adders carry circuit.

So, the most interesting thing about this is, if a and b are same, so that is condition of return here, if a and b are same, c of j plus is actually either a or b itself. However, if a and b are not same, then c of j plus 1 is c of j, let us go and look at this, let us validate this statement. Whenever, a and b are same, so that this row, whenever a and b are same, carry of j plus 1 is a same as, so a and b same here. So, carry of j plus 1 is same as carry of j.

So, in the 1st two rows and to the last two rows, a of j is same as b of j, you will see that c of j plus 1 is a same as a of j or b or j, we can see that here. However, in the middle 4 rows, this row, this row, this row and this row, a of j j is not equal to b of j, in this case, c of j goes to c of j plus 1. So, the way to think about that is, if I am go to add 0 and 0 in the current location, so in this case a of j is equal to b of j. So, if I have a of j is 0 and b of j is 0, even if there is incoming carry 1, it is not going to generate an outgoing carry.

Similarly, if I have a of j is 1 and b of j is 1, if both of them are 1's, no matter what the incoming carry is, there will be an outgoing carry. However, only when you have this condition, where one of them is 0 and one of them is a 1, you really have to know what the input incoming carry is. Once, the incoming carry is known, only then you can decide the outgoing carry. You cannot settle the outgoing carry, until you know the incoming carry, only under the condition where a and b are not same. So, this is interesting case, this is some interesting thing about carry.

(Refer Slide Time: 19:04)



So, what we are going to do is, we are going to explain that. So, what we are going to do is, if a of j and b of j are 0, if both of them are 0, I do not have to know c of j at all, this is a very important thing. If I know a of j and b of j are both 0, I do not need to decide based on c of j, I can say that c of j plus 1 is 0 without worrying about, what the incoming carry is.

So, I am going to make c of j 1 plus 0 and the condition I am going to call that condition as a kill condition. The kill condition happens when no matter what the carry is, it is going to kill the carry. So, you look at state j, there is some carry that is coming in c of j, no matter what c of j is, if a of j and b of j are both 0, we are going to kill the carry. So that incoming carry is 1, it gets killed become 0, if the incoming carry is 0, it still gets killed and the output is 0, the carry output is 0. So, this is called the kill condition.

The kill condition happens when both a of j and b of j are 0, because without looking at c of j, I can say that c of j plus 1 is killed or it is going to be 0. Similarly, if a of j and b of j are both 1, again without looking at c of j, I can say that, I have to generate c of j plus to be a 1, I have to make that carry to be a 1, I will call that the generate condition. The other two conditions happens when a of j is 0 and b of j is 1 or a of j is 1 and b of j is 0, if one of these conditions happen, I need to know, what c of j is, but c of j decides, what c of j plus 1 is.

So, let see what happens here, let say a of j is 0 and b of j is 1. So, in the current stage, the inputs 0 and 1, if the incoming carry is 1, then it suppose to be 1 plus 0 plus 1, the

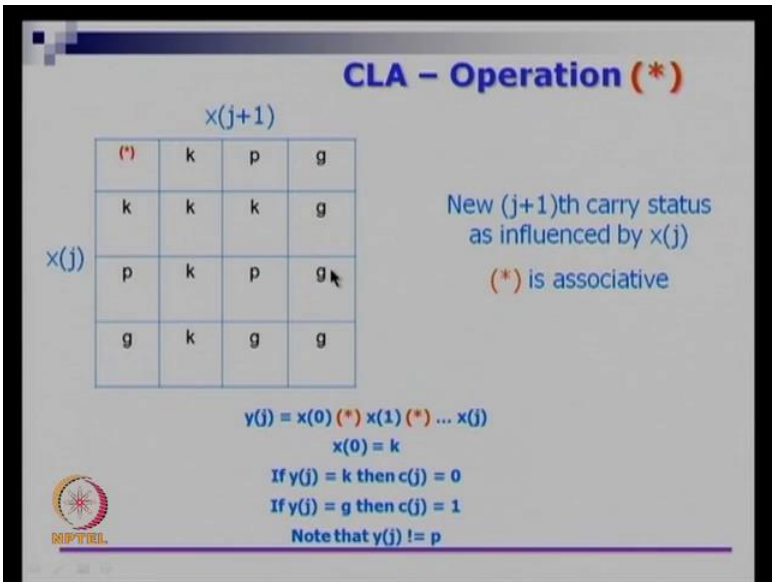
sum is 0 and the carry is 1. So, c of j plus 1 is 1, which is same as c of j, if the incoming carry was 0, then I have 0 plus 1 plus 0, 0 plus 1 plus 0 is sum 1 carry 0. So, c of j plus 1 is 0, which is the same of j itself. The same thing happens, when you have a of j is 1 and b of j is 0.

So, essentially what it says is, if the current conditions at the j th location, if a of j is 0 and b of j is 1 are the other way round, then I will take the carry that is coming in from the input stage, I will propagate it to the next stage. So, this is called the propagate condition. So, we have three conditions namely, kill, propagate and generate. So, if I know that, if both the inputs are the same, either we are killing the carry into the next stage or we are generate a carry for the next stage.

However, if the inputs are different at a particular stage, then the incoming carry is passed on to the outgoing carry and notice that is incoming carrying being passed on to the output carry is the problem in our ripple carry adder. So, if every stage was let say, they where kills in every stage, then there is no carry propagation. Similarly, if there was a generate in every stage, again you can do something better.

So, you have a problem, when you have this extra condition that the inputs are 0 and 1 or 1 and 0 and there is a carry that is coming in, it may have to go to the next stage. Of course, if the input carry is 0, you do not pass on anything, but it is possible to the input carry that is coming in so 1 in which case the next stage will get a 1. This is a series problem and we want to see, how to do this in parallel.

(Refer Slide Time: 22:50)



So, let me write this as a table. So, I am going to look at condition x of j plus 1 and condition x of j. So, condition x of j plus 1 means, we looked at a of j plus 1 and b of j plus 1 and based on that, I generated either kill or propagate or generate. So, that is listed in the columns here and if I go and look at the x of jth stage, again it have taken a of j and b of j and I would have had either kill or propagate or generate for that stage.

Now, since j plus 1th stage is following the jth stage, I am going to look at what happens in the columns here. So, let say the j plus 1 stage says, it is a kill, which means a of j plus 1 is 0 and b of 1 plus is 0. So, this stage is generating in kill. So, no matter, what is coming from the right side, since this stage is generating a kill, it stopping the carry from moving forward and the carry that you have give to the next stages is solve kill.

So, x of j plus 1, essentially to j plus 2th stage, which is on the left side of the j plus 1, it can say that, this is killing after a kill or killing after a propagate or killing after a generate in all these cases, your killing the carry, so it is as good as kill. So, the input carry to j plus 2th stage is not 1, it is going to be 0. Similarly, if j plus 1 stage, the conditions are in such a way that, you have a of j plus 1 is 1 and b of j 1 is 1, the carry to the next stage will be a 1.

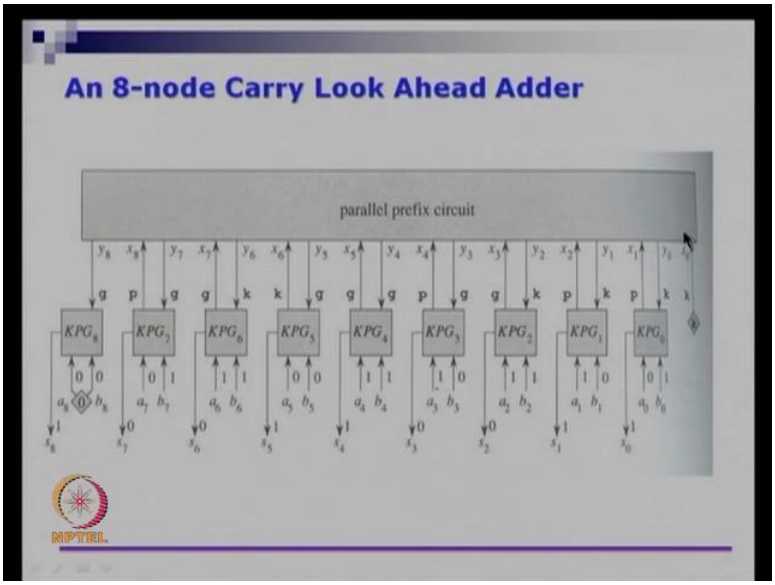
So, generate which is to the left of kill is still a generate, generate to the left of the propagate, it is still a generate, generate to the left of generate, it is still a generate, because this is the higher order bit and this is the lower order bit. Higher order bit has one and one in the inputs. So, no matter what the inputs are coming from the previous stage, the next stage to this one will have a 1. The propagate as only place, where we have to copy the stage from the previous one.

So, fixed kill you have to propagate a kill, if it is a propagate, you have to propagate it, you do not know, how to resolve it yet and if is it a generate, you have to propagate a generate. So, propagate a kill is kill, propagate of propagate is propagate, propagate a generate is generate. And I am going to have this y of j, y of j is all the states operate at using this table. So, this operator I am going to call star.

Star is not multiplication, so this is an operator called star and y of j is x of 0 star, x of 1 star, x of 2 all the way up to x of j and I assume that, x of 0 is a kill, which means the very first location does not have any input, it is a kill. And what I am going to do is, I will resolve this y of j, I resolve it using this table, I will take repeated locations and resolve it. If y of j is k, then c of j is 0, if y of j is g, then c of j is 1.

So, at the end, if I say that, if I can resolve it as a kill, then c of j is 0, at the end, if I resolve it to be a generate, then c of j is 1. Note that, when you finally finish it, y of j cannot be a propagate, it cannot be a propagate, because at the end you will always have either a propagating something that is coming from the previous side, at the end you any way have a x of 0 is k. So, in the worst case, it will just become, even if you have a serious of propagates at the end you have a k. So, it will be a kill in that case.

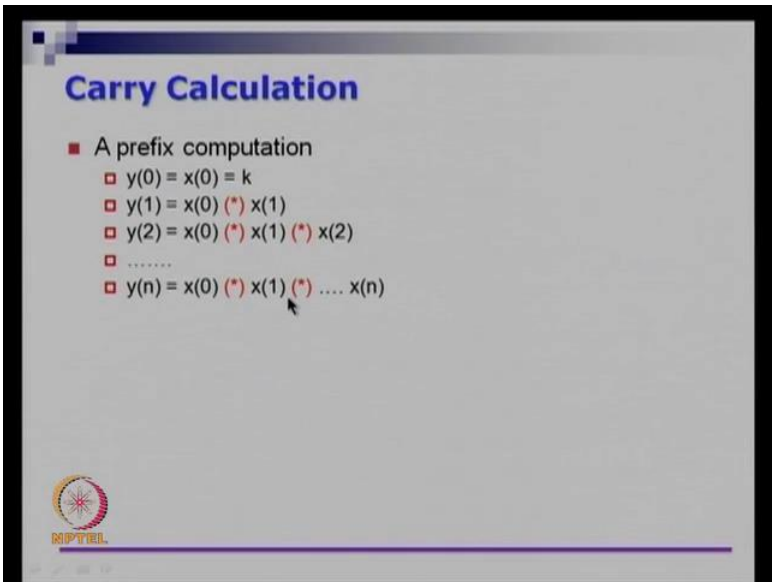
(Refer Slide Time: 26:29)



So, let us look at carry calculation. So, I am going to design a circuit call the parallel prefix circuit, I will show that in a little while and there will be gates of this kind. So, there is a module call KPG 0, KPG 1, KPG 2, 3, 4 up to 8 and there is this stage, which is kill, it is going to the parallel prefix circuit. So, what we are doing here is, we are giving the inputs to KPG, this KPG intern is a generator this will.

So, each stage will give it is resolution to the parallel prefix circuit. So, let us go and look at the inputs here, a and b are 0 and 1. So, this is giving a propagate, then a and b are one1 and 0, this is giving a propagate, a and b are both 1 here, this is giving a generate, then this is 1 0 here, it is giving a propagate and so on. So, what you see on the left side is what is coming in from the inputs a and b, then the parallel prefix circuit is suppose to compute the parallel prefix.

(Refer Slide Time: 27:32)



So, this parallel prefix is because, what we have seen here. So, you look at this y of 1 is x 0 and star x of 1, y of 2 is x of 0 star x of 1 star x of 2, you can notice that, this is very much like prefix. So, the prefix thing that we talked about, when we adder numbers, it is very similar to that. So, for 2, I need everything from 0 to 2 operator upon, for n, everything from x of 0 to x of n, must be operator upon using this operator.

So, since every stage i, requires everything from i, all the way to 0 to be operator together, this is a prefix calculation we are doing, we can construct a circuit which is a parallel prefix circuit. So, I am going to show you, how to do that circuit later. So, we can have a parallel prefix circuit, which will then resolve the prefix and give it back. So, once you do that, then you can create the sum very easily.

So, here you have one and one, if you added one and one, the sum 4 here would have been 0, but it gets a generate, which means, the incoming carry of 1. So, the summation becomes 1. So, let us look at each one of them. So, the summation of 0 and 1 is 1, but it have a propagate signal, the parallel prefix will return a kill, kill means there is no incoming carry which is a 0. So, 0 plus 1 is 1 and then you add a 0, so sum naught is 1 itself.

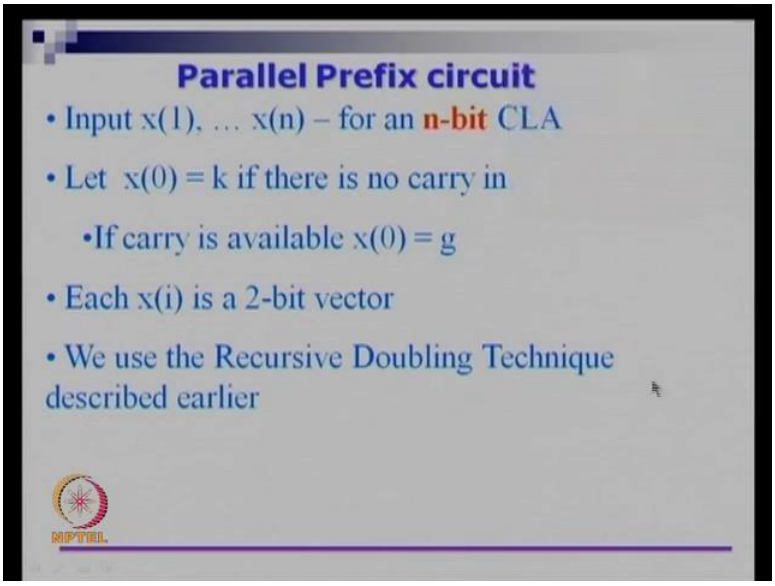
Similarly, a 1 plus b 1 is a 1 and it is also giving a propagate, but this is propagate after a kill. So, that will be a kill. So, this kill is treated as a 0 and 0 plus 1 plus 0 is 1, here it is 1 plus 1, it is actually a 0 and you are getting a kill, which is means 0, so the sum is 0. So, this is the stage where it is interesting, both are 1, so the summation is actually a 0, but it

gives a generate signal here.

The parallel prefix circuit and intern return a generate signal, this g when it comes to KPG which be treated as a 1. So, 1 plus 1 plus 1, the summation is 1, the carry can be ignore, there is no carry, because you already taken care of passing on the carry to the other stages. So, similarly here this 1 plus 1, which is 0; however, there is no incoming carry, because it says the incoming carry is a kill or a 0.

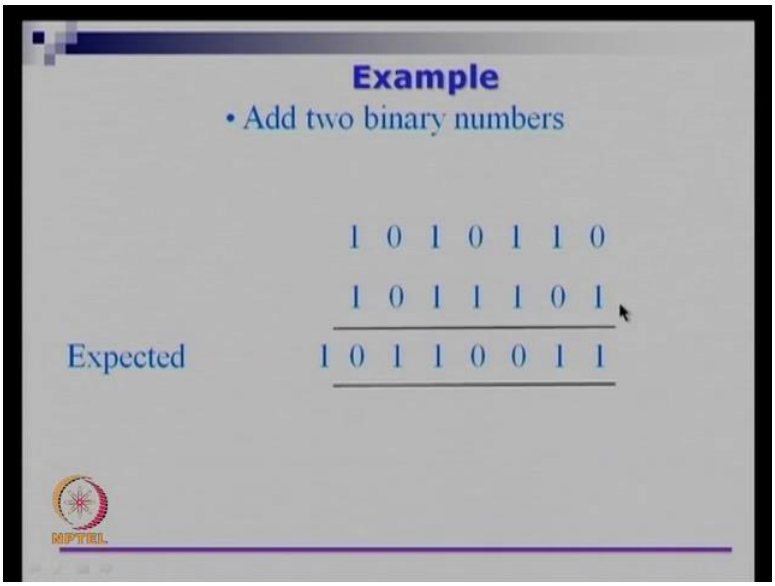
So, which means 1 plus 1 is a 0, you do not have to add anything more and so on. I recommend that you go and give this set of inputs add it yourself as a binary number and ensure that is the summation is correct.

(Refer Slide Time: 30:03)



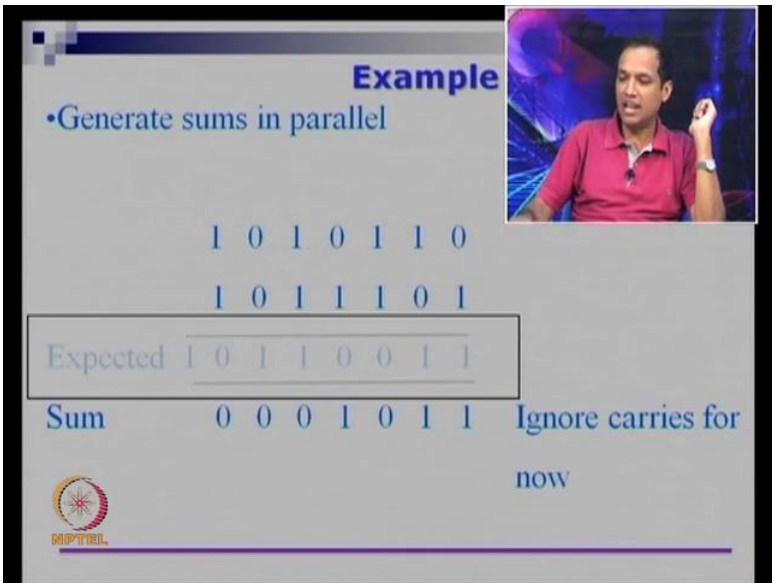
Let me show, how the whole thing works.

(Refer Slide Time: 30:05)



So, let I will take a specific example, let us say I get to this example, 1 0 1 0 1 1 0, I am going to add with 1 0 1 1 1 0 1, the expected sum is this. So, the only thing that have to notice is, I am not taking of A of 8 to A of 1, I have taken A of 7 to A of 1, assuming that A of 0 is 0. So, this is the expected result, you can verify this may directly adding.

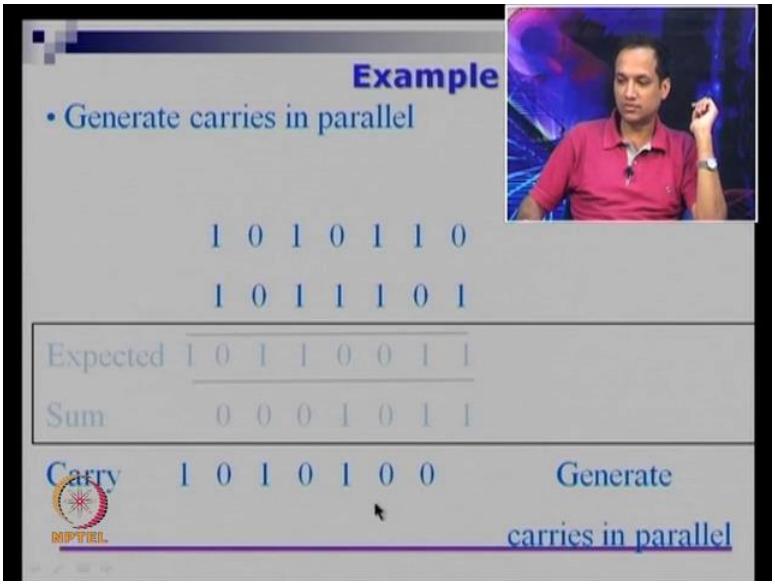
(Refer Slide Time: 30:31)



So, I am mark that as gray area here, that is the expected result. So, I am going to do first stage what I am going to do is, I am going to generate the sums in parallel. So, 0 and 1 is 1, 1 and 0 addition is 1, 1 and 1, the addition is 0, I will ignore the carry for now, 0 and 1, addition is 1, 1 and 1, addition is 0, 0 and 0 addition is 0, 1 and 1, addition is 0. In doing this, I do not look at what is the carry that is coming from the previous stage. So, I

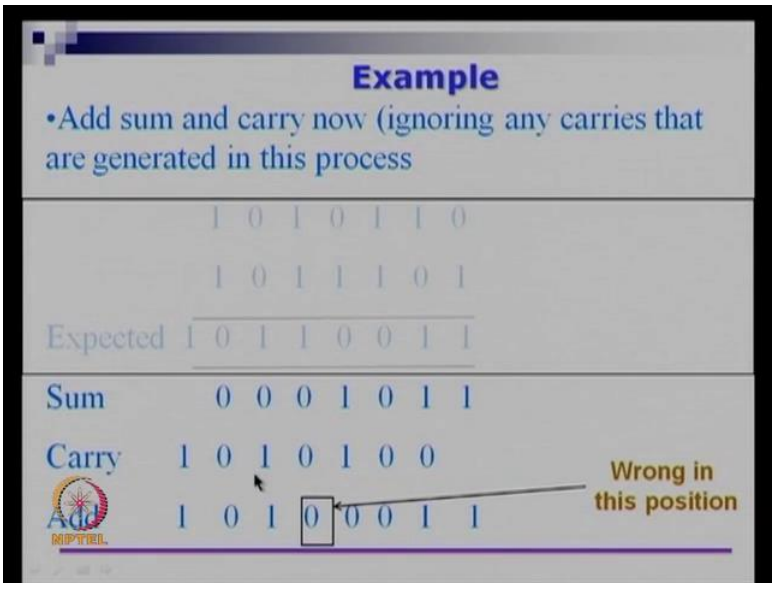
generated all the sums in parallel.

(Refer Slide Time: 31:02)



And I can also generate all the carries in parallel. So, 0 plus 1, it should generate a carry of 0, 1 plus 0 should generate a carry of 0, 1 plus 1 should generate a carry of 1 and so on.

(Refer Slide Time: 31:17)

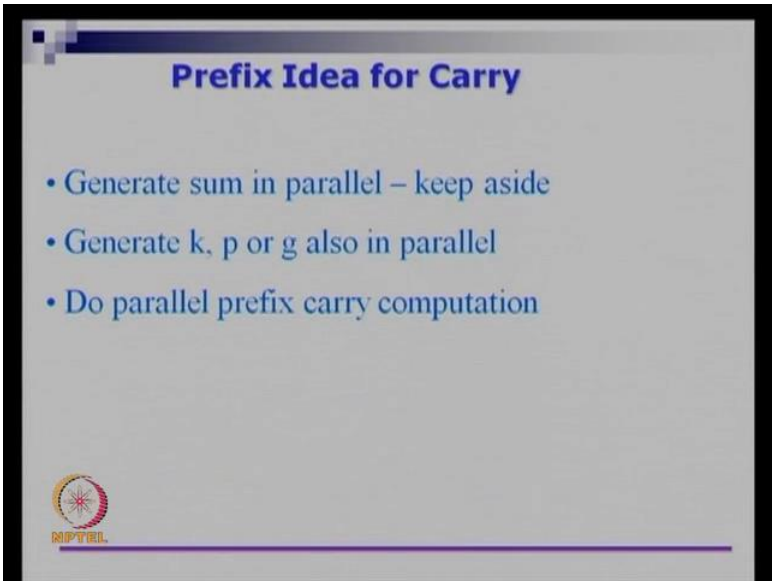


And if I go and blindly add sum and carry as it is now. So, this is the sum that I generated in parallel, this is the carry that I generated that in the parallel, if I add both of these together, then I get this. So, now, if I look at it, so when I say adding, I am only doing, so for example, one is directly here, 1 plus 0 is 1, 0 plus 0 is 0, 1 plus 1 is 0 and I am

ignoring the carry that coming due to this addition. Similarly, 0 plus 0 is 0, 0 plus 1 is 1, 0 plus 0 is 0 and this is 1.

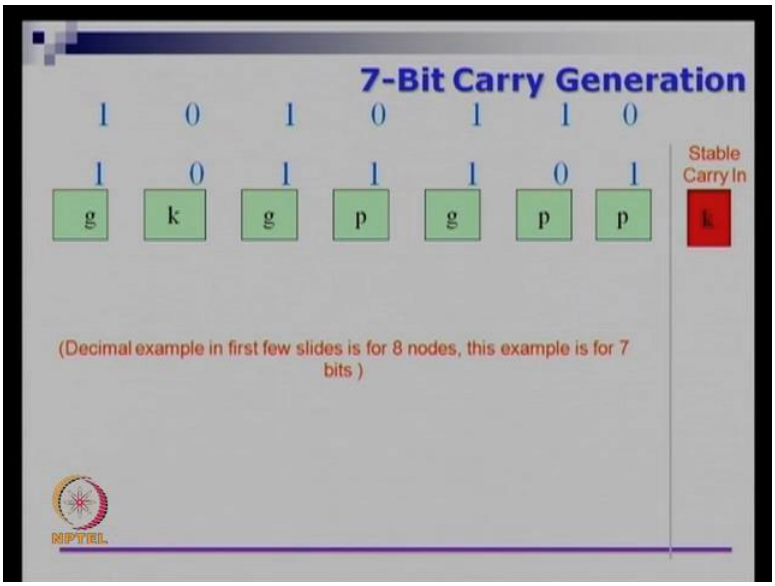
If I leave it as it is this is not the expected result, if you go and look at the expected result, it is 1 0 1 1 0 0 1 1. So, what I want is, I cannot do just addition of… So, the addition of these two actually means it AND of these two, so the XOR of these two that is the sum. So, sum is XOR, OR. So, this final result is this intermediate sum x or this intermediate carry, but that will be incorrect, because the carry signal, we have taking care of only that stage, we are not seen what is happening from the right side, this is where we need the parallel prefix circuit.

(Refer Slide Time: 32:27)



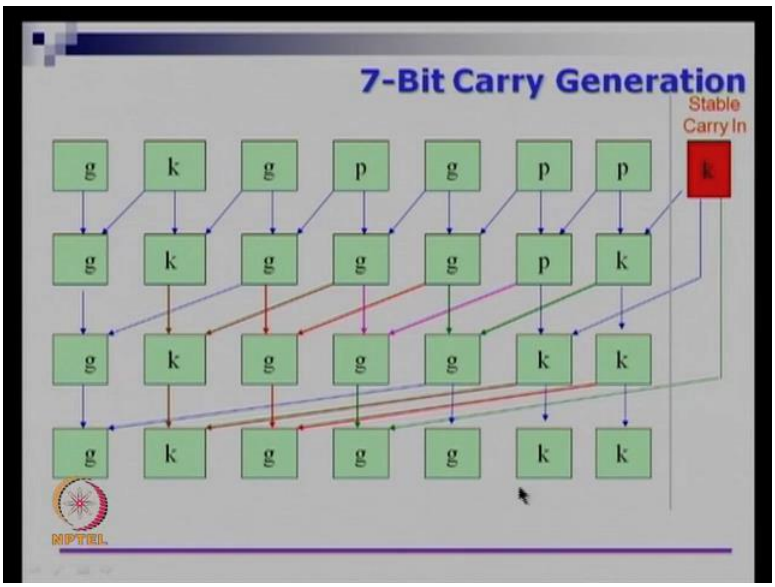
We will generate the parallel prefix. So, we will generate the sum in parallel, keep it a side, generate k p or g also in parallel, do parallel prefix computation. Then will get a carry vector, which we will add, I will show you the steps.

(Refer Slide Time: 32:44)



So, let us say, these are the numbers, these are the two numbers that I want to add. So, I have I will calculate the sum and I will put it aside, let it be aside, now I am going to look at only the parallel prefix stage. So, 0 and 1 means, it propagate, 1 and 0 means propagate, 1 and 1 means generate, 0 and 1 means propagate, 1 1 is generate, 0 0 is kill, 1 1 is generate, I have the very first stage gpk. So, this is before the prefix, this is the carry prefix that I wanted, this is before the prefix.

(Refer Slide Time: 33:18)



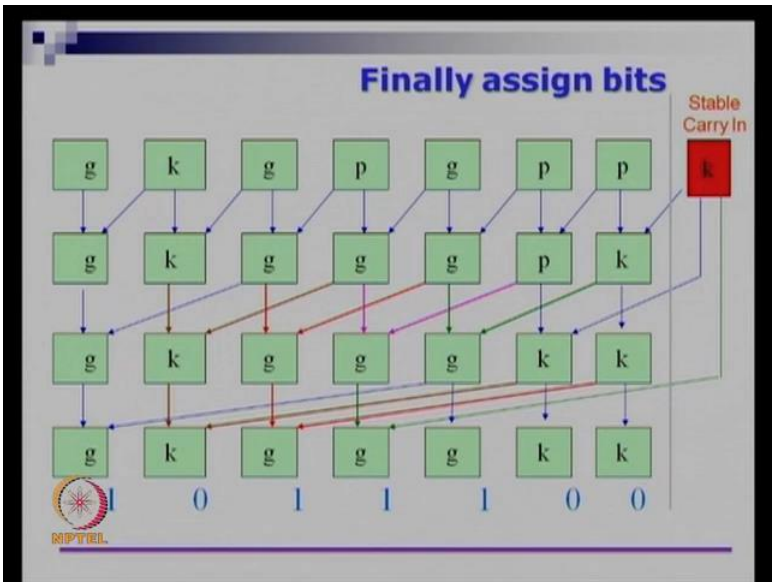
What I am going to do is, I am going to make each and every element talk to the neighbor first and resolve it. So, g after the k is g, k after k, g is k. So, generate after a propagate is generate, propagate of a generate is generate, generate of a propagate is

generate, propagate a propagate is propagate, propagate a kill is kill. So, this is my round 1, at round 1, so this location is settled, it will not change any more. This is just like what we did in the parallel prefix addition, this stage is settled.

Now, in the next stage what I will do is, the next step I will take each location and communicate with neighbor too away. So, generate after this generate is generate, kill after a generate is kill, generate after this generate is generate and so on, you can notice that, this is already taken care of the two locations. So, if you look at this blue line. So, this blue line as already taken care of locations this and this, this blue line has already taken care of locations this and this.

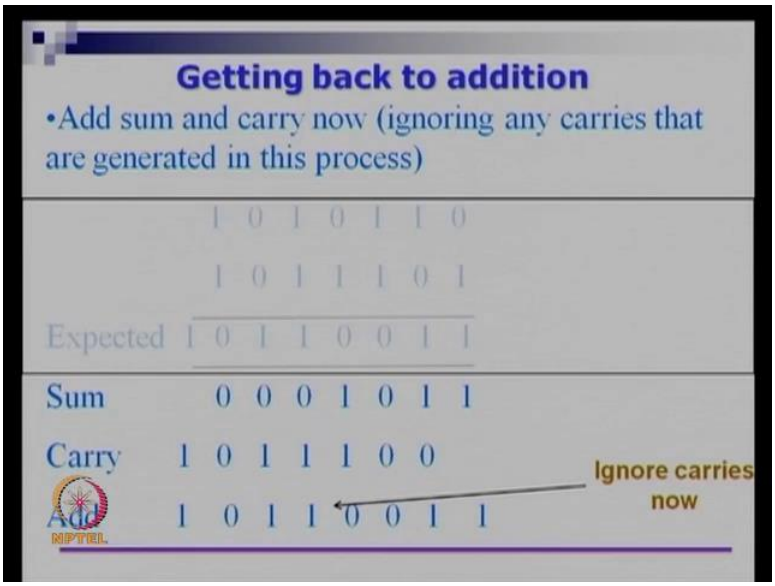
So, the left for most things will come here at this round. So, we can see the arrows and you can convince yourself that you are talking to a neighbor a distance too away in the second step. And at this point, this is settles, this was settled in the previous round in this round these two will get settled, the next round these 4 will get settled. So, one remark is that, this is only 7seven locations, it is not eight locations at the end of 3rd round itself, this is over. So, now, I have g k g g, g k k, I will go and interpret that as 1's and 0's.

(Refer Slide Time: 35:01)



So, generate means the incoming carry is a 1, kill means the incoming carry is 0. But, this is after resolving all the stage to the right side of you.

(Refer Slide Time: 35:13)



Now, I can take this vector this, this is 1 0 1 1 1 0 0 and I can a XOR with some locations appropriate some locations. So, 1 0 1 1 1 0 0 along with sum, it is take produce and parallel. So, I will now XOR these two and now it is only XOR these two, there is no carry; this will be the final result. You can go and verify that the expected result and the correct result are the same.

So, I will give you the summary of the step now. So, we are given to binary numbers which are unsigned numbers, what you do is, you calculate the sum which is just XOR of these 2 bits, keep is keep it is aside and calculate the carries, the carries are all just… So, you for each stage, we calculate the condition whether it is a g or a b or a k. So, you can calculate this just the location sum and all these carries simultaneously.

Then, you do log n steps of parallel prefix, carry calculation and that will be done in a log in steps. At the end, you will have only g's and k's, take the g's and k's interpret them as 1 and 0 respectively that will give you the carry vector. The sum vector was available in the first step itself and we took log in steps to get the carry vector, take the sum vector and carry vector with appropriate shifting and XOR it together, you will get the final sum.

So, all of them may looks like magic, but if you follow through the steps carefully, all I have done is, we have followed the same idea as in parallel prefix circuit. In log in steps we have resolved the carry condition, instead of n steps that we required in the ripple carry set up. So, this is a fast adder, because n grows, you are delay is not growing linear

with respective n, it is going only logarithmic with respect to n.

So, this brings me to end of module 51, what I will suggest is, you take more numbers, calculate the summation directly keep it is a side and see, whether you understand this whole process of doing parallel prefix or not. So, take different kinds of numbers, so that you get different kinds of conditions and ensure that, you have actually understood the whole process.

So, this is the fast adder for the reason that it is able to do in a logarithm n as suppose to n, so for n which is law fairly large. So, in current computers you are looking at 64 bit addition or the two bit addition and so on, you can do this in a few gate delays as suppose to 64 gate delays. So, this is the very handy thing and this is the circuit that you see in current day process and what not. So, this circuit is called the carry look ahead adder.

So, in the text books we write in a slightly different way, they write the g's and b's and c's and they have complicated equations to do the g's, b's and c's. So, here I have shown you very logical way in which you get with g, b and c. So, if go back to the text book, if you see gi, pi, ci it is nothing more than the g p and k that we have seen here, it is not a complicated thing. And once, you know the generate and propagate and kill you can resolve it as 1's and 0's appropriately.

So, the fairly interesting way of doing it and this is not something that I see in a undergraduate text books, it is a very useful thing to know. So, we have a fast adder, one fast adder, there are several other adders like carry, save order and so on. So, I cannot cover many of them here and all of them in somewhere will try and break down the number of operations from n to something less than n. So, this brings me to the end of module 51 and module 52, we will see some more high speed arithmetic circuits not adders, we will see other arithmetic circuits, which are high speed circuits.

Thank you and see you in a little while.