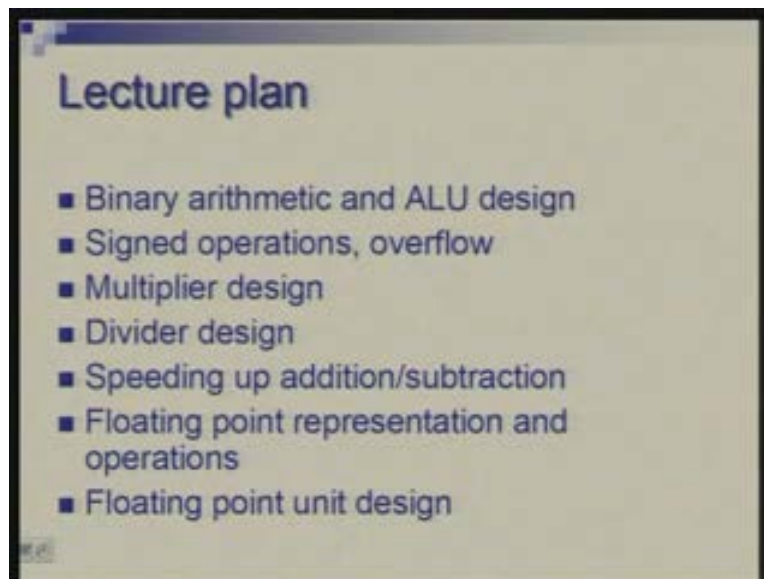


Computer Architecture
Prof. Anshul Kumar
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi
Lecture - 15
Fast Addition, Multiplication

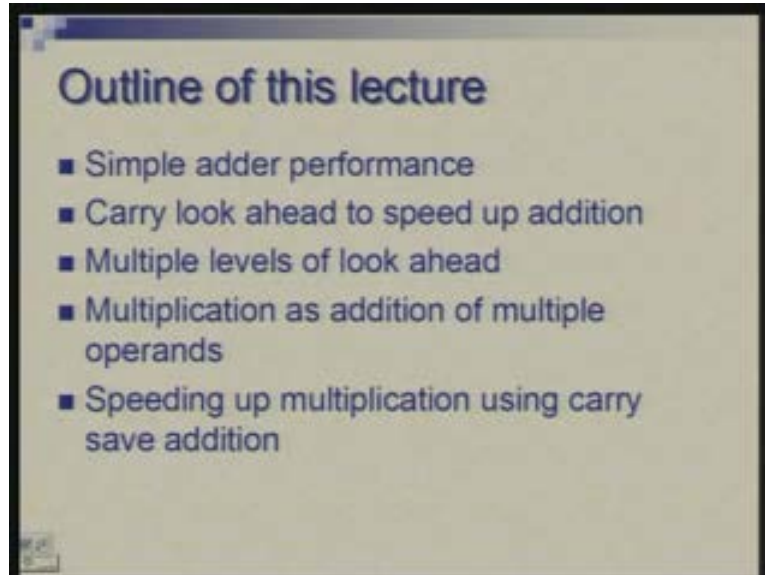
We have discussed how to carry out basic arithmetic operations and our focus was to do them in a very simple manner without worrying about the performance. Today we are going to see what techniques can be used to speed up the operation; improve the performance. So in the overall sequence of lectures in this broad topic, as you see that we have discussed the basic design of various operations and today we are going to talk of speeding up of addition, subtraction as well as multiplication and finally then we will move on to floating point operations.

(Refer Slide Time: 01:15 min)



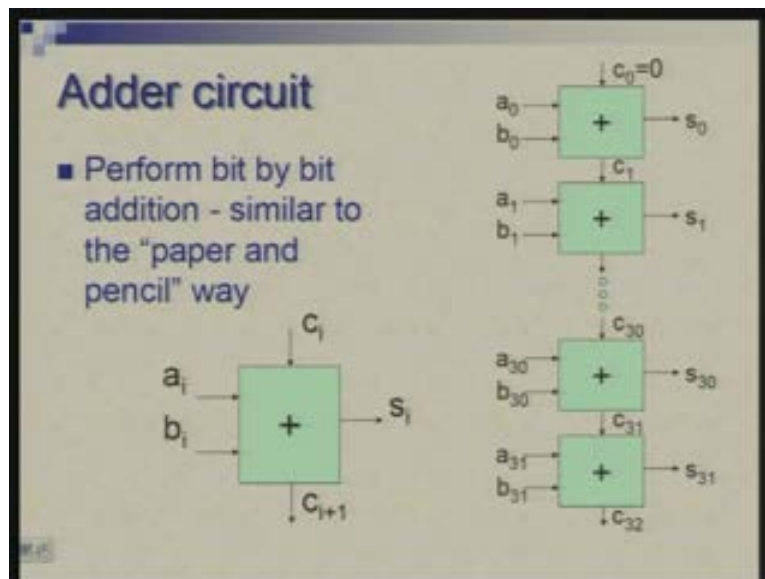
So first what we will do is we will look at a simple adder which we have discussed and see what actually governs the performance; what is the bottleneck; we will see a technique which is called carry look ahead. We will notice that it is carry which is taking time and some way to compute the carry fast is what is required. Then we will go over to multiplication and then try to see how multiplication can be carried out in a fast manner. The technique use there would be what is called carry save addition where we postpone the use of carry; rather than to immediately look for carry and wait for it we will postpone that and add at a different stage so in the process we will speed up our operation.

(Refer Slide Time: 02:31 min)



So, coming back to the simple adder circuit which we have discussed we had discussed how we do addition using paper and pencil and our attempt was to capture that in a simple circuit exactly the way we do it. That means we take the corresponding bits of 2 operands, perform the addition and let the carry go to the next stage.

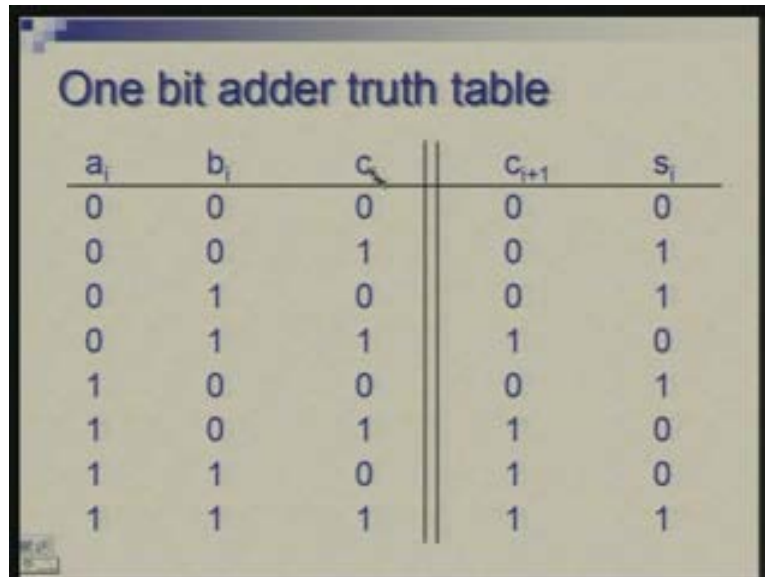
(Refer Slide Time: 2:55)



This is a circuit which adds two individual bits of the 2 operands and these 1-bit adders are cascaded or chained together to form adder of the right side. So, for example, if you are adding two 32-bit words then there are thirty two units which are put together like this. So in this you would notice that c_1 which is the carry out of the first unit is

dependent upon c_0 and c_2 is dependent upon c_1 and so on. So it is this dependence of carry from one stage to the other stage which has to be taken into account to complete the operation and our attention would be focused on this particular aspect of carry going from one end to the other end; from LSB side to the MSB side.

(Refer Slide Time: 03:57 min)



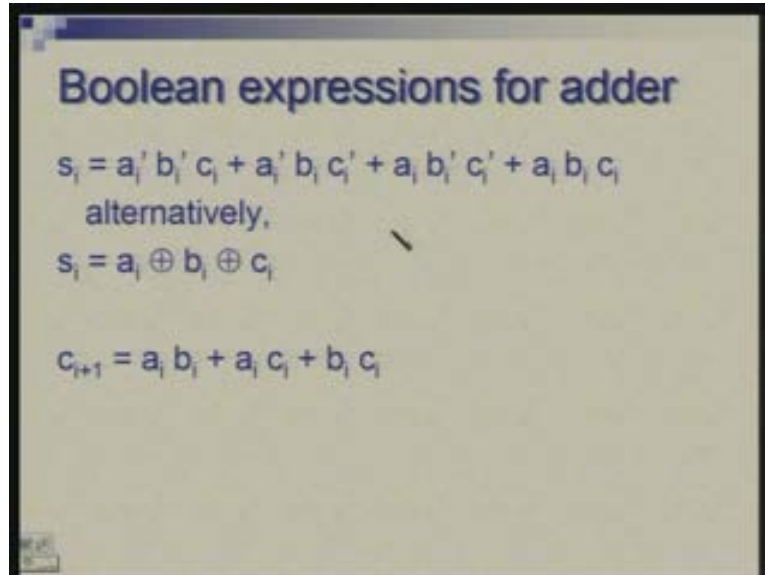
The image shows a truth table for a one-bit adder. The title is "One bit adder truth table". The table has five columns: a_i , b_i , c_i , c_{i+1} , and s_i . The first three columns are inputs, and the last two are outputs. The table lists all possible combinations of these inputs and the resulting carry and sum.

a_i	b_i	c_i	c_{i+1}	s_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

What each adder does is a simple task of looking at 3 inputs i th bit of a and b the 2 operands and the carry c_i as an input and it tries to define c_{i+1} the carry for the next stage and s_i as the sum for that particular stage. So it is a simple combination circuit with 2 outputs and 3 inputs. There are different ways in which this can be realized. One possible realization is shown here. In fact two: s_i can be expressed as a sum of product form of expression with the a_i , b_i , c_i as the inputs.

In a more compact form you can write this as an exclusive OR of a_i , b_i and c_i . c_{i+1} plus 1 on the other hand, is also a simple a simpler form it is in the sum of product form of three terms.

(Refer Slide Time: 04:55 min)



Boolean expressions for adder

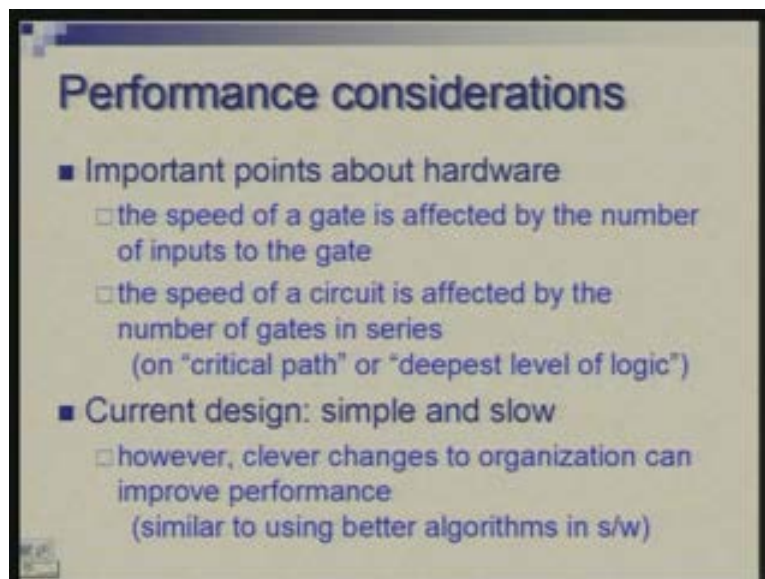
$$s_i = a_i' b_i' c_i + a_i' b_i c_i' + a_i b_i' c_i' + a_i b_i c_i$$

alternatively,

$$s_i = a_i \oplus b_i \oplus c_i$$
$$c_{i+1} = a_i b_i + a_i c_i + b_i c_i$$

So in this how do you define the performance of such a circuit?

(Refer Slide Time: 5:06)

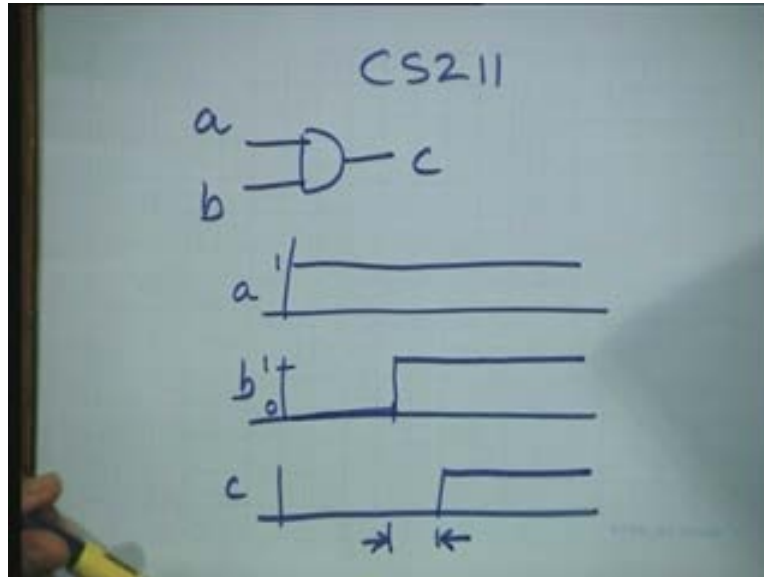


Performance considerations

- Important points about hardware
 - the speed of a gate is affected by the number of inputs to the gate
 - the speed of a circuit is affected by the number of gates in series (on "critical path" or "deepest level of logic")
- Current design: simple and slow
 - however, clever changes to organization can improve performance (similar to using better algorithms in s/w)

So, basically we should go down to how a gate behaves when input changes. So, at some point of time suppose the input changes, let me..... suppose we have AND gate and if you look at waveforms in time; suppose a is 1 and b is 0 up to some point when it changes to one then we expect that c would become 1 in response to this change but there would be some delay c is 0 at when at least 1 input is 0 so it is this delay which is attributed to the gate.

(Refer Slide Time: 6:24)



So, gate has transistors which switch from one state to the other state and it is that which takes time. Also, the gates are driving some load or some capacitance and that load has to be charged so the charging time of the capacitance is what dictates this delay. Without going much into details of this delay we assume that each gate has some inherent delay.

Actually there is also delay in the signal propagating from one point in the circuit to the other point in the circuit over the wire. When you are talking of gates which are extremely fast where the delays are of the order of picoseconds or tens of picoseconds then the delays of wires also....., the time taken for the signal change to propagate over the wire that also becomes significant and comparable.

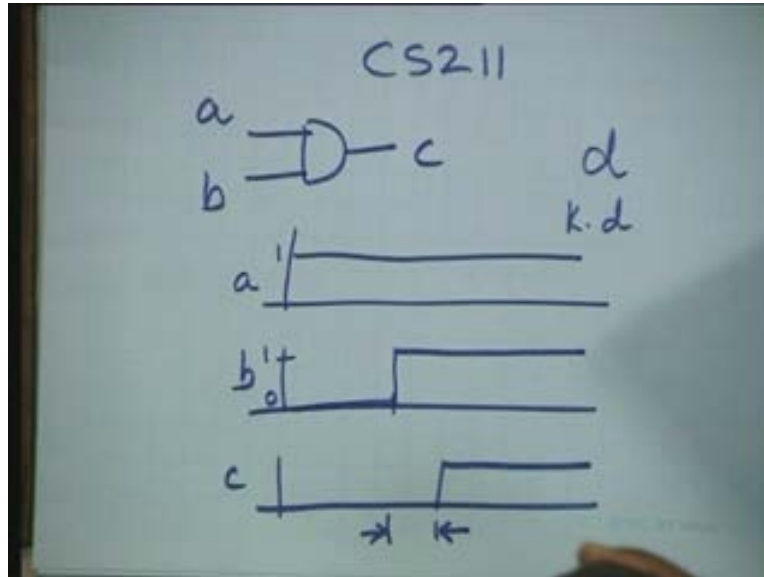
On the other hand, if the gates were working in nanoseconds or larger, then the wire delay may be negligible. For the present discussion we will assume that wire delays are not significant because our focus is on the logic part of the design and we will take into account the delays which are caused because of the gates. So now, in a larger circuit when there are number of gates; you have one gate feeding another gate and that feeds another gate then this delay gets accumulated and the delay would depend upon how many gates you are putting in a series. So, more the gates you have in a series the larger the delay is. So roughly speaking you could say that if in ideal condition delay of one gate is let us say d units of time d picoseconds or whatever the unit is and you chain k gates then the delay will be k times d .

Now the question is; is this factor d independent or not?

Strictly speaking, this d would depend upon how many inputs the gate has and how many other gates this gate is feeding which means the delay is a function of fan-in as well as fan-out. But it is a, for example, a 3 input gate would have slightly more delay than 2 input gate and 4 input gate will have a larger delay but the delay does not grow proportionally; it varies, it increases slowly it does increase but increases slowly. So in a

idealized situation when you are not using gates with very large number of inputs or outputs then you might assume up as an approximate situation that D is more or less a constant. But **we must remember** we must keep this in mind that we are idealizing; we are making an approximation.

(Refer Slide Time: 09:40 min)



So, as I mentioned speed of the gate is affected by number of inputs to the gate and more strongly speed of circuit is affected by the number of gates in the series and there may be many paths you can trace in a circuit from input to output; the longest path the path which goes through the maximum number of gates in the chain is called critical path or the deepest level of logic.

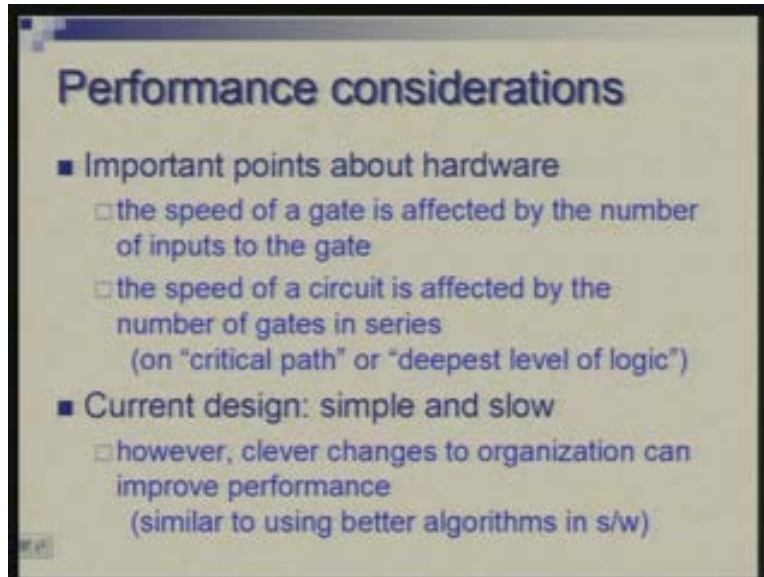
So, in fact this is really the motivation why we always try to express circuits, Boolean circuits as two level sum of products or you can alternatively do product of sum. You can have more complex expression where there is a parenthesis and deep nesting of ANDs and Ors. But a sum of product form or a product of sum form is most efficient from the point of view of this depth of logic consideration.

So in the current design of the adder which we have talked of, you could imagine that the delay of each 1-bit adder can be brought down to 2 times d . You can always express that as we did earlier, as we have shown here we look at this realization of s_i and c_i plus 1 so you will have 1 level of ANDing, you have a series of AND gates and then an OR gate. **I am ignoring again inverters, these compliments for the moment**; they will also have delays which has to be counted for but it will be smaller as compared to AND and OR.

So this circuit can compute the sum and carry in $2d$ time. So now, if you have thirty two of these put together or n of these put together in general then the delay will be n times $2d$.

Now the question is how can we reduce this; so what changes; how do we restructure these gates; how do we reorganize so that the delay is reduced. So this is like trying to improve your algorithm doing the same computation in a faster time.

(Refer Slide Time: 11:50 min)



So now, as we have seen the cause of the delay cause of the large delay is rippling of the carry from one stage to the other stage to the next stage. It is because c_{i+1} is generated from c_i by this expression (Refer Slide Time: 12:30) so that depends..... means that c_1 is expressed as a function of c_0 ; c_2 is expressed as a function of c_1 and so on.

Now the question is can we do some look at; can we avoid this ripple? Can c_{i+1} , for example, be generated directly from $a_0 a_1 a_2 a_3$ up to a_i and $b_0 b_1 b_2$ to b_i and c_0 . So, of course c_0 is a primary input which we have to count for but if you can write for example, can we write c_4 directly in terms of $a_0 a_1 a_2$ and a_3 as well $b_0 b_1 b_2 b_3$ and c_0 . So we can think of c_4 as a function of these eight inputs; 4 bits of a , 4 bits of b and 1 carry. So in principle it is possible and let us see how do we do it.

(Refer Slide Time: 13:30 min)

Speed of ripple carry adder

- Ripple is caused because c_{i+1} is generated from c_i , i.e.,

$$c_{i+1} = b_i c_i + a_i c_i + a_i b_i$$

$$c_1 = b_0 c_0 + a_0 c_0 + a_0 b_0$$

$$c_2 = b_1 c_1 + a_1 c_1 + a_1 b_1$$

$$c_3 = b_2 c_2 + a_2 c_2 + a_2 b_2$$

$$c_4 = b_3 c_3 + a_3 c_3 + a_3 b_3$$

- Can c_{i+1} be generated directly from $a_0.. a_i, b_0.. b_i$, and c_0 ?

So let us go step by step. First we write c_1 in terms of a_0, b_0 and c_0 that is the usual expression we have and we wrote c_1 we wrote c_2 in this form. Now all you need is you take the expression for c_1 and substitute that in the second one and also we expand because we want to ultimately have a two level expression. You could have a left that in parenthesis you could say that b_1 plus a_1 within bracket and then expression for c_1 in another bracket and then you can say that these two are ANDed but then we are increasing the level of logic. So our objective here is to keep the level low or number of gates which you find in a chain low so we expand it and we get a two level expression for c_2 .

(Refer Slide Time: 14:38)

c_{i+1} in terms of $a_0.. a_i, b_0.. b_i$, and c_0

$$c_1 = b_0 c_0 + a_0 c_0 + a_0 b_0$$

$$c_2 = b_1 c_1 + a_1 c_1 + a_1 b_1 =$$

$$b_1 b_0 c_0 + a_1 b_0 c_0 + a_1 b_0 b_1 + a_1 b_1 c_0 + a_1 a_0 c_0 + a_1 a_0 b_1 + a_1 b_1$$

$$c_3 = b_2 c_2 + a_2 c_2 + a_2 b_2 =$$

$$b_2 b_1 b_0 c_0 + a_2 b_1 b_0 c_0 + a_2 b_1 b_0 b_1 + a_2 b_1 b_1 c_0 + a_2 b_1 a_0 c_0 + a_2 b_1 a_0 b_1 + a_2 b_1 b_1 +$$

$$a_2 b_2 b_1 c_0 + a_2 b_2 b_1 c_0 + a_2 b_2 b_1 b_1 + a_2 b_2 a_0 c_0 + a_2 b_2 a_0 b_1 + a_2 b_2 a_0 b_1 + a_2 b_2 b_1 +$$

$$a_2 a_1 b_0 c_0 + a_2 a_1 b_0 c_0 + a_2 a_1 b_0 b_1 + a_2 a_1 b_1 c_0 + a_2 a_1 b_1 c_0 + a_2 a_1 b_1 b_1 + a_2 a_1 b_1 +$$

$$a_2 a_2 b_0 c_0 + a_2 a_2 b_0 c_0 + a_2 a_2 b_0 b_1 + a_2 a_2 b_1 c_0 + a_2 a_2 b_1 c_0 + a_2 a_2 b_1 b_1 + a_2 a_2 b_1 +$$

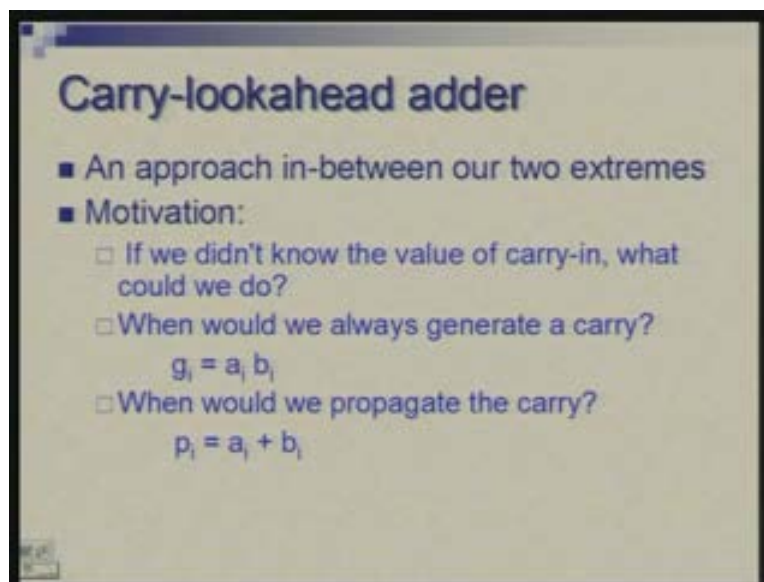
$$c_4 = b_3 c_3 + a_3 c_3 + a_3 b_3 = \dots$$

c_{32} will have 4 billion terms !!
 Effect of large fanin and fanout ??

Next we take expression for c_3 and take the values of c_2 from this in an expanded form and substitute it here. You get again a large expression but still it is possible to get a two level expression. One could keep on doing this. **you could** From this you can generate expressions for c_4 then c_5 and you can go all the way go up to c_{32} and what will happen is that c_{32} will have 4 billion terms. Although theoretically possible it is an impractical solution and the assumption we made initially that we will consider that delay of each gate is constant **that was** that is possible only if the number of inputs number of fan-ins is not very large. But now what is going to happen is that we will have gates with.... these product terms will become very very wide they will have 32 inputs. So a 32 input gate will no longer have the same delay as a 2 input, 3 input or 4 input of gate although we will keep things in two levels but our assumption about delay of individual gates will break down.

Similarly, there will be large fan-out for at least the primary inputs. You would notice that each of these variable there are after all 32 a inputs, 32 b inputs and a c_0 all these will be appearing in so many different terms 4 billion terms so there will be lot of fan-out as far as the primary inputs; all these a i's and b i's are concerned so it is an unworkable circuit. You will able to verify that this is..... the number of terms will be observed as 4 billion because you know the first one has three terms, second has seven terms, then you have fifteen terms, then there will be 31 and so on so it is some power of 2 minus 1 so you can verify it yourself.

(Refer Slide Time: 17:00)



Carry-lookahead adder

- An approach in-between our two extremes
- Motivation:
 - If we didn't know the value of carry-in, what could we do?
 - When would we always generate a carry?

$$g_i = a_i b_i$$
 - When would we propagate the carry?

$$p_i = a_i + b_i$$

So, why do we not take an approach which is somewhat in between?

We do want to have look ahead. You want to directly compute c_i 's without looking at intermediate carries. So what we can do is we can basically allow some preprocessing to be done on a_i and b_i . We will look in to what role a_i and b_i are playing in determination of carry and try to do something before we really compute all the c_i 's. So we have actually two stages: In one stage we will let individual circuits for each bit

position, digest a_i and b_i do something compute something which is more useful and then do something what we try to do in the previous slide. So, instead of computing c_i 's directly from a 's and b 's we will compute it from some preprocessed form of a and b .

What we notice here is that the condition under which one particular stage generates a carry and condition under which propagates carry. So, when a_i and b_i both are one then irrespective of what carry is coming from input side a new carry will be generated. So we denote this by g_i . So g_i is the condition when carry gets generated in i th position irrespective of what has happened towards its LSB side.

Secondly, when any of a_i or b_i is 1 if the carry is coming from the input side it will propagate to the output side. So c_i plus 1 will be 1 if any of these is 1 or sorry I should say that c_i plus 1 will be 1 if c_i is 1 and any one of these is 1 so we denote this expression as p_i or the propagate condition. So now we try to rewrite all the c_i values in terms of p 's and g 's.

(Refer Slide Time: 19:27)

Carry-lookahead adder

- Did we get rid of the ripple?

$$c_1 = p_0 c_0 + g_0$$

$$c_2 = p_1 c_1 + g_1 = p_1 p_0 c_0 + p_1 g_0 + g_1$$

$$c_3 = p_2 c_2 + g_2 = p_2 p_1 p_0 c_0 + p_2 p_1 g_0 + p_2 g_1 + g_2$$

$$c_4 = p_3 c_3 + g_3 = p_3 p_2 p_1 p_0 c_0 + p_3 p_2 p_1 g_0 + p_3 p_2 g_1 + p_3 g_2 + g_3$$

Feasible! To what extent?

So we can say that c_1 is true if either carry gets generated by stage 0 or carry coming in gets propagated at this stage. Similarly, we can write for c_2 c_3 c_4 when c_2 is $p_1 c_1$ plus g_1 c_3 is $p_2 c_2$ plus g_2 and so on. Now in this we can make substitution of the kind we did earlier in terms of a 's and b 's. So the value of c_1 we take from the first line and substitute here we get another sum of product term; this term we substitute here we get this and we substitute here we get this. Again we have growing expressions but you can see that the growth is much more contained.

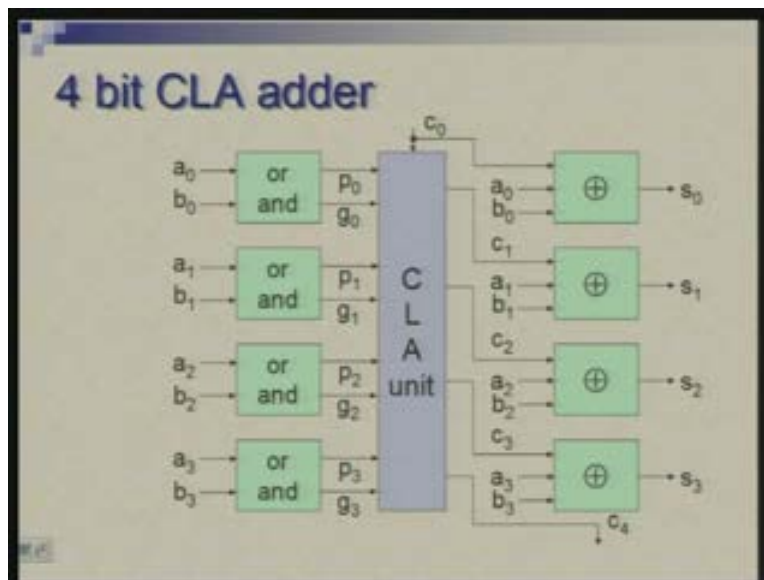
Of course this gives a feasible way of doing it but to what extent we can do. We still cannot keep on doing this for thirty two stages. So we will do it for limited stages and then we allow may be carry to go in the normal way.

Actually coming back to these expressions we can **we can** reason this out directly also. So, for example, here for c_2 what we can say is that either carry gets generated at stage one or carry gets generated at stage zero and gets propagated through 1 or there is an incoming carries c_0 which propagates through p_0 and p_1 through stage zero and one and so on. So, similarly let us look at the last one.

The condition for c_4 is that either there is a generation at stage three or generation at stage two which propagates through 3 or generation at stage one which propagates through 2 and 3 or generation at stage 0 which propagates through stage one, two and three and incoming carry which propagates through all these four stages. So you can either derive these expressions by substitution or reason them out directly based on this logic of what these terms p 's and g 's are.

So now, this, as I mentioned this can be done to a limited extent; we will not try to do this for thirty two stages and we will do it for let us say four stages which is not very large and then see what happens.

(Refer Slide Time: 22:04)



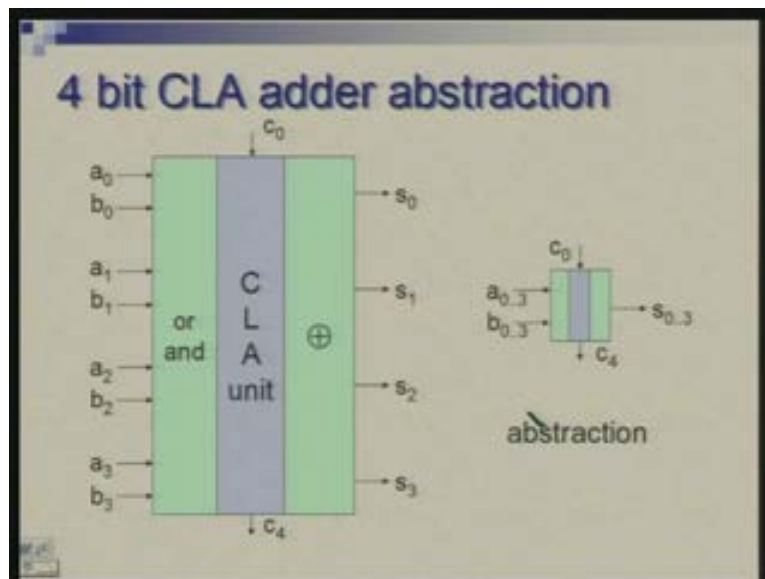
So, first of all let us understand the circuit which will do this. We have the first stage which I have captured in these green boxes; each one of these contains basically one AND gate and one OR gate. AND gate generates g_i 's and OR gate generates p_i 's. So we have these p 's and g 's computed out of a 's and b 's then we have this block (Refer Slide Time: 22:35) which computes all carry values looking at p 's and g 's. c_0 is the input and it will compute c_1 , c_2 , c_3 and c_4 which has to go out then we have the circuits computing the sums out of a 's, b 's and carries.

So, on the whole there are actually three stages and we can see **what is** how much delay this circuit will have. So can you figure out what is the maximum delay?

See, the first stage delay is one unit; there is either one AND gate or OR gate single level, this second stage has two levels, the third stage also has two levels although I have shown for brevity I have shown Exclusive OR but you can write as sum of product. So basically there is one level plus two levels plus two levels there is a five level; so in 5 d time we can compute all the sums and the final carry. Of course the carry is coming in 3 d time and the sums are coming in 5 d time.

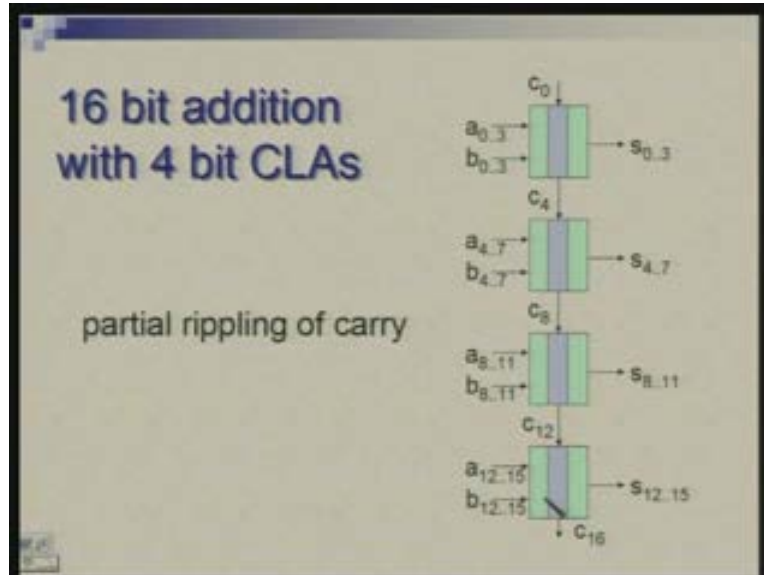
Now we will take this as a module and try to build larger adders. So first we try to represents this in somewhat an abstract form. I have just [col.....24:00] three stages now we are suppressing internal details so that we will look at the whole thing as a block box and try to put more of these to build larger adders. So I am just showing that on one side you have a and b inputs; one side you have sums, there is a carry c_0 and carry out c_4 and three stages we are not now separating them out but there is a box which has three stages and in a smaller form I will show it like that.

(Refer Slide Time: 24:28 min)



So now, suppose we connect four of these in a chain form the way we did for ripple carry adder but now we have units of 4-bit adders and within 4 bits we have look ahead and across these blocks of 4 bits we do not have look ahead we have carry rippling through. So now there will be, let us see, let us try to find out how much delay this circuit will have.

(Refer Slide Time: 25:05)



So remember that the carry comes out in 3 d time and sum comes out in 5 d times so let me **let me** write the time when the thing will come out at different points. so **this is** this comes at 5 d **I will omit d**; this (Refer Slide Time: 25:36) is coming out in time 3 and from this 3 to this output it will take another 2 so this **just a minute** this will take d plus 2 d 3 okay the carries which are coming out in this will be 3 plus 2 5 and 2 this will come out **at after** 7 d.

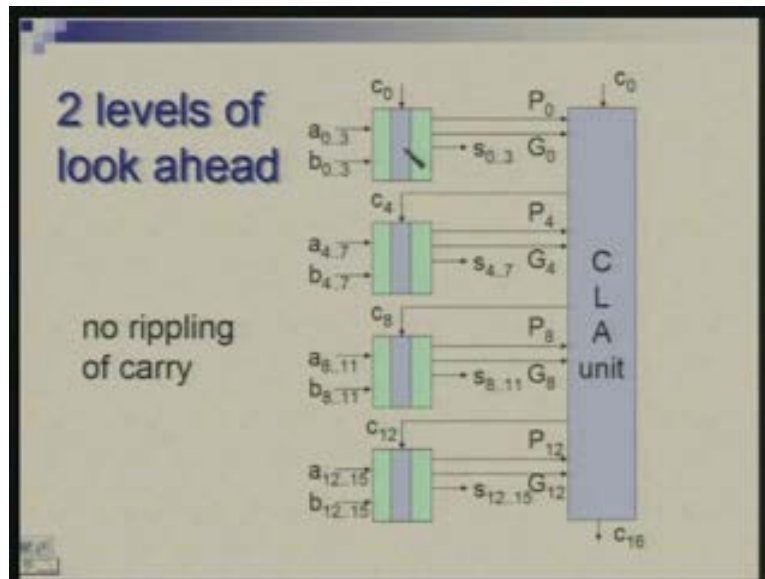
You can see that that 3; this purple box will take 2 d time and another 2 d time here so 7 d, this will come out at 5 7 9. So first let us look at the carry times. so the time was 0 here 3 here, 5 here, 7 here, 9 here and from this 3 we get 7 here, from this 5 we get 9 here, this 7 we get 11 here (Refer Slide Time: 26:44) so the whole thing has taken **11 d of** 11 d units of time. If we had allowed normal ripple carry to operate, for 16 bits we would have required 16 into 2 that is 32 d so there is a considerable saving and saving will show up more and more as we increase the value of n or number of stages.

So now here, there is partial rippling of carry. Question is can we even avoid that. one way would have been that extend this block or this carry look ahead to sixteen stages or thirty two stages but that increases the cost too much and the gates become wider so the delay also starts increasing because of that reason. But what we can do is we can apply the same idea once again at a higher level.

So, we have two levels of look ahead. These four blocks are as OR but rather than letting the carry ripple through I try to do look ahead at that level. That means try to determine c_4 directly from c_0 c_8 directly from c_0 and c_{12} directly from c_0 at the next level. So what this will require is something which is equivalent to propagate and generate signal at block level. we have blocks of 4 bits and each level we generate block propagate and generate signals P_0 and G_0 P_4 and G_4 and so on and this unit which I am showing here which is looking at these capital P's and G's and generating these carries c_4 c_8 and c_{12}

and also c_{16} would be exactly identical to this small purple box which I have shown. The same idea that c_0 is coming here and there are P's and G's signals so either carries get generated through these G's or carry gets propagated according to P signals.

(Refer Slide Time: 28:55 min)



So now what are these block propagate generate signals or group propagate generate signals that is what we need to see next. So we call these group propagate and generate or block propagate and generate signals. Recall that expression for c_1 c_2 c_3 c_4 in terms of P's and G's over these. So, on the whole looking at all 4 bits together we say that this block propagates carry is all small p's are **one**. That means p_0 p_1 p_2 p_3 if all of them are one then incoming carry will get propagated. So we call this as capital P 0; this is a propagation condition across four stages.

G_0 is a **generate condition** generation condition in these four bits which means that generation takes place either at level three or stage three or stage two, stage one or stage 0. Generation may take place at any of these four stages and propagate out to the extreme left. We say it is g_3 or g_2 and p_3 or g_1 and p_2 p_3 or g_0 and p_1 p_2 p_3 . So given this you can write c_4 in terms of capital P 0 and c_0 that is this block propagates c_0 or this block generates a carry. So we will have similar conditions for all the blocks and then carries can be computed in this look ahead manner.

(Refer Slide Time: 29:25 min)

Group propagate & generate

$$\begin{aligned}
 c_1 &= p_0 c_0 + g_0 \\
 c_2 &= p_1 c_1 + g_1 = p_1 p_0 c_0 + p_1 g_0 + g_1 \\
 c_3 &= p_2 c_2 + g_2 = p_2 p_1 p_0 c_0 + p_2 p_1 g_0 + p_2 g_1 + g_2 \\
 c_4 &= p_3 c_3 + g_3 = \\
 &\quad p_3 p_2 p_1 p_0 c_0 + p_3 p_2 p_1 g_0 + p_3 p_2 g_1 + p_3 g_2 + g_3 \\
 P_0 &= p_3 p_2 p_1 p_0 \\
 G_0 &= p_3 p_2 p_1 g_0 + p_3 p_2 g_1 + p_3 g_2 + g_3 \\
 c_4 &= P_0 c_0 + G_0
 \end{aligned}$$

So, in general we say that P_i is the p_{i+3} and p_{i+2} and p_{i+1} and p_i ; G_i is again composed of this g_{i+3} or g_{i+2} propagating through this stage, g_{i+1} propagating through these two stages, g_i propagating through these three stages and once these are done we can have c_4 c_8 c_{12} c_{16} expressed in much the same way as we had expressed the carries c_1 c_2 c_3 and c_4 . So basically it is the same form of expression and I will leave it for you to verify.

(Refer Slide Time: 31:02 min)

Group propagate & generate

$$\begin{aligned}
 P_i &= p_{i+3} p_{i+2} p_{i+1} p_i \\
 G_i &= p_{i+3} p_{i+2} p_{i+1} g_i + p_{i+3} p_{i+2} g_{i+1} + p_{i+3} g_{i+2} + g_{i+3} \\
 c_4 &= P_0 c_0 + G_0 \\
 c_8 &= P_4 P_0 c_0 + P_4 G_0 + G_4 \\
 c_{12} &= P_8 P_4 P_0 c_0 + P_8 P_4 G_0 + P_8 G_4 + G_8 \\
 c_{16} &= P_{12} P_8 P_4 P_0 c_0 + P_{12} P_8 P_4 G_0 + P_{12} P_8 G_4 + \\
 &\quad P_{12} G_8 + G_{12}
 \end{aligned}$$

One thing I would like you to notice here about comparison of this form with this form (Refer Slide Time: 31:58); here the growth of the number of terms was exponential; the

number of terms is roughly doubling whereas in this case the growth is linear so that is what has saved as and why it is linear here and why it is exponential there is because here you have the carry appearing at two places so let us take this.

(Refer Slide Time: 32:22 min)

c_{i+1} in terms of a_0, a_i, b_0, b_i , and c_0

$$c_1 = b_0 c_0 + a_0 c_0 + a_0 b_0$$

$$c_2 = b_1 c_1 + a_1 c_1 + a_1 b_1 =$$

$$b_0 b_1 c_0 + a_0 b_1 c_0 + a_0 b_0 b_1 + a_1 b_0 c_0 + a_0 a_1 c_0 + a_0 a_1 b_0 + a_1 b_1$$

$$c_3 = b_2 c_2 + a_2 c_2 + a_2 b_2 =$$

$$b_0 b_1 b_2 c_0 + a_0 b_1 b_2 c_0 + a_0 b_0 b_1 b_2 + a_1 b_0 b_2 c_0 + a_0 a_1 b_2 c_0 + a_0 a_1 b_0 b_2 +$$

$$a_1 b_1 b_2 + a_2 b_0 b_1 c_0 + a_0 a_2 b_1 c_0 + a_0 a_2 b_0 b_1 + a_1 a_2 b_0 c_0 + a_0 a_1 a_2 c_0 +$$

$$a_0 a_1 a_2 b_0 + a_1 a_2 b_1 + a_2 b_2$$

$$c_4 = b_3 c_3 + a_3 c_3 + a_3 b_3 = \dots$$

c_{32} will have 4 billion terms !!

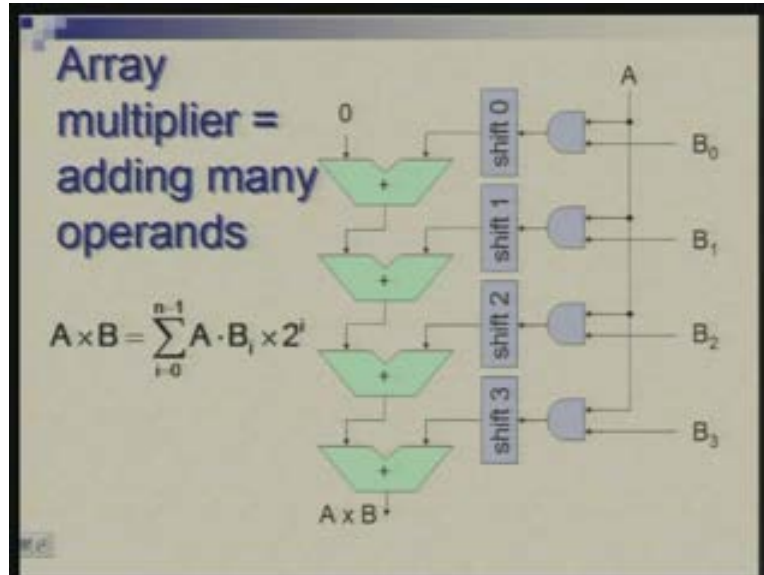
Effect of large fanin and fanout ??

When you take $c = 2$ and substitute its value it gets substituted twice so that is the reason why it roughly doubles every time whereas in this case you have only one term containing carry and when you make substitution it grows by just that amount so next time again you substitute once so it grows by that amount so it is growing linearly.

And another way of explaining these p's and g's is that the generate term is here which we have sort of abbreviated as g 0 and propagate term is essential taking this as a common factor. What is getting multiplied with c 0; what is getting ANDed with c 0 it is a 0 plus b 0 so that is the propagate term. So once we have taken this out in the process we have made things slower because we have added more stages but we have got a circuit which is reasonable and feasible.

Now let us move our attention to multiplication. You recall that I talked about two different types of multiplication: One was when you put the number of multipliers number of adders to add the partial products and the other one was when we did this iteratively using a single adder. So let us look at this approach where you have number of orders as we do not have to go through sequentially because the sequential approach would mean that you have to perform addition, store it somewhere then redo the addition so on. So here also the signals propagate through various adders but there is no storage involved so there is some advantage in terms of speed. So such multipliers are called array multipliers for natural reasons.

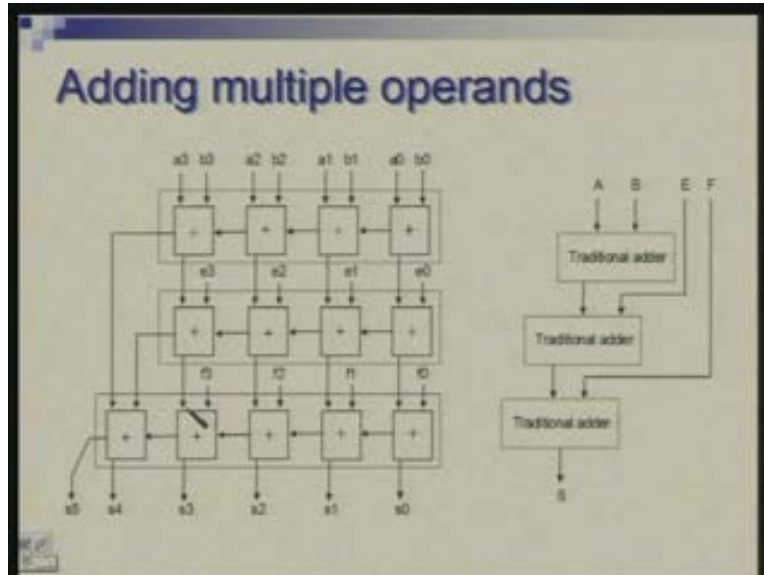
(Refer Slide Time: 34:29)



What we will try to see is how we can speed up such an array multiplier. There are many different approaches for multiplication; you can have actually tree multipliers. For example, you are basically adding four terms; rather than adding them in cascade what you can do is you add two terms separately, two terms separately and then add the two sums. So when you have to add several terms you can actually form a tree so that is another class of multipliers. We focus our attention on these so-called array multipliers where you are adding different terms in a cascade form.

So we will look at the question of addition of multiple terms. I am not quite showing a multiplier here but a circuit which adds four numbers. In the case of multiplication these numbers will happen to be those partial products but let us say we have four arbitrary values to be added; you will have 1 adder which adds two terms, next one adds third, next one adds fourth term. So, suppose we put the usual ripple carry adder for each of these stages then what is the consequence. The signal has to propagate horizontally; there is a carry rippling through and there is propagation vertical also; you are going from one stage to another stage. So the last thing which will get computed is the MSB of the final result; it has to account for propagation in this direction as well as in that direction.

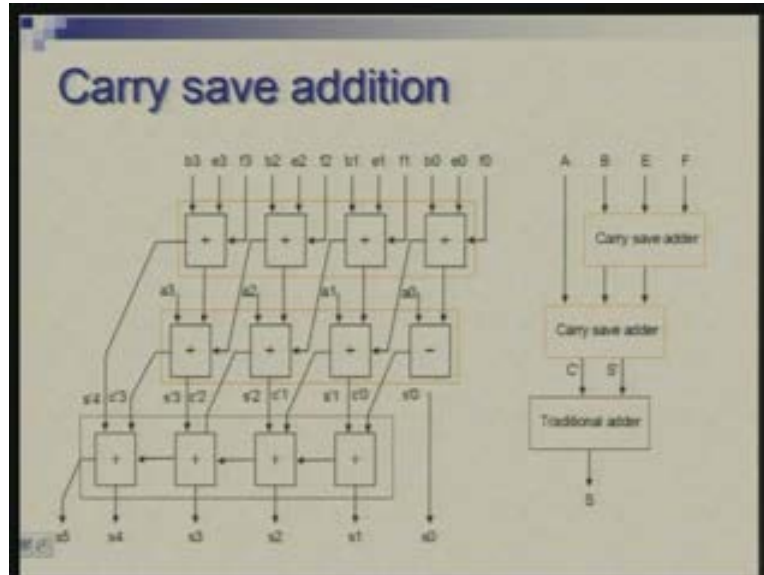
(Refer Slide Time: 36:25)



So here what we can do is since we have to go to the next stage we have to wait for signals to propagate vertically; we can take advantage of that and prevent propagation of carry to just the left neighbor of 1 adder cell. So logically it is equivalent that let us say this carry instead of going from here to here suppose it was to be sent to this one this adder (Refer Slide Time: 37:02) has to wait for some of this to come down. So meanwhile the carry of this also can be made available here and we can also postpone this carry let it go to next stage and so on.

So what we are doing here is that rather than making carries go to left we make them go diagonally left and down and therefore the additional delay which might come because of leftward carry propagation will be cut down. So here is a circuit which will do that.

(Refer Slide Time: 37:40)



Therefore, now you can do this; this is called carry save addition because we are saving carry for the next stage. You can do this upto some stages but the last stage has nothing else to fall back on so in the last stage carry has to propagate which might give an impression that you would require one extra stage so you keep on saving carry for the next stage and the last one you add. But what happens is that in the first stage since you have no previous stage here to feed carries you can actually have the third operand fed as carries.

So, if you really see let us go back to this expression of the sum (Refer Slide Time: 38:42) or even if you look at that it is symmetrical with respect to a i, b i and c i so it is only our interpretation that one of these we are calling as carry input but as far as the circuit is concerned you can think of this as a 3 input and 2 output adder. It reduces 3 inputs to 2 inputs and these inputs are interchangeable. So what we have done is that three of the inputs have been added in the first stage itself. So, from the first stage you have sums coming out as well as carries coming out. Both have to go to the next stage so the next stage is taking these sums and the carries; carries have to move diagonally, sums have to move vertically and the third **sorry** the fourth input gets added here; so a i's have got added here so one could have written a, b and e in the first stage and f at the next stage but these are all interchangeable.

Well, this is showing as.... plus **there is a** resolution problem..... this is showing as minus take it as plus. So, in the last stage we have again sums coming out and the carries. **carries are** These carries are being added at the sums position and because we have to have a carry propagation here also. So you would notice that the number of these adder cells is exactly the same no more no less but they have been rearranged in such a manner that **delays are** delays are smaller. **So how much is** Let us compare the delays.

In this case what is the maximum delay; what is the longest path you can think of?

You try for many inputs try to go any output the longest path would be obviously something which leads to this; and there many ways because of any horizontal movement or any vertical movement you could either go like that and like that so that makes it 1 2 3 4 5 6 7 or if you go like this 1 2 I am sorry 1 2 3 4 5 6 7 whichever way you go there is a chain of 7 adders.

Now, here in the last stage you have one extra you are noticing one extra adder here; our numbers were four bits; because as you keep on adding the magnitude keeps on increasing and you have to accommodate for extra bits so these carries you do not throw away let them be absorbed by an adder here. So there is a delay of seven units here whereas in this case again we can try to find what is the longest path. You have, for example, let us take paths leading to this we have 1 2 3 4 5 6 (Refer Slide Time: 42:34). There is a saving. Since we are talking for small circuit it does not show so much. Or we go way other path let us see that nothing is longer than 6; 1 2 3 4 5; 1 2 3 4 5. So the other path also are of five length; this is the longest path of length 6.

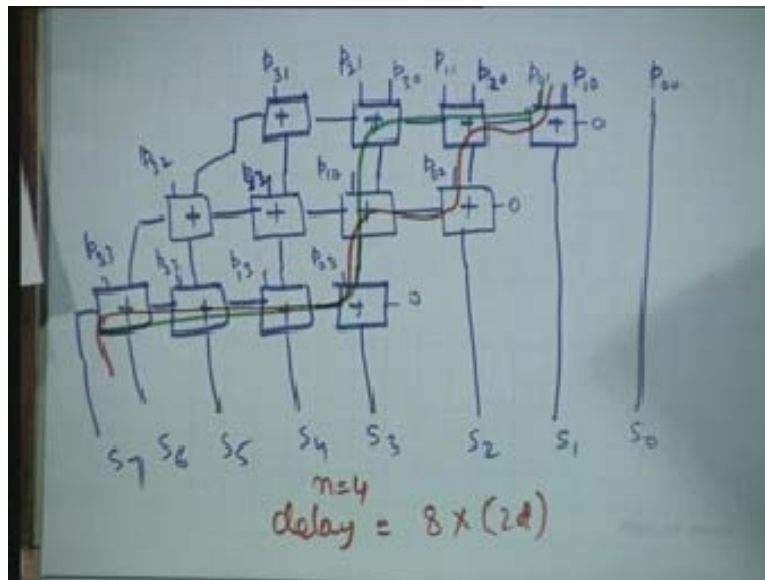
One more thing which can be done here in this is that you can replace this with a..... the last stage could be actually carry look ahead adder. See, carry look ahead adder is more expensive but faster. We have seen that you have to put extra logic for carry computation so we can afford to have one of these adders that means the last adder has a faster and expensive adder and then effectively it will be less than 6. Wherein in this case (Refer Slide Time: 43:43) if you were to speed up you will have to speed up at each stage to make sense and it will be much more expensive.

Now let me show you this principle applied to a 4 by 4 multiplication. Let me switch to.... let me denote by p_{ij} rather a_i and b_j so we have to have a_i 's multiplied by b_j 's. Well, you cannot see that now. Recall that we had A multiplied B equal to $\sum A_i \times B_i \times 2^i$ raised to the power i so 2^i raised to the power i is only a matter of waiting it or shifting it which we will do through wiring. So A_i each bit of A will have to multiplied by each bit of B effectively so I am [d...45:17] to simplify things denoting with p_{ij} . So we have a p_{ij} 's for i going from 0 to 3, j going from 0 to 3 and we have to somehow feed these two various adders. So to this let me feed p_{00} p_{10} p_{20} p_{30} so this is A multiplied by B_0 the first term and the second term we need to add is p_{10} p_{20} p_{30} sorry 10 20 I made a mistake, let me redo it.

p_{00} actually will stand alone; I will put p_{10} here, p_{20} here, p_{30} then next will be second index will be 1 so p_{01} comes here, p_{11} p_{21} and p_{31} . So this is the first stage where I have added A multiplied by B_0 and A multiplied by B_1 . Then in the next stage I will start adding p_{02} and so on; p_{12} p_{22} p_{32} sorry sorry p_{22} p_{12} p_{02} and then p_{32} this can be added here and in the last stage we add p_{03} p_{13} p_{23} and p_{33} and we have we require eight result bits; let me call this S_0 S_1 S_2 S_3 S_4 S_5 S_6 and S_7 ; these initial carries are 0 so we have this with normal carry ripple through and you can see that the delay of this circuit would be again declared by the path leading to S_7 output and the longest path would be this. Let me trace it out. So, for n equal to 4 this delay equal to 1 2 3 4 5 6 7 8 eight times delay of 1 adder which is $2d$.

In general you can say that this is..... You are going to; you have propagation horizontally, you have propagation vertical and you also have some horizontal propagation which is across these stages (Refer Slide Time: 51:17). So, if you extend this if you extrapolate these two ends you will have **one two let us see**; actually it will be easiest if you look at this path (Refer Slide Time: 51:47) which is of the same length but easier to quantify. So you have some horizontal propagation, some vertical propagation, some horizontal propagation. So you can calculate exactly but you would find that it will be something like three times n . This is approximately n , approximately n , approximately n .

(Refer Slide Time: 52:18 min)



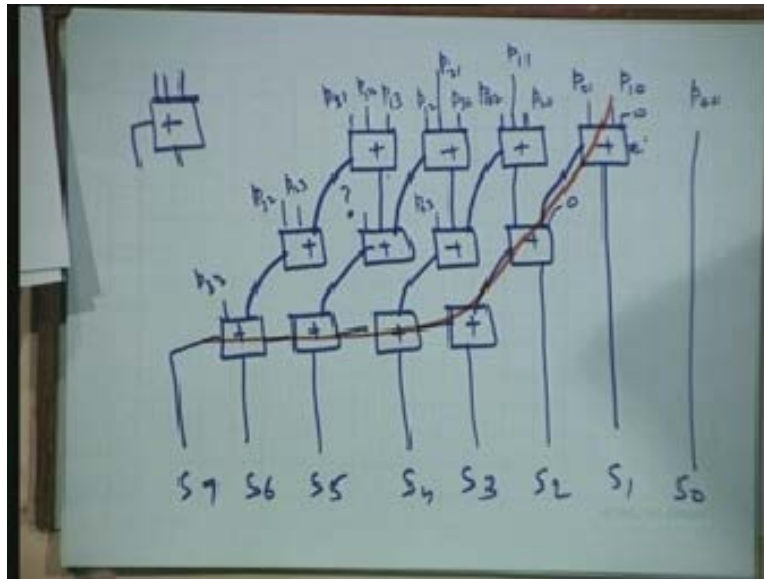
Now, when we do a carry saving, you carry save addition then what do we get?

So, we will require the same 12 adder units and each of these I will look at in this form (Refer Slide time: 52:59) takes 3 inputs you can put them in any order, generates a sum and a carry. So, p 00 I let go straight, p 10 and p 01 are fed here; I have nothing else to feed here, I could have fed 1 more here but there is nothing else to be fed here so I will maybe I can leave 1 0 here; here I can feed p 20 p 11 and p 02.

You would notice that what I am feeding along a column the sum of indices would be same. So 2 plus 0 1 plus 1 are 0 0 plus 2. So here I will feed p 30 p 21 and p 12 so I have a need for feeding p 03 also at this stage so I can leave here. these four terms are to be fed in this particular column and here again I will be left with I will have still four terms so I think I left 3 only so I can take p let me see p 00 p 10 p 20 p 30 this I have finished; p 10 11 12 and I can have p 13; then 02 12 22 and 32 can go here (Refer Slide Time: 55:36); then next one is 03 here p 13 is already accommodated, 23 will come at this stage and 33 will come at this stage. So **I have** I have added all 1 2 3 4 5 6 7 8 9 10 11 there is something missing, can you figure out? 03 13 23 33 02 12 22 and 32; 01 11 21 yeah 31 is missing; I think I should have put that here so that should be..... where is p 22?

So all that one has to do is take care that a term gets added in the right column whether it is added earlier or later it does not matter. **Doubtful whether something is left out here or not** so you can check but anyway this is the overall structure of this adder as this multiplier and you would notice that the delay now is dictated by this path (Refer Slide Time: 59:05). So you have one diagonal movement and one horizontal movement so roughly it will be a proportion to $2n$ instead $3n$.

(Refer Slide Time: 59:13 min)



(Refer Slide Time: 59:30 min)

Summary

- Speeding up addition
 - Ripple carry adder (carry propagate adder): $O(n)$
 - Carry look ahead: $O(\log n)$
- Speeding up array multiplier
 - Using ripple carry adders: $\sim 3 n d$
 - Using carry save adders: $\sim 2 n d$

So let me just conclude; what we have seen is that as far as addition is concerned and same would be true for subtraction the key idea was to go from ripple carry to carry look

ahead and go from $O(n)$ type of delay to $O(\log n)$ type of delay; why we get $(\log n)$ is because when you do multiple levels of look ahead; suppose **we had** we went from 4 to 16 we got two levels; if suppose you have to go for 64-bit addition so we had one unit looking across 4, another unit at the next level will look across 16 and then yet another level will look across 64 so it grows like a tree and the number of such look ahead levels will be given by.... will be proportional to $(\log n)$. I am not putting a base here; if you have individually look ahead across 4 so it will be $(\log n)$ to base 4 but that is only proportional factor whereas in case of multiplier we moved from delay which is proportional to $3n$ to delay which is proportional to $2n$ by doing carry save addition; there is a sort of 3:2 roughly which is the saving. I will stop with that.