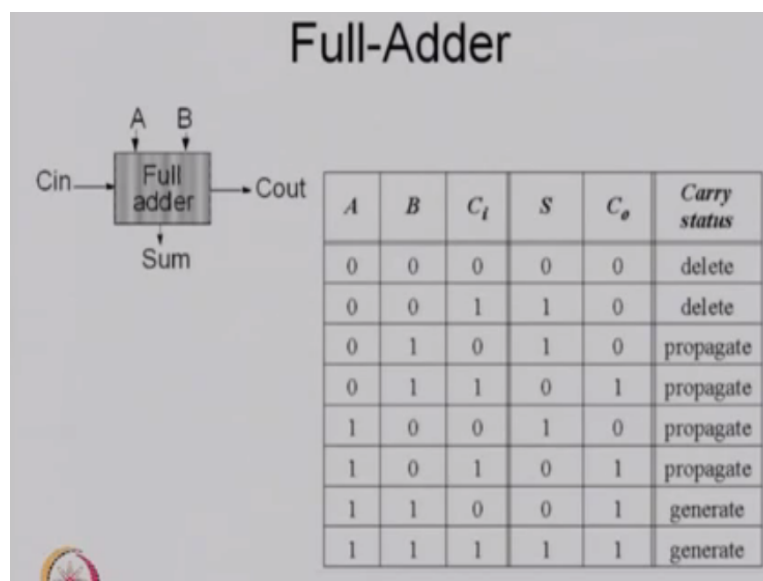


Advanced VLSI Design
Prof. A. N. Chandorkar
Department of Electrical Engineering
Indian Institute of Technology – Bombay

Lecture – 12
Arithmetic Implementation Strategies for VLSI – Part III

We are last time already started working on adder circuits. Today, we start; we will just recapitulate what we did last time, we said that there is a full adder requirement for almost all arithmetic data parts.

(Refer Slide Time: 00:37)

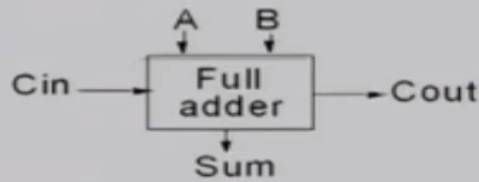


And in this full adder, we have possible shown here is 2 input A and B which receives an input C in and produces a sum and C out. The typical, the 2 table for such adder is shown on your right which says A, B and C_i are the inputs and C_i is the carry input, SNC out are the output which is sum and output carry and we also discuss last time that depending on the input as well as the carry in.

The status of a carry will be either to keep zero or to propagate it or to generate the one, delete means 0, propagate means 1, propagate means passing C out as C in and generate means create ones. So depending on the input requirements or input data available and the carry input available, we can always figure out that we can generate specific carry or delete or propagate.

(Refer Slide Time: 01:33)

The Binary Adder



$$\begin{aligned}
 S &= A \oplus B \oplus C_i \\
 &= A\bar{B}\bar{C}_i + \bar{A}B\bar{C}_i + \bar{A}\bar{B}C_i + ABC_i \\
 C_o &= AB + BC_i + AC_i
 \end{aligned}$$

This we discussed last time, we also said that a typical adder logically can be explained as $\text{sum} = A \text{ x } B \text{ or } C_i$, where C_i is the input carry and we can even, we can also expanded logically into sum, sum of product terms, which is $AB\bar{C}_i$ and something like this and a output carry will be also a sum of the $AB + BC_i + AC_i$.

(Refer Slide Time: 01:58)

Express Sum and Carry as a function of P, G, D

Define 3 new variable which ONLY depend on A, B

Generate (G) = AB

Propagate (P) = A ⊕ B

Delete = $\bar{A}\bar{B}$

$$C_o(G, P) = G + PC_i$$

$$S(G, P) = P \oplus C_i$$

Can also derive expressions for S and C_o based on D and P

Note that we will be sometimes using an alternate definition for



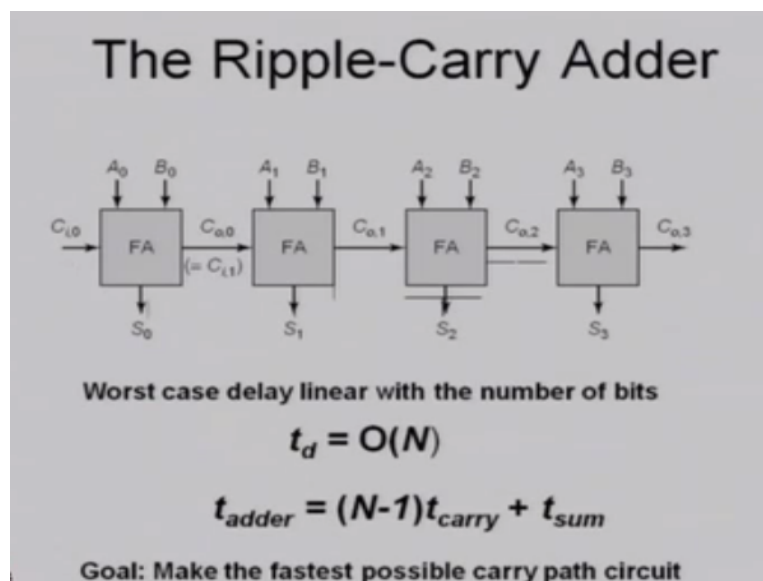
Propagate (P) = A + B

We also said that instead of every time doing such functions, other method of doing is to generate 2 terms. Basically, 2 terms called propagate and generate. Propagate P essentially is $A \text{ x } B$ and generate term is $A \text{ dot } B$ and there is also a delete term sometimes required which is $\bar{A} \text{ dot } \bar{B}$. If we go back and look at the expressions for sum and carry, output carry in the last slide.

We could see that C_0 which is now a function of A and B with through their functions G and P . Then it can be written as $G + PC_i$ and some can be written as P_x or C_i . Now one can see from here that if I know from, clearly from this expression that if C_i zero that is input carry is 0, then the output carry is always G . On the contrary, if inputs A and B are same, that means A_x or B is 0, then P_0 and then C_0 will always equal to G .

So since we know this that in case A and B are not same, P is one and since A and B are not same AB is 0, which means G_0 , that means C_0 is C_i .

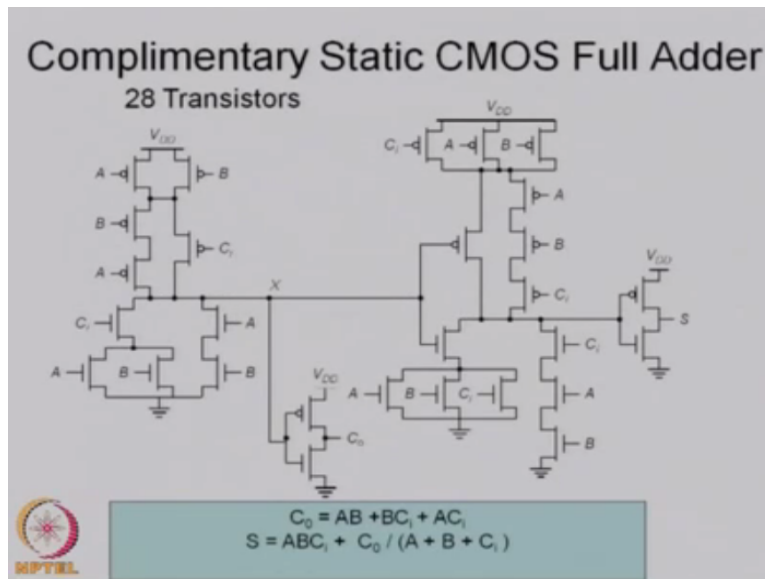
(Refer Slide Time: 03:26)



This is exactly what we wrote in the table and these expressions can actually represent the two table which we wrote back. I am going ahead, we can see that such a using this functions, we can generate something like a simple adder. This is a 4-bit adder shown here, you have 4-bit full adder shown here, each receives 2 bits, $A_0, B_0, A_1, B_1, A_2, B_2, A_3, B_3$, FA the 2 words and this is the input carry.

But the fact is here you can see here unless the output carry generated by this first adder, one cannot do addition for this adder, unless it generates the next carry, it cannot do something from here and unless we find out here, we cannot find out the final carry and final sums. So essentially, if you see the adder requirement, time required to go from input to the output when the 4-bit output can be made available or N bit output is available.

(Refer Slide Time: 04:39)



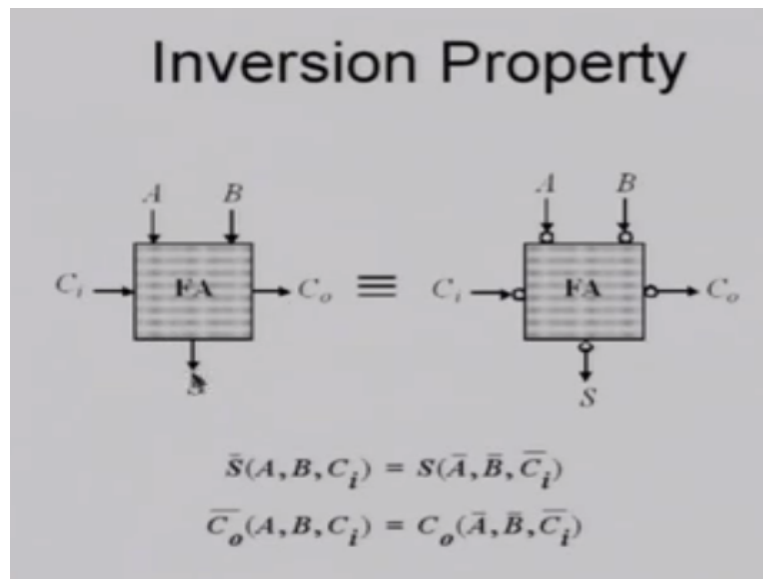
The added time is essentially $N - 1$ t_{carry} each 1,2, 3 carries and $- 1 t_{\text{carry}} + 1$ some registered which final sum has to come, so this is the added time. Now our; what is the goal at the end of the day for any data path or any arithmetic? We are looking for fastest possible carry path. We said that okay such a function can also be implemented using a Static CMOS.

And in this we could show here that since we know the expressions for C and S, in this form, we can actually create C term from here and finally using that C term we can create S term here. So, now this kind of implementing the; this 2 expressions requires almost 32 transistors, sorry 28 transistors in total and if you add little more inversion here, it may be actually 32 transistors are normally required to actually create a normal static CMOS adder.

I may now show you that in any full adder circuit, we already said that there is a path which we say critical path that is from input to the output, the verse delay path is called the critical path and we are actually always trying to deduce the delay in the critical path and we are also trying to simultaneously deduce power dissipation and that was important because we are trying to deduce power dissipation, is the major criteria of today's technologies or today circuits.

We are working on 45 nanometre down technologies and the power supply voltage is around 0.8 Or 0.2 volt or even lower these days in some chips, so for those purposes, we are really looking for a very low power circuit and features of this present discussion from now onwards would be also looking into low power part and the same circuit even if it is not as speeded up as normally would have been.

(Refer Slide Time: 06:49)



But at least trying to reduce power and increase speed as much as we can. So what we did? We actually, have just now said, if you see the previous slide, if you see this expression, since there are XOR GATES are available; XOR GATES are available are required to generate A, propagate function. So obviously based on an XOR GATE, one can make an adder, which can be shown.

(Refer Slide Time: 07:12)

If We expand XOR Function & then Represent SUM & Carryout in terms of ONLY AND/OR GATES, we CAN REDUCE TRANSISTOR COUNT AS WELL AS DELAY IN CRITICAL PATH

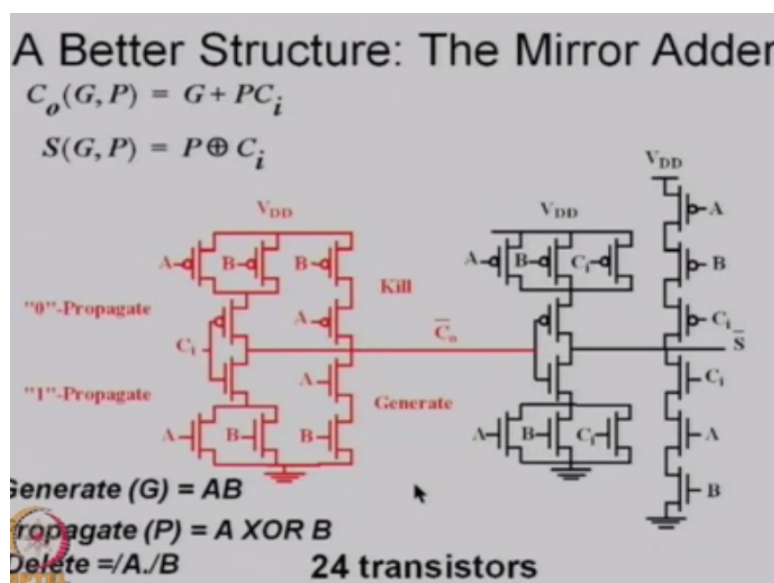
A 28 Transistor Full Adder is one such Adder.

That if C or XOR GATES, one can then actually create an interesting normal adders, which is essentially 2 bits added at a time, okay. Please remember they have 2 inputs A and B and they also have one input carry and then they generate the sum and the output carry. Typically, as I said require 32 transistors. However, I just now or earlier shown you, if we expand an XOR

GATE function XOR functions and just represents sum and carry outs in terms of and OR GATES.

One can reduce the transistor count as many and also it can also speed up the circuit. Typically, the circuit which I showed you last has 28 transistors may be you can see from here, this is 28 transistors circuit, which is essentially has using only GATES, please remember there is no XOR function here, we are just representing a s some of the product terms or at all terms in this kind of complex logics representation in mean terms method or max term methods.

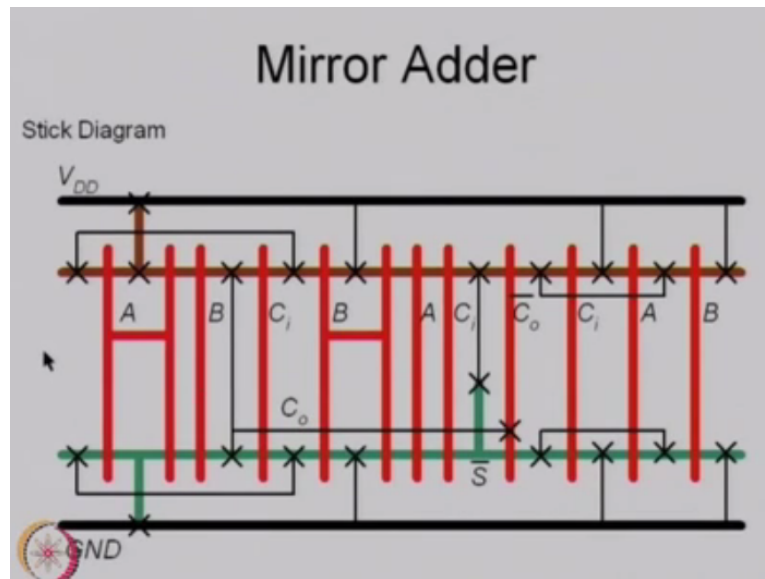
(Refer Slide Time: 08:05)



We can actually create sums and carry using only 28 transistors. We can also look into the better structure of the same as we discuss this. We can try to put things more in the symmetry form across the carry propagation path upper and lower, and upper and lower, is you can see is identical is symmetric to the nature. It generates both AB, A XOR B and also A bar, B bar and using modified form of the same, you required 24 transistors, okay.

You require 24 transistors to do the same job. This of course, you can have this circuit anyway analysed, it is very simple, we are trying to generate C through this A, B and these are the inputs corresponding to that bits you have, please remember again that when A and B are same, then the generate function is AB is available. If A and B are different, then G is always 0, and that fact can be actually used in the implementation.

(Refer Slide Time: 09:02)



This circuit, which I shown here is laid out on and to create a chip and this is called the stick diagram. The yellow redlines of course are the poly lines, green lines are the diffusion lines, this is P channel devices, these are N channel devices, please remember this is source drain, source drain, source drain across poly source drain. Similarly, for P channel this is source, this is drain and think of that or this is source this is drain.

(Refer Slide Time: 09:35)

The Mirror Adder

- The NMOS and PMOS chains are *completely symmetrical*. A maximum of two series transistors can be observed in the carry-generation circuitry.
- When laying out the cell, the most critical issue is the minimization of the capacitance at node C_0 . The reduction of the diffusion capacitances is particularly important.
- The capacitance at node C_0 is composed of four diffusion capacitances, two internal gate capacitances, and six gate capacitances in the connecting adder cell .
- The transistors connected to C_1 are placed closest to the output.
- Only the transistors in the carry stage have to be optimized for optimal speed. All transistors in the sum stage can be minimal size.

So this stick diagram represents so called 24 transistor adder. What are the advantage we said last time? We said NMOS and PMOS chains are complementary and symmetrical, completely symmetrical. A maximum of 2 series transistors can be observed in carry generation circuit to reduce this delay. When laying out the cell, the most critical issues the minimisation of capacitance had node C0.

At this node, which is the output node, capacitances are associated from all kinds of transistors sitting at this point, may be if you look at the circuit once again you can see from here, the capacitance from this side, capacitance from this side, capacitance from this side capacitance from this side, capacitance input side CGD kind of this, so there are too many capacitances which are associated at the output node of C0.

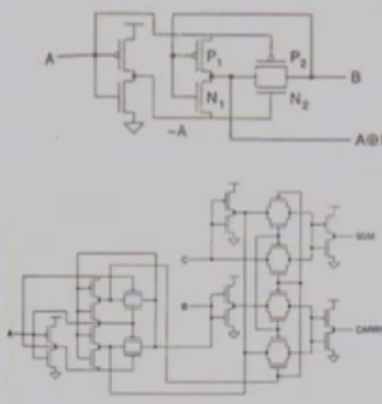
And if you go find for the down to some they are further some more capacitances are added. So in the case of the capacitances at node C0 is composed of 4 diffusion capacitances to internal gate capacitances, six gate capacitances in connecting adder cell. Now, this means larger the value of capacitance at a node the time taken to charge or discharge will be higher and therefore speed will be reduced.

Therefore, the transistor which are connected to C_i must be placed closest to the output. This is the standard technique of any logical layouts which we always observed. On the transistors in the carry stage, you have to optimise for optimal speed and therefore their sizes has to be so designed, so that the net de provocation delay is minimal. Now, based on this, I just now shown you equivalent transmission line adder was implemented.

(Refer Slide Time: 11:24)

Transmission Gate Adder

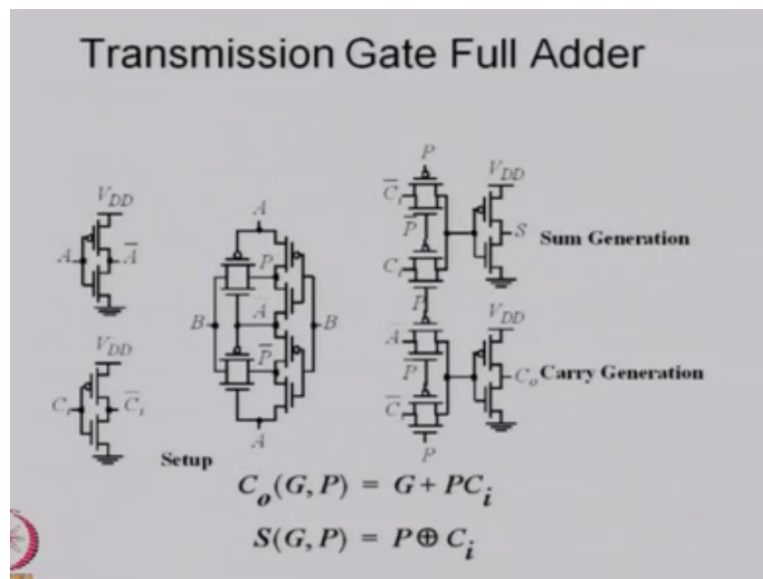
- Thus, by using 4 transmission gates(8 transistors), 4 inverters(8 transistors) and 2 XOR gates(12 transistors), an adder can be constructed.
- We characterize this for -
More area,
+ Equal Delay for sum & carry.



Transmission gate adder was implemented, basically this generate $A \oplus B$ and then from there, you can actually generate equivalent of this by using 4 transmission GATES, 8 transistor, 4 invertors, 8 transistor, 2 XOR 12 transistor. Essentially, I am saying it is $16 + 12$, 28 transistor circuits same as what we have and that can generate a function which is someone carry for you.

The advantage of this is it should equally delay for both sum and carry, there is an issue which I have not earlier discuss, the most adder problems come at when you are bit size increase as 64 bit or 32 bit or higher, then the delay which it will create for the last bits will be so large that the some may come later than the carry propagate times and you have to actually equalise the delays.

(Refer Slide Time: 12:23)



And to do this, there is always an issue, how to minimise and also equalise the delays. Typical transmission get full adder, blocks are shown here, this is for some generation part, this is carry generation part, this is set up part to create A B, A bar B bar, AB bar and then using this, we can create the sum as well as the carry. Now, why I am showing you this, most people believe that the transmission line get adder will be low powers simply.

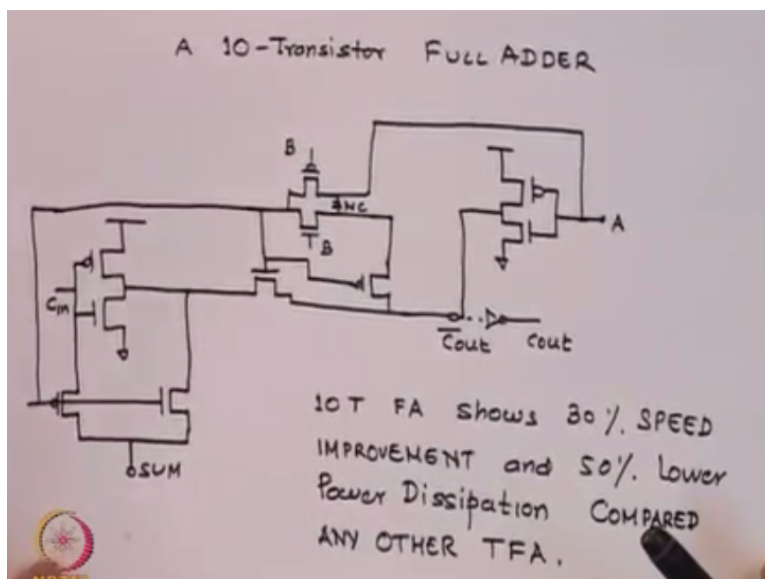
Because, there is no direct power supply requirements as far as the transmission gets are concerned. It is only different from their inputs is essentially a kind of marks, since it reduces the power, this became very popular as far as the low power circuits are concerned but the area wise also the speed wise, they are not really very high speeds. So, to deduce power as well as the; to see that the speed is not as bear or literally brute.

(Refer Slide Time: 13:22)

IN QUEST OF BETTER SPEED A
 LOWER POWER DISSIPATION, MANY
 FULL ADDER CIRCUIT IMPLEMENTATIONS
 HAVE BEEN TRIED like
 17 Transistor FA, 14 Transistor FA
 and 10 Transistor FA.
 A 10-T FA is as shown in
 next Slide

Many structures were tried out of 28 transmissions there is a structure which is available, which is called 17 transistor full adder which uses one XOR, then there is a 14 transistor full adder and finally the one which is most lightly use for low power high speed applications, the 10 transistor full adder, which I may now show you as this, you can show from here 10 transistor full adder.

(Refer Slide Time: 13:35)



For example, this is my inverter, this creates C_m , these are two pass gates P channel, N channel pass gate, this is another pass gate, which is this is input; P channel, N channel, no connection please they get. This is creating A bar, this is an inverter; CMOS inverter and one can see from here, I do not want to write now work on it but may be one of them I can show you.

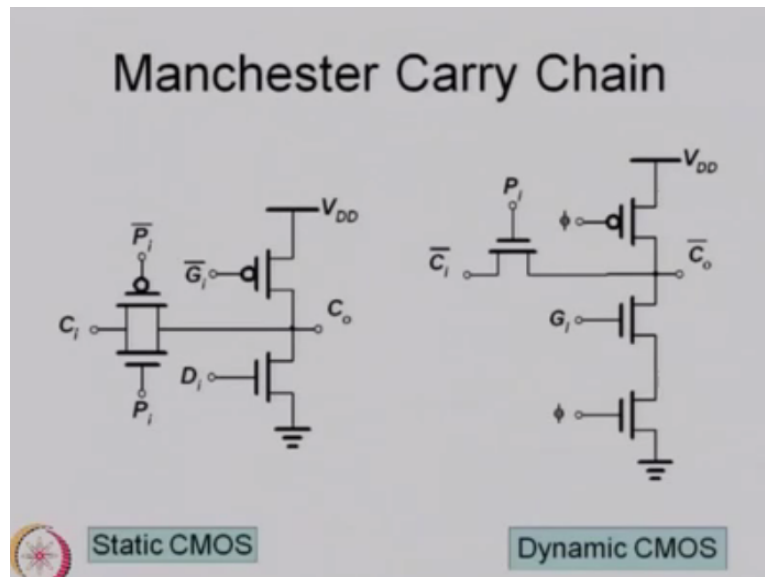
Let say A is 0, let say A is A0 B0, then one can see this input is 0; A0, this is one, so this is one, since B0 P channel turn some; so this zero is transmitted. Since this is 0, since this N channel works okay, P channel works and you can see from here, whatever is the input carry is the sum from your expression, one can see from here if G is 0 and P is one than sum is acquired as only C_i .

So essentially you can put this XOR GATE, this XOR GATE is essentially not full XOR it is called pseudo XOR and say 2-mark system in which we are not connecting this, this is somewhere I made mistake, so one can see from here using only ten transistors, I will able to generate both G and P as well as I can use C_i to create C_0 and C_0 as well as sum. Please remember why we were looking for such structure.

Because we thought that the number of transistors if they are larger they add 2 capacitances, so there reduce the speed any way. The second issue is the area, the VLS have we always worry about the silicon real estate, so if you reduce the number of transistors obviously we should be able to reduce the area and the third of course if there are pass gates used, transmission lines gets used.

Then one can think that the power itself will be little lower and it can be showed compared to any other full adder structure shown here 17 transistors, 28 or 32, typically it gives 30% speed advantage and it also reduce 50% of the power. So, if you see that if at all you are going to use a full adder circuit using transmission line gate method, the best available adder right now for you full adder right now is this 10 transistor full adder.

(Refer Slide Time: 17:01)



And this is what most of these represent circuits use whenever they are using static CMOS. Going ahead as soon as we looked into this last expression, we can use; we start looking at this expression again and again, I repeat again a page one obviously G will be 0, so C_0 is C_i . If P_0 , C_0 G is anyway, P will be 0 only when A and B are not, they are same and in that case G will be either 0 or one.

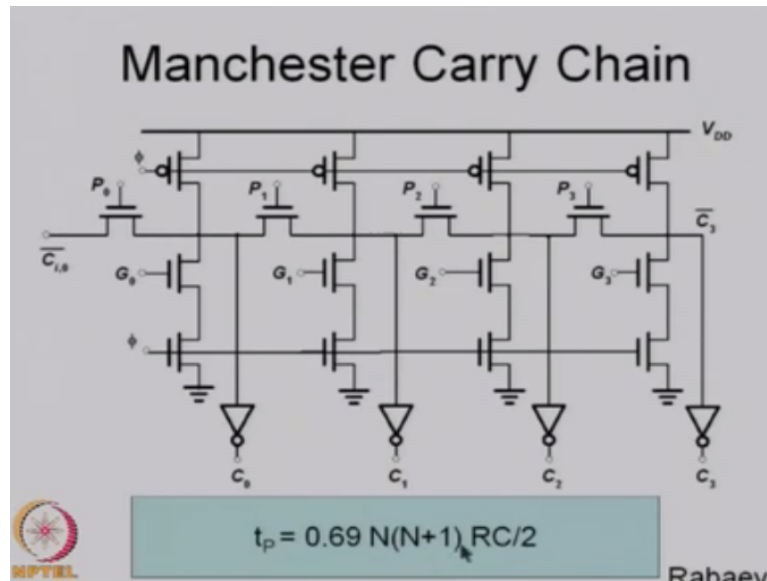
So, C_0 can be either 0 or one, so I know what is the output carry and sum always is if p is 0 one then one obviously one XOR C will depend if C_i is 0, then it is one, if it is C_i is one, it is zero. So, I actually know my sum as well as my carry as soon as I know what is my G and P for a given data, this fact can be used in generating a chain of adder circuits which is called Manchester carry.

He had that first cell of a Manchester carry chain, which shows there is a, this is of course based on static CMOS, but one which is most likely used by us to reduce the power is static dynamic CMOS, in which you have a P channel transistor and N channel transistors which are dynamically connected to a clock ϕ , whenever ϕ is 0 C_0 goes high, when ϕ is 1, it evaluates. So, this is standard dynamic process.

The carry is inputted through a pass gate it can be a transmission gate, a CMOS, right now it only shown in channel whose driven by P_i that is the propagation term, okay now please remember if A and B are same either P you can see from here depends on if A is 0 and B is one or A is 1 and B is 0 and P_i 0 otherwise it can be 0 or 1 depending on the ENB values, so the corresponding to only when P is one that the C_i passes.

Otherwise, C_i bar does not pass so this is the logic and therefore whatever we already said from the expression, a P_i is one, then G is zero, and therefore C_i , C output and input, so this is direct transmission process. Otherwise, if that is not possible then P is 0, in that case, this will be blocking and so only it will be C_0 will be G . So, this essentially what we said in the expression can be represented by simple dynamic or static CMOS circuit.

(Refer Slide Time: 18:48)



Here is the 4-bit adder shown here, you can see here this is P_0 , P_1 , P_2 , P_3 which are propagate functions given to four pass transistors, this is one cell of carry Manchester carry chain I showed you, the output of course is inverted because you are getting bars here, you may; it is an advantage to put always bars because generally some other day in a logic I explained you, why zeros are much preferred compared to ones.

Because there is a power supply droop problem in most cases. So, passing zero is much easier. At the end of course, you may generate the outputs once for all by inverting them. So, the method here is simple and you keep depending on P_0 , P_1 , P_2 , P_3 values that is $A_0 B_0$, $A_1 B_1$, $A_2 B_2$, $A_3 B_3$ values either this input carry will propagate directly, if not you have to generate at wherever P is 0 that block has to generate the carry for the next block and keep doing this.

If you look an equivalence of this in the circuit method this is like OR and at this node this is, this transistor will act like a resistor, very small capacitance will also, the output capacitance here + the capacitance of all this node here which come from here which come from here,

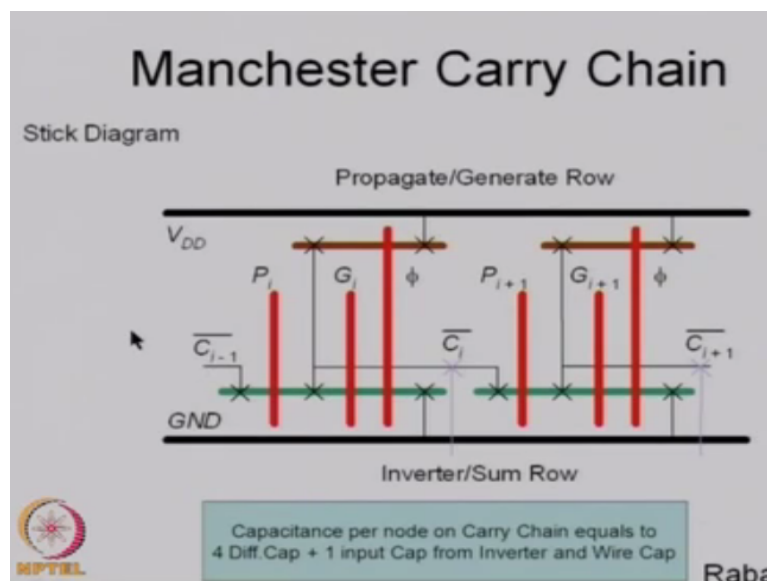
which come from here, so there is a capacitance here, so this is like a RC network, RC, RC, RC.

And using NMOS time constant method one notes that if R and C is the net R , please remember R is essentially some of R_i or R_j s, where j is the other number of such R s, so it is like R_1, R_2, R_3, R_4 , they may be same, but in case they are not they can be summed up by $\sum R_i$ or $\sum R_j$ and then using NMOS theory, we know transmission line theory, we say that delay associated is $0.69 N$ times $N+1$ RC by 2.

Now, this fact is very relevant to understand because if you have a N is larger, for example, this is only 4 bits, if you have 16 bits, 32 bits, 64 bits, obviously the delay will be proportional to N square, that means the larger the bit size additions you do, larger is the delay which you are going to get, therefore it is essentially a ripple carry system in which carries a repelling and as many stages it has to ripple that much delay to create.

Of course, please remember in this expression, I am only showing the propagation delay because of this RC network. In real life you can actually do a spice analysis and you can figure out that you will come very close to this value, the reason why we do not calculate any time constant for evaluating G s, so you should add little more time for the final G essentially, because during this G calculation anyway the other processor is also being continued.

(Refer Slide Time: 22:13)

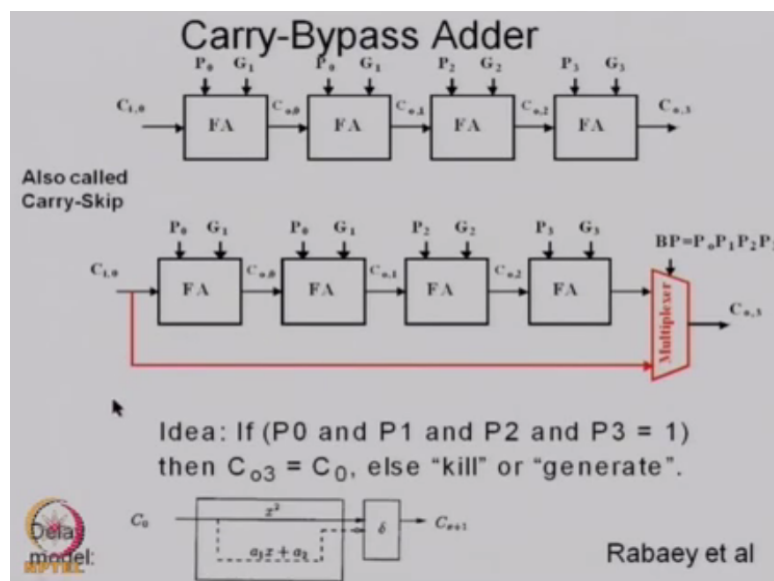


So, they do not really contribute times but the final one, because this final carry and with this it will decide time and therefore additional some more time but this time of evaluation is

smaller than the delay which is coming through this and therefore typically one may say the net propagation delay is roughly equal to $0.69 N \ln(N + 1) RC$. Typically, layout of such a carry chain is shown here.

Please remember again and again, actually capacitance per node on carry chain equal to 4 diffusion capacitance, one input capacitance from the inverter and wire capacitances. So, if you wish to speed up the first thing you should try is to go on technology in which the net capacitance of this is much smaller increase the size of transistors so that the R is minimised. But increase the size of a transistors essentially you are giving an area, okay.

(Refer Slide Time: 22:58)



And since you are giving a larger area, larger wire will require will produce larger currents and therefore larger powers. We looked at into the expression again and again for P and G and we figured out P , G , and C and S . We already said if P is one, G is always 0, so output carry is same as the input carry, this was known to us. So, if you have a ripple carry adder, slightly modified shown here.

You have 4-bit adder has what we have shown, they may be Manchester carry also or normal full adder also. In that case, if we see that P_0 into P_1 into P_2 into P_3 that means for all the 4 bits are not identical in any of this bits are like 1001, you may have now 0110, so they are complemented all the times. In that case $P_1 P_2$, $P_0 P_1 P_2$ all are ones and if they are ones because they are XORs of A and B .

Then the product we called BP which is P_0, P_1, P_2, P_3 that will be one. We now say this is the multiplexer, which is receiving a carry through chain of adders at zero and that one we have the input carried directly given. We know very well, we just now said if the propagation is P_0, P_1, P_2, P_3 are one, obviously G is zero, G_s are zeros and therefore C_{out} is n , so what we say okay bypass you give as soon as the select signal of a multiplexer is one, bypass the carry to the output.

In any case, if this is not any one of them is not equal to 1 which means G has to be evaluated for that and that will be creating the carry. In that case, whichever part of the chain you have to go through generate G and move ahead, you keep moving and finally at that when since it is 0, the zero means the final carry will propagate. So it is not that it will reduce drastically huge powers.

But for a particular data in which or at least even a partial data it may actually bypass the carry directly to the output without actually evaluating it. Now, this is called carry skip adder once a while because it skips the carry directly to the outputs without going through the adder parts. Now, typically if you look at the equivalence in a signal flow part, you can see from here that you have some kind of delay coming through the delta function which creates this.

And you have a feedback path which is nonlinear and that is where the whole issue, in all this in, why I showed this kind of signal path is many people believe that everything is linear in the system, in real life nothing varies because all transitions character itself are nonlinear and because of that the actual delay between this and this is not so linearly connected as one things.

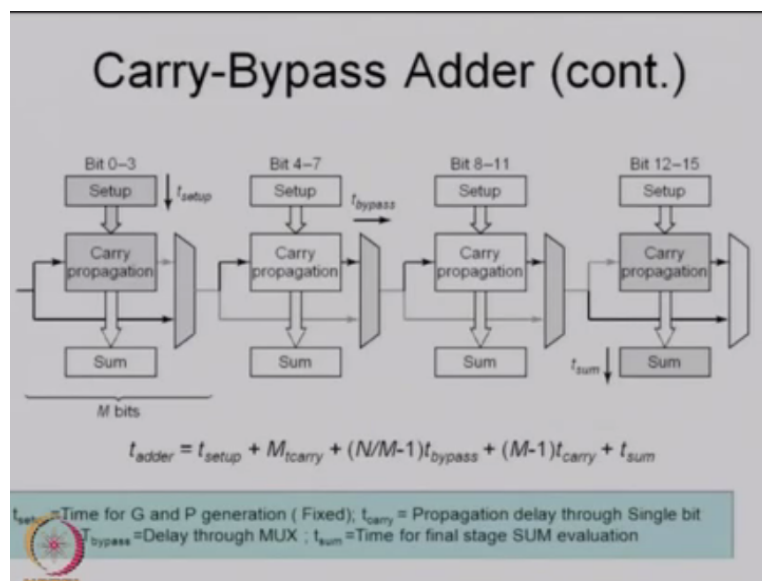
And that is where the issue of the equalisation of delay starts varying as extremely when you increase the number of bits. This model essentially is to show you that the carry delay model is not linear and since is not linear, one has to always worry because you can see it is one path of the different delay the other path of the different delay. So, equalising the delay at the output is a major worry every time.

So you may require some kind of register which will actually wait for all the data to appear and then put it out which essentially means, you will reduce the speed. In a full adder, if the carry has to propagate it will take longer time, if P_s are ones then it will directly, so the time

taken from this path to time taken to this path is not same. So for a given data, this will be available early or in the other data it may not be available.

So, for the next block to receive it, it any has to wait because it does not know when to take that data, so even if you ask period data, it did not really work out well because for other data you may have to still wait okay and this is the model which I keep showing that the delay model is very important and therefore something has to be done so you may increase the multiplexer delay itself and if you do that and then why do this?

(Refer Slide Time: 27:43)



Okay so there is a; there is always a VLSI issue in the case of; so as well as logic is concerned everything looks to be great but if you really look at the timing part, you feel it, it is not that you are achieved really great. Here, is the 16 bit added shown here using carry bypass the method is; what we do is? If you have N bit adder divide into number of blocks. Let us say each block has 4 bits, so N bits.

And N is the number of total bits so 16, and I choose n4, so each block has a 4-bit adder going on, so 0 to 3, 4 to 7, 8 to 11, 12 to 15 and we do individual operation for 4 bits. Now the advantage is this that now you are doing 4 bit operations parallelly okay. Now, what we can; why are looking for this? In some cases, you may be actually directly passing the carry but during this process, you are also completing this process.

Because that time all are getting data simultaneously, so you are actually generated, so you have this data output okay. So, the idea is to break into number of such blocks and believe

that some of the blocks will actually great faster this overall delay may be lesser in larger number of bits. So, with the way it is; it is the same carry by pass adder as we have and each of them. This set up time is essentially taken for calculation of P and G and this, so this is called the set up time for each of them.

Then, there is the propagation time which we just now said. What is the propagation time from input carry to the output carry? and there is of course the final some part depending on the last carry you have and of course the first may not. because generally see input is carry this may be fast available to you but depending on whether P1 P2 for all of them is one or zero, you can actually directly pass the propagation.

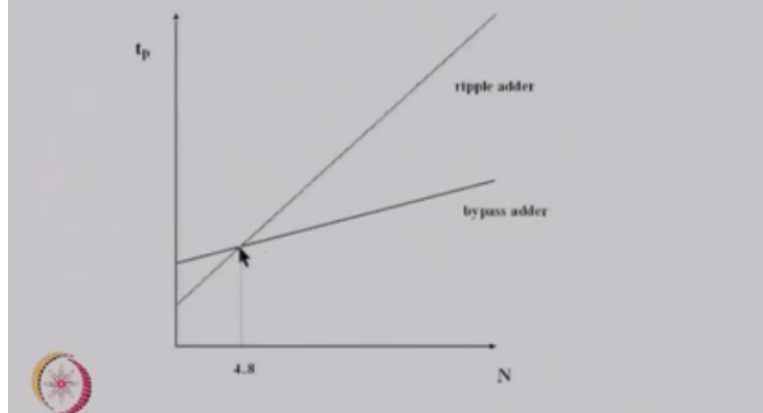
So, if you look at the delay to reach the final sum, this is the sum adder, please remember I am not adding time here simply because during this propagation and some this anyway was performed only the last has to do the final time. So, we say there is the sometime, one sometime, then you have a setup time and all will be setup simultaneously so one set up time then there will be as many 4 bits so you have a carry time for 4 bits.

Let us say, one bit carries one, so m, there are 4 bits will require 4 bit carry time; maximum worse case time. Then seen there are propagation in its are only one, two and three; so $n - 1$; so $M - 1$. So, out of 4, only three are required for total delay, so you say N divided by $M - 1$ into this is called bypass, okay is possible carry and then $M - 1$ carry because you will have to carry for each of them okay.

So, the $M - 1$ carry and finally the sum time. So, this is essentially we say that for a typical 16 bit or any N bit carry bypass adder which has 4 bit has block, one can have this. You can see from here again, if N is larger okay if N is larger, T adder will be larger but if you have at least divided into 4, you have at least N by $M - 1$ primes this, because if N is 64 and you use M -, M say 4, then at least you have one third of 64 that is 20 on numbers carry bypass is only required.

(Refer Slide Time: 31:40)

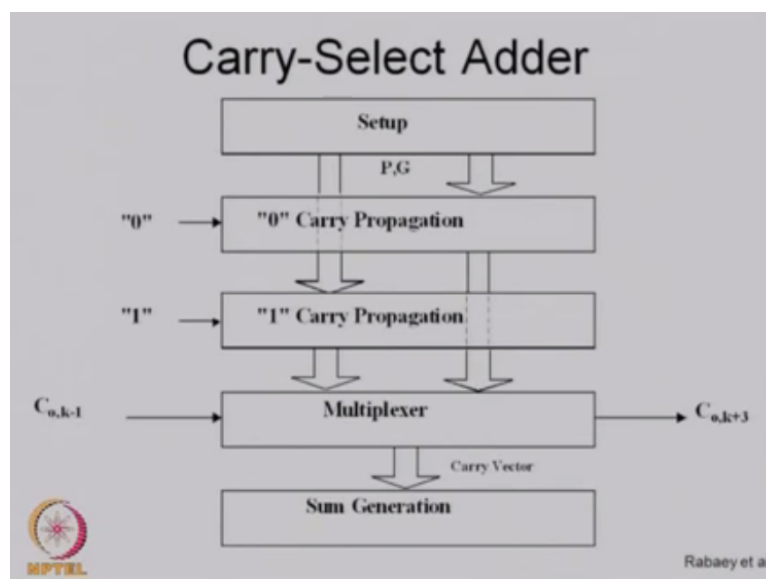
Carry Ripple versus Carry Bypass



So, the idea behind dividing into such blocks is to overall reduce the carry chain time. If one plots the propagation delay versus the carry versus the number of bits which you want to add then it is found that around 4 or 5, if the bits of size is 4 or 5, both ripple carry essentially is better because their delay is actually proportional to this. Please remember this additional time coming from here is the other part in the blocks which we have that will always be there.

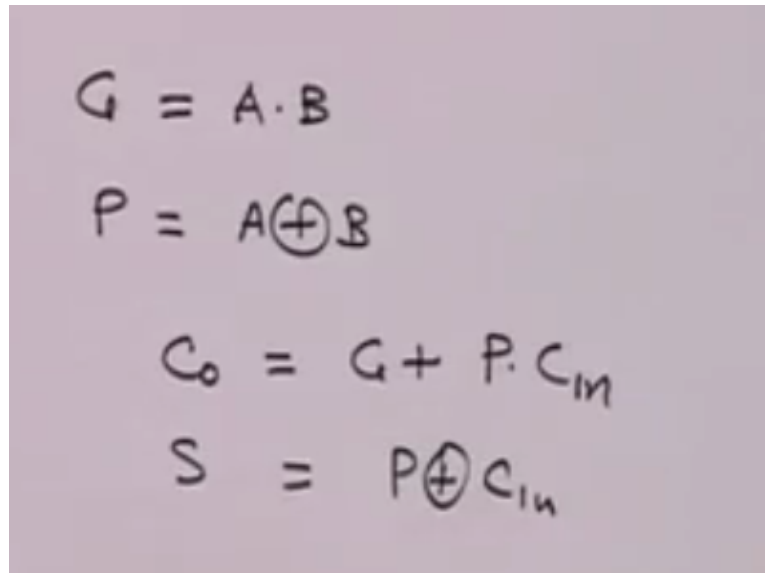
However, if you exceed the 4 bit or 4.8 has the actual number I have shown but any way beyond 4 bits, the carry bypass adder increasing time is much smaller compared to ripple carry because this is directly proportional to N . This is N upon $M - 1 +$; some other terms, so obviously this will; for a larger bit therefore it always advisable not to use simple ripple carries but use bypass carries.

(Refer Slide Time: 32:32)



Another variant of similar carry bypass adder is shown here, which is called carry select. Now, in our expression of P and G, we are seen, I keep repeating again and again that expression and do not leave that expression any time.

(Refer Slide Time: 32:50)



$$G = A \cdot B$$

$$P = A \oplus B$$

$$C_0 = G + P \cdot C_{in}$$

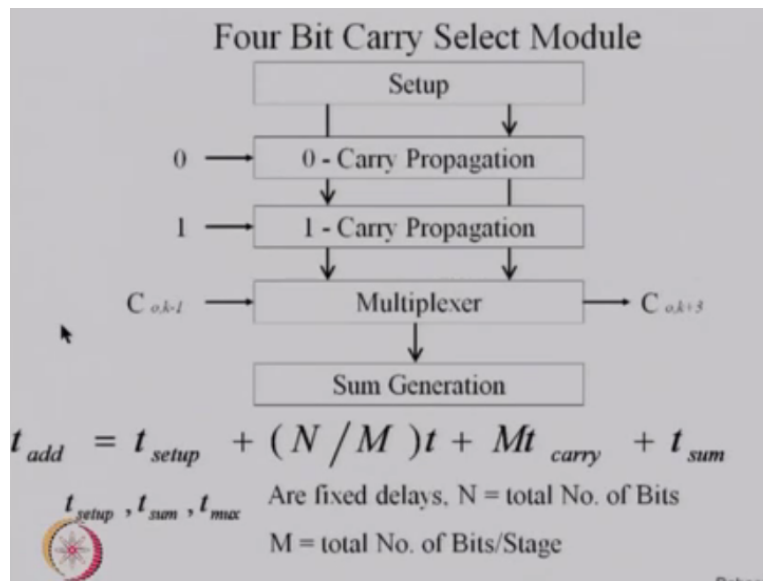
$$S = P \oplus C_{in}$$

In your idea G is A dot B, P is A XOR B and then C out is G + P times C in and sum is P times XOR is in it? Just show and check it; P XOR C in, okay. Now, one can see from here, that either I can have carry generated one or I have a carry generated 0, that means whatever is; there are only 2 possibility, carry can be zero or carry can be one. So, we say okay, take your P and G whatever number and evaluate a carry with the next term either using carry 0 or using carry one.

These in the 2 possibilities carry can be 0 or 1, so we actually evaluate the output to evaluate the output carry. We actually have both terms simultaneously available for any kind of P and G because either it will generate a carry of 0 or it will take a carry of 1. So, using this now, we have now depending on the last carry, we know whether to propagate or not to propagate so if it is 2, then we say okay take this otherwise take this, so we have already generated the 2 term, we are not waiting for finding whether it is 1 or 0.

Depending on the input we receive, we know; what is the next carry output. Because we have already pre evaluated them here okay. Then of course some of course is using this; P XOR Cn we can always generate this; so the idea is same vary bypass in some way but instead of just bypass we say okay we have both possibility availability with carry 0 or carry 1, pre evaluated availability and multiplexer depending on whether in this expression.

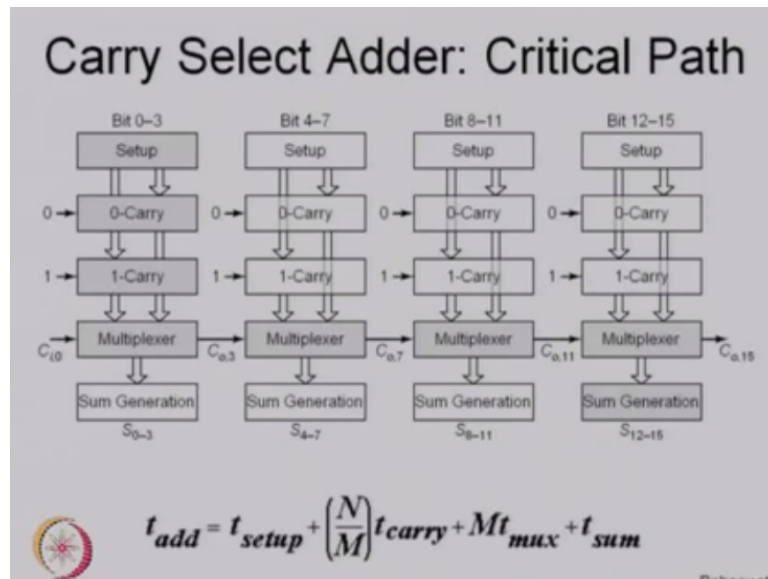
(Refer Slide Time: 35:17)



What is the value of this may decide whether carry will be 1 or 0, so we just accordingly pass the output carry. So, this is trying to save time because we already pre evaluated of course you have added little more time here but that time will be much smaller than propagates. So, here is a typical timing calculation done here for the adder, set up time of course is standard then if you have a 4 bit carry system like N by M.

Now, you do not need because for everything, you have to do this, so $N/M t + N$ times which each for them you will have to carry + t_{sum} this is of course t_{mux} , okay I forgot here, this is t_{mux} . So, one can see from here they are fixed in N total number of bits, M total number of bits per stage. Since we can now see; it is we already reduced to some extent, initial timing of calculating this, the net carry time, the nets added time will be smaller than the normal carry save adders okay.

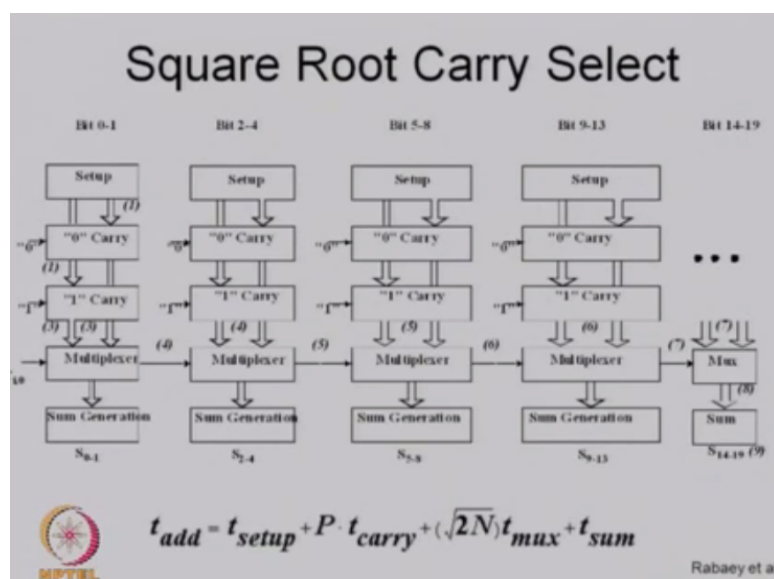
(Refer Slide Time: 36:10)



Now, if you look at the 16 bits similar adder, we now do a same procedure 4 bit, 4 bit, 4 bit blocks we create same 0, 1 availability for given any P and Gs and you have a multiplexes which receives the last carry bypass or nor bypass and if you now this is called the critical path. Because it will take one set up time, all will be setup same of one setup time, you have to take 0 and 1 calculation for this during this everyone will be doing.

So, only one time or this, okay then this is what I say N by Mt carry, then you require a multiplexer times if there are 4 bits, you require 4 bits of this okay and then you have of course a sometime which is this, so this essentially, one can see from here it has slightly reduce the m times t carry has been reduced in this case, so this will be slightly faster than the earlier ones, not greatly but at least to some extent.

(Refer Slide Time: 37:19)



Now, we figured it out that unnecessarily if you see just show you in this case, for this one to see you know, now it has to propagate all of it till it does something here, so the delay which every one of them receiving is not the same okay and in the circuit I keep saying if the delays are not equal and the system is dynamic kind of adders you use, then there is the timing issue will be very crucial for you.

(Refer Slide Time: 38:17)

$$N = M + (M+1) + \dots + (M+P-1)$$

$$= MP + P(P-1)/2 = \frac{P^2}{2} + P(M - \frac{1}{2})$$

For adders,
Here we have N-bit adder containing P-stages, first stage has M-bits, Second stage has (M+1) bits and so on.
If $M \ll N$ (generally true), then

$$N \cong \frac{P^2}{2} \text{ or } P = \sqrt{2N}$$

$$t_{add} = t_{setup} + Mt_{carry} + \sqrt{2N}t_{mux} + t_{sum}$$

While in Linear case we had

$$t_{add} = t_{setup} + Mt_{carry} + \frac{N}{M}t_{mux} + t_{sum}$$

So, we say if anyway if it is to be done like this we may not do, we may initially start with only some kind of a logarithmic number increment or square root increments, have the first block has 2 next may be 3, next may be 4, then 5, 6 and so on and so forth. Why we do this? of course this final term I recalculate. We can see from here now the number of bits are first stage is M, next is M + 1, and P is the number of stages you are; M + P - 1.

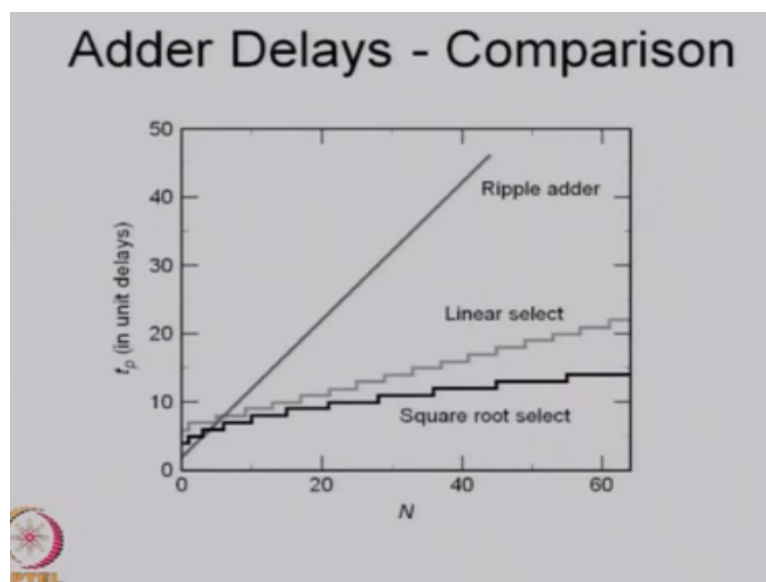
This is the series; this is called infinite series and if you actually put this is equal to $M^2 + P^2 - 1$, please remember P is the number of stages okay, N is the number of bits, P is the number of stages, please remember this is not same N/M for the reason you can see here this is 2 bit, 3 bit, 4 bit, 5 bit, 6 bits so number of stages are depending on how many Ns you have finally okay.

So it is not N/M because it is, M is increasing every stage okay. So if I see this, then I add, I see this N number is $M^2 + P^2 - 1$ which by series evaluation one knows is equal to $MP + \frac{P^2}{2} - \frac{1}{2}$, okay. If M is very small which should be M is 2, 3, 4, 5 whereas N is of course is 64, 128, or 32 or higher, so N is much larger than M in most cases particularly you should use when N is much larger than M.

If that is show, this second term can be neglected, M is very small compared to this or this term is very small compared to square terms, so N is equal to P^2 by 2 or P is now $\sqrt{2N}$, so I know now how many stages I have to go. Let us say if I have a 32-bit addition to be done, so I should have 8 stages to go through okay 2, 3, 4, 5, 6, 7, 8, in keep on increasing the bits every stage, so the 8 stages will be required.

And then if you add; find out the carry total add time setup time + Nt carry + now P stages means $\sqrt{2N}$, t_{mux} this; since it is a root, please remember earlier case it is N/M , let us take a M is 4; normal case, then you can see from here, let us say 32, so this only 8, this is $64/4$ is 16. So we are actually reduce the delay by going a square law terms which is essentially we say logarithmic system.

(Refer Slide Time: 40:51)



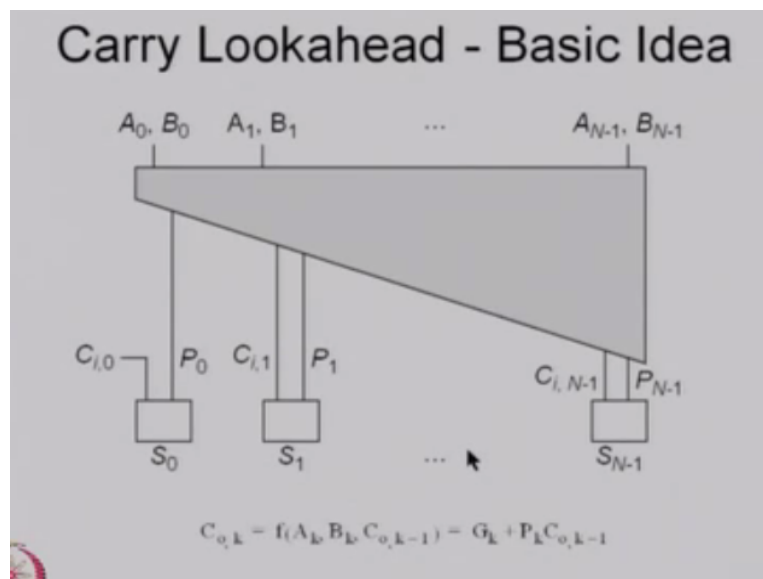
So, here is a table again taken from Rabaey's book, which says the number of bits verses the delay for the adder in units of delays, it can be nano second, pico second depends on technology. Say, if you see if you use the ripple carry adder and you keep increasing N , the delay will keep on linearly rising as proportional to N in fact okay. If you do a normal linear select, okay even with 0, 1 pre computed for a every bit of you know chain these are all equal.

You can see here there are 4 bits, 68 bits or whatever size of M you choose, so there is the stepping going on for every size, so we can see from here. There is an increase of delay but certainly much smaller than this, but if I do square the delay is much more constant, so one

can see for a larger and larger bits of operations one should prefer square root select adders. because that will reduce the delay to a great extent.

So, this is the ultimately I must like to tell you that when to choose a simple ripple carry kind of a Manchester carry kind of ripple adders, the minimum thing one has to do is to deduce this. However, if you look at very carefully if you do this square root select the number of Ps will be very large may be 16 or 8 or 30 or 32 in case of larger bits which means the area of the; this circuit will not be smaller in any case, okay.

(Refer Slide Time: 42:59)



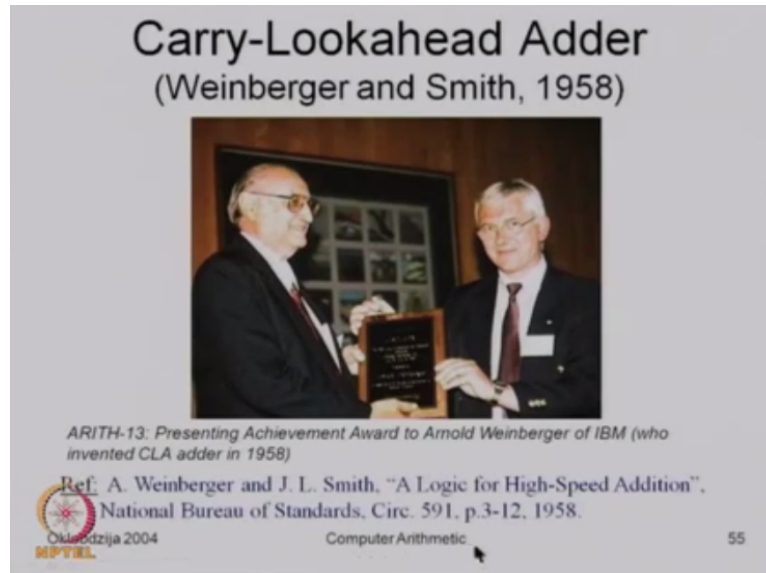
Now, one of the feature of the adder is we are keep telling again and again same thing okay, we keep telling you that major worry of adder speed is decided by the availability of carry before the next add operation is pre performed and if that is to be done what is the way we do it? Here is the simple method we say here are A0, A1, B1, AN, Bn these are; let us say N bit of data.

The first A0 B0 creates P0, this and you add Ci0 here to create sum, now this will also create Ci1 which you have to create and then the A1 B1 and then it will create S1 and this. Now, if since if you can figure it out this without coming from here, then I am saving time, okay. So, the whole trick in look ahead word is that I do not have to wait the last carry outputs before I do the next operation.

I am not trying to say that it is so linear as I am making it there are hidden problems, which I will show you now but still I can say that at least if I make it some kind of algorithm in which

Ci 1 is not the output coming from this first block to a great extent I would say, I will not say it is 100 % 1 to 1 then I have some generation much faster compared to any other block. Now this is very idealistic thinking.

(Refer Slide Time: 44:35)

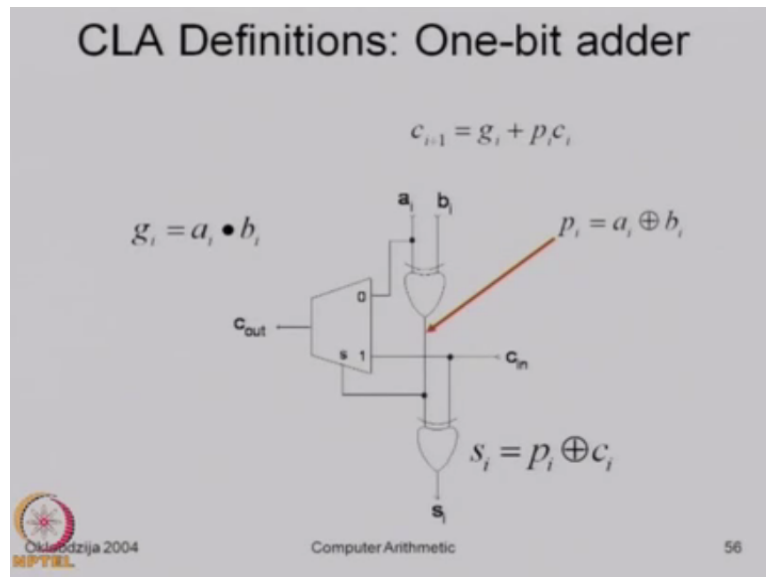


Let us say, how do I implement this my basic idea. As I you have seen my first lecture in this course was historic perspective of VLSI and from there you can understand that why I am so fuzzy okay. There is a trend among the present generation, I hope it is not true for all of you; you forgot the history and probably I am now in the history of your age I already 65, so obviously I am historical person for you.

And unless you know history you cannot progress in future even if you do not know history 200 years ago at least you should know history of last 25years even to know what moment wrong or what went right that the path is in future is better. So, here is my feelings, I just tell you that Weinberger and Smith in 1958 suggested this idea of look ahead carry and they received IBM award for that time which was more prestigious award in those days.

Achievement award for their work which they called ARITH 13 and they invented this CLA adder this was in 58, okay just want to therefore give you the reference Weinberger and Smith logic for high speed edition in National Bureau of Standards 91, 1958, okay this has been taken from Oklabdz 2004 ppt slide, he has the lecture some computer arithmetic either the computer scientist, okay.

(Refer Slide Time: 44:35)

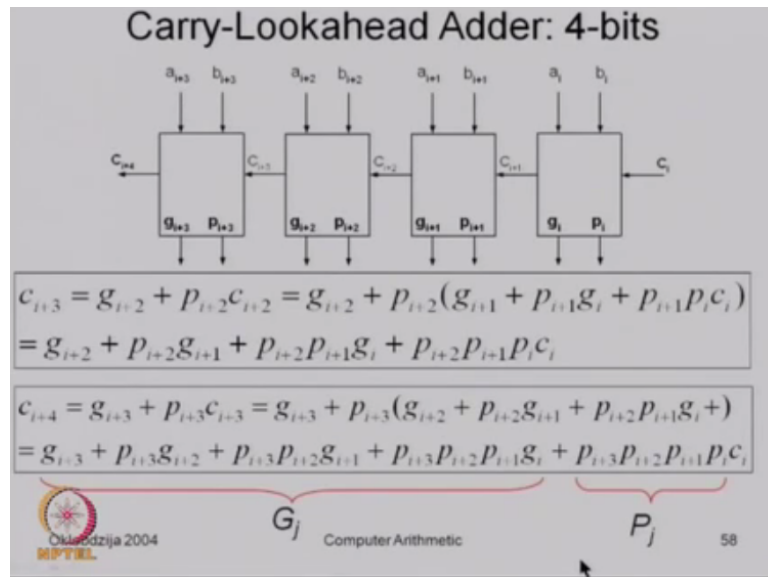


Of course, by the way this Weinberger was a mathematician okay so what is that CLA is trying to do is the following okay. What is he trying to do is? We have all maths available to us okay, $c_i + 1$ is $g_i + p_i c_i$, p_i is $a_i \text{ XOR } b_i$, g_i is $a_i \text{ dot } b_i$, okay and s_i is $p_i \text{ XOR } c_i$ okay. So now implement this whole function in this form. You create inputs a_i and b_i XOR of this will be obviously p okay.

And you know the c_i , okay, now if you do XOR of this with c_i , then you get the sum, s_i is $a \text{ XOR } b \text{ XOR } c_i$ is the sum and to generate a carry you need a g_i , which is $a_i b_i$ one of them, so you should of course that adder is not sure, $a_i b_i$ this dot essentially in meaning that I had taken care of addition here, n product here. This is my g_i and you can see from here if c_i is 0, okay please remember this is the $a_i b_i \text{ XOR}$ that is p_i .

Okay, p_i is one, then g_i 0 I know, okay. If g_i is 0 and p_i is 1 and $c_i + 1$ is c_i , so obviously, whatever is C input here will pass here. Because I am choosing this has 1, so 1 the lower one will pass mux. In case c in it 0 okay in case p_i is 0 which is from $a_i b_i$, g_i can be 1, and if that is one, then C in what we it anything it does not matter because g_i is one means, one + anything is one, so this is directly transfer to C out.

(Refer Slide Time: 48:25)



So, C out g_i when p_i is 0, so that thinking is so simple that since I look at the expressions, I now see there are methods I can utilise it. because I do not have to evaluate carry every now and then if I look at this kind of expressions. Now, look at the way it is done okay. Here is the definitions we gave you have a 4 bit adder, these are the input carry C_i C_{i+1} , C_{i+2} , C_{i+3} this is a_1b , $a_i +$, $a_i + 2$, or call it as a_0 , b_0 , a_1 , b_1 , a_2 , b_2 , a_3 , b_3 .

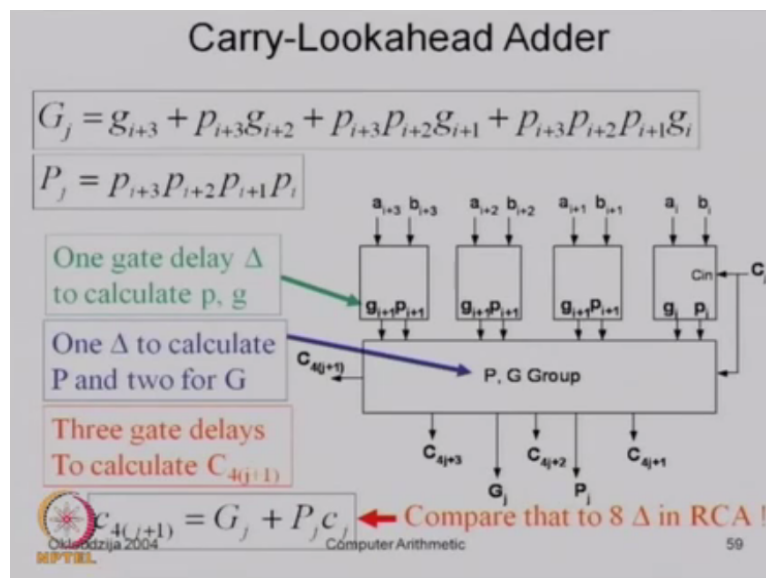
This is your standard addition what we do so that is nothing big about it but if you look at the C_{i+1} the carry which is generated here out of this add or circuit, instead of writing XOR functions you can write it as $a_i \text{ bar } b_i c_i + a_i b_i \text{ bar}$ like a logical expressions and then you say this is also we know c_{i+1} is nothing but $g_i + p_i c_i$, essential substitute in g_i as $a_i \text{ dot } b_i$ okay.

And substituting p_i by a XOR b_i which is essentially $a_i \text{ bar } b_i + a_i b_i \text{ bar}$ which is the XOR function, so I am representing those functions by their equivalent mean terms, not exactly full mean terms partial mean terms. So, if you get this expression, you can see now look for the other one; c_{i+2} is $g_{i+1} + p_{i+1} c_{i+1}$. Now, I know c_{i+1} okay. So I say okay it is $g_{i+1} + p_{i+1}$ + on bracket of $g_i + p_i c_i$.

So, I expand that, so I say it is $g_{i+1} + p_{i+1} g_i + p_{i+1} p_i c_i$, okay just make product from that. You calculate c_{i+3} now it is $g_{i+2} + p_{i+2} c_{i+2}$ now I know c_{i+2} so I am representing term here, expand it here, I do same thing for four also, so if you see the final one for example it is $g_{i+3} + p_{i+3} g_{i+2} + p_{i+3} p_{i+2} g_{i+1} + p_{i+3} p_{i+2} p_{i+1} g_i + p_{i+3} p_{i+2} p_{i+1} p_i c_i$, so the first 3 terms for the fourth carry has G terms, g_{i+2} , g_{i+1} , g_i .

It has p_{i+1} , p_{i+2} , p_{i+3} , 3 propagation terms, is that clear? You have g_{i1} , g_{i2} , g_{i3} and you have p_{i1} , p_{i2} , p_{i3} , so 3 P terms for the last stage okay. However, the last term we say is p_{i+3} , p_{i+2} , p_{i+1} , p_i into c_i , this does not have any G term no generate term in this. So, we say okay this we call as block G, for all of the block of this, this is the capital G_j , in which all G s appear okay. This does not contain any G only P terms, so we call it P_j okay.

(Refer Slide Time: 51:40)

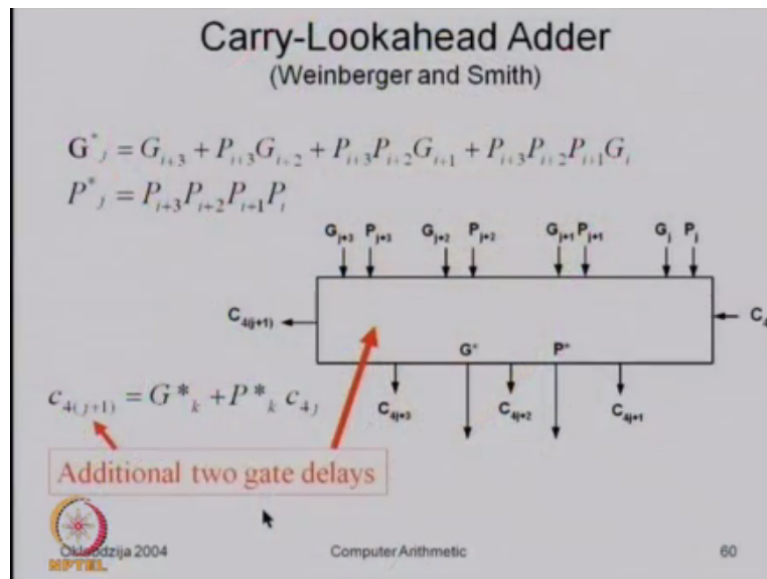


This is the definition what Mr. Weinberger has given them okay. Now, if I do this, then I write G_j is this expression P_j is this expression, is that clear? So, now one can see from here, to evaluate P_j you do not need C ; to evaluate G_j you do not need C okay, so your whole worry of propagating carry ahead I do not need to know carry anyway, I can evaluate my P_j s and G_j s irrespective of what carry I am going to get okay.

And because of that if you see here is the input carry and this is your this, this PG group which is here will immediately generate to me G_j s and P_j s which will give me an output carry anyway. So, without actually going through this typically if you calculate this, if you add 8 stages of carry has the last one, there say each gives the delta delay then it compared to that to 8 delta in RCA.

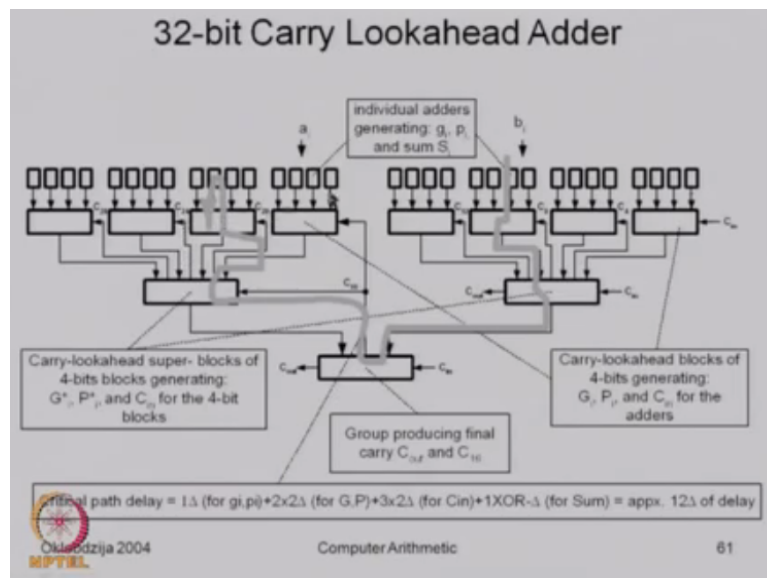
This is essentially has only 5 delta as will give the calculation numbers okay. One get delay for any one of this operation, one get delay for this operation p and 2 for g, okay and 3 get delays to calculate the final ones. So you have 3 + 2 + 1, 6 get delays okay. So essentially, what I am trying to say if you have done a normal ripple carry adder and if you do look ahead I have anyway saved the time in a great extent.

(Refer Slide Time: 54:06)



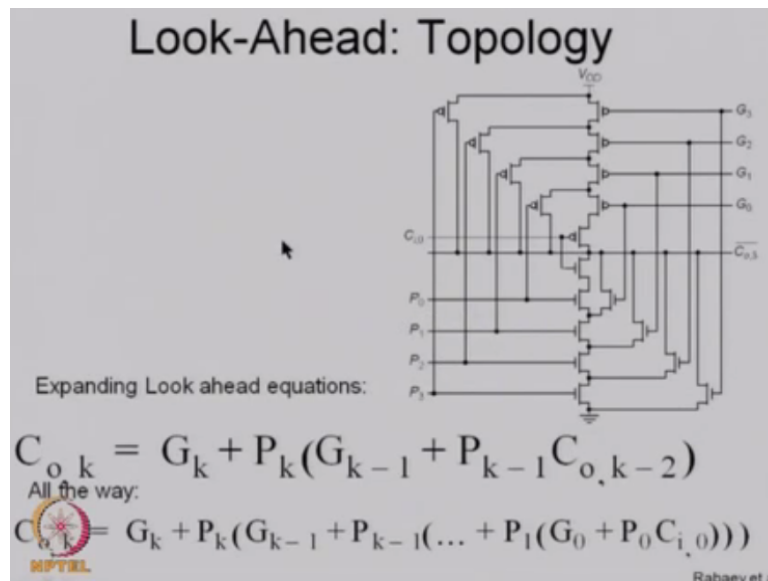
Also look at the features, there are no additional adder blocks like carry save stages we did, no 16 stages or 34, we are keeping normal addition, we are not changing from our normal carry adders which we had, but just modify the way we evaluate things and we figured it out that because of this we can always improve the speed of this. So, Weinberg method, which I shown here one can see from here, okay the same I think is repeated.

(Refer Slide Time: 54:09)



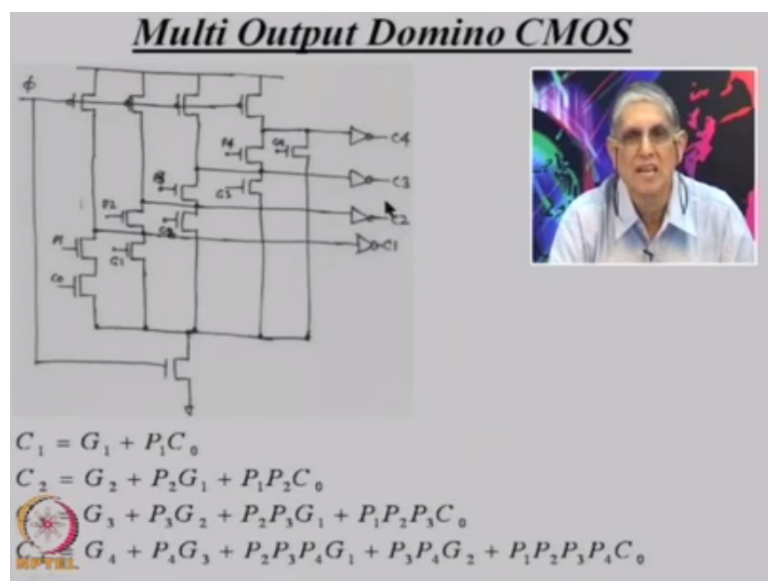
Using Weinbergs technique or look ahead method, the 32-bit carry look ahead will look something like this and I also written the critical path is around one delta for GP, 4 delta for GP into it and keep on, because you know each they block, each one block I have shown you, repeat so many blocks and you will get the net delay coming out. Please remember there is compared to any look ahead, compared to any ripple carry adder, this will be the fastest.

(Refer Slide Time: 54:41)



Just to give an idea, you look at this static representation.

(Refer Slide Time: 54:49)



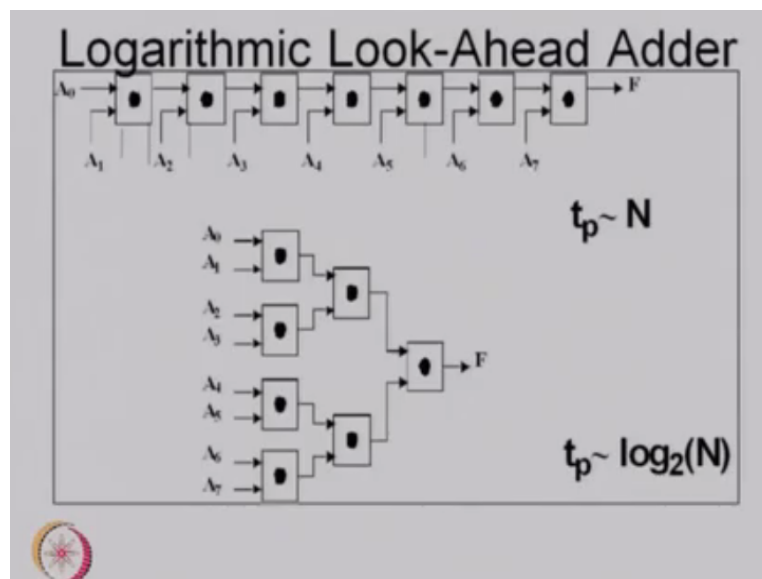
Maybe, it is better if I show you domino ones for the simplicity, you can see from here C_1 is $G_1 + P_1C_0$ only 4 bits I show you, C_2 is $G_2 + P_2G_1 + P_1P_2C_0$; C_0 is the input carry and C_1, C_2 is the output carry, $G_3 + P_3G_2$, these are the terms, these are AND terms and OR terms. In a logic, transistor in series creates ANDS parallel creates ORs, okay.

And since it is an inverting logic, it will create bars of that so you require finally, you will get \bar{C} bars instead of C s. However, you can do start looking at it. The first stage you have P_1C_0 as the input, this one and all to that is G_1 , this block P_1C_0 OR G_1 , this output of this, this is

C1, okay, this output of C1 is transmitted okay, now this which function you have generated here, this into P1 + parallel combination, now these terms I am calculating.

So, I generate C2. I use the last Cs which contains all these terms, use that term as one of the input and generate C3 use this term to create C4, so now you can see from here $4+3$, $7+3$, $10+14$ transistors are required with much faster speed than any ripple carry, okay, this is domino in the sense why it is advantages, you can reduce power because that no time P and N channels at least the static power or half power will be smaller.

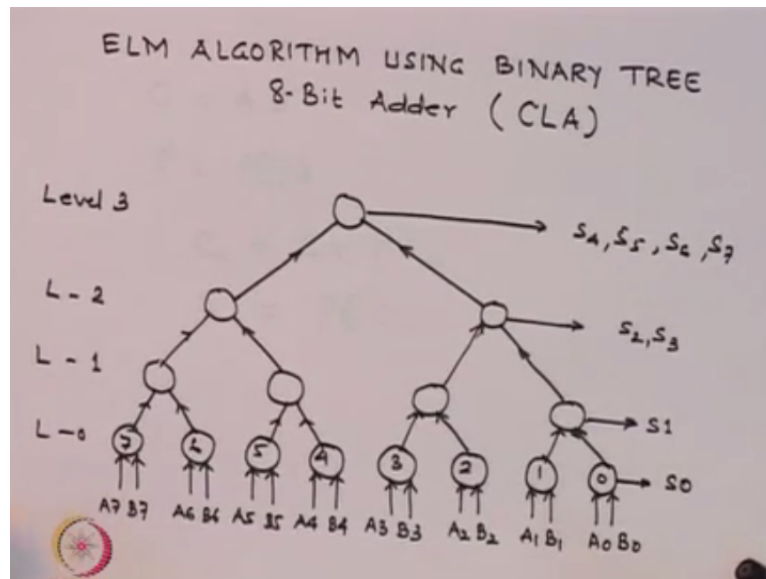
(Refer Slide Time: 56:50)



So, one can see it multi output dominos CMOS can implement, a CLA kind of structure in much low power and relatively high speeds. Now, we looked into little differently the look ahead. We said okay can; this is the standard carry what we do proportion in any way, we do block wise, we say okay we do only for $A_0 A_1$ create something this, $A_2 A_3$ 4 blocks they will generate 2, each of them, then use those 2 to generate this.

So, this is called trees and if you see this 2 into 4 into 2 into one this is like a logarithmic system. o if you look at now, you do partial evaluations and keep calculating ahead because after all carry is not required in anyway, so you can keep doing in partial sums and add next to that if you do instead of series connection like this, this will be proportion to law, these tree structures which is what is very popular is also uses the algorithm, logarithmic algorithm which is called ELM, logarithmic models.

(Refer Slide Time: 58:03)



The biggest advantages of ELM, which I can show you here in my case, there are 4 levels for 8 bit adder, A0 B0 you do an evaluation here and pass some output immediately create carry, go create next P and Gs, take A1 B1 and now work with them, create S1 and go ahead keep doing like this and you create S0 S1, S2S3, S3 S4 S5 S7 in 1, 2, 3, 4 stages, okay instead of; it is a logarithmic system in which one can reduce the speed simply.

(Refer Slide Time: 59:01)

Comparison Between Various Adders of 32 Bits

SPECs:
 $V_{DD} = 3.3\text{ V}$, Clock Frequency = 100 MHz

Type	Area ($10^6 \mu^2$)	No. of Transistors	Delay ns.	Av. P. Dissipation (mW)
CLA	2.27	2132	15	114.6
ELM	2.36	2078	10	104.1
RCA	0.80	1204	55	89.2

Clearly ELM Binary Tree Adder is
BETTER ON PERFORMANCE & Power Dissipation
K. Roy et al

Because you are doing parallel operations okay, this is called tree structure additions and this is very popular because all tree structure will reduce the average time and they will be useful much more in something called multipliers little later when we discuss. Now, if I do for all adders shown so far, the most common Manchester carry or ripple carry adders then the modify tree adders, see ELM and we have SCLA, which is standard Weinberger system.

There is a comparison taken from Roy's papers, this is a 32 bit addition, for the simplicity they are used very high voltages just to separate things, actually they were working on one micron, point 2.5 micron process. So 3.3 volt, 100 mega hertz is the frequency they used and lambda as I said is 0.25, so this is area, number of transistors, delay nano seconds and power dissipation, average power dissipation.

So, you can see from here, for CLA, for the same 32 bit addition, you have 2.27×10^6 to the power 6 lambda square is the area, transistor required as 2132, the average delay is 15 nano seconds and power dissipation in milliwatts is 115 or 144.6 milliwatts. If you do a typical RCA then the area which is the smallest in fact 0.80×10^6 to the power 6 lambda square requires 1204 least number of transistors but the delay is 55 nano seconds, power is also relatively smaller.

If you look at ELM, between these 2, that is why I put in between, it has slightly larger area than CLA, because you are doing square terms, okay 3 terms, it requires even if area little larger, the number of transistor is slightly lower than CLA, it gives delay of 10 nano seconds which better than CLA, and it has the power dissipation of 104 milliwatts. Now you can see from here, RC has 87 milliwatts, CLA has 114, EMS 104.

This is 55 nano second delay in RCA, 15 in CLA this is 10, transistor of course area wise this is much higher than but it is low as much as CLA so in most binary tree adder requirements as in the case of multipliers, Wallace tree multipliers. If you are going to use an CLA as your adder then most likely ELM version of that adder will be used. This is very relevant because people should know, when to use what?

(Refer Slide Time: 01:01:55)

Ripple-block carry look-ahead adder (RCLA)

- **Ripple-block carry look-ahead adder (RCLA)**
- The idea of the ripple-block carry look-ahead addition is to lessen the fan-in and fan-out difficulties inherent in carry look-ahead adders. A ripple-block carry look-ahead adder (RCLA) consists of N m -bit blocks arranged in such a way that carries within blocks are generated by carry look-ahead but carries between blocks are rippled.
- The block size m is fixed to 4 in the generator. The RCLA design is obtained by using multiple levels of carry look-ahead. If there are five or more blocks in a RCLA, 4 blocks are grouped into a single superblock, with the second level of look-ahead applied to the superblocks.
- Figure in next slide shows the parallel prefix graph of a 64-bit RCLA, where the symbol (solid circle) indicates an extension of the fundamental carry operator described at [Parallel prefix adders](#).

I just want to give you numbers so that you know this okay. The same thing I discussed here just to give an ideal ripple block look ahead adder, this is what essentially ELM means, the ideal of ripple block carry look ahead addition is lesser than fan in and fan out difficulty inherent in carry look ahead. The block size m is fixed to 4 in generators, RCLA designed is obtained by using multiple levels of carry look ahead.

(Refer Slide Time: 01:02:34)

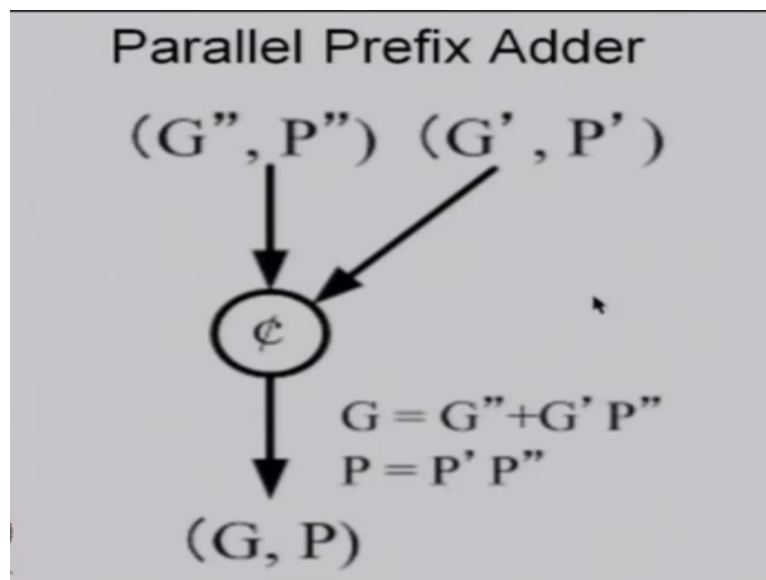
Parallel Prefix Adder

- Parallel prefix adders are constructed out of fundamental carry operators denoted by ϕ as follows:
 $(G'', P'') \phi (G', P') = (G'' + G' \cdot P'', P' \cdot P'')$,
where P'' and P' indicate the propagations, G'' and G' indicate the generations. The fundamental carry operator is represented as shown in next figure.

If there are 5 or more blocks in RCLA, 4 blocks are grouped into a single super block this is what logarithmic systems are known, combined, combined okay. Now, if you look at this, there is an equivalent method was suggested for CLA and that is called parallel for its adders, okay you see into this later. Now, what is parallel prefix adder is going to say, it use a some kind of a little algebra which is different from normal.

Parallel prefix adders are constructed out of fundamental carry operators denoted by C bars, C slash as we call as C carry, whatever I do not know what I should say, see slash I would say. Now, the way definition is that Gs and Ps are the carries and propagations indicated generations and propagations then G dash P dash contained in G dash P dash equal to and that the expression they derive this is G double dash + P dash G dash this.

(Refer Slide Time: 01:03:21)



Now, the fundamental carry operator is represented as shown next figure, this will be more understood to you. If you give this kind of an operator C Bar or C slash, you give G double dash P double dash, G double this, it will create an output which is of G and P where G is G double dash + G dash P double dash. Please look at what we are trying, we are looked into the expressions of carry look ahead by expansion; Weinberg expansion.

We were looking into terms which will get $G + GP, G_{i+1}, G_i P_i +$ and you see that the G terms contains $G_i P_i$ products and P terms contains only P terms there okay, so block carry and block generate, so we are just trying to utilise those terms, we say okay where G is G double dash G dash P_i and P is the same what we said in the case of block, this is same thing I am talking again but this is what it generates.

(Refer Slide Time: 01:04:27)


DOT Operator (Associative Law valid)

Using dot operator

$$(G,P) \cdot (G',P') = (G + PG', PP')$$

Example: $(G_{3,2}, P_{3,2}) = (G_3, P_3) \cdot (G_2, P_2)$
 OR $(C_{0,3}, 0) = [(G_3, P_3) \cdot (G_2, P_2) \cdot (G_1, P_1) \cdot (G_0, P_0)] \cdot (C_{1,0}, 0)$

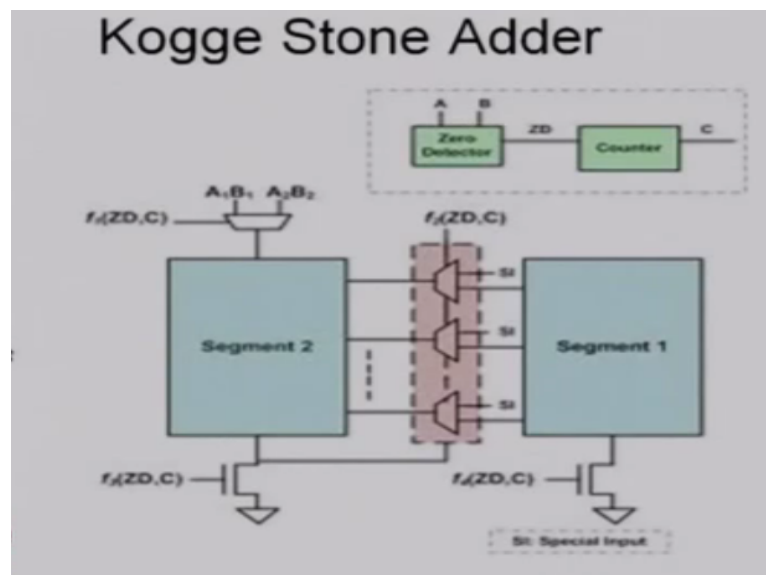
$$(G_i, P_i) = \begin{cases} (g_0, p_0), & \text{if } i = 0 \\ (g_i, p_i) \circ (G_{i-1}, P_{i-1}) & \text{if } 1 \leq i \leq n-1 \end{cases}$$



$C_i = G_i \text{ for all } i$

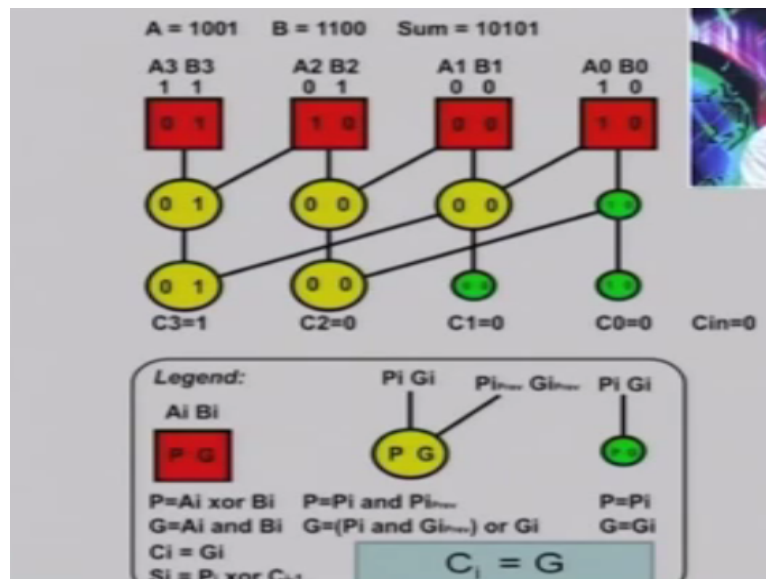
This kind of operations can be directly generated; this is what the parallel prefix adders are. There is something called as associative law, which dot operators allowed and one can see $G_{3,2}, P_{3,2}$ can be written as G_3 into P_3 dot G_2 into P_2 this is associative law or $C_{0,3}$ zero can be written as $G_3 P_3 G_2 P_2 G_1 P_1 G_0$ into $C_i 1$ zero. Now, this interesting mathematics can be implemented something like this, the block G and P which we have derive is equal to $G_0 P_0$, if i is 0.

(Refer Slide Time: 01:05:26)



If not, $G_i P_i$ into G_i - this if $i < n-1$ or > 1 okay. Now, if you look at very carefully using this i is, C_i is always G_i , okay. C_i will be always the part of the G_i , okay, now this is very interesting okay. Now you can see from here this is the first adder which uses this prefix adder technique is called Kogge Stone. I am not going to detail of this, first let me show you what it does?

(Refer Slide Time: 01:05:36)



There are 3 blocks shown to you here okay, in Kogge Stone adder. Let us say I want to add A1001 with B 1100 an I know normal adder I am expect addition and expecting is 1 0 in 1 is one, 0 and 0 is 0, 1 and this is 1, and 1 and 1 is 1 + carry 1, so 1 and 1 is 0 but carry 1, so I expect a sum as 1 0 0, please remember this is the last carry is the first bit, okay which I know in anyway in additions.

So now, I figure out there are 3 blocks are generate. The first blocks are normal addition blocks you can see or PG generating blocks, the one prefix adders, part of the normal adders I said, P and G. So what is the reason for this? This P and G block receives Ai Bi as the input and it says the P is Ai XOR Bi, G is Ai and Bi and then Ci is Gi and Si is Pi XOR Ci, okay, Ci - 1, last carry.

The yellow blocks which are the prefix adders block C block as we say it receives previous P, previous G okay and it receives the current P and current G, this is what say P double dash, G double dash, P dash, G dash, just now I said and this is that C slash adder, C slash operator. So, what it will generate? it now says P is equal to Pi and Pi inverts and of this and G is equal to Pi and G previous RGi, those expression which I wrote can be expressing this now okay.

And this is called transmission blocks in which it receives Pi Gi and transmits as they are oaky and now if you see here the way it is done, I have fed this A0 B0, A1 B1, 010000111 to these normal adder blocks okay so now this will create; let us say this will create P and G

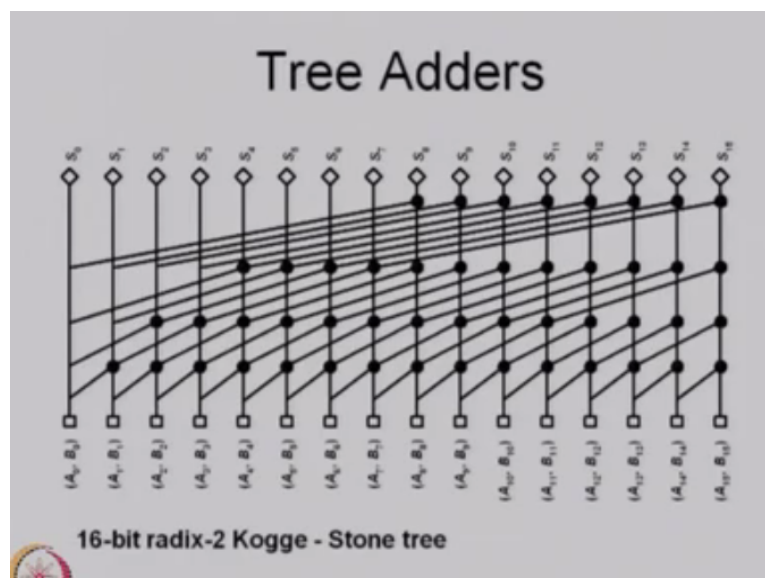
which is 1, 0 okay you can see here why 1 and 0, the expression I already written this it evaluates creates 1 and 0 and in the next they say just transmits.

Now, we also know the C_i is equal to G , so if G_0 output is 0, so carry is 0, okay. Now for the next block yellow block we need the last P to get the P , so I have already generated P and G for the last okay. So, I use them and use this current P and G using this expression, I evaluate both P and G here, which I get 0, 0 okay. Please remember take it an example, this is P_i and P_i inverse 0 into 1 is 0 and G is P_i and P_i and G inverse.

So, it is 0 into 1 is still 0, so 0, 0 okay. Now this I propagate down okay. So, I get next C as my G_i which is 0 because this is 0, so output is 0, carry 0. For the third block, I need this is 1,0, I need the last coming from here $P_i G_i$ inverse, so I get 0, 0 transmit 0, however for this when I do this, I take these calculations; for this calculation I need; please remember $P_i + 2$, last 2 last one because unless that is known I cannot generate the C_2 .

Because you know the third term is required $P_1 P_2$ third term, so the last 2 terms I needed so I create 2 term from here and reach here, i need 3 terms. So one from here one from one from here, so I get 3 terms to evaluate when I get the output carry C , some I can directly get from the expressions which P_i XOR which is not shown here, some can be directly parallel out here itself okay.

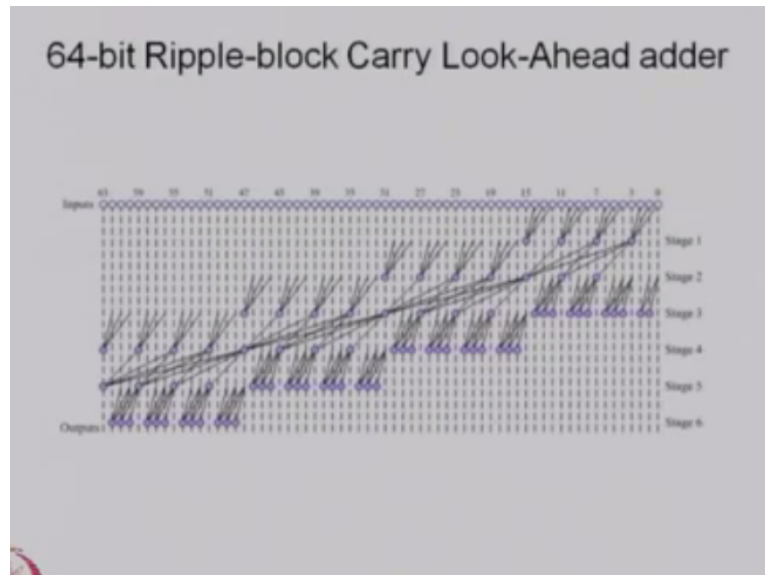
(Refer Slide Time: 01:10:25)



So, now every time I do an operation I create some and I generate C and then I can get this, so what is the advantage of this Kogge Stone method? You can see from here, you can feed

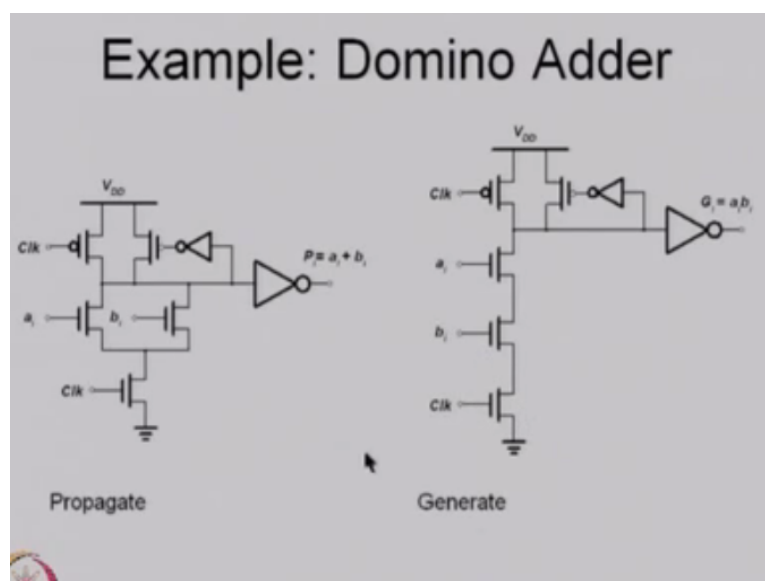
all the data 16 bit, 32 bit any number of bits data and you can create partial sums, sums 0 this and transfer P and G for the next one P and G to the next one, P and G to next one transmit, keep down going down and keep create new sums and new carries okay.

(Refer Slide Time: 01:11:19)



Or rather new Ps and new Gs and Gs are finally seen so we always know what we are actually doing, so using this kind of structure of the net delay essentially because there is no carry propagations since there is no carry propagations but dot products or dot operators allow you to create parallel sums and parallel carries simultaneously. A typical ripple block carry 64 bit if you I do not want to discuss it same as this.

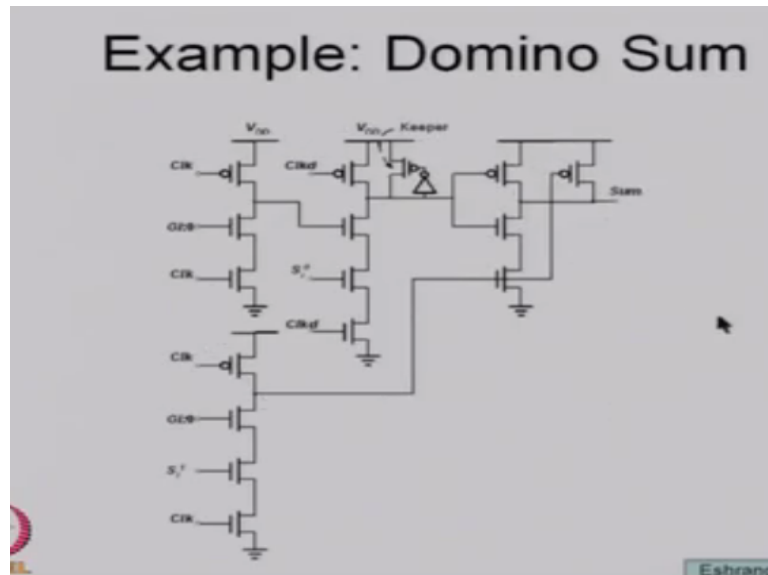
(Refer Slide Time: 01:11:48)



You can now require 6 stages due to power 6 stages you can see this figure you start 64 bits like this keep doing pass on pass on pass on pass on, so the larger version of 64 bit carry

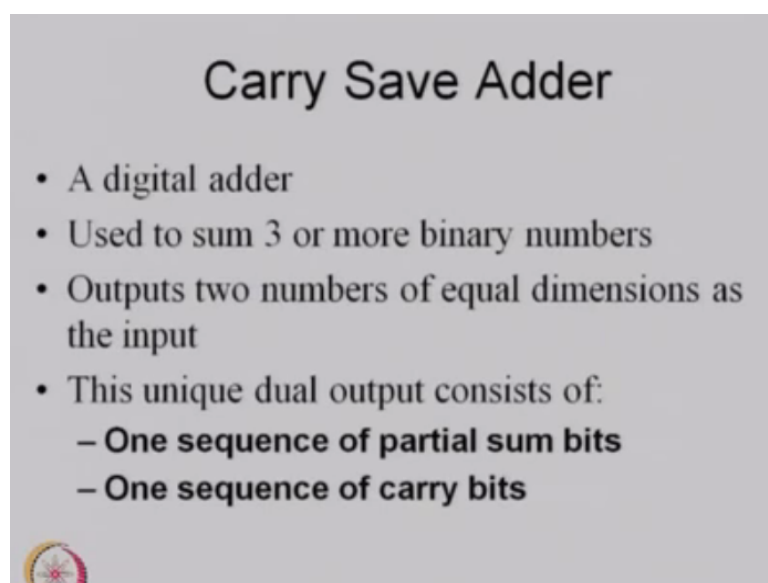
block carry look ahead look something like this using prefix adders. Another advantage of using dominos circuits is shown here for generating P and P this is a propagate signal and this generate signal you can see why I actually shown you here is the circuit immediately.

(Refer Slide Time: 01:12:13)



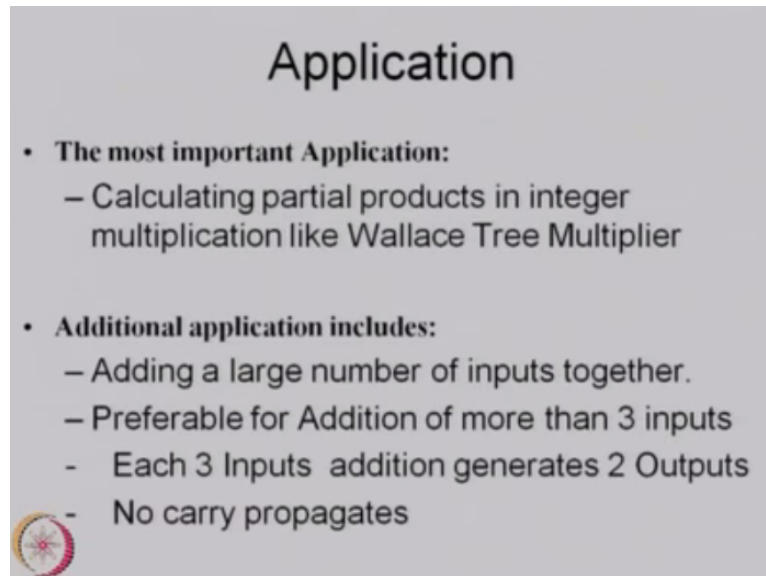
Because I thought that it is not the static way we will do it in most cases, we will like to reduce the transistor themselves and therefore power and we will always require P and G generation so you can use P and G blocks in the domino form as shown here. You can create domino sums out of this number of transistors as we show you. So normally the blocks which I show you may be using domino circuits rather than the normal statics CMOS or simple dynamic CMOS.

(Refer Slide Time: 01:12:48)



Now, I come back to you in different kind of adder which is very popular in the case of the multiplier before we quit this area is called carry save adder which is very very relevant for multipliers not so much may be for adders though it is important. A carry save adder is the digital adder, it used to some 3 or more binary numbers, output 2 numbers of equal dimensions as the inputs this unique dual output consist of one sequence or partial sum and one sequence of carry bits.

(Refer Slide Time: 01:13:13)



Application

- **The most important Application:**
 - Calculating partial products in integer multiplication like Wallace Tree Multiplier
- **Additional application includes:**
 - Adding a large number of inputs together.
 - Preferable for Addition of more than 3 inputs
 - Each 3 Inputs addition generates 2 Outputs
 - No carry propagates

I will actually, this is the statement I make, I will examples to show you what I am talking. The most important application is calculating partial products in integer multiplication like Wallace tree multiplier, just now I said because please remember in multiplier you need partial products okay and then you will sum them. So, in those generation, I am remember partial products are nothing but and GATES.

(Refer Slide Time: 01:13:57)

Advantages and Disadvantages

- **Advantages:**
 - Produces all of its outputs in parallel resulting in the same delay as a full adder
 - Very little propagation delay when implemented
 - Carry Save Adder plus a ripple adder = $n+1$
 - Two Ripple Carry adders = $2n$
 - Allows for high clock speeds



And therefore this kind of tree structures helps to generate very fast radiations. Additional application include adding a large number of inputs to you, you can have more than 3 input, 4 inputs simultaneously preferable for addition of more than 3 inputs, addition generates 2 outputs, no carry propagates which is the biggest thing it can do okay. It has advantages, it produces all of its outputs in parallel resulting in the same delay as the full adder okay.

Very little propagation delay when implemented, carry save adder + a ripple adder additional adder will be required to make $n + 1$, total if n carry save adder for n bit, then you need additional final ripple adder to make total sum, 2 ripple carry adders will require $2n$ and terms allow; please remember that is the number if I have 2 such thing is require $2n$, otherwise I require only $n+1$ and that is the trick of the trait.

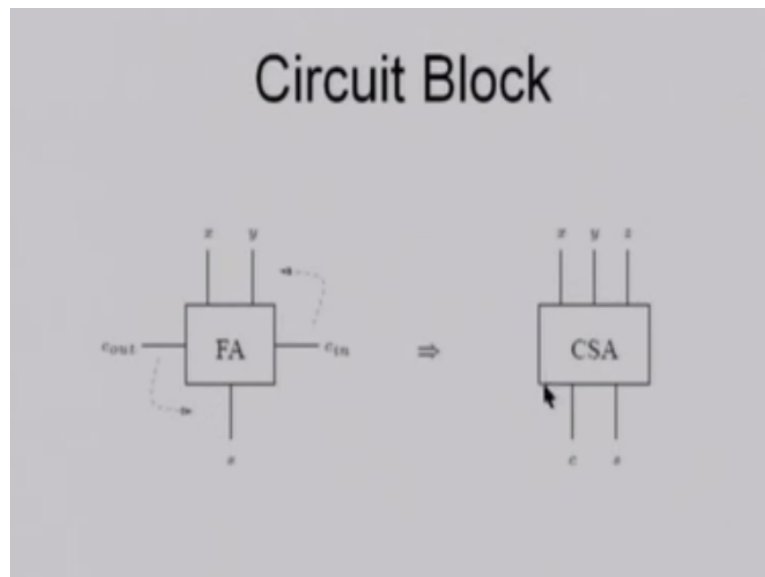
(Refer Slide Time: 01:14:40)

Advantages and Disadvantages

- **Disadvantages**
 - We do not know whether the result is positive or negative
 - This is a drawback when performing modular multiplication since you don't know whether the intermediate result is greater or less than the modulus

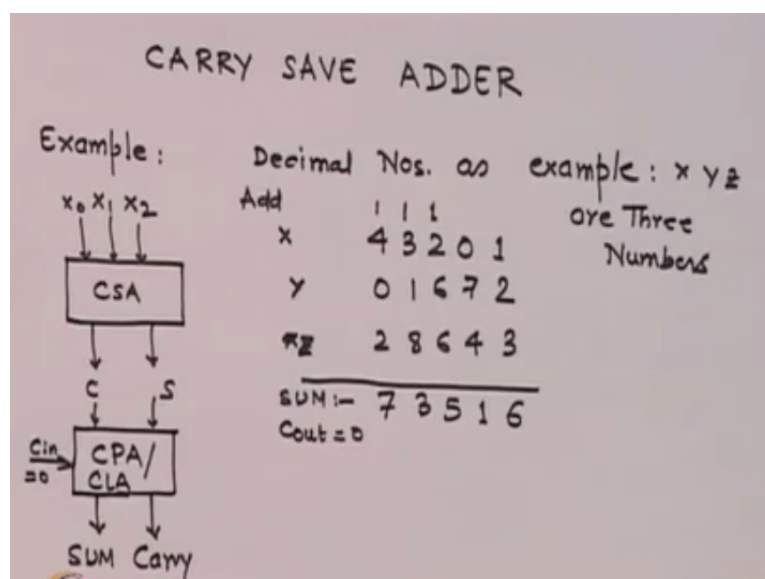
And it allows you the very high speeds. Of course disadvantages, we do not know whether the result is positive or negative, no signs, no sign digits. This is the drawback when performing modular multiplication since you do not know whether the intermediate result is greater or less than modulus okay.

(Refer Slide Time: 01:15:01)



Now, I will give you an example before, this is a typically I may show you a typical full adder in a way can be converted to a CSA, okay. A CSA has 3 inputs 2 outputs, x , y , z are inputs and c and s are the outputs. You can see here if this C in is introduced here, then it becomes 3 input and if this C out is taken as it from here it generates like this so it is not identical the way I said but it is verse C that the full adder in a form is easily convertible to CSA. Here is my simple examples, which will prove the method okay.

(Refer Slide Time: 01:15:40)



Now, first thing I show you because since this is not very popular adder as for the adder is concerned let me do addition with the decimal numbers to prove the principle and then show you one of the binary carry save adder. A typical carry save adder addition process will be you have a block called CSA which receives an input x_0 , x_1 and x_2 generates 2 output called c and c okay.

These 2, along with the actual input carry, which is normal CPA or CLA can generate your sum and carry. So this is the final addition block and this generates, this reduces your initial inputs to a lower number of outputs. Here is an example which will verify what this CSA is doing? Let us take, I have a 3 bit numbers, 3 numbers x , y , z and in decimal they are in 10s or 1000s, okay.

For example, x is 43201, y is 1672, and z is 28643 and I want to add. So, what do I add in normal decimal numbers, I will say okay, $3 + 2 + 1$ is 6, $7+4$ is 11, so 1, I put 1 carry here then I say $6 + 6$, $12 + 2$, $14 + 1$, 15; carry one, $8 + 1 + 3$, $12 +$; 13, 1; $5 + 2$, 7. So, I got and no carry because last digit so I will say the sum of these 3 numbers is 73516, this is what decimal does okay. Now how CLA will do the same operation, CSA, here is what CSA does?

(Refer Slide Time: 01:17:44)

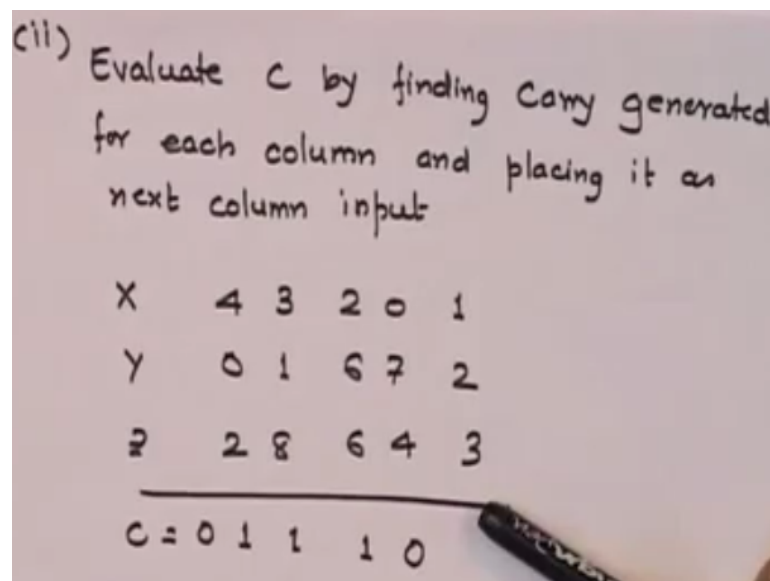
Two Steps :

(i) Evaluate $s = x + y + z$ without taking carry generated

x	4 3 2 0 1
y	0 1 6 7 2
z	2 8 6 4 3
	$s = 6 2 4 1 6$

It does this operation in 2 steps, it evaluates term S by just adding them x , y , z with no idea what carry it is generating, is that okay? For example, $1+2 +3$ is 6 but $7+4$ is 11, so I damn care what is carry? I just leave 1, I forgot 1, $6 + 6$, $12 + 2$, 14; I put 4 4 leave 1, $8+1$, $9+3 + 2$, leave 1, and $4 + 2$ is 6. So the first addition I performed without bothering the carry okay the first thing I did.

(Refer Slide Time: 01:18:46)



So, the S term which I got is 62460, please remember the first step is to evaluate some of the actual numbers bit by bit without taking care of any carry, it is only vertical line whatever it comes the LSB is retained. Then we calculate the second term which is C, I already said if I have 3 inputs x, y, z, I will create C and S 2 outputs, so S; I have calculated, now I have calculate C.

Now, instead of looking at what has been available to us as the sum part, I only look for carries, the only thing I do is; since the carry generating the first column is transferred to the next column, so whenever the carry is available to me I actually write the output carry below that column. So, for example if I do this addition and I do not see carry so put it 0, if I do this I generate 1 carry I put it here, I generate 1 carry I put it here.


(Refer Slide Time: 01:20:04)

(ii) Add S and C (shifted by left position) to get SUM & Cout

S	6	2	4	1	←
C	0	1	1	1	0
SUM	0	7	3	5	1 ←

Cout = 0

This number is Same as what Normal Addition Gets.



I generate 1 carry I put it here, I do not generate any carry if I put it 0, so I have actually shifted my carry as 0, 1, 1 as I did like this . So 2 steps, first I calculated S without worrying C, next I calculate C without worrying S okay 2 steps. Then, I actually add CNS so I say, 61537; 73516, the same sum no carry okay, this number is same so now I have an idea that well if I have a larger number of bits I can keep doing this operation.

(Refer Slide Time: 01:20:42)


Binary No Example : Three Nos. X, Y, Z

X =	1	0	1	1	0	1	45
Y =	1	1	0	0	0	1	49
Z =	0	1	1	1	0	0	28
SUM	1	1	1	0	1	0	122

Cout →

(i) Step I: Evaluate S (without use of Carry)

X	1	0	1	1	0	1
Y	1	1	0	0	0	1
Z	0	1	1	1	0	0
S =	0	0	0	0	0	0



And finally I will then; I may put the initial carry and find out what has happen, okay. So this is the trick which carry save does and here is example for binary, same example. I have a 3 numbers 101101, 110001, and z is 011100, which represent 45, 49, 28 number 122 is required. If I would have done a normal binary addition I would have got 111010, which is essentially equal to 122, okay.

(Refer Slide Time: 01:21:26)

cii) step II: Evaluate C

X	1	0	1	1	0	1
Y	1	1	0	0	0	1
Z	0	1	1	1	0	0
<hr/>						
C	1	1	1	1	0	1

ciii) Add S and Left shifted C as above

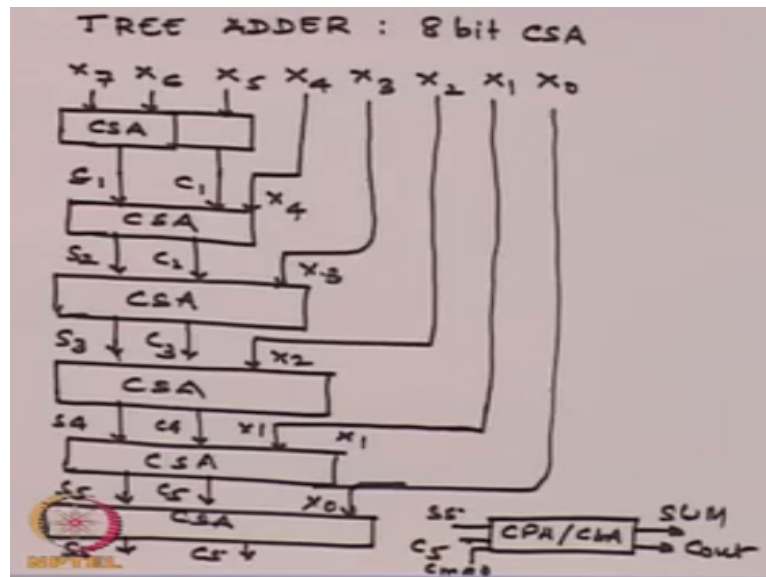
S	0	0	0	0	0	0
C	1	1	1	1	0	1
<hr/>						
SUM	1	1	1	1	0	1

→ 122 in Decimal

Now, and this final one is coming at the output carry, now what do we in the case of CSA, carry save adders, we first calculate S, 1 and 1; 0, 0, 0, 1 and 1; 0, 1 and 1; 0, 1 and 1; 0, 1 and 1; 0. So S is all 0s, calculate C please remember in calculating C, we do not worry about what S was, 1 and 1 will generate carry, 0 and 0 will not, 1 and 1 will generate carry, 1 and 1 will generate carry, 1 and 1 will generate carry, 1 and 1 will generate carry, 1 and 1 will generate carry.

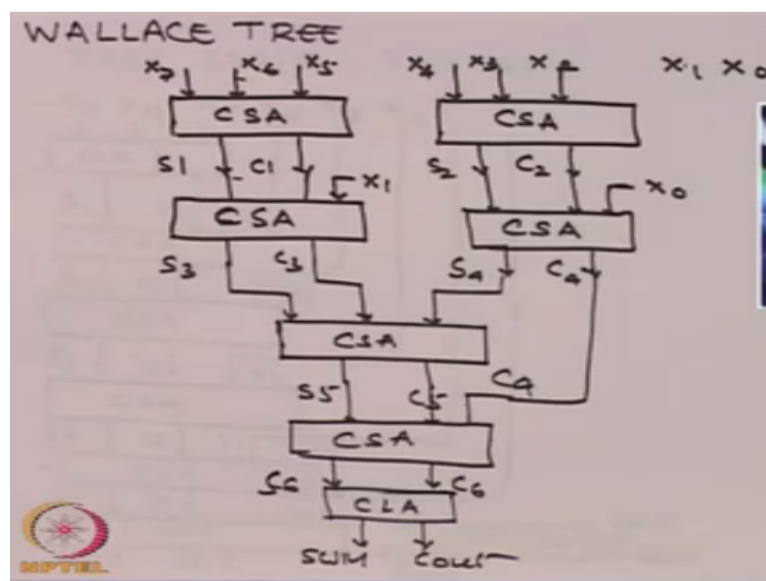
And the last one which has been generated that is kept here okay. Please remember 1 and 1 but 1 will transfer, so the last one okay. Now this is C, as my next third step was to add S and C so 0000 and please remember I have already shifted 1 so keep that as it is, at this you get 11110110, which is 122 in decimal. So, the operation which I did please remember I have done 2 operations generating C and S and final addition to get my final sum and the output carry.

(Refer Slide Time: 01:22:28)



Now, why it saves time? or why is so important can be visible from this tree adder which is a 8 bit CSA, you can see from here a tree bit 8 bit CSA has x_0 to x_7 8 bits. So, first CSA takes care of x_5, x_6, x_7 generate S_1, C_1 , the next CSA takes S_1, C_1 , takes x_4 okay generate S_2, C_2 , next CSA takes x_3 generate S_3, C_3 then takes x_2 generate S_4, C_4 takes x_1 and finally it generate S_5 and take x_0 and you get S_5, C_5 .

(Refer Slide Time: 01:23:29)

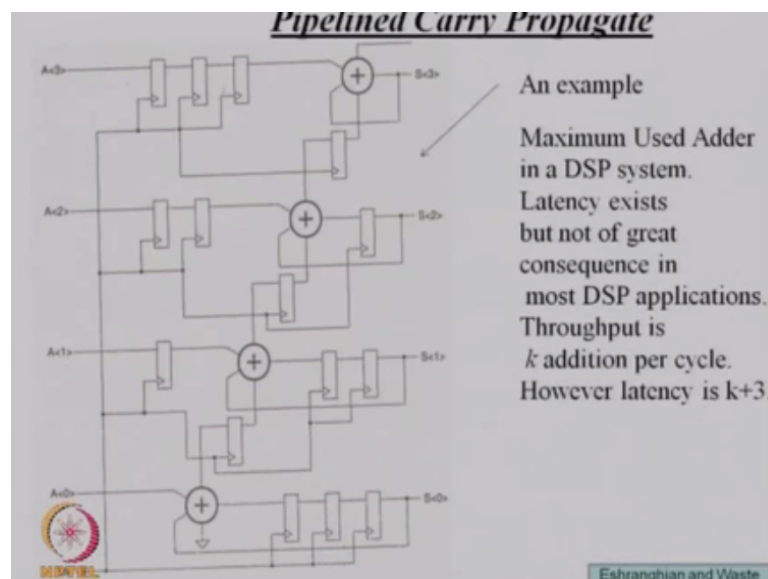


Then, add S_5, C_5 of you actual input carry if you have do a normal addition if input carry this and you have sum and this output. Now based on this tree adder this here is what a typical Wallace tree was done, Wallace tree allows as I said it is like a ELM systems okay, let us say tree additions; simultaneously I do a operation of 3 bits here and 3 bits here generate S_1, C_1 and then this x_1 I gave it here and x_0 I gave it to the next block okay.

So, I generate S3 C3 S4 C4 then in next S2 C3 and one of the S4 I brought here and in the next, I brought the C4 here then I generate S6 then you have the normal CLA. So how many stages I have gone through? 1,2, 3, 4, 5, stages is only required to go through it bits of that, any other adder will give larger timing than this, okay. This is like an ELM which I already said it is a parallel processing and this allows us and why it is allow like this?

Because we are not using any carry in between whatever CNS is getting we are just going add with the numbers. We are not waiting for carry and please remember there is of course some new methods can be found, you can actually have a look up table available for all possible xj, xs, okay and it can be stored in the RAM, given address and you have an output so this is something a fantastic method of actually doing addition which can be utilised in a Wallace tree, which as I say most famous multiplier is a Wallace tree integer multiplier.

(Refer Slide Time: 01:25:39)



Last but not the least okay let us go ahead, okay this is same circuit, before we go to the last one. The another problem which many of us do not feel very obvious is worry is that if you have a large chain of operations like a filter; fir filter or comb filters in the DSP system one finds that there are large number of adders and multipliers required in a DSPs. However, the problem one sees is as I keep saying for the first bits of operation you have one bit delay.

Now, till it generates, you will have 2-bit delay, 3-bit delay, 4-bit delay but at the output you cannot generate all of them till all 4 bits are available okay. Now, this means the delays are not equal okay so throughput is waited any of the 4 factor. So, what we do now? For each of

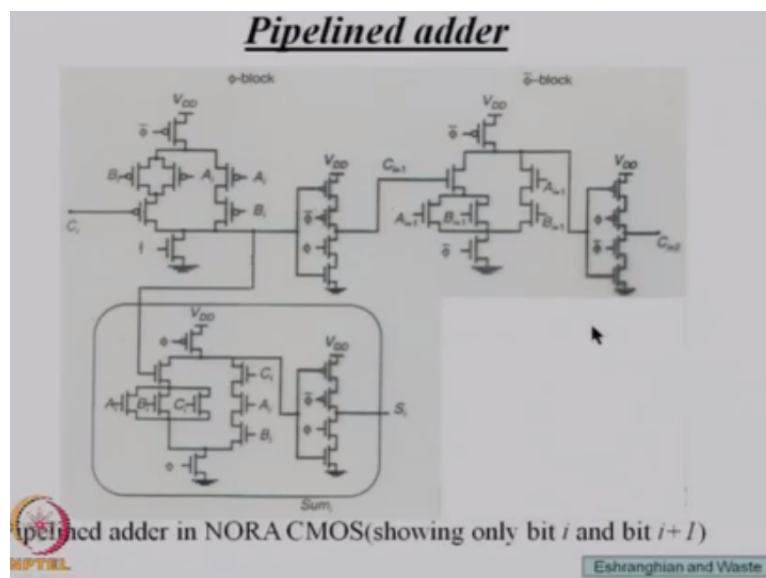
them, we actually equalise the delay; say for example, for this I put 3 register delays or 3 latches.

For the next one, I actually delayed the output here and then I put 3 here to make equal same thing I keep doing, so that at a given input once I give all S3 S2 S1 S0 are available at a given time. The first time it will take all 4 bit delays but when the next bit occurs you are any way moving ahead. So when the first bit it may require what we call latency of 4, it will require 4 clock cycles to reach the output.

But once that first clock, four cycles are over, every clock you have output bits, which are equivalent times is that correct, this is essentially the advantage of pipe lines, okay. This actually has the same throughput though I mean you can see it as a larger throughput and all 4 bits for everyone will keep coming, is like putting thing in a pipe, it will take some time before first thing come out or once in then the continuous flow is available.

So, the frequency or the speed is much higher except for the first 4 cycles, it also allows you to data at a given time available for the next processing, okay. So, in many case irrespective whether which kind of this +es you use, please remember this + can be of any kind okay, but the pipe lining may be still done to avoid the delay problems and also to generate the equal delay outputs, okay.

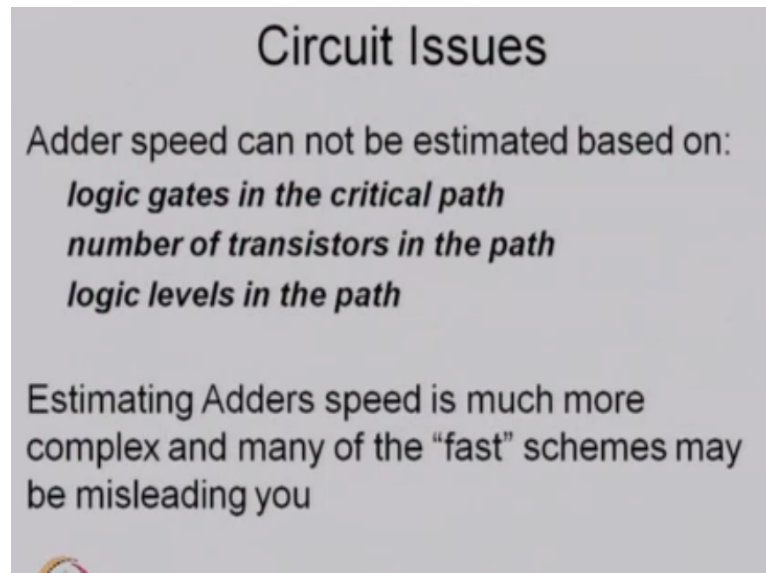
(Refer Slide Time: 01:27:55)



This is an example using an NORA CMOS pipe lined adder could be done. This is available in the Eshraghian and Weste. I just explain what it is? This is essentially a NORA; please

remember it has 2 clocks; 5 and 5 bar runs throughout and it is very interesting. Some are called fly box. Some evaluate in 5 bar which is called 5 5 bar or NORA or modified domino as a sometime some people call and it shows you.

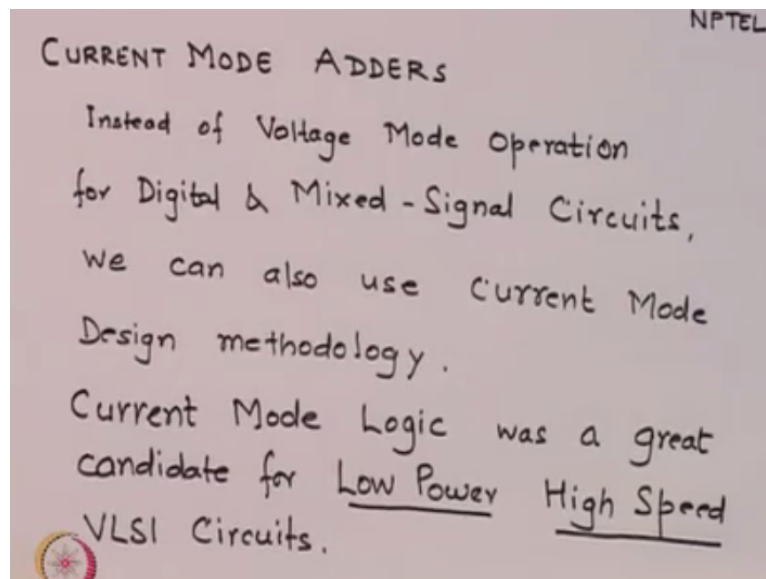
(Refer Slide Time: 01:28:29)



One should not forget that adders speed cannot be estimated just by the critical path delays in logic, number of transistor in the path or logic levels in the path, actually it is very complex system, estimating adder speed is much more complex and many of the fast schemes may be actually very slow at the end of the day, when the net delays you calculate of the inter connect everything, you will figure out that the fastest one device are not really the fastest one.

So, please always worry about when you design, you stimulate and only then verify which one for your application is the fastest adder, do not go by that CLA with CSA kind or something is the fastest in logic yes but in reality every other thing should be taken before you actually look for it. The last and the most interesting adder which has you know, which I thought normally people do not talk so much.

(Refer Slide Time: 01:29:42)

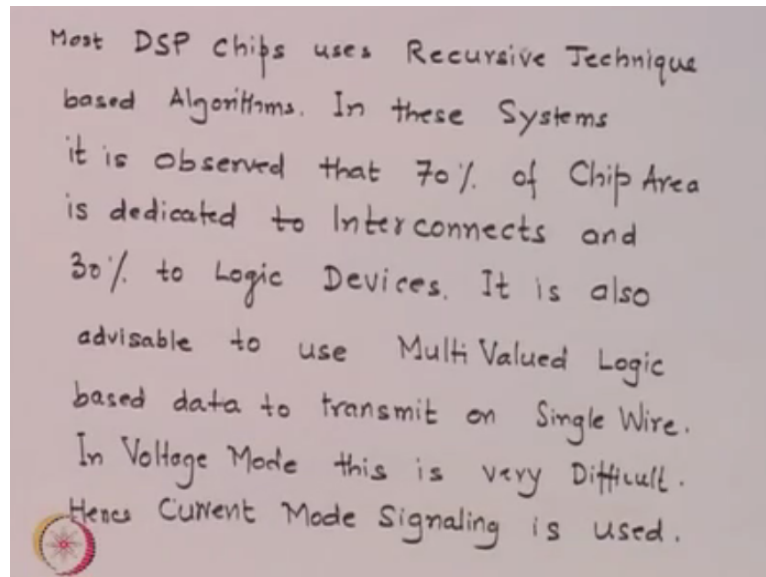


So, I thought may be in my course of advanced VLSI, I will talk about that. The present day systems if you see in the real life consumer electronic systems or even other useful systems like microprocessor based systems, one is finding that, most of this systems are multi modes, that is; they both are analogue operations as well as digital operation called mixed modes. Since you have a mixed mode operation, almost all operations in digital preferred with 2 logic binary numbers 0 and 1s.

So, that is why they are actually digits which use binary numbers. Now and since binary numbers are very easy to generate by the 2 voltage levels, almost all digital logic binary logic is based on voltage logic, 2 voltage logics, okay. But, if you have anyway going to a mixed mode signalling then why not have at least some part of the adder part or some part of the logic; some part of the function which can be done analogue a way.

And one of the major advantage of analogue way is you do not actually add anything or subtract or we will divide in voltages, we actually do in currents. Because, analogue allows you to add currents much more easily. So, I give you some hints. Current mode logic was a great candidate for low power high speed VLSI circuit, particularly mixed mode circuits, pure digital may or may not okay.

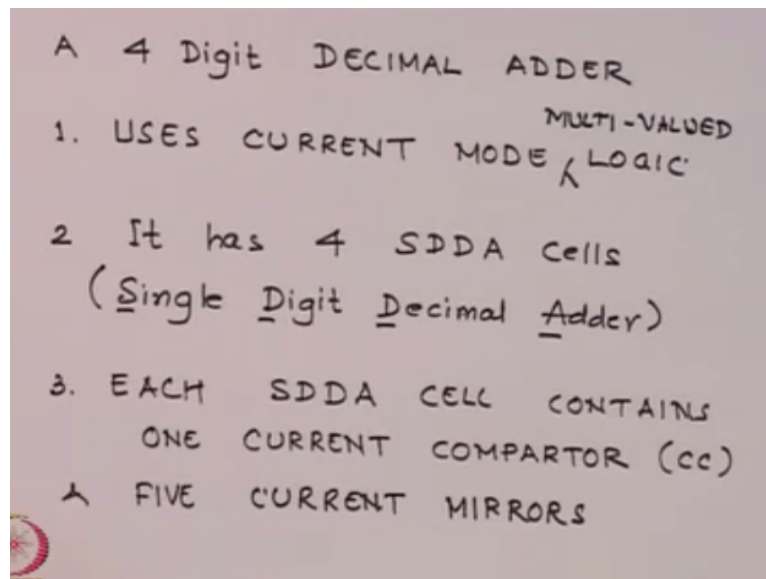
(Refer Slide Time: 01:31:17)



Most DSP chips use recursive techniques, they based on those algorithms. In this systems, it is very interesting data that 70 percent of the chip area is dedicated to interconnects and just 30 percent to the logic. It is also therefore advisable to use multi valued logic to transmit on a single line. Because if you have only 2, this then you have the problems, if you can put multiple levels then you can say the data can be separated by their levels okay.

So, a single inter connect then will pass larger data compared to only one bit at a time, okay. So is called multi valued logic okay which is essentially uses of course you can say always you have a, 0 to 1 volt you can have 0 to 0.4, 0.6, 0.8 as multi valued but to separate and compare these 2 voltages, separate voltages is very very difficult compared to the current separations and therefore almost all multi value logic which is used is use current mode signalling.

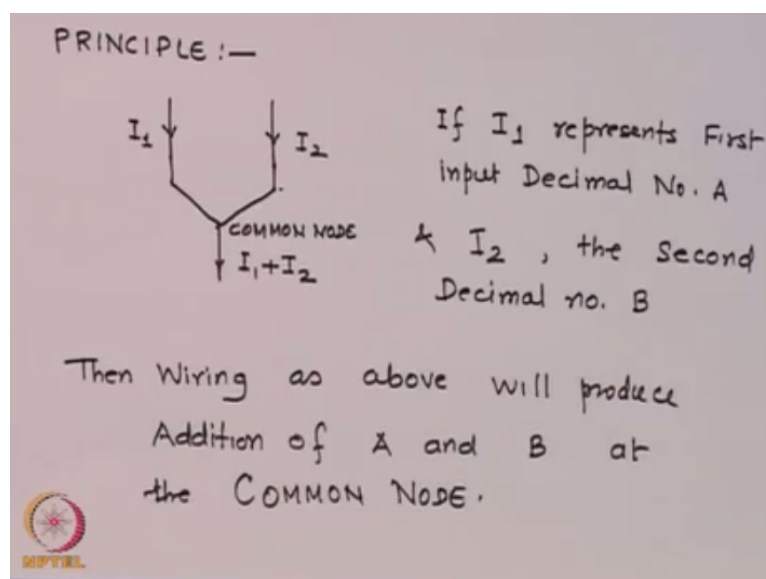
(Refer Slide Time: 01:32:55)



And therefore and current modes are very very easy to implement on analogue blocks and therefore if I am going to add numbers why should I convert my digital analogue number into a digital binary number and then add and then come back to the analogue numbers, okay. So, I may directly added decimal numbers in analogue fashion and here is the one which is an example, which show you that these are the fastest may be fastest.

I will never say all the time but certainly low power. It uses current mode logic, it has 4 cells called SDDA, single digit decimal adder I repeat, it uses current mode multi valued logic, I should say so very clearly sorry is uses current mode multi valued logic. It has 4 SDDA cells single digit decimal adder, each SDDA is cell contains one current comparator, 5 current mirrors.

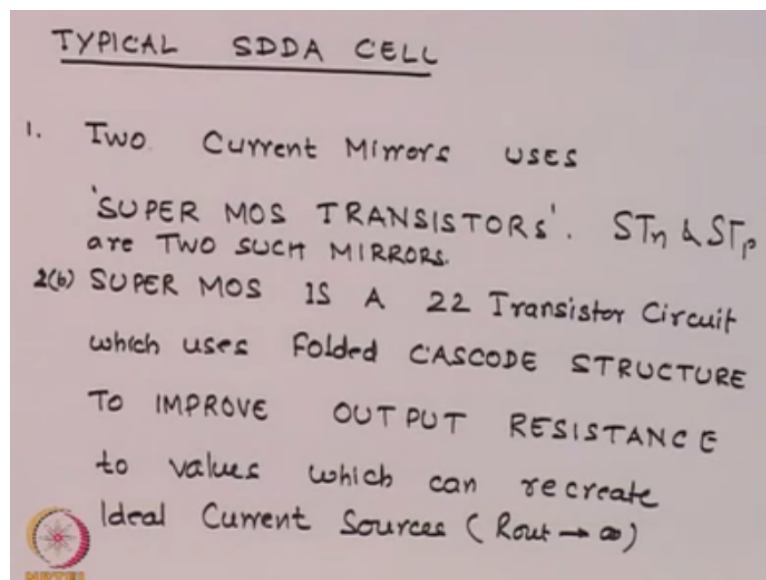
(Refer Slide Time: 01:34:10)



I repeat, each SDDA has one current comparator and 5 current mirrors okay. Now what is the principle I am really looking in this add part there. Very simple principle is used, if you have 2 wires and you connect them at a common node. Let us say, each is carrying a current I_1 and the other one carrying I_2 , let us say I_1 represents one decimal number, I_2 represents a another decimal number.

Then if you connect them we know by simple circuit theory then the net current will be $I_1 + I_2$ that means corresponding to I_1 prior to which is the addition of I_1 and I_2 which is same equivalent to addition of your equivalent decimal numbers okay. The wiring as above will produce addition of A and B at the common node okay. This is the simplest principle of addition we do in the case of currents.

(Refer Slide Time: 01:35:16)

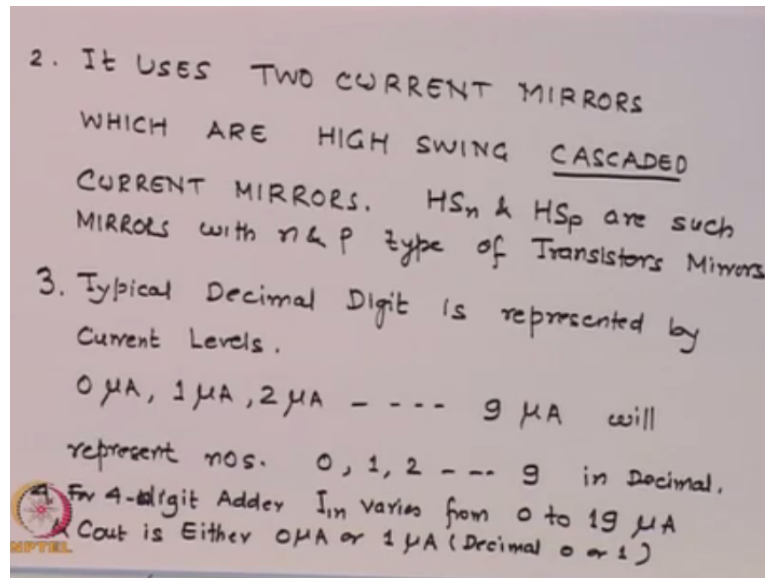


Please remember, if I connect it to voltage, do you believe that this will be possible because then it will actually compete each other to get common voltage okay. Here you just add okay. So for a adder application, current modes seems to be very very simple solution for you. I just now said a typical SDDA cell contains 5 current mirrors, there are 2 kinds of current mirrors I am using.

Two current mirrors uses, what we call using super MOS transistor. I will show little later what they are and they are 2 kinds. One is ST_n , other is ST_p . Please remember you have 2 source current and sink currents possible, P channels normally use for sourcing, N channels are normally use for sinking, okay. So you have 2 source or sink currents mirrors coming from ST_n and ST_p .

They use super MOS transistor, we will show what they are. A typical super MOS transistor either 22 transistors circuit using a normal folded CASCODE structures which improves output resistance in a very very large fashion. Please remember if there is a gain of the CASCODE gains of 1000 or 10000 then the R_{out} will be 10000 square times the output available for a single mass transistors.

(Refer Slide Time: 01:36:53)



So, it boost up the output resistance this, why are we looking for higher output resistance? because for a good current source, the resistance of a current source should be infinite, okay as preferable. Super MOS allows you to do as good a current sourcing or sinking as possible. It uses other 2 current mirrors which are called CASCADED current mirrors which are called high swing.

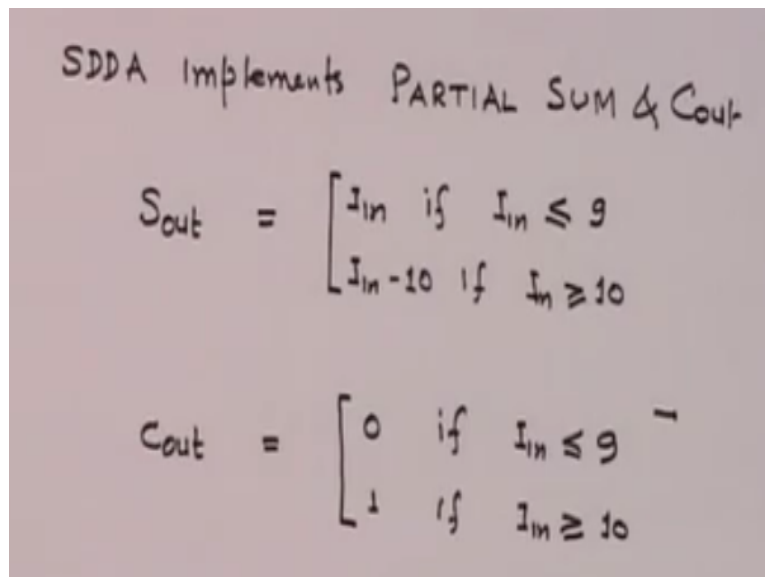
Full swings larger current swings and they are named as H_{SN} and H_{SP} for N and P channels. The typical the way we define the typical decimal digit is represented by current levels. I already said we are talking a multi valued current logic okay. So we say there are let us say 10; digits are 10; 0 to 9, so we have 10 currents, 0 micron, 1 micron, step of 1 micron, you can have any other step but example is one micron, okay.

They may represent numbers 0 micron shows 0, 1 represent 1, 9 represent 9, 9 micron represents 9 in decimal. Now 4 digit adder receives an input current I_{in} , which can vary for 0 to 9, why I 4 I said? You can see from 4 digit adder okay. If you have this, if I had 2 numbers

at best 7 + 8 is 15, 9 + 9 is 18. So the highest number you will require is only 19 is that correct. So above 10, the next number is the 2 bits when you add just 19 in decimal.

So, we now say a 4 bit digit adder that is A0 A1 A2 A3 A4 kinds. It varies from, if we are adding only 2 bits in decimal, you require only 19 steps, 0 to 19 that is 20 steps actually. Now you also know in the case of decimal numbers since you have only 2 bits of decimal you can either have a carry of 1, or if 5 + 7 has a carry 1, 3 + 5 does not have a carry. So either the carry is 0, or carry is 1, there is no other, you are never going beyond 20 in this numbers.

(Refer Slide Time: 01:39:25)



SDDA implements PARTIAL SUM & Cout

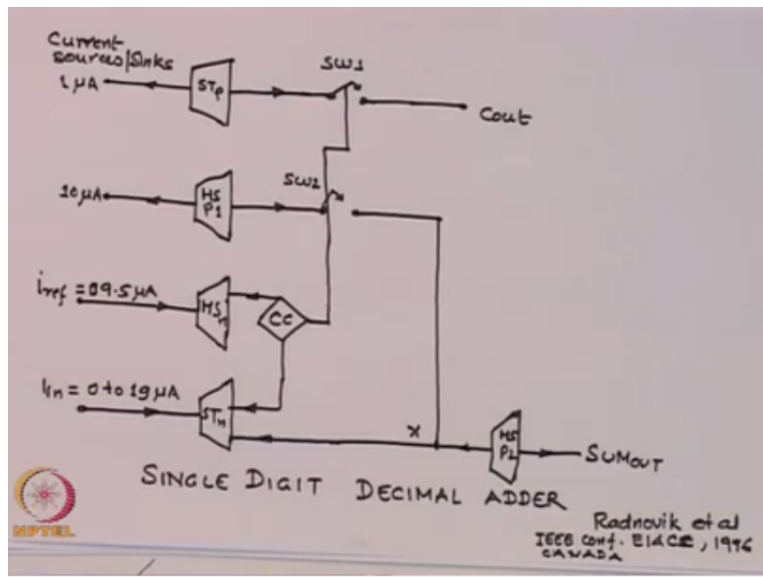
$$S_{out} = \begin{cases} I_{in} & \text{if } I_{in} \leq 9 \\ I_{in} - 10 & \text{if } I_{in} \geq 10 \end{cases}$$

$$C_{out} = \begin{cases} 0 & \text{if } I_{in} \leq 9 \\ 1 & \text{if } I_{in} \geq 10 \end{cases}$$

So, at no time that 2 carry or 3 carry are available. So you have possibility of 2 carries; either 0 carry or 1 carry, 0 stand for 0 microns, 1 stand for 1 micron, okay. So, I have already created requirements, I implement partial sum and output which I show you a circuit of SDDA, you just look at expression representing that. A sum output is equal to input current, we will show you this how it happens.

If $I_{in} < 10$, okay please remember I_{in} is nothing but $I_1 + I_2$ whatever number you were looking for, is that okay? Now, if it is less than 9, the sum will be always equal to the whatever I_{in} new are introduced, the same sum should come, 5 + 4 is the 9, so sum is 9. If it is 7 + 8, then the sum is 15 which means you have 10 above, so $I_{in} - 10$ is the new sum now. You generate carry out of it then the sum term is only 5, okay.

(Refer Slide Time: 01:40:57)



So, S out is $I - 10$ if $I_n > 10$, okay. By same logic C out can be always 0, if $I_n < 9$ because $5 + 4$; $3 + 5$, anytime no carry, anything beyond 10 will be carry, so you say C out is 0, if $I_n < 9$. C out is 1, if $I_n \leq 10$. Quickly we show you the circuit and will switch off this adder part. Here is 4 current I already told you that there are mirrors this is STP, this is HSP, this is HSN, this is STN, this is HSP 2.

Now, take the circuit from here, the first one has the sink current of 1 micron, the second one has a 10 micron, okay, these are that switch over above 10, anything above high in greater than 10 or less than 10, so we have 2 currents created for the 1 and 0, okay. Please remember, 1 stand for carry 1, okay and 0 stands for carry 0, so what we do is the following. If it is more than 10, you have to subtract and generate 1 for that.

Okay, this is what we have to do. We create a reference current which is 19 by 2, okay the maximum current divided by 2 which is called, so that 50 , 50 above and below, so 9.5 and we have the input current which is coming from where? $I_1 + I_2$ addition of that, which is my input current which can be 0 to 19 any number which is sourcing to STN current mirror. Please remember, current mirror can always be created out of a standard 2 to 4 transistor circuits; Widlar current source, Wilson current source or even normal source or current mirrors.

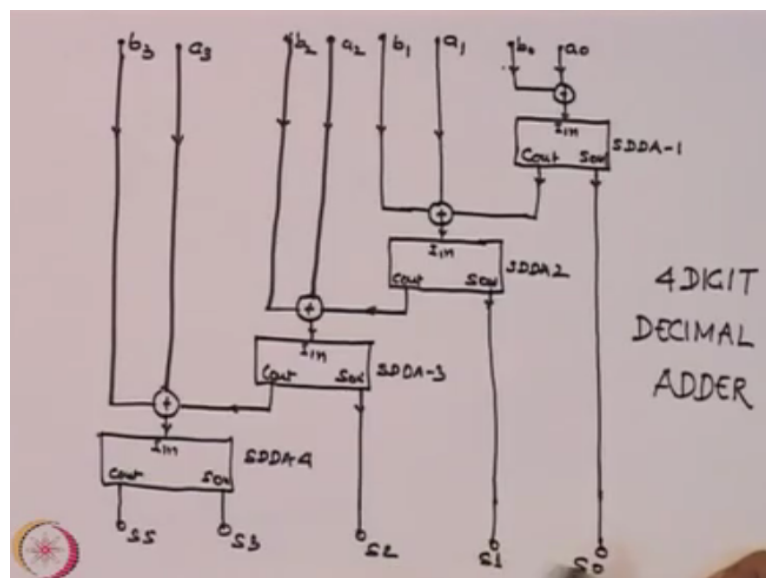
Here is the current comparator, the way it operates. The way it operates is if your I_n is less than 9, okay less than 9, we know S out going to be 1, sorry S out is going to be I_n . So, I want this output at that time you want C out to be 0, so switch 1 and switch 2 less remember this is

less than 9, you switch of these both of them output remains 0 initially, no pass. But if would have been greater than 10, it would have pull it down and this 1 would have transmitted.

Is that clear? So the carry 0 and carry1 can be created depending on the CC output okay. Some can be created from this, if it is 10 micron and this switch closes down this is 10, now please remember this is anything between 10 to 19 and if you see the sum has to be either I_{in} or $I_{in} - 10$, depending on whether this condition is more than 9.5 or less than 9.5, this sum will be either this sum or coming from this sum.

So, this adder allows you single digit adder allows you do this function, you can go and look yourself once again then you will verify. You want sum to be I_{in} okay; if this is current source and let say this is less than 10 or 9, okay. If it is less than 9, this of course would have actually this is less than 9 means this is 0, okay so whatever is there will be outputted okay. If it is more and greater, then this will be outputted which will switch on this.

(Refer Slide Time: 01:44:38)



And then the 10 will be outputted, so that is the way the logic goes, is that clear? Okay, now how to implement the 4 digit? Here are last slide for this adder part, you have 4 SDDA shown here, SDDA1, SDDA2, the first 2 bits A0 B0 which means currents equivalent of that, this is our normal adder which is nothing but connecting wired and nothing more. Please add is nothing there in the circuit just connect.

This creates sure I_{in} for this SDDA, we have just seen the operation on SDDA, it will create output carry and the first sum okay, S_0 is out, carry is generated. Next 2 bits add and now add

this additional current C, please remember I can add any number because these are 3 parallel current pass adjoint, the sum 3 will actually appear. The only thing you should see here all are in the same directions, which will not subtract there okay.

Otherwise, there is a hogging problems okay. As soon as I get A1 B1 + whatever C there, I can get new I_{in} immediately creates S out with the less than 10 or less this, generates new C 1 or 0 whichever it is. Take next 2 bits use this C, find out again generate C last 2 bits generate this, so you get S₅, S₃, S₂, S₁, S₀, please remember last C out is your S₅, so you have created without going through any logic per say as voltage logic does only adding operations or comparing operations.

And we can generate the 4 digit decimal adders okay. Now this is the current mode, please remember these are small analogue transistors the currents could be less than microns or tens of Nano amps. Therefore, the power dissipation will be extremely small, okay. So in any case, since their current driven, they are not capacitively charging or discharging on them they are just passing the current through a transistor okay.

So, there is no question of propagation delays through that, okay. There is the stability issue which I did not discuss any analogue has an issue but otherwise current mode multi valued logic can create a decimal adder which is extremely low power very high speed which can compete any of the digital block any time is that okay? So, in a multimode system which requires both analogue digital applications.

Sometimes there is even an advantage of converting your digital data to decimal like a BCD converters and actually do decimal operation and then go back to decimal to binary converters which may still be faster and low power. So, this is a new technique which in the current technology of 2012 find more and more use. Because, as we say effort in 2000 onward is to deduce low power.

They are working on 0.6-volt process and now small channel devices, all kinds of things but current requirements may not be large high because in all digital logic that is the major issue, all current has to be high enough to deduce the charging or discharge time, here we do not require large currents because we are not using charge discharge techniques okay. Therefore there will be always high speed.

There will always low power but then why do not you use? Use as I say is there are many analogue issues which probably we are missing here and we do not want to discuss in this part of the adder course, some other time, some other day. Thank you for the day.