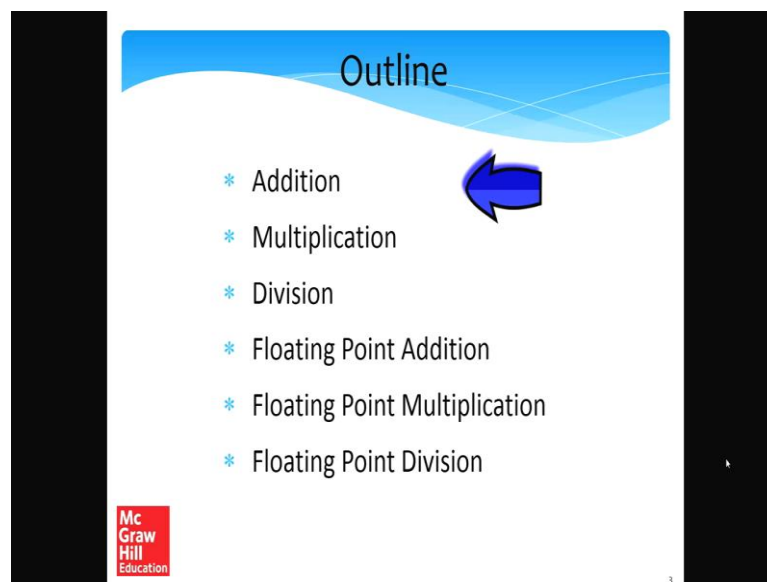


Computer Architecture
Prof. Smruti Ranjan Sarangi
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Lecture – 17
Computer Arithmetic Part-I

Welcome to chapter 7. This is a chapter on Computer Arithmetic. We shall use all the knowledge that we have gained in designing digital logic, all the knowledge that we gained in chapter 2; the language of bits to design circuits in this chapter, to implement adders, multipliers and dividers. So, this is chapter 7 of the book computer organization and architecture published by McGraw hill in 2015.

(Refer Slide Time: 01:29)



So, we will have we have these 6 parts in this chapter, we have addition, multiplication and division. So, these are integer addition, multiplication and division, and then we have floating point addition multiplication and division. So, the entire chapter is broken down into 6 sections.

(Refer Slide Time: 01:49)

Adding Two 1 bit Numbers


- * Let us add two 1 bit numbers – a and b

$\begin{array}{r} \overset{1}{1} \\ + 1 \\ \hline 10 \end{array} \quad (2)$	$\begin{array}{r} \overset{1}{1011} \quad (11) \\ + 0010 \quad (2) \\ \hline 1101 \quad (10) \\ \hline \overset{1}{752} \\ + 129 \\ \hline 881 \end{array}$
---	---

- * $0+0=00$
- * $1+0=01$
- * $0+1=01$ (Sum)
- * $1+1=10$ (Carry)

- * The **lsb** of the result is known, as the **sum**, and the **msb** is known as the **carry**

4



So, let us take a look at the simplest problem, we just to add 2 1 bit numbers a and b. So, if we add to 1 bit numbers, what is it that we can get? The largest number that we can get will actually require 2 bit, let us see how and why. If we add 0 and 0, so let us assume the output is 2 bits, the output is 0 0. If you add 1 and 0, the output is 0 1. Similarly if you add 0 1 1, the output is 0 1. And if you add 1 and 1, the output is 1 0. So, 1 plus 1 is 2, and representation of 2 is 1 and 0. So, what we see is that for the first three results 1 bit is sufficient, for the last result we need 2 bits. So, pretty much we can say that we add to 1 bit numbers. The final result can take 2 bits, is better if you allocate 2 bits.

So, least significant bit of the result of, it is a 2 bit result right. The least significant bit which is let us say this bit, is known as the sum. And the most significant bit is known as they carry. So, we shall see why. If you add 0 and 0 the sum is 0 nothing has being carried. Same is true for 1 and 0 and 0 and 1, but we add 1 and 1, 1 plus 1 then. So, if you do a binary addition of 1 and 1. So, what we get? So, what we would get is, that essentially the unit's position. I am sorry the least significant position will be a 0, and we will be carrying 1 to the next position. So, basically the carry will be 1 and the sum will be 0. So, 1 0 in decimal is 2 all right.

So, let me give you some more examples of let us say binary addition. We would do it exactly the same way as we do in base 10, and we can take a look at the carry business in some more detail. Let us assume is 1 0 1 1 plus 0 0 1 0. So, 1 0 1 1 if you consider to be

an unsigned number, its 11 and base 10, and 0 0 1 0 is 2 and base 10. So, we add it should be 13 and base 10. So, 1 plus 1 is 1 1 plus 1, it is 2 in binary it 1 0, so the sum is 0 and the carry is 1. So, the carry goes to the next position, this position. So, 1 plus 0 plus 0 is 1, and we again have 1.

So, what is the sum it is 8 plus 4 12 plus 1 - 13 which is exactly what we would expect. So, additional binary is exactly the same as addition in base 10. There is no difference at all, and in base 10 we also carry a 1. So, so let us in, I was to do the same similar kind of an addition. Let us say in base 10, which would maybe p something of this type. So, 2 plus 9 is 11 I write 1 and then I do I carry. So, it is 1 plus 5 is 6 plus 2 is 8 and 7 plus 1 is 8. So, same way we do a carry in base 10, we do a carry in base 2 as well and, in the case of a 1 bit, we call the least significant bit as the sum, and the next bit as the carry.

(Refer Slide Time: 05:57)

Sum and Carry

$$\begin{array}{r} + \\ a \\ b \\ \hline \end{array}$$

carry

sum

a	b	s	c
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

$$S = a \oplus b = \overline{a} \cdot b + a \cdot \overline{b}$$

$$[c = a \cdot b]$$

Truth Table

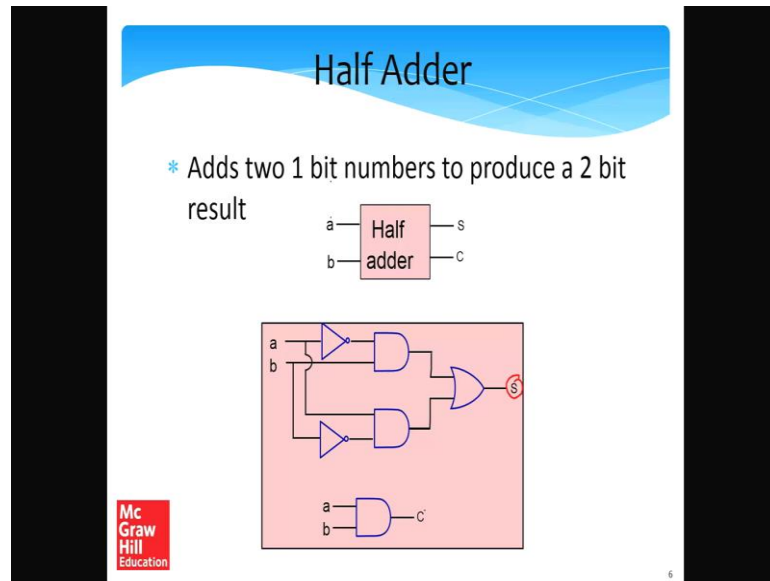
McGraw Hill Education

5

So, if I draw a simple truth table of what are the values of a and b, and what is the sum and carry. So, we see of a and b are 0 and 0, the sum is 0, the carry is 0. If a is 0 and b is 1, the sum is 1 and carry is 0. Likewise if a is 1 and b is 0, the sum is 1 and the carry here is 0. And lastly if a is 1 and b is 1, the sum is 0 and the carry is 1. So, the sum is the LSB mind you, and carries MSB. So, what we can see if you want to implement this, you know a Boolean system. We can see that the sum bit over here is actually a XOR b. So, why is this case? So, basically if 0 and 0 the XOR of them is 0, 0 and 1 exclusive r is 1, 1 and 0 is 1 and a 1 and 1 its 0.

So, we can say that this is this function. So, go back to chapter 2, again for an explanation of this is exclusive or operator, which can be expanded to this. So, the plus sign is denoting the OR operation, and the dot sign is denoting the AND operation. So, is essentially a compliment and b or a and b compliment, so that is a sum. And the carry will happen in only 1 case, then both a and b are 1.

(Refer Slide Time: 07:51)



So, we can see it is a and b that is when we will have a carry. So, we can then implement a simple Boolean circuit called a half adder, which adds to 1 bit numbers to produce a to 2 bit result. So, in this case we will add a and b, and the result will have a sum and carry bit. So, we just need to implement the circuit, the Boolean formulae showed over here. So, what we see over here is, to that this particular formula needs to be implemented. So, we take a compliment, we add it with b, and another a and added with b complement. After 2 and gates a compute or function and this becomes sum, and the carry was anyway a and b. So, what we do is, we pass a and b by an and gate, and the result is the carry.

(Refer Slide Time: 08:50)

Full Adder

Add **three** 1 bit numbers to produce a 2 bit output

$$a + b + c_{in} = 2 * c_{out} + s$$

a	b	c _{in}	s	c _{out}
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

The slide includes a truth table for a full adder, a handwritten binary addition example (1101 + 1101), and a block diagram of a full adder with inputs a, b, and c_{in}, and outputs s and c_{out}. The truth table shows the sum (s) and carry-out (c_{out}) for all combinations of inputs a, b, and c_{in}. The handwritten diagram shows the addition of 1101 and 1101, resulting in 11000, with a carry of 1 from the second bit to the third bit. The block diagram shows a full adder (FA) block with three inputs (a, b, c_{in}) and two outputs (s, c_{out}).

Let us now consider full adder. So, basically full adder will be required, when we want to add 3 bit and here is when will need to add 3 bits. So, let us assume that there is a large Boolean number right, something of this type. So, in this case when you perform the addition for the first, we can for the LSB bits, we can use a half adder, it will produce the sum of 0 and a carry of 1. So, mind you for the second position, we are actually adding 3 bits; we are adding 1 plus 0 plus 1 right.

So, this will produce a sum of 0 and a of carry a 1, but essentially a full adder will be something that takes 3 bits as input. So, you take 3 bits as input; 1 plus. The ma the largest possible sum that we can have, will essentially be 1 plus 1 plus 1 which is 3, this binary representation is 1 and 1. So, 2 bits are sufficient. So, in this case a full adder will have 3 inputs, and it will have 2 outputs. So, in this sense 2 bits are sufficient to get the outputs. So, what we can say is, that the 3 input bits, can basically be called a b and I carry in. So, in this case you know the first number here can be called a, this number can be called b, and there is a carry which is actually coming from the right. So, this can be called the carry in.


Then what will be the output, the output well will the output will be a sum, the sum s, and we will have a carry as an output in to differentiate that from carry in let us call it carry out all right. So, there are 3 bits coming into full adder which is a b and carry in, if you are adding 2 large, if you are adding to multi bit binary numbers. So, a will be let say

the i th bit of 1 number, b will be the i th bit of the second number, and carry in will be the carry that is coming in from the right, and the outputs will be the sum and the carry out. So, the number will be equal to 2 times carry out plus s ; simple binary expansion, and if this is what the truth table will look like. So, if a b and carry in are all 0s. So, naturally the sum and carry out both are 0. Similarly if a and b are 1 1 each carry in are 0, its 0 and 1s. So, read just needs to verify all the entries in this table and ensure that they are correct. So, let us take a look at the last entry. So, this is the sum a b and carry in are all 1s. So, 1 plus 1 plus 1 is 3, here 1 1 and binary. So, sum is 1 and the carry out is 1.

(Refer Slide Time: 12:02)

Equations for the Full Adder

$$\begin{aligned}
 s &= a \oplus b \oplus c_{in} && \begin{matrix} 1 \oplus 1 \oplus 1 \\ = 0 \oplus 1 \\ = 1 \end{matrix} \\
 &= (a \cdot \bar{b} + \bar{a} \cdot b) \oplus c_{in} \\
 &= (a \cdot \bar{b} + \bar{a} \cdot b) \cdot \bar{c}_{in} + \overline{(a \cdot \bar{b} + \bar{a} \cdot b) \cdot c_{in}} \\
 &= a \cdot \bar{b} \cdot \bar{c}_{in} + \bar{a} \cdot b \cdot \bar{c}_{in} + \overline{(a \cdot \bar{b}) \cdot (\bar{a} \cdot b) \cdot c_{in}} \\
 &= a \cdot \bar{b} \cdot \bar{c}_{in} + \bar{a} \cdot b \cdot \bar{c}_{in} + (\bar{a} + b) \cdot (a + \bar{b}) \cdot c_{in} \\
 &= \{ a \cdot \bar{b} \cdot \bar{c}_{in} + \bar{a} \cdot b \cdot \bar{c}_{in} + \bar{a} \cdot \bar{b} \cdot c_{in} + a \cdot b \cdot c_{in} \} \\
 &\xrightarrow{2/3 \text{ of } a, b, c_{in}} c_{out} = \underline{a \cdot b} + \underline{a \cdot c_{in}} + \underline{b \cdot c_{in}}
 \end{aligned}$$


8

So, what are the equations for the full adder, let us take a look at this. So, the sum if you take a look at the sum let us again go back. So, essentially in the sum if. So, when is the sum equal to 1, the sum is equal to 1, when b is 1 and a and carry in are 0, or when a is 1, or when carry in is 1, the rest 2 are 0, or when all 3 are 1. So, what we can see is the sum is 1 only when, if and only if out of a b and carry in an odd number of them are equal to 1 right. So, basically in this case b is 1 the rest are 0, in this case a is 1 the rest are 0, in this case C in is 1 the rest is 0, or all 3 of them are equal to 1.

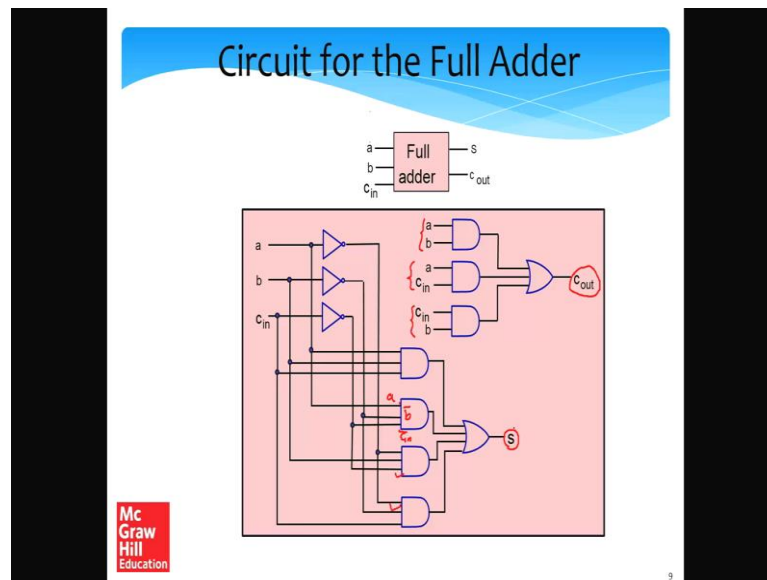
So, typically any such function for the sum can be represented like this as a XOR b XOR carry in. So, this will be true only when, an odd number of a b and carry in are 1. So, either 1 of them is 1 the rest are 0, 0 XOR 1 is 1, or when all 3 of them are equal to 1. So,

basically $1 \text{ XOR } 1 \text{ XOR } 1$ can be thought of like this, that $1 \text{ XOR } 1$ is 0 and then 0 exclusive or with 1 is 1.

So, we can just do a simple algebraic expansion of this. We can take a sense the rule of associativity is followed. We can take a XOR b and expand to a and b complement, or a complement and b, and just keep on expanding it. So, finally, if you do the algebra, this is the final result that we get. So, in this expression n over here, what we see is that when will this be true. So, this expression will be true when a is 1; both b and the carry in are 0. Similarly in this expression this will be true when b is 1, and both a and the carry in are 0. Similarly for this, this takes care of the first 3 cases. In the last case will happen, when the last case will be true, then all three of a b and the carry in all three of them are equal to 1. So, this is exactly what we have in this is truth table over here. This is exactly what we are trying to capture.

Now, let us come to that carry out. So, when is the carry out equal to true? So, the carry out is equal to true, when a and b both are 1 carry in a 0, when b and carry in are 1 a is 0, or a and carry in r 1 all right, when all 3 are. So, essentially the carry out is true when at least 2 of a b and carry in are equal to 1. So, this can be nicely captured in this thing. So, so let me get summarized when is carry out equal to true. Carry out is equal to true at least 2 out of 3 of a b and carry in, are equal to true. So, this will become this is the way to express this is. If a and b are true, then a and b this expression will be true. If a and carry in are true then this will be true, or this expression will be true. So, essentially if these 3 bits any 2 of them are true, at least 1 of the conditions here should be true, and thus carry out will be equal to 1; otherwise carry out will be equal to 0.

(Refer Slide Time: 16:35)



So, for creating a full adder, we basically need to create a circuit that will realize these Boolean equations in hardware. So, this is the circuit of a full adder looks complicated, but it is not complicated. So, carry out is slightly easier to understand. So, carry out is a and b or a in carry in or b and carry in. So, what we can see is, that this is a and b, this is a and carry in, this is b and carry in. So, what we do is, that we have 3 AND gates, and we have an OR gate whose output is carry out.

Now, let us consider the sum. So, in the case of the sum what we need to do is. We need to figure out the value that each of these expressions. So, let us consider the first 1 a and b complement and complement of carry in. So, this will get captured like this; a is coming here, then ah. I am sorry yeah this 1 a is coming here. So, this is a, this value is b complement, and this value over here is a complement of carry in. So, this is getting captured over here. So, the output is a and b complement, and complement of carry in.

Similarly, the rest of the 2 and gates are capturing the rest of the 2 terms, which are this and this, and we need 1 and gate to capture this term, which is a and b and carry in which is the first and gate, where we are just computing and other values. And at the end to compute the, or we have a 4 input or gate, so this gives us the sum. So, computing the sum is more difficult requires more hardware, as compared to computing the carry out, but this is pretty much the circuit of a full adder for you, which takes 3 bits of the input. Adds them, and there is a 2 bit output. For the LSB, the least significant bit is the sum,

and the most significant bit is the carry out. So, both of these are getting computed in these fashion.

(Refer Slide Time: 18:53)

The slide shows the addition of two 4-bit numbers: 1011 and 0101. The result is 10000, where the leading 1 is the carry out. The diagram uses arrows to show the carry propagation from right to left. The steps listed are:

- * We start from the **lsb**
- * Add the corresponding pair of bits and the **carry in**
- * Produce a **sum bit** and a **carry out**

Mc Graw Hill Education logo is visible in the bottom left corner of the slide.


Let us consider the addition of 2 n bit numbers. So, the way that we start is we just. So, so these are 2 n bit binary numbers. The way we add them is, exactly the same way as we add base 10 numbers. So, we start from the right and move towards the left. So, we first add the first pair of numbers and. So, if we add the first pair of numbers what we see, is that we add 1 and 1. So, 1 plus 1 is 1 0, we have 0 has the sum bit, and we carry over 1 to the next position. In the next position we need to add 1 plus 1. So, again the sum bit will be 0, and again we will carry over a 1 to the next position, again we add 1 plus 1. So, the sum bit is again 0, and we carry over 1, so we have 0. And then finally, we again carry over a 1 to an x position, and the addition completes. So, the final result is 1 and 4 0s.

So, what do we do, we start from the least significant position, we start from the LSB, then we add the corresponding pair of bits and the carry in and we produce a sum and the carry out. So, the carry out pretty much of the i th position, is the carry in for the i plus 1 th position.

(Refer Slide Time: 20:24)

Observations

- * We keep adding pairs of bits, and proceed from the **lsb to the msb**
- * If a **carry is generated**, we add it to the next pair of bits
- * At the last step, if a carry is **generated**, then it becomes the **msb of the result**
- * The carry effectively **ripples** through the bits



11

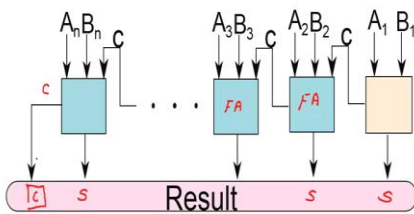
So, some quick observations are due. So, we need to keep adding the pair of bits, pairs of bits, and proceed from the LSB to the MSB. If a carry is generated we add it to the next pair of bits, and at the last step if a carry is generated, then see if you see here in the last step a carry is generated. So, it essentially becomes the most significant bit the MSB of the result. So, the carry effectively ripples through the bits.

(Refer Slide Time: 20:58)


Ripple Carry Adder

$$\begin{matrix} A & A_n & \dots & A_1 \\ B & B_n & \dots & B_1 \end{matrix}$$

c → carry Full adder Half adder



Result [c] s s s



12

So, let us first design a very simple adder called a ripple carry adder. So, in this case we will add 2 multi bit binary numbers a and b. So, let us assume that they have n bits each,

and the bits are a_1 to a_n right; that is the addition that we do plus. So, b_n means the n th bit and b_1 means the first bit. So, when we add, we will first add a_1 and b_1 . So, mind you there is no carry input, so we can use a half adder. So, the half adder will produce a sum bit, and it will produce a carry. Subsequently we need to add a_2 b_2 and the carry that was generated. So, we need a full adder.

So, again we save the sum bit over here, and we move the carry to the next position. So, here also we use a full adder, and we add a_3 b_3 and the carry. So, in a similar fashion they go till the n th position which is the last point. We add the carry a_n and b_n . So, finally, we have a sum bit and we have a carry out. So, since we have nothing more to add, the carry out becomes the most significant bit of the result right. So, this is this carry out, the bit that comes out. So, this is exactly the same as the addition that was done over here. The only important point to note here is that at the least significant position, we have half adder, because there is no carry input, and the rest of the adders are full adders.

So, this particular adder let me actually go back once again. This is called ripple carry adder. This is essentially the carry is passed from 1 added to the other, it is like a ripple, it is like a wave of carry is going from you know right to left.

(Refer Slide Time: 22:58)

The slide is titled "Operation of the Ripple Carry Adder" and contains the following text:

- * Problem : Add $A + B$
- * Number the bits : A_1 to A_n and B_1 to B_n
 - * lsb $\rightarrow A_1$ and B_1
 - * msb $\rightarrow A_n$ and B_n
- * Use a half adder to add A_1 and B_1
- * Send the carry(c) to a full adder that adds :
 $A_2 + B_2 + c$
- * Proceed in a similar manner till the msb

McGraw Hill Education logo is visible in the bottom left corner of the slide.

So, in this problem what we are essentially doing is. So, this sort of a textual description of what I described on the previous slide, that we start from the right, and we gradually move to the left, and we keep on adding the carry values.

(Refer Slide Time: 23:20)

How long does the Ripple Carry Adder take ?

- * Time :
 - * Time of half adder : t_h
 - * Time of full adder : t_f
 - * Total Time : $t_h + (n-1)t_f$


Mc Graw Hill Education

14

So, how long does the ripple carry adder take right? So, the question is when we design an adder, we should know how long it takes, and you know to perform the addition; such that we can, when we compared to adders, we can compare them on the basis of the time. So, let us assume that time a half adder takes is t_h , and the time a full adder takes is t_f . So, as we see the total time that this adder will take. Let me go back is essentially the time for 1 half adder to come to compute its carry, and then since these are in series, it will be t_h plus n minus 1 in a n bit addition we will have n minus 1 full address. So, it will be t_h plus n minus 1 time t_f .

(Refer Slide Time: 24:13)

Asymptotic Time Complexity

- * Most of the time, we are primarily interested in the **order of the function**
- * For example : we are only interested in the n^2 term in $(2n^2 + 3n + 4)$ ~~$3n + 4$~~ $\approx n^2$
- * We do not care about the **constants**, and terms with **smaller exponents**
 - * $3n$ and 4 
- * We can thus say that :
 - * $2n^2 + 3n + 4$ is order of (n^2)

Mc Graw Hill Education

15

So, let me now define the notion of asymptotic time complexity, which is very important in computer science. So, most the time we are primarily interested in something called, the order of the function for example, let us consider this expression $2n^2 + 3n + 4$. So, if we take a look at it. Well $3n$ is important, 4 is important, but considered n is a fairly large value. So, the only term that actually dominates, this expression is $2n^2$. So, the rest of the terms, in certain sense can be ignored.

Again its being mathematically inaccurate, but we shall see there are some advantages to this. So, basically these terms can technically. Well not technically, but in a certain sense can be ignored and. So, basically even that is say the constant $2n^2$, we can foil in a broadly thinking of the approximate time, we can even ignore the constant 2 , and we can say that this function sort of grows as n^2 right ignoring constants.

So, it is possible to say that a function of the form $2n^2 + 3n + 4$ is, you know the order of n^2 , and the reason it is possible to say this is, because we have decided to get rid of constants. And say if you see you know the maybe the graph of this function will essentially be quadratic graph, might be something like this, and maybe n^2 will sort of be a half of a it will be a something like this, but pretty much you know we sort of want an approximate representation where we do not want many constants. So, we can say that look, as long as 2 curves are, sort of 1 curve is a multiple of the other curve right, it is some constant times another curve will ignore the constant. So, we can say that this is approximately growing as n^2 would grow.

(Refer Slide Time: 26:32)

The O notation

- * Formally :
 - * We say that: $f(n) = O(g(n))$ $2n^2 + 3n + 4 = O(n^2)$ $2n^2 + 3n + 4 \leq 10n^2$ ($n > 100$)
 - * if, $|f(n)| \leq c|g(n)|$, for all $n > n_0$. Here c is a positive constant.
- * In simple terms:
 - * Beyond a certain n , $g(n)$ is greater-than-equal to a certain constant times $f(n)$
 - * For example, beyond 15, $(n^2 + 10n + 16) \leq 2n^2$
 $n_0 = 15$
 $c = 2$

McGraw Hill Education

16

So, let us try to formalize this notion. So, let me introduce the O notation. So, this notation is called O. So, formally we say that a function $f(n)$ is order of. So, O of $g(n)$, if $f(n)$ is less than equal to the constant time $c g(n)$, for all n greater than some n_0 where c is a positive constant. So, let me, I will explain it with an example, but let me say what this means in a slightly colloquial since. So, so let us consider 2 functions $f(n)$ and $g(n)$. So, let us say these function, is the same old function $2n^2 + 3n + 4$.

And we are saying that this is order of n^2 . So, what is basically means, is that we can say that $2n^2 + 3n + 4$. So, let us consider positive n . So, absolute value will go away. So, whenever the absolute value is more of a mathematical artifact. We can say that this is in a less than equal to $10n^2$, for n greater than hundred right. You can definitely say that, this is less than this in this case c will be equal to 10 and n_0 will be equal hundred. So, it can be said that you know this function $2n^2 + 3n + 4$ will essentially grow the same way as n^2 will grow, ignoring constants. So, let us look at another example. So, in simple terms what we are saying is that beyond the certain n , beyond a certain value of n $g(n)$ is greater than or equal to, a certain constant times $f(n)$. for example, beyond 15 $n^2 + 10n + 16$ is less than equal to $2n^2$. So, in this case we can say n_0 is equal to 15, and c is equal to 2.

(Refer Slide Time: 28:44)

Example of the big O Notation

$f(n) = 3n^2 + 2n + 3$. Find its asymptotic time complexity.
Answer:

$$f(n) = 3n^2 + 2n + 3$$
$$\leq 3n^2 + 2n^2 + 3n^2 \quad (n > 1)$$
$$\leq 8(n^2)$$

Hence, $f(n) = O(n^2)$. $n_0 = 1$
 $c = 8$

$8n^2$ is a strict upper bound on $f(n)$ as shown in the figure.

Mc
Graw
Hill
Education

17

Now, let us consider an example of the big O notation. So, let us look at $f(n)$ is equal to $3n^2 + 2n + 3$, and let us find its asymptotic time complexity. So, $f(n)$ is $3n^2 + 2n + 3$, which is less than or equal to $3n^2 + 2n^2 + 3n^2$ when $n > 1$. Well, $2n$ is less than $2n^2$ when n is greater than 1 and 3 is also less than $3n^2$ when n is greater than 1. If I sum of all of these terms $f(n)$ is less than or equal to $8n^2$. So, what can be said, is that $f(n)$ is actually order of n^2 , where the constant n_0 is equal to 1, and c is equal to 8. So, you plot the graph will notice in the interesting thing, that after this point 1 right, which is roughly be here. We can see that $f(n)$ is clearly greater than no. I am sorry $8n^2$. So, $8n^2$ is this green curve over here, and $f(n)$ is the red curve.

So, we can see that $8n^2$ is clearly greater at all points, as compared to the original function $f(n)$. So, we can say that $8n^2$ is the strict upper bound on $f(n)$ as shown in the figure. So, essentially the point is that we can have, in a any kind of functions, or any number of polynomial terms we want to find a simple upper bound, and ignore multiplicative in additive constants. So, the order notation pretty much gives us a mathematical tool of achieving that.

(Refer Slide Time: 30:33)

Big O Notation - II

Example:
 $f(n) = 0.0000n^{100} + 10000n^{99} + 234344$. Find its asymptotic time complexity.

Answer: $f(n) = O(n^{100})$

$f(n) \leq cn^m$
 $n > n_0$

$f(n) = O(n^m)$

* We shall use the asymptotic time complexity metric (big O notation) to characterize the time taken by different adders

$O(n)$ $O(n^2)$ $0.1n^2 + 30n + 20$ n_0, c $f(n)$ $cg(n)$

Mc Graw Hill Education

18

For example, let us consider this function; $10000n^{100} + 10000n^{99} + 234344$. So, it is true that you know this coefficient is very small, and this coefficient is very large, but definitely we can find a value of n , at which you know this term will dominate right. So, keeping that in mind we can use the same algebraic formula listed in the previous slides, and essentially say that $f(n)$ is order of n^{100} . So, a pretty much we have to take the highest exponent, while computing the order of a functions, say in this case is n to the power hundred.

Why is this the case, because we can always find. So, if this is $f(n)$ right, we can always find a constant c such that c is less n^{100} for certain n greater than n_0 right. So, basically we can always find something of that nature. So, one thing is if I set c as 10000 . So, $10000n^{100}$ is more than this term, but again these 2 terms will increase. So, that is if I keep on increasing the value of n , ultimately I will find out some value of n_0 and c ; such that this function is strictly less than cn^{100} for all values of n . So, hence we can say that $f(n)$ is equal to order of n to the power 100.

So, 1 simple ways is that if it is a polynomial expression, take the highest power of n right. So, that is the order of the function. So, we shall essentially use the O notation which is actually very easy, and in a, very easy to use, and very easy to compute to characterize the time taken by different types of adders. So, so let me you know talk

about some issues that students typically raise. So, students might say that look, we might have a function of this type let us say point $1 n^2 + 300n + 20$. And say in this case for a lot of values of n pretty much this term will dominate. So, the question is it still order n^2 , or is it order n , and how do we use it to compare.

So, the question is that. So, the big O notation is an approximate representation, is sort of telling you that a function for sufficiently large values of n is bounded by another function, after taking multiplicative constants into account right. So, basically it is all that its saying is a look if you have a function of maybe this type, then this is bounded by some other function of this type right. So, it is sort of a boundary of the function. So, this function, if this function is $f(n)$, this is something out the form $c \times g(n)$ right; that is what we are saying, and when you are saying that $f(n)$ is order of $g(n)$, this is exactly what we are trying to say that we are sort of creating an envelope for the functions; such that $f(n)$ is less the envelope at all points.

So, this has some value in the sense; of course, the value of n matters the multiplicative constant matters. So, essentially the value of n_0 and c definitely matter, either no doubt about that, but still it gives us a good idea and it is very unlikely, that in real life, if you have one function whose asymptotic complexity is higher than that of another function, then it will actually be faster in practice. So, that typically does not happen, in a sense and order n time algorithm or a circuit right, is typically much faster than order n^2 . Of course, n and there might be more constants in this equation which are getting ignored, but typically it is the case, that an order n time algorithm is much faster.

(Refer Slide Time: 35:16)

The slide features a blue header with the title "Ripple Carry Adders and Beyond". Below the header, there is a list of bullet points: "* Time complexity of a ripple carry adder :" followed by "* $O(n)$ ". A red circular icon with a white question mark is positioned to the left of the text "Can we do better than $O(n)$?". A blue speech bubble containing the word "Yes" points towards the question mark icon. A small cartoon fish is located to the right of the speech bubble. The McGraw Hill Education logo is in the bottom left corner, and the number "19" is in the bottom right corner.

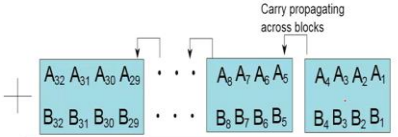
So, let us take a look at the complexity of a ripple carry adder. So, let us actually go back. So, at this point what we see is that the time complexity is t_h plus n times t_f . So, basically these t_h and t_f are constants. So, the only variable in here is n , and so we can say that this is equal to order of n , where n is the number of bits that we are trying to add. So, the complexity of a ripple carry adder does become order of n , in the sense that you know asymptotically, roughly we will take order in steps, if we have n bits.

So, if it is a 32 bit addition, roughly we will take you know some k times 32 steps, and if its 16 4 bit addition, roughly take c times 64 steps. So, the question is, can we do something faster, can we do something which is much better than order of n , that is the question that we need to ask. Well the answer is yes.

(Refer Slide Time: 36:29)

Carry Select Adder $O(\sqrt{n})$ time

- * Group bits into blocks of size (k)
- * If we are adding two 32 bit numbers A and B, and $k = 4$, then the blocks are :



- * Produce the result of each block with a **small ripple carry adder**

Mc
Graw
Hill
Education


20

So, let us look at a carry select adder, which takes order of square root of n time. So, let us do 1 thing, let us group bits into blocks of size k . So, if you are adding to 32 bit numbers a and b and k is equal to 4, then the blocks are. So, what will do is, we will group them into blocks. So, we will consider 4 bits at a time a_1 to a_4 b_1 to b_4 , then we will add them a carry will propagate across the blocks. So, may carry will propagate. And for each block of bits 4 and 4, will produce the result of each block with a small ripple carry adder. So, we will have separate ripple carry adders for each such small block. So, in this case what will happen is, that instead of treating adding a bit at a time, we will be adding 4 bits, entire block at a time.

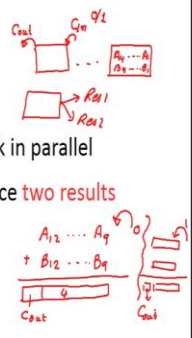
(Refer Slide Time: 37:21)


Carry Select Adder - II

- In this case, the **carry propagates across blocks**
- Time complexity is $O(n)$

 Idea :

- Add the numbers in each block in parallel
- Stage I : For each **block**, produce **two results**
 - Assuming an **input carry of 0**
 - Assuming an **input carry of 1**





21

So, the carry will then propagate across blocks, which still take order of n time, but let us come up with a novel idea of how to speed it up right. So, what was the original aim? The original aim was that from an order n time complexity, we want to move to order of square root of n time complexity. So, here is what we want to do. Let us do 1 thing. So, so it is considered the structure of the addition right. So, what are we done? We have essentially divided the numbers into blocks of 4 bits. So, let us consider 1 block. So, what is the inputs or what are the inputs to the block. The input to the block is the carry input from the previous block. What is the output from the block? The output from the block is the carry from the block right. So, these are the 2 inputs and outputs, there is a carry in, and there is a carry out. If we know the value of the carry in, we can perform the addition correctly.

So, let us consider any block. So, what are the possible values of carry in, they are either 0 or 1 right, only 2 values or the carrying in can be there. So, what we can do. So, here is a brilliant idea, behind the carry select adder right. So, this is very important. So, what we can do is that for each block we can actually produce 2 results right. So, when you are doing addition, you can actually produce two results; result 1 and result 2. So, basically when we are adding 2 numbers. So, let us say you are adding you know 2 blocks of number from a 4 to, let say. Well a 4 might not be the right example, let me just, may be deleted.

Let us say that we are trying to add a 12 plus. So, for this particular block the value of the carry input, can either be 0 or 1 right. So, we do not know that. So, let us first assume the carry input is 0 and let us produce an output. So, the output will have 5 bits; 4 of them are sum bits and 1 of them is the carry out then in parallel what we can. So, in the since it is in hardware right, we can do a lot of competitions that are in parallel right. So, there is no need of doing one after the other. In parallel what we can do is that we can perform another addition, for the 2 blocks, assuming that the carry in was actually 1. So, that will also produce its result and a carry out all right.

So, what we can do is that for each block we produce 2 results. We assume an input carry of 0 into the blocks. So, we produce one result and we produce one more result to assume an input carry of 1, both of these since it is an hardware, both of these can happen in parallel right. So, for what we can do, is that for each block. So, assume that we are adding 2 32 bit numbers, and we have blocks of size 4.

(Refer Slide Time: 41:25)

Carry Select Adder – Stage II

- * For each block we have two results available
- * Result \rightarrow (k sum bits), and 1 carry out bit n/x
- * Stage II
 - * Start at the least significant block
 - * The input carry is 0
 - * Choose the appropriate result from stage I
 - * We now know the input carry for the second block
 - * Choose the appropriate result
 - * Result contains the input carry for the third block

Mc Graw Hill Education

22

So, we will have essentially 8 pairs, where 1 8 blocks for 1 block we will have 2 pairs of 4 bit numbers right. So, for each block we actually produce two results; the first result assumes that the input carry coming from the previous block is 0, and a second result assumes the input carry coming from the previous block is 1. The result will have the sum bits and a carry out bit.


Now, So, what we have, for let us say 2 32 bit numbers that we are trying to add a plus b, we have a sequence of 8 blocks right, where each block has 4 bits. So, we can generalized it let say each blocks is k bits and that is also fine, and their mostly equal size rectangle side and. So, then essentially what we can do at this point. So, s in stage 1 what have we done. What we have done, is that we have for each block we have done the addition, assuming an input carry of 0 and an input carry of 1. So, we have two results. In stages two what we do is we start at the least significant block. So, clearly the input carry into this is 0. So, then we quickly choose the appropriate result from the first stage, and again sent. So, the result will have the carry out. So, we again send the carry to the next block.

Then we quickly choose the appropriate result from the first stage and send the, you know the appropriate carry out to the next block and so on and so forth. So, how long does this process take? So, let us first find out how long this stage 1 take, and how long does stage 2 take. So, let us assume that each block has k bits. So, we are essentially doing the editions in all the blocks in parallel, you know mainly, because there is no carry business right. We are assuming a carry of both 0 and 1, and each block, since it is k bits we can use a ripple carry adder. So, that will take order of k time.

And in this case what we do is we start at 1 block. So, we quickly decide which output it is, we send the carry, let us say the carry here is 1 then in the previous stage we have already computer 2 results. So, we choose that result which corresponds to an input carry of 1, and we quickly propagate the output carry again and again and again. So, we can say that in each stage it takes roughly take constant time, which is ordered 1, and since there are in a each blocks k bits right. So, basically there are n by k blocks. So, we can say that the entire operation takes order of n by k time units all right.

(Refer Slide Time: 44:38)

Carry Select Adder – Stage II

- * Given the result of the second block
 - * Compute the carry in for the third block 
 - * Choose the appropriate result
- * Proceed till the last block
- * At the last block (most significant positions)
 - * Choose the correct result
 - * The carry out value, is equal to the carry out of the entire computation.

23


So, what I can let me just summarize what we do in stage 2. Given the result of this let us an n th block, in this case of second block. We compute the carry in for the third block which is very easy; we just take the carry that is coming in. So, let say for this second block 1 in a block number 2, we know the carry in, since we have the pre computed results and immediately find what is the carry out, and then this goes to the next block the third block. And since again, we know what is the carry in we can choose from one among the two precomputed results, and send the carry out and so forth. So, in this manner we proceed till the last block.

(Refer Slide Time: 45:37)

How much time did we take ?

- * Our block size is k
 - * Stage I takes k units of time
- * There are n/k blocks $k \rightarrow$ block size
 - * Stage II takes (n/k) units of time $n \rightarrow$ # of bits
- * Total time : $(k + n/k)$ $= \sqrt{n} + n/\sqrt{n} = 2\sqrt{n}$

32



$$\frac{\partial (k + n/k)}{\partial k} = 0 \quad O(\sqrt{n})$$

$$\Rightarrow 1 - \frac{n}{k^2} = 0$$

$$\Rightarrow k = \sqrt{n}$$

24

At the last block. We choose the correct result, and basically this finishes the entire competition. So, how much time did we take in doing this, right. So, since a block size is k as I said, stage 1 roughly takes order k units of time and. So, this can be k times some constant it does not matter. So, that is the reason I have removed it for the sake of simplicity, but the reader is encourage to assume it takes c times, c 1 times c 1 times k time and. So, the constants do not matter they will get factor out in our equation. So, for the sake of simplicity let us assume that stage 1 take k units of time.

Since there are n by k blocks, stage 2 will take n by k units of time. So, total time is k plus n by k . So, what is k ? k is the block size and number of bits in 1 blocks. So, let me write it down; k is the block size and n is the number of bits that we are trying to add. So, n is clearly cannot be changed, once the problem is given to you, but the block size can be changed. So, let us do one thing, for finding out the minimum of this function, it is differentiate it with respect to k . So, whenever the derivative is 0, we will be able to find maxima or minima.

So, in this case you want to minimize the time. So, we want minima; let us differentiate it with respect to k . So, it is because 1 minus 1 by k square and. So, the optimal value of k for which the total time is minimized. It is actually square root of n it is a very important point to take into account, which means that let us say if you have a 32 bit addition right, we are adding to 32 bit numbers. So, square root of 32 is between 5 and 6. So, any of the numbers are fine. So, what we essentially need to do is that we need to create 5 bit blocks of numbers. So, let us say a 5 to a 1. Similarly a 6 to a 10, and similarly you know we need to add these blocks and use a ripple carry adder, I am sorry use a carry select adder to add the squares of blocks. So, that will give us the fastest possible time, and what is the fastest possible time. So, it is essentially k plus n by k when k is equal to square root of n becomes square root of n plus n divided by square root of n which is two times square root of n , which can be written in an approximate notation on order of square root of n , because all additive and multiplicative constants are ignored.

(Refer Slide Time: 48:39)

The slide features a blue header with the title "Time Complexity of the Carry Select Adder". Below the title, there are two bullet points: "* $T = O(\sqrt{n} + \sqrt{n}) = O(\sqrt{n})$ " and "* Thus, we have a \sqrt{n} time adder". To the right of the first bullet point, there is a handwritten note in red: " ~~$O(n)$~~ $O(\sqrt{n})$ ". Below the bullet points, there is a red question mark icon. To the right of the question mark, there are two lines of handwritten text in red: "Ripple Carry Adder: $O(n)$ " and "Carry Select Adder: $O(\sqrt{n})$ ". Below this text, the question "Can we do better?" is written. A blue thought bubble containing the word "Yes" is shown above a cartoon fish. The McGraw Hill Education logo is in the bottom left corner, and the number "25" is in the bottom right corner.

So, pretty much what we have achieved in our carry select adder, is that we have created an adder that takes order of square root of n time, and so which is fast we thus have our root n time adder. So, basically how fast is a ripple carry adder? The ripple carry adder which was very simple took order of n time, and the carry select adder takes order of square root of n times. So, it is expected to be much faster. So, mind you as I said, it is not necessary that for relatively small values of n , any functions which is order of n is actually slower, in the sense I can always give a pathological example, for I am sorry. I can always give a pathological example that consider this function, which is an order of n versus consider this function. The most values of n actually this function is smaller, but this is a very pathological example it is typically not the case.

So, even if I encourage students to actually implement the carry select adder in hardware and see for themselves find out for themselves; that it is genuinely faster than the ripple carry adder. So, I mean asymptotic time complexity definitely gives a broad direction that yes maybe it is faster, but also you can implement it in hardware, and see that it is faster, and definitely as you increase the number of bits 64, 128 and so on, a carry select adder is definitely much faster, as can be seen from the asymptotic complexity as well.

Now, let us ask a magic fish, can we do better. Well magic fish says yes, we can do better, we can do much better. In fact, so, how much better can we do that is the

important question that we need to answer. Let us now consider the carry look ahead adder which is much faster. So, we will roughly do it in order of logging time, which is significantly faster as compared to n and square root of n .

(Refer Slide Time: 51:13)

The slide features a blue header with the title "Carry Lookahead Adder ($O(\log n)$)". Below the title, there are three bullet points: the first states "The main problem in addition is the carry", the second says "If we have a mechanism to compute the carry quickly, we are done", and the third says "Let us thus focus on computing the carry without actually performing an addition". A diagram shows three square blocks in a row, with red arrows indicating carry propagation from left to right. The first arrow is labeled c_i and the second is labeled c_{i+1} . To the right of the blocks, the expression $O(\sqrt{n})$ is written. The slide also includes the McGraw Hill Education logo in the bottom left corner and a small number "26" in the bottom right corner.

So, let us first take a look inside. So, since you know, since this adder is much faster, and also this is the adder that is used in most commercial processors, it is also far more complicated. So, we need to take it slow.

So, let us first take a look at the main insight. So, the main problem in addition is it is a carry right. So, the carry is the one that goes from 1 block to the other and that is essentially the sequential bottleneck. Otherwise in hardware if we can do things in parallel, then things will be much faster right. So, if you can. So, if you take a look. So, basically where did the speed up in the carry select adder come from? It came from the fact that in stage 1, we can treat each block separately in parallel right.

So, essentially we can treat all the blocks simultaneously. we assume we do not know what is the carry, but we produce two results, so we do extra work, and we produce two results for input carry of 0 and 1, so that is stage 1. Then in stage 2 the carry quickly ripples and instead of rippling across bits it ripples across blocks. So, it takes biggest ripples.

And we quickly choose from 1 of the 2 pre computed outputs. So, essentially we get a time for complexity which is k plus and by k and we differentiate it and we find the optimal value of k to be square root of n , and that is the total time is order of square root n , but essentially all are benefits came, from computing the carry very quickly. So, let us thus focus on computing the carry, without actually performing the addition if required, some kind of a shortcut to get rid of the carry business.

(Refer Slide Time: 53:18)

Generate and Propagate Functions

A
+B

- Let us consider two corresponding bits of A and B
 - A_i and B_i
- Generate function** : A new carry is **generated** ($C_{out} = 1$)
- Propagate function** : $C_{out} = C_{in}$
- Generate and Propagate Functions are :

C_{in}	0	0	1	1
A_i	0	0	1	1
B_i	0	1	0	1
C_{out}	0	0	1	1

$G_i = A_i \cdot B_i$
 $P_i = A_i \oplus B_i$
 $C_{out} = C_{in}$

$\times \oplus \oplus$
 $A_i \ B_i \ A_i \oplus B_i$
 0 1 1
 1 0 1

McGraw Hill Education

So, let us do one thing. So, we need to define a little bit of algebra before we actually proceed. So, let us consider 2 corresponding bits of a and b . So, a and b are the two numbers that we are trying to add, we are trying to add a plus b and both of them are n bit numbers. So, let us define a generate function, a generate function means that a carry is generated after the addition. So, C_{out} is equal to 1. On similar lines, let us define a propagate function. So, basically if I consider a block of bits irrespective of the input carry, irrespective of the input carry we are generating an output carry, we say that the block is generating a carry.

For example let me consider this block $1\ 1$ plus $1\ 1$ irrespective. So, if I irrespective of the import carry, we will always generate an output carry. So, input carry is 0. So, 1 plus 1 will be 0 , and this will be 1 and output carrier of 1 will be generated. Let us assume that input carry is 1 . So, 1 plus 1 plus 1 is 1 carry of 1 again 1 and I carry out of 1 . So, if

this is the contents of the block, irrespective the input carry, the output carry will always be 1. So, we are saying that we are generating.

Now, let us define a propagate function. So, the propagate function says that the output carry is equal to the input carry. So, let us consider the example. So, let us consider single bit example, which will be far easier to understand. So, let us assume. So, let us considered the values of a_i and b_i . So, let us consider a multiple values. So, let us assume these are 0 and 0. If they are 0 and 0 and let us consider the carry, irrespective of the value. So, let us consider a carry in, irrespective of the value of the carry in, the carry out will be 0 right. So, even if the carry in is 1 we will still not generate a carry, if its 0 still not generate a carry.

Now, let us assume, now assume that a_i is 0 and b_i is 1 right. So, in this case, it is, in this case let us see. So, let me may be draw a quick line here, to remark it. If the input carry is 0, then the output carry is 0, but if the input carry is 1, then we are adding 1 plus 0 plus 1 which is 1 0 right. So, the output carry is 1.

So, what we can clearly see is that output carry is the same as the input carry. Now let us consider one more example. In this case let a_i be 1 let b_i be 0, and let us assume that input carry is 0. The input carry is 0 we are adding 1 plus 0 plus 0, no carry is being generated, so the output is output carry out is 0. If however, the input carry is 1 then we are adding 1 plus 1, the sum is 0, the carry out will be equal to 1. Hence the output carry will be equal to 1. So, in both of these cases, the carry out is equal to the carry in. in the first case irrespective of the carry in the carry out in 0. So, let us consider the last case. In this case a_i is 1 b_i is 1, irrespective of the input carry. So, let us assume that the input carry is 0. So, then we have 0 plus 1 plus 1 carry is generated.

Let us assume carry in as 1. So, we add 1 plus 1 plus 1, again a carry out is generated. So, irrespective of the input, irrespective of C_{in} C_{out} is always 1. So, what we can say, is that if a_i and b_i both of them are 1, which is when a_i and b_i is true. We can say that this combination of bit of this block right, which has 1 bit of a and 1 bit of b , is generating a carry. So, when do you generate a carry? You generate a carry when a irrespective of the input carry, you always generate an output carry that will happen if a_i is 1 and b_i is 1. So, I can define a function g_i which is a_i and b_i . So, if g_i is true, means that a carry always generated.

Now, let us consider this case, when a_i is 0 b_i is 1 or a_i is 1 and b_i is 0. So, how can I represent this case? The way that this case can be represented is, by a_i exclusive or b_i . So, if you go back to our discussions in chapter 2, the XOR function. So, let me just in are do the quick recap of the, XOR function if anyway seen that many times. So, the XOR function is 1 when a_i is 0 b_i is 1 or a_i is 1 b_i is 0 that is when a_i exclusive or b_i is 1, in the remaining cases it is 0. So, when a_i exclusive or b_i is true, we can say that the carry is being propagated.

What is propagation mean? Propagation means that the carry output is equal to the carry input. So, if carry in is 0. So, then carry out C_{out} is 0, if C_{in} is 1 then C_{out} is 1. So, that is when the propagate function will be true, and if both a_i and b_i are 0 then irrespective of the carry in, the carry out will be 0, in a sense the carry is being absorbed. So, what is that take away point from these slides? The take away point from this slide is the generate function that we have defined which is a_i and b_i , means irrespective of the carry in and carry out is always generated and p_i which is propagate i which means that this function, if this function is true it means that the carry out C_{out} is equal to c_n . So, keeping those in mind let us move forward.

(Refer Slide Time: 60:28)

Using the G and P Functions

- * If we have the **generate** and **propagate** values for a bit pair, we can determine the **carry out**

$$C_{out} = g_i + p_i \cdot C_{in}$$

McGraw Hill Education

So, let us use the G and P generate and propagate functions for a bit pair. So, the carry out, can we said it is equal to g_i . So, plus means, plus here means an or and dot means an and. So, this is. So, if g_i is true then a carry out is definitely being generated.

Otherwise or, the carry input is true and the propagated is true. So, there are only two ways that C out can be 1, either the block is generating at carry or the block is propagating a carry and a carry input is 1. So, we can say it is g i or p i and carry input.

(Refer Slide Time: 61:15)

Example

Example:
Let $A_i = 0, B_i = 1$. Let the input carry be C_{in} . Compute g_i, p_i , and C_{out} .

Answer:

$$g_i = A_i \cdot B_i = 0 \cdot 1 = 0$$

$$p_i = A_i \oplus B_i = 0 \oplus 1 = 1$$

$$C_{out} = g_i + p_i \cdot C_{in} = C_{in}$$

29

So, let us consider an example let us say a i is 0 b i is 1, let the input carry be c n compute g i p i and c out. So, g i is a i and b i which is 0 and 1 which is 0. So, this is not generating, p i is b i XOR a i which is 0 XOR 1 which is 1. So, it is propagating. So, C out is g i plus p i and carry input. So, this is 0, it is propagating, and we do not know the input carry. So, we can say C out is equal to C in this case.

(Refer Slide Time: 61:55)

G and P for Multi-bit Systems

- * $C_{out}^i \rightarrow$ output carry for i^{th} bit pair
- * $C_{in}^i \rightarrow$ input carry for i^{th} bit pair
- * $g_i \rightarrow$ generate value for i^{th} bit pair
- * $p_i \rightarrow$ propagate value for i^{th} bit pair

So, let us now consider the multi bit system where a block does not have 1 bit each of a and b, but it has multiple bits. So, let us define some simple terminology first. So, let us say that C_{out}^i means output carry for the i^{th} bit pair. Similarly let us say C_{in}^i is the input carry for the i^{th} bit pair. Similarly g_i is a generate value is the generate function for the i^{th} bit pair, and p_i is the value of a propagate function for the i^{th} bit pair; a bit pair is a pair of bits, so when you are adding we having a multiple bit of a and multiple bits of b right. So, one corresponding pair of bits is called a bit pair. So, now, let us write some basic equations for multi bit systems and let us see what it looks like.

(Refer Slide Time: 62:45)

G and P for Multibit Systems - II

$$C_{out}^1 = g_1 + p_1 \cdot C_{in}^1$$

$$C_{out}^2 = g_2 + p_2 \cdot C_{out}^1$$

$$= g_2 + p_2 \cdot (g_1 + p_1 \cdot C_{in}^1)$$

$$= \underbrace{(g_2 + p_2 \cdot g_1)}_G + \underbrace{p_2 \cdot p_1 \cdot C_{in}^1}_P$$

$$C_{out}^3 = g_3 + p_3 \cdot C_{out}^2$$

$$= g_3 + p_3 \cdot ((g_2 + p_2 \cdot g_1) + p_2 \cdot p_1 \cdot C_{in}^1)$$

$$= \underbrace{(g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1)}_G + \underbrace{p_3 \cdot p_2 \cdot p_1 \cdot C_{in}^1}_P$$

So, let us see c we need, we have already derived this that for the first bit pairs $C_{out 1}$, it is g_1 plus p_1 times the carry that is. So, it may not times p_1 and the carry that I coming in which is $C_{in 1}$. Now let us consider the second bit pair which is a basically. Let us consider the 2 bits $a_2 a_1$ and $b_2 b_1$. So, basically for the first bit pair this is the equation right this is the equation for the first bit pair. If I consider the second bit pair over here, the carry that is coming out $C_{out 2}$ is g_2 or p_2 and the carry that is coming out of the first bit pair. So, then I can do a little bit of algebra and replace $C_{out 1}$ with this expression here, which comes to here and I just rearrange the terms.

So, what I see, is that for finally, I have $C_{out 2}$ which is the carry which is coming out from this c_{out} , and the carry which is going in which is $C_{in 1}$. So, in this way I finally, have an equation in terms of $C_{in 1}$, and generate and propagate functions of the pair of bit. So, so the way that the generate function for essentially the pair of bits right. I can think of this as you know 1 big block now right. So, for this block the generate function is actually g_2 or p_2 and g_1 and the propagate function is p_1 and p_2 with basically means, that the input carry will show up in output if every pair of bits is propagating it; that is the reason for the and function.


And a nice way of understanding regenerate is that, either the last pair the second pair is generating or the first pair is generating, and second pair is propagating right. So, this can be considered as generate for the 2 bit pair and this has propagate. So, let us consider a 3 bit system. So, we can write a similar equation, that either C_{out} for at the end of 3 bits, either the third bit pair is generating, or the third bit pair is propagating, the carry generated out of the second addition out of the addition, of the first 2 bits. So, then here for this equation, we can substitute this result after doing some algebra, we again come at this. So, this is the generate function for a 3 bit system. So, when will a 3 bit block generate, then either the third bit pair is generating or the second bit pair is generating, and the third bit pair is propagating or the second and third bit pairs are propagating, and the first bit pair is generating. And when will you propagate, it will propagate, when all the 3 bit pairs $p_1 p_2$ and p_3 all of them are propagating. So, the input carry will reflect in the output.

(Refer Slide Time: 66:24)

G and P for multibit Systems - III

$$\begin{aligned}
 C_{out}^4 &= g_4 + p_4 \cdot C_{out}^3 \\
 &= g_4 + p_4 \cdot (g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1) + p_3 \cdot p_2 \cdot p_1 \cdot C_{in}^1 \\
 &= \underbrace{(g_4 + p_4 \cdot g_3 + p_4 \cdot p_3 \cdot g_2 + p_4 \cdot p_3 \cdot p_2 \cdot g_1)}_G + p_4 \cdot p_3 \cdot p_2 \cdot p_1 \cdot C_{in}^1
 \end{aligned}$$

$$\begin{array}{r}
 A_4 \ A_3 \ A_2 \ A_1 \\
 + B_4 \ B_3 \ B_2 \ B_1 \\
 \hline
 \end{array}$$



32


So, similarly we can have much more complicated algebra, but all that you want you can see, is that all of these are essentially following the same pattern, exactly the same pattern. So, for a 4 bit system the generate function looks something like this, the pattern is the same. So, for a 4 bit block, let us assume a 4, let me write A 4, A 3, A 2 and A 1 being added to, so when will this actually generate, it will generate when either you know this bit pair over here.

Let me use a laser pointer laser. So, when this bit pair over here is generating, which we see over here, or this bit pair is propagating, and this bit pair is generating, or these two are propagating, and the carry is being generated here, or all of these 3 are generating and a carry is being generated over here. When will carry propagate across this block, and every single bit pair p 1 and p 2 and p 3 and p 4 all of them are propagating the carry. So, in the same way for an n bit system, we can create a generate function and the propagate function.

(Refer Slide Time: 67:48)

Patterns

1 bit	$C_{out}^1 = \underbrace{g_1}_{G_1} + \underbrace{p_1}_{P_1} \cdot C_{in}^1$
2 bit	$C_{out}^2 = \underbrace{g_2 + p_2 \cdot g_1}_{G_2} + \underbrace{p_2 \cdot p_1}_{P_2} \cdot C_{in}^1$
3 bit	$C_{out}^3 = \underbrace{g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1}_{G_3} + \underbrace{p_3 \cdot p_2 \cdot p_1}_{P_3} \cdot C_{in}^1$
4 bit	$C_{out}^4 = \underbrace{g_4 + p_4 \cdot g_3 + p_4 \cdot p_3 \cdot g_2 + p_4 \cdot p_3 \cdot p_2 \cdot g_1}_{G_4} + \underbrace{p_4 \cdot p_3 \cdot p_2 \cdot p_1}_{P_4} \cdot C_{in}^1$
n bit	$C_{out}^n = G_n + P_n \cdot C_{in}^1$

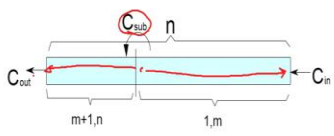


So, let us take a look at the pattern. So, coming to the pattern here for a single bit block C_{out} is g_1 plus g_1 or p_1 , and carry input where is a simple functions for a 2 bit system for a 2 block. We can see the generate function is slightly more complicated, the logic has been explained, and the propagate function is p_1 and p_2 , which means both the bit pairs need to propagate. So, we just keep going on and on. And for a 4 bit system, we can clearly see the generate function is more complicated, in a sense either the fourth bit pair is generating or the fourth bit pair is propagating, and a third bit pair is regenerating and so on. And when does a 4 bit system actually propagate, when all the bit pairs are propagating, the carry input carry to the output. Similarly for an n bit system, we can create a generate function, and a propagate function, and given the input carry, we can find out the output carry.

(Refer Slide Time: 69:02)

Computing G and P Quickly

- * Let us divide a block of n bits into two parts



- * Let the carry out and carry in be : C_{out} and C_{in}
- * We want to find the relationship between
 - * $G_{1,n}, P_{1,n}$ and $(G_{m+1,n}, G_{1,m}, P_{m+1,n}, P_{1,m})$

Mc
Graw
Hill
Education

34

So, now, the idea is, can we compute the, generate and propagate functions quickly. So, let us look at some other results for the G n P functions. So, let us do 1 thing, let us divide a block of n bits into two parts. So, let us consider this block of n bits. So, we divided into 1 parts, is 1 to m and the other is m plus, m plus 1 m bits here and n plus 1 to n bits over here. Let the carry out and the carry in to this block, we carry in here and the carry out here. So, we want to find the relationship between the generate functions of the entire n bit block, the propagate function for the entire bit block, and generate and propagate functions for this sub blocks. So, generate from 1 to m propagate from 1 to m , generate from m plus 1 to 1, and propagate from m plus 1 to n , or what do they exactly look like.

(Refer Slide Time: 70:10)

Computing G and P Quickly - II

$$\begin{aligned}
 C_{out} &= G_{m+1,n} + P_{m+1,n} \cdot C_{sub} \\
 &= G_{m+1,n} \\
 &+ P_{m+1,n} \cdot (G_{1,m} + P_{1,m} \cdot C_{in}) \\
 &= \underbrace{G_{m+1,n}}_{G_{1,n}} + \underbrace{P_{m+1,n} \cdot G_{1,m}}_{P_{1,n}} + \underbrace{P_{m+1,n} \cdot P_{1,m}}_{P_{1,n}} \cdot C_{in}
 \end{aligned}$$

$G_{1,n} = G_{m+1,n} + P_{m+1,n} \cdot G_{1,m}$

$P_{1,n} = P_{m+1,n} \cdot P_{1,m}$

35

So, let us take a look at this equation, and try to work it out. So, in this equation let me. So, the carry out is essentially equal. So, so let us do 1 thing let us call the carry which is propagating from, let us say the sub block of the first m bits to the remaining n minus m bits. So, c sub right let us refer to this carry a c sub. So, a final C out the carry out is a generator of m plus 1 to n. So, this block final carry out what, how will it, how will we compute, either this simple block over here of n minus m bits, either this is generating in that case. We will have a carry out or it is propagating the carry that has come from its right, which is c sub that is the only two ways in which we can get carry out. So, carry out will be $G_{m+1,n}$ or $P_{m+1,n} \cdot C_{sub}$ for those m n minus m bits and did with c sub. So, either you are generating, or you are propagating right, there is no other way to generate a carry.

So, there is no other way if I carry out to be one. So, you do a little bit of algebra we will keep $G_{m+1,n}$ and this and the c. So, so what is c sub? c sub is basically the carry out of this other block over here, consisting of the first m bits. So, we can write that as. So, when will its carry out be true, when either it is generating a carry $G_{1,m}$, or it is propagating the carry input and. So, which is $P_{1,m}$ ended with carry in, if I do a little bit of algebra, and rearranging the terms what we see, is that the generate function from 1 to n will essentially be the generate function of the first n minus m bits or the propagate functions value from for the first n minus m bits, and the generate function the first 1 to m bits, which means that either the first 1 to m bits generating a carry in the carry is

being propagated through $n - m$ bits or the $n - m$ bits are generating a carry, and the carry is being reflected in, the carry out that is the only two ways. So, what we can write, ultimately C_{out} is $g_{1:n}$ plus or $p_{1:n}$ and c_{input} . So, the generate function for the n bits system can be represented as a function of generate and propagate functions of the first m bits in the remaining $n - m$ bit.

So, the important equations that we get from this particular section over here, is that generate 1 to n . So, basically the first 1 to n bits, considered a block of the first 1 to n bits, when will it actually generate a carry it will generate a carry, if the bits from $m + 1$ to n , you know the right most sub block if that is generating a carry or if the, or you know, if this sub block over here from the last $n - m$ bits if that sub blocks is propagating, and the other sub block consisting of the first n bits, is actually generating a carry. So, that is pretty much the two ways in which a carry can be generated from the n bit blocks.

So, let me explain this in the different way. So, let me consider n bits we are dividing it into two sub blocks. So, 1 sub block consists of the bits from $m + 1$ to n , and the other sub block consists of the bits of from 1 to n , we have a carry input and we have a carry output. So, let us try to understand this particular equation slightly intuitively. So, when will the carry output, to be actually 1 the carry output, will be 1 if the entire block of n bits, either generates a carry, which means irrespective of the carry input the carry output will be 1 right.

Or the entire sub block or the entire block from 1 to n propagates the input carry and input carry if it is 1 carry out will be 1 . So, basically for the generate function of 1 to n bits. So, when will this n bit block, actually generate a carry it will generate a carry, if let us see this block, let us call this left block, and let us call it right. So, when the left block over here, is generating a carry or the left block from $m + 1$ to n is propagating a carry, and a right block is generating a carry.

So, in this case a carry will be generated, and get propagated till the end. Now when is the n bit block actually propagating a carry? It is propagating a carry when. So, essentially carry comes here, I will sort of gets propagated till the end. This will happen if both the blocks the left 1 and the right 1 both of them are propagating a carry. So, the propagate function will $p_{m+1:n}$ ended with $p_{1:m}$. So, which is a similar

logic that we used, when we are writing equations for multiple systems and. So, this is also 1 way of writing the equation. So, we shall find it very handy when we discussed a carry look ahead adder, but the important take home point over here, is that a carry out is generated only under two circumstances. The circumstances is that either the n bit block generates a carry which means irrespective of the input carry the output will always be 1, or when it propagates a carry which means that the input carry is 1 output carry will be 1, and if the input carry is 0 the output carry is 0.

So, the problem that we wanted to solve over here is that given an n bit block can be represented. Can we represent it is generate and propagate function in terms of generate and propagate functions of the different sub blocks. So, the left one and the right one, we were able to do that, and these are the two equations to do that. So, the take home point here, is that the generate function for the entire block is essentially the generate function of the n minus m left bits or with the propagate function of the n minus m left bits with the generate function of the m right bits, and the propagate function for the entire n bit block is essentially, the end of the propagate functions of the individual sub blocks.

(Refer Slide Time: 78:11)

Insight into Computing G and P quickly

- * **Insight :**
 - * We can compute G and P for a large block
 - * By first computing G and P for smaller sub-blocks
 - * And, then **combining the solutions** to find the value of G and P for the larger block
 - * Fast algorithm to compute G and P
 - * Use **divide-and-conquer**
 - * Compute G and P functions in $O(\log(n))$ time

McGraw Hill Education

36


Now, let us give an insight of computing generate and propagate functions quickly. So, the idea is that we can compute G and P for a large block by first computing generate and propagate function for smaller sub blocks, then we can combine them using these equation two actually. So, you know if you see this diagram we have generate and

propagate functions for smaller sub blocks, and these have been combine to create, the generate n functions for a larger blocks. And once we have that we can, let me just clean the screen of annotations. Once again we can then use this equation to sort of given the carry in compute the carry out right.

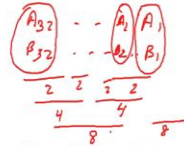
So, what is the idea, we take smaller sub blocks compute the G and P function, we combine the solutions for the smaller sub blocks to find generate and propagate function for a larger block and so on and so forth. So, fast algorithm to compute G and P will have to use some sort of a, what is called a divide and conquer strategy, which means, first solve the problems for very small blocks and gradually combine it to get the solutions for bigger and bigger blocks right. So, basically this we want to do in order login time, let us see how.

(Refer Slide Time: 79:47)


Carry Lookahead Adder – Stage I



- * Compute G and P functions for all the blocks
- * Combine the solutions to find G and P functions for sets of 2 blocks (or 2 bit pairs)
- * Combine the solutions to find G and P functions for sets of 4 blocks bit pairs
- * ...
- * ...
- * Find the G and P functions for a block of size : 32 bits



$$G_{32} \quad P_{32} \quad [C_{out} = G_{32} + P_{32} \cdot C_{in}]$$


37

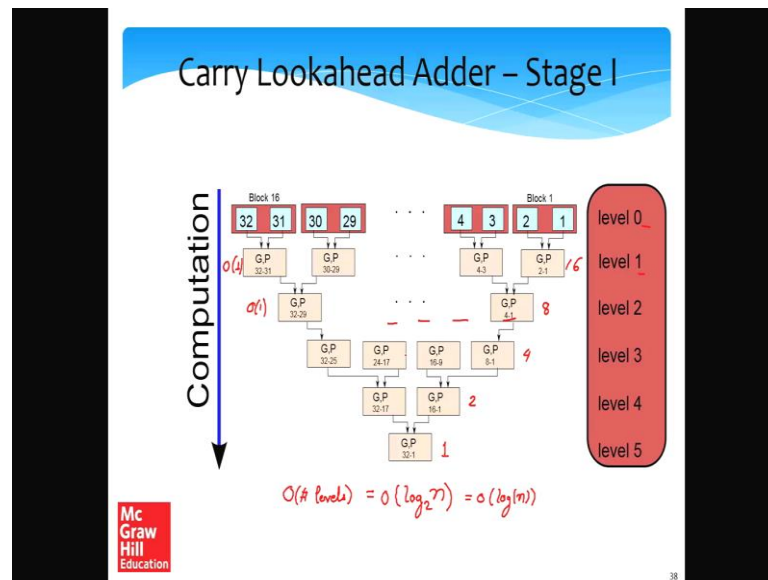
So, the carry look ahead adder is similar to the carry select adder in the sense there are 2 stages. So, in stage 1, we compute the generate and propagate functions for all the block, how do we do it? We combine; first we find the solutions to for G n P functions of a single. All the single bit pairs we combine the solutions to find G n P functions for sets of to block, then we combine the solutions to find G n P functions for sets of 4 blocks and so on. So, so basically you know two blocks means two blocks of bits. So, basically what we do let me explain this, if you have a and b and a contains 32 bits b contains 32 bits. So, let us contain considered each bit pair.

So, what we do, is initially we compute the generate and propagate functions for each bit pair, and then what we do, is that we sort of multi bit system, where we computed for 2 blocks right, 2 blocks are, let us it or 2 bit pairs. And similarly we combine solutions to find G n P functions for sets of 4 bit pairs. So, I should mainly make it bit pairs, it is slightly easier to understand. So, similarly we just keep on increasing the size of each block the number of bit pairs on each block and. So, finally, so what do we do? So, we start out with assuming that a has 32 bits and b has 32 bits. So, we start out in a parallel fashion by computing the G n P functions where each pair of bits.

So, at this point each block contains 1 pair of bits, then what we do is, that we combine you know for let us say $A_1 A_2 A_1 B_1$ and $A_2 B_2$ we combine the solutions to get the solution for a 2 bit system, and we take all the solutions for 2 bit blocks, and we get solutions for 4 bit blocks right. solutions means G n P functions, then we combine the solutions for 4 bit blocks, and then you get for 8 bits, then from 8 and 8 16, and finally, 16 and 16 32. So, you find the G n P functions for a block size of 32 bits, at the end right a block of size 32 bits means 32 bit pairs. So, also in a I would instead I have writing blokes here I would rather say you know for 2 bit pairs and 4 bit pairs. So, basically at the end of this exercise, what is it that we have we essentially have G 32 and P 32.

So, what is the final carry out? The final carry out of the 32 bit addition is G 32 ORed with P 32, and the carry in if there is a carry in right. So, so that is the simple equation that we have and. So, this is essentially. So, what do we get in stage 1. What we get in stage 1, is that we sort of create hierarchy, and we consider larger and larger block sizes and we compute the G n P functions.

(Refer Slide Time: 83:49)



So, this is shown in the diagram. So, in this diagram what we do is that. So, so let us do 1 thing. So, let us say let us number the bit pairs first bit pair second third fourth and so on.

So, let us say that in level 0, we considered 2 bit pairs, it can be done. We have the solutions for a 2 bit system. So, we compute the G n P functions for the first 2 bit pairs, which is 1 to 2 3 to 4 till twenty 9 to 30 and 31 to 32 for this is level 0. This is level 1, and the next level what we do is, we compute the G n P functions for a 4 bit system, for 4 bit pairs from 1 to 4. Similarly another groups of 4 and so on. So, at this point, so we actually have 16 you know we have 16 blocks at this point, we have 8 blocks, then what we do, is we take those 8 blocks and coalesce them into 4 blocks for completing the G n P functions from bit pairs 1 to 8 G P functions, from 9 to 16 G P functions, from 17 to 24 and 25 to 32. And then we again quails them to find the G P functions from bit pairs 1 to 16 and G P functions from bit for bit 17 to 32.

And then finally, we compute the G P function for the entire block of 32 bits right, which is from 1 to 32. So, basically there are two important points that we need to understand here. So, this thing can be done, and mind you in each level all of these computations are happening in parallel right. They are not happening in series. So, hardware by definition is parallel it is not in series. So, pretty much in level one. So, we will have many of these hardware circuits that will be computing the values of the G P functions for, you know for the bit pairs and parallel. So, this all of these things will happen simultaneously, in

the same time can happen. In the same time there is no dependency per se between these circuits and it will be 16 of such circuits.

So, for example, the G P circuit here will compute the G n P functions for the first 2 bit pairs this for the third and fourth bit pairs, and then we gradually, we shall you know we shall merge. So, the G P function here will take the G P functions of this unit and this unit, and merge them as per this equation right, can easily be done as for these two right. So, what needs to be done, and you need an or gate and a NAND gate very simple and in this case you need only and gate. So, so these are simple competition. Similarly this block will take the inputs from this circuit will take an input from this circuit, and another circuit here, and we will just keep on doing this. And at every point for the propagate function we need to do an AND AND for the generate function, we need to a single or and a single AND.

So, what we can say is that, for each level the time that it takes is the time that any one of the circuit takes, because all of them are working in parallel, and they take roughly order of 1 time, which means constant time. So, this also takes order one time and. So, roughly all of them, since the same circuit take the same time. So, the total time that you require is essentially order of the number of levels right. So, how many levels would you have the number of levels you will have? So, consider the fact that we have 32 bits, and at each level the number of circuit. So, we start from 16, because you are considering maybe in this case 2 bit pairs at the same time.

So, we start with 16 and then go half of that 8 4 2 and 1. So, the number of levels since we are decreasing by 2 in every, in an every layer right in every level the number of levels, is basically order of $\log_2 n$. So, in computer science whenever we says you know say that a number is of the form $\log n$, we typically mean it is $\log_2 n$. So, that is the reason this can be written as simple as order of $\log n$. For any case $\log_2 n$ or base 10 or base 100 does not matter, because $\log_2 n$ and $\log_{50} n$, they only differ by a constant factor with anyway, the order notation takes care off. So, the case does not matter is just order of \log of n .

So, essentially computing this part there G n P functions for you know blocks of bits starting from consecutive bit pairs to 1 to 4, then 5 to 8, 9 to 12, 13 to 16 and so on. And then you know blocks of 8 blocks of 16 and then the entire blocks of 32 will essentially

take order $\log n$ time, because for each level we do a constant amount of work right, one and one and for the propagate and one on one and for the generator and we have order of $\log n$ levels. So, the total time it will take his order of $\log n$, and we will have essentially in a tree of G P functions, like an inverted triangle for different, we will have $G n P$ functions for different ranges of bits. So, the question is, we need to use it to actually do an addition.