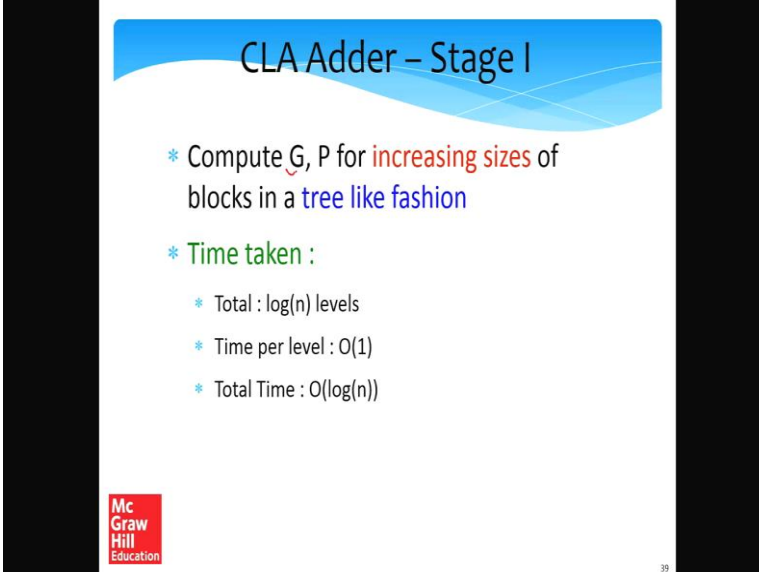


**Computer Architecture**  
**Prof. Smruti Ranjan Sarangi**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology, Delhi**

**Lecture – 18**  
**Computer Arithmetic Part-II**

(Refer Slide Time: 00:25)



**CLA Adder – Stage I**

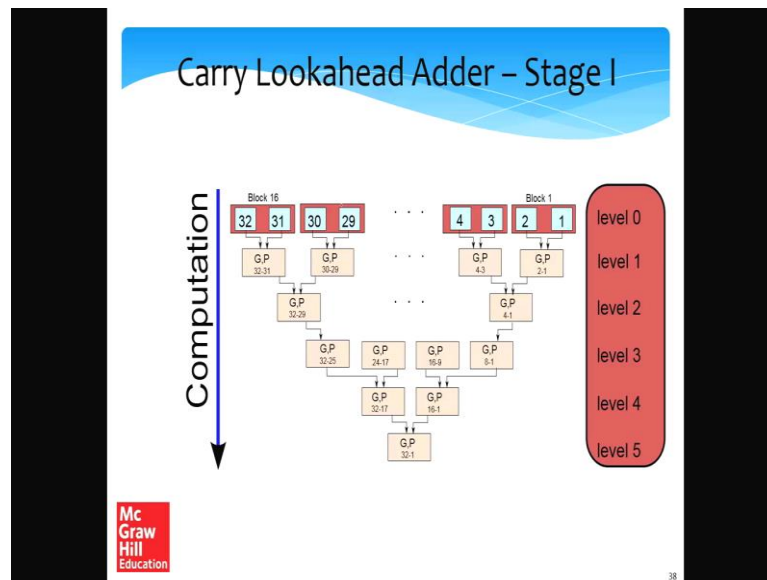
- \* Compute G, P for **increasing sizes** of blocks in a **tree like fashion**
- \* **Time taken :**
  - \* Total :  $\log(n)$  levels
  - \* Time per level :  $O(1)$
  - \* Total Time :  $O(\log(n))$

Mc  
Graw  
Hill  
Education

39

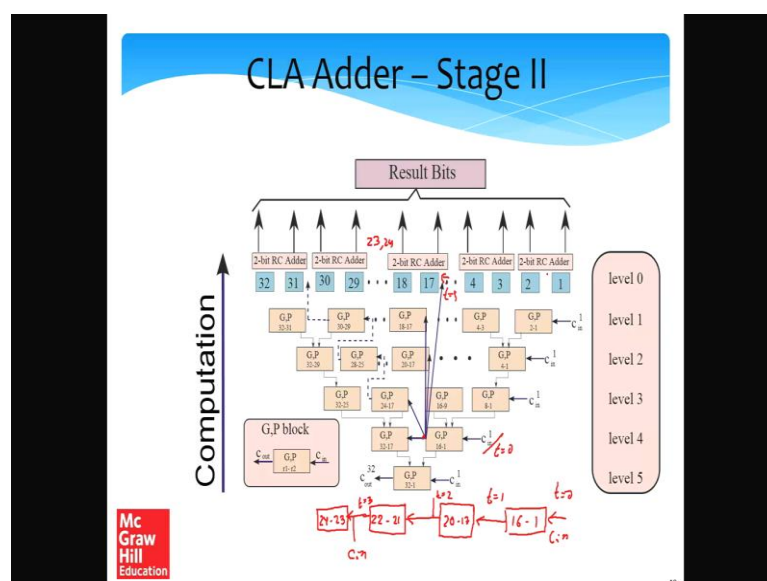
Let us just summarize the results of stage one. So, in the carry look ahead adder of this stage, we compute the generate and propagate the G, P functions for increasing sizes of blocks in a tree like fashion.

(Refer Slide Time: 00:45)



So going to the previous slide, we initially do it for small blocks of 2-bit pairs, 4-bit pair, 8, 16 and 32. So, assuming that there are  $n$ -bits that we want to add, there are total of approximately  $\log_2 n$  levels,  $\log_2 n$  to the base 2. The time per level is  $O(1)$ ; which means it is constant time. And, the reason that it is constant time is because we do not have to do a lot of work to compute  $G$  and  $P$  function; an AND gate in one case and, OR and AND gate in some other case. So, total time for stage one is order of  $\log n$ . So, that is for stage one. So, we have not done the addition yet. But, let us take a look at how the addition would be realized.

(Refer Slide Time: 01:30)



Let us now look at stage two in all its detail. So, recall that this is exactly the same diagram that we have been looking at earlier also for stage one. There are a few couple of changes. So, I will discuss that.

So, in stage one the aim was different. The aim was to compute the generate and propagate functions for a level by level. So, we pretty much (Refer Time: 03:05). See, if you would recall the diagram in stage one, we are considering, we started out with 2 bit blocks. So, basically the reason we did that is that an equation for generate and propagate function for a 2-bit block or a 2-bit system is not that complicated. So, we started out with that - reduces the number of levels. And then, we just created a tree of G, P blocks and aggregated this information till we went to the top, say a total of six levels, actually from 0 to 5.

So, we have the diagram here. It is just we have added some more inter connections for stage two. So, in stage two the idea is that once we know the generate and propagate function for each block, we should be able to efficiently compute the carry. And, once we know the carry, we can quickly do the addition. So, since you have considered 2 bit blocks, the first thing that we do is in level 0. So, what we do is that we consider this group of 2 bits, right, 2 bit pairs actually. So, since (Refer Time: 03:08) adding 32 bit, there will be sixteen 2 bit pairs. Each one of them can be added with a 2-bit R C. R C is a ripple carry adder.

But, the main thing that stop us from doing the addition is that we do not know what is the carry. For example, for this 29th and 30th bit pair, we do not know what is the carry that is coming inside or we do not know that what is the carry coming inside the 31st or 32nd adder. If we know that, if we know all the carries that are coming into these adders, we can quickly in order one time, do the addition. So, the idea is to quickly compute all the carries, such that we can use the ripple carry adders here to do the addition and then compute the result bits. So result bits, well, if you are adding 32 bit numbers, there can be 33-bit result. You are not showing the additional bit just to keep the diagram simple, but that needs to be kept in mind.

So, this, to quickly compute the carry, let us view a G, P block slightly differently. So, each G, P block has a range, a range of bits. For example, this G, P block over here is consisting the range of bits from the 17th position to the 20th position. So, we are calling

it  $r_1$  and  $r_2$ . And, so this is like from 29th to 30th. So, the range is set. For each of them if we get the carry in, we can always compute the carry out with the simple equation; which is carry out is to generate value or with propagate value and C in. So, this is a very simple thing. This requires one OR gate here and one AND gate. So, constant time per block, but the aim is to quickly compute all the carries. And that is what we shall try to do now. So, let us do one thing. So, there are lot of wires actually. Over and above, the wires that are shown here, but not all of them are been shown. I have just showed a couple for the purposes of illustration. So, let us consider the carry.

So, let us consider two 32-bit number that we are adding. So, in this case you know there is no carry in. But, let us consider a general case where there can be a carry in into the first bit. So, what we do is that we take all the right most G, P blocks; right most when we are looking at the screen. So, at  $t$  equal to 0, we essentially connect all the carry in, the C in values to the right most G, P blocks. So, the ranges are 1 to 2, 1 to 4, 1 to 8, 1 to 16 and 1 to 32.

So, let us assume for the purpose of explanation that the time it requires to; you know, the time it requires one G, P block to compute C out, which is same for all the block shown over here is one-time unit. So, to compute C out; it is one-time unit. It takes one-time unit to compute C out. And, so similarly we need to find out how many time units it takes for to do the entire addition.

So, our main aim is that for each of this 2 bit adders that we have shown here, we need to find all the carries. If you can do that very quickly, we are all done. So, let us again consider the beginning of stage two. When  $t$  is 0, we connect all the C input. So, the carry in values, I am sorry. And, so then after one-time unit, essentially all of these G, P blocks are ready with their carry out. So, we know the carry out after these second bit pair, after the fourth bit pair, eighth, sixteenth and thirty-two. So, we can again, you know, draw one more front, that it, so, this front is when  $t$  equal to 1. So, everything to the left of it, we do not know the carries. But, to the right of it, we know all the carries that are there; all the carry outs we know.

So, now let us do one thing. Let us look at how do we interconnect these block. So, let us take a look at one example. And then, rest of the blocks are also connected in the same way. So, we do not have to take a look at everything with so many wires. So, let us

consider this block which after  $t$  equal to 1 has its carry out ready. This range is from 1 to 16. So, for this particular block, since we know the carry out after the 16th bit pair, it can be connected to all the other G, P blocks which start from the 17th bit pair - for example, this one, this one, this one and this one. So, at  $t$  equal to 1, when the carry out of this block is available, it can immediately be sent to the rest of the blocks, these four blocks. And then, at  $t$  equal to 2, they carry out at the end of these blocks would be there.

For example, at  $t$  equal to 2, we will find that this G, P block would be ready with the carry out, out of the 24th bit pair. So, let us again erase the ink. So, we will realize that at this point at  $t$  equal to 2, so basically at  $t$  equal to 0, we are here, right at pretty much at the very beginning. At  $t$  equal to 1, we know the carry out at this point. And, similarly for other points I am just showing one particular path. They are  $t$  equal to 2. We are at this point where we know the carry out out of the 24th bit pair. So, this G, P block is connected to all the other blocks which; so, always we will connect something at a higher, which is sort of higher on the screen and whose level is lower, so that is the way that we will proceed.

So, in this case we will connect this with this block. And, so there can be other blocks also. But, let me just show one. So, in this case we will connect it with this block. So, this, the range is 25 to 28. And, so since this block here produces a carry out at the 24th bit position, we should be able to connect it. So, basically at  $t$  equal to 2 it will get the value of its carry in. Then, at  $t$  equal to 3 we will get. So, this block will finish its computation. We will get the carry in at this point; for the blocks which takes, which looks at the range 29 to 30.

Then, at  $t$  equal to 4, we will, this block will finish its work. This block will finish its work, the input again to this adder. So, the carry in will be available because this block would have computed its carry out. So, this is 29 to 30. So, the carry in at the 31st position would be ready.

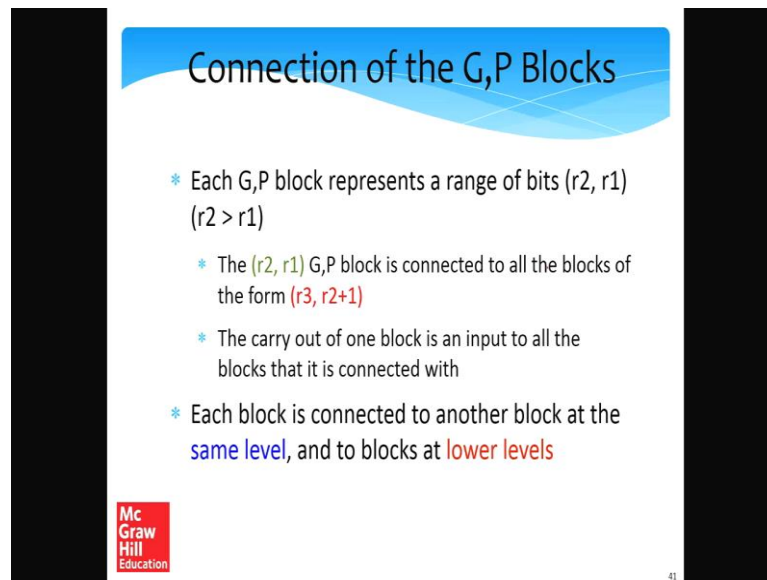
So, in a similar manner let me again, you know, delete, erase all the ink on the slide. So, basically what is the idea? The idea is that we start at the beginning with the right most blocks and at each stage we send the computed carry out to one of the blocks at lower levels. So,  $t$  equal to 0, we are here; then at  $t$  equal to 1, we are here. So, the front keeps on in advancing and, all the blocks at the lower level sort of in a keep on computing the

carry. So, basically within four steps we were able to compute the carry out here, I am sorry, the carry in at this point. So, this holds true for all the other points as well. That within a maximum of in a five steps, the values of the carry input values of all of these adders would be there.

So, let us may be consider the example with bit pair 17 and 18. So, let us see how its carry in will reach it. So, let us take a look at 17 and 18. So,  $t$  equal to 0, we are over here; at  $t$  equal to 1, we have computed the carry out over here. And, this immediately reaches the ripple carry adder, which is adding the bit pair 17 and 18 at  $t$  equal to 1 and an addition can be done. So, let us, we may be look at 23 and 24. Let us look at the bit pairs 23 and 24. So, essentially the way that things would pass is something like this that first, so I will show the sequence of G, P blocks that they arranges. So, first we will go from 1 to 16. So, this is, you know that the carry in is flowing this way.

So,  $t$  equal to 0, we are here; at  $t$  equal to 1, we will be here. This will be connected to a block; whose range is between 17 to 20. So,  $t$  equal to 2, we will have a carry out value over here. This will again be connected to one more block, whose range is 21 to 22. So, the end of  $t$  equal to 3, we will have the carry in over here. And finally, this will be connected to the block, whose range is 23 to 24. So, basically by  $t$  equal to 3, the carry in here will be available and we can then perform the additions. So, we can, you know, convince ourselves that for each of these blocks, that the end of in a five steps, we are guaranteed to have the carry in available for each of these adders. And then, we can perform the addition which is a quick step. It takes a constant amount of time.

(Refer Slide Time: 13:45)



## Connection of the G,P Blocks

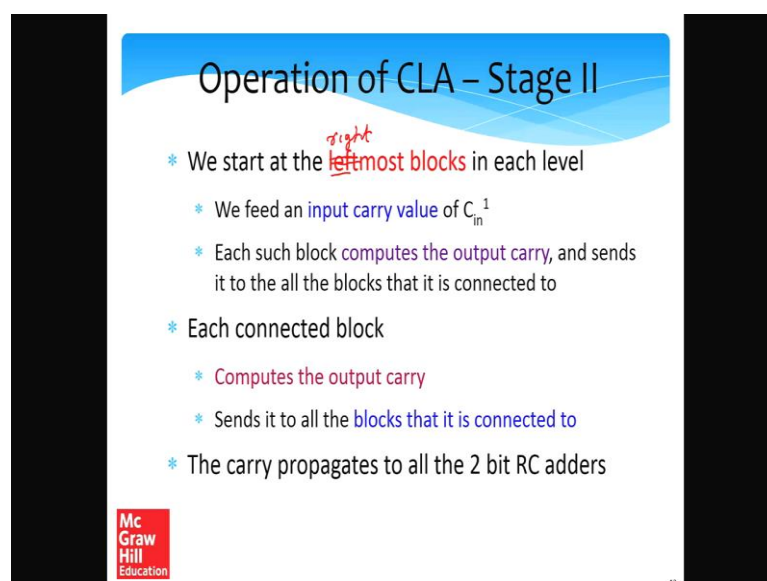
- \* Each G,P block represents a range of bits  $(r_2, r_1)$  ( $r_2 > r_1$ )
  - \* The  $(r_2, r_1)$  G,P block is connected to all the blocks of the form  $(r_3, r_2+1)$
  - \* The carry out of one block is an input to all the blocks that it is connected with
- \* Each block is connected to another block at the **same level**, and to blocks at **lower levels**

McGraw Hill Education

41

So, keeping this in mind let me once again explain this in a textual way. See, each G, P block represents a range of bits  $r_2, r_1$  to  $r_2$ , where  $r_2$  is greater than  $r_1$ . So, the  $r_2, r_1$  G, P block is connected to all the blocks of the form  $r_3, r_2$  plus 1. And to avoid, you know, extra connections we can assume that these blocks are at a lower level. So, the carry out of one block is an input to all the blocks that it is connected with. And, each block is connected to another block, either at the same level or at lower levels as we, you know, discussed. Sometimes, same level connections are also not necessary. So, we clearly want to minimize the number of wires.

(Refer Slide Time: 14:28)



## Operation of CLA - Stage II

- \* We start at the ~~left~~<sup>right</sup>most blocks in each level
  - \* We feed an input carry value of  $C_{in}^{-1}$
  - \* Each such block computes the output carry, and sends it to all the blocks that it is connected to
- \* Each connected block
  - \* Computes the output carry
  - \* Sends it to all the blocks that it is connected to
- \* The carry propagates to all the 2 bit RC adders

McGraw Hill Education

42

So, we start at, well, it should not be the left most blocks in each level, I am sorry, should be the right most. Well, it depends on how you are looking at it. It depends on how you are looking at it, but if we you are starring at the screen it should be the right most blocks actually, not the left most. And then, we feed it an input carry of C in 1. So, then each block will compute the output carry and send it to all the blocks is connected to till it reaches the ripple carry adders. And, so basically this is again the same diagram shown once again just to reinforce the facts of how this is done. And then, let us move to 44th slide.

(Refer Slide Time: 15:14)

**Time Complexity**

- \* In a similar manner, the carry propagates to all the RC adders at the zeroth level
- \* Each of them compute the correct result
- \* Time taken by Stage II :
  - \* Time taken for a carry to propagate from the (16,1) node to the RC adders
  - \*  $O(\log(n))$
- \* Total time :  $O(\log(n) + \log(n)) = O(\log(n))$

McGraw Hill Education

44

So, in this slide let us take a look at the time complexity. So, time complexity is that it is a time taken for a carry to propagate from, you know, one node to all the other nodes. And, since we go down one level in each time unit, it has its limit. It has order of log n steps. So, basically stage one took order of log n steps, stage two takes order of log n steps. And, addition at the end is constant time, so we ignored. So, here we go. The total time that it takes is order of log n, which is exactly what we had set out to prove that we can have an adder. In fact, a very faster adder and, this adder is also used in commercial circuits and it takes order of log n time to finish its computation.



(Refer Slide Time: 16:12)

Time complexities of different adders:

- Ripple Carry Adder:  $O(n)$
- Carry Select Adder:  $O(\sqrt{n})$
- Carry Lookahead Adder:  $O(\log(n))$

The slide includes handwritten diagrams. The first diagram shows a ripple carry adder with four full adder blocks connected in series, with a carry-in  $c_0$  and carry-out  $c_n$ . The second diagram shows a carry select adder with two parallel paths of full adder blocks, with a carry-in  $c_0$  and carry-out  $c_n$ . The third diagram shows a carry lookahead adder with a carry-in  $c_0$  and carry-out  $c_n$ , and a carry propagation delay  $\Delta$  indicated by a triangle and arrows.

Mc Graw Hill Education

45

So, let us summarize the time complexities of different adders. The ripple carry adder takes order of  $n$  time. The carry select adder that we discuss, so what was the idea of the ripple carry adder once again? The idea of the ripple carry adder was that we basically have, you know, this fixed size blocks. And may be, you know, with one-bit pair or two-bit pairs, does not matter. And then, we do the addition and carry sort of ripples. That is the reason. You know, in worst case it can take order of  $n$  time.

What we did with the carry select adder is that we had slightly larger blocks. And, but in the first stage we actually computed two results. One with an input carry of 0, one with an input carry of 1. So, we had two results. And so, then what we did is that we quickly took a look at.

So let me, may be clean this up slightly such that I can re explain in a better way. So, what we did is that we computed two results. One is input carry. And, input carry assuming 0 and assuming 1. So, given the value of the input carry at the beginning, we were quickly able to compute the output carry, may be this way and choose a set of results, you know, based on the carry out. So, we figure out, we got an equation of the form order of  $k$  plus  $n$  by  $k$ . And, so this we differentiate it. So, and then after some, you know, calculus, the time came to order of square root of  $n$ .

So, carry look ahead adder is a latest adder in our portfolio of adders. It is very fast. It is the fastest. It is also used in commercial processors. So, this takes order of  $\log n$  time.

And,  $\log n$  is clearly, you know, much much faster than of  $n$  or square root of  $n$ . So, the advantage here is that you know it is again a two stage algorithm, which is very fast. That is the advantage. And, the way we do it is basically via creating a tree that computes the G, P function. See, first walk in a downwards, compute the G, P function for each block. And then, we walk upwards where we compute the carry outs for each bit pairs. And finally, we use a ripple carry adder to do the addition.

(Refer Slide Time: 18:51)

### Outline

- \* Addition
- \* Multiplication
- \* Division
- \* Floating Point Addition
- \* Floating Point Multiplication
- \* Floating Point Division

46

(Refer Slide Time: 18:58)

### Multiplicands

$$\begin{array}{r} 13 \\ \times 9 \\ \hline 117 \end{array}$$

(a)

$$\begin{array}{r} 1101 \\ \times 1001 \\ \hline 1101 \\ + 0000 \\ \hline 110100 \\ + 110100 \\ \hline 1110101 \end{array}$$

(b)

Partial sums

- \* 13 → Multiplicand
- \* 9 → Multiplier
- \* 117 → Product

47

So, we shall take a look at algorithms for in a fast multiplication next. So, let us go back to class two and take a look at how multiplication is done. And then, once we figure out how multiplication is done, we will find, we will make a computer circuit for it.

So, let us try to multiply 13 times 9. So, 13 times 9 in base 10 is 117. So, this is in base 10. So, let us now try to multiply 13 times 9 in binary. Let us consider them unsigned binary numbers. So, 13 is 1101; 8 plus 4 is 12, plus 1 is 13. And, 9 is 1001. So, the way we perform multiplication in binary we shall see is exactly the same as the way we do multiplication in decimal. There is no reason why it should be different.

So, so before looking at the multiplication let us define some terminology. So, the number that we write on top, you know, it is the matter of tradition is called the multiplicand. And, the number that we are writing at the bottom, which in this case is 9 is called the multiplier. So, of course the multiplicand and multiplier are symmetric terms. But, you know one of the numbers has to be in our system; one of the number which is written at the top. Let us just call it the multiplicand and, the one that is written at the bottom. So, in this case 13 is written at the top. So, that is the multiplicand. And, 9 is the multiplier. And, 117 is the product. So, in this case 1101, which is the binary representation of 13 is the multiplicand and 9 is the multiplier. So, let us do one thing. So, what do we do while multiplying base 10 numbers is that we start from the right and move towards the left. For each digit, we multiply the multiplicand at the top with the digit.

So, let us first consider the first digit here; that the right most. So, this is 1, so we multiply the entire multiplicand by 1. So, we get 1101. Then, we move to the next digit which is 0. So, 0 times any number is 0, we just write four 0's below the other; 0000. Next, in the next digit we again have a 0. So, we write four 0's; once again 000 and 0. And finally, we come to the left most digit or the MSB digit. So, since we are looking at unsigned numbers, we are relatively say for now this is not the sign bit. So, we multiply again 1101 times 1, where the result of that multiplication is 1101.

So, now what we need to do is that we; so each of these numbers that we computed is called a partial sum. So, 1101 is a partial. So, each of these, you know, roundish, bluish, rectangular boxes are known as partial sums. So, we need to add up these partial sums. And, mind you the one partial sum is offset by one position to the left as compared to the

previous partial sum, in the sense is multiplied by 2. So, what we do is that we take 1. So, 1 is essentially added with 0's. We can write a 0 again. So, just to make the addition easier to visualize, we can write 0's here as is. So, this is exactly the same as the normal base 10 multiplication that the little children learn when they are in a second grade or third grade or fourth grade.

So, what we do is that we add 1 with 0. So, we get 1. At this point, we get 0; we add 1 with three 0es. We again get a 1. Then, 1 plus 1, we use it to get a 0. And, there is a carry of 1. So, 1 plus 0 plus 0 becomes 1, then 0 plus 1 is 1. And, again we have 1 here. So, this is what we have. So, what is this number? So, this is power of; this is 1 plus 4 plus 16 plus 32 plus 64. So, this number is. So, if we add 64 plus 32 is 96. 96 plus 16 is 112 plus 4 is 116 plus 1 is 117.

So, we clearly see that if we multiply the same numbers in binary, we get the same result; which is 117. And, in this case we get the binary representation of 117, which is 110101. While (Refer Time: 24:10) there you see that multiplying numbers in base 10 and base 2 is exactly the same and we also get the same result.

So, the important take home points here are we define the term multiplicand, which is this number; we define a multiplier, which is this number; then, the final result is a product and each of these blue rectangles is a partial sum.

(Refer Slide Time: 24:33)

**Basic Multiplication**

- \* Consider the **lsb of the multiplier**
  - \* If it is 1, write the value of the multiplicand
  - \* If it is 0, write 0
- \* For the next bit of the multiplier
  - \* If it is 1, write the value of the multiplicand **shifted by 1 position to the left**
  - \* If it is 0, write 0
- \* **Keep going ....**

McGraw Hill Education

48

So, what did we do? We started from the LSB, least significant bit of the multiplier, kept going left. If any of these bits is 1, we write the value of the multiplicand; if it is 0, we write 0. Then, we move to the next bit of the multiplier. And, essentially at every point we shift the partial sum one place to the left.

(Refer Slide Time: 24:56)

**Definitions**

**Partial sum:** It is equal to the value of the multiplicand left shifted by a certain number of bits, or it is equal to 0.

**Partial product:** It is the sum of a set of partial sums.

- \* If the multiplier has m bits, and the multiplicand has n bits  $(2^m - 1) \times (2^n - 1) < 2^{m+n}$
- \* The product requires (m+n) bits

49

So, let us now have two quick definitions. Partial sum: it is equal to the value of the multiplicand left shifted by a certain number of bits, as we just saw or it is equal to 0. And, a partial product is essentially a sum of a set of partial sums. And, a product is definitely sum of all the partial sums, but a partial product is a sum of a set of partial sums. We also know that if the multiplier is m bits and the multiplicand is n bits, the maximum size of the product will be. So, in m bits, let us see. With m bits assuming that it is an unsigned number, the largest number that we can represent is 2 to the power m minus 1. Similarly, the largest number that we can represent with n bits is 2 raised to the power n minus 1, which is strictly less than 2 raised to the power m plus n. So, the product requires m plus n bits.

(Refer Slide Time: 26:07)

**Multiplying 32 bit numbers**

- \* Let us design an **iterative multiplier** that multiplies two 32 bit signed values to produce a **64 bit result**
- \* What did we prove before
  - \* Multiplying two signed 32 bit numbers, and saving the result as a 32 bit number is **the same as**  $u \rightarrow F(u)$   
 $-3 \rightarrow F(-3) = 1101$   
 $3 \rightarrow F(3) = 0011$
  - \* Multiplying two unsigned 32 bit numbers (assuming no overflows)  $F(u \times v) \equiv F(u) \times F(v)$
- \* We did not prove any result regarding saving the result as a 64 bit number  $F(2 \times (-3)) \equiv F(2) \times F(-3)$   
 $0010 \times 1101$   
 $= 1010 = F(-6)$

McGraw Hill Education

So, now let us look at multiplying two 32-bit numbers. So, let us first design a simple multiplier called an iterative multiplier that will multiply two 32 bit signed values. So, this is the most general case. So, multiplying unsigned number is always easy. But, let us deliberately complicate our life and look at two 32 bit signed values. Let me just erase that. So, we need two 32 bit signed values. And, we will produce a 64-bit result.

So, what did we prove before? Well, we have proved before that multiplying two signed 32 bit numbers and saving the result as a 32-bit number is the same as multiplying two unsigned 32 bit numbers assuming no overflows. So, what we have essentially proved in chapter two is that so if you can go back to chapter two, but what we have, we proved this result that if there are no overflows, then we can very well; see and let us see want to multiply two signed to 32 bit numbers, which can be positive or negative. Then, what we do is that we take that two's complement representations. So, basically if a number is  $u$ , its two's complement representation is  $F u$ .

So, basically let us consider an example. Consider a 4-bit system. So, if a number is minus 3, then essentially  $F$  of minus 3 is plus 13, which is 1101. And, if the number is plus 3, then the representation of plus 3,  $F$  of plus 3 is 0011. So, what we have essentially proven is that to multiply two numbers. And, you know the representation in two's complement of the product of two numbers is essentially the same as; consider each of the numbers individually, take the two's complement and then multiply them assuming


that they are regular unsigned 32 bit numbers. You will be just fine as long as there are no overflows. We however did not prove any result regarding saving the result as a 64-bit number. But, this is what we know. So, what we know is that we want to multiply 2 times minus 3.

So, let us see the 4-bit number system. We want to multiply 2 times minus 3. What we know is that multiplying this is equivalent to multiplying F of 2 times F of minus 3. So, F of 2 is 0010. That is being multiplied with 1101.

So, actually if you multiply 2 times 13 we get 26 and, since we cannot save. But 26, we will discard the highest bit because it is a fifth bit. So, we will be left with 26 minus 16 which is 10. So, we will essentially get 1010. 1010 is F of minus 6. So, we thus see that, you know, treating these numbers as unsigned numbers and multiplying them is good enough. But, in this case this is only if the size of the result is also 32 bits. However, in this case we are considering the possibility of saving the result as the 64-bit number. And, we have not proven, you know, any theorems or any axioms or any lemmas for what should be done in this case.

(Refer Slide Time: 30:04)

**Class Work**



$$-4 = 1100$$

$$A_{1 \dots n-1} = 4$$

$$-4 = 4 - 1 \times 2^{4-1}$$

$$= 4 - 8$$

**Theorem:** A signed  $n$  bit number  $A = A_{1 \dots n-1} - A_n 2^{n-1}$ .  $A_i$  is the  $i^{\text{th}}$  bit in  $A$ 's 2's complement based binary representation (the first bit is the LSB).  $A_{1 \dots n-1}$  is a binary number containing the first  $n-1$  digits of  $A$ 's binary 2's complement representation.

51

So, let us take a look at very very important theorem that we need to prove. Once we are able to do that, you will find that a lot of the multiplication work that we do will become very easy to understand. So, it is very important. We take a look at this.

So, let us consider a signed  $n$  bit number  $A$ . So, let us first consider a number which is made by the first  $n - 1$  bits. So, what we want to say that the number  $A$  is equal to the number made by the first  $n - 1$  bits minus the MSB, multiplied by  $2$  to the power  $n - 1$ . So, here  $A_i$  is the  $i$ th bit in  $A$ 's two's complement based binary representation. The first bit is LSB. And,  $A_1$  to  $n - 1$  is a binary number containing the first  $n - 1$  digits of  $A$ 's binary two's complement representation. There is a lot of text. Let us consider an example. It will become crystal clear.

So, let us consider  $3$ . And, let us consider a 4-bit representation. This is  $0011$ . This is exactly the same as we consider a number made with the first  $n - 1$  bits. So, this is  $011$ ; is still  $3$  minus the MSB is  $0$ ,  $0$  times  $2$  to the power  $4 - 1$ . So, but  $0$  times any number is  $0$ . So, basically this is equal  $3$ . So, the important, for a positive number proving this is trivial but, the important case is a negative number. So, let us consider a minus  $3$ . So, minus  $3$ 's representation in a two's complement system is  $1101$ . So, basically this is equal to let us consider this number. So,  $101$  is  $5$  minus MSB is  $1$ . So, it is  $1$  times  $2$  to the power  $4 - 1$ ;  $2$  raised to the power  $4 - 1$  is  $2^3$ ,  $8$ . So, this is same as  $5 - 8$ . So, this is the important result that we need to prove that if we consider; let us consider one more example. So, let me delete this part. And, let us consider one more example and here is what we need to prove.

So, let me consider, sorry, let me consider, let us say the number minus  $5$ . In a 4-bit representation, the number of minus  $5$  will be represented as plus  $11$ , which is  $1011$ .  $1011$  can be interpreted in another way. So, this is  $011$ . This is  $3$ . So, we are seeing that minus  $5$  is equal to  $011$ ,  $3$  minus  $1$  times  $2$  to the power  $4 - 1$ , which is actually the case;  $3 - 8$ , which is minus  $5$ . So, in a sense what we want to prove which is this result that given a sign  $n$  bit number  $A$ , if we consider the number made by its first  $n - 1$  bits. If that number is  $A_1$  to  $n - 1$  and we subtract the MSB times to the power  $n - 1$ , we will get the original number  $A$ . That is what we need to prove. So, what we have seen is that for positive number this is trivially true because for positive numbers, the MSB is  $0$ .

So, since the positive number is  $0$ , we put  $0$  here and  $0$  times any number is  $0$ . So, essentially this part goes away. So, in any positive for, in any positive two's complement number the MSB is  $0$ . So, naturally if we get rid of the MSB, the number that is left will be equal to the number itself, here we are assuming that this number is unsigned. So, it




will be equal to the number itself. But, the important point that we need to prove is that for negative numbers this result holds.

That if I take any negative number, again consider one more example; minus 4. Minus 4 is 1010. Minus 4 is also equal to this. Let us take 0101, which is 2 minus 1 times 2 to the power 4 minus 1. Oops! Sorry, I made a mistake in that. I will just go once again. So, let us consider minus 4. So, minus 4 is actually plus 12. So, as per this theorem over here,  $A_{1 \text{ to } n-1}$  is actually this number here; 100, which is plus 4. And, so essentially minus 4 is equal to 4 minus 1 times. We are considering (Refer Time: 35:20) a 4-bit number system, which is 4 minus 8. So, the question is that why does this thing hold? We have already proven this in chapter two. But, we want to prove it once again because we will really be using this result. We will be in a heavily using this result in all our work on multiplication as well as division. So, we sort of want to you know ensure that we understand this very well.

(Refer Slide Time: 35:43)

### Class Work



$-u \ (2^n) \dots$

$A < 0 \quad -1 \ -3 \ -2$

$\hookrightarrow F(A) \quad 15 \ 13 \ 14$


$F(A) = A_{1 \dots n-1} + 2^{n-1} \dots (1)$

$F(A) = 2^n + A \quad \dots (2)$

$2^n + A = A_{1 \dots n-1} + 2^{n-1}$

$\Rightarrow A = A_{1 \dots n-1} - 2^{n-1} \times A_n$

**Theorem:** A signed  $n$  bit number  $A = A_{1 \dots n-1} - A_n 2^{n-1}$ .  $A_i$  is the  $i^{\text{th}}$  bit in  $A$ 's 2's complement based binary representation (the first bit is the LSB).  $A_{1 \dots n-1}$  is a binary number containing the first  $n-1$  digits of  $A$ 's binary 2's complement representation.


51

So, as I said for positive numbers, this is trivially true because  $A_n$  is 0. So, basically if we just discard the MSB bit, the number remains the same and  $A$  is equal to the number made by the first  $n$  minus 1 bits. For negative numbers, the situation is slightly different. So, let us consider a proof.

So, let us consider the number  $A$  to be less than 0. So, if the number  $A$  is greater than 0, this is a trivial result. So, if it is less than 0, let us also look at the representation of  $A$ . Let

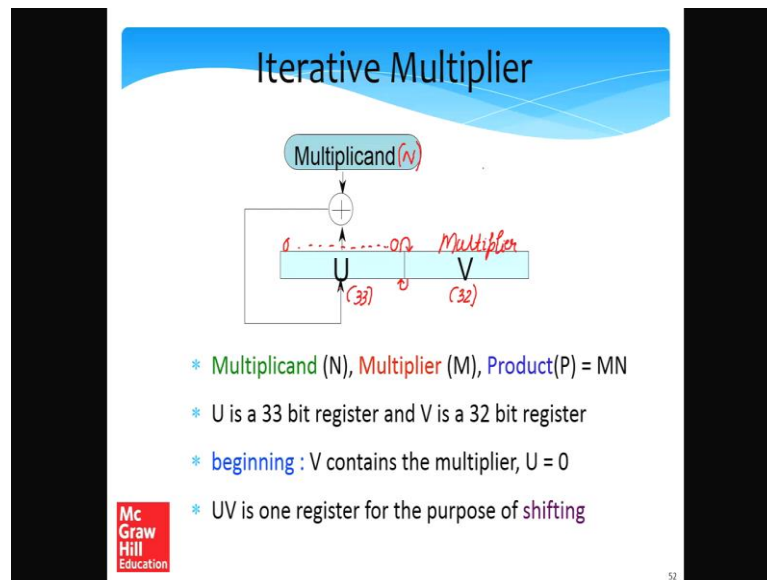
us consider the unsigned number that represents the two's complement representation of  $A$ . For example, if  $A$ , in a 4-bit system, if  $A$  minus 3, then  $F$  of  $A$  is plus 13. If  $A$  is let us say minus 2, what will be  $F$  of  $A$ ? It will be plus 14 so on and so forth. So, what we can say is that this number  $F$  of  $A$  is essentially equal to the number which is formed by considering the first  $n$  minus 1 bits of the two's complement representation of  $A$  plus  $2$  to the power  $n$  minus 1. The reason that I can write it this way is because for every negative number, the most significant bit is 1, it is 1. So, basically any number of this form will expand to  $2$  to the power  $n$  minus 1 plus something. So, that is the reason if the MSB is 1, I can write  $2$  to the power  $n$  minus 1 over here, plus the number formed by the rest of the digits, which is rest of the bits which is  $A$  1 to  $n$  minus 1.

Similarly, I can write one more equation. So, in this equation what I can write. So, this follows directly from, you know, that the definition of the two's complement that  $F$  of  $A$  is equal to  $2$  to the power  $n$  plus  $A$ . So, why is this the case? Let us just consider these examples to find out more.

So, when we want to find the two's complement representation of minus 3, what we do is we actually add 16 to it. So, when we add 16 to it, we get plus 13, which is its representation. Similarly, when we want to find the two's complement representation of minus 2, we also add 16 to it in a 4-bit system to get plus 14. Essentially, if you want to find the two's complement representation of any number of the form minus  $u$ , where  $u$  is positive, its representation is basically  $2$  to the power  $n$  minus  $u$ .

So, so basically in this case since the number is negative, we just add it to two to the power  $n$ . And, so for example, I want to find the representation for minus 1. I just add it to plus 16. So, it is plus 15, which is 1111. So given these two equations, given equations one and two, what is it that we can write? We can write. And, since we know that the MSB bit is 1 for a negative number, we can multiply this with  $A_n$ , when  $A_n$  with a  $n$ th bit or it is the MSB. So, this expression over here;  $A$  is equal to the number formed by considering the first  $n$  minus 1 bits in  $A$ 's two's complement representation, which is this number over here, minus  $2$  to the power  $n$  minus 1 times  $A_n$ , where  $A_n$  is the  $n$ th bit. And, in this case we know it is 1. So, you proved it for the case that  $A$  is less than 0 and, the proof for the case that  $A$  is greater than 0 is trivial. The main reason being that  $A_n$  is 0. And, so we remove 0 out of the number. We have the same number. So, given the fact that we armed with this proof, let us move forward.

(Refer Slide Time: 40:49)



And, let us try to design a multiplier. So, so let me just make one more point before we proceed. I just want to repeat it because it is very important. So, let me go to this slide. So, what we proved in chapter two was something like this that when we multiply two signed 32 bit numbers and save the result again as 32-bit number without any overflows, it is exactly the same as considering the unsigned representations.

You know, considering that, not the unsigned representations, but considering both the 32 bit numbers to be unsigned numbers and performing regular unsigned multiplication on them and then, using the last 32 bits as the result. So, that is what we proved and we showed examples that it works. And, we also theoretically proved that this result is correct.

In this case, what we want do is slightly more generic and more complicated. We want to multiply two 32 bit numbers, no doubt. So, we want to multiply numbers; let say A times B, if they are 32 bits and we want to save it in a number called C. But, this number is mind you not a 32-bit number; it is a 64-bit number. It is a larger number. So, this result will not hold, but we will find that with the help of this theorem over here, we will be able to you know solve this problem very easily.

So, let us now take a look at the iterative multiplier. So, the iterative multiplier is a very simple structure as can be seen. So, we have the multiplicand, and the multiplicand is stored in a register. So, let us call the register N, then we have the multiplier. So, we will

be referring to the multiplier as  $M$ . So, the aim is to compute the product, which is  $M$  times  $N$ .

So, in this particular scheme we have two registers that we shall use. So, we are doing mind your 32-bit multiplication.  $U$ , register  $U$  is a 33-bit register. So, this is 33 bits and register  $V$  is 32 bits. So, why is this 33? Well, we have an extra bit to take care of the overflows. That is the only reason. Otherwise, you know there is no other reason.

So, basically both these registers act as the single register for the purpose of shifting. What this means is that I can shift both the registers to the right. So, then the least significant bit of  $U$  will become the most significant bit of  $V$ . Similarly, I can shift both the registers to the left. Then, the least significant, sorry, the most significant bit of  $V$  will become the LSB of  $U$ . So, for the purpose of shifting, they act like a single register.

So, when we shall start the algorithm, what we will do is that we will load the multiplicand into the register  $N$  and we will load the multiplier into  $V$ . So, we will start with having the multiplier. And, register  $U$  will contain all 0's. So, so that is the way that we start. We load the multiplier in register  $V$ . Register  $U$  will contain all 0's and the multiplicand will be in register  $N$ , which is shown on top. And, we will have a circuit over here, where we load the current contents of  $U$ . We will add it to the multiplicand. And, the result of the addition will be fed back to the register  $U$ . So, that is pretty much the only circuit. But, let us take a look at the algorithm that will show how exactly this circuit works.

(Refer Slide Time: 45:02)

### Algorithm

**Algorithm 1:** Algorithm to multiply two 32 bit numbers and produce a 64 bit result

**Data:** Multiplier in  $V$ ,  $U = 0$ , Multiplicand in  $N$   
**Result:** The lower 64 bits of  $UV$  contains the product

```
i ← 0
for i < 32 do
  i ← i + 1
  if LSB of V is 1 then
    if i < 32 then
      U ← (U + N) (U ← N)
    end
  else
    U ← U - N
  end
  UV ← UV >> 1 (arithmetic right shift)
end
```

Mc  
Graw  
Hill  
Education

53

So, the algorithm is as follows. So, we want to multiply two 32-bit numbers and produce a 64-bit result. So, we start out with having a multiplier in  $V$ . So, the multiplier is loaded in register  $V$ . And,  $U$  is 0 as shown. And, the multiplicand is in register  $N$ . So, essentially to avoid overflows, we made  $U$  a 33-bit register. So, the lower 64-bits of the  $UV$  register combined will contain the product at the end. So, what we shall do is that we shall start a for loop that will iterate for 32 times; a standard way we have a variable and we iterate for 32 times. So, essentially at this point  $i$  will go from 1 to 32, 32 times because there are 32 bits.

So, let us consider the first iteration. If the LSB is a least significant bit of  $V$ , which contains the multiplier is 1; which basically means we come here and we take a look at the least significant bit. If this bit is 1, so what is it that we need to do? Well, what we need to do is that we need to add the multiplicand to the accumulating partial product. So, since the first iteration, the partial product will just be the multiplicand itself. So, clearly in the first iteration  $i$  is 1. This is less than 32. So, what we do is we add  $U$ . So, we set  $U$  to  $U$  plus  $N$ . So,  $U$  initially is 0. So, essentially what we do is that we set  $U$  to  $N$ . So, that is anyway the first thing that we should do. That if we see that the least significant bit of the multiplier is 1, then what we should do is that we should, essentially the product will be the multiplicand itself. So, we set  $U$  to  $U$  plus  $N$ . And, so let us not talk about this part right now. We will talk about it later.

And then, what we do is that we set, we take the register U V and we shift it to the right; one arithmetic right shift to preserve the sign bit. We shift it to the right by one position. So, let us see what is happening.

So, initially we have the register U and the register V. And, the register V contains the multiplicand. After that, let us say that this bit is 1. If this bit is 1, then the multiplicand is connected, is sort of the multiplicand occupies register U. And, register V will contain the multiplier M. So, then we shift it to the right. If we shift it to the right by one position, then you know I still, this is still, you know, register U and register V. So, if I do it, then what will happen is that the product will sort of V and U and as well as in one bit in V. So, this will sort of be the partial product. (Refer Time: 48:17). The partial product will scan both the registers. And, the least significant bit of the multiplier will fall out from the right. So, this part will connect, will contain the multiplier. And, if the multiplier initially at 32 bits, we will only have thirty one bits of the multiplier left because we shifted it to the right by one position.

So, then again we will come to the next iteration. Again take a look at the least significant bit of the multiplier. If it is 1, we will again add it to the partial product. If not, we will see if the least significant bit of V is not 1, then we do not need to add anything because if we go back to the original, to this example; so, what we are essentially doing is first we are considering this bit. So, this is 1. So, essentially we set U to the multiplicand, then we shift both the multiplicand and the multiplier by one position. So, this bit falls off from the right. So, then we consider this bit. Since this bit is 0, we do not have to do anything. We again shift one step right. So, this bit becomes the least significant bit. And, since this is 0 also we do not do anything till we come to the last position.

We will now discuss what we do in the case of the last position because this varies for signed and unsigned operands. So, but the important point to note from this particular diagram is that if the bit under consideration in the multiplicand is 0, then we do not do anything. So, in a sense we do not update the partial product. So, let me once again explain what is the partial product. So, partial product is a sum of partial sums. So, what we essentially do in a multiplication is that we first create the partial sums. So, the second partial sum which is 000 is written in such a way that it is left shifted by one place as compared to the previous partial sum. So, essentially the  $i$ th partial sum is

shifted to the left by one position as compared to the  $i$  minus 1 th partial sum. So, if we consider the first  $i$  partial sums, we can add them up to make a partial product. So, then the partial product will be added to partial sum, to the next partial sum.

(Refer Slide Time: 50:56)

**Algorithm**

**Algorithm 1:** Algorithm to multiply two 32 bit numbers and produce a 64 bit result

**Data:** Multiplier in  $V$ ,  $U = 0$ , Multiplicand in  $N$   
**Result:** The lower 64 bits of  $UV$  contains the product

```

i ← 0
for i < 32 do
  i ← i + 1
  if LSB of V is 1 then
    if i < 32 then
      U ← U + N
    end
  else
    U ← U - N
  end
end
UV ← UV >> 1 (arithmetic right shift)
end
  
```

The slide also features hand-drawn diagrams in red ink. One diagram shows a register  $V$  with its least significant bit (LSB) circled and labeled '1'. Another diagram shows a register  $U$  with a value  $N$  being added to it, indicated by a plus sign and an arrow. A third diagram shows a register  $UV$  with a value  $N$  being subtracted from it, indicated by a minus sign and an arrow. A final diagram shows the  $UV$  register with a rightward arrow and the text 'arithmetic right shift'.

So, let us come back to the algorithm. In the algorithm, we run the loop for 32 times. For the first 31 times, this is what we do. We take a look at the least significant bit of  $V$ . If it is 0, we do not do anything. We do not have to do anything because in this case the partial sum is 0. However, if it is 1, we add it to the register  $U$  and we proceed. At the end of this 'if statement', we shift the  $U V$  combine, a one step to the right using an arithmetic right shift operation because you want to preserve the sign.

Now, the first thing that needs to be explained is that why do we shift one step to the right. So, let us look at a general case. Let us look at, you know, some iterations in the middle. Let say the  $i$  th iteration, where  $i$  th, where  $i$  is clearly not 32. So, in the  $i$  th iteration. So, in iteration number  $i$ , let us consider this point. You know exactly at this point. So, here we have already computed the partial product for the first  $i$  partial sums. So, the partial product is contained partly in register  $V$  also. So, maybe we can say that this entire region contains the partial product, which is the sum of the  $i$ , first  $i$  partial sums. And, the remaining region contains the bits of the multiplier. Then, what we do is that we shift the  $U V$  register combine one step to the right. So, effectively what this means is that we have one bit. So, basically when we shift it to the right, we maintain the

sign bit as well, but this regarding that the partial product is pretty much contained in this fashion. This is for register U. And, again for, you know register V, the partial product will again come here and we will have the remaining set of multiplier bits, where one bit just fell out because of the right shift operation.

So, what we do now is that we come to the next iteration, in the next iteration if the partial sum is 0, well and good. We do not have to anything. But, let us consider the case where the partial sum is 1, sorry, when the least significant bit is 1. So, in this case what we do is very interesting. So, in this case what we do is that we add multiplicand to the register U. So, the question is that why do we add the multiplicand to the register U. And again after adding, the result of the addition is used to update the register U is essentially add N plus U. So, right, we have an adder like this. We add N plus U. And, the result of the addition is fed back to register U. So, this is pretty much the same as.

So, what we are doing if we just; so, if I just take it out, it is pretty much the same as doing the following addition, where I write register N here, which is the multiplicand. And then, I consider the partial sum, where the partial sum is starting one bit to the right. So, so basically if you think about it. So, this is pretty much where the partial product is ending, and the partial sum has one extra bit. So, I am just reproducing the same thing, but I am just writing it in a reverse fashion. If I write it to the; so, in this if I write it this way, the multiplicand is one step to the left. But, if I see from the point of view of the multiplicand, the partial product is shifted one step to the right. So, essentially the partial product will be a number of this fashion.

And then, we perform the addition. So after performing the addition, we get a new partial product which adds the multiplicand as well. Subsequently, what we do is we come to this step and we again shift it one step to the right and again we add the multiplicand. So, why we are doing this will be very very self-evident, if we take a look at this once again.

So, let us consider the last step. So, in the last step, here, mind you, we are multiplying unsigned numbers and in an algorithm we are multiplying sign numbers. So, there is little bit of a difference. But, let us still take the unsigned example. So, in the last step what we are doing is that we are writing the multiplicand which is N and we are adding it to the partial sum. So, basically as far as we are concerned, the multiplicand is essentially shifted one bit to the left as compared to the previous partial sum. So, we essentially, this



is exactly the same as we take the partial sum. We take the first  $i$  minus 1 partial sums and we shift that one step to the right as compared to the last partial sum. So, the effect is the same. You shift one number to the left and then add or you keep one number constant and you shift the others to the right, the relative movement is the same.

So, in the case of a regular multiplication we just write the partial sums and we keep on shifting them one place to the left. Another idea can be that for example, we take the partial sum as 1101 first and we shift 1101 one step to the right and then we add 0000. Then, we shift it one step to the right and then we again add 0000. Then, we again shift this one step to the right and then we again add; which is exactly what we are doing. So, instead of shifting the partial sums one step to the left here, what we can do is that we can shift the partial product, which is our current accumulated set of partial sums, one step to the right. And, of course not to lose any bits and, keep on adding the multiplicand, if need be. So, the effect is the same.

Essentially, shifting a number left and shifting a number right depends on from where we are looking at it. So, what we are essentially doing in this line, over here, is that the current computed partial product is being shifted one step to the right. And then, we are adding; if the LSB, the current LSB is 1, we are adding the multiplicand. So, the effect is the same. The effect is basically that after shifting this, after shifting the partial product  $k$  steps to the right, we are essentially multiplying the multiplicand by  $N$  to the power  $k$ , which is fine. That is exactly what we wanted to do.

So, let us consider the partial sums in the regular multiplication. So, if we consider the  $k$ th partial sum; right, let us consider the  $k$ th partial sum. So, essentially this partial sum is either 0 or it is the multiplicand multiplied with 2 to the power  $k$  because it has shifted  $k$  places to the left. So, here what we are doing is that we are essentially doing the same, but we are just keep changing the baselines and we are moving it to the right. So, it will appear that the multiplicand is first being displaced by  $k$  steps, then by  $k$  plus 1 and  $k$  plus 2 and so on; because this boundary over here is moving to the right. So, the multiplicand over here will first appear to be left shifted by  $k$  steps. In the subsequent step, once these boundaries move one step to the right, from this point the new multiplicand will look at shifted  $k$  plus 1 steps to the left and so on.

So, basically the crux of the issue is that instead of shifting the partial sums which contain the multiplicand 1111 steps to the left, it is a better idea in this scheme to keep the multiplicand where it is, but shift the accumulating partial product to the right. So, it is the same thing. Mathematically, it is a same thing. It is just that is easier to implement in hardware.

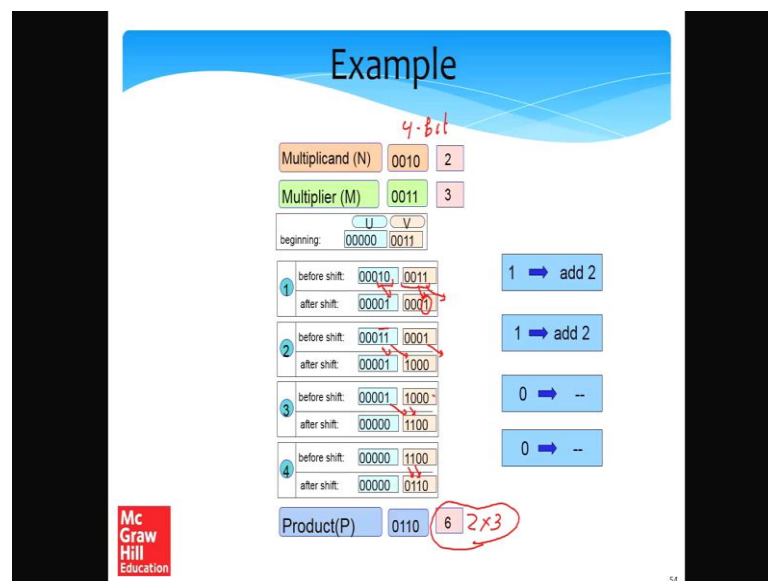
So, let us now consider the last iteration. So, the last iteration will pretty much bring us over here. The last iteration is somewhat special, the reason being that this is the signed number. And, so we are essentially we have the multiplicand on top and multiplier at the bottom. So, when we reach the last iteration, which is the thirty second iteration, if the LSB 0, we do not do anything, which is fine. But, if the LSB of V is 1; so LSB of V basically means that in the last iteration if I consider the register pair U and V, then only the last one bit of the multiplier will be left, which will be its MSB. So, the multiplier is negative, then the MSB will be 1. So, in this case what we are suggesting is that we do not follow this part of if loop, if statement. But, we come here. And, instead of adding a multiplicand to the partial product, we subtract the multiplicand from the partial product. That is all that we do. And, we do one more shift and we are done.

So, the question is why are we doing it. So, the reason that we are doing it we have to go back to this theorem over here, which essentially tells us that the number is. See, if I consider its two's complement representation, it is the same as an unsigned number that is formed by the first  $n$  minus 1 bits. And, subtract; and from this, we subtract  $A_n$  which is the MSB times  $2$  to the power  $n$  minus 1. So, since the MSB is 1, so basically  $A$  is equal to; when  $A$  is negative. So, when  $A$  is negative,  $A$  is equal to. So, assume that  $A$  is the multiplier and we are trying to multiply  $B$  times  $A$ , where  $B$  is the multiplicand. So, then what we will have is that we will have  $B$  times, which will happen in the first  $n$  minus 1 iterations. Then, we need to subtract  $B$  times  $2$  to the power  $n$  minus 1, which is a left shifted version of  $B$ . And,  $B$  is the multiplicand over here. And that is exactly what we are doing here. So, we are subtracting multiplicand from  $U$ . And, since in the last iteration, we have reached here.

So, effectively our; so, you can think of it that our base, our least significant bits position is over here. So, from here if I look at the multiplicand which is over here, then it appears to me that the multiplicand is left shifted by  $n$  minus 1 places. And, so what I am doing is in effect I am left shifting the multiplicand by  $n$  minus 1 places and I am subtracting it

from the accumulated partial product which is this value. So, this justifies this step that if the multiplier is negative, then in a last iteration, instead of adding multiplicand, instead of adding n to the partial product, which is essentially; I am adding it to register U. What I do is I subtract. These are very important thing should be kept in mind. And, a lot of time student actually make mistakes. But, the important point is that this has to be kept in mind that in this multiplication operation, in the last iteration, instead of adding the multiplicand to U to register U, we subtract it.

(Refer Slide Time: 64:35)



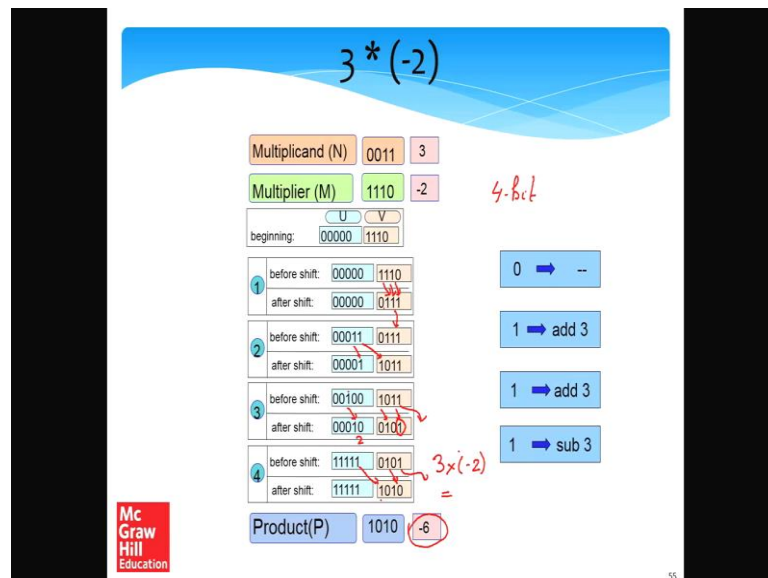
So, let us consider an example. Nothing is clear without an example. So, it has to be considered. And, let us see. So, let us assume that the multiplicand is the number 2; 0010. The multiplier M is the number 3, which is 0011. And, we have a 4-bit number system. At the beginning, the register V contains the multiplier which is 0011 and U contains 4 plus 1 five 0's just to take care of overflows.

So, let us consider two points, which is before the shift, let me call it before the shift. And, let us consider one more point which is the after the shift. So, before the shift we will have. So, so we will consider the LSB of the multiplier. So, the LSB is 1. So, we will add the multiplicand to register U, we will add 2. So before the shift, we will have 0010 because the multiplicand is added. And, we will have the multiplier which is 0011, which was already loaded at the beginning. Then, we will do a right shift. So, right shift

means that this bit will come here and this bit will come here and this bit will essentially fall off.

Then, let us come to the next iteration. So, what is the least significant bit of V? It is 1. So, if it is 1, what we will do is that again we will add 2, which is the multiplicand. So, 1 was there. We will add 10 plus 1 to make it 1 1. The rest remains the same. And, subsequently we shift. So, this bit will come here, this bit will come here and one will fall off. The rest of the bits of the multiplier are 00. So, we will not, nothing will happen before the shift. But, after the shift the bits will each moved by one position. And then, the same happens in the fourth iteration, where the bits move by one more position. So, at the end of four iterations because it is a 4-bit number system, the computation is done. The product P is 0110, which is 6, which is exactly what you would expect; 2 times 3 is 6. This is exactly what we would expect. And, how is this happening? We are considering each bit of the multiplier. And based on that, we are either adding the multiplicand to the partial product or we are adding or we are doing nothing, if the bit is 0.

(Refer Slide Time: 01: 07: 20)



Let us now consider the more complicated example, where the multiplier is negative. This is where, you know, lot of our understanding will be tested. So, let us look at it in some more detail. So, you first load 3 into register N, which is the multiplicand. So, this

is 0011. And, we load minus 2 into register M, which is also loaded into register V. And, minus 2 in a 4-bit number system is plus 14, which is 1110. Then, we start.

So, the first thing that we do is that we take a look at the LSB of V. So, LSB of V is 0. So, we do not do anything. So, before the shift we will have U as all 0es and V as 1110, then we shift. See, each of these bits we shifted one step to the right. Subsequently, we find the LSB of V to be 1. So, we add 3 to U. So, we add 3. So, we have 011 over here. And, we have the existing multiplier over here. So, we do one right shift. So, all the bits move one step to the right and then they remain there. After that, we take a look at the LSB of V once again. It is again 1. So, we again add 3 to register U. So, 3 plus 1 is 4. So, this becomes 100 and 1011. And, we do a right shift. So, all the bits again move one step to the right and 1 falls off.

Finally, we go to the fourth iteration which is the fun part. And, here we find that the least significant bit of V is 1. And, this is the last iteration. It is a 4-bit number system and it is a last iteration. Since it is a last iteration, instead of adding the multiplicand, we will subtract the multiplicand. So, basically what was the number we had before? We had before plus 2. Now, we subtract 3 from 2. So, then the result is minus 1 and minus 1 is 1111. So, this is what we get before the shift. And, after the shift, we, essentially 1 will fall off. This 1 comes here, this 1 comes here; the rest of the number get shifted and we have a sign extension.

So, if we take a look at the product, the product is basically 2 plus 8 is 10. And, in the 4-bit number system 10 is the same as minus 6. Did we expect that? Of course, we did. 3 times minus 2 is equal to minus 6. So, there you go. (Refer Time: 70:08) And, behold it works. And, it works, you know, fantastically well. So, here the only, the trick here was that in the last iteration which was the fourth iteration, instead of adding the multiplicand, we actually subtracted the multiplicand. And, the reason we did this is because of the following theorem which basically said that look, you treat a number. So, let us assume that.

Let us go with conventional terminology, where multiplicand is being multiplied with the multiplier. So, this can be written as; so the multiplier let us consider the first 1 to n minus 1 bit. So, that is one unsigned number. And, if the number is negative, then the MSB is 1. So, essentially in the first n minus 1 iterations, we multiply the multiplicand

with the first  $n - 1$  bits of the multiplier. So, this is the partial product at this, in a particular stage. After that, we need to subtract it with the multiplier shifted in a small  $n - 1$ . You know, there is a capital  $N$ , which is multiplicand and there is a small  $n$  which is a number of bits in a number system. So, we want to make this distinction. So, it is not the multiplier shift, it is the multiplicand shifted by small  $n - 1$  steps.

So, that is essentially what we want to ensure in the last iteration, so from the partial product, we need to subtract the multiplicand shift it by  $n - 1$  steps; which basically means that you take the partial product and we sort of  $U$  shift it  $n - 1$  steps. And then, you keep the multiplicand as it is and then we add it.

So, so we have been looking at, you know, different versions of, you know the same idea. But, the main reason that we do a right shift is basically to create a relative movement between the multiplicand and the partial product. So, always the multiplicand should appear; the next partial sum should appear one bit left shifted as compared to the previous partial sum. And, the way that this is ensured is basically by having a combined register. And, initially the partial product is only in  $U$ . And, the end of the partial product is over here. So, we keep on moving here  $n$  by 1, one place. So, we keep on moving the end. If we start looking at the multiplicand from the end of the partial product, it will appear shifted to the left by increasingly one position, which is exactly what we wanted to achieve.

(Refer Slide Time: 73:17)

**Operation of the Algorithm**

- \* Take a look at the **lsb** of  $V$ 
  - \* If it is 0 → **do nothing**
  - \* If it is 1 → Add  $N$  (multiplicand) to  $U$
- \* **Right shift**
  - \* Right shifting the partial product is the same as **left shifting the multiplicand, which**
  - \* Needs to be done in every step
- \* Last step is **different**

Mc Graw Hill Education

56

So, after seeing these examples, let us textually summarize the algorithm. So, we take a look at the LSB of V. If it is 0, we do nothing at all; if it is 1, we add the multiplicand to U. And, right shifting the partial product is the same as the left shifting the multiplicand which is done in every step, except the last step which is different.

(Refer Slide Time: 73:36)

**The Last Step ...**

- \* In the last step
  - \* lsb of V = msb of M (multiplier)
  - \* If it is 0 → do nothing
- \* If it is 1
  - \* Multiplier is **negative**
  - \* Recall :  $A = A_{1..n-1} - 2^{n-1}A_n$
  - \* Hence, we need to subtract the **multiplicand** if the msb of the multiplier is 1

57

In a last step, if the MSB of multiplier is 0; which means the multiplier is positive, we do nothing. If it is 1 means the multiplier is negative, so we use this relation over here. And, this number is equal to 1, which is the MSB of the multiplier. So, we basically subtract the multiplicand if the MSB of the multiplier is 1 from register U.

(Refer Slide Time: 74:03)

**Time Complexity**

- \* There are  $n$  loops
  - \* Each loop takes  $\log(n)$  time
  - \* Total time :  $O(n \log(n))$

Mc Graw Hill Education

58

So, what is the time complexity? So, the time complexity is there are  $n$  loops, so that it will take, you know, order and time; so, times of each loop. And, each loop what does it have? It has an addition or a subtraction. So, in addition or a subtraction takes  $\log n$  time. As we have seen from our previous discussion, we can use a carry look ahead adder. So, the total time that is required is order  $n$  times  $\log n$ , which is order of  $n \log n$ .

(Refer Slide Time: 74:35)

**Booth Multiplier**

- \* We can make our iterative multiplier faster
- \* If there are a continuous sequence of 0s in the multiplier
  - \* do nothing 😊
- \* If there is a continuous sequence of 1s
  - \* do something smart 🧠

$$M = \sum_{k=i}^{k=j} 2^k = 2^{j+1} - 2^i$$

Mc Graw Hill Education

59

So, given the fact that we have seen this iterative multiplier, can we make it a little bit faster? May be if not asymptotically faster, at least faster in practice, well, it turns out



that we can make our iterative multiplier faster. So, if there is a continuous sequence of 0's in the multiplier, we do nothing, which is good, which is in a very good for us that we do nothing at all. So, there is no addition or multiplication involved. However, if there is a continuous sequence of ones, then in iterative multiplier we keep on doing additions and subtraction at the end. Maybe we can do something smart. So, that is where you know when idea strikes us.

So, what is the idea? So, let us take a look at the simple identity over here. So, let us consider this. Let us consider the sum of powers of 2, 2 to the power k from k equals i to k equals j, where j is greater than i. So, in this case if I do a simple G, P summation, it will turn out that this is equal to 2 raised to the power j plus 1 minus 2 raised to the power i. So, this is a simple summation that you can do. So, 2 raised to the power j plus 1 minus 2 raised to the power i.

(Refer Slide Time: 76:04)

**For a Sequence of 1s**

- \* Sequence of 1s from position i to j
- \* Perform (j - i + 1) additions
- \* **New method**  $0 \dots 0 \underset{i}{\uparrow} 1 \dots 1 \underset{j}{\uparrow} 0 \dots 0$
- \* Subtract the multiplicand when we scan bit i (count starts from 0)
- \* Keep shifting the partial product
- \* Add the multiplicand(N), when we scan bit (j+1)
- \* This process, effectively adds  $[2^{j+1} - 2^i] * N$  to the partial product
- \* Exactly, what we wanted to do ...

$$A = \sum_{k=i}^{k=j} 2^k$$

$$N \times (2^i + \dots + 2^j)$$

Mc Graw Hill Education

So, this means that for a sequence of ones, from that, that are from position i to j, we typically perform j minus 1 additions. And, so basically we, you know, this is what we do that we just keep on performing additions. So, let us do a new method. So, new method is called a booth multiplication method. So, let us see. That when we scan bit i, so yes, so basically let us take a look at this. Let us assume we have a sequence of ones from, you know, bit position i to j in the multipliers, in the multiplier we have a sequence of ones.

So, when we scan bit position  $i$ . So, so assume that before that in the sequence of 0's (Refer Time: 77:13) of 0's, we suddenly have a sequence of ones from  $i$  to  $j$ . So, when we scan bit position  $i$ , let us subtract the multiplicand from  $U$ . And, let us keep then shifting the partial product as we do. And, when we scan bit position  $j$  plus 1, right, after  $j$  when we again see that again there is a transition from 1 to 0. Let us add the multiplicand when a scan bit  $j$  plus 1. So, since we right shift the partial product at every stage, in effect what it does is that it effectively adds  $2$  raise to the power  $j$  plus 1 because if you remember a previous discussion, we had said that the partial product occupies the register  $U$  entirely and a part of register  $V$ . And, in every iteration the LSB of the partial product keeps moving one place to the right. So, if we start looking from the LSB towards the multiplicand, it will look like it is moving one step to the left.

So, basically in  $i$  th iteration, since we are subtracting the multiplicand, we are subtracting  $n$  raised to the power  $2$  to the power  $i$ . And, in a  $j$  plus 1 th iteration, since we are adding the multiplicand, it looks like we are adding  $2$  to the power  $j$  plus 1 times  $n$ . So, in effect we are adding this number to the partial product. Well, this is exactly what we wanted to do. I mean, in this case when we have a continuous sequence of ones from  $i$  to  $j$ , what we are adding to the partial product is essentially summation of  $2$  raise to the power  $i$  till  $2$  raise to the power  $j$  multiplied by the multiplicand, which is the multiplicand times. So, summation of this as we have seen is  $2$  raise to the power  $j$  plus 1 minus  $2$  raise to the power  $i$ . So, this is exactly what we wanted to do.

So, the insight is that if we have, let us say a string of ones. And, this is 0 at the beginning and the 0 at the end. Then, so basically this is what we are trying to say that even if let us say that we are looking at the first bit position. And, the first bit position is 1. We will assume that just before it there was, there is a hypothetical 0. So, this is what this line means. So, the moment we see a 1 and we see a continuous run of ones between positions; let us say  $i$  and you know position  $j$ . And, it also says that the count starts from 0; which means that if, you know, the first position, then we will have  $2$  raise to the power 0. But, that is position  $j$  and  $i$ .

In effect to the partial sum, we are adding  $n$  times. You know, at this point it is left shifted by  $i$  positions and then plus plus  $j$  positions. There are little bit of algebra. This is what we get, which is same as subtracting it once and adding it once more. Subtracting it

once is the  $i$ th position and adding the multiplicand once more at the  $j$  plus 1th position to the register  $U$ .

(Refer Slide Time: 80:41)

**Operation of the Algorithm**

- \* Consider bit pairs in the multiplier
  - \* (current bit, previous bit)
  - \* Take actions based on the bit pair
  - \* Action table

Handwritten bit strings:  $0110$  and  $0110$

(current value, previous value)	Action
0,0	×
1,0	subtract multiplicand from $U$
1,1	×
0,1	add multiplicand to $U$

Mc Graw Hill Education

So, the operation of this algorithm is fairly simple. Let us consider bit pairs in our multiplier; a current bit and previous bit pair. And, we take actions based on the bit pairs. So, we have an action table current value and previous value. So, if the current bit is 0 and the previous bit is 0, well nothing needs to be done. We are in the run of 0's. Absolutely nothing needs to be done. If the current bit is 1, then the previous bit is 0 means a run of ones is starting.

So, as discussed on a previous slide we subtract the multiplicand from  $U$ . Then, if you are in the middle of a run of ones, so current bit is 0 and the previous bit is, sorry, current bit is 1 and previous bit is 1. We do not do anything. And, again when there is a transition from a 1 to a 0, then we add the multiplicand to  $U$ . See, in effect what we are doing is that we are adding this quantity to  $U$ , which would have been the same as you know  $j$  minus  $i$  plus 1 repeated additions. Instead of doing this, so we are essentially using this identity over here, which is saying that instead of using  $j$  minus 1 repeated addition, we can subtract ones and add ones, as simple as that. That will save us some work. Asymptotically no because in the worst case you can always have a multiplier that is of the form 01010101 but, in most cases if there is a continuous run of 0's or a continuous run of ones, we can really save a lot of effort.

(Refer Slide Time: 82:18)

### Booth's Algorithm

**Algorithm 2:** Booth's Algorithm to multiply two 32 bit numbers to produce a 64 bit result

**Data:** Multiplier in  $V$ ,  $U=0$ , Multiplicand in  $N$   
**Result:** The lower 64 bits of  $UV$  contain the result

```
i ← 0
prevBit ← 0
for i < 32 do
  currBit ← LSB of V
  if (currBit, prevBit) = (1,0) then
    U ← U - N
  end
  else if (currBit, prevBit) = (0,1) then
    U ← U + N
  end
  prevBit ← currBit
  UV ← UV >> 1 (arithmetic right shift)
end
```

62

So, let us take a look at the Booth's algorithm, which uses exactly the same hardware plus a little bit of extra circuitry. But, the hardware is the same as iterative algorithm. So, we multiply two 32 bit numbers to produce a 64-bit result. The multiplier is in  $V$  and  $U$  contains a 0. The multiplicand is in  $N$  and lowest bits of  $U$ , 64 bits of  $UV$  will contain the result. So, let us do the same thing. Let us set  $i$  to 0 and let us iterate for thirty two times, each times increasing  $i$ .

Let us consider, let us start with by considering as mentioned before the previous bit to be 0. And, let the current bit be the LSB of  $V$ . The current bit that we are considering of the multiplier let us take a look at the total of the current bit and previous bit. So, if a run of one is; just if a run of ones is just starting, you know, in the multiplier 00 and 1. So, we are making a transition from 0 to 1. This is the time to do a subtraction;  $U$  is  $U$  minus  $N$ .

Similarly, if runner ones is just ending. So, we have runner ones over here. And, this is just ending. Then, what we do? From 1, we are moving to 0. Current bit is 0, previous bit is 1. So, as discussed before we add the multiplicand to  $U$ . So, in the first case we will subtract. So, in a runner one is the beginning and in the second case we will add. Then, we need to set previous bit to the current bit and then perform a right shift as was done before and keep going on.

So, let us try to outline a proof, but again the proof is complicated. So, I will not go into the details of the proof. And, the readers are expected that you want to know more about the proof, which anyway star marked inside the book. So, you should take a look at the proof inside the book. So, the proof is difficult. But, I will tell you what makes the proof difficult. The fact that makes the proof difficult is like this that let us assume that the multiplier is a number of the form 1110. So, after that, you know, we can have any combination of bits; does not matter. So, which means the multiplier is essentially negative. So, when we come at this point, we are making a transition from 0 to 1. So, we subtract the multiplicand, which is this line over here. And then, we just keep going. We just keep going on and on and on. Finally, we reach the last bit. And, so the runner ones is continuing. So, we do not get a chance to actually perform this addition, and then we finish.

So, we need to say that for sign two's complement numbers. This is absolutely fine. In the sense that we can do a subtraction over here and, essentially never do the addition, that is exactly what is happening. If we have a runner ones and the runner ones continue, continues till the most significant bit, then we will do a subtraction. But, we will never get the opportunity to actually do an addition. And, when it is say in a two's complement world, this is fine. So, this requires a little bit of algebra. And, it is somewhat difficult. But, we shall see examples of this working.

So, let me explain the insight once again. I will give a brief overview of the proofs. But, I will be very very brief because the proofs are complicated. And, an online lecture is probably not the best place to discuss the proofs of the Booth's algorithm. But, we will discuss the examples.

So, the main insight is this; a loop. In a multiplier, if we have 0's we do not do anything. So, that is the best thing for us. That if a number is 0, it essentially saves us effort. But, if a number is 1 in iterative multiplier, we are either doing an addition or a subtraction which used to take effort. It is used to take  $\log n$  time.

So, since this was taking effort, we said what happens if we have several consecutive ones. So, let us say we have, you know,  $k$  consecutive ones separated by 0's. Then, in the case of an iterative multiplier, we are essentially doing  $k$  additions, which was a lot of work. What we are saying is that it is not required. What we can instead do is that we can

do one subtraction at this point and, one addition at this point; that is equivalent to actually doing  $k$  additions because it is a simple sum of a geometric series. So, instead of adding, you know,  $k$  numbers, we can essentially do one subtraction and one addition. And, we are done. So, that is the reason we look at transitions from 0 to 1 and 1 to 0. And, if the transitions are fine, you know, if the transitions are there, then there is nothing much that we need to do.

We just need to work on whenever the transitions are there in 1, 0 to 1 transition; we need to subtract the multiplicand in a 1 to 0 transition. We need to add the multiplicand. That is pretty much the only thing that we need to do. But, proving it for two's complement sign number is a different issue altogether.

(Refer Slide Time: 88:06)

The slide is titled "Outline of a Proof" and contains the following content:

- \* Multiplier (M) is **positive**
  - \* msb = 0
  - \* Divide the multiplier into a **sequence of continuous 0s and 1s**
    - \*  $01100110111000 \rightarrow 0, 11, 00, 11, 0, 111, 000$
  - \* For sequence of 0s
    - \* Both the algorithms (iterative, Booth) **do not add the multiplicand**
  - \* For a run of 1s (length  $k$ )
    - \* The **iterative algorithm** performs  $k$  additions
    - \* **Booth's algorithm** does one addition, and one subtraction.
    - \* **The result is the same**

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

So, let me give a very quick overview. So, assume that the multiplier  $M$  is positive. If the multiplier  $M$  is positive, then so the way the proof works is that we try to match the state between iterative algorithm and the Booth's algorithm. And, we shall see that since at the end, you know, things the MSB is 0. So, all the runs of ones are in a sense taken care of because we do a subtraction and then we do an addition.

(Refer Slide Time: 88:37)

Outline of a Proof - II

- \* Negative multipliers
  - \* msb = 1
  - \*  $M = -2^{n-1} + \sum_{(i=1 \text{ to } n-1)} M_i 2^{n-1} = -2^{n-1} + M'$
  - \*  $M' = \sum_{(i=1 \text{ to } n-1)} M_i 2^{n-1}$
- \* Consider two cases
  - \* The two msb bits of M are 10
  - \* The two msb bits of M are 11

Mc  
Graw  
Hill  
Education

64

So, the main problem comes when we have negative multiplier. So, then we further divide it into two sub cases. So, when the MSB bits of the multiplier are 10 and when they are 11. And, so, I will not discuss these because this is fairly complicated.

(Refer Slide Time: 88:51)

Outline of a Proof - III

- \* Case 10
  - \* Till the (n-1)<sup>th</sup> iteration both the algorithms have **no idea** if the multiplier is equal to M or M'
  - \* At the end of the (n-1)<sup>th</sup> iteration, the partial product is:
    - \* **Iterative algorithm** : M'N
    - \* **Booth's algorithm** : M'N
  - \* If we were multiplying (M' \* N), **no action** would have been taken in the last iteration. The two msb bits would have been 00. There is **no way to differentiate** this case from that of computing MN in the first (n-1) iterations.

Mc  
Graw  
Hill  
Education

65

(Refer Slide Time: 88:57)

## Outline of a Proof - IV

- \* Last step
  - \* Iterative algorithm :
    - \* Subtract  $2^{n-1}N$  from U
  - \* Booth's algorithm
    - \* The last two bits are 10 (0 → 1 transition)
    - \* Subtract  $2^{n-1}N$  from U
  - \* Both the algorithms compute :
    - \*  $MN = M'N - 2^{n-1}N$
    - \* in the last iteration

66

And so, but the main idea is that it is possible to prove that in the case, the Booth's algorithm works correctly. So, when the multiplier is negative, it is represented in the two's complement fashion. But, how it is done? It is better to take a look at the book. And, even in the book the proof is fairly long a couple of pages, if I remember correctly.

(Refer Slide Time: 89:20)

Multiplicand (N)	00011	3
Multiplier (M)	1110	-2
beginning: 00000 1110 0		
1	before shift: 00000 1110 after shift: 00000 0110	00 → --
2	before shift: 11101 0111 after shift: 11110 1011	10 → add -3
3	before shift: 11110 1011 after shift: 11111 0101	11 → --
4	before shift: 11111 0101 after shift: 11111 1010	11 → --
Product (P)		1010 -6

70

So, let me skip that. But, let us take a look at the example which is a nice and fun part. So, in the example, let us multiply 3 with 2. So, the multiplicand N is 3; 0011. And, multiplier is 2; 0010. So, let us again consider two points before the shift and after the



shift. So, the initial previous bit is 0. So, basically the state that we see here is the LSB of  $V$  is 0 and the previous bit is 0. So, the current bit, previous bit pair is 00. So, essentially we do not do anything, but we shift. So, the 1 will come over here. That is all.

Subsequently, what we do is that we look at the current bit, previous bit pair, which in this case is 1 0. So, we see a 0 to 1 transition. So, in this case what we do is we add we essentially subtract the multiplicand; which means that we add minus 3 and minus 3 would be this in a 5-bit representation. Why because you just add plus 3 to it. So, you will clearly see that the result become 0. And, so this is what minus 3 is. So, basically we can also see for a 4 bit minus 3 will be 1101. And, we are just extending its sign. So, this is minus 3. And, subsequently we do a shift and we maintain the sign bit. So, we perform arithmetic right shifts. So, the one bit is thrown out and the rest of the bits get shifted by the one position to the right. Exactly the same as the iterative algorithm and, this is the sign extended bit.

Now, the least significant bit is 0 and the previous bit is 1. So, there is basically again a transition from 1 to 0. So, in this case we add 3. We add 3. See if we add 3, then we are adding 0011; 1 plus 0 is 1. Then, when we add 1 to these string of four ones, we will get four 0's. And, the rest remains the same. So, then we do a right shift. So, the current state of the; so, the current bit and previous bit are now 00. So, nothing needs to be done. We do one more right shift and the product that we end up with is 0110. 0110 is 6. And, this is exactly 2 times 3. So, this is exactly what we wanted. So, two times 3 is 6. And, this algorithm is faster; the reason this algorithm. Well. So, so one thing is we did not see a run of ones. So, we are not able to see the power of it. But, had there been a run of ones, we would have definitely seen this algorithm; you know the Booth's algorithm to be faster.

So, let us now consider multiplier with a run of ones and also the multiplier is negative. So, this represents a fairly complicated case for us. So, let us start the way that we should start by first loading the multiplier into  $V$ . So, the multiplier is minus 2 in this case. So, it is 1110. We load all 0's into  $U$ . And, before the shift we have  $u$  as 0000 and  $V$  as 1110. The reason being that the previous bit and current bit pair is 0 0. So, since it is 00, we do not do anything. We do not do anything at all. So, we just do a right shift. So, all the numbers here gets shifted to the right by one position. Subsequently, we need to take a look at the least significant bit of  $V$ , which is the current bit 1 and the previous bit is 0.

So, our current bit, previous bit pair is 10; which means that there is a transition from 0 to 1. So, given a transition, what we need to do is that we need to subtract the multiplicand. In this case, add minus 3. So, adding minus 3 to 0000 would be the representation of minus 3 in a 5-bit system, which is 111101. Why is this the case because this is; so, in a 5-bit system minus 3 will be actually plus 29. So, this is 1 plus 4 is 5, plus 8 is 13, plus 16 is 29. And, once after doing this, after the shift, we just shift it. So, this 1 falls off to the right. We have all the numbers shifted to the right by one position and we extend the sign. So, the sign bit is 1. So, we just extend it and we add a 1 over here.

Now, let us take a look at the current bit and previous bit pair. So, in this case it is 1 1. Since it is 11, nothing needs to be done. So, this is exactly where we are saving work. So, we just need to do a shift. So, let us do a shift. This 1 will again fall off and each number is shifted by one place to the right. And, there is sign extension. Again, let us take a look at the LSB. The LSB is 1 and the previous bit is 1. So, again the current pair of bits is 11. So, nothing needs to be done. We just need to do a shift; one arithmetic right shift. So, this is where we end up at.

So, the final product is when if you consider a 4-bit system, it is 1010. And, 1010 is minus 6. And, we consider a 8-bit system; it is basically 1010 extended, it is sign extended, which is still minus 6. Is minus 6 the correct answer that we expect? Of course, this is what we expect; 3 times minus 2 is minus 6.

But, let me tell you the advantage of applying the booth multiplier. So, had it been an iterative multiplier, we would have performed two additions here and one subtraction. So, if the addition and subtraction take more or less the same amount of work. So, essentially we would have performed the work of three additions because there are three ones. But, the interesting thing of the greatness of the Booth's algorithm is that in this case we took the advantage of the runner ones and we just performed one addition. If we esteemed that the addition takes most of the time, so this algorithm is faster by a factor of three. So, it is three x faster, which is great, which is fantastic. That is exactly the kind of speed ups that are required in modern high performance adders.

Also, the other thing that you need to note is that the two's complement system got taken care of automatically. So, in this case the multiplier is negative. And, so we did not have

to do anything special and it is just the runner ones ended and we finished the algorithm. So, we did not have to do anything else. So, the product was, you know all set for us; minus 6. So, the fact that nothing needs to be done is something that should be proved. And, we have proven that in the book. And also, those who have read the proof in the book can actually also read the last five slides that will sort of help you refresh the knowledge of the proof. Otherwise, the proof is somewhat involved. But, in any case the important point is that a Booth's algorithm takes advantage of a run of ones and can has the potential of significantly reducing the numbers of add and subtract operations.

(Refer Slide Time: 97:21)

**Time Complexity**

- \*  $O(n \log(n))$
- \* Worst case input
- \* Multiplier = 10101010...10  $\log(n)$

71

Well, what is the time complexity? Well, the worst case input is 1010; which means all the time there is a transition. So, we will have to do an add or subtract. So, the time complexity per iteration still remains  $\log n$ . Well, that is how much an add or subtract takes. And, if there are  $n$  iterations, it is still  $n \log n$ . So the worst case, asymptotic complexity still remains the same. But, in practice we will always have numbers that have some runs of ones.

So, in practice we will definitely see some speed up. So, this is pretty much where, you know, the asymptotic notation fails to show of its benefits. In any case, the Booth's multiplier is popular. It is heavily used, and it is particularly heavily used in small embedded processors. So, that is where the algorithm finds a lot of utility and a lot of use. So, now let us look at substantially faster multipliers that can in a work in order  $\log$

$n^2$  or a  $\log n$  time. So, these multipliers will find that used in very very high performance implementation. So, let us look at it next.