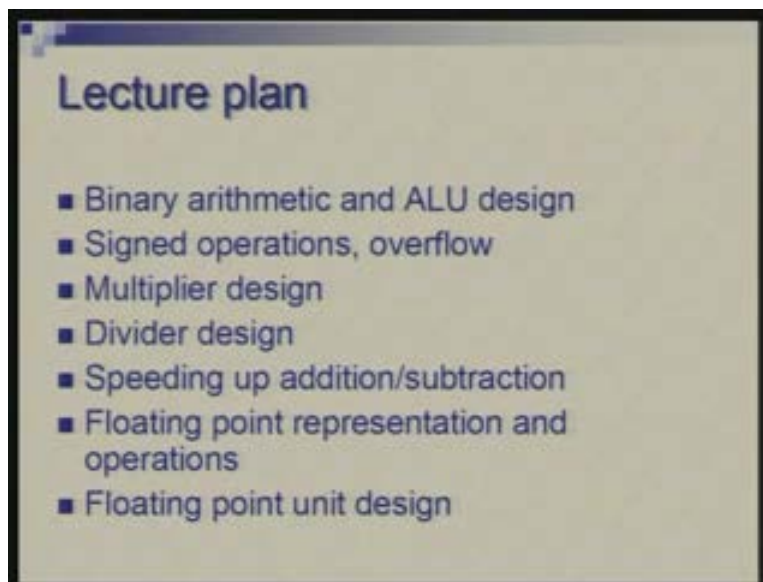


**Computer Architecture**  
**Prof. Anshul Kumar**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology, Delhi**  
**Lecture - 14**  
**Divider Design**

Today we will discuss how binary division is carried out and how we build the circuit to carry that out for the processor. We have been discussing various arithmetic operations and logical operations and we have discussed the design of ALU.

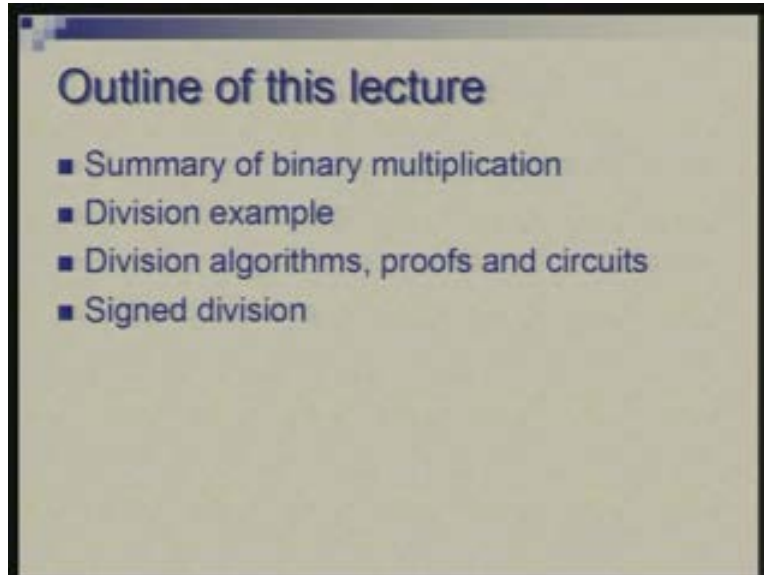
(Refer Slide Time: 01:15 min)



Lastly we discussed how we carry out multiplication and we talked about different designs of multipliers. We will continue with that talk of binary division and design of divider.

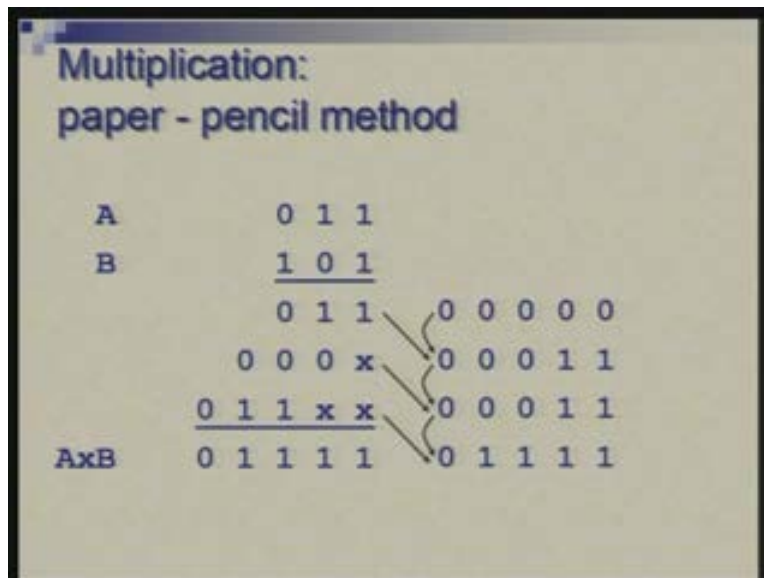
First of all, I will summarize how we did multiplication because there is some overall similarity between how multiplication and division is carried out; and we will illustrate division operation by an example, by taking two numbers, the same way we did for multiplication then from that we will derive the algorithms. We will ensure that the algorithm we are talking of are correct and discuss the circuits. Initially we will talk of unsigned division and then talk about signed division.

(Refer Slide Time: 02:00 min)



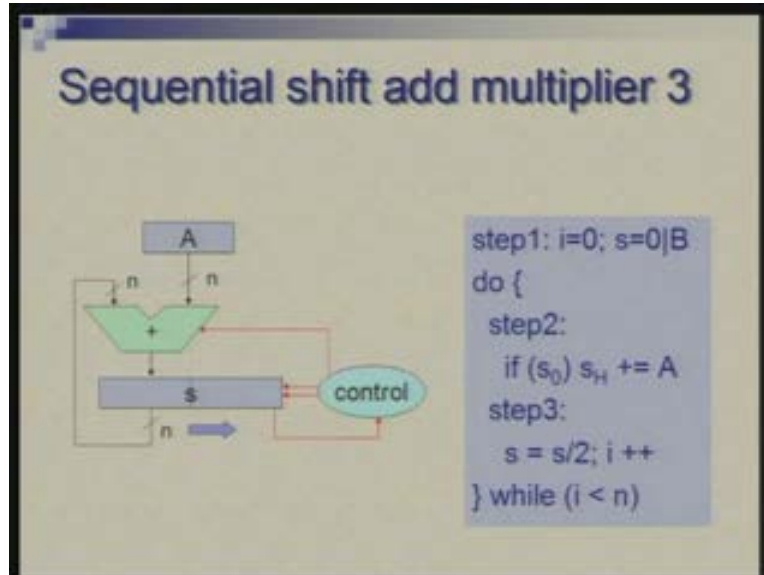
So, just to summarize what we did for multiplication; we looked at an example and saw that when you work on paper how you manually carry out multiplication. So the same process was captured by a circuit.

(Refer Slide Time: 02:20 min)



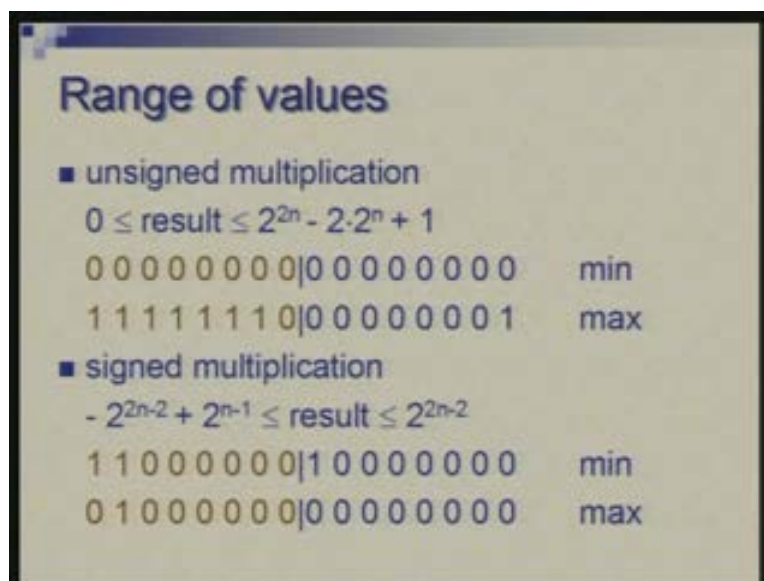
We went through a set of refinements and finally we had a circuit in algorithm which looked like this.

(Refer Slide Time: 2:35)



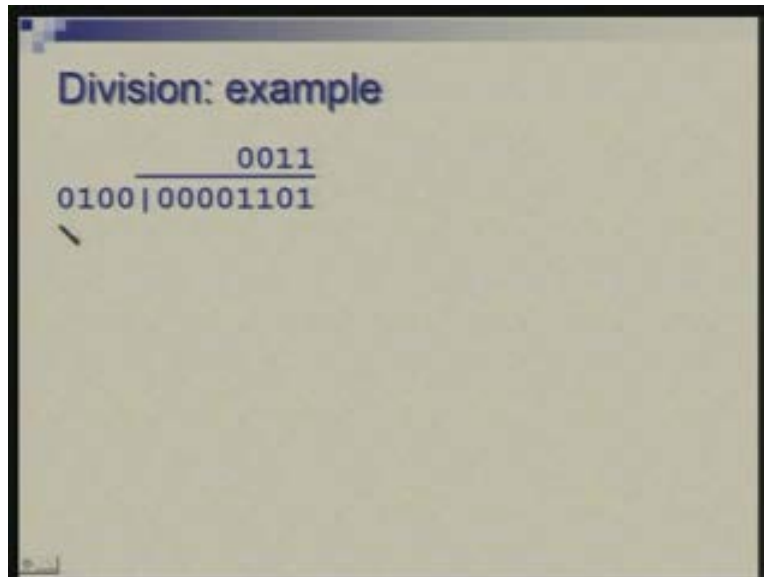
Essentially the multiplication operation was broken down into shifting and adding. So we then saw that for signed multiplication you need the ability to add and subtract and we talked about the Booth's algorithm. We also discussed the range of values which could result when you multiply. There are two types of multiplication in MIPS which try to produce signed result or unsigned result and there are also pseudo instructions which work with a single word output. Normally when you are multiplying two one-word integers the output could be accommodated in two words so there are pseudo instructions which will check if the output is exceeding one word and accordingly give the overflow.

(Refer Slide Time: 02:57 min)



Now let us start with a division by taking two small 4 bit numbers and try to see how division is carried out. Let us say we have a number 00001101 which stands for 13 how that is carried out, how that is divided by a number 0100 which is four in terms of its binary equivalent.

(Refer Slide Time: 4:07)



We will expect that when you divide 13 by 4 you get three as the quotient so in anticipation I have put that here and we should be left with 1 as a remainder. Naturally this division would be carried out as a set of shift and subtract operations. So initially we see if **this multiplier** this divider can be placed in this position here we can subtract. So we examine this position (Refer Slide Time: 4:52) because this will correspond to a bit in the LSB position for the quotient.

We are now targeting for a 4-bit quotient we have a 4-bit divider and if you are able to subtract 0100 in this position then it will result in a 1 here but in this case we cannot because this is larger in this position we are only subtracting a 0 and we place a 0 bit as part of the result. So, after subtraction we still have the same number and the next time we try subtracting in this position it leads to another 0 bit for the result and there is no change here. So we get 0011; I am just dropping of zeros on the left side which are insignificant and then we try subtracting in this position. And indeed now we can subtract so we get 1 in the quotient and after subtraction we get 00101 then finally we subtract in this position which is right justified and this result is 1 in the LSB of the quotient and finally there is a remainder 1.

(Refer Slide Time: 06:17 min)

Division: example

$$\begin{array}{r} \phantom{00}0011 \\ 0100 \overline{) 00001101} \\ \underline{0000} \phantom{01} \\ 0001101 \\ \underline{0000} \phantom{01} \\ 001101 \\ \underline{0100} \phantom{01} \\ 00101 \\ \underline{0101} \phantom{01} \\ 0001 \end{array}$$

So we carried out these four steps. At each step we try to see if we can subtract or not. If subtraction is possible subtraction is carried out and 1 gets regarded in the result. If subtraction is not carried out then 0 gets regarded in the result. That is a very simple mapping of normal decimal division we are used to in binary domain. We require basically comparison. Before we can subtract we compare, if necessary we subtract and we record a 0 or 1 and from step to step we shift the position of the divider. So, in a nutshell it is result.

**you would notice** Let us call this as A which is the dividend, divisor is B, quotient is Q, remainder is R and the values which are being subtracted is nothing but multiples of B the divisor multiplied with powers of 2 and weighted with either a 0 or 1. So, in the first position it was B multiplied with 2 raised to the power 3 which we try to subtract from here so you could see that B has been shifted 3 bits to the left in relation to A. And in position number 3 counting it as 0 1 2 3 in Q we record the result as bit 0.

(Refer Slide Time: 07:50 min)

**Division: example**

$$\begin{array}{r} \text{0011} \text{ --- Q} \\ \text{0100} \overline{) \text{00001101}} \text{ --- A} \\ \underline{\text{0000}} \text{ --- } 0 \times B \times 2^3 \\ \text{0001101} \\ \underline{\text{0000}} \text{ --- } 0 \times B \times 2^2 \\ \text{001101} \\ \underline{\text{0100}} \text{ --- } 1 \times B \times 2^1 \\ \text{00101} \\ \underline{\text{0101}} \text{ --- } 1 \times B \times 2^0 \\ \text{0001} \text{ --- R} \end{array}$$

So next time the attempt was to subtract B into 2 raised to the power 2 and then B into 2 raised to the power 1 and B into 2 raised to the power 0 and the last value being subtracted is actually 0100 and not 0101.

What we are trying to get is given A and B we are trying to find Q and R which will satisfied this.

(Refer Slide Time: 8:18)

**Unsigned Division**

$$A = Q \times B + R$$

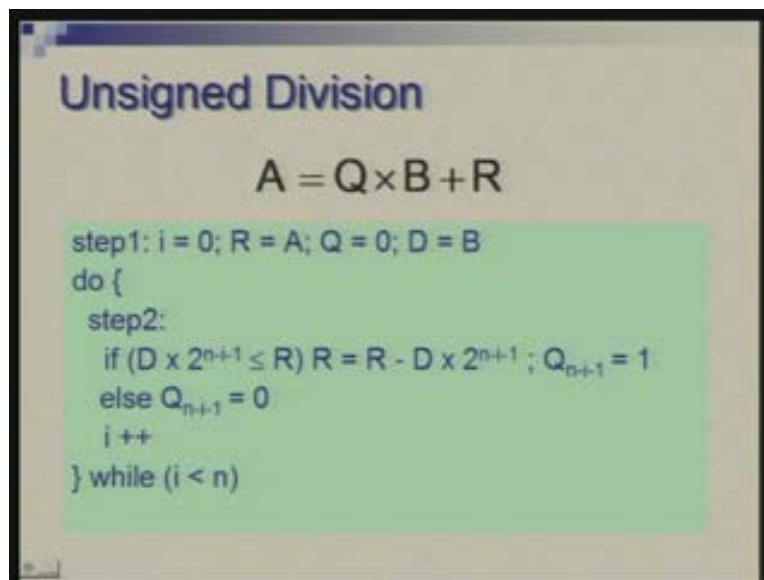
step1:  $i = 0$ ;  $R = A$ ;  $Q = 0$ ;  $D = B$

So we take a register R which we initialized with A. A is the dividend and from this we will keep subtracting. Hopefully what will be left in A left in R would be the remainder.

The quotient Q initially is put as all 0s and another register D which will hold the value B which is the divisor. Then there is a loop where we repeat this comparison and subtraction step. So, if D multiplied by appropriate power of 2 is less than R. So this is D positioned at appropriate position. So you would notice that when i equal to 0 in the beginning in the first step you need to displace it by n minus 1 position. So with i equal to 0 the power of 2 is 2 raised to the power n minus 1 so that is it.

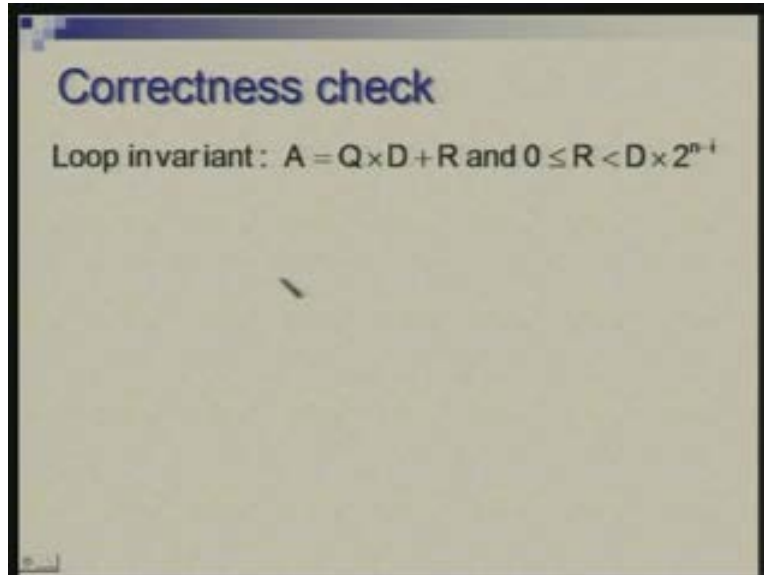
In our case n was four so we are shift we have shifted the divider by three positions to the left. We are comparing it with R and if it holds then R is replaced with R minus D into the same factor and Q n minus 1 minus 1 that particular bit of Q is recorded as 1. If the condition does not hold we do not really have to carry out..... either you say subtract 0 or you just leave as it is and the corresponding bit of the quotient is recorded as 0. Simply then step up i and repeat it for n steps. Is this clear?

(Refer Slide Time: 10:06 min)



This is our basic unsigned division algorithm. We will analyze it to make sure that what we are doing is correct and then we will do some modifications and improvements, transformations and implement this in the form of a circuit. So we want to make sure that what we are doing actually computes what we want. Our requirement is that ultimately..... there is a loop which is executed and we will ensure that there is certain invariant it maintains. We will ensure that Q into D plus R is always equal to A.

(Refer Slide Time: 11:15)



Therefore, as the loop proceeds this will be ensured and also we will ensure that  $R$  remains within this limit. So intuitively we have put up this invariant and first of all we must make sure that this invariant is a useful one which will give us the required result. So, at the end of the program when the loop ends I would have reached a value of  $n$  and if we put  $n$  in this invariant put  $i$  equal to  $n$  in this invariant this will ensure that the remainder we are getting is less than that divider.

You see, this unique, this equality  $A$  equal to  $Q D$  plus  $R$  can be satisfied by many  $Q$  are combinations. We are given  $A$  and  $D$   $D$  is initially  $B$  and we are finding  $Q$  and  $R$  quotient and the remainder. So this is a single equation with two given values and they are multiple  $Q R$  combinations which can be satisfied. We want the one where  $R$  is within  $0$  and  $D$ ; the remainder should be less than the divider and it should be positive or non-negative as I would say.

So, with  $i$  equal to  $n$  this condition will be ensured. So, at the end of the program if you are ensuring this loop invariant then we have the right quotient and remainder and remainder is within the range we expect it to be. Therefore  $R$  is the correct remainder and  $Q$  is the correct quotient. So **this is the** this is to show that the loop invariant we are talking of is a meaningful one; it will actually ensure that we are getting a correct result.



(Refer Slide Time: 13:14 min)

**Correctness check**

Loop invariant:  $A = Q \times D + R$  and  $0 \leq R < D \times 2^{n-i}$

At the end of the program,  $i$  has a value  $= n$

For  $i = n$ , the loop invariant  $\Rightarrow 0 \leq R < D$

Therefore  $R$  is the correct remainder and  $Q$  is the correct quotient

Now let us prove it inductively. First we will ensure that this holds in the beginning. When  $i$  equal to 0 when you start, after initialization, this holds or not. So our step 1 which is initialization is making  $i$  equal to 0,  $R$  equal to  $A$ ,  $Q$  equal to 0 and  $D$  equal to  $B$ . So, if you put these values then we can initially be sure that this holds because  $Q$  is 0 and  $R$  equal to  $A$  which is ensured by this initialization so this holds. In this (Refer Slide Time: 14:04) if you put  $R$  equal to  $A$  and  $Q$  equal to 0 it is satisfied. And also, since  $i$  is 0 this relationship that **A is less than** if  $A$  is less than  $B$  into 2 raised to the power  $n$  then this will also hold.

(Refer Slide Time: 14:30 min)

**Base case**

Loop invariant:  $A = Q \times D + R$  and  $0 \leq R < D \times 2^{n-i}$

step1:  $i = 0$ ;  $R = A$ ;  $Q = 0$ ;  $D = B$

This ensures the truth of the loop invariant initially, provided  $A < B \times 2^n$

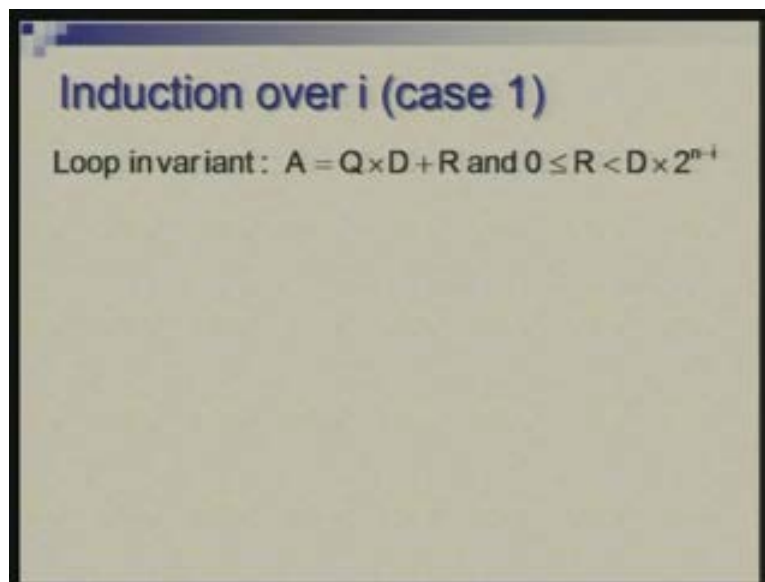
This condition is met if we assume that  $B \neq 0$  and  $A$  is contained within  $n$  bits.

Now how do you meet this condition?

Initially R is same as A and D is same as B and i is 0. So, in order to ensure this what we need is A should be less than B into 2 raised to the power n. So, if we assume that A is n-bit number and B is not 0 so B is at least 1 then A is less than 2 raised to the power n or A is less than B into 2 raised to the power n. Therefore, this is satisfied if you choose A as an n-bit number although **we can** in the illustration which I took I had taken 8 bits for the dividend and that was with the intuition that when you multiply two 4 bit numbers you will get 8 bit product. So I started with an 8-bit dividend and tried dividing it with 4 bit divisor. But there are cases where this may not work. So **we are** to begin with we are talking of a specific case where A is a 4 bit number or n-bit number in general. That means the left n bits will be 0s.

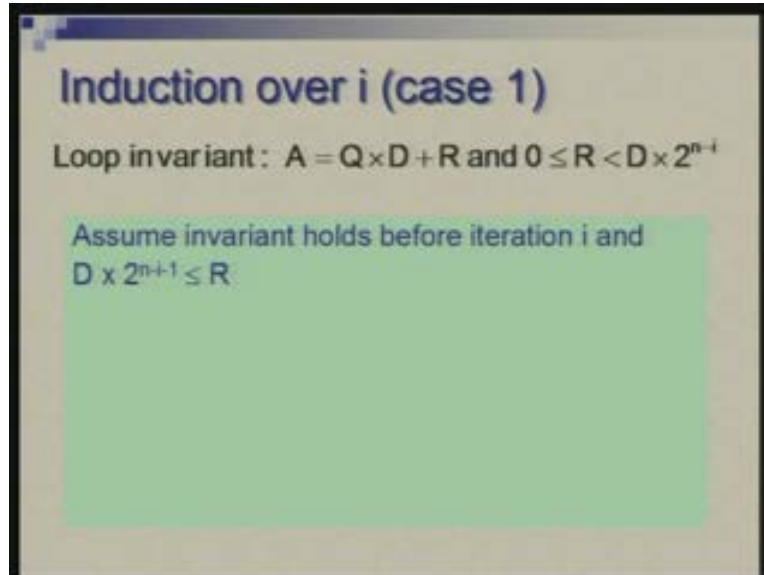
Let me go back to this (Refer Slide Time: 16:04) although I had the right value I was taking I was working with 8 bits but I had the right values because I had left 4 bits as 0. So what could happen is if you have A larger then algorithm may not work. We are trying to.... what we are learning here is that we will ensure that this algorithm works correctly if A is contained within n bits.

(Refer Slide Time: 16:52)



Now let us try to show this in general in the inductive manner. So what we will do is we will use mathematical induction; assume that before a particular iteration this holds we will show that it holds after the iteration that means it will hold throughout. So there will be two cases because in the algorithm we have a condition which is being checked and the action depends upon that. So we will take the first case.

(Refer Slide Time: 17:21)

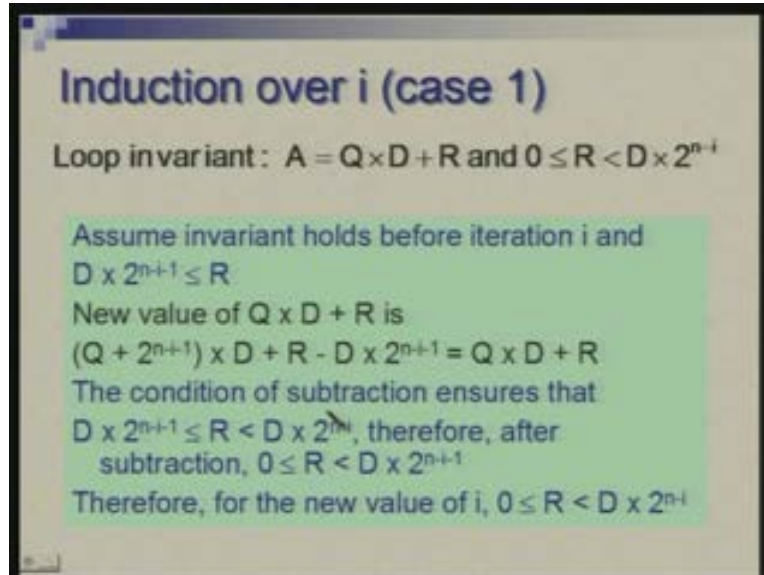


Here we assume that the invariants hold before the iteration  $i$  and we also assume this condition  $D$  into  $2$  raised to the power  $n$  minus  $1$  minus  $1$  is less than equal to  $R$  which means subtraction will be carried out, so this is one case. So what happens in this case is what would be examined now. So, given to us is that this condition holds, this inequality holds (Refer Slide Time: 17:51) and we know that under this condition subtraction will be carried out. So the new value of **Q plus**  $Q$  into  $D$  plus  $R$  is given by this where we have value of  $Q$  updated, we are setting this particular bit in  $Q$  and  $R$  has changed because we have carried out subtraction. So  $Q$  has become  $Q$  plus  $2$  raised to the power  $n$  minus  $i$  minus  $1$  and  $R$  has become  $R$  minus  $D$  into this power of  $2$ . So the new value of this expression  $Q D$  plus  $R$  is given by this. So, if you expand this you will be again left with  $Q$  into  $D$  plus  $R$ . What it means is that at least this equality holds even after the iteration.

We can also check if this inequality also holds. Thus, the condition of subtraction is ensuring that  $D$  is less than this and we know that  $D$  into this greater than  $R$  which means that  $D$  into  $2$  raised to the power  **$n$  minus  $1$**   $n$  minus  $i$  minus  $1$  is less than equal to  $R$  less than equal to  $D$  into  $2$  raised to the power  $n$  minus  $i$ . So the left part of the inequality is ensured by our assumption and the other part is ensured by noticing that invariant holds before the iteration.

Therefore, now  $R$  lies in this range and after the iteration once you subtracted this quantity from  $R$  (Refer Slide Time: 19:44) you can see that  $R$  is going to be lying in this range. So  $R$  was in this range and from both sides you can subtract this quantity and you can say that, now, after the iteration,  $R$  lies in this range.

(Refer Slide Time: 20:00 min)



**Induction over i (case 1)**

Loop invariant:  $A = Q \times D + R$  and  $0 \leq R < D \times 2^{n-i}$

Assume invariant holds before iteration i and  $D \times 2^{n-i+1} \leq R$

New value of  $Q \times D + R$  is  
 $(Q + 2^{n-i+1}) \times D + R - D \times 2^{n-i+1} = Q \times D + R$

The condition of subtraction ensures that  $D \times 2^{n-i+1} \leq R < D \times 2^{n-i}$ , therefore, after subtraction,  $0 \leq R < D \times 2^{n-i+1}$

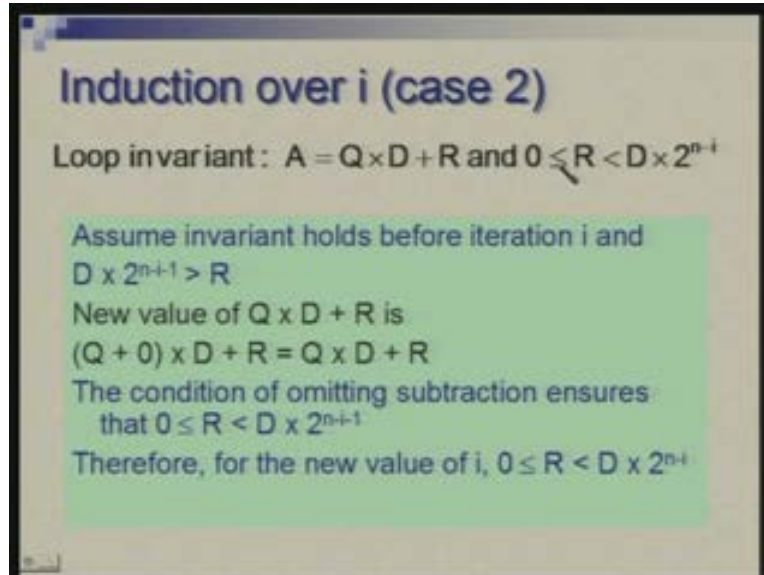
Therefore, for the new value of i,  $0 \leq R < D \times 2^{n-i}$

Therefore, now, after the iteration i also changes to i plus 1. So therefore, we can say that for the new value of i R lies in this range and hence the invariant both components of invariant hold after the iteration. Is that clear?

This was actually a more difficult case which we have ensured. The other case is when subtraction is not carried out is in fact easier. So we assume that invariant holds before iteration and the subtraction condition is false. That means  $D \times 2^{n-i} > R$ .

Now in this case the new value of  $Q \times D + R$  is a basic it trivially same as  $Q \times D + R$  because you are setting a bit 0 your recording a 0, Q was initially 0 so essentially there is no change there is no change in R so there is no change in  $Q \times D + R$ . And the condition of omitting subtraction that is this one ensures that initially we have R lying in this range. So basically R is in the half range of this and we are not changing the value of R, what we are changing is the value of i. So with the new value of i you can say that R lies in range 0 to  $2^{n-i}$ . Therefore, in this case also invariant holds and therefore we have regressively ensured that we have a correct algorithm.

(Refer Slide Time: 21:44 min)



**Induction over i (case 2)**

Loop invariant:  $A = Q \times D + R$  and  $0 \leq R < D \times 2^{n-i}$

Assume invariant holds before iteration i and  $D \times 2^{n-i+1} > R$

New value of  $Q \times D + R$  is  
 $(Q + 0) \times D + R = Q \times D + R$

The condition of omitting subtraction ensures that  $0 \leq R < D \times 2^{n-i+1}$

Therefore, for the new value of i,  $0 \leq R < D \times 2^{n-i}$

Now we will modify it and prove it so that we get a suitable hardware implementation.  
Is there any question so far?

Now in the algorithm which I had written we were talking of D multiplied by 2 raised to the power some factor. So, rather than every time multiplying by that factor we will shift it step by step as we did in case of multiplication. Of course the directions of shift may be different here. So we start with a value in D which is B into 2 raised to the power n minus 1. So initial shift is there by position n minus 1 and then we will always ensure that D has the right value so that you can always subtract D from R straightaway and this will be ensured by making D shift right every time. So we do  $D = D / 2$  and this comparison becomes comparison of D and R and subtraction becomes subtraction of D from R.

Also, instead of shifting instead of setting different bits of Q we will be shifting Q left effectively multiplying it by 2 and adding a 1 or adding a 0. Shift left and set the last bit set the least significant **bit by** bit to 0 or 1 so operationally this becomes easier. It is basically the same thing. First time we were placing B such that there was a shift of n minus 1 position. So we ensure that D is initialized to this number and subsequently move D right by one position every time.

(Refer Slide Time: 23:45 min)

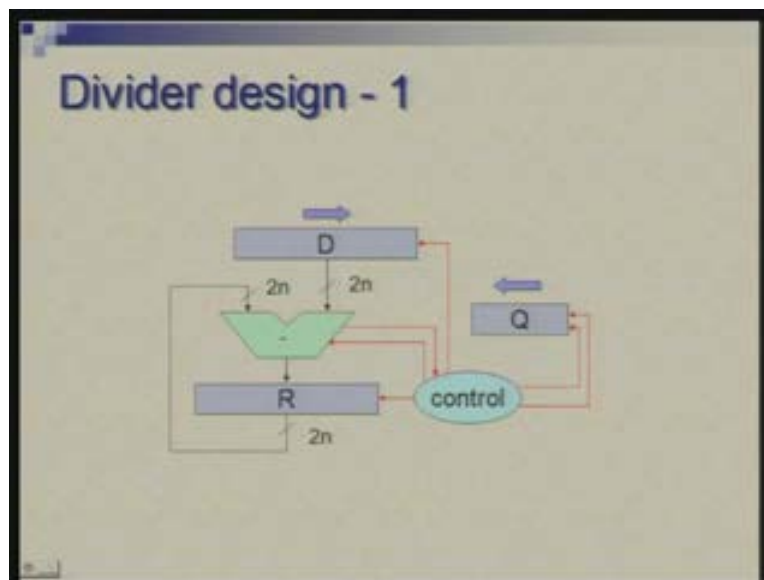
### Introducing shift registers

$$A = Q \times B + R$$

```
step1: i = 0; R = A; Q = 0; D = B x 2n-1
do {
  step2:
    if (D ≤ R) R = R - D; Q = 2 x Q + 1
    else Q = 2 x Q
    D = D / 2; i ++
} while (i < n)
```

So, once we have done this we can draw a circuit which carries out this operation.

(Refer Slide Time: 23:57)

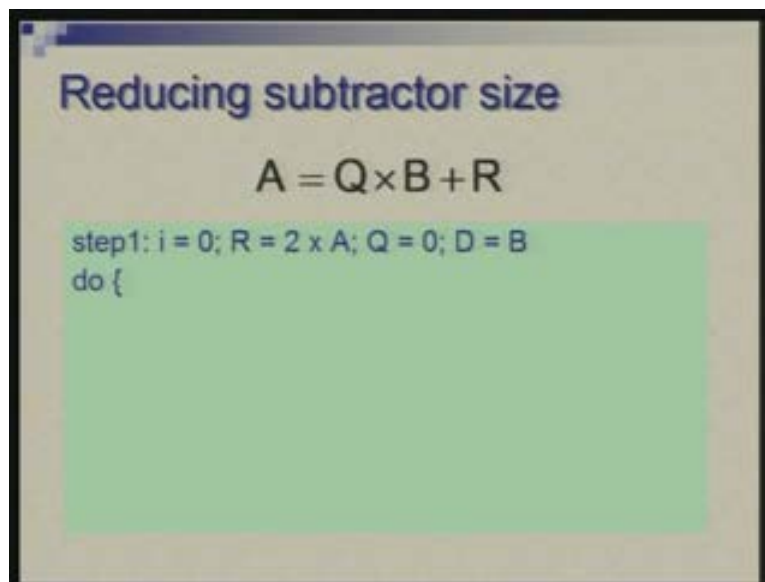


So you would notice that the structure is quite similar to that of a multiplier except that the shifts are in different directions, addition has been replaced by subtractions. So here is a subtractor which will subtract D from R so R is one input, D is another input, result goes back to R. This D shifts right by one position every time, Q shifts left by one position every time and on the right side we will insert a 0 or 1 as the space gets created in Q. So there is a controller sitting here which will look at the result of comparison. Basically we are also assuming that this subtractor is doing comparison. So the result of

subtraction is available actually at the output of this subtractor; we may not store it actually if the result is negative. So basically we are assuming that there is an indication that the result is positive or negative and this control acts accordingly. It will modify the value of R or not accordingly and it will set a 0, 1, Q as Q shifts to the left. So this is a this circuit captures the straightforward algorithm which we had just worked out.

Now we will carry similar kind of improvements in this as we did for multiplier. We will first make sure that subtractor is reduced in size. Right now it is a 2 n-bit subtraction because D D gets D has the divider which can be placed anywhere; initially it is quite to the left and gradually it is shifting right so all bits are significant at some time or the other and therefore we are keeping a 2 n-bit subtractor. What we will do is that we will change this situation and make D stationary; achieve the same effect by shifting R to the left so their relative position is same. We are shifting D right keeping R stationary; instead of that we will keep D stationary and shift R to the left. So, in terms of algorithm what we do is as follows.

(Refer Slide Time: 26:35)

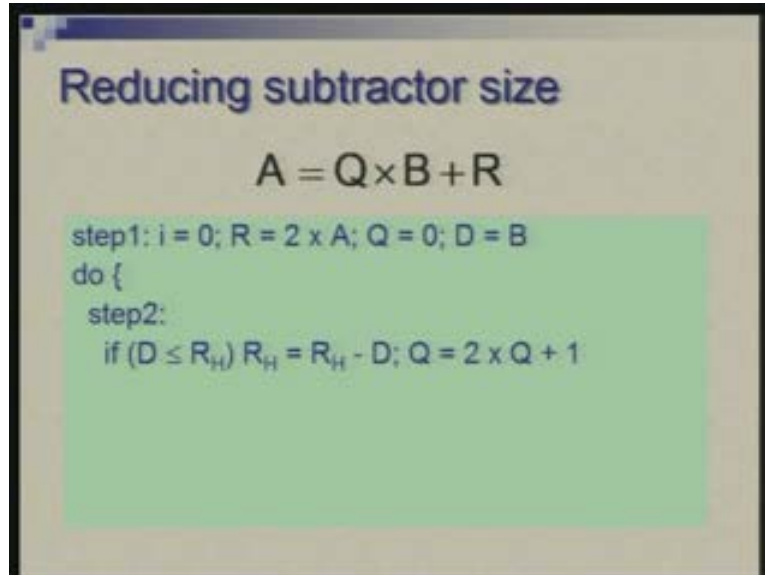


Now you would notice that D will start with the value B. It is not B into 2 raised to the power n minus 1, D will have value B, D will be an n-bit register whereas R has R has the dividend placed in correct position so that the dividend divisor are relatively positioned correctly. So you recall that the first time we shifted D was by n minus 1 bits and not n bits so the same position is maintained here because D will be now subtracted from R in its left n positions as you will see now.

We are comparing D with the higher n bits of R. R is still a 2 n-bit register and I am assuming that the two parts left n bits and right n bits will be referred to as R H and R L high and low.



(Refer Slide Time: 27:35 min)

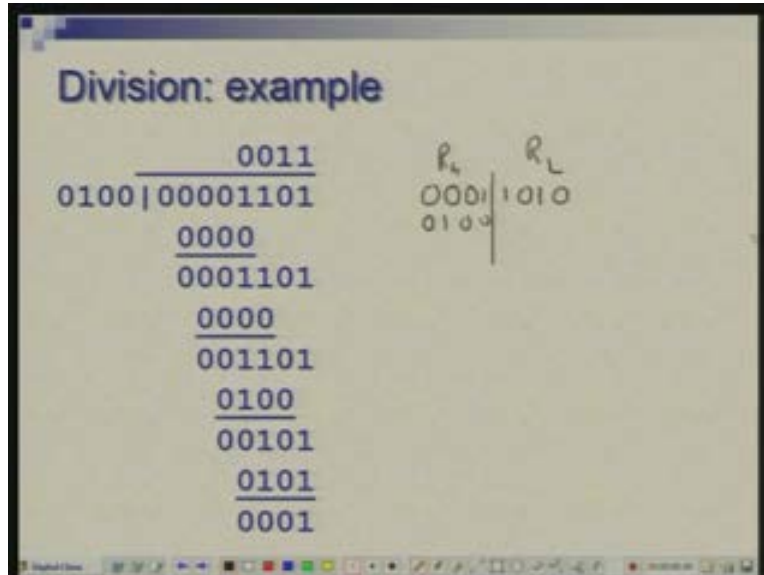


So now it is  $R_H$  or the left half of  $R$  which will participate in this subtraction. so let me again show you this diagram (Refer Slide Time: 28:05); I think it will be easier if I get back to..... so to get this same effect **what we have** we are going to place  $A$ ; if you shift  $A$  to left 1 bit..... you remember that we have placed two  $A$  in  $R$  which means  $A$  has been shifted to one position left and then placed in  $R$  so actually what it will contain..... **let me get this**.....

So initially I would have obtained this position..... **not writing very clearly**. I have placed  $A$ , 1 bit shifted left and  $B$  will be compared in this position. So it is **I am sorry** so this is  $R_H$ , this is  $R_L$  and I will always be working with  $D$  which is  $n$  bits and  $R_H$  which also is  $n$ -bit so they will be compared and the subtraction will be carried out. Next time  $R$  will be shifted left  $D$  will remain here. Instead of shifting  $D$  we are shifting  $R$  left so the starting position is here which must be carefully noted.



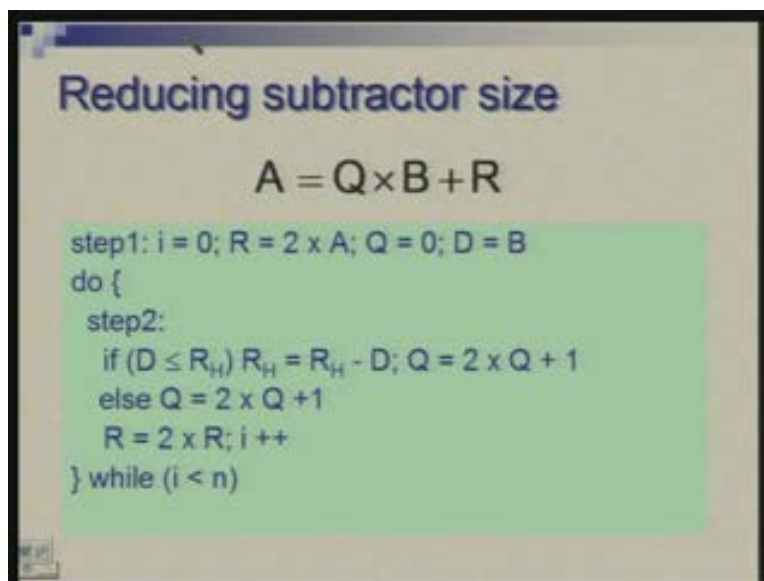
(Refer Slide Time: 31:00 min)



So, coming back to the same point (Refer Slide Time: 31:58) we are comparing D and R<sub>H</sub> and if this condition holds R<sub>H</sub> becomes R<sub>H</sub> minus D and this part is same Q gets two Q plus 1 and if the condition does not hold Q gets..... I am sorry this should have been..... ignore this let me correct this (Refer Slide Time: 32:19).

So, if the condition does not hold Q simply shifts left and R is also shifted left now.

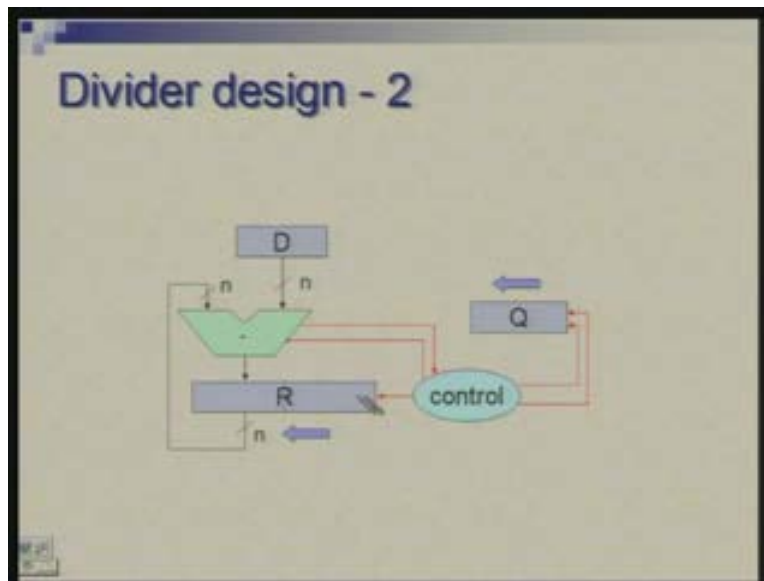
(Refer Slide Time: 33:00 min)



Therefore, with these two changes we can have a circuit now that subtractor is only n bits, size of register D is reduced and both the registers shift in same direction. So now

the next modification is very straightforward. We have R shifting to the left and as the position gets vacated here we can keep on stuffing the bits of Q so we do not need to keep a separate register for Q and utilize the part of R towards the right end which is getting vacated to accumulate the bits of Q so that is a very straightforward change and we have omitted register Q here.

(Refer Slide Time: 33:02 min)



(Refer Slide Time: 33:52 min)

### Reducing registers

$$A = Q \times B + R$$

```

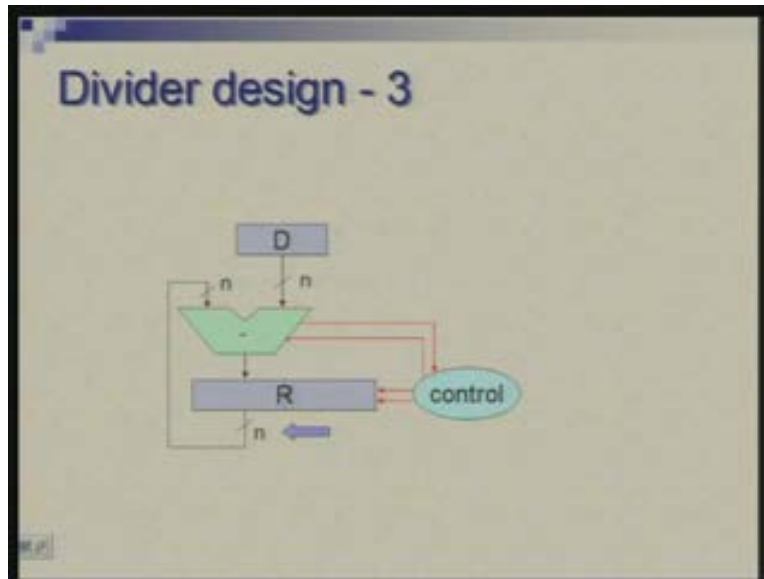
step1: i = 0; R = 2 x A; D = B
do {
  step2:
    if (D ≤ RH) RH = RH - D; R = 2 x R + 1
    else R = 2 x R
    i ++
} while (i < n)  # RH = remainder, RL = quotient

```

Therefore, we are basically doing R equal to 2 R plus 1 or R equal to 2 R. So, effect of shifting Q and R is sort of combined and at the end of the iteration R<sub>H</sub> the left half of R will contain the remainder and right half will contain the quotient so the same n-bit

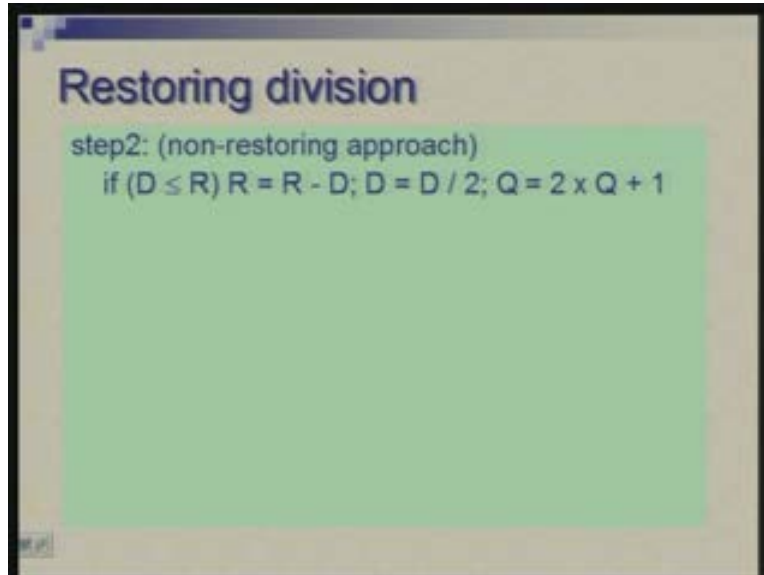
register contains the results. So this leads to a circuit which is now in the final form where we have one register shifting, another register which is n-bit and a simple control which looks at this result of subtraction and looks at the sign and accordingly controls the operations.

(Refer Slide Time: 34:22 min)



Now we will introduce another form of division which is called restoring division where what we do is; this step (Refer Slide Time: 35:03) which was there **where we are** I am now getting back to the first form of the algorithm where we had just introduced, after the basic algorithm we had introduced the shifting operation. So we were comparing D and R and accordingly we were subtracting so this is called actually non-restoring approach. We first check and then subtract and the alternative is that we carry out subtraction in anticipation and if we have made a mistake then we restore or we make a correction so that is called the restoring division.

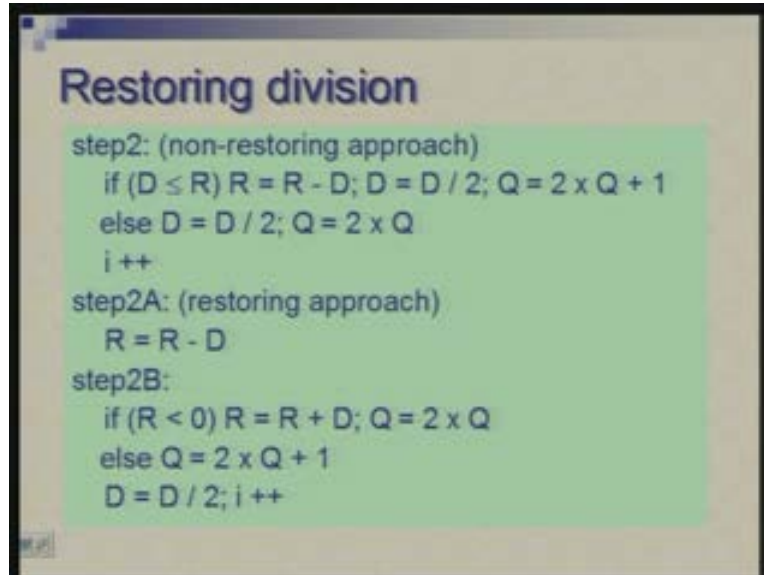
(Refer Slide Time: 35:29)



This was a complete step 2 as I have listed here and in restoring approach it changes in the following way, we subtract unconditionally and the next step within the iteration is that if now the result is negative then you correct, make R equal to R plus D, rest of it will remain same. It is in this case Q becomes 2 Q and in the other case 2 Q plus 1. So, if R is not negative then you do not need to make any correction and **D part** D equal to D by 2 and i plus plus is common.

So basically change has occurred here that we have introduced an unconditional subtraction and there is a conditional addition. So now in this case **we we** we are actually using two steps in the iteration. So two clock cycles would be used because first you have to carry out subtraction it is only then you can carry out addition. So, apart from the fact that you are using additional steps the subtractor has to be replaced by a circuit which can do addition or subtraction but that is not a big deal we have seen how that can be done.

(Refer Slide Time: 37:06 min)



### Restoring division

step2: (non-restoring approach)

```
if ( $D \leq R$ )  $R = R - D$ ;  $D = D / 2$ ;  $Q = 2 \times Q + 1$ 
else  $D = D / 2$ ;  $Q = 2 \times Q$ 
i ++
```

step2A: (restoring approach)

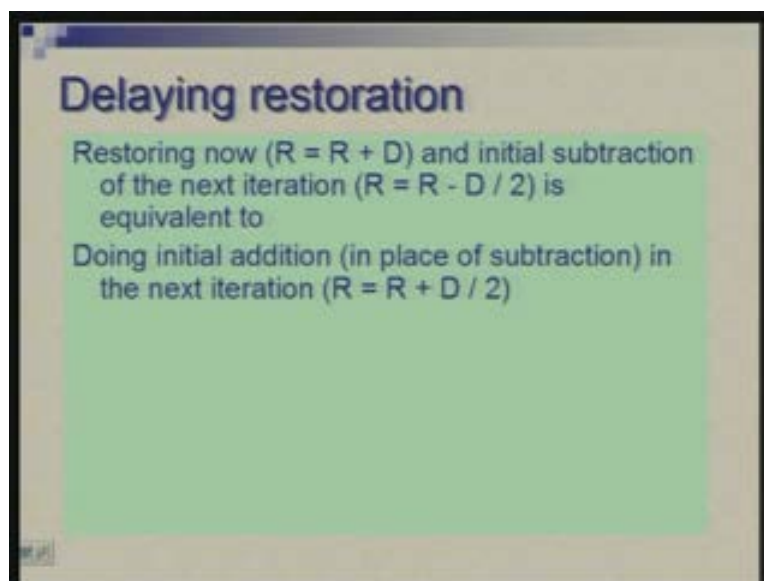
```
 $R = R - D$ 
```

step2B:

```
if ( $R < 0$ )  $R = R + D$ ;  $Q = 2 \times Q$ 
else  $Q = 2 \times Q + 1$ 
 $D = D / 2$ ; i ++
```

The motivation for doing this is what it is going to follow that we can actually postpone these restorations by making this following observation that if you are restoring now in any particular step you are restoring by adding  $D$  to  $R$  and in the next iteration there will be again an initial subtraction,  $D$  would have reduced by a factor of 2 so now you are subtracting  $D$  and in the next iteration you will subtract  $D$  by 2. The same effect can be achieved by not doing any restoration now and in the next iteration the initial unconditional subtraction may be replaced by addition. So adding  $D$  now and subtracting  $D$  by 2 later is equivalent to adding  $D$  by 2 later.

(Refer Slide Time: 38:05 min)



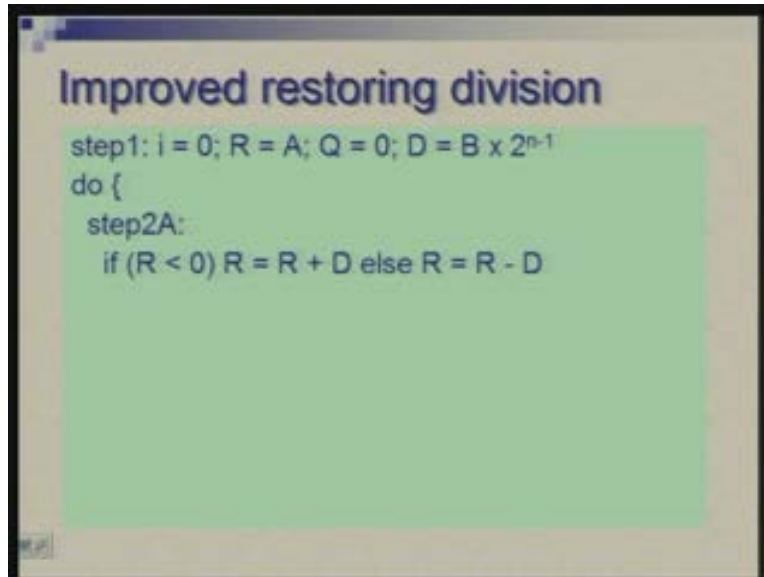
### Delaying restoration

Restoring now ( $R = R + D$ ) and initial subtraction of the next iteration ( $R = R - D / 2$ ) is equivalent to

Doing initial addition (in place of subtraction) in the next iteration ( $R = R + D / 2$ )

So what we are doing is we are wording an additional step of restoration now and achieve the same effect by choosing the initial unconditional step to be either an addition or subtraction. So the algorithm now looks like this.

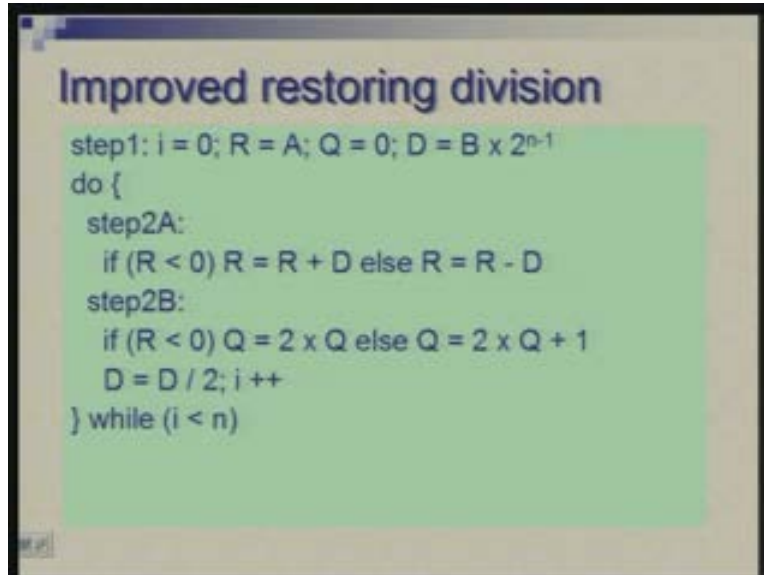
(Refer Slide Time: 38:48)



Initially we will look at the sign of R because possibly there might be a pending restoration requirement. In the previous iteration we might have subtracted where subtraction is not to be done and as a result we may have a negative R value. So, if R is negative in the beginning of the iteration we make it R plus D else you make it R minus D as usual. So negative R implies that there is a pending restoration which we have postponed and we start with an addition. Otherwise in the normal case we start with the subtraction.

I am just showing a separate step where after this we are recording a bit 0 or 1 in Q and D is halved, i is incremented and **this is** that completes the iteration.

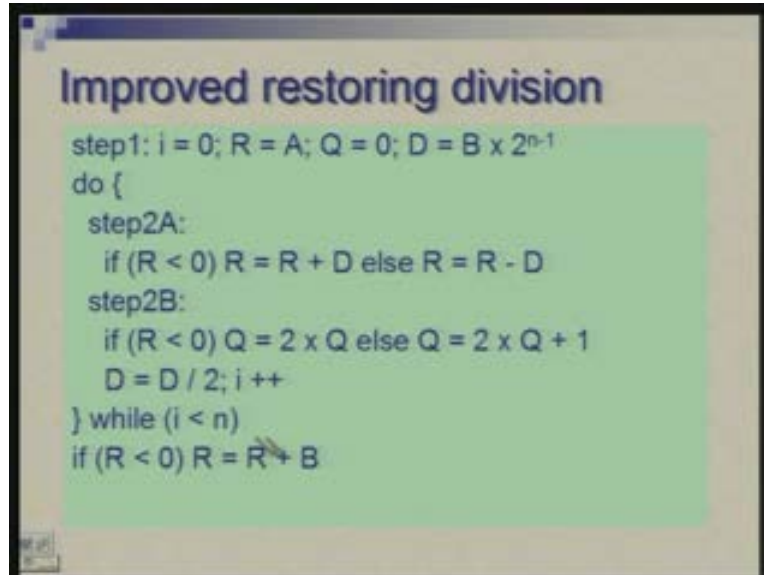
(Refer Slide Time: 39:56 min)



So now, of course I am not yet eliminated two steps; I still have two steps but I have just taken care that restoration is postponed. So in each iteration you are doing only one addition or a subtraction. Earlier there was a possibility of one addition and subtraction both in the same iteration and they had to be necessarily done in sequence. Now with some modification we can actually [.....40:38] these two steps.

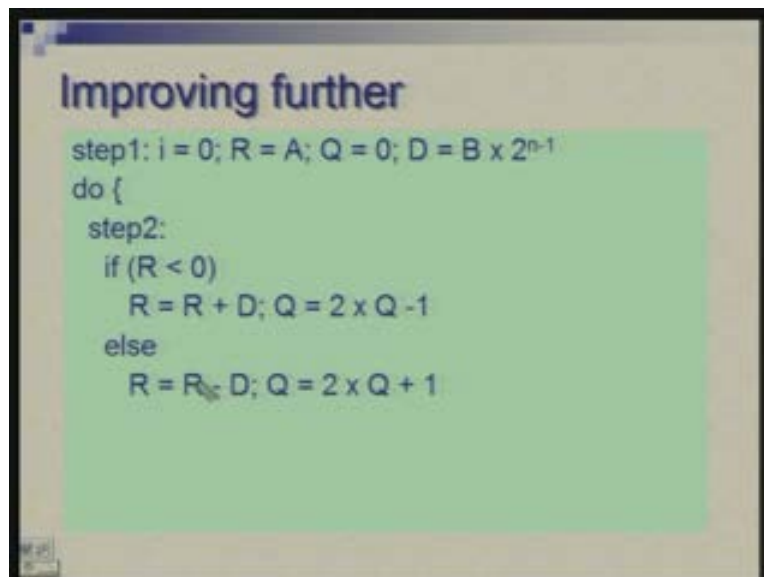
But one important thing is that since you are postponing your restoration; in the last step if you postpone and no more iteration is left you may still have a pending restoration. So a final adjustment may be required here. So I am adding a step there; at the end if R is negative finally then you would need to make a final correction.

(Refer Slide Time: 41:00 min)



We will improve it further where we are actually combining these two steps. So, recording of that bit in Q is actually brought within this condition in a single step so if R is negative then you are doing initial addition and **you are** here you make Q equal to 2 Q minus 1 otherwise you start with initial subtraction and make Q equal to 2 Q plus 1.

(Refer Slide Time: 41:47)

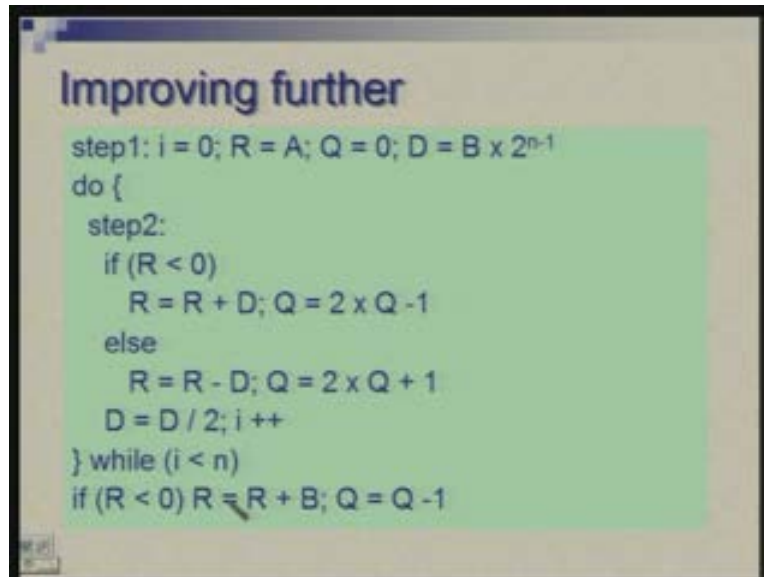


So basically what you would notice is what is happening is that as you are doing.... Let us this is an anticipatory subtraction which you do and in anticipation you are also recording a bit 1 in Q. So, later on, when you possibly correct this you also would correct this effectively. So recording of bit in Q is also done in anticipation and possibly



corrected in the next step if necessary. So, that effect is achieved by doing  $2Q - 1$  and this part remains as it is and there is a final step. So now finally also we may require a correction in  $Q$ . This is a step which if  $R$  is negative finally we have a correction done in  $R$  as well as in  $Q$ .

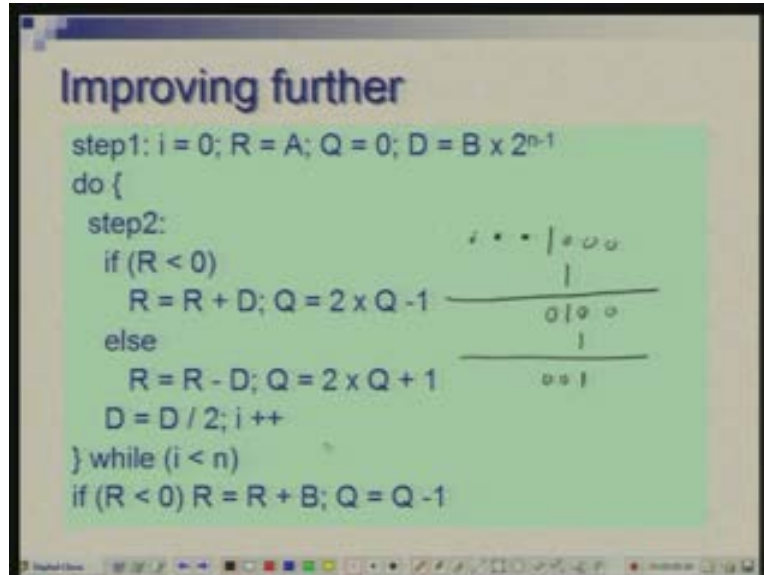
(Refer Slide Time: 42:46 min)



So let us.... let me..... you want me to illustrate how this correction  $Q$  is getting done or you can see it through. Let me illustrate this.

What will happen is that, see, let us say we have some bits of  $Q$ , we have reached some point and let us say we have put one in anticipation and if it is getting if it is correct it will be left as it is. If in the next position we are doing corrections then we will be subtracting one from this position next to it which means this 1 I mean if you just see it locally this 1 0 and from that if you are subtracting 1 it will become 0 1. Or if let us say if this need for correction continues; if you if you are subtracting a 1 again here then this one also become 0 and you get a 1 here and so on. So essentially what is happening is that the 1 which you have put in anticipation gets converted to 0 and you are putting a 1 in the next position and if even that is not correct next time when correction is done even that gets moved further.

(Refer Slide Time: 44:33 min)



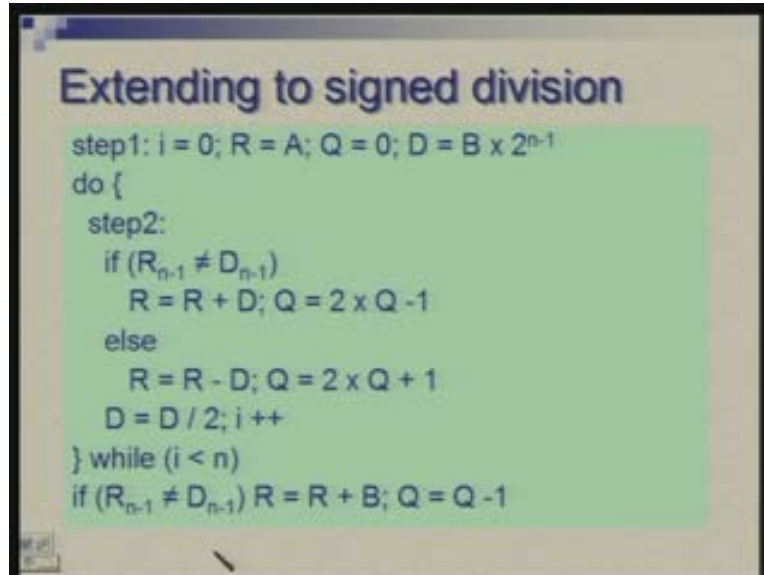
So I will suggest that you actually take out you take some examples and work through these algorithms so that you I have convinced yourself that it works correctly.

Now, once we have brought the algorithm to this form effectively we have now made it almost ready for a signed division. What is effectively happening is that in every iteration by adding or subtracting  $D$  we are trying to bring  $R$  closer to 0; you are starting with some value and your attempt is to bring it successively close to 0 so you subtract initially a large value then you subtract half, subtract half of that, then half of that and so on. So essentially you are trying to bring it close to 0 and if we observe it in that sense whether the value was initially negative or positive it can still work.

If the dividend was negative we are still by adding a positive value we will bring it close to 0 or if divisor is also negative by subtracting divisor we will bring it close to 0. So essentially attempt is to look at signs of R and D and accordingly either subtract or add. So, instead of checking whether R is positive or negative we will see if sign of R and D are the same or different. If they are different then we subtract **sorry** if they are different then we add because if opposite sign values are being added then the result will be small and if they are of the same sign then we subtract so that is the change in the logic.

I am looking at the MSB of R and D the sign bit and if they are not equal then we add otherwise we subtract and accordingly Q becomes  $2Q - 1$  or  $2Q + 1$  so that is the only change and we can use the same thing for a signed division. The final correction step also makes a similar check; it compares  $R_{n-1}$  and  $D_{n-1}$ . If they are still of the opposite sign then there is a final addition which is being done.

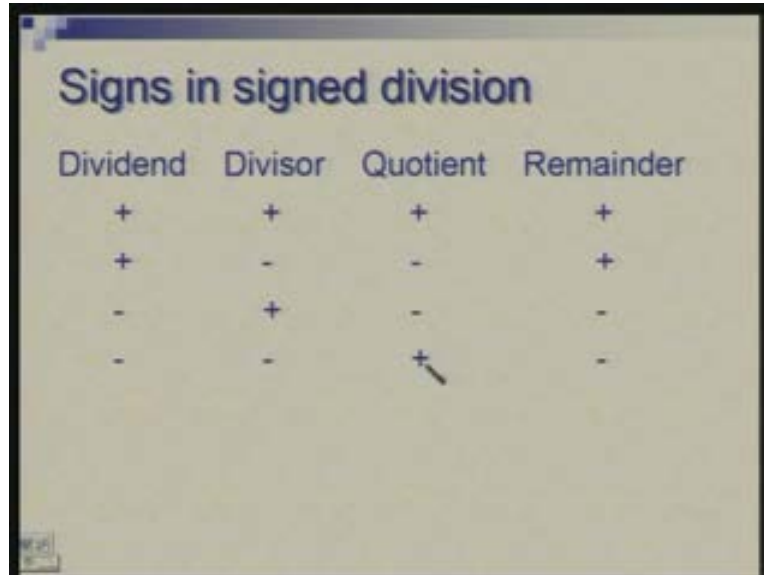
(Refer Slide Time: 47:46)



So now intuitively is this correct; what is the relationship between signs of dividend, divisor, quotient and remainder? So let us have a look at that. We are given basically dividend and divisor and we are looking at all possible sign combinations. Any of these could be positive and any of these could be negative and the two right columns show the corresponding signs of quotient and remainder.

So, what is the logic which is intuitively governing this; that sign of the quotient would follow similar logic as you have in multiplication. When you multiply two numbers of the same sign that is positive into positive or negative into negative the result is positive so same thing we will do. You are dividing a positive number by a positive number you get a positive quotient. You divide a negative number by a negative number you get a positive quotient. So, the quotient is following the same logic as multiplication.

(Refer Slide Time: 48:42 min)



Dividend	Divisor	Quotient	Remainder
+	+	+	+
+	-	-	+
-	+	-	-
-	-	+	-

Therefore, when the signs are opposite the quotient is negative; when the signs are same the quotient is positive. The remainder always takes the same sign as that of dividend because remainder is something which is left out of dividend you try to reduce the dividend to 0 but something is still left so it has to be of the same sign and if you look at this (Refer Slide Time: 50:08) so the final correction actually which is being made here will ensure that remainder is ultimately of the same sign; **this is.... no**, actually that is not obvious from here; I think it is..... one could prove that the remainder would be of the same sign although it is not obvious from this condition.

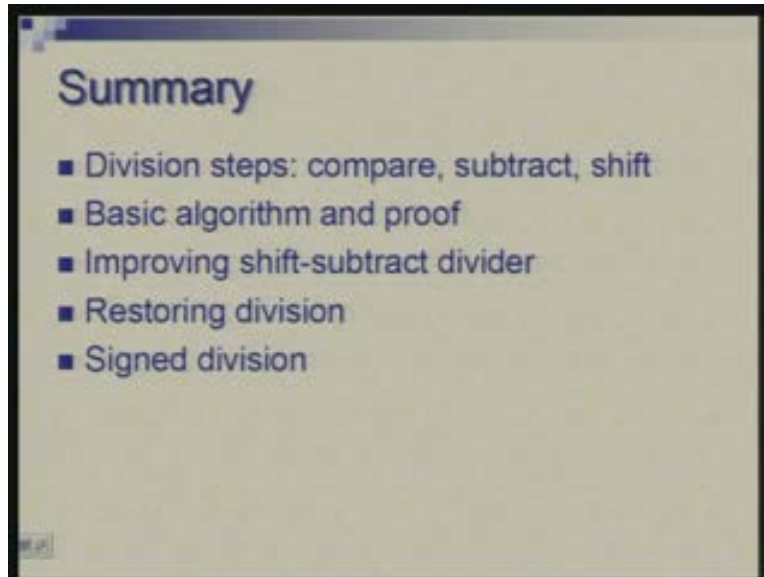
Well, actually I **would like you to I** have written several algorithms and the kind of proof I did using invariants you should try for a few more particularly this one. So it should ensure that the correct signs are being obtained here. In particular the remainder should be of the same sign as the dividend and the quotient would depend upon whether the sign of divisor and dividend are same or opposite. I leave it to you to ensure that very regressively.

So let me summarize what we have done. We started with simple handworked example of 4-bit division in unsigned case and we notice that using basically compare shift and subtraction operation you can carry out division in a sequential manner. So based on that we developed a basic algorithm, we analyzed it thoroughly to ensure that it is correct and from there we derived the circuit. In the circuit then some improvements were made. First improvement was to reduce the size of the subtractor; instead of  $2n$  bits subtractor we reduced it to  $n$ -bit subtractor and then we did some improvisation and reduced the number of registers so ultimately we worked with one  $n$ -bit register and two  $n$ -bit register. Then we brought in the concept of restoring division.

Initially you carry out anticipatory subtraction so there is no comparison involved here now; subtraction is an unconditionally or I should say blindly you subtract and then

restore. Then we saw that restoration can be postponed which simplify each iteration and of course it made it necessary for us to have a final step where a final restoration of final correction was necessary.

(Refer Slide Time: 53:22 min)



Then with slight changes we were able to modify this algorithm for signed numbers and remember that when we are talking of signed division we are doing addition, subtraction with 2's complement number so therefore the addition, subtraction need not be done or addition, subtraction are not conscious of whether the numbers are signed or unsigned as long as representation is 2's complement; so that fits and nicely and we can have the algorithm work on two signed integers and perform divisions. I will stop with that. Next time we will take up floating-point operations and with that this chapter will be done.