Computer Architecture and Organization

Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 37
Design of Dividers

In the last few lectures we had seen how we can implement adders and multipliers. In this lecture we shall be looking at the design of dividers, the various algorithms we can use and how we can implement them in hardware. So, the topic of our discussion today is design of dividers.

(Refer Slide Time: 00:46)

Introduction

- Division is more complex than multiplication.
- *Example*: Typical values in Pentium-3 processor.
  - Not easy to construct high-speed dividers.
- The ratios have not changed much in later processors.

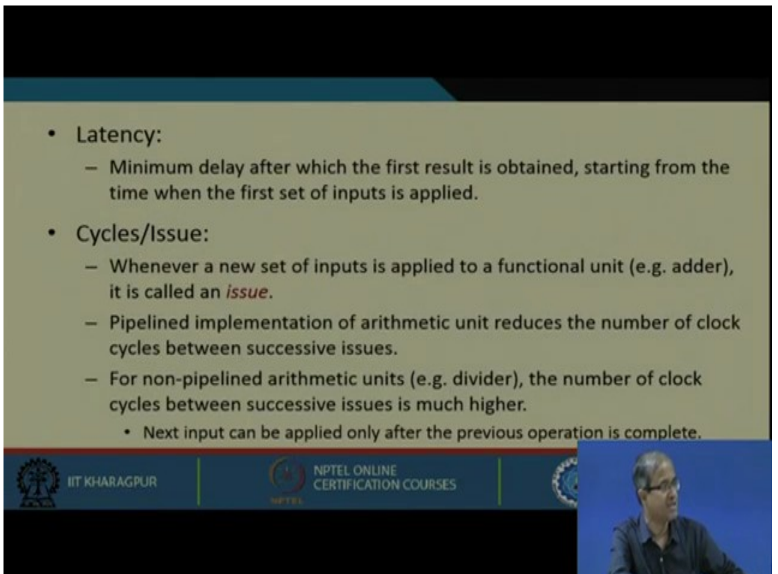
Instruction	Latency	Cycles / Issue
Load / Store	3	1
Integer Multiply	4	1
Integer Divide	36	36
Floating-point Add	3	1
Floating-point Multiply	5	2
Floating-point Divide	38	38

Before we start let us look into a basic problem that is there in the division operation; because the first thing is that division is more complex than multiplication because of various reasons that we shall be seeing during the course of this lecture.

Now, to appreciate this point that division is indeed more difficult than multiplication or any other arithmetic operation, let us look at some typical facts and figures that are present in one of the commercial processors, Pentium 3. Here in a table we are showing for some typical arithmetic operations two things, one is called the latency other is called the cycles/issue. One thing we can see from this table is that the values for divide operation (either integer or floating point whatever) is much higher as compared to the values for load, store, multiply, addition, this kind of operations.

These are the values for Pentium 3 and even in the very recent processors the relative differences still remain. We shall be coming back to this slide once more; first let us see what do you mean by latency and cycles per issue.

(Refer Slide Time: 02:26)



Let us look at this first. Latency refers to the minimum delay starting from the time when the first set of input is applied and up to which the first result is obtained. So, latency talks about the just the initial delay, the delay for the first time. Suppose I have a circuit it can be an adder it can be a multiplier divider whatever I apply a set of inputs. Now, the question is after how much time I get my result --- this time duration is called latency. So, minimum after how much time I can get back my first result.

Now there is another thing. Well, you see I apply an input I get back an output after some time that is equal to the latency. Now the second point is that how frequently can I apply the inputs; do I have to wait until the operation is complete before I apply the second input, or I can overlap my operation in some way.

Well, if I have to wait for the operation to be complete before I apply the next input, then this is so-called cycles / issue. This is called cycles per issue --- after minimum how much gap I can apply my second input, that will be the same as latency. But if we are allowed to have some kind of an overlap, meaning before the first operation is complete I am allowed to feed in the next input or the next to next input, this is what happens in a

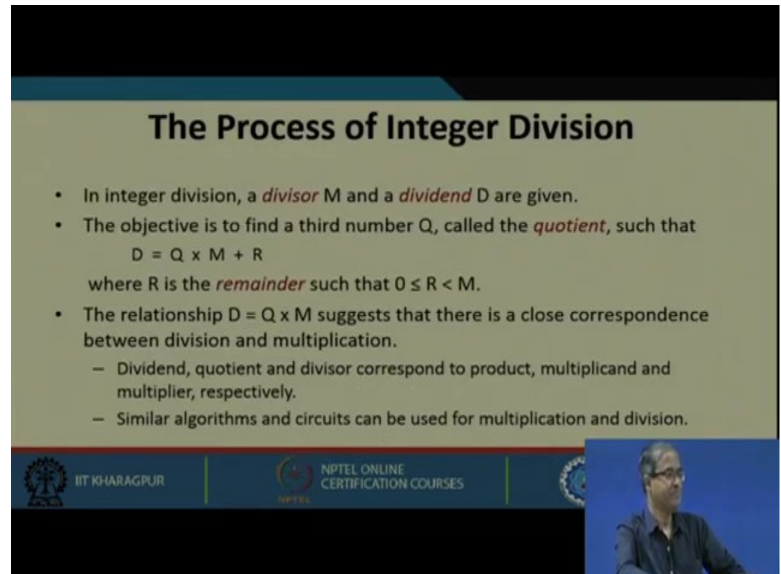
typical pipeline processor which you shall be looking into detail much later during the course of these lectures.

In a pipeline implementation we can apply inputs much faster although the latency value remains basically the same. I apply one input, after how much time the result comes out that remains same, but the inputs that I apply one after the other the gaps between them that can be less. So, whenever we apply a new set of inputs to some functional unit like an adder or a multiplier, we call it an issue; that the inputs have been issued.

Now, as I have said that if we have pipelined implementation, we can reduce the number of clock cycles between successive issues, but for a circuit like divider it is very difficult to have a pipeline implementation, most of the dividers are non-pipelined. So, for such circuits the number of clock cycles between successive issues is much higher; next input can be applied only after the previous operation is complete. So, let us go back to the previous slide and see, for load store kind of instructions latency is 3 clock cycles while cycles per issue is 1. That means, I can issue 1 such instruction every clock cycle; there is an overlap possible.

For integer multiply again latency is 4. So, I can complete multiplication in 4 cycles, but I can apply new sets of inputs every 1 clock cycle. For floating point addition again 3 is the latency, 1 is the clock cycles per issue; and for floating point multiply it is 5 and 2. But divide latency and cycles per issue values are same 36 here, and for floating point 38; this means that division units are not pipelined, they actually work as a single non pipeline block. You apply an input get the output, after it is finished only then you apply the second input. So, cycles per issue is equal to the latency.

(Refer Slide Time: 07:06)

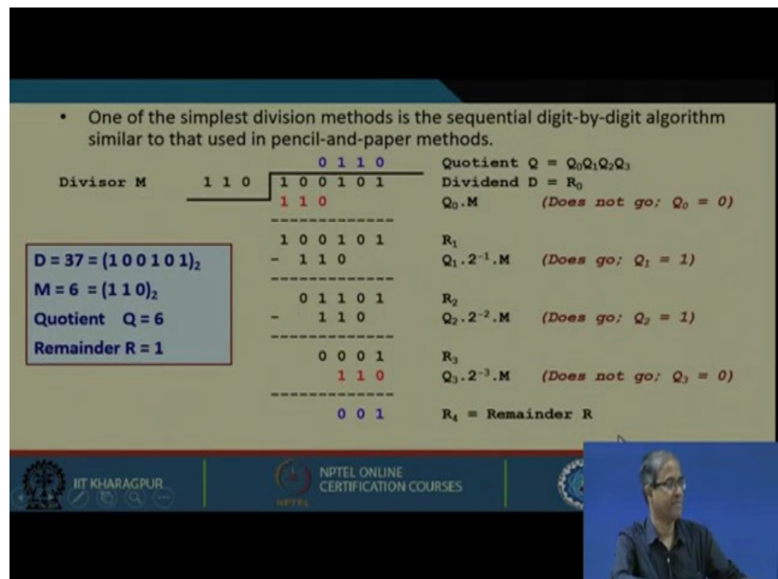


Let us now look into the process of integer division. What you really do when you divide 2 numbers. For integer division we have something called a divisor, something called a dividend. We divide the dividend by the divisor, D is greater than M. So, the objective is to find a quotient that is the result of the division and of course a remainder; remainder R should be less than this divisor M, and this M, D, Q and R are related by an a equation like this.

Now, just if we ignore the remainder for the time being, D is Q x M. So, you see there are basically two things that you are multiplying, and you get D. So, there is an analogy you can draw between division and multiplication, the operations look quite similar. Here we are talking about dividend, quotient and divisor; and in multiplication we talked about product, multiplicand and multiplier. So, there is a correspondence between dividend and product, quotient and multiplicand, divisor and multiplier.

Now, this correspondence will be clear when we look at the basic data path, or the circuits that we use for division. If you compare this circuit with what we had used for multiplication, you can see that see this correspondence immediately. Because of this correspondence very similar circuits and very similar kinds of algorithms can be used for multiplication as well as division.

(Refer Slide Time: 09:10)



So, let us just work out a simple example division using the traditional approach which is sequential digit by digit, shift and add kind of thing.

This is quite similar to what we do using the so called pencil and paper approach. So, the example we take is for a dividend D which is 37, divisor is 6; we are dividing D by M. So, D we are writing here this is 37, 1 0 0 1 0 1 and this is my divisor 1 1 0 which is 6. What we do in a normal division step, we see if divisor goes with the first 3 digit here it is 1 0 0, if it goes I can subtract, but here we see that 1 1 0 is greater than 1 0 0 which means it does not go, so what to do we set the next quotient bit to 0 if it does not go.

So, the next quotient bit is set as 0. Because it does not go we do not make any change to the dividend, it remains the same next step. The divisor we shift by 1 place and repeat the same process, we shift right by one step and we compare it with this 1 0 0 1. Now we see 1 0 0 1 is greater than this; that means, we can subtract which means it does go and the next quotient bit will be 1. So, because it goes now, we can do the actual subtraction.

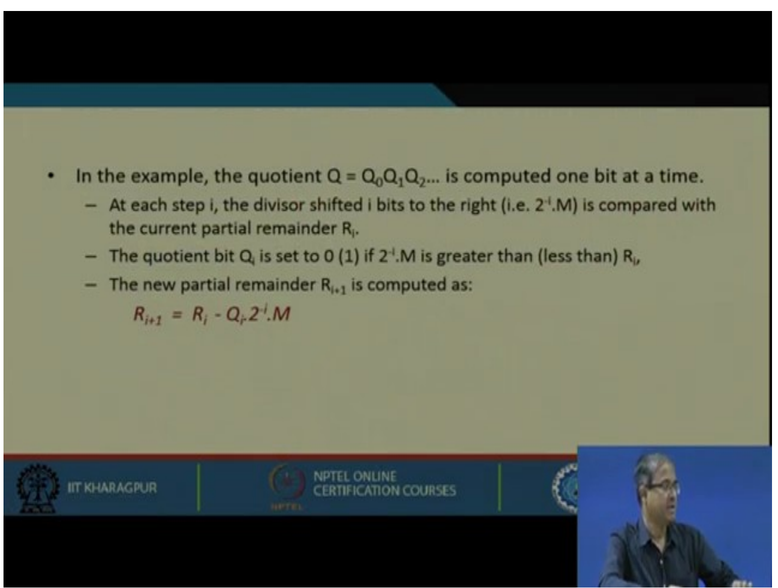
So, you repeat the same process. What finally remains here will be my remainder and I have already found out my quotient. So, my quotient is 6 in decimal my remainder is 1 in decimal.

So, you see here the steps that I have shown side by side. So, the quotient bit that we are generating for example, Q1 equal to 1 and this divisor MI am shifting it right by 1

position, I am expressing it as 2 to the power -1 x M2. To the power -1 means divide by 2, and you recall divide by 2 means shifting the number right by 1 position. So, we are doing exactly that here, similarly here we are shifting M right by 2 positions, 2 to the power -2 means dividing by 4, here we are shifting with right by 3 positions, 2 to the power -3 x M.

So, actually whatever we are trying to subtract it is actually the quotient bit multiplied by 2 to the power some -I x M. This partial remainders you can say R1 starting with R0, R1, R2, R3 and finally, R4 you get we subtract these values from this R i's this is what we do here.

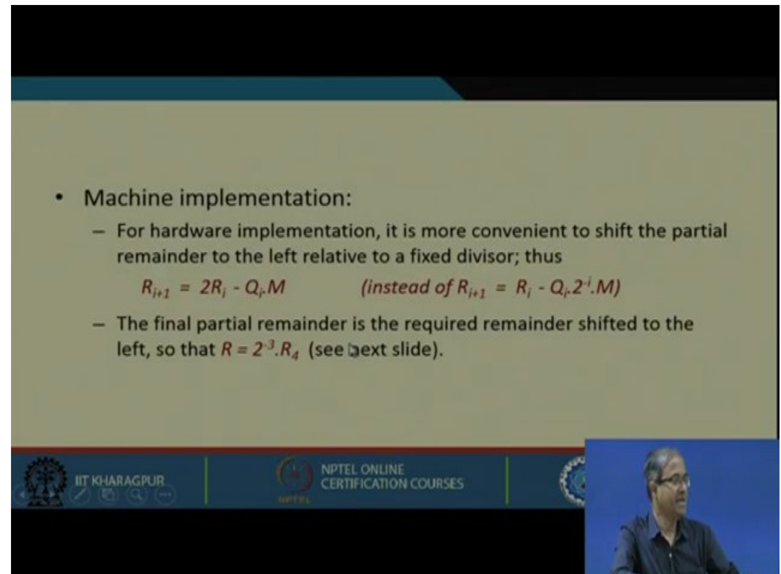
(Refer Slide Time: 13:16)



Just to recall exactly what we saw. We were computing the bits of the quotient one at a time, and at each step what we are doing --- the divisor was being shifted I bits to the right you see here it was 1 bit, 2 bits, 3 bits. So, I goes from 1 2 3, 2 to the power -1, -2, -3 like this.

So, we shift I bits to the right; that means, we compute 2 to the power -I x M and this compared with the current partial remainder whether it goes. If it goes then we set Qi = 1; if it does not go we set Qi = 0, and the new partial remainder is computed by subtracting. So, if it does not go Qi is 0. So, actually this is 0 anyway. So, we do not make any change Ri minus 0, but if it is 1 we subtract the shifted divisor 2 to the power -I x M from Ri .

(Refer Slide Time: 14:24)

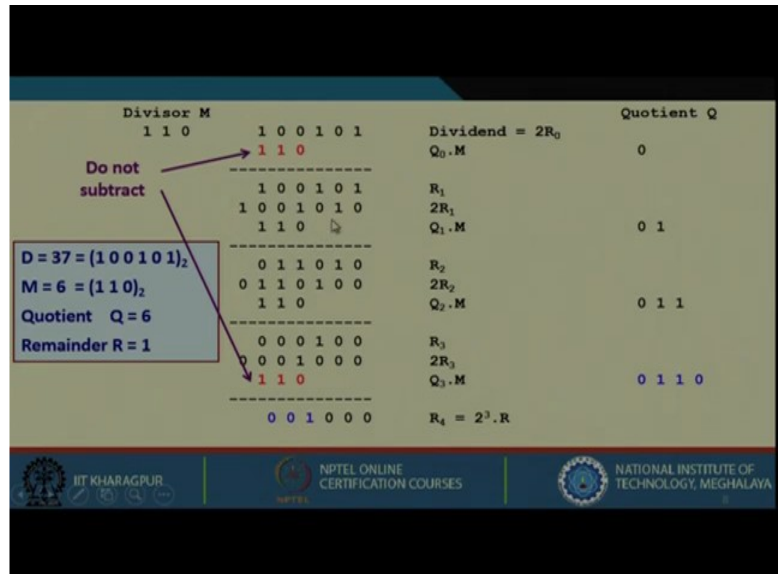


If we want to implement by hardware just like in multiplication algorithm we were keeping the partial product shifted and the multiplicand we are keeping in the same position, we are following the same principle here the partial remainders we shall be shifting, but the divisor will be keeping fixed in one position that will help us in implementing it in hardware.

So, here it is more convenient to shift the partial remainder to the left, but we keep the divisor in a fixed location, which means in the earlier case we are doing something like this --- we were shifting the divisor to the right and we are subtracting from the partial remainder, but now we shall be shifting the partial remainder to the left; that means, multiplying by 2, and we will be subtracting this divisor which is at a fixed location. So, there is no 2 to the power -I here, just Qi x M; if the next quotient bit is 0, we do not subtract anything, if it is 1 we subtract the divisor

But here the only one change is there because we are not shifting the final partial remainder that remains. Actually to get the remainder from there we have to shift it right by 3 places; this is the only correction we have to make.

(Refer Slide Time: 16:05)



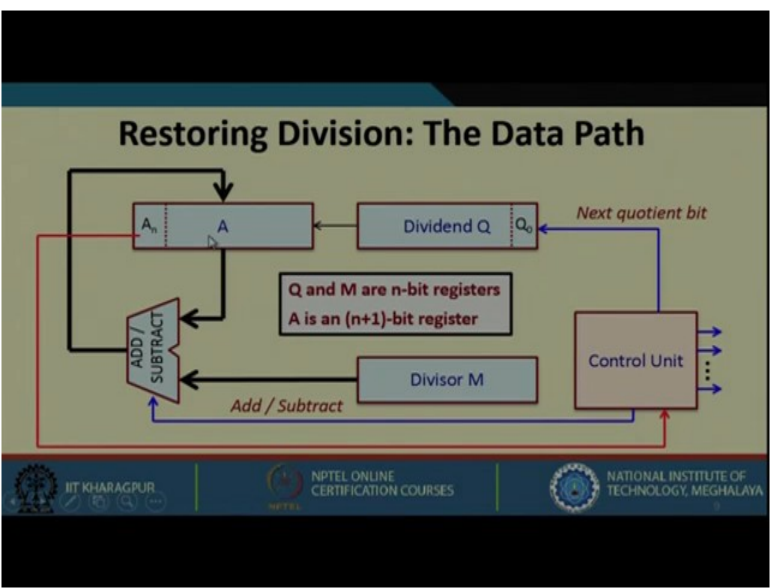
So, here we have an example worked out; that same example 37 divided by 6. Here you see 37 is the dividend here we expressed in 6 bits and divisor is we see that whether it goes 1 1 0 divisor is in fixed place you see 1 1 0 is in fixed place. So, it does not go. So, we said the next quotient bit to 0 and no subtraction.

So, the partial remainder remains the same this is R1 we shift R1 to the left by 1 position shift left. So, this becomes 1 0 0, 1 0 1 and a 0 goes in. Now you again try to subtract 1 1 0 from here, here it goes. So, the next quotient bit will be 1 you do an actual subtraction 1 1 and 0 and these 3 bits remain. This is my second partial remainder R2, shift it left again 2R2. So, again you subtract the divisor it again goes 1 1 0 1 1 0. So, next quotient bit is again one. So, you subtract this is R3, shift it left divisor try to subtract you see that it does not go.

So, next quotient bit is 0, and you do not subtract and whatever was there it remains. So, remainder is this. So, to get the remainder will have to shift it right by 3 positions that was actually mentioned here. Remainder multiplied by 2 to the power 3 is actually what you are getting here. So, this will be my quotient and this will be my remainder. So, one thing you have seen from this example is that at every step we are checking whether the divisor goes; that means, whether we can subtract it or not, but in an actual hardware circuit how we shall check that whether it goes or does not go we can do it by making a trial subtraction.

Let us do a subtraction and see that the result is becoming negative or not, if the result is becoming negative then you can say that it does not go; if the result remains positive then it is fine because we are talking about unsigned numbers, no negative number. But if you find it does not go and you have already subtracted, we will have to add the divisor back to restore the correction, restore the original value. So, a restoring step may be required as the correction.

(Refer Slide Time: 18:59)

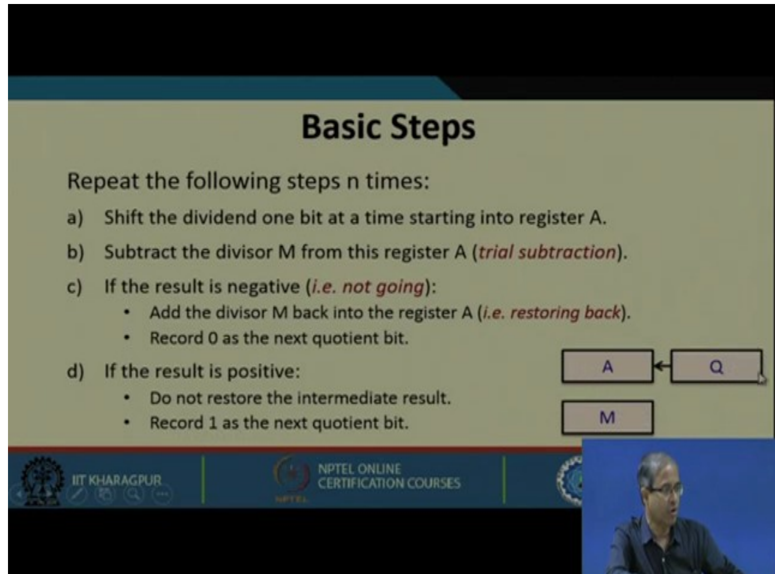


So, here as it said we do not subtract here we do not subtract here these 2 places, and the concept that we are saying that we make a trial subtraction and then you can add it back to make the correction step. When we see that it does not go, that can be very easily implemented by using a hardware circuit or a data path as we can see in this diagram. This diagram looks very similar to the multiplier data path, which is why we said that there is a very strong correspondence. So, we have a temporary register A which will be initializing to 0, the dividend we store here, and the divisor we store in another register and this A is a this is An, n plus 1 bit register; one extra bit because we have to check this sign after trial subtraction.

So, what we do from a at every step we do a trial subtraction of the divisor and check the result of the subtraction; that means, the sign bit is 0 or 1. The control unit will be checking that. If it sees that the result is negative then it will again activate an addition step.

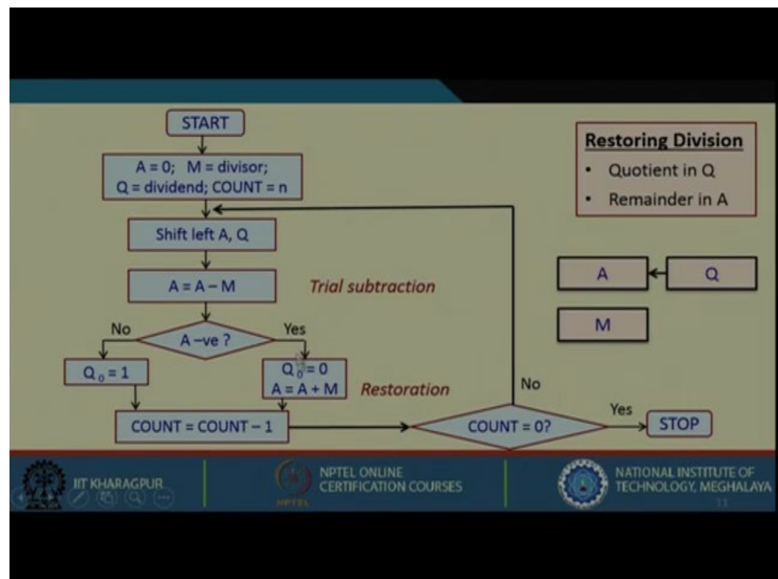
The divisor will be added back to A to restore back the value; that means, we have done a subtraction wrongly we add back to restore the previous value. This is basically what we are trying to do, and this whole process we repeat and in every step we shift dividend and this a register left by one position, this was exactly what we are doing here right we are shifting this every step by one position, one position, one position we do exactly the same thing here right in hardware.

(Refer Slide Time: 20:54)



In terms of the steps; we have this A register, Q register and the divisor in another register M. So, we shift the dividend 1 bit at a time into register A. So, both A and Q we are shifting left, subtract divisor M from A; that means, M is aligned to A. We always subtract M from A; this is our trial subtraction. If the result is negative which means it is not going then we add M back to register A; this is the restoring step, and because it was not going the next quotient bit; that means, last bit of Q we record as 0, but if the result is positive; that means, it goes you do not do any restoration or addition and record 1 as the next quotient bit.

(Refer Slide Time: 21:53)

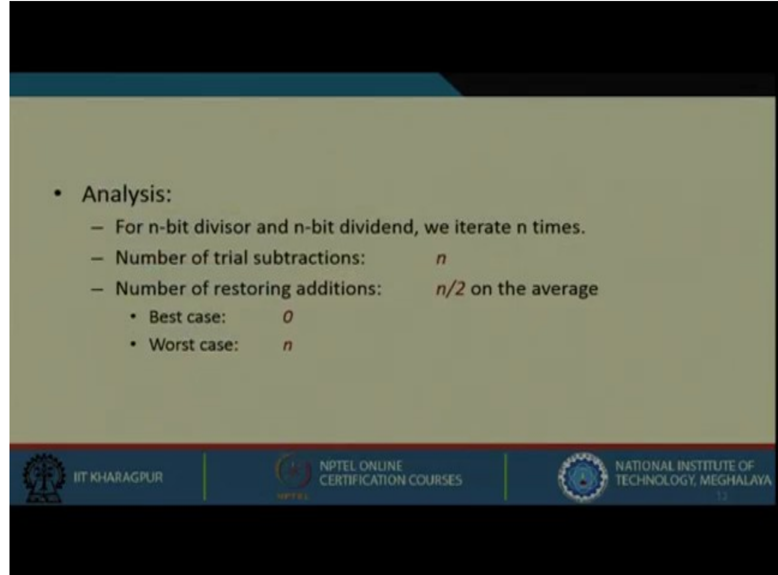


This can be depicted in the flowchart form as follows the same thing which we said this you see we load A with 0, M contains the divisor, Q contains the dividend.

So, we start by shifting left AQ; that means, the first bit of the dividend gets into A and we make a trial subtraction, this is your trial subtraction after shifting you make a trial subtraction then you check after trial subtraction whether a is becoming negative or not. If you say A is negative which means it does not go, in that case the last bit of Q, Q0 is set to 0, and you add M back to A --- this is a restoration step. But if you see A is not negative after subtraction, then it is fine --- you simply set the quotient bit to 1 and repeat this n times. count was initialized to n, decrement by 1 as long as it does not reach 0; you repeat this loop, if it reaches 0 you stop rght.

This is the basic restoration division algorithm. Now you see from this flowchart that you are looping n times, now every time you are starting by doing a subtraction and depending on the sign of a negative or positive you are either doing an addition or not doing an addition. So, we can say that on the average we shall be doing the corrective addition 50% of the time; that means, n/2 time on the average.

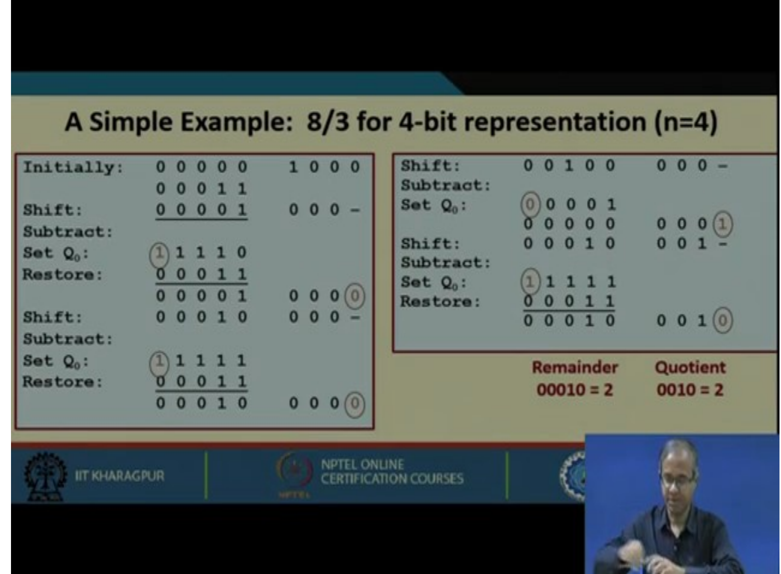
(Refer Slide Time: 23:39)



So, this is the analysis. So, for n bit divisor an n bit dividend we iterate n times. So, every time we carry out a trial subtraction; that means, there are n trial subtractions and on the average number of restoring additions will be n / 2, because in the best case you will not need any restoration 0 worst case it will be n.

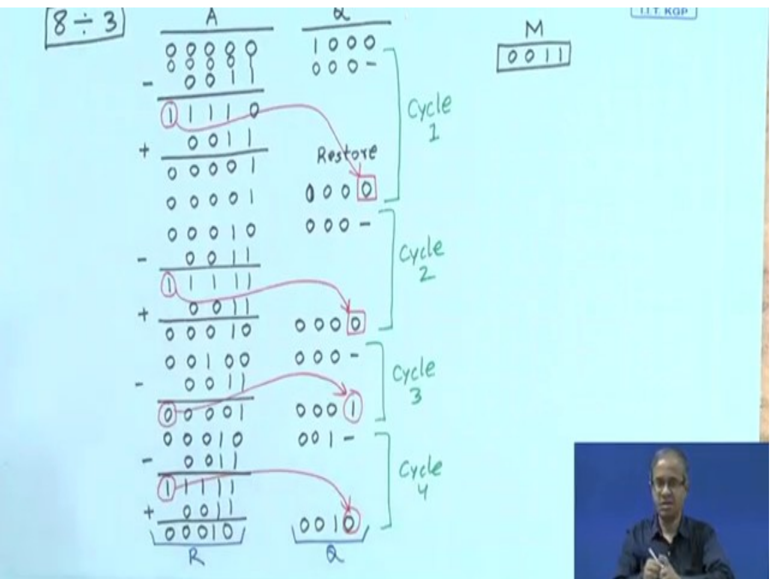
So, average will be (n + 0) / 2 = n / 2. So, the total number of addition / subtraction will be n + n / 2 here.

(Refer Slide Time: 24:13)



There is an example that is worked out here. I shall actually just work out this example on paper just to show. Finally, you get the remainder and quotient let us work out this example. So, we want to carry out a division operation 8 divided by 3.

(Refer Slide Time: 24:43)



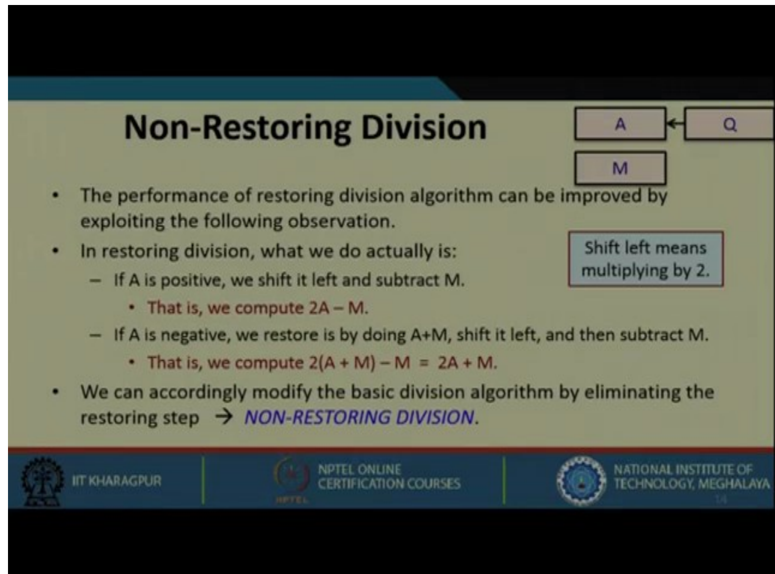
So, what we do we start like this we have this A register. A is a n + 1 bit register; suppose we are representing everything in 4 bits. So, A will be a 5 bit register, initialize to 0, and side by side there is a Q register. Q will contain the dividend 8 (1 0 0 0), and the multiplicand here M is 3. M in binary is 0 0 1 1. So, you start the operation. You will have to make a trial subtraction from the first step. You subtract 0 0 1 1 from A. After subtraction the result will be 1 1 1 1 0. What you see after subtraction is that the result is negative; that means, the sign bit is 1, because your sign bit is 1 you will have to restore it back; that means, you add 0 0 1 1 again to restore back the original value of course.

You have to start by making shifting. You first make a shift and then do this addition. So, this will be a shift 0 0 0 -, and then you do the addition or subtraction. This continues.

So, this is how you can work out the algorithm step by step by hand.

Let us look into an improvement now, the restoration division algorithm that you have talked about here what we have seen is that we are doing subtraction every time in the iteration and on the average half a time we are also doing a corrective addition.

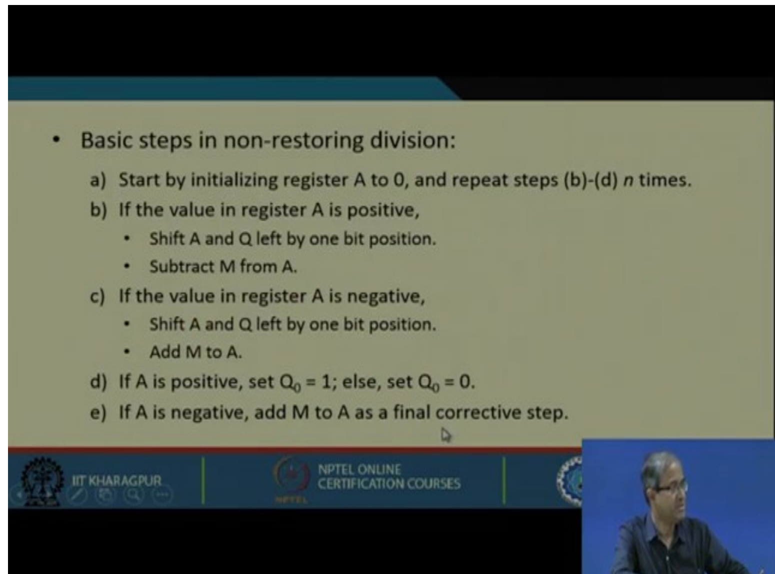
(Refer Slide Time: 31:32)



Now, let us see if you can reduce the number of addition and subtraction our objective is to carry out only one operation in every iteration --- total addition subtraction will be n only. So, we make an observation that in the restoring division algorithm that you have already seen what you are doing is if after the trial subtraction A was positive, we shifted left and subtract M; that means, shift left means we are doing 2A, then we are subtracting M shift left means multiplying by 2, but if A was negative we were first restoring it by adding back M, then shifting and then again subtracting M for the next iteration.

So, this is done in the current iteration and again you go back next iteration you again do a subtract. So, if you combine the two what it means is that this A + M is done shifted left means twice of that, then subtract M. So, this is 2A + M. So, effectively in one case you have been 2A - M, and in the other case we are doing 2A + M. So, if we can modify our algorithm by making this observation then we can reduce the number of operation; this method is called non-restoring division.

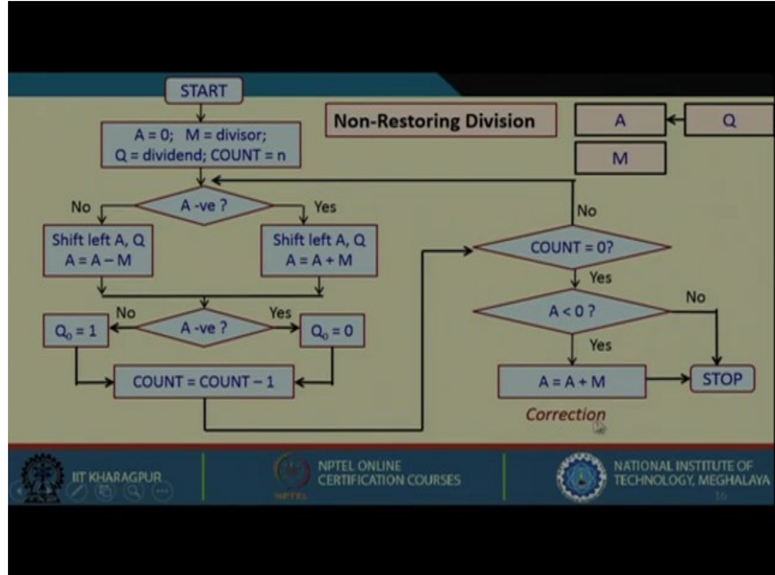
(Refer Slide Time: 33:26)



So, let us see what we do here we similarly start by initializing register a to 0 and repeat these steps b, c, d n number of times. The method is simple; if A is positive you shift A and Q left by 1 position and subtract M from A, but if it is negative you add M to A. This was the observation; in one case we subtract M other case you add M, then you check if after this addition or subtraction A is positive or negative, if it is positive set the quotient bit to 1 else set the quotient bit to 0.

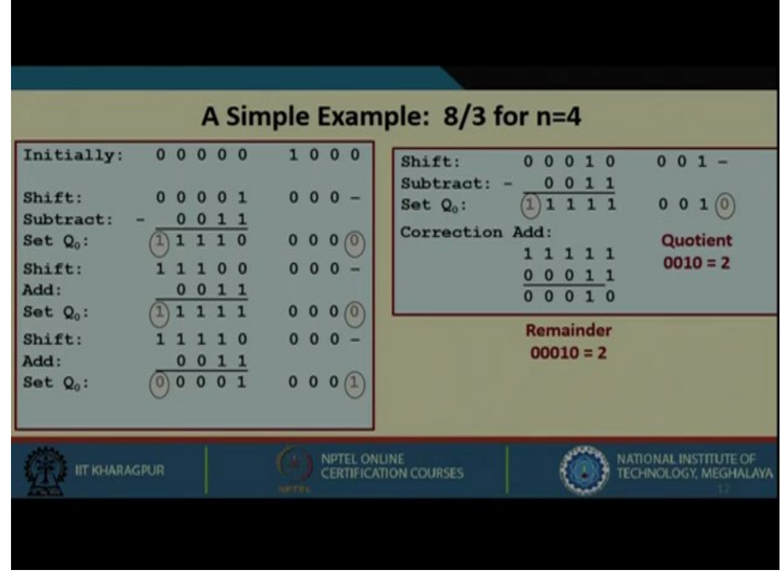
Now, see here we are including one additional subtraction that is there in the next iteration with this operation. So, for the last time there may be an addition subtraction operation which we have carried out. So, we may have to carry out a corrective addition only at the very end. So, that we are doing here if at the very end A is negative; that means, we have carried out a wrong last subtraction we have to make a corrective addition at the end.

(Refer Slide Time: 34:44)



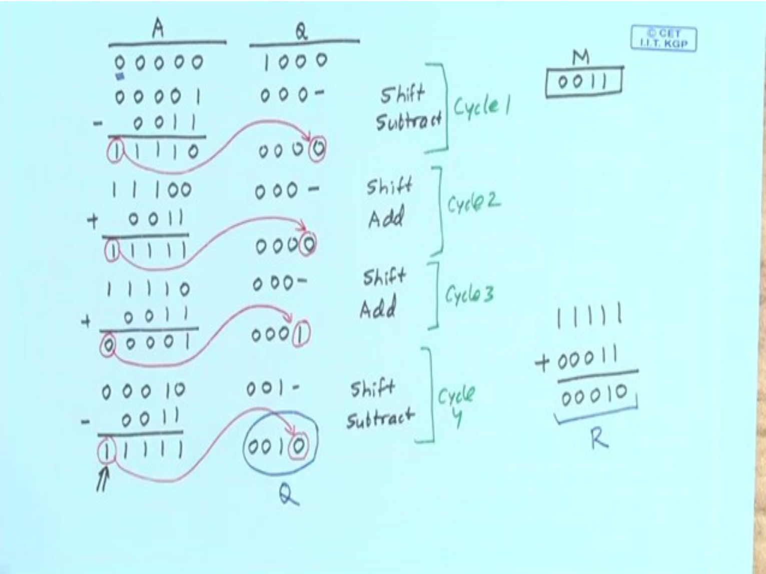
In terms of flow chart, I just showing it diagrammatically like this. We initialize the registers as earlier, we check A is negative or not; if yes we shift left and add, if not shift left and subtract. Then we check whether A is still negative or positive; if it is negative the quotient bit is set to 0, or else quotient bit set to 1. This thing you repeat n times and after you complete n times you check finally, whether A is still negative or not, if it is still negative you have a corrective addition step.

(Refer Slide Time: 35:39)



This is what the algorithm is. So, here again I shall be working out this simple example 8 x 3 by hand. So, the same example I will be taking I am just showing this animation first. So, let us work this out again.

(Refer Slide Time: 35:59)



So, here just exactly similar to what we did for the restoring division we start with this register A containing all 0’s and register Q containing the dividend. You recall our divisor was 3 which was 0 0 1 1 this was M. So, according to the algorithm, we shall be starting by checking whether A is negative or not. accordingly we do a shift and then addition or subtraction.

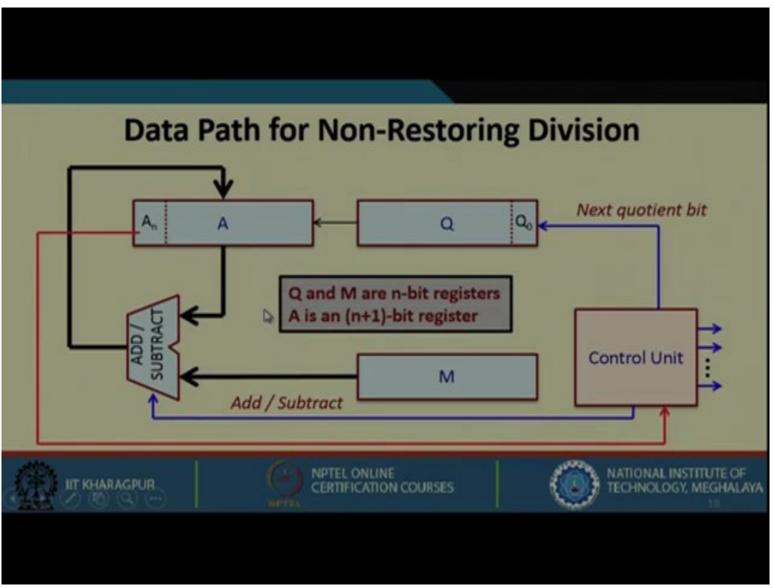
So, we are doing that same thing. So, we are seeing that the number is positive. So, we will be doing a shifting and then a subtraction, if you do a shifting this becomes 0 0 0 0 this 1 comes here this will be 0 0 0 -, this is your shifting then you do a subtraction, subtract M from this minus 0 0 1 1. So, this result will be 1 1 1 1 0 and this 0 0 you see after subtraction your sign is 1. So, your next bit here will be 0. So, this one will decide the next quotient with 0. You see the flowchart once more that here we do a subtraction and after subtraction we check whether A is negative or not, when if it is negative we set Q0 to 0.

So, because it is negative we are setting Q0 to 0. So, let us continue with this. We have a shift step, we have a subtract step --- this is your cycle 1. So, you repeat this step shift left again 1 position, now this is 1 that is how you have to shift, and add if it is 0 we will

be subtracting if it is 1 we will be adding. So, we do a shift first it will be a 1 1 1 0 0 this 0 will come in 0 0 0 dash and we will be doing addition here plus 0 0 1 1. So, this will be 1 1 1 1 1 and 0 0 0 see here again you see that the sign is one. So, the next bit that will be coming in the quotient will be 0 just like in the previous case.

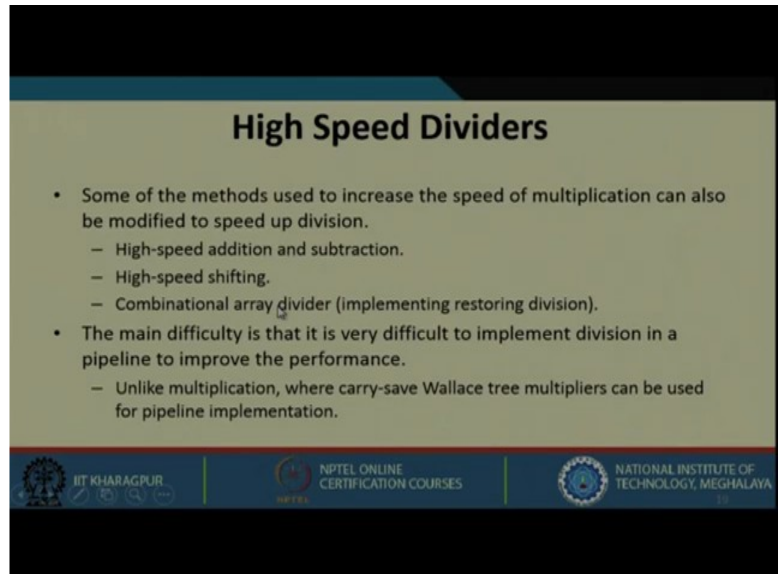
The process repeats. So, you see this step by step we have seen how the restoring division algorithm works.

(Refer Slide Time: 42:12)



Regarding the data path, it is exactly identical to what we saw for the restoring algorithm. There is no difference and the same hardware can be used; only your control unit will be different. The advantage is that we need only n number of addition or subtraction steps here.

(Refer Slide Time: 42:32)



Now, if we want to go for high speed division, some of the methods that we have already seen to design high speed multipliers can be used here also, like you can use high speed adders and subtractors using carry look ahead adders, carry select adders etcetera.

We can use high speed shifters like barrel shifters, we can use combinational array dividers which are similar to combinational array multipliers, and they means actually can implement the restoring division algorithm. But the problem with division algorithm is that you can do it, but it is extremely uneconomical and ineffective to build a combinational array kind of a divider. The other thing is that by including sign in the division process; signed division algorithm is not easy normally, it is done using a separate step by separately checking the sign bits and making a corrective step at the end to correct the sign of the product.

So, signed division is again a problem. The main difficulty that you have seen earlier in the very beginning of the lecture we said that for division the latency as well as the number of cycles per issue are very high. That is mainly because of the difficulty in implementing the division algorithm in a pipeline. Multiplication can be very effectively implemented in a pipeline in a Wallace tree kind of a multiplier, that is how you have seen that the number of cycles per issue for multiplications can be as low as 1. E;very cycle you can feed a new data to a multiplier and the total latency will be 3 it takes 3 cycles to complete the multiplication.

But for division it is not so, division still remains the bottleneck. So, as a programmer whenever you are developing some applications or programs you should keep it in mind that division is an expensive operation, and you should replace division by another arithmetic operations wherever possible. This can increase the effectiveness of the program or application in terms of speed and performance.

In the next lectures we shall be moving on firstly the floating-point operations. So, how you can extend whatever we have learned so far to handle floating point numbers, numbers with decimal points and suddenly we shall be looking at other logical operations how everything can be integrated within the arithmetic logic unit. So, we come to the end of this lecture.

Thank you.