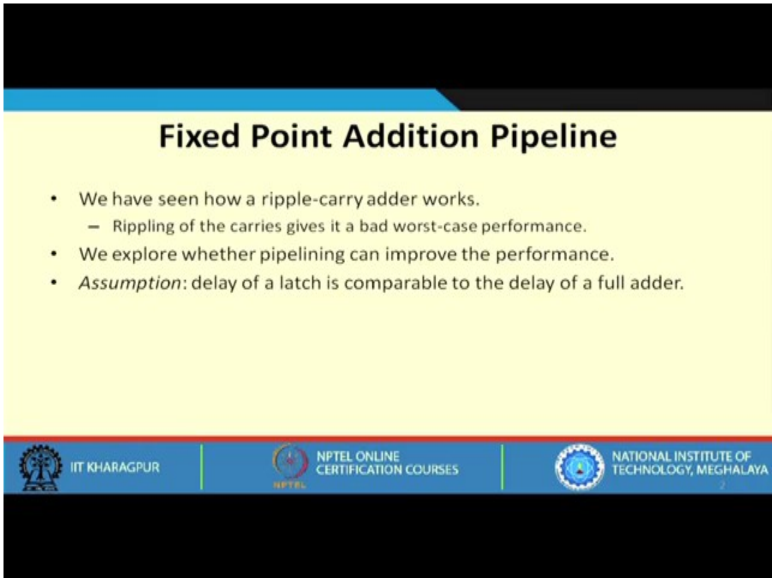
Computer Architecture and Organization
Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 42
Arithmetic Pipeline

In this lecture we shall be talking about arithmetic pipelines. Now, in a computer system if you see the way pipelines are used or utilized, you will find that they are used in broadly two areas or in two ways. One to speed up the execution of the instructions that is called instruction pipelining, and second to speed up the arithmetic operations that is called arithmetic pipelining.

So, today in this lecture, we shall be looking at some examples of the design of arithmetic pipelines.

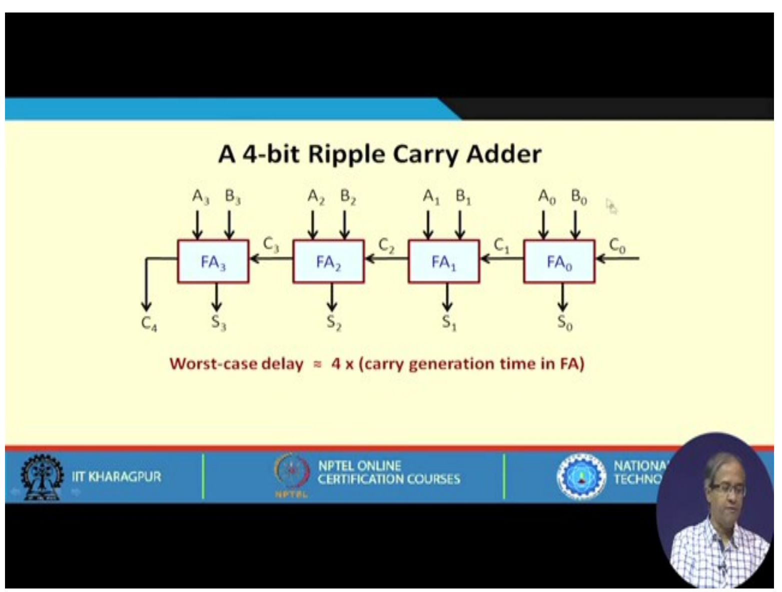
(Refer Slide Time: 01:09)



We start with a very simple example. We have already studied fixed-point addition; we have seen how a ripple carry adder works. In a ripple carry adder we use a cascade of full adders. Every full adder is carrying out the addition of a pair of bits, it generates sum, it also generates a carry, and the carry goes to the input to the next stage. So, this carry ripples like this and that is why the name is ripple carry adder. Earlier we have seen that due to the rippling nature of the carry, the performance of a ripple carry adder is not good. The worst case delay is pretty high in fact.

So, it has a bad worst case performance, but now let us see can how we use pipelining to improve the performance of a ripple carry adder. Well, one assumption we shall make. We shall be requiring latch or registers if we want to make a pipeline. Our assumption is that the delay of a latch will be comparable with the delay of a full adder.

(Refer Slide Time: 02:35)

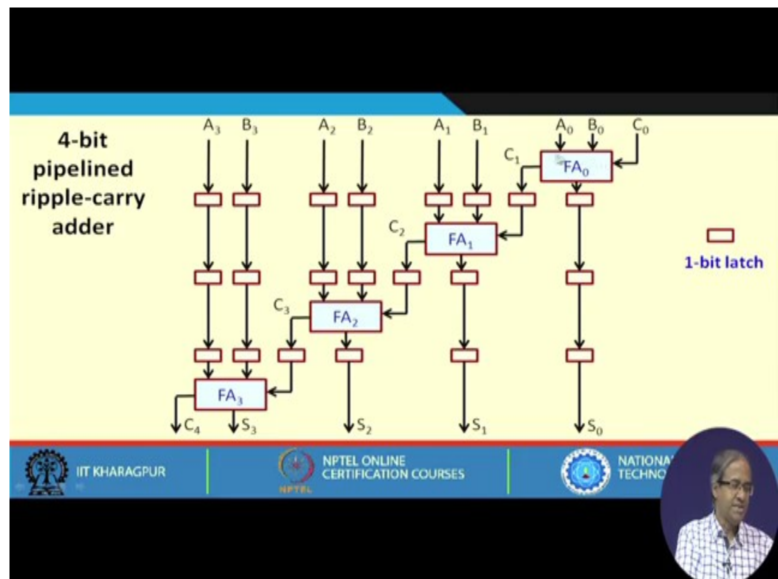


Let see this is the full adder design that we have seen earlier, this is a 4-bit ripple carry adder. In fact, there are 4 full adders. As I have said each of the full adders will be adding a pair of bits of the input numbers A and B, generating the sum bit and generating the carry bit for the next stage.

So, worst-case delay will be the first full adder will be generating a carry and because of that this carry will be generated, and again this carry will be generated. So, 4 multiplied by carry generation time in full adder, this will be roughly the worse case delay. Of course, there will be one more because of this sum generation. Anyway I am looking at the carry only. So, roughly speaking because it is a 4-bit ripple carry adder, the worst case delay will be four times the carry generation time of a single full adder.

Now, let us try to do one thing. Let us try to stretch this ripple carry adder in a skewed way, let us move the different stages of the ripple carry adder in the different time steps. Let us make them different stages in the pipeline.

(Refer Slide Time: 04:19)



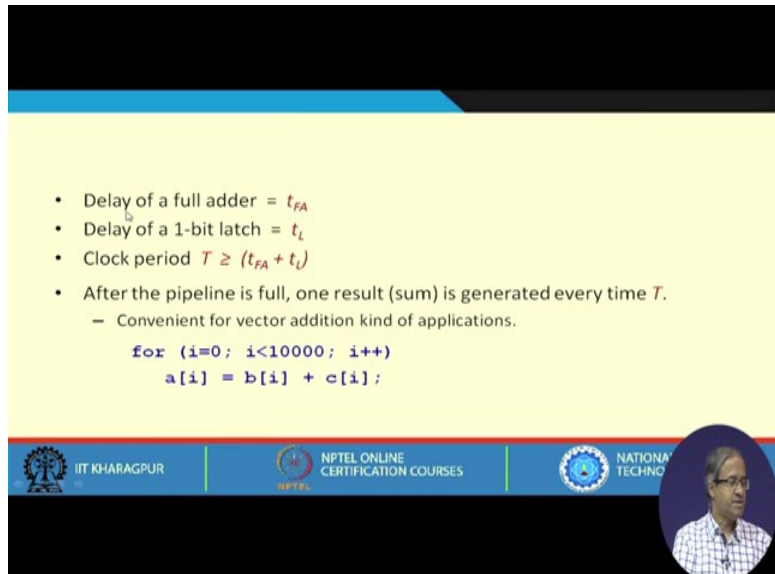
What we are doing is that whatever ripple carry adder was here like this, we are making it like this. We are stretching it, you see we just ignore these pink boxes for the time being.

You see that the full adder is still a full adder, but what I have done is that we have drawn it in a skewed way, and these are the four stages of the pipeline we are defining, and the small pink boxes will be our latches. See in the first stage only, the last full adder will be working and the other input bit pairs which are coming; they will be simply stored in the latches. They will be used later not now.

After the first stage is finished, this carry C1 has been generated and they will be stored in this latch, and this sum will be stored here. So, now stage 2 comes into the picture. In stage 2, only the second full adder is active. The others are simply dummy. These values are copied into the next lecture. Simply this sum is copied.

So, again this carry is copied, this sum is copied in the third stage again. There is a full adder here. These sums are copied, these bits are copied and in the last stage there is a full adder. So, you see we are using many latches, but when the input data is transferred to the second stage, we can parallely feed the next set of data to the first stage. So, there can be the possibility of overlapped execution just like in a pipeline. This is the advantage we gain here.

(Refer Slide Time: 06:14)

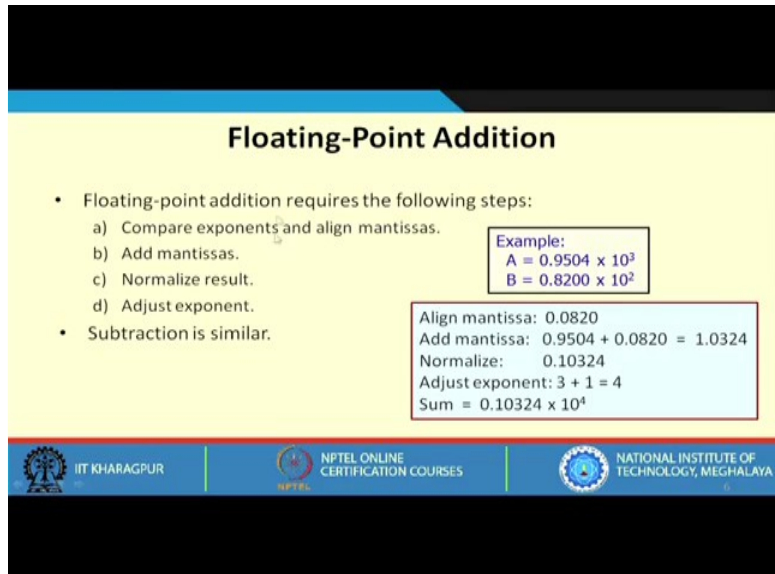


Here is a simple calculation. If we just assume that the delay of a full adder is tFA and a delay of a latches tL, so the clock period T has to be greater than equal to this full adder delay plus latch delay. After the pipeline is full, we can expect one result in every time T. See earlier in a full adder roughly we are getting one result every time 4 x delay of a full adder, but now in the pipeline implementation, we will be getting one output every clock cycle, and that is the advantage.

Suppose I have a program like this, where I am doing some addition a large number of times. Let say there are 10,000 additions. I can use this kind of a pipeline to advantage. After my pipe is full, I can get one output every cycle. Instead of waiting for one addition to be complete before starting the next addition, if we have a pipeline, it can definitely give you speedup for this kind of vector.

So, just like this if you are adding two arrays and number or the elements are being fed to the pipeline in sequence one by one, then this kind of arithmetic pipeline can be used to benefit. Basically wherever there is something called vector kind of arithmetic, pipelining can give you great benefit. This again we shall come back to later.

(Refer Slide Time: 08:22)

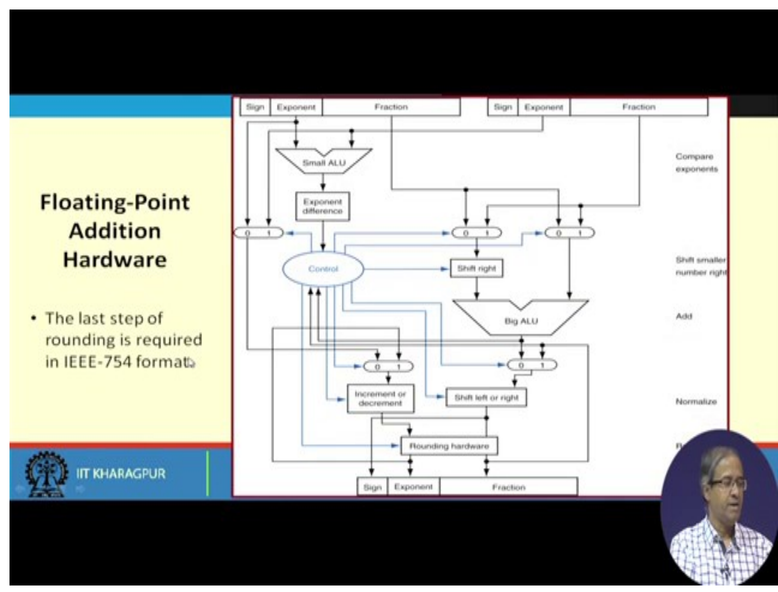


Let us now come back to floating point arithmetic. That is how it can be pipelined. Let us look at floating-point addition, this we have already seen in detail. Earlier floating-point addition requires these steps of comparison of exponents and alignment of mantissas. Then, addition of mantissas, normalization of the result, and because of normalization, you may need to adjust the exponent. Subtraction will be very similar. Instead of addition we will be subtracting; an example for two decimal numbers are shown here.

Suppose I have two numbers A and B; with exponents 2 and 3. I make the second one also 3. This is called alignment of mantissa. After alignment I add 0.9504 and 0.0820. I get this let say normalization means it starts with 0. So, here I do a normalization. That means, I shift right by one position.

So, exponent was 3, but because I have shifted right, I have to adjust the exponent and I have to make it 4. So, my final result will be this into 10 to the power 4. These steps can easily be mapped into the various pipeline stages.

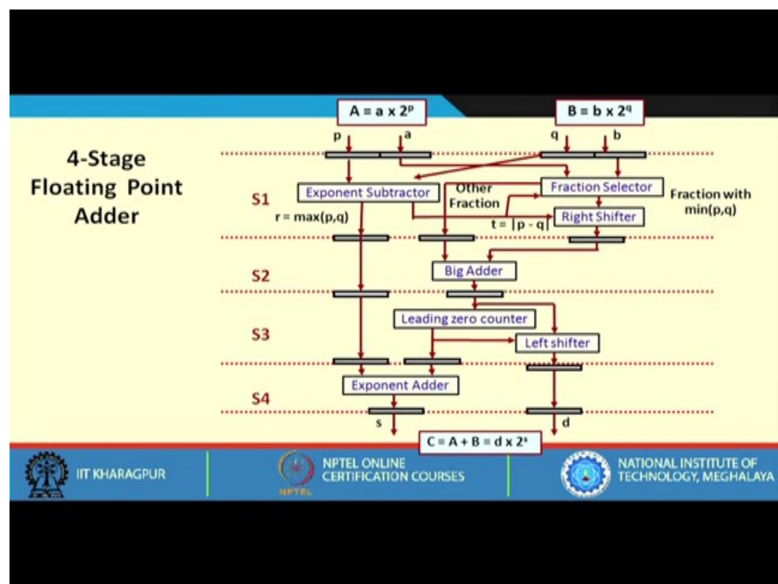
(Refer Slide Time: 10:02)



This is the floating-point addition hardware that we saw earlier. This is a non-pipelined version. Just a quick recap: the two numbers to be added over here, sign, exponent and the mantissa of the fraction, first we are comparing the exponent by using a subtraction using a small ALU, and after subtraction you see the difference in the exponents will have to this, smaller number fraction by that many bits. So, there will be a shifter. After shifting, there will be multiplexer to select the correct fraction, there is an ALU which will be carrying out addition and after addition, we will have to do normalization. We will have to check whether you have to do increment or decrement, according the shifting left or right and for IEEE format, you may have to do a rounding in the last step. So, that step is also shown.

This already we have seen earlier. Now, these basic steps you can easily break into as pipeline stages. I am showing one simple implementation of a pipeline like this. Let say the two numbers are A x 2 to the power P, and B x 2 to the power Q.

(Refer Slide Time: 11:14)

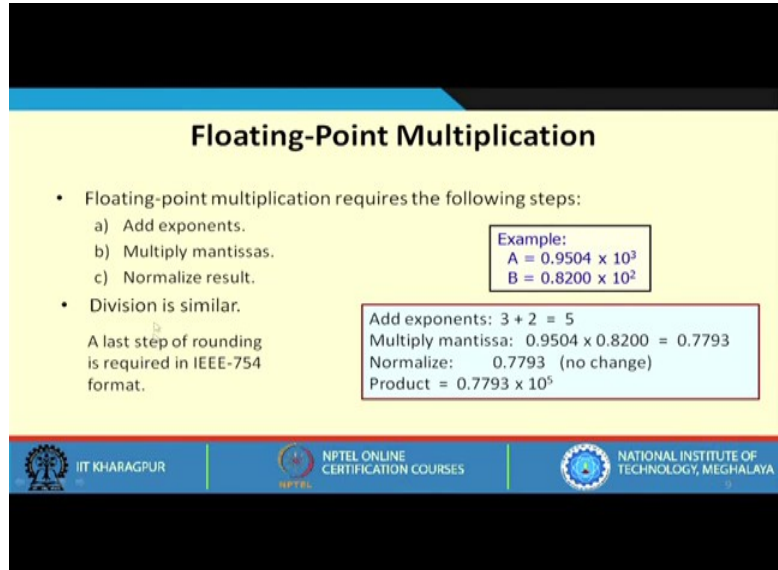


Here I am not showing the sign, just the mantissa and the exponent. This is P and this is A, this is Q and this is B. In the first stage S1, I am comparing the exponent. There is an exponent subtract. This P is coming and this Q is coming. That is subtracting.

The output will be R which will indicate which one is greater. That means, R is the greater of the two, P or Q. That also is known and after doing this, you select the fraction which one is smaller. So, either A or B, one of them is selected and after subtraction whatever is the result that many time P - Q, these many positions you are doing a right shift. This is your mantissa alignment. So, after mantissa alignment, you take the other fraction here and this shifted fraction here, you do an addition in stage S2. You are adding the mantissas.

In stage S3, after adding you have to do normalization. You will have to have a circuit that will count the number of leading 0's and depending on that, it will be shifting the mantissa left by that many positions. This is stage S3 and depending on how many positions you have shifted your exponent will have to be added by that number. So, this exponent correction, well here I am not showing normalization. Just simple floating point addition as a pipeline and these small rectangular boxes are the latches. So, finally you get the result, and it is a 4-stage simple pipeline.

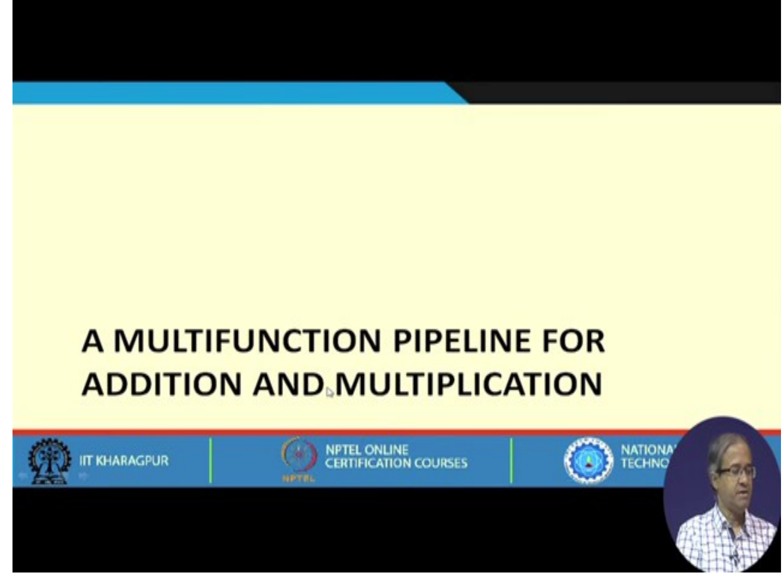
(Refer Slide Time: 13:38)



Now, floating-point multiplication is a little simpler because here you do not need to align the mantissas before the operation. You simply add the exponents and multiply the mantissas. Then, normalize like if you have two numbers like this. You add the exponents 3 + 2 = 5. You multiply the mantissas straightaway and it is 0.7793. It is already normalized. No change and see your result is this into 10 to the power 5.

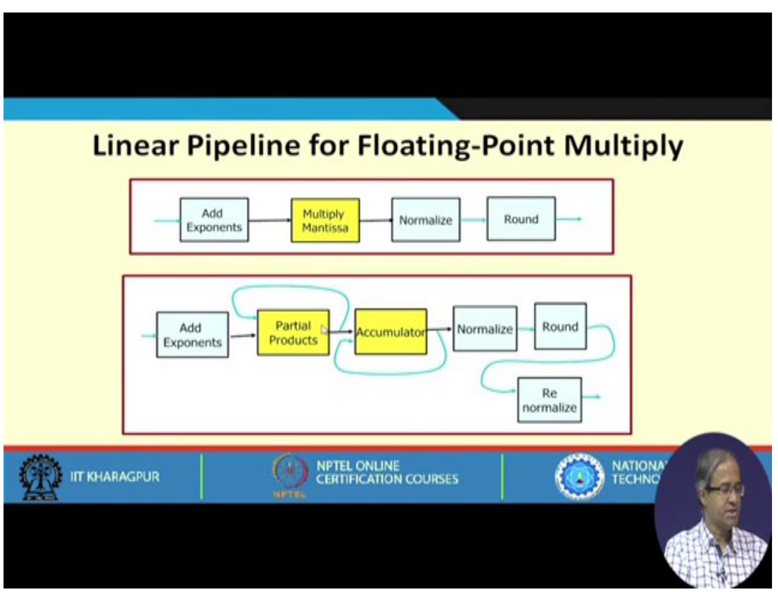
Division is similar to multiplication. Here you will be doing subtraction, here you will be doing division and again for IEEE format at the end, you may need a step of rounding.

(Refer Slide Time: 14:31)



For multiplication and division we have already seen earlier how we can implement that. Now, we will be seeing a multifunction pipeline for addition and multiplication together, because already we have seen non-linear pipelines in our last lecture. This is a simple pipeline for floating point multiply.

(Refer Slide Time: 14:49)

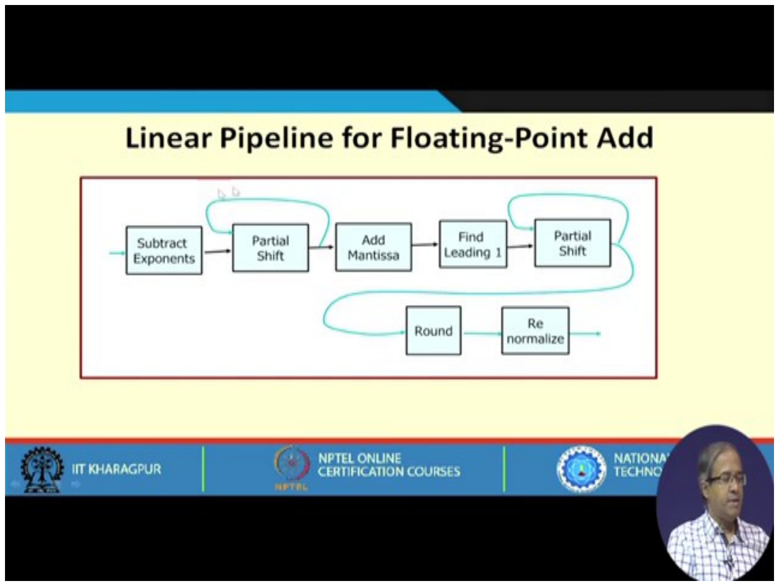


I am showing a simplified diagram. You do not add exponents, multiply mantissa, normalize rounding, but the thing is that when I am saying add exponents, multiply mantissa for example, or normalize these may require multiple steps because multiplication and addition are not of the same complexity. Multiplication takes more time than addition. You may have to compute the partial products, then add them. It may need multiple clock cycles.

So, technically more correct visualization will be something like this multiply mantissa. You can break it up into two boxes. One is the generation of partial products. This also will be an iterative process, and for every iteration, you will be doing an addition, you will be accumulating the partial result sum.

So, both these two stages may be required to execute it for multiple clock cycles and normalize and rounding and at the end, you may need to do renormalizations after rounding because the number might become again un-normalized.

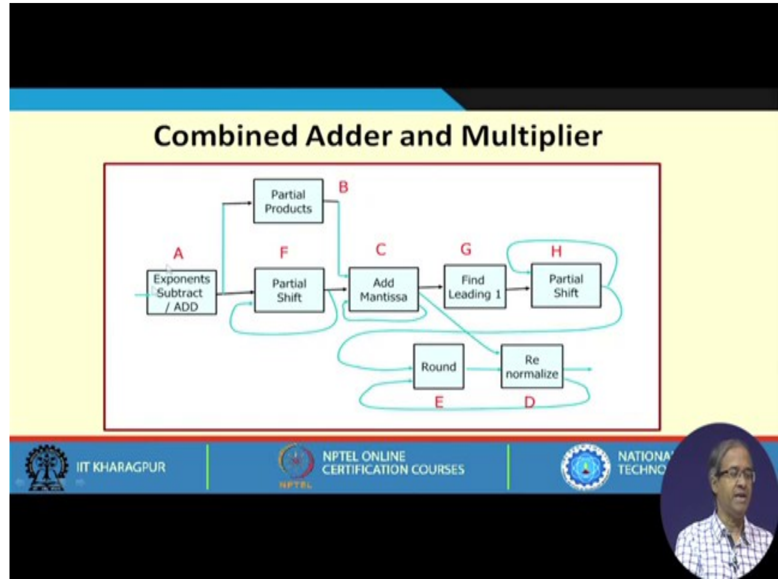
(Refer Slide Time: 16:37)



This is a more technically correct version of a floating point multiply pipeline with the IEEE format, because for IEEE format, you need this last step. And if you look at the floating point add, you need a mantissa alignment initially. For mantissa alignment, you have to shift the mantissa certain number of times. This again can be an iterative step; after the shifting, then you do an addition, you find the number of leading ones, then you will have to do a right shift or left shift.

Here left shift by that many position, this also can require multiple number of cycles, the rounding and for IEEE format renormalize. So, you see with respect to the previous diagram, there are some similarities.

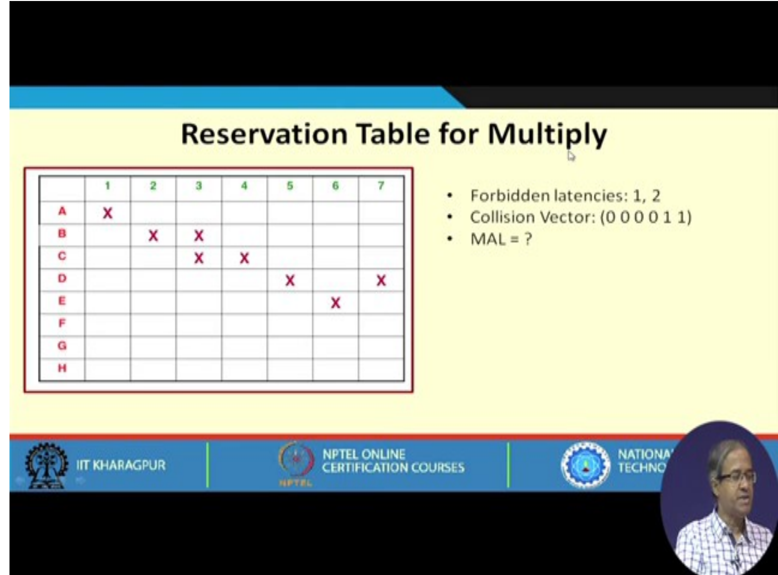
(Refer Slide Time: 17:29)



We can combine both the diagrams into a single pipeline like this. This is a combined adder and multiplier. You see this partial shift will be required for addition, partial product will be required for multiplication, but for both, these cases mantissa addition is required. Similarly partial shift will be required for addition, but for multiplication you can straight away go from add mantissa to renormalize. You can skip these two steps because leading one and partial shift will be required only for addition. From multiplication you can straightaway skip G and H; you can straightaway come to D here.

So, if you have a multifunction pipeline like this, I am not going into much detail on this. I will just give you an idea and then, you can similarly construct the reservation tables for addition.

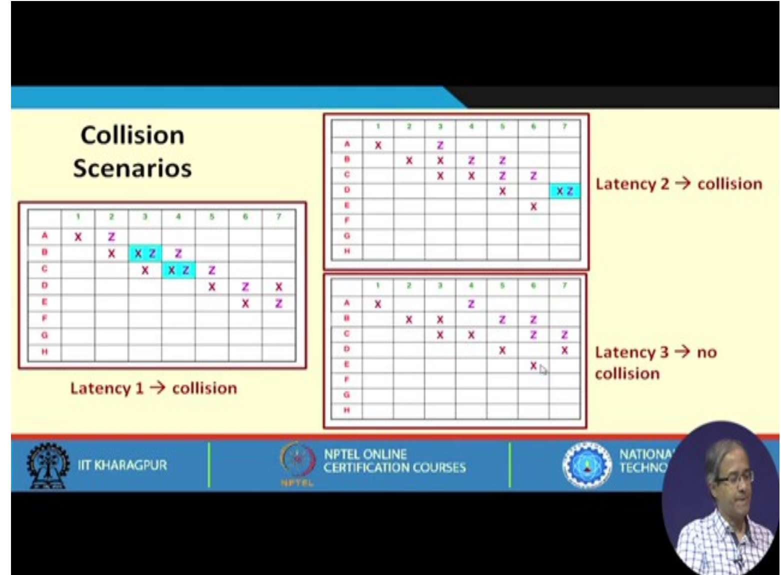
(Refer Slide Time: 18:28)



For multiply operations, you will be needing ABCD and what we are saying is that steps B and steps C will be requiring little more time. We are seeing that it is taking two time steps because partial shifting or the partial products and the add mantissa, this will take more time, and because of that I am assuming that this stage is B and C are being used for extended period.

For this reservation table, you see that the forbidden latencies are 1 and 2. So, the collision vector will be this. So, intuitively you can say that your minimum average latency will be 3, because 1 and 2 are both forbidden. So, 3 we can use definitely here. So, let us see for latency 1, there will be collision.

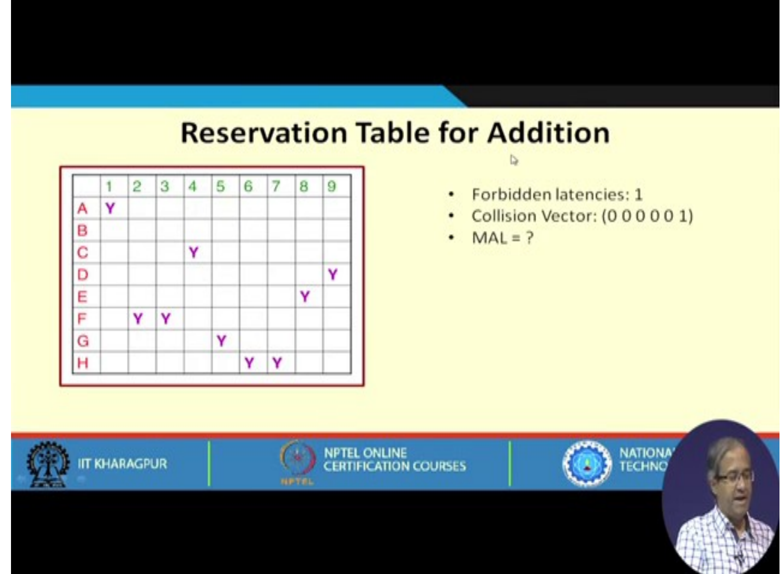
(Refer Slide Time: 19:43)



You see this was the reservation table. So, this X is the first data. Suppose I am feeding the second data, I mean it is after one cycle. So, Z has shifted here, Z has shifted here, X as you see X and Z will collide in B and C. So, latency 1 will lead to a collision. So, X and Z refer to consecutive data items which are fed, Z is fed earlier X is fed now.

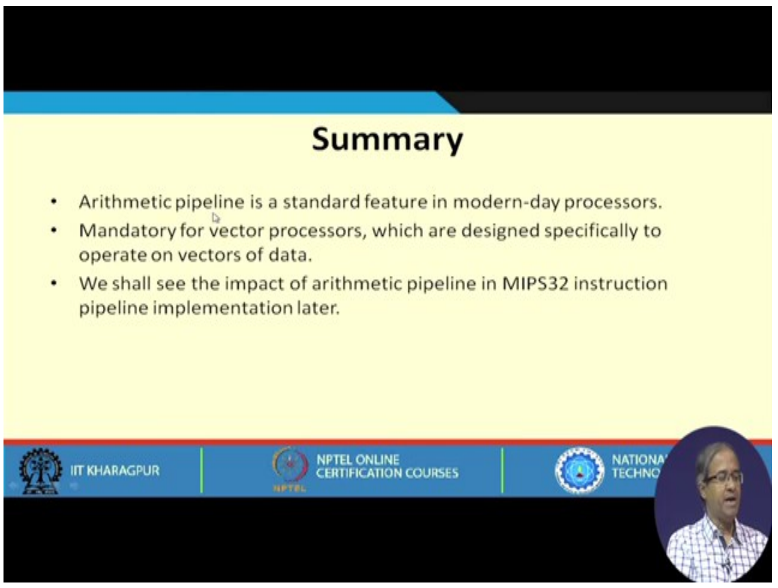
Now, if I use latency 2 that will also lead to a collision. Latency 3 does not lead to collision.

(Refer Slide Time: 20:53)



For multiplication you can feed the data with a latency of 3 without any problem. Similarly for addition, the reservation table will look like this. So, here again this F and H are been used extended period here. There is only one forbidden latency, 1. Here minimum average latency will be equal to 2. So, here you can similarly justify that.

(Refer Slide Time: 21:17)



To summarize this arithmetic pipeline is a very standard feature in modern day processors. We shall see later in some detail about vector processors. Vector processors are computer systems that are specifically designed to operate on vectors of data very fast. We shall of course first be discussing later in the next week that how we can build or how we can convert our MIPS architecture that we have seen earlier into a pipeline, how we can execute the instructions faster, but then we shall see that how we can also augment that pipeline with arithmetic pipeline concepts, so that vector arithmetic operations can also be made faster.

What we have seen today? We have seen some very simple examples of some arithmetic pipelines, and one instance of a multifunction pipeline of multiplier and added together. We have seen that we can speed up operations by implementing arithmetic circuits as a pipeline, provided we have continuous stream of data to be fed to that pipeline. That is possible only if we have vectors of data to be operated on.

We shall see the impact of arithmetic pipeline, we shall see later that arithmetic pipeline can complicate the design of the MIPS32 pipeline, but this we shall be seeing only later.

With this we come to the end of this lecture. There are a few things we shall be discussing after this; we shall be looking at the input-output process in a computer system, how peripheral devices are interfaced to a computer system, what are the different characteristics of the peripheral devices that makes the interfacing somewhat more complicated or complex as compared to interfacing memory devices.

And also, you shall be looking at some of the commonly used bus standards that we see everywhere today, and of course, after that we shall be looking at how we can make our computers faster by incorporating pipeline at the instruction level, and also as I had said at the arithmetic level. So, with this we stop for today.

Thank you.