**Computer Architecture**
**Prof. Madhu Mutyam**
**Department of Computer Science And Engineering**
**Indian Institute of Technology, Madras.**

**Module – 05**
**Lecture - 13**
**Fundamentals of Pipelining (Part 1)**

So, in this module we are going to discuss the fundamentals of pipelining. And we find pipelining applications in many scenarios in our real life. We will consider one example. So, for example, consider the admissions in an engineering college or in a university. This consists of the set of steps a student has to go through for taking up seat in university or college. So, first he has to go for verification of medical records through a doctor.

Next the student has to go to a person who will verify the certificates. And finally, once the seat is allotted the student has to go to a person to pay the fee. So, effectively in this process we can see the student go through different stages before paying the fee. And now while doctor is busy with verifying medical records of a student, the person who is verifying the original records, the original certificates will be free. And similarly, the person who is collecting the fee also will be free. So, now if, at any point of time if one student is taking services of all these three people, so, overall the efficiency will not be there.

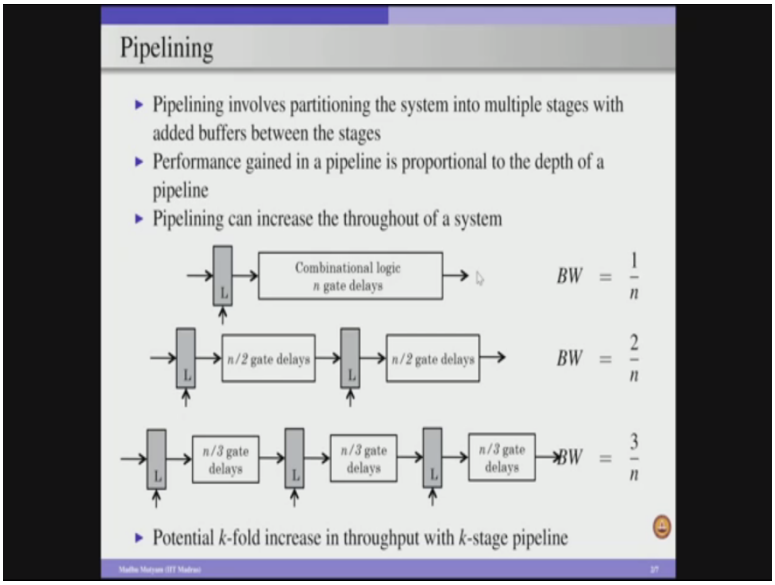
So, overall efficiency will be significantly degraded. So, rather because these three persons are doing three different tasks, so, what we can do is, while one student is consulting a doctor for medical record checking the other student can go to the person who is verifying the certificate. So, the third person can go to person who is collecting the fees. So, in other words while the first student is busy with the payment of fee, the second student will be busy with certificate verification and the third student or the new student entering the queue will be busy with the medical record verification.

Let us assume that each of these tasks will take x amount of time. And the first student will complete his task after the x amount of time and after that every x units of time one student will complete his task, because all these three tasks are independent and then, so as a result, like so different students will be there in different phases of their admission process. So, the

overall the time it takes for completing the admission process for, let us take for hundred students, will be significantly reduced.

And such types of applications we can find in real life scenarios plenty. So, effectively we can see pipelining applications everywhere and as a part of computer architecture course we are going to focus on pipelining of instruction execution, but in this particular module we are going to see the basics of pipelining. And then we will illustrate this pipelining with an example by considering a combinational circuit.

(Refer Slide Time: 03:36)




So, a pipelining can be defined as partitioning a system into multiple stages and each of these stages are separated by a set of buffers. So, that means like between two stages we will put a buffer and that buffer is going to store the computations whatever is done by previous stage. So, effectively if I divide my overall computations into n stages. So, I will consider n buffers intermediate to each of these pair of these stages. They store the intermediate computed values. So, once we have this, a process of dividing a system into smaller pieces.

So, that multiple independent computations can be processed through this different stages. And as a result the performance can be improved. Performance improvements we get through pipelining is proportional to the depth of the pipeline. If I divide a computation into n tasks, in an ideal scenario, I can get a performance improvement of n times. And this performance is actually, we can also represent using throughput and the pipelining will improve the

throughput of the system. So, consider an example here, let us assume that we have here a combinational circuit, which takes n gate delays to compute a task.

So, here given an input to the combinational circuit, it takes n gate delays to produce the output. Now, if this entire combinational circuit is not partitioned, then effectively we will process the given inputs for every n gate delays. So, in other words, we will get a bandwidth of one by n. So, here one computation will be completed for every n gate delays.

Now, if I want to partition this combinational logic into two pieces, in an ideal scenario, each piece is going to take n/2 gate delays and since I partitioned the combinational logic into two parts and I separated these two parts by using latch. And so the first part, half of the computation will take n/2 gate delays and after n/2 gate delays, then I will get the partial output and when we apply the clock then so, this will be latched in the intermediate buffer. And once the data is stored in intermediate buffer and then this will be taken as input to the second stage of computation which takes another n/2 gate delays. And we will get the output at the end of n gate delays with added latency of the propagation delay through the latches.

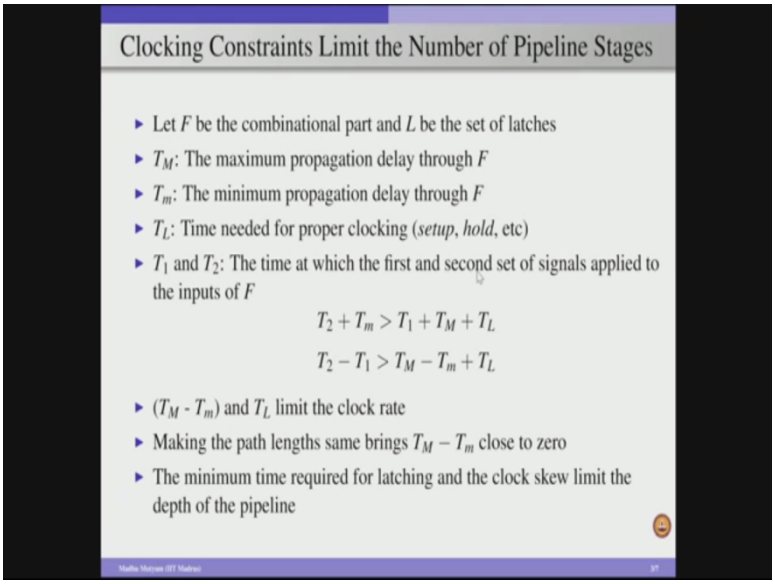
So, here each of these latches have a clock. So, effectively we latch the input, we latch the output from the previous stage only at a particular clock pulse. And once it is latched, then we can use this as input to the next stage and this will continue. So, as a result, once two stages is separated by latch so, while the second stage is taking input from latch, the first stage can work with new set of inputs and the first stage output will be written to latch only at regular clock pulse and that we need to determine based on the computational delay required for each of these individual stages. And once we divide the computation into two parts our bandwidth will be effectively 2/n, because we will get the output at every n/2 gate delays except the first one. The first one is going to take n gate delays excluding the delay through the latch, but after that every n/2 delays we will get the next output. So, effectively our bandwidth will be 2/n.

So, in other words, in n gate delays we are going to compute two inputs. And now, if I continue the logic again, for example, divide the combinational logic into three parts, by considering the three stages and each adjacent stages are separated by a latch. And now each of these sub stages is going to take n/3 gate delays and effectively our bandwidth will be 3/n. In other words, if I divide my combinational logic into k stages, I can get a bandwidth of k/n.

In other words, the bandwidth will be improved by k times. So, potential k fold increase in throughput is obtained by k stage pipeline.

Again, this is an ideal scenario, where we will get this k fold increase, but in reality we may not get this k fold increase. So, now having discussed this, so one can say that OK, I can just go for any k, so that, I can improve throughput significantly. If we consider k very large, then you can get a significant improvement according to this statement, but in reality that is not the case. So, there are several factors impact the number of pipeline stages or the depth of the pipeline.

(Refer Slide Time: 09:23)




So, the major constraint which limits the number of pipeline stages or depth of pipeline is the clocking constraints. So, let us assume that we have a combinational part which is denoted as F and the set of latches denoted by L. And now this combinational logic will take some amount of time for processing the inputs. And since the combinational logic can have multiple paths from input to output, so, as a result we need to consider two timing parameters one is the $T_M$ which is the maximum propagation delay through the combinational logic F.

And there is another one $T_m$ which is the minimum propagation delay through the combinational logic. And since we also considered latches, which are separating the adjacent stages of the pipeline and it takes $T_L$ amount of time. The $T_L$ amount of time is needed for proper clocking because a latch requires a setup and the hold times and so on.

So, effectively so given an input, the input will take either $T_m$ or $T_M$ amount of time to reach the output from the inputs and it requires another $T_L$ amount of time to be latched properly in the buffer or pipeline register. Now assume that, we are planning to give inputs to this combinational logic at $T_1$ and $T_2$ and these are the timings at which we want to apply the pipeline the clock.

So, $T_1$ , $T_2$ represent the time at which the first and second set of signals applied to the inputs of F. Now once we have mentioned $T_1$ , $T_2$ and also we have $T_M$, $T_m$ and $T_L$, we will give an equation, inequality which is $(T_2 + T_m) > (T_1 + T_M + T_L)$. This indicates that, the second set of inputs can be given to this combinational logic or pipeline logic, at a time which should be greater than the time at which the first signal is given plus the time it takes for processing the first set of inputs.

The maximum time it takes for the first set of inputs to reach the output plus the time it takes for proper latching minus the minimum amount of time the second set of inputs will go from input to the output. In other words $(T_2 + T_m) > (T_1 + T_M + T_L )$. So, $(T_2 + T_m)$ actually represents the time at which the second set of inputs are applied to the combinational logic and the minimum amount of time these inputs take to reach from the input to the output.

And $T_1 + T_M$ indicates the time at which the first set of inputs applied to this combinational logic and the maximum amount of time, these inputs are taken to reach from input to the output. And once the inputs are processed by the combinational logic and these outputs have to be stored properly in the latch. So, that is going to take $T_L$ amount of time. So, if we rearrange this inequality, we will get $T_2 - T_1$, is the interval between the first set of signals applied to the combination logic and the second set of the inputs applied.

In other words, the clocking time between two sets of inputs applied to the pipeline, of pipelining, of this combinational logic, which is greater than $(T_M - T_m + T_L)$. In other words, we have to apply clocking such a way that it should be greater than the difference between the maximum time and minimum time for the inputs to be processed by the combinational logic plus the time it takes for proper latching.

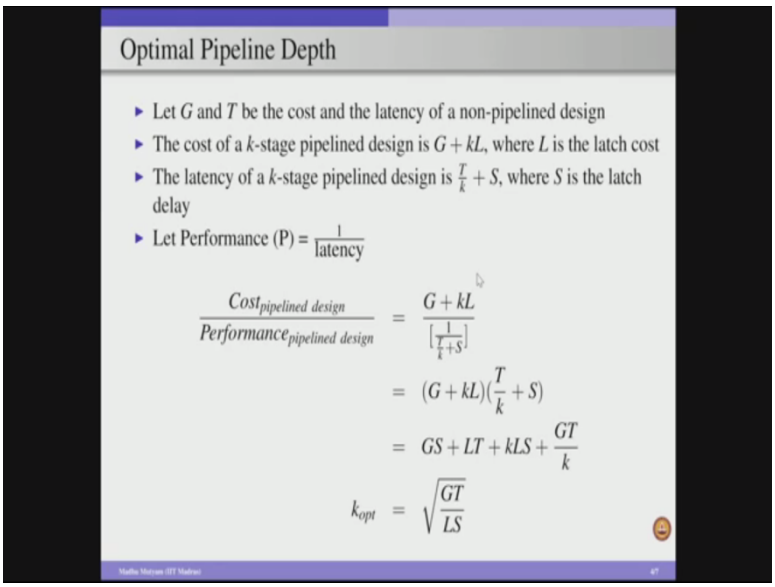
And again like, if we design our combinational logic in a proper way such that all the paths, critical paths are of equal length, if we make that way then $T_M$ is almost equal to $T_m$. Once we have the input to the output path length is same, even if there are n numbers of paths from

inputs to outputs. And if we make the lengths of these paths equal then our maximum propagation time and the minimum propagation time will be almost same.

So, as a result our $T_2$ minus $T_1$ will be greater than just $T_L$. In other words, our depth of the pipeline is determined by the $T_L$, the minimum time required for latching and the clock skew limit, the depth of the pipeline. Here the clock skew is because when we are dealing with synchronous circuits, so all the components in the synchronous circuits will work synchronously by using a clock. So, there is a clock which generates this clock signals. And these clock signals may reach different components at different times.

There may be as slight variation and that variation is called clock skew. So, effectively our number of pipeline stages is determined by this clocking constraint. So, once we have this, then we can clearly see that we can divide our combinational logic to a maximum number of pipeline stages by satisfying this clocking constraints. As long as we take care of $T_L$ and the clock skew, we can divide our combinational logic into n number of pipeline stages. And this foil is actually determining the maximum number of pipeline stages that we can come up with for a given combinational logics, or combinational circuit computation, but again not always this maximum number of pipeline stages is optimal. So when I say optimal, so, depending on the requirements our optimality is determined.

(Refer Slide Time: 16:23)



So, we will consider optimal pipeline depth by considering the cost as well as the performance. Let us assume that, G be the cost of a non-pipeline design and T be the latency of non pipelined design. And since, we are planning to make this design into a pipelined design so, we will need to consider set of latches and the cost and the latency associated with the latches are determined by L and S. So, the cost of a k stage pipeline design is G + kL because we know that once we divide our non-pipeline combinational circuit into a pipelined design by dividing that into stages k stages.

So, we need to consider k latches. So, effectively our overall cost associated with our earlier non pipeline design is increased by k into L. So, effectively the total cost associated with this k stage pipeline is G + kL, where G is the cost associated with the non-pipeline design and k is the number of stages and each stage we require a latch and L is the cost associated with the latch. This cost can be defined as the area it takes and now, we will now look at the latency point. So, we know that a non-pipeline design is going to take a latency of T. So, once we consider a pipeline design and that too k stages.

So, our overall latency is decreased significantly and that is represented by (T/k) + S because we divided our non-pipelined design into k stages. So, each stage is ideally going to take T by k amount of time and there is some latency associated with propagation delay of the latch that is represented by S. So, effectively for each stage our latency is now (T/k) + S. So, in summary once we divide a non-pipeline design into k stage pipeline, then we have, we incur a cost of G + kL, but the latency is reduced to (T/k) + S. Let the performance be defined as one by latency. Now, we want to see whether how many pipeline stages are required if our design is optimal in terms of cost per performance.

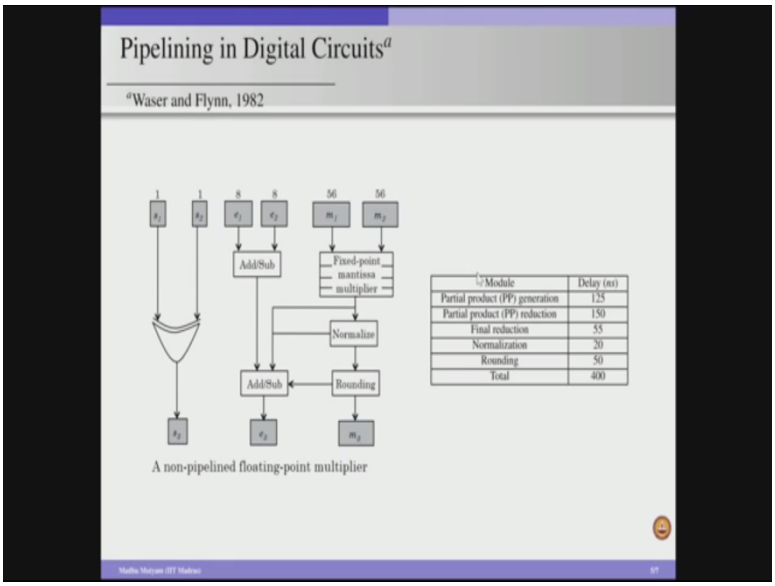
So, cost per performance is cost into latency according to the above equation. So, effectively we will have the overall expression becomes GS + LT + k * LS + GT by k. And now we want to find the optimal number of pipeline stages which minimizes our cost per performance. We have to incur least cost, at the same time; we have to get maximum performance. So, effectively we take the first derivative of this expression and equate that to zero and the derivative with respect to k because we want to find the number of pipeline stages.

Once we do that, then we get k optimal equal to a square root of GT / LS. So, in other words, for a given combinational circuit which has a cost of G and the latency of T and latch cost is

L and latch latency is S, then we can come up with the optimal number of pipeline stages for the combinational circuit by considering the square root of GT / LS number of stages.

So, this indicates that given a combinational circuit, if we want to optimize the design for cost per performance, then we cannot divide the combinational circuit by not more than square root of GT / LS number of pipeline stages. So, having discussed this now we will consider an example of a floating point multiplier.

(Refer Slide Time: 20:52)




And this example is taken from research paper published by Wasser and Flynn in 1982. So, here we are considering a fixed point multiplier for floating point numbers. And we know that the floating point numbers have sign bit exponent and the mantissa. Of course, the example here the authors considered are not following the IEEE standards and as a result we have a mantissa of 56 bits with the hidden bit and 8 bits of exponent. So, we are considering a biased 128 exponent and a sign bit. And given two floating point numbers when we are multiplying, the sign will be determined by XOR of the two numbers and that will be obtained in S3 which is going to give the resultant sign of this multiplication.

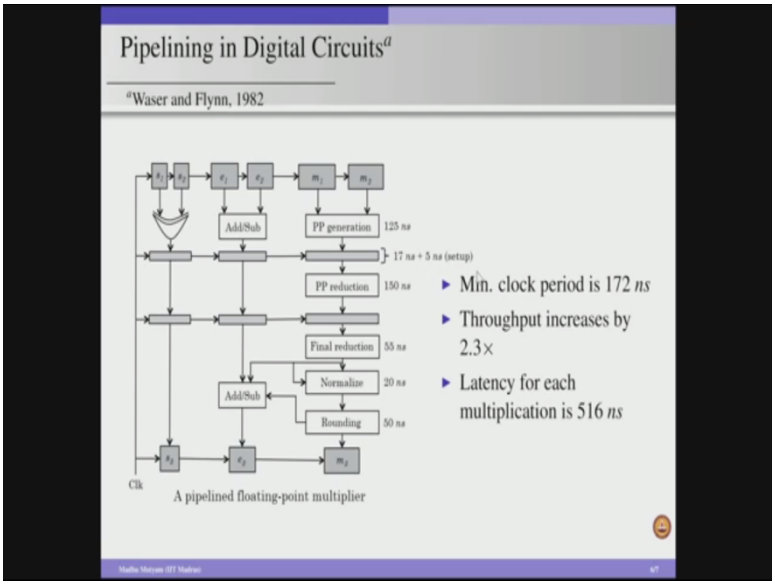
And we have exponents $e_1$ and $e_2$ and we need to perform the addition because we are performing a multiplication of two floating point numbers. So, we will just add the corresponding exponents. And in the case of mantissa we use a fixed point mantissa multiplier logic, which actually consists of the partial product generations and then we add

this partial products. So, that is represented as partial products reduction. And finally, once we have, we keep on adding these partial products to get intermediate results and finally we end up with two partial products and we just add that by using carry look ahead adder. Finally, we get the actual the value.

And once we value, we need to normalize that and also we may need to perform rounding. So, effectively, to perform this floating point multiplication of two floating point numbers and the authors observed that these are the total amount of time it takes for each of the components - 125ns for partial product generation, 150ns for partial product reduction, 55ns for final reduction and 20ns for normalization of the result, 50ns for rounding off. And overall latency it takes is 400ns for performing multiplication of two floating point numbers.

So, effectively, if we do not, if we are not going for pipelined design, we can perform a multiplication on two floating point numbers with a latency of 400ns, a delay of 400ns. So, if we want to pipeline this design, then how the performance is improved that we are going to see now.

(Refer Slide Time: 29:30)




So, we know that the multiplication part. The multiplication of these mantissa components consists of three portions; one is the partial product generation, partial product reduction and final reduction. And each is taking 125ns, 150ns and 55ns respectively. And the next one is normalization which is going to take 20ns and rounding is going to take 50ns. Now, given

this, delay values for each of these components, we can divide our total floating point multiplication process into a different number of stages, but this particular division is optimal. The reason is here, you can see, the first stage is going to take 125ns because add or subtract is going to take less amount of time than the partial product generation. So, as a result when we decide a pipeline stage, we have to consider a stage, a component in the pipeline stage is going to take maximum delay. So, in this particular case, the partial product generation is going to take more time. So, we are going to consider 125ns for this.

And now for second stage we can see the partial product reduction which takes 150ns. We cannot combine the generation and reduction into a single stage. If we combine this then effectively our overall delay for that stage is going to be 275ns, which is not optimal. And similarly, the last three components are put together, are considered as a single stage which is going to take 125ns. In other words, when we divide this floating point multiplication computation into three stages, the first one is going to take 125ns, the second one is taking 150, third one is taking 125ns.

And now, once we have this three divisions, the three stages when we divided the non pipeline floating point computation, we have to decide the pipeline cycle time. And whenever we divide any computation into different stages in a pipeline, the pipeline stage latency or the delay is determined by the maximum delay of any of stages. Here in this particular case, two stages are taking 125ns and one stage is taking 150ns. So, effectively the middle stage is actually determining the overall pipeline stage cycle time and which is 150ns plus because we are separating stages by using pipeline registers or latches.

And we need to consider the latency associated with the latch also. In this particular case, we considered 17ns, for processing, for storing the data in the pipeline, pipeline latch or pipeline register and 5ns for setup time. Effectively 22ns is the overhead we get, the overhead we incur because of this pipeline registers. In other words, our pipeline stage time or pipeline stage delay is equal to the delay associated with the stage computation plus the delay associated with the pipeline registers.

In other words, in this particular example, we get a pipeline stage delay of 172ns. So, once we have this pipeline design for our floating point multiplication after the first computation is done, if we are supplying the inputs to this floating point multiplier continuously, we can get outputs at every 172ns as compared to 400ns that we incur in the non-pipeline design. So, as

a result, we can easily see that this pipeline design of floating point multiplier is going to improve the throughput by 2.3 times, which is nothing but 400 by 172. And remember the pipelining is going to improve the throughput, overall throughput, of the system. As long as we supply the inputs to the pipelined unit continuously, but if we are supplying only one pair of inputs to this floating point multiplier, the actual latency it takes to compute the output is actually 560ns, as compared to 400ns in the case of non-pipeline design.

So, as a result, so if we are performing a single computation then pipelining may not be efficient. So, which actually incurs extra latency because of this the pipeline registers, but again when we are designing a pipeline, when we are considering a pipeline design, for computation our assumption is we are going to use that pipeline design for processing a large collection of inputs. So, that is the reason why the pipelining is going to improve throughput significantly. So, with that I am concluding this module and in the next module I am going to discuss instruction pipelining.

Thank you.