

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

**Lecture - 40**  
**Basic Pipelining Concepts**

In this lecture we shall be talking about some Basic Pipelining Concepts. Now you may wonder why we are discussing pipeline in the context of floating point arithmetic and floating point; I mean hardware implementation of arithmetic functions.

I would like to tell you here is that, pipelining is a very commonly used and widely used technique to enhance the performance of a system without significantly investing. In hardware if I want to make something faster I can always replicate some functional units. So, instead of one adder I can use 5 adders; my addition time will speed up 5 times. But here the philosophy is different; we are not replicating hardware, we are using a different kind of a philosophy by which we can promote something called overlapped execution whereby the performance improvement can be quite close to what we can achieve by actually replicating the hardware.

We are getting good return of investment without making any significant enhancement in the hardware. This is the basic concept of pipelining. The reason we are discussing pipelining here is that, we shall see how we can use pipelining to improve the performance of arithmetic circuits that you have already seen.

Later on we shall also see how we can enhance the performance of the processor, that is called instruction level pipelining; how instructions in the MIPS32 data path can be executed faster by implementing a pipeline there. But before that you will have to understand what is a pipeline, how it benefits the designer, and what kind of speed up you are expected to get. So, here in this lecture we shall be talking about the basic pipelining concepts.

(Refer Slide Time: 02:57)

**What is Pipelining?**

- A mechanism for overlapped execution of several input sets by partitioning some computation into a set of  $k$  sub-computations (or stages).
  - Very nominal increase in the cost of implementation.
  - Very significant speedup (ideally,  $k$ ).
- Where are pipelining used in a computer system?
  - **Instruction execution**: Several instructions executed in some sequence.
  - **Arithmetic computation**: Same operation carried out on several data sets.
  - **Memory access**: Several memory accesses to consecutive locations are made.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

As I had said pipelining is nothing but a mechanism for overlapped execution of several different computations. Several different computation means they are possibly carrying out some computation on several input data sets. The basic principle is that we try to divide our overall computation into a set of  $k$  sub-computations, called stages.

As I had said if we do this there is a very nominal increase in the cost of implementation. Again I shall be illustrating it with a real life example. But we can achieve very significant speedup that can be very close to  $k$ . This is really fantastic; we are not really investing in additional hardware, just we are dividing the hardware into smaller functional sub-blocks, and by doing that we get an architecture where we can equivalently get a speedup of up to the number of sub-functional units that you have divided our overall computation into.

This kind of pipelining can be used in several different areas of computer design. For instruction execution we shall see later that several instructions can be executed in some sequence, because you have already seen earlier that when instruction is executed in a processor basically there is a fetch and execute cycle. During fetch we fetch an instruction, then we decode it; depending on the type of instruction we can do the appropriate actions or operations to complete the execution. Now the idea is that when an instruction is being decoded and it is being executed, why not fetch the next instruction

and try to decode it at the same time; this is what I mean by overlapped execution. This is what is there for instruction pipelining.

Similarly, for arithmetic computation we shall be seeing how we can speed up arithmetic computations by implementing pipeline. Similarly for memory access also we shall see later again that we can use the concepts of pipelining to speedup accesses from consecutive locations in memory.

(Refer Slide Time: 06:16)

**A Real-life Example**

- Suppose you have built a machine  $M$  that can wash ( $W$ ), dry ( $D$ ), and iron ( $R$ ) clothes, one cloth at a time.
  - Total time required is  $T$ .
- As an alternative, we split the machine into three smaller machines  $M_W$ ,  $M_D$  and  $M_R$ , which can perform the specific task only.
  - Time required by each of the smaller machines is  $T/3$  (say).

Diagram 1: A single machine  $M$  performing tasks  $W$ ,  $D$ , and  $R$  sequentially. The total time for one cloth is  $T$ . For  $N$  clothes, the total time is  $T_1 = N.T$ .

Diagram 2: Three specialized machines  $M_W$ ,  $M_D$ , and  $M_R$  performing tasks  $W$ ,  $D$ , and  $R$  in parallel. Each machine takes  $T/3$  time. For  $N$  clothes, the total time is  $T_2 = (2 + N).T/3$ .

Logos: IIT KHARAGPUR, NPTEL ONLINE CERTIFICATION COURSES, NATIONAL INSTITUTE OF TECHNOLOGY KHARAGPUR

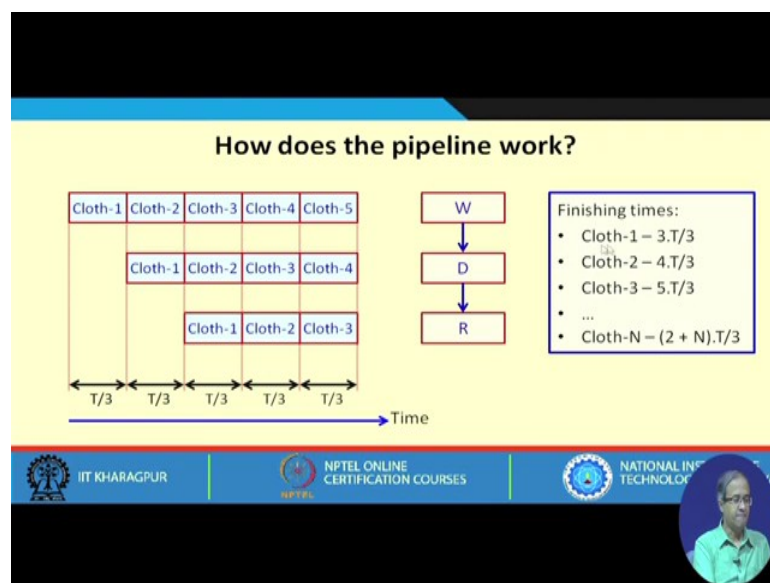
Let us take a real life example say of washing. Suppose we have a requirement where we need to wash clothes, dry them and iron them. Suppose I have a machine, which can do washing, which can do drying, which can do ironing. So, there are 3 stages of the machine. I feed my clothes to the machine. The clothes will be moving through the 3 stages and then I can feed the next cloth or next set of clothes whatever.

Let us assume that this whole thing takes a time  $T$ . So assuming that I can feed one cloth at a time, if there are  $N$  clothes I need to do washing, drying and ironing, the total time will be  $N \times T$ . Now as an alternative what we are saying that we are not buying 3 such machines, but rather we are breaking the machine into 3 smaller parts. There will be one machine which can do only washing, there will be one machine which can do only drying, and one machine which can do only ironing. So, roughly the total cost remains approximately equal to the total cost of the original machine. Assuming that they take

equal time, let us assume that earlier it was taking a time of  $T$ , now each of these will be taking a time of  $T/3$ .

We shall see very shortly that if I do this then for washing, drying and ironing  $N$  clothes, I need a time  $(2 + N) \cdot T/3$ . So, if  $N$  is large you can ignore 2; it is approximately  $N \cdot T/3$ , which means I have got a speed up of approximately 3 (equal to the number of pieces I have broken my original machine into). This is the essential idea behind pipelining; I can get a significant speedup.

(Refer Slide Time: 09:01)



Let us see, washing - drying - ironing; after washing is completed I will be giving it to the dryer, after drying is completed I shall giving it to ironing.

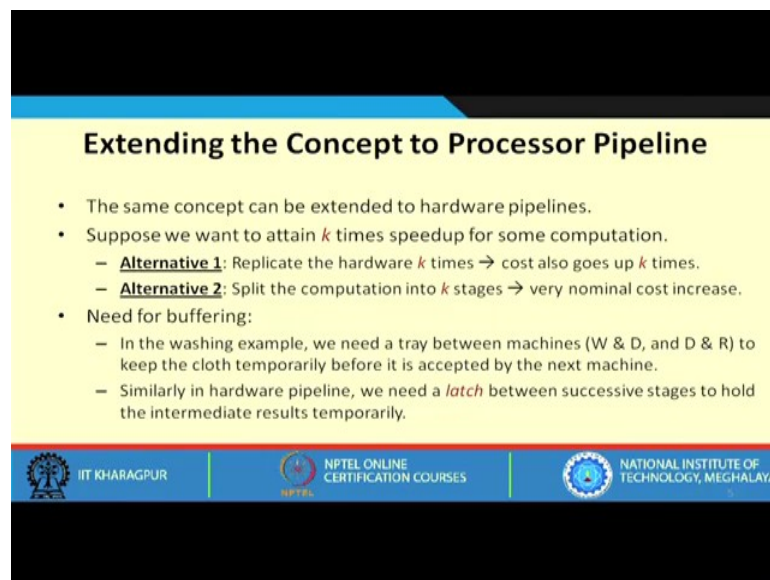
To start with, during the first time slot let us say the time slots are all  $T/3$ ,  $T/3$  each, the first cloth reaches the washer. It finishes washing at the end of this time slot. Then this cloth-1 will be moving to D. What will happen in the next time slot? This cloth-1 will be moving to D, but now the washer will be idle. So, I can feed this second cloth to the washer.

Now, the washer and dryer are working on two different clothes in an overlapped way. So, at the end of this time slot, dryer has finished with cloth-1 and washer has finished with cloth-2. So, what will happen next? Cloth-1 will come here to the ironer, cloth-2 will come here, and cloth-3 will come to washer.

Now you see you see all these 3 machines are busy. Now at the end of this time what will happen? Cloth-1 is done and will be taken out; at the next time cloth-2 will come here, cloth-3 here, and next cloth will come here.

You see after this initial two periods of time that is required for this pipeline to be filled up, after that in every time slot I am getting one output; that means, one cloth is finishing and I can take them out. That is why I talked about the total time as  $(2 + N) \cdot T/3$ .

(Refer Slide Time: 11:36)



**Extending the Concept to Processor Pipeline**

- The same concept can be extended to hardware pipelines.
- Suppose we want to attain  $k$  times speedup for some computation.
  - **Alternative 1:** Replicate the hardware  $k$  times  $\rightarrow$  cost also goes up  $k$  times.
  - **Alternative 2:** Split the computation into  $k$  stages  $\rightarrow$  very nominal cost increase.
- Need for buffering:
  - In the washing example, we need a tray between machines (W & D, and D & R) to keep the cloth temporarily before it is accepted by the next machine.
  - Similarly in hardware pipeline, we need a *latch* between successive stages to hold the intermediate results temporarily.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

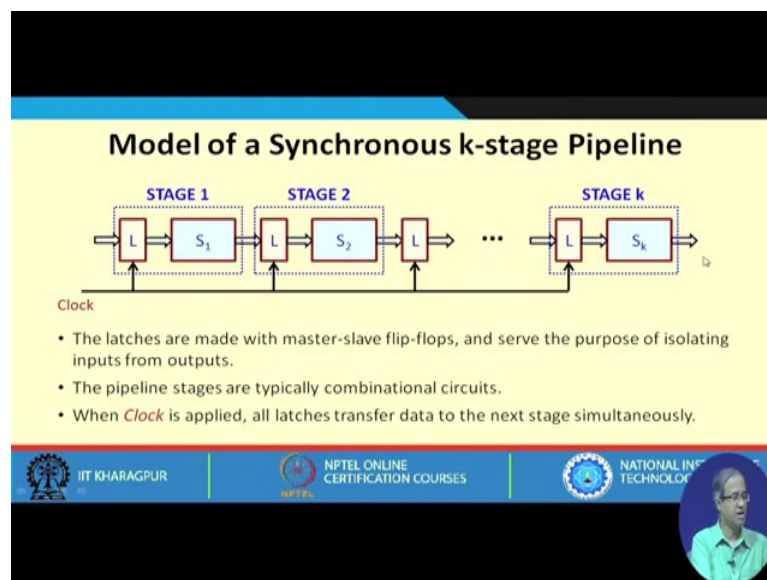
We can extend the concept discussed so far to actual processor designs. Suppose for some computation we want to achieve a speedup of  $k$ . The two alternatives are with us. We can use  $k$  copies of the hardware, this will obviously give us a speedup of  $k$ , but the cost will also go up  $k$  times because we are replicating the hardware.

Alternative 2 is pipelining. We are splitting the computation into  $k$  number of stages. This will involve very nominal increase in cost, but potentially it can give you a speedup of  $k$ . But one thing is required which we ignored; you see if you think of the washing example – washing, drying and ironing, see after one machine finishes and a cloth comes out and goes to the input of the other machine, I need some kind of a tray or a buffer in between. The cloth will be temporarily stored in that buffer before it can be fed to the next machine. Because it may so happen then when the first machine finishes with washing, the dryer is still working on the previous one, and it is still not free.

So, only when it is free then only the cloth can be given to the dryer. Thus, I need some kind of a buffering mechanism in between the stages. This is what we talk about here – the need for buffering. For the washing example it is said we need a tray between the machines to keep the cloth temporarily before the next machine accepts it. In exactly the same way when we want to implement pipeline in hardware, we need something similar to a buffer. It is nothing but a latch or a register between successive stages, because one stage finish some calculation and prior to giving it to the next stage, maybe the next stage is still not finished.

So, that value is temporarily kept in the latch, so that the next stage whenever it is free can take it from the latch. This is the idea behind hardware pipelining.

(Refer Slide Time: 14:41)



I am showing a schematic diagram of a k stage hardware pipeline. This is called a synchronous pipeline because there is a clock which is synchronizing the operation of all these stages. So, as you can see there are k number of stages. Each stage involves some computation, S<sub>1</sub>, S<sub>2</sub> to S<sub>k</sub>, and also there is a latch. The latch will temporarily store the result of the previous stage before the next stage is ready to accept it. The clock will be generating active signals periodically and clock period will be large enough, so that all these stages can finish their computation and also it should take care of the delay of the latch. This we shall come a little later how the clock frequency or the clock period can be calculated. The stages S<sub>1</sub>, S<sub>2</sub>, ... are typically combinational circuits. So, whenever the

next active edge of the clock comes, the next data is fed to S1, the output of S1 goes to S2, S2 goes to S3, and so on. There is a shift that goes on whenever the clock signal appears. In synchronism with the clock the pipeline shifts result from one stage to the other in a lock step fashion.

(Refer Slide Time: 16:34)

**Types of Pipelined Processors**

- Can be classified based on various parameters:
  - a) Degree of overlap
    - Serial, overlapped or pipelined
  - b) Depth of the pipeline
    - Shallow or Deep
  - c) Structure of the pipeline
    - Linear or Non-linear
  - d) How the operations are scheduled in the pipeline?
    - Static or Dynamic

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

You can classify or categorize pipelined processors based on various parameters like degree of overlap, depth of the pipelining, structure of the pipeline, and how we schedule the operations.

(Refer Slide Time: 16:57)

**(a) Degree of Overlap**

- Serial
  - The next operation can start only after the previous operation finishes.
- Overlapped
  - There is some overlap between successive operations.
- Pipelined
  - Fine-grain overlap between successive operations.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA




Let us very briefly look at these. When you talk of the degrees of overlap, one extreme can be serial that is a strictly non-pipeline implementation. The next operation can start only after the previous operation finishes. So, here I shown it schematically like this, this is an operation let us say which takes 1 2 3 4 5 steps. So, only after the first operation is completed only then the second operation can start. It is strictly sequential, with no overlap. There can be partial overlap as the second diagram shows. Partial overlap naturally results in a speedup because earlier it was taking so much time, but now it is taking 2 time units less. In the extreme case you can have something called pipeline as I said. There is almost complete overlap; when the first operation he is finished with the first step it moves to the second step, and the second operation can start with its first step. This is called fine-grained overlapped execution.

Here naturally the time required is much less. So, depending on the degree of overlap you can classify how deep or how efficient your pipeline implementation is. Ideally you should have something like this.


(Refer Slide Time: 19:12)

**(b) Depth of the Pipeline**





- Performance of a pipeline depends on the number of stages and how they can be utilized without conflict.
- Shallow pipeline is one with fewer number of stages.
  - Individual stages more complex.
- Deep pipeline is one with larger number of stages.
  - Individual stages simpler.



Shallow



Deep


IIT KHARAGPUR

NPTEL ONLINE CERTIFICATION COURSES

NATIONAL INSTITUTE OF TECHNOLOGY


Then comes the depth of the pipeline. We have seen earlier that if there are  $k$  number of stags, then we can have a potential speedup of up to  $k$ . Then you can argue why do not have a very large value of  $k$  so that we can get a very large speedup. But there are some limitations to increase the value of  $k$ ; we cannot break up a computation into smaller



pieces beyond a limit. Beyond a limit it does not make sense, maybe the delay of the latch will be more will become more than the delay of the computation.

So, the depth of the pipeline is an issue. You can either have a shallow pipeline with a few number of stages, or you can have a deep pipeline with large number of stages. If you have 9 stages then potentially you can have a speedup of 9. Depth of pipeline is a design issue. But it is also very important to evaluate that you can increase the depth all right, but you must also see in what way you can allow the computations to proceed so that you can have overlapped execution without any conflict. We shall be seeing later that conflicts in a pipeline are very important and there are so many techniques to handle them.

For a shallow pipeline because we are making the number of stages smaller, individual stages are more complex. But if I have a very large number of stages, each stages will become simpler.

(Refer Slide Time: 21:18)

**(c) Structure of the Pipeline**

- **Linear Pipeline:** The stages that constitute the pipeline are executed one by one in sequence (say, from left to right).
- **Non-linear Pipeline:** The stages may not execute in a linear sequence (say, a stage may execute more than once for a given data set).

A possible sequence: A, B, C, B, C, A, C, A

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY

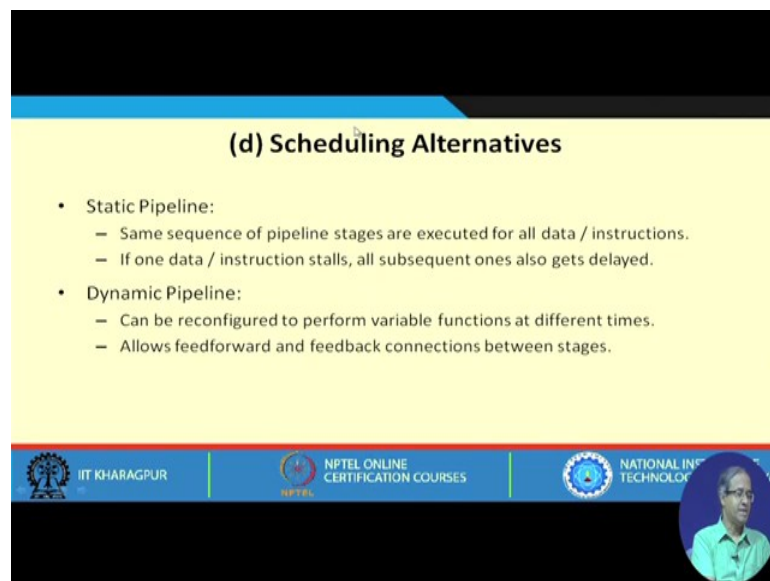
Next comes structure of the pipeline. We can have a linear pipeline that is most conventional. Linear pipeline means I have divided the computation into a number of stages that are supposed to be executed linearly one after the other. That means, there is some kind of a straight-line sequence; for every data input first A, then B, then C, and then you take out the result. This is the order of execution for every input data. But for a non-linear pipeline, see for very complex pipeline implementation you can have this kind

of non-linear pipelining. What non-linear pipeline says is that the stages do not execute in this kind of a straight-line or linear sequence. You see this is a pipeline again with 3 stages A B C, of course I have not shown the latches.

The arrows indicate the connection between stages. Like you can see from A there is a connection to B, from A you can also move to C, and also you can take a result out from B, you can go to C, from C either you can take a result out or you can move it back to B, or we can move it back to A, and new data is coming always to A. Here in this example a possible sequence can be A B C B C A C A. You can have some other sequence also, say A B C B C.

This is so-called non-linear pipeline where the stages may not execute in a linear sequence. As this example shows a particular stage may execute more than once for a given data set.

(Refer Slide Time: 24:10)



**(d) Scheduling Alternatives**

- Static Pipeline:
  - Same sequence of pipeline stages are executed for all data / instructions.
  - If one data / instruction stalls, all subsequent ones also gets delayed.
- Dynamic Pipeline:
  - Can be reconfigured to perform variable functions at different times.
  - Allows feedforward and feedback connections between stages.

The slide footer contains logos for IIT KHARAGPUR, NPTEL ONLINE CERTIFICATION COURSES, and NATIONAL INSTITUTE OF TECHNOLOGY KHARAGPUR, along with a small circular portrait of a man in a blue shirt.

The last classification depends on how we are scheduling the pipeline; it can be either static or it can be dynamic. Static means you have the pipeline stages and in the previous example you have seen that either I can execute them linearly or I can execute them in some particular order by feedback, etc. Static says same sequence of pipeline stages are executed for all data sets, say you can have non-linear piping all right, but the sequence of stages you are executing will always remain the same, that will not change with time.

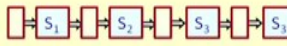
So, if one data or instruction stalls, I cannot feed the new data. Stall means due to some ongoing computation, I have to stop the pipeline temporarily. Here it says if I stall a particular data, then all the subsequent data sets also gets stalled.

In contrast we can have a dynamic pipeline where with time the pipeline can be configured, so that different sequence of pipeline stages can get executed. This allows feedforward and feedback connections as the previous example showed. This is a feed forward connection and this is a feedback connection. One example of a dynamic pipeline is one that is able to carry out both floating point addition and multiplication.





(Refer Slide Time: 26:27)

**Reservation Table**

- The *Reservation Table* is a data structure that represents the utilization pattern of successive stages in a synchronous pipeline.
  - Basically a space-time diagram of the pipeline that shows precedence relationships among pipeline stages.
    - X-axis shows the time steps
    - Y-axis shows the stages
  - Number of columns give evaluation time.
  - The reservation table for a 4-stage linear pipeline is shown.



	1	2	3	4
S <sub>1</sub>	X			
S <sub>2</sub>		X		
S <sub>3</sub>			X	
S <sub>4</sub>				X

When you are doing addition then certain sequence of stages will be executed, when you are doing multiplication some other sequence of stages will be executed. This is example of dynamic pipeline scheduling. Reservation table is a very commonly used data structure that represents the utilization pattern of successive stages. Let us take a simple example. Suppose I have a linear pipeline S1 S2 S3 S4. When a data comes, in the first cycle it goes to S1, second cycle it goes to S2, third cycle to S3, fourth cycle to S4.

You see the reservation table represents exactly that. On one side I show the stages other side I show the time steps. First time step we use S1, second time step we use S2, third time step S3, fourth time step S4. So, this is basically a space-time diagram that shows precedence relationship among the pipeline stages. The x-axis shows the time steps and

y-axis shows the stage. Number of columns indicates the total time required by the pipeline to evaluate the result. Let us look at a more complex reservation table.

(Refer Slide Time: 27:51)

• Reservation table for a 3-stage dynamic multi-function pipeline is shown.

- Contains feedforward and feedback connections.
- Two functions X and Y.

• Some characteristics:

- *Multiple X's in a row* :: repeated use of the same stage in different cycles.
- *Contiguous X's in a row* :: extended use of a stage over more than one cycles.
- *Multiple X's in a column* :: multiple stages are used in parallel during a clock cycle.

	1	2	3	4	5	6	7	8
S <sub>1</sub>	X					X		X
S <sub>2</sub>		X		X				
S <sub>3</sub>			X		X		X	

	1	2	3	4	5	6
S <sub>1</sub>	Y				Y	
S <sub>2</sub>			Y			
S <sub>3</sub>		Y		Y		Y

Here we look at that non-linear pipeline example we showed earlier.

For the same pipeline example I am showing two possible reservation tables. One computation X is here other computation Y is here. So, what it shows? It says that for computation X we need 8 steps: first time step S<sub>1</sub>, second time step S<sub>2</sub>, third time step S<sub>3</sub>, fourth time step again S<sub>2</sub>; that means, from S<sub>3</sub> again you go back to S<sub>2</sub>, fifth time step again S<sub>3</sub>, sixth time step S<sub>1</sub>. So, from S<sub>3</sub> you go back to S<sub>1</sub>, then again S<sub>3</sub>. You follow this path to S<sub>3</sub> then again S<sub>1</sub> again and you are finished. The Y computation is like this: first time step S<sub>1</sub> then S<sub>3</sub>. So, you follow this path straight away then S<sub>2</sub>, then S<sub>3</sub>, then S<sub>1</sub>, S<sub>3</sub> again S<sub>1</sub> again, from here you come out.

So, X comes out of here and Y result comes out of here. For the reservation table although all of these are not being shown in these examples, the first one is shown. Multiple cross marks in a row means that for this computation this stage S<sub>1</sub> will be used in time cycles 1, 6 and 8; stage S<sub>2</sub> will be used in time cycles 2 and 4; S<sub>3</sub> will be used in 3, 5 and 7. So, multiple X's in a row means repeated use of the same stage in different time cycles. Sometimes you may need a stage for an elongated period of time.

You can have two consecutive X's side by side, which will mean extended use of a stage over more than one cycle. Sometimes it may be required you may need for a computation stages required for two cycles. So, there will be two X's side by side. It is not shown in this example, you can also have multiple check marks in a column. That means, in a particular time step you can have a check mark here as well as here, which means both S1 and S2 are working together on the same data set.

(Refer Slide Time: 30:58)

**Speedup and Efficiency**

Some notations:

- $\tau$  :: clock period of the pipeline
- $t_i$  :: time delay of the circuitry in stage  $S_i$
- $d_l$  :: delay of a latch

Maximum stage delay       $\tau_m = \max \{t_i\}$

Thus,                               $\tau = \tau_m + d_l$

Pipeline frequency         $f = 1 / \tau$

— If one result is expected to come out of the pipeline every clock cycle,  $f$  will represent the maximum throughput of the pipeline.

IIT KHARAGPUR     
 NPTEL ONLINE CERTIFICATION COURSES     
 NATIONAL INSTITUTE OF TECHNOLOGY

So, multiple stages are used in parallel during a clock cycle. In general you can have all these characteristics in a reservation table. Let us make a quick calculation how we can compute the speed up and efficiency of a pipeline. We define some notations; tau is the clock period of the pipeline,  $t_i$  denotes the time delay of the stage  $i$  circuitry, and  $d_l$  denote the delay of a latch. There are  $k$  numbers of stages  $S_1$   $S_2$  to  $S_k$ . So, the delays will be  $t_1$   $t_2$  to  $t_k$ . The maximum stage delay I am denoting as  $\tau_m$  which is max of  $t_i$ . The minimum clock period should be satisfying this criteria, it must not be less than  $\tau_m + d_l$ ; that means, the maximum stage delay plus the delay of a latch. The pipeline frequency  $f$  would be the reciprocal of tau.

$f$  will represent the maximum throughput of the pipeline. That means, in this rate the output results will be coming out of the pipeline. If you have a linear pipeline you are expecting one result to come out every cycle, but for a non-linear pipeline things will

become more complex as we will see later, you may not be generating a result every clock cycle.

(Refer Slide Time: 32:41)

The slide contains the following text and formulas:

- The total time to process  $N$  data sets is given by  
$$T_k = [(k-1) + N].\tau$$

$(k-1)\tau$  time required to fill the pipeline  
1 result every  $\tau$  time after that  $\rightarrow$  total  $N.\tau$
- For an equivalent non-pipelined processor (i.e. one stage), the total time is  
$$T_1 = N.k.\tau$$

(ignoring the latch overheads)
- Speedup of the  $k$ -stage pipeline over the equivalent non-pipelined processor:  
$$S_k = \frac{T_1}{T_k} = \frac{N.k.\tau}{k.\tau + (N-1).\tau} = \frac{N.k}{k + (N-1)}$$

As  $N \rightarrow \infty$ ,  $S_k \rightarrow k$

The slide footer includes logos for IIT KHARAGPUR, NPTEL ONLINE CERTIFICATION COURSES, and NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA.

The total time to process  $N$  data sets is given by this expression.

Now if you have a equivalent non pipelined processor where we do not have pipelining, the total time will be  $N \times k \times \tau$ .

Here of course I made an assumption that all stage delays are equal and the latch overheads are ignored, but still this will help us to make a very fair comparison. Let us see how much improvement we are getting by pipelining. As  $N$  becomes very large this speedup tends to  $k$ .

(Refer Slide Time: 34:57)

• Pipeline efficiency:  
– How close is the performance to its ideal value?

$$E_k = \frac{S_k}{k} = \frac{N}{k + (N - 1)}$$

• Pipeline throughput:  
– Number of operations completed per unit time.

$$H_k = \frac{N}{T_k} = \frac{N}{[k + (N - 1)] \cdot \tau}$$

The slide also features logos for IIT KHARAGPUR, NPTEL ONLINE CERTIFICATION COURSES, and NATIONAL INSTITUTE OF TECHNOLOGY KHARAGPUR, along with a small video inset of a speaker.

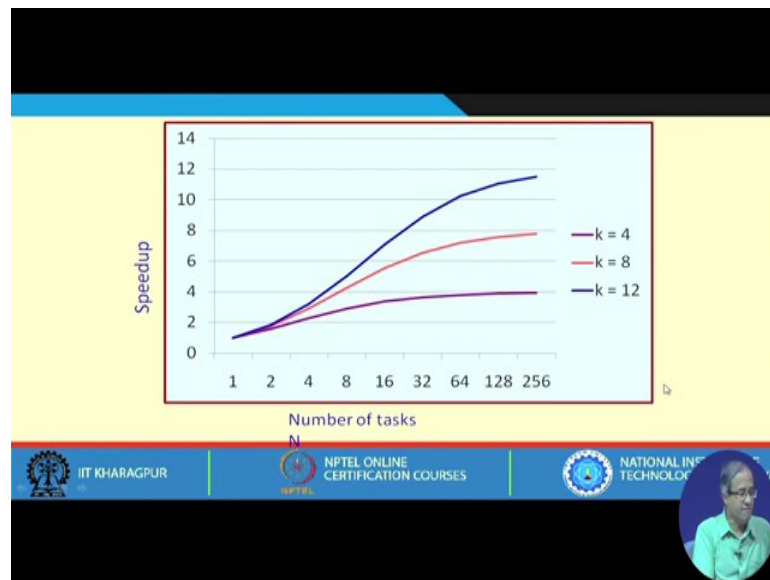
This simple expression tells you that we can achieve a pipeline speedup of maximum  $k$  as  $n$  tends to infinity; and we can define a term called pipeline efficiency which is defined as how close the actual pipeline performance is towards ideal value. You see the actual speed up is  $S_k$  just as we had expressed, and the ideal speed up is  $k$ .

So, I can divide  $S_k$  by  $k$ . How close this fraction is to 1; if it is 1 which means I have got the ideal value maximum efficiency. But in practice the denominator is greater than the numerator. So, efficiency is a little less than 1.

The pipeline throughput can be defined as number of operations completed per unit time.



(Refer Slide Time: 35:59)



This is a graph I am showing. As you vary the number of tasks  $n$  the speedup also increases. But for a 4-stage pipelining it levels up to 4; it cannot cross 4. For 8 stages it levels to 8, for 12 stages it approaches 12. So, it can never cross  $k$ .

(Refer Slide Time: 36:42)

### Clock Skew / Jitter / Setup time

- The minimum clock period of the pipeline must satisfy the inequality:  
$$\tau \geq t_{\text{skew+jitter}} + t_{\text{logic+setup}}$$
- Definitions:
  - Skew:** Maximum delay difference between the arrival of clock signals at the stage latches.
  - Jitter:** Maximum delay difference between the arrival of clock signal at the same latch.
  - Logic delay:** Maximum delay of the slowest stage in the pipeline.
  - Setup time:** Minimum time a signal needs to be stable at the input of a latch before it can be captured.

Lastly I talk about the clock period. Actually the clock period depends on few other things. The minimum clock period must satisfy an expression like this.

When the clock signal is generated it goes to all points in the pipeline latches. Because of unequal distance of the wires the clocks may not reach all the latches at the same time,

and there will be small time difference. This is called clock skew. Clock jitter means because of noise is on the neighboring lines due to capacitive affects, the delay of a signal line can vary slightly from one time to other; maybe for the same latch the period of the first one was  $\tau$  for the next one it is a little less than  $\tau$ , for the next on it will be little greater than  $\tau$ , there can small variations this is called jitter.

So, skew is the maximum delay difference between the arrival of clock signal at the stage latches, and jitter denotes maximum delay difference between the arrival of clock signal at the same latch. Designers always try to lay out the clock signal in such a way that skew and jitter is minimized, but still there will be a worst case value.

This is the more accurate expression that gives you some lower bound of the clock period that can be used.

With this we come to the end of this lecture. If you recall what we saw in this lecture, we have basically tried to convince you that pipelining is a concept where you can have significant speedup without making a significant investment. In our next lectures we will see that there are some complexities in the pipeline; for example, for non-linear pipeline you may be using the same stage more than once for the same data. So, you are not allowed to feed the input data at every clock; if you do it, there can be some conflicts or clashes in some stages.

Something called pipeline scheduling to decide when I need to feed my next data is very important. We shall be looking into pipeline scheduling aspects and then we shall be seeing how some of the arithmetic operations in particular the floating point operations can be implemented in a pipeline.

Thank you.