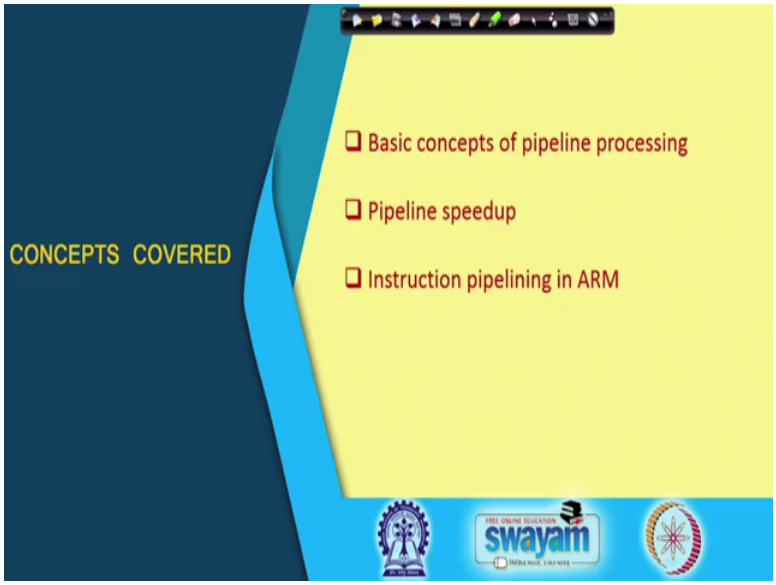
Embedded System Design with ARM
Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 05
Architecture of ARM Microcontroller (Part II)

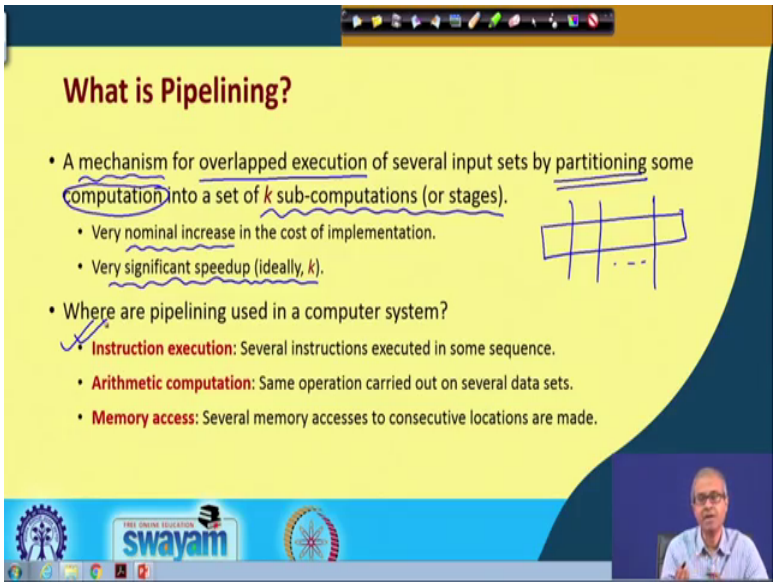
So, we continue the discussion on the architecture of ARM controllers, ARM architectures, ARM microcontrollers. So, this is the part 2 of a discussion architecture of ARM microcontroller.

(Refer Slide Time: 00:32)



Now, in this lecture, we shall be mainly talking about the pipeline features that are present in the ARM processor. Now, because the concept of pipelining may be new to some of you, so what I shall be doing I shall be diverting more time in explaining the basic concept of pipeline; subsequent to feature I shall very briefly tell how the pipeline in the ARM processor look like, and what is expected to be gained out of it, why do use pipeline, what are the advantages that you have out of it ok, these are the concepts that will be covered.

(Refer Slide Time: 01:16)

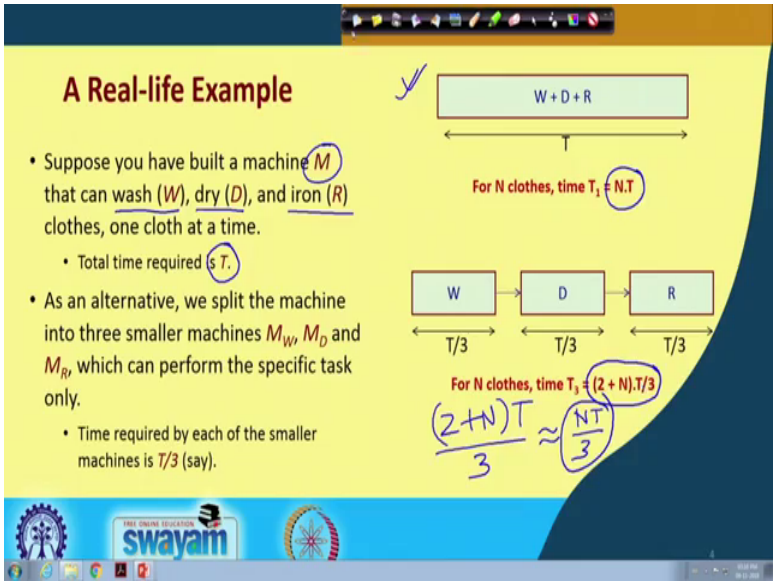


Now, let us start with the basic question what is pipelining, what is pipelining? Basically pipelining is a mechanism for overlapped execution. Overlapped execution means well we do not necessarily associate pipelining only for instruction execution for other cases also as we see. When you say that we means such we are caring out certain task, normally what we do, we finish a task complete it and then start with the next task. Now, in pipelining concept the thing is that even before you finish the first task, we start with the second task, even before we finish the second task we start the third task. So, different parts of the tasks can be carried out in parallel in an over lapped fashion. This is a basic concept behind pipelining right.

Now, the basic idea behind this is that we partition the computation that is being carried out into k number of sub computations or stages. Let the total computation that was there, we try to divide them up into several pieces of sub computations. If there are k number of sub computations we can divide into, then it is possible to have very significant speed up we shall see maximum up to k. But the very interesting thing is that we are not increasing the cost by a factor of k. Well, we can have k number of processes naturally we will be getting k times speed up. So, we are not doing that by very nominal increase in cost we are achieving close to k speed up. So, the increase in cost is very nominal right, this is the main feature of pipelining.

Now, in the present context we are talking about instruction execution, but pipelining can be used very efficient and effectively in other domains of computer processing also, during computation or arithmetic operations, during memory access of a vector of data. I want to load a vector of 64 words from memory one by one, there also the concept of the pipeline can help a lot. But in the present context, since instruction execution is the only thing you are concerned about. So, we shall not be going to too much detail about the other ones.

(Refer Slide Time: 04:18)



Let us understand first the concept. So, we take we take a real life example, which has nothing to do with computers. Let us assume that we want to wash some cloths think of a problem like this. Suppose I have built a machine M which can do three things one by one, wash, dry and iron. So, I give the machine of cloth, the cloth will wash it, dry it, iron it, and output that cloth in the iron form. So, let me assume that the total time required for the being the whole thing is T.

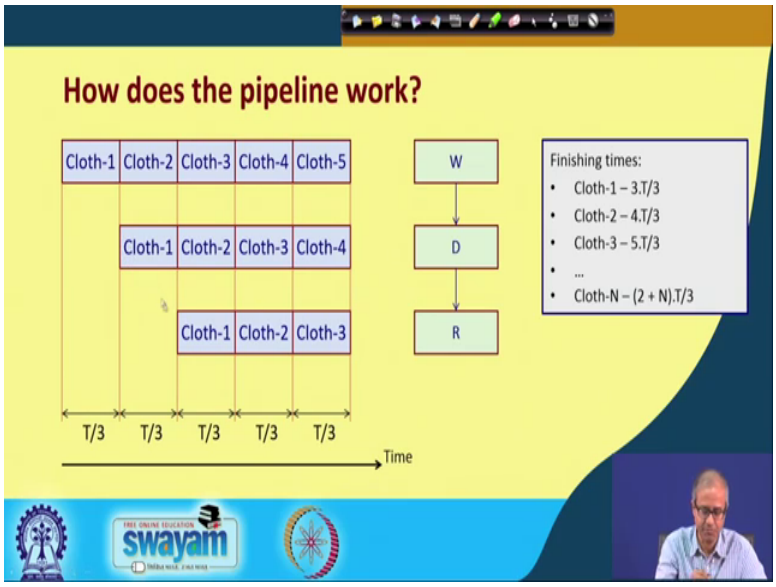
So, pictorially I show it like this. So, I have my machine which does washing, drying and ironing let us say R, the total time taken is T. So, if I have a N number of clothes, then the total time required will be N multiplied by T right.

Now let us assume that instead of a single very complex machine which can do everything let me divide this machine into three smaller pieces. Three is simple machines one which can only wash, one which can only dry, and one which can only iron.

Naturally this will not be costing three times as compared to the original machine, this will be cheaper. So, let us make another assumption the total time early was T, let us say for each of these time is T by 3 T by 3 and T by 3. So, that the total time still remains T.

Now, it can be shown we shall be seeing in later in the next slide that here for processing N number of clothes the total time required will be only 2 plus N T by 3, 2 plus N multiplied by T by 3, say earlier it was N into t. So, if N is very large, then this 2 can be neglected. So, you can proximate into N T by 3. So, we see earlier the time is N T, now I have reduced the time to N T by 3. This three comes from the number of stages; I have divided into three stages. So, I have got a three time speed up, but I have not paid three times more right, I am using simpler machines this is the concept of pipeline, but let us see how this expression is coming.

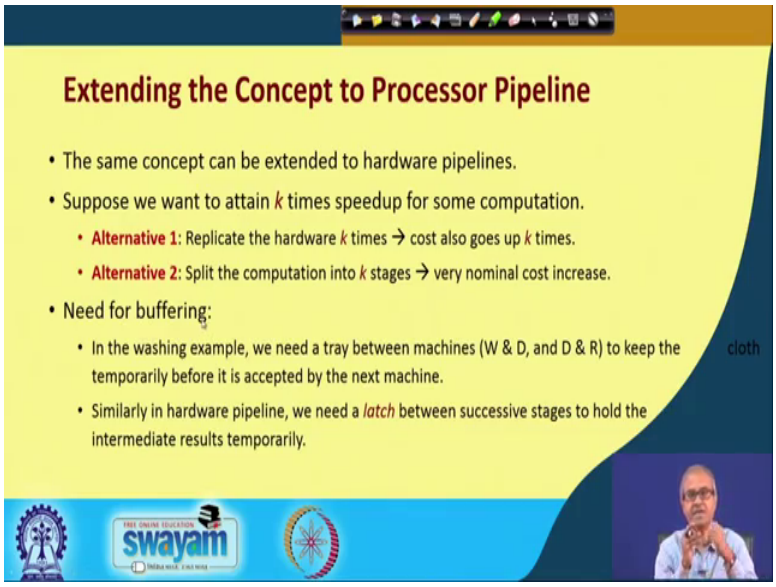
(Refer Slide Time: 07:11)



Let us look at this diagram in an animated form. So, washing, drying and ironing. First cloth comes for washing. So, this will be taking T by 3 times. So, after it is finished with washing, this cloth one can go for drying. And when cloth one has come for drying, the machine one is free again. Machine one can start with cloth 2; cloth 2 is being washed. Next step, cloth one is coming for ironing, cloth 2 is coming here, cloth 3 and so on, like this it will go on. So, you see after this machines are all filled up means every T by 3 time, here, here, here there will one cloth coming out earlier one cloth is coming out every time T. But now in this case every time T by 3 a cloth is coming out. So, I am

getting a three times speed up effectively right. This is the essential idea behind pipelining. So, a cloth N if you just calculate is a initial two time is taken for the pipe to fill up, and after that one T by 3 for every cloth for N clothes N into T by 3 like this it comes.

(Refer Slide Time: 08:44)



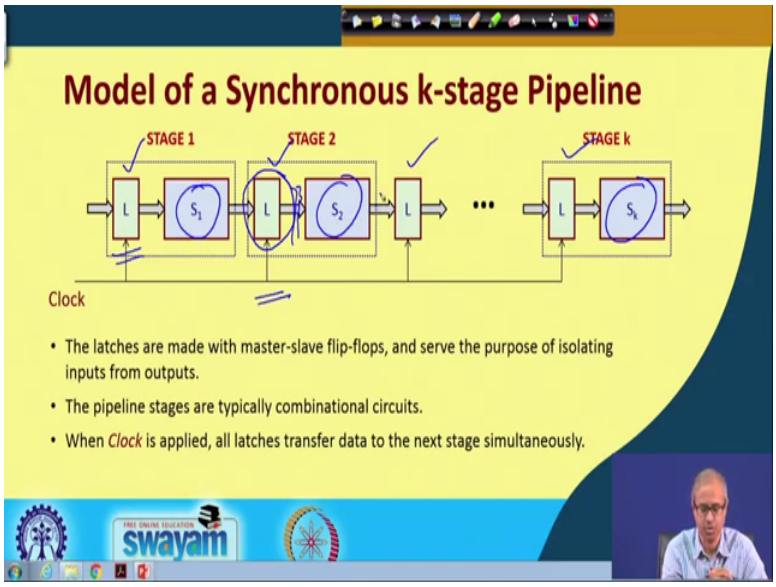
Now, extending the concept to process a pipeline, we want to increase this speed of a processor now. Suppose, I have a CPU I want to make it faster; I want to make it k times faster. So, I have two alternatives I can buy k copies of this CPU and run k instructions in parallel, so that is one way to have k times speed up, or I can divide the execution process in to k-stages of pipeline. So, instruction will be executing in a pipeline fashion just like that washing drying and ironing I mentioned. So, there also you can expect a k time's speed up.

So, alternative 1 is replicate the hardware k times, but if you do it, the cost will also go up k time, because you have to buy k copies of this CPU. Alternative 2 is split the computation into k-stages here you are not multiplying the hardware by k rather the you are splitting the hardware into k pieces very nominal increase in cost ok.

But in order do this, you need some buffering let say actually you think of the earlier example of clothing. When the first clothes your washing is finished, you want to give it for drying. So, you must keep it somewhere in between, so that you can accept the next cloth for washing. So, there must be a buffer or a tray between the machines, these are

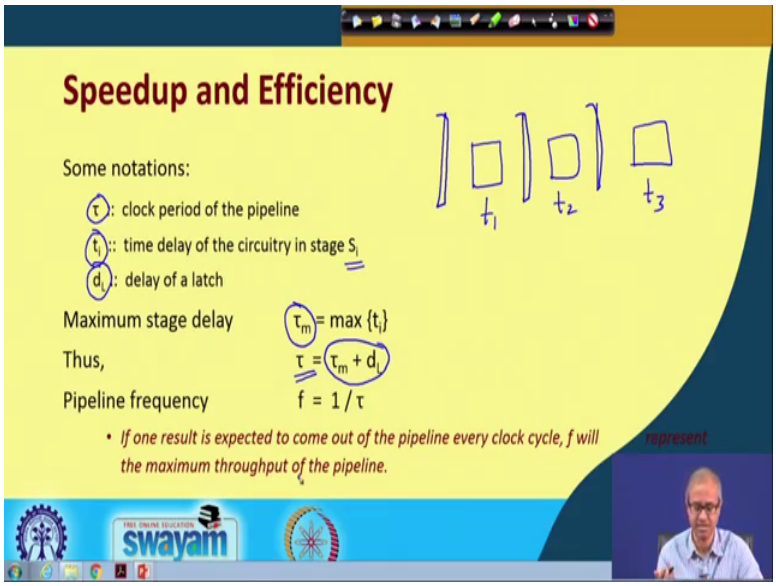
something called buffering requirements. So, for a instruction pipeline also we need some registers or latches in between the stages which will be temporary storing the data, the result of the previous stage which will be used by the next stage for processing. Well, if you do not use this registers, then the previous stage can go and modifying this value, so the next stage might carry you are some wrong computation, wrong calculation because of that.

(Refer Slide Time: 11:05)



Now, k-stage pipeline in general looks like this. There are k numbers of stages S 1, S 2 up to S k with a latch or registers in between preceding every stage. So, whenever a stage completes its calculation, it will store the data into this latch, and it will take the data from the next competition into its input latch. Because if this latch was not there, then while the next calculation is going on this input will go on changing right. So, this calculation can become a wrong ok. This is how it works.

(Refer Slide Time: 11:51)

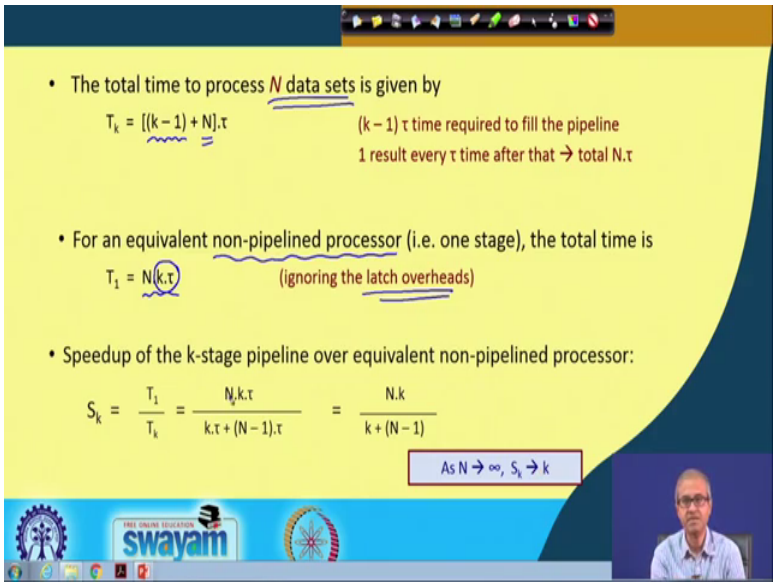


Now, let us carry out a quick calculation of the speed of an efficiency of a pipeline in the general sense. So, earlier we showed that calculation for that washing example. Let us say let us consider tau is the clock period of the pipeline, that means, every tau time data moves from one stage to the other, and new data was comes tau, tau, tau like that. And t i is the time delay for the circuits that are there in stage S i. And I told you there are some latches in between right. This latches delay will be dL, dL will be the delay of latch that's it.

So, whenever you have the pipelines like this, these are the pipeline stages and there are some latches in between these are the latches. So, what will be the maximum stage delay, see this is t 1, this is t 2, this is t 3. So, this slowest stage in the pipeline will determine that what is the maximum speed in which we can shift the data, because this slowest stage will be become the bottle neck. So, maximum of t i, let us call it tau m the maximum delay. And to it we have to also add the latch delay dL. So, tau m plus dL that is what is your tau is that will be your clock period right that is how you calculate the clock period.

Now, the pipeline frequency will be 1 by that tau. Now, f will also be the maximum throughput of the pipeline if you are expecting one result to come out every clock that is what I said earlier [FL] ok. Now, with this assumption let us try to make a quick in calculation.

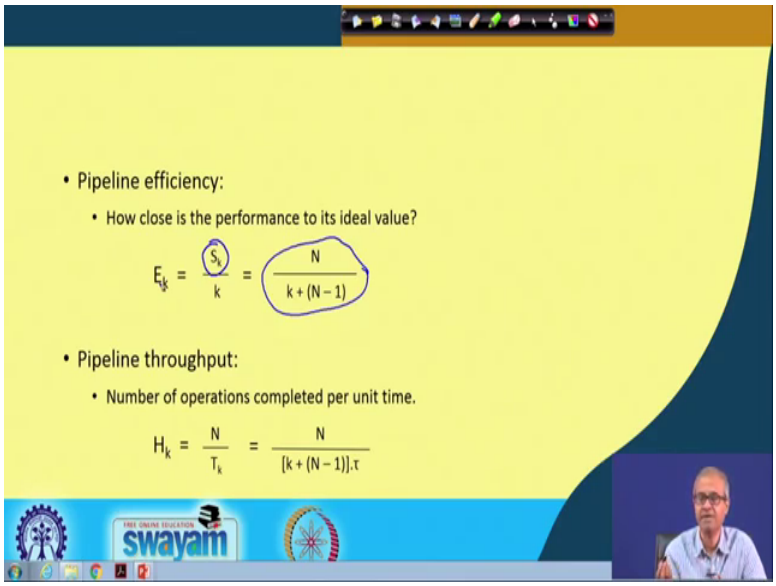
(Refer Slide Time: 14:02)



The total time to process N sets of data. How do you calculate? Tau is my clock with every tau a new data is coming. Now, k minus 1 clocks are required to fill up the pipeline. There are k-stages, so I need k minus 1 clocks to reach a stage where all the k-stages are working on something. After that every tau time there will be one new result being generated tau, tau, tau right. So, k minus 1 into tau will be your initial time for the pipe to fill up, and then this N into tau for the output to be generated, N number of data to be calculated on data sets, so N number of outputs so this whole thing into tau. This will be the total time to process in data sets.

Now, if we have an equivalent non-pipelined processor, well if you ignore the latch delays for the time being, they are small, then the total time can be estimated as capital N into k into tau. Let us say k into tau will be the total computation time let us say for the non pipeline processor. Capital N is the number of data sets, capital N multiplied by that this will be the data time. So, in a pipeline how much speed up we are getting, T 1 divided by T k. T 1 was the original time and T k is the time with pipeline. So, you have this. So, you this in tau gets canceled out, you get this expression. Now, as N becomes very large if you divide by N k by k by n. So, it will become 1 as k N tends to infinity this S k approaches k. So, for a large number of data that you are processing the pipeline, your speed up will be close to number of stages k. This is an important result.

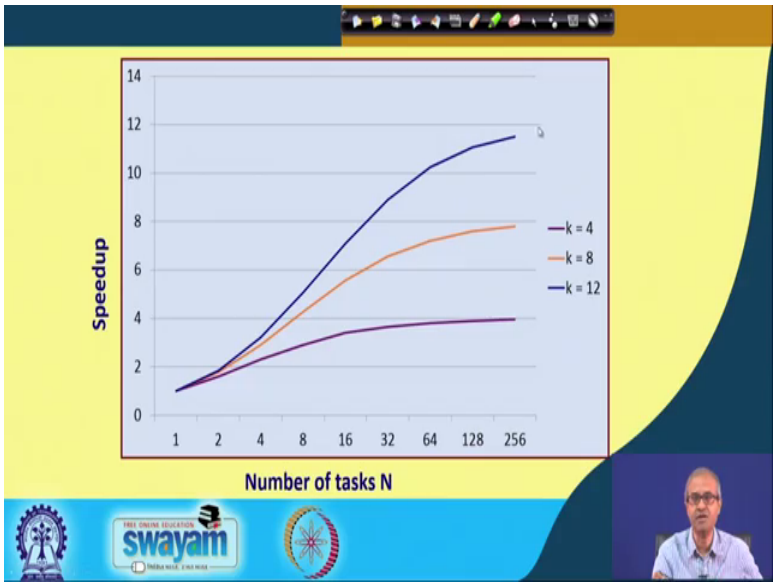
(Refer Slide Time: 16:21)



Now, this another term we define called pipeline efficiency that how much is the performance close to the ideal value. Well, this S k we just calculated this is the pipeline speed up right pipeline speed up is N k divide by this. And divide by k is the ideal speed up. I told that the maximum speed up can be k as N tends to infinity the speed up will tend to k, so that is when the pipeline is operated at maximum efficiency. If I divided by that, k, k can cancels out, so I get a factor. This I can define as the actual pipeline efficiency. So, it will never be 100 percent maybe is work in 90 percent efficiency or so on I can say that.

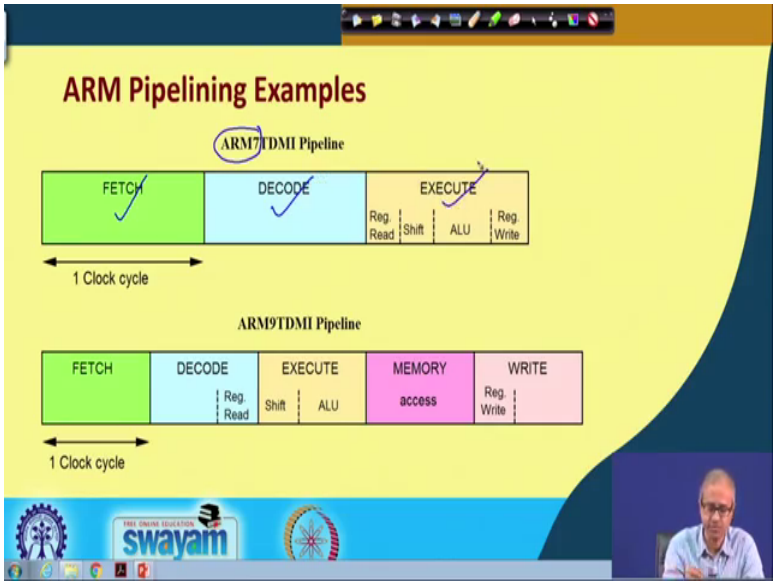
And another term which is of course, not that important the present context pipeline throughput number of operations completed per unit time. Total number of operation is N. And the time taken is T k. If you divide it, you get an expression, this is pipeline throughput. So, this is how you can make some simple calculations for a pipeline ok.

(Refer Slide Time: 17:48)



And this is a very typical plot I am showing number of task N versus speed up for some various values of k. Let us say k equal to 4 you see as the number of task increases, the speed up increases increase it levels to very close to 4. For k equal to 8, it levels to very close to 8; 12 it levels close to 12. So, it will never reach to N, it tends to 12 ok, it will never touch 12, but it will get closer and closer to 12 as the value of N increases. Here I have shown up to 256 right. So, here you get some idea what is actually happening ok.

(Refer Slide Time: 18:35)

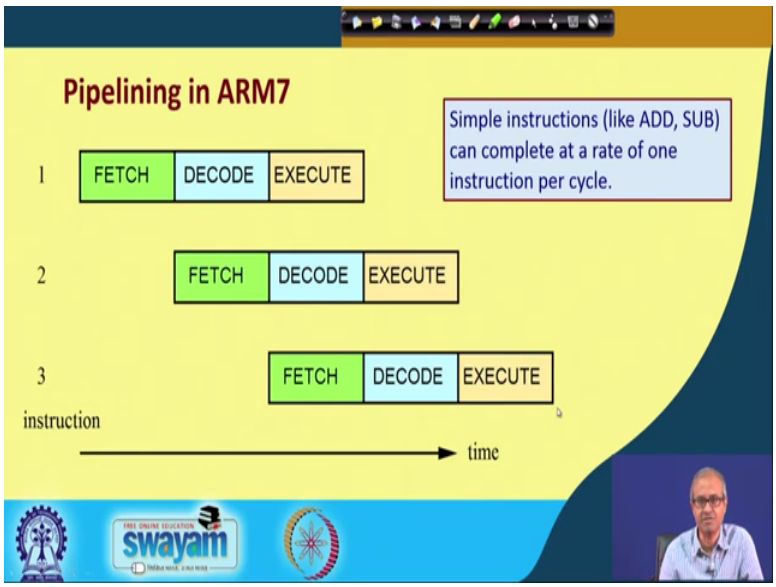


Now, coming to ARM I am not going into the details now because this is not a course where I am teaching computer architecture rather I am trying to tell you that ARM uses instruction pipelining and why pipelining is useful that I try to explain. If I have a k-stage pipeline, I can or I can expect to have k times speed up.

So, in your ARM seven architecture this is one particular processor TDMI pipeline. There are three stages, fetch, decode and execute. For each for these were executed in one clock cycle that is tau you are talking about. So, if everything else is fine, we are expected to get about three time speed up in terms of instruction execution. Similarly ARM 9 has a five stage pipeline, fetch, decode, execute, memory, write. So, we can see something smaller things.

So, within the decode, the register values are also read whatever registers are required. During execute the shifting, the barrel shifter is also working ALU operations also carried out, memory access are carried out here; during write, the results written back into the register bank, so it is done here. So, this captions actually indicate that. And for ARM 7 everything was done during execute. The register read, shift, ALU, register write everything was done in execute. But in ARM 7 ARM 9, you are making it more elaborate and more flexible. So, here this speedup can be maximum 5, so if I improve on that.

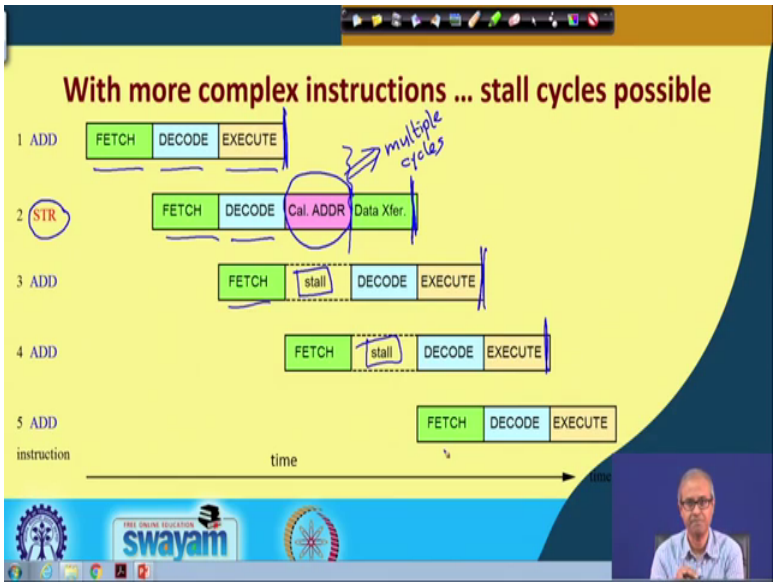
(Refer Slide Time: 20:33)



And in ARM 7 just an example just the way I said that wash, dry and iron your fetch decode execute. So, first instruction is doing fetch, decode execute, second instruction

can start fetch here while the first one is being decoded. Third instruction can start fetch while the first one is executive, second one is decoding. So, starting from the third clock, you will be finishing one instruction every clock cycle right. This is the main having this advantage and purpose of a pipeline.

(Refer Slide Time: 21:15)



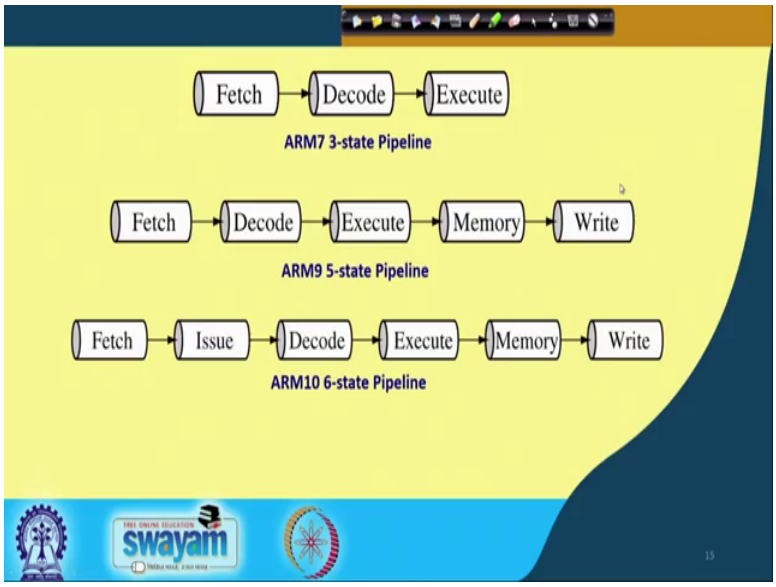
And just one thing I want to mention here this speedup of k that I am talking about that is an ideal speed up that is a speed up you can get when the pipeline is operating in its full speed, but sometimes you may see that due to some reason, you cannot operate the pipeline at full speed. So, in those cases, the speed up obliviously will become less much less than k. So, I am giving one example. Suppose, there are some instructions which are executed and let us say these are all add instructions and there is one complex multi register store instruction let us say I talked about multi register store load and store. So, multiple registers can be stored and loaded together. So, it will need multiple clock cycles.

So, the idea is that normally everything finishes in one clock, one clock, one clock one clock, this also proceeds. But for STR instruction what might happen that this pink colored box this all though I am showing it like this, but this can actually require multiple cycles. What does that mean? Multiple cycle means here you cannot decode this next instruction you see and this unless this execute is over you cannot decode it. So,

there will be some delay here because it is requiring multiple cycles. Such delays will referred to as stalls; we call them as stall cycles.

Stall cycle means wastage some cycles are wastage. You see here first instruction is finished here at this point; second instruction was finished here, but yeah third instruction here, fourth instruction here. But because of this delay this instruction was supposed to finish here, but it got delayed. Not only this, all subsequent instructions got delayed and there can be many such instructions like this in between. So, for every such instruction there will be some stall cycle inserted. And once a stall is there this stall will be carried by all subsequent instructions until that instruction exits the pipe right. So, such cases can slow down the maximum operational speed of a pipeline ok. Just remember this

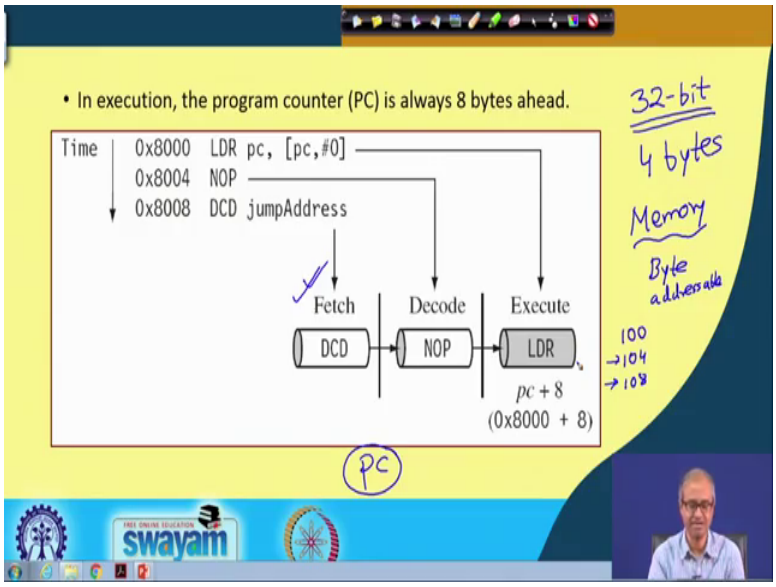
(Refer Slide Time: 24:07)



Just a bar side view of the ARM 7, ARM 9 and ARM 10 pipelines. ARM 3 I mentioned fetch, decode, execute; ARM 5 also I mentioned fetch, decode, execute, memory, write. In ARM 10, there is just this additional stage of issue which has been inserted between fetch and decode. This issue is a feature which is an advanced architectural feature, where you check whether to issue an instruction or not depending on various things. There are something called pipeline hazards. There can be data dependency whether you can feed the next instruction or not. There are lot of architectural issues which prevent the pipeline from operating at its maximum speed.

Stall instructions can occur due to one example I gave because of a complex instructions, but there can be other reasons there are situations called hazards data hazards, there can be structural hazards, there can be control hazards. Because of various sequence of operations that are being carried out, and some instructions like jumps and branches you may have to insert stall cycles. So, all of these prevent the pipeline from operating at the maximum clock frequency.

(Refer Slide Time: 25:39)



Just another thing let me tell you, in this ARM 7 kind of architecture a 3-stage pipeline let us say when an instruction reaches the execute phase. Now, one thing you remember I told you all instructions are 32-bit instructions, this we will shall see later also in some detail bit. 32-bit instruction means 4 bytes. And your memory that you have this memory is byte addressable. So, if the first instruction is stored in memory location 100 the next instruction will be stored in memory location 104, next location will be stored in location 108, because each instruction will be requiring four bytes or four memory locations right.

So, the point is that when some instruction is executed the program counter will always be 8 bytes a head. So, you add it because you will be fetching this. So, each instruction will be 4 plus 4. So, each instruction will be adding 4 to the memory address. And this program counter we shall see PC is a special register which always stores the address of the next instruction to be fetched right. So, when you are fetching this instruction, this will be the program counter of the current instruction plus 8, because current instruction

is here. If we add 4 to it, we will be getting the next instruction; if we add 8 to it, we will be the next to next instruction. Now, here you are always fetching the next to next instruction right because there are three stages that is why the program counter is always 8 bites a head you say like that ok.

Because when you are executing here already incremented this pc two times it is always 8 bytes ahead, so that when you are fetching you should accordingly adjust and fetch accordingly right that calculations to be done. So, with this we come to the end of this lecture. Now, in the next lecture, we shall be looking at some of the unique features of ARM with respect to the register organization, the various execution modes and so on. This we shall be discussing in the next lecture.

Thank you.