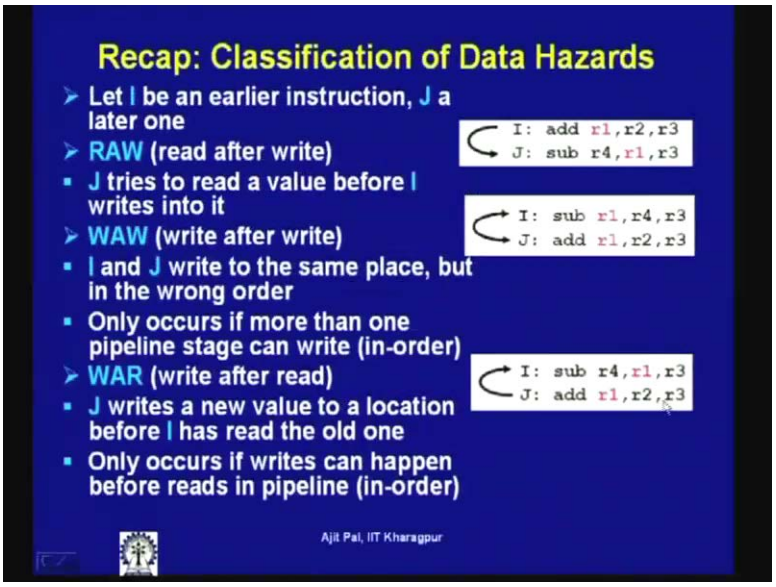
High Performance Computer Architecture
Prof. Ajit Pal
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 9
Data Hazards

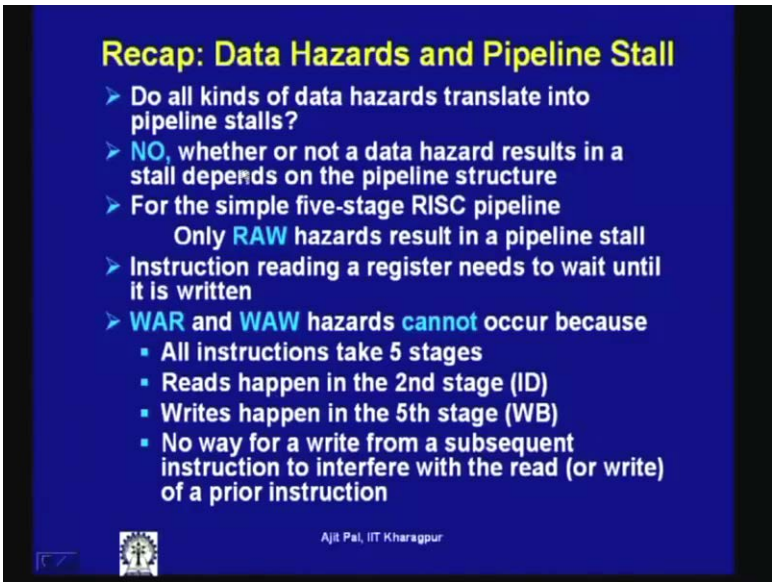
Hello and welcome to today's lecture on Data Hazards, we have started our discussion on various types of hazards that arise in pipeline implementation of processors.

(Refer Slide Time: 01:12)



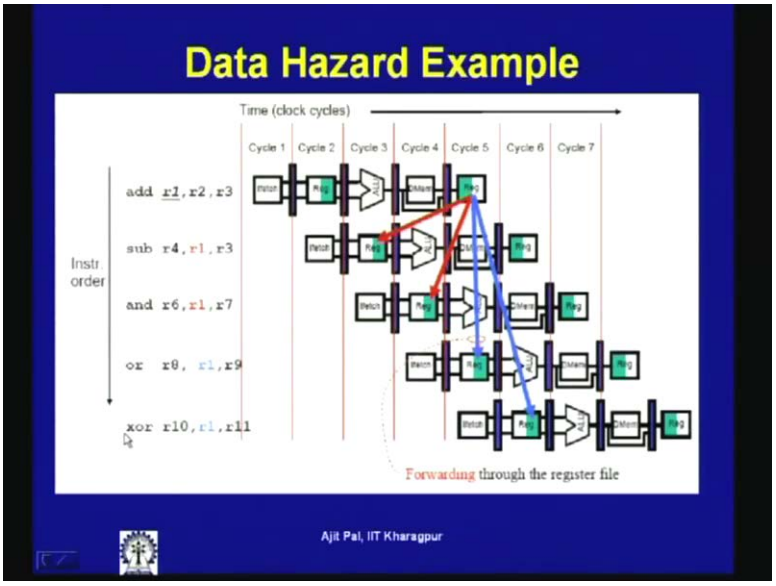
And you may recall that the data hazards is one of the 3 types of hazards, and in the last lecture we have discussed about the structural hazard. And we have seen how structural hazards can be overcome with the help of additional hardware. Now, coming to the data hazards as you have seen, the data hazards can be classified into 3 types; first one is read after write in where J tries to write a letter in instruction J tries to read a value before I writes into it, and the other two are write after write WAW type, and WAR write after read type.

(Refer Slide Time: 02:03)



However as you have seen for the pipeline that we have been discussing the simple 5 stage RISC pipeline. They are only read after write type of hazards may result in a pipeline stall, on the other hand write after read and write after write hazards cannot occur because of different reasons that we have discussed in the last lecture.

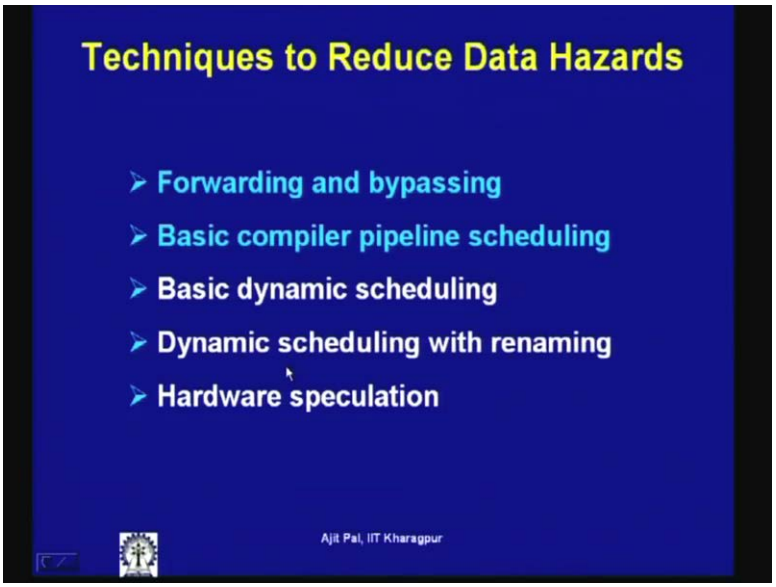
(Refer Slide Time: 02:31)



Now, before I mention about this let us consider what is outcome of this data hazard, here is a code sequence add r 1 comma r 2 comma r 3 followed by sub r 4 comma r 1 comma r 3, then and r 6 comma r 1 comma r 7 and so on. So, we find here you have got

5 instructions, and data hazard is arising because the second instruction is trying to read a data which has not been written into the register. And this is leading to a hazard, similarly the third instruction is also facing hazard because it is trying to read data before the writing has taken place into the register. However, the subsequent instructions will not face any hazard, because they are reading the data from the register, after writing has taken place.

(Refer Slide Time: 03:45)

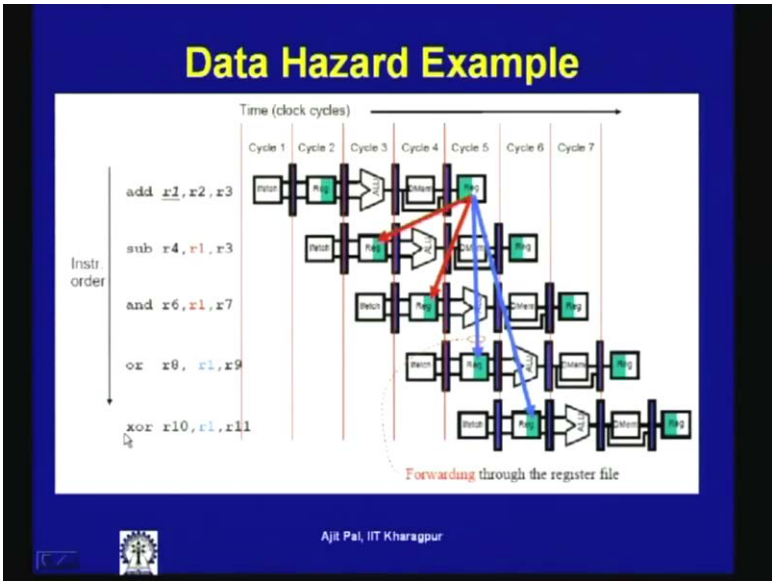


Now, this is the how can we overcome these hazards, there are several techniques by which this data hazards can be overcome. The first technique is known as forwarding and bypassing, forwarding and bypassing technique is based on hardware approach; that means, you have to add some additional hardware to overcome data hazards. And we shall see how it can be implemented, the other techniques are essentially software based first one is basic compiler pipeline scheduling.

And we shall see, how compiler can help in scheduling instructions such that the data hazard is eliminated or it is impact is reduced. Then this the scheduling that is being done by compiler is known as static scheduling, on the other hand this scheduling can be done with the help of hardware, which is known as dynamic scheduling which I shall also discuss.

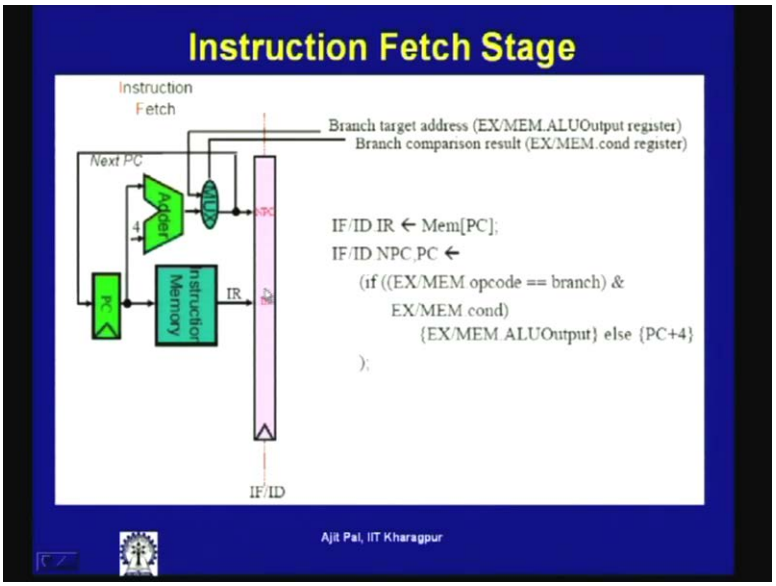
And this dynamic scheduling can be done without renaming, and with renaming and finally, we shall discuss how hardware speculation also help in reducing data hazards, so first let us focus on forwarding and bypassing which is a hardware based approach.

(Refer Slide Time: 03:45)



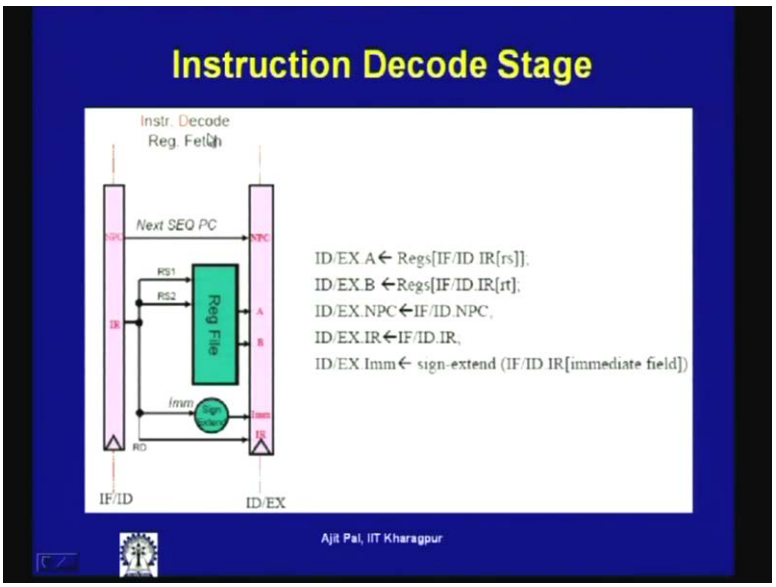
If we look at this particular pipeline execution of these instructions. We find that data hazard is arising in spite of the fact that results are already available in the pipeline registers. Although it has not had been data has not had been written into the register that register where it has to be written that is r 1, but those values which was computed here in the third clock cycle are already available in different pipeline registers.

(Refer Slide Time: 05:38)



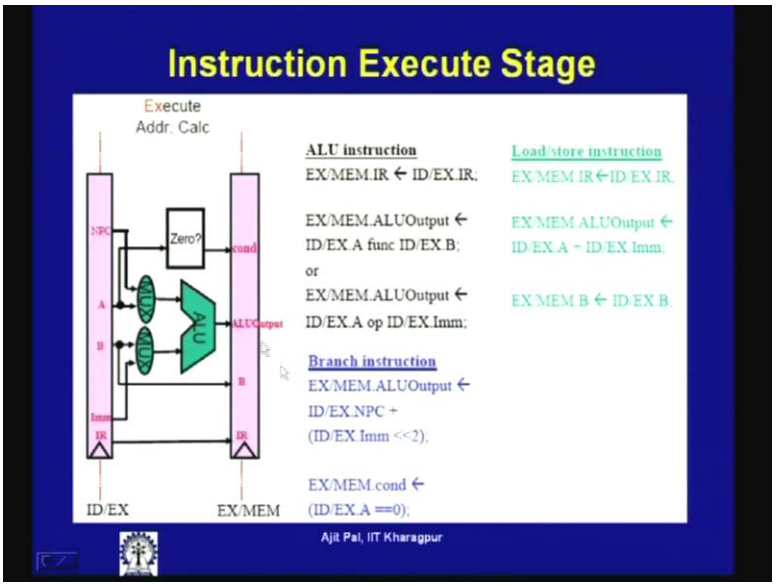
So, as you can see the pipeline registers which are holding different values after the execution in different stages. For example, after instruction phase means instruction is stored in this instruction register.

(Refer Slide Time: 05:56)



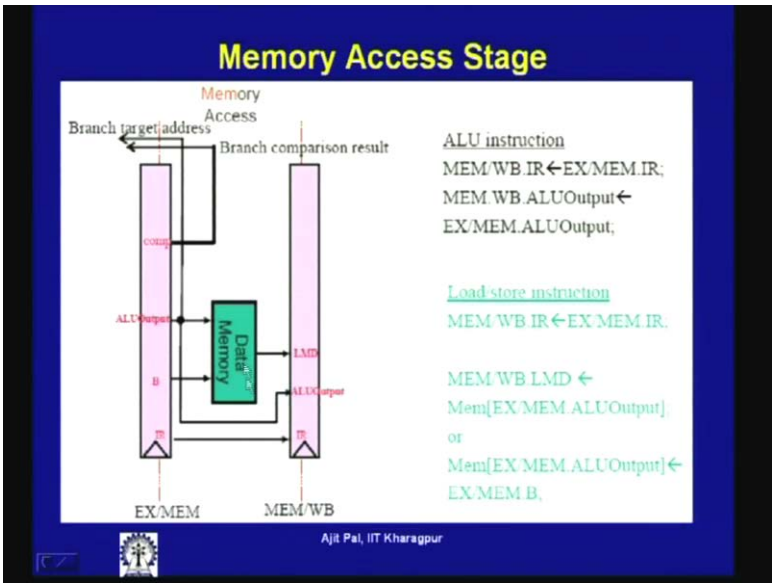
After the instruction decode phase, the data which is being read from the registers are again stored in the pipeline registers. So, various values which are generated in a particular stage are stored in those register, in these pipeline registers.

(Refer Slide Time: 06:13)



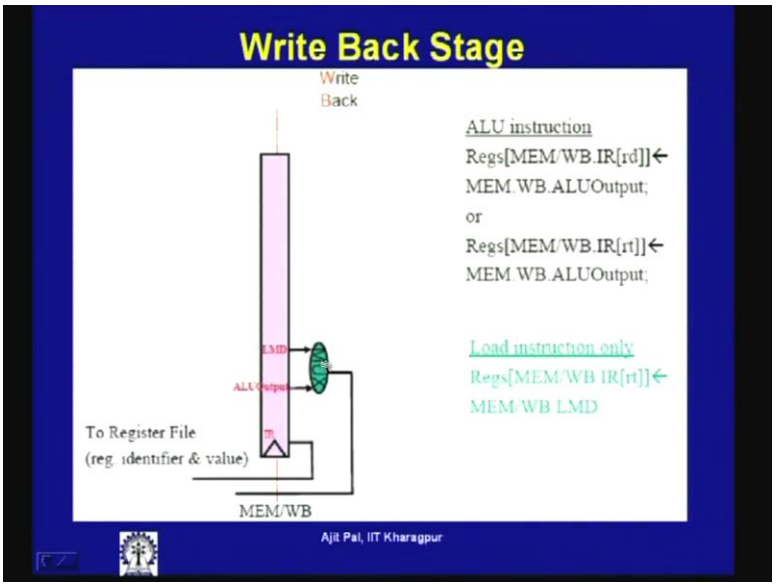
Similarly, after execution the results are available in these pipeline register.

(Refer Slide Time: 06:20)



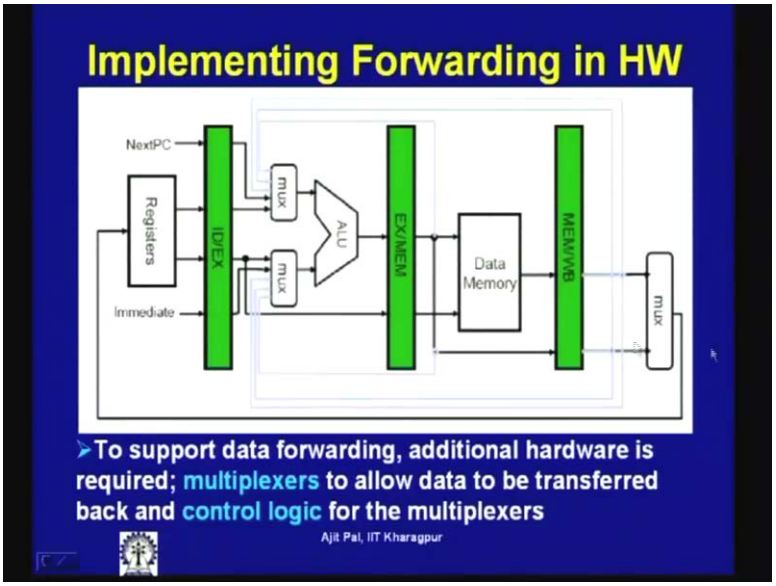
So, similarly after whenever memory access is being done the result is again stored in the pipeline registers.

(Refer Slide Time: 06:28)



And finally, when write back is taking place data is taken from the pipeline register, and which is being written to the proper register. So, we find that the key idea of forwarding comes from the fact that, data which is already available in pipeline registers we are not utilising it, we are trying to read it from the final register where it is stored at the end of fifth cycle. So, can we not read the operant's from pipeline registers instead of the I mean before it is written into the register of the ALU, that is your in our case the register is r 1.

(Refer Slide Time: 07:18)



So, how this can be done is shown here we shall require several multiplexers, as you can see ALU inputs are coming from 2 multiplexer. Earlier they were coming only from the this particular stage; that means, the pipeline stage; that means, the first pipeline register, so inputs were taken from there or from the immediate data that was taken. Now, from different pipeline registers, the outputs are applied here to the input of the marks and for you can see from this the ALU output is applied to the marks.

Similarly this ALU output is also applied to the other marks and similarly, from the data which is being read from the memory. They are also written into the they are applied to the input of the marks because we shall read from the pipeline register instead of reading from the registers. Similarly, finally, whenever it comes of course, it will go to the register, so the basic idea is that with the help of multiplexers having more than 2 inputs say 5 inputs to each of these multiplexers the inputs can be taken not only from the final register.

But, also from different pipeline stages because the intermediate results which are hold in the pipeline registers are available in them. And they can be taken from there and applied to the ALU, and that is how this hazard is overcome, so to support data forwarding additional hardware is required, first hardware is multiplexers to allow data to be transferred back. And second is the control logic, we know that the control logic is performing the control of different data path hardware that is present in the processor.

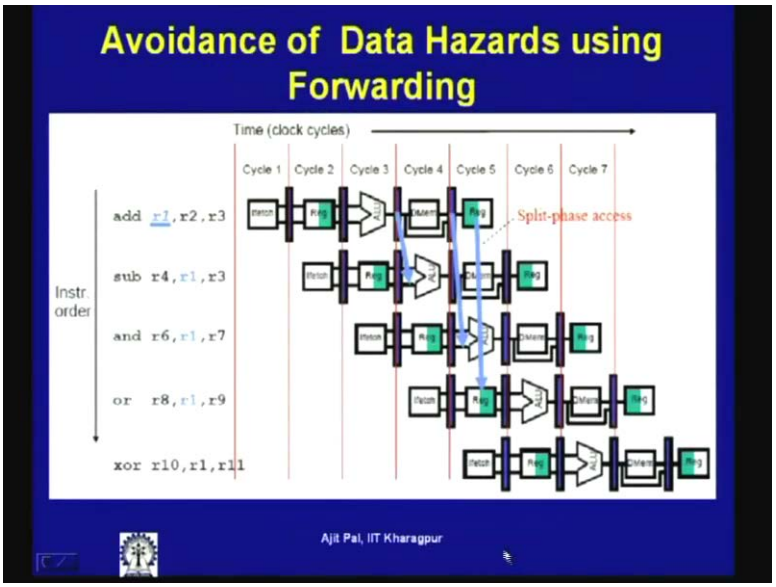
That means, it is controlling the different registers, it is controlling, the ALU the operation to be performed and so on. So, here whenever we add multiplexers with multiple inputs for example, this multiplexer is having 5 inputs, so you will require 3 control signals, which will select one of the 5 inputs and that will be applied to the ALU. Similarly, here also you have got another multiplexers, 5 inputs are coming from 5 different sources, and you have to select one of the five depending on the instruction that is being executed.

That means the depending on the off code, the controller will decide which inputs will be selected and that the data will go to the ALU. So, the control logic of the multiplexer will be also be has to be implemented, in other words I am trying to tell that the controller without pipelining will be simpler, than sorry controller with forwarding. That means, if forwarding is not done, then the controller will be simpler, but whenever forwarding is

implemented then the controller will be complex because it has to generate additional control signals.

And those control signals can be generated by analysing different values, different I mean instructions. That means, you have to see in what situations the operant's will come from different those pipeline registers, and accordingly by that accordingly the logic has to be implemented in controller. So, this implementation will be little more complex, so this is how forwarding is done, this is called bypassing. Because, we are bypassing reading from the final register by reading it from different pipeline registers, which are present where the intermediate values are temporarily stored this is forwarding.

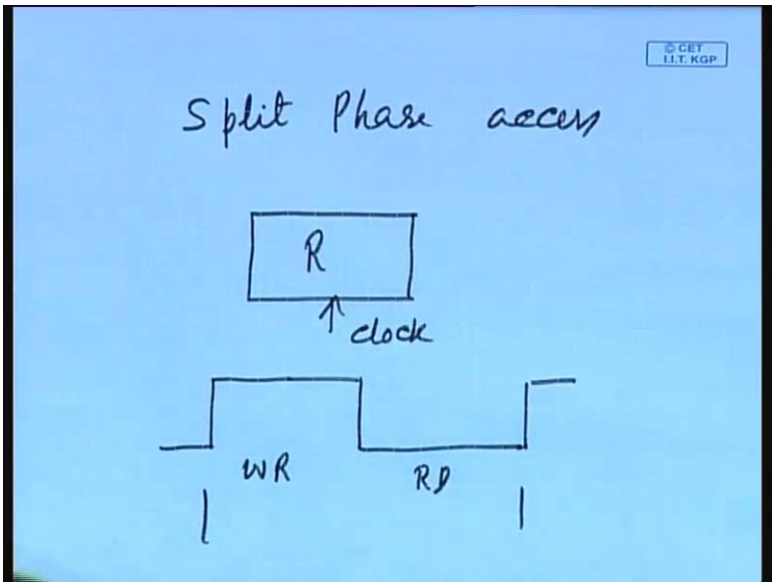
(Refer Slide Time: 11:42)



Now, whenever we do forwarding this particular diagram shows, how forwarding is helping in overcoming data hazards. And as you can see, earlier we were trying to read this value from this register, instead of reading from this register now, for the second instruction that the value will be coming from this particular pipeline register, instead of the register bank that is present in that ALU. So, it is reading from there, so since it is in the forward direction there is no hazard.

Similarly, we can see here also in the third instruction is also getting the upper hand from the pipeline register; that means, this r 1 is also read from the pipeline register, and that is being applied to the one arm of the ALU. And similarly of course, the third instruction that operant is being read from the register itself.

(Refer Slide Time: 13:06)

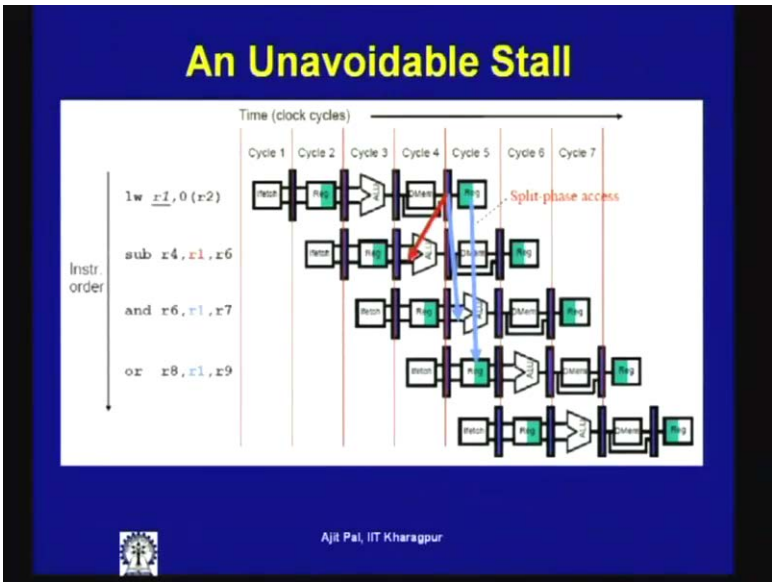


And of course, this is possible because we are using what is known as split phase access, so you have a register and that register what is being done, it is controlled by a clock there is a clock which is applied. And the clock is applied to control the operation of the register, now what is being done writing into the write operation is being performed in this the first cycle, first part of the cycle. So, and then read operation is done in the second phase, as a consequence after the writing takes place in the same phase, same clock during the same clock you can read it.

So, that is what is happening in this particular case; that means, writing of the data is taking place into the register, in the first part of the clock cycle. And in the second part of the clock cycle, which is shown by green shade, the reading operation is taking place from the same register. So, from the same register in two phases you are able to do two operations, writing as well as reading that is why it is called split phase access, and by if the split phase access is not allowed.

That means, if the only reading or writing can be taken place in a single clock cycle in such a case of course, again there will be a data hazard in this case. And in such a case you have to read it not from the register, but again you have to read it from the pipeline register. So, now the question arises this type of data hazard can be overcoming by forwarding or bypassing, can all possible data hazards be overcome by bypassing or forwarding the answer is no.

(Refer Slide Time: 15:01)

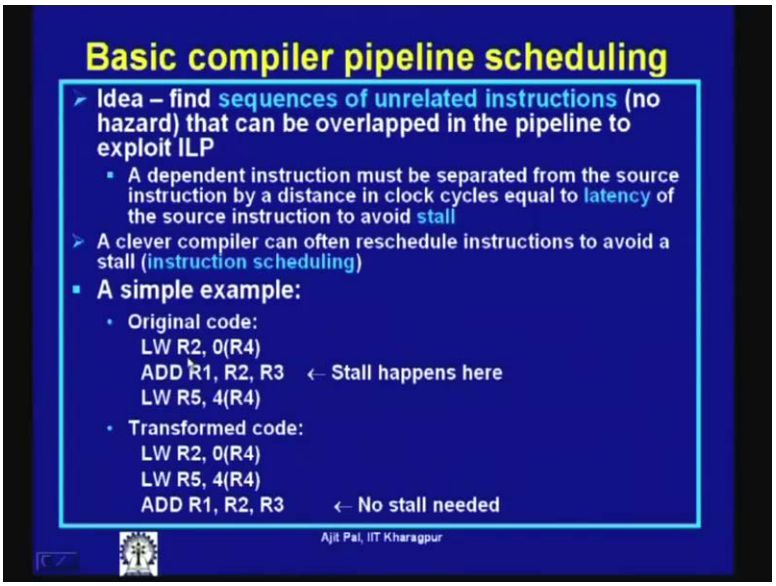


There are situations where I will see even by using that complicated bypassing hardware, the hazard cannot be overcome. So, in this particular case it involves a load operation, so you will be loading some value into a register r 1 and so you will require a memory access. So, as we have seen the in the fourth clock cycles the memory access is being done, so; that means, the value that is being read from the memory will be available only at the end of clock cycle 4.

On the other hand, the next instruction that sub r 4 comma r 1 comma r 6 is reading the value in the fourth clock I mean in the in the beginning of the fourth clock cycle. So, in the beginning of the fourth clock cycle it is trying to read the value, and at the end of the clock cycle data is available. So, in this particular situation there will be hazard and so this hazard cannot be overcome by bypassing, so you have to introduce a stall to overcome this situation.

However, this subsequent reading operant's will take place from the register, so this will lead to a stall which we call unavoidable stall. So, this types of stalls will be present, whenever you are reading an operant from a memory and that is being used in the next cycle; however, whenever it involves ALU operations, then this problem will not arise.

(Refer Slide Time: 16:46)



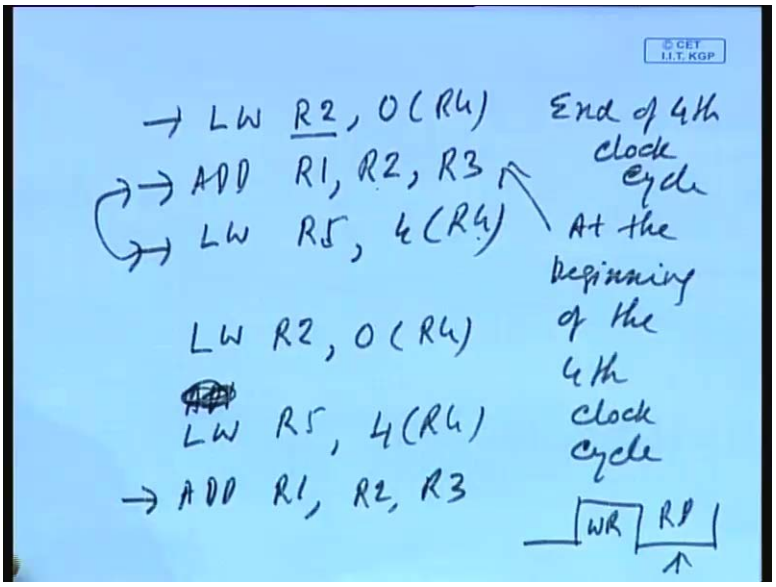
So, this is all about the data hazards, now we shall discuss about the use of compiler for overcoming data hazards. And basic idea is find sequences of unrelated instructions that can be overlapped in the pipeline to exploit ILP; that means, in case of our ideal implementation of pipeline, we assume that all the instructions are independent originally. But, subsequently we discover that there is data dependency and because of data dependence there is some kind of I mean the instructions are not really independent.

So, you have to maintain a sequence and you cannot arbitrarily change that position; however, there are instructions which are independent, and which can be identified by the compiler. And then they can be scheduled; that means, the original sequence can be modified by the compiler, and change their sequence in such a way that the data hazard will be overcome that is the basic idea. And, so a dependant instruction must be separated from the source instruction by a distance, in clock cycles equal to the latency of the source instruction to avoid stall.

So, this is the key idea behind this compiler based scheduling of instructions; that means, an instruction which is dependent on the previous instruction has to be separated, from the source instruction. How much separation has to be done to avoid data hazard, which is the latency of the pipeline that is being implemented; that is dependent on the pipeline that we have implemented. So, this is how we can avoid stall, let me illustrate this with the help of a simple example, so a clever compiler can often reschedule instructions to

avoid a stall. So, here you have got an instruction load I have already mentioned about this whenever it involves reading data from a memory.

(Refer Slide Time: 19:12)



That means, the sequence of instruction is load word R 2 comma O R 4, then you are performing additional operation, using involving the same register R 2 and R 3. So, you are adding the content of R 2 with the content of R 3 ensuring the result in R 1, and then another instruction is there load word R 5 and comma 4 R 4. So, here we can see that this particular reading of data, will take place at the end of fourth clock cycle, on the other hand here the reading will take place, at the beginning of the fourth clock cycle.

So, what we find that here there is latency of only one clock cycle, if this instruction can be separated from this by one instruction our job will be done. That means, the data hazard will be overcome, so this particular instruction is independent, if we consider the 3 instructions we find that, this instruction has no dependency on the previous two instructions.

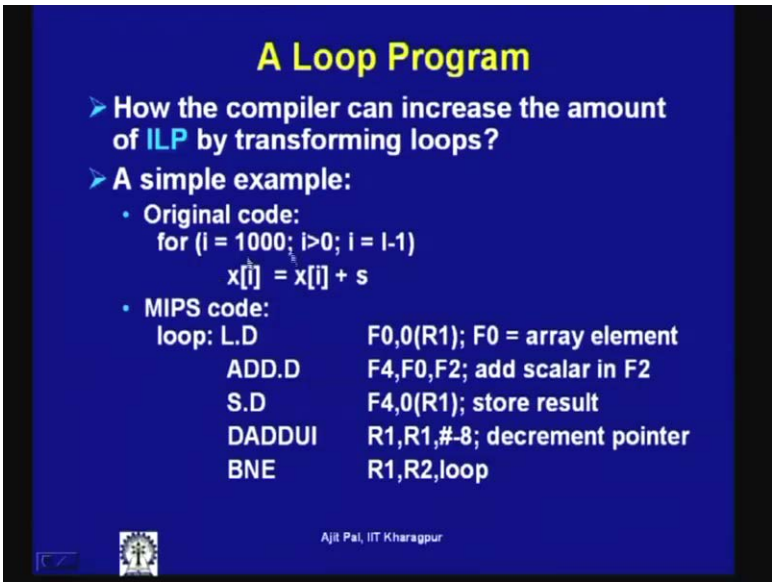
So, these two instructions can be interchanged, so these if these two instructions are interchanged. That means, if we write in this manner load word R 2 0 R 4, then second in load instruction load R 5, 4 R 4, and if the reordering is done in this way R 1, R 2 R 3. So, we find that in this particular case by the time this instruction will read the data from the register R 2, writing has been already completed because at the end of fourth clock

cycle rather in the first phase of the first half of the fifth clock cycle, writing will take place into the register by this instruction.

And reading here is taking place in the fifth clock cycle, and the second phase, so here the reading will take place, here the writing will take place. And as a consequence the hazard will not arise, and you will be able to read from the register, so this how the hazard is being overcome in this particular case. So, by reordering the instruction this is a transformed code there is no stall needed, so earlier there was a stall needed here, at the I mean after load word, you have to introduce a stall, if you have to correct data and this is not required in the transformed code.

So, this type of compiler rescheduling can be done with the help of a optimising compiler, we call it optimising compiler. Now, this type of situations of identifying independent instructions is very difficult in a small straight line code, usually the number of instructions present in a straight line code is very few. And since the number is very few, finding independent instructions among them is difficult. So, how can we improve the instruction level parallelism.

(Refer Slide Time: 23:23)



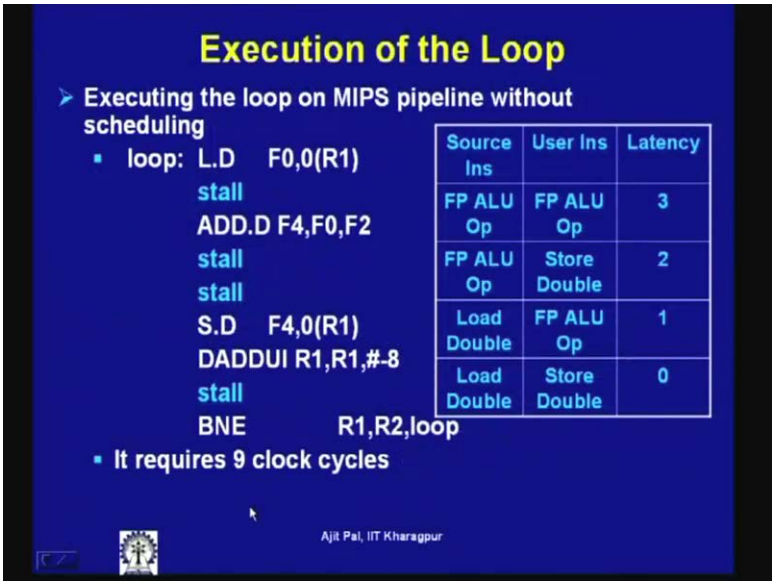
One very important technique which has been exploited that is known as unrolling, loop unrolling and that is actually related to loops. So, let us consider an example, you have got this original code i is equal to for 1000 semicolon i greater than 1 i is equal to i minus

1. So, it is reading data from a array and then performing the operation x i is equal to x i plus s, so s is a constant scalar value which is available in a particular register.

So, this is the high level language code, they corresponding MIPS code is given here, so first of all you are loading the value, you are loading it into the register. In a register F 0, F 0 is the first array element you are loading, then you are performing the additional operation assuming that the second the constant scalar value is present in register F 2. So, this is and then the content of F 0, and content of F 2 are added and that is being stored in F 4, and then you are storing the data in register F 4 using the register R 1 as the pointer and result is being stored.

And the next 2 instructions are essentially loop manipulation instructions, which are used for housekeeping the loop. So, the R 1 is decremented by, so these are double words, so you require 8 bites 64 bits, so 8 bites are required, so R 1 decremented by 8, and then it goes back it also checks whether that you know that 100 is being stored in R 2 that is why it compares and if branch not equal. That means, if they are not equal, then it goes back to the first one. So, in this way it keeps on doing the looping 1000 times looping will take place, so these are the 3 instructions, as you can we have already seen there is data dependency. And because of that dependency among these instructions if the instructions are present in this order you have to introduce stall.

(Refer Slide Time: 25:53)



So, executing the loop on MIPS pipeline without scheduling will require, how many stalls as you can see after load you will require 1 stall, this is based on this table. So, this is the source instruction, and this is the user instruction or dependant instruction you can say. And if it is a floating point ALU operation, and next instruction is also floating point ALU operation, and then if the dependant instruction is I mean if there is a latency of 3; that means, this from the source instruction to dependant instruction, you have to separate them by 3 instruction you have to if you want to avoid stall.

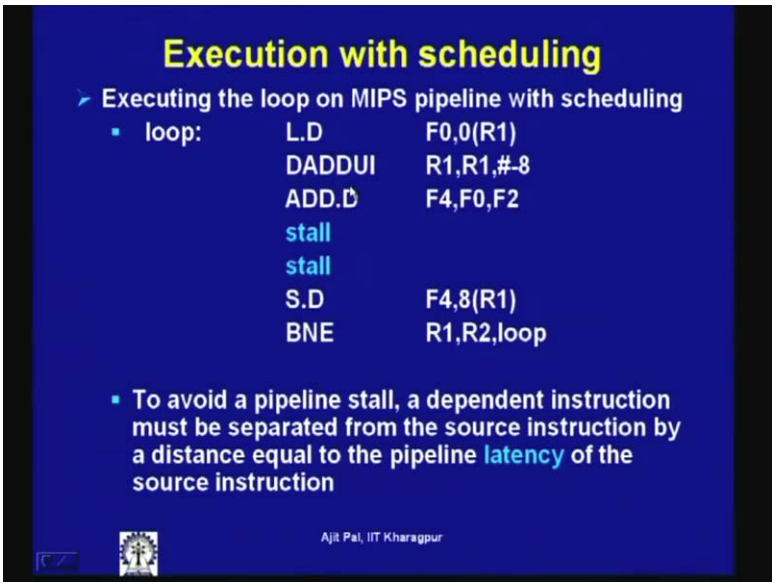
And the second one is floating point ALU operation, and the next one is store double in such a case it will involve the latency is 2. That means, you have to introduce to stalls to overcome data hazard in such a situation, the third one is load double by followed by floating point ALU operation, in such a case latency is 1. So, one stall has to be introduced, so in the first case that first stall is coming out of this that is means it you have got a load double, and it is followed by an ALU operation floating point ALU operation.

So, the latency is 1, so you have to introduce one stall, then coming to the second instruction add double F 4 comma F 0 comma F 2 here, you have to read the value from the register in this third instruction. And this belongs to the first type of problem that is floating point ALU operation, followed by floating point ALU operation, so dependant instruction has to be separated by you know there is a sorry store double. So, this store double sorry this is the second category floating point ALU followed by store double.

So, you will require 2 stalls, because latency here is 2 if we want to avoid stall, then here again you are decrementing a register. So, it is an ALU operation, and here this is a branch operation, so in this case also it will involve one stall because you are performing not equal, you are performing ALU operation. And, so loads double I mean you will require one stall here, it will belong to this type of problem.

And one stall has to be introduced here because you will not get the proper value in R 1, and I mean unless one stall is introduced you will not get correct value, it will not jump to the proper value. So, we find that 8 stalls I mean 1, 2, 3, 4, 4stalls are required and total number of clock cycles that is required to execute each parts of the loop is 9 clock cycle. So, 1, 2, 3, 4, 5, 6, 7, 8 and 9, 9 clock cycles, now let us see how by rescheduling these instructions the number of hazards can be reduced.

(Refer Slide Time: 29:32)



What has been done here, the that loop manipulation housekeeping instruction has been shifted moved earlier, earlier it was present here somewhere here from after stored data it was present here. Now, it has been shifted earlier, and since these two instructions are not dependent, so there is no need for introducing stall here, and these two load and add data has been separated by 1 instruction. And as a consequence no stall is required since they have been there was a latency of one clock cycle.

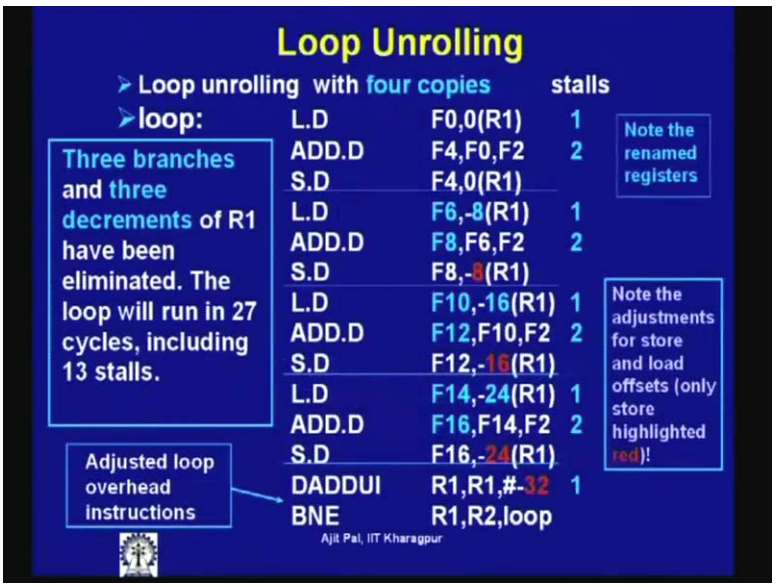
So, there will be no need for introducing stall here; however these two stalls will be present because add data it is followed by store data as per this you will require two stalls to be introduced. However, whenever this type of rescheduling is done you have to change the instruction, you may see that in the previous case this was you are decrementing by 8 the value of R 1 was decremented by 8 to point to the next element of the array.

So, but in this rescheduled code we are performing decrement earlier than, storing the value. So, that is why to store the result in the same memory location, instead of decrementing we are adding it because already decrement operation was done, so addition of 8 is performed. So, this instruction is modified, so that the array element; that means, result is stored in proper locations from the proper memory locations. So, it was present; that means, this will point to the same memory, as it was done by this instruction.

That means, although the instruction I changed, but the value that will be that effective at this that will be generated by this instruction, and this instruction will be same. So, as a result it will store the value in the same memory location because the effective add data is same. So, in this situation we find that two stalls are still present is there any way by which these two stalls can be overcome, so to avoid a pipeline stall a dependant instruction must be separated from the source instruction by a distance equal to the pipeline latency of the source instruction.

So, in this situation whenever you have got very few instructions in your straight line code, you cannot really overcome these stalls; however, this can be overcome, if you do loop unrolling. So, the loop will be unrolled by several times may be twice or thrice or fourth times, and then you have enough number instructions in your straight line code. They can be rescheduled and you will be overcome the stalls, let us see how the loop unrolling is being done.

(Refer Slide Time: 33:00)



So, here the loop unrolling has been done and you have got 4 copies, so 3 instructions load, add double, load double, add double and store double, these instructions which are repeated fourth times. You can see, this is 1 copy, this is 2'nd copy, this is 3'rd copy, this is 4'th copy, and after unrolling is being done; obviously, these 2 instructions are to be appropriately modified. That means, this is a unrolled code, where I mean unrolling has been done 4 times.

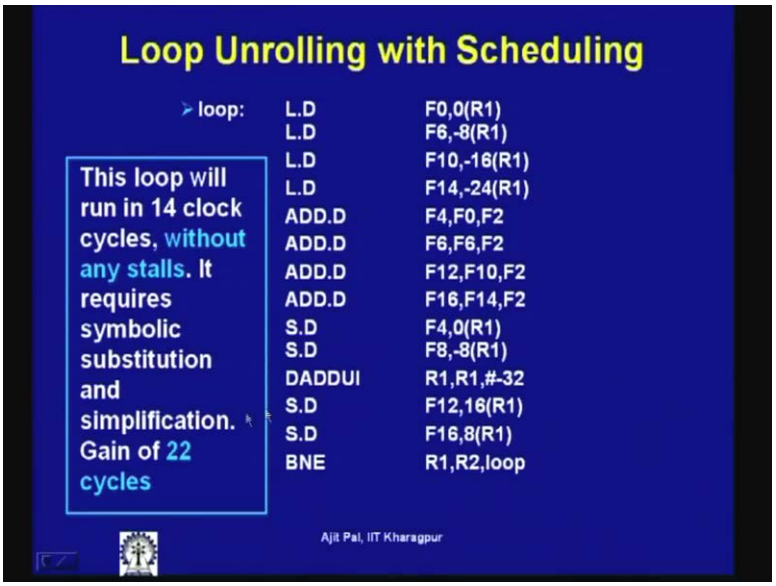
So, in this case how many times this loop has to execute, earlier the loop was executing 1000 times. But, now since each loop is performing 4 operations of a single loop earlier loop, you have to loop only 250 times because the number of I mean operations that is being performed is 4 in this case in a single loop. So, by this unrolling number of loops iterations will be reduced. However, the size of the code is increasing, the source size of the code is increasing.

Now, if it is present in this form; obviously, those stalls will be present 1 stall at the end of after load double, and 2 stalls after 8 double. And again for the next code again 1 and 2 stalls in this way we find total of 3 plus 3 plus 3 plus 3 plus 3; that means, 4 4 net 12 plus 1 13 stalls would be required and only a gain of 9 cycles. I mean without doing unrolling earlier what was happening, so 2 cycles were saved, but in this particular case how we are saving, we are saving because these two instructions is repeated only once instead of 4 times.

Because, you know since the unrolling has been done the number of iterations will be reduced from 1000 to 256. And as a consequence the number of instructions that will be executed will be reduced, these two instructions number of such instructions will be reduced leading to a gain of 3 cycles. Because, here you know it is executed only once instead of 4 times, so one stall plus this 1, so 3 cycles were required I mean for each loop earlier.

Now, you require only 1 and of course, the number of loops is reducing that is why a gain of 9 cycles just for this computations. So, 4 computations are being performed and you are saving 9 cycles, now you see the size of the straight line code is quite big, so we have a long straight line code, earlier you were having only 4 or 5 instructions we have seen here, very few instructions were present. And now you got large number of instructions, so the compiler has much more opportunity for rescheduling the instructions. So, scope for instruction level parallelism is increasing here, so you can exploit instruction level parallelism more whenever this unrolling is being done.

(Refer Slide Time: 37:00)



And you can see how it is being done, in this particular case this is the loop unrolling with scheduling. You can see, all the load operations have been performed because they are independent, and all the load after performing all the load operations and; obviously, the instructions will change. Because, you have to read the array element properly that is why the first, you can see after the first learning is done here it is minus 8; that means, next array element is R 1 minus 8, and next 1 is minus 16 and third one is minus 24.

Because, each element requires 8 bites and that is why this adjustment of the address is required to generate effective address for proper elements of the I mean. So, that it points to the proper array elements, so after loading is done you will be able to perform the add operations, now you see this load and you know that here you are writing in F 0 and this F 0 is here, it which is separated by 3 instructions. So, latency is 1 and it is separated by 3 instructions. So, there is no question of any hazard whenever you execute this instruction add double F 4 comma F 0 comma F 2.

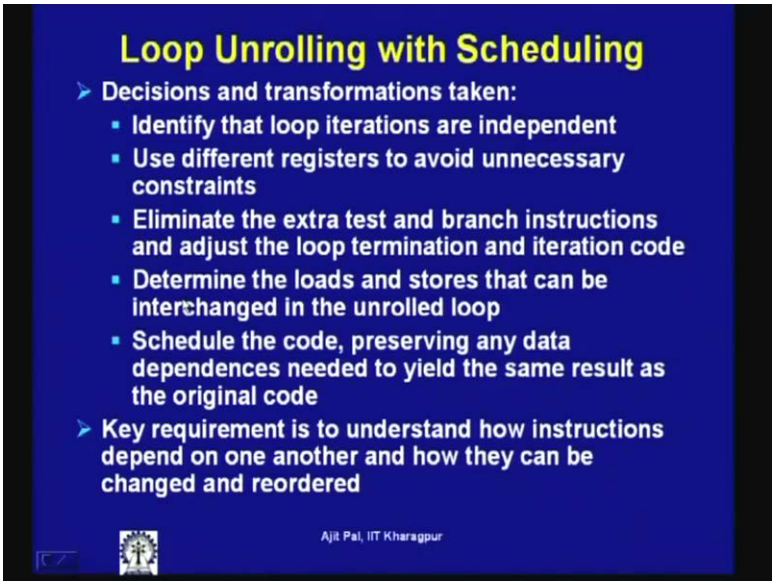
Similarly this instruction this load, and this instruction where F 6 is being read is separated by again by 1, 2, 3 instructions. So, latency is 1 and separation is by 3 instructions that hazard is being overcome, so in this way there is no hazard present here. Similarly, whenever you stored the data here you are performing and that F 4 was loading was done here, and it is separate by several instructions 1, 2, 3, 3 instructions and second load is also being performed.

Now, after this load second data loading is done, here again one this subtraction operation which is loop manipulation operation has been shifted. The reason is in this add double is being performed here, wherever writing in the register F 16 and that value sorry F 12 and that value is being read here. So, this separation is being again increased, so that there is no hazard, so in this way by scheduling the instructions in this manner, you are able to overcome all the hazards.

So, no hazard present here, so gain of 22 cycles you are able to achieve a gain of 22 cycles, whenever you perform this type of unrolling and scheduling up instructions. So, loop unrolling with scheduling we find we have been able to overcome all the hazards that was present in the execution. Now, the question arises what kind of problem, what is the problem we may face whenever we do this kind of loop unrolling and scheduling, it is definitely overcoming all the hazards.

But, will it introduce any new problem which may degrade the performance, by overcoming hazards stalls are removed; obviously, it will improve the performance we have seen you are gaining 22 cycles. But, is there any possibility of any reduction in performance because of this loop unrolling and scheduling.

(Refer Slide Time: 41:17)



We can see here, before I come to the disadvantages what kind of decisions and transformations had taken for loop unrolling and scheduling is explained here. What are the things you have to do, rather not you have to do rather the compiler has to do is being

explained, first of all identify the look iterations independent. It may, so happen that different iterations may not be independent, which is being performed in iteration i and in iteration i plus 1, they may not be independent.

So, loop unrolling is profitable only when the loop iterations are independent, in our example we have seen the operations that is being performed in 2 different iterations are completely independent. Because, the operations performed on different elements of array, since they are performed on different elements of array they are independent. But, it may happen for all kinds of loops, so first thing that you have to see is identify the loop iterations are independent.

Second is use different registers to avoid unnecessary constraints, we have seen in our original programme we have used very few register R 1, R 4 and R 2 only these 3 registers are being used in our original programme. But, whenever we did loop unrolling we have seen, we have used a large number of registers, we have used not only those floating point registers R 1, R 2 and R 4. We have also used a number of other registers F 4, F 6 you know F 6 were not used earlier, F 8 were not used earlier, F 10 were not used earlier, F 12 were not used earlier, F 14 were not used earlier, F 16 were not used earlier.

These all these registers were not used earlier; that means, whenever we perform loop unrolling, it is essential to use to have large number of registers to avoid name dependences. Because, you know if you use the same registers again this will lead to some king of dependency, and this dependency although not real dependency, but it will lead to hazard in some kind of pipelines. So, that is a reason why we have used a large number of registers and that register use of different register to avoid unnecessary constraints that is important, whenever you do loop unrolling.

Eliminate the extra test and branch instruction adjust the loop termination and iteration code, we have seen we have removed a large number of these 2 instructions this one, this instruction and branch not equal. And we have modified accordingly, such that it goes to the next iteration, determine the loads and stores that can be interchanged in the unrolled loop. So, we have this was our original code that was present immediately after unrolling, now this has been done by scheduling after unrolling.

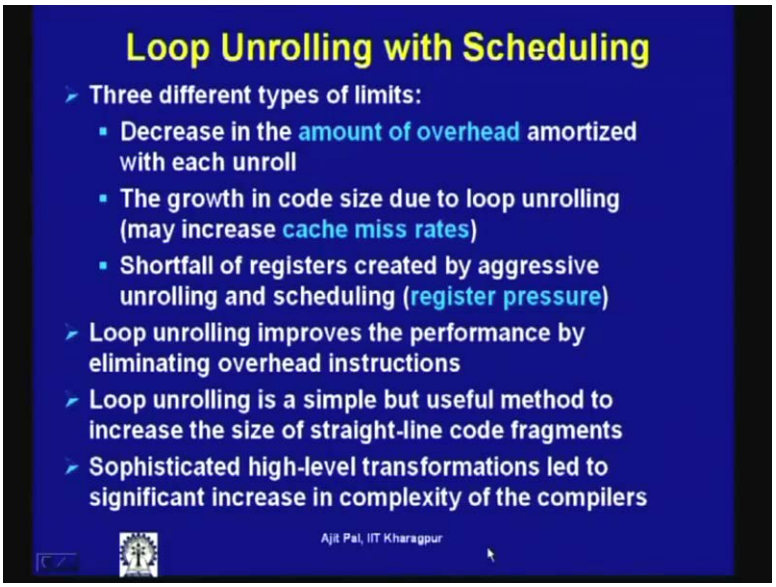
And we had to identify independent instructions, which can be placed one after the other, so determine the loads and stores that can be interchanged, in the unrolled loop. Schedule

the code, preserving any data dependences needed to yield the same result as the original code. So, this is the very important ultimately you have your results should be same, so maintaining the same result you are able to avoid the dependences, by scheduling the instructions.

So, basic idea is key requirement is to understand how instructions depend on one another, and how they can be changed and reordered. So, this is dependent on processor architecture, so what we are trying to tell that compiler must understand the processor architecture. Because, in the processor how many registers are available, fortunately for RISC processer, large number of registers are available for example, we have got 32 registers.

So, availability of large number of registers is available, secondly, that dependency that latency between instructions that is also dependent on the pipeline implementation. And those latency are to be understood by the compiler and accordingly scheduling of the instructions had to be done, and this is how a compiler can do the job.

(Refer Slide Time: 46:39)



Now, this loop unrolling has got three different types of limits, number one is decrease in the amount of overhead amortized with each unroll. So, the decrease in the amount of overhead as you are doing the unrolling, with each unroll you are able to reduce the amount I mean amount of overhead,. And we have seen that for each unroll we are able

to reduce two instructions, and as you do, so that decrease in amount of overhead amortized with each roll on each unroll.

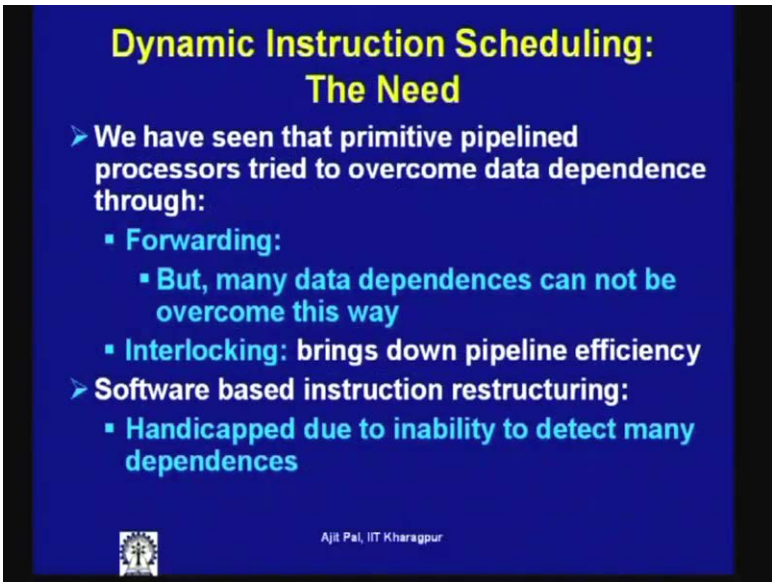
So, this is happening second is the growth in code size due to loop unrolling, so we have already seen that size of the code is increasing. That means, this has to be stored in the cache memory before the programme can be executed, as you know now a day's all processors are provided with cache memories. And those if the code size is small, a small number of instructions had to be cache memory, but if the size of the code is big then you know this will lead to what is known as cache misses.

So, this will lead to cache misses, so; that means, the growth in the code size due to loop unrolling may increase the cache miss rates. So, this will degrade the performance, later on I shall discuss about these cache miss how this cache miss occurs, whenever you have got cache memory. So, program that is transferred to the cache memory, there cache miss may occur if the size of the code is big, so this problem will also limit on the loop unrolling.

Third is shortfall of registers created by aggressive unrolling and scheduling, we have seen we are using registers, additional registers for each unrolled code to avoid various types of constraints. And whenever we are doing, so I know that a point will reach whenever we may not have any further register available, so this is known as register pressure. So, within the you have to exploit the registers which are available, and as the unrolling is done there will be a shortfall of registers. So, this is known as register pressure.

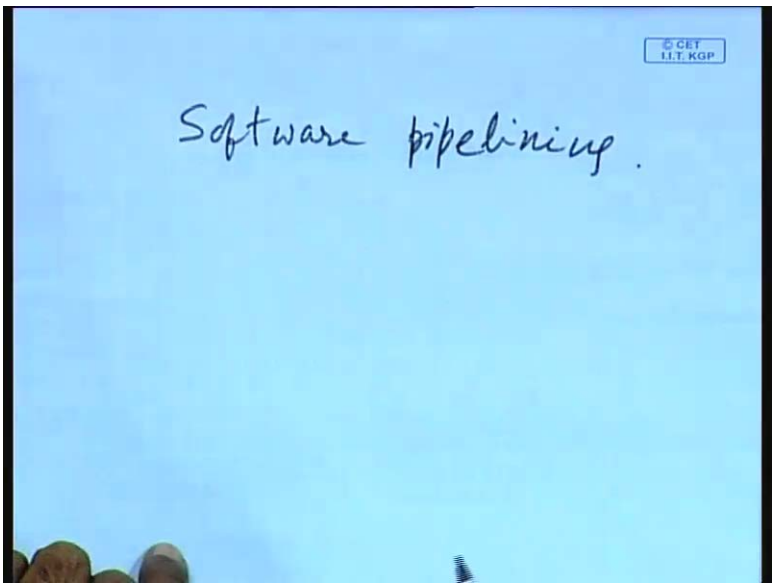
So, this will again limit on the put a limit on the loop unrolling, so loop unrolling improves the performance by eliminating overhead instructions that we have already seen. And loop unrolling is a simple, but useful method to increase the size of the straight line code fragments that we have already mentioned, and sophisticated high level transformations led to significant increase in complexity of the compilers. So, these three statements are very important because how loop unrolling is improving performance at the same time it is telling, in what way the complexity of the compiler is increasing. That means, the compiler writer has to be knowledgeable about the processor architecture, and then only the unrolling can be done in a effective way to overcome the stalls.

(Refer Slide Time: 50:15)



Let us stop here, we have discussed about the static scheduling of instructions with the help of compiler. And later on in a next class we shall discuss about another technique which is known as software pipelining.

(Refer Slide Time: 50:39)



The hardware pipelining we have seen in detail, where the instructions are executed in a overlapped manner different instructions. Now, we shall see software pipelining where we will see, instructions from different loops, different iterations will be executed in a overlapped manner. This is also a compiler based approach, by which the instructions

level parallelism is improved, and there will be stalls are avoided, so that I shall discuss in the next lecture.

Thank you.