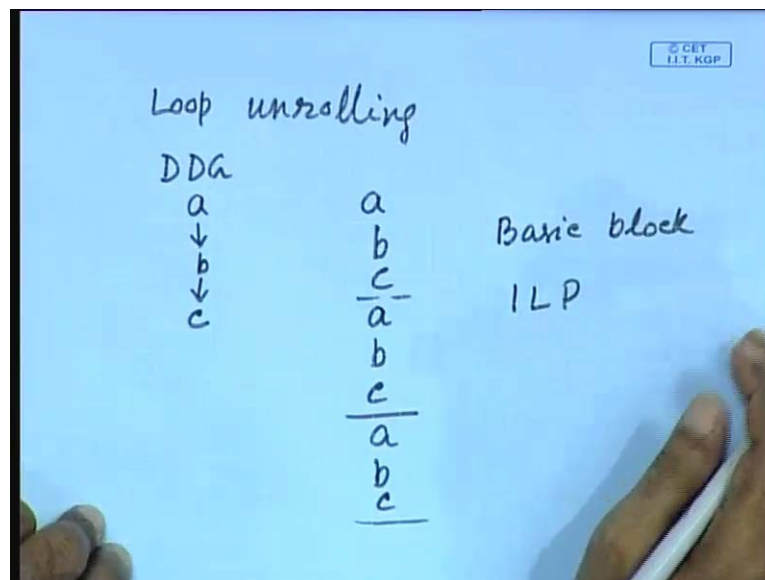


**High Performance Computer Architecture**  
**Prof. Ajit Pal**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology, Kharagpur**

**Lecture - 10**  
**Software Pipelining**

Hello and welcome to today's lecture on Software Pipelining, we are discussing about data hazards. And in the last lecture we seen how we can minimize or reduce the data hazards; that means overcome the stalls with a help of a software approach know as loop unrolling.

(Refer Slide Time: 01:15)

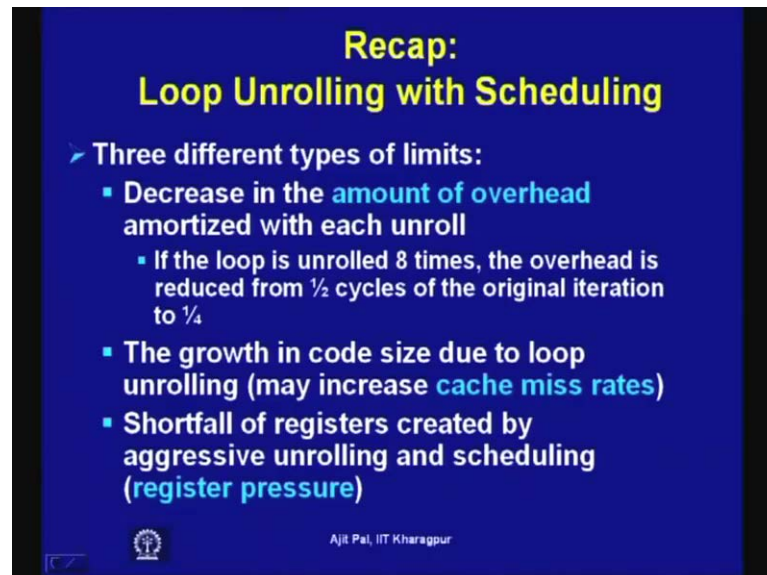


What it essentially does, it increases the size of the basic block, so suppose you have got a loop in which these are the three instructions, which are data dependent. So, in the main body of the loop you have got three instructions a b c, and this is the data dependency graph. So, these three instructions are dependent as a consequence, they cannot be executed simultaneously or in a overlap manner.

So, by using loop unrolling what is being done, a b c then if we unroll 3 times a b c and then a b c, so now you have got 6 instruction in your basic block. And since the instructions of the of different iterations or different loops are usually independent, within these 9 instructions the data dependency is obtain. So, instruction level parallelism is increased by using this basic block, and then these instructions are re

scheduled to execute the program in such a way that the stalls are reduced, that is basic approach.

(Refer Slide Time: 02:53)

A blue presentation slide with yellow and white text. The title is 'Recap: Loop Unrolling with Scheduling'. Below it, a bullet point lists 'Three different types of limits:'. This is followed by three sub-bullets: 'Decrease in the amount of overhead amortized with each unroll' (with a further sub-bullet about unrolling 8 times), 'The growth in code size due to loop unrolling (may increase cache miss rates)', and 'Shortfall of registers created by aggressive unrolling and scheduling (register pressure)'. The slide also features a small logo and the text 'Ajit Pal, IIT Kharagpur' at the bottom.

### Recap: Loop Unrolling with Scheduling

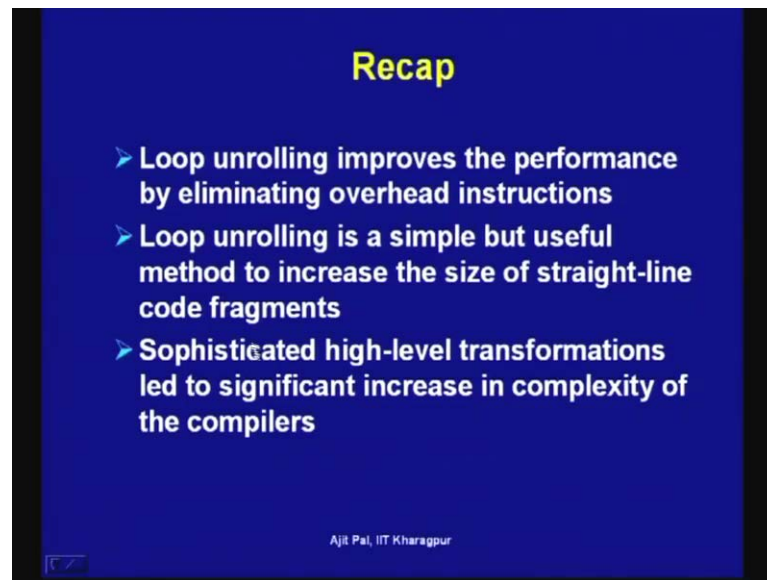
- Three different types of limits:
  - Decrease in the **amount of overhead** amortized with each unroll
    - If the loop is unrolled 8 times, the overhead is reduced from  $\frac{1}{2}$  cycles of the original iteration to  $\frac{1}{4}$
  - The growth in code size due to loop unrolling (may increase **cache miss rates**)
  - Shortfall of registers created by aggressive unrolling and scheduling (**register pressure**)

Ajit Pal, IIT Kharagpur

We have discussed in the last lecture and it is a software based approach, and we have also seen that loop unrolling with instructions scheduling have three different types of limits. Number 1 is that decrease in the amount of overhead, which is amortized with each unroll for example, if the loop is unrolled 8 times the overhead is reduced, from half cycles of the original iteration to 1 by 4.

So, that happens and the growth of the code size due to loop unrolling, that leads to cache miss rates and consequence you cannot really unroll many times. Third limit that that is due to shortfall of registers created by aggressive unrolling and scheduling, and this is known as register pressure we have already discussed in detail issues.

(Refer Slide Time: 03:41)

A blue slide with a black border. The title "Recap" is in yellow at the top center. Below it, there are three bullet points in white text, each preceded by a blue arrow. At the bottom center, the text "Ajit Pal, IIT Kharagpur" is written in small white font.

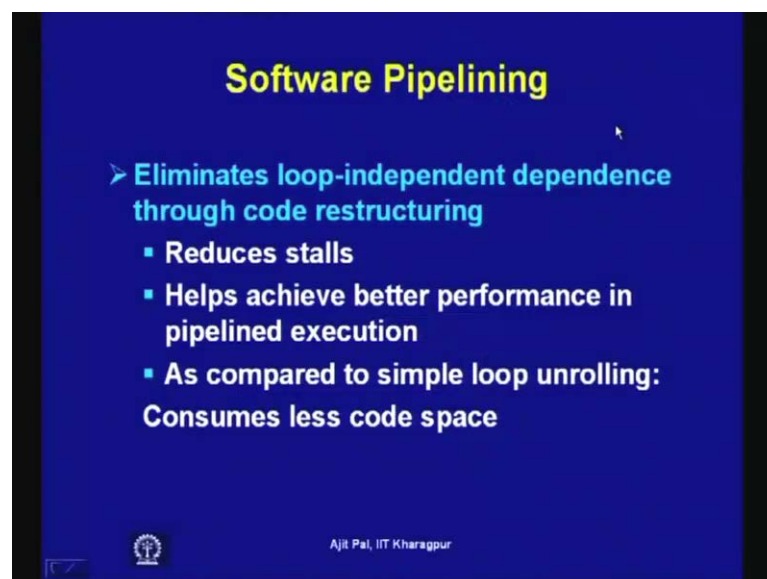
## Recap

- Loop unrolling improves the performance by eliminating overhead instructions
- Loop unrolling is a simple but useful method to increase the size of straight-line code fragments
- Sophisticated high-level transformations led to significant increase in complexity of the compilers

Ajit Pal, IIT Kharagpur

And you have seen that loop unrolling improves the performance by eliminating overhead instructions loop unrolling is a simple, but useful method to increase the size of straight line code fragments. This is a sophisticated high level transformations, which leads to significant increase in complexity of the compiler, so this loop unrolling and instructions schedule is done at the cost of increased compiler complexity. But, we have seen in spite of that fact it has got some limitations, because it increases the size of the code that leads to cache misses and other things.

(Refer Slide Time: 04:34)

A blue slide with a black border. The title "Software Pipelining" is in yellow at the top center. Below it, there are two main bullet points in white text, each preceded by a blue arrow. The second main bullet point has three sub-bullet points in white text, each preceded by a small white square. At the bottom center, the text "Ajit Pal, IIT Kharagpur" is written in small white font.

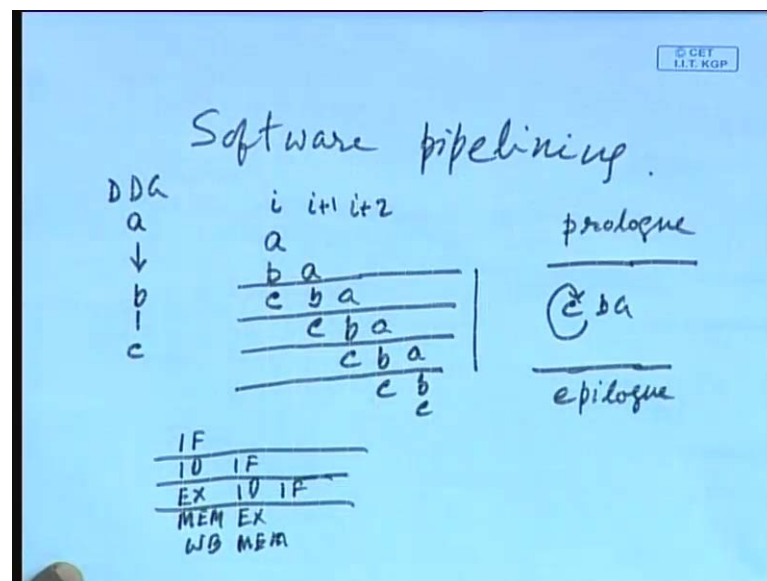
## Software Pipelining

- Eliminates loop-independent dependence through code restructuring
  - Reduces stalls
  - Helps achieve better performance in pipelined execution
  - As compared to simple loop unrolling: Consumes less code space

Ajit Pal, IIT Kharagpur

We shall discuss about another approach where some of these limitations are overcome, and this is known as software pipelining, and this eliminates the loop independent dependence through code restructuring. So, by restructuring the code the loop independent, dependences are eliminated and; obviously, it leads to reduced number of stalls it helps to achieve better performance in pipeline execution. And one more one very important factor is as compared, to simple loop unrolling consume less code space; that means, size of the code is small, now let me explain how exactly it is been done.

(Refer Slide Time: 05:19)



So, let us start with the gain the same data dependency graph a b c, this is your data dependency graph, and whenever we try to execute in a pipeline manner. We may write it this way a b c, then in the next cycle a b c, next cycle a b c, next cycle a b c, a b c and so on. Now, if we look at these we find that we can divide these into three parts, in this case you can see this instruction c of the first iteration, say maybe iteration i, instruction b of iteration i plus 1, and instruction a of iteration i plus 2.

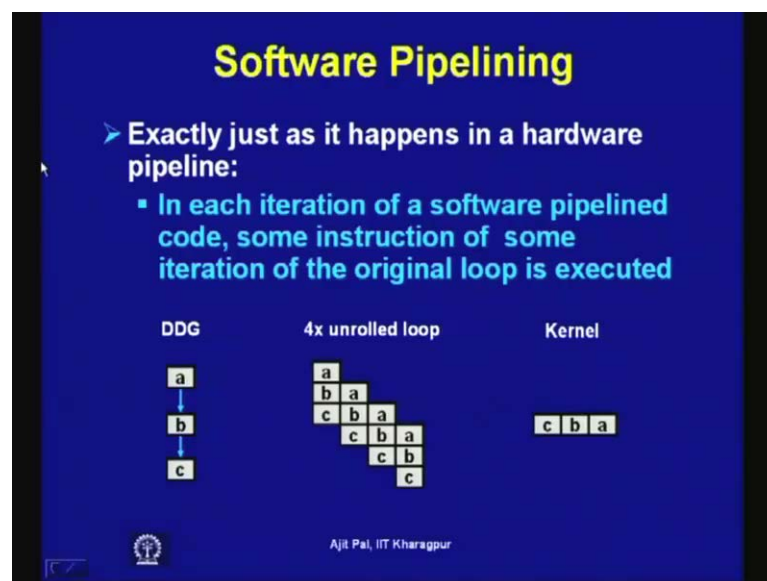
We can combine them, and for single loop where these three instructions will be executed, since they have been taken from different iterations. They are presumed to be independent; that means, this c this b this they belong to different iterations, and as a consequence they are independent; that means, here we can see that c b a, c b a, this will form a loop. That means, this part can be considered as a loop like this c b a, this will

form a loop and of course, you will have the these three instructions will be there they are to be executed.

And you have to execute the remaining instructions, which are left out, so it will have kind of prologue and epilogue, but this part can be executed in the form of a loop and by doing this. You find that there will be these three instructions are independent, there will be no stall in pipelines, so that is what is being done in loop unrolling; that means, we are taking instructions from different loops, as it is done in case of hardware pipeline.

We have seen in a hardware pipeline, what was done say instruction fetch, instruction decode, instruction execute, then memory, operation, then right back. So, what was done this instructions were executed or these instructions, instructions fetch they were executed in a pipeline. Similarly, here also you are doing the same thing, but here it was done for a single instruction, now we have taken instructions from three different iterations.

(Refer Slide Time: 08:55)



So, that is I mean from this simple feature, that means exactly just as it happens in a hardware pipeline. We are doing it in the same manner; that means, in each iterations of software pipeline code some instructions of some iteration of the original loop is executed. So, this what I have explain is shown in this, so we form a kernel c b a, which can be executed in the form of a loop, and this is essentially software pipelining.

(Refer Slide Time: 09:23)

## Software Pipelining

- Central idea: **reorganize loops**
  - Each iteration is made from instructions chosen from different iterations of the original loop

Software Pipeline Iteration

Ajit Pal, IIT Kharagpur

Let me, illustrate this with the help of an example, so this is the same I explained same thing explained central ideal reorganize loops, each iteration is made from instructions chosen from different iterations of the original loop. That means, here for example, we had iteration  $i_0$ ,  $i_1$ ,  $i_2$ ,  $i_3$ ,  $i_4$  and  $i_5$ , but some instruction from iteration  $i_0$ , some from  $i_1$ , some from  $i_2$ , some from  $i_3$  and some from  $i_4$ , are taken to form a loop. And then similarly, in the next iteration the instructions from  $i_1$ ,  $i_2$ ,  $i_3$ ,  $i_4$  and  $i_5$  is taken that forms another loop, so this is the software pipeline iteration and 2 iterations are shown here.

(Refer Slide Time: 10:10)

## Software Pipelining

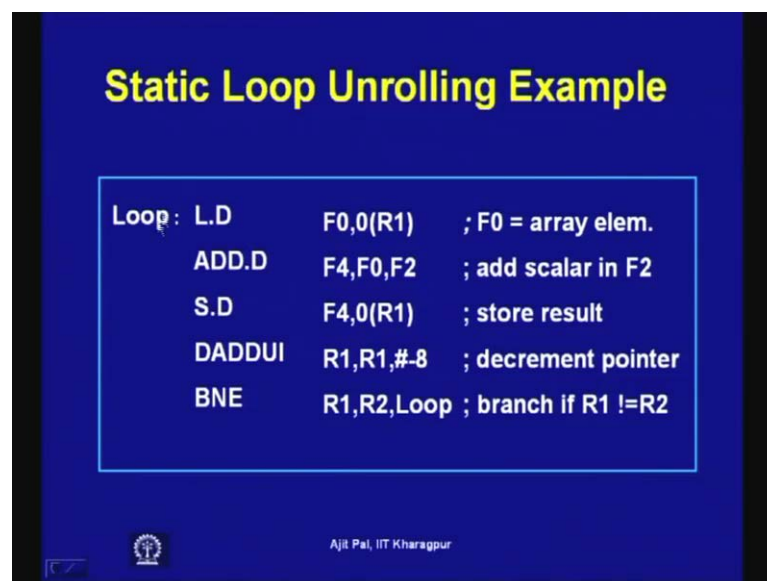
- How is this done?
  - 1 → unroll loop body with an unroll factor of  $n$ . (we have taken  $n = 3$  for our example)
  - 2 → select order of instructions from different iterations to pipeline
  - 3 → “paste” instructions from different iterations into the new pipelined loop body

Ajit Pal, IIT Kharagpur

Let me illustrate this with the help of an example, but before that it is explained how is this done, how these software pipelining is implemented or done, so number 1 step is unroll loop body with an unroll factor n. So, here also just like the loop unrolling is done to improve the instruction level parallelism, which I have explained in the last lecture. Here, also you will be doing loop unrolling, but for a different objective, here the objective is different the way the instructions are executed different from the previous case.

So, unrolling is done then select order of instruction from different iterations to pipeline, so here you have to select instructions from different iterations to form a pipeline. And then paste instructions from different iterations into the new pipelined loop body, so it will form a pipeline loop body, and taking instructions other pasting instructions from different iterations.

(Refer Slide Time: 11:26)



**Static Loop Unrolling Example**

```
Loop: L.D      F0,0(R1)    ; F0 = array elem.  
      ADD.D    F4,F0,F2    ; add scalar in F2  
      S.D      F4,0(R1)    ; store result  
      DADDUI   R1,R1,#-8   ; decrement pointer  
      BNE      R1,R2,Loop ; branch if R1 !=R2
```

Ajit Pal, IIT Kharagpur

Let me comeback to our original example, this was the static loop unrolling example that we with the help of this we explained the loop unrolling, and subsequently. How loop unroll loop unrolling was done to improve the instruction level parallelism, that I explained in the last lecture. Now, in this case you have got three instructions in the main body of the iteration, and these 2, the third and I mean fourth and fifth instructions they are essentially loop manipulation instructions, which are used for housekeeping.



(Refer Slide Time: 12:08)

### Software Pipelining: Step 1

Iteration i:	L.D	F0,0(R1)
	ADD.D	F4,F0,F2
	S.D	F4,0(R1)
Iteration i + 1:	L.D	F0,0(R1)
	ADD.D	F4,F0,F2
	S.D	F4,0(R1)
Iteration i + 2:	L.D	F0,0(R1)
	ADD.D	F4,F0,F2
	S.D	F4,0(R1)

**Note:**

1. We are unrolling the loop. Hence no loop overhead instructions are needed!
2. A single loop body of restructured loop would contain instructions from different iterations of the original loop body

Ajit Pal, IIT Kharagpur

Now, here as I mentioned the loop body is unrolled 3 times, so you have got 3 instructions, first 3 instructions belong to iteration 1, second 3 instructions belong to iteration i plus 1, then the last 3 instructions belong to iteration i plus 2. And of course, whenever loop unrolling is done, the loop overhead instructions are not needed, they have been removed. And so 2 instructions which are present here have been removed, 2 instructions which are present have been removed, and in a single loop body of restructured loop would contain instructions from different iterations of the original loop body.

(Refer Slide Time: 12:58)

### Software Pipelining: Step 2

Iteration i:	L.D	F0,0(R1)
	ADD.D	F4,F0,F2
	S.D	F4,0(R1)
Iteration i + 1:	L.D	F0,0(R1)
	ADD.D	F4,F0,F2
	S.D	F4,0(R1)
Iteration i + 2:	L.D	F0,0(R1)
	ADD.D	F4,F0,F2
	S.D	F4,0(R1)

**Notes:**

1. We'll select the following order in our pipelined loop:
2. Each instruction (L.D ADD.D S.D) must be selected at least once to make sure that we don't leave out any instruction of the original loop in the pipelined loop.

Ajit Pal, IIT Kharagpur

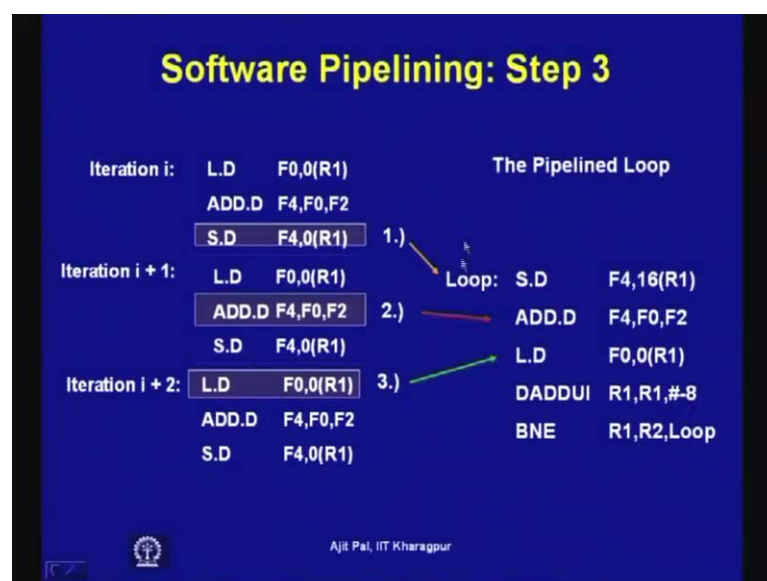


Let us, see how this is being done, now what you are doing, we are taking three instructions, first instruction from iteration 1, second instructions from iteration i plus 1, and third instructions from iteration i plus 2. So, whenever you take instructions from different iterations, you have to select it in such a way that each instructions must be selected at least once, to make sure that we do not leave out any instruction of the original loop in the pipelined body loop.

That means, here ultimately you have to execute your program, and it has to be a correct result and it will give correct result only when all the instructions are executed. And that 2 do that you have to be careful; that means, in this particular case for example, you are taking third instruction from iteration 1, second instruction from iteration i plus 1, first instructions from iteration i plus 2. So, you are taking all the 3 instructions, maybe from three different iterations in your loop body of the software pipeline.

And; that means, in one iteration these three will be executed, in the second iteration the other 2 instructions other; that means, another instruction from iteration 1 will be executed another instructions for iteration i plus 1 will be executed. And another instruction for i plus 2 is executed, so this is how it will be done, so this is a very simple program having only 3 instructions in the loop body. If you have more number of instructions in the loop body, you have to be very careful to pick up instructions and put them in your software pipeline.

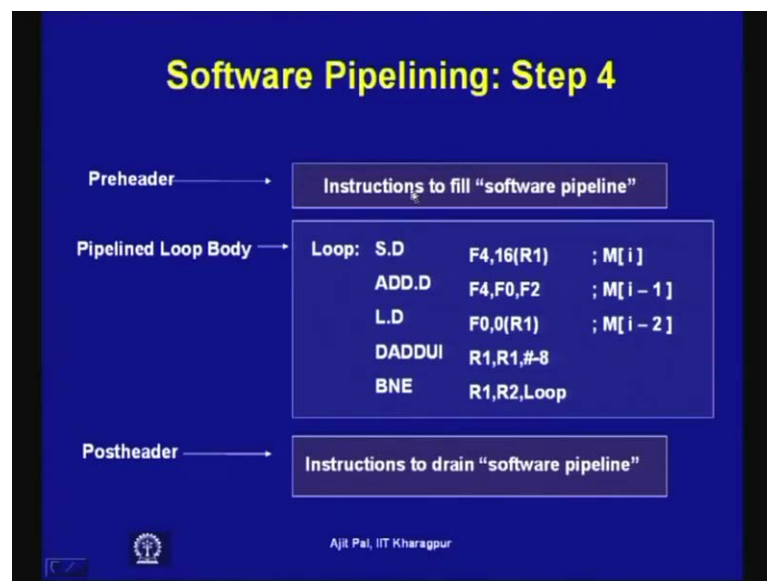
(Refer Slide Time: 15:01)



So, with these 3 instructions, now we have formed a loop, so here you have formed a loop as you can see taking instruction 3 from iteration 1, instructions 2 of iteration 2 iteration  $i$  plus 1 and instruction one of iteration 3. So, these three is forming the loop of the software pipeline, so they belong to 3 different iterations, then of course, since we are interested in forming a loop here. We have to put those loop manipulation instructions, this one will reduce the value pointer to point to the next element of the area, and this will decide how many times they looping will be done; that means, you have to carry out the execution of these loop.

That was for 1000 times our original program, it is not shown here, but in our original program it will loop for 1000 times, so to facilitate that that register  $r_2$  was to stored with the value I mean it was 0. So, initially  $R_1$  is initialized with 1000, and then it is decremented and until the then you keep on comparing, and then you whenever this two are equal, then you stop it from out of this loop, this is how it is being done.

(Refer Slide Time: 16:32)

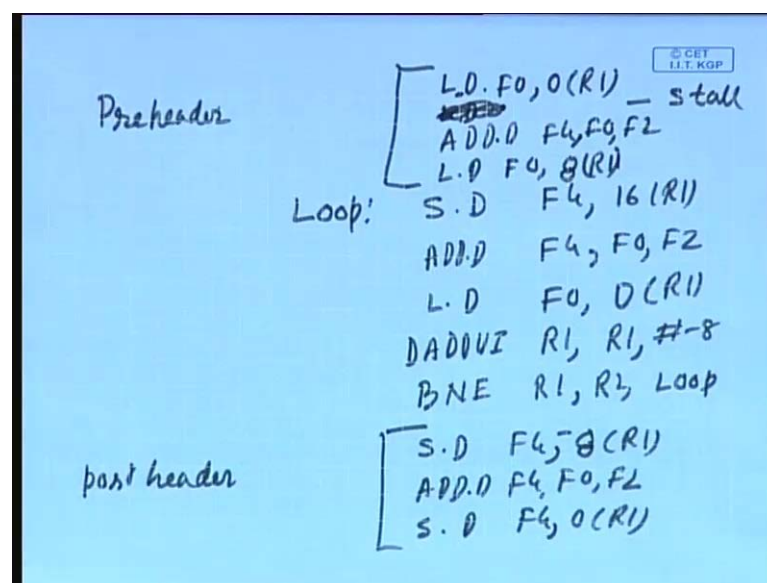


So, we find that the software pipeline will have now, this is the forth step where you will be having the pipeline loop body, and pipeline loop body you can see you have to adjust the the various values. I mean they, so the effective at this points to be right element of the (Refer Time: 16:58), and that is why here it is 16  $R_1$ , then you are loading the value of that era element register  $F_4$ . And you are adding it with the constant that is being stored in 0, this is the stored data, add data and load data, you have taken from 3 different

instructions, so you cannot really explain the operation from this thing, but you have to go back to original program to do that.

So, these are this is forming the basic loop and here this instruction has been taken from m i, m i minus 1, m i minus 2, so and you have the pre header, where you have to fill the add the instructions, which are to be executed. Before, you execute this loop, similarly after these loop execution is complete you will require several instructions present here, let us see what are the instructions they are present in the pre header and post header.

(Refer Slide Time: 18:24)



So, if we consider this your loop body is loop stored double, F 4 sixteen R 1, then add double F 4 F 0 F 2, then load double F 0 0 R 1. and this is decrementing the pointer DADDUI and R 1 comma R 1 comma minus 8. So, you are decrementing it and BNE R 1 r 2 loop, now this is your main loop body, what will be in your pre header you will must have pre header and post header. So, here we are left with 3 instructions that load F 0 comma R 0 R 1 at the F 4 F 0 comma F 2, so you have to execute these three instructions; that means, load double F 0 comma 0 R 1, and add double F 4 comma F 0 comma F 2.

Now, if these two instructions are executed one after the other, we know that this will lead to a stall here, what can be done another load that, second load that is present here this load can be filled up in between these two instructions. So, if you do that then load

double, you have to fill this one, with we cannot fill it this way I think we have to then you have to use different register value.

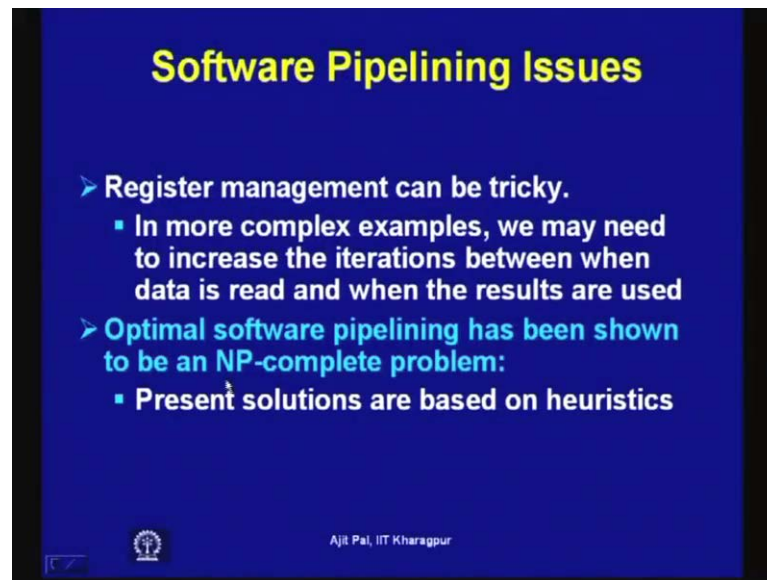
So, you have to load double F 0 0 R 1, so to here you have to actually increase the value by 8, because it points to the next element, so that you have to do. So, first instructions is done this way, and then second instruction is done this way, this is how you have to execute, I mean the pre header will form from these instructions. Maybe, you have to insert stall here, and your post header will require 3 instructions that a stored data F 4 0 R 1, add double F 4 comma F 0 comma F 2, and then stored double F 4 comma 0 R 1.

Now, you will see your R 1 was you have to adjust the value here, besides that these two are stored in two different places, so here actually this will be minus eight and this will point to be proper elements; that means, storing has to be done in a different way. So, these three instructions, and these three instructions are to be executed. before and after this loop body, so that will form the pre header and post header.

So, you have to fill those instructions, and these part that pre header and post header part may have few stalls, but main body of the loop will not have any stall, because they have been taken from different iterations and they are independent. So, this is the basic ideas, now let us consider the important issues related to software pipelining, so register management can be tricky in more complex example, we may need to increase the iterations between when data is read, and when the results are used.

Actually, if we do back to our original this problem, we find that in the pre header part, then the post header part the register management has to be done properly, such that ultimately the program to correct result. So, the way I have written it may not give you the correct result, because the register management part has not been taken to consideration properly, and that you have to take into consideration.

(Refer Slide Time: 23:51)



**Software Pipelining Issues**

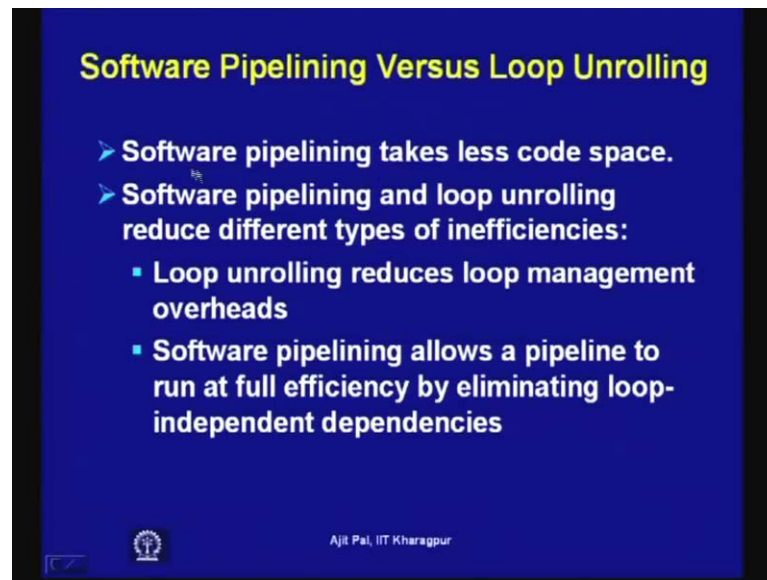
- Register management can be tricky.
  - In more complex examples, we may need to increase the iterations between when data is read and when the results are used
- Optimal software pipelining has been shown to be an NP-complete problem:
  - Present solutions are based on heuristics

Ajit Pai, IIT Kharagpur

Then optimal software pipelining has been shown to be an NP complete problem, so I have illustrated these with the help of very simple example, and it appeared to be very simple. But, whenever you consider real life problems, which are where the number of instructions in the loop body is more and later on, we shall discuss when it is being done in the context of your superscalar architecture. Then it then it becomes a very non-trivial problem, it has been found that it is NP complete problem and whenever the problem is NP complete, and you have to solve it.

There is no deterministic algorithm, which will give you an optimal result, so what you have to do you have to use a heuristic based approach, so that is being tried present solution are based on heuristics. So, heuristic based approach is used for software pipelining, and some lot of research has been carried out to achieve proper software pipelining, to improve the performance of the program execution.

(Refer Slide Time: 25:22)



**Software Pipelining Versus Loop Unrolling**

- **Software pipelining takes less code space.**
- **Software pipelining and loop unrolling reduce different types of inefficiencies:**
  - **Loop unrolling reduces loop management overheads**
  - **Software pipelining allows a pipeline to run at full efficiency by eliminating loop-independent dependencies**

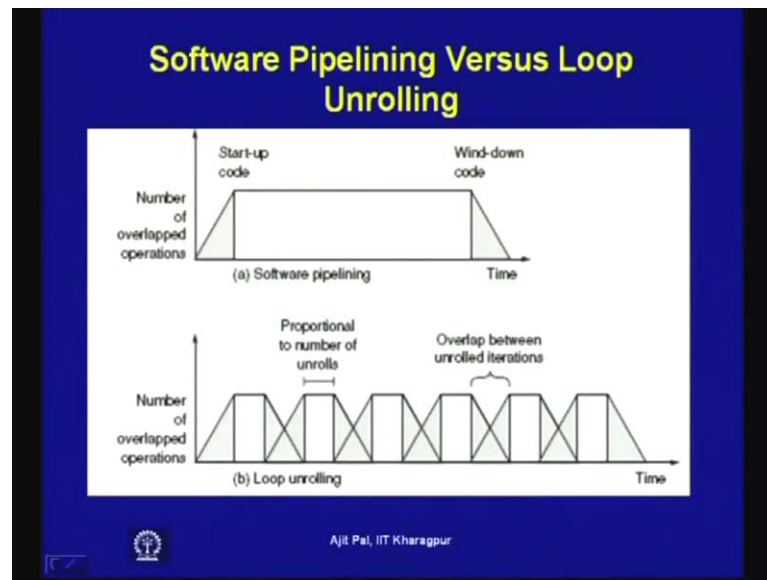
Ajit Pai, IIT Kharagpur

And another very important aspect is you can see, if we compare loop unrolling and software pipelining, we find that software pipelining takes less code space. We have seen that the one very important limitation of loop unrolling was that it increases the size of the code. So, since the size of the code is big, and you have to load it in the cache memory before we execute it will lead to cache misses, but when the size of the code is small that problem does not arrive.

So, this software pipelining facilitates that it has got less code space and Software pipelining and loop unrolling reduce different types of inefficiencies. So, the inefficiency in terms of instruction level parallelism that is present in the program, and it the two approaches reduce the inefficiencies in two different ways or other that different types of inefficiencies are reduced.

So, loop unrolling reduces loop management overheads, as you have seen it the additional loop management overheads, which are present if you we unroll each time you unroll, you reduce the number of those overhead. So, those overheads are reduced, and software pipelining allows a pipeline to run at full efficiency by eliminating loop independent dependencies. So, in case of software pipelining it allows the you know it improves efficiency by eliminating loop independent dependencies. That means, by taking instructions from different loops, which are independent, you are forming a loop body, and that is how you are increasing the efficiency of the program.

(Refer Slide Time: 27:26)

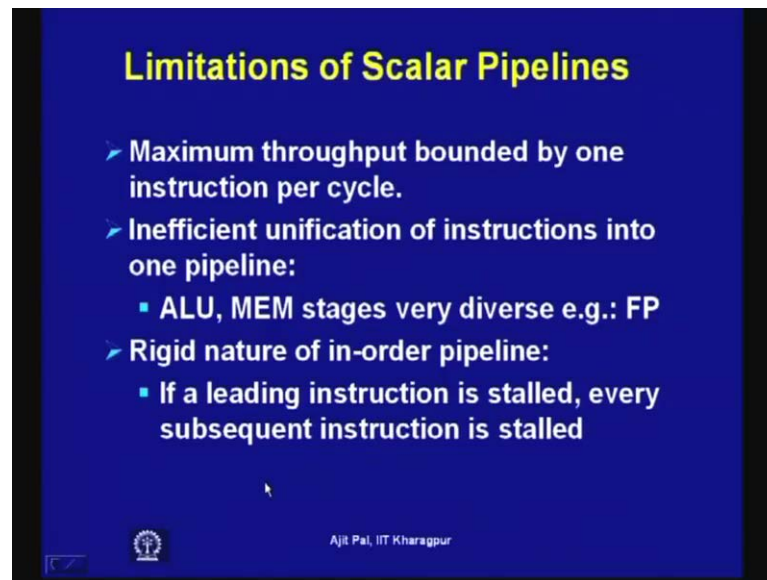


So, you can visualize the two approaches software pipelining and loop unrolling, the top diagram corresponds to software pipelining. So, as I mentioned there will be a start-up code and wind down code, you have got the software pipelining and no of overlapped operations such as one here. Here the number of overlapped operation is maximum, but this part and in this part the number of instructions, which can be executed in overlapped manner that will be reduced.

On the other hand the bottom diagram corresponds to loop unrolling, where only the middle part which is proportionate to the number of unrolls, where you have got the maximum number of instructions, which can be executed in a overlapped manner, but the other parts overlapped between unroll iterations. They are the you know number of instructions, which can be overlapped in a executed in overlap manner that is that is smaller, so these two diagrams compares or visualizes these two basic approaches.



(Refer Slide Time: 28:44)



Now, so far what we have tried, we have tried to unroll the loop or we have tried another approach that is software pipelining by which we have tried to execute the program. So, that the cycles per instruction CPI, the CPI of 1 is achieved; that means, maximum throughput bounded by 1 instruction per cycle. That is the maximum that can be achieved in both the approaches; that means, what you are doing that per cycle 1 instruction is executed, when it can happen where there is no stall.

So, only when there is no stall, we shall be able to achieve a upper limit of CPI is equal to 1, and beyond that cannot be done by using these approaches. So, inefficient unification of instructions into one pipeline; that means, we are trying to combine different types of instructions for example, ALU operations, memory stage operations, floating point operations. We are trying to do inefficient unification of instructions into one pipeline; that means, you are forming a single pipeline, where these are being inefficiently combined and this rig rigid nature of in order pipeline.

That means, we are trying to execute one instruction followed by another instruction, so they will be executed in a in order, but that is very rigid. In the sense that if a particular instruction execution is stalled, because of some reason, then the second instruction also installed, that is not allowed to progress, and so that is problem arises in a scalar pipelines.

(Refer Slide Time: 31:05)

## Higher ILP Processor

- **Pipelined Processors**
  - An ideal CPI of 1 can be achieved by eliminating data stalls using the techniques discussed so far
- **CPI less than one**
  - To improve performance further we may try to achieve **CPI less than 1**
  - Two basic approaches:
    - Very Large Instruction Word (VLIW)
    - Superscalar

Ajit Pal, IIT Kharagpur

Now, so higher ILP processor, how can we increase the ILP or we can have, how we can have CPI less than 1.

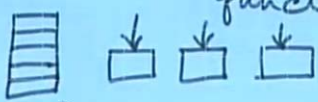
(Refer Slide Time: 31:33)

© CET  
IIT KGP

$CPI = 1$       Per cycle one instruction

$CPI < 1$

- VLIW (Very Large Instruction Word)
- Superscalar      More than one functional unit in a single CPU

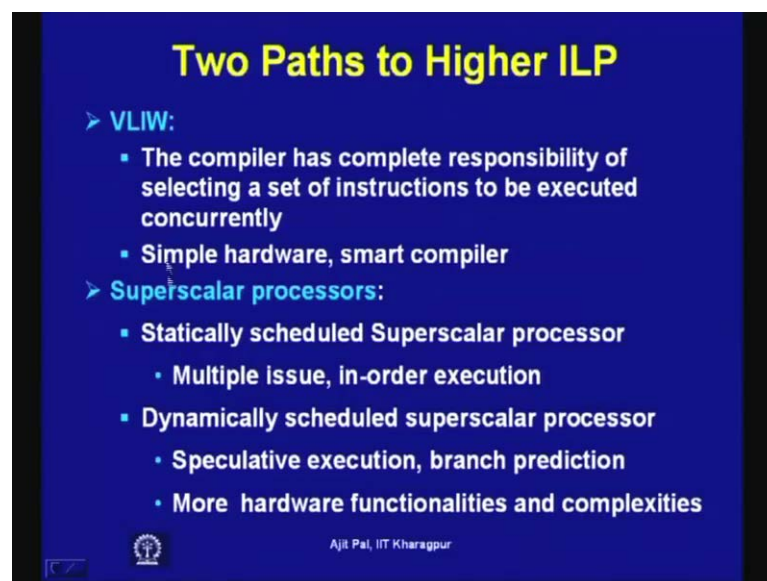


So, far we have assumed that our upper limit is CPI is equal to 1, we cannot still reduce it, CPI can be more than 1 whenever you have got stall, but now we are trying to achieve more than 1, I mean CPI which is less than 1. In other words we are trying to execute more than one instruction in a single cycle, and there are two basic approaches, one is

known as VLIW Very Large Instruction Word, and another approach is known as superscalar.

So, in both the cases the basic approach is to have more than one functional unit, so the number of functional unit that is present both in VLIW approach or superscalar approach is more than 1. So, far we assume that you have got only one functional unit, and where which is pipeline. Now, In VLIW or superscalar we shall be trying to, we shall be having more than one functional unit, since we have got more than one functional unit. We shall be able to issue more than one instruction at a time, more than one operation at a time; that means, this will help in getting the CPI which is less than 1. So, in a superscalar or VLIW processors you have got more than one functional unit in a single CPU, so you have got only one CPU, but unlike one ALU present in a CPU, you have got more than one functional unit.

(Refer Slide Time: 33:58)



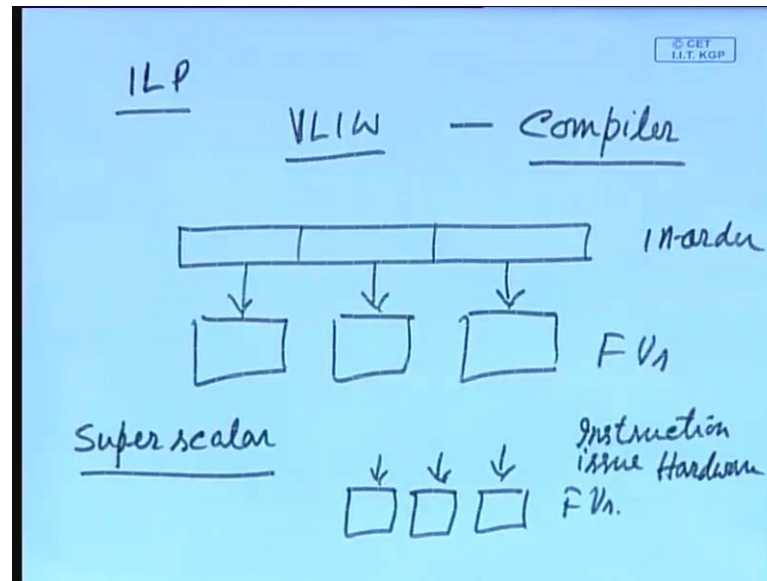
**Two Paths to Higher ILP**

- **VLIW:**
  - The compiler has complete responsibility of selecting a set of instructions to be executed concurrently
  - Simple hardware, smart compiler
- **Superscalar processors:**
  - **Statically scheduled Superscalar processor**
    - Multiple issue, in-order execution
  - **Dynamically scheduled superscalar processor**
    - Speculative execution, branch prediction
    - More hardware functionalities and complexities

Ajit Pal, IIT Kharagpur

So, that is the basic approach followed, but the way these two are done in two different cases are different, in case of VLIW the compiler has complete responsibility of selecting a set of instructions to be executed concurrently.

(Refer Slide Time: 34:19)



That means, that instruction level parallelism ILP that is being exploited in superscalar or VLIW, they are done in a different way, in VLIW the responsibility is given to compiler. That means, compiler identifies, which instructions can be executed in parallel, and those instructions for corresponding to those instructions you have got separate functional units and they are executed. That means, the compiler is given the complete responsibility for identifying see the instruction level parallelism, and then the single instruction will be having more one operation, which can be fed to different functional units, that is done in case of VLIW.

On the other hand in case of superscalar approach, there compiler is ordinary and simple, but the hardware identifies, which instructions can be issued simultaneously can be executed concurrently, so responsibilities done by concurrently. Then you know the several more than one instruction are issued, which are fed to different functional units, so these are the functional units here also.

So, both the cases here, but functional units, but in case of VLIW these instructions are formed with the help of compiler, then they are executed in order. On the other hand in superscalar the hardware finds out which instructions can be executed, so you have got a instruction issue hardware, which will generate several operations to be performed by different functional units. Now, in case of super superscalar processors, it can be done in two ways.

Number one is statically scheduled superscalar processor, where multiple issue is performed, but in order execution take place, on the other hand dynamically scheduled superscalar processor, which we will use very specialized feature. Like specialized property like speculative execution branch prediction, and where you will allow out of order execution, so this will require more hardware functionalities and complexities. So, later on, I shall discuss about these two techniques, which we will provide you higher ILP and of course, loop unrolling will be necessary software pipelining will be necessary to have more number of I mean to increase the ILP.

(Refer Slide Time: 37:46)

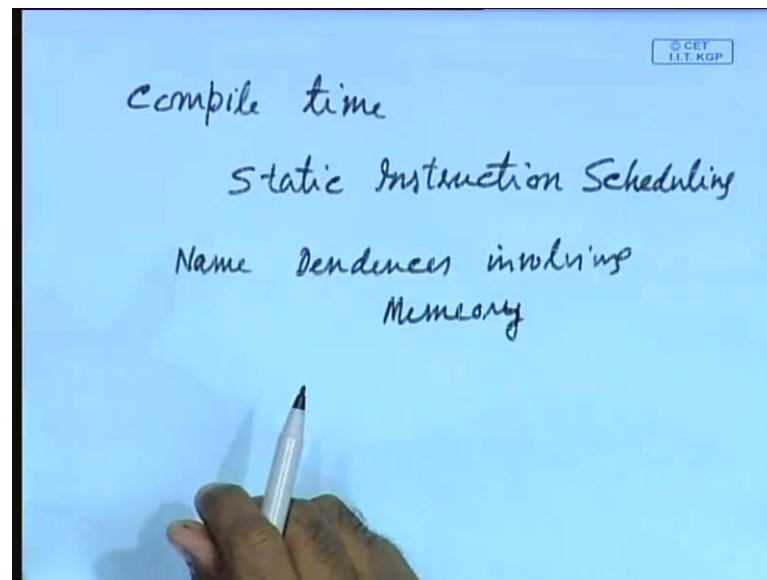
**Dynamic Instruction Scheduling:  
The Need**

- We have seen that primitive pipelined processors tried to overcome data dependence through:
  - Forwarding:
    - But, many data dependences can not be overcome this way
  - Interlocking: brings down pipeline efficiency
- Software based instruction restructuring:
  - Handicapped due to inability to detect many dependences

Ajit Pat, IIT Kharagpur

Now, so far what I have discussed is known as static instruction pipelining, which is done by compiler, another approach is known as dynamic instruction pipelining (Refer Time: 38:05) dynamic instruction pipelining is needed. And the incase of pipelining we have seen some hardware technique forwarding developing technique, where the there is a stall I mean when there is a hazard stalls are introduced, or software based instructions is done. But, this software based instruction is structuring is handicapped due to inability to detect many dependences, we have discussed about different types of dependences.

(Refer Slide Time: 38:46)



Those dependences, which are visible at compile time can be done with the help of static instruction scheduling, so it is very conservative in nature, on the other hand particularly there are situations; that means, name dependences involving a memory. If name dependences involving the memory is present in your program, this cannot be identified by the compiler at compile time, because they will be evident only when the program is executed that at one time. So, the dependencies which not revealed at compile time will be visible at runtime, and that is what is been done, in case of dynamic instruction scheduling with the help of a hardware.

(Refer Slide Time: 40:03)

### Dynamic Instruction Scheduling

- **Scheduling:** Ordering the execution of instructions in a program so as to improve performance
- **Dynamic Scheduling:**
  - The hardware determines the order in which instructions execute
  - This is in contrast to statically scheduled processor where the compiler determines the order of execution

Ajit Pal, IIT Kharagpur



So, the hardware determines the order in which instructions execute, so this is in contrast to statically scheduled processor, where the compiler determines the order of execution. And later on I shall discuss a technique by which this hardware scheduling is done, we will require a very specialized hardware, which will do this instructions scheduling. And the loop unrolling another thing which is been done by the compiler, will not be necessary whenever you do it with the help of a hardware. And various other things like the that registering, and other thing they are also incorporated at the time of dynamic instruction scheduling.

(Refer Slide Time: 40:52)

### Points to Remember

- What is pipelining?
  - It is an implementation technique where multiple tasks are performed in an overlapped manner
- When can it be implemented?
  - It can be implemented when a task can be divided into two or subtasks, which can be performed independently
  - The earliest use of parallelism in designing CPUs (since 1985) to enhance processing speed was Pipelining
- Pipelining does not reduce the execution time of a single instruction, it increases the throughput
- CISC processors are not suitable for pipelining because of:
  - Variable instruction format
  - Variable execution time
  - Complex addressing mode
- RISC processors are suitable for pipelining because of:
  - Fixed instruction format
  - Fixed execution time
  - Limited addressing modes

Ajit Pal, IIT Kharagpur

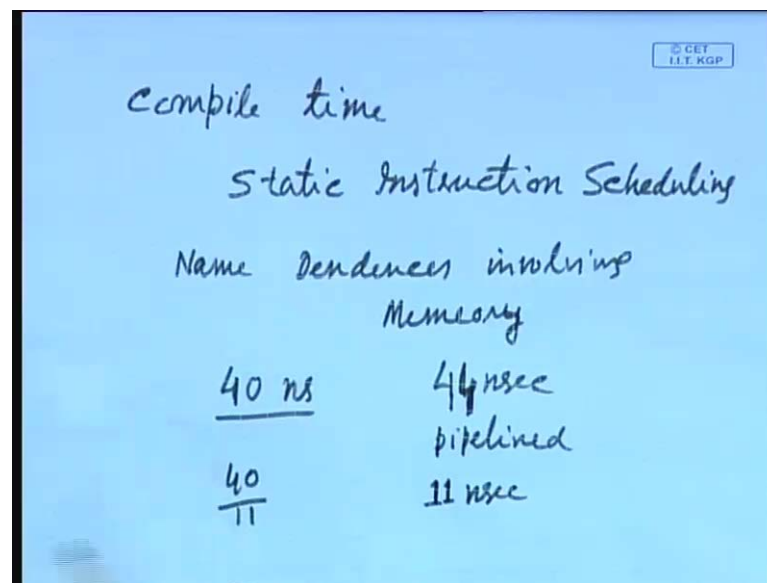
So, that we have come to the end of a very important topic that is instruction level parallelism, where the simple pipelines are used, and that instruction level parallelism is incorporated to achieve CPI 1. So, some of the important points, you should remember before we leave this topic is given here, first of all what is pipelining that has been that I defined in the beginning.

What is an implementation technique, where multiple tasks are performed in an overlapped manner, you may recall that, and when can it be implemented I mentioned that it can be implemented when a task can be divided into two or subtasks, which can be performed independently. And second point is the earliest use of parallelism in designing CPU's to enhance processing speed was pipelining, so pipelining was the first parallelism, that was incorporated in processors.



And pipelining does not reduce execution time of a single instruction, it increases the throughput, that I have highlighted many times. Whenever, you execute instructions in a with the help of a pipeline processor, time needed to execute a single instruction is not reduced rather it increases. Because, you are performing different parts of an instruction by different stages, in different stages, but in between you have got those pipeline registers or latches, which introduces some delay.

(Refer Slide Time: 42:38)



compile time	
Static Instruction Scheduling	
Name	Dependencies involving Memory
40 ns	44 nsec
	pipelined
40	11 nsec
11	

So, if you consider the time needed to execute a single instruction maybe instead of 10 nano second, it may take 11 nano second or more, so time need to execute a single instruction reduces it is not reduced. So, you have seen it was 40 nano second and nano second is non pipeline processor and 44 nano second was required, in a pipeline processor. We can see for each latch one additional you all have done, that is why in a pipeline processor single instruction was taking 44 nano second; however, if you consider the through you will find that on the average far 11 nano second.

You are getting 1 output, and that is how the throughput is increased, and giving you speed up of 40 by 11. So, this is in highlighted in this point and another issue that I mention in detail we have discussed about two types of processors CISC and RISC having different features. CISC processors are not suitable for pipelining because of variable instruction format, variable execution time, and complex addressing mode, on

the other hand RISC processors are suitable for pipelining, because of fixed instruction format, fixed execution time, and limited addressing modes.

So, we have restricted our discussing to pipelining of this processor; however, later on I shall discuss about that INTEL series of processors, which are essentially CISC. But, in those processors internally the complex instructions are decomposed into RISC like operations, and they are executed in a pipeline manner later on I shall discuss about it.

(Refer Slide Time: 44:38)

**Points to Remember**

- There are situations, called **hazards**, that prevent the next instruction stream from getting executed in its designated clock cycle
- Three major types:
  - **Structural hazards:** Not enough HW resources to keep all instructions moving
  - **Data hazards:** Data results of earlier instructions not available yet
  - **Control hazards:** Control decisions resulting from earlier instruction (branches) not yet made; don't know which new instruction to execute
- **Structural Hazard** – can be overcome using additional hardware
- **Data Hazards** – can be overcome using additional hardware (**forwarding**) or software (**compiler**)

Ajit Pal, IIT Kharagpur

Then I discussed about hazards, there are situations called hazards that prevent the next instruction stream from getting executed in its designated clock cycle, and we have discussed about 3 different types of hazards. Structural hazards which arises due to non availability of enough hardware resources, and data hazards results up earlier instructions not available. That means the incase of data dependency results needed by subsequent instruction is not available, because the it has not yet been completed.

So, that is why data hazards occurs, and we have discussed various techniques over occurring data hazards. Later on, we shall discuss about the control hazards techniques of overcoming control hazards, so control decisions resulting from earlier instructions not yet made; that means, that decision is sometime is required to take a decision, whether a branch will be taken or not taken.

And, because of that delay there will be some stalls to be introduced, and how that can be minimize that we shall discuss later, so we have discussed techniques for overcome structural hazard and data hazards. And in the next class we shall discuss in more detail about the superscalar and VLIW processors.

Thank you.