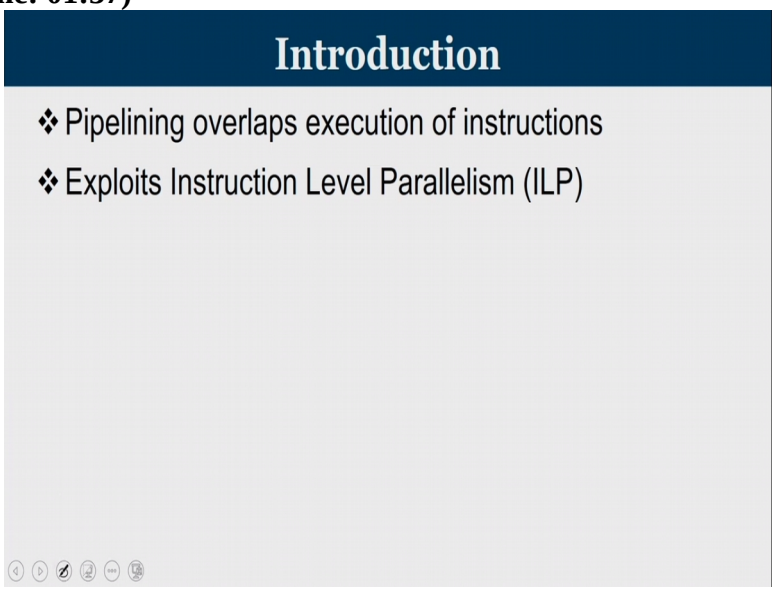
Advanced Computer Architecture
Dr. John Jose, Assistant Professor
Department of Computer Science and Engineering
Indian Institute of Technology Guwahati, Assam

Lecture 7
Compiler Techniques to Explore Instruction Level Parallelism

Welcome to lecture number 7. Today, our discussion is on compiler techniques to explore instruction level parallelism. Over the first 6 lectures, we were trying to understand how a basic pipeline works. And we have seen multi cycle pipelines as well. And the hazards that is happening in terms of data hazard, then control hazard, structural hazard, we have seen certain techniques, how these hazards are been dealt with.

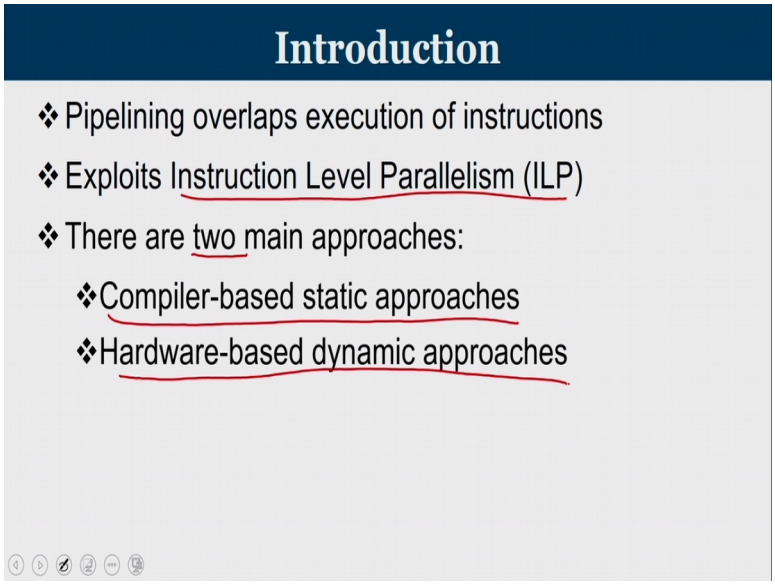
In today's lecture, our focus of attention is on how can a compiler helps the hardware, we know that compiler is going to act as a software layer above the bare hardware, if the architectural features of the hardware is been shared with the compiler can compiler help in optimizing the code such that once the optimized code runs on the hardware, can it be running at a faster rate. So, the title for today's lecture is compiler techniques to explore instruction level parallelism.

(Refer Slide Time: 01:37)



We know that pipelining overlaps execution of instructions and we are basically focusing on how to improve instruction level parallelism.

(Refer Slide Time: 01:51)



So, when you have multiple instructions that is available, the whole concept of your pipeline is revolving around. Can you exploit instruction level parallelism across multiple instructions, if there is parallelism between them that means, they can be run parallelly that is what we are looking in pipelining concept. Now, when you deal with the exploration of instruction level parallelism, there are basically 2 approach.

First is compiler based approach they are known as static level approaches and second one is hardware based dynamic approach. So, the broader difference between these 2 terms, that is compiler based techniques and hardware based technique is. In the case of a compiler based technique, whatever is the code given by the programmer, this programmers code is been given to the compiler.

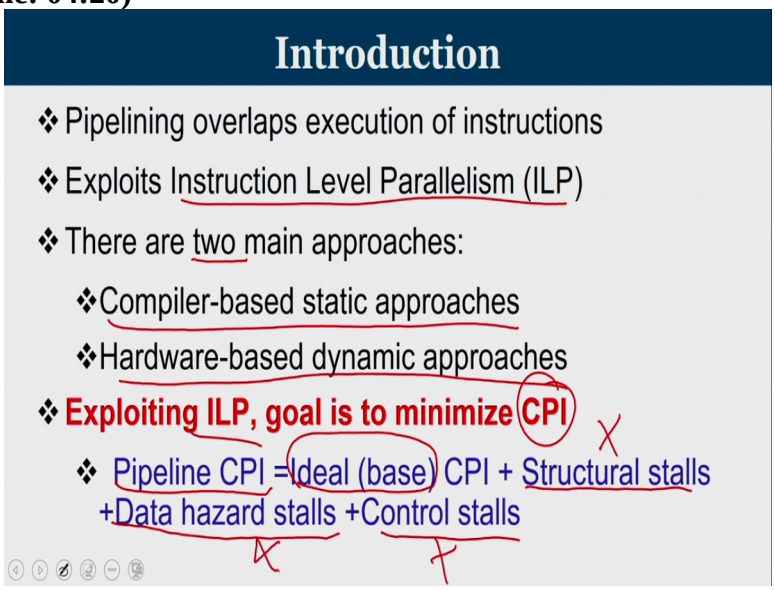
Compiler will look into the impact of when these codes when these set of instructions are going to hardware, will there be any stall between them that is what has been studied and compiler exploits the option of is it possible to rearrange these instructions such that when certain instructions are kept a little apart, then there would not be the direct dependency or the stalling that happens.

In short, a compiler is trying to reorder the instructions when it is been supplied to the hardware. So, as far as the instructions once they reached the hardware, hardware does not look into any kind of reordering, hardware has to just run these instructions in a pipelined fashion. So, by this reordering compiler make sure that once the instruction is running in the hardware, the number of stalls are minimized.

The second approach is called hardware based approach they are also known as dynamic approaches, wherein, the compiler simply translates the code and then supply the instructions in the same order what it got. Now, the hardware will have intelligent units or control units which will look into these instructions and trying to see whether these instructions if run parallelly or if then adjacent slots will it create stalls or not.

So, in compiler based approaches, the instructions that the hardware received they are executed in the order itself whereas, in the dynamic approach or hardware based approach, the instructions are executed out of order.

(Refer Slide Time: 04:20)

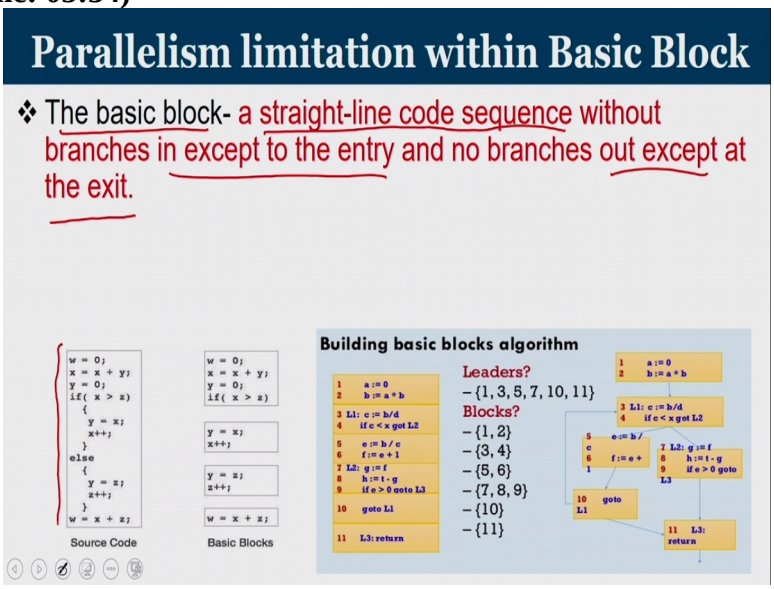


Now, when you try to understand how can you exploit instruction level parallelism, our main purposes is can we reduce cycles per instruction. The number of CPU clock cycles needed to complete an instruction is going to increase if there are stalls. So, the pipeline CPI is ideal CPI or base CPI that is the case wherein you are going to complete 1 instruction say per calclation then ideal CPI is going to be 1.

Now on top of this whenever we have structural hazard, there are certain stalls that are introduced, we have data hazard stalls, and we have control hazard stalls. Ideally, if there are no structural hazard, no data hazard and no control hazard, then the CPI in a pipelined processor is known as the base CPI, cycles per instruction every cycle you are going to fetch an instruction and naturally every cycle you are going to complete an instruction that makes the CPI equal to 1.

And when there are hazards, then CPI value will increase. And the whole purpose of exploiting instruction level parallelism is can I reduce structural hazard, can I reduce control hazard, and can I reduce data hazard?

(Refer Slide Time: 05:34)

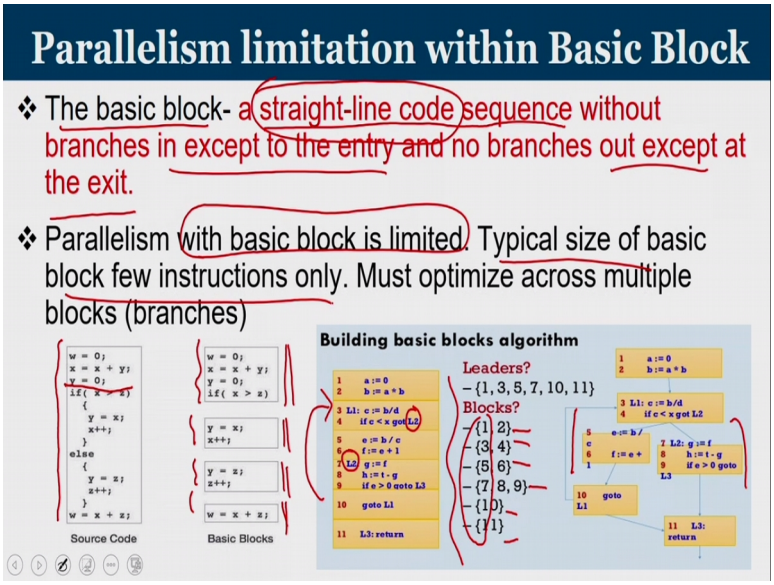


Now, what are the limitations, when we try to optimize the code that is coming to hardware for execution? Before that, let us try to understand a term called as basic block. A basic block in a program sequence is a straight line code sequence, set of straight line code without any branches, except at the entry point there can be a branch, and there would not be any branch at all, except at the exit.

So, here is there is a piece of line that has been given let us try to understand this. So, you have a set of lines that is been given and the concept of basic block is within a block there would not be any branches at all. What do you mean by this? When you have sequence of instruction, whenever a branch instruction starts, you are starting a new basic block. And if at all a new branch instruction is somewhere there in between there a new basic block starts.

So, the idea of basic block is control will reach the basic block only at that point and control will leave the basic block only at the exit. If at all there is any other point where there is multi point diversion as far as the control is concerned, a new basic block is starting.

(Refer Slide Time: 06:58)



So, in this given example, you can see that, w = 0, x = x + y, y = 0 if x greater than z. So, this is the point where in a branch condition is going to start and we know that there is an else component that is coming. So, when you consider about the basic block, this is one block, this is another block, this is another block and this is the third block. And the peculiarity of each of the basic block is there are no branches inside the statements in the basic block that is a straight line code sequence.

All these blocks that we have identified, we can know that there is only straight line sequence. There are no branching in between. Now let us see another example where, a = 0, b = x * p and then you have a line 1, that is the point at which the control will reach, so there is a go to line 1 here control can reach line number 3, control can reach line number 3 from line number 10 or control can reach line number 3 from line number 2, as far as the basic sequence is concerned.

Now in line number 4, there is a jump to L2 and L2 is a label that is given up line number 7. So if you look at line number 1 and 2 will form 1 basic block, 3 and 4 will form another basic block because there is a branch, it is a target of a branch and whenever you get a new branch, the very next line after the branch should be reachable in different ways. So, line number 3, 4 is the second basic block.

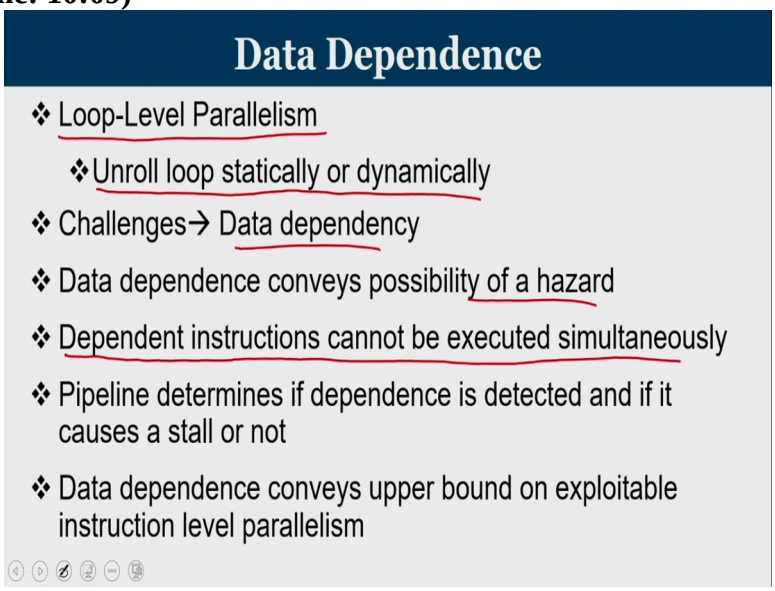
5, 6 is the another basic block, 7, 8 and 9 will become another basic block. 10 is a basic block and 11 is a basic block. So, these are the basic blocks that we have identified. So, this means

line number 1 and 2 is 1 basic block 3, 4, 5, 6, 7, 8, 9, 10 and 11. Now, the very first line of each of the basic block is known as leaders.

So line number 1, line number 3, line number 5, 7, these are all leaders, because control can reach at this point only from 1. What do you mean by this when you represent the basic block in a graphical fashion? At the end of line number 4, there is a possibility that you will look into line number 5, 6 or you can go for line number 7, 8, 9 as well. What is the purpose of understanding a basic block in the case of an instruction level parallelism, what are we trying to do is we are trying to exploit instruction level parallelism within a basic block only.

So the number of lines within a basic block is very limited, you get marginal performance improvement by exploiting instructions from parallelism, and we know that typical size of basic block is very few instructions only. So, the kind of performance improvement that we get by reorganizing instructions within a basic block is limited. So, if you wanted to improve the performance beyond a basic block, then we have to come up with techniques that will work across basic block and that is what we are going to deal today.

(Refer Slide Time: 10:09)

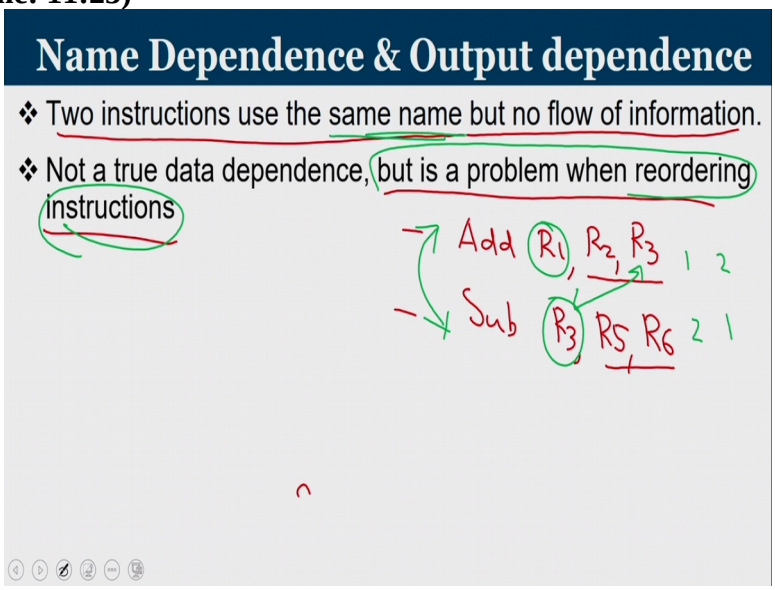


Let us revisit some of the concepts. We are going to learn a technique called loop level parallelism in this lecture. What do you mean by that, we are going to unroll and loop statically or dynamically, and when you unroll a loop there can be data dependency across iterations. And whenever there is a data dependency, we know that there is a possibility of a hazard, 1 data value depends on the next value.

And when there are hazards when you have dependency, dependent instructions cannot be executed parallelly. And pipeline determines if dependence is detected and if it causes a stall or not. So, the idea of pipeline is trying to understand whether there exist a dependence between 2 instructions, if there is a dependence, then I cannot execute those instructions parallelly. If there is no dependence then I can execute them parallelly.

Data dependence conveys upper bound on exploitable instruction level parallelism. When there is a data dependency between a pair of instructions, I have to wait for 1 instruction to produce results, such that I can forward it to the next instruction.

(Refer Slide Time: 11:23)



So data dependence is very important. The other concept is named dependence and output dependence. We have seen this thing when we discussed about WAR and WAW hazards. 2 instructions use the same name, but there is no flow of information that is known as a name dependence. It is not a data dependence. But it becomes a problem when we are going to reorder instruction. So when you execute instructions in a normal fashion, then there would not be any problem. As for as the name is concerned.

Let us now take an example. Consider the case I am going to work with an add instruction.

Add R 1, R 2, R 3

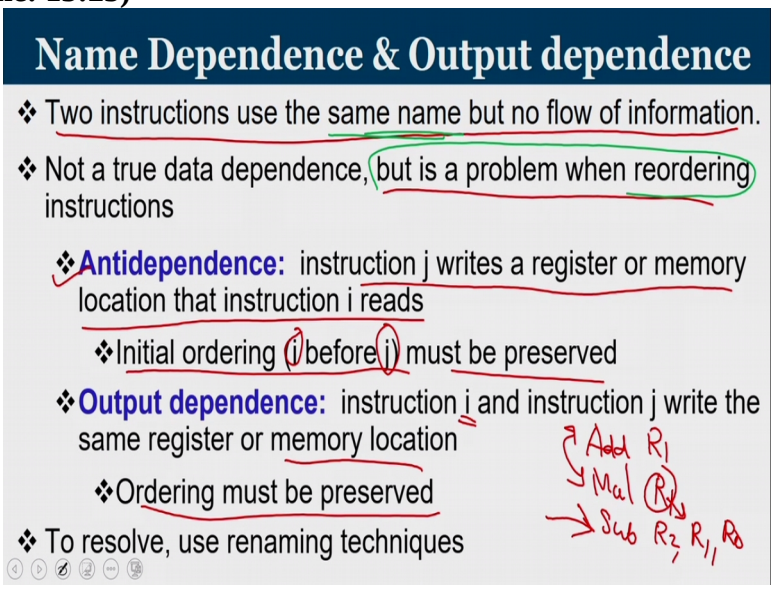
This instruction is going to write its result in R 1. Now consider the case that I am going to perform a subtraction operation on R 3, R 5 and R 6. So, if you look these 2 instructions, there is no data dependency. First instruction operates on data R 2 and R 3. Second

instruction operates on data R 5 and R 6 and the result of the first instruction is returned to R 1 and R 1 is not used.

Now, if you try to reorder this instruction, then subtraction is going to write a result into R 3 and which in turn is used as a source operand by the add instruction. So, if you do ordering, the normal ordering is 1, 2 add will be done first followed by 2, it is not creating any hazard, but if the order is changed. Then it can lead to a WAW hazard. Then it can lead to a WAR hazard.

Such a kind of a dependency is known as a name dependency. So when there is a reordering that is happening, when you reorder instructions, due to the same name that is used, you will be getting a hazard.

(Refer Slide Time: 13:25)

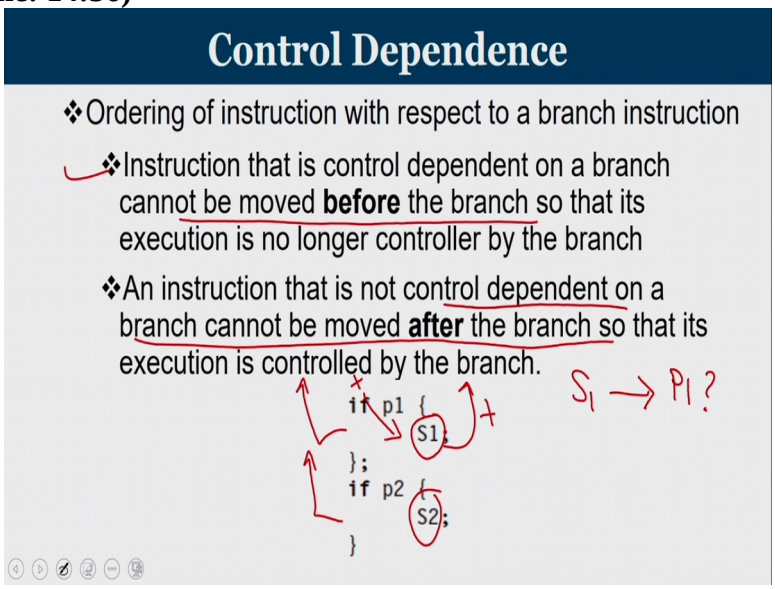


So antidependence means when an instruction j writes or register or memory location, that instruction i reads, that is what we have seen just now. The initial ordering, that means i should happen before j must be preserved. If you try to execute j before I, then this antidependence is going to create a hazard. Similarly output dependence means an instruction i and instruction j write the same register or memory location.

Here also the ordering must be preserved. Meaning let us say I am going to have a first add instruction which is going to write into R 1. Let us say there is a multiplication instruction later, that also going to write into R 1 and thus imagine you have a subtraction operation, which is going to write to R 2 where R 1 is 1 of the source operand. In this case ideally the subtraction operation should make use of the result of multiplication.

But if add and multiplication are going to be doing in different order, then subtraction will actually read a wrong data. So, this is the meaning of output dependence. Instruction i, in this case add and instruction j, in this case multiplication, write onto the same register R 1. So, in this case, even though there is no dependency between them, but if you try to reorder these instructions, then it is going to create a hazard.

(Refer Slide Time: 14:56)



So, the result is basically we use register renaming techniques and that is what we will deal. The third one is known as control dependence. So, we have seen what data dependencies, we have seen what is antidependence and what is output dependence and this is called control dependence. Ordering of instruction with respect to a branch instruction is been dealt by the control dependence.

Instruction that is control dependent on a branch cannot be moved before the branch, so that execution is no longer controlled by the branch and instruction that is not control dependent on a branch cannot be moved after the branch. So, that is execution is now controlled by the branch. So, let us try to understand in this particular fragment of code if p1, let us say it is a condition then write s1. So, s1 is a conditional statement.

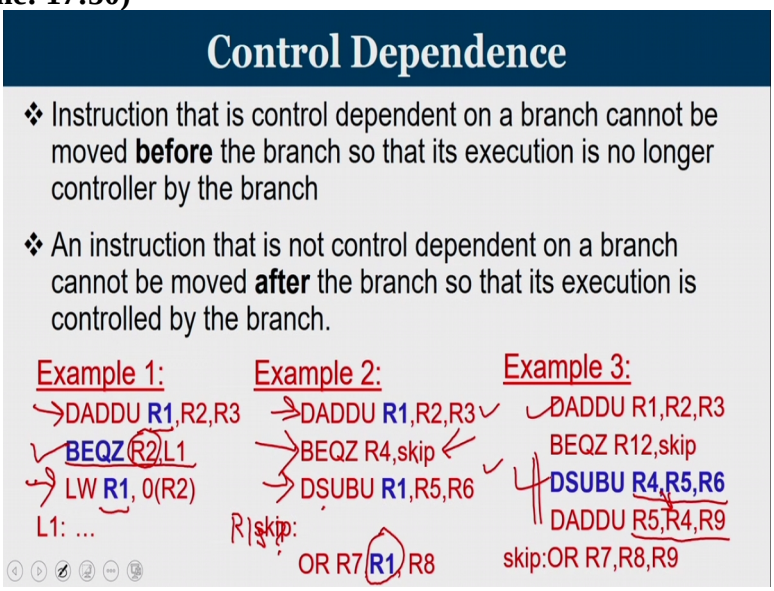
Similarly, if p2 then write s2. Now what this tells is, an instruction is control dependent on a branch. So, here the statement s1, whether the set of statements x1 is executed or not, it depends on what will happen to the condition p1 and instruction that is control dependent on

a branch cannot be moved before the branch. So, the s1, I cannot take it out, because then the moment s1 is taken out of the condition, then s1 is no longer control dependent on p1.
Similarly s2, I cannot take it out of that, then once I take it out then s2 is no longer control dependent on this. So, why we try to study about this control dependence, the whole purpose of compiler level assistance to instruction pipeline is can compiler reorder instruction such that you will get better performance. So, if compiler decide let us let me take s1 and put it above or let me take s2 and put it above then s1and s2 will no longer be dependent on p1 and p2 respectively.

Similarly, an instruction that is not control dependent on a branch cannot be moved after the branch. So that it execution is not control. So, imagine that there are some statements x which is before this if p statement I cannot put the x down the moment they put x inside this if condition then x is now control dependent on p1. Initially x was not control dependent. X was a must execute instruction before the condition check of p1.

So, any moment of x inside this p1s body make x control dependent on p1 that is not required. So control dependency is also there. Anything that is inside a branch condition, we cannot take it, out anything that is outside a branch cannot be taken in as well.

(Refer Slide Time: 17:50)



Instruction that is control dependent on a branch cannot be moved before the branch or instruction that is not control dependent on a branch cannot be moved after the branch. Now, consider this example, you are going to write the result of R2 and R3 and is putting the result

in R1, then you have a branch statement, which will check the value of R2, then you have a load word.

Now, if I am going to put a value, so the value of R1 let some other instruction is going to check the value of R1 at some point later, the value of R1 is actually control dependent on this instruction, the moment is branch is equal to R2, L1. If the value of

$$R2 = 0,$$

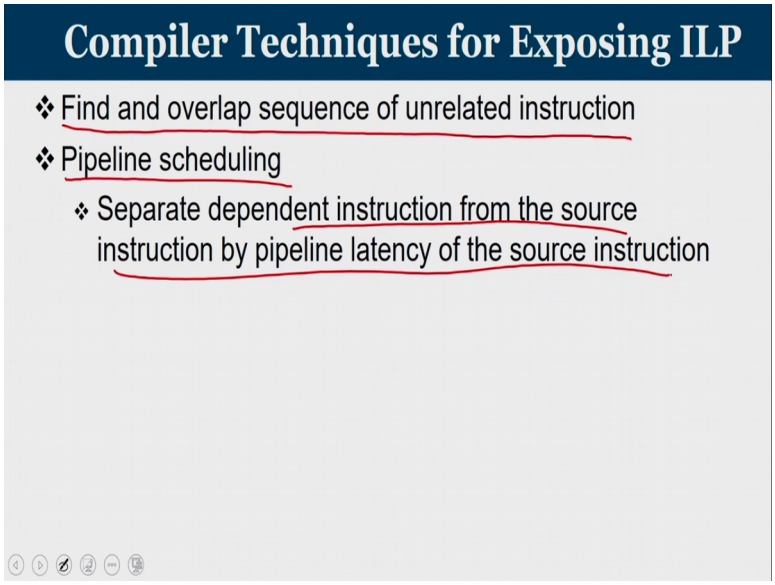
then I am jumping to L1, in that case the R1 value is determined by this DADDU direct add unsigned. So, this DADDU instruction will determine the value of R1, if the branch is taken.

If the branch is not taken, then load word will determine the value of R1. So the value of R1 is control dependent on this branch. Similarly, in this case, you have a branch that is going to happen and the operand in the OR operation. R1 is an operand, the value of R1 is dependent either on DSUBU or on DADDU. So, they are actually dependent on both of these instruction whichever is going to run.

Now based on whether this branch is taken then R1's value is determined by the DADDU instruction. If the branch is not taken, then it would not go to skip it to run and come and execute DSUBU instruction and that will update the value of R1. Let us look into the third example. Here you have a value that is return to R1 and here we have a value that is return to R4 and here we have a value that is return to R5.

We know that once you take the skip then these 2 instructions would not be executed that means the value of R4 and R5, because this value of R4 is dependent on this. So only if this instruction is executed then only its value will vary. So, these are all places wherever you see the blue mark these are all places where control dependence is going to impact.

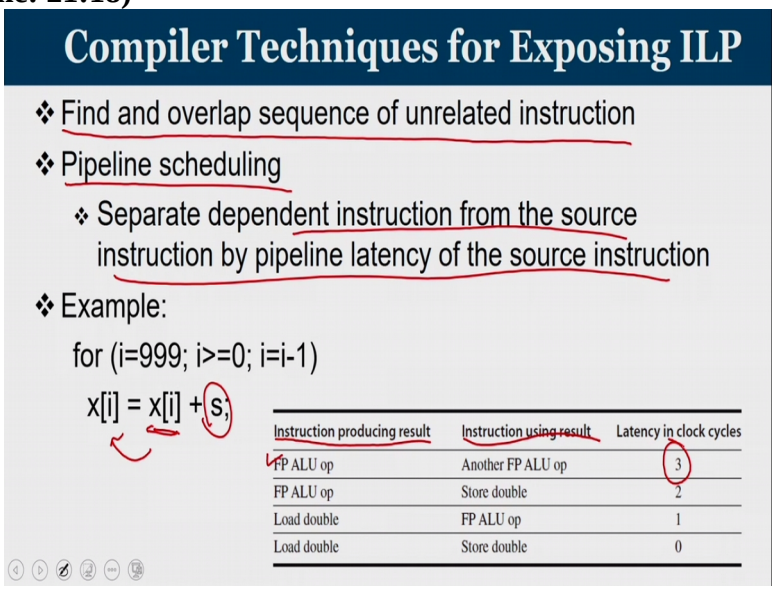
(Refer Slide Time: 20:35)



So, what are the compiler techniques? Compiler basically does find an overlap sequence of unrelated instruction. Compiler tries to see whether there are any unrelated instruction or not. If it find out some unrelated instruction, then it tried to schedule these instructions separate the dependent instruction from the source instruction by pipeline latency of the source instruction.

So if compiler is seeing that certain instructions are dependent on each other, it is a and b, if b is dependent on a, then b and a has to be sufficiently separated, such that by the time b is going to run all dependency with respect to a is resolved.

(Refer Slide Time: 21:18)



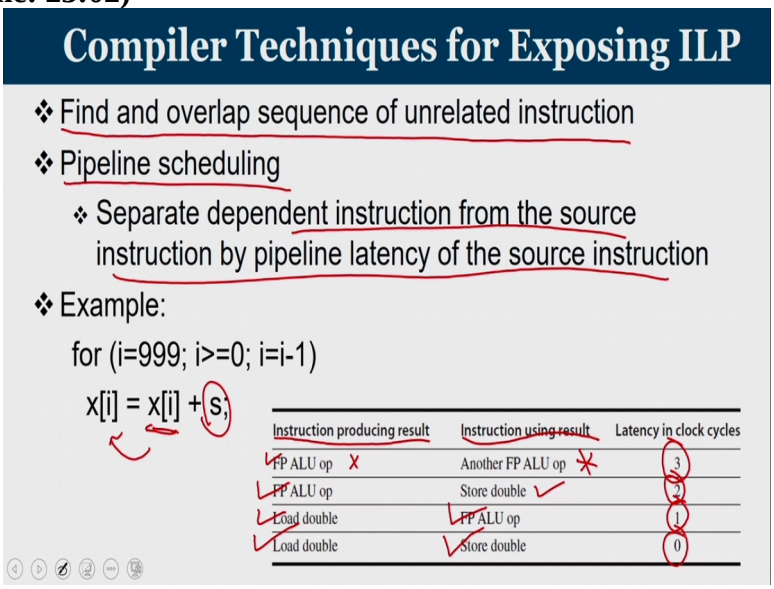
So consider an example, see code that is given, it is a loop that i tried from 999 all the way up to 0, by decrementing the value of i by 1 in every iteration and it is accessing an array. So X[i] is an array whose index is from 0 to 999. We are going to start from the last element.

Take an element, add X scalar value a constant value S to it and store it back in the same location. So, for this piece of line, let us assume that all these values are doubled, each value is going to be stored in 8 bytes.

Now, this table tells that, what is the dependency level if a floating point ALU operation is going to produce the result and another floating point ALU operation is going to use the result, then between them there is a latency of 3 cycles. This is basically your floating point ALU which takes floating point add and subtract. When we discussed the multi cycle pipeline, where in a1, a2, a3 and a4, which represents a 4 stage floating point add operation.

We know that each floating point operation will take 4 cycles in the execution stage. So, if there is a dependency like if I am producing a floating point result, which has to be used by another instruction, then these 2 instructions has to be separated by a minimum of 3 cycles.

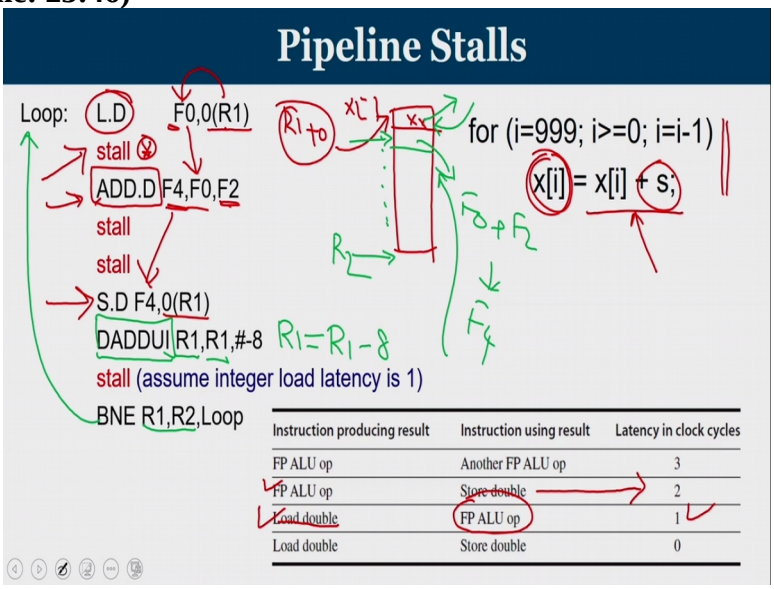
(Refer Slide Time: 23:02)



So the table tells that if a floating point value is producing a result, and I have to use the result, then minimum of 3 cycles. Similarly floating point ALU is going to produce a result which I have to store it immediately, then there should be a delay of 2 cycles between them. This is a standard one if you load the value and if we wanted to use the value in floating point ALU, there is a delay of 1 cycle that is latency. Similarly, when you load the value and immediately we are going to store, then by operand forwarding there is no delay.

So it is 0 cycle delay. So let us try to understand we are going to see some sample set of code and dependency how the compiler can help.

(Refer Slide Time: 23:46)



So this is the instruction that we are going to see a sequence of instruction inside a loop, the loop is going to operate from 999 all the way to 0. Take an element, add a constant value to it and then store it back in the same location. And this has to be repeated. Let us see how the MIPS risk code for this one will be. You are going to store the first element. So 0 of R1 will point to the first element.

Let us say this is your array, the array X, so R1 + 0, we will actually point to this location. So from the location, copy the contents, these contents you are getting, you are going to copy to a register called as F0. Just to make you understand the notations that has been used, if the register is starting with an F, it is a double register, it is a floating point register, if the register is starting with R it is an integer register.

So 0 and R1, they are basically used to compute the address, and with the address you go to the location, contents are transferred into a register F0. So you are loading a value. Now, the loaded value, I have to add a constant S to it. Let us see, the value of S is stored in register F2. So whatever is loaded from memory that is F0, I am going to add F2 to it and store the result in F1. So it is a floating point add operation.

We know that whenever we have a load operation, which is producing a result in F0 and the result has to be used by another floating point operation, then there is a stall of 1, that is what this stall represents, that means a load and an add instruction, which make use of the result of load has to be separated by 1 instruction or there will be 1 stall between them. Similarly, we

are going to have a ALU operation and the result of the ALU operation have to store there will be 2 stalls.

So now you take the value in F0 add with F2, the result is stored in F4. Now, I have performed

$$X[i] + S$$

Now the result I have to store back in the same location. Storing is done with the help of a store and 0(R1) will tell the same address and the new value is available in F4. So

store F4, 0(R1)

So essentially what are we trying to do using this load instruction you get the value and then you add, and the new store the value. This is what is happening now.

So with this, the first number is taken and the updated value is stored back. Now I have to go to the next location. Since, it is a loop that is counting backward. The next instruction is obtained by I am changing the value of R1.

$$R1 = R1 – 8$$

So this instructions meaning is

$$R1 = R1 – 8$$

Add unsigned immediate value, that is the meaning of it. And then I check whether the value of

$$R1 = R2$$

So

$$R1 = R2$$

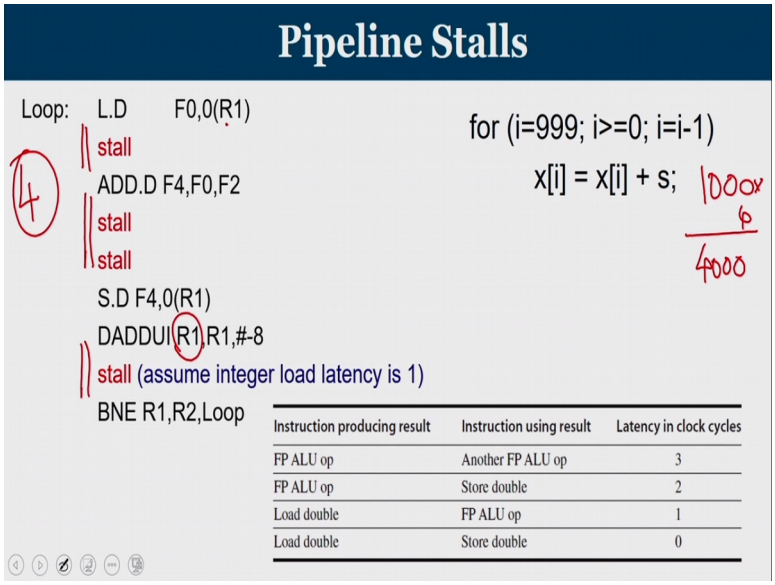
means R1 value is decremented every time and then see whether it reached R2 or not.

So if the value of

if R1 != R2,

I am going to repeat the loop. So now the new value of R1, I perform 0 for one that will point to the next location, take the value to F0, add F2 to it and produce the result in F4 and then store it back. Again subtract R1. So, if you do the loop, then the R1 value is decremented every iteration and with the new value of R1, the effective address of the next element of array is obtained. Now if you look at this particular loop.

(Refer Slide Time: 28:00)

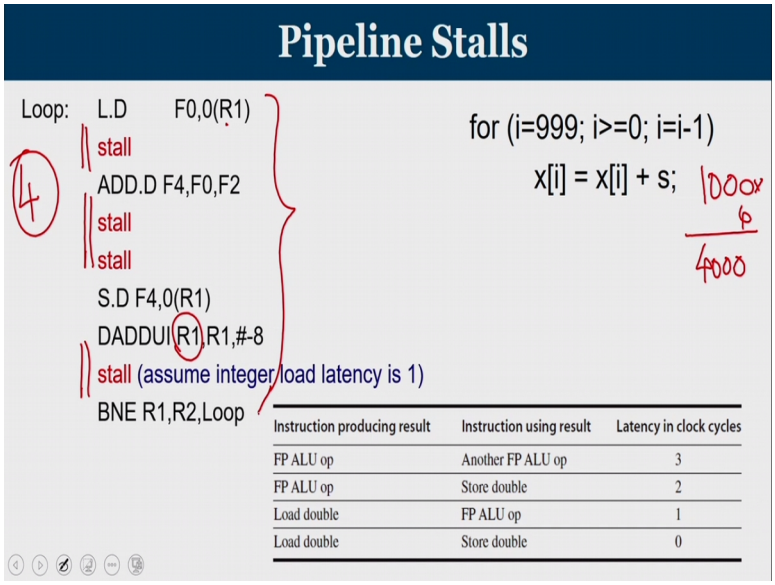


If you look at the loop, you can see that the load and add instruction of separated by a stall, add and store instruction are separated by a stall and here also because the value is updated in R1, there is a stall. So we have 4 stalls in 1 iteration of the loop and this loop is going to run for 1000 times. So total number of stalls that I am going to get is 4000 stalls. So, because of the dependency between instruction, each of the iteration is going to create 4 stalls and we are going to run the loop for 1000 times that makes it 4000 stalls.

Imagine that this particular set of stalls are inside a nested for loop then the number of stalls are going to be even high. If it is a triple nested for loop then because of the stalls CPU is actually wasting closed minutes.If the program is going to be delayed by few minutes, and if it is a critical program, which runs on embedded systems or specific, mission specific applications, then that is a big time. Can we get rid of these stalls?

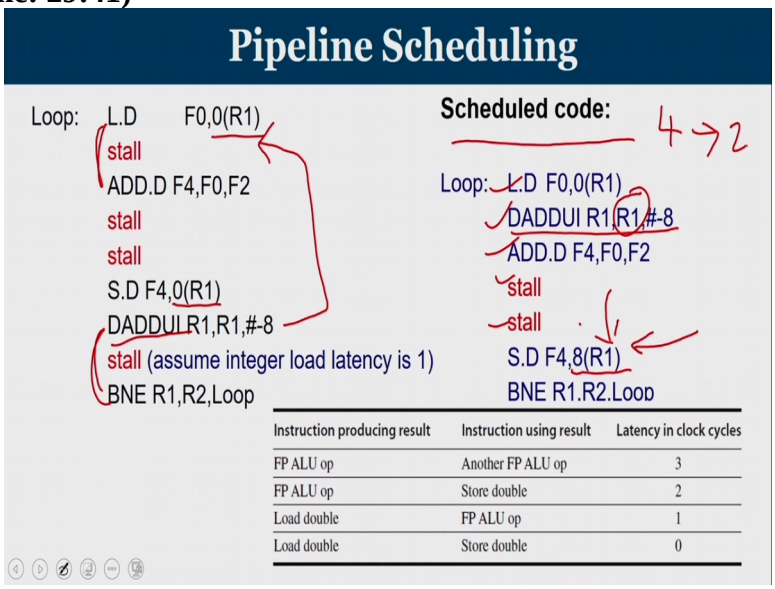
If you look at general operand forwarding technique, what we have seen there is a lot of limitation in these set of instructions, because essentially, we have only 4 or 5 instruction.

(Refer Slide Time: 29:27)



And you have to understand that these are part of 1 basic block up to this, this is part of 1 basic block. So, you cannot do much between this because a number of instruction is limited.

(Refer Slide Time: 29:41)



So, this is what we have now. Now, let us see how compiler can help in this. So compiler sees that this load and add anyway has to be separated by one instruction. There is a stall between them. So can I find out some other instruction compiler identify this as an independent instruction and put it there. So, that is the first step. Now, by the time the add is running, there are sufficiently separated by one clock cycle.

And rather than wasting that one clock cycle, during that cycle, this instruction is fetched and executed. But between add and store still we have these 2 stalls. So you are now scheduling the code and find out an instruction for rearrangement of this instruction. Now we have to

understand by this time the value of R1 is now updated.So when I store the value I have to compensate it.

Previously, both load was using 0 of R1 and store was also using 0 of R1. Both are same. But now in this case, since the value of R1 is updated in order to get the same one, because

$$R1=R1 - 8$$

to compensate that I have to use an 8 as the index. And one more thing is since these 2 have to be separated by one instruction. Once if this, that UI is going up, they are already separated.

So, from 4 stalls that we had initially, we are moving into 2 stalls, that is the advantage that we get. So, since the number of instructions within the block is limited, any kind of rearrangement within a block also will give you minimal robots. So this technique of rearranging instruction is known as scheduling.

Now, let us see what is loop unrolling. So, this is the scheduled code that we have. Now in loop unrolling, so what are the essentially trying to do in the next iteration, I recompute the address of the location. So R1 value is updated. And when I visit the load instruction again, it is a new value of R1 and I am going to load and store. Now look at this pair of instructions. You are going to have a load, add and store. So take the value that has been done in load, add a constant value and then store.

So what I mentioned previously, we have different elements in the array, take an element that is called load operation and then perform the add operation and then you store it back, that is called a store operation. So basically this load, add and store is the heart of this code. So load the first value, add and then store. So you get the result in F4 and then you are storing the value in F4 into the same address.

Now let us see previously we had statements like DADDU. These statements were trying to manipulate the value of R1. This is a manipulation of R1.

$$R1 = R1 – 8$$

Now I am using

$$\text{load F6} – 8 \text{ (R1)}$$

I am not using this instruction, rather than that, in the current loop itself, I am finding out what is the next element. So and then that F6 to F2, you are getting the new result there.

So, these load, add and store what is marked here, that takes care of the next value in the array, which would have normally been fetched only in the next iteration. Similarly, these 3 that I am taking a value in F10 then F10 is added with F2 to get result in F12 and F12 value is stored. This will take care of the next element in the array. Similarly, if you look into that, you are going for -24, so loading the other value get the result in F16 and store the value that is in 16.

So, to summarize what we discussed, we are trying to drop these 2 instructions, which were part of our normal scheduled code and I perform load, add and store for 3 set of numbers. Load, add and store, load, add and store, load, add and store for 3 more iterations. And then at the end, I perform

$$R1 = R1 – 32$$

And then I am comparing. So one thing is I am removing these branch instructions, which can create a control hazard and trying to see what are the potential values that I should get.

If I could have run the subsequent iteration, so I am not going for next iteration. By standing in the current iteration trying to get the next values by -8 of R1 , -16 of R1, -24 of R1. So, this -16 of R1, -24 of R1 and -8 of R1 will give you the adjacent numbers.So, you unroll it 4 times and then you do a single subtraction of

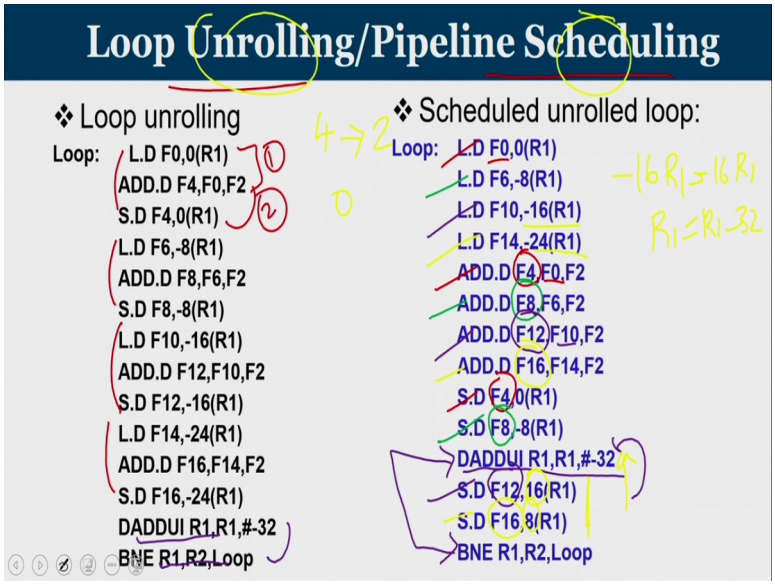
$$R1 = R1 – 32$$

Remember, in the normal fashion, this will create 4 stalls, this also will create 4 stalls.

Now, you have essentially this is, So this load and add will be having one stall between them and add and store will have the 2 stalls. So, that makes it total of 3 stalls between them. 3 stalls here, 3 stalls here and 3 stalls there and there is going to be one stall between them. So, the number of stalls have significantly come down since we are eliminating this branch and this direct add instruction.

Can you do anything better, so loop unrolling still we have stalls, only thing is we have removed the branch instruction which can potentially create a hazard. Now the next stage is, here we are creating lot many registers, previously our code worked with a F0, F4 and F2. Now we are using F6, F8, F10, F12, F16 sort of registers are been used.

(Refer Slide Time: 36:24)



Now can we club 2 techniques. One is scheduling reordering instruction and unrolling that is called loop unrolling and pipeline scheduling. This is what we have seen by unrolling. Now, because there is dependency between them and they are going to create hazards, we put up all the loads together. I wanted to separate this load and add which is creating one cycle delay and the add and store which is creating 2 cycles delay, my load, add and store are now sufficiently separated.

So, what do you see in this red colour these are corresponding loads. See the one that is making use of F0 producing result in F4 and that F4. So add and store is sufficiently separated. Similarly, if you take up the next set, this is a value that is loading to F6. F6 value is now added to get the result in F8 and the F8 value is be now stored. So the green will show their operations on the second set. And this violet will show operations on the third set.

So you load a value in to F10. And then you are going to get the result in F12 from F10 and we are going to store. But in between you have the subtraction instruction. Why I am taking off subtraction instruction is normally these 2 we have a 1 cycle stall between them. So if I separate this R1 subtraction instruction directly add -32 and the jump instruction, now they are sufficiently separated.So that stall also we are eliminating.

And the last pair of values which is been marked with this yellow colour, that will take the value in F14 and get the result in F16, and you are going to store the F16. So, any store that is happening after the updation of

$$R1 = R1 - 32$$

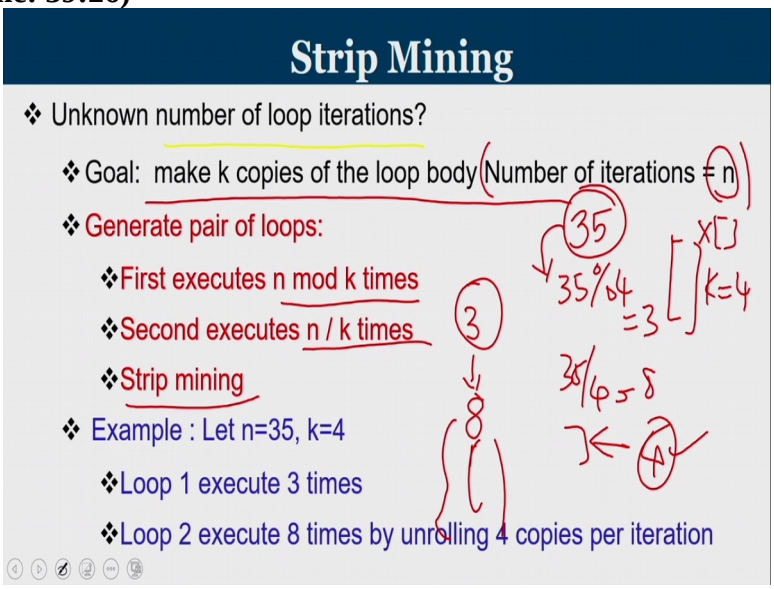
which is appropriately adjusted, so that it will write into the same location as that of load. So this -16 of R1 is same as 16 of R1 here, because in between we have an

$$R1 = R1 - 32.$$

So this is how you are able to when you look at here, there are no stalls at all. So when we had 4 stalls, that 4 stalls we reduced to 2 stalls by scheduling. Now we are reducing to 0 stalls with a combination of loop unrolling and pipeline scheduling. So how will you do this? So here, you know that the number of time the loop has to run is thousand times and we divide thousand into units of 4.

So the loop has now reduced to 250 iteration, but each iteration is taking care of 4 values of the array. What if the number of iterations of the loop is not a multiple of 4. And that is what we do with the help of strip mining.

(Refer Slide Time: 39:26)



So we do not know the number of loop iterations. So what is the goal, the goal is to make k copies of the loop body where the number of iterations is going to be n. So we have to generate pair of loop first executes n mod k times. Second, execute n by k times and this technique is known as strip mining. So think of a case that you have a loop with 35 numbers. So and then, in each of the loop, let us say I am going to process 4 values of the array X.

So the value of k,

$$k = 4.$$

So what I do is out of this 35 I perform

35 mod 4

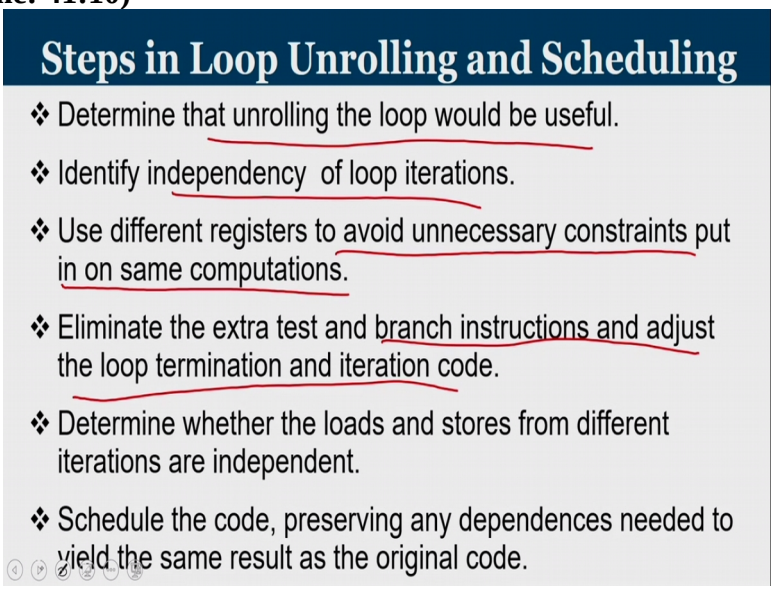
that will give you 3 and

35/ 4 =8

that is going to give you 8. So there is a loop, which will take care of only the 3 numbers separately take a number and perform the operation and remainingly I run the loop only for 8 times, where in each time a loop is going to handle 4 numbers. So first iteration will take care of 4 numbers, second iteration will take care of 4 numbers like that they have only 8 iteration rather than 35 iterations.

Now I have a loop which will run 3 times and a loop which will run 8 times a second loop will take care of 4 numbers. So the unrolling happens here. This technique is known as strip mining. Let us example, what is been given. Loop 1 is execute 3 times, loop 2, it execute 8 times by unrolling 4 copies per iteration. The strip mining is a technique that has been generally used in order for unrolling where the number of iterations of the loop is not clear.

(Refer Slide Time: 41:10)

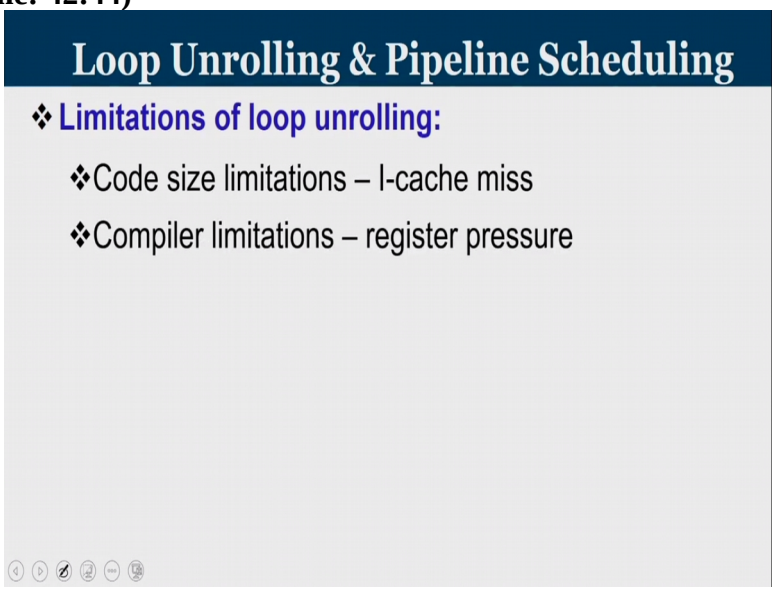


So what are the steps in loop unrolling and scheduling, you have to determine that unrolling the loop would be useful, identify independecy of loop iterations and then use different registers to avoid unnecessary constraints that are put on the same computers. That is what we have seen, when you take the first number to take the number to F0, next number you are taking into F6 like that, Eliminate extra test and branch instructions wherever possible to adjust the loop termination and iteration code.

And determine whether the loads and stores from different iterations are independent or not. And if all these conditions hold properly, then scheduled code preserving any dependency is needed to yield the same result as that of the original code. Let us see what are the limitations of loop unrolling. Loop unrolling is a very good technique you unroll, once you unroll you are getting many instructions and essentially what you do in loop unrolling is, your size of the basic block is increase.

You have a branch statement only in the beginning and then at the end, all other branch statements are you removing and fusing all these straight line sequence instructions together so that your basic block is big. Once the basic block is big, you have so many instructions potential instructions are there, so that you can perform this reorganization. So once you rearrange, you can find out some instructions which are dependent between them and try to rearrange, keep them close, keep them far, and then try to reduce the stalls.

(Refer Slide Time: 42:44)



But when you look into the limitations of loop unrolling, there are 2 limitations. First 1, when you have a code, that is a 3 line code inside a loop that says 1 or 2 loops statements and this 3 lines so it is hardly 4 or 5 lines of code, when you unroll the program size increases even though the time is same, the size of the program increases and when the size of the program increases it requires more space in the instruction cache and that leads to i cache misses.

The second problem with respect to loop unrolling was we are making use of more number of registers. So, previously our example program we work with an F0 where value is loaded, F2 which contains the scalar value and F4 the result is stored. So with 3 registers I could

manage. Now, we unroll it 4 times, then roughly that many number of registers are there and then that is what is known as register pressure.

So, with that, we are going to conclude this lecture. Let us try to summarize what we learned in this lecture. We were trying to find out how can you improve instruction level parallelism over the past few lectures, we were trying to see about operand forwarding techniques, branch prediction techniques to take care of data hazards control hazards and all. Now in today's lecture, we were trying to see how if compiler knows the knowhow of the architecture.

Compiler knows these instructions, if they come together there should be minimum of these many stalls between them. If you pass out this information to a compiler, then compiler can effectively rearrange the instructions, if at all it is possible that is called scheduling, rearrange the instruction and then pass the modified set of instructions to the hardware. So that hardware up and running will not see any dependent instructions coming close by.

But there are cases wherein, the number of instructions that is available inside the basic block is limited. So we are trying to increase the size of basic block and that is done with the help of loop unrolling. So unroll the loop, increase the basic block and then rearrange instruction we can reduce the hazards and strip mining techniques are used whenever we cannot perfectly divide the number of iterations into the number of loop and rolling expansions. So with that we conclude. Thank you.