Computer Architecture

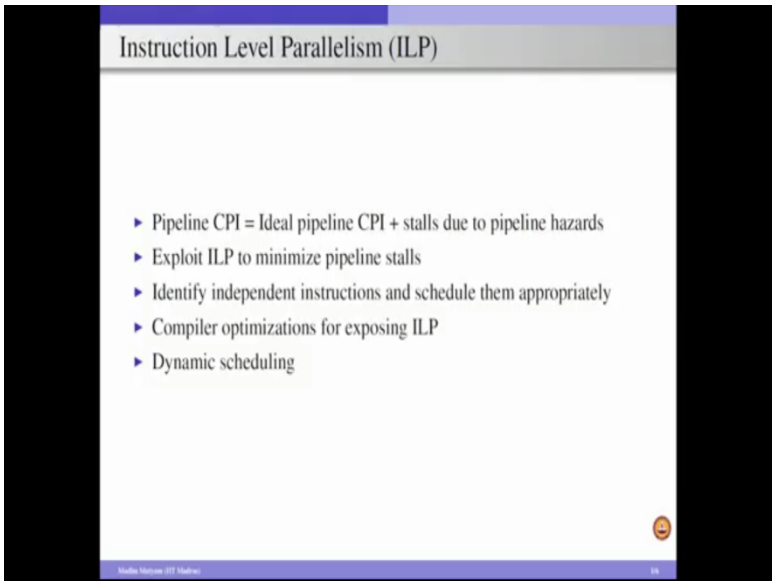
Prof. Madhu Mutyam
Department of Computer Science And Engineering
Indian Institute of Technology, Madras

Module – 06
Lecture – 19
Instruction Dependencies

So, in the last module we discussed the limitations of scalar pipeline design and suggested super scalar pipeline design. In super scalar pipeline design so, we can execute instructions in out of order fashion. In order to execute the instructions in out of order fashion, so at the dispatch stage we will select the instructions which are independent and if the functional unit is available we can issue these instructions to the functional units. And as a result the instructions can be executed in an out of order fashion. And all these executed instructions will be stored in the buffer, the write back buffer, and from there we will commit the instructions in the program order.

But in order to execute the instructions in out of order fashion, so first of all, we have to know which instructions are independent. And only the independent instructions can be proceeded further before the leading instructions. So, for that first we have to identify which instructions are independent and which instructions are dependent. So, effectively so this module and the next couple of modules we are going to look at techniques which exploit the instruction level parallelization. And as the starting point of this instruction level exploitation we are going to discuss the dependencies among the instructions. And in this module we are going to look at what type of dependencies exists between instructions in a program.

(Refer Slide Time: 01:50)



Instruction Level Parallelism (ILP)

- Pipeline CPI = Ideal pipeline CPI + stalls due to pipeline hazards
- Exploit ILP to minimize pipeline stalls
- Identify independent instructions and schedule them appropriately
- Compiler optimizations for exposing ILP
- Dynamic scheduling

So, we know that a CPI of a pipeline design is,

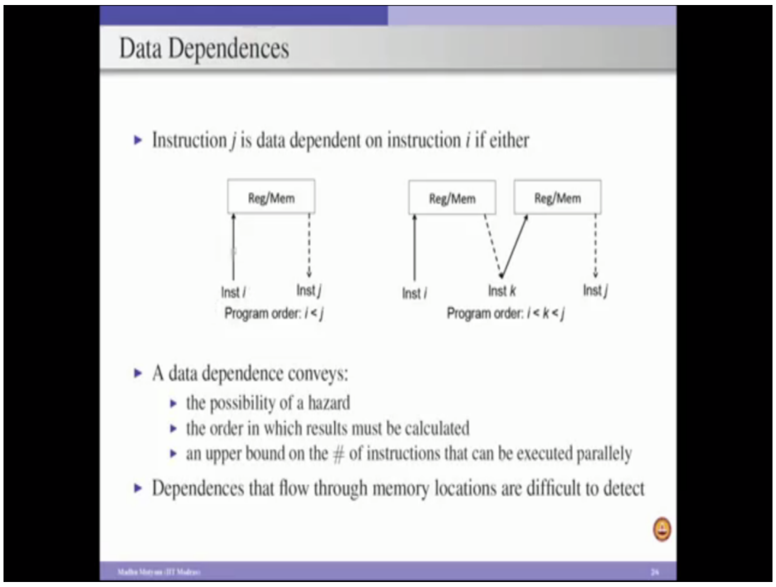
$$Pipeline\,CPI = Ideal\ pipeline\,CPI + Stalls\ due\ \text{to}\ pipeline\ hazards$$

So, in an ideal pipeline design the CPI will be equal to 1 and so if there are stalls because of pipeline hazards then our, the overall CPI will be degraded significantly. So, because of the stalls in the pipeline our overall CPI will increase and that results in performance degradation. So, as a result we need to come up with techniques to minimize the stalls due to pipeline hazards. And as part of this unit we are going to look at techniques which exploit the instruction level parallelization to minimize the stalls associated with the pipeline hazards. So, we consider several techniques as part of this instructional parallelization unit, to minimize the pipeline stalls.

And so, the major thing involved in this instruction level parallelization mechanism is to identify the dependencies among the instructions. So, identify independent instructions and schedule them appropriately, so that the pipeline stalls can be minimized. But for identifying the independent instructions we can take the help of the software as well as the hardware. In the case of software, compiler can identify independent instructions in the program and schedule the instructions statically. Or in the case of hardware we can use a dynamic scheduling. So, at the run time we will identify independent instructions and then issue these instructions to the functional units appropriately to minimize the overall pipeline stalls so that the performance can be improved.

So, in this module we are going to concentrate on identifying the dependencies among the instructions in a program. So, we first start with the data dependencies. So, for example, if we have two instructions i and j,

(Refer Slide Time: 04:09)



Data Dependences

- Instruction *j* is data dependent on instruction *i* if either

Reg/Mem
Inst *i*
Inst *j*
Program order: *i* < *j*
Reg/Mem
Reg/Mem
Inst *i*
Inst *k*
Inst *j*
Program order: *i* < *k* < *j*

- A data dependence conveys:
  - the possibility of a hazard
  - the order in which results must be calculated
  - an upper bound on the # of instructions that can be executed parallely
- Dependences that flow through memory locations are difficult to detect

and where j is data dependent on instruction i, such that in the program order i comes first and j comes later. In this diagram we can clearly see instruction i is writing a value to a register or a memory and instruction j is reading the value from the register or the memory. So, if that is the scenario then instruction j is said to be data dependent on instruction i. And this data dependence can also happen in a transitive way, that is instruction i is writing the value to a register which is consumed by instruction k and instruction k writes value to another register or a memory and instruction j is consuming.

So, effectively now we can say instruction j is data dependent on i because this is chain dependence. So, k is dependent on i and j is dependent on k in a transitive way. So, instruction j is dependent on i and this dependence is the data dependence. So, effectively here the actual data flow happens between the producer and the consumer. So, here instruction i is the producer and instruction j is the consumer. So, effectively this data dependence is the true dependence, where the actual data flow happens between two instructions where one is a producer the other one is a consumer.

So, once two instructions are data dependent then based on this data dependence we can know that there is a possibility of a hazard and it also conveys that the order in which the results must be calculated and also it gives an upper bound on the number of instructions that can be executed parallely. So, for example, in this case since instruction j is going to need the data supplied by instruction i. So, if we execute instruction j before instruction i then there is a data hazard. And as a result instruction j is going to take the old value and using that it computes and overall computation will go wrong.

So, as a result when there is a data dependence between two instructions then the instruction which is going to require the data produced by the instruction which is supplying the data should not be proceeded before the instruction which is going to supply the data. In other words when there is data dependence between instruction i and j, where j is going to consume the value supplied by instruction i, so instruction j should not be executed before instruction i. So, that indicates that there is a possibility of a hazard if we are going to reorder these instructions.

And also once there is a data dependence a between two instructions, we know that we have to execute instruction i first before instruction j because there is a data dependence between instruction i and j. So, that is nothing but the second point. So, the data dependence conveys that the order in which the results must be calculated. And finally, once there is a data dependence between instruction i and j, even when we have multiple functional units in our super scalar pipeline design we cannot execute these two instruction simultaneously.

At any point of time we can execute only one instruction. So, that also says that because of the data dependence we will have a limit on the number of instructions that can be executed parallely. In this example we can clearly see instruction i is writing to a register or a memory and instruction j is reading that. So, that means like this true dependency can be there on using a single register, otherwise like instruction i can be writing to a register and from that register instruction j is reading or instruction i is writing to a memory location from which instruction j is reading the value. In other words, true dependence can happen because there is a read and write operation on a register or onto a memory location.

If the true dependence happens because of the registers, register read and write, we can easily identify the dependence because the number of registers we use in our ISA will be limited. As a result we can easily name each of these registers and because of that it is easy to identify the data dependence between instructions if they are using the same registers for read and write operations. But on the other hand if the data dependence happens because of writing to a memory location and reading from a memory, same memory location, then it is very difficult to identify the data dependence.

So, that is the reason why the dependence is that flow through memory locations are difficult to detect. To clarify this point, let us consider an example, where instruction i is writing to a memory location which is specified in the instruction as 10(R1). So, it is effectively, the effective address of the memory location is the contents of R1 plus 10. And there may be another instruction, instruction j which is trying to read from a memory location which is specified in the instruction as 20(R2). In other words effective memory address location used by this instruction j is the contents of R2+20.

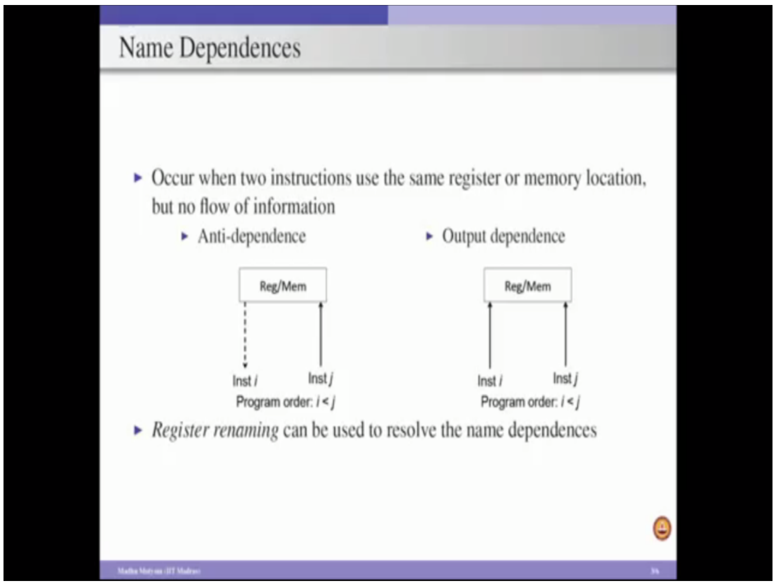
Now in this case, it may so happen that,

$$contents\ of\ R1+10=contents\ of\ R2+20$$

but it is very difficult to detect this equality unless we calculate the effective address calculation for both the things. Because of that when the data dependencies involved in the effective address calculations, then it is very difficult to identify the dependencies. And also it takes significant amount of time to calculate this effective address and as a result we incur the stalls in the pipeline. So, as a result compared to the data dependencies that involve in registers, data dependencies that involve memory locations will be hard to detect or it is going to create the stalls in the pipeline.

Whereas, in the case of data dependencies that involve register locations, for example, consider a scenario where instruction i is writing to register R1 and instruction j is reading from register R1. Here in these two instructions because in one instruction R1 is the destination register and in the other instruction R1 is a source instruction. We can easily identify the dependencies and as a result we can detect the data dependencies easily. So, after these data dependencies we have name dependencies.

(Refer Slide Time: 11:41)


Name Dependences

- Occur when two instructions use the same register or memory location, but no flow of information
  - Anti-dependence
  - Output dependence

Reg/Mem
Inst i
Inst j
Program order: i < j
Reg/Mem
Inst i
Inst j
Program order: i < j

- *Register renaming* can be used to resolve the name dependences

So, name dependencies occur when two instructions use the same register or memory location, but there is no flow of information. To clarify this point we will consider an example. So, here in this case, instruction i is reading a value from register or a memory location and instruction j is writing a value or data to a register or memory. And here we assume that instruction i is preceding the instruction j. So, in the program order instruction i is coming first followed by instruction j. So, now because the instruction i is coming first and when it is reading a data from a register or a memory, it is not going to create any problem with respect to the following instruction which is instruction j writing to the same memory location or a register.

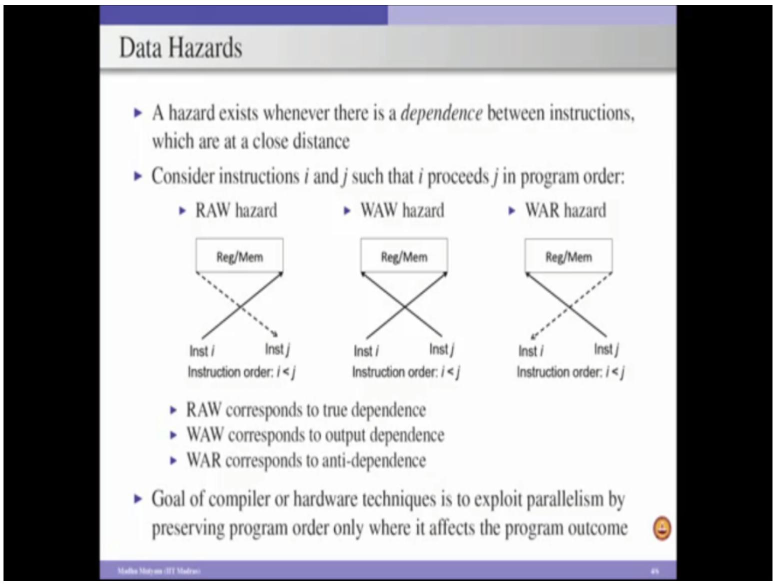
So, this is actually called as anti-dependence. So, there is no flow of information between producer and consumer. Here consumer is instruction i producer is instruction j, but instruction i is coming before instruction j. In other words consumer is coming before the producer. So, as a result there is no flow of information and this is called as the anti dependence. And the dependence here is mainly because these two instructions are using the same register or memory location. And the other name dependency is output dependence.

So, here again consider instruction i which is coming before instruction j in the program order and both these instructions are writing different values to a register or a memory location to the same register or a memory location. So, once that happens, now if instruction i completes its execution first and instruction j is completing afterwards, then the final value that is stored in register is the value written by instruction j. On the other hand if instruction j is executing first and instruction i is executing later, then the final value that is stored in the register will be the value written by instruction i, but in this case also again there is no flow of information between these two instructions and in fact these two instructions are producers.

There is no consumer in these two instructions. So, effectively name dependencies are false data dependences because there is no data flow between producer and consumer, but because they are using the same register or a memory location. So, as a result there is a dependency and we have to eliminate this dependency in our program, so that we can schedule these instructions in an out of order fashion to exploit instruction level parallelization. So, using the register renaming we can eliminate anti dependence and output dependencies or otherwise the name dependencies among instructions.

And as part of this unit we are going to discuss register renaming concept and by the way this register renaming is used only for eliminating the name dependencies between instructions which are using the same register, it is not for the memory location. In the fundamentals of pipelining unit we discussed data hazards informally, now we are going to discuss the data hazards more formally in this module. So, we know that the data hazards happen because of the dependencies between the instructions.

(Refer Slide Time: 15:19)


Data Hazards

- A hazard exists whenever there is a *dependence* between instructions, which are at a close distance
- Consider instructions *i* and *j* such that *i* proceeds *j* in program order:
  - RAW hazard
  - WAW hazard
  - WAR hazard

Reg/Mem
Inst *i*
Inst *j*
Instruction order: *i* < *j*
Reg/Mem
Inst *i*
Inst *j*
Instruction order: *i* < *j*
Reg/Mem
Inst *i*
Inst *j*
Instruction order: *i* < *j*

  - RAW corresponds to true dependence
  - WAW corresponds to output dependence
  - WAR corresponds to anti-dependence
- Goal of compiler or hardware techniques is to exploit parallelism by preserving program order only where it affects the program outcome

So, hazard exists whenever there is dependence between instructions which are at a close distance. When I say close distance because we are dealing with k-stage pipeline, if two instructions are within a distance of the number of pipeline stages, then and if the instructions have some dependence then there may be a hazard. And if two instructions are separated by a gap which is more than the number of pipeline stages then we cannot get any hazard because of the dependences because by the time the dependent instruction comes the producer instruction completes its execution. So, there would not be any hazard because of this dependence.

So, consider two instructions i and j such that i comes before j in the program order and because of this dependencies we can have RAW hazard, this is called as "Read after Write" hazard. So, in this particular scenario, instruction i is coming before instruction j and instruction i is writing to a register or a memory location whereas instruction j is reading from the same memory location or the register. Now, if instruction j proceeds before instruction i because of our out of order execution then it is going to create a hazard, that hazard is called as read after write.

Actually this read has to be performed after the write is completed, but when we reorder these two instructions. So, instruction j can read the value from a register or a memory location. As a result it reads an old value from the register or memory location and that is going to create wrong computation. So, whenever we have such type of hazards, read after write hazards then we should not reorder these two instructions. Effectively we should not execute instruction j before instruction i, if there is read after write hazards between instructions i and j and instruction i is proceeded instruction j in the program order.

And the second type of hazards is "Write after Write" hazards. So, in this scenario again instruction i coming before instruction j in the program order and both these instructions are writing to a memory location or a register. Now, in this scenario if we are going to execute instruction j before instruction i, finally, the register is going to store the value written by instruction i, but the program may be requiring the value to be written by instruction j in the register. So, in that case then there is a "write after write" hazard. According to the program register at the end of these two operations should contain the value written by instruction j, but if you are executing instruction j before instruction i, we are going to have the value written by instruction i in the register and that is going to create "write after write" hazard.

And the third hazard is "write after read" hazards. In this case so instruction j is writing to a register before instruction i is reading from the register and instruction i is actually proceeded instruction j in the program order. So, according to the program instruction i should read the old value and after that instruction j should write a new value to the memory location or a register, but if you are executing instruction j before instruction i. So, effectively this is going to create “write after read” hazard.

And the fourth case is because we considered here read and write operations between two instructions. So, effectively we can have four scenarios. One is read after write, write after write, write after read and the last case is read after read. But read after read is not going to create any hazards because you are just the value from a register and it does not matter whether we are reading for the first instruction or the second instruction because the read operation is not going to change the value that is stored in a register.

So, as a result there would not be any hazards because of read after read. So, among all these four scenarios, only the read after read is not going to create any hazard, but all the other three can create hazards. And effectively when we are planning to reorder our instructions to exploit instruction level parallelization we have to take care of these hazards. And out of these three hazards, read after write hazards is actually a true dependence because there is an actual data flow happens between the instructions which have this read after write hazards. And in the case of write after write hazard, this hazard happens because these two instructions i and j are actually using the same register or the memory location.

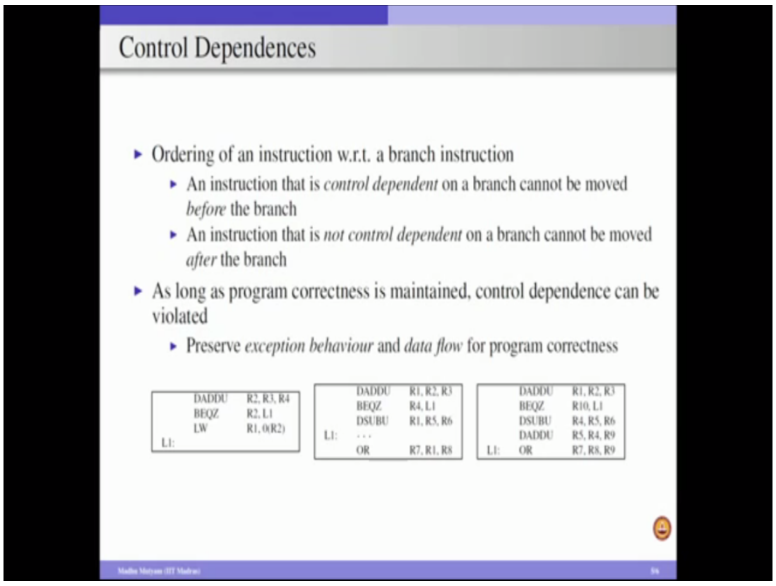
In other words, instructions i and j are actually using the same output register because they are writing to a register or a memory location. So, in other words, there is output dependence and this is a part of name dependence. And similarly, the write after read hazards is corresponding to anti dependence because here actually there is no flow of information similar to the write after write hazard and this is also part of the name dependency between the instructions. In other words write after read and write after write hazards are corresponding to name dependencies and read after write hazard is corresponding to the true dependence or data dependence.

So, in the case of name dependencies as we mentioned earlier, if we apply register renaming concept then we can eliminate this name dependence as a result we can eliminate these hazards, such as like write after write hazards and write after read hazards. The goal of a compiler or the hardware techniques is to exploit parallelization by reordering the instructions where ever there is a possibility, but at the same time preserve the program order in scenarios where reordering the instructions is going to affect the outcome of the program.

So, as long as the program outcome is not affected by reordering the instructions, our compiler or hardware techniques can reorder the instructions to exploit instruction level parallelization so that we can improve the overall performance. So, in summary without affecting the program order, rearrange the instructions in our program to improve the instruction level parallelization. So that our underlying hardware that is Superscalar pipeline design or super scalar processor design can exploit these independent instructions and execute them in out of order fashion and improve the overall performance.

So, we discussed the data dependencies, we discussed the name dependencies and then we discussed the hazards associated with these dependencies such as read after write hazards, write after write hazards and write after read hazards, now we will move onto the control dependencies. So, control dependencies happen because of the dependence between branch instruction and the following instructions.

(Refer Slide Time: 23:06)



Control Dependences

- Ordering of an instruction w.r.t. a branch instruction
  - An instruction that is *control dependent* on a branch cannot be moved *before* the branch
  - An instruction that is *not control dependent* on a branch cannot be moved *after* the branch
- As long as program correctness is maintained, control dependence can be violated
  - Preserve *exception behaviour* and *data flow* for program correctness

```
        DADDU   R2, R3, R4
        BEQZ    R2, L1
        LW      R1, 0(R2)
L1:
```

```
        DADDU   R1, R2, R3
        BEQZ    R4, L1
        DSUBU   R1, R5, R6
L1:     . . .
        OR      R7, R1, R8
```

```
        DADDU   R1, R2, R3
        BEQZ    R10, L1
        DSUBU   R4, R5, R6
        DADDU   R5, R4, R9
L1:     OR      R7, R8, R9
```

So, ordering of an instruction with respect to a branch instruction needs to take care of these points. So, instruction that is control dependent on branch instruction cannot be moved before the branch. If we move an instruction which is actually control dependent on a branch instruction before the branch instruction what happens is, this instruction will not be controlled by the branch anymore. So, as a result if this instruction is supposed to be executed only when the branch condition is true, now that scenario will be violated, when we move this instruction out of this the branch reach. In other words, so any instruction that is control dependent on a branch instruction should not be reordered.

It should be kept as it is, so that the branch condition can control the execution of that instruction. And similarly, an instruction that is not control dependent on a branch instruction should not be moved after the branch. If we move such an instruction after the branch then the execution of that instruction depends on the branch condition. In other words the branch is now going to control the execution of that instruction and which is actually violating the program correctness. So, as a result whenever we want to reorder the instructions with respect to a branch, we have to see whether rearranging of these instruction, rearranging of these instructions with respect to the branch, is going to create any problem. Like one of these two and if it is so, then we should not reorder that instruction.

Whereas, if an instruction which is not going to be affected by the branch instruction, then we can reorder the corresponding instructions. So, effectively as long as the program correctness is maintained, the control dependencies can be violated. And to maintain the program correctness we have to preserve the exception behavior as well as the data flow. So, we just consider an example here. Consider a simple code where we have an add instruction which is writing a value to register R2 and there is a branch instruction branch equal to 0 condition is check with the register content R2.

So, if register content R2 is equal to 0 then we are branching to an instruction pointed by L1 and if it is not equal to 0, then we execute the load instruction. So, in this case if we are going to rearrange these instructions, that is nothing but if we are going to rearrange load and the branch instruction. In other words if we are executing load instruction before the branch instruction, it may create an exception. For example, if R2 is equal to 0 and R now the load instruction is going to read from the memory location 0 and which can give an exception.

So, as a result, like, we should not execute this load instruction before the branch instruction. In other words, so here in this case, the load instruction is actually dependent on the branch. So, according to our first point, it clearly says an instruction that is control dependent on a branch cannot be moved before the branch. When we consider another example, see here an add instruction which is writing the value to R1 and there is a branch instruction which is comparing whether R4 content is equal to 0 or not.

If R4 content is equal to 0, then we are going to an instruction which is labeled by L1 and if R4 is not equal to 0, then we are going to execute this subtract operation which is also going to write value to R1. And finally, there is a OR instruction which is actually requiring R1 as one of the source operands to perform this OR operation and the result will be written to the register R7. Now, in this scenario we can clearly see that OR instruction requires R1 and this R1 can be supplied by add instruction or subtract instruction depending on the branch condition.

If the branch is taken then R1 will be supplied by add instruction, if branch is not taken R1 supplied by subtract operation. So, effectively now, we can clearly see that here the input for the OR instruction depends on the branch outcome. So, in such scenario, we cannot move our subtract operation before the branch instruction. If we move this subtract operation before branch instruction. So, as a result what happens is now R1 will have only the value supplied by subtract operation and because the add is performed earlier to subtract and because the subtract is performed later as a result that, whatever the value written by add operation will be over written by the subtract operation.

And even when the condition in the branch is true, we are actually taking the value supplied by the subtract operation. And that is actually not correct according to the program. So, according to the program if the condition is true, then we have to get the value supplied by ADD as an input to the OR instruction and if the condition is false then we have to get the value supplied by SUB as an input to the OR. So, now because of that when we rearrange our subtract operation and the branch instruction as a result the program correctness will be violated.

So, that also says that. So, when an instruction is dependent on a branch instruction, should not be moved before the branch instruction because when we move subtract operation before the branch instruction, now branch cannot control the subtract operation anymore. So, as a result we will have program correctness issue. So, effectively in the first example we know that when we rearrange the instruction there may be an exception occur in the program. And in the second case because of this rearranging the instructions there may be a change in the data flow because the OR instruction is supposed to get the value supplied by ADD instruction when the condition is true, but because of this rearrangement.

So, now OR instruction is going to get the value supplied by subtract instruction. So, that means there is a change in the data flow that is actually violating, that is giving, the program incorrectness. Similarly, in the first example because of the rearrangement, it may create an exception and that is also leading to program incorrectness. So, as long as the program correctness is maintained, we can rearrange the instructions, but if rearrangement of instructions is going to create program incorrectness then we should not do that.

And finally, we consider another example. Here we have ADD instruction which is supplying the value to R1 register and there is a branch instruction which is checking whether R10 is having value 0 or not. And if it is 0 then we are going to an instruction labeled by L1 that is OR instruction, which is going to perform OR operation on R8 and R9 contents and the result will be stored in R7. And if the condition is false here then we are going to execute SUB instruction and ADD instruction on the respective registers. And assume that here the R4 register is not used any more in the program.

And if that is the case, even when we move this subtract instruction before the branch instruction then it is not going to create any problem. And it is not going to create any issue with the program correctness because when they move the subtract operation before the branch instruction. So, branch is using R10, but the subtract operation is not dependent on the branch outcome because we know that the subtract operation value whatever the value produced by the subtract operation, is not used anywhere in the program.

So, as a result when we execute the subtract operation before the branch condition is known, it is not going to create any issue with the program correctness. So, when such instructions are there in the program, the compiler can identify and compiler can rearrange these instructions efficiently. So, that we can execute this in out of order without having any issue with program correctness. So, with that we conclude this module.

Thank you.