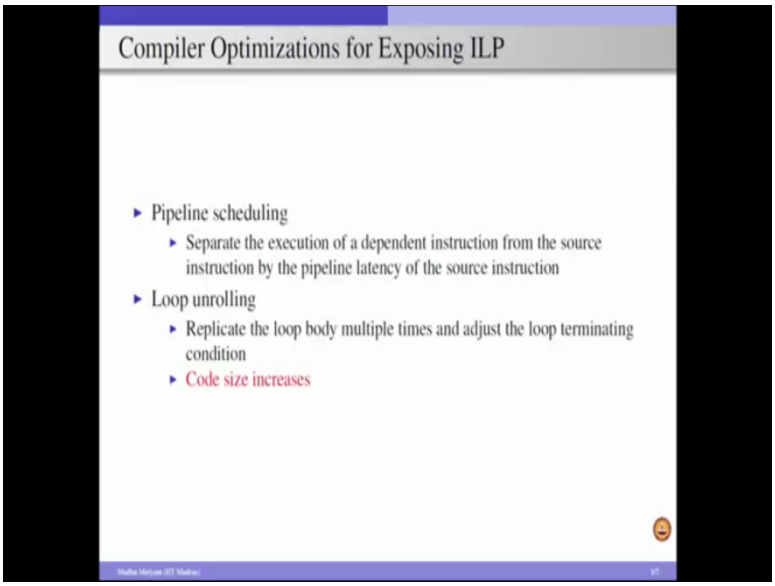
Computer Architechture
Prof. Madhu Mutyam
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Module – 06
Lecture - 20
Compiler Optimizations for Exposing ILP

So, as part of the last module we looked at different dependencies such as data dependencies, name dependencies and control dependencies and we also discussed various hazards associated with dependencies. And in this module we are going to look at the compiler techniques to exploit instruction level parallelization. So, the compiler can rearrange the instructions in such a way that, the overall performance can be improved in the program, overall performance of the program can be improved.

(Refer Slide Time: 00:52)



So, here in this we are going to consider 2 basic optimizations associated with the compilers. One is pipeline scheduling and other one is loop unrolling. In the case of pipeline scheduling, so, it is also called as a software pipelining. So, we separate the execution of a dependent instruction from source instruction by the pipelined latency of the source instruction. In other words, let us consider 5 stage pipeline and if instruction i and j are having data dependency. For example, the instruction i is producing a value which is required by instruction j. Now, if i scheduled instruction j at least after 5 pipeline cycle time as that of instruction i. Then even when there is dependency, so instruction j will not have any impact. Because by the time

instruction j is going for the execution stage the instruction i would have completed and written its value to a register. So, as a result, so, in the pipeline scheduling concept we have to separate the dependent instructions by a no of cycles which is equal to the no of pipeline stages. So, that the dependent instructions can go ahead with execution without incurring any stall. And the second optimization is loop unrolling. So, here the loop body will be replicated multiple times.

So, that will have more number of instructions in the body and as a result we can apply our pipeline scheduling whatever the first technique is proposing and so that we can find more independent instructions and as a result we can minimize the stalls significantly. So, but the problem with loop unrolling is, it increases the code size. For example, consider simple loop

$$\text{for}(i=0;i<100;i++)$$
$$x[i]=y[i]+z[i]$$

If this is the loop we consider then the loop body has only one instruction, but where as if I unroll this loop 5 times then effectively the loop body is going to have 5 instructions, which are like,

$$x[i]=y[i]+z[i]$$
$$x[i+1]=y[i+1]+z[i+1]$$
$$\ldots$$
$$x[i+4]=y[i+4]+z[i+4]$$

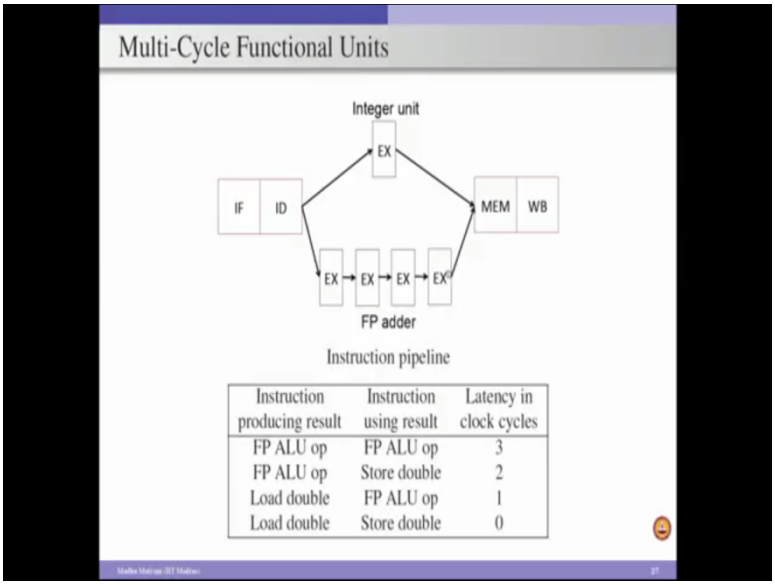
And once we unroll this 5 times then we have to adjust the terminating condition accordingly. The previously, the loop body was executed for hundred times now the loop body is going to execute for twenty times because the loop body is unrolled 5 times.

So, effectively using pipelined scheduling, we can schedule dependent instructions in such a way that these dependent instructions are separated by some number of pipelined cycles. So, the dependent instructions can get value from the producer instructions without having any stalls. And once we separate the dependent instructions from the source instructions, we fill these slots with the other instructions in the program. And using the loop unrolling, we unroll the loop body such a way that the loop body now going to have more number of instructions.

And once we have more number of instructions, we can try to see whether pipelined scheduling can be applied on this unrolled loop body. So, that we can minimize the number of stalls associated with the dependencies in the program. So, in another words, so we exploit

both pipeline scheduling and loop unrolling to minimize the stalls associated with these dependencies in the programme and improve the overall performance of the system. So, we explain these two concepts by considering set of examples in this module. Before discussing the example, we will consider our, the basic pipelined unit, we use that unit for the executing our sequence of instructions given in our examples.

(Refer Slide Time: 05:32)



So, we consider a multicycle functional units in our system, where we considered a 4 stage pipelining for floating point adder and one pipelined stage for the integer unit. And our remaining stages that is instruction fetch, instruction decode, memory access and write back, all are taking single pipeline cycle. So, given this pipeline design, so, when we apply operand forwarding for minimizing unnecessary stalls for our instructions, these are the latencies that the dependent instructions can incur with respect to the producing instructions.

So, if there is a floating point ALU operation is executed in the pipeline and if there is trailing floating point ALU operation which requires the value produced by the leading floating point ALU instruction then this trilling instructions has to incur a latency of 3 cycles, to get value produced by the leading instruction. In other words, let us assume that the instruction i and j are following in the program order where the instruction i is coming before the instruction j and both these instructions are floating point ALU instructions.

And now, instruction j can get the value supplied by instruction i only after 3 cycles of latency. The reason is, in this particular case we can see here, instruction i produces the value

only at the end of 6th cycle because in the floating point instructions will follow through this pipeline path and at the end of 6th cycle, it will complete the execution. And as a result it will ready to forward the value whatever it is computed by the floating point adder to any dependent instruction. Because instruction i is issued in cycle, let us say t and instruction j can be issued only at a cycle t plus 1.

So, that means already the instruction j is delayed by one pipelined cycle. And now instruction j requires operand, the source operand, only at the start of the third cycle. And we know that instruction i is producing the value at the end of 6th cycle and instruction j requires an operand in its third cycle because instruction j is already issued one cycle delayed to instruction i. So, effectively the instruction j can get the value with the forwarding only after 3 clock cycles, when instruction j is scheduled one cycle later to the instruction i.

And if we consider instruction j is store instruction and instruction i is floating point ALU instruction, in that case, instruction j requires at least 2 cycle stall. The reason is the instruction j is now going through this path because this is a store instruction and store instruction will go through this. And this integer execute stage is required for store operation because the store operation has to compute the effective address. And for computing the effective address calculation we use integer unit and which is going to take one pipeline cycle.

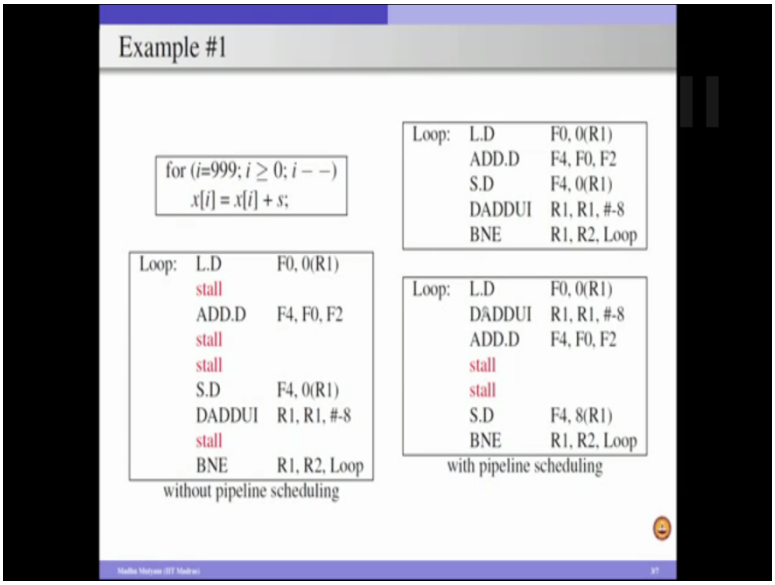
And now because instruction j is scheduled one cycle latter to instruction i and instruction i is producing the value at the end of 6th cycle and instruction j which is a store instruction requires the value only at the start of its 4th cycle. So, another words instruction j requires the value at the start of the forth cycle and instruction i is producing the value at the end of 6th cycle, but instruction j is scheduled one cycle latter to instruction i.

Effectively, the instruction j requires the 2 cycles stall or 2 cycles delay. So, that is what is given here. And now consider the other scenario where instruction i is load instruction and instruction j is floating point ALU instruction. And we know that the load instruction will follow through this path, because there is an effective address calculation for which we require integer ALU unit. So, load will complete the memory read operation at the end of its 4th cycle and because floating point instruction requires the value at the start of its third cycle.

So, as a result, it requires one cycle delay or one pipeline stall will be there if you want to schedule the floating point ALU instruction, immediately after one cycle delay to the load instruction. And finally, if we have 2 instructions, one is load double and other one is store double. And the load is producing the value which is required by the store instruction. In this scenario there is no delay associated with that or it is not going to create any stall. The reason is the load is producing the value at the end of the 4th cycle and the store is requiring the value at the start of 4th cycle because the store instruction is scheduled one cycle later to the load instruction.

So, as a result by the time store requires the data, the load is read the value from the memory and as a result using this load forwarding concept we can supply the data to the store instruction. And remember both the load and store will follow the same pipeline path, which is instruction fetch, decode, integer execute stage, and memory. So, as a result like there is no stall for a subsequent store operation which requires the value produced by the leading load instruction. So, we use these delays in our calculations for an example that we consider in the next foil.

(Refer Slide Time: 12:08)



So, consider a simple example, here this is a for loop, for (i = 999; i >= 0, i--) and we are computing this loop body which has a single instruction x[i] = x[i]+s and this loop body is executed for thousand times. We are performing this operation on an array which has thousand elements and we read a value from the array and we add some constant to that. And

finally, we store the value back into the same location in the array and this array has thousand elements. And we perform this operation thousand times and each time we are performing operation on a single array element.

Effectively, this operation requires loading the value from the array, adding the constant to that value, and storing the value back to the array, because array is stored in our memory. So, we require a load operation, followed by add operation, followed by store operation. When we convert this simple c code into assembly language, then this is what the set of instructions. It consists of load, because we are performing all this on floating point doubles. So, effectively each floating point double is going to take 8 bytes of space and when we use the instructions also, we have to specify that effectively.

So here considering l.d is load double. Similarly, add.d is performing add operation on 2 double values. Similarly, store double and so on. So, we load value from location in the array or location in the memory which is specified by 0 of R1. So, here in this code all the contents which are specified with F indicate the floating point registers and all the contents which are specified with R indicates that these are the integer registers. So, effectively R1 R2 are integer registers here and F0 F2 F4 are floating point registers. And R1 is actually, R1 is initially having some value and we load some data which is stored in some address which is specified by the contents of R1. So, we go to the memory and we get the value from that and we load that value to the floating point register F0. And then, because once we read the value from the i th location in the array we have to add some constants to that value.

So that constant is stored in another floating point register which is F2. So, we perform an add operation on the contents of F0 which has the value from, the read from the memory and the F2 has the constant value. And these 2 are added and the result is stored in another floating point register F4. And once this add operation is completed, now we are going to write the contents of F4 to the memory location which is specified by R1 because we are going to write to the same memory location from which we read the earlier value.

So, effectively we are going to write to the memory location which is specified by 0 of R1. So, once we perform this store operation, next we are going to decrement the R1 value, because we have to now pointing to the second array location, so that is specified by R1 minus 8. So, we are performing this, this, an immediate addressing mode is used. So, we have this minus 8. So, we are going to decrement R1 by 8. In other words, we perform an add

operation with R1 and minus 8. So, that now R1 is pointing to next location which is 8 bytes away from the previous location.

So, that we can get the next floating point value that is stored at that particular address. And after this add operation is performed to set, our, the subscript value, now we will check whether condition is true or false, because we are going to execute this loop only for thousand times and every time, after every iteration of this loop body we have to see whether condition is satisfied or not. So, branch not equal condition is checked here with respect to the R1 and R2. So, we are going to see whether R1 value is equal to R2 or not, if R1 value is not equal to R2 then we jump to the instruction which is pointed by this loop.

So, that is effectively we are again start reading second value from the memory by using the load operation. And then we perform an add, store and will continue. And if this condition is false then we will exit this loop and then we will continue with the subsequent instruction after this branch instruction. So, this is a piece of code for our high level language code here. And now we will see how much time this code is going to take to execute after considering the stalls associated with the dependencies between these instructions. And we know that, so this add instruction is actually dependent on load, because it is using this F0 which is produced by load.

And similarly, this store is dependent on add because the store has to wait for add to be performed. And so this add operation cannot be executed before this store operation because if we perform this add operation then it is going to check R1 contents as a result of the store operation is going to perform on different location. So, as a result this add i is also dependent on store instruction. And finally, this branch instruction is dependent on this add instruction because it is using this R1. So, R1 is updated in add instruction here and as a result this branch cannot be executed before add instruction.

Effectively, there is dependency among the different instructions in this code and as a result if you schedule these instructions as it is, we are going to get significant penalty in terms of stalls. So, when we scheduled these instructions as it is, so we are going to have these many number of stalls. So, here we can clearly see so this is a floating point, this is a load operation and there is a floating point ALU operation and which is taking one cycle stall because we know that from previous foil, we know that, the load instruction and there is a floating point

ALU instruction. Then the floating point ALU instruction will wait for one cycle to get the value supplied by the load.

So, as a result like this add operation cannot be executed immediately after the load instruction. So, it has to incur one cycle stall or it has to be delayed by one pipeline cycle. So, that the value produced by the load will be available and add can perform the operation. And similarly, there is a store instruction, which is a consumer instruction, and there is floating point add instruction which is a producer. So, a floating point ALU operation is producing a value which is required by the store instruction.

So, which is going to take 2 pipeline stalls, because we know that the store instruction which is dependent on the previous floating point ALU operation, which requires 2 pipeline cycles. Remember that, these latencies are after applying this operand forwarding or load forwarding techniques. And finally, there is a branch instruction which is dependent on previous add instruction, which is actually requiring one cycle. So, because of branch is actually computed in the second stage of our pipeline, but the add is going to produce the value only at the end of the execute stage.

So, as a result it has to wait for one cycle, though we have not mentioned the delay associated with the branch instruction and an integer ALU operation in this table, but because we are using the same 5 stage pipeline design, whatever we considered in the fundamentals of pipelining design and where our the branch conditions will be completed in the second stage that is ID stage itself.

So as a result, our branch instruction requires the one cycle stall compared to the, with respect to the, previous integer ALU operation. So as a result, so, without any pipeline scheduling if we schedule these instructions as it is. So, we are going to complete this task, or one iteration of the loop in 9 cycles. And in these 9 cycles the 4 cycles are just the stalls and 5 cycles are actually performing the computation required with respect to the loop body and remember. In this 5 instructions these 3 instructions are actually performing the required operations which is like the load, add and store, but last 2 instructions are actually the overhead associated with this for loop.

So, one is decrementing the iteration value and the other one is branch condition check. So, effectively these 2 are overheads for performing one iteration of the loop. Effectively for every iteration we are going to incur 2 extra instructions and also because of the

dependencies among the instructions we are, in addition to these 2 extra instructions and we are also incurring extra 4 stalls. In other words out of these 9 cycles that we incur for executing one iteration of the loop, we are actually wasting 6 pipeline cycles time because of either the stalls or because of the overhead associated with this loop iterations. Now, without worrying about the loop iteration overhead, if we schedule these instructions using the pipeline scheduling concept, we can minimize these stalls. Now, you can see here in the pipeline scheduling the compiler rearranges this code in such a way that, so we move this integer ALU instruction before the floating point ALU operation. And after that we have not rearranged any other instructions.

So, now we can see here, we know that floating point add instruction requires the value supplied by our load instruction, but it takes one cycle stall if you are executing the ADD instruction immediately after the load instruction. And now in order to minimize this stall, what we can do is, we fill this pipeline cycle with, this stall with, useful instruction that is add instruction. So, when we move this add instruction here we are not incurring any stall and we delayed the floating point add operation by one more cycle and by the time this add operation comes to the execute stage our load is having the value.

And using the load forwarding, we can supply the value to the ALU operation. So, as a result this ADD operation is not going to incur any more stalls. And that minimizes the stall associated with this ADD instruction with respect to the load. But now, we already discussed earlier, our store operation actually dependent on the integer ALU operation. And we mentioned that our integer ALU operation should not be moved before the store operation. Because when we move this integer ALU operation before the store operation then store is going to write the value to a different location, because already this integer ALU operation decremented R1 value by 8.

So, in order not to have any impact, we rearrange or, we format this “S.D” instruction also. We do some changes to this store instruction such a way that, previously our store instruction has 0 of R1, now our store instruction has 8 of R1. This indicates that we calculate the effective address by adding 8 to the contents of R1. Already R1 is decremented by 8. So, now new R1 is going to be the old R1 minus 8 and for that we are going to add 8. So, effectively we are pointing to the old R1 value.

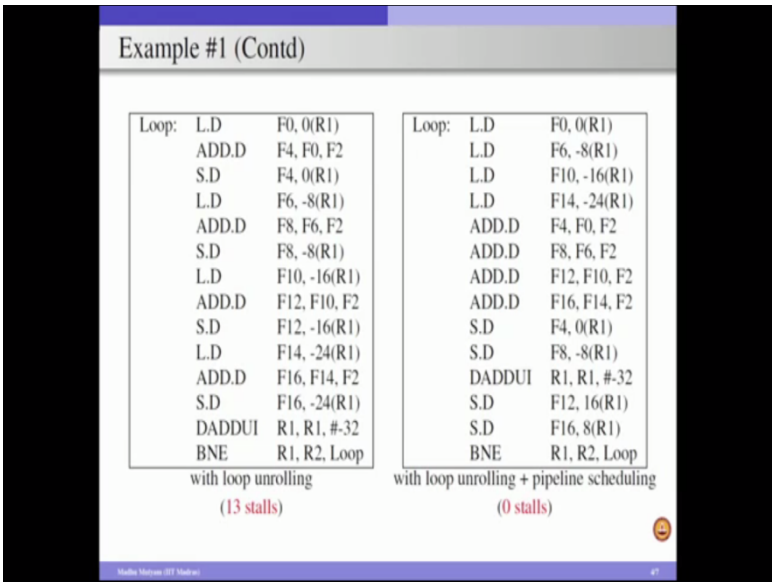
So, as a result there would not be any change and store is going to perform the store operation to the correct location in the memory. So, as a result there would not be any correctness issue in the program. And after that there is a branch instruction, previously this branch instruction was incurring one pipeline stall because this is waiting for R1 to be computed by the integer ALU operation, but now this R1 is already computed way ahead with respect to the schedule of this branch instructions. So, as a result it is not going to have any stall.

So, because of this pipeline scheduling, we minimize the number of stalls from 4 to 2. So, as a result, with this pipeline scheduling, now we have only 2 stalls per iteration. So, as a result out of these thousand iterations for thousand iterations of this loop, now we are going to have two thousand stalls as compared to four thousand stalls that we incur by using no pipeline scheduling mechanism. So, effectively by using this pipeline scheduling for the simple code, we minimize the stalls significantly.

Almost fifty percent of the stalls are minimized by using this pipeline scheduling, but still here we have not eliminated the overhead associated with this iteration overhead that is corresponding to the performing this integer ALU operation to decrement the subscript value then and branch condition check. So, in order to minimize the overhead associated with these 2 instructions, what we have to do is we have to exploit the loop unrolling. So, when we consider the loop unrolling, when we unroll the loop by 5 times for example.

So, we need the overhead associated with this iterations only one for every 5 instructions. The previously, in our unrolled, in our base instructions, we know that we have to compute the branch instructions and we have to change the subscript at every iteration of loop body. And the loop body consist of only one instruction, so effectively for every one instruction, we have to change the subscript as well as we have to check the branch condition. So, as a result our overhead is significant with respect to these loop body iterations. But when we consider the loop unrolling, and if we unroll the loop by 5 times we can minimize the overhead associated with the iterations 5 times. We minimize the overhead associated with these loop iterations by almost 5x. So, in other words, when we unroll the loop,

(Refer Slide Time: 29:12)



for example, 4 times in this scenario, so rather than performing add operation on single element, we are now going to perform add operations of 4 elements. So, effectively we have total of 3*4 = 12 instructions plus 2 extra instructions associated with the loop body iterations. So, these 2 are the instructions associated with the loop body iterations. One is for changing the loop subscripts and the other one is checking the condition and remaining 12 instructions are associated with load, add and store for each of the instructions, because there are 4 instructions in the loop body and each instruction is going to take load, add and store.

Effectively, 12 instructions for this plus 2 instruction. So, we have 14 instructions, when we unroll the loop 4 times and so this total computation is going to take total of 27 cycles, in that 14 cycles are required to perform this operations, and 13 cycles are required for stalls. These 13 cycles are because of one cycle stall between this integer ALU operation and the branch condition and one cycle stall between this, the load instruction, and add instruction and 2 cycle stall between add instruction and store instruction, because we have 4 such a pairs of load and add in this code.

So, we have 4 cycle stall and 4 pairs of add and store, we have 8 stalls, effectively 12 stalls plus 1 stall 13. So, when we unroll the loop 4 times, we are going to take a total of 27 cycles to execute the loop body which consists of 4 instructions in it. And remember when we unrolled the loop, we have not applied any pipelined scheduling. So, as a result there is a scope to minimize these stalls if we apply pipeline scheduling. So, when we combine this

loop unrolling with pipeline scheduling. So, we eliminate all the stalls, effectively our unrolled loop executes without any stall, but we exploited pipelined scheduling also here after unrolling the loop.

So, when we unroll and then apply pipeline scheduling, we can rearrange the instructions such a way that, all the dependent instructions are delayed by a time which is more than the time the dependent instruction is going to get the value from the producer. So, now you can see here, all the load instructions are scheduled before the add instructions and the store instructions. Now, it can clearly see this add instruction requires a value produced by this load instruction. And in earlier case when we scheduled this add instruction immediately after the load, it is going to take one cycle stall, but now this add is separated with respect to this load by 3 other instructions.

So, as a result by the time comes for the execution this has already produced the value. So, as a result there is no stall for this add instruction. Similarly, this add is separated with this load by another 3 instructions. So, this also can get the data without any stall and that can be explained by the other add instructions also. And previously this store is taking 2 cycles stall with respect to this add, because add is supplying the data only after 2 cycle delay. So, but now this store is actually requiring the data from this add and these 2 are separated by 3 other add instructions.

So, as a result which is more than delay incurred by the store previously with respect to this add. So, there is no stall for the store operation and it can peacefully get the data from produced by this add instruction. And that can be explained for the other store operations also. And this add instruction is scheduled 2 cycles ahead of this branch instruction. So as a result, this branch instruction is not going to incur any stall associated with this add instruction. So, we eliminated all the stalls in the code. So, as a result this overall loop body is going to take 14 cycles to execute.

So, effectively, 4 instructions are completed with 14 cycles and none of these instructions are incurring any stalls. So, as a result, we can improve the overall performance. So, this shows that the compiler looks at the independent instructions. And even if the dependency is there the compiler can rearrange these instructions in such a way that, it can minimize the stalls so that the overall performance can be improved. And in order to that the compiler applies the

concepts such as that the pipelined scheduling and loop unrolling. Loop unrolling gives a larger scope for rearranging the instructions.

And once we increase the scope then we apply our pipelined scheduling on top of that. So, that we can schedule the instructions properly and minimize the stalls significantly in our code, that improves overall performance. So, with that I am concluding this module and in the next module we are going to discuss techniques associated with the hardware mechanisms to improve the overall performance of Superscalar pipelined design.

Thank you.