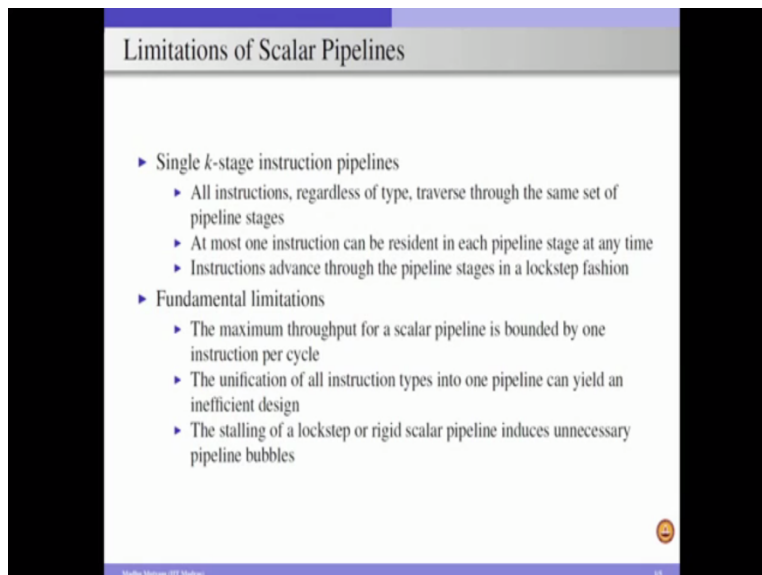


Computer Architecture
Prof. Madhu Mutyam
Department of Computer Science And Engineering
Indian Institute of Technology, Madras

Module – 06
Lecture – 18
Scalar Pipeline to Superscalar Pipeline


So, as part of fundamentals of pipelining we discussed that k stage pipeline will give k -fold increase in the throughput in an ideal scenario. And in this module we are going to discuss the limitations of the scalar pipeline design. And will discuss the need for the superscalar pipeline design for improving the performance further.

(Refer Slide Time: 00:43)



The slide is titled "Limitations of Scalar Pipelines" and contains the following content:

- ▶ Single k -stage instruction pipelines
 - ▶ All instructions, regardless of type, traverse through the same set of pipeline stages
 - ▶ At most one instruction can be resident in each pipeline stage at any time
 - ▶ Instructions advance through the pipeline stages in a lockstep fashion
- ▶ Fundamental limitations
 - ▶ The maximum throughput for a scalar pipeline is bounded by one instruction per cycle
 - ▶ The unification of all instruction types into one pipeline can yield an inefficient design
 - ▶ The stalling of a lockstep or rigid scalar pipeline induces unnecessary pipeline bubbles



Madhu Mutyam (IIT Madras) 44

So, we know that in a k stage instruction pipeline, we get in an ideal scenario a k -fold throughput increase, but in the k stage pipeline whatever we discussed in the last week, we know that all the instructions regardless of the type traverse through the same set of pipeline stages. In our discussions we consider five stage pipeline and each instruction whether it is memory instruction or an ALU instruction or a branch instruction, all these instructions will go through all the five pipeline stages.

And as a result sometimes we incur unnecessary penalties and we sometimes waste pipeline cycles and because of that we cannot get increased performance. And also we know that at any point of time only one instruction will be there in a single pipeline stage. So, in other words in a

five stage pipeline at most we will have five instructions in five different pipeline stages. And if you want to increase the number of instructions that are there in pipeline stages, then we have to consider a different organization compared to this scalar pipeline design.

And one more thing is the instructions advance through this pipeline stages in a lockstep fashion. So, in other words if an instruction is stalled in one pipeline stage so, the following instructions will not proceed further and the trailing instructions are waiting for this stalled instruction to proceed further. And as a result we will get unnecessary bubbles or stalls in the pipeline which decreases the overall throughput improvement or the performance improvement. So, in summary because of this scalar pipeline we have several limitations. One is the maximum throughput we achieve is at most one instruction per cycle. So, though we have a k-stage pipeline which improves the throughput by k times.

At any point of time only one instruction will be coming out of this pipeline design. So, as a result out instructions per cycle will be at most 1. And it will be 1 only in the ideal scenario and non ideal scenarios it will be less than 1. So, we have to improve our instructions per cycle so, that the overall performance can be improved. Note that instructions per cycle will be inversely proportional to the cycles per instruction. We already discussed in our processor performance equation the term CPI clock cycles per instruction.

And if you take the inverse of that then we will get instructions per cycle and the processor performance can also be represented in terms of instructions per cycle. So, in this scalar pipeline design even if we consider k stage, where k can be any number, our IPC will be at most 1 per cycle. And we also know that the deeper pipelines have disadvantages such as the penalties associated with mispredictions and dealing with the branches and so on. So, as a result we cannot go for a very deep pipeline design to improve the overall speed up. Even if you go for the deeper pipelines because of our IPC is limited to 1. So, we cannot gain significant performance improvement with the scalar pipeline design.

So, we have to look for some other alternatives. And the second fundamental limitation with the scalar pipeline design is, the unification of all instruction types into one pipeline can yield an inefficient design. We know that irrespective of the type of the instruction all the instructions are going through the all pipeline stages. And as a result we will get unnecessary external fragmentation. For example, branch instruction we know that, it can be completed in 2 pipeline stages if our branch is resolved at the end of id stage. And similarly if it is a store instruction it

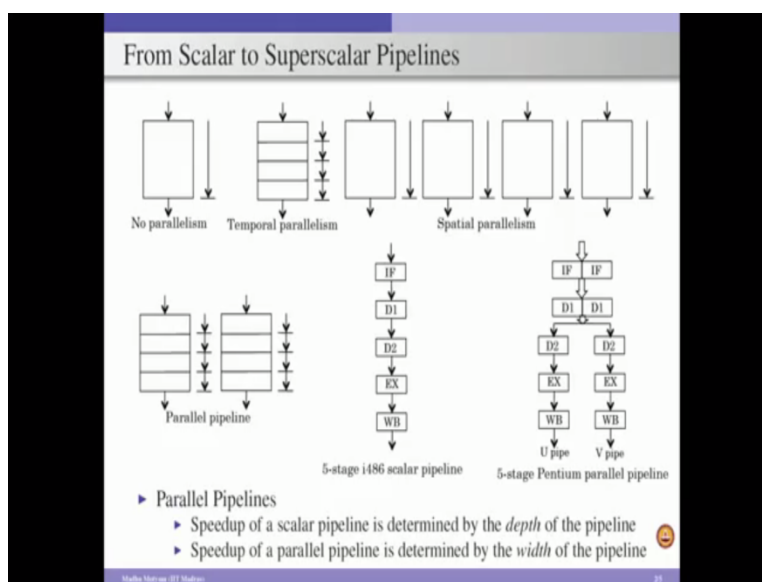
requires only four pipeline stages in a five stage pipeline, but whereas ALU or load instructions require five pipeline stages.

So, now when we treat all the instructions equally then we will unnecessarily degrade the performance because of this external fragmentation. So, that means like we need to consider a diversified pipeline design for the different type of instructions. And finally third fundamental limitation is the stalling of lockstep or rigid scalar pipeline induces unnecessary pipeline bubbles. We know that in the scalar pipeline design whatever we discussed previous modules. So, the instructions are following through in a lockstep fashion and it will proceed one pipeline stage to the next pipeline stage in a lockstep fashion.

So, as a result if leading instruction is stalled in any of the pipeline stages all the trailing instructions will be stalled. It cannot proceed further even if the trailing instructions are independent with respect to the leading stalled instruction. So, as a result we will unnecessarily get bubbles in the pipeline and that degrades the performance. So, because of all these things we have to look for some other alternative design. And in this week we are going to discuss one such alternative design that is called as superscalar pipeline design, which take care of all these limitations and then improve the performance of the overall system.

So, in the remaining foils we are going to discuss the techniques to deal with these 3 fundamental limitations. So, in order to go from scalar pipeline design to a superscalar pipeline design so, we will discuss the different designs here and the first one is consider a non pipeline design.

(Refer Slide Time: 07:16)



So, there is no parallelism involved, and this entire hardware component is going to execute only one instruction at a time because of that our overall throughput will be significantly degraded. It will be like one instruction processed per n gate delays, assuming that this entire instruction data path takes n gate delays. Now, in order to improve the throughput we already discussed that the scalar pipeline will improve our throughput by k -fold if we consider the k stage pipeline data path. So, when we consider the pipelining in terms of temporal. So, this entire, the data path is divided into k stages and in this particular case we consider the four stage.

So, effectively at any point of time four different instructions will be there in this data path at different stages. The data path is utilized by different instructions and sharing these hardware components temporally. So, as a result we get temporal parallelism with this design and this improves the overall throughput by k fold, where k is a number of the pipeline stages we considered here, compared to design where no pipelining is considered. But the parallelization can be obtained even with the spatial way. So, that is like by using this spatial parallelism where rather than dividing a single data path into multiple stages here we are replicating this non pipelined data path several times.

So, in this particular example we are replicating this non pipelined data path into four, so that at any point of time four instructions are executed by this four non pipelined data path units. And finally we will get four fold increase in the throughput. So, though these 2 temporal and the spatial parallelism mechanisms will give a through put improvement, the temporal one is efficient from the hardware point of view because it is reusing the same hardware for processing different instructions, but whereas in the spatial parallelism case we are unnecessarily wasting the hardware resources.

So, now if we combine this temporal parallelism and the spatial parallelism, then we can get parallel pipeline. So, here in this particular example we are considering a 2 stage parallel pipeline and each pipeline is, each stage is further divided into four stages. Effectively in our parallel pipeline we replicated our temporal parallelism units twice. So, we considered 2 units where each unit is temporally parallelized. So, effectively we have system where 2 instructions can be processed in each pipeline stage and overall pipeline is consists of four stages.

Effectively at every cycle we can get 2 instructions processed. So, this is going to improve the throughput significantly compared to this just considering temporal parallelism. But again if we consider the hardware point of view this parallel pipeline is actually taking double hardware as

compared to the temporal parallelization, but the advantage with the parallel pipeline is we are going to process more number of instructions per cycle. So, now having discussed this parallel pipelines and temporal parallelism, spatial parallelism, now we will consider an example of real processors where these designs are considered.

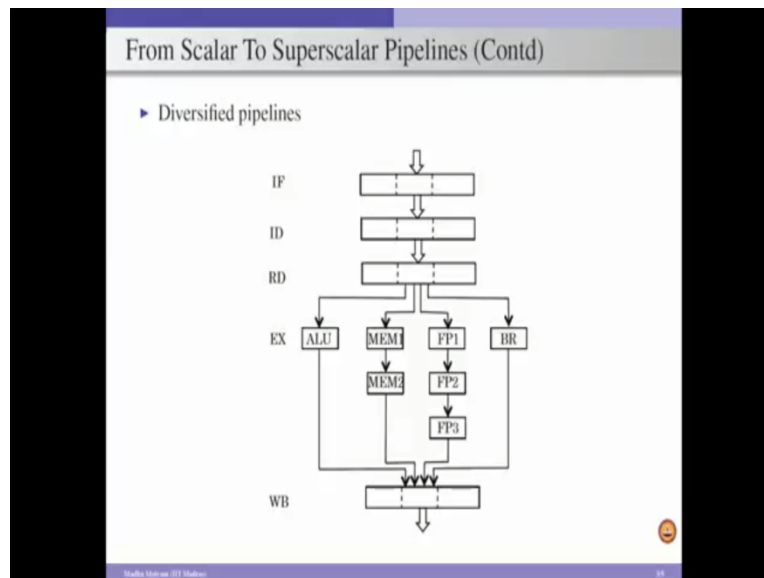
If we consider Intel 486 processor, it was using a scalar pipeline and in this particular design they have considered five staged pipeline consists of instruction fetch, decode, a dispatch, execute stage and write back. Here the execute stage consists of performing the ALU operations as well as the memory access. And so, in this design they have considered scalar pipeline. So, effectively only one instruction at any point of time will be there in a single pipeline stage. And the overall performance improvement will be related to at most 1 instruction per cycle. And the intel came up with advanced design compared to this in their next processor design, by considering the parallel pipelines.

So, in the Pentium processor they have considered five staged parallel pipeline. Here they consider a pipeline width of 2. So, when I say pipeline width 2, so each cycle they are going to fetch 2 instructions and they are going to decode 2 instructions and after the decoding two instructions, then we have 2 parallel pipelines, one is a U pipeline and other one is a V pipeline. And based on the instructions and the based on the functional units associated with this pipelines the instructions will be steered to the different pipeline units of this parallel pipeline. And once an instruction is placed in either a V pipeline or U pipeline, it will proceed further in the same pipeline and it completes its execution. So, as a result like once we consider a parallel pipeline we can improve the overall performance. And we can eliminate the first limitation associated with the scalar pipeline. So, that is the first limitation, as I mentioned earlier, is using the scalar pipeline we can get at most one instruction per cycle, that means like the IPC can be at most of 1 in the scalar pipeline, but once we consider a parallel pipeline we can improve IPC to 2.

So, in this case for example if we see here at the end of every cycle we can complete 2 instructions. So, in other words 2 instructions can be completed per cycle. So, IPC will be 2 here. So, in order to improve our IPC instructions per cycle, then we have to go for parallel pipelines. So, in summary, we know that the speed up of a scalar pipeline is determined by the depth of the pipeline. So, if we have a k stage pipeline, we can get a speedup of at most k, but we also know that the deeper pipelines have an impact on the performance penalty associated with mispredictions associated with the branches.

So, as a result the deeper pipelines may not be always efficient in improving the overall performance. And when we consider parallel pipeline the speed up of a parallel pipeline can be determined by the width as well as the depth, because a parallel pipeline also consists of multiple stages in each of the pipeline unit. So, effectively we can improve the overall performance and that is determined by both the depth and the width. So, then now we consider the next component that is a diversified pipeline. So, we already discussed that the second limitation with the scalar pipeline is unification of all the instructions and all the instructions are processed through the same pipeline component, but because the different instructions require different pipeline stages, so, as a result, we can diversify this pipeline design.

(Refer Slide Time: 15:36)



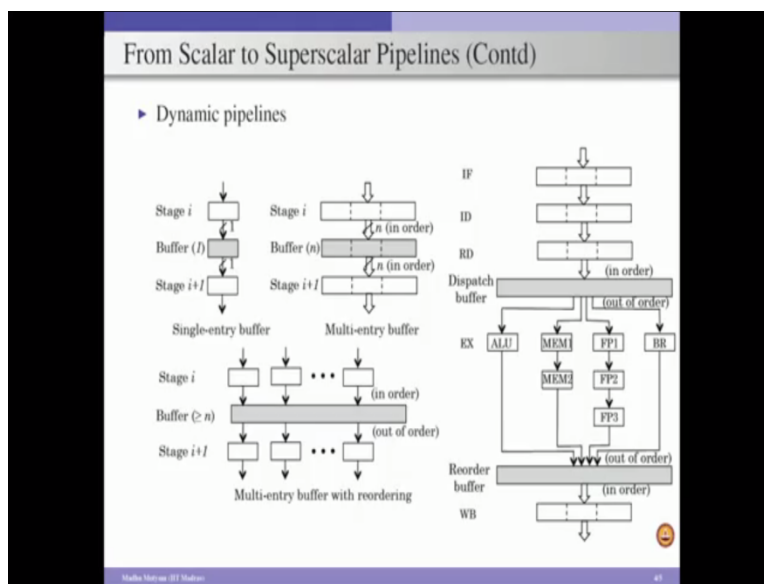
So, in our diversified pipeline design and with the parallel pipeline of width three, so when I say pipeline width of 3 we have in each pipeline stage 3 instructions can be processed. So, in the instruction fetch stage we fetch 3 instructions, in the instruction decode all the 3 fetch instruction will be decoded thus by using the respective decoders. And then we will perform register read for these 3 instructions that are decoded in the previous cycle. And after register read is performed then we dispatch instructions into different functional units depending on the type of the operation we are going to perform.

And finally based on the number of cycles it takes to perform the operation, these instructions after completing their execution they will go to write back stage and it will be again written to the buffer associated with the write back stage. Note that here except the execute stage all other

stages of the pipeline have uniformity in their design, but in the execute stage because depending on the type of instruction we consider different type of functional units and each functional unit will take different number of pipeline cycle.

So, we can design efficient pipeline design if we consider the diversified pipeline. So, as a result once we consider this different pipelines cycles for the execute stage. So, we can minimize unnecessary stalls associated with the dependent instructions and we can improve the overall performance.

(Refer Slide Time: 17:23)



We consider dynamic pipelines in order to deal with the third limitation. We know that in a scalar pipeline design at any point of time only one instruction will be there in a single pipeline stage. And once it is processed in that pipeline stage, it will be returned to the buffer associated with the corresponding pipeline stage. So, in other words between 2 pipeline stages in our scalar pipeline design we have a buffer that can store one instruction. In order to deal with parallel pipelining design one thing we can do is, we can consider multiple entries in the buffer between adjacent pipeline stages.

So, that is nothing but, by considering a multi entry buffer between stage i and stage i+1, we can move the instructions that are processed by the previous pipeline stage to this buffer. So, that the next pipeline stage can read the data from this multi-entry buffer and it will proceed further. Once we consider each of these entries in the multi entry buffer as an independent component,

we can move the instructions efficiently through this multi-entry buffers and as a result we can improve the performance efficiently.

By the way in this multi entry buffer even though we consider the entries are working independently, but we still consider inorder processing in these buffers that is nothing but the instructions which are fetched in the cycle I will be proceed to the buffer and after that the next instructions which are fetched in the inorder from the program will be moved further. And also from this buffer when we are issuing these instructions to the execution unit, the dispatch will also happen in the in order.

Effectively once we have 3 instructions waiting in the buffer to be dispatched to the functional units, we will issue in the program order only. Even if there is an independent instruction which is waiting to be dispatched to the functional unit, if there is a leading instruction which is waiting for the functional unit to be available we cannot issue this independent trailing instruction. So, once we use this Inorder design, the overall pipeline design will be simple, but we cannot exploit the independent nature of the instruction in the program. As a result the performance cannot be improved significantly.

So, in order to improve the performance by exploiting the instructional parallelism or instruction level independence we have to go for out of order execution. So, that we can consider in a multi entry buffer with out of order issue. So, when we consider this multi entry buffer with reordering, the main advantage we get is we can select independent instructions from this buffer which can store at least n instructions in it. And we select n independent instructions and send those n instructions to n stages, n units of stage $i+1$. So, note that the difference between this multientry buffer and the multientry buffer with reordering.

Here we are actually following the inorder between all the stages, but whereas here we follow in order till stage I, but for the stage $I+1$ we are actually going for an out of order. When I say here inorder and out of order, these are with respect to the program order. The inorder is with respect to the program order, whatever the order the instructions are arranged by the compiler that is called as the program order. And an out of order is dynamically we can see which instructions are independent and if we have some independent instructions we can schedule those instructions ahead of the previous instruction which are in the program order.

And this out of order execution actually improves the overall performance and that is the reason why we can use this out of order execution and we exploit this instructional independence and

improve the overall performance. So, in summary in a scalar pipeline we consider only one buffer and buffer can store only one instruction at any time. And this buffer is pipeline register between 2 adjacent pipeline stages and when we deal with the parallel pipelines.

So, we extend our single entry buffer to a multi-entry buffer. So, that a buffer can store multiple instructions in it and, but the processing can be done in the in-order in this multi-entry buffers between multiple stages. So, as a result we cannot exploit instructional independence and that can degrade the performance. In order to exploit the instruction level parallelization we have to go for out of order execution and that is where we can modify this multi-entry buffer with multi-entry buffer with reordering concept, where we move n instructions which are processed by the previous stage to the buffer, but when we want to select the instructions from this buffer to the next stage, we can select n independent instructions.

And the selection can happen based on the independent nature of the instructions. So, as a result we can improve overall performance. And also in order to exploit the instructional parallelization we have to design this buffer with more than n entries, if we are considering an n width pipeline. So, that there may be scenarios where sometimes we may not get all n independent instructions. As a result the instruction may be stalled in the buffer. So, we need to have a capacity of the buffer such a way that it can accommodate more than n instructions, because our pipeline width is n . At every cycle n instructions will come to the buffer and if the previous n instructions are not sent to the next stage. So, as a result we have to have storage for keeping the next set of n instructions.

So, that is a reason why when we say buffer here we are consider a buffer width of greater than or equal to n . And once we have this multi-entry buffer with reordering, now we will see a design a processor design which will be accommodating all these diversified pipelines, multi-entry buffers as well as processing multiple instructions simultaneously. And we get a superscalar pipeline design. So, in this superscalar pipeline design, in this particular example we are considering a pipeline width of 3. So, effectively our instruction fetch is consists of 3 instructions can be fetched from the instruction cache or the memory.

And all this 3 instructions will go the next stage that is the decode stage. We decode 3 instructions at after that we read operands for these 3 instructions in the next stage. And that also having 3 entries in the stage and finally once we perform the register read for all the 3 instructions, now we will send all these to a dispatch buffer and whose capacity is more than 3.

So, we keep all these instructions which are processed till this stage and store in this dispatch buffer. And till this point our processing is happening in the in-order that is in the program order, but from this dispatch buffer we dispatch the instructions to the execution units in an out-of-order fashion by exploiting the instruction independence.

And in the execute stage we consider four different functional units. One is an integer ALU unit, memory operations or memory units or floating units and finally the control unit, that is performing that performs the branch instructions, that processes the branch instructions. And these instructions will be dispatched from the dispatch buffer in an out-of-order fashion and executed on the various functional units. And as and when instruction is completed it will be written to the buffer. And this buffer is called as a reorder buffer, where instructions will be written to the reorder buffer in an out-of-order fashion with respect to the program order, but finally when we want to write back these instructions to the actual registers, actual architectural registers we have to write in the program order only.

In order to maintain the correctness of the program as well as in order not to have any complications because of exceptions and so on. So, as a result, when you are going to write the outputs to the registers destination registers, we have to write in the program order. That is a reason why we consider a reorder buffer here. So, the role of reorder buffer is to rearrange these instructions which are executed in out-of-order fashion to be rearranged in the program order. So, effectively input to this reorder buffer comes in out-of-order fashion, but the output from this reorder buffer to the next stage will happen only in the program order.

And once we take the instructions from the reorder buffer in the program order, we will perform the write back that also again of width 3 and with that we complete the instruction processing. By using this, the parallel pipelining diversified pipeline as well as multi-entry buffers, we can improve the performance significantly and we can achieve IPC more than 1. In this particular case if everything is working smoothly without any stalls we can achieve IPC of 3, 3 instructions per cycle can be completed with this super-scalar processor design.

And having discussed this superscalar pipeline architecture, now in the next module we are going to discuss the instruction dependencies because that is required for us to identify independent instructions in our program, to exploit instruction-level parallelization. And once we have instruction-level parallelization then we can execute those instructions independently on this super-scalar pipeline design. So, with that I am concluding this module.

Thank you