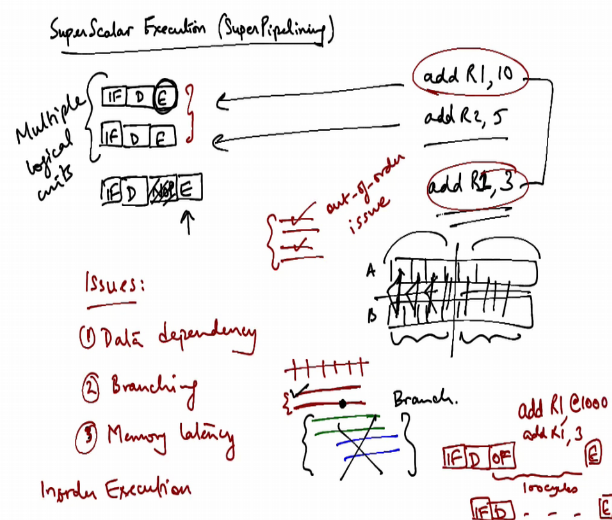
Introduction to Parallel Programming in OpenMp
Dr. Yogish Sabharwal
Department of Computer Science & Engineering
Indian Institute of Technology, Delhi

Lecture - 04
Superpipelining and VLIW

(Refer Slide Time: 00:02)



So, the next feature that pushes this a little further is Superscalar Execution, and it is also called Super Pipelining. What happens in this is that the processor if it determines that there are instructions like in the previous example, we saw add R 1 10 and add R 2 5. So, if it is able to figure out that these instructions are nothing to do with each other. Then it can actually execute them both in parallel. So, you have instruction fetch decode and execute and simultaneously, you could be doing; you could do the instruction fetch decode and execute for the second instruction. Let we could do both of these in parallel, but what does this require; now this requires multiple logical units. So, you need multiple hardware for each of the stages.

Because you want to be fetching 2 instructions at a time you want to be decoding 2 instructions at a time you want to be executing 2 instruction at a time and of course, if you have something like let us say that the second instruction instead of add R 2 5, it was let us say add R 1 R 3, right, then could you execute that in parallel along with the first instruction. So, you could do the instruction fetch in parallel. There is nothing wrong

with fetching both these instructions in parallel you could decode both these instructions in parallel, but while this first instruction is getting executed can you execute the second instruction no essentially what happens; in this case is that for the second instruction, you have to execute a no op cycle. No op is no operation, right, you have to wait because it has a dependency on some other instruction, right and then you could execute in the next cycle, right. Once 10 has been added to R 1, then you could add 3 to R 1 in this cycle can you think of some examples where this kind of architecture some charity would be very useful.

So, one common I mean a lot of common operations is when you are doing linear algebra operations like when you are working on matrices and vectors, right. So, suppose you are trying to scale a vector or computer dot product you have lots of operations which are very similar in nature and which are completely working on independent data sets, right. So, for instance, if you are computing the dot product of 2 vectors A and B; so what do you do you pick up the first element of each of them, you multiply them together, you pick up the second elements of each of the vectors multiply them together the third elements and so on.

So, each of these is independent; yes, you want to add them up finally, to one common value, but you know there are easy ways you can design easy algorithms to take care of that right for instance you could keep track of the result of let us say first half of the vector in a separate variable and you could keep the dot product of the second half of the vectors in a separate variable and the end you could add them up together, right. So, you can paralyze this to large extents. So, when you are doing the operations on the first half of the vector and if you are doing the operation on the second half of the vectors, they are very much independent, they do not have anything to do with each other.

So, you could possibly rearrange your code. So, that the instructions for the first half and the second half get executed together and that would make very good use of super scalar execution. So, what are the issues that we typically phase with pipelining and with super pipelining. So, there are several issues the first one of them we have already seen is data dependency this can take various forms, but one of them is that for instance over here right we were adding ten to R 1 and then we were adding 3 to R 1, but you could not add 3 to the register and till that register was not free. It was already participating in some operation something was being written into that resistant, right. So, you could not you

know perform that execute cycle at that point in time. So, there are different kinds of data dependencies. So, we will be talk about them later the second issue is branching. So, what are we doing with this pipe with these pipelines, right?

So, we are filling up these pipelines let us say that these are the cycles and this is how the pipeline how the instructions are getting executed right. So, this lets say that we have issued 2 instructions together. So, this is super pipelining and then in the next cycle, we issued 2 more instructions. And in the next cycle, we issued 2 more instructions and at some point of time what we realize is that this particular instruction let us say was a branch instruction, right when you decode that instruction you realize it is a branch instruction. So, what happens now? So, what happens is that the instructions that are executing before it they are fine, right, they have to be executed anyways, but what about all the instructions which are being executed after it.

So, here we are assuming that you know the processor is just picking up the instructions in sequential order and putting them into the pipeline. So, what happens to all these instructions which come after the branch instruction? So, you have to basically get rid of them right because you are going to jump to some other piece of code. So, these instructions have to be thrown away. So, that is wasted work right you picked up all these instructions you put them in the pipeline, but eventually you know that was wasted you did not complete these instructions you could not finish them another issue which is memory latency well this goes beyond pipelining this is an issue in general the problem is that the processor operates at frequencies of you know something like 3 gigahertz four gigahertz.

If you do some memory operation if you want to fetch some data from the memory it takes a substantial amount of time for that data to come it can take hundreds of cycles even though your instruction can execute and about 4-5 cycles in 4-5 nanoseconds, but just because it is trying to get something from the memory, just to get that data it can take hundreds of cycles, right.

So, there is a huge gap between the performance of the processor and the time it takes to get data from the memory. So, what will happen to pipelining in this case let us say that I have an instruction say add R 1, but instead of adding a value like 10 to it, a constant which is which you know I do not need to go to the memory for suppose I am trying to

add to register R 1 data that resides in memory location 1000. So, what would happen over here; what would happen in the pipeline; well you would have the instruction fetch you would have decode and then you would have the operand fetch. So, what does operand fetch do? It fetches the data from the memory how long is this going to take to execute well this is going to eat up about maybe 100 cycles, there it is. So, what happens to the next instruction which was put into the pipeline after this instruction fetch decode let us say the next instruction was add R 1 comma 3,

So, what do I do? Now I cannot execute that instruction; that instruction is waiting also for R 1, right. So, it is also stuck only when this data comes and finally, this instruction gets executed after that can I execute this instruction. So, when we talk about superscalar execution, what are we trying to do over here? We are trying to issue multiple instructions together, I am trying to pick up the 2 instruction and execute them together.

Now if I am doing in order execution, right, what is in order execution in order execution is that I am simply picking up the next instruction which is there in the code and trying to execute; I pick up the current instruction, I issue it, I pick up the next instruction and I have to check; can I issue it simultaneously or not, maybe I can; maybe I cannot; depends on the dependencies depends on various things, right, but I may be able to issue it. I may not be able to issue it, right and you know the chances have being able to issue the next instruction together with the current instruction may be quite small.

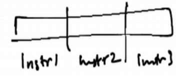
So, how do I deal with that? So, typically what is done in modern day processors is that there is a window that is maintained. So, if this is your code it generally maintains the window of few instruction and it examines all those instructions and figures out that which are the instructions that can be picked up to be executed in parallel. So, if this instruction is picked up, maybe the next instruction is dependent on it. I cannot pick it up, but maybe the instruction after that is completely independent, right.

So, then it will pick up that instruction and issue it together with the current instruction. So, there are lots of intricacies involved which we are not going to get into for instance you can issue instruction a different order, but you have to be very careful that they complete in the order in which the code appears. So, we are not going to get a whole lot into those intricacies and this is actually called out of order issue because it has issued instructions which are not in sequential order as they appear in the code.

(Refer Slide Time: 09:25)

VLIW (Very Long Instruction Words).

- Compiler



Superpipelining	VLIW
More complex h/w	Simple H/W.
Realtime	Offline.
	No view of dynamic
	Static

So, another way of handling the same situation is VLIW; very long instruction words. So, what happens here is that just as in the case of super pipelining, you were trying to issue multiple instructions together, but who was determining which instructions can be executed together, all of that was being done by the processor at runtime, when it was in the code at that time, it was maintaining a window and trying to decide which instructions cannot execute together.

So, that makes the hardware complex, right that makes a logic complex. So, instead another approach is that offload this to the compiler, right, why do not I do it in the compiler. So, when I am compiling the code at that time, I can look at all the instructions and figure out which instructions can be executed together and just club them up together and that is the idea behind VLIW architectures.

So, here you have instruction words which actually comprised of multiple instruction; instruction 1; instruction 2; instruction 3 and so on, right and the compiler figures out that these instructions have no dependency amongst them. And therefore, I can issue them together, right. So, there are advantages and disadvantages of both these approaches. So, we just compare VLIW versus super pipelining, right, being done dynamically; what are the advantages or disadvantages. So, one is in super pipelining, this involves a more complex circuitry whereas, in VLIW, these decisions are not being made dynamically; it is being made by the compiler.

So, the circuitry can be much simpler the hardware can be much simpler again when you are doing super pipelining when the processor is trying to determine which instructions to issue dynamically by itself rate, it is doing this in real time, right, this is in real time whereas, this is offline and because it is doing it in real time. There is a limit to what it can do, because it has to eventually I mean it is looking at the code and it has to issue the instruction in the next couple of cycles.

So, there is a very small window in which it has to make the decisions that which instruction and I am going to pick up to issue simultaneously, right because it is all in real time, but that is not a restriction in when you are doing it in the compiler because in the compiler I mean it is being done offline you can take your own sweet time the compilation is going to be slow, but that is fine, but you can take your own sweet time and you can you know look at a larger window you can try to look at more permutation combination and figure out which instructions can be issued together.

But of course, one major drawback of VLIW as opposed to super pipelining is that it does not have a view of the dynamics state, right that what is currently going on because when you issue an instruction and let us say that the data is not present and you have to go to the memory to fetch that data which takes a substantial amount of time, right based on that you can make decisions of you know which instructions you can issue in which you cannot.

So, also the branch history plays a role. So, again we are not going to get into that, but you know previously whether a branch has been taken repeatedly or not like for instance in a loop when you are executing a loop a tight loop, right you take the branch again and again. So, that you that information is maintained in a branch history table based on which you know you can when you are executing the instruction you can say that you know, I have taken this branch condition I have taken this loop ten times before. So, I am probably going to take it again. So, let me fetch the instructions from there and start working on them. So, this kind of dynamic state is not available to the compiler.

So, whereas that is all this information because it is being done in real time in super pipelining, it is available to the processor. So, it can you know use that information to make decisions.