Computer Architecture and Organization
Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

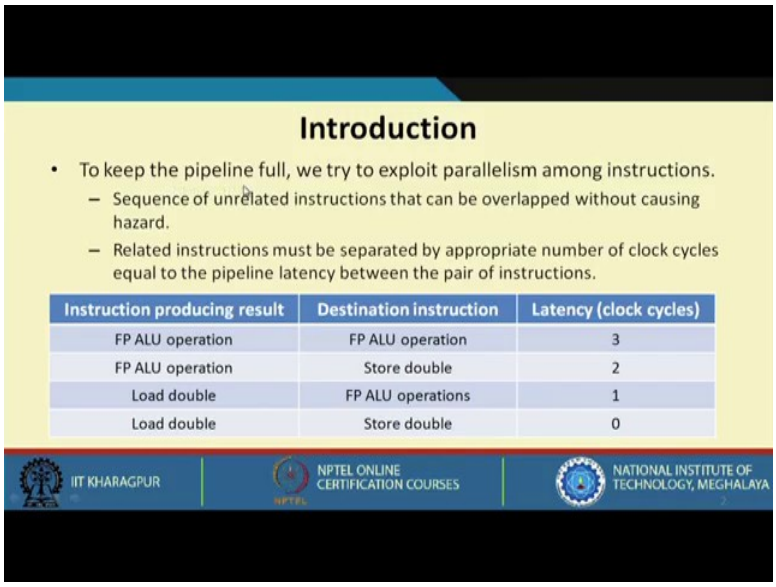
Lecture – 60
Exploiting Instruction Level Parallelism

In this lecture, we shall be discussing about instruction level parallelism. If you recall, we talked about multicycle operations. One thing we saw is that they occupy more number of clock cycles in the EX stage, and if there are data dependencies then even with data forwarding significant number of stall cycles are required.

For the integer operations, earlier we saw that in the worst case only 1 stall cycle is required; here there can be several. So, we can again look at the compiler and give it the responsibility to try an fill up the delay slots.

Now the compiler has to work much harder because now we have so many delay slots. There are many interesting techniques that the compilers use; they use something called instruction level parallelism. Well unless there is parallelism, you cannot move things around that freely. So, the compiler tries to expose more parallelism and then utilize that to reduce the number of stalls. We shall try to illustrate this with some illustrative examples in this lecture.

(Refer Slide Time: 02:11)



Instruction producing result	Destination instruction	Latency (clock cycles)
FP ALU operation	FP ALU operation	3
FP ALU operation	Store double	2
Load double	FP ALU operations	1
Load double	Store double	0

Our objective is to keep the pipeline full to reduce the number of stall cycles as much as possible. For that purpose, we will have to exploit parallelism among instructions. What do we really mean by parallelism? No parallelism means instructions are executed in sequence. In a particular order, second instruction depends on the first instruction, third instruction depends on the second instruction, fourth depends on the third instruction, etc. But if the instructions are independent then we can run them parallel; this means even we can exchange the order of the instructions without any problem.

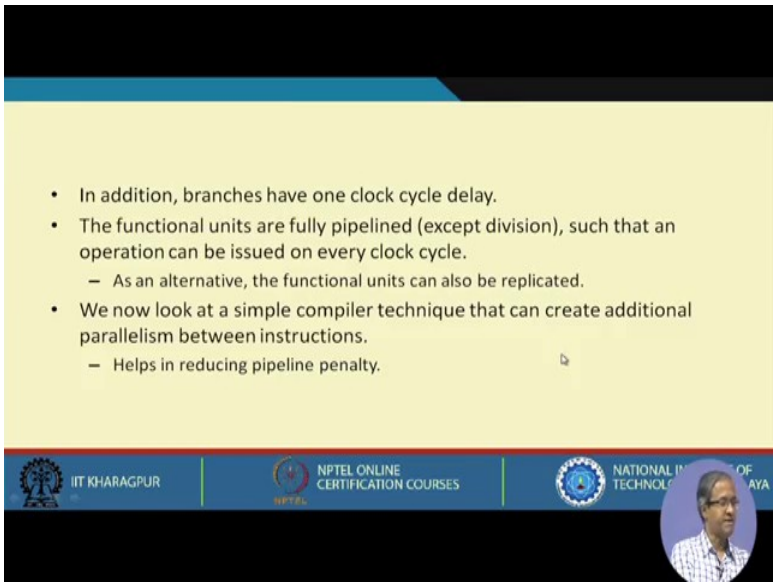
Whenever I can expose more parallelism, I can do two things. I can move instructions around much more freely, and if I have multiple functional units available, then I can try to execute an addition and a multiplication instruction together.

So, when we talk about parallelism, it means sequence of unrelated instructions that can be overlapped and if there are unrelated; obviously, there will not be any data hazards.

There will not be any problem, but if the instructions are related; means the output of one instruction is used as the input by another instruction, then they have to be separated by appropriate number of clock cycles. That means, the latency that depends on the type of the operations. Now this table summarizes the typical latency figures between the broad kinds of arithmetic operations, particularly floating point. When there 2 floating point ALU operations, the latency is 3; these we are assuming.

The various latency values are shown.

(Refer Slide Time: 05:34)



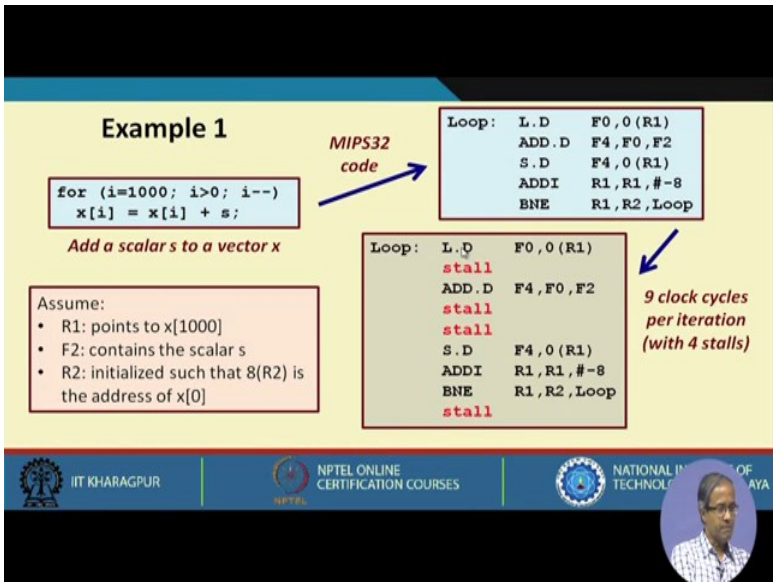
We will incur one cycle delay. The other assumption that we make is: we discussed that the functional units like adder, multiplier are fully pipelined, except the division unit. And because of the pipeline, you can initiate an operation like addition, subtraction, multiplication in every clock cycle.

The alternate philosophy could have been to have multiple functional units instead of pipelining. Let us say the adder unit that consists of 4 stages; we could have used 4 adders, but that unnecessarily would increase the cost 4 times. So, pipelining is a much more elegant method where cost is not increasing that much, but effectively you are getting 4 times throughput approximately.

We look at a compiler technique, where some additional parallelism can be created. Suppose I have written a program; just by seeing the program some parallelism can be identified. A compiler will say that is fine, but let me try to generate some more parallelism.

If it is done then the pipeline stall cycles can be reduced quite significantly.

(Refer Slide Time: 07:27)

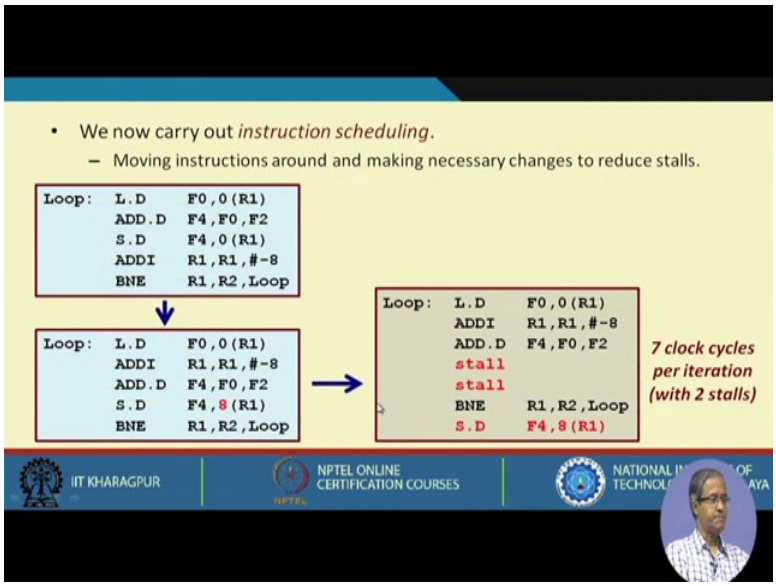


Let us take an example like this, with which we will be illustrating the process. We assume that there is a vector; that means, an array of size 1000, s is a scalar number, you are adding s to all the elements of the array. This is the C code, where we are assuming register values like this. That is, R1 points to the last element of the array, F2 contains the scalar s, and R2 is pointing to the element that is just before the first element of the array. That means, if I add 8 with R2, it will be pointing to x[0]. So, actually R2 is pointing to 1 element before x[0]. The corresponding MIPS32 code is shown, assuming that R1, F2, R2 are a loaded like this.

Let us analyze this code first. You see there is a load followed by an add. So, according to our earlier table, we will incur one stall. Also, add is followed by store, for which we will incur 2 stalls.

For the branch, there will be 1 stall cycle as usual. So, you see that for this loop that consists of 5 instructions, for every loop it will actually require 9 clock cycles; 9 clock cycles per iteration with 4 stalls this is what this code gives.

(Refer Slide Time: 11:01).



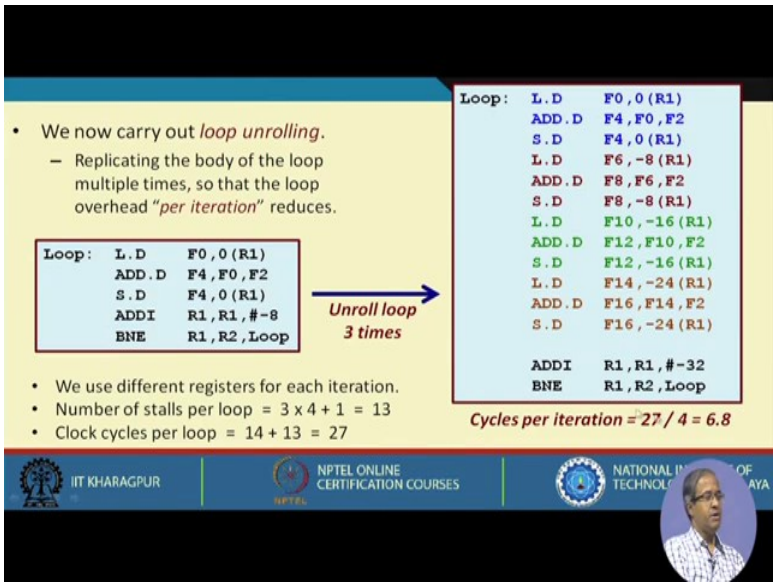
Let us now try to do instruction scheduling that we learnt earlier. Let us keep the same code, let us try to move some instructions here and there and try to reduce the stalls. This was our original code. Let us do instruction scheduling like this. The first thing is that this ADDI instruction we are moving it here, such that between load and add the stall disappears.

So, now this store becomes 8(R1) because already we have decremented. So, this 0 is change to 8, then we have the add followed by store and BNE, this was the modified thing. Another thing we do, this store is moved to after BNE to fill up the branch delay slot, this also you can do because the store and branch are independent.

If you do these then you see for this loop, there will 7 cycles per iteration, and there are 2 stalls. Our program had 1000 iterations in the first version, there were 9 clock cycles per iterations which means there were total of 9000 clock cycles required, but now in this version where having 7 clock cycles per iteration.

So, from 9000, we have brought it down to 7000. But you see with this code, however hard the compiler tries, it cannot improve any further. So, what is the way out? The way out is to do something called loop unrolling.

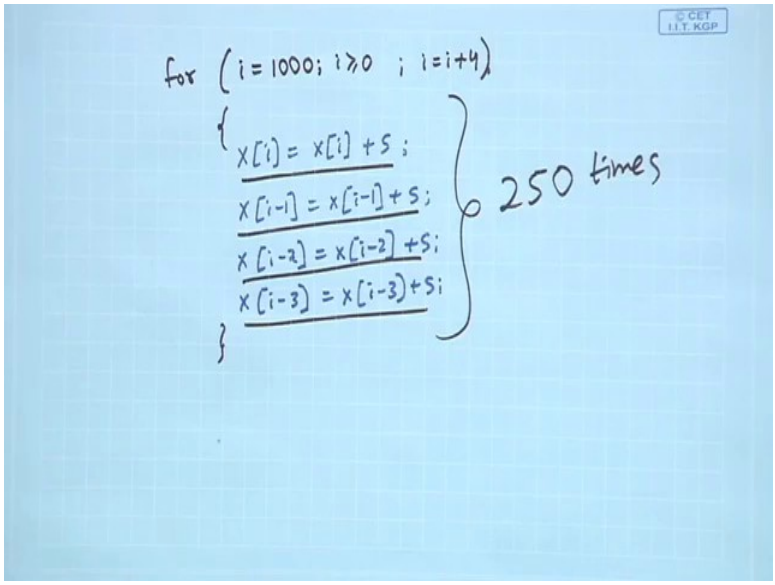
(Refer Slide Time: 13:21)



Let us go back to the original loop. The original loop was looping 1000 times. Now what I am doing; there are 1000 elements, I have written a small loop for adding 2 numbers, I am repeating 1000 times.

In the modified version; what I do? I unroll the loop --- this is called unrolling. Unrolling means see earlier I was only adding x[i] with s.

(Refer Slide Time: 14:17)



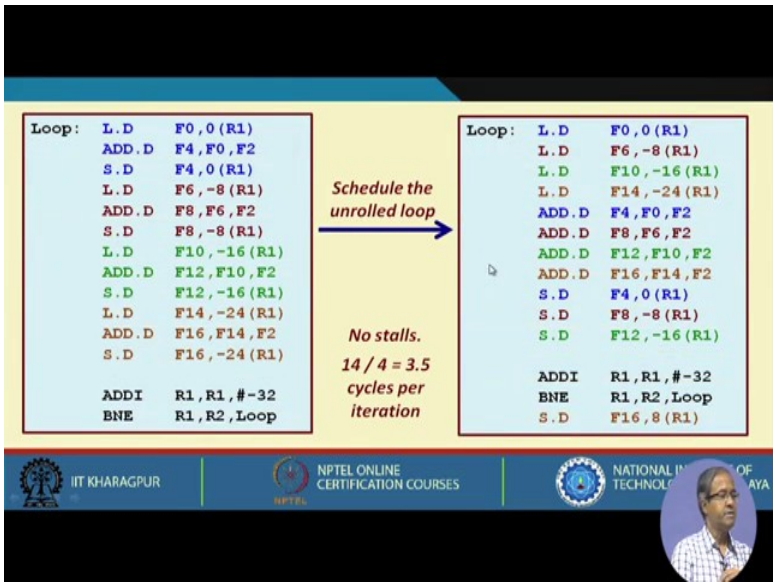
Now what I do? I write x[i] = x[i] + s, then write x[i-1] = x[i-1] + s, then x[i-2] = x[i-2] + s, and x[i-3] = x[i-3] + s. I unrolled the loop 3 times to make it 4 copies.

There is no data dependency between these four blocks right this is called loop unrolling.

The stall cycles in the unrolled version are also shown. In this version, cycles per iteration will be 27 / 4 = 6.8.

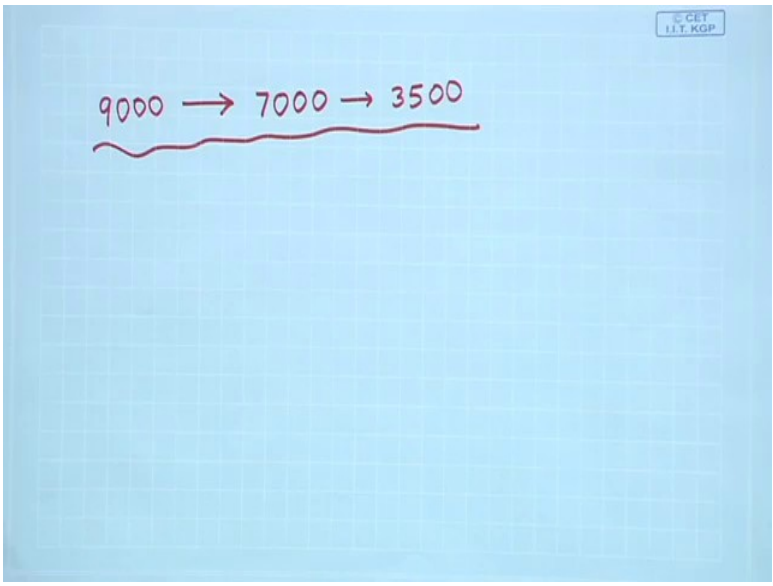
Now you see we have exposed so much parallelism. Now you can move instructions around much more freely to eliminate stalls where possible.

(Refer Slide Time: 18:07)



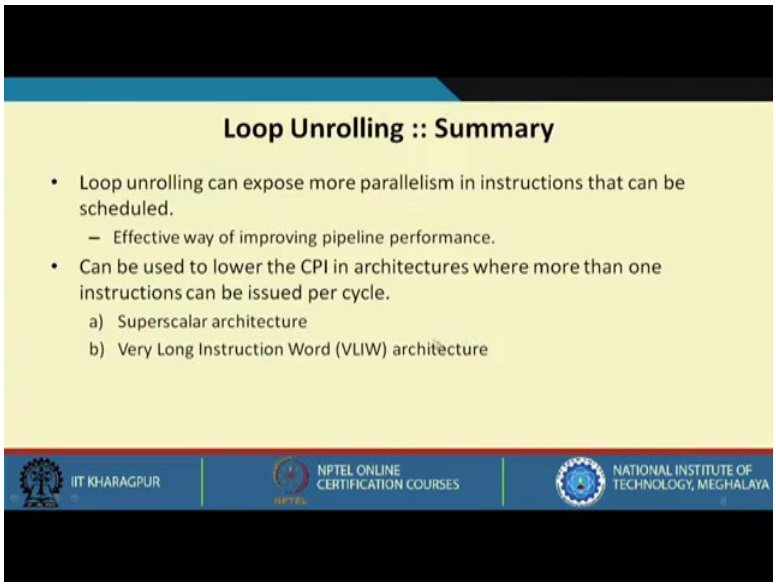
You see in this version, there are no stalls because there are no dependencies. So, no need for any additional stall cycles. There are 14 instructions, which require 14 clock cycles. This gives 14 / 4 = 3.5 cycles per iteration.

(Refer Slide Time: 19:58)



So, you see there is a quite drastic reduction in the number of clock cycles.

(Refer Slide Time: 20:44)

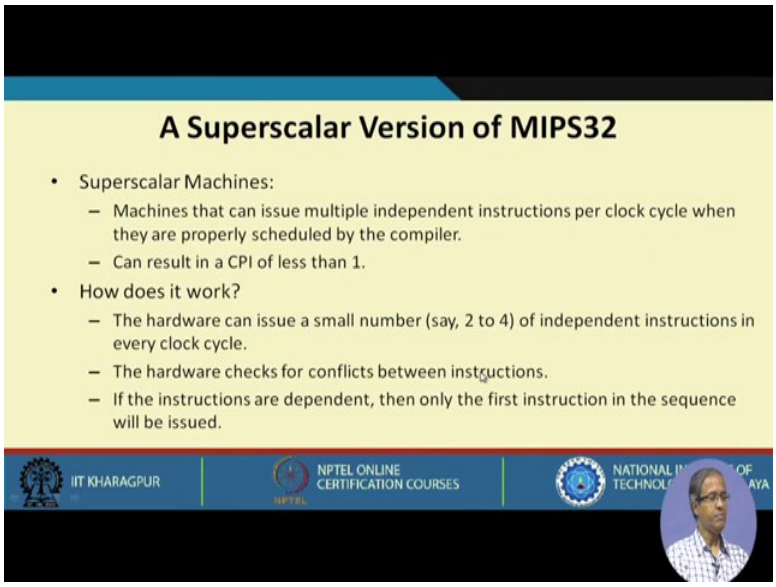


So, if the compiler does this instruction scheduling, it can bring down the clock cycles to a great extent. To summaries, loop unrolling can expose more parallelism.

So far in the pipeline, we said that our ideal CPI was 1, but now we are talking about machines where this CPI value can be less than 1; what does that mean? We are using some kind of parallelism.

Let us say two instructions are executing together. In every cycle, we are executing 2 instructions. The CPI will be 0.5. Broadly there are 2 kinds of approaches we will be talking about, one is called superscalar architecture, other is called very long instruction word or VLIW architecture.

(Refer Slide Time: 22:04)

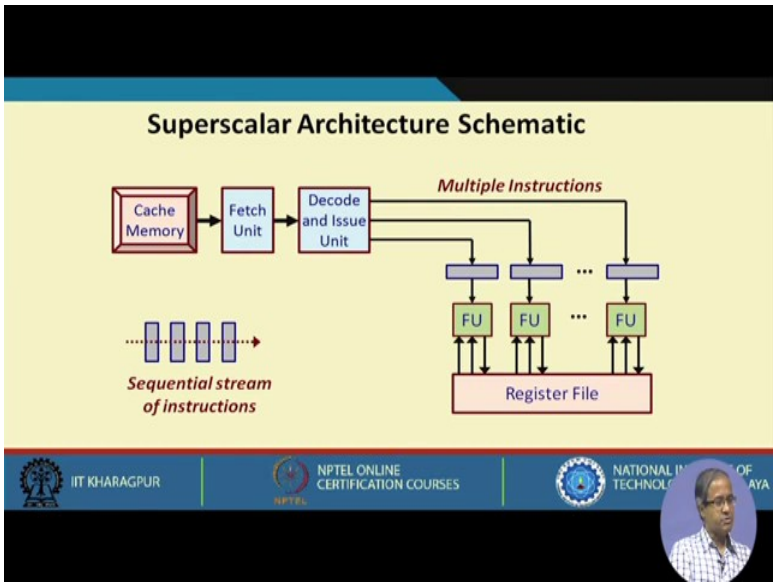


First you look at a superscalar version of MIPS32. A superscalar machine means it is a computer system, which can issue multiple instructions in every clock cycles. You imagine a superscalar version of MIPS where there are 2 pipelines.

In every clock cycle we will be fetching 2 instructions, and feeding them to the 2 pipelines. In general, the number of such pipelines can be more; than it can be 4 even higher, this is what is meant by superscalar.

So, machines can issue multiple independent instructions; if there is dependency you cannot start them together. The hardware can also check for conflicts whether they can start together; if there is a conflict then only one of the instruction can be issued and the others have to wait.

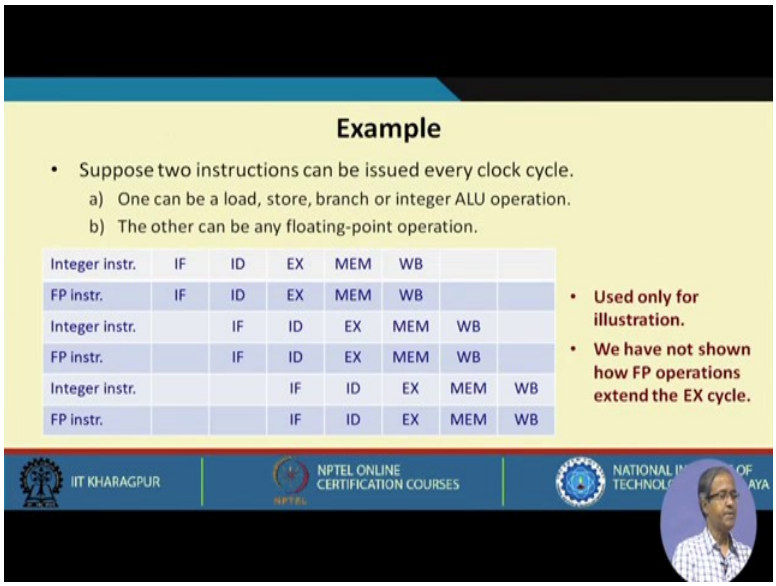
(Refer Slide Time: 23:41)



Superscalar architecture schematically looks like this. The fetch unit will be more sophisticated. From the cache memory it will have to fetch more than one instructions every cycle. Decode and issue units will also be decoding several instructions together and there will be multiple pipelines, I am calling them as functional units. You can imagine as if they are independent pipelines accessing maybe a common register file.

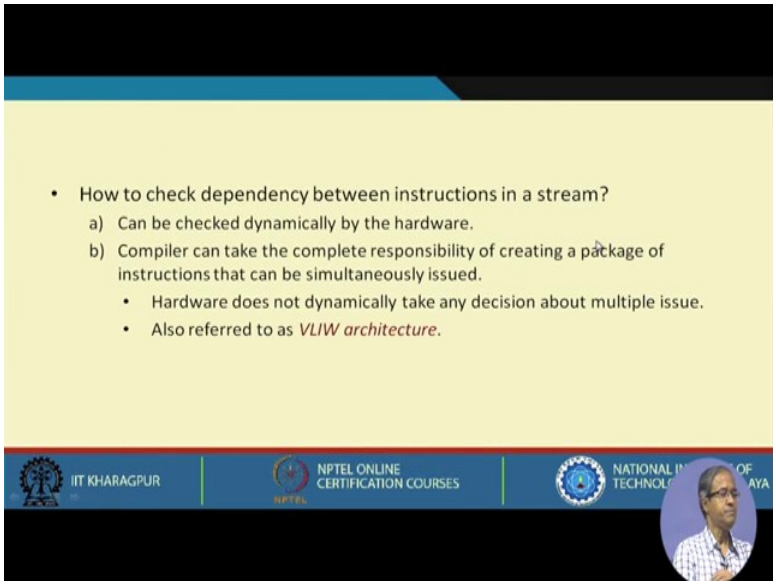
Depending on the capability of the machine suppose there are 4 such pipelines. So, 4 instructions will fetch together, they will be issued 4 concurrently to the 4 pipelines. So, conceptually it is like this. The instructions are stored sequentially, then fetched much faster and 4 of them are fed to the 4 pipelines. This is the concept of superscalar architecture.

(Refer Slide Time: 24:54)



Consider an example for a superscalar architecture with 2 functional units, one of them can handle load, store, branch, and integer operations, and the other functional unit can handle floating-point operations. In every clock cycle, you can start 2 instructions, 1 integer and 1 floating-point. Here we are not showing multicycle operations because in general for floating point there will be multicycle; just for the sake of illustration we are showing like this.

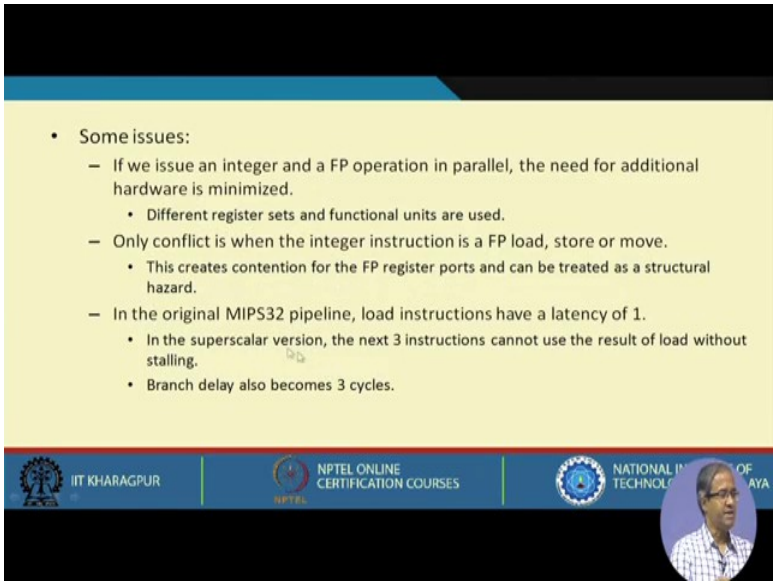
(Refer Slide Time: 25:42)



The dependency between instructions will be checked dynamically in the hardware, this is important. As an alternative, we can again give some responsibility to the compiler, but that is not for superscalar architecture, but for the other kind of architecture VLIW. So, what the compiler can do? See for superscalar; instructions are fetched, the hardware is dynamically checking whether there are conflicts; if no conflicts they are fed to the pipeline together.

But now I am saying there is another kind of architecture called VLIW. Here the compiler is trying to create packets of instructions, like each packet will consist of 4 instructions and the compiler will ensure that the 4 instructions are such that there is no conflict between them. The hardware did not check for anything here.

(Refer Slide Time: 26:52)

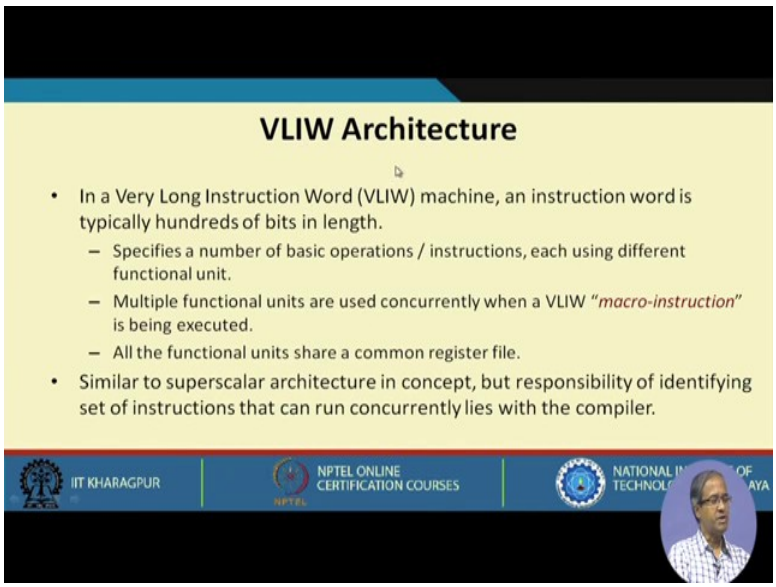


So, this is the idea. Now there some issues like for example, if we issue an integer and floating point operation in parallel because they use different register sets and different functional units, additional hardware required is less because they do not normally share register sets. The only conflict is when the instruction that is handling integer unit is a floating point load where the loaded value has to be loaded into a floating point register. So, that can lead to a hazard.

Another issue is that for the original MIPS pipeline; whenever there is a load instruction latency was 1, but for the superscalar version, it is not 1, it will be 3 because not only the next instructions; the next 2 instructions also have to wait because of that latency of 1

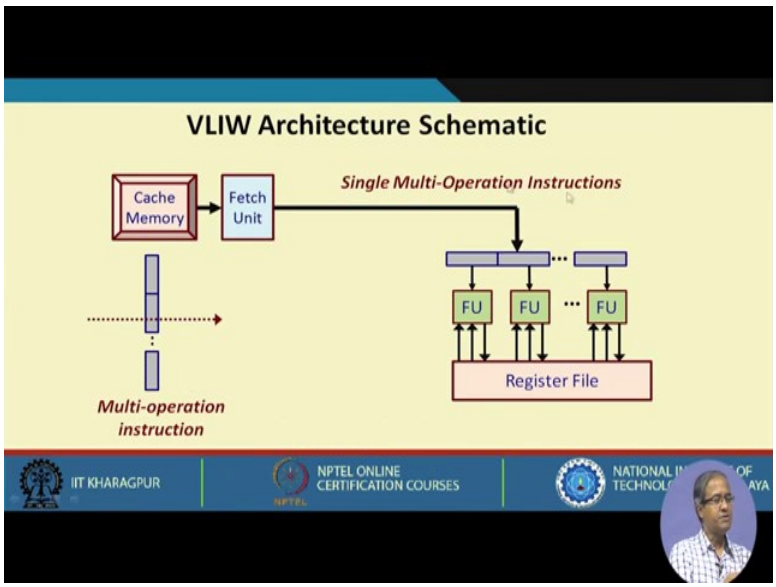
cycle. So, now latency will be 3, 3 instructions will have to wait rather than 1. Similarly branch delay will also become 3 cycles and not 1.

(Refer Slide Time: 28:11)



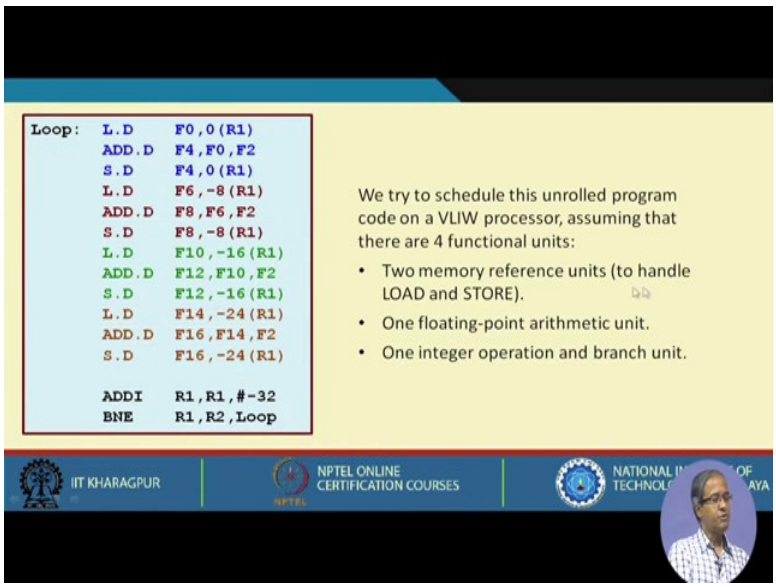
An instruction word can store several instructions together; several instructions together is referred to as a macro instruction, may be 2 or 4 like that. There are similarly several functional units, the compiler will be generating these macro instructions, will be trying to group the instruction together. The responsible to identify the set of instructions that can run concurrently lies with the compiler.

(Refer Slide Time: 28:50)



So, you see architecture wise here it looks similar, but now instructions are coming as packets and not one at a time.

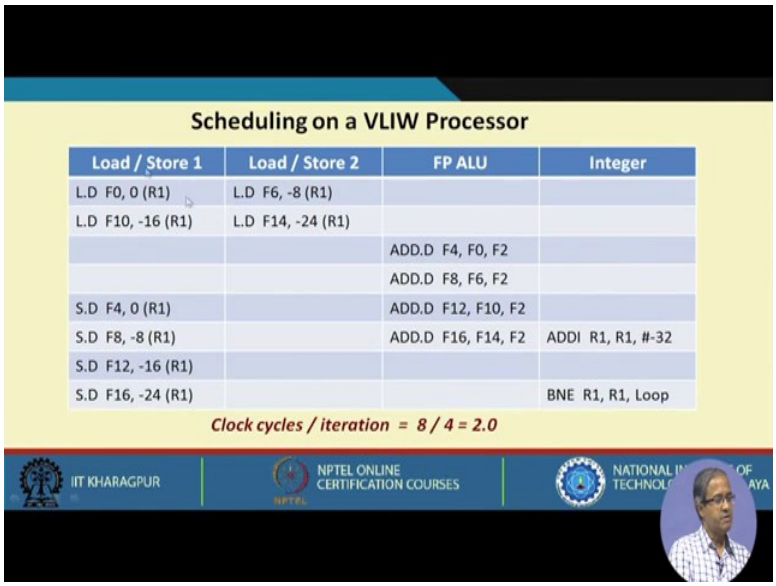
(Refer Slide Time: 29:20)



Let us see how we can run the unrolled code on a MIPS processor and schedule it, where we assume that the processor has 4 functional units, where 2 of the functional units are memory reference units that can handle load and store. There is one floating point and one integer operation which can also handle also branch.

We see for load and store, there are several instructions for floating point; there are these add instructions and for integer operation there is only this branch and this add immediate, let us see how it can be scheduled.

(Refer Slide Time: 30:13)



Here I am showing one possible scheduling. These are the 2 load/store functional units. This is the floating point functional unit, this is the integer functional unit. The loads can be placed together in the first 2 cycles. After the loading is done, you can put the adds here then stores. There will be delay up to 2 cycles as shown. s.

So, here the CPI becomes 2.0 because of the parallelism that is supported by 4 parallel hardware units. For processing 1000 numbers number of clocks will become 2000.

With this we come to the end of this lecture. We have looked at some of the ways of parallelizing or speeding up the basic MIPS32 processor that contains not only integer units, but also floating point units.

Nowadays many of the processors that we see around us are actually based on superscalar architectures.

Thank you.