# GPU Architectures and Programing

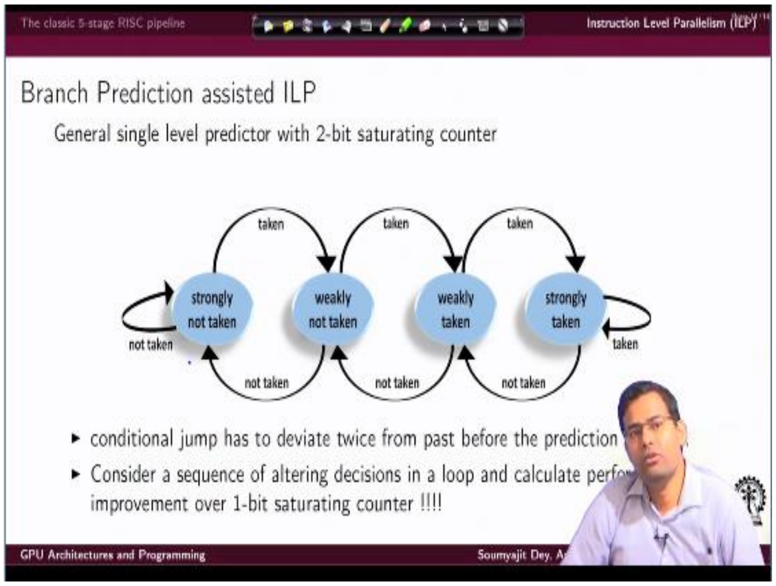
**Prof R. Soumyajit Dey**
**Department of Computer Science and Engineering**
**Indian Institute of Technology, Kharagpur**

## Lecture No. 4
## Review of Basic COA w.r.t. Performance (contd.)

Hi. So, we were discussing about different possibilities of branch prediction. In order to increase the available parallelism in a system or reduce pipeline stalls. That can happen due to control hazards that is branch hazards. So we were discussing about this 1 bit branch predictors, and we have thought that okay there can be generalized right? I can have a 2 bit branch predictor essentially a branch greater.

Which will try to predict a branch decision based on more amount of branch history. So all that I want to be predicted does is, it just looks at the previous decision

**(Refer Slide Time: 01:01)**



Branch Prediction assisted ILP

General single level predictor with 2-bit saturating counter

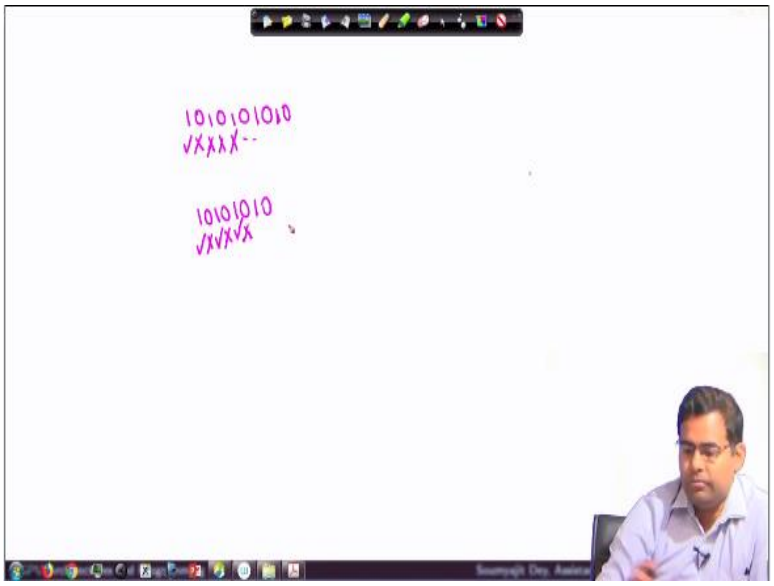
- conditional jump has to deviate twice from past before the prediction
- Consider a sequence of altering decisions in a loop and calculate perfo improvement over 1-bit saturating counter !!!!

And repeats it a 2 bit branch predictor. Looks like the last two decisions and tries to do something like that right? So this is an example of a branch predictor designed with a 2 bit saturating counter. So, I mean, just to motivate. Why does a good thing. First of all we have already understood earlier that there are 4 states strongly not taken weakly not taken weakly taken, and strongly taken state.

So strongly taken means, I am always deciding to take the branch and this we have already discussed. If once I go wrong in my decision I come to weakly taken state, here also I want to again take the branch. Again, if I go wrong in my decision. I come to weakly not taken state. Here I am always predicting that I will not take a branch. If I, if I am correct, that means a branch is really not to be taken then I go to the strongly not taken state.

Where also my decision always is to not take the branch. And I shift from strongly not taken to weakly not taking if I go wrong once that is a branch is taken. I shift from weakly not taking two weakly taken, if I go wrong again. That means I predict that the branch is not taken but it is again taken, and so on so forth. So, just to observe the performance of the system. Let us go back and think of the performance of a one bit branch predictor, you know, worst case scenario.

**(Refer Slide Time: 02:27)**



So, worst case scenario would be something like Suppose I have a problem which is executing with sequence of,I mean the actual brunching decisions, denoted by 1s and 0s, like this. They are going to be alterations. So let us say I am executing a loop inside which there are two branches is always the case that the first branch is taken and the second branch is not taken. So that gives me this pattern. How do you think, then the one bit branch predictor will work.

So let us say, is initially is in the second state, so it will predict this correctly. And then it will think that okay in the next branch also I should take the branch. So this will go wrong. Now it seems that it has gone wrong. So in the next state it again wants to not take the branch, whereas the real thing is that the branch has to be taken, so it will go wrong, and the behavior will continue right? So, you are always predicting wrong.

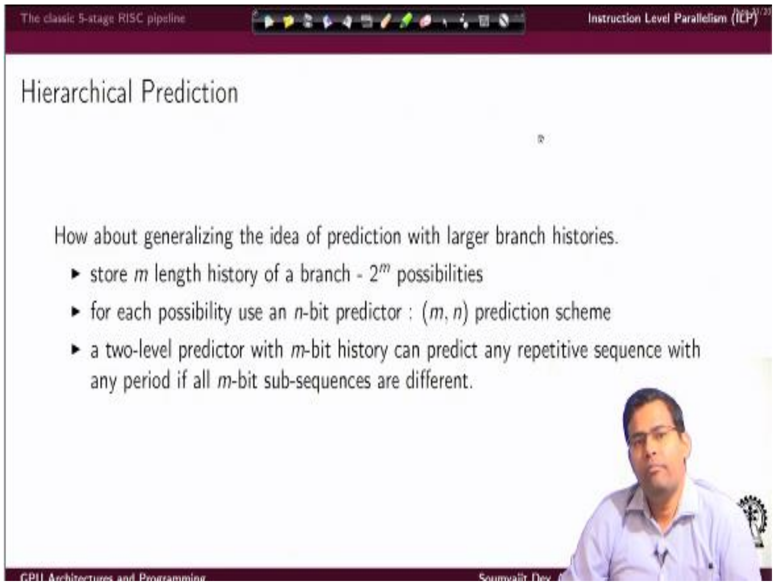
How about using this to 2 bit branch predictor  here. Of course, the behavior will depend on my, I mean initial state. let us assume that I am in the strongly taken state. Okay. So, assuming that. Let me again write the sequence here. So I am in the strongly taken state, that means, this one I predict that I will take and I am correct. Then here.Again, I am in the strongly taken state. But the real branch is not supposed to be taken.

So I go wrong, because I am always thinking I will take it. So then I will also have a state change right so assuming that in the automata You'll see that I am now going to switch from the strongly taken state to the weakly taken state, because, although I thought that the branch is taken, it is really not taken, but still my decision is, since is a weakly taken state for the next branch am again going to take the branch, right?

So, I will go correct here. So then, that also means that I switch, since the branches really taken a switch to the strongly taken state right? So, again, for the next prediction, it will go wrong. And then again I go correct, and they go wrong like that. So, I have an improvement in performance here over a one bit saturating counter. So in that way I can see that with addition of some history of information. I get an improvement in performance.

Now, of course, I can keep on making a more complex branch predictor by looking into more amount of history and taking suitable logical decisions. Or, alternatively.

**(Refer Slide Time: 05:39)**


The classic 5-stage RISC pipeline

Instruction Level Parallelism (ILP)

## Hierarchical Prediction

How about generalizing the idea of prediction with larger branch histories.

- store $m$ length history of a branch - $2^m$ possibilities
- for each possibility use an $n$-bit predictor : $(m, n)$ prediction scheme
- a two-level predictor with $m$-bit history can predict any repetitive sequence with any period if all $m$-bit sub-sequences are different.

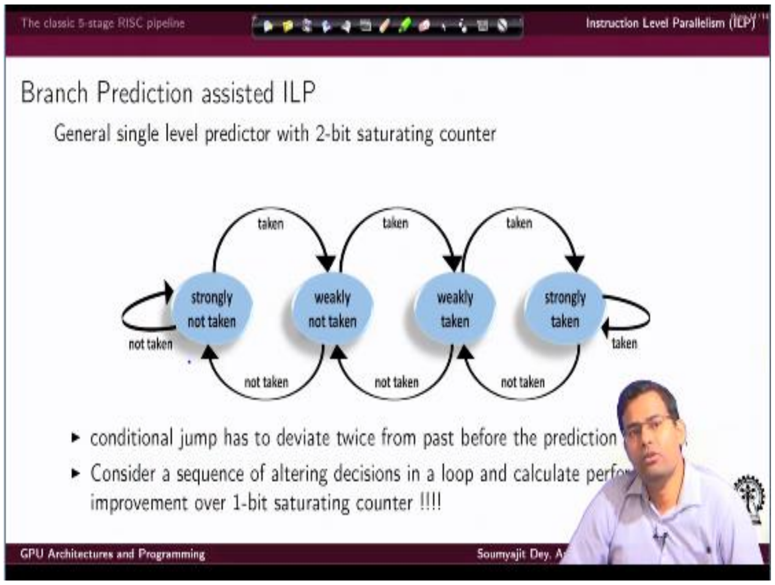
GPU Architectures and Programming

Soumyajit Dey

I can make the branch predictor even smarter and go for what we known as a hierarchical predictor. So if we generalize the idea, it would be something like this, that we look into branch histories and take decisions. Now, suppose I am always looking at an M length history. So there are 2 to the power m possible branch histories, so I store these different. This is history in a location, maybe a shift register.

So, I have 2 to the power m possible values of this history. And for each of these situations that means for each of 2 to the power m possibilities. I have one n-bit predictor. So just to understand back.

**(Refer Slide Time: 06:26)**


The classic 5-stage RISC pipeline

Instruction Level Parallelism (ILP)

Branch Prediction assisted ILP

General single level predictor with 2-bit saturating counter

- conditional jump has to deviate twice from past before the prediction
- Consider a sequence of altering decisions in a loop and calculate perfo improvement over 1-bit saturating counter !!!!

GPU Architectures and Programming

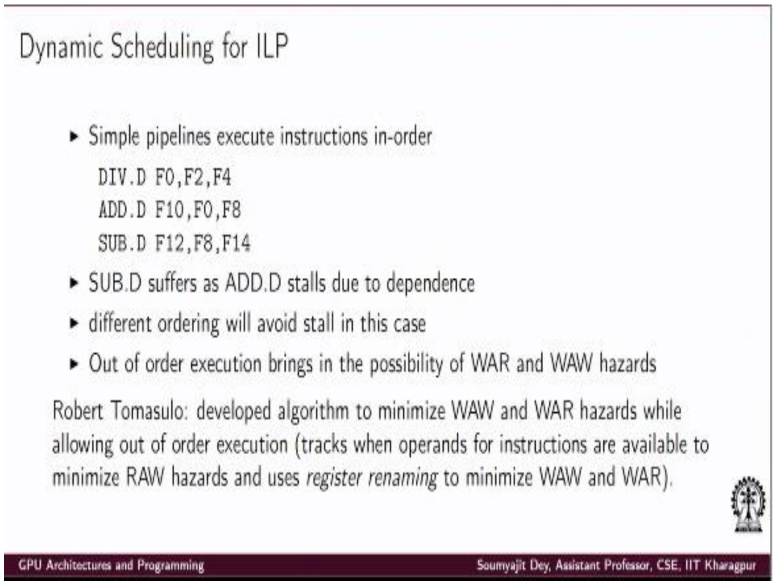
Soumyajit Dey, A

Here what we are doing. We are looking at the last decision and doing a prediction. So the prediction decision is based on 2 bits right? It is a 2 bit saturating counter decision and but I am looking at the last decision right? So, I can also do is, I can look at the history of m length, and for each possible search sequence. I'll take the decision based on another n-bit predictor. So this is a combination scheme.

I have a hierarchy, first I see what is the branch history, and then I look at for this history for this specific history. What is the state of my n-bit predictor among those weakly taken strongly taken less weakly taken more strongly taken. They will they will be more steps because if n is greater than 2 and all that. So, I mean, in that way I can take a more informed decision. It is known that if I have a two level predictor with an m-bit history.

It can predict, any repetitive sequence with any period. If all the m-bit sub sequences are different. Let us understand what it means, suppose I am doing a two to level predictor. And my choice of m means I am always deciding based on an m-bit history. It can predict any sequence which will repeat with some period. If all the corresponding sub sequences, these m-bit sub sequences are different. that means, for the m-bit subsequence. The decision is always unique.

**(Refer Slide Time: 08:15)**



## Dynamic Scheduling for ILP

- Simple pipelines execute instructions in-order

```
DIV.D F0,F2,F4
ADD.D F10,F0,F8
SUB.D F12,F8,F14
```

- SUB.D suffers as ADD.D stalls due to dependence
- different ordering will avoid stall in this case
- Out of order execution brings in the possibility of WAR and WAW hazards

Robert Tomasulo: developed algorithm to minimize WAW and WAR hazards while allowing out of order execution (tracks when operands for instructions are available to minimize RAW hazards and uses *register renaming* to minimize WAW and WAR).

GPU Architectures and Programming    Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So that was a bit of diverging into the idea of branch predictors. And looking into some amount of the intricacies, if not all, and we are also not getting into the details because, in general, paralegal prediction is a very complex phenomenon. There are a lot of other predictors that tournament predictors you can just go into those details. If you are interested, from an advanced architecture point of view.

Now, the next important topic here for us is the issue of dynamic scheduling for instruction level parallelism. So what is that. Now, we know that simple pipelines execute instructions in order. Like, I have the sequence of instructions. Divide for double type, ADD SUB and all that. Now look at the operants here, from which it is very easy to decipher that  the subtract instruction will suffer unnecessarily.

As the ADD instruction gets stall to to dependence. So what is that that that the DIV instruction is going to write to  F0. Now this is again the read operand  for ADD.  So, this is something we have already discussed earlier, it is a read after write dependency, due to which ADD will stall. Now SUB has no dependency on ADD. I mean, it is basically reading on the only common thing is, F8.

But it is a read after read so there is not not not any kind of hazard or stall that is going to happen. So there is practically no reason why sub should suffer due to end getting stalled with the preceding instruction DIV. However observed that if I do an alteration here in the ordering. I can avoid the stall, because I can just execute because ADD will get stalled here I can just execute so me before at the end there is no stall.

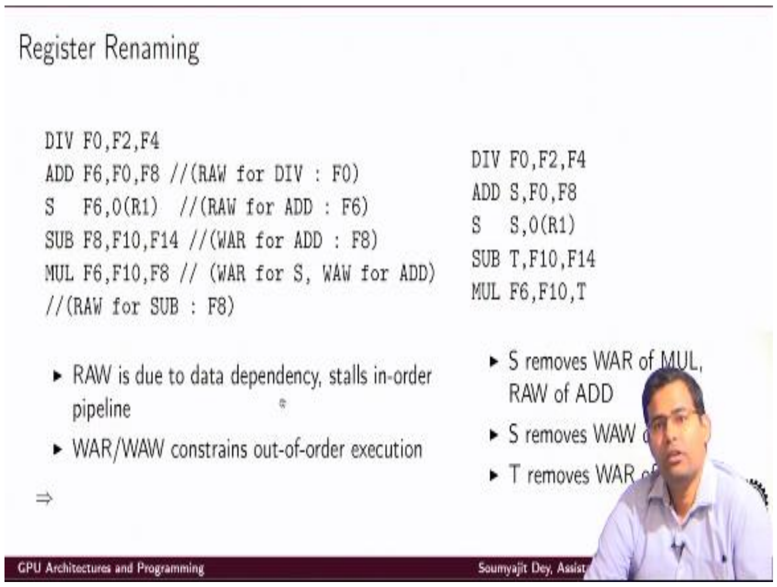
Now, this is a good thing for this specific case because SUB has no dependency and observed that I can have a recognizer system in the hardware, which can actually identify that yes there is no dependency and this can go before this, right? So, I can have this kind of modified real, or I can have this kind of reordering of instructions that is executing instructions, out of order to get more instruction level parallelism, and increase the effective CPI right?

However, there are issues, I cannot do it in an ad hoc way, in this case it is good. But if we just think of the earlier issues, like, as we discuss the hazards of type WAR or WAW, those can come in. If I am doing out of order execution. So, I have to be very careful here, that when I am executing out of order. I should  choose the order in such a way that these possibilities of WAW and WAR type hazards.

Do not creep in. In this way, there is a very famous algorithm by Robert tomasulo well known as store over tomasulo  algorithm solos algorithm, which is basically a way of choosing instructions that would execute out of order by minimizing these  WAW and WAR type hazards and resulting stalls. So essentially the algorithm will track when occurrence for instructions are available to minimize this hazard.

And they actually employ some optimization techniques will known as the register renaming to minimize this kind of hazards.

**(Refer Slide Time: 11:58)**



## Register Renaming

```
DIV F0,F2,F4
ADD F6,F0,F8 //(RAW for DIV : F0)
S   F6,0(R1)  //(RAW for ADD : F6)
SUB F8,F10,F14 //(WAR for ADD : F8)
MUL F6,F10,F8 // (WAR for S, WAW for ADD)
//(RAW for SUB : F8)
```

```
DIV F0,F2,F4
ADD S,F0,F8
S   S,0(R1)
SUB T,F10,F14
MUL F6,F10,T
```

- RAW is due to data dependency, stalls in-order pipeline
- WAR/WAW constrains out-of-order execution

⇒

- S removes WAR of MUL, RAW of ADD
- S removes WAW
- T removes WAR

GPU Architectures and Programming Soumyajit Dey, Assist

So let us look at an example of this register renaming. We are not I mean just to be very clear we are not getting into tomasulo algorithm. There's quite a lot of stuff to be covered right now for our purposes not also necessary discussing about to the register renaming option. So if you look to the code in the right hand side. As you can see, we have introduced two new registered options S and T.

And there is some significant modification that has been done, into the original code of the left hand side,the modification is as follows the add instruction that which is the second instruction. Earlier it was writing to the register F6 now it is writing to another register S. And similarly, F 6 has also been replaced by S in the next instruction, there is a stall instruction. And in the sub instruction, which was writing in the register, F8.

This is being removed, and there is a new register T, in which is writing.And similarly, in the subsequent instruction MUL also, F8 has been replaced by T. So, this helps in removing several of the possible hazards that can come in. So let us look into the issues that get solved by doing this register naming operation. So as you can see this, including inclusion of this register S removes the read after write dependency of the multiply instruction.

Now, where is that read after I dependency. So, this multiply instruction was reading from F8. Whereas, and and F10, and it was writing to F6 right? It was writing to F6. Now, what is happening is this read after write was dependent essentially on this F6, this S, the store instruction. Now, let us explain that once again. So as we can see, this multiply instruction is writing to F6,whereas the store instruction is reading from f6.

Now, this is a read after right, which would mean the store instruction, cannot be executing after the multiplied there cannot be any ordering and reordering possible between multiply and stall. But now, once I make them, independent, that is, this S, is being used here instead of F6 for the stall instruction. That means there is no read after right i mean right after sorry write after read dependency it in the multiply.

And the story instruction so now the story instruction is reading from the register is and writing to the location in our one, whereas the multiply instruction is writing to F6. So now, there can be an instruction reordering between this store and this multiply. So in that way. This new register S removes this read after write of sorry right after read dependency of multiply on the store instruction.

So essentially this issue gets resolved, as we as I am pointing it out here this issue of right after read is getting resolved. Now there are some more similar issues that are getting resolved. For example, there was a right after right issue also, that is, as we can see that F6 is being written by multiply. And F6 is also being written by ADD. So, earlier there was a strict ordering that ADD should happen before multiply, but since.

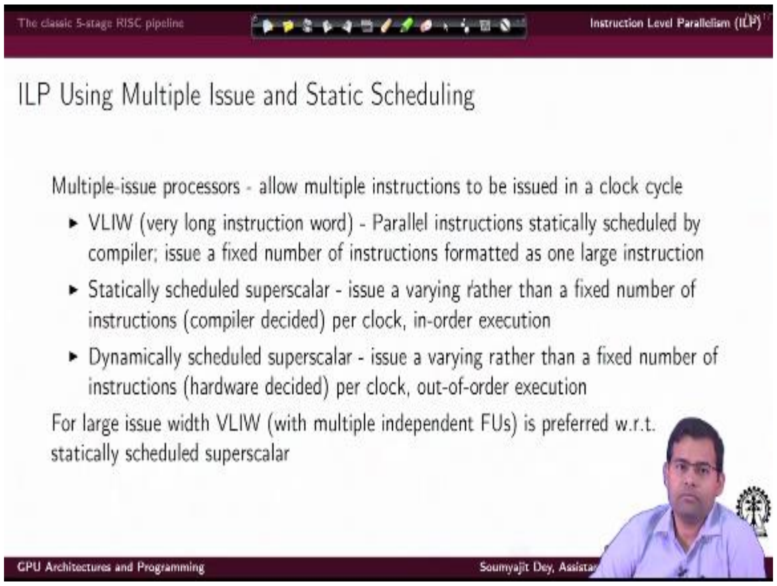
Now that is also being changing right because ADD is writing to S  and multiply writing to F6. So, they are different registers so in that way, this register renaming of S removes the right after right dependency of multiply also. Now what else is getting resolved. Let us see, because we have also included some other register which is the T  register. So, earlier there was this right after read dependency of the SUB instruction.

So it was writing to F8, and F8. This substantial instruction was writing to F8. And F8 was read from by the ADD instruction. So again, there was no reordering possible between ADD and SUB. But now, since SUB starts writing to T. There is a possibility of reordering between ADD and SUB instruction. So in that way. This register renaming by T removes the. So in that way this register removing renaming of T removes this right after read dependency of SUB.

Because it was writing here to F8 and Addition was happening from F8 also. But now, Addition is have a reading from F8, but SUB is writing to T. So, they can be ordered differently. So in that way. Several of these dependencies gets removed with with this register renaming operations. And that creates a case for suitable really I mean, in ordering reordering of instructions which can help in increasing the effective parallelism.

If there is a possibility of executing multiple instructions in multiple pipelines in parallel. So we are just trying to show that these are the possibilities here. This example has been also taken from the classic book by Hennessy Patterson similar to most of the other examples we discussed in this introductory slide. And we are just trying to point out that with this kind of optimizations. They can be easily being carried out by compilers. And once they're turned, they really help in exploiting more amount of parallelism.

**(Refer Slide Time: 18:03)**


The classic 5-stage RISC pipeline
Instruction Level Parallelism (ILP)
GPU Architectures and Programming
Soumyajit Dey, Assistan

ILP Using Multiple Issue and Static Scheduling

Multiple-issue processors - allow multiple instructions to be issued in a clock cycle

- VLIW (very long instruction word) - Parallel instructions statically scheduled by compiler; issue a fixed number of instructions formatted as one large instruction
- Statically scheduled superscalar - issue a varying rather than a fixed number of instructions (compiler decided) per clock, in-order execution
- Dynamically scheduled superscalar - issue a varying rather than a fixed number of instructions (hardware decided) per clock, out-of-order execution

For large issue width VLIW (with multiple independent FUs) is preferred w.r.t. statically scheduled superscalar

So, as we have discussed earlier that there can be several possible ways to do these kinds of optimizations. They are there they I mean they can be done with hardware support as well as software support by software support we mean smart compiler can do these kinds of operations. I mean, by operations I mean deciding, which instructions are going to be issued in parallel, or not. Also, a smart hardware can do the same.

Now, it depends on you. We like to tackle this problem of deciding independence of instructions. That we sleep well at the compiler level or the personal level. Now of course, whatever can be done at each level is also different. I mean, just to take a case like when I am trying to do some optimization by the compiler, the compiler is not aware of the arguments that are provided to the program.

So all that we could do is some static optimization by looking at the problem, more or less, whatever we have discussed is all static optimization because we are not. I mean, using any concept of dynamic optimization, which is based on what is the value in the register. We are just looking at the dependencies and the hazard. So these are all static optimizations which can be done by a compiler, or can be done by a smart hardware.

So, depending on where you want to do this. There can be several different possible processor design styles. So, When we are looking into the issue of how I can resolve dependencies among instructions and issue multiple instructions to be executed in parallel. So, based on this classification, we have the following three possibilities here. The first one is the VLIW style of execution VLIW stands for very large instruction word.

So this is a style of micro architecture design, where the effective parallelism is exploited by the compiler. So, the compiler decides which are the instructions that should execute in parallel. And so essentially the compiler schedules. These instructions and they're parallel execution issues, a fixed number of instructions formatted as one large instructions so essentially the compiler will look at the actual instruction stream.

And come up with packets of large instructions in each packet kind of stickers together. A set of independent instructions which can be executed in parallel. And that is fed to multiple independent functional units so that they can execute in parallel. So here. This is more of a compiler intensive approach, the parallelism is decided by the compiler, the hardware is blindly following it.

And executing the instructions in parallel using multiple functional units which are already present there in the VLIW style of architecture. Now, as example there are well known processors, which do digital signal processing, which are designed based on this philosophy. So, apart from this VLIW style of processors, we also have the other two options which is statically scheduled superscalar and dynamically scheduled superscalar.

So by statically scheduled we mean that the compiler decides which are the instructions that are to be issued in parallel. However, the execution of this instructions are in order. So, we have by pipelines executing the instructions in parallel. But the compiler still decides which are the instructions that will be issued the instruction issue with may very coming to the last option which is dynamically scheduled superscalar here.

The hardware decides instead of the compiler that which are the instructions that are going to be issued in parallel. Also the other important thing is the hardware execute this instructions, out of order. Now, if the design philosophy is that we should have a very large issue with, then VLIW is preferred. With respect to statically scheduled superscalar, simply for the reason that both approaches require support from the compiler.

And if we are really deciding that will the compiler should be packetizing or doing the issuing of parallel instructions. The VLIW makes sense because of the large issue, because if we are keeping large issue it. But still doing statical is scheduled superscalar, then the problem is the number of instructions that will be issued may vary. When this varies and it is not always a compant I mean, I mean filling up this large issue it.

Then it doesn't make much sense to have a large issue with. So, typically this idea of statical is scheduled superscalar is used for narrow issue which may be too. But for large issue VLIW is the preferred way to design, but most popularly used for most of the microarchitectures the design that is followed his dynamically scheduled superscalar. So, in short summary, these are the different architectural styles.

That are practically employed for deciding how to exploit the parallelism resident inside instructions and execute more number of instructions per clock cycle. And over this lecture, we have tried to summarize the topics like what how a basic risk pipeline operates, how the memory is organized, along with this specific RISC pipeline, what are the limiting factors, with respect to, I mean, instruction instruction execution in the pipeline.

And to elevate those limiting factors, what are the optimizations that can be done. So that would be our overall review of existing computer architecture notions. That we will be using for building upon them for the GPU architectures, and programming part. So with this will conclude this topic, which is overall review of basic computer architecture, and we will continue with the part on more fundamentals of GPU architectures.

How GPU architectures and their execution differ from traditional computer architectures. Now, one thing that is to be very important in this case is this idea of instruction level parallelism. Because we will see that GPUs essentially are designed with the basic philosophy that large number of operations have to be carried on by the processor. Now, at this, even at this level, there are different philosophies of computing,

Like how really large number of operations can be carried out. Will it be carried out by executing different instructions in parallel, or will it be carried out by executing an instruction over multiple points data points in parallel. And among those philosophies, which is the one that is used for GPU is that the topics from which we will get on with the Next part of the lecture where we delve more deeper into GPU architectures. Thank you.