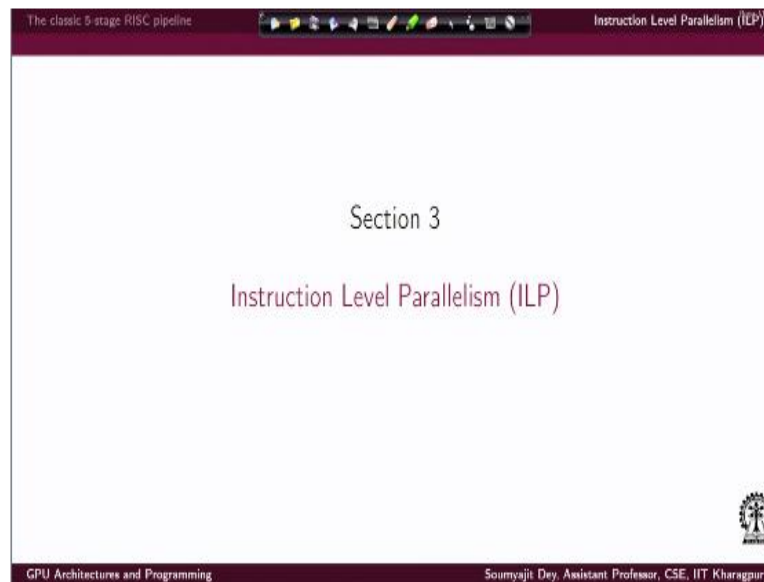
GPU Architectures and Programing
Prof R. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture No. 3
Review of basic COA w.r.t. Performance (contd.)

Hi, so let us get on with the section three of our.

(Refer Slide Time: 00:30)



First part of our course, where we are kind of reviewing the basic background on RISC file processing of basic computer architecture. Now, Just, I mean, if you recall, we have discussed about the five stage RISC pipeline. Next we have discussed about the memory hierarchy that is prevalently, popularly used. Now, I mean, we have just covered the basics here.

And after that, we are trying to figure out what are the different techniques that are popularly employed inside microarchitecture in order to exploit the parallelism that is present. While instruction processing, because, although we started with the basic background that I have a basic five stage RISC pipeline in modern microprocessors the pipelines are far deeper. Moreover, pipelines are hardly paralyzed.

That means there are multiple parallel functional units are in a mean, you can just simply view it that in a modern microprocessor there are a collection of pipelines which are processing

instructions in parallel, although that that element is not that simple. So, this whole idea of figuring out how microprocessor can functions in parallel, through pipelining and parallel processing kind of techniques.

This comes into his purview of the title, that we have here for section three which is instruction level parallelism.

(Refer Slide Time: 02:05)

The classic 5-stage RISC pipeline — Instruction Level Parallelism (ILP)

Actual Pipeline CPI

Pipeline Cycles per instruction (CPI) = Ideal pipeline CPI + Structural stalls + Data hazard stalls + Control stalls

- Handling hazards require both architectural and compiler techniques
- Data hazard types while executing instruction i followed by j in a pipeline
  - RAW — j tries to read a source before i writes it, so j incorrectly gets the old value
  - WAW — j tries to write an operand before it is written by i. Will not happen in simple RISC, but in pipelines that write in more than one basic stage or allow an instruction to proceed even when a previous instruction is stalled
  - WAR - j tries to write a destination before it is read by i, can happen in case instructions are reordered
  - RAR - not a hazard

instr i → instr j ⇒ IF ID ..

Now, to discuss about how I can make what are what are the architectural and compiler techniques using which I can make microprocessor compute on instructions very fast. Let us go back to the basic formula that we had. Which was that the basic index of how how fast the pipeline is executing is CPI that is cycles per instruction. So I have a pipeline, through which instructions are flowing starting from the fetch states, up to its completion in terms of register write-backs and memory updates.

So, we are thinking that let us say a pipe. I mean a single instruction take some n number of cycles on the average so I would say that the pipeline's performance is m. That is the there is a CPI there's it takes m cycles per instruction. Now, if I am trying to compute the CPI inside the pipeline execution. What is the actual CPI that depends on several factors, as we have discussed earlier, considering a multi stage pipeline

Where an instructions operation is broken down into several basic stages. And every basic stage is actually engaged in processing some instruction. Whenever it is executing without any kind of stole that gives me an ideal pipeline scenario. And inside an ideal pipeline. At the end of the pipeline I can see one instruction getting completed by every clock cycle. So that gives me an ideal pipeline CPI for a single pipeline as 1.

That means one instruction gets completed in every clock cycle. However, it doesn't really happen because as we have seen, there are several kinds of hazards structural, data as well as control hazards. Which introduced what we known as what we call us pipeline stalls. So due to a structural hazard that can there can be a stall in the pipeline, that means some instruction thing inside the stage is not progressing in the next stage in the constitutive crop cycle.

So that induces a structural stall and similarly there can be data hazard induced stalls as well as control hazard induced stalls Now the question is, how are such hazards detected and handled? There are various techniques for that. Now as we can understand that as a collection of these all different phenomenons, we have the actual pipelines CPI deviating from the ideal pipeline CPI. Now of course the calculation becomes more complex.

When I have more complex pipelines, which can execute multiple instant which can actually get multiple instructions issued in parallel, instead of just having one fetch unit issuing. I mean, executing and then starting to fetch one instruction in one clock cycle. So, starting from this perspective. Let us first discuss what are the what are the classifications of hazards and what kind of techniques are practically used.

For getting around them by the architecture of course for handling hazards. There are well known techniques based on compilers, which can also help us to alleviate certain types of hazards now this is also something we will look into. So, I mean the overall point here is that these hazards are kind of impeding the overall performance. Hence, there has to be certain techniques. Some of the things which can actually help you to approach the ideal CPI.

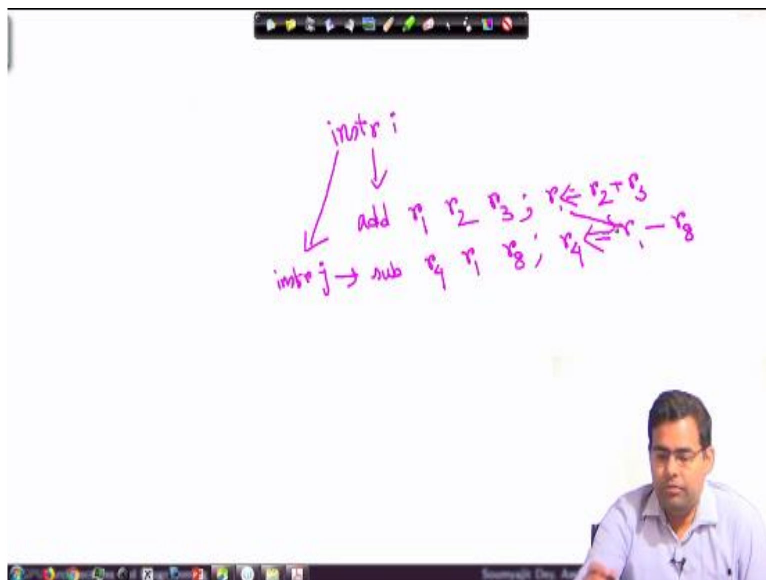
Some of these techniques can be implemented in hardware, so they are architectural techniques. While some of the techniques can be implemented by the compiler so that it can generate the assembly.

And corresponding machine instructions smartly enough so that the architecture can process them with in a more parallels i mean of providing us with higher throughput and get a automatically able to exploit higher level of parallelism. So, starting with a classification of different kinds of hazards. So consider the different scenarios for which we have some data hazards. Now for that.

Let us start with a scenario that I have an instruction i executing and is followed by the execution of an instruction sheet. So, let us consider instruction i. This is followed by an instruction j Now, what are the different problems that can happen when this sequence is entering the instruction fetch unit, followed by instruction decode unit, and so on so forth. So, the first. kind of hazard that may happen is, what we know as read after write dependencies.

Or read after write class of hazards. So suppose instruction j is trying to read a source. Before instruction i writes it So, let us let us try and understand how that can happen. So let me pick up a simple example here.

(Refer Slide Time: 07:35)



Suppose I have this instruction i. It is going to perform some operation. So let this instruction be an addition. And it is going to add values from registers r2 and r3, and write it to r1, right. So, overall, I have r1 is getting updated with the content of r2 and r3. Now what if I have an instruction j which is going to use the content of r1 for some purpose so let it be substract instruction.

It is going to substract the content of r1 it is going to subtract from the content of r1, some content of some register let say r8 and put it to r4. So, in terms of pseudo codify right. I am trying to update r4 with r1-r8. Now the thing is, I want these instructions to be scheduled. Like, i is to be followed by j. Which means, when I am executing instruction j r4 deserves an updated value of r1, right?

That means r1 must have been updated by the previous i-th instructions so here. I have this dependency. Now, coming back to our description of the hazard. So j is trying to read from some source. And that is some place where i is supposed to right now as we know in the pipeline is going to right, execute the right part in the write-back stage. That means, this destination register for i will be updated in the write-back stage.

Whereas the read is going to read has to wait until this update happens. Otherwise, what can happen is, j will incorrectly read the old value, this is something we have discussed earlier also. So, this is kind of a hazard, known as, write, read after write hazard. So unless I put in the stall. In the execution of instruction j, it is going to read the older value. Of source, instead of reading the updated. value, of the source, register.

So this is a read after write kind of dependency. Now, there are several other kinds of dependencies that may happen. So read after write is the most simple one, and also the most frequent one of them to understand that. Unless I am going to delay the execution of j. It is bound to read from the registers, an old value. And it is not going to consider the updated value, that is supposed to be written by the preceding instruction that is instruction i.

So this is your example of read after write and to preserve the semantics of the instructions. You know to force that the read happens only after the update that is done by the write we have to insert the stall. So this is a data hazard of type read after write. Similarly, there can be write after write now consider this. So I am always considering that i execute followed by j execution. Earlier instance I had that j was going to read from a source. Before i write the source.

Now, this consider this is execution, that is going to write something. And j also went to write something. Now, j tries to write an operant and before it is written by, i. Now why it can happen in a general simple pipeline of where register updates will only happen in the writeback stage. This problem cannot happen right because instruction i is being followed by instruction j in the pipeline.

So only after i will update it. write. Only after that, j will be updating its write, it cannot happen in such a simpler pipeline but it may happen in a, in a more complex pipeline. Where you can have the facility to do write in multiple stages, rather than in one stage. So, in summary, this will not happen in simple RISC. But in pipelines that write in more than one basic stage or allow an instruction to proceed.

Even when a previous instruction is stalled. These are the situations where the situation may arise. Now, the other type of dependencies, write after read. Now what is really that? j is the principle of succeeding instruction. It tries to write a destination, before it has been read by i. Now as we have discussed already that i is being followed by j. So how can realy j write before I am reading the instruction.

So, before proceeding with this again, let me go back to WAW type and repeat because it is related toWAR in WAW the problem can only happen when in the pipeline there are multiple possible states where write can actually be executed. And if the previous instruction is doing some write and the later stage of the pipeline, whereas the following instruction is trying to do the right at some earlier stage of the pipeline.

Then this problem can come. But what about WAR? Here we are talking about read after write where the write after read, where the write is an event of the succeeding instruction and read is the event of the preceding instruction. So read is by i. And j is the succeeding instruction trying to do write. Now. In an inorder pipeline this cannot be a problem. So we have a new word here what is an inorder pipeline.

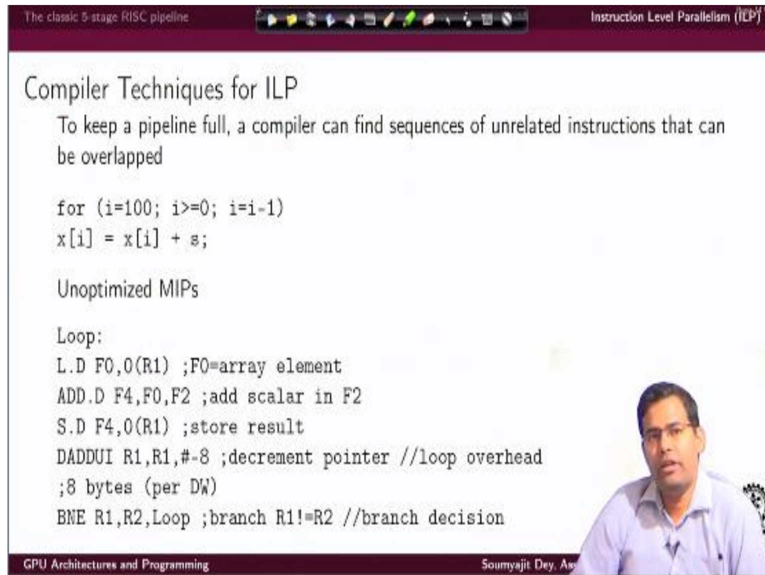
Whatever we have discussed till now is extremely in order that means I am given an instruction sequence that is exactly the sequence in which I am feeding the instructions into the pipeline, what the point is, it is not always necessary. There are situations we exclude them soon, where we will see that instructions may get reordered or instructions execute out of order. When that happens, So suppose there is a smart mechanism inside your micro architecture.

Which is deciding that okay I will do a reordering of instructions. It has to do the reordering in such a way that this issue doesn't arise that if it is the case that i is going to read some location and j is going to write some location. And the original ordering is i followed by j. If i switch the order. Then, j, will be writing a destination. And after that, I will be reading it, which is not the actual order as it is indeed there is kind of indicated by the original program.

So that is again a violation so WAR is a violation. So, we see that write after write and write after read are violations that may happen in certain scenarios for write after read instructions.Have to execute out of order for write after write, the pipeline has to have the fecilitated from multiple stages write are possible. So, apart from this, the read after write is a scenario, which can occur even in a basic pipeline.

But that can be elevated with suitable stores. Now what about the other combination that can happen, combining reads and writes, so we have read after read that is never a hazard, because the reads, do not update state any register.

(Refer Slide Time: 16:12)



Now, as we have discussed earlier,that there are various possible techniques in which the parallelism of instructions can be exploited, some of the techniques are architectural and some of the techniques are also compiler based. So, let us kind of review what are the different optimizations that a compiler can do to extract instruction level parallelism. I mean, of course there are many possible optimizations.

We will just talk about one or two of them for our basic motivation. So consider. At this point, the simple C code segment that we have for a loop. So we have a loop with the loop index down counting from hundred to zero. And all we are trying to do is we are doing a scalar addition we are adding the scalar value s to all elements of the vector x. Now, consider the corresponding MIPS assembly, which is unoptimized.

So, what do we really have in the MIPS assembly, you have this loop. So, we are assuming that the address of this array, x, which is storing in the vector x, is located in the base addresses and this is r1 is containing the end address is essentially x hundred at the start. That address with of set zero gets loaded to the register is F0. F2 is a register containing the scalar value, that is, s right. So, you execute ADD.D basically ADD for double type values.

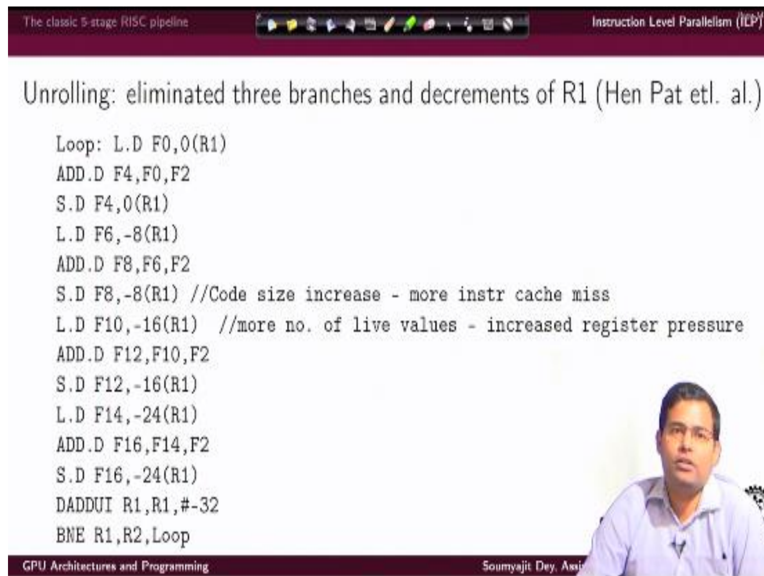
And you store the content in f4 register F4. So, F4 essentially has the content of x 100 +s. Right. And then you store the result. Where do you store it, you are supposed to store it exactly in the

same place from which you read it right? So, you store it by this instruction, exactly at that location. What do you do after that you are supposed to browse back in the array. How do you do that. So, you decrement the pointer by 8 bytes, there is a width of the double type words.

So you decrement the pointer. That is the content at this content of R1/8. And then you branch again into the loop. In case you still have data to be processed, that means the base address is inside R2, as long as you are not done. Essentially, this is going to contain that and as long as you are not done, you will be continuously going inside the loop and doing the addition. and in that way you to the scalar addition here.

Now, what is the issue here. Why do I say that it is unoptimized. It can be considered unoptimized, because the reason we can have an alternate version of the loop where we may have, we may end up with the less number of instructions to execute.

(Refer Slide Time: 19:32)



So let us see what such a solution. So, suppose the compiler is provided with this kind of an unoptimized code in the initial part of the code generation process. And then the compiler does some architecture ever optimization, which is this kind of loop unrolling. So what it does is the compiler will instead of executing coming back here instead of executing these many hundred iterations of the loop.

It thinks that Okay, I will execute these many divided by four iterations of the loop. And inside each of the iterations, I will do for others additions. Let us see how the code looks like in that case. So now I have the loop, containing a sequence of 4 scalar additions of constitutive locations in the array. So these are load, followed by add, and then store. Again load of the previous byte, followed by add, and then stored.

Again the load of the previous to previous byte. that is why of set is -16, followed by add followed by stored. So at this point I am expecting that from your basic UG background on computer architecture. You should familiarize yourself with MIPS assembly, so that this kind of a program is readable for you. Just a reminder here. So we do a store here again. And then again, again we do a load at the offset of the fourth position.

And then again we do the add and store it back right?. So I am inside one iteration of the loop instead of doing one scalar addition, I am doing 4 scalar addition. And then, I am shifting path. The pointer by eight times for 32 bytes, and again executing the loop with the branches branch, if not equal instruction right? So what does this really mean I am doing the same functionality. Not a problem.

All I am doing is inside one iteration of the loop i am doing more number of additions, how does it tell. It tells because I am executing the branch if not equal instruction, less number of times, the previous number of times /4 that many number of times. So, in every iteration, I am eliminating three branches, and I am executing the 4th branch right? So, and also, I am saving on another instruction which is the pointership thing.

That is a decrementing of R1, right. So, I have eliminated so many decrements operations and so many branch operations. Why is that a good thing, because the pipeline that takes the equal amount of time for decrementing by eight or 32, but it just has to execute that many instructions list. It also has to execute the branch if not equal instruction that many times lists right? So the for every branch we know there can be associated stalls and all that.

So all those problems get elevated. In that way, I have lesser number of instructions executing. And also, if we look you look at the code here.I mean, with lesser number of instructions executing, I have an overall speed up for this scalar Addition that is going on. And we have more overlapped execution in the pipeline, because the instructions are unrelated. I do not have branches.

So, I do not have sequence of branches for each ADD. So that makes this execute faster. However, what is the problem with this. In general, there is no problem but of course, you can always argue that if we follow this process that again, I can unroll any loop up to any bound differently that is not a good solution. Because programs are considered good for the loopy behavior right? That means there has to be a sweet spot.

That means what is a good unrolling factor. Now let us consider the side effects, or bad things due to unrolling. Your code size is increasing right? Earlier your assembly was this. now you have a bigger assembly to be executed right you have loopy behavior But inside each iteration of the loop you have more work to do. So your code size increases that will create more number of instruction cache miss.

Because you are now fetching more instructions from the cache. I mean since your code size has increased. So there will be more number of misses. And you are, you are fetching more instructions of basically which are the same instructions, but they are duplicated and stored in separate memory addresses right? So there's definitely a problem. And the other problem is when this code is executing, you have more number of live values.

So more registers may get engaged due to renaming and other things which will cover letter, and that creates a problem called register pressure. So these issues are there, which will be clear, of course, as we progress through a bit more of content, but just to make a point that one should not think that okay unrolling is a smart thing to do but that means I should completely unroll any program and forget loops.

There's also not a good thing, because of this possible. These, these problems. Now, coming to some other techniques that also help with respect to exploiting the instruction level parallelism. Now, in unrolling the help came because we have less number of branch computations we also elevated some arithmetic instructions in terms of pointers increment and decrement. So with all this.

That helped in executing the pipeline faster with more overlapping structures and eliminating certain specific assembly instructions. This is more of a compiler optimization. Not something done by the architecture is not a hardware optimization. There are, as we have discussed earlier, that optimizations can be done both by the hardware, as well as the software. So what can we have possible software optimization.

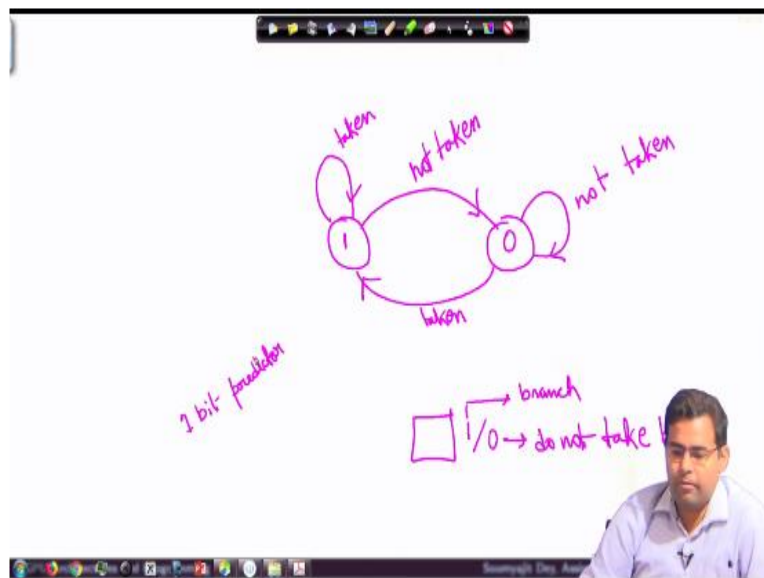
Let us speak of that. Sorry, what can be a possible good hardware optimization, let us look into that.

(Refer Slide Time: 25:58)

So, in modern microprocessors there is a specific hardware unit called a branch predictor. What it tries to do is to decide whether a branch should be taken or not taken, and we have discussed a bit about this earlier also, that I can, I can say that a branch will be always taken, or I can say that the branch will be never taken that can be a static kind of prediction. So, because in general for a uniform from a uniform statistics, there is a 50-50 chance about that.

However, things can be done with a bit more statistical sense. Like, we can look at the previous history of branches and decide. Okay, whether to take the branch or not. Now, by looking at the previous history, I mean if I think that I will, I will implement predictor which is much more sophisticated. Let us think of the behavior of such a predictor in terms of a finite automata. So what can be a very simple automata. So, consider.

(Refer Slide Time: 27:08)



A simple situation that I am going to decide whether to take a branch or not based on a one bit content is a flip flop containing either a 1 or a 0. If it is 1 that wouldn't mean are you want to take the branch. And if it is 0. You do not take the branch.Now, I want to implement finest admission logic to decide on this, and I have the previous history of branches, based on only 1 bit. So, suppose the value is 1, which is indicating that the previous branch was taken.

And right now, if I get a branch instruction, I will take the branch. And if this is a correct decision. Then again, the state gets upgraded with one only because the branch was really taken. Now let those value being 0 denote that. Okay, although I thought that the branch was taken. But in reality, it was not taken the nice trip to another state. That means I update this one with a 0, meaning that this last branch I did not take right?

So, next comes another branch instruction, I will also decide to not take it. And that will be kind of a self loop for the state, denoting that I remain in this not taking state that means I do not update the content zero in this flip flop. Now, following the similar logic and completing this automata. Why is my decision is that if any instruction comes I do not take the branch. And now, a branch instruction,

I mean if any branch instruction comes I will not take the branch. Now, if. Finally, it is found that will the branch has to be taken. Then I switch back to 1. So, this completes the design of a very simple one bit. I mean, a branch greater with 1 bit. History is just putting the information about the last branch now. So to summarize, this is 1 bit predictor. Things can be much more complex.

As we can see, so this is nothing but just a extension of that same concept. And we are now thinking that okay I have predicted with 2bit and is a formal term that you use you call it a 2bit saturating counter saturating counter for obvious reason that okay if I take a decision and I that decision is reinforced twice. I keep on taking the decision. So that is what I can. That can be nice somebody, I took the decision that the branch is taken.

And then I find that Okay, again it is taken. So now I switch to a state that I call as strongly taken. So, if I find that in a next sequence. There is a branch instruction and is not taken. I come to a state that is called a weakly taken that means Still, if an instruction comes, I will consider the branch of instruction, there is a branch instruction comes. I will consider to take the branch, but then again I see that this also wrong.

So then, again, the branch is not taken. Then I come to a state that is weakly not taken all that is happening is, if I go, if I think of the original construction that I just did. I was simply remembering the history of the previous branch and trying to do the same. Now, I am trying to do something based on the history of last two branches. So, if last two branches were taken I will definitely take the branch.

If the last word one branch was taken, but the last branch was not taken. Still, I will take the branch provided, I have switched to the weakly taken straight from the stronger taken state, and will continue like that. So, you can just include this behavior, as we can see in the form of a two bit saturating counter. And with this kind of a switching logic, which will actually tell you when or not to take a branch.

All again I will just repeat that the weird things have started different now is, in my simpler design, it was a one bit predictor in these design, you said 2 bit predictor. I am trying to decide on branches, based on a history of size 2. Earlier, I was trying to take them decide on branches, based on a history of size 1. So maybe we will take this up again. Yeah. Thank you.