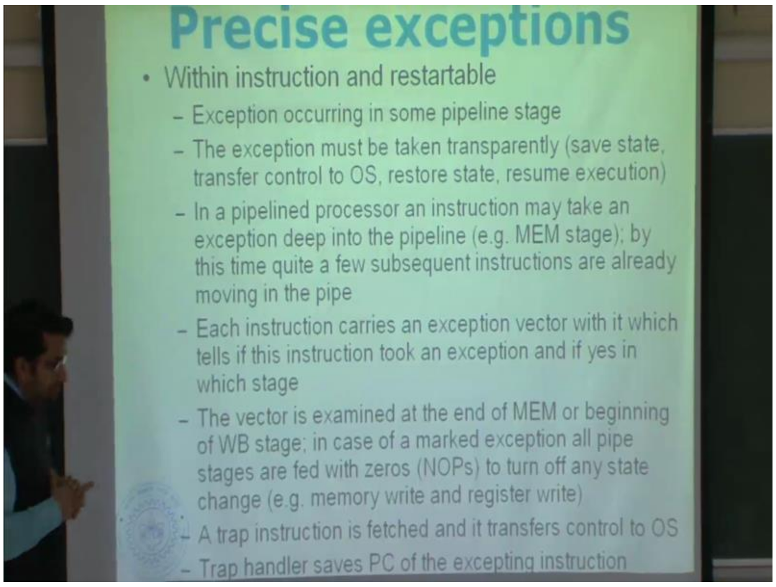
Computer Architecture

Prof. Mainak Chaudhuri
Department of Computer Science and Engineering
Indian Institute of Technology, Kanpur

Lecture - 19
Basic Pipelining Branch Prediction

(Refer Slide Time: 00:19)



Precise exceptions

- Within instruction and restartable
  - Exception occurring in some pipeline stage
  - The exception must be taken transparently (save state, transfer control to OS, restore state, resume execution)
  - In a pipelined processor an instruction may take an exception deep into the pipeline (e.g. MEM stage); by this time quite a few subsequent instructions are already moving in the pipe
  - Each instruction carries an exception vector with it which tells if this instruction took an exception and if yes in which stage
  - The vector is examined at the end of MEM or beginning of WB stage; in case of a marked exception all pipe stages are fed with zeros (NOPs) to turn off any state change (e.g. memory write and register write)
  - A trap instruction is fetched and it transfers control to OS
  - Trap handler saves PC of the excepting instruction

So, we were discussing precise exceptions, so these are exceptions that happen within an instruction and restoring that is the definition of a precise exception. The main comes from the fact that when exception happens, the system state is such that everything before it has completed and nothing occurring.

So, the system state is precise in that sense when the when does the exception case handle and these are exceptions that happen within an instructions. So, an instruction executes one two from pipeline in some stage in cases as exception hollow, not all within instruction exceptions may be results, let me say, but precise exceptions are always within instruction. This means a precise exception is always tagged with this and that is why the restart able clause is important, because it says that after you handle the exception, you should be able to resume the instruction and execute it.

So, exception occurring in some pipeline stage and the exception must be taken transparently meaning that you save state transfer control to the operating system then restore state and resume execution. In a pipeline processor, an instruction may take an exception deep into the pipeline such for example; it may happen in the memory stage you could take a base form. For example, by this time quite a few in subsequent instructions are already moving in the pipe right next to instruction must have in fetched something on that in the decoder something must have in execute mode.

(Refer Slide Time: 02:18)

If you look at the pipeline timing when this instruction gets to MEM, the next one is with x next to next be decode and another is been fetched. So, that means by the time the exception happens there are three mode instructions inside the pipeline and you have to maintain the precise semantic, you have to do something to make sure that these instructions do not get to complete. They do not get to modify any state to the processor and of course, you have to make sure that everything above it that is completed by the time you take the exception because the previous instruction is currently write back.

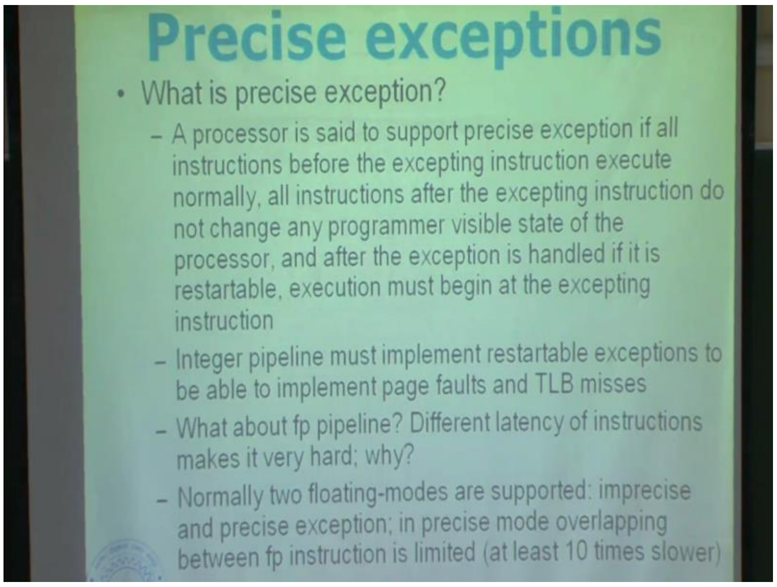
So, the way it is done is that each instruction carries an exception vector with it which tells if this instruction took an exception and if yes in which stage. So, it is called a vector because usually the length of the vector is equal to the number of five stage. So, you mark vectors on a width on if for example, if instruction takes an exception in x stage we marked how we drawn. So, on the vector is examined at the end of MEM or beginning of write back stage, if you tell notice that the write back stage cannot raise an exception.

So, when you are in the write back stage you know that you are done, you have no further competition left, and the only thing that you have to need to do is you store your result back to the corresponding register. So, write back stage is usually exception free and that is why this is the stage where you usually change this vector that tells me if this instruction should write the register or not. So, in case of a marked exception all pipe stages are fed with 0 s to turn off any stage change.

So, essentially what I am saying is that suppose this particular instruction takes a page fault in memory stage. So, you mark that vector and when this instruction reaches the write back stage you examine the vector then vector says that this instruction actually will get accepted. So, now you have to actually modify the instructions which already in a pipeline, so that is exactly what you say all pipe stages are going to be fed with 0 s. So, that essentially they are nops, so they move through the pipe they do nothing and in our five stage pipe, we know that by the time this instruction in the write back stage the previous one has completed.

So, this is that enough to maintain preciseness as long as you do this you know that nothing after this particular instruction has done anything in the in the processor stage and then what happens is the trap instruction is fetched. That controls that transfers controls to the operating system and the trap handler saves the program counter of the excepting instruction. So, that after the exception is handled you can restore the program counter and start the execution counter. So, program counter of these instruction would be saved is a general mechanism.

(Refer Slide Time: 05:56)



Precise exceptions

- What is precise exception?
  - A processor is said to support precise exception if all instructions before the excepting instruction execute normally, all instructions after the excepting instruction do not change any programmer visible state of the processor, and after the exception is handled if it is restartable, execution must begin at the excepting instruction
  - Integer pipeline must implement restartable exceptions to be able to implement page faults and TLB misses
  - What about fp pipeline? Different latency of instructions makes it very hard; why?
  - Normally two floating-modes are supported: imprecise and precise exception; in precise mode overlapping between fp instruction is limited (at least 10 times slower)

So, a processor is said to support precise exception if all instructions before the excepting instruction execute normally all instructions after the excepting instruction do not change any programmer visible state of the processor and after the exception is handled. If it is restartable, execution must begin at the excepting instruction integer pipeline must implement restartable exceptions to be able to implement page faults and TLB misses. These are the two highly required restartable exceptions that you have to do it what is a TLB does anyone know? Look aside, so why is it used is it something edible or you can what is it, sorry, what is it?

Student: It is pastable.

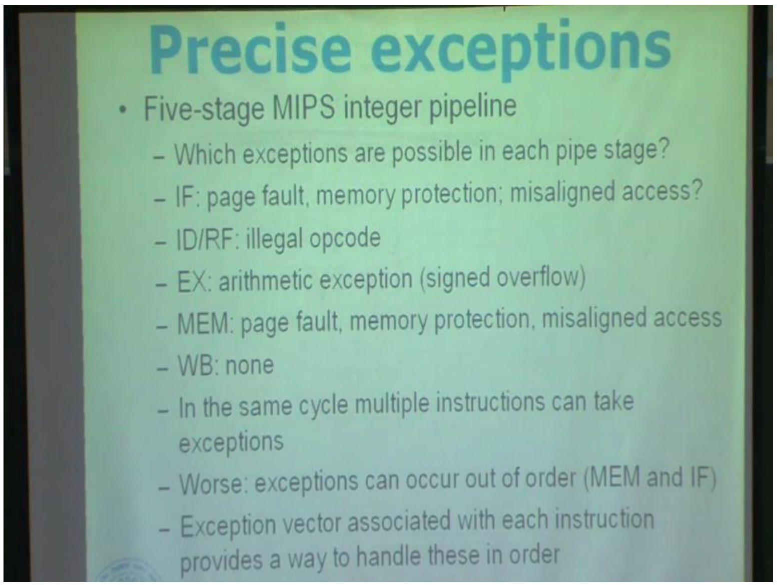
Yeah it is a pastable, exactly, so these are small structure that caches the recently used pastable, what about the floating point pipeline. So, these figures very tricky, so we will we will soon talk about these actually the fundamental problem here is that the pipelines do not really look like this.

So, by the time a particular instruction completes guarantee the previous instruction is completed because this instruction could be an addition instruction. You could remember a pipeline addition under that is your four sides this instruction might be a multiplication which takes longer. So, there is no such guarantee that by the time this instruction competes the previous one is completed. There is no other we know such guarantee that by the time this instruction raise an exception nothing after it has completed because it could be that this instruction is an is an instruction between less than that instruction.

So, by the time it raises an exception these are the only completed, so now how do we maintain precise. So, we will talk about this soon and normally the way this is handle this there are two floating point modes supported with processors what is called an imprecise mode, then the other one is a precise mode. In precise mode overlapping between fourteen point instructions is limited and usually at least an order of magnitude slower.

So, the imprecise mode has no problem, essentially you can ignore all exceptions. So, it is designed to be the performance, so we talk about how exactly we can implement the precise mode. So, that is what makes to be hot about how can I implement the precise mode in a correct performance.

(Refer Slide Time: 08:37)



Precise exceptions

- Five-stage MIPS integer pipeline
  - Which exceptions are possible in each pipe stage?
  - IF: page fault, memory protection; misaligned access?
  - ID/RF: illegal opcode
  - EX: arithmetic exception (signed overflow)
  - MEM: page fault, memory protection, misaligned access
  - WB: none
  - In the same cycle multiple instructions can take exceptions
  - Worse: exceptions can occur out of order (MEM and IF)
  - Exception vector associated with each instruction provides a way to handle these in order

So, we will talk about that, so let us first take a look at the integer pipeline, so we have this, so this is our five stage five fit, so first thing we need to understand is what kind of exceptions bit nice. So, in the first stage, the fetched stage what can happen you can have a page wise we have tried with page instructions and instruction page is. So, that is an instruction page fault you can run to memory protection exceptions, you may be trying to access some hours some of the right or you have to try in the access your own data.

So, that will make to protection valuation could there be a misaligned access in the page stage, so it needs all instructions are by the line. So, can I have a misaligned access, which essentially means can I have a misaligned PC here where my program counter be not above the load and not 0, these are possible. So, what are the sources of PC, we want a sources right every PC + 4 it can be predicted and if it is PC + 4 all the time it cannot be ever non zero module 4 to draw the zero module. What about the predicted precise and did I confess, so when is the jump instructions will assume last with you and strictly in the last two minutes.

Student: (Refer time 10:19)

Then, there is no right, but how can I have non legitimate.

Student: (Refer time 11:20)

There is only one instruction then they jump, but that shifts in the last two minutes and no other instruction. So, it has to do with indirect jumps the question is jump instructions take your target from register, how can you get a long register there is only way to get something that is misaligned hazard.

Student: (Refer time 12:10)

Well, assume that the pipeline is correct you hope or say, otherwise it will be it will be right to think talk about these. No, this instruction is not the target.

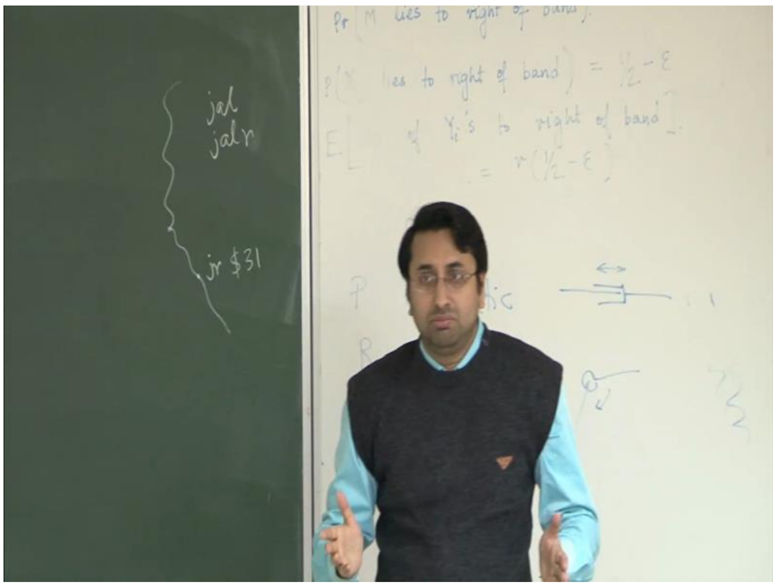
Student: (Refer time 12:30)

That would come from the register, register how can that happen? While using a compiler generating code happen in case of miss prediction. How does it happen, predictions are coming from the branch target it stores previously seen targets right it does not put up any. So, they cannot be wrong they cannot at least see they cannot be missed out, so what is it about?

So, if your target is some little legitimate instruction, so we are going along the predicted path and you can counter a retired instruction, but there also no matching call because we are just going along the wrong path. You should not be along this path here, in correct execution you should never be taking this path without taking some other path from which you actually call them multiple procedure.

So, now we have a retired instruction which is a literary jump and it use a register at the target and the register content was never set to something. So, that can misaligned access does anybody follow what I just said in the other branch somewhere in my program.

(Refer Slide Time: 14:17)



So, through some control path I will reach the point these all also may be predicted and here I take a prediction I going along this path and here I encounter a return statement. The point is that through whatever path I came, there was no procedure call statement simply because this path is wrong actually I should not be following this path at all. But, because of prediction I will follow this path and I encounter this j r dollar 31, dollar 31 was never saved because only the 100 corresponding jal or jal r instruction you would actually put to return as just dollar 31. You may able to come across such instruction because this whole thing is actually wrong, you were going along the prediction path.

Student: Why do not you take this path because of this been state or something, that is why we have to return it.

Provided that register has not been saved to tag that is what supposes your procedure wants to use this register for storing some computation branch right.

So, it saved on the memory, but has not been restored and then you start going along the wrong path and suddenly you ignore that this and you start using dollar 31 data complete a value. It does not, actually it is a 32 bit value, so it is just you should reach this as the program counter you could do that what was suggesting. So, when you take execute a jal instruction, it would actually store only that you know or something, but please do not do that.

Student: (Refer time 16:27)

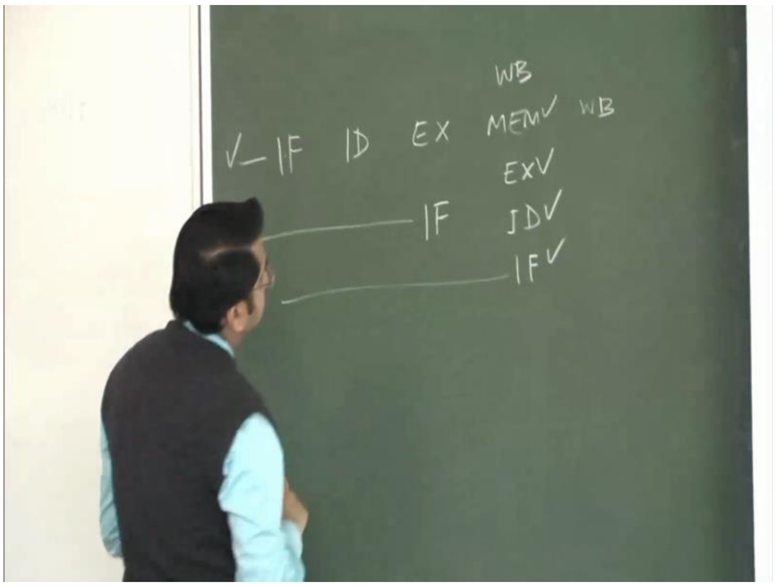
So, this will happen with very low likelihood in this pipeline this will happen if you have a deeper pipe and this will happen if instruction starts executing, how do the program, which we will discuss very soon. So, we will visit this example again, but keep in mind that it can happen misaligned access in the in the fetcher because of execution along misaligned path. This arises particularly from indirect jumps well indirect using a wrong register to take the second stage decoder illegal op code again aligned as because of same reason because of this particular path.

So, I would actually give the same example, it could happen that in such a case actually this turns out to be an aligned access. So, the fetcher does not know it innocently goes and fetches what and all is there in that particular address or anything that as an address goes and fetches whatever is there. It may be completely a garbage value, does not even make sense through the decoder. So, immediately there is an illegal op code exception in the sets. Third stage can have arithmetic exception, for example in integer pipeline the only exception that will happen is signed overflow.

So, you can go and look up your list of refractions, these are only bad thing that can happen in the third stage. Fourth stage can have page faults memory protection and misaligned accesses, again this mix complier are actually very careful about this. So, legitimate load store operations will never have misaligned addresses except for those l w l, l w r s w and s w r.

So, this can again happen because of execution along miss paths, we will start using wrong adjacent and we will start doing wrong things, write back stage does not have any exceptions any question. So, now the problem is in the same cycle multiple instructions can take exceptions and even worse exceptions can occur out of order.

(Refer Slide Time: 18:53)



So, consider this particular instruction and this instruction we are looking at these two instructions alright. So, this instruction is a MEM stage may not be this one let us take this one its better this one and this one these two instructions this instruction can take an exception in the MEM stage this instruction can take an exception in the fetched stage. So, a later instruction can actually take an exception earlier in time, this instruction will actually this exception will show up earlier than these exception in the pipeline because this is how my time knows.

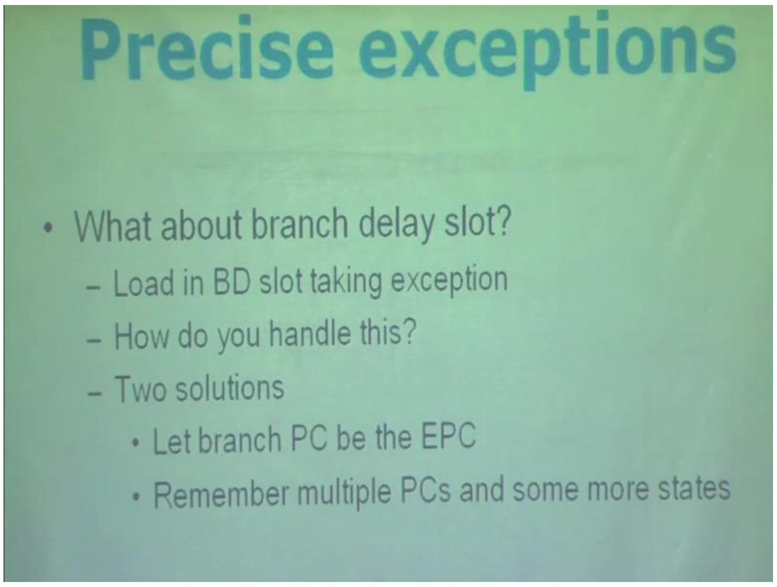
So, that is exactly all the same exception can offer it out of order and in the same cycle multiple instruction can make exception. So, you can look at this cycle right there could be exception here, here, here; here all the four instructions can raise exception in the same size. So, we have already devised a method to handle this, so exception vector associated with each instruction provides a way to handle these in order because we say that. Well, let this instruction take an exception I am not going to handle immediately I will only mark it in the exception vector.

Finally, this instruction which is right back stage I will actually handle the exception fortunately, by then these exception will already have been handled. So, this instruction will actually not be the pipeline any more when you say that would you handle the exception we will feed zeroes in the pipe. So, these three instructions will actually get nullified, so when these exception is handled the exception restarts these instructions re-executes.

Then, again this instruction will appear in the pipeline may take again an exception then we will handle it later. So, exceptions will be handled one after another exactly by total amount because say exception bit any question can actually handle the exception. So, suppose that the third instruction page fault, there would not be any instructions or it cannot form the instruction would actually stall.

It will not stall, so written in it whatever in the we will just carried for does not matter, what it is you will mark the exception written as a single bit in the fetch stage and that is it you'll really not fetch anything. Registers are not protected registers, they do not belong to you they belong to processor, so any question, is this clear now.

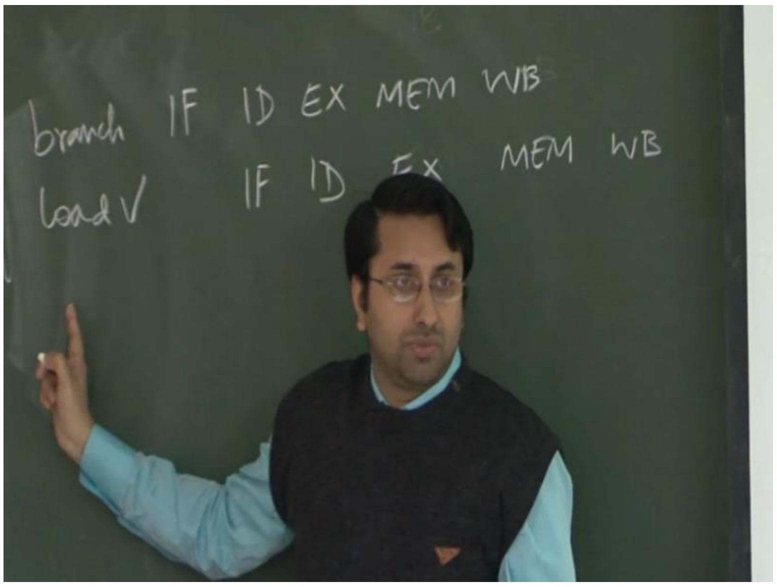
(Refer Slide Time: 22:02)


Precise exceptions

- What about branch delay slot?
  - Load in BD slot taking exception
  - How do you handle this?
  - Two solutions
    - Let branch PC be the EPC
    - Remember multiple PCs and some more states

I handle exceptions is the standard five stage five, so few small problems that you have to worry about. So, I mention these because you will have to handle system calls in your second assignment and these system calls are actually exceptions. So, the handling will be exactly same and face the exaction problems, so once more problem arises about the branch delay slot. So, let us try to understand why a point is very special what the difference from others, so let us suppose that you have a load instruction in a branch delay slot.

(Refer Slide Time: 22:37)



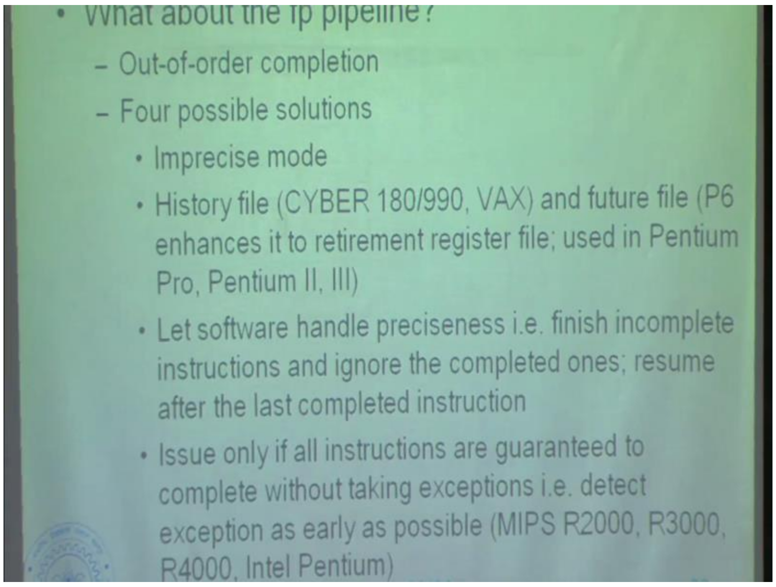
We are taking an exception, so essentially I have a branch instruction and in the slot I have a load instruction and then I have target of the fall through depending on which will a branch goes. So, let us assume that the load in the branch delay slot takes an exception, so the question is how you really handle this the problem is depending on where the exception is taken. So, first of all to maintain the preciseness of exception you cannot execute the next instruction that is what if you take the exception of the load come back re execute the load.

Then, only you can re-execute the next instructions, which essentially means if you take this exception of the load, you have to remember which was the branch is out because then only you will know which PC is used after the load instruction. So, the exception PC is this one this instruction, so in a normal case you will only remember this PC and that is it.

So, if you only do that the problem is you get the exception then you resume execution here what is the next PC this is PC may not be you have to remember which of the branch is out the previous instruction. It could be PC plus four or it could be some target instruction, so there are two solutions, let the branch PC be the exception PC. So, if you want to do is you take the exception, but start execution from here, you re execute the branch. Then, you re execute the load and then load and then you do not have to remember which of the branch is out.

The second option is you remember multiple PC s and some more space, essentially you remember the load PC and also remember the next PC to execute after the load. So, of course, the good news is that now are the codes that are handle to do for the assignment they are system calling branch. So, you will actually never face this particular boundary that in instruction a load branch is not getting an exception. Of course, we do not model page faults or anything without similar there is no question on having a load instruction taking an exception along this. So, the exception in your second assignment is six sigma’s that that is what you have to handle, so this case will not arise in your assignment.

(Refer Slide Time: 25:31)

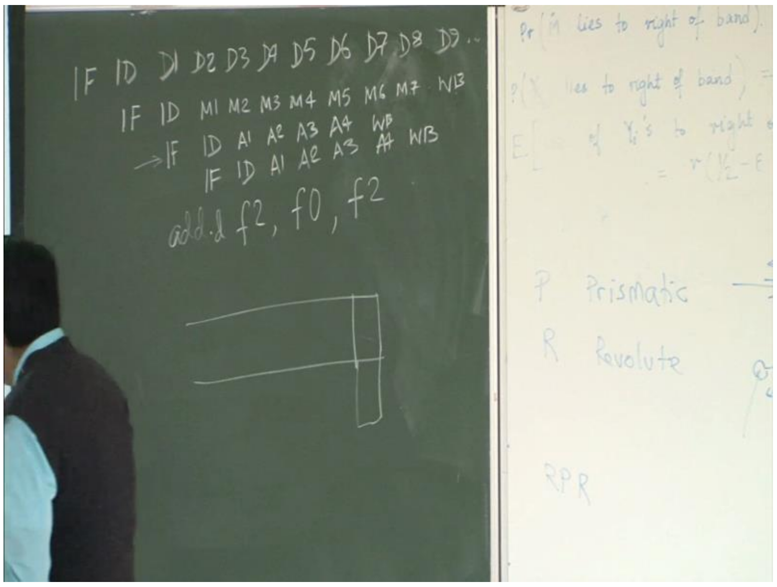


- What about the fp pipeline?
  - Out-of-order completion
  - Four possible solutions
    - Imprecise mode
    - History file (CYBER 180/990, VAX) and future file (P6 enhances it to retirement register file; used in Pentium Pro, Pentium II, III)
    - Let software handle preciseness i.e. finish incomplete instructions and ignore the completed ones; resume after the last completed instruction
    - Issue only if all instructions are guaranteed to complete without taking exceptions i.e. detect exception as early as possible (MIPS R2000, R3000, R4000, Intel Pentium)

So, now the hardware problem what about the floating point pipeline, suppose let us try to understand why this is harder. So, just to remind you we had this floating point handle which had a four cycle literacy the multiplier and seven cycle literacy and the divider had twenty five cycle literacy, so what will happen the pipeline is that.

So, let us suppose I have a multiplication operation, so your book actually puts a memory stage here which I will because the multiplier does not need a memory stage. Now, things really do not change, we will put a new stage here, so that is the multiplication instruction suppose the next instruction is an add instruction, so what we will put?

(Refer Slide Time: 26:56)



So, let us suppose that this multiplication instruction brings an exception to stage m 7, the last by now the add instruction is complete it has written back already, well how we maintain precise exception in this case. So, that is exactly what the first point says here instructions complete out of out is a problem, now somehow to maintain precise to recover the multiple value for by this instruction. For example, these instruction could be something like this f 2 coma f 0 comma f 2 a double position add let us say.

So, it has over f 2 already try to recover that value to maintain preciseness so that when this instruction goes into the operating systems exception handle the precise case of the processor. It goes that everything before it has completed and nothing after it has started executing, so now and also there could be other problems think about the previous instruction, this instruction could be a division here etcetera, it will take 25 size. So, if you try to replicate the solution that you are trying to do for integer pipeline in an exception vector that does not really work because by the branch.

This instruction reaches right branch when I am when I am examining this exception vector there is no guarantee that this instruction is completed and there is no guarantee that these instruction has not completed. So, this might have completed this may not have completed, so that does not really want any more we have to do something else to maintain preciseness. So, here are four solutions one is a imprecise mode that is a easy way out we say that I do not offer precise exception, but you get good performance. The second solution is to have a history file anybody guess what it might be from the name what could be a history file, sorry save.

So, before it overwrites a register we save the register to a separate register part that is called history file it tells you what the value was before this instruction. So, how does it help in maintaining precise exception, how will I use the history part? Save register for example, you know why is it you want them to write for same, what is the then there is a problem, then also problem.

So, I basically this is the point of precise exception is that I do not want anything to be modified after this if the multiplication is taking an exception. So, whatever register this add instruction is written to must be recovered now and that is what you use to recover the history file. Before you allow the add instruction to this stage before you allow this instruction to right to f 2, you actually save the previous f 2 in the history file.

So, why do you finally, copy the value to the main register file, sorry re starting the execution, the new value is already in the register file right to the main register file, it is already written back handling the exception. Now I am saying in the history file the new value is already written to a main register file the history files scores the own value. So, you do not have to do anything extra as such just get it for free right is that, but if the register is being written multiple time.

So, what if I have another add instruction here, so this one also completes before the multiplication gets to the write back stage when I examine the exception better right and this instruction may be writing to f 2 also. So, now, what we have to do now the received file will actually have the value of this instruction not the previous values.

So, I want the value before this instruction, but that is now when you go over written by this particular value because every time you update a register in the main file whatever the main file is having you copy that into this file is the problem clear to you. So, how do you resolve this, sorry you will not get back the second, so you will stall the instruction. So, that is one possible solution, so what you say is that you have a big vector length equal to number of registers. If one register is already to the history file you mark that bit, so this instruction comes to write back; that means, its already marked, so you say cannot write.

How long do you keep that bit in a bit why you deserve in that bit so some structure will have to maintain this order, so that you have to you have to essentially have a one to one correspondence between the instruction and a history file. So, you need a separate FIFO, which actually maintains this particular order of fetch whenever you fetch the instruction to make you that in the FIFO. So, instructions will actually bringing for the FIFO in that order that will make sure that whenever an instruction is removed from the FIFO you know that now that history file entry can be overwritten we feel that history file is in that sense.

So, it is not at as easy as said your history file and we are done no there are many small things you have to take care of. The other option is the future file, so probably you can guess now what that mean. So, essentially what I mean is that well in this case you do not actually change f 2 in the main file install its how else that tells you what the future values the main file does not change in this case. So, in this case now what will happen is this is actually when this you still have to maintain this before when your instruction goes out to the FIFO you copy the drives on the future file into the main file because.

Now, that value becomes visible to everybody, so do you have here also this multi version problem do not you have in a history file multiple versions of values I given the same instruction to the same register, so I will have the same problems here. So, now essentially what you can do is since we are maintaining this FIFO anyway why do not you have with each entry of the FIFO a value fit the value that this instruction is produced that will become your future. So, essentially we have a FIFO queue wherever these instruction is fetched you will put this instruction line here and also it will have a value filled we should be populated by the value in this instruction produces.

Whenever this is removed out of the queue you move this drawing to the main one that automatically takes care of this multiple version, you have to the next instruction we have to bypass exactly your future file. Now, it becomes part of your bypass network because what if the next instruction is f 2 what we will get it from the main file does not have the most accurate values here sitting edging in the queue. So, to bypass from the future file, so anyways none of these actually is an easy solution those have complications.

So, that the PC architecture enhances the future file to a retirement register file we will talk about p 6 we will also cover a few more concepts p 6 micro architecture is used in Pentium pro Pentium 2 and Pentium 3. So, these processors actually use this particular structure it is called retirement register file this particular whole queue, it has a certain size of course, which essentially means that if this queue is full, you cannot fetch any more you have to stop.

So, we will talk about this this skew in the more detail later it has various other names and all, but just keep in mind that it is associated in maintaining the order of instructions the order in which you are fetched, any question on history file or future file? The other solution is you can let the software handle preciseness that is forget about whatever is completed whatever is partial whatever is unfinished, what happens is that whenever this lets say this multiplication instructions take an exception by this one, we just right back. I get this exception better, immediately there is an exception without worrying about what happens to the instructions before it what happens to the instructions after it.

So, the software handler that is when we handle the exception we will have the responsibility to finish the incomplete instructions and ignore the complicated ones and resume after the last completed instructions. So, some of the instructions may already be completed here, so if you actually resume execution after the last completed instruction wherever that is the last solution is issue. If all instructions are guaranteed to complete without taking exceptions these are very hard think to guarantee.

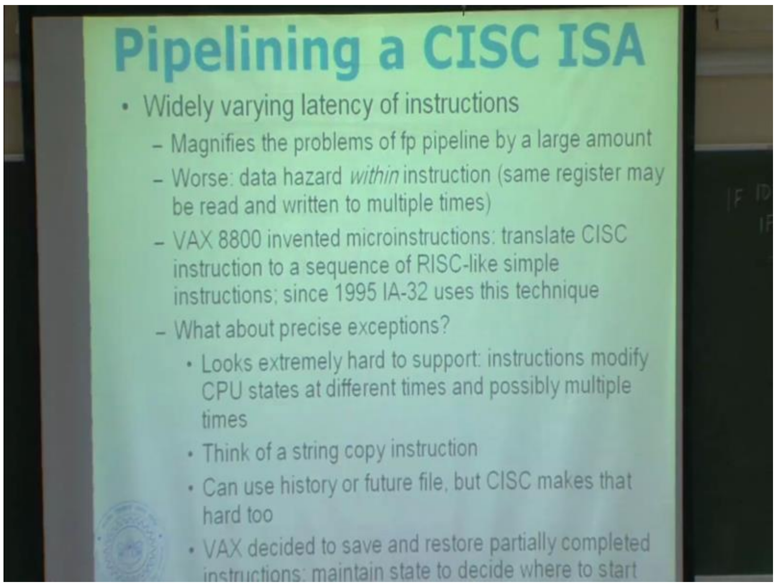
So, essentially what I am saying here is that you detect exceptions as early as possible. So, if possible whenever you are issuing an instruction you ask the following question tell me that the instructions before me with their exception a lot asking the decoder this question.

So, whenever you issue this load instruction here you ask all the instruction that kind be pipeline before me will they take an exception or not if the answer is yes, then you stall this instruction. So, solve this problem that you know there will be an instruction after this excepting instruction to the pipeline, but how we really answer this question. So, what at all you see how you knew that this multiplication is going to take an exception in the m seven stage how does this particular decoder siting here can say that.

So, it is possible you have to design how every functional unit takes for exceptions before starting the computation in the very first stage the multiplier could actually check if this instruction is going to get an exception or not that is possible. Actually, it can examine the operands and then figure out if there is an there is going to be an exception or not.

So, this exactly what is done in bits r 2000, r 3000, r 4000, so they actually solve instruction issue if there is a chance of an exception happening in any of the instructions band width. So, we talk about r 4000 very soon see how it actually does any questions?

(Refer Slide Time: 40:13)



Pipelining a CISC ISA

- Widely varying latency of instructions
  - Magnifies the problems of fp pipeline by a large amount
  - Worse: data hazard *within* instruction (same register may be read and written to multiple times)
  - VAX 8800 invented microinstructions: translate CISC instruction to a sequence of RISC-like simple instructions; since 1995 IA-32 uses this technique
  - What about precise exceptions?
    - Looks extremely hard to support: instructions modify CPU states at different times and possibly multiple times
    - Think of a string copy instruction
    - Can use history or future file, but CISC makes that hard too
    - VAX decided to save and restore partially completed instructions; maintain state to decide where to start

So, little bit about pipelining CISC is till now we have to integrate a RISC is where we heard this soon simple instructions where you know where the instructions requires more futures like in which stage will access memory and so on and so forth. In SISC, you have widely varying agency of instructions that magnifies a problems of the fourteen point pipeline by large amount which we have seen here and worse there could be data hazard within instruction. So, we have label in counter index that usually have errors from one instruction to another here you can have data hazard within instruction because the same register may be read and written to multiple times in one instruction.

So, VAX 8800 invented something called micro instructions what is that in places we translate CICS instruction to a sequence of RISC like simple instructions. Since 1995, your intel architecture uses this technique essentially internal you know CISC instructions get transfer into RISC like micro instructions, so that takes care of many bad things about the CISC.

What about precise exceptions in a CISC, it looks extremely hard to support because instructions modifies few states at different times and possibly multiple times it is not as well defined in the re startable. For example, you can think about string copy instruction which could actually take multiple page faults because for example you can try to copy particular strings. Then, planning multiple pages to solve other page faults planning multiple pages source pages can have page faults the discussion pages can have page faults.

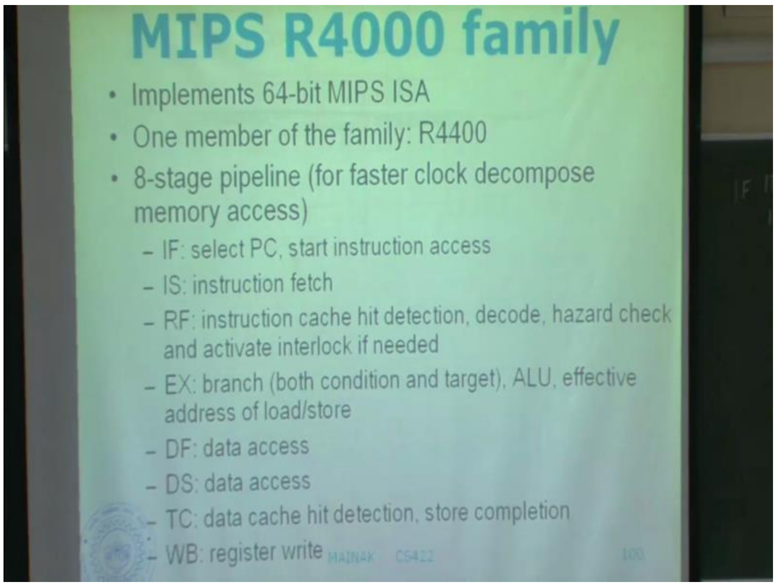
So, maybe which can happen that you have copied some part of the string and then we encounter a page fault because when you move to second page, by the way page is not there actually. So, there could be many problems in handling precise exception, so what is the solution here could be. Before you start doing anything you make sure that all the pages that you need on a memory, which you cut on the pages bring all the pages and then start string copy.

You can use history on future pipe as we discussed, but CISC makes that hard to do why is that actually all we discussed why that is. So, in CISC you just you know magnify the multiple times, so because of this problem same register may be little bit multiple times in one instruction. So, that problem of versions gets ninety five several times in this particular, so the same instruction may require multiple versions of the same register because you can have an exception anywhere in an instruction. Suppose you have return to some register to three times you need to go back to it may be several more times mount in exception, so you will have to maintain all these versions.

So, VAX decided to save and restore partially completed instructions essentially what we are saying now is that you maintain state to decide where to start. So, it may be a copied part of the string and then you take an exception you just you know remember that I have done this much of this instruction I will resume the execution middle of that instruction.

So, now essentially you are changing that the precise exception definition which we were saying that we do not really resume at the beginning of an instruction. You can actually resume in the middle of the instructions, because exceptions now will happen in buildup instructions.

(Refer Slide Time: 44:09)



MIPS R4000 family

- Implements 64-bit MIPS ISA
- One member of the family: R4400
- 8-stage pipeline (for faster clock decompose memory access)
  - IF: select PC, start instruction access
  - IS: instruction fetch
  - RF: instruction cache hit detection, decode, hazard check and activate interlock if needed
  - EX: branch (both condition and target), ALU, effective address of load/store
  - DF: data access
  - DS: data access
  - TC: data cache hit detection, store completion
  - WB: register write

MAINAK CS422 100

So, I will quickly go over this particular processor it just shows ah that the pipeline bypass and all these things just from a slightly different prospective, compare to a five stage five. So, this one this particular processor would give raise to 64 bit MIPS ISA and that you know bits R 3000 which is the 32 bit processor and R 40 is a family and R 4400 is a member of that family you can discuss which has an eight stage pipeline. Essentially, what they have done is they are taken this five stage five and as decomposed further to get 3000 frequency.

So, what are these five stages we have a fetched stage which selects PC you have that multiplexer which selects your PC and it starts instruction access and instruction fetch. So, starts instruction access and that is second cycle which actually completes instruction fetch. So, now, we have two cycle fetch although pipe let hard stage is register file access where you actually get to know if the instruction cache have to hit or not it is very interesting.

So, we start the second string of the pipe without actually knowing whether you hit the instruction cache or not you might know it only in this particular cycle and if you hit the cache then of course, everything is fine. Otherwise, whatever you have decoded will be discarded because you also start decoding in parallel, you decode you also do the hazard check and activate interlock if needed. You save that the RISC philosophy was that you detect all hazards in a decoder and it would use interlock cycles at that point third stage is execution. Here, you execute branches both condition and target you will also do ALU operations and compute effective matters of load store.

Then, there are three stages of memory access there is a data access DF DS, there are two stages that that do the data access in TC, you get low if you hit the data cache we still do not know actually with the beginning of the this side. You will only get to know whether the data that you are dealing with is actually correct or wrong and also in this stage you complete the store. So, essentially how do you do a store in first look up, suppose you are you have a cache box right which is 64 bytes and a store operation modify, let us say 4 bytes somewhere. So, you first read out the cache work completely modify those and then write it back to the cache.

So, these data access is actually get you the 64 byte out of the cache and in this cycle the t c cycle you get to know if the hit on the cache hit actually and then you do the store. So, the question now is what am I really accessing in the cache without knowing if it is a hit or not anybody guess you know what is the meaning of this or not. Same thing here from saying that you start the instruction access into an instruction fetch, but you really do not know whether you hit the cache or not because usually the way you would understand is that you first look up the cache if you hit you access the drive.

So, who is going on here I am doing in the opposite way, how you figure out if the cache hit anybody or may be do you know caches let cover caches. Who does not what the cache is raise hands everybody knows. So, what how you detect the cache?

Student: Match the tag, it should be (Refer time 48:22).

Match the tag.

Student: See whether the connectivity is there or not.

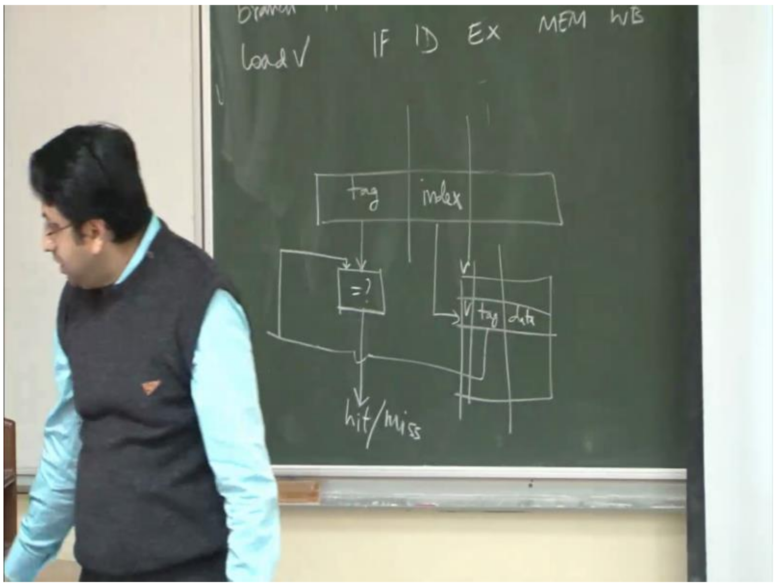
Management of what?

Student: Of the dashboard.

Of the dashboard.

Student: (Refer time 48:38)

(Refer Slide Time: 48:49)



So, from the from the given address, we take out some part of it called the tag and you match that against whatever is stored in the cache right in that block the tag of that block. If I match that the value bit is set which states the cache hit which means I can now access later I am doing it here in opposite way my first access the data and the check if the data was valid or not. Can I do that how do I access the tag how do I get this one actually cache is an array of blocks, now how do I locate this particular tag how will you locate the block I will tell you anyway.

So, there are some wigs which I will call index wigs in the address which we use to find out which block to get right. So, this will actually point to this is my cache, we will point to some block and this block will have data tag and a valid vector and essentially this is what I am leaving here. So, nothing really stops me, so I have the address from an instruction, let us suppose I have a load instruction or if I have the program counter I have this two laterals on me.

So, I can calculate the index and nothing stops me from accessing the data without accessing the tag right because the data will also be at the same index. So, that is exactly what I am doing here I access the data I access the tag in parallel and while I am accessing the data I make this comparison later and figure out if this data was actually correct or not. So, that is what I am doing here, why does it why does it help, because now these two operations are not realized tag check tag access tag check and data access I can actually you may be in parallel.

I can save time what will I be losing, I am losing a lot of power in my cache I invest I actually spend the lot of energy here now because some of the some of the data accesses are going to be useless, so I end up earning energy.

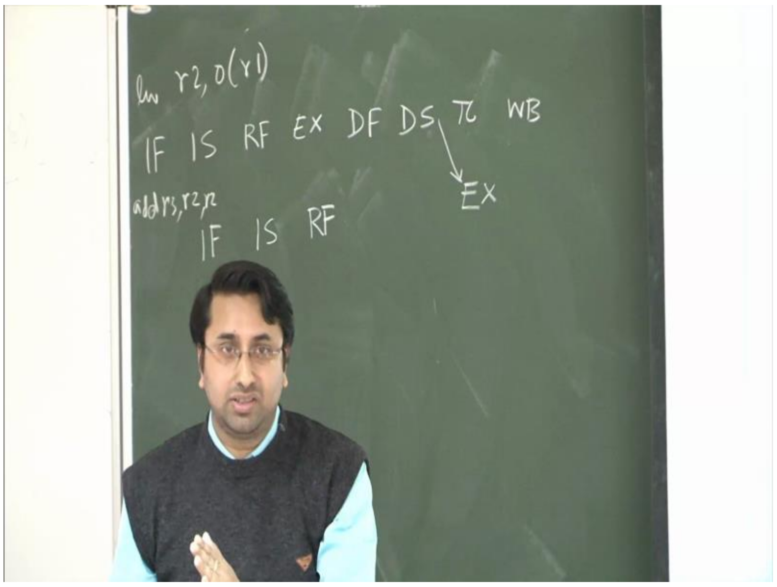
Student: All the tags in a set are actually all the RISC.

Quite for if cache, well yes when I say cache, this is visible in much more detail. So, you will have to you have to access all the data and pick one of them depending on that necessarily. So, there we will we will end up spending even more.

Student: (Refer time 52:09)

I do not remember exactly it will have at least two cache. So, does everybody follow what is mentioned, so if your cache will end up actually doing more uselesss, but this is these are done to save time. Otherwise, I would not be able to do this in three sizes I will leave you more time actually same here exactly same here and then finally, I do the register write back, so what are the implications?

(Refer Slide Time: 52:48)



I have a load delay of two cycles, let us see what that two is that is my pipeline alight. So, let us suppose that this is the load instruction this particular one, so values available here right, but I do not know the value is correct until I get to the stage right, is that clear to everybody. This is where I get to know if the cache actually that I am using is valid or not, so now think of a think of an instruction that uses the down rate value. For example, this instruction took the load mode R 20, R 1 and then after that we have an add instruction which gives us R 2 R 3 R 2 R 2 like.

So, it is the value, so let us try to see what happens to these instruction, so these instructions fetches here issues here register files access. So, it needs a value here, so there is no bypass on this getting that value it is impossible, the value is available here and I only get to know if it is correct here. So, if I introduce cycle stoles I will of course, again assuming that all stoles are introduced after this particular stage after the decoder internal stoles.

So, I stole it for two cycles, I move the execution here then I can at least get the value I still do not know if it is correct or not and these were things get very interesting. So, they say that the load delay stole is actually two cycles that these are the two cycles, but I will I will not make sure if I am doing the computation correct. So, what you do is you are actually making a speculation we are saying that well what a common case the cache hit usually well is done.

So, most of the time I am going to be correct if I lose the value there will be some cases where I am going to be wrong. So, essentially in those cases you will have to go back one cycle and re execute the x h. So, if it happens that at the end of this stage, you get to know oh this value actually was a cache miss. So, essentially what we do is now you introduce NOPS to the pipe and keep these instructions not here until the load miss resolves even though it is going to take some time actually for the data to come back.

So, this actually called a blind speculation where you are not using any history you are just regarding on the fact that cache is a good. So, there is going to give you majority of miss minority misses got it. So, this is called low hit speculation this is widely used today all microprocessors, they use a more sophisticated predictor these are prediction right that is what I am telling that that is what I am doing.

They will actually look at the history of this particular load instruction and decide if it is going to hit on this in the cache alright now worse case is three cycles as I just mentioned. Also, we need a hardware to back up by one cycle because miss may take longer the backup hardware turns the dependent issue in last cycle to a node and then stoles the pipe until the miss.

So, essentially we will cover this two to the load and keep this instructions, stole it until that the load data is available the pipeline interlock is implemented to stole dependent for two cycles and the interlock is actually decided it. So, whenever the situation arises this decoder can figure out and then it will not issue this instruction for two more cycles quiet clear and then send it here and again to do a tag check here and if needed introduce a load and stole instruction is this clear.

Branch delays three cycles, this is much easier to understand actually, so I will show you that these are branch instruction. So, I produce my branch target and condition here, so I cannot fetch here, I cannot fetch here, I cannot fetch here, I can fetch here and three cycle parts later and one of these is filled by the compiler which is your…