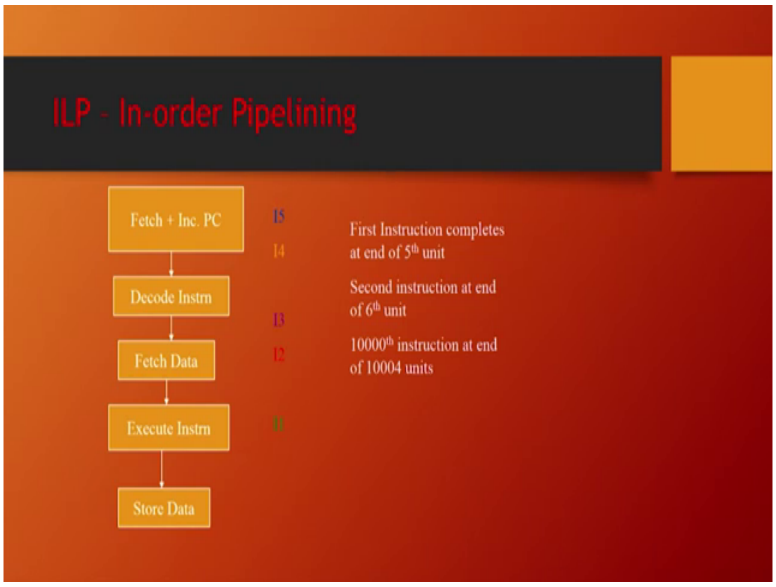
Information security-IV
Prof. Kamakoti V
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Lecture -59
Out–of-order Execution (Ref: IS-2)

Now, we will see out of order execution. So, this is also very important for understanding meltdown right. So, that is the second step, as I told you in the three steps. I have covered out of order in deep detail in my information security 2 course, but I want if you want, you can go and look it, look into it, but I will give you a very quick 10 minutes summary of what is out of order.

There are five stages in execution of instruction, the first one is the fetch the instruction and increment the program counter, the second is decode the instruction. The third is fetch the data, the fourth is execute the instruction, fifth is store data.

(Refer Slide Time: 00:39)



ILP - In-order Pipelining

First Instruction completes at end of 5th unit

Second instruction at end of 6th unit

10000th instruction at end of 10004 units

In an ideal scenario, let us assume that each takes 1 unit of time, whatever that unit be nano seconds, pico second; one unit of time, each instruction will take 5 units of time. So, if I have 10000s instructions then it will take 50000 units of. Now, the interesting thing as I told in the earlier course, when an instruction is fetched, all the remaining 4 units, namely decode fetch, fetched instruction, fetched data, execute instruction store data. They are not doing anything, when the instruction is decoded all the remaining 4 stages like fetched instruction or fetched data or execute instruction store data, they are not doing anything.

So, that five stages of the instruction is executed by five different hardware pieces, the hardware in charge of fetching the instruction will do nothing, after that the hardware in structure, in charge of decoding, the instruction will only decode the instruction. In the remaining part it will not do anything. So, every unit ideally is executing for 1 unit of time and then instruction takes 5 units in which each unit will take 20 percent or 1 unit of time idea, ideal scenario, just to understand the remaining 80 percent of the time, that each unit is free, ideally it is free.

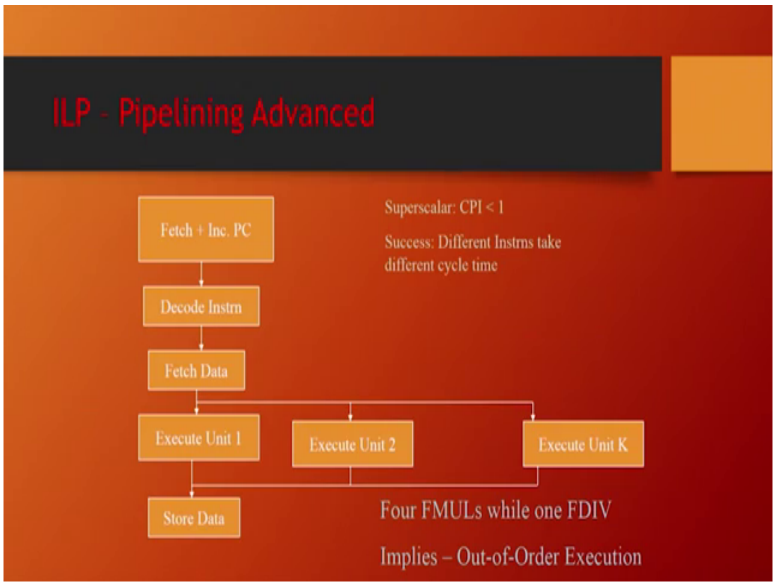
So, can they reuse it? Can I use it? This is the basic observation that motivated the notion of pipeline. In pipeline what happens? First, I 1 comes, I am sorry. It should have a very close look at your screen. I do not know why I made that color.

So, I 1 comes in, then, when I 1 is decoded I 2 is fetched, when I 1s data is fetched I 2 is decoded and I 3 is fetched, when I 1 is executed I 2s data is fetched I 3 is decoded and I 4 is fetched, when I 1s data is getting stored, I 2 is executed, I 3 is fetched, I 4 is decoded and I 5 is fetched. So, I 3 data is fetched I 4 is decoded and I 5, as an instruction is fetched. So, the first instruction finished at 5th unit, second instruction at the end of 6th unit, just 1 unit delayed can; so, your 10000s instruction at the end of 10004 units, it will finish so.

So, now, your entire thing becomes, 5 times faster at this point of time. Now, the ideal scenario is I always assumed that executing an instruction, which is happening at this fourth stage, as you see on the slide will take constant time right. So, we will take the same unit of time whether I am doing addition or multiplication or division or floating point addition or floating point multiplication or floating point division or whatever bit twice and everything I assume, it will take unit amount of time, which is not the case.

So, this is ideal as the ideal, proof our ideal scenarios for me to explain pipelining in say 2 minutes, which hopefully, I have done and hopefully, I have understood, but this is not a reality. So, the execute instruction there could be different instructions taking different amount of time and that leads to the concept of what we call as, the out of order execution processes. So, what has happened here is that I, what we have done is that there is one fetch that instruction get decoded, one data gets fetched, then there are multiple execution units.

(Refer Slide Time: 04:14)



ILP - Pipelining Advanced

Superscalar: CPI < 1

Success: Different Instrns take different cycle time

Four FMULs while one FDIV

Implies – Out-of-Order Execution

Each unit is capable of doing some operation, each unit; all units need not be symmetric. So, there can be some units, which is doing load and store. There can be some units, which are doing add, minicher arithmetic. There is be some units, which is doing single persistent floating point. There will be some unit, which is doing double precision floating point like that. Now, when the instruction is decoded and it is available. Now, the hardware by looking at the instruction has to decide whether, it should go, it is an integer instruction or it is a Boolean instruction or it is a jump branch transfer instruction or it is an floating point instruction.

It is a single precision floating point or double precision floating point then there are several instructions like sim D, single instruction multiple data instruction. So, it has to find out what and correspondingly allocate to that unit as I have multiple units. So, I have an execution in, it is not as necessarily a everything be symmetric, but for a simple understanding of the MM, out of the, out of order execution.

Let us assume that all these units are all symmetric, every unit is capable of taking every operation ok. Now, suppose, I have to say 4, FMUL, floating point multiplication while one floating point division, floating point division.

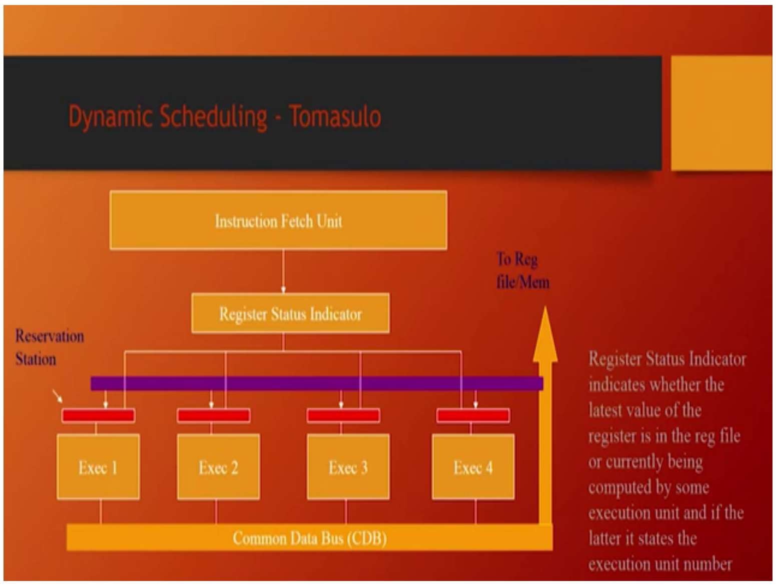
Let us say floating point division is the first instruction to be executed in your program order, in your assembly program order FDIV comes first, followed by 4 FMULs. FDIV will take 40 cycles FMUL will take only 10 cycles. Let us say all these units can do floating point. I do the FDIV in 1 unit by the time FDIV finishes the remaining 4 FMUL can also finish right. So, that is a very-very interesting thing right. So, FDIV comes first, but FMUL, FMUL 1, FMUL 2, FMUL 3 comes after FDIV. Now, before FDIV actually finishes, FMUL 1, FMUL 2, FMUL 3 can finish and be ready with the answer, the moment FDIV finishes all the answers, can be sent back.

Now, when is this possible, if the result of the division is not used by the subsequent instruction then what could happen? That four instructions can execute together at that same place and this the, the, this is the out of order execution, introduction to out of order execution. So, what does essentially happened here is that you are the 4 multiplications, at least 2 or 3 of the multiplications that followed FDIV got executed finished and they were ready, before FDIV finished so; that means, that there is a program order.

There is an assembly program order in which the instructions appear and in this case, the, an instruction that came after a particular instruction, actually finished. It is execution before the previous instruction like FMUL has come after FDIV, but before FDIV has finished, couple of FMULs can also finish and this is possible, if the, the input to the multiplayer multiplying instruction. The data to the multiplying instruction, should not be the output of FDIV, if FDIV is calculating the output of the instruction then FMUL has to wait, but if it is not doing then FMUL can still proceed and it can keep the answers ready as soon as FDIV finishes, this can basically, put the answers back into the memory.

So, this is how the, advance pipelining work and we have explained, this in a good detail in the I S 2 course.

(Refer Slide Time: 08:00)


Dynamic Scheduling - Tomasulo

Register Status Indicator indicates whether the latest value of the register is in the reg file or currently being computed by some execution unit and if the latter it states the execution unit number

Now, I am just basically covering this Tomasulo algorithm, because this is something very important for us to understand, the meltdown, this I have done in I S 2 course, but I will again do this. So, this is how the entire micro architecture basically, looks like. So, there is an instruction, which unit, which can fetch 1 or 2 or many instructions and then there is a register status indicator, which says that, you know, if the register status indicator is 0; that means, the latest value of the register is in the register file, if the register status.

Indicator is 1; that means, the latest, it is nonzero say some I; that means, the execution unit, I suppose, the register status indicators 2; that means, the execution unit 2, it is currently computing, the value of the register and that is the latest value.

So, the latest value if, if the, if I have a nonzero value in the register status indicator; that means, the latest value is not in the memory. It is currently under computation by, by one of the execution units and what like, we also do is instead of just storing something, we can store that say the value 3 means, the current value of the register is not in the register, but it is going to be output by the execution unit 3.

So, this is how the, whole thing, this is how the register status, indicators is after the register status indicator. There is a, there is 1 unit, which basically, schedules the instruction depending upon each unit. Let us assume that each unit cannot do everything, every unit cannot do everything. There will be some unit doing integer arithmetic's, some unit doing floating point arithmetic etcetera.

We have multiple floating point units multiple integer units. Now, the, the, the register status indicators subsequent to the register status indicator the, the instruction should be, if you have an integer instruction should go to an integer, execution unit. If they have a floating point instruction, go to the floating point execution in it. So, some scheduling happens that and this scheduling happens dynamically that's. So, it is called a dynamic scheduling, dynamically means when the program is in execution. It is not done by the compiler; it is done by the program, when it is an execution.

So, what will this, So, if I get that load store, instruction or say let us say I get an, arithmetic instruction and let us assume, without loss of generality on this slide, when you see the execution unit 1 basically, can do arithmetic instructions. So, and that is free, now your instruction is basically, you know put into this execution unit 1, it comes to the reservation station and then, you know starts executing at the execution state, execution unit 1, after that the result of that is basically, written on the common data bus, which goes to the register file or memory right, wherever the destination is and also that values are fed back into this execution.

It immediately their suspect before, it actually goes to the memory unit or register fail, etcetera. The answer is basically, HM you know fed back to all these. So, these two things happen in parallel, I send something to the register file or the memory saying, though this operation is over before, this actually reaches, the you know, register file or the memory, that can also that the, the whole data actually, gets populated back in some of these reservation station, if there is any right.

So, that is very important, because, if I want to implement out of order execution and of a very very strong out of order execution, when I want to implement that, I need to have these type of, execution profiles right. I need to start executing multiple instructions, at the same time and if that is not possible then you will not get the performance and that. So, this is extremely performance driven the out of order execution is performance driven and this entire dynamic scheduling that we are talking of is also performance.

So, instruction; So, let me just very quickly some of this slide, the instructions are fetched one by one and they are decoded to find the type of operand operation that it is add or subtract or anything and dependent and, and also it is decoded to find, I find the source operands, for that operation. So, if I say add two numbers, where are those two numbers stored, is it in memory? It is in general purpose register that is the sort of, input that is needed from the instruction fetch unit.

So, as you see in the slide, the register status indicator indicates whether, the latest value of the register is in the register file or it is currently being computed by some execution unit, and if the later is it states, the execution unit number. So, the registration, register status indicator, which says 0 0 means, there are current values, there in the register file. If you say it is pi, means the current value is currently computed by execution unit pi right. So, you wait for that and take the data right.

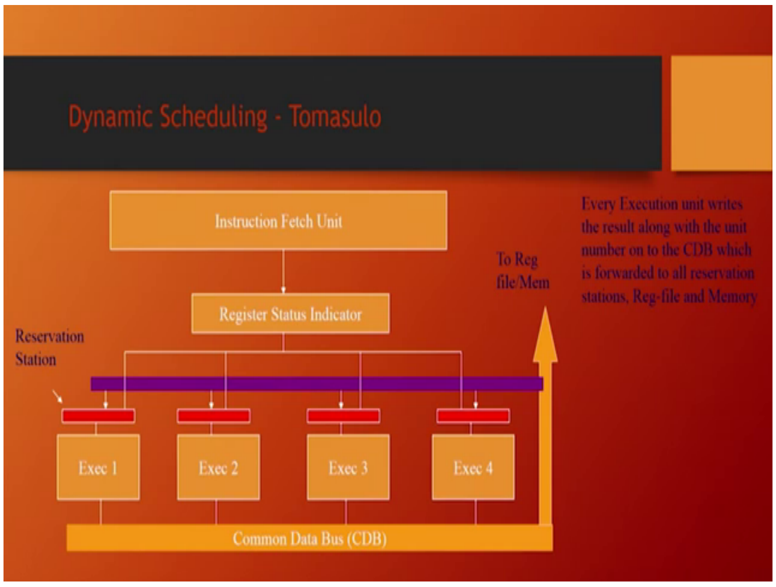
So, that is what I am reading the last sentence HM, register status indicator indicates whether the latest value of the register is in the register file or currently being computed by some execution unit and if the latter is true, it states the execution unit number. So, the register file itself will say if it has pi and say the latest value is going to be written by pi. So, what happens after this, you come after the register, status indicator, you just come if all the data input, data are available then you start executing the program.

If some data is not available then what you do you, you keep waiting on the bus right and whenever, suppose, I say, I need the value of, some, some register and it says that the execution unit pi is actually reading that register currently, handling the register. So, I go to the reservation station marked in red here and wait for pi to give an answer, when pi completes.

It writes back in the common data bus that, whatever brown color that you have now, on the common data bus, it writes the value, which will actually go to the register file or memory file, but at the same time, it also gets propagated back into this, through the violet bus onto each one of those reservation stations right.

Once so when I am waiting for say unit phi, if phi writes that whenever I see a phi writing into that well. So, when I, when something is written on the common data bus. They also write the execution unit number. So, when I am looking at the, violet line, I find a execution unit phi is coming. So, immediately I can start taking that data and start handling it. So, these are all some of the very interesting important fact about, you know dynamic instruction schedule.

(Refer Slide Time: 15:19)

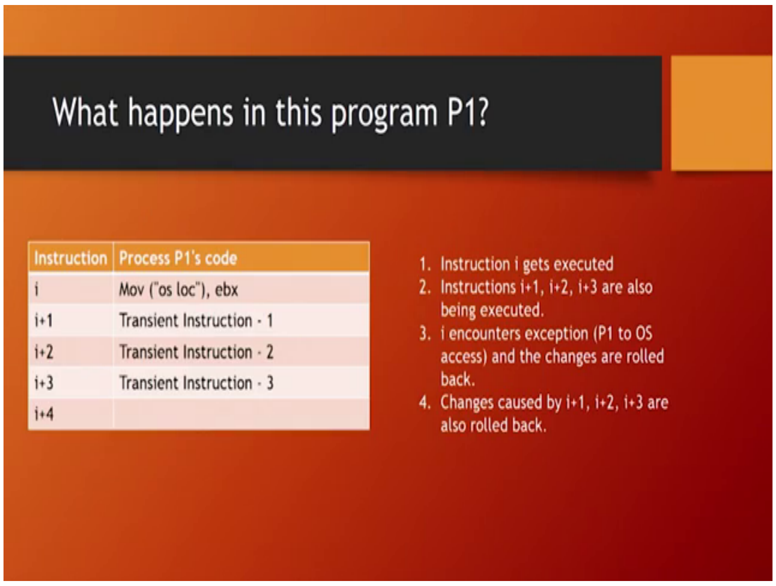


Dynamic Scheduling - Tomasulo

Every Execution unit writes the result along with the unit number on to the CDB which is forwarded to all reservation stations, Reg-file and Memory

So, every execution unit as you see on here, writes the result along with the unit number, along with the unit number on to the common data bus, which is forwarded to all the registration, reservation stations, register file and memory ok. So, you got, a quick overview of, what we are calling as the out of order execution. Now, let us now, we are coming to this program. Suppose, I have, say this is a block of instruction, instruction i, i plus 1, i plus 2, i plus 3 right.

(Refer Slide Time: 15:58)



What happens in this program P1?

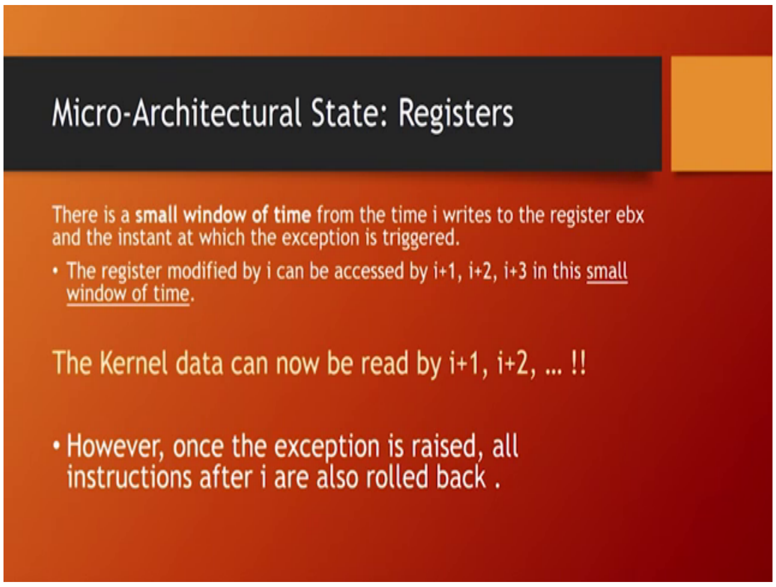
Instruction	Process P1's code
i	Mov ("os loc"), ebx
i+1	Transient Instruction - 1
i+2	Transient Instruction - 2
i+3	Transient Instruction - 3
i+4	

1. Instruction i gets executed
2. Instructions i+1, i+2, i+3 are also being executed.
3. i encounters exception (P1 to OS access) and the changes are rolled back.
4. Changes caused by i+1, i+2, i+3 are also rolled back.

Now, what I do? It is I go and my objective is to go and read one O S page and show that I have written, read it. So, what will happen? Instruction, I will try to move the O S location to a register e b x right. When instruction, I gets executed, what will happen is the instructions, i plus 1, i plus 2, i plus 3 are also being executed, because it is a superscalar processor and suppose, there is no data dependency between the, Mov instruction and the three transient instructions, then basically what will happen is, the instruction I gets executed by the time, your instruction i plus 1, i plus 2, i plus 3, are also being executed.

What will happen is I will encounter an exception, because I am trying to read O S page, Mov O S lock comma i b x. So, first see the ith instruction. It will immediately get yeah exception. So, all the changes caused by i plus 1, i plus 2, i plus 3, etcetera have to be rolled back, because now, an exception has come. Now, after curing this exception again I start running, i 1, i 2, i 3 that is the correct way of doing it. So, so, but then it is, it is quite, large amount of time. So, when I encounters an exception peon to OS access, then all these changes that are done by i plus 1, i plus 2, and i plus 3, are basically, annulled or we call it as rolled back.

(Refer Slide Time: 17:50)



Micro-Architectural State: Registers

There is a **small window of time** from the time i writes to the register ebx and the instant at which the exception is triggered.

- The register modified by i can be accessed by i+1, i+2, i+3 in this small window of time.

The Kernel data can now be read by i+1, i+2, ... !!

- However, once the exception is raised, all instructions after i are also rolled back .

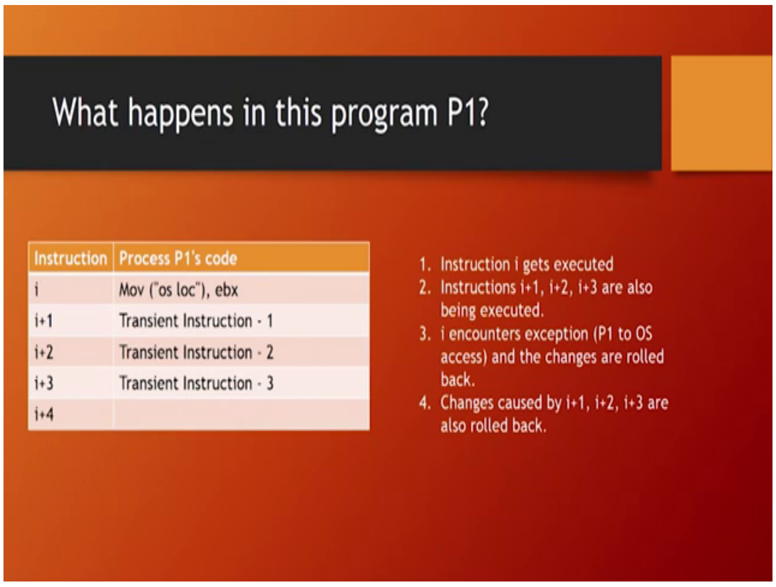
Please note here that, there is a small window of time, very small window of time. From the time I writes to the register e b x and the instance, at which the exception is together right. So, the register modified by i, can be accessed by i plus 1, i plus 2, i plus 3, in this small window of time. So, essentially what happens is for performance reasons? i, i as a process try to read a OS page right. Some entry in the OS address space. Now, when will that deduct? So, two things happen, one thing is there is something sent, to verify whether I can access or not right, whether it is a user proof mode page or a supervisor can access or not. Another side, it will go and fetch and keep, it will not go and commit it into your register right. So, that when this chick gets over, I can just take that and commit it and take it forward ok.

So, what happens essentially is when you, when I go and access a Kernel space, a memory location in the kernel space, the data is actually fetched and it is ready before, actually an exception can happen. So, the data that the instruction gets executed per say, but then when an exception happens, everything is rolled back right. So, this is very very important here right. So, this is, because I have things like out of order execution, which allows the next instruction to execute, when I am still executing, or the next instruction to go forward while I am still, executing right the so, that is one thing.

So, that is the major, part of what we say it is the out of order execution and it is sometimes, it is also called speculative execution. There is a branch i. So, after that conditional branch, there are some instructions, we do not know whether that instruction, whether that branch will be taken or not taken.

So, we just speculate that ok, the branch will be taken and then go, start executing those instructions, if your, thing is correct, you have fine, if your thing is wrong then you roll back. Roll back is a very costly affair, so that is, that is also, one very, important thing that we need to keep in mind.

(Refer Slide Time: 20:21)



What happens in this program P1?

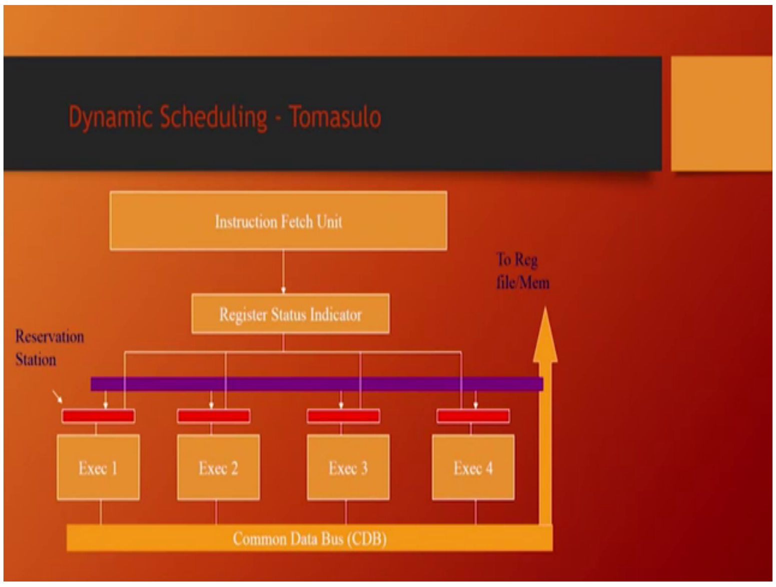
Instruction	Process P1's code
i	Mov ("os loc"), ebx
i+1	Transient Instruction - 1
i+2	Transient Instruction - 2
i+3	Transient Instruction - 3
i+4	

1. Instruction i gets executed
2. Instructions i+1, i+2, i+3 are also being executed.
3. i encounters exception (P1 to OS access) and the changes are rolled back.
4. Changes caused by i+1, i+2, i+3 are also rolled back.

So, what happens here, I basically access a Kernel location and that value is already brought and stored in ubx after that only somebody raises an exception. Now, along with this Mov, there could be other instructions like transient instructions, which are waiting for ebx right. So, what will happen is, when this instruction Mov OS loc comma ebx executes before an exception is raised, as you go back to see here.

(Refer Slide Time: 20:52)

Dynamic Scheduling - Tomasulo



Let us say, execution to unit is executing that value of the other three transient, instructions are waiting in execution 1 3 and 4 for the value of ebx, when execution 2 unit finishes and when it is trying to write into the register or memory or, so, every other execution 1 3 and 4 will get this value true that you know that violet, bus it will get back that well.

So, I love transient instructions say suppose that I am reading into a Kernel memory into any, in your register ebx, all the transient, three transient instructions would now, have the value of the ebx, after this only that is her exception saying a, what he, this is wrong, he should not have accessed, then the hardware itself goes and rewinds everything. It undo's everything right. So, by the time this hardware undo something, that the entire meltdown attack is some instructions get executed right. Before the exception is raised done, done and that thing basically we need to, that, that is the point where these, these instructions get executed and by executing this instruction, they somehow see that, some information is leaked right.

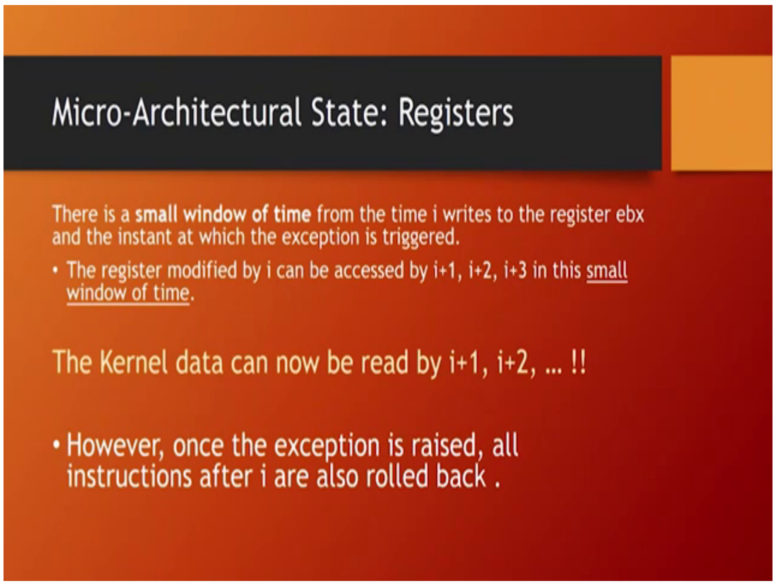
So, let me again tell you I am having an instruction say, executing an execution unit 1 which is going to basically read a Kernel at that space into a register ebx then there are three transient instructions as you see, which all will start using ebx. So, all those three transient instructions are waiting in execution 2 3 and 4, waiting for this value of e b x. Now, the first instruction will read from the memory, OS Kernel location space and it will, it will come here and it is about to go and write into memory on its way. The data whale, it is going to memory, it will also get fed into execution 2 3 and 4. They are waiting for this execution 1 unit itself.

The moment it comes your execution, 2 3 and 4 will have the value of the Kernel address space and it will do something. So, that it can communicate to the external world. This is the value all right. So, this is the basic thing, where out of order execution has helped us, if it is in order then there is no bypass that I need to do, there will be no, no multiple units, there will be no violet bus and.

So, it is gone, but now, since I have done out of order, I am now, in a position to access that ebx just this is the code sequence. So, instruction I gets executed, instruction i plus 1, i plus 2, i plus 3 are also being executed. Now, I encounters an exception P 1 to O X has access changes are rolled back and the changes caused by i plus 1, i plus 2, i plus 3 are also rolled back right.

So, this is the important thing. So, when so, that is why we, we still now, we are thinking that we are safe. Now, the, I tries to access the Kernel address space, then immediately, you know and, and store it in ebx. Now, that happens and before that the hardware says that this should not happen actually, the other three instructions, which are waiting for this value say ebx, will get that value of ebx and now, they can start doing whatever they want and at some point of them they will all be removed, because, i itself is wrong. So, i plus 1, i plus 2, i plus 3 will, will get annulled.

(Refer Slide Time: 24:37)



Micro-Architectural State: Registers

There is a **small window of time** from the time i writes to the register ebx and the instant at which the exception is triggered.

- The register modified by i can be accessed by i+1, i+2, i+3 in this <u>small window of time</u>.

The Kernel data can now be read by i+1, i+2, ... !!

- However, once the exception is raised, all instructions after i are also rolled back .

All the things that it has done will get annulled, because of this, but by the time it gets annulled, that is, what we were said there is a small window and in that window, we can basically leak the information of ebx to the external birth right. So, this is, this is basically the important. So, now at a micro architecture level I have registers, and the register store value, the registered modified by I can be accessed by i plus 1, i plus 2, i plus 3, in this small window of time before everything gets entered, it will not. So, the values of this ebx, the instructions which are i plus 1, i plus 2, i plus 3, it will not only take this latest value of ebx, but then, it can take any value from there.

So, the Kernel data can now, be read by i plus 1, i plus 2, etcetera and; however, once the exception is raised, all the instructions are rolled back and all these will go somehow before I lose this information as an attacker, I need to basically somehow, store that information in some form. So, that, the, the parent process, basically access it and then fix it out. So, this is the.

So, in the next section, we will talk about recovering from exception, and then, we will continue with the meltdown attack accordingly.

Thank you.