# Computer Architecture

**Prof. Smruti Ranjan Sarangi**
**Department of Computer Science and Engineering**
**Indian Institute of Technology, Delhi**

## Lecture - 29
## Principles of Pipelining Part-IV

Let us now talk about interrupts and exceptions.

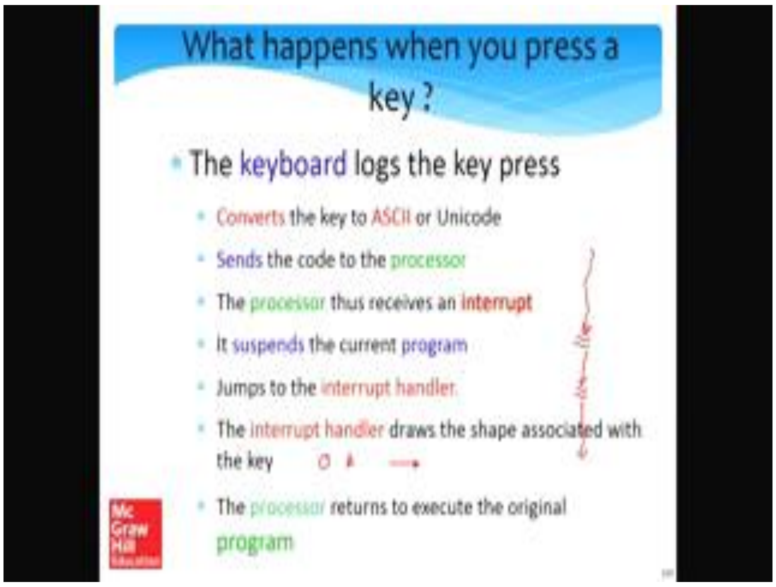
(Refer Slide Time: 00:25)

### Outline

- Overview of Pipelining
- A Pipelined Data Path
- Pipeline Hazards
- Pipeline with Interlocks
- Forwarding
- Performance Metrics
- Interrupts/ Exceptions

So, these are very important features in modern pipeline processors and they are required particularly while interacting with IO devices. What are IO devices input output devices like the keyboard the mouse.

(Refer Slide Time: 00:45)


## What happens when you press a key ?

- The keyboard logs the key press
  - Converts the key to ASCII or Unicode
  - Sends the code to the processor
  - The processor thus receives an interrupt
  - It suspends the current program
  - Jumps to the interrupt handler
  - The interrupt handler draws the shape associated with the key
  - The processor returns to execute the original program

So, what happens when you press a key? So, when you press a key on keyboard, this is the precise sequence of actions. So, what happens is that the keyboard first logs the key press right? So, every time we press a single key, the keyboard records which key was pressed with converts the key to it is equivalent as key code, which is 7-bit number or its equivalent unicode code which can be a multiply number it depends on the language right, this is this step. Then it senses the code of the key that was pressed to the processor. So, once this is sent to the processor what the processor receives is an interrupt right. It is receiving an interrupt. An interrupt is a certain message which can appear anytime like you can press a key anytime.

So, the key board sends an interrupt to the processor, which is essentially single bit signal that look there is some data arriving it is like doorbell. You can think of an interrupt like a doorbell where the keyboard pretty much rings the bell and tells the processor that there is message waiting for it. And then what the processor does at that point you know the processor is ideal or it is running a program, if the processor is running a program it suspends the execution of the current program which stops it.

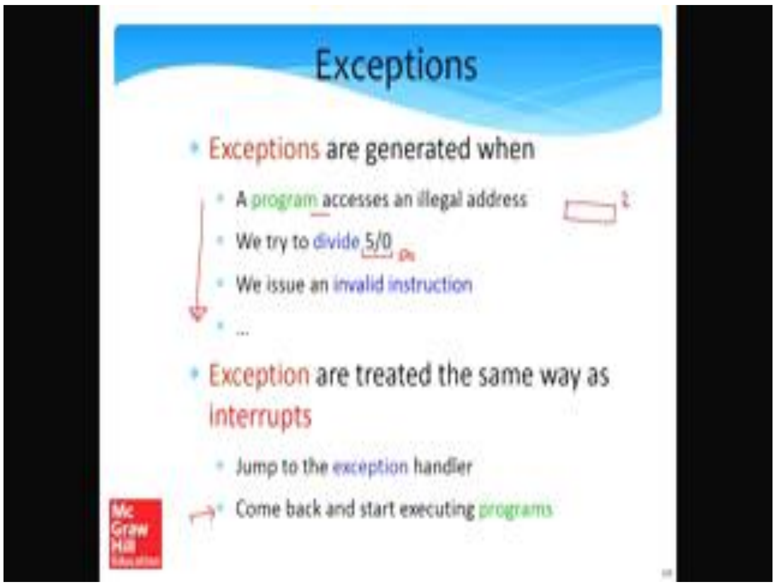
And then it jumps to the specialized program other program, which is within the operating system right. So, operating system is basically certain specialized programs to manage normal program as well as the devices connected to the computer such as the mouse in the keyboard. Once we have received an interrupt we have jumped to the interrupt handler, which is a very small program which meant to process the input. So, the interrupt handler will then. So, if I will you know press key then the key is coming the screen right. So, it is job is to invoke key that it would be itself or invoke another program to separate issue, but essentially draw the shape associated with key on the screen.

So, for example, if we click the o button, then what is drawn on the screen is pretty much in o right. So, what if I let us press o at certain point then an o is drawn. If I press k, then k is drawn. Similarly, I can receive an interrupt from the mouse or the tablet that I am using. So, my tablet is saying that the pen has moved let us say 1 centimeter to the right. So, what the processor is doing is that it is running the interrupt handler program which is moving the dot that you are seeing on the screen one centimeter to the right and also drawing a line this case.

So, after the interrupt has been processed, which means that ones you know the value of the letter that was pressed on the keyboard that comes on the screen, or let us say the mouse pointer moves the processor returns to execute the original program that it was executing right before the interrupt.

So, the way we can visualize that the program is executing then on interrupt comes. Interrupt handler executed then again the program starts, again the interrupt handler executes. So, that is the way it continues. So, this is interrupt.

(Refer Slide Time: 04:25)



## Exceptions

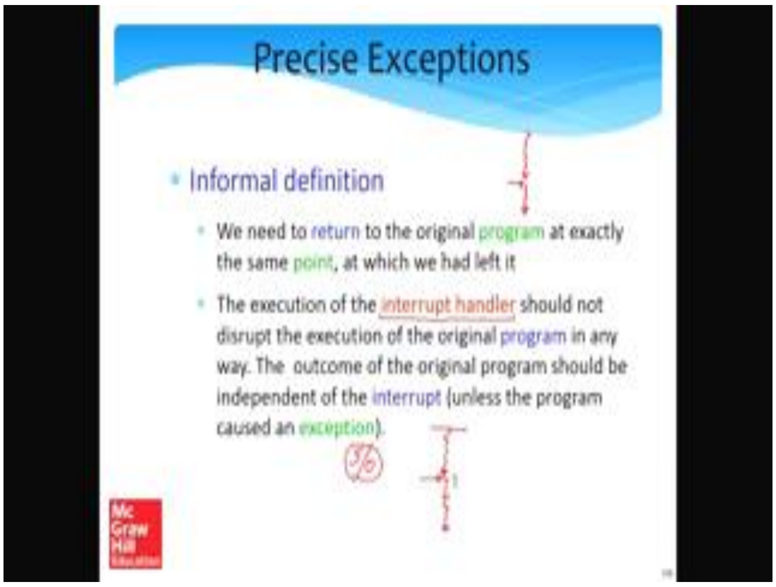
- Exceptions are generated when
  - A program accesses an illegal address
  - We try to divide 5/0
  - We issue an invalid instruction
  - ...
- Exception are treated the same way as interrupts
  - Jump to the exception handler
  - Come back and start executing programs

So, then there is a term called exceptions? So, exceptions are generated from the program access and illegal, you know it can be generated in many situations, but pretty much if the situation which is not normal it is an exceptional situation. So, one way can be when a program access and illegal address right. From the address itself is not legal right. So, you know some sort of an address it is sort if this is the address spaced that we allowed to use, some sort of an invalid address right if the program access that, then it is a calls of an exception.

Or if we divide you know 5 divides by 0 and it an interior division the result is infinity, but you know infinity does not have a representation and the integers. So, that is also an exceptional case or if you issue invalid instructions you know in invalid out code, that is also an exceptional case. So, we can have a many such exceptions in this fashion, where essentially the program does something which is not right not correct not in the normal course of actions and this is an exception.

So, exceptions are handled the same way as interrupt. So, the interrupt has an interrupt handler exceptions have an exception handler. So, the access and illegal address then the exception handler might just terminate the program, and the let the user know that an illegal address been accessed. So, once the interrupt and exception handler execute after that we come back and start executing programs, might be the same program that was executing before the interrupt or exception or might be a other program.

(Refer Slide Time: 06:19)



## Precise Exceptions

- Informal definition
  - We need to return to the original program at exactly the same point, at which we had left it
  - The execution of the interrupt handler should not disrupt the execution of the original program in any way. The outcome of the original program should be independent of the interrupt (unless the program caused an exception).

So, the most important property in this lecture that we want to define is something called precise exceptions. So, we will give informal definition first, and then we will give a formal definition of precise exceptions.

So, the informal definition is that we need to return to the original program at exactly a same point at which we had left it. So, let us assume that on the original program, we are executed some instructions and then on the interrupt came. So, when we go back we will pretty much have to start from exactly at the same point at which had left right. So, we cannot skip an instruction neither can be execute an instruction twice. So, what it is saying is that the execution of the interrupt handler.

So, you know will use term interrupt handler and exception handler inter changeably, see in this case interrupt handler means both the interrupt as well as exception handler. So, in a though meaning has to be invert from the context. So, what we are saying is that the execution of the interrupt handler should not destruct the execution of the original program in any way. So, what we mean by the fact that the execution is not disrupted in any way.

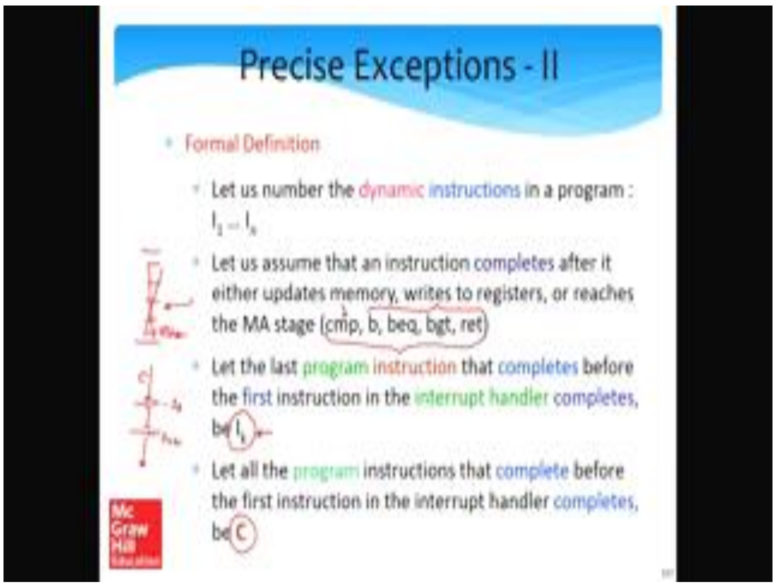
It basically means that the outcome of the original program should be independent of the fact that there was an interrupt right. Again unless it is dependent on the interrupt right in the interrupt is unrelated to the program, then the outcome of the program the final output or any of the outputs of the program or the behavior of the program should be absolutely independent of the interrupt right. Unless of course, you know it is dependent on the interrupt for example, interrupt is an exception. So, then you know something like if you do 5 by 0 then of course, you know some then you know in some systems the program may be terminated.

Then of course, it is dependent, but let us assume that in this case the interrupt or exception whatever it may be you know related to the program it is not an exception. It is just a key press, then the execution of the interrupt handler or the interrupt handlers should disrupt the execution of the program in any way. So, the output of the program will remain the same, right; the final output outcome of the program will remain the same.

And there should be destructions what so ever right. So, the same instructions will execute we need to ensure that this happens right. The process of interrupt processing is seamless. Otherwise what will happen is let us assume that I am playing a chess game, well after that in the middle right click some buttons, or let us say open an open other program into something. So, then you know of the moves of my chess game will change, but that is no happen. The reason that does not happen is because we have some notion of precise exception.

So, note that the precise exception or the precise interrupt is the same. So, the word exceptions and interrupt are used interchangeably and the reason that that does not happen is basically because we ensure that you know when a program executes is executes completely on it is own if an interrupt comes and the interrupt is not related to the program, then we process the interrupt to the executive interrupt handler and then the original program starts. With the original program otherwise is oblivious of the fact there it was interrupted in the middle.

(Refer Slide Time: 10:10)



## Precise Exceptions - II

- Formal Definition
  - Let us number the dynamic instructions in a program : $I_1 - I_n$
  - Let us assume that an instruction completes after it either updates memory, writes to registers, or reaches the MA stage (cmp, b, beq, bgt, ret)
  - Let the last program instruction that completes before the first instruction in the interrupt handler completes, be $I_k$
  - Let all the program instructions that complete before the first instruction in the interrupt handler completes, be C

So, let us now have a slightly more formal definition. So, let us consider the sequence of dynamic instructions in the program, as I 1 to I n. Let us further assume that an instruction completes after it either obtains memory right to registers or reaches the MA stage which is in this case which means that you something has happened and what is happen well let us see. So, for the cmp instruction the flag registers would have been updated, b beq bgt and ret the newth the value newth pc would have been written to the pc register right. So, let us define a notion of completeness of an instruction once again, an instruction completes if let us see either after it updates memory or right to the registers or reaches the MA stage whichever is earlier.

So basically for the branch instructions by the MA stage would have updated the program counter the compare instruction would have updated the flag suggested. So, basically that is how we defined brought notion of completeness which means that some notion of permanent stage some changes to permanent state or permanent registers would have happen. For example, if an instruction has return to memory. We say that it is complete became it is job been having been done. Similarly, if an instruction has return a register we say it is done or it is return to the flags or updated the pc we say it as that it has done.

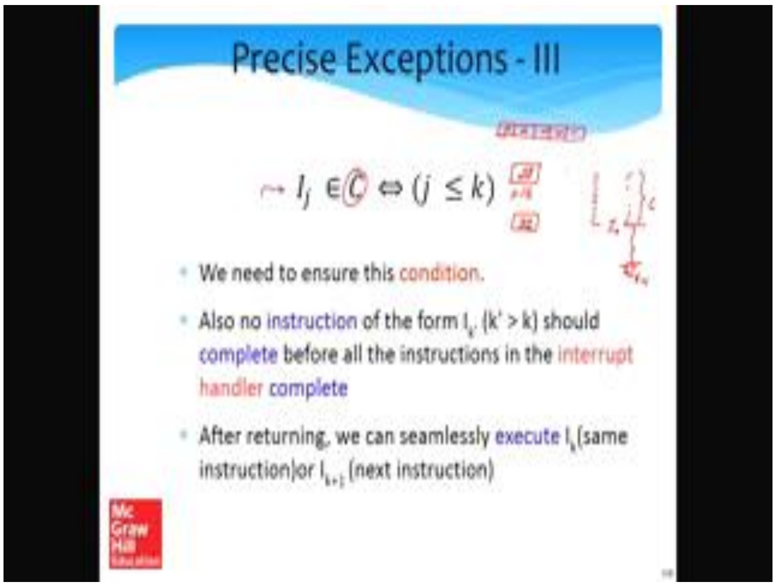
So, let the last program completes before the first instruction in the interrupt handler completes let it be Ik. See that the reason that we are defining so much is basically in a pipeline there will be multiple instructions right at one point is that, this the pipeline we will have 5 instructions at the same time. Let us assume that this point interrupt arrives. So, what do we do?

So, let us say if the interrupt arrives if we kill all of these instructions that will be wrong, because the instructions in the maybe RW stage has is maybe a store instruction which is already completed in the MA stage if we remove all of this and then you know later on start from this instruction it will be wrong because we redoing work. So, we do not want to do that. So, that is the reason we have to be slightly more careful when we define the notion of instruction completeness, and also how we ensure that we stop in already executing program very carefully, execute the interrupt handler and again come right.

So, this entire process has been done pretty carefully. So, that is the reason we define one additional terminology Ik. This is the last instruction in the program that completes before the first instruction in the interrupt handler completes. So, let all the program instructions that completes before the first instructions interrupt handler completes be represented by the set C. So, what we want is that for an outsider, he will see that all instruction in the set C has completed, and then the interrupt handler executes. It executes and totality and then the execution start. So, the last instruction to complete here is Ik, and then the execution starts from Ik plus 1.

So, recall that in some situations you might want to execute Ik once again, we will discuss when are those situations in chapter 10 but right now you need take word for it.

(Refer Slide Time: 14:24)



## Precise Exceptions - III

$$I_j \in C \Leftrightarrow (j \leq k)$$

- We need to ensure this condition.
- Also no instruction of the form $I_{k'}$ $(k' > k)$ should complete before all the instructions in the interrupt handler complete
- After returning, we can seamlessly execute $I_k$ (same instruction) or $I_{k+1}$ (next instruction)

But in general if we have done till Ik, you can start here from Ik plus 1. So, the formal correctness condition or the formal correctness idea that we want to define any Ij is element of C which is set of instructions that have completed before the instruction before the first instruction of the interrupt handler, then j is less than equal to k. Which means that Ik in program model is the last instruction to complete all other instruction that are a part of C complete.

So, recall that C was declared here that C was declared here as set of all the instructions are complete before the interrupt handler. So, if any instruction Ij is the part of C then this implies j is less than k less than or equal to k which means that this more of the instruction before it. And no instruction after Ik will actually complete. So, we need to ensure this condition. In addition, we need ensure one more condition that no instruction of the form Ik dash where k dash is greater than k should complete before all the instructions in the interrupt handler complete right.

So, if we consider Ik and Ik plus 1. So, between all the instructions in the interrupt handler have to complete right. They have to completely finish and only then will be start executing instructions Ik plus 1. Then after returning we can seamless execute Ik this either the same instruction or some cases you know especially when the execution Ik was not correct. So, it requires to executed once again or we need to execute Ik plus 1, which is the next instruction.

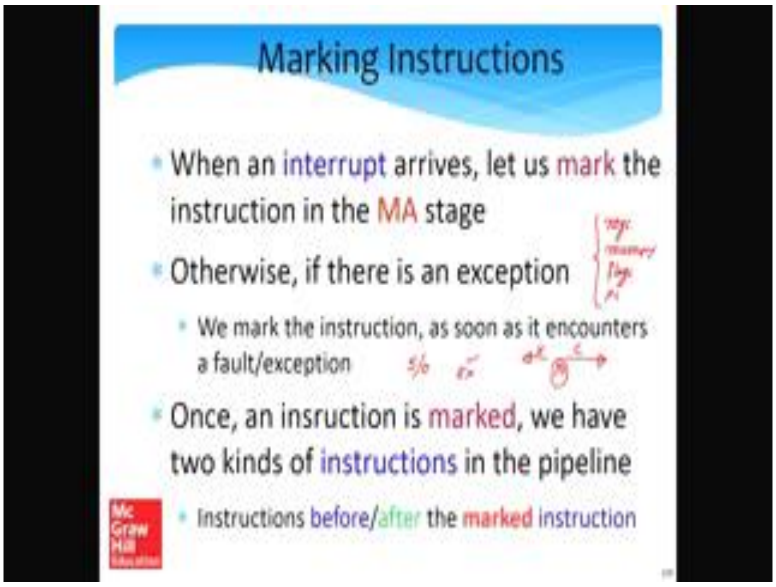
So, this is something like we are trying to ensure that the execution the interrupt handler is completely invisible. It is somehow happening in the middle it is finished it is your and went out these are not able know. And this problem is particularly difficult. You have it been a single cycle processor this would have been very easy that once the interrupt comes is finished the execution of the current instruction that becomes Ik, and all the instruction before it which the set C have already completed. And then we execute the interrupt handler and then we restart, but since it is a pipelined processor right with 5 stages 1 2 3 4 and 5.

If an interrupt comes it is slightly difficult, the reason it is slightly difficult is because you know some of these instruction you know may be this one, some of these instructions might have made some changes to the permanent state or they should be allowing to complete. And some of these instructions might not have made you know any changes to the permanent state. So, then may be instruction the fetch stage can be discarded instruction and decode stage can be discarded. And for the rest we need to see. So, you know it is like that that we need to ensure that the instructions are that have at least made some problems need to complete and some which have not they can be discarded because they have not updated any state.

So, this is required to ensure both these conditions that before Ik all the instructions have completed after it none has completed. And also when we exit when we finish which also says that once the interrupt handler starts it needs to complete all of it is instructions and no instruction the form Ik dash with k dash is greater than k you know some instruction after Ik should complete before all the instructions in the interrupt handler complete.

So, that is the broad precise definition. This is the same as informal definition means that in the interrupt handler handlers’ execution should be you know in the sense informal. And so we can clearly see that you know if you do not follow this precisely a lot of problems can happen. Consider a branch statement. See if the branch statement we allow it to a upgrade the so, I am sorry update the program counter with a new value right. So, instead of 10 offset we make it 20. And we somehow end up executing the same branch statement once again, and let us say the offset in the branch statement is 12. It will again add 20 plus 12 and make it 32 it will essentially add the offset twice. So, you have ensured that one instruction does not execute to twice unless we want it to.

(Refer Slide Time: 19:27)



## Marking Instructions

- When an interrupt arrives, let us mark the instruction in the MA stage
- Otherwise, if there is an exception
  - We mark the instruction, as soon as it encounters a fault/exception
- Once, an insruction is marked, we have two kinds of instructions in the pipeline
  - Instructions before/after the marked instruction

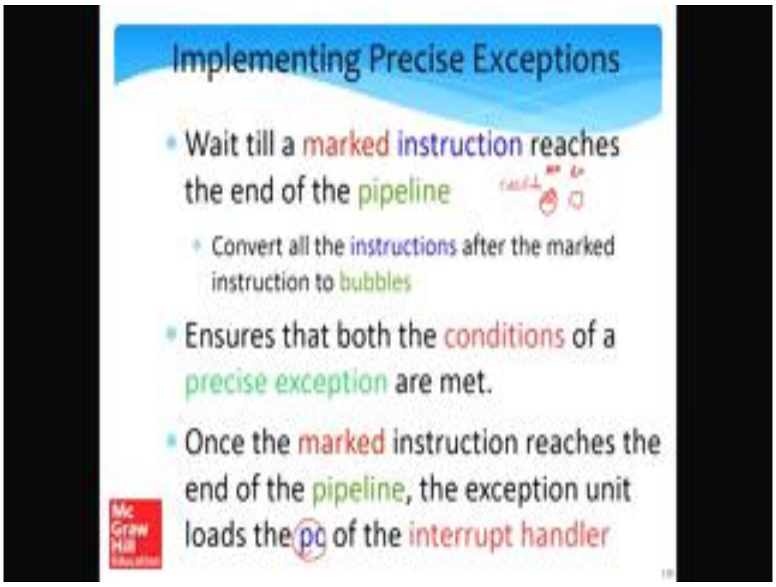
So, what we do? The way to do it is that final interrupt arrives, we mark the instruction in the MA stage because you know all the changes are being made in the MA stage even the branch instruction is making the changes in the MA stage. So this marking the instruction we will if this only if interrupt arrives, then only we will mark it in the MA stage. We will mark whatever instructions is there MA stage, otherwise if there is in a exception we mark the instruction as soon as it encounters see fault or exception and this instruction can be in any stage.

So, for example, in the fetch stage, if you know there is an exception we will mark the instruction in the fetch stage right. So, whichever stage causes an exception, or if we have a divide by 0 like 5 by 0. And the EX stage it causes an exception you know mark the instruction in the EX stage. So, recall that the way that we are marking, this is kind of simple therefore, any interrupt that often that arrives. So, we basically whatever instruction is in that cycle in the MA stage, we will mark it and for an exception we will mark it whenever we see in exceptional in whichever stage there is an exception we will mark that instruction.

So recall that the way that we are trying to this is to basically ensure precise exceptions. And there are all kinds of a state in a program. For example, there are the registers which are permanent state. There is memory which the store instruction writes then there is flag register and then there is the pc. So, whether we access all of them store all of them or you know how we manage them will be gradually clear as we keep on describing, but it is important to know that these are all the pieces of the state that we have. So, now, once an instruction is marked, we have 2 kinds of instructions in the pipeline. One set of instructions after the marked instructions and one set of instructions before the marked instructions one after and one before.

Our job is to ensure that all the instructions after the marked instructions complete right, and none of instructions before the marked instructions complete that is our job to ensure that everything after it complete is nothing before it completes right. Completes means either reaches the MA stage or makes the changes to the registers or memory. So, the flag register is not a part of this.

(Refer Slide Time: 22:27)



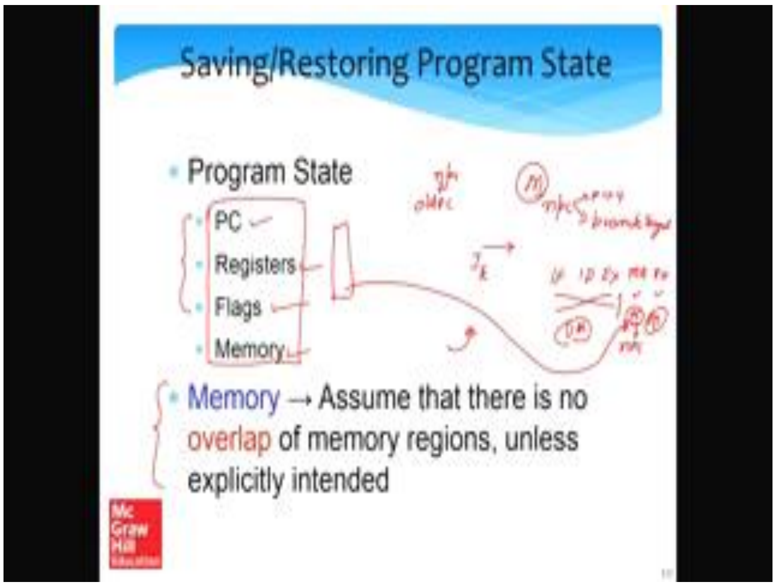
## Implementing Precise Exceptions

- Wait till a marked instruction reaches the end of the pipeline
  - Convert all the instructions after the marked instruction to bubbles
- Ensures that both the conditions of a precise exception are met.
- Once the marked instruction reaches the end of the pipeline, the exception unit loads the pc of the interrupt handler

So, we now go to the next. So, now, what we do is we wait till a marked instruction reaches the end of the pipeline. After that we convert all the instructions after the marked instructions which are there in the pipeline to bubbles. So, in the sense they are cancelled. And we also ensure that all the instructions after the marked instruction are not able to make any permanent changes right.

So, basically you know this is the marked instruction all the instructions after a sort of converted to bubbles, and once this reaches the end of the pipeline we can pretty much clean up the pipeline. So, this ensures that both the conditions of the precise instruction are met, and the reason being that we you know at least the conditions we have said that is let us assume a marked the instruction MA stage. So, wait for these 2 instructions to get out, and all the other instructions are converted to bubbles. So, they have no affect at all. So, once the marked instruction which have the end of the pipeline the exception unit will load the pc of the interrupt handler.

(Refer Slide Time: 23:37)



## Saving/Restoring Program State

- Program State
  - PC
  - Registers
  - Flags
  - Memory
- Memory → Assume that there is no overlap of memory regions, unless explicitly intended

So, the exception unit is a special additional unit that we are adding. So, as I said the program state the state of a program is it is program counter, the set of all the 16 simple risk registers the flag registered and memory. So, we assume that you know in the case of memory there is no overlap of memory regions between the program and the interrupt handler unless explicitly intended.

So, let us assume that you know memory for our program and memory for the interrupt handler is separate. So, this is all the state that we are talking about. So, we need to ensure there are instruction Ik and all the instructions before it in the pipeline, update the state and whatever state we have the same state is exactly loaded back after the interrupts handler finishes execution that is our aim. So, what is the state again that pc registers flags and memory.

So, let us add a little bit of additional word, but maybe we can have a brief discussion right here, and I will discuss the logic behind marking. So, let us assume that we have marked our instruction. So, basically in the MA stage we mark the instruction which means that this instruction and this instruction need to complete. So, if you want to exactly start our position in the instruction after the MA stage what is it that we require? So, what we would require is that if we have the pc of the instruction that is there in the MA stage then and also if we if just in case this instruction or the branch. So, we also have the next pc right the branch target, and then you know we can happily cancel all of these instructions right. It does not matter all these instructions can be cancelled. And next time we will start from the next pc which is either this pc plus 4 or the branch target right. So, this ensures that we can recover the program straight for the pc.

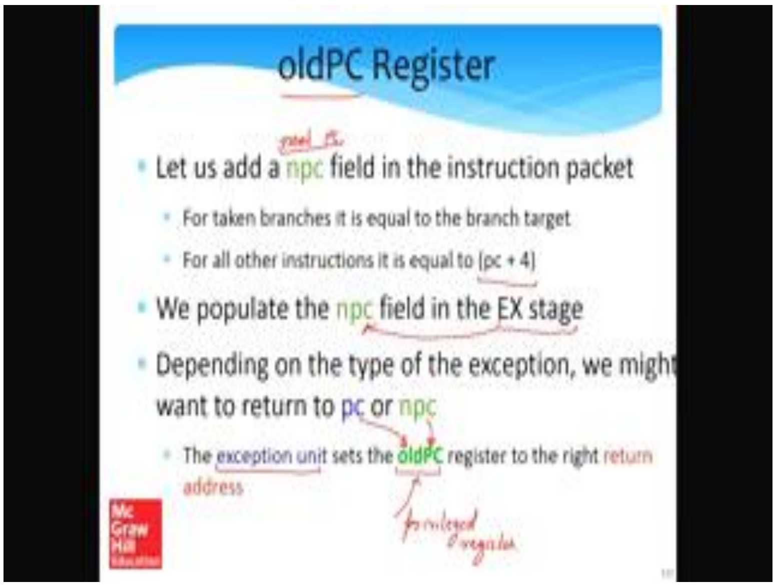
Similarly, you know no instruction before the RW stages actually modify the set of registers. So, if we do not allow any of these instructions to modify the registers and this allow these 2 instructions to leave the pipeline, the state of registers at the end if we have some way of capturing and restoring it will be able to come to the same state, as it would have been after executing the instruction after executing the marked instruction and the MA stage. So, this also can be done. Subsequently what we need to do is that when we return we need to restore the state of the flags register to the flags that the m a instruction would have seen or in the case of cmp instruction would have set.

So, we need to have some way of doing that we will find out how. And also in the case of memory what we need to do is that well. So, this can be a store instruction that is the reason we marked this stage the instruction in this stage, but none of the instructions prior to this would have actually return something to memory. So, in many cases they have been invalidated to convert to bubbles, but at least after this point all are changes to memory would have been done, and no additional changes to memory would be possible. So, memory will retain it is state because you know there no overlaps. No other program is coming and changing the data in the memory regions.

So, now let us see how to implement all of these things. So, memory as such you know saving restoring the memory is not difficult at all, because the data remains in memory. So, next when we come back we can start from the exactly the same point. The reason being that none of the so, these are all the marked instructions right. And these are the unmarked instructions none of the unmarked instructions are actually changing anything in memory we are not allowing them to. Neither is the any other program including the interrupt handler changing anything in the memory. So, the memory will remain as is. The problem is basically the pc the registers in the flags have to be the; these 3 have to be restored to exactly the same state at which we had left right.

So, this instruction in MA stage whatever it had left the value of these 3 instructions with these 3 state elements with we need to come back to exactly the same state right restore the pc the register in the flags as seen or as updated by the instruction in the MA stage right. Or let us we marking some other instructions as seen by the instruction that was marked.

(Refer Slide Time: 28:44)



## oldPC Register

- Let us add a npc field in the instruction packet
  - For taken branches it is equal to the branch target
  - For all other instructions it is equal to (pc + 4)
- We populate the npc field in the EX stage
- Depending on the type of the exception, we might want to return to pc or npc
  - The exception unit sets the oldPC register to the right return address

next pc
privileged register

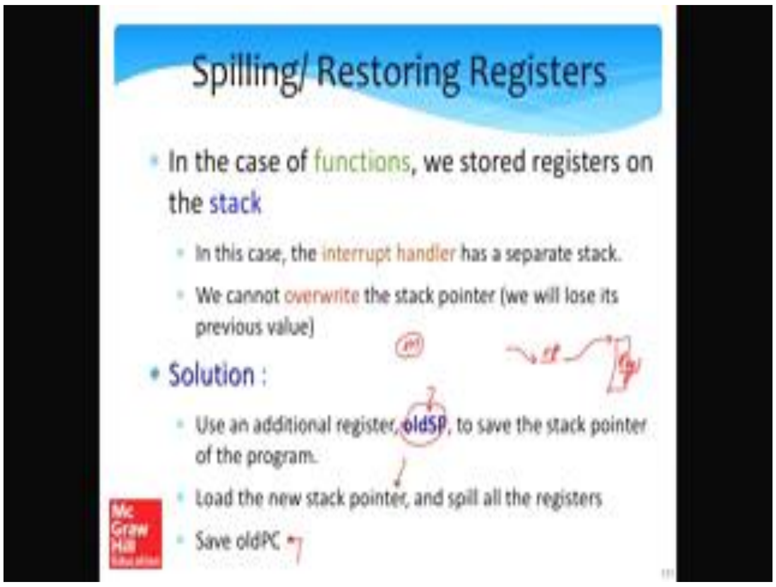
So, for the pc and registered and flags let us first look for the look at the pc. So, let us add a npc n pc is next pc filling in the instruction packet. Recall that the instruction packet is basically set of bits of travels along with instruction in the pipeline. So, for taken branches it will be equal to the branch target. For all other instructions it will be equal to pc plus 4. So, we will have this in the field. So, this will ensure well this will give some benefits, but I will talk about it slightly later.

So, in the npc in the EX stage when we compute the branch target we populate the npc field of the instruction packet. And this moves along with the instruction in the pipeline depending on the type of the exception we want to return to pc or in pc, but that is the exception unit will take care of it. So, the exception unit which is specialized hardware sets the oldPC register to the right written address right. So, which can either the pc or in pc. So, depending up on the type of the interrupt or exception one of these values will be saved in the oldPC registers right.

So, this is in the additional register it is an extra register it is also called a privilege register because normal programs cannot see it. It is an extra register that we are adding and the dedicated hardware called the exception unit. What it will do is that it will take the program counter or the next the program counter whichever the maybe the case depending up on the type of the interrupt or exception and save it in the in a new register called oldPC which normal programs cannot access, but the interrupt handlers can.

So, this will ensure that we can come back exactly the same point at which we had left right. So, basically the instruction that is marked over we can either do this instruction or the next one if it is a branch, but at least will be able to save this information via this npc field in the instruction and the oldPC register right.

(Refer Slide Time: 31:04)



## Spilling/ Restoring Registers

- In the case of functions, we stored registers on the stack
  - In this case, the interrupt handler has a separate stack.
  - We cannot overwrite the stack pointer (we will lose its previous value)
- Solution :
  - Use an additional register, oldSP, to save the stack pointer of the program.
  - Load the new stack pointer, and spill all the registers
  - Save oldPC

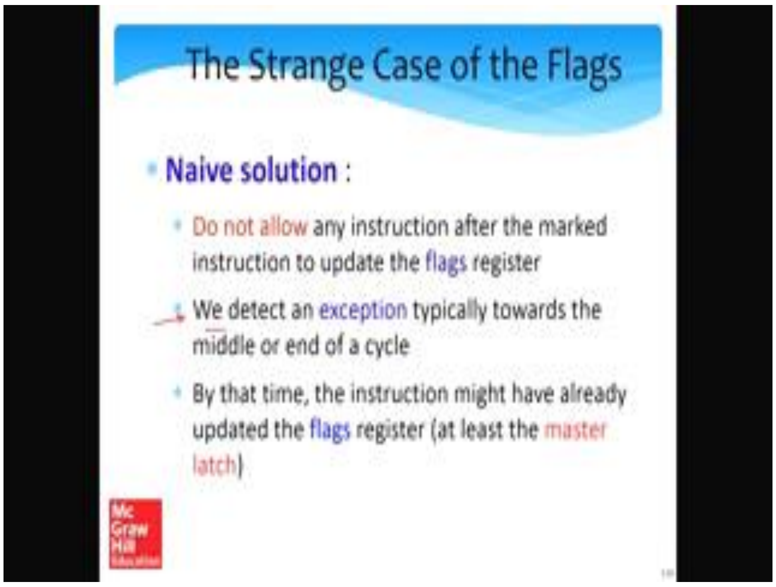
So, now let us take a look at spilling restoring registers rights. So, what where that the element of our state that pc. So, we have a way. So, you have a given a marked instruction. So, for every marked instruction you have an npc field, you know for every instruction rather. So, npc field either stores the pc plus 4 for this would pc is the pc of this marked instruction or the branch target, if it is a branch in any case is the next pc.

So, whenever there is whenever the marked instruction reaches the end of the pipeline in a special register or new register called oldPC either the npc field is stored right

And sometimes depending up on the type of the exception of course, we want to store the current pc in an execute the marked instruction once again. Now let us take a look at registers. So, we had 3 parts of the state that we needed to store that pc is taken care of. The registers we have to store and restore them the set of walls exchanges the registers and the flags registers. So, flags registers are the third. So, this is the second point. In the case functions we used to store registers on the stack rights. We used to do spilling. In this case the interrupt handler has to have a separate stack. It cannot be the same stack. Because if this is the same stack I mean it cannot be the same area because we will have to update the stack point or else what will happen to it is old value of the stack point.

So, we cannot overwrite this stack pointer right. We will lose it is previous value. So, let us consider the fact that you know we have a program and a stack pointer is pointing somewhere. Then you know we do not want to now let us say we want save it on the interrupt handler stack. Then we will have to overwrite the value of the stack pointer right. So, is that it is point to somewhere else where all the resisters can be saved. And this will kill it is previous value. So, what we do is we add an additional register called oldSP, to save the stack pointer of the program. When the marked instruction reaches the end of the pipeline and you know that is when all our previous instructions are completed we save the value of the current stack pointer which is registered 14 in a simple risk to a new register called oldSP, old stack pointer. Then we load a new stack pointer which is the stack pointer of the interrupt handler whatever it is, and we spill all the registers there. And we in addition to that we save oldPC and the oldSP of course in that region right. This is taken care of the storing of the old program counter and the old stack pointer.

(Refer Slide Time: 34:03)

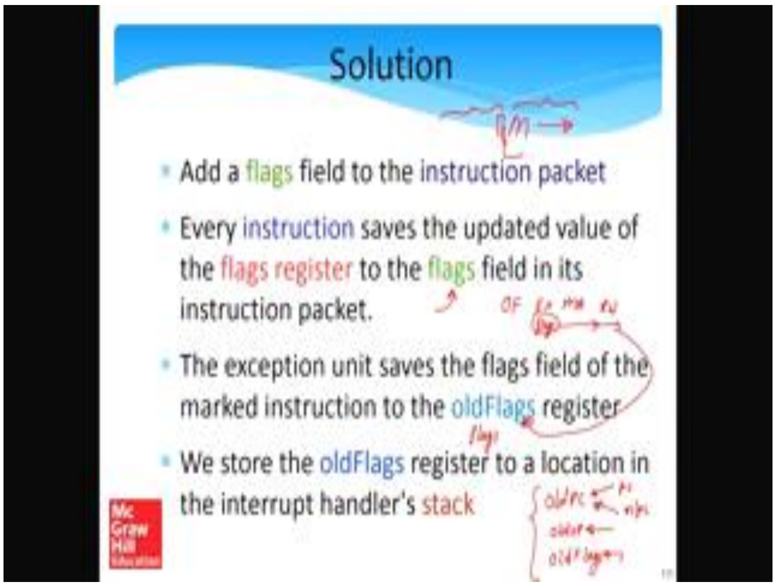

## The Strange Case of the Flags

- Naive solution :
  - Do not allow any instruction after the marked instruction to update the flags register
  - We detect an exception typically towards the middle or end of a cycle
  - By that time, the instruction might have already updated the flags register (at least the master latch)

Now, let us take a look at the flags registers. So, the strange case flags register like this let us see. So, simple naive solution will be that we do not allow any instruction after the marked instruction to update the flags register then. So, the thing is we detect in, but the problem is that we detect interrupt or in the exception.

Typically it was the middle or end of a clock cycle by that time the instruction might have already updated the flags register right or at least the latch at least the master latch of the master flip pop if there is one. So, if this is having to do you know let us not do this, because we will not know exactly when the flag registers are being updated and when it is being restored. So, that can be a problem. So, let us look at a better solution.

(Refer Slide Time: 34:55)



## Solution

- Add a flags field to the instruction packet
- Every instruction saves the updated value of the flags register to the flags field in its instruction packet.
- The exception unit saves the flags field of the marked instruction to the oldFlags register
- We store the oldFlags register to a location in the interrupt handler's stack

So, it is true that the flag register is a part of permanent state, and if let us say this marked instruction and all the ones before the marked instruction in the pipeline all of these need to complete and none of these can complete before the interrupt handler starts executing. The flags register that this marked instruction would have seen or updated is what needs to come back when we restart the program right when the instruction over here start like execute, this is what it means to see. So, one approach will be that let us add a flags field in it is instruction packet.

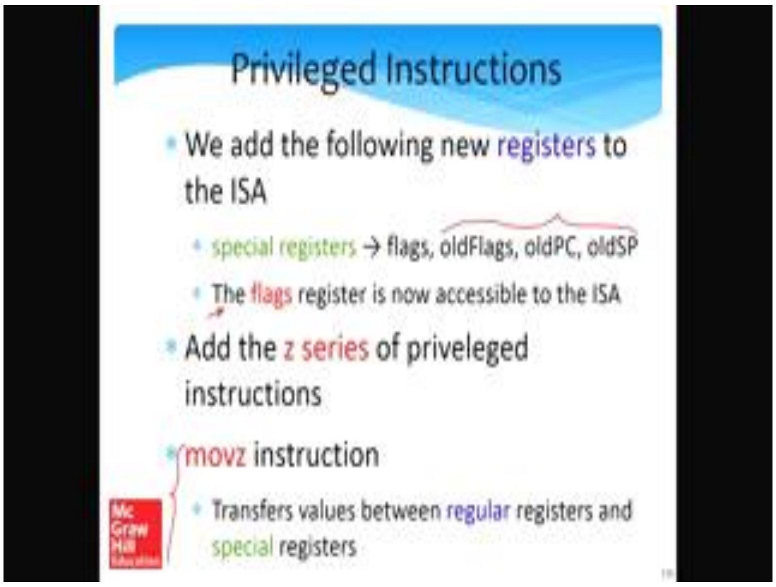
Say every instruction, will save the updated value of the registered to flags field in it is instruction packet if it updates it or the value that it sees when it passes by the EX stage. So, only this cmp instruction updates the flags register, but the rest of the instructions when they passed by the EX stage whatever value the C they will put it in the flags field of the instruction packet. This is the set of bits that it takes down the pipeline. So, the exception unit then at the end when the marked instruction reached the end of the pipeline will save the flags fields of the marked instruction to the old flags register. So, this is the nice we are doing it.

So, this basically this as you know goead and do whatever you want with actual flags register, but we will in addition have a flags field in instruction packet, which means that whenever any instruction passes by the EX stage, whatever value of flags that is it sees or it updates it carries it along with it you know suitcase, it carries along with it when it exists the RW stage, what the exception you need does is that it writes this value to the old flags register right, only for the marked instruction. So, we will store old flags register to a location in the interrupt handler stack.

So, what are the new registers that we have added we have added oldPC to store the value of the pc that we need to go back to, what are the values well this can be the pc of the marked instruction for some kinds interrupts, or at or it can be the next pc for some other kind of interrupts. Then we have oldSP. OldSP is the old stack pointer. So, the old stack pointer is basically you know the stack pointer that the marked instruction sees. So, this needs to be stored otherwise how do we update the value of the stack pointer. So, this needs to be stored this is a new instruction we are adding and also old flags. So, this stores the value of the flags register you know, we are we essentially have to one bits flags naught t n flags naught g t, but let us treated as a 32-bit register for the sake of simplicity.

So, the flags that the marked instructions see or updates, they will be stored in the old flags register once the marked instruction register the end of the pipeline. So, these are the 3 registers that we add. They are not visible to normal programs; they are visible only to special programs like interrupt handlers. So, that is the reason we will call them privilege registers with that privilege they are special.

(Refer Slide Time: 38:24)



## Privileged Instructions

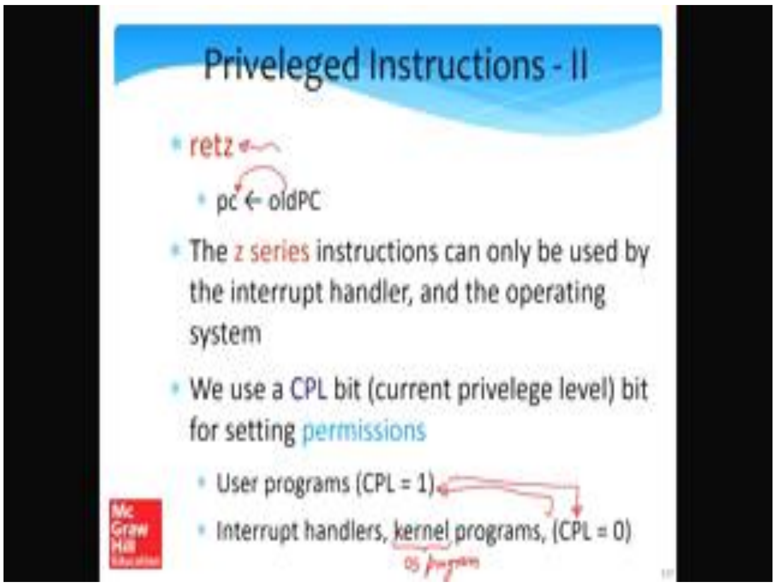
- We add the following new registers to the ISA
  - special registers → flags, oldFlags, oldPC, oldSP
  - The flags register is now accessible to the ISA
- Add the z series of priveleged instructions
- movz instruction
  - Transfers values between regular registers and special registers

So, along with privilege registers right. So, what we do is that we add some privileged instructions. So, we will add the following new registers to the ISA.

So, in the ISA you know what we can do is that we will also expose the flags register that we have over here to ISA. So, we shall explain in the short while why we need to expose the flags register to the ISA. So, these are the other privilege registers that we have added old flags oldPC and oldSP. So, as I said you know this will discuss it a short while when you see the assembly code of an interrupt handler. So, let us call this as the as the z series of privileged instructions to x series of privileged instructions. So, let us add a few more instructions to operate on these registers right the new once that we have added.

So, recall that we had discuss the simple risk instructions at in chapter 3 and we have discussed all the registers, but we are not discussing these ones because this is the new once that we are adding now. So, in general what most process designers do is that they have normal registers setm then they have a special register set, for special programs like you know interrupt handlers. So, they will typically have a slightly different encoding. So, we need to change the decode unit as well to take care of these instructions. So, with the first instruction in this class that we want to introduce is the movz instruction which transfers values between regular registers and special registers, another instruction we want to introduce is retz.

(Refer Slide Time: 40:26)



## Priveleged Instructions - II

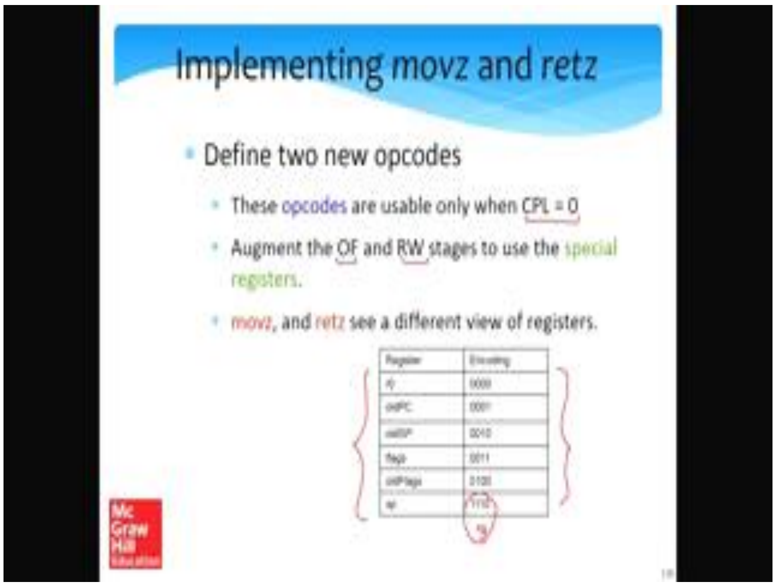
- retz
  - pc ← oldPC
- The z series instructions can only be used by the interrupt handler, and the operating system
- We use a CPL bit (current privelege level) bit for setting permissions
  - User programs (CPL = 1)
  - Interrupt handlers, kernel programs, (CPL = 0)

So, retz transfers the value in the oldPC privileged register to the current PCs. So, means we can start executing from z. So, the series instructions can only be used by the interrupt handler and other you know and the operating system. So, how do we ensure. So, the way that we ensure that normal programs cannot access these registers or these instructions is that we use a bit called the CPL bit and the CPL bit is called the current privilege level bit for setting permissions where user programs said CPL bit to be 1 and for the interrupt handler kernel programs you said it to be 0.

So, what this means is the user programs will use a certain kind of registers and instruction encoding. So, they will not be able to see the special additional registers or flags oldFlags oldPC oldSP or even the flags register for that matter. They can see that implicitly, but not explicitly; however, once an interrupt or exception handler comes the CPL bit will automatically convert from you know change 1 to 0. So, then our interrupt handlers and kernel programs will be able to access all the special registers as well as special instructions.

So, once then you know when it is kernel is not defined. So, the kernel is. So, operating system code is also known as the operating system program is also known as the kernel. And this is the same thing it is just known in our computer science called as the kernel. And a kernel is basically a special program to manage other programs right which is the job of the operating system. And once we execute once we exit the kernel and go back to the user program the CPL bit will convert from 0 to 1, which means that the special instructions and registers cannot be used.

(Refer Slide Time: 42:32)



## Implementing movz and retz

- Define two new opcodes
  - These opcodes are usable only when CPL = 0
  - Augment the OF and RW stages to use the special registers.
  - movz, and retz see a different view of registers.

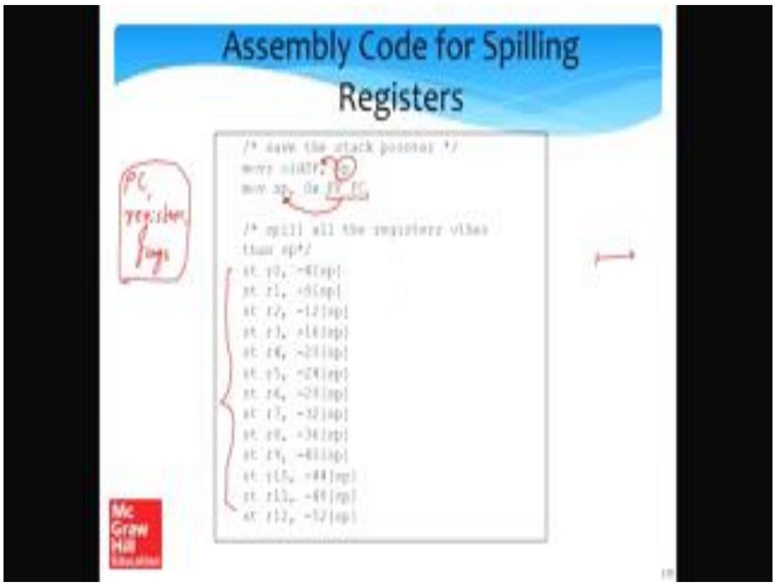
| Register | Encoding |
|---|---|
| r0 | 0000 |
| oldPC | 0001 |
| oldSP | 0010 |
| flags | 0011 |
| oldFlags | 0100 |
| sp | 1110 |

14

So, implementing movz and retz; so these of course, usable only when CPL is 0. See augment the OF and RW stages to use these special registers if there is a requirement and also in this mode we will see a different view of registers. So, let us you know it see just 6 registers. Let us see r 0 let us enquire it as 0 0 0 0. OldPC as 1, oldSP as 2 flags as 3 oldFlags as 4 and the stack pointer sp as 1 1 1 0. So, this is the same as 14. So, it maintains it is existing value for the rest we give the new value.

So, essentially in the privilege mode we see a different enquiry of a registers and these this encoding also corresponds to different registers right different privileged registers.

(Refer Slide Time: 43:31)



## Assembly Code for Spilling Registers

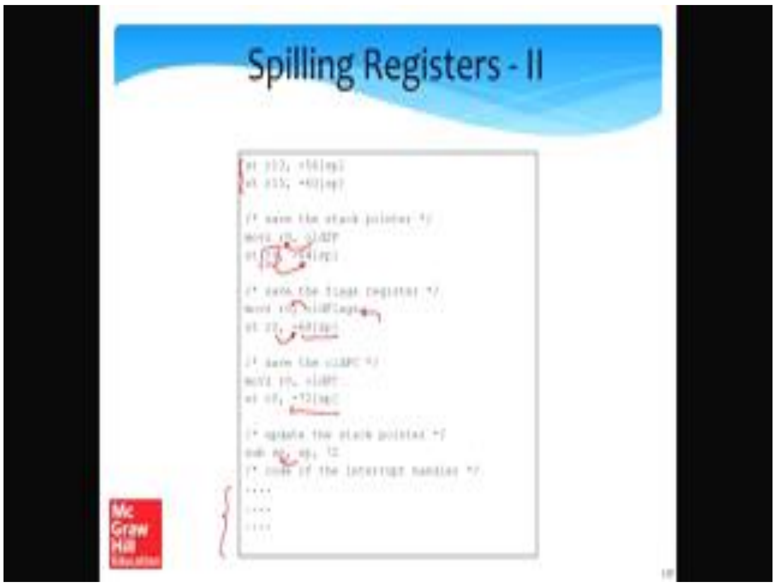
```
/* save the stack pointer */
movz oldSP, sp
movz sp, 0x FF FC

/* spill all the registers other
than sp*/
st r0, -4[sp]
st r1, -8[sp]
st r2, -12[sp]
st r3, -16[sp]
st r4, -20[sp]
st r5, -24[sp]
st r6, -28[sp]
st r7, -32[sp]
st r8, -36[sp]
st r9, -40[sp]
st r10, -44[sp]
st r11, -48[sp]
st r12, -52[sp]
```

So, let us look at the assembly code for the interrupt handler which is to spell the register; so the first thing that we do. So what is the idea? The idea is that are entire state by it is memory is taken care of for the pc the registers and the flags all 3 of them have to be stored somewhere in memory. So, as the later on we can graded from there and we can restore it. So, where will we store it? We will store it in the stack of the interrupt handler and from there we can store it somewhere else, but essentially we need needs to be stores somewhere in memory. So, the first thing that we need to do is that we need to save stack pointer.

So, once the marked instruction reaches the end of the pipeline. So, we can sort of clean up the pipeline because it is rest of the instructions we have converted to bubbles. So, we will have the circuitry anything after the marked instruction is all the bubble. So, then we load the interrupt handler and the first thing the interrupt handler does is that it transfers the value of the current stack pointer to the oldSP register. So, mind you this can be done automatically as well. So, it can be done in this case via an assembly instruction. So, both are possible in this case we are doing it by assembly instruction. Now that this is total let us assume whether stack pointer at which we want to store the set of registers in the interrupt handler is 0 x FF FC. Let us assume this is the address when that is loaded directly into the sp register and let us transform there.

So, now essentially we have loaded a new location which is the stack pointer of the interrupt handler. So, there we will start storing all our registers; so no problem. So, we start storing or registers like this we start storing r 0 at sp minus 4 r 1 r 2 r 3 r 4 r 5 r 6 r 7 r 8 r 9 r 10 r 11 r 12. So, they are simply storing the values of all are registers on the stack of the interrupt handler.

(Refer Slide Time: 45:55)



## Spilling Registers - II

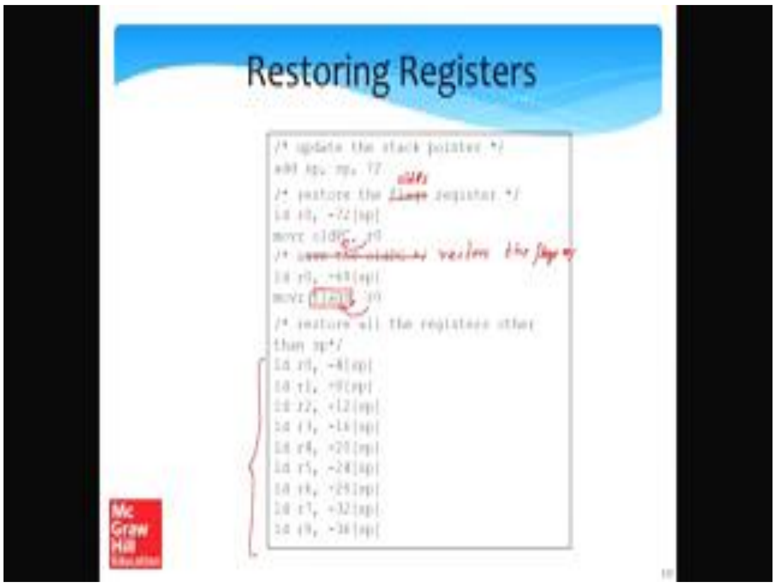
```
st r13, -56[sp]
st r15, -60[sp]

/* save the stack pointer */
movz r0, oldSP
st r0, -64[sp]

/* save the flags register */
movz r0, oldFlags
st r0, -68[sp]

/* save the oldPC */
movz r0, oldPC
st r0, -72[sp]

/* update the stack pointer */
sub sp, sp, 72
/* code of the interrupt handler */
....
....
....
```

And we store mainly r 13 and r 15. We do not stored r 14 because r 14 is the stack pointer itself. So, we do not do that then we save the stack pointer the old stack pointer in oldSP. So, what we do is, is that we move it r 0 which is the one of the registers that visible to us in this mood. And then we store the value in r 0 to memory to sp minus 64. So, after storing the old stack pointer, we now save the flags register right. So, the flags that the marked instructions have seen or updated have been automatically transferred to the old flags register. So, this can be transferred to r 0 and then r 0 can be stored in in will you use the regular store instruction.

So this can be stored in this particular memory location. And lastly we will store the we will store the oldPC. So, the oldPC we will store it in this particular memory location. So recall that you know just to keep things simple, we have not you know introduced any additional instructions beyond movz and retz and movz. Also we have said it can move data between a special and a normal register. And the rest of the instructions we have kept the same. So, once we done this we need to update the stack pointers as a you know the size of the stack. So, it is its sort of the stack is increased by 72 bytes. So, this is reflected. So, in the subtract 72 from the stack pointer and you use it update the value of the stack pointer sp.

So, now that we have saved our entire state which includes all our registers, the old stack pointer of the old flags in the oldPC. We can start the code of the interrupt handler. Because we know that all the state is same and we just have to restore all of the state to start back again. So, for restoring registers we pretty much work in the reverse fashion.

(Refer Slide Time: 48:12)



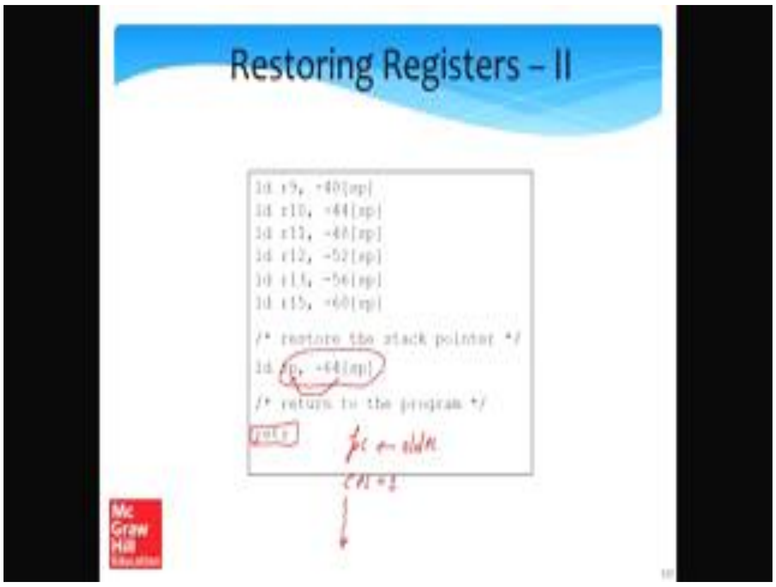
## Restoring Registers

```
/* update the stack pointer */
add sp, sp, 72
             oldPC
/* restore the ~~flags~~ register */
ld r0, -72[sp]
movz oldPC, r0
/* ~~save the oldPC~~ restore the flags */
ld r0, -68[sp]
movz flags, r0
/* restore all the registers other
than sp*/
ld r0, -4[sp]
ld r1, -8[sp]
ld r2, -12[sp]
ld r3, -16[sp]
ld r4, -20[sp]
ld r5, -24[sp]
ld r6, -28[sp]
ld r7, -32[sp]
ld r8, -36[sp]
```

So, we update the stack pointer by 72. Then we restore the oldPC, it is not the flags registers actually the oldPC. So, we restore the oldPC. So, what we do as recall that the oldPC we have saved in this location. So, we restore the oldPC right and so we transfer this to r 0, and then we have a movz instruction to that. And then what we do is that we restore the flags.

So, here what we do is that we take value the flags register and also we take the value in the old flags register. Now which was actually stored in memory and we restore it to the flag. So, the way that we do it, is that we need the value first into one register and we then we do a movz. So, we set the flag register with whatever value was previously stored. So, recall that is why we know we had exposed the flags register to assembly software just that we can restore it is value. Then we sort of walk reverse and restore the value of all the other individual registers starting from r 0 to r 15 it is just that we leave stack pointers.

(Refer Slide Time: 49:54)



## Restoring Registers – II

```
ld r9, -40[sp]
ld r10, -44[sp]
ld r11, -48[sp]
ld r12, -52[sp]
ld r13, -56[sp]
ld r15, -60[sp]

/* restore the stack pointer */
ld sp, -64[sp]

/* return to the program */
retz
```

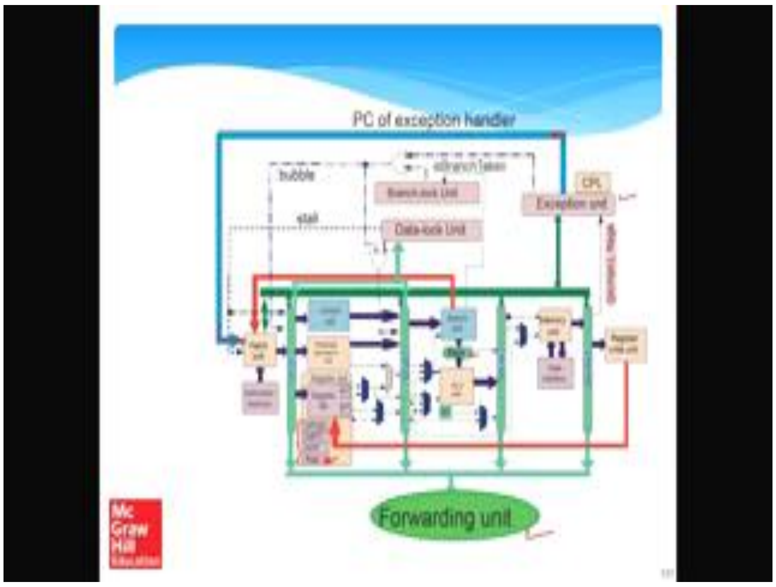
So, just know everything, but not the stack pointer we will do it at the end. So, at the end what we will do, is that we will restore the stack pointer, so the stack pointer where is it stored. Well the stack pointer is stored at this location the oldest period at minus 64 you know that the contents of sp minus 64. So, we will have this instruction over here which will take the contents of the stack pointer subtract 64, and then write it to the sp register.

So, now that we are all done and you know oldPC has been loaded we execute that retz instruction, which will basically do what the retz instruction will transfer the oldPC to the current pc it will start execution. And it will you know start execution the original program at the CPL level of 1. So that is what is it is going to do, that it will transfer and recall that the oldPC we have restored in these lines. So, once you called retz. The program counter will be equal to whatever is there in the oldPC register. And also the CPL level will be the same as that has a user program which is 1. So, a normal user program can start executing and it state would have been restore it no way of knowing that were interrupt and actually come in the middle. It is as simple as that.

So, summarize how did we do it? Well the way we did it is by actually introducing some special instructions and registers; so in specific the instructions that we introduced. So, let me clean up this line. So, in specific the instructions that we introduced or movz and retz; so movz essentially the transfers' data between normal and special registers. And retz what it does is that it is essentially returns from an interrupt handler.

So, the contents of oldPC are transfer to the program counter pc. And we start running the user program. And the user program runs with the privilege level of 1 CPL is equal to 1. And these are the 4 additional special registers that we introduced. So, why do we introduce special registers? Well to transfer all the general registers from point a to point b we need to write assembly code and the assembly code would require registers. And then you know some of the registers random up getting overrated. That is the reason we have to introduce additional registers such that our general purpose registers are not over it.

(Refer Slide Time: 52:54)



So, now that we have some understanding of this let us take a look at updated right line diagram of average diagram of course. So, what we have done is that we have of course, this new marking circuitry which can mark an instruction in any stage. In addition, that we see slash in pc obligate pc any one of these values and the flags one or both of these values, and the flags field from the instruction packet are going to the exception unit along with this CPL registered over here. So, the exception unit will control the execution of interrupt handler.

So, once the marked instruction reaches the end, we will update the CPL bits start the interrupt handler or the exception handler depending up on you know interrupt obligate exception handler. So, depending on what kind of interrupt or exception it is. So, it will essentially send the command to the fetch unit, and we will stop the regular execution and they will also follow the same pipeline pattern of execution as we have been following. The only additional detail that we need to consider out here is that in our register file we have augmented it.

So, we have added 4 additional register oldFlags, oldPC, oldSP and flags, there you know this flags and this flags are conceptually the same. And so we just assume that whatever you know the contents of this flags register and this one here are the same. So, you know this flags registers can actually be physically located over here right, but you know since it is a register we have put it in the register unit, but you know this and this are you know the same entity.

So, this is sort of our entire pipeline processor with this stall logic the bubble logic and your data has in their control has it handling logic with the forwarding unit with the exception unit to take care of interrupt and exceptions which are extremely useful things. You know otherwise; how would you interrupt with the IO devices; how would you interrupt with programs that are faults. So, an exception unit is required and we need additional instructions of course, which are decode unit will take care. And we need you know some additional registers which are saved in the register unit and of course, in the exception unit with the CPL bit.

So, this brings us to an end of the pipelining chapter. So, we have now discussed in a fair amount of detail that design of a processor and we have further optimized it with pipelining. So, now, this part of the book ends subsequently we will move to the design of memory system.