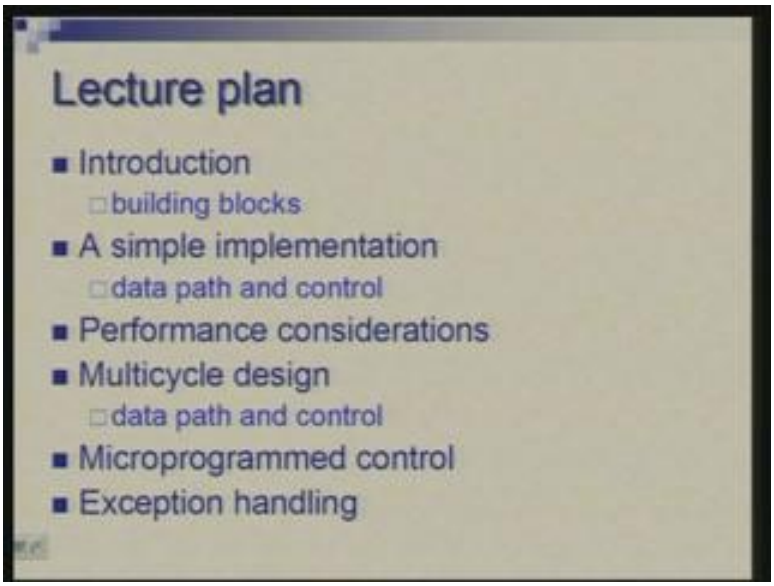
Computer Architecture
Prof. Anshul Kumar
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi
Lecture - 23
Processor Design Exception Handling

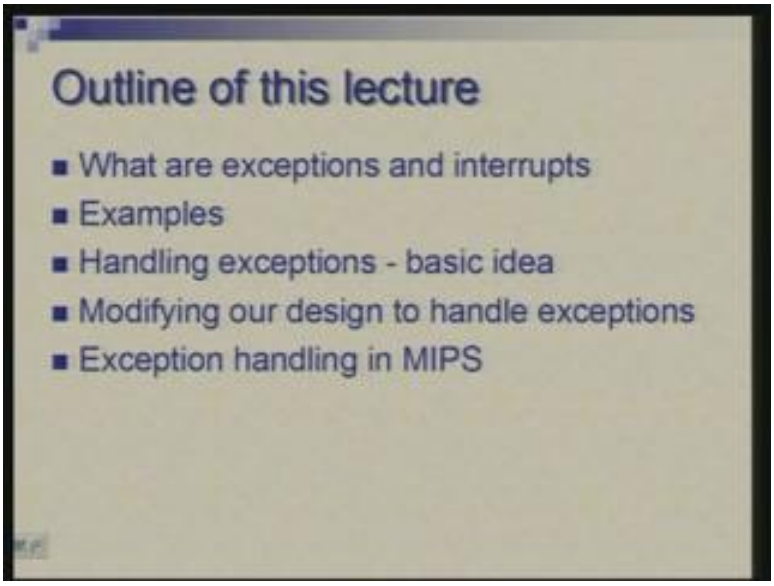
We conclude our discussion on processor design today by introducing the capability to handle exceptions. Exceptions are basically unusual circumstances or unusual events which a processor has to handle. As you would recall that we started with design of a simple datapath.

(Refer Slide Time: 1:25)



We looked at the performance issues and modified design to include a multi cycle datapath where we can first improve the utilization of resources and allow as far as possible just the right time for every instruction. We saw a specific way of controlling a datapath with the help of a microprogram and finally now we are looking at the issue of exception handling.

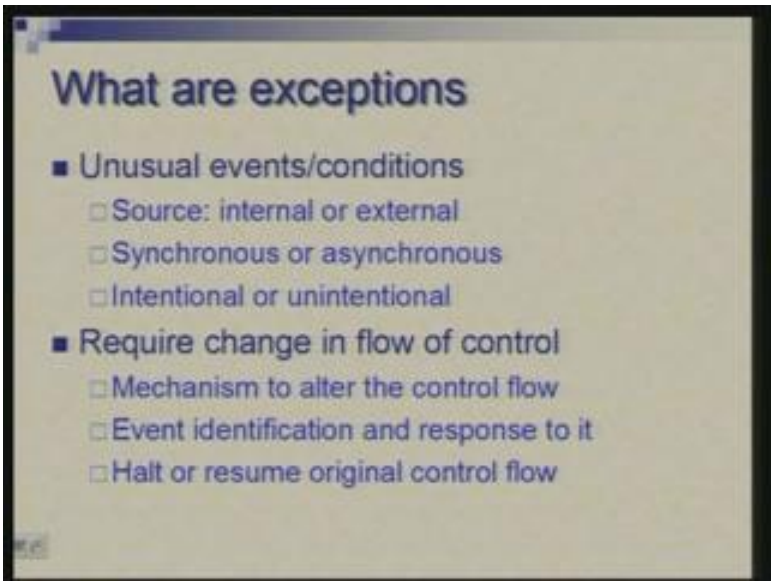
(Refer Slide Time: 1:50)



So first we will try to understand what are the exceptions and also I will look at a term which is similar in terms of meaning that is interrupt. That is what are these two terms, what are the differences, what are the similarities, look at some examples and then particularly look at them in context of the design we are doing. So we will then see how do you handle these exceptions, what is the basic idea, what is that we require to do when such a thing occurs and we will take our design which was done and incorporate this idea of handling exceptions.

Finally we will elaborate further on this and look at how exceptions are handled in MIPS processor in general so we will take a little broader picture there.

(Refer Slide Time: 2:43)



So, to begin with let us try to understand what exceptions are. As the name implies exceptions are unusual events or unusual happenings; the source of these could be internal that is within the processor or it could be something which is outside the processor. These could be synchronous or asynchronous.

Synchronous means that these are synchronized with the clocks and the instruction. So one knows exactly when it can occur, it may occur or not is a different issue but what are the time slots or clock period when we have to look for a particular type of exception. It may be synchronous in that sense that you know its timing when it is expected or it could be asynchronous it may appear or it may occur anytime and you have absolutely no idea.

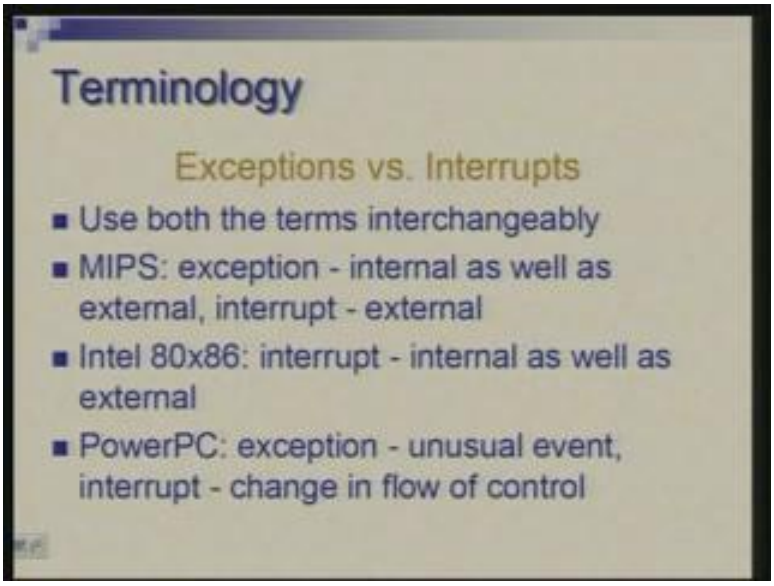
Then the motivation behind it could be intentional that is you want some action to be done but it is not the main action but it is exception so you want to handle it separately but it is something which you desire. On the other hand, exception can also indicate something which you do not desire, which you do not want to happen but for something beyond your control it happens. So still you have to react; you have to react whether it is intentional or unintentional. You will try to see situations of both kinds.

So this is something about the nature of these exceptions. The other question is what do you do when exception occurs. So the main thing is that you are executing a sequence of instructions and you need to go beyond that sequence and do something else. So the flow of control which is implied by a sequence of instructions or may be the embedded branches and jumps you have to break off from there go somewhere else execute may be some instruction which represent your response to the exceptions. So do that and then possibly you come back or you may not come back. So the question here is that what is the mechanism of making this happen. How do you alter the flow of control and the second question would be that how do you identify what event was it which is causing this break or what is causing this change of control and what exactly is the response so you may transfer control but after having transferred what do you do. It may depend upon what was the reason for exception what caused it and how do we respond to it.

If it was a intentional exception of course there is a clear defined response which is expected which has to be taken but if it is unintentional something which you don't want to happen then you have to take some suitable action in response to that. And then finally after having taken that action do you terminate the program because situation may be that you cannot proceed further or after having handled this exception you come back to what you have been doing earlier and continue. So this will again depend upon what was the cause of exception and whether it leaves the processor in a state where you can continue or not.

Now, before we go further let us bring these two terms together and look at the terminology. I mentioned that in literature there are two terms used exception and interrupts which are often used interchangeably meaning the same thing. The meaning is same and they are just used as synonyms but there are some architecture and some designers who tried to distinguish them and use them for some more different meanings.

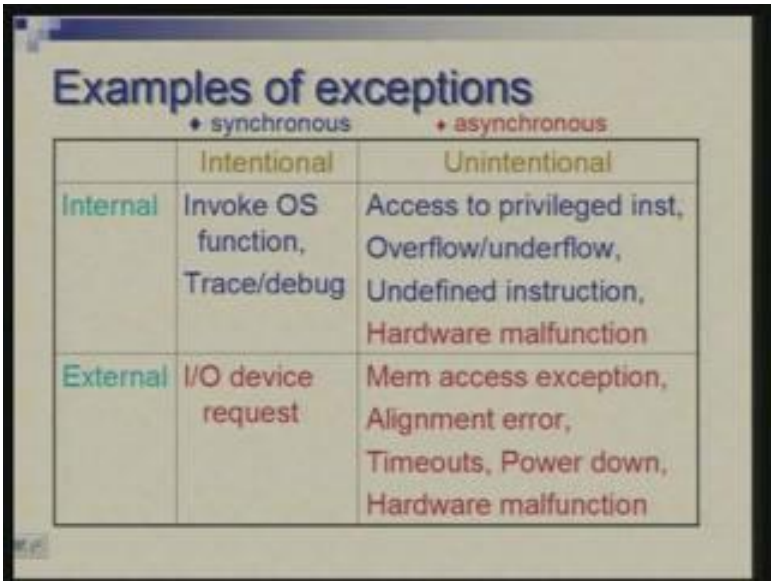
(Refer Slide Time: 6:58)



In MIPS terminology for example, exception is used as a general term it could mean internal cause or external cause; any event occurring in the processor or outside whereas the term interrupt is used only for external events. So exception is more general; interrupt is more specific. Where as in Intel 8086 80x86 terminology the term used is interrupt and it is referred to both internal as well as external events. Whereas in power PC you have exception which is the term used for the event which is occurring it could be internal or external and interrupt is a term used for the effect of it so one is the cause one if the effect and to me this sounds more sensible if you really look at the natural meaning of these two terms; exception is that event which is happening and interrupt is the effect of it that you were doing some program it gets interrupted you have to break off and do something else.

But one must be while reading literature or reading about processor one must be aware of this variation in that terminology so that one does not get confused. Now let us look at this slide which tries to list few common causes of exception and this will give you a reasonable picture of what we mean.

(Refer Slide Time: 8:40)



I have tried to classify all these depending upon whether they are internal or external that is one thing; secondly, whether they are intentional or unintentional and thirdly, whether these are synchronous or asynchronous. So if you look at the two rows first row corresponds to internal events and the next one corresponds to external events. Column-wise this column (Refer Slide Time: 9:16) refers to intentional events and this refers to unintentional events and color-wise I have tried to distinguish between synchronous and asynchronous. So you would notice that a variety of combination exists; one characteristic does not necessarily imply. So let us go over these one by one. I am not necessarily going to go in sequence but let us look at some of these.

Look at this one: underflow and underflow or overflow. So this is something some condition we sense when arithmetic is being performed and as we have discussed that when the result of some operation exceeds the limit which are prescribed by certain word size then we call it underflow or overflow. So this is unintentional; you normally when you are performing operation you would expect the result to be within the limits but if you are not making a check before hand the result could exceed the range and this could happen.

Question is what do you do when it happens. Again it may be dependent upon the intention of the programmer whether you would still like to continue with rest of the calculation or you look at it as a situation where something has gone wrong which was not intended and you want to abort the execution and stop there.

Then and of course as it is a clear internal and it is synchronous in the sense that you know when it is going to occur in a program. In an instruction when you are doing arithmetic instruction when you are doing arithmetic operation in that particular cycle you would know whether things went wrong or went right.

Undefined instruction: suppose you have 6-bit opcode which means there are 2 raised to the power 6 or 64 possible patterns of opcode which you can have but your instruction set may not

exactly be that so there may be some opcode which are undefined. What happens when you are executing a program and by whatever reason you find an instruction where the opcode field has something for which you have not designed your hardware for?

So in the simple case we have been discussing we have looked at only nine instructions but we still have 6-bit opcode field and there are many opcode which are not defined as far as our design is concerned. The hardware must be ready to respond in a meaningful manner if some unexpected instruction comes some code which is there which you cannot understand. So again this is unintentional, it is internal and it is synchronous.

Continuing within this cell (Refer Slide Time: 12:31) the set of instruction is often in a real case divided into privilege instruction and general instruction. So, privilege instructions are special instructions which are only allowed for operating system so these are used for overall resource management which operating system has to carry out and the user program contains those instructions the hardware has to check and not allow that to proceed. So, if there is an access to privilege instruction; privilege instruction is the one which the ordinary user program is not allowed to execute.

Then there is an event called hardware malfunction which means that something goes wrong in the hardware and assume that hardware has some capability there is some logic which can self test which has ability to determine whether some fault has occurred in the hardware so this can signal this condition that there is a fault in hardware or hardware has a malfunction.

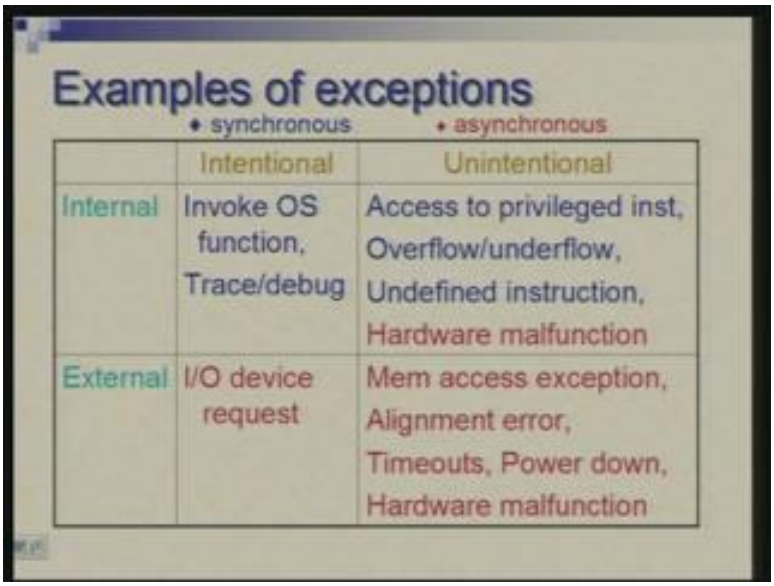
So now this is as you can imagine this is asynchronous at which point this may get detected may not necessarily be synchronized with the instruction. But it could be internal or external so therefore you will find that I have included this in both cases so some hardware which is outside the processor may malfunction and signal may come to the processor that something has gone wrong and it could be internal.

Let us look at this one (Refer Slide Time: 14:27) memory access exception. Suppose you have 2 raised to the power 32 or 4 GB of memory space but physically in a given piece of hardware you may not have entire memory filled up here; you may have a smaller memory although the addressing capability is larger but an instruction can generate an address which is beyond what the physical memory is so that is the case of memory access exception. Or also what could happen is that if in a time shedding mode there are many programs running and the operating system will assign areas of memory to different programs. So each program is expected to access memory within the area allocated to it. What happens if it goes beyond that that is again the cause of exception.

[Conversation between student and Professor: memory yeah because (Refer Slide Time: 15:36)] here I am assuming memory as external although the design which we have done so far is somewhat misleading in the sense that we have looked at memory, instruction memory, and data memory are all part of the process of design but in reality memory is a separate sub-module and then memory access actually need not be in a single cycle so memory may operate asynchronously or synchronously with the instruction. So it could be if it is a synchronous memory then you know precisely when the request is made to memory when it responds

everything synchronized with clock but often that clock could be different from the process of clock, it could be.... bus may have a different clock the processor may have a different clock. So in general it is more reasonable to assume that exceptions related to memory are asynchronous.

(Refer Slide Time: 16:45)



Here is another memory related error which is alignment error. As you know that addresses which are generated are often byte addresses whereas memory is organized word-wise. So a word may start at an address which is multiple of 4 or not. If the addresses of word are multiple of 4 the logical word and physical word are aligned. If it is not then they may be misaligned that means a logical word may be spread over two physical words some bytes of the first word some bytes of the next word.

Now, some processors may have this provision that you can disallow unaligned memory accesses because there is a performance penalty with unaligned access and then a check has to be made whether access to be made is aligned or not. Then if you are preventing unaligned accesses then any attempt to make unaligned access would be flagged as an error so that is one kind of error again related to memory access.

There could be timeouts particularly relating to input output devices.

Power down: this is something which may beyond the control of the processor and as I mentioned earlier hardware malfunction.

Now there is other world of exceptions which are intentional. And examples are that invoking operating system function so the way a user program gets some service from operating system is to create a situation which is same as exception. So the response to that will be that the control gets transferred to operating system, operating system provides the service and control gets back to the user program. So the reason why it need to be done like that is operating system is able to execute all privilege instructions and the service you may require may need that and OS has

access to all the resources all the peripherals and so on which the user program may not have direct access to. So this is a sort of a special mechanism to transfer control to operating system.

You cannot write a jump instruction or a branch instruction where the target address is some address which lies in the operating system area you cannot jump into that because that would be very uncontrolled very sort of arbitrary access to operating system you can jump into arbitrary area and may cause something which is unpredictable. So this is like a restricted area where you are taken holding your hand you are shown around and you are brought back. So, control gets transferred to OS, you follow certain path, carry out some operation and then you are not allowed to look around here and there and you come back.

So therefore, this is an intentional event (Refer Slide Time: 20:20) and it is actually some special instructions are there which are actually treated like exception the response to these are like exception and the control gets transferred to a specific area of OS.

Another main example of intentional exception is request from a input/output device for carrying out data transfer. So let us say you wanted to transfer data from disk so your program may give a command and when the disk is ready with the transfer it may send a signal which causes exception or causes interrupt and then the required service is provided.

Then lastly tracing and debugging. So in some processor or in fact most practical processor there is hardware debugging facility which would for example allow you to run the instruction step by step; one instruction at a time. So one way it could be done is that after every instruction there is an exception generated. So you do one instruction and then exception takes you to a special handler where you would interact with the user who is trying to debug the program you could display contents of memory location, contents of register and so on and after that when the user wishes to resume the program you will come back do one more instruction but again exception happens you go back and so on. So, in the user program only one instruction gets executed and you are back to the exception handler which is actually supporting or debugging.

Is there any question about any of these events or happenings?
[Conversation between student and professor:..... yeah, yeah....] external exceptions are always asynchronous yes because the way we have defined synchronous on that, that there is something which happens at a predictable time within the instruction so I think it is a straightforward implication that external events are asynchronous. But internal ones may be synchronous or asynchronous.

Yeah, timeout is that you want something to happen but it does not happen by that time and so what is done is there are timers which you set in motion so it is like an alarm you set the timer and when the time expires it sends a signal which is taken as an exception.

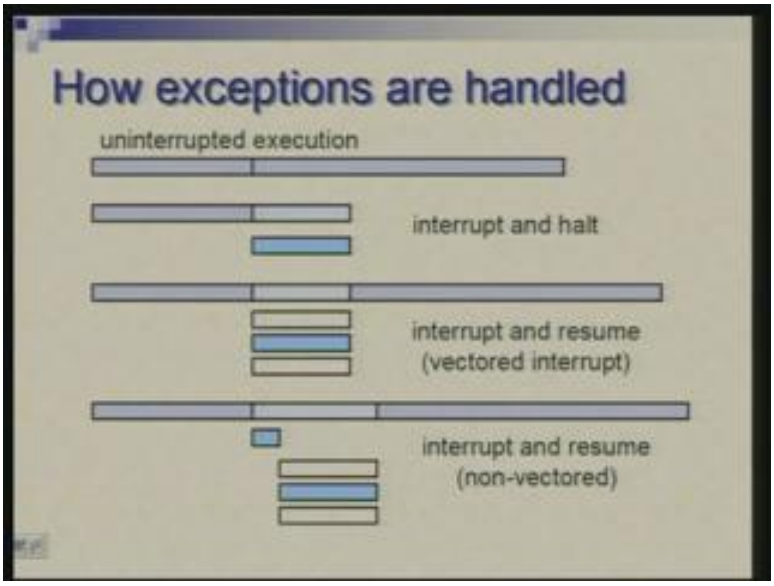
For example, suppose you wanted to do some input output you send a request to input output device and you expect response let us say within 1 millisecond. You set the timer for 1 millisecond if the device responds it is fine you shut the timer off if it does not then after 1 millisecond you will if device does not respond the timer will respond and then you can take necessary action.

Any other question?
Power failure? It could be, well, in general, what we mean here is that something goes wrong with the power you may often have some time to take an action which will for example save your current status of computation so you can save the registers and you cannot do much to save what was in the memory. Suppose you were executing a program there were some partial results in memory may be some something in the arrays but what you have in registers can at least be saved in memory. You can't save from memory to disk you may not have that much time because there may be some acceptable level of voltage beyond which the system has to be shut down. So you can generate a warning signal little before that so that you get some time to take some action.

In some embedded processor where you are running on battery you may intentionally do power down. When you see inactivity and there is no need to keep processor in active state some processor provides this mechanism that you can reduce the voltage you cannot totally shut off but reduce the voltage to the extent that contents of memory are preserved but processor does not execute instruction. So that is a state which will consume much less power so for power consumption power conservation you get into that state.

So in that state also you may have to take some action before you change the state.

(Refer Slide Time: 26:10)



Next now we understand what these exceptions are. Let us try to see in general what is it that we are expected to do. So imagine a timeline horizontally so you have let us say a sequence of instruction which is some activity is happening, so suppose this (Refer Slide Time: 27:02) represents the normal action some program is running and it is at this point you identify you notice that some event has occurred. Then a possible response is that at this you transfer the control to some other activity, carry it out and at the end of it you just stop the processor, so just

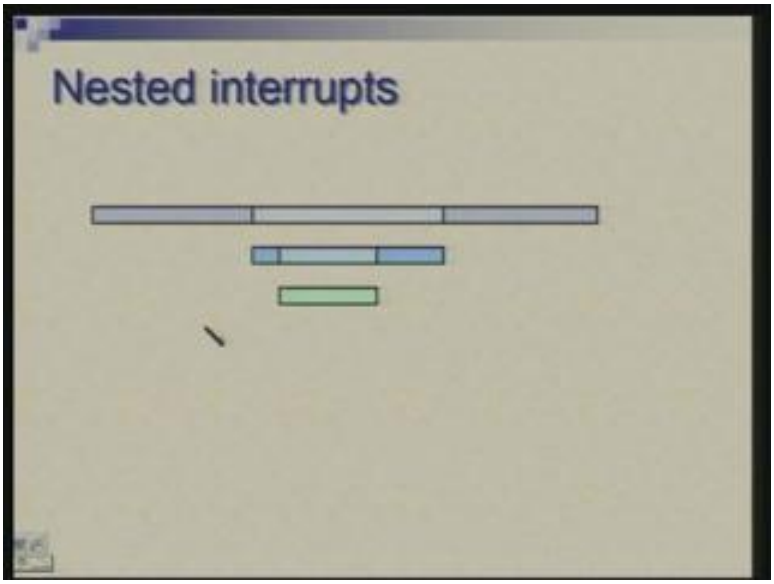
interrupt and halt, the other possibility is that after doing this what is called exception handling or interrupt handling you come back and resume what was being done.

So here (Refer Slide Time: 27:42) there could be multiple choices of what you need to do and it depends upon what was the cause of exception. So in this interrupt and resume situation I am showing two alternatives: one is called vectored interrupt where as you transfer the control you also identify what is the cause of exception and transfer control to one of the many possibilities. So, for example if it is overflow you go to one address, if it is illegal instruction you go to another address, if it is hardware malfunction you go to another address and so on. so directly in one shot you go to one of the many addresses and each address you have some piece of code which we call as interrupt service routine or interrupt handler or exception handler which takes the necessary action.

On the other hand other alternative is which is not vectored is that you transfer control to a specific address irrespective of what was the cause and then execute a few instructions here to determine the cause and then branch off have a multi-way branch and go to one of these.

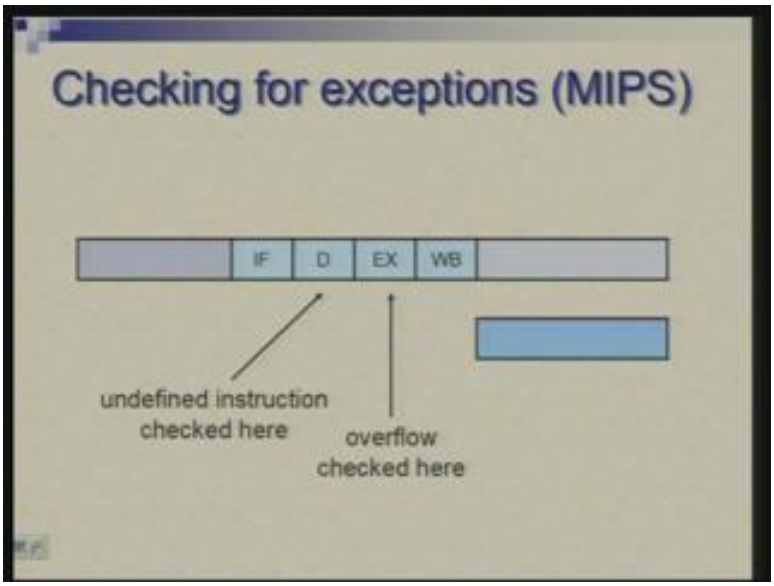
As you can see this there is some extra time spent here (Refer Slide Time: 29:03) by a few instructions to figure out what was the cause of exception and then you are branching off to take appropriate action whereas in this case there has to be a hardware mechanism so that the next address is influenced by the cause of the exception. So this is faster of course this takes more time but it needs more expense to the hardware.

(Refer Slide Time: 29:35)



Then complications can arise when there are multiple sources of exception. You may be servicing one exception and some other exception can arise. So even this gets interrupted you do something else, you do the third thing come back to this when you have finished come back to this so there could be nesting of interrupts or nesting of exceptions which can complicate the issues.

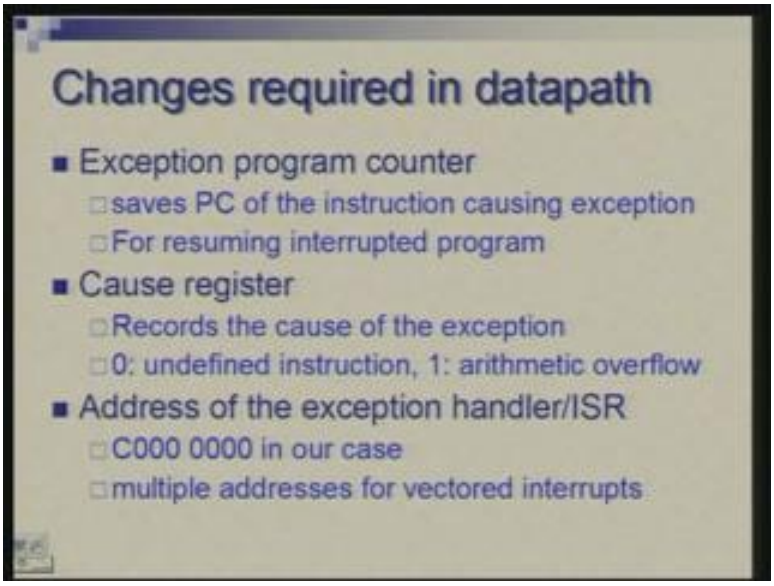
(Refer Slide Time: 30:04)



Now let us try to be specific to our design of those nine instructions which we have talked of and let us look at specific cases which can occur here. So we will focus our attention on these two exceptions which are shown here: exception resulting from undefined instruction and exception resulting from overflow. So here I am showing an instruction like add instruction which goes through four cycles. If you recall in the first cycle you are fetching the instruction updating the PC, the next cycle you are decoding the instruction you are looking at the opcode at the same time you are getting two values from the register file A and B, here you perform (Refer Slide Time: 30:52) actual arithmetic so I am calling it execute cycle and finally you write the result back into register file I am calling it write back cycle.

The instruction fetch decodes, executes and writes back so these are the four cycle instructions goes through. And it is in this cycle (Refer Slide Time: 31:09) that you will identify, you will come to know if the opcode is not the one of which you have catered for whereas it is in this cycle that you will know that overflow has occurred. So you can take action anywhere after this after the occurrence of after you have detected in any other following cycle you can take a suitable decision to branch off to some other point.

(Refer Slide Time: 31:50)



So now to do this what is it you need in the design. Suppose we have the intention of coming back to the program which is interrupted so you will have to save the values of the program counter somewhere. So this is something like subroutine like jal instruction where you need to transfer the PC to return address ra register and then you can do jr and come back. So we want to do something similar. One possibility could have been that we use dollar ra for doing this before transferring the control; you have the PC value in ra but what could happen is that this exception may occur at a point where you are interfering with normal procedure call.

So, for example, what you do is after jal instruction the control gets transferred and the first thing you do is save ra in the stack for example. It might take one or two or three instructions before you could do that and if exception occurs exception may sometime be unpredictable if exception occurs at that time where you have made jal you have made the transfer but you have not saved ra and if exception causes another transfer and PC value gets stored in ra, the old value will get lost. So for that purpose a different register can be provided. In MIPS it is called EPC or exception program counter. The purpose of this is to save the value of PC save PC of the instruction causing exception and then it can be used it can be read out that you want to resume the interrupted program so this is on part of it.

Actually in some processors where the call instruction that directly saves program counter on stack unlike our MIPS case where it is first saved in ra and then ra gets saved in stack and there are some processors where call instruction means program counter directly gets pushed on the stack. In those cases usually in case of exception also the program counter will get pushed on the stack and there is no register like EPC provider.

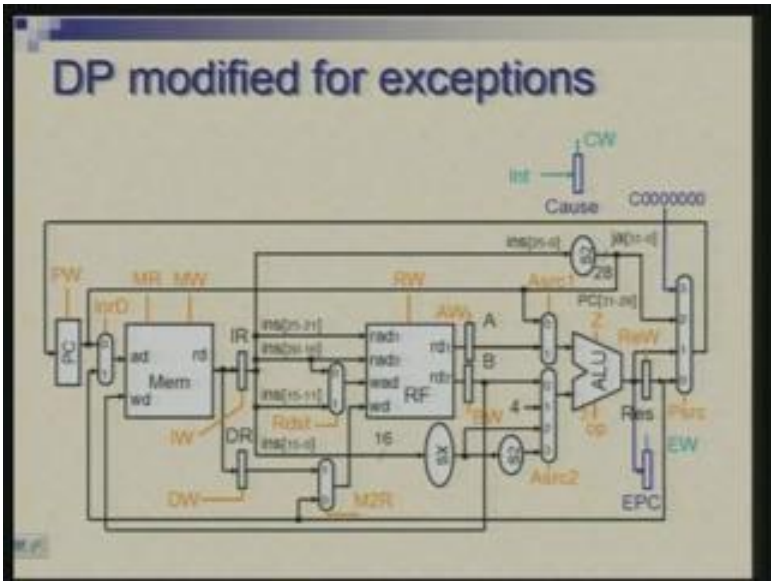
Second thing we need is what is called a cause register which will generally record the cause of exception. In our case there are two possible exceptions so it could be one bit register only but in general in real cases there are many causes of exception and the registers are bigger. So often it

may be a full-fledged 32-bit register you may not use all the 32 bits but several bits do get used. So let us code it as follows.

Value 0 in this would mean undefined instruction exception and 1 would mean arithmetic overflow exception. Third thing we need to arrange for is to which address we need to transfer the control to. So address of exceptions handler or interrupt service routine; so let us assume that this is the address (Refer Slide Time: 35:10) in hexadecimal where we need to transfer the control. So this is the value which needs to be loaded in PC when we detect that exception has occurred.

In case of vector interrupt you have an array of such addresses and you transfer control to one of those depending upon cause of exception. So these addresses could be typically uniformly spaced you say they are spaced by sixteen words then you may not necessarily be able to accommodate each exception routine within sixteen words what you can do is you can write some initial code and then branch off to some other place.

(Refer Slide Time: 36:02)



So here is the same good old datapath which we had designed multi cycle datapath and what it shows in different colors are the additions or alterations which are required to handle these exceptions. So let us look at this one by one. This (Refer Slide Time: 36:21) this is the EPC or exception program counter where we will store the address of instruction which has caused exception. Now by the time we detect exception PC would have incremented by 4. So if you want to get the original value back we need to do PC minus 4 again and it is this ALU which can be made to do that so PC and 4 are already available as operand to ALU by suitably controlling these multiplexors. All that you need to do is tell that this operation now is minus and output of ALU can be stored in EPC. So I assume that the control signal which will allow something to be loaded into this is ew or EPC write this is one part.

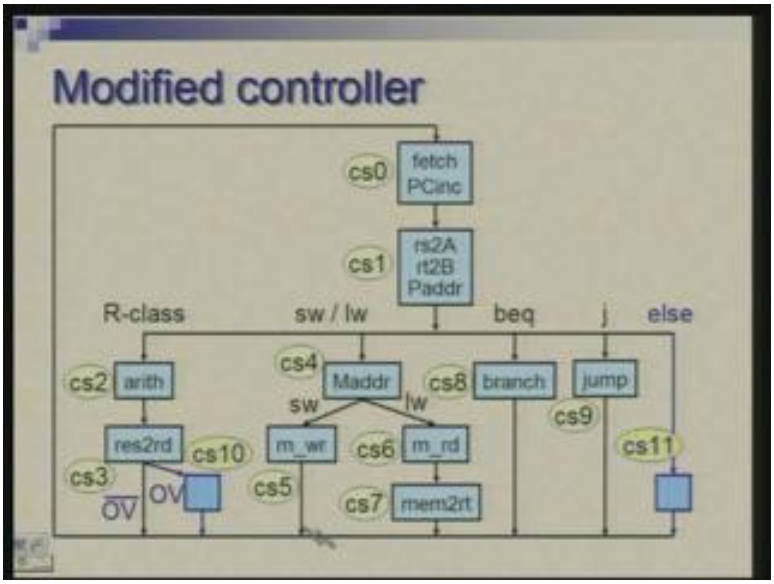
Second thing is the cause register which will store a value 0 or 1 depending upon cause of the exception. So it is simply a small register with a control signal which controls loading something into it and the value which gets loaded into it a 0 or 1. So we are assuming that this value (Refer Slide Time: 37:36) will also come from the controller, the controller will go into a state where exception has been detected and there it will generate a signal 0 or 1 depending upon to which exception has been detected.

And the third thing is an alternative source for the next PC value. So earlier we had a 3 input multiplexor here which now has been extended to 4 input multiplexor and the 4 input value is this address of the exception handler. So now with these three small alteration your datapath is ready to handle exceptions.

We will now go back to the controller design; basically look at the flowchart and see what are the stages at which you had these exceptions and there you will cause transition to new states where the new action will be taken.

Is there any question about this one?

(Refer Slide Time: 38:48)



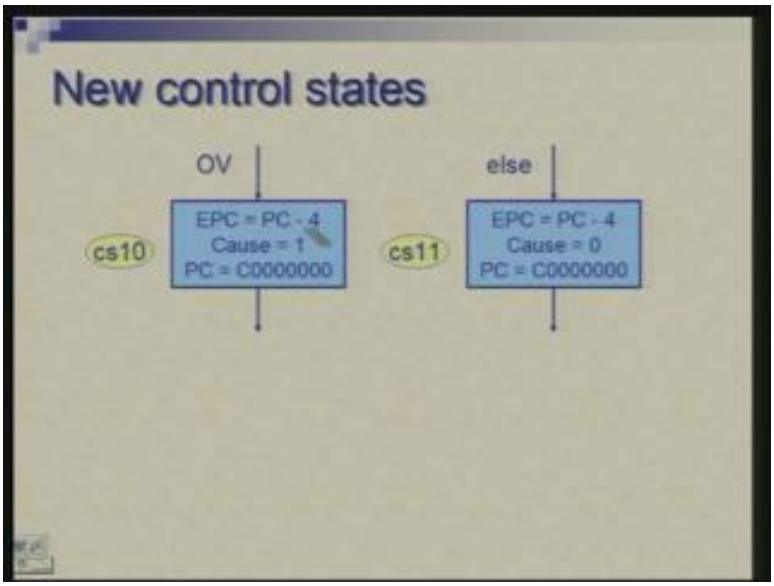
So this is the same controller diagram, we have made some change here and some change here. So in this stream of in this chain of states (Refer Slide Time: 38:59) suppose this is the state where you did the arithmetic where you performed addition subtraction and as a result you will either have a successful operation or you would have detected overflow. So after this actually you can branch off but we have actually postponed here, this choice actually depends upon whether you want to store the result in register file irrespective of whether there was overflow or not. So here the decision is that you will store the value you will store the result in register file even if there was an overflow. That may be necessary if the programmer wants to do something with that value and continue further but sometimes you may like the state of the register file to remain clean and not get any erroneous value in that case you must bifurcate at this point itself

after this state but I am branching off here so there is a here either you go through the normal path when there is no overflow or when there is overflow you go to another state which I am labeling as cs10 and I will show the action of this in a separate figure so this is a new addition.

Similarly at this point when you are branching off to R class load, store, beq, j if it is none of these you branch off to another state. So this is a state where I know that an illegal or undefined opcode has occurred and a suitable action can be taken here. So these are the two patches (Refer Slide Time: 40:47) or two small changes which are there in this control state transition diagram.

Now let us see what we do in this new states cs10 and cs11.

(Refer Slide Time: 41:02)

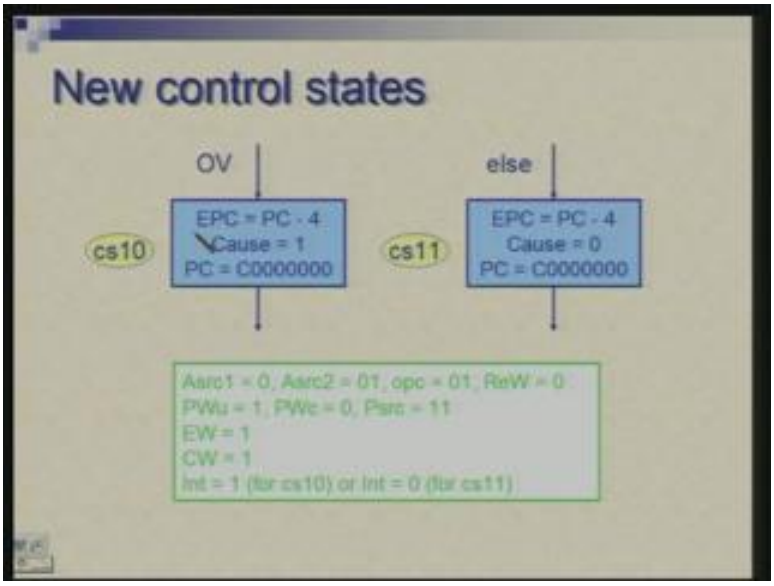


In cs10 which has to respond to arithmetic overflow what we do is EPC gets the value of PC minus 4 the cause is set to 1 and the new value PC is C followed by seven 0s. Cs11 is identical the only difference is that the cause register gets a different value.

In a vectored interrupt we will not have a cause register we may still have a cause register; different values will go to PC but here we are assigning the same value to PC and we will figure out later on by looking at the cause register; cause register will determine what was the cause of the exception and suitable action can be taken accordingly.

So now to achieve this what values we need to put to the control inputs that is shown here.

(Refer Slide Time: 41:58)

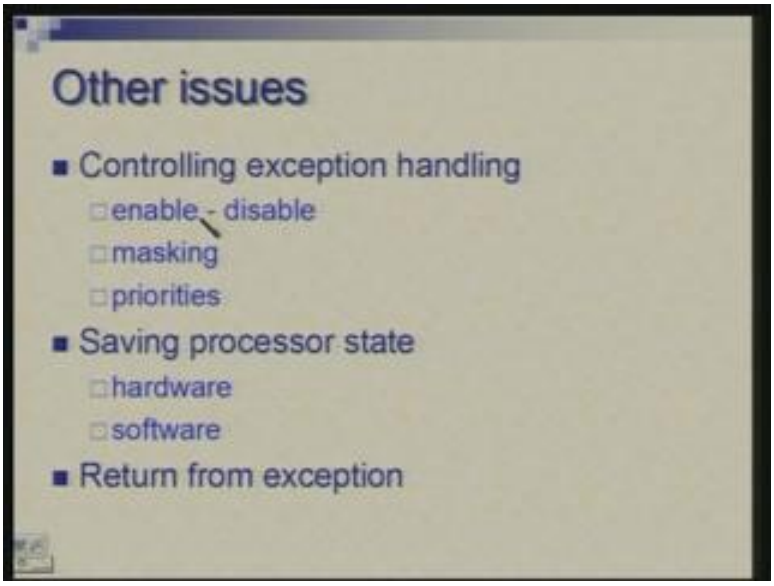


For making this happen EPC is getting PC minus 4 we need to give appropriate value to A source 1 and A source 2 so A source 1 equal to 0 will select PC as the first operand A source 2 equal to 01 selects 4 as the second operand and opc equal to 01 will result in subtraction being performed. The result we do not want to load into result register but on the other hand we will load it into exception register so its write signal we are making here as one.

Then we need to write something into PC so unconditional write signal for PC is 1, condition write is 0 well, actually it could be don't care and source is 11 which means the new input to the multiplexor will be selected and that value C 0 0 0 0 will get transferred to PC.

This is made one (Refer Slide Time; 43:03) for writing something to cause register; the value to be written into cause register is this signal int which will be 0 or 1, 1 for cs10 state and 0 for cs11 state both are identical except for the different values getting written here. So this was a very simple case but there may be many more issues. You may like certain amount of control of exception handling, you may like to for example disable or enable. There may be some critical portions in the program where you may not like to get distracted and handle exceptions and that is particularly true of exceptions coming from outside.

(Refer Slide Time: 43:55)

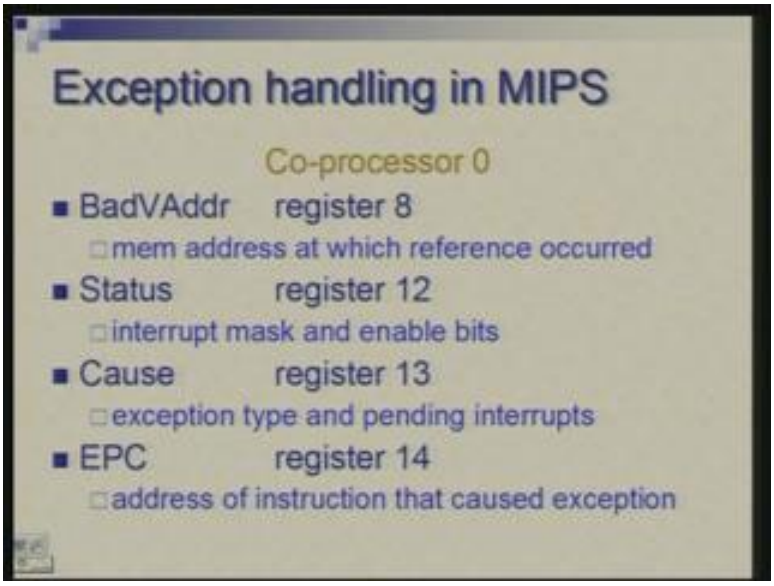


Internal exceptions like overflow etc or hardware fault you have to do something I mean you cannot skip over that. So enable disable is a mechanism to enable or disable the interrupt or exception system as a whole but it is also possible to mask or unmask specific interrupts. You may want to say that I want to look at this, this, this interrupt but I want to ignore for the moment these, these, these interrupts. Also, you can define priorities for various events and you may handle them according to your priority.

Then we talked about saving of only the program counter so that we can return back but there may be often a need to save other registers because in the interrupt handling you may have the need to use other registers which may be containing the values of partial computation which is going on. So, in some processors it is the responsibility of the software to do saving of those registers, in some case hardware does it for a fast reaction.

And then how do you return from exception?
It may not be simply a matter of transferring control, you may have to restore if something got saved. So let us have a quick look at the exception handling in MIPS processor on the whole.

(Refer Slide Time: 45:30)

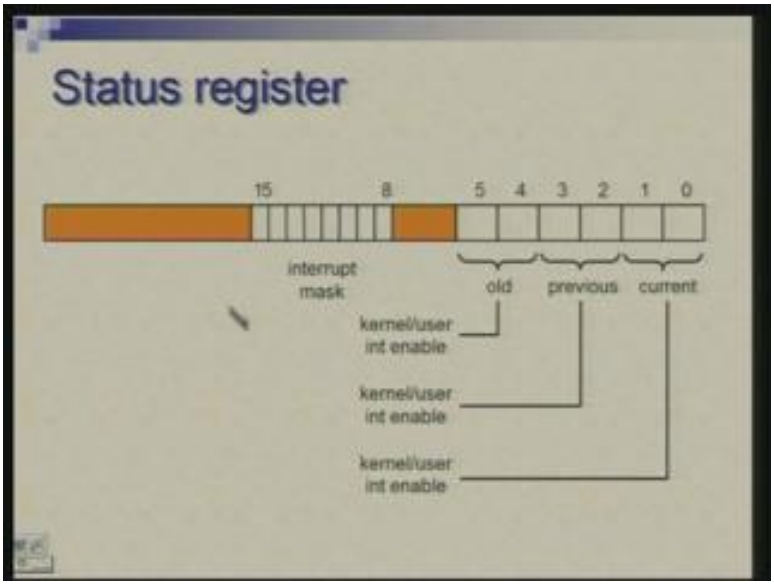


So in MIPS the portion of hardware which handles exception is treated as a co-processor. You remember that when we talked of floating point I mentioned that it is called co-processor 1. So co-processor 0 is actually exception handler and it has its own set of registers which are again numbered 0 1 2 3 4 and there are specific names and specific purposes of these registers these are not general purpose registers. Some of these I have listed here.

Register 8 12 13 and 14; 13 and 14 are what we have already talked of. 13 is Cause register which actually carries the information about what was the exception type, which event cause exception and if there are pending interrupts, you may have several exceptions several interrupts coming in here you may be handling only one, others may remain pending. EPC contains address of instruction that cause exception.

Apart from this you have a status register which carries interrupt mask and enable bits. So these are the controls for enabling or disabling the overall interrupt system or specific events. This register will carry the memory address when you, suppose you made the memory reference and some exception occurs there so that address is actually contained in this register.

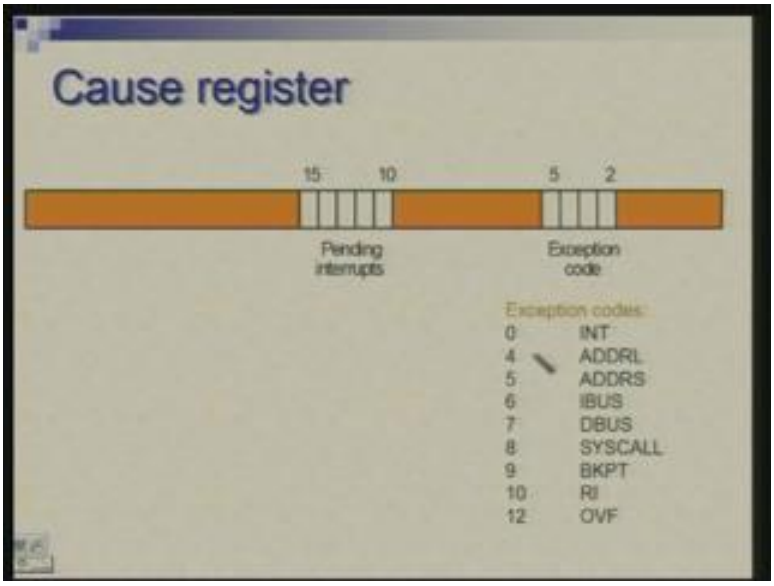
(Refer Slide Time: 47:00)



Some details of a status register: There is an a few bits of these are considered as an interrupt mask so here one bit is for one different type of interrupt so bit number 8 to 15 that means eight specific interrupts can be individually masked or unmasked and then overall enable disable is stored here (Refer Slide Time: 47:27). As you would see there are three sets of values: the current value, the previous value and one before that and each of these is two bits; one bit determines whether you are in kernel mode, or privilege mode or user mode or the normal mode and at the same time the other bit is the enable bit.

So now these six bits actually form a stack of depth three so you are storing the current enable disable value and current mode whether it is kernel or user and similarly you have one for the previous state and one before that. So as exception occurs this stack gets pushed down, when you return from the stack you know whether you were in kernel mode or in user mode and whether in that interrupt enables or disables so all that information is carried in this status register.

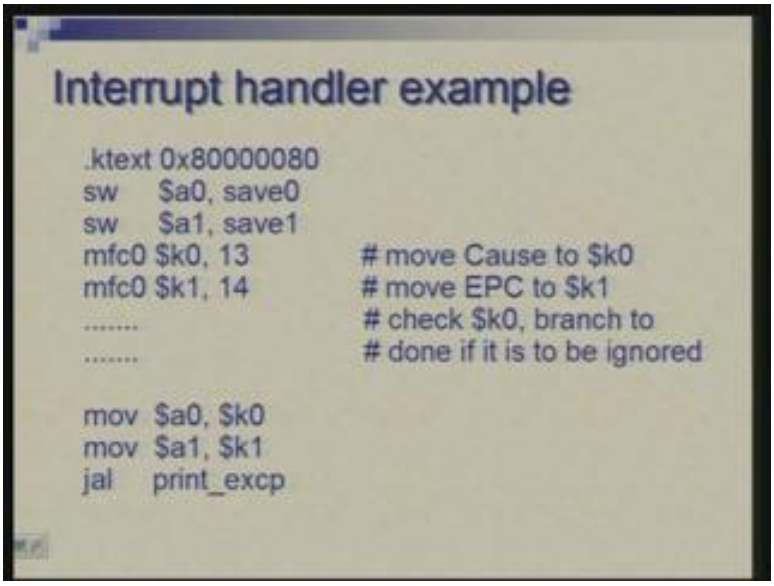
(Refer Slide Time: 48:35)



The cause register has exception code which indicates which kind of exception it is so there are various causes I will not go into the details of all of those, let me refer to some of those which I have already mentioned; this is overflow (Refer Slide Time: 48:50), this refers to undefined instruction this point, syscall is for OS service, INT is external interrupt and these are related to memory errors and here there is information about pending interrupts.

So as you would notice that the overall interrupt system could be really complex and all the operating interrupt routine are part of the operating system, it is not typically the user who writes the routines so they have to very carefully define responses to various events and they all form part of an operating system kernel.

(Refer Slide Time: 49:37)

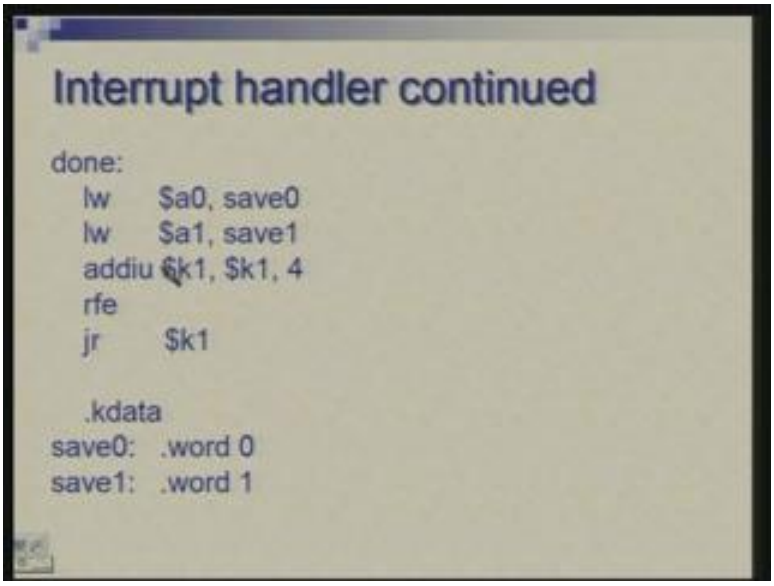


So here is a small example of interrupt handler. I will not show full code there are just some usual actions which are there. First thing which is happening here is that register a0 a1 are being saved because they are required inside and then two registers k0 and k1 are one of those thirty two main CPU registers which were at some point I must have mentioned that these are reserved for the operating system. So this is where they get used.

Look at these two instructions: move from co-processor zero. So from register 13 which is the cause register of co-processor 0 you get information into k0 similarly 14 which is EPC you get in k1. Then you may do something about this (Refer Slide Time: 50:30) so k0 is causing the k0 is containing the cause register so here you may decide which exception you want to handle in which particular way and if there is something which you do not want to handle you can decide that in branch off you can skip remaining processing.

Here (Refer Slide Time: 50:51) what we are trying to do is calling a print routine which will print the value of these; print the value in k0 and k1 which are cause and the EPC. Then when you are done you will restore values of a0 a1 from these memory locations, you were containing the addresses of instruction that caused exception and now you want to resume from the next instruction onwards so you add 4 to that and do jr .

(Refer Slide Time: 51:20)



There is another instruction new instruction which you see here which is rfe standing for return from exception. So this return from exception will take care of that stack of..... those are the three level stack of two bits so when you get into an exception handling routine you are pushing down that stack when you come out you pop up that stack. And here is the way these two memory locations are defined (Refer Slide Time: 51:56). I think this has to be 0, the idea was to initialize these two to 0.

So now what the important thing to be noticed here is that the information available to you here in our simple case was that the cause register and EPC. EPC allows you to go back and cause register tells you what is to be done. So in most simple case this is what you need. And here (Refer Slide Time : 52:36) you might wonder that we are trying to undo something which we apparently did extra. We already had PC plus 4, for saving we went back to PC and now we want to get PC plus 4 again.

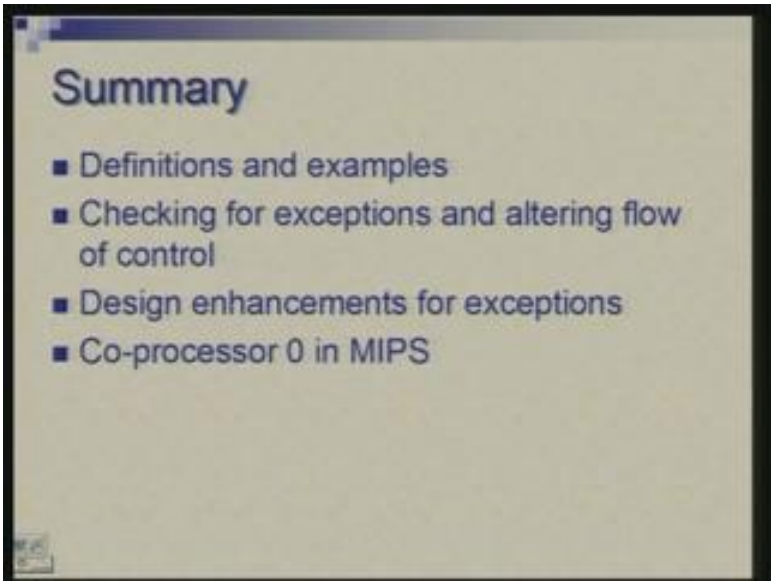
Now the reason why we save address of the instruction which cause exception and not the one which is following is that in the exception handling routine sometime you may like to analyze what the instruction was and therefore you want a quick access to that address. So, for example, if it was undefined opcode you want to see what opcode was so that suitable message can be printed or action could be taken. So therefore we did PC minus 4 that time so that we have EPC contains the address of instruction which caused exception.

But if you are resuming, in this case we wanted to resume from the next instruction. There may also be cases when you resume from the instruction which cause the exception. So, for example, suppose it will happen in case of virtual memory, so suppose you wanted to access some memory location which is not actually present information has to be brought from disk so in such a case exception will occur and that instruction will not get completed. So you have to come back and complete that instruction suppose if it was a load instruction you are trying to load some word but that word has been swapped out it; is in disk but not in main memory so the

instruction does not complete but your intention is to complete it. So you handle the exception which will bring the data into memory you will come back to the same instruction complete it and then proceed further.

So, if you are handling exception like that you will not do this (Refer Slide Time: 54:34) you will simply return to the same instruction where exception has caused.

(Refer Slide Time: 55:05)



I will close with that and just to recapitulate what we did we looked at the meaning of exception and the term interrupt, how they can be defined and compared, looked at various conditions which can cause exception and interrupt, we looked at the mechanism of checking the exception and altering the flow of control. We took our simple data path design multi cycle design and caused alterations or additions to handle exception and finally we had a brief look at the overall interrupt mechanism in MIPS processor. Thank you.