Computer Organization
Part – III
Prof. S. Raman
Department of Computer Science & Engineering
Indian Institute of Technology, Madras
Input/output
Lecture – 27
Device Service Routines

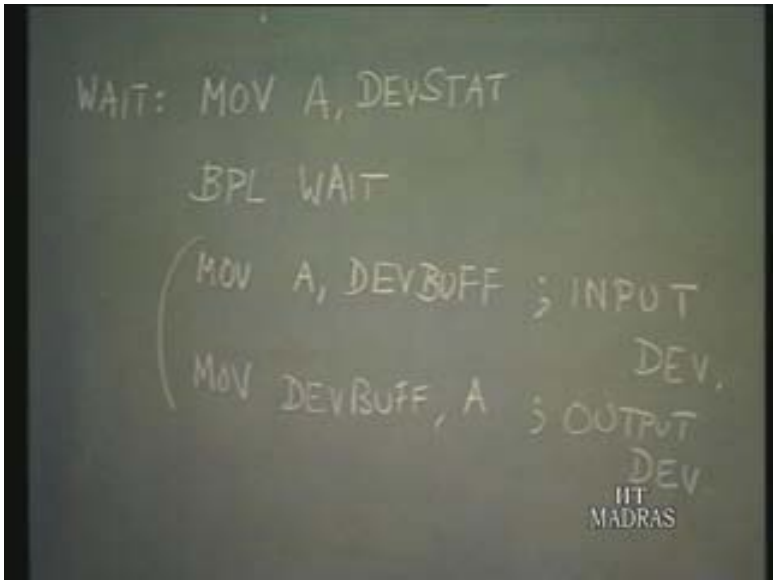
In the previous lecture, we took a look at the concept of memory mapped I/O and also started the discussion of device service or device service routine, specifically with reference to programmed I/O.

(Refer Slide Time: 01:19)

Now what is this device service routine? After all it is the CPU which is involved in executing a program. Now whenever the device needs attention, either the CPU will look and see whether the device needs attention or the device we said will interrupt the CPU and demand attention. In either case, the CPU must stop whatever program it is currently executing and take up a program or what we call as a routine essentially to service that particular device which needs attention. Now in the case of programmed I/O, we saw that the CPU, which takes the initiative, checks for the status of the device and decides whether the device is ready or not. If the device is not ready, it waits in this particular loop – I mean as written in this particular program it just has to wait; we can also rewrite the program so that if one particular device is not ready then the CPU goes and checks another device. It just depends on how exactly you write the program; that is why it is called the programmed I/O. Now in this particular one, it just waits indefinitely and then when the device is ready it is indicated by 1 bit; that is a ready bit in the status word; we have discussed that in detail.

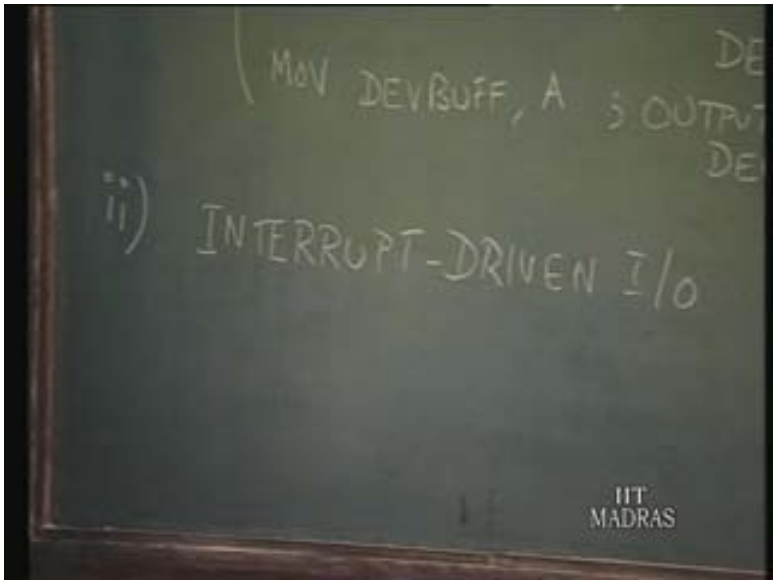
Now when it is ready, it moves out of this loop and then moves on to service that particular device, in which case it just has to effect the data transfer. Since we had assumed an input device, what we are saying is that we are moving the data which is available in the data buffer of the device to the accumulator, which means, to the CPU. So in other words, it is the input. On the other hand if you happen to have an output device, it is just the other way but this is for an input device; if it were an output device instead of this, what we will be having is just the other way because of the way we have assumed the format of the instruction. We had assumed also the memory mapped I/O scheme: just try to recall; instead of this we would be having just the other way, that is, move whatever we have in the CPU that is accumulator of the CPU – this is the source this is the destination you remember the move destination source is the format we had followed earlier – so move what is there in the CPU to the device buffer, which means outputting. So this is what we will be having for output device; so this in fact is the essence of an I/O program.

(Refer Slide Time: 04:22)



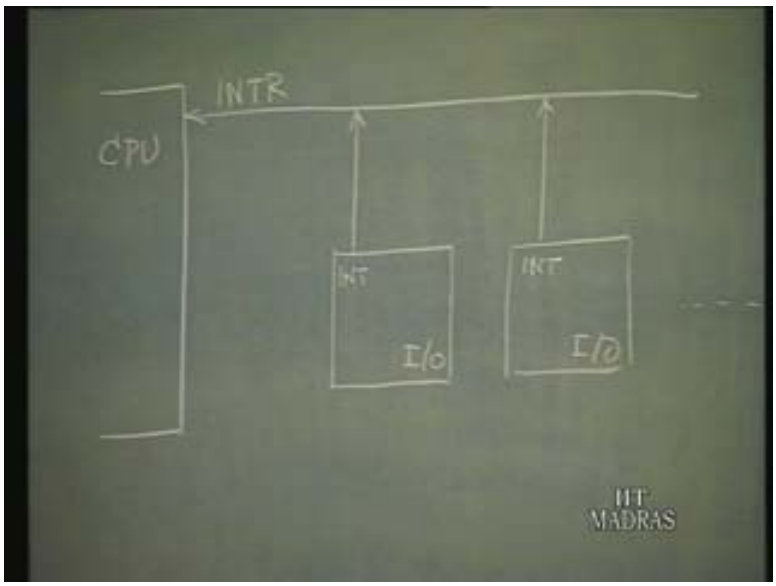
An I/O program would mean granting that the device is ready, making the data transfer. Now in programmed I/O, the CPU checks; we will see this is all that is there in data transfer. Of course it all depends on the particular size of the buffer; for simplicity's sake we have assumed a buffer of only 1 character or byte, let us say, assuming the accumulator is also a character or byte size. So we are just moving 1 byte. Suppose the device happens to generate a lot of data, then we have to put a counter. Suppose it generates 10 bytes of data, then we have to set up a counter and then assuming this data transfer is byte by byte, we have to keep counting and then checking. After 10-byte transfer is over, this whole I/O program is over. This is how we have to do; it is part of the programming – I will skip that detail. Now let us move on from this programmed I/O to the next scheme. The next scheme is one in which the CPU is not taking the initiative; it is up to the I/O to interrupt the CPU whenever it is ready. We were calling this interrupt driven I/O scheme.

(Refer Slide Time: 06:13)



As per this mode of transfer, since CPU need not check, we go back to this previous program. You see this part is only for checking whether the device is ready or not; so that is not there now. It is only the data transfer that may be affected, depending on input or output – the data transfer must be effective. I may not be showing all the details of the bus here; I will show the necessary connections between the CPU; we will also not bother about the memory in this particular case.

(Refer Slide Time: 07:44)



Suppose we have two or three I/O interfaces that is, one for each device, that is good enough. So now the I/O is going to interrupt; the interrupt input is there to the CPU.

When the I/O is ready, it is putting up the interrupt signal. So the CPU is not going to check whether the device is ready; it is only this part that is needed. That is, the device is ready when its data buffer is full in the case of input device. In the case of output device the data buffer will be empty; that is all the difference between these. So we will continue a discussion for input device and then I will just tell you whenever we have talk about output device also.

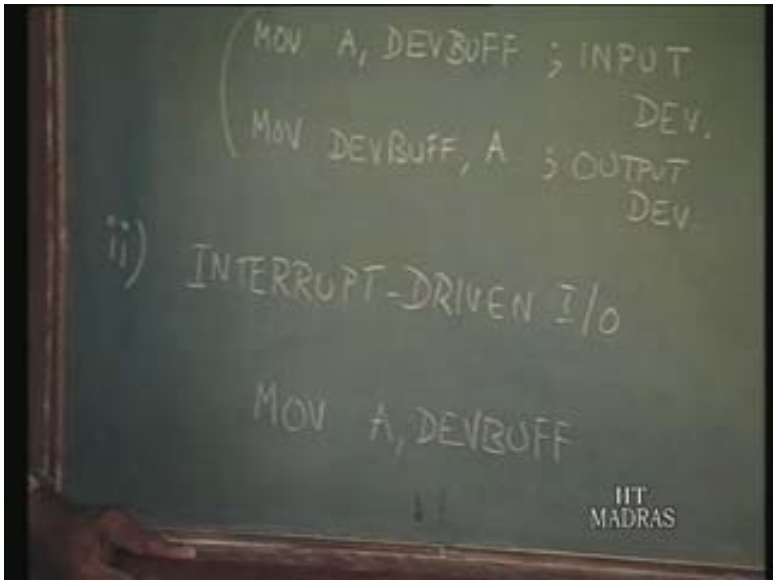
So when the input device data buffer is ready, it is going to interrupt. The CPU is going to stop whatever it has been doing and it has to turn its attention to this service part, device service. So we said earlier in the case of interrupt driven I/O, there is a context switch; that is, a switch from the main program, main routine to device service routine. Because it is interrupt, this particular one may also be called interrupt service routine, interrupt routine, and CPU handles these situations; so it is also called the interrupt handler – these are all different names for the same thing. So let me list these: device service routine or DSR; interrupt service routine or interrupt handler and so on and so forth; interrupt handler.

(Refer Slide Time: 09:50)

Now what is it essentially? Essentially this particular one is going to service the device and the service is just seeing that the data transfer takes place. That is, the device status checking is not there in this case; only because the device is ready it has interrupted the processor. If that is so what is there in the program? The program will only consist of the interrupt service program or the interrupt service routine will essentially consist of the data transfer part and anything else associated with it as I said earlier, counting of how many bytes and so on and so forth. That is, suppose 10 bytes have to be transferred as part of the interrupt service, then the CPU must take up this data transfer 10 times; it must carry out the data transfer assuming only 1 byte is transferred at a time. So the counting part and all those things are there.

That will also become part of the interrupt service. so other than that, you can now see that the service part is nothing but moving these; so if you assume that we are dealing with an input device, the interrupt service routine essentially consists of this particular statement.

(Refer Slide Time: 11:24)

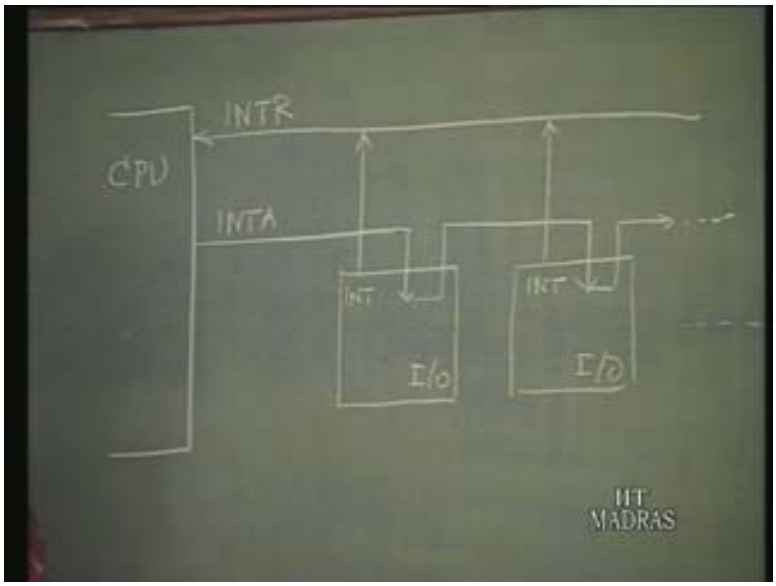


That is, input device – moving whatever is there in the device data buffer to CPU, of course, not over this line but over the data bus, the data part of the bus, which is not shown in this figure as such. So what has been shown here is only when the device, rather how the device, interrupts the CPU – that is all that is shown in this particular figure. The data bus will be connected to this, so that will be moved and the appropriate read write control signal will also be there. We will not bother about it; hope you follow that without any difficulty.

The interrupt service or the interrupt routine is sometimes called just an interrupt program. It is nothing but the CPU executing a main routine and now the CPU executing something like a subroutine, which is now interrupt routine. So there is a context switch. Some very interesting things arise in this particular situation. First of all when you have many devices how will the CPU know which is the device which must be serviced? This is number one because if you have say 10 devices, you cannot have 10 interrupt inputs to the CPU. Especially in the case of microprocessor situation, the number of pins limits that. So you may have only 1 or 2 or may be 3, say maximum 5; we do not really have more than between 3 and 5. In the case of even 1, it should be possible. In the case there is only 1 interrupt input, CPU should still be able to identify which is the device which must be serviced because this may be a key board; and this may be say light pen input. So the kind of service needed for the keyboard is different from the kind that is needed for the light pen. So the programs are different, leave alone the other electrical interface, leave alone that program itself will be different. Now assuming that the devices there can be any number, all put a request on a single interrupt input.

How will the CPU know what it is to do? We already discussed this earlier. First of all, the CPU will not allow itself to be interrupted till the end of that particular instruction cycle; that we know. Now at the end of that instruction cycle, it is going to generate the interrupt acknowledge. This particular signal comes at the end of that. Now one scheme can be that the interrupt acknowledge signal goes to one device and then to the other device and so on.

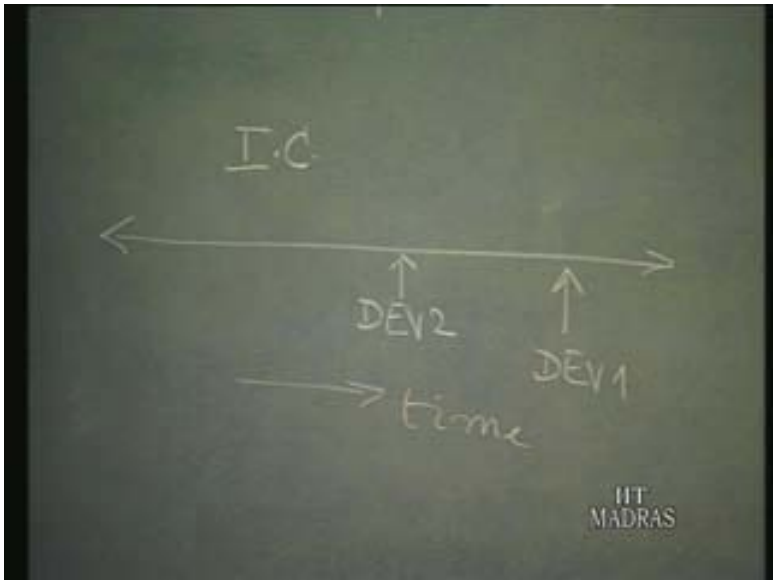
(Refer Slide Time: 14:59)



Now shown in this way, what does it mean? You may be having n devices and any one of these n devices may generate an interrupt signal. At the end of that instruction cycle, when the interrupt acknowledge signal is generated, this is the device which is going to get the interrupt acknowledge signal first. Suppose this is ready, we do not have to know when exactly this became ready. When the interrupt acknowledge signal comes, if this is found to be ready, this will respond and what is the response? It has to essentially identify itself; that is, it has to generate a code to the CPU, that is, we call that a vector, a code. The code essentially points to where its interrupt service program is. So let us say for the first device, interrupt service program is starting from location 100; for the second device it starts from location 200 and so on for third device from location 300. Essentially this device must say my service program starts from location 100. So it has to send a code, which the CPU will make use of and link it to start address 100 and this second device will have to generate a code, and then, the CPU, will understand from that that it will have to start the service program from 200 and so on and so forth.

For instance the simplest thing is this device itself can put that address 100, which can go; this device can itself put 200 in response to the interrupt acknowledge. See, after all, an instruction cycle consists of many states. Now it is possible that we will just say this is one instruction cycle; it is possible because, after all, this is time axis. It is possible that device 1 or device 2 generated interrupt at this point in the instruction cycle and for device 1, we have just considered two devices.

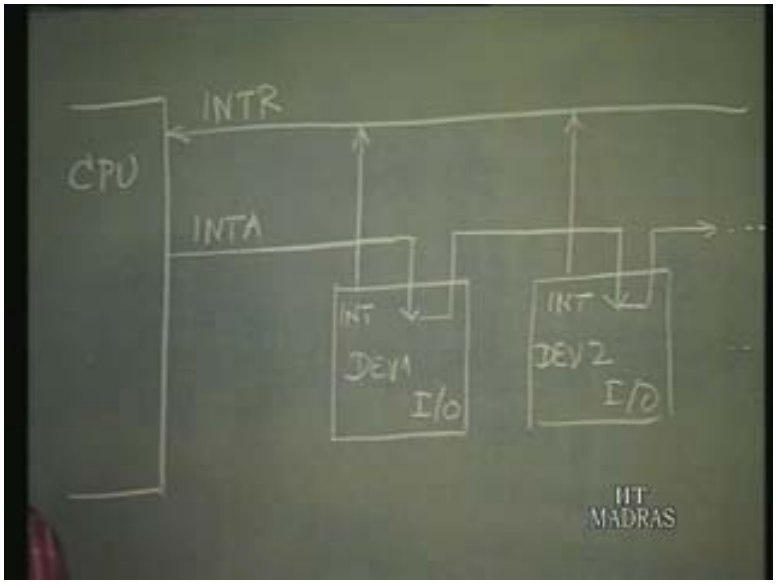
(Refer Slide Time: 17:45)



So we will continue and device 1 has generated an interrupt at this instant. Now this is an interrupt time, at which device 2 generated the interrupt, and this is the time in which device 1 generated the interrupt. So in either case, both had generated the interrupt; the interrupt input has come to the CPU and CPU responds with an interrupt acknowledge only at the end of the instruction cycle. Now by the time we reach the end of the instruction cycle, we see that device 1 is ready; that is why it has interrupted.

And so now you can see even though device 2 has generated an interrupt earlier, when the CPU responds, it sees device 1 as ready. I just said device 1; I think you all followed – this is device 1 and this device 2. Now both are ready and since CPU sees that this is ready, this I/O device will generate over the data bus; actually it will send the code and identifies itself as its service program is from such and such and the service program is all essentially moving the data. So the service program for this, starting from 100, will be executed at the end of it.

(Refer Slide Time: 19:03)



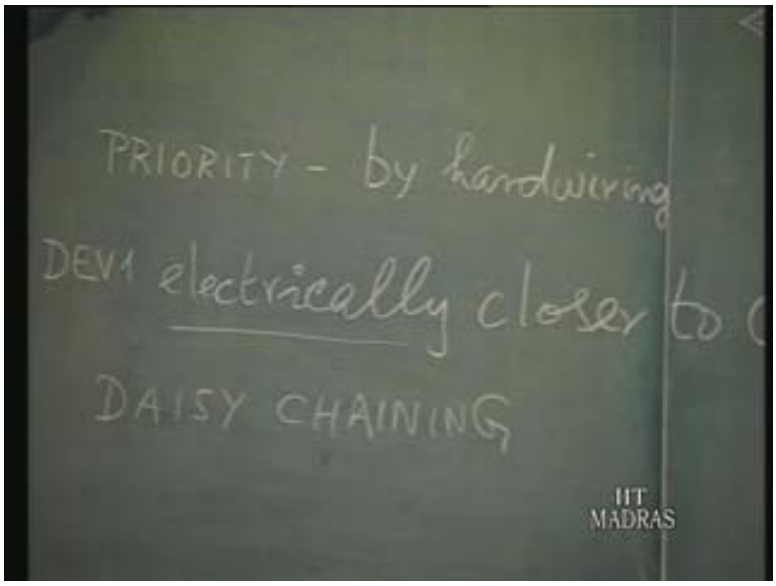
At the end of the service, device 2 will still be waiting because that is also ready and it not been served. So at the end of that device 2 will be served. Now this is one simple way of looking at it. We will not go into certain issues in this I said at the end of device 1 service there is device 2 and I also said it is not really necessary. We will talk about it; we will come to that a little later.

Why is it that even though device 2 generated the interrupt signal earlier than device 1, device 1 was serviced first? It is mainly because this is the route by which the interrupt acknowledge signal goes. In other words, this is nothing but hard wiring of the priority of the devices. This is nothing but defining the priority of this device by hardwiring because it has been hardwired this way.

The acknowledge first goes to this point; electrically this goes. So we also say that device 1 is electrically closer; because of the hardwiring device 1 is electrically closer to CPU. I hope you understand why we say electrically closer. It is because device 1 may be physically farthest. For instance, if the interrupt acknowledge first goes to device 2 and then it comes to device 1, then we would say device 2 is electrically closer, though physically it may be further away. So the priority of device 1 service is high mainly because we have achieved it by hardwiring the acknowledgement.

Now by this hard wiring, we see that device 1 is electrically closer to CPU and that is the reason device 1 was serviced first. We have already seen what the service is; service is essentially data buffer, moving the contents to the data buffer. Now this particular arrangement is also called a DAISY chaining; so it is something like chaining all the devices.

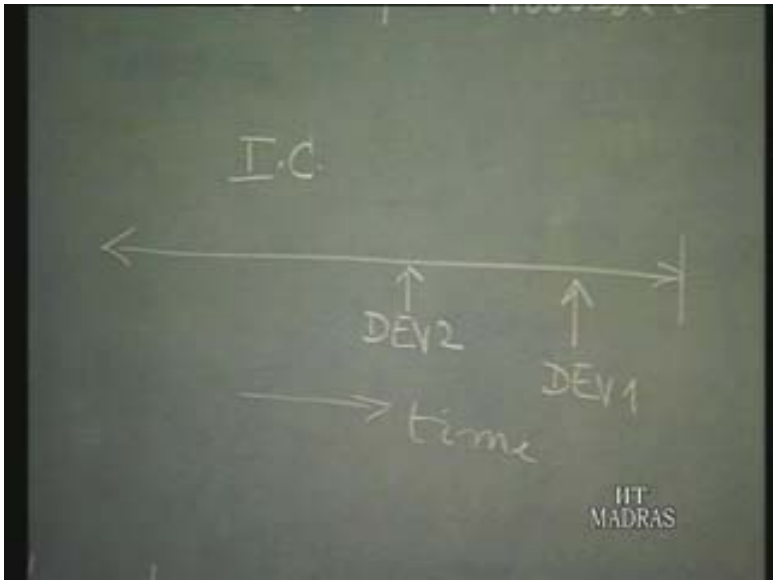
(Refer Slide Time: 22:28)



So take the signal through one device and then through another device and through another device, creating a chain; this particular thing is also called a DAISY chaining. So this is one way in which the priority can be defined. Now are there other ways of defining the priority? Obviously the moment we say through hardwiring, we define the priority; that means there must be another one by which, through software, we may be able to define the priority.

How is it? Now let us go back to the original situation itself. Device 2 generated interrupt and then device 1 generated interrupt; at the end of the instruction cycle when the interrupt acknowledge goes to device I mean is generated by the CPU, it goes to device 1 first and so device 1 service is taken up. We had assumed a simple one-line interrupt service program; now assuming that this consists of say some 5 instructions, suppose the device service routine is the interrupt service routine specifically in this case we can call it as interrupt service routine. Suppose this consists of 5 instructions.

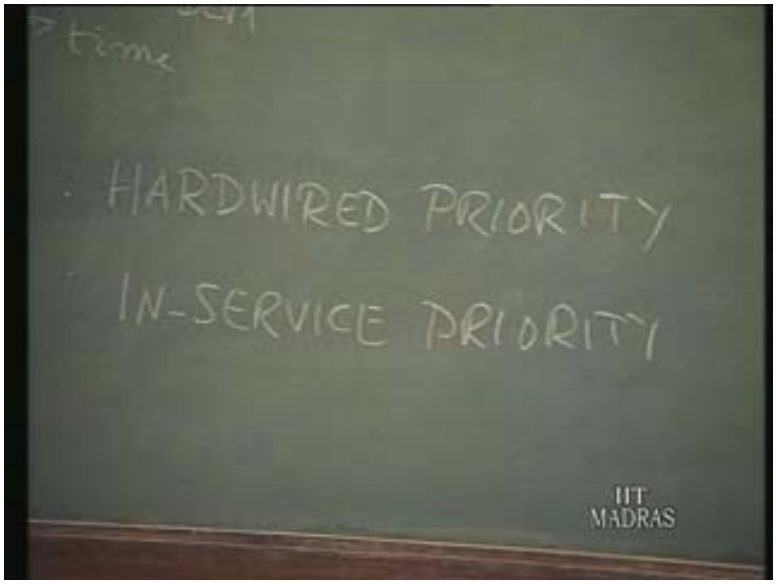
(Refer Slide Time: 23:28)



That means essentially 5 instruction cycles will be gone through. Now what did we say earlier? We said that whenever there is an interrupt at the end of the instruction cycle, the interrupt acknowledge will be generated. Now device 1 service we say consists of 5 instructions or 5 instruction cycles. Now the moment the first instruction cycle of the interrupt service for device 1 is taken up, if this interrupt is still there, what will happen? After all from CPU's point of view, it will respond whenever there is an interrupt, and in this particular case device 1's request had already been responded and it is servicing, that is, it is executing a program. Whether it is a main program or an interrupt service program, it is just a program and the interrupt acknowledge has gone to device 1 and the service program has been started. So it will be meaningful to assume that this device itself cannot interrupt itself again.

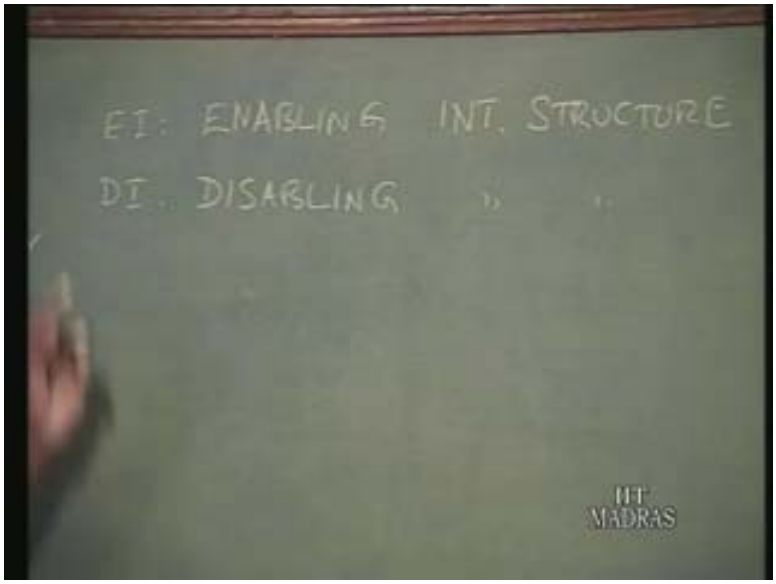
So this interrupt say input would have been cleared. Now this interrupt signal will not be there and CPU will see this interrupt signal. The device 1 service has started, and it is going through the first instruction cycle, and at that instant, it is meaningful to assume that this interrupt input will not be there. Now this interrupt input will be there. CPU, after it finishes 1 instruction cycle for device 1 service, will be interrupted because device 2 shows the interrupt input, that is, it has flagged the interrupt input. What happens? The service of the device has been interrupted; mainly it is just a program. As long as the instruction cycle is completed, it can be interrupted. So that can be interrupted and with device 2 – what happens to our earlier priority definition? We now have to talk about two things: one is what is the priority before the interrupt and what is the priority while interrupt program is not there, that is, as hardwired priority that is as in this and then what is the priority in service. So we have to talk about the two priorities, hardwired priorities. Of course, there is a way in which this problem can be circumvented; anyway let us make a note of this also.

(Refer Slide Time: 27:41)



In service priority, it is already a device which is being serviced. What is its priority? Now if you want device 1's service to be completed, what did we assume? We assumed that device 1's service involves 5 instructions and so 5 instruction cycles. Now throughout these 5 instruction cycles, if the interrupt itself is disabled, then device 1 service can be completed. That would be needed some times. So we have to also talk about enabling or disabling the interrupt structure.

(Refer Slide Time: 29:08)



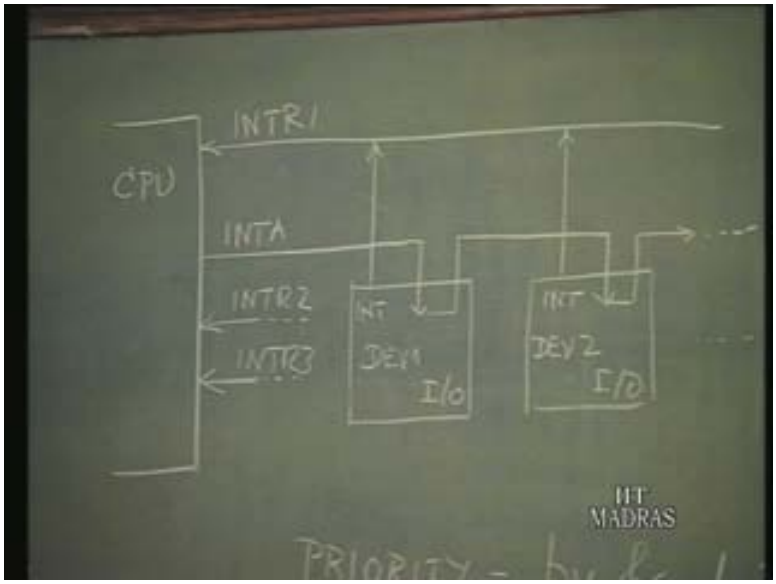
That is, the CPU should not respond; I mean the CPU enables to respond and in this case of disabling interrupt structure, the CPU is disabled from responding to the interrupt.

Actually there are instructions called enable interrupt and disable interrupt. By executing these enable interrupt and disable interrupt instructions, the structure can be enabled to respond to an interrupt input or disabled. Now let us go back to our original problem: device 2 and device 1 both interrupt at the end of an instruction cycle. The interrupt acknowledge is generated and device 1 interrupt service is taken up. The very first instructions in device 1, if it were disable interrupt, then what will happen? No more interrupt will be accepted by the CPU and the very last instruction of device 1 interrupt service can be enable interrupt, in which case that instruction will be completed and then only at the end of that instruction cycle only, the interrupt input will be honored. In this case we can complete the device 1 interrupt service and then respond to it.

So enabling interrupt structure and disabling interrupt structure in the CPU is done by having the instruction; it is part of the instruction set of the CPU. By including these in the service program in the service routine, we can selectively enable and disable and then keep redefining or rather, in this particular case, maintain the priority. Now the same thing holds good. Suppose device 1 and device 2 generate interrupt at different times. Let us say only device 2 has interrupted and, to start with, device 1 has not interrupted; then device 2 service will be taken up, and if the interrupt has not been disabled, then at any time when device 1 interrupts, device 2 service will be interrupted. Device 1 will be taken up mainly because of the hardwired priority. So the interrupt acknowledge will be generated. Now even though by hard wiring we have assumed that device 1 interrupt will be taken up first, suppose during device 2 service the interrupt structure has been disabled, by executing a disable interrupt instruction as part of the device 2 service program, even though device 1 has a higher priority because the structure is disabled, during service we can keep redefining the priority and there are many other ways also by having a separate priority arbitrator and so on and so forth.

That is, through software, we can keep redefining the priority. That is why we say we have a hardwired priority and then we have in-service priority and this particular thing can be changed. What can be changed is what can be done through software. The one which is hardwired can never be changed. Now there are many ways – we assumed only one interrupt input. Suppose we have say two or three interrupt inputs. Then, given an interrupt line, we can talk about the priority and given these interrupts, say we call these interrupt 1, interrupt 2, and interrupt 3, suppose there are 3, we would be connecting the devices to these.

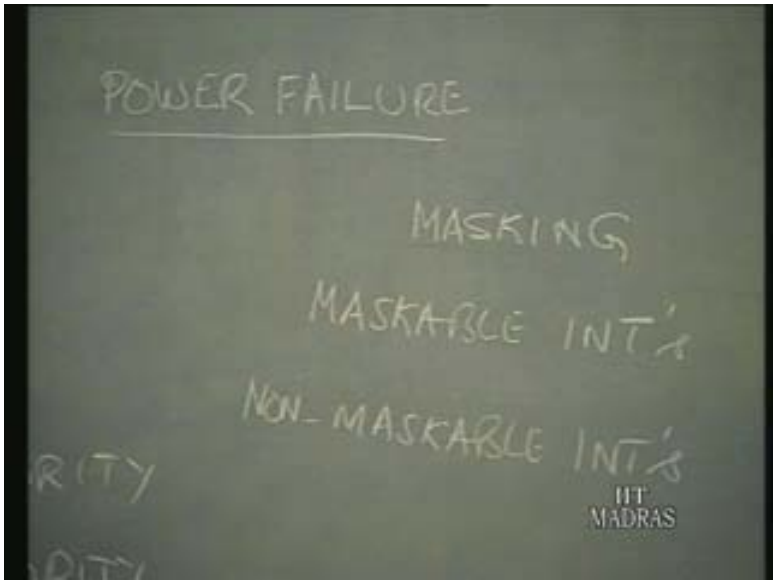
(Refer Slide Time: 33:49)



Now we can talk about a two-dimensional priority structure. Generally what happens is the CPU itself will define the priority among these interrupt 1, 2 and 3. The CPU will define a priority and then, by this external arrangement, on the given line by hardwiring and DAISY chaining and what not, we can redefine the priority. So you have a two-dimensional priority. That also will be taken into account during the service. Earlier I had also said that it is meaningful not to disable some aspect of the interrupt structure. There may be some crucial inputs, which you should never disable; for instance in case power failure is indicated. Suppose you are able to disable the interrupt, then before we can re-enable, the power may fail and we would lose everything that is in the semiconductor memory, volatile memory.

So generally what happens is certain inputs will be accorded the highest priority; not only highest priority but also that it will not be masked; you cannot disable them. The term that is used for that is by disable or enable, we can selectively enable; that is, we will selectively enable certain inputs and we will selectively disable certain inputs. So that particular thing by selection is called masking. We talk about mask able interrupts and non-mask able interrupts. For instance, crucial inputs such as power failure will all come under non-mask able interrupts.

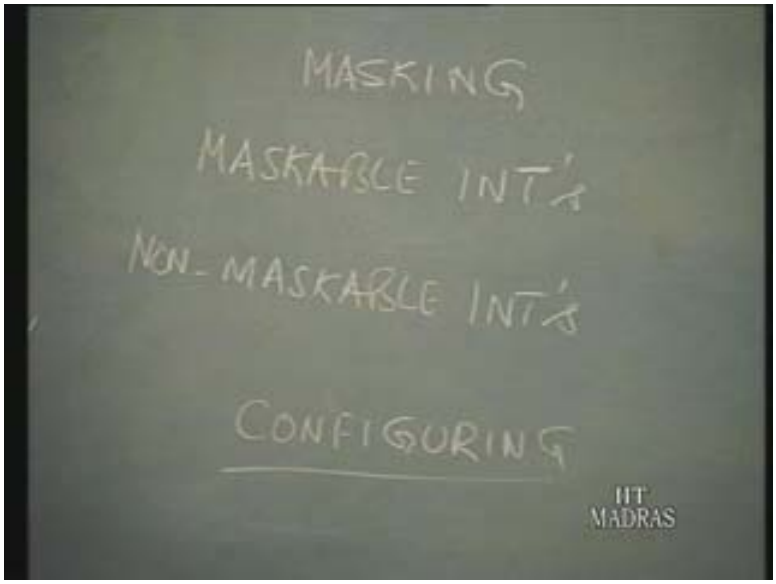
(Refer Slide Time: 36:33)



It means, suppose interrupt 1, interrupt 2 and interrupt 3 have a priority such that interrupt 1 is the highest priority. Generally the highest one will be given the non-mask able character so that it will never be disabled. Suppose we have three different priorities like this. Then we may be selectively enabling this or disabling this; enabling this or disabling this whichever way we want; that is, at any time, I can selectively enable interrupt 1 and interrupt 3 alone; interrupt 2 is disabled, or any other combination: interrupt 1 is enabled, interrupt 2 is also enabled; interrupt 3 is disabled. So this is done obviously to the CPU; we have to send an appropriate command and that command word will have the appropriates bits like masking or non-masking. So it is nothing but what we knew as configuring. Earlier remember, we were talking in terms of peripherals, we were talking about configuring and data transferring – that is configuration phase.

So selectively masking any interrupt; we just cannot do anything. Whenever the input comes on non-mask able interrupt, it will be responded to by the CPU. Sometimes it may be so critical that even the interrupt instruction cycle may not be completed, but anyway if it is called an interrupt, the instruction cycle must be completed. Now during configuration phase, we may selectively mask, that is, selectively enable and disable certain interrupts; non-mask able cannot be touched at all.

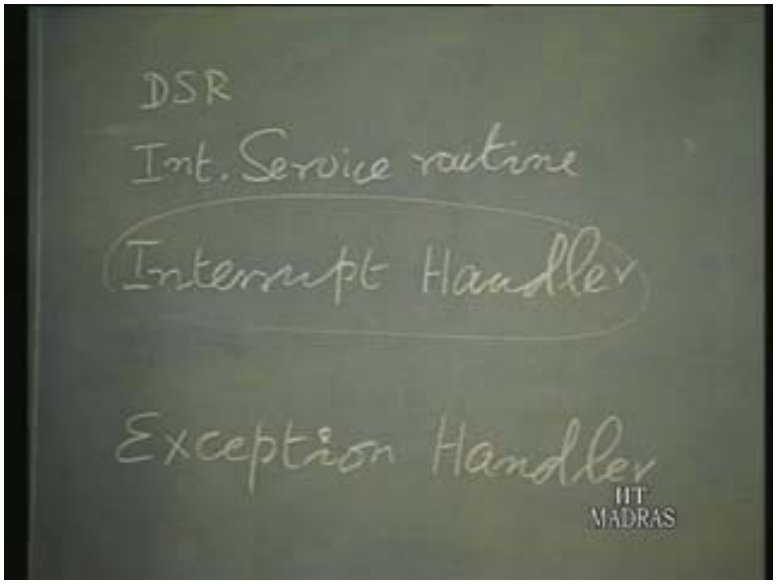
(Refer Slide Time: 37:55)



Generally these non-mask able interrupts are used for very critical things such as power failure. Some other systems have failure aspects. This is about what we need to see about interrupt in general. However while talking about the interrupt and the interrupt handler, that is, the service routine; I would like to touch upon a few other aspects also, which are related to it. Essentially interrupt is carried out through a context switch. We saw that a device is interrupting. Now it is also possible that interrupt can be achieved by executing an instruction; it may sound a little strange because there are some interrupt instructions. So the main routine is executed and then some situation arises and the CPU cannot proceed further. Now that situation may be because a device is interrupting or because the CPU has come across an instruction, which is called an interrupt instruction.

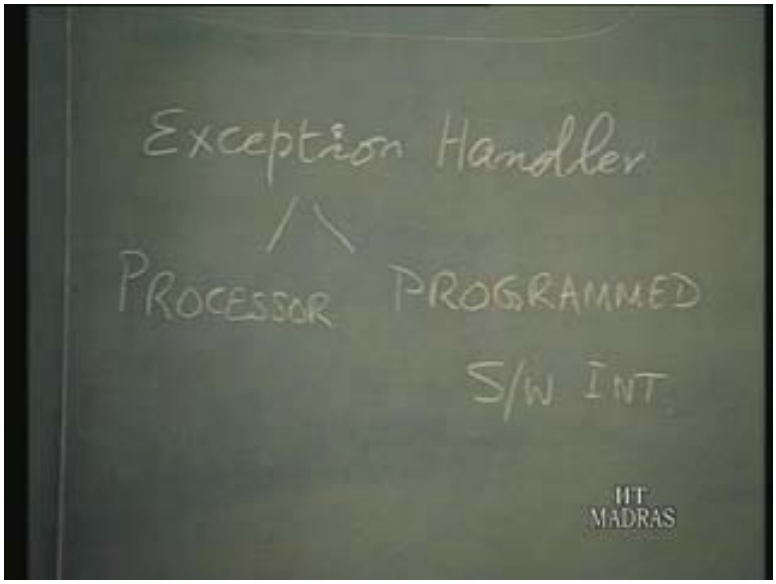
There are some software; we call these as software interrupts, though all the time we have been talking about an interrupt and it is through hardware and so on, there are software interrupts too. Similar situations may arise – like a call to a subroutine; it is somewhat like this. A call to a subroutine or the main routine is interrupted and the subroutine is taken up. Similarly in the software interrupt also, execution of software is an appropriate instruction; then some situation may arise. For instance, there is an instruction which is not part of this particular implementation of the processor, that is, an instruction has been reserved for future processor, but not for this version. Suppose the instruction comes, then it is really an illegal instruction. The processor cannot proceed or divide by 0; if it comes the processor cannot proceed. So the processor may get interrupted and we say in general the flag, such as an exception flag, is raised. That is, an exception has arisen and the processor has to respond to that exception; it is similar to the interrupt.

(Refer Slide Time: 41:42)



Now if we know precisely what the exceptions are, we can also store the routines or handlers for these exceptions. So the interrupt handler and exception handler essentially is something which the processor cannot proceed with, and instead of going rule by rule, now an exception has come. The processor has to respond to that. There are two types of exceptions: one is processor exception; the other one is in fact called a programmed exception.

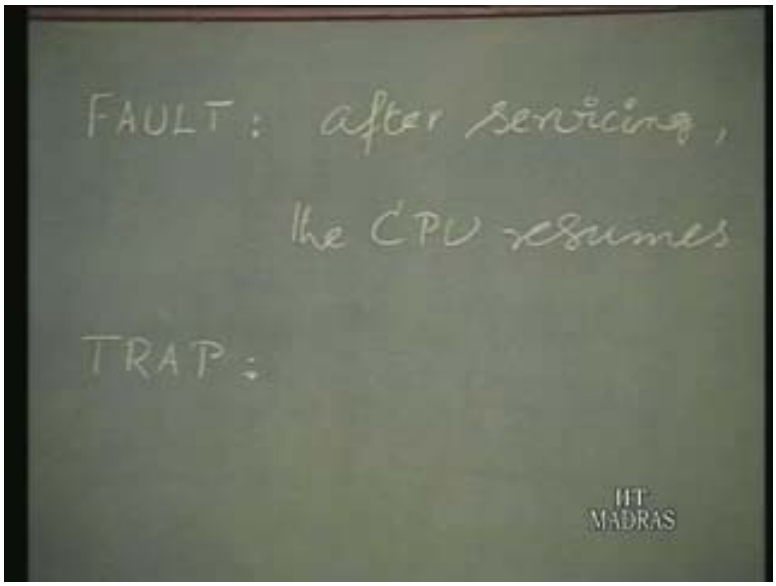
(Refer Slide Time: 42:27)



This programmed exception is similar to the software interrupt I was mentioning, which is essentially caused through software.

For instance, a software interrupt programmed exception is like this. Now let us see what the processor exception is. That is, the processor cannot proceed further because the programmed exception is given specifically through some instruction or other. The moment we say software, it is something which is acceptable by the processor, that is, essentially some instruction. What is this processor exception? We will see this in detail. In this, the response by the processor will be similar to what it would otherwise be in the case of interrupt; that is why we are taking it up. There are essentially three types – one is a FAULT.

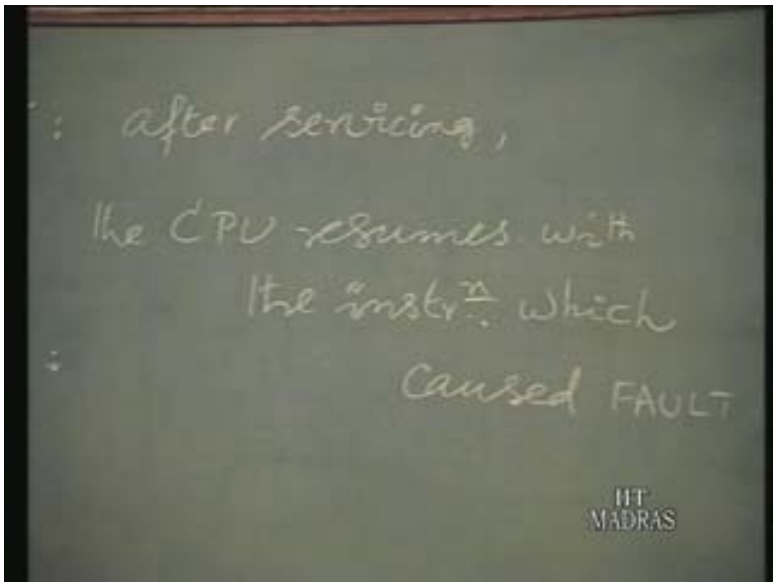
(Refer Slide Time: 45:32)



FAULT is the processor exception. Now what is the FAULT? We have come across this FAULT, a segment FAULT; what is it? The page is not accessible to the processor; the segment is not accessible to the processor because mainly the memory is not holding what the CPU wants. Now what is the situation? What is the remedy for that? The CPU has to bring in the new page or bring in the new segment, and then after it has brought in whatever the CPU requires, that CPU can continue with the same instruction, meaning an instruction has come across an address and that particular address, the contents of that address, are not available in the memory. So we call that a page FAULTS. The processor comes across a page FAULT; then we say an exception is raised and this is a FAULT exception. So a page FAULT handler – it is a program that is run and that sees to it that it swaps out one of the pages and swaps in a new page that is required, and at the end of it, the processor can resume from where it left off. So we say after servicing for whatever fault, the CPU resumes with that instruction; we will say resumes with the instruction which caused the fault. Now there is another type of exception called a TRAP. Now why are we talking about this?

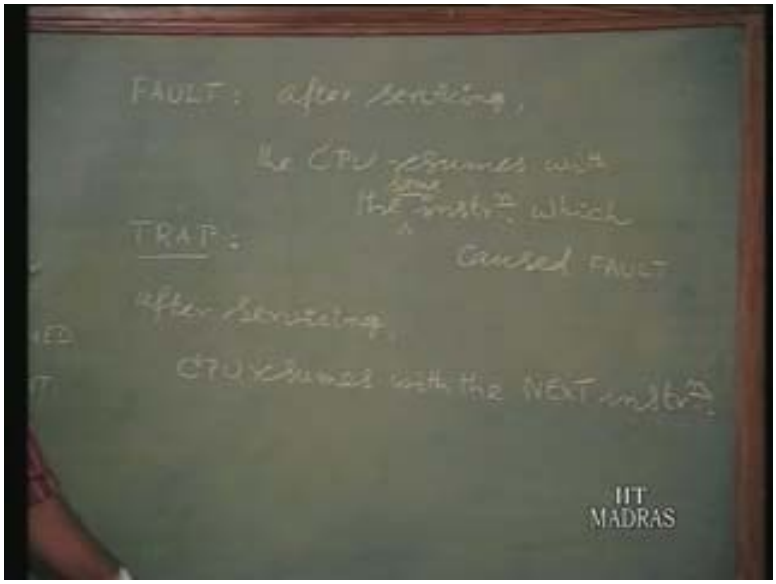
They are similar interrupt handlers; a fault handler is coming under exception and a trap handler is another exception; that is, the CPU cannot proceed; it is getting disturbed or interrupted. But it is not the same as interrupt; there are different names for it. What is a TRAP? When we say power failure causes the CPU to be trapped, meaning the AC power is dipping and DC power will go off in another few milliseconds, before that, a few instructions will be executed and those few instructions will possibly be for trap handling. It means to bring it to a satisfactory or rather, bring it to a logical portion; that is, end of that current instruction, and stop. May be a few instructions will be necessary, and then, stop, save everything so that later on, when the power resumes, the service can be restarted with the next instruction. In the case of FAULT, after servicing the CPU resumes with the instruction which caused the FAULT.

(Refer Slide Time: 47:23)



Now in the case of TRAP, after servicing, CPU resumes with the next instruction because a TRAP is a situation with which the CPU cannot proceed any further. So it will be brought as it is a legal instruction, but because of power failure and something like that it cannot proceed. So it will complete that instruction and the CPU resumes with the next instruction – this is important.

(Refer Slide Time: 48:20)



Now here we can emphasize – here it continues with the same instruction, which causes FAULT; and here, it is with the next instruction. So you can see there are minor differences; there is another type called ABORT.

(Refer Slide Time: 48:58)

That is, a situation arises and the CPU cannot proceed any further. So it has to ABORT; in other words the CPU stops. An abort handler causes the CPU to stop and it cannot proceed any further. That is, it is not like a FAULT, where some corrective action can be taken and we can proceed with the next instruction. It is not like a trap where the current instruction can be executed and then we can proceed with the next instruction.

Here the CPU cannot proceed any further because of this situation; that is, there is no correction that is possible. So we have a FAULT handler; we have a TRAP handler; and we have an ABORT handler and interrupt handler. These in fact are somewhat related. In both these cases the CPU stops and takes up some other service. Now specifically we were talking about device interrupt service here; these are all internal; the processor itself faces this. In addition to this, as part of the program, by including appropriate instruction in the software, we can also cause something similar to interrupt. But all these are internal to the processor, unlike this, in which the external device is involved. So in the next lecture, we will briefly take a look at the DMA and then proceed to trace the evolution of I/O over these few years.