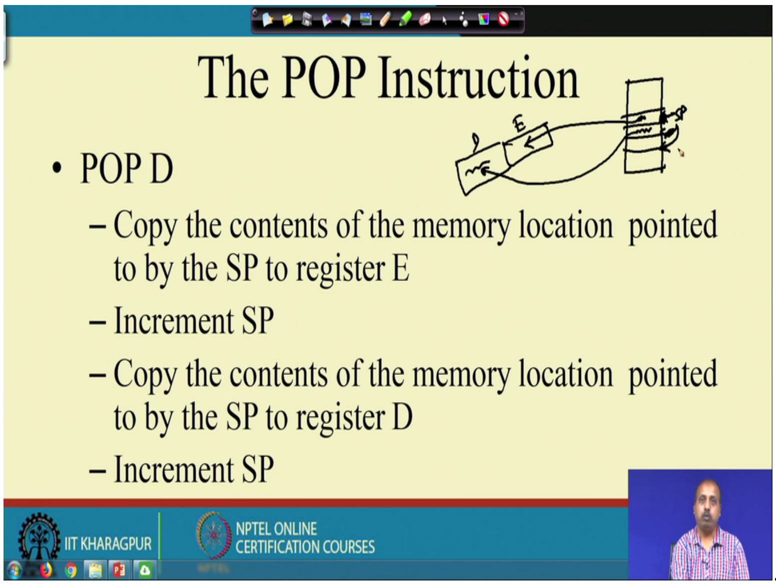
Digital Circuits
Prof. Santanu Chattopadhyay
Department of Electronics and Electrical Communication Engineering
Indian Institute of Technology, Kharagpur

Lecture - 56
8085 Microprocessor (contd.)

Next, we will be looking into the POP Instruction. So, this is just a reverse of push.

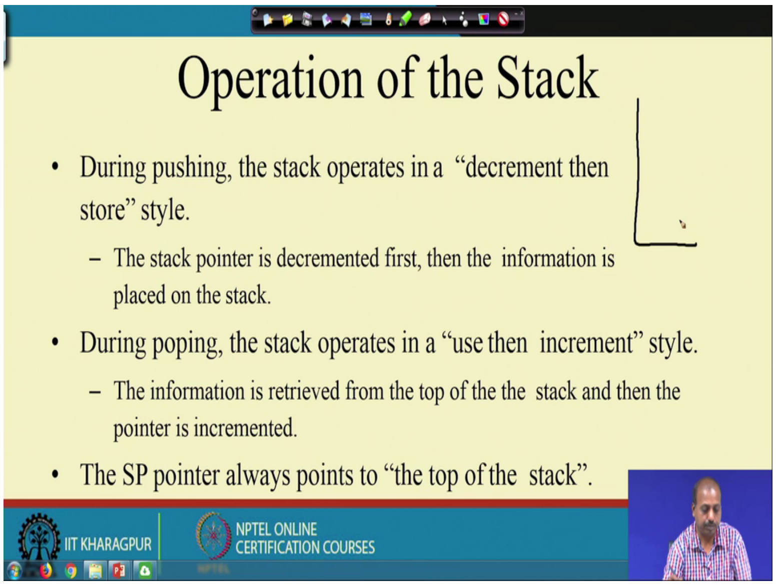
(Refer Slide Time: 00:18)


The POP Instruction

- POP D
  - Copy the contents of the memory location pointed to by the SP to register E
  - Increment SP
  - Copy the contents of the memory location pointed to by the SP to register D
  - Increment SP

So, in case of push instruction so, we are saving the content of this register pair on the stack. So, this POP instruction is to retrieve the register pair value from the stack. Again the POP works with register pair not with a single register. So, the way it operates is that it copies the content of memory location pointed to by the stack pointer to the E register. Then implement the stack pointer and then it will be copying the content of this stack point memory location pointed to by stack pointer into the D register and then implement stack pointer again.

(Refer Slide Time: 01:01)



Operation of the Stack

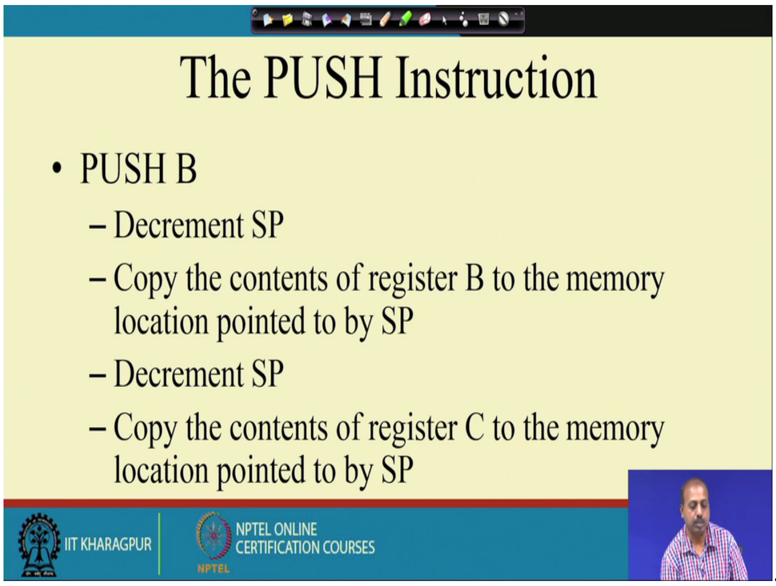
- During pushing, the stack operates in a "decrement then store" style.
  - The stack pointer is decremented first, then the information is placed on the stack.
- During poping, the stack operates in a "use then increment" style.
  - The information is retrieved from the top of the the stack and then the pointer is incremented.
- The SP pointer always points to "the top of the stack".

So, if we think if we if we this is the if this is the sorry so, if we if we look into this one. So, if these suppose this is the stack this is the portion of the stack, that we had this is the memory location and this is the stack. So, stack pointer is now pointing to this place. Now, when we do a POP D then so, these locations content so, these locations content in is first copied on to the E register.

So, this the D E pair so, this content it will come to the E register and then the stack pointer will be implemented. So, stack pointer now points to this location, the next location and that locations content will be copied on to D register. So, whatever be the content here will be copied on to the D register and then the stack pointer will be implemented further. So, that it points to the next memory location ok.

So, that way this POP instruction is executed. So, it is just a reverse of push. So, these push POP instructions can be useful for saving some value temporarily into the memory. So, now, how this stack is used? So, during pushing the stack operates in a decrement then store style.

(Refer Slide Time: 02:28)



The PUSH Instruction

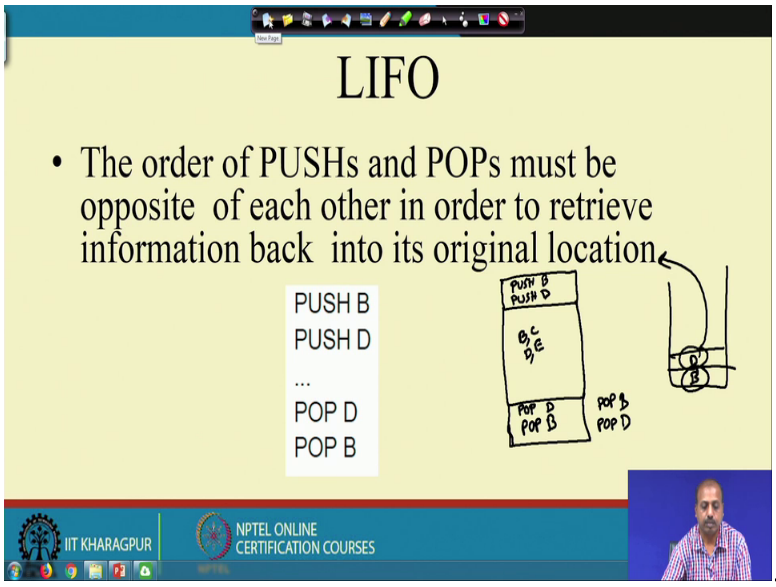
- PUSH B
  - Decrement SP
  - Copy the contents of register B to the memory location pointed to by SP
  - Decrement SP
  - Copy the contents of register C to the memory location pointed to by SP

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
NPTEL

So, first as if you look into the instructions then you see that here, it for the push instructions so, it is decrement SP and then after do the copy then again decrement. So, it is first decrement and then copy, whereas for POP it is first copy then increment. So, we can say that during push operation the stack operates in a decrement then store style, the stack pointer is the decremented first and then the information is placed on to the stack and for popping the stack operators as use then increments. So, as if it is all the copy then increment style.

The instruction is retrieved from the top of the stack and then the pointer is implemented. And stack pointer always points to the top of the stack. So, that is by the definition of this stack data structure, that it to always points to the top of the stack. So, that is taken care of in the 8 0 8 5 stack operation also.

(Refer Slide Time: 03:20)



LIFO

- The order of PUSHs and POPs must be opposite of each other in order to retrieve information back into its original location

```
PUSH B
PUSH D
...
POP D
POP B
```

Stack works in a LIFO fashion last in first out fashion, because the basic 8 0 8 5 designers. So, they have not put any restriction like the order in which you do a push POP operation, but as a user of the system. So, we have to be careful. So, the order of push and POP they must be opposite of each other in order to retrieve information back into it is original location. So, may be I have brought a piece program and that in that. So, we do not want that wherever from what wherever place I have called this program. So, whatever register this program uses.

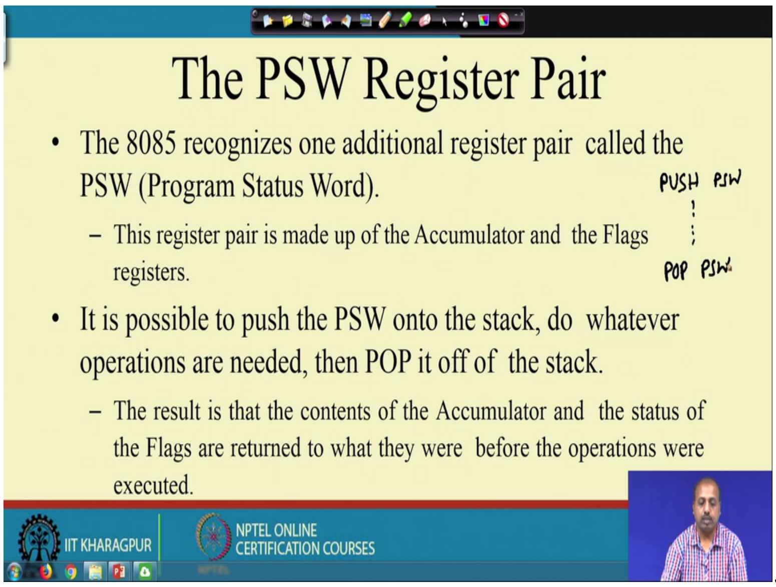
So, let us see that let us say that this uses the registers B C D and B C D. So, these register pairs it is using, then what I want is that wherever be the program from where this one has been called. So, after this program has finished. So, that the B C D E register they should get their old values back. So, we will see later this type of situation occurs when we consider the interrupts in a system. So, what we want is before coming to this the routine, whatever be the values of B C D E after finishing the routine the values should be restored to those.

So, for that purpose we have to we have to save this register. So, you should have 2 instructions here the push B and push D. So, these 2 instructions should be there for putting it or for saving them on to the stack. Now, at the end what is what is required is that we have to retrieve the content of this 2 this 2 this 4 register B C D E and how do we do that? How do we do that is by means of this POP instructions and this popping has to be done in the reverse direction.

So, that is POP D and POP B. So, if we do not do that so, if we ju[st] by mistake. So, if we write as first POP B and then POP D then POP D, then the content will be reversed is not it, because in stack we have saved this B register first and then the D register. So, while popping out while popping out. So, I should get I should POP out D register first then the B register.

So, the order in which you have pushed into the stack. So, while popping out you should do it in the reverse order. So, that is why it is called a last in first out structure or LIFO structure.

(Refer Slide Time: 06:15)



The PSW Register Pair

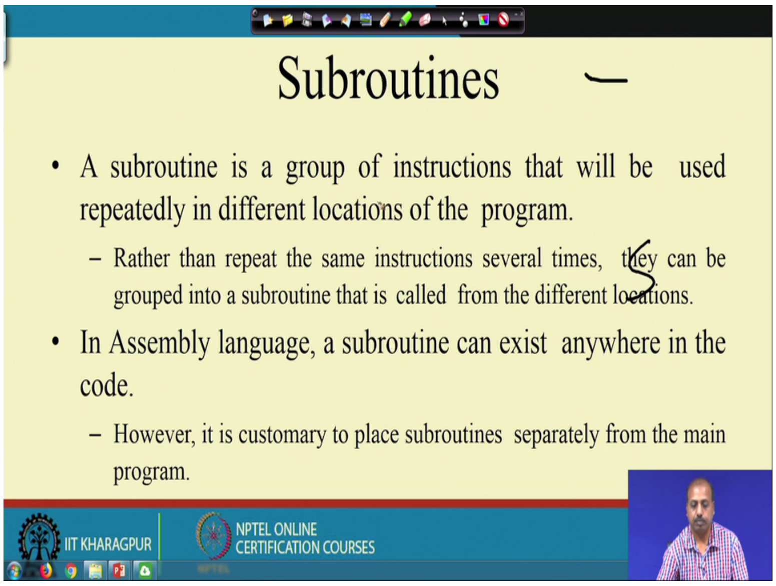
- The 8085 recognizes one additional register pair called the PSW (Program Status Word).
  - This register pair is made up of the Accumulator and the Flags registers.
- It is possible to push the PSW onto the stack, do whatever operations are needed, then POP it off of the stack.
  - The result is that the contents of the Accumulator and the status of the Flags are returned to what they were before the operations were executed.

PUSH PSW
⋮
POP PSW.

Another, very interesting register that this 8 0 8 5 has is PSW register pair. So, this is PSW physically the register consists of the accumulator and the status word register ok. So, this is called program status word or processor status word sometimes. So, this register pair is made up of the accumulator and the flags register. So, this accumulator and flag, they constitute this PSW register. It is possible to push the PSW onto the stack do whatever operations are needed and then POP it of the stack.

So, that way we can use this you can use this PSW register. The result is that the content of the accumulator and status flags status of the flags are returned towards they were before operation were executed. Again the same thing many times we want that this a register and the status they should be available at the end of the program at the end of the execution of a piece of program. So, for that purpose at the beginning you can do a push PSW instruction. So, you can have an instruction like push PSW at the beginning and at the end. So, we can have POP PSW, we can have push PSW and POP PSW as the 2 instructions. So, that is so, that is the utility of this PSW register pair.

(Refer Slide Time: 07:43)



Subroutines

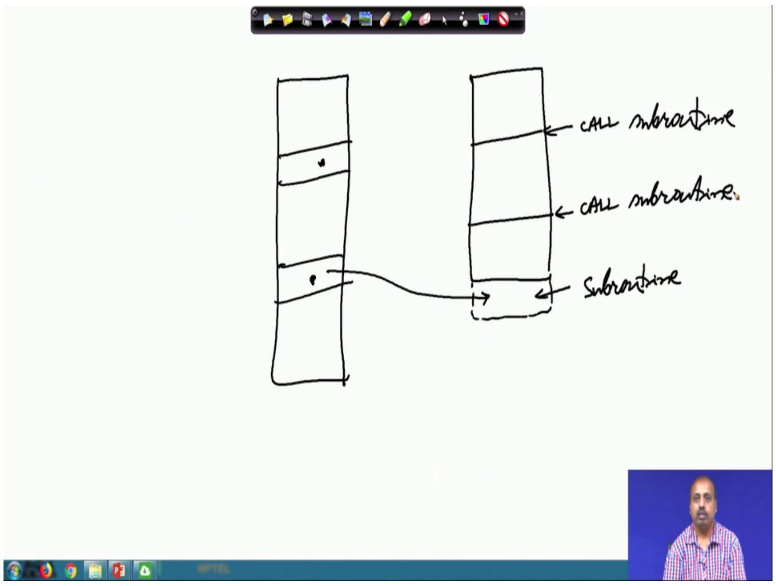
- A subroutine is a group of instructions that will be used repeatedly in different locations of the program.
  - Rather than repeat the same instructions several times, they can be grouped into a subroutine that is called from the different locations.
- In Assembly language, a subroutine can exist anywhere in the code.
  - However, it is customary to place subroutines separately from the main program.

Next, we will be looking into one very important concept, which is known as subroutine. So, a subroutine it is a group of instructions that will be used repeatedly in different locations of a program. So, rather than the same instructions putting same instructions several times they can be grouped into a subroutine and that is called the that is called from the different locations.

So, this is similar to the procedures that we have or the functions that we have in high level languages. So, here we write in case of assembly language program some part has to be a repeated. So, you just put it separately and then we can you can just call this subroutine.

So, if I if I have a piece of program.

(Refer Slide Time: 08:33)



And, if we find that in that program so; this some parts of the program are similar. Only some parameter some register are changing and things like that, then what you can do? So, if this code and this code are very similar. So, you can reorganize this entire code like this. So, you put a you put a small portion at the ends. So, that is actually the code that we have here.

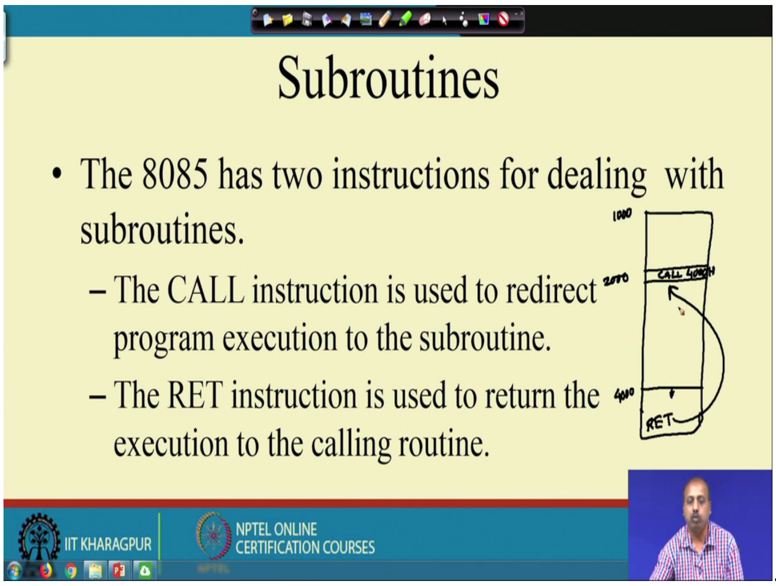
So, the since the codes are similar so, put the code only once. And so, this is this will be called a subroutine and this subroutine is called from different places, where this place you have got a call. So, this is a call to the subroutine. And, similarly sometime later again you give a call to the subroutine; so, this is a call to the subroutine.

So, this way what we are saving is we are saving the extras program length that we had. So, because of this is only twice that I have shown here. So, maybe 1 routine is needed very often. So, as a result so, it is called several time it is it is copied several times in it is in a straight line code.

So, if you are putting a subroutine, then that part can be saved and we can make the program size small particularly in microprocessor base system. So, since the space is a concerned. So, making them small saves the space. So, that is the utility of this subroutine. So rather than repeating the same instructions several times. So, they are grouped together into 1 subroutine and they are called from several locations.

In assembly languages subroutine can exist anywhere in the codes. So, though in my example I have shown the subroutine to be at the end, but it is not necessary. So, you can put the subroutine at any place. So, it does not put any restriction in assembly language programs. So, there is no restriction, but for the readability purpose. So, they should be puts separately from the main program otherwise readability becomes a problem.

(Refer Slide Time: 10:51)

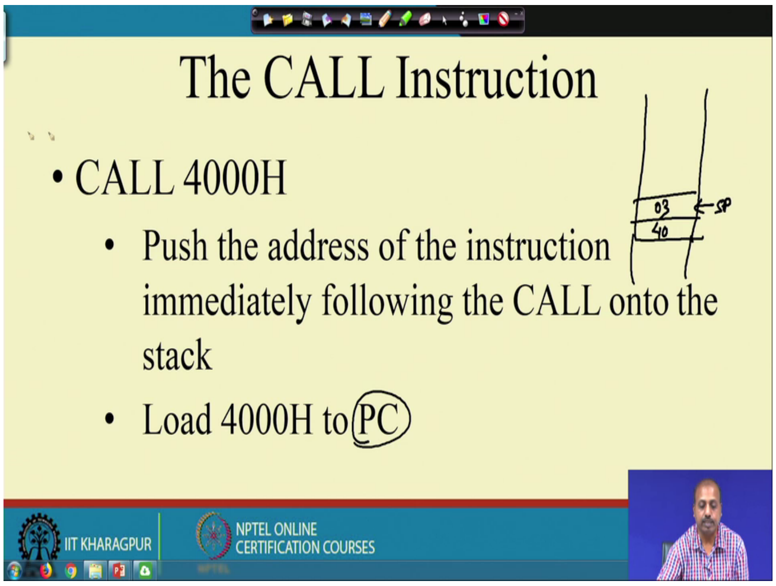


Subroutines

- The 8085 has two instructions for dealing with subroutines.
  - The CALL instruction is used to redirect program execution to the subroutine.
  - The RET instruction is used to return the execution to the calling routine.

Next in case of 8 0 8 5 there are 2 instructions that deal with subroutines, the call instructions that is used to redirect the program execution to the subroutine. And the return instruction or ret instruction used to return from the execution of the execution to the calling routine.

(Refer Slide Time: 11:08)


The CALL Instruction

- CALL 4000H
  - Push the address of the instruction immediately following the CALL onto the stack
  - Load 4000H to PC

So, for example, these call 4000 hex. So, that will that will we that for calling the subroutine and we have got it return instruction. So, it is like this that if this is my main program and then so, suppose your main program starts at memory location 1000 and it goes like this. And, then this is the subroutine that you have starting at location 4000. Now, somewhere here at location say 2000 I want to give a call to this subroutine. So, for that purpose the instruction is call this the call 4000 hex. So, the size of this instruction is 3 bites as you can understand that call will take 1 byte and this 4000 H will take 2 bytes. So, total it is a 3 byte instructions.

So, location 2000, 2001 and 2002 so, they are holding the instruction. And, then when this instruction is executed what the system does it jumps to the location 4000 and start executing from this point onwards. Now, any subroutine it should end with a return instruction or ret. So, when the processor it is your find this ret instruction, because ret is again 1 byte instruction because it has only the up code part. So, when this ret is found then the processor will try to come back to the location just after the call instruction, that is the 4002 ok. So, it will be 4003. So, 4000 4001 and 4002 they were the call instructions.

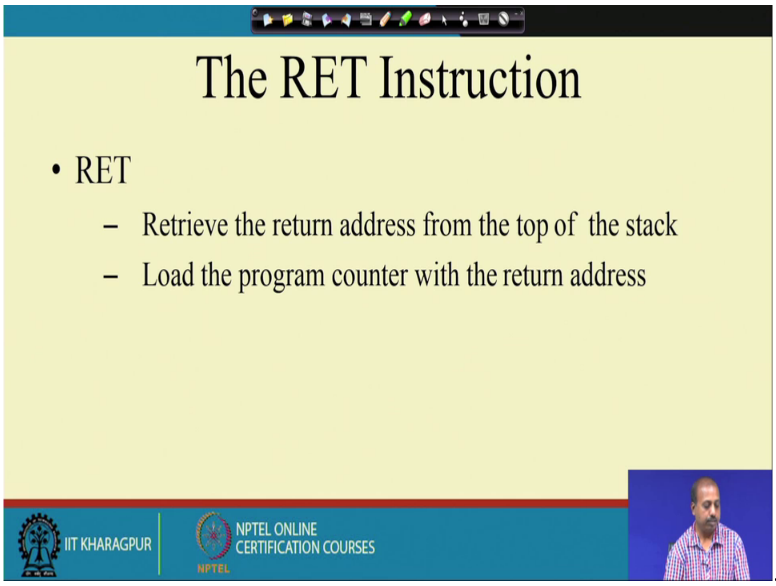
So, in this return is executed it will try to come back to 4003. Now, the question is how does it come back? So, how the processor will know in the return address I am not mentioning like where to go? So, how the processor will know where to go?.

So, this is actually accomplished by the call mechanism. So, at the time of calling itself so, this is taken care of that we have got this return address saved on to the stack. So, in this call instructions push the address of the instruction immediately following the call into the stack. And, as I have said that call is a 3 byte instructions. So, that immediate next instruction is at address 4003.

So, this 4003 value will be pushed on to the stack and then. So, what happens is that? So, if this is the stack. So, value 4003 will be pushed on to the stack and as we know that it is a 4 0 is the higher order byte and 0 3 is the lower order byte. So, this is the. So, stack pointer will now point to this location ok.

Now, it will be it will be calli[ng]- now the program counter value program counter register will be loaded with the 4000 hex as a result from the next instruction the processor will execute from location 4000 hex. So, after some time so, it will be for finding the return instruction, and then it will be popping out the content from the stack to get the return address. So, we will see how is it done?.

(Refer Slide Time: 14:23)



The RET Instruction

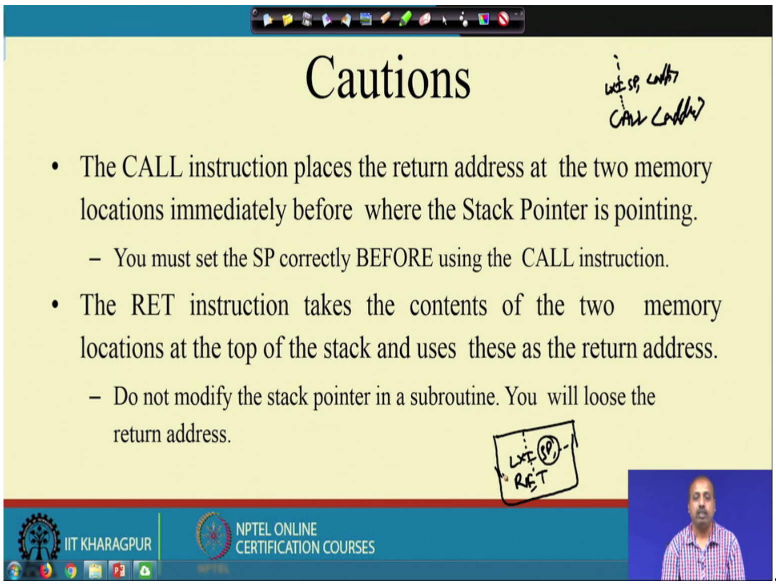
- RET
  - Retrieve the return address from the top of the stack
  - Load the program counter with the return address

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
NPTEL

So, return instruction so, it will retrieve the return address from the top of the stack. So, it will get the content from the top of the stack and it will load the program counter with the return address. And, as we know that the program counter register it actually determines the instruction, which will be executed next.

So, for the return what we need to do is to get the content from the stack and put it on to the program counter. So, that is exactly what is done in the return instruction.

(Refer Slide Time: 14:53)



Cautions

LXI SP, <addr>
CALL <addr>

- The CALL instruction places the return address at the two memory locations immediately before where the Stack Pointer is pointing.
  - You must set the SP correctly BEFORE using the CALL instruction.
- The RET instruction takes the contents of the two memory locations at the top of the stack and uses these as the return address.
  - Do not modify the stack pointer in a subroutine. You will loose the return address.

LXI SP, -
RET

So, there can be some caution like the call instruction places return address at the 2 at the 2 memory locations immediately before where the stack pointer is pointing. So, we must stack pointer correctly before using the call instruction. So, this is so, 8 0 8 5 designers what they have done? So, they have made the call instruction execution like this that, this program counter value will be saved on to the stack.

Now, if so, but it does not do anything regarding the stack pointer initialization. So, if by mistake this stack pointer contains some garbage value, then the program then the program counter values will be saved onto that memory location. So, that makes it very difficult like. So, it is the user’s responsibility to ensure that the stack pointer is pointing to a valid memory address, from where which is not used for any other purpose and the return address can really be saved onto that address and can be retrieved later. So, before the call instruction is execute call instruction is put into a code. So, before the first call instruction, if this is my program somewhere I am doing say call to some address. So, before doing this call somewhere previously I must have executed this instruction LXI SP comma some valid address.

So, that the stack bottom is loaded on to the stack pointer stack pointer points to the bottom of the stack that should happen. And, after that only this call should be executed.

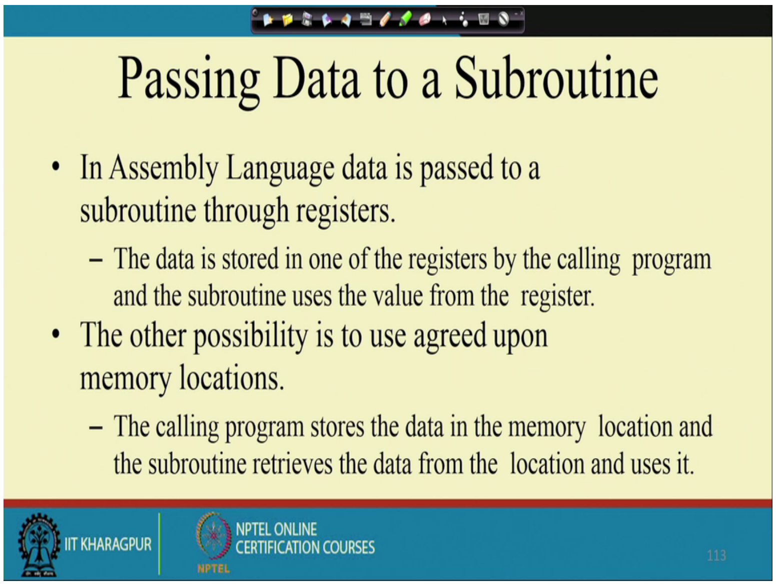
So, this is just a precaution. So, if you do not do this if you do not execute the LXI instruction then also this call will execute, but the effect may be disaster.

So, this may be that it collapse some other values of your program other data of your program, because the stack pointer points to those arbitrary locations and then it tries to write this program counter value there. So, you must save the stack pointer correctly before using the call instruction. Similarly, the return instruction it takes the contents of 2 memory locations at that top of the stack and uses these as the return address.

So, you should not modify the stack pointer in a subroutine. So, if in a subroutine if we modify the stack pointer then this will be lost well I if this is my subroutine and there somewhere I change the stack pointer values. So, LXI is P 2 to something else. And, then I put a return then what will happen? So, it will be it will be trying to load this stack it will load the stack pointer is something else. So, when the return is executed. So, it will try to get the return addresses from that other location not the location, where the stack this return address was saved while the call instruction was being executed.

So, should not modify the stack pointer in a subroutine, otherwise we will lose the return address value. So, that we have to be careful.

(Refer Slide Time: 17:54)



Passing Data to a Subroutine

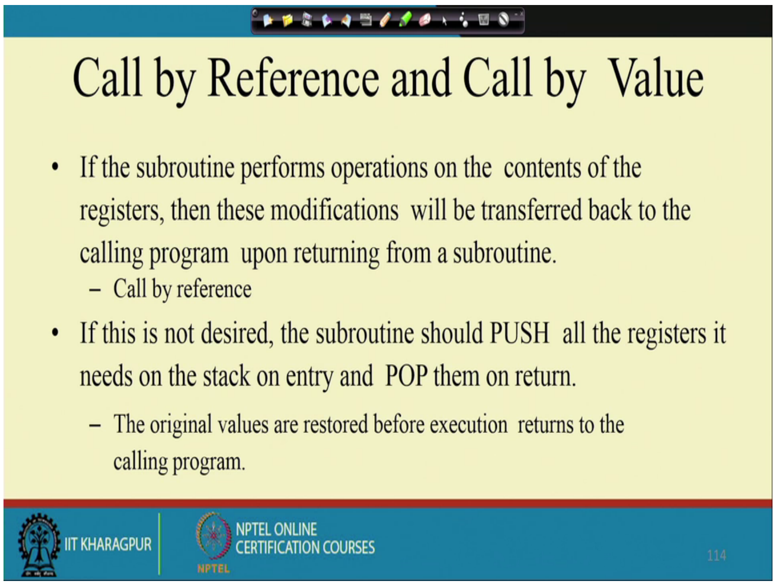
- In Assembly Language data is passed to a subroutine through registers.
  - The data is stored in one of the registers by the calling program and the subroutine uses the value from the register.
- The other possibility is to use agreed upon memory locations.
  - The calling program stores the data in the memory location and the subroutine retrieves the data from the location and uses it.

So, you can also use this stack to pass data to a subroutine. So, in assembly language program. So, we can pass the data to the subroutine through registers that is a 1 possibility. So, so this data is stored in one of the registers by calling the program and the subroutine uses the value from the register. So, by the calling the calling program for example, if it has to pass and 8 bit value, it may be that it saves the value in the B register and then calls the subroutine. And, in the subroutine we access the B register to get the value that has been passed. So, that is one possibility. So, you can use one register to hold some hold some parameter that you want to pass to the subroutine.

But, since the number of registers are limited and they are also used for some other purposes. So, it is it is it is better that we have some option ok. So, and that option is via the stack so, if because memory is very large compared to the data the space requirement of a program of for the data space requirement of a program. So, we can use this stack for the purpose of this transfer of this parameters.

So, what the what the program does the calling program instated of storing the content of this data onto a register, it can store the content in a memory location and the subroutine will retrieve the data from the location and use it. So, that is possible. So, that way we can have some memory location and to do that operation.

(Refer Slide Time: 19:40)

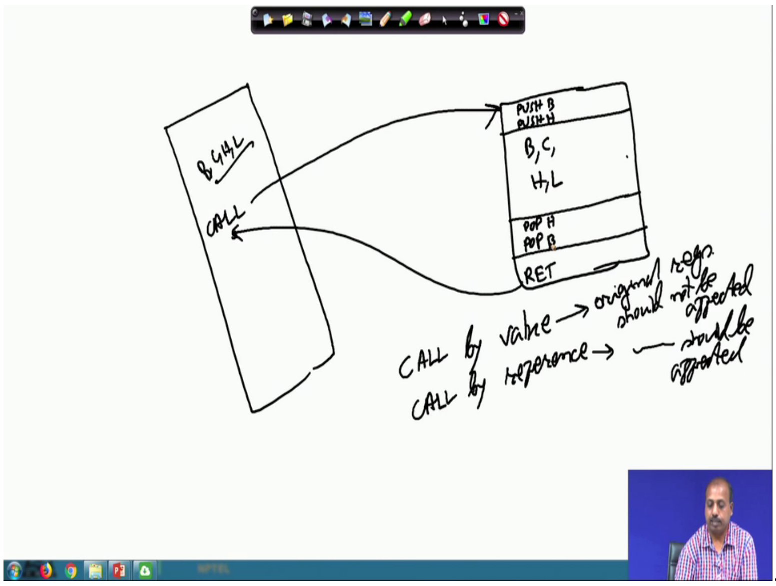


Call by Reference and Call by Value

- If the subroutine performs operations on the contents of the registers, then these modifications will be transferred back to the calling program upon returning from a subroutine.
  - Call by reference
- If this is not desired, the subroutine should PUSH all the registers it needs on the stack on entry and POP them on return.
  - The original values are restored before execution returns to the calling program.

Now, if a subroutine performs operations on the contents of the register and then these modifications will be transferred back to the calling program upon returning from the subroutine. So, this is called call by reference and if this is not desired then subroutine should push all the registers it needs on the stack on entry and POP them on return. So, this is the original value will not be lost. So, what we mean is like this that suppose I have a subroutine ok.

(Refer Slide Time: 20:17)


B,C,H,L
CALL
PUSH B
PUSH H
B, C,
H, L
POP H
POP B
RET
CALL by value → original regs should not be affected
CALL by reference → — should be affected

So, I have a subroutine where the so, this is my subroutine and there I am using this registers like it this part of the code I am using the registers B C H and L fine. Now what can happen? When these modifications are done? So, this is so, if this is the main program from where I am calling this subroutine. So, there is a call instruction here which actually calls this subroutine and somewhere here I have got the return. So, it will be coming back to the this point.

Now, if this subroutine modifies this B C H L registers, then once you come back here you see that this B C H L are all modified. So, whatever be the values of this B C H L registers here. So, they are modified by the subroutine ok. So, if you want that so, there may be situations in which you want it. So, maybe it is doing some computation in the B C H L register values in the registers and then there those values are needed by the calling program.

So, that is one possibility, other possibility is that this B C H L. So, they were used as some temporary storage by the calling by the by this subroutine, and the values of the of those registers of the on the original program they should not be modified.

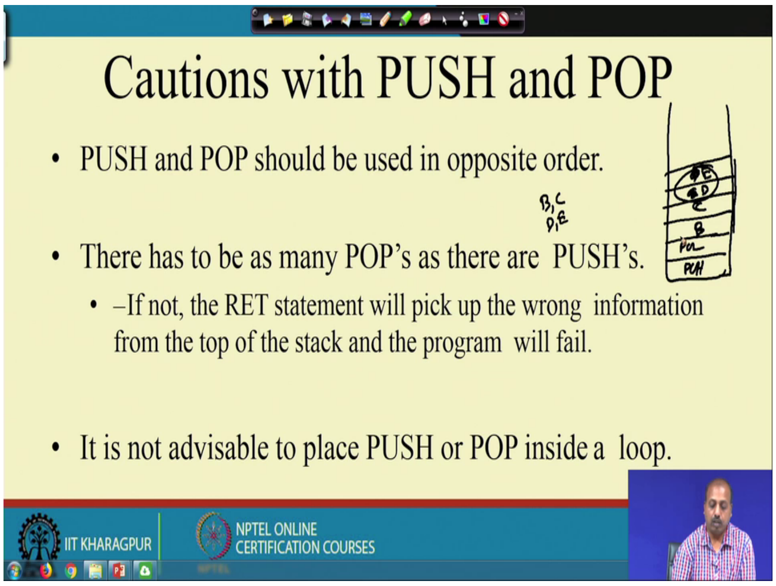
So, for that purpose so, we have got 2 type of situation one is call by value another is call by reference. So, in case of call by value the original registers should not be affected, original registers should not be affected. So, this is the situation I was talking in second in the second case that is whatever be the values of B C H L here. So, that that should not be should not get disturbed and call by reference means original registers should be should be affected they should be affected.

Now, the second part the call by reference implementation is very simple. So, you do not have to take any precaution to restore the values of B C H L, but if you really want that this B C H L content should not be lost you have to have a call by value, then the first few instructions of this subroutine it should be push B and push H.

So, they will be saving those 2 registers and before doing this return instruction ok. So, I should POP them out. So, I should use instruction like POP H and POP B before the return. So, that the values are restored. So, the original content of B C H L registers. So, they are not lost. So, this way we can do this call by value and call by reference type of implementation by using the stack.

. So, if the subroutine performs the contents of the registers if these modifications will be transferred back to the they will be transferred back to the calling program upon returning from a subroutine. So, this is the call by reference and if this is not desired then the subroutine should push all the registers, it needs on to the stack on entry and POP them on return. So, this is the original values are restored before execution returns to the calling program. So, that has to be done.

(Refer Slide Time: 24:01)



Cautions with PUSH and POP

- PUSH and POP should be used in opposite order.
- There has to be as many POP's as there are PUSH's.
  - –If not, the RET statement will pick up the wrong information from the top of the stack and the program will fail.
- It is not advisable to place PUSH or POP inside a loop.

Now, push and POP as we have already say that the push and POP should be used in the opposite order the order in which you push the registers you should POP in the reverse order. And, there has to be as many POP's as there are pushed ok. So, if the number of suppose for example, when these see the number of push is say 4 and number of POP's is three; that means, the situation is like this is that when you are calling a subroutine you know that first if a first the return address is saved on to the stacks. So, P C high and P C low. So, if you draw in terms of a stack so, it is like this first the P C high will be stored then the P C low will be stored.

And, then if you if you say that I will be I have pushed this registers B and C, then the C register B register will be saved here, then the C register is saved here then if I push D and E, then E register will be saved here and D register will be saved sorry D register will be saved here, and E register will be saved here. Now, while doing the POP so, naturally when you are returning so, it is desirable that when you are returning the stack everything has been erased from the stack and the stack has got this P C L and P C high at the 2 top most entries.

So, if there is a mismatch in the number of POP's push and POP's like if may be I have just popped out these 2 by using a POP instruction and I have did not do any further POP's. So, the stack top actually contains the register values B and C the old values of B and C. Now, if you do a return now then this C register value will go to P C low and the

B register value will be go to P C high, and that is dangerous. So, it will take your program to somewhere else not to the point from where the subroutine was called. So, this ret statement will pick up the wrong information from the top of the stack and the program will fail and it is not advisable to push put to this push and POP inside a loop.

Because, it is often very difficult to judge like how many pushes are pops will be done and whether they are really balanced or not, because the last iteration of the push POP last iteration of loop normally it comes out without executing the body. So, in those cases it is difficult to ensure that this push POP they will be matching.

So, you try to push this push and POP instructions outside the loop and if you are if it is very much necessary to put inside the loop body, then ensure that it is put at the beginning of the body and popped out at the end of the body and they are always executed. So, it is not there in some cases some the POP may not be executed it should not happen like that. So, that way we have to be careful with this push and POP instruction while writing the subroutines.