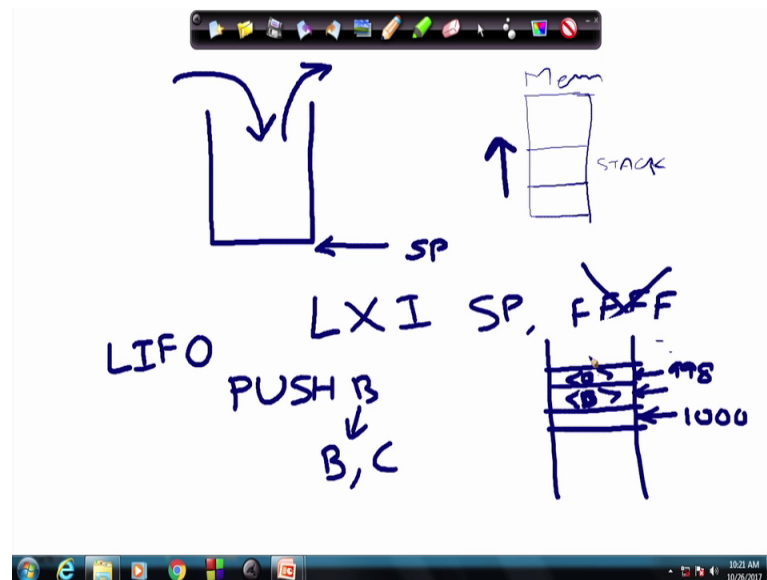


Microprocessors and Microcontrollers
Prof. Santanu Chattopadhyay
Department of E & EC Engineering
Indian Institute of Technology, Kharagpur

Lecture - 15
8085 Microprocessors (Contd.)

In our last class we have seen the structure of stack and we have taken the seen the corresponding statements like PUSH and POP. So, today before going further we will explain the stack structure a bit more, and with some examples and then go towards the subroutine and calls and returns.

(Refer Slide Time: 00:38)



So, as we said that in 8085 or for that processor, and any other processor for any other processor the stack happens to be a part of the memory. So, if this is your memory, then this stack is a part of the memory, which is declare which is the stack and then this stack can grow in this stack can grow in one direction, this they normally it grows in the upward direction.

Though it is not mandatory that it will always grow in the upward direction, but by in general it is made to grow in that way. So, conceptually we can view that stack as if a data structure like this, where at the beginning we are at the bottom of the stack, and that is pointed to by one special register which is the stack pointer in case of 8085.

Similarly, in other processors there may be some other name, but this is called stack pointer in 8085. So, at the beginning of the program if a program is expected to use this stack like say you use the PUSH and POP instructions subroutine calls etcetera, then the user is expected to initialize this stack pointer to some proper value. Otherwise the stack pointer will contain some garbage value, this register will contain some garbage value, and this PUSH POP they will start updating memory or some arbitrary locations. And the by if there is a possibility that with the stack pointer not being initialized properly it modifies some essential part of the program or the essential part of the operating system.

So, that way we should initialize this stack pointer. So, the way we should initialize the stack pointer is by the LXI instruction LXI SP comma some value. So, we in our last class we have seen that we initialize it to FFFF hex. So, that it means that it is the highest possible address at which the 8085 can have, and it is initialized to that so it is initialized to the bottom of the memory.

So, but this FFFF hex so, this is not mandatory. So, you can if you find that your program in your system you can define the stack to be the maximum address you say 4000, then you can initialize the value to 4000. But that way you are absolutely free to initialize this stack pointer value, but we have to make sure that it does not corrupt other part of the data and program in the memory.

Now, once this stack pointer has been initialized. So, you can use the PUSH and POP instructions to save the content of some registers into the stack, or get the content of some content of the stack onto some register. And this is a LIFO structure last in first out structure. So, the PUSH operation so, this will put some data element into the stack, and the POP operation will take it out. So, PUSH and POP they both occur from this one same end so, when you PUSH it. So, it is pushed it is copied on the topmost element in the stack and when it is popped out the topmost element is popped out.

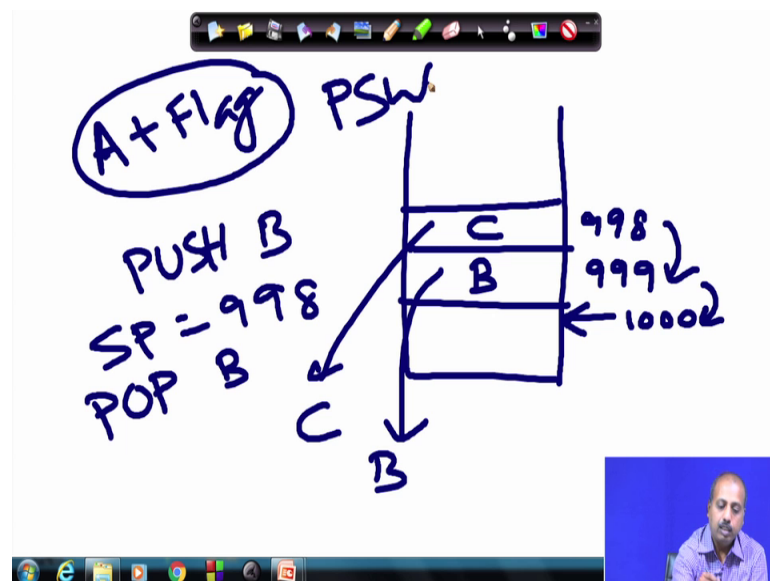
And as we said that these PUSH and POP instructions they work on the register pairs. So, if you can have an instruction like say PUSH B; so this PUSH B instruction. So, this will modify the stack in this way. So, we can say suppose at present the stack is suppose at present my stack is like this. So, it is this is my memory, and the current stack pointer

is pointing to this location. So, this location is say the address 1000. I am talking I am writing in decimal. So, I assume that the value is 1000.

So, then so, when this PUSH B instruction is executed, the first the stack pointer will be decremented. So, it will come to this location, and the content of C register will be put here. The content of the C register will go there. And then stack pointer will be decremented further. So, stack pointer will become I will get the value 998, and the at 998 the B register content will be put.

So, for this B, B is as A register pair it consists of the registers B and C. So, the C goes to the sorry, this actually as for intel convention the higher order byte will go to the higher order address. So, this will be the B value will be coming here. This will be the B value, and this will be the B value. And this will be the C value, let me draw a fresh diagram like, if this is the stack then this PUSH B.

(Refer Slide Time: 05:58)



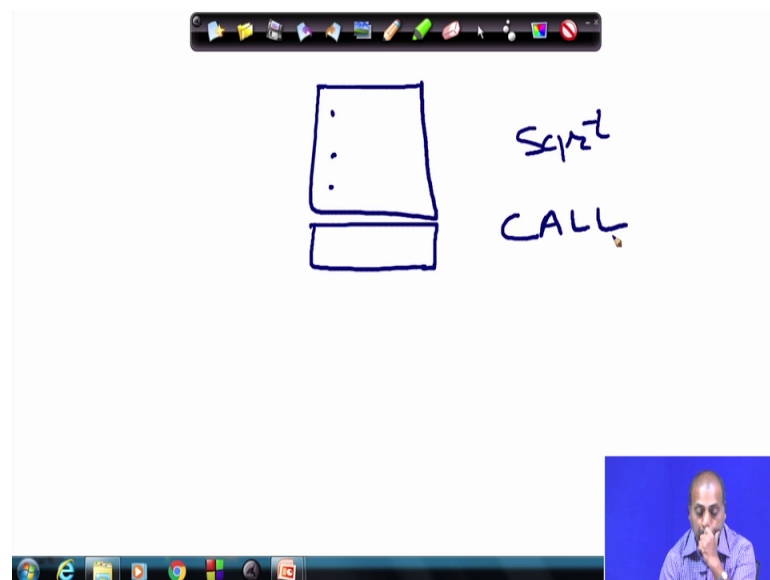
So, if previously the stack pointer was a here, the B value will be coming here, and the C value will be coming to the previous location, if so, if these value is now this is memory location 1000. So, this is 999 and this is 998. So, this where the since the lower order B C is the lower order byte. So, it will go to the lower order address location 998, and B being the higher order byte. So, it will go to the higher order address that is 999. So, this way the PUSH instruction is executed, and the stack pointer value Sp will new value will become 998.

Similarly, the POP instruction will do just the reverse. So, if you are executing POP B in this configuration, then first the content of this location 998 which is pointed to by the stack pointer will be taken to the C register. So, this value will go to the C register stack pointer value will be incremented to 999, and then the content of this will come to the B register, and the stack pointer value will be implemented 2000; as if 2 top most entries have been taken out from the stack.

So, this way this PUSH and POP instructions are executed. And we have got for so, for a B C D E and H L, we have got registered pairs. Also, for the accumulator register we it is not paired accumulator A is a single register. So, what is done with this a this p the flag register is added. This flag register is added, and this whole thing is given the name PSW; so processor status word or PSW. So, that includes the accumulator and the flag. So, the accumulator will be saved first, and then the PSW with in the flag register. So, that way the values will be pushed into the stack.

So, next we will look into the subroutine structure.

(Refer Slide Time: 08:10)

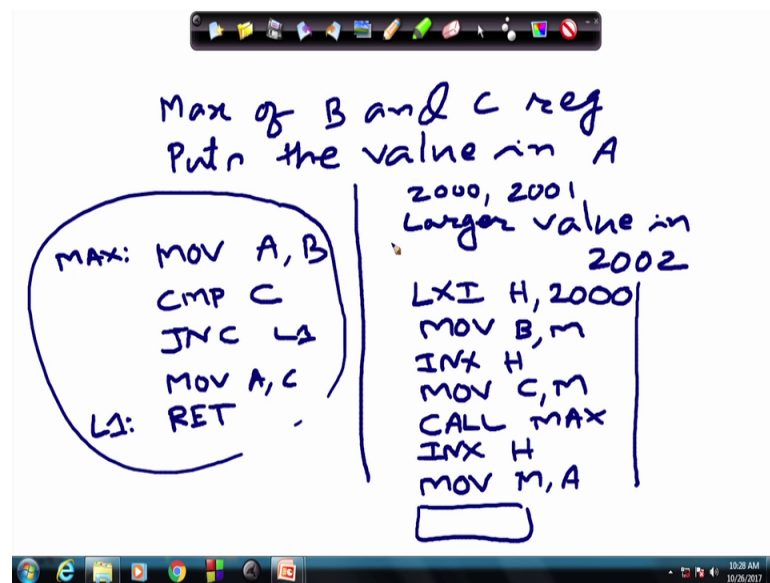


And in the subroutine structure, if the subroutine structure so, what happens is that a part of the program which is required here several times. Like, in the last class we are telling, that in my I may have a program, where the particular function say the square root calculation function is required several times in the program, instead of putting it putting the code several times repeating the code several times.

So, we can just we can just take it say we can write it only once. So, we write this square root routine here, and wherever we need this square root routine, we call it from those points. So, there is an instruction call by which you can call a subroutine, and that subroutine will be executed by calling it at that position.

A typical example; can be like this say, we have got suppose we have got a program.

(Refer Slide Time: 09:13)



We have got a we have we write a function or subroutine that finds the maximum of say B and C registers, and puts the maximum puts the value in A. So, if I want to write this particular program. So, I can do it like this say MOV A comma B, then compare with C. So, I have got the content of B register into a and that is compared with C. So, if it happens that B is already larger. So, if B is if this accumulator is smaller than C that is C is larger then the carry flag will be saved.

So, if there is if there is no carry jump on no carry to say some level L 1, otherwise we know that the maximum is the B. So, jump on no carry that is B is already larger than C. So, the in that case nothing happens otherwise if the carry is there; that means, C is larger than B. So, in that case I should move the content of I should move the content of C register into a MOV A comma C, and then there here is my level L 1, and here I return from the subroutine by means of a ret instruction. So, we call it this this subroutine, let us give it a name. So, let the name be max.

Now, this subroutine we can use it to find say maximum of 2 numbers, like say in the memory. So, suppose I want to come compare the contents of memory location 2000 with 2001, we want to compare the contents of these 2 memory locations, and store the larger one, the larger value, I want to store in say 2002.

So, this program we want to write. So, we can do it like this, first we do an LXI H comma 2000 MOV, first value I get in B register B comma M, then INX H MOV C comma M, and then I call this subroutine. So, I can call this subroutine by the instruction call, CALL MAX, and we know that at the end of this call, the A register will have the maximum value. So now, I can save the value in 2002. So, I can say another INX H, and then I can say MOV M comma A.

So, by this I can save the larger of the 2 numbers in to the location 2003. So, this way I can write down the subroutines; so this part of the subroutine that I have written. So, it can be placed anywhere in the program. So, so ideally, we put it at the bottom of the main program. So, if this is the main program so, at the bottom of that we put the subroutine, but it can be put anywhere. So, it can be at any stay say somewhere before this or after this it does not matter.

So, with that we will go back and discuss on the subroutine and all.

(Refer Slide Time: 13:11)

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.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | 114 | 10:28 AM 10/26/2017

So, subroutine it is a group of instructions that will be used repeatedly at different locations in the program; so rather than repeating the same statement same instructions several times. So, they are put only once. So, in this way it will save in the space requirement. So, the total program size will be reduced, but what about the execution time.

So, execution time will be more, because we will see shortly that why whenever you are going to execute a subroutine the return address has to be saved on to the stack or when you are coming back from the subroutine. The return address has to be popped out from the stack. So, that way the execution time will increase, but the size of the program reduces the readability of the program improves.

So, in assembly language program is subroutine may be anywhere in the program, but it is customary to put at the end of the main program.

(Refer Slide Time: 14:11)

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.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

So, next we will have next we will see some subroutines in 8085. So, there are 2 instructions that deal with 8080 subroutines in 8085, one is the call instruction. So, the call instruction will be used for calling a particular subroutine. So, redirect the program execution to get subroutine and how will it happen. So, this the way it takes place is that the program counter value will be loaded with the address of this subroutine and that way the next instruction to be executed will be from that particular address.

So, this way execution will be happening in case of call and this return instruction. So, return instruction is used for returning from the subroutine. So, in the subroutine ends at the subroutine we will end with a return or RET. So, getting this ret instruction the processor will POP out from the stack the 2 topmost entries, and it will put those values into the PC low and PC high those 2 locations though those 2 registers. So, PC will now have the return address.

So, you can understand that it becomes a responsibility for the processor, that when it is going to the call. So, before starting the subroutine it should save the instruction it should save the address of the instruction just after the call so that after the call instruction. So, in case of so, that it can return to the appropriate address.

(Refer Slide Time: 15:39)

The CALL Instruction

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

1000
1003

IIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES

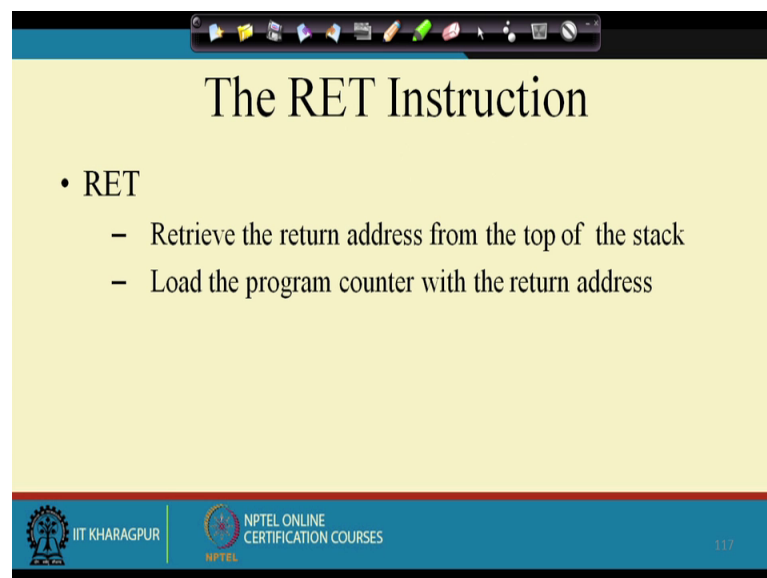
So, next so, the structure of this call instruction is like this call followed by a 16-bit address. So, what it does is that the PUSH the address of the instruction immediately following call on to the stack, and then the program counter value is loaded with 4000.

So, it is like this that say this is the call instruction, if this is the call instruction then, and this is at say location 1000. And call instruction you can understand the size of this call instruction is 3 byte, for 1 byte will be going for the opcode call, the second byte will be 4 0 third byte will be 0 0. So, this is a 3-byte instruction. So, my next instruction is at location 1003. So, this is my next instruction.

So, what the system will do is that it will store this value 1003 into the stack. So, it will put it will PUSH this 1 0 and 0 3 these 2 bytes into the stack, which is the current program counter value. And then only the program counter will be loaded with this value 4000. So, that my return address is saved onto the stack, and now I am ready to start execution of this of the procedure the of the subroutine that we are talking about. So, we can do that.

So, next we will see like how this return instruction is executed, this return execution is like this.

(Refer Slide Time: 17:09)



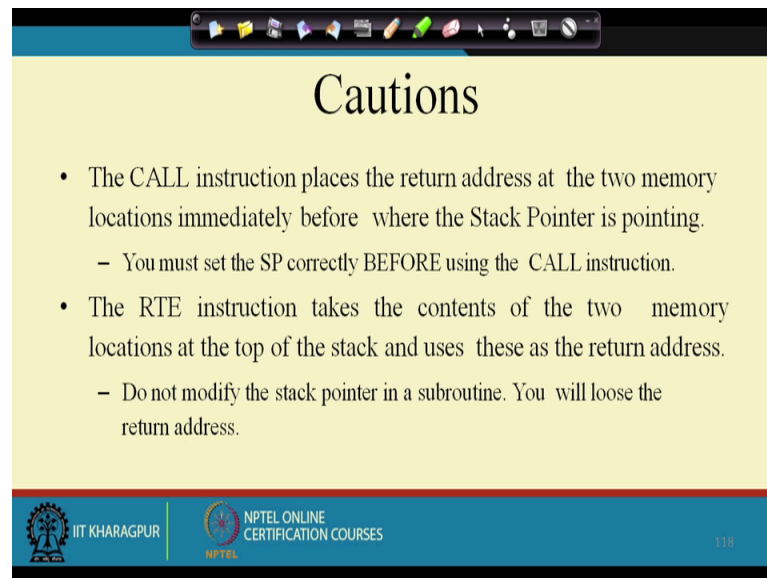
The slide is titled "The RET Instruction" and is presented in a yellow-themed window with a standard operating system taskbar at the top. The content is as follows:

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

The footer of the slide contains the logos for IIT KHARAGPUR and NPTEL ONLINE CERTIFICATION COURSES, along with the number 117.



So, it will retrieve the content of the return address from the stack. So, in the previous example the 1003 is in the top of the stack. So, when this procedure finishes this 1 0 is put into the PC high higher order 68 bits and 0 3 will be put in into the lower order 16 bits. So, that is the 2 topmost stack entries will be popped out, and they will be put into the program counter so that your program execution resumes from location 2 1 1003.

(Refer Slide Time: 17:49)



Cautions

- 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 RTE 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 lose the return address.

 IIT KHARAGPUR |  NPTEL ONLINE CERTIFICATION COURSES

118

So, this is the usage of this usage of this call and return instruction. Now there are some cautions like this call instruction places return address at the 2 memory locations immediately before or at the stack pointer is pointing. So, this is the real. So, some time back I was telling that before starting the actual code of the program. So, we should save the we should we should set this stack pointer to some proper values.

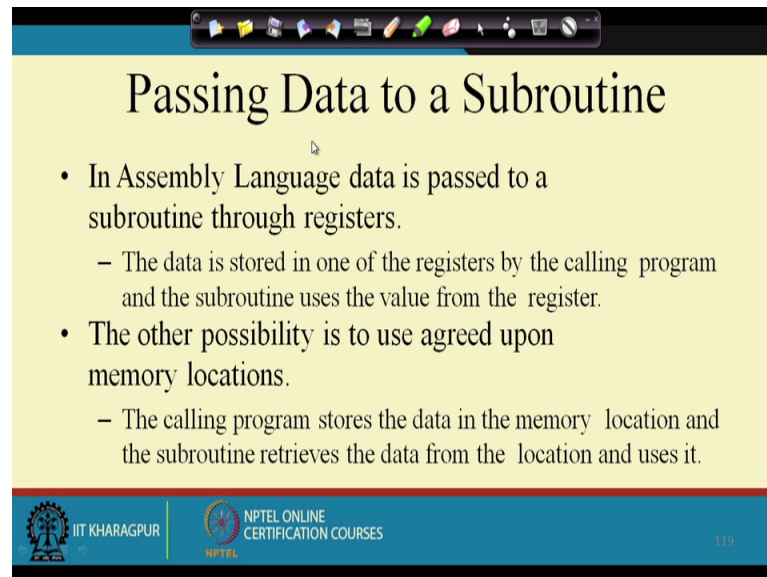
So, if that is not done, and your program uses the call instruction, then it will be putting those return addresses into whichever location pointed to by the stack pointer, and by even by unfortunately if that points to some valid program statements program bytes. So, that will also get overwritten. So that, make create a difficulty.

On the other end this return instruction, or though it will be it will be taking out the content or 2 topmost memory locations; so naturally if in a subroutine. So, if you are not careful enough and modify the stack pointer register inside the subroutine, so that may cause harm because while returning from the subroutine. So, whatever be the stack pointer value, the processor will access the top 2 most locations and put them onto the PC.

So, if the stack pointer is not proper, then it may again get some the PC may get some garbage value, and your program instead of really returning to your desired location. So, it may return to some arbitrary place, it may not be even in the same program it may be some arbitrary place within the memory. So, it returns there.

So, we have to be careful while manipulating this stack pointer register inside procedure. So, you should normally you should not do we should not do this, but if it is absolutely necessary. So, we should be taking enough care, to ensure that the stack pointer value is restored before the return instruction.

(Refer Slide Time: 19:44)



The slide is titled "Passing Data to a Subroutine" and contains the following content:

- 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.

The slide footer includes the IIT KHARAGPUR logo, the NPTEL ONLINE CERTIFICATION COURSES logo, and the number 119.

So, next thing that we do is a subroutine how are you passing data, passing the value on which the subroutine will operate. Now we have already seen one technique by which we can do this like the example we have taken to get the maximum of the 2 values are stored a 2002 2001.

So, there we have used that B and C registers to hold those 2 values. So, those are actually passing the parameters to the subroutine. So, this way the first option for doing this is to pass the values are pass data to subroutine by means of registers. So, data stored in one of the registers by calling the program, and the subroutine uses the value from that register. So, whatever data so, in that case we required 2 values to be passed. So, you use 2 registers, if you requires on a single value to be passed we can use one register, that way.

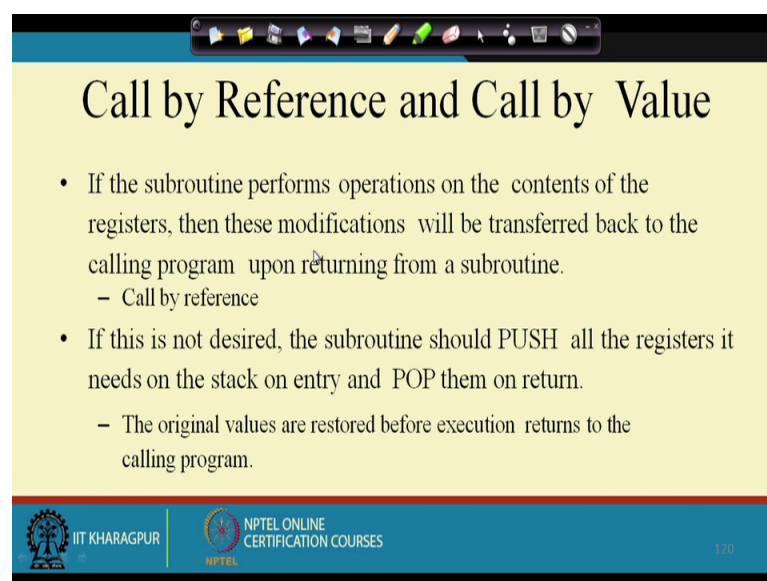
So, the so, this before calling the subroutine we should initialize that register with the proper value, and subroutine will use that register value. So, as a result it will get the correct content. And then later on it is when the program returns or subroutine returns then maybe we can have some return value. But the so but that is true if the number of

values that we want to pass are less. Like I want to say we have got some general-purpose registers like Vcd H L, and you can also say that I will also use the A register. So, that is total 7 bytes. So, if your data that you want to pass is restricted to 7 bytes. So, this CPU registers may be sufficient, but if you are we asking for larger size data to be passed for example, a and add I v on to write a program a subroutine that sorts numbers and you want to sort to say 100 numbers. So, 100 numbers cannot be passed through registers. So, there we have to pass through some memory locations.

So, we can earmark some memory locations that will contain the input array. So, maybe we can fix some location like we can say that that the location the memory location 1000 onwards the 100 integers will be stored which are to be sorted. So, that way the destina the subroutine will know that my parameters are from location 1000. Other possibility is that the there is agree the agreement may be like this that the HLPR. So, this will point to the memory locations that will contain the data that will call from where the data will start.

So, in that case before calling the subroutine I should initialize the H L pair with the proper value. So, that when I am in the subroutine. So, I use the H L pair to a index through the array elements.

(Refer Slide Time: 22:30)



The slide is titled "Call by Reference and Call by Value" and contains the following text:

- 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.

The slide footer includes the IIT KHARAGPUR logo, the NPTEL ONLINE CERTIFICATION COURSES logo, and the number 120.

So, there can be 2 types of 2 2 types of modifications that can occur in the in the parameters that we have passed. So, in one case like in the normal case like if you are

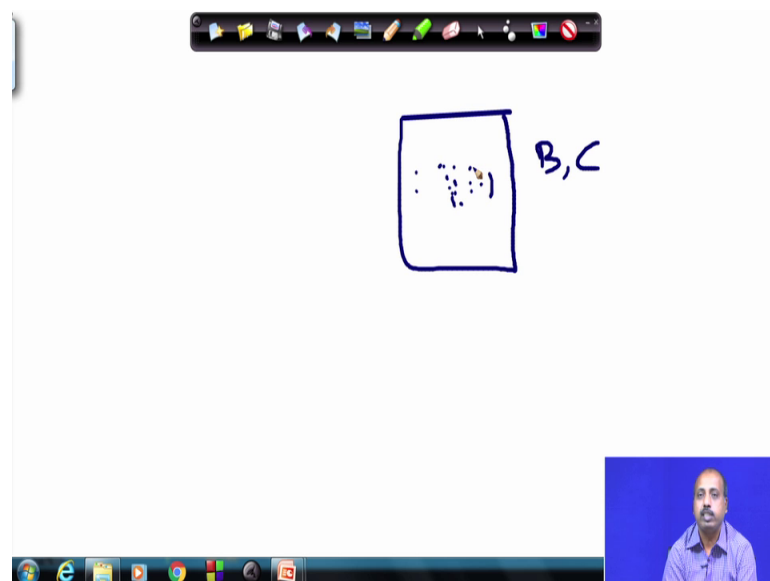
say I have passed the parameters being through the registers B and C, and inside the program it is just modifying those values. So, it is modifying the contents of B and C registers.

Then once you return to the program you do not get the original values of B and C, but you get the modified values of B and C. So, this may or may not be desirable. Sometimes we want that the modification carried out by the by the subroutine is really the modification that we desired and in another case. So, we may just say that this value I passed just for as an input value I do not want the register to be get to get modified.

So, that way these parameters so that that can be classified into 2 classes one is called call by reference other is called call by value. So, in case of call by reference, the values are values are modified in the subroutine. So, those register contents are modified, and as a result. So, we do not get back the original values now if you say that I do not want to destroy the contents of those original registers, I just want to use it and do some calculations around that, but the result should not modify those registers.

Then for doing it we need to save the original registers using PUSH instructions before using them in the subroutine, and we have to before exiting the subroutine, we should POP them. So, we will take a simple example, say like this.

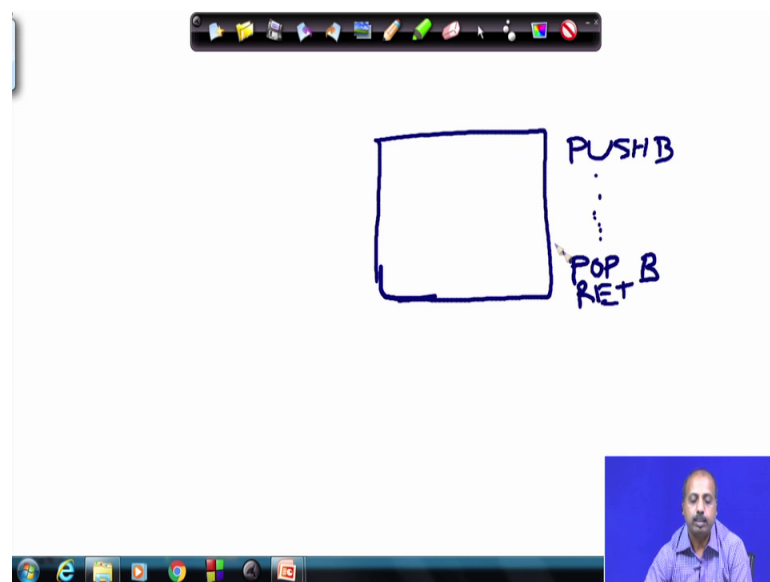
(Refer Slide Time: 24:25)



So, this is the subroutine and we want that in this subroutine. So, this uses the registers B and C, and we will expect that these values should not be modified. This register content should not be modified. So, this in the in the body of this subroutine it modifies this B and C. So, it works with this B and C register. So, that way it gets modified, but we do not want that these modifications will get reflected into the B and C values of the origin of the of the calling program or the or the main program.

So, how to do this is there in that case at the beginning of this at the beginning of the subroutine at the beginning of the subroutine we have to save the contents of those 2 registers.

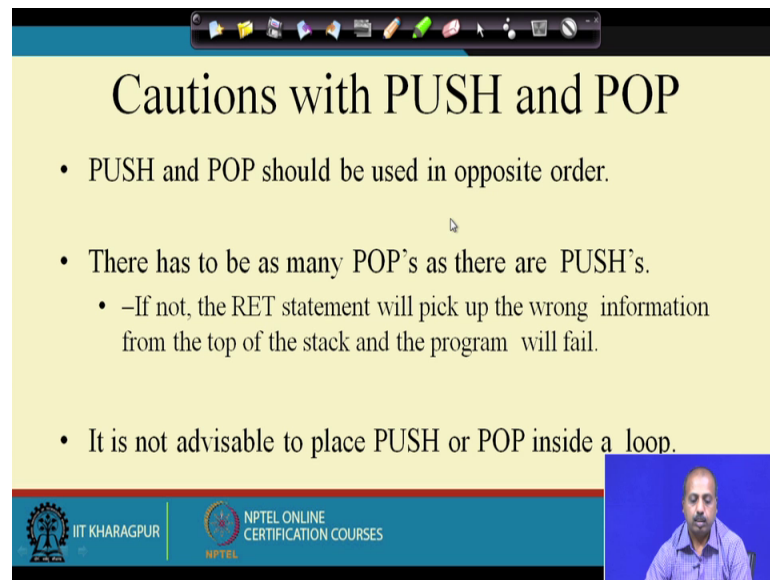
(Refer Slide Time: 25:06)



So, if this is the address for the first statement in this case should be A PUSH B so that the B C pair will be saved onto the stack. And after that I should have the code whatever code is there that will be there, then at the end before the return instruction I should have the POP; so this POP B. So, it will restore the content of those B and C registers from the stack. So, the original values were saved in the stack at the beginning original values are restored from the stack. At the end after that I should have the return instruction.



So, this way I should do. So, that I can get and I can get the implementation of this call by value. So, original values are restored before execution returns to the calling program.


(Refer Slide Time: 26:11)



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.

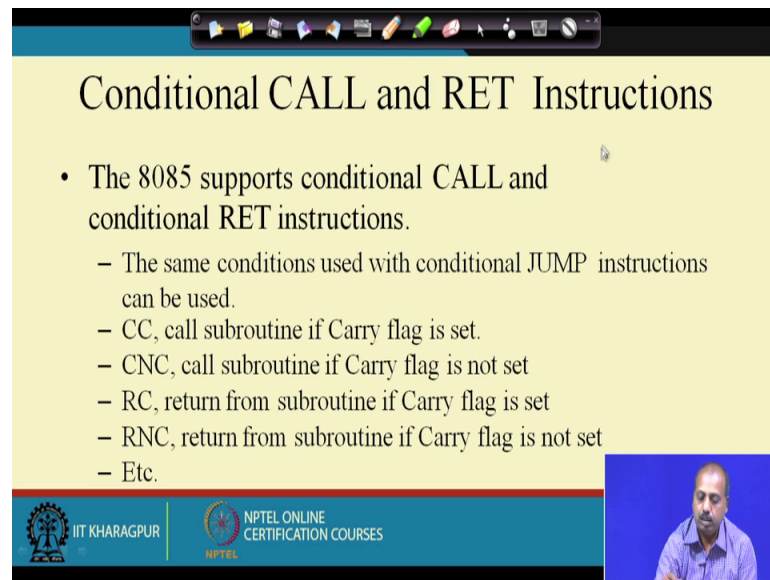
 IIT KHARAGPUR |  NPTEL ONLINE CERTIFICATION COURSES



So, this is required, then another interesting thing that you should have is that PUSH and POP should be used in the opposite order. So, if you PUSH in at the beginning if you have first pushed the register B then PUSH the register D, then the POP should be in the other order it should be POP D first, and then for B first then the POP B.

So, because it is otherwise what will happen is that the values that you get are not correct. So, maybe at the beginning I have pushed in the order B and D, while popping if I POP out B first and not D, then I will get the wrong value. So, if you have pushed like say if you have if you have pushed like say PUSH at the beginning you have written like PUSH B PUSH D.

(Refer Slide Time: 26:54)

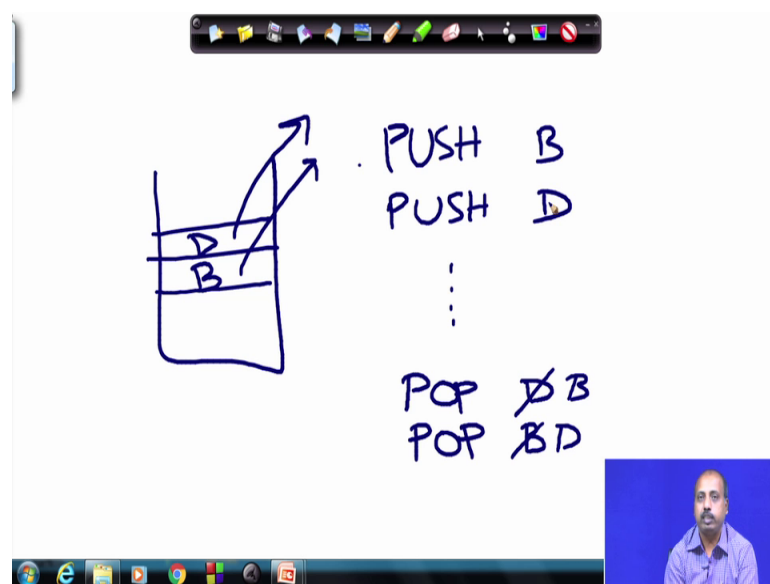


Conditional CALL and RET Instructions

- The 8085 supports conditional CALL and conditional RET instructions.
 - The same conditions used with conditional JUMP instructions can be used.
 - CC, call subroutine if Carry flag is set.
 - CNC, call subroutine if Carry flag is not set
 - RC, return from subroutine if Carry flag is set
 - RNC, return from subroutine if Carry flag is not set
 - Etc.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

(Refer Slide Time: 27:00)



A hand-drawn diagram on the left shows a stack structure with two cells. The bottom cell contains 'B' and the top cell contains 'D'. Two arrows point upwards from the 'B' cell, indicating the push operation. To the right, the following assembly code is written:

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

The video inset shows a speaker.

Then at the end you should do like POP D first, and then you should do POP B.

If you change the order, then what will happen is that we will get a wrong value, because first B has been pushed. So, in a LIFO structure; so first the value B has been pushed, and then the value D has been pushed. So, while popping out I should POP out this D value first and then the B value. So, if we change the order here they D here, B here and D here; then, I will get the original content of D register into B register and original content of B register into D register.