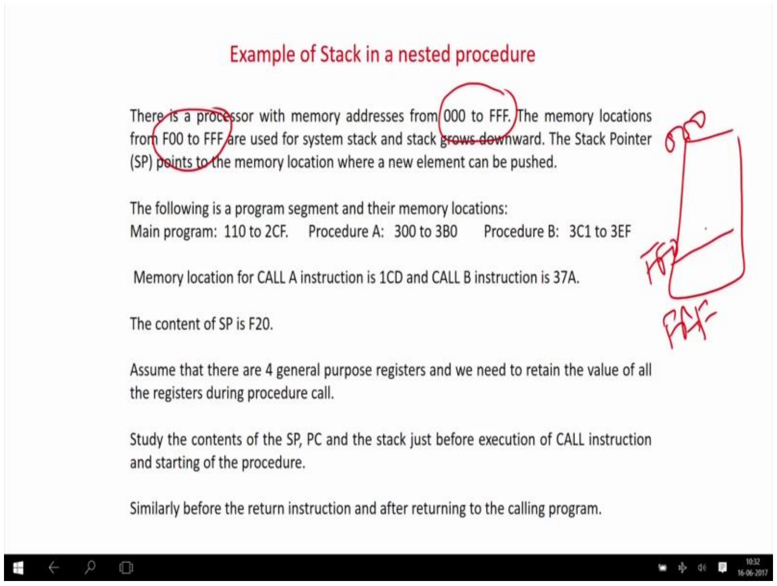
So, this is overall the basic comp, CPU components which are involved in a procedure call all the components are quite similar to all other functions and all other instructions we have discussed till now, but stack pointer is a register which plays a very special role over here.

(Refer Slide Time: 17:27)



Example of Stack in a nested procedure

There is a processor with memory addresses from 000 to FFF. The memory locations from F00 to FFF are used for system stack and stack grows downward. The Stack Pointer (SP) points to the memory location where a new element can be pushed.

The following is a program segment and their memory locations:
Main program: 110 to 2CF. Procedure A: 300 to 3B0 Procedure B: 3C1 to 3EF

Memory location for CALL A instruction is 1CD and CALL B instruction is 37A.

The content of SP is F20.

Assume that there are 4 general purpose registers and we need to retain the value of all the registers during procedure call.

Study the contents of the SP, PC and the stack just before execution of CALL instruction and starting of the procedure.

Similarly before the return instruction and after returning to the calling program.

Now, we have talked enough theory and this is actually a nutshell what I have told you completes the theory of a procedure call it is basically only 3 very simple steps, you call a procedure, before calling a procedure place everything in this stack, call the procedure, jump to the instruction where the first instruction of the procedure is there jump to that memory location, after completing the procedure return back to the main program from where the procedure is called and at the same time before jumping there jumping back to the main place where we have started; where the procedure is called if you regain back the values of program status word, the program counter, variables which you have saved before going to this one.

So save, call, then call means save the context, call the procedure go to the procedure and complete the procedure, return back to the main part of the program from where the procedure was called and before returning back you have to regain what you have saved. So, this is a very simple way of implementing a procedure. Now, it is better that this was all from a theoretical context now we are going to look at in a simple example.

So, we have a processor whose memory address range is 000 to FFF. So, it is a address bus will be 12 bits the memory contains and as I told you. So, this is your main memory. So, it will have 000 to say FFF and a part of this is reserved for the stack. So, what is reserved for the stack if you see FF0 to FFF maybe some from here. So, this part of the program memory is allocated for a stack when your stack will be saved and it is assumed that the stack goes downwards; that means, you will be going downwards as a stack implementation ok. And the main program is located from 1100, 110 to 2CF, procedure A 300 to 3B0, procedure B is located from memory location 3C1 to 3EF.

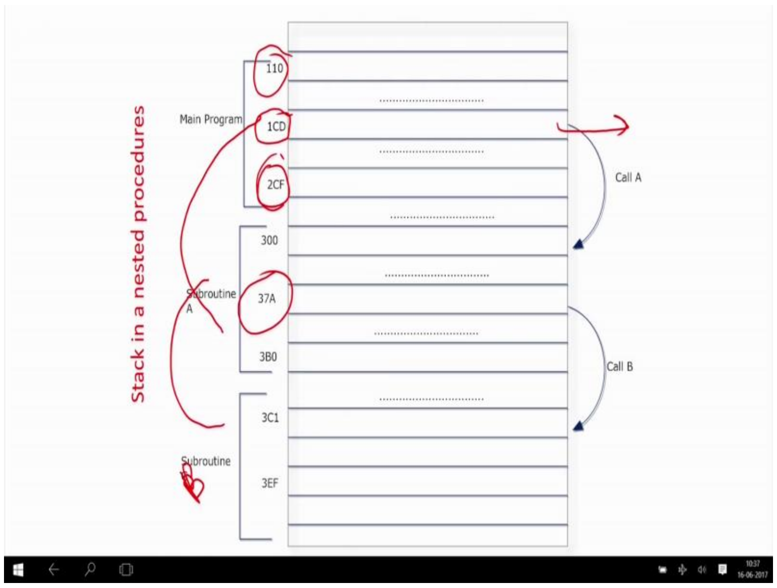
So, basically it has 3 programs one is the main program whose location is 110 to 2CF, procedure A is loaded in 300 to 3B0 and procedure B is loaded at 3C1 to 3EF. And memory location from the main memory at location 1CD which is in between this will call procedure A and in sorry main memory location call A instruction is at CD. So, in the main memory main program at 1CD in memory location call A instruction is present that is where the main program will be at instruction which is located in memory location 1CD it will be calling procedure A.

Similarly, while I am executing procedure A then this instruction which is in place of 37A as a part of procedure A it will call procedure B. That is when I mean executing the main program it will call procedure A at memory location 1CD that is the call A instruction is located in memory location 1CD when I am executing procedure A at memory location 37A the instruction call B is there and after executing call procedure B you will return back to A and then again you return back to the main program.

So, we look at details what exactly what is going to happen the stack pointer at present is F20 that is the top of the stack it is assumed. So, let us assume that the stack pointer is this at some point of time this is your stack. This is the part of the main memory which is allocated for this stack, the stack pointer value is there we are having 4 general purpose registers which will hold the values, that is your scratch pad kind of a thing which are the general purpose registers which are given to you to basically for user programming, for use in user programming. Stack pointer, program counter all are also present as a general CPU architecture, so they are all available over here.

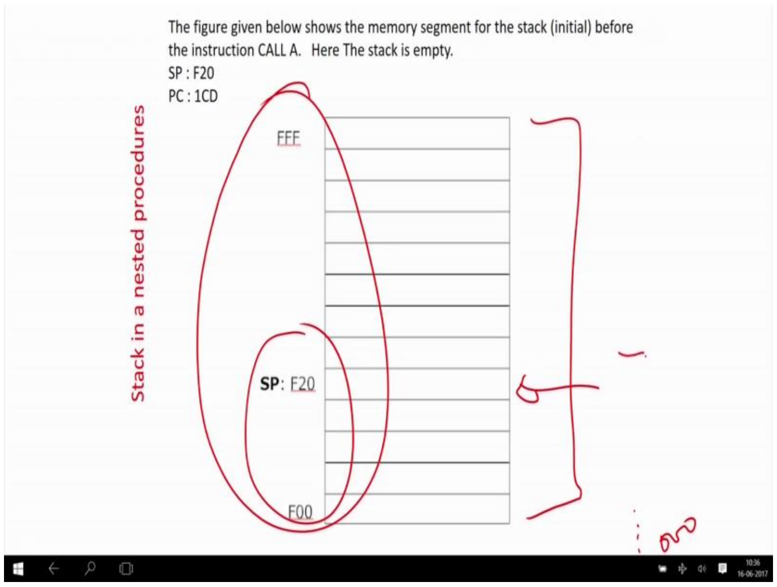
So, now we will see how the stack is implemented or how the stack is modified when such a code is executed using jump instructions in a nested procedure with the example is given below.

(Refer Slide Time: 21:32)



So, if you look at the pictorial representation we start main program at 110 end at 2CF and then if you look at it 1CB is the instruction of the main program which is calling function A subroutine A is starting at 300 ending at 3B0, at 37A location it is calling procedure B who starts at 3C1 and which ends at 3EF and after ending of the procedure again you go back. So, this is the pictorial representation of the nested call which we are going to see and this is the example of the stack.

(Refer Slide Time: 22:00)


Stack in a nested procedures

The figure given below shows the memory segment for the stack (initial) before the instruction CALL A. Here The stack is empty.

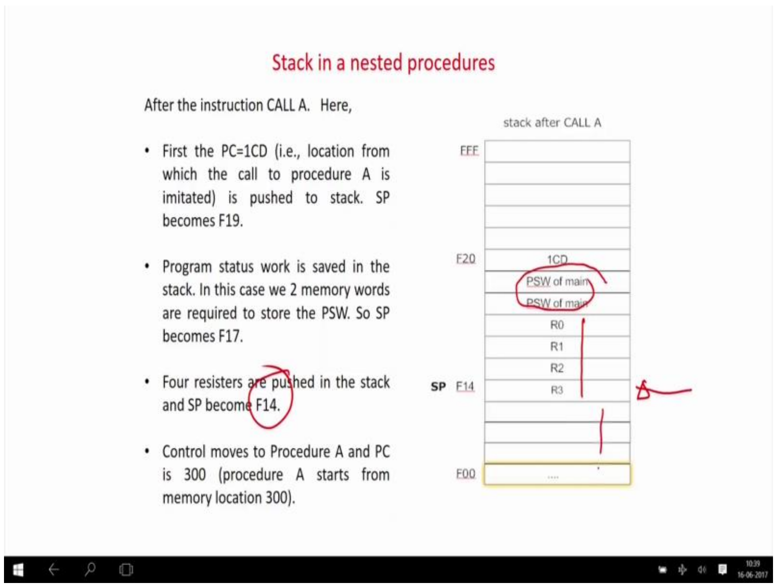
SP : F20

PC : 1CD

So, this part of the memory has been allocated for the stack. So, the memory is written in a reverse manner basically FFF to FF0 in fact, if you go … this will be 000. So, they have not written the memory in this fashion, like 000 to FFF they just illustrated in a reverse manner that is not a problem. Now just we have to think about that is the last location and the 0th location is down the line. So, and this part of the memory is allocated for your stack implementation sorry, this part of the whole part is allocated for stack implementation and at present the stack pointer is here.

There may be some other elements over here which we are not bothered for the time being, but our stack pointer is this point and, so we have to think about the stack which actually starts from here, in this range ok sorry ok.

(Refer Slide Time: 22:53)



Stack in a nested procedures

After the instruction CALL A. Here,

- First the PC=1CD (i.e., location from which the call to procedure A is imitated) is pushed to stack. SP becomes F19.
- Program status work is saved in the stack. In this case we 2 memory words are required to store the PSW. So SP becomes F17.
- Four resisters are pushed in the stack and SP become F14.
- Control moves to Procedure A and PC is 300 (procedure A starts from memory location 300).

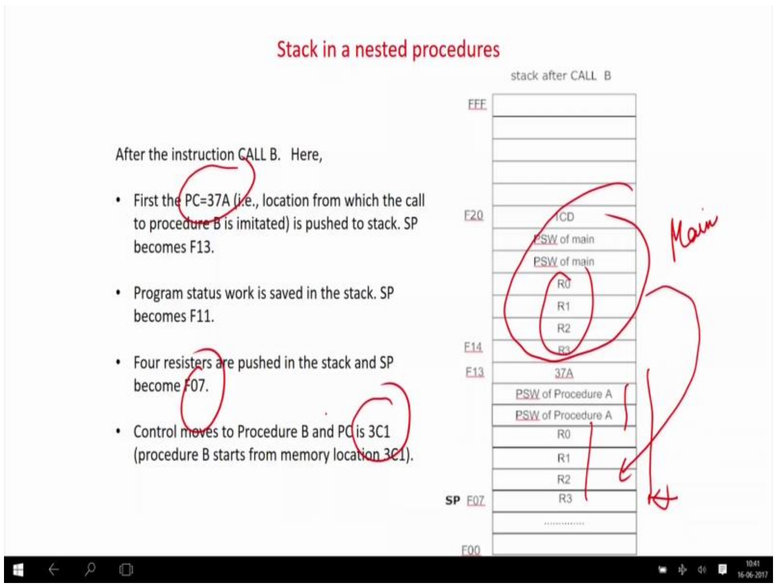
So, now we have built our context and now let us go ahead with the implementation. So, our stack was at 20 if you remember our stack was pointing over here. Now main program is calling the procedure A. So, now, we are at 13B. So, at this point the program main program is calling subroutine A which with is located at location 300. So, basically our stack pointer was here then first PC = 1CD location from which the call to procedure A is intimated. So, you see 1CD is the location from which the call is there, call has occurred. So, what you have to do first you have to store the value of 1CD in the stack this one because when I will be returning I will be returning back to this point.

When I will be returning from subroutine A to main program after of course, going for subroutine B sorry this is subroutine B this is actually subroutine B. So, it is a mistake over this it's subroutine nothing but subroutine B. So, then after executing subroutine B you are going to execute subroutine A and then after that you are going to come to the main program. So, I have to remember the value of 1CD because. After that I have to start executing from 1CE that is the next instruction. So, I have to save the value of 1CD. So, the value of 1CD is saved over here.

Next, here we are going to save the program status word of the main memory that is a lot of context, flag registers etcetera will be saved and then I will also save the value of the 4 registers. This is the context which is saved of the main program in stack before I go to memory location number 300 where the program procedure A starts, so 1 2 3 4 5 6 7. So, 7 values have been stored and now your stack pointer will be placed over here because as I told you stack pointer points to the memory location in this case F14 where the last element was pushed into the stack for the current set of codes being executed. So, this actually slide very clearly shows that when I am calling procedure A from main program the main location per when the procedure call is initiated 1CD so that is the program counter at that point it is first saved then the program status word of main memories main program is saved which is the flag registers and several other values of the intermediate variables are stored then the user registers like $R1$, $R2$,R3, R4, R0, $R1$, $R2$ and R3 are saved and then you go to jump to memory location 300 where the procedure A starts. But now the 7 elements have been saved so your stack pointer will now point to F14.

So, now whenever from A to you will be calling procedure B. So, it will start filling up from here so now look at here.

(Refer Slide Time: 25:36)



Stack in a nested procedures

After the instruction CALL B. Here,

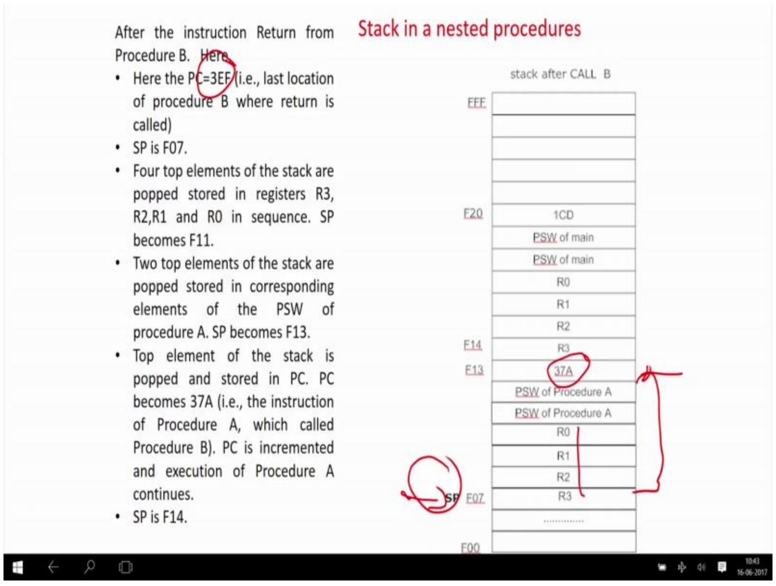
- First the PC=37A (i.e., location from which the call to procedure B is imitated) is pushed to stack. SP becomes F13.
- Program status work is saved in the stack. SP becomes F11.
- Four resisters are pushed in the stack and SP become F07.
- Control moves to Procedure B and PC is 3C1 (procedure B starts from memory location 3C1).

So, here call B. So, after this if you remember this was the part for main program. Now, from here I am going to call program B. This procedure A is going to call procedure B. So, while jumping to procedure B you have to do the same thing you have to save the value of 37A. So, 37A is nothing but the part of procedure A or the location or procedure A from where the procedure B is called. So, you have to store 37A because whenever you are completing procedure B you have to return to the point in procedure A from where procedure B was called any you have to complete A. So, you have to remember basically 37A from where procedure A called procedure B because after completing procedure B you have to return to memory location 37A and then it will start executing the next instruction that is 37B.

Of course, you have to again store all the values of the context of A user registers of procedure B because these user registers while executing the main program we are saving some values which were local to the main program. When I am going to procedure B the same set of registers will be shared within main program and procedure A, but then procedure A will again save some values which are required to be saved or called or this one which are local to A.

For example, I may use in the main program to store the value of C + B in $R0$ when I am calling procedure A then I can use the same register R0 to compute some other stuff which may be F + 3. So, the resources are same. These registers are same for procedure A, main program and procedure B. So, therefore, the local values involved in the registers has to be saved when you are leaving that procedure and calling some other procedure. Then again the program stack will again get incremented by C. So, it will come down by already saved and the 7 values. So, the now this stack pointer will be pointing over 7 and then it will move to 3C1 which is nothing, but the place where this one is called. So, I am going to 3C1. So, this is nothing, but the instruction location where procedure B starts.

(Refer Slide Time: 27:51)



Stack in a nested procedures

After the instruction Return from Procedure B. Here

- Here the PC=3EF (i.e., last location of procedure B where return is called)
- SP is F07.
- Four top elements of the stack are popped stored in registers R3, R2,R1 and R0 in sequence. SP becomes F11.
- Two top elements of the stack are popped stored in corresponding elements of the PSW of procedure A. SP becomes F13.
- Top element of the stack is popped and stored in PC. PC becomes 37A (i.e., the instruction of Procedure A, which called Procedure B). PC is incremented and execution of Procedure A continues.
- SP is F14.

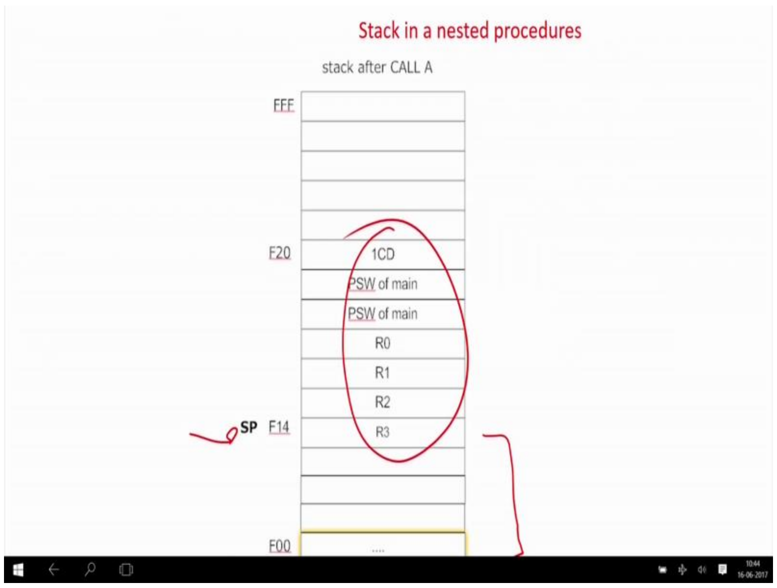
And then procedure B is done. So, then procedure B will start executing and when procedure B will be completed it has to regain back. So, after procedure B is completed what do you have to do? You have to start executing procedure A from the point which we have left. So, if you see if you remember 37A is the point where I called B from A. So, you have to when I am completing. So, my stack pointer is here. So, when I am completing procedure B see that is the 3EF which is the last location of procedure B then it is start unwinding. So, what will happen? Then procedure B is finished now it has to return. So, now, what will happen the stack will try to find out that where I have to return. So, it is nothing but procedure A where you have to return. So, how we how the whole computer will know that what I have to do, so it does not understand or does not remember in a very explicit sense that A called B, B called C not in that way. B has ended that is the last instruction of procedure B has completed now it is a return instruction. So, whenever return instruction is executed it will first look at the stack pointer. So, stack pointer will start refilling the values in the respective registers.

So, procedure B has completed return instruction is executed, now this part is invoked what it is doing stack pointer is 7. So, whatever is in this stack first stack pointer will be fed to register

3 2 1 0, PSW of A, PSW other parts will be loaded in the corresponding registers and main mem and the registers and the flag registers etcetera. So, up to this the whole context of A will be loaded into the respective registers; that will be automatically done when B will be executing the return instruction. B is executing a return instruction means the stack pointer starts incrementing one by one that is they are popping out all the elements and feeding in the respective registers.

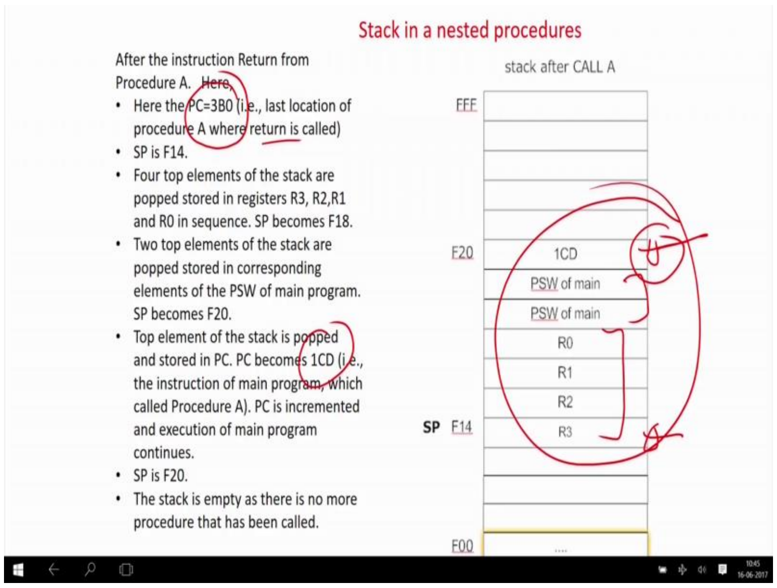
Then the program counter will be loaded at 37A that is now it will return back to A. Then at procedure A, 37A is the location where I left A to go to B then program counter will be incremented by 1. So, it will be 37B and procedure B will be executed, basically this is what is the situation.

(Refer Slide Time: 30:03)



So, in this case the whole stack which was corresponding to A is gone, it has been fed to the corresponding registers and the program status part of A, procedure A code has been started to execute from memory location 37B and A will complete. So, when procedure A is running. So, this is the part of this step which is in context the stack pointer is over here and these are the elements which corresponds to the main program, but at present we don’t require the context of the main program right now, because now procedure A is running.

(Refer Slide Time: 30:40)



Stack in a nested procedures

After the instruction Return from Procedure A. Here,

- Here the PC=3B0 (i.e., last location of procedure A where return is called)
- SP is F14.
- Four top elements of the stack are popped stored in registers R3, R2,R1 and R0 in sequence. SP becomes F18.
- Two top elements of the stack are popped stored in corresponding elements of the PSW of main program. SP becomes F20.
- Top element of the stack is popped and stored in PC. PC becomes 1CD (i.e., the instruction of main program, which called Procedure A). PC is incremented and execution of main program continues.
- SP is F20.
- The stack is empty as there is no more procedure that has been called.

Once procedure A will be completed then what is going to happen the same thing which I discussed from B to A the same thing will repeat for the case of A to main program. So, the stack pointer was at 14. So, we rem we remember 3B0 is the last location in which the procedure A was called from main location. So, you have to, from 3B0 is the last location of procedure A and where return is called. So, last location of program counter is 3B0 in procedure A when return is called. That means, at 3B0 A has stopped then when as I told you whenever a return instruction is called these $R0$ $R1$ $R2$ $R3$ will be now fed to the respective registers.

So, what will happen? So, the $R0$ $R1$ $R2$ $R4$ till now are having the values corresponding to procedure A now they will be eliminated and the values of registers corresponding to the main program will be loaded into the register. Similar thing will happen for the program status word.

Now, what is going to happen? Now 1CD will be loaded in a program counter that is very important because 1CD is if you remember 1CD is the place where the main program called the procedure A. So, now, this will be incremented it will be 1CE and the main program will start executing from 1CE and finally, the stack pointer also after start before starting the execution in the main program which called A the stack pointer will be pointing over here the whole stack will be empty, the main program will be executed and finally, the whole code will stop. So, this is what will be your stack.

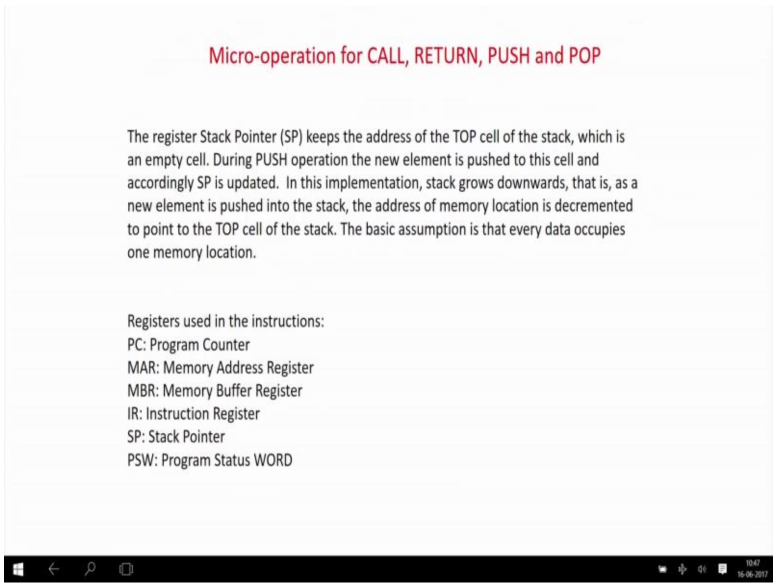
Because after I am loading all the context to main program then there is no other code which remains to be executed in the nesting. So, procedure, main program, A, B from B to A, A to main program and the whole stack is clean. So, this is a very very simple way in how a procedure is basically implemented.

So, now we are going to see a slight because we are this last lecture of this unit module sorry. So, in the next module which we will be going into more integrated way of understanding how basically the codes are executed in terms of micro instructions. So, what we are going to see now, so when you are going to say pop, when you are going to say push, when you are going to say jump, when you are going to say return, how actually the what are the set of assembly language instructions that are executed. Like push, pop, actually, involve some kind of micro instructions. Like when I say push so what is going to happen? Push means first we have to take the value of the stack pointer because it is pointing to the main memory. So, that value has to be loaded to the memory address register then the value from the memory buffer register will be loaded to the memory and then it will be again decremented.

So, because when I say pop, so what is going to happen? Again the stack pointer value has to be put to the register memory address because only a part of the main memory is given for the stack implementation. So, again when there is a pop means this stack pointer value will be loaded to the address register, then the address register will point to the means the memory address which is having the corresponding value for that stack which you I mean I want to pop. So, that value of the memory will be going to the memory data register and that will go to the instruction register and so forth. So in fact push, pop, call and return. So, you want to call then what you have to do before you have to call you have to push so many instructions into the main memory into the stack and then you have to jump unconditional to a place. So, all these push, pop, call and return will have some kind of micro instructions.

So, now we will see basically or micro operations. So, how basically cost call, return, push and pop are implemented in a micro level.

(Refer Slide Time: 34:15)

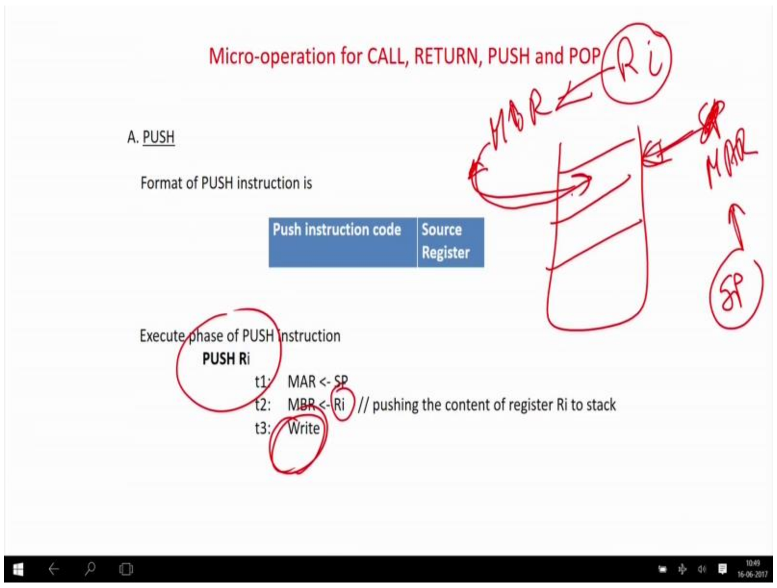


Micro-operation for CALL, RETURN, PUSH and POP

The register Stack Pointer (SP) keeps the address of the TOP cell of the stack, which is an empty cell. During PUSH operation the new element is pushed to this cell and accordingly SP is updated. In this implementation, stack grows downwards, that is, as a new element is pushed into the stack, the address of memory location is decremented to point to the TOP cell of the stack. The basic assumption is that every data occupies one memory location.

Registers used in the instructions:
PC: Program Counter
MAR: Memory Address Register
MBR: Memory Buffer Register
IR: Instruction Register
SP: Stack Pointer
PSW: Program Status WORD

(Refer Slide Time: 34:17)



Micro-operation for CALL, RETURN, PUSH and POP

A. PUSH

Format of PUSH instruction is

Push instruction code	Source Register

Execute phase of PUSH instruction

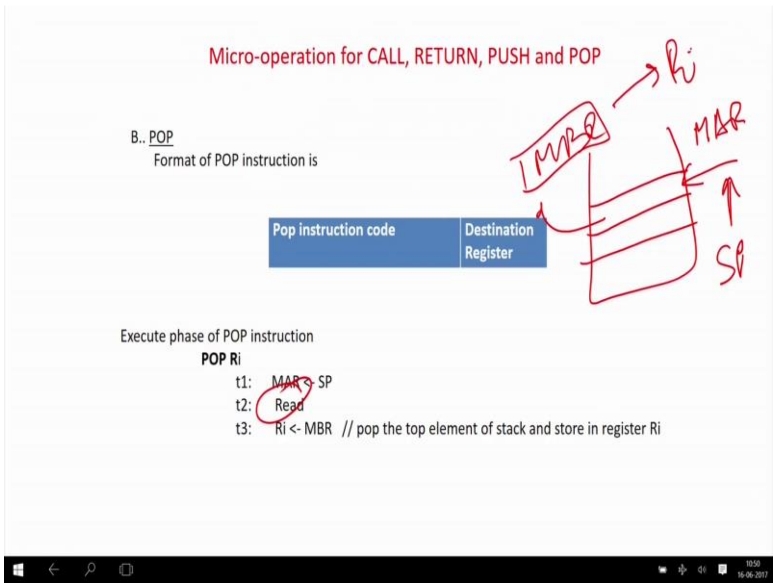
PUSH Ri

```
t1:   MAR <- SP
t2:   MBR <- Ri  // pushing the content of register Ri to stack
t3:   Write
```

So, when I say push its very simple. So, the stack pointer will be loaded to the memory address register and then, so, now, the memory address register is having the value of the point in the stack which I want to push a value. So, the register Ri will be written to the memory buffer register and finally the memory will be written it is something like this. So, this is your stack pointer. So, stack pointer basically will be pointing to the memory that is your stack. So, this one will be fed by the stack pointer. So, stack pointer is a register it will be written to the memory address register simply pointing over here then the memory the data will be going to the memory buffer register and the memory buffer register sorry this will be stack pointer the memory it’s a push, so it will be the other way around. So, the memory buffer register will be written from the Ri because I want to store the value of memory register Ri to a memory location.

And in which memory location I want to put that is the stack pointer. So, the value of stack pointer will be given to the memory register memory address register. Memory address register is pointing to the point in the stack where I want to push the value, but who writes to the memory if you remember the memory buffer register is written to the memory as well as read to the memory. So, in this case it is a write operation because you are pushing it. So, the write signal will be made high in the memory, the register Ri will be the written memory buffer register already the address register is pointing to the top of the stack and then the value of the memory buffer register will be written over here.

(Refer Slide Time: 36:01)



Micro-operation for CALL, RETURN, PUSH and POP

B.. POP

Format of POP instruction is

Pop instruction code	Destination Register

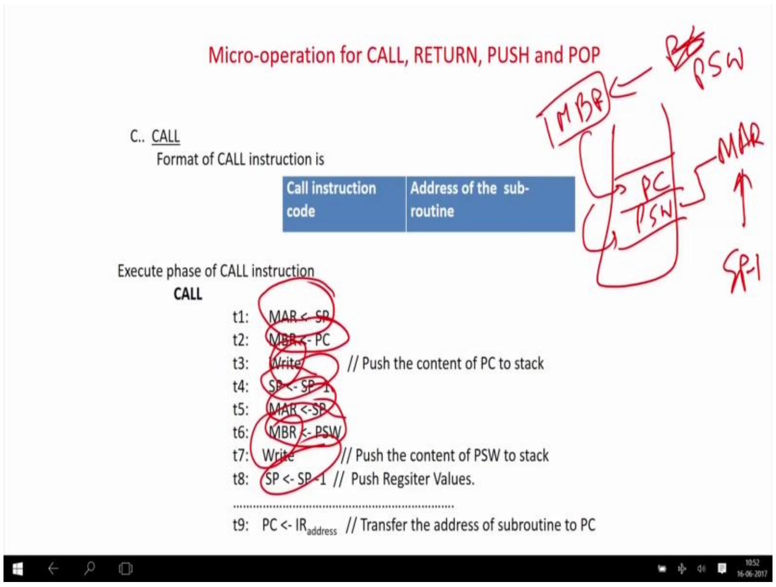
Execute phase of POP instruction

POP Ri

```
t1:   MAR <- SP
t2:   Read
t3:   Ri <- MBR  // pop the top element of stack and store in register Ri
```

In case of pop it is just the reverse same thing. So, this is your memory address register you are writing the value of the memory register from the stack pointer. So, we are pointing to the exact location of this stack from where we want to pop the value then you make a read so that it will come to nothing but the memory buffer register. So, the stack pointer will give the address to the memory address register, memory address register will point to the address location in the stack from where you want to read, you make a read signal and then your memory buffer register will be getting the value and memory buffer register will be writing to the memory location from to the register. So, Ri is going to gain back the value which you want. So, it is very simple of a push pop operation.

(Refer Slide Time: 36:42)



Micro-operation for CALL, RETURN, PUSH and POP

C.. CALL

Format of CALL instruction is

Call instruction code	Address of the sub-routine

Execute phase of CALL instruction

CALL

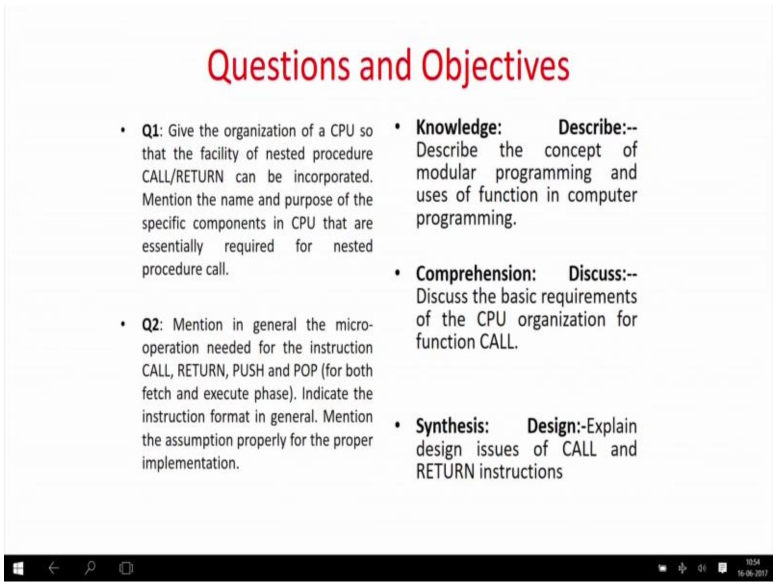
```
t1:  MAR <- SP
t2:  MBR <- PC
t3:  Write         // Push the content of PC to stack
t4:  SP <- SP-1
t5:  MAR <-SP
t6:  MBR <- PSW
t7:  Write         // Push the content of PSW to stack
t8:  SP <- SP-1  //  Push Regsiter Values.
.......................................................................
t9:  PC <- IR_address  // Transfer the address of subroutine to PC
```

Now, call, so call is slightly complex means before you call there are so many stuff that has to be done. So, say I want to do a call location call. So, what I will have to do? So, call is nothing but a jump instruction. So, before doing it we have to push all the elements to the stack like the program status word, where you have to return, the context and then you go for a unconditional jump. So, what you have to do? So, this is your stack. So, whenever you want to do any instruction like return, call, push and pop you have to first load the value of the program, you have to first load the value of the memory address register from the stack pointer. You know stack pointer is a temporary register sorry the register whose which contains the value of the top element of the stack which can be pushed and popped right.

So, memory address register is going to get the value of SP that is stack pointer, memory buffer register you store the value of PC. So, in this case there is a memory buffer register you have to put the value of PC and then you say write. So, if you write the memory PC will be saved over here. Then you make $SP = SP - 1$ then you again load the memory address register, so now, your memory address register will be pointing to the next element of the stack because this one you have stored PC now you decrement $SP = SP - 1$ give the value of the memory register and memory buffer register now you put your program status word. So, now, you will have program status word.

And you keep on doing it and then you write it then again you decrement the PC you keep on doing it till you want to save all the elements. So, what I am doing? I am taking the SP writing to the memory address register and then in the memory buffer register I am putting whatever I want to save. PC, program status word, all registers values I will write in this manner and then finally, I will give PC I will load with the address where I want to jump because now my PC has already been saved. So, I can rewrite the PC to the jump instruction. So, in this way we will jump to the respective location.

(Refer Slide Time: 38:47)



Questions and Objectives

- **Q1**: Give the organization of a CPU so that the facility of nested procedure CALL/RETURN can be incorporated. Mention the name and purpose of the specific components in CPU that are essentially required for nested procedure call.
- **Q2**: Mention in general the micro-operation needed for the instruction CALL, RETURN, PUSH and POP (for both fetch and execute phase). Indicate the instruction format in general. Mention the assumption properly for the proper implementation.

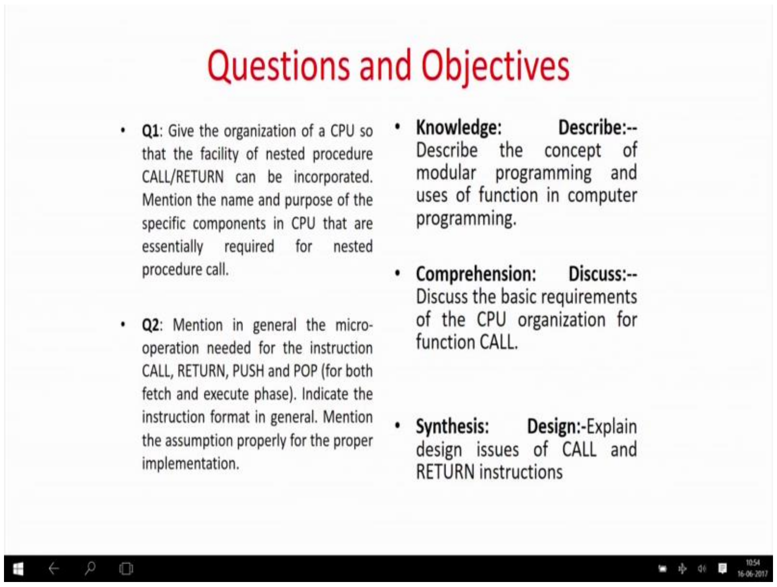
- **Knowledge: Describe:--** Describe the concept of modular programming and uses of function in computer programming.
- **Comprehension: Discuss:--** Discuss the basic requirements of the CPU organization for function CALL.
- **Synthesis: Design:-**Explain design issues of CALL and RETURN instructions

Return will happen, reverse will happen when you want to return back. So, what I want to do? Again memory addresses register this is the fact. So, this means de facto. So, whenever you want to do any kind of an operation of call, return, push and pop returning to this function call stack pointer will be writing to the memory address register then you will read it. So, again you will read the value of memory buffer register to register 1 and you will keep on doing it. So, what I am doing? Memory this is the stack pointer is now pointing to this memory location, now this is a memory address register this is the stack pointer this is default.

So, now whatever in this case is now in the memory buffer register, now I will first write to $R1$ $R2$ $R3$ $R4$ there is all the in this case it will be a read will be there. So, you will keep on reading the values that means the read operation. So, first from the memory buffer register you will read $R1$ $R2$ $R3$ then we will be reading the value of program status word, then we will be reading the value of program counter and so forth. You will keep on doing it from everywhere you will do register $R1$ you will read, then increment the value of program SP by 1. So, now, it will be your new SP you will read $R2$, $R3, R4$ program status word and finally, you will read the value of the program counter.

If you remember I have saved the program counter then I have saved all the programs status word and the registers. While I am popping I am reading $R1$ $R2$ $R3$ $R4$ program status word and finally, I am reading the program counter. So, when you are returning the program counter; obviously, you have returned from where you have left. So, basically these two slides are showing the micro level instructions or the micro level operations required to implement a call, return, push and pop. Very simple first you have to give the value of stack pointer to the memory address register, then either do a read or write and you have to keep on doing it till you have fetched all the or pushed or popped all the required elements. When you are calling any function you have to change the value of program counter to jump instruction and when you are returning you have to write the program counter from the stack because this stack will be remembering the value of the program counter where I have to return.

(Refer Slide Time: 40:59)



Questions and Objectives

- **Q1**: Give the organization of a CPU so that the facility of nested procedure CALL/RETURN can be incorporated. Mention the name and purpose of the specific components in CPU that are essentially required for nested procedure call.
- **Q2**: Mention in general the micro-operation needed for the instruction CALL, RETURN, PUSH and POP (for both fetch and execute phase). Indicate the instruction format in general. Mention the assumption properly for the proper implementation.

- **Knowledge: Describe:--** Describe the concept of modular programming and uses of function in computer programming.
- **Comprehension: Discuss:--** Discuss the basic requirements of the CPU organization for function CALL.
- **Synthesis: Design:-**Explain design issues of CALL and RETURN instructions

So, this basically completes this module as well as this unit. Where we have started from what is a very basic CPU architecture, then how instructions are designed and finally, we have ended to a very very complex set of instructions which are involving procedure call, return calls and what do I say that is your conditional jumps etcetera.

So, in the next module will be trying to look into more internal details of a CPU architecture and how these basic instructions are executed in terms of hardware signals. So, before we close down let us see one or two simple questions. So, given the organization of a CPU how are a nested procedure how call and return can be incorporated. Mention the name and purpose of several of the hardware elements required to do this. So, if you are able to answer this question it will means suffice for the objectives like describe the concept of modular programming, discuss the basic requirements of CPU design and also discuss in details about the return and issues of call and return instructions because when you will be solving this problem you will be able to justify these objectives.

Second similar question can be mention the general micro operations needed for the push and pop, return and call. So, as we have last discussed what are the basic micro level instructions required for a proper procedure call and return. So, this actually directly satisfies the objective of synthesis design that is explain the issues of design and call return instructions that how internally they are designed and implemented.

So, with this we come to the end of this module as this lecture as well as module. So, in the next we will be looking at in a newer module which will be looking into more details on the hardware aspects of execution of the instructions.

Thank you.