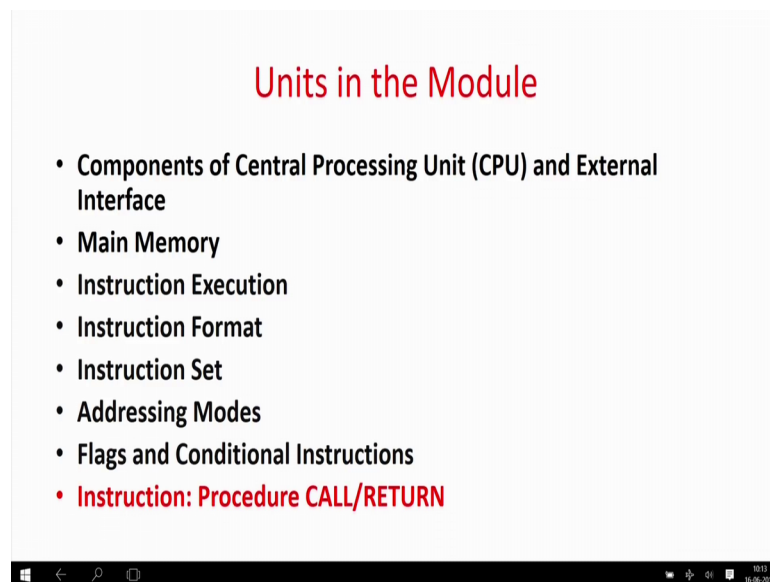


**Computer Organization and Architecture A Pedagogical Aspect**  
**Prof. Jindra Kr. Deka**  
**Dr. Santhosh Biswas**  
**Dr. Arnab Sarkar**  
**Department of Computer Science & Engineering**  
**Indian Institute of Technology, Guwahati**

**Lecture – 14**  
**Instruction: Procedure CALL/RETURN**

So, welcome to the last unit of the module on addressing mode instruction set and instruction execution flow.

(Refer Slide Time: 00:36)



So, throughout this module in the last 7 units basically we have covered mainly different types of instructions, their format, their operand types, what are the addressing modes and finally, in the last unit we covered a very special type of instructions which are actually called conditional instructions. That is which are not a very sequential flow, but depending on some is conditions of variables like 0 flag or some other conditions like equality, parity, etcetera we can jump to a required memory location to execute the instruction from that part or if the condition is true we can just go ahead to the next instruction.

And then we had seen that flags play a very very important role in control instructions flags are some of the registered bits which are set or reset depending on the conditions of some mathematical or logical expression. Like for example, if you add two numbers then if the answer is 0 then a 0th flag is set then you can use a conditional instruction called jump on 0 using that flag bit.

Now, today the last unit that is we are going to use such conditional instructions in a very very practical application which is called the presidio return and code. So, if all of us we have written some C or C plus plus high level language codes and we know that functions or procedures are a very very important part and parcel of all programming language because you can module arise your code.

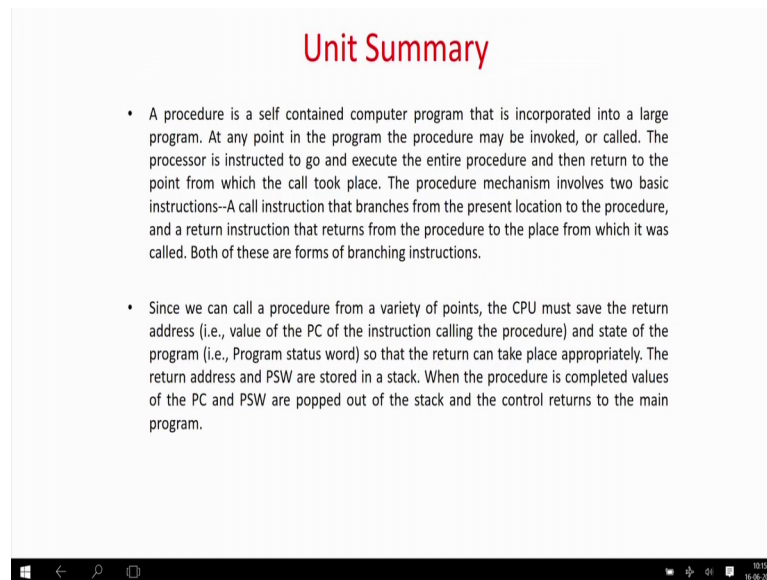
But what happens is that whenever you are writing running a main code from there you have to call to a subroutine or a function and then from that function you can call another function and it can go on. But what are the issues there the first very simple issue is that you have to know at which location I am going to jump. So, there can be there should be a jump instruction. So, it can be conditional or it can be unconditional.

Conditional means if it is based on some condition if you want to call the instruction then it will be a conditional jump for procedural call, but in most general cases which generally just call the function from certain part of the code. In that case it will be just a function call which would be an unconditional jump.

But then whenever you are leaving your main program then; obviously, you want after executing the procedure you have to come back to the main program and start executing from that point. So, you have to save the context of the program when you are (Refer Time: 02:36) jumping to a sub routine. So, all those issues we will be covering today in this unit.

So, this is the last unit of the module which is on procedure call and return.

(Refer Slide Time: 02:46)



### Unit Summary

- A procedure is a self contained computer program that is incorporated into a large program. At any point in the program the procedure may be invoked, or called. The processor is instructed to go and execute the entire procedure and then return to the point from which the call took place. The procedure mechanism involves two basic instructions--A call instruction that branches from the present location to the procedure, and a return instruction that returns from the procedure to the place from which it was called. Both of these are forms of branching instructions.
- Since we can call a procedure from a variety of points, the CPU must save the return address (i.e., value of the PC of the instruction calling the procedure) and state of the program (i.e., Program status word) so that the return can take place appropriately. The return address and PSW are stored in a stack. When the procedure is completed values of the PC and PSW are popped out of the stack and the control returns to the main program.

So, as a pinnacle pedagogical format so what we are going to study, what is the unit summary. So, basically as we have told that procedure is a self contained computer program which is a part of a larger program. So, whenever a procedure is called basically it invokes a jump to the memory location or when the first instruction of the procedure is placed. So, it is a some form of a unconditional jump.

But sometimes you can also have a procedure whose call may be depending on some condition. In that case you have to call the procedure based on some conditional instruction. But in a very general sense we generally have a unconditional jump or unconditional call to the procedure.

Then again as we are jumping to a new procedure, so we will be reusing all the temporal registers the accumulators, the program counter, program counter actually gets changed from the current programming location to the location of the first instruction of the procedure, but again when you want to return back you have to again come back to the point where I have left the main program you may also regain the values of the registers where there are some temporary variables which I were working which I was working on and so forth.

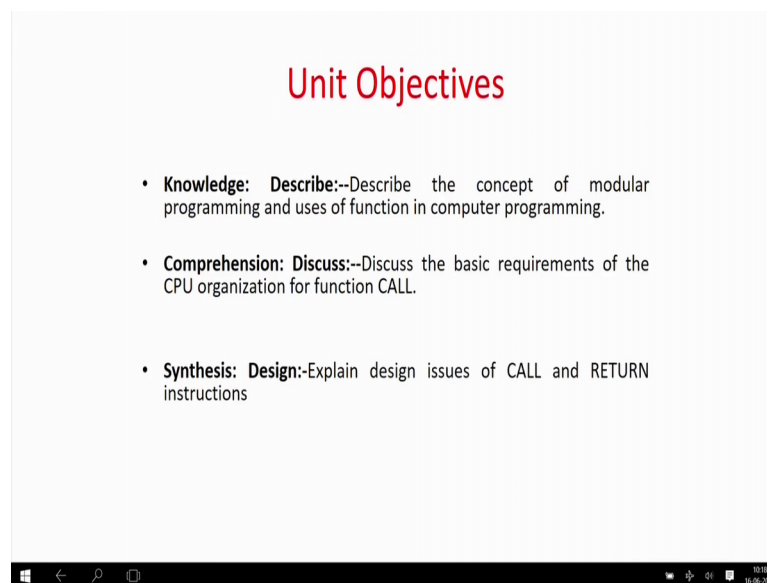
So, in other words the whole context of the program has to be saved in a stack. So, very very important components of this stack are basically we call it as the program status word which will have all the values of the flags, which will have the value of the

program counter or in other words the context of the current program has to be stored in a stack when you are calling a procedure. And when we are returning to the returning from the procedure all the values of the context like the value of the program counter, the value of the registers etcetera are fed back to the corresponding registers and the program and the program counter so that you gain the real context for what I have left in the main program and you can start re executing.

And if they are nested procedure as we will see in an example, so procedure a will call procedure b, procedure b will call procedure c and at every call the context of the calling procedure will be stored in a stack and when you are returning unwinding the procedure calls you are going to get back one context after the other from the stack that we are going to see.

So, they are two very important concept jump and before you jump you store everything like the return address the program status word, the value of the registers etcetera or the temporary variables or the temporary context of the procedure or the main function which calls another procedure are saved in a stack. And when you have returned back after completing the called the procedure they have you regain back the values and you can start executing.

(Refer Slide Time: 05:14)



**Unit Objectives**

- **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

10:10  
14-06-2017

So, this is if any two basic concepts you are going to deal in detail today and then what is the unit objectives. There are 3 basic objectives that is knowledge, in which you are

(Refer Time: 05:24) to describe the concept of modular programming which uses a procedure of a function. Then you will be able to comprehend and discuss the basic requirements of CPU organization for a call function, that what are the registers required what is a stack required etcetera.

And as the synthesis objective you will be able to design explain the design issues of written and call instructions, that how can you design the call instructions, how can you design return instructions, what are the issues, which are the registers involved, what are the stack involved etcetera, what are the hardware of the CPU involved to design in call and return instruction. So, these are the basic objectives which you are going to fulfill after running this unit.

(Refer Slide Time: 06:00)

**Subroutine**

```
main() {
    int a, b;
    a = 5;
    b = sqr(a);
}

/* subroutine of squaring a given number */
int sqr(int val) {
    int sqval;
    sqval = val * val;
    return sqval;
}
```

```
MOV R1, a // Move data
present at memory location for variable
a to R1
JMPS sqr // jump to
subroutine whose address is sqr

MOV b, R2 // Move data present at
R2 to memory location for variable b

sqr: //Subroutine for squaring
starting at memory location sqr

MUL R1, R1 //Squaring the value
at R1 and storing at R1

MOV R2, R1 // Move data present at
R1 to R2

RTS // Return to the main program and
execute the next instruction "MOV b,
R2"
```

*Handwritten red annotations:  $b = a^2$  and  $a^2$*

So, as we are all very much familiar with C programming. So, we are going to take a very simple C program with a subroutine and see how it reflects in a assembly language. So, if int, int a b, a equal to 5, b is equal to square of R. So, this is a function in which you are making a square then in this case actually this b equal to a square.

So, what is the function? So, you are calling this the function. So, you are passing the value of a then in square then square value equal to val into val, val is passed over here. So, a is passed is the function a. So, we are going to get the value of a square because the variable value as the value of a which was passed as a parameter and then you are returning square of val that is the square val is nothing but value of the square of val and

that is nothing, but in this case a square and that is going to be returned to b and you are going to get b equal to a square, very very simple C procedure.

Now, if you look at it how the subroutine would look like. So, first we are taking a value of the memory location a into register 1, then it is a. So, now, register one has the value of a. So, now, what you are doing? You are making a jump unconditional to a procedure whose name is square. So, again this square here is a subroutine, but as we are discussing yesterday as a part and parcel of every jump instruction is something called label. Label means nothing is the name given to an instruction when and when the code is loaded into a memory it is going to be replaced with the exact memory location of the instruction which you are going to call. So, in this case square root is the label here and it is nothing, but it is the pointing to (Refer Time: 07:37) instruction or it is the name of the instruction.

So, jump is an unconditional jump subroutine it will just go over here. Square root means the name of the instruction or it corresponds to the memory location of the instruction when this instruction is stored. So, it is MUL R 1 into R 1. So, already have the value of a into R 1. So, you are going to give the value of R 1. So, it is nothing, but it will be R 1 square R 1 into R 1 it will be stored into R 1. Then R 1 is going to have the value of a is basically a square now you are going to move the value of R 1 to R 2. So, that we now are two as the value of a square and then you are saying that I am going to return.

(Refer Slide Time: 08:13)

**Subroutine**

```
main() {
    int a, b;
    a = 5;
    b=sqr(a);
}

/* subroutine of squaring a given number */
int sqr(int val) {
    int sqval;
    sqval = val * val;
    return sqval;
}
```

Assembly instructions and annotations:

- MOV R1, a // Move data present at memory location for variable a to R1
- JMS sqr // jump to subroutine whose address is sqr
- MOV b, R2 // Move data present at R2 to memory location for variable b
- sqr: //Subroutine for squaring starting at memory location sqr
- MUL R1, R1 //Squaring the value at R1 and storing at R1
- MOV R2,R1 // Move data present at R1 to R2
- RTS // Return to the main program and execute the next instruction "MOV b,

Handwritten annotations: 'PC 5' with an arrow pointing to the JMS instruction; '5' in a circle; 'a 2' and 'PC 10' with arrows pointing to the MUL instruction.

So, when I am executing the return statement what happens? So, when I am calling this one you have to store all the value of the temporary variables temporary registers as well as most importantly you have to store the value of the memory location for the jump instruction because when I am going to return from here I have to go back here and then I will do, what I will do is that I have come back with it from the point which I have left increment the value of PC and you come over here in other words whenever you are jumping from main procedure program to procedure you save the value of the program counter of this state at this part.

Then now this is the may be the program counter say 5. So, 5 is stored in a stack. Here program counter may become 20 because 20 may be our way assuming that the memory location starting address of the subroutine. So, 20 21 22 it will go on and whenever it is going for the return instruction the value of 5 will be popped from the stack so that you can know that now the program counter value is 5 which is the place which I had left.

Then it will become 6 and you are going to execute instruction in which is called move R 2 to b. So, in this case the value of R 2 will be moved over here and if you look at the program carefully R 2 is nothing but it is basically R 1 square that is nothing, but a square. So, m square will be transferred to the value a sorry the R this one this one instruction will be transferring the value of a square to b which is actually the requirement of the program.

So in fact, this is some background information we have given you which is required to understand and how a subroutine is converted into an assembly language code.

(Refer Slide Time: 09:44)

### CPU components for procedure call

- CALL/RETURN is used to execute subroutines of the main program.
- When a subroutine is to be executed, the main program is stalled and the subroutine is to be executed. After the subroutine is completed, the control returns to the main program at the point where it was stalled and the execution of the main program continues.
- If the subroutine calls another subroutine which calls another subroutine and so on, the call and return process is executed in a nested manner.
- When the control moves to the subroutine the state of the main program is to be saved so that when control returns the main program it should be able to execute without loss of data. So prior to the starting of a subroutine, PC (Program Counter), PSW (Program Status Word) register variables etc. are saved and after the return they are retrieved back.

Then now we are going to see basically one of the components required or what is the hardware required in the pros in this as CPU for a procedure call. So, as I told you when a subroutine is executed the main program is stalled and the subroutine is executed when the subroutine is completed it returns to the main program.

So, there is some, program counter will play a very very important role over here and the value of program counter when calling a subroutine has to be stored if the subroutine calls another subroutine in a nested manner then actually the storage of the program status were the storage or the storage of the registers the storage of PC will be in a recursive manner as a for recursive simple recursive function call in a stack. So, when a calls b, b calls c c calls d. So, you put all the context in as register in a stack and whenever you are finishing one supporting after another you are popping up the corresponding program status word registers PC in a return procedure return manner.

So, basically that is what is being said when we move from one subroutine to another you are saving program counter program status for register variables etcetera and when you are popping back. So, it is when you are returning from one procedure to another in a nested manner or returning back.

So, one after another the context will be fetched from this stack. So in fact, you require basically a program counter memory everything is required and in addition you require a stack which is holding all the components or the context of the programs or the



subroutines when a call is made for a month's are put into another a stack is a very very important component of a procedure call, if you look at the hardware terms.

(Refer Slide Time: 11:19)

**CPU components for procedure call**

- The PC (Program Counter), PSW (Program Status Word) register variables etc. are stored using a stack.
- A group of main memory cells are used to implement the stack. The address of the last filled cell (i.e., top element) will be saved in a Stack Pointer (SP) register. When a subroutine is called, the contents of the calling program's register files (i.e., PC, PSW, and variables) are copied into the stack. After the subroutine returns to the calling program the contents of the stack are popped.

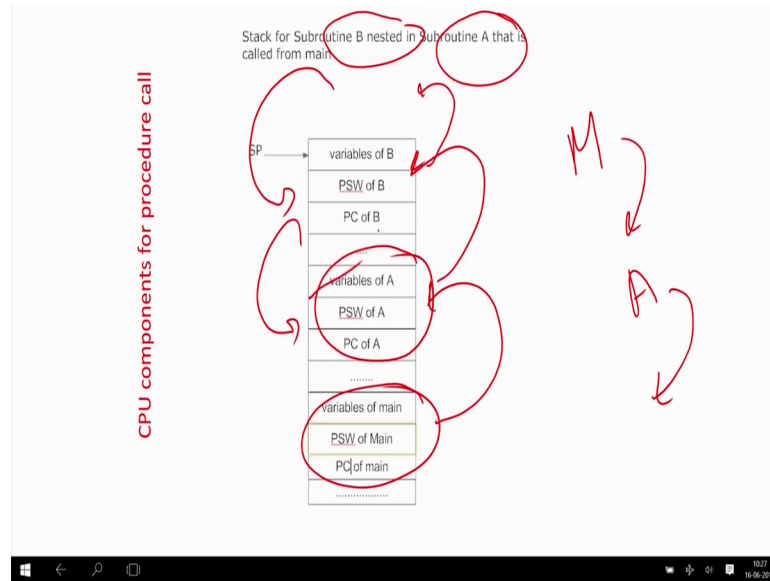
The slide features a title 'CPU components for procedure call' in red. Two bullet points describe how the PC, PSW, and register variables are stored in a stack, and how the stack is implemented using main memory cells with a Stack Pointer (SP) register. Red circles are drawn around the terms 'PC', 'PSW', 'Stack Pointer (SP)', and 'top element' in the text.

So, as I told you, so program counter program status word and register variables are very very important components which are told in a stack which defines the current context of a code. In fact, the flag registers also saved basically because when you are running a current set of instructions the flags are set or reset which are also used for some conditional statement when you go to a subroutine the same program that is their flag registers will be used, but now it will be used in the context of that procedure.

So, in that case you have again same back the value of the flag resistor and you have to again when you come back from the procedure to the main code again you have to regain then because they are shared resources this program status word, program counter, variables, registers they are actually shared components of the CPU among functions and main function and several procedures. So, you have to save the value if you are moving from one procedure to another.

Now, where is the stack implemented? So, the group of main memory is used to implement the stack, that is very important you can save a particular part of the main memory to implement the step and there is a special stack pointer which will actually locate that with the top element of the stack is. So, basically in fact, a part of the main memory is reserved for this stack.

(Refer Slide Time: 12:31)



So, something like this is a pictorial representation. So, it says that subroutine B is nested in subroutine A so in fact, what it says there is a main program then it will call program A and it will call program B. So, in this case if you see when the main program is calling procedure A then what happens the variables of main will be saved program status of word of main will be saved the PC of main will be saved and other temporary flag registers of the main memory will be saved and so forth.

Then actually from here you are going to go back to the procedure A. From procedure A procedure B will be called. So, whenever you are going to proceed your A to procedure B of course, you have to store the same many elements as of the main program when you are calling procedure A from procedure B.

So, you are saving the value of variables of A, program status of word of A, registers of A, program counter of A, flags of A in this stack then you are calling B. So, now, from B you may go to C and so forth and then afterwards what happens when you return from that you regain the value of PC you regain the value of program status word you regain the variables and complete the execution of it. Once it is that you will return from B to A.

Similarly now again you get back the value of PC of A where you have left from while calling A to B then you again reached we start from the point where you have left in B. So, before that you regain the values of variables of A, program status of A etcetera and the PC will be loaded at a point from where you have left A to execute B after

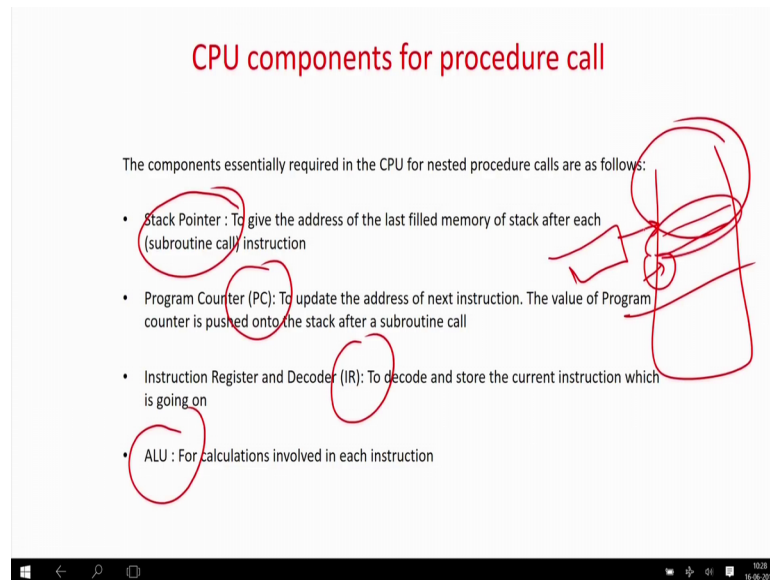
completing execution of A you do the same thing for the main program and finally, the whole code stops the execution. So, what I was telling you in of a nested manner in a stack is represented in a nice pictorial manner in this slide.

(Refer Slide Time: 14:12)

### CPU components for procedure call

The components essentially required in the CPU for nested procedure calls are as follows:

- Stack Pointer : To give the address of the last filled memory of stack after each (subroutine call) instruction
- Program Counter (PC): To update the address of next instruction. The value of Program counter is pushed onto the stack after a subroutine call
- Instruction Register and Decoder (IR): To decode and store the current instruction which is going on
- ALU : For calculations involved in each instruction



So, important components of a procedural call everything program counter, instruction decoder, arithmetic and logic units, and very important is a stack pointer and a stack. In fact, the main memory is actually the part of implementing the stack and stack pointer is a very very important it can be actually nothing but another variable or a register of the stack pointer which will actually contain the address of the main memory well which is the top of the stack.

So, the whole main memory actually is there. You are allocating a part of the main memory first stack. So, maybe you can say that this is my stack pointer. So, there will be a special register which will hold the value of the top of this stack because after adding some more elements it will come over here. So, every time you recollect which is the top of the stack.

So, whenever you want to pop some element from the stack. So, you will read the variable or the register which is having the stack pointer you will load the value of the stack point of the memory address register and then you can easily start popping up the values.

(Refer Slide Time: 15:13)

The components essentially required in the CPU for nested procedure calls are as follows:

- Memory Address Register (MAR) and address bus: To transfer the address from PC or IR and back through address bus and access that address location in main memory.
- Memory Data Register (MDR) and data bus : To transfer the data from main memory location to CPU
- System Bus: For all interconnections between the CPU components.
- Registers : To store data when it is being processed in CPU
- The figure below illustrates the CPU architecture for nested system call.

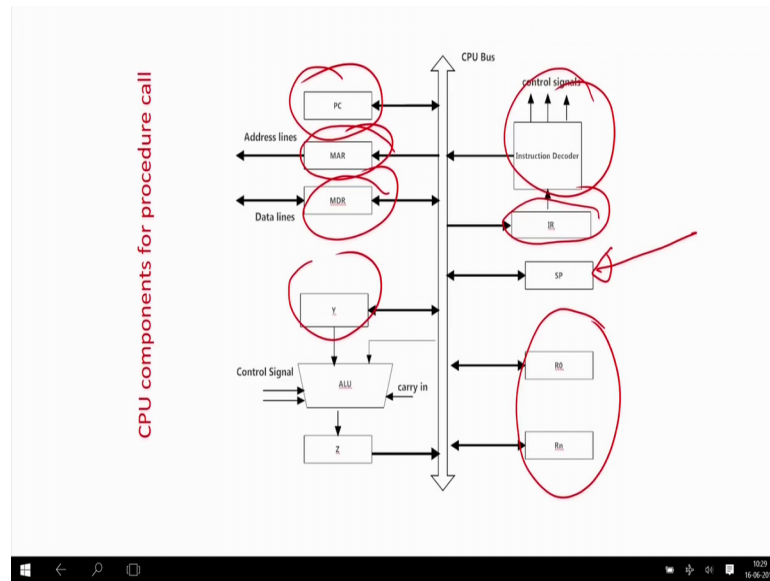
The slide features a red handwritten circle around the text 'Memory Address Register (MAR) and address bus' and another red handwritten circle around 'Memory Data Register (MDR) and data bus'. The slide is presented in a window with a Windows taskbar at the bottom showing the date 16-02-2017.

Obviously as I told you and the stack is implemented in the main memory. So, main memory is a very very important component of the function procedure card, of course memory address register will be required because you are going to pop up the elements. Where is the value of the memory register memory address register in the case of stack? It is in the stack register a stack pointer. So, stack pointer is nothing, but it points to the name, it points to the top of the top element of the stack.

So, anyone to face the element from the top element of the stack what you have to do; you have to get the value of stack pointer to memory address register and then you can fetch the value in the memory data register. But importantly stack I need a variable for a register to implement the stack pointer it is nothing, but the address of the main memory which is stored in a register and it corresponds to the address of the main memory where the last element which was inserted in this stack is present then of course, system bus registers everything is involved in the implementation of a procedure call.

So, this is nothing, but your whole how was CPU looks like only the CPU bus. So, therefore, you can see there is an instruction register, there is decoder, the instruction register there is all the user registers program counter, memory address register, memory data register that is the memory buffer register arithmetic logic unit you can call it as the accumulator. So, all the components which we have talked till now are all available.

(Refer Slide Time: 16:18)



Importantly I want to point out is the stack pointer. So, stack pointer here is nothing, but a special register which is going to hold the address of the ready main memory for the last element that was whose in the stack is present.

So, whenever you want to pop a element from the stack what you have to do you have to just SP will put the value in the memory address register. So, it will say that I have to pop one element so SP will be given to the memory address register that value upon the memory will be fetched and it will be fed back to the memory data register and from the memory data register instead it can go to the instruction register sorry instruction register it will go to the instruction decoding register instructions will be decoded and the code will be executed.

So, this is overall the basic comp, CPU components which are involved in a procedure called all the components are quite similar to all other functions and for 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 (Refer Time: 17:32) this is theory of a procedure code it is basically only 3 very simple steps, you call a procedure, before calling a procedure figure everything in this stack, call the procedure jumble an 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 form 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 use of the programs status what the program counter variables which you have saved before going to this one.

So, same call, come call, that call means call save, the context call the procedure go to the procedure and complete the procedure return back to the main part of the program from any procedure of score and before returning back we have to regain what you have seen. So, they are 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 very simple example.

So, we have a processor whose memory address range is 000 to FFF. So, all these are address bus will be one twelve bits the memory contains and as I told you. So, this is your main memory. So, it will have 000 to say FFF and as a part of this result for this stack. So, what is that for the stack if you see FF0 to FFF maybe some from here. So,

you know, this part of the program memory is allocated for a stack when your stack will be saved and it is assumed that this stacks goes downwards; that means, you will be going downwards as a stack implementation. 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.

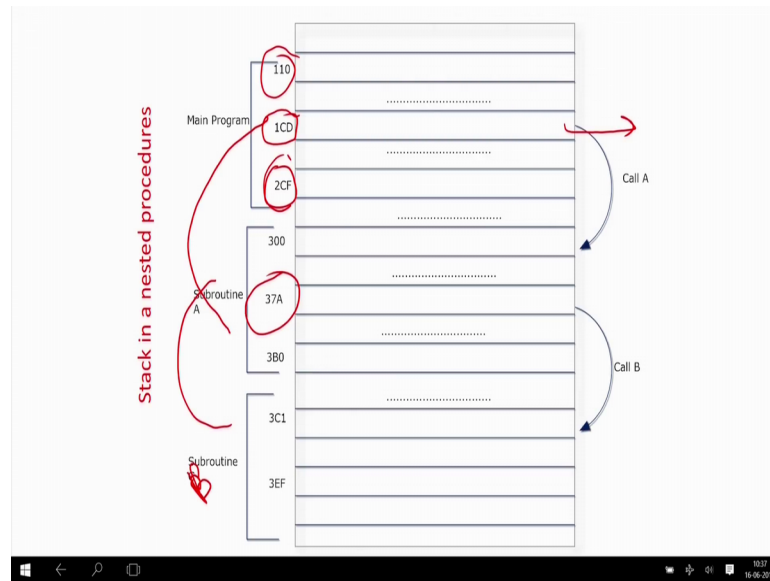
So, basically it has 3 programs one is the main program which location is 110 to 2CF, procedure A is loaded in 300 to 3B0 and procedure B is loaded at 3C1 to 3EF. And memory location from the main memory at location 1CD which is in between this will call procedure A and in sorry main memory location 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 when 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 call A, sorry 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 exhibiting procedure in at memory location 37A the instruction call B is there and after executing call procedure B you will return back to A and then you can 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 using user programming. Stack pointer, program counter all are also present as a general CPU architecture, so they are all available over here.

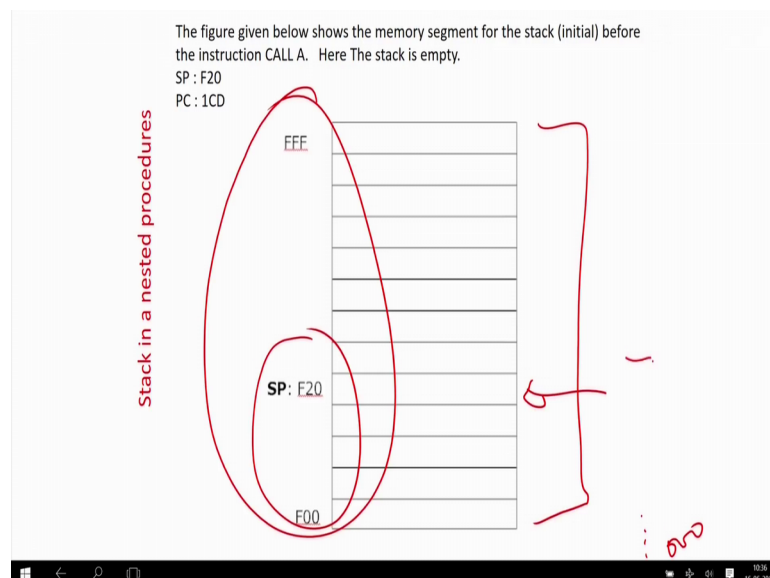
So, now we will see how this stack is implemented or how the stack is modified when such a code is executed using jump instructions in a nested procedure with the example is given below.

(Refer Slide Time: 21:32)



So, if you look at the pictorial representation we start main program at 110 in that 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 you are going to see and this is the example of this stack.

(Refer Slide Time: 22:00)





So, this part of the memory has been allocated for this stack. So, the memory is written in a reverse manner basically FFF to FF0 in fact, if you go dot dot dot this will be 000. So, we have they have not written the memory in this fashion, like 000 to FFF there 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 sorry.

(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 initiated) 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 registers 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).

stack after CALL A

|        |             |
|--------|-------------|
| EEE    |             |
|        |             |
|        |             |
|        |             |
|        | 1CD         |
| E20    | PSW of main |
|        | PSW of main |
|        | R0          |
|        | R1          |
|        | R2          |
| SP F14 | R3          |
|        |             |
|        |             |
|        |             |
| F00    | .....       |

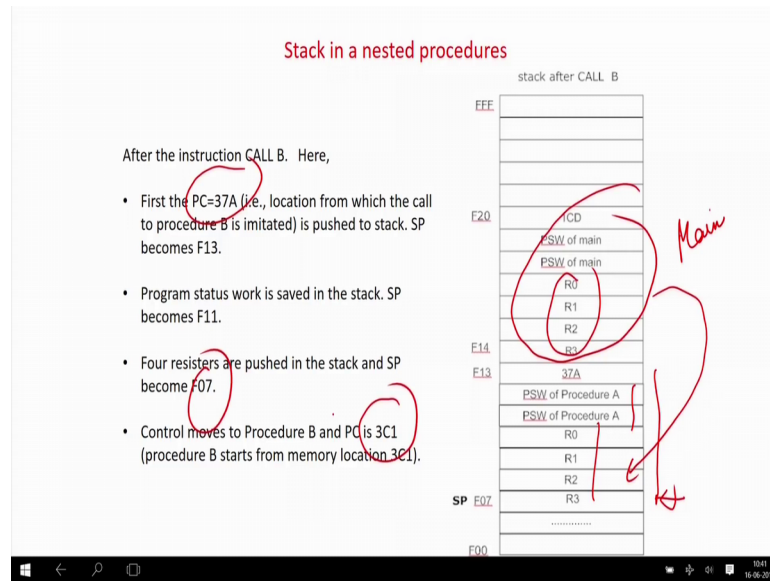
So, now we have built our context and now let us go ahead with the implementation. So, our stack was that 20 if you remember our stack was pointing over here. Now main program is calling the procedure A. So, now, we are at 13B. So, at this point the program main program is calling subroutine A which with is located at location 300. So, basically our stack pointer was here then first PC is equal to 1CD location from which the call to procedure A is initiated. 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 this 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 is subroutine nothing, but subroutine B. So, when 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 control 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 (Refer Time: 24:41) I told you stack pointer points to the memory location in this case F14 where the last element was pushed into this stack for the current set of course, 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 R 1, R 2 ,R 3, R 4, R 0, R 1, R 2 and R 3 are saved and then you go to jump to memory location 300 well they 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 cleaning up from here tonight only now look at here.

(Refer Slide Time: 25:36)



So, here call B program. 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 the procedure will be 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 be completing procedure B you have to return to the point in procedure A from where procedure B was called any of the completes A. So, you have to remember basically 37 in from when procedure A called procedure B because after computing procedure B you have to return to memory location 37A and then you it will start executing the next instruction that is 37B.

Of course, you have again store all the values of the context of a user registers of procedure B because this user register well what executing the main program we are saving some values which were local to the main program. When I am going to consider me the same set of registers will be shared within main program and procedure A, but then procedure even against 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 plus B in R 0 when I am calling procedure A then I can use the same register R 0 to compute some other stuff which may be F plus 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 implemented 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)

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.

**Stack in a nested procedures**

stack after CALL B

|     |                    |
|-----|--------------------|
| EEE |                    |
|     |                    |
|     |                    |
| E20 | 1CD                |
|     | PSW of main        |
|     | PSW of main        |
|     | R0                 |
|     | R1                 |
|     | R2                 |
| E14 | R3                 |
| E13 | 37A                |
|     | PSW of Procedure A |
|     | PSW of Procedure A |
|     | R0                 |
|     | R1                 |
|     | R2                 |
|     | R3                 |
|     | .....              |
| F00 |                    |

And then procedure B is done. So, then procedure we 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 n.

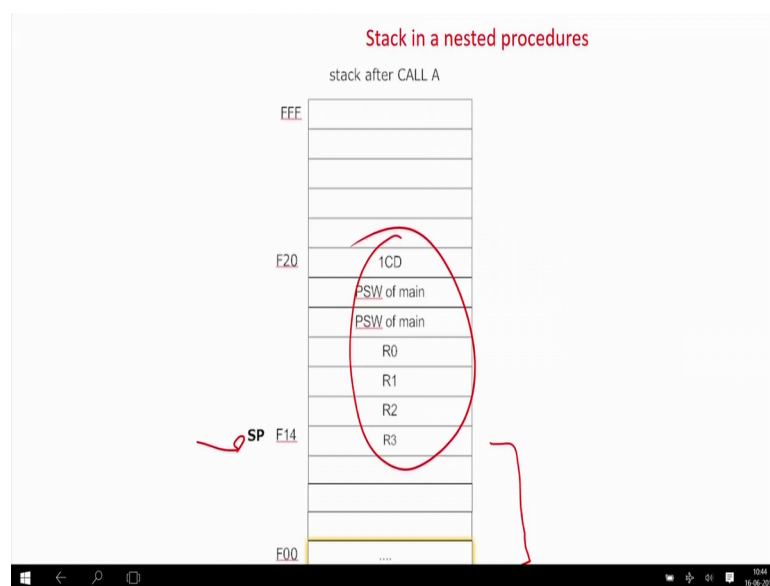
So, when I am completing procedure B see that is the 3EF which is the last location of procedure B then it is start unwinding. So, what will happen? Then procedure B is finished now it has to return. So, now, what will happen the stack will try to find out that where I have to return. So, it is nothing, but procedure A very where you have return. So, how we how the holds computer will know that what I have to do, so it does not understand or does not remembered in a very explicit sense that A called B, B called C is not in that way B has ended that is the last instruction of procedure B has completed now

it is a return instruction. So, when a 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, when procedure B has completed a return instruction is executed, now this part is involved 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, 4 PSW of a PSW other parts will be loaded in these corresponding registers and main name and these 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 either by doing a 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 it 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 (Refer Time: 30:14), 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 are do not require the context of the main program right now, because now procedure A is run.

(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.

stack after CALL A

|        |             |
|--------|-------------|
| FFF    |             |
|        |             |
|        |             |
|        |             |
|        |             |
|        |             |
| F20    | 1CD         |
|        | PSW of main |
|        | PSW of main |
|        | R0          |
|        | R1          |
|        | R2          |
| SP F14 | R3          |
|        |             |
|        |             |
| F00    | ....        |

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 remain you remember 3B0 is this 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 when return is called. So, last location of program that is 3B0 in procedure A when return is called. That means, at 3B0 has stopped then when it then as I told you whenever a return instruction is called these are R 0 R 1 R 2 R 3 will be now fed to the respective registers.

So, what will happen? So, the R 0 R 1 R 2 R 4 till now were having the values corresponding to procedure A now they will be eliminated and the values of registers correspond in the main program will be loaded into the register. Similar thing will happen for the program status one.

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 for senior 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, proceed your 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 be 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, what is going to happen? Again the stack pointer value has to be put to the register a memory address because only a part of the main memory is given for the stack implementation. So, again when have the pop means this stack pointer value will be loaded to the address register, then the address register will point to the mean 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 our 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

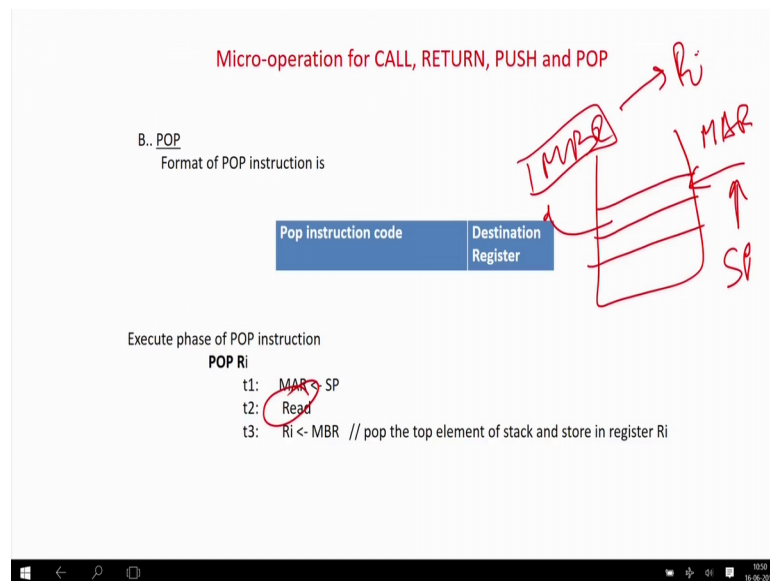
So, when I say push its very simple. So, the stack pointer will be loaded to the memory address register and then, so, now, the memory address register is having the value of the point in the stack which I want to push a value. So, the register Ri will be written to the memory buffer register and finally, the memory will be returned it is something like this. So, this is your stack pointer. So, stack pointer basically will be sorry sorry sorry. So, this is your memory, address register will be pointing to the memory that is your stack. So, this one will be fed by the stack pointer. So, stack pointers is a register it will be returned 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 sorry sorry the stack pointer the memory is a push, so it will be the other way round. So, the memory buffer register will be returned 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 register. Memory address register memory address register is pointing to the point in this stack what 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 right 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)



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 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 pop 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:  $MAR \leftarrow SP$

t2:  $MBR \leftarrow PC$

t3: Write // Push the content of PC to stack

t4:  $SP \leftarrow SP - 1$

t5:  $MAR \leftarrow SP$

t6:  $MBR \leftarrow PSW$

t7: Write // Push the content of PSW to stack

t8:  $SP \leftarrow SP - 1$  // Push Register Values.

.....

t9:  $PC \leftarrow IR_{address}$  // Transfer the address of subroutine to PC

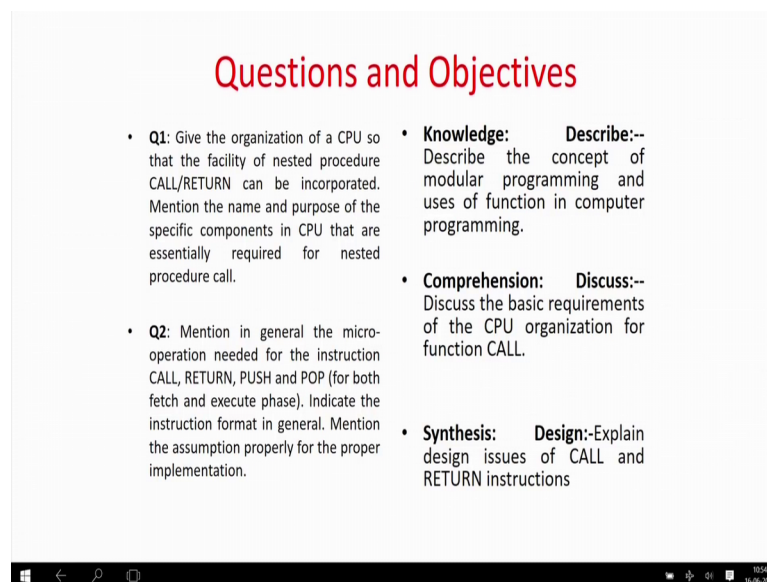
Now, call, so call is slightly complex means before you call there are so many stuff has that has to be done. So, say I want to do a call location call. So, what I will have to do? So, call is nothing, but a jump instruction. So, before doing it we have to push all the elements to the stack like the 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 the 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 at in stack pointer memory buffer register you store the value of PC. So, in this case there is a memory buffer register here you 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 equal to SP minus 1 that is SP minus one then you again load the memory address register, so now, your memory address register will be pointing to the next element of this stack because this one you have stored PC

now you decrement SP equal to SP minus 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 step program status word all registers values I will write in this manner and then finally, I will be 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, so the jump instruction. So, in this way he will jump to the respective location.

(Refer Slide Time: 38:47)



### Questions and Objectives

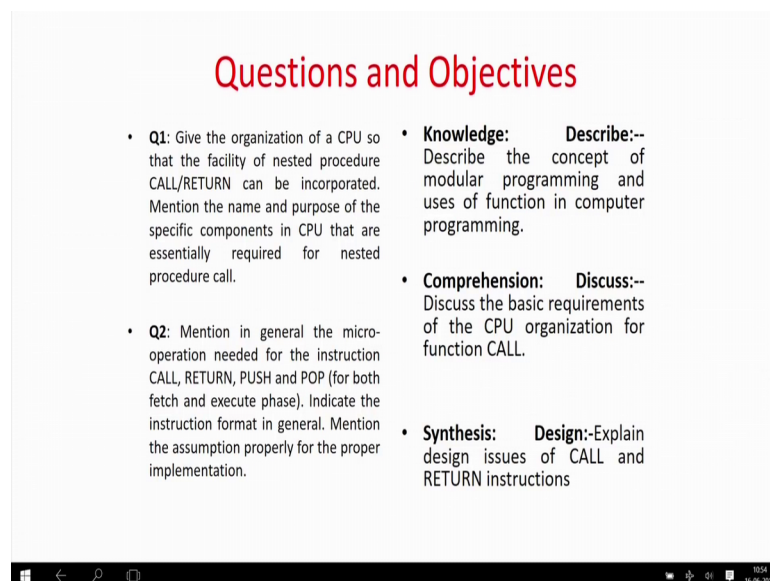
- **Q1:** Give the organization of a CPU so that the facility of nested procedure CALL/RETURN can be incorporated. Mention the name and purpose of the specific components in CPU that are essentially required for nested procedure call.
- **Q2:** Mention in general the micro-operation needed for the instruction 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, if you 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 the fact. So, whenever you want to do any kind of an operation of call returns 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 R 1 R 2 R 3 R 4 there is all the in this case it will be R read will be there. So, you will keep on reading the values if you see if the read operation. So, first from the memory buffer register you will read R 1 R 2 R 3 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 R 1 you will read, then increment the value of program SP by 1. So, now, it will be your new SP you will read R 2, R 3, R 4 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 handy registers. While I am popping I am reading R 1 R 2 R 3 R 4 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 read or write and you have to keep on doing it till you have fetched all the pushed or popped all the required elements. When you are calling any function you have to change the value of program quality 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

104  
16-09-2017

So, this basically completes this module as well as this unit. Where you 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 the conditional jumps etcetera.

So, in the next module will be trying to look into more internal medians 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 CPU how or 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 object is 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 called written 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.