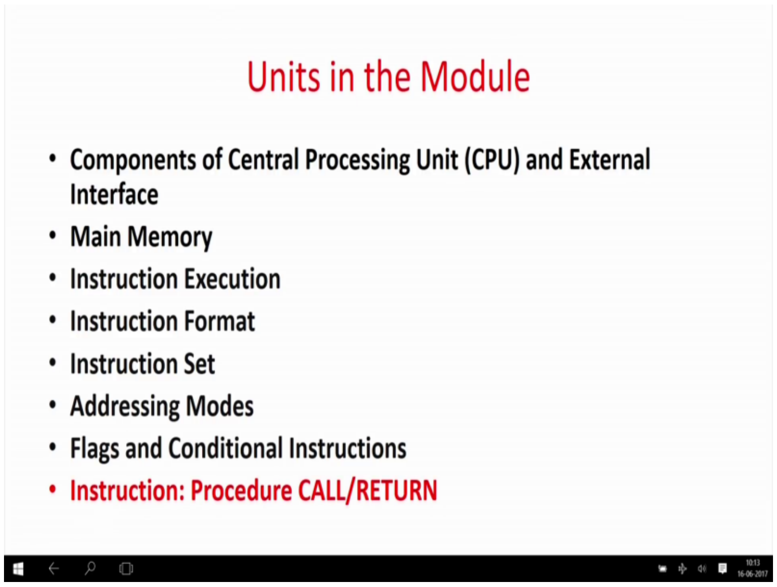
Computer Organization and Architecture A Pedagogical Aspect

Prof. Jaindra 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)



Units in the Module

- Components of Central Processing Unit (CPU) and External Interface
- Main Memory
- Instruction Execution
- Instruction Format
- Instruction Set
- Addressing Modes
- Flags and Conditional Instructions
- Instruction: Procedure CALL/RETURN

So, throughout this module in the last 7 units basically we have covered mainly different type 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 conditions of variables like zero 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 procedure return and call. So, if all of us we have written some C or C++ 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 modularize your code.

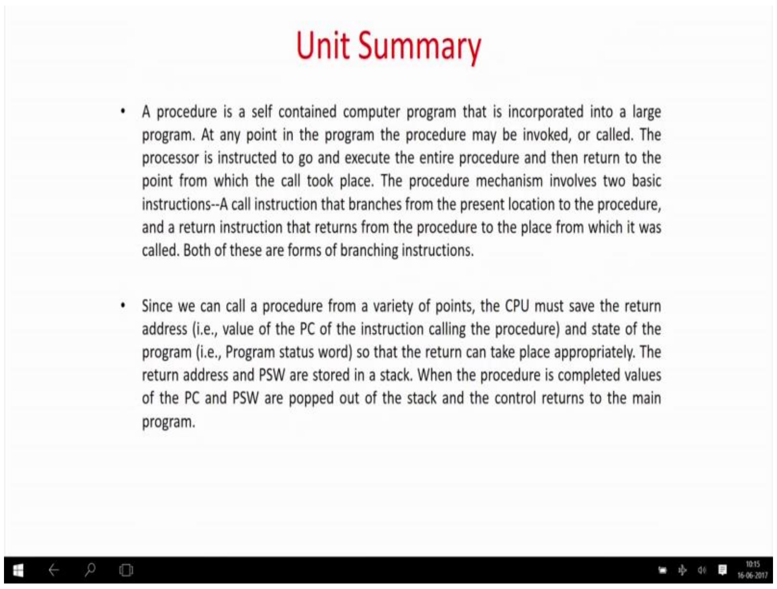
But what happens is that whenever you are writing running a main code from there you have to call to a subroutine or a function and then from that function you can call another function and it can go on. But what are the issues there the first very simple issue is that you have to know at which location 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 we 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 the save the context of the program when you are jumping to a subroutine. 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 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 where the first instruction of the procedure is placed. So, it is some form of an 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 an unconditional jump or unconditional call to the procedure.

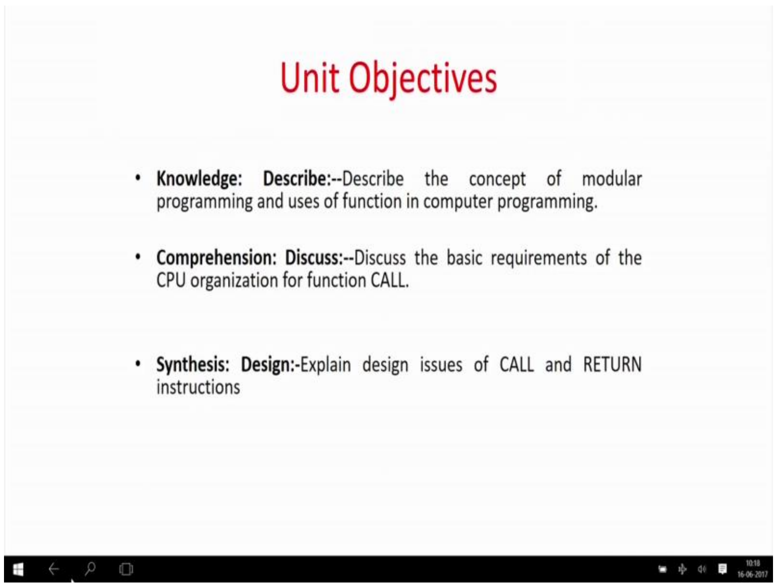
Then again as we are jumping to a new procedure, so we will be reusing all the temporal registers the accumulators, the program counter, program counter actually gets changed from the current programming location to the location of the first instruction of the procedure, but again when you want to return 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 the 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 from 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 world, 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 procedure they you have to regain back the values and you can start executing.

(Refer Slide Time: 05:14)



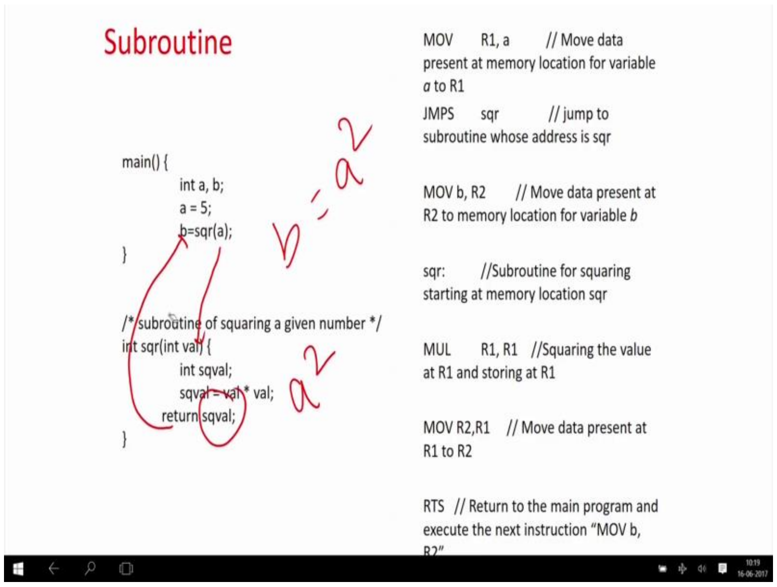
Unit Objectives

- **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

So, this is the two basic concepts we 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 will be able 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 return 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 this call and return instructions. So, these are the basic objectives which we 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"
```

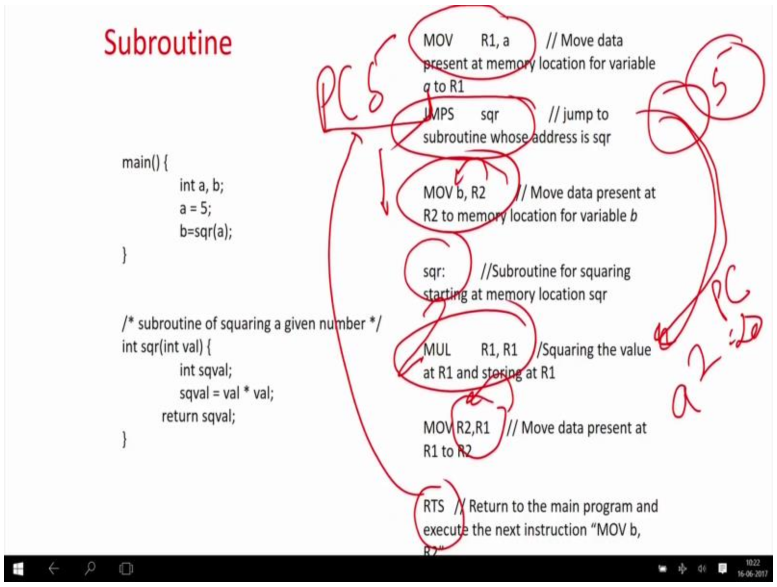
So, as we are all very much familiar with C programming. So, we are going to take a very simple C program with a subroutine and see how it reflects in an assembly language. So, if $int\ a, b$; $a = 5$, $b = r^2$. So, this is a function in which you are making a^2 then in this case actually this $b = a^2$.

So, what is the function? So, you are calling the function. So, you are passing the value of a then $int\ sqr$, then square value $sqrval = val \times val, val$ is passed over here. So, a is passed as a function. So, we are going to get the value of a^2 because the variable val has 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^2 and that is going to be returned to b and you are going to get $b = a^2$, 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 $R1$, then it's a so now $R1$ 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 this 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 where this instruction is stored. So, it is $MUL\ R1, R1$. So, already we have the value of a into $R1$. So, you are going to give the value of $R1$. So, it is nothing, but it will be $R1^2$ into $R1$ it will be stored into $R1$. Then $R1$ is going to have the value of basically a^2 now you are going to move the value of $R1$ to $R2$. So, that now $R2$ as the value of a^2 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;
}
```

```
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"
```

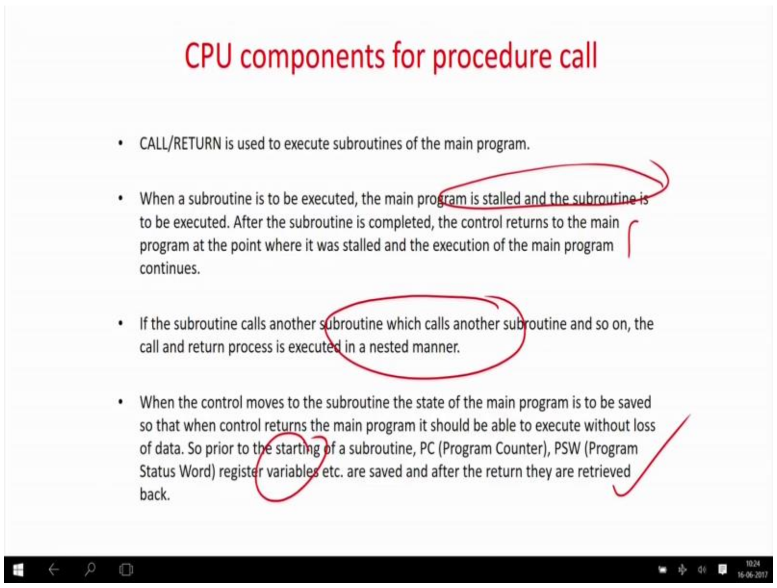
So, when I am executing the return statement what happens? So, when I am calling this one you have to store all the value of the temporary variables temporary registers as well as most importantly you have to store the value of the memory location for the jump instruction because when I am going to return from here I have to go back here and then I will do, what I will do is that I have come back 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 at this space 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 we are 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 the instruction in which is called move $R2$ to b. So, in this case the value of $R2$ will be moved over here and if you look at the program carefully $R2$ is nothing but it is basically $R1^2$ that is nothing but a^2. So, a^2 will be transferred to the value a sorry this one instruction will be transferring the value of a^2 to b which is actually the requirement of the program.

So in fact, this is some background information which I 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

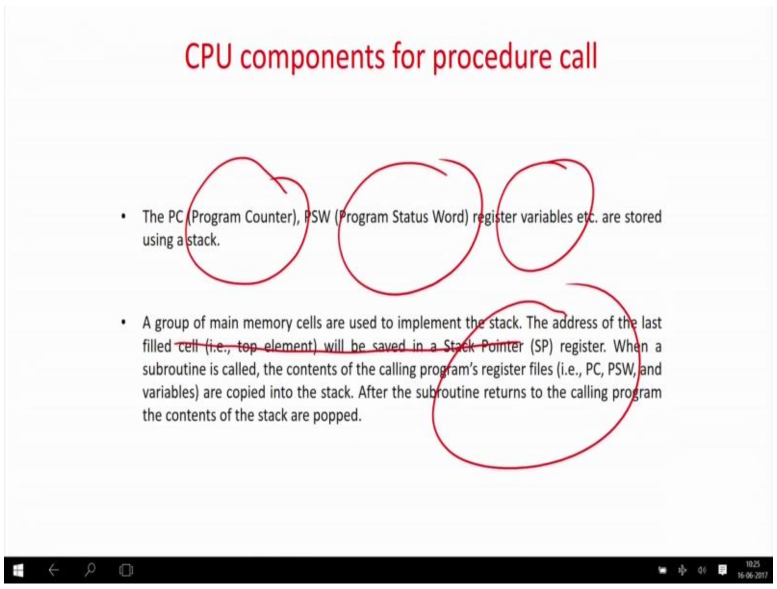
- CALL/RETURN is used to execute subroutines of the main program.
- When a subroutine is to be executed, the main program is stalled and the subroutine is to be executed. After the subroutine is completed, the control returns to the main program at the point where it was stalled and the execution of 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 what are the components required or what is the hardware required in the pros in the 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 word, the storage of the registers, the storage of PC will be in a recursive manner 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 a register in a stack and whenever you are finishing one subroutine after another you are popping up the corresponding program status word, registers, PC in a return return procedure manner.

And so, basically that is what is being said when we move from one subroutine to another you are saving program counter, program status word, 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 from one subroutine to 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

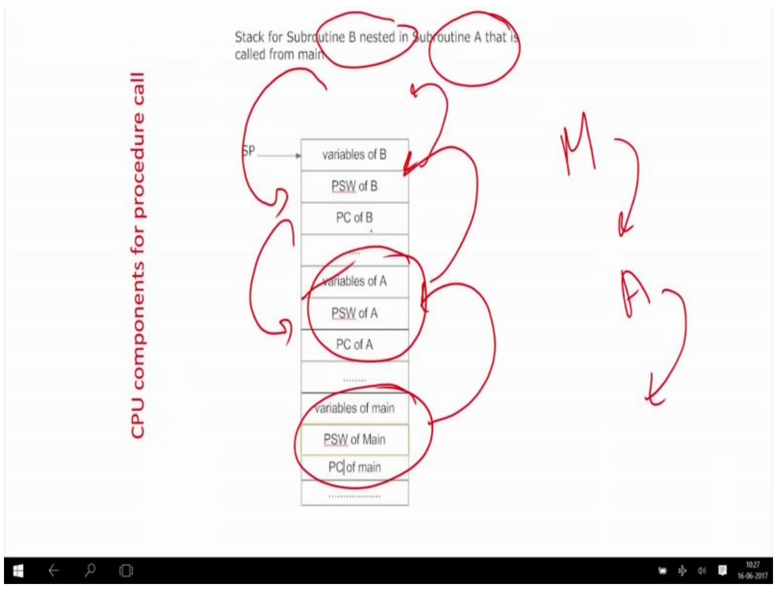
- The PC (Program Counter), PSW (Program Status Word) register variables etc. are stored using a stack.
- A group of main memory cells are used to implement the stack. The address of the last filled ~~cell (i.e., top element) will be saved in a Stack Pointer~~ (SP) register. When a subroutine is called, 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.

So, as I told you, so program counter, program status word and register variables are very very important components which are stored 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 save back the value of the flag register 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 stack and there is a special stack pointer which will actually locate that where 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)


CPU components for procedure call
Stack for Subroutine B nested in Subroutine A that is called from main
SP
variables of B
PSW of B
PC of B
variables of A
PSW of A
PC of A
variables of main
PSW of Main
PC of main
M
A

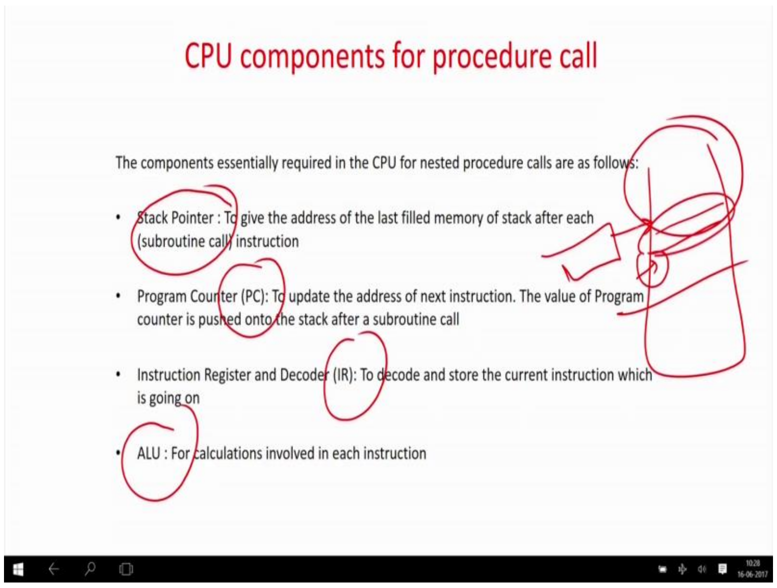
So, something like this is a pictorial representation. So, it says that subroutine B is nested in subroutine A so in fact, what it says there is a main program then it will call program A and it will call program B. So, in this case if you see when the main program is calling procedure A then what happens 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 go to the procedure A. From procedure A procedure B will be called. So, whenever you are going from proceed your A to procedure B of course, you have to store the same 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 the 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 done 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 restart 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:

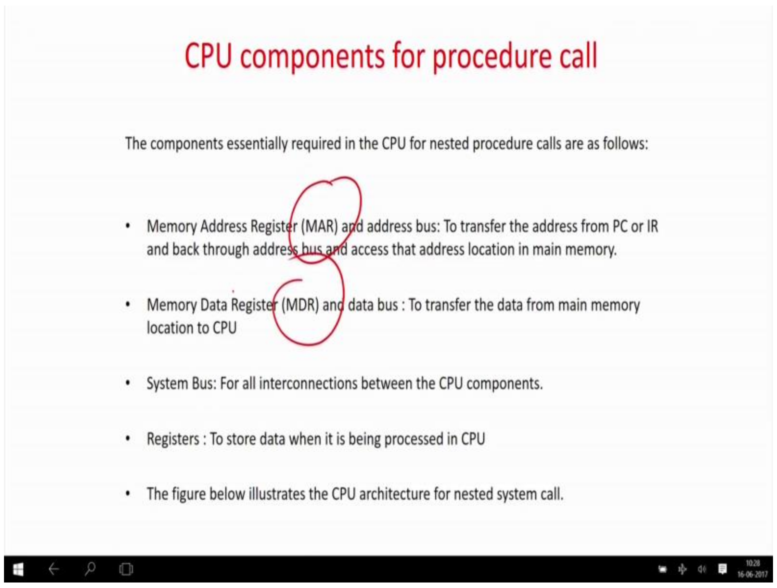
- Stack Pointer : To give the address of the last filled memory of stack after each (subroutine call) instruction
- Program Counter (PC): To update the address of next instruction. The value of Program counter is pushed onto the stack after a subroutine call
- Instruction Register and Decoder (IR): To 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 which is the top of the stack.

So, the whole main memory actually is there. You are allocating a part of the main memory for 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 have to recollect which is the top of the stack.

So, whenever you want to pop some elements 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 pointer to the memory address register and then you can easily start popping up the values.

(Refer Slide Time: 15:13)



CPU components for procedure call

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

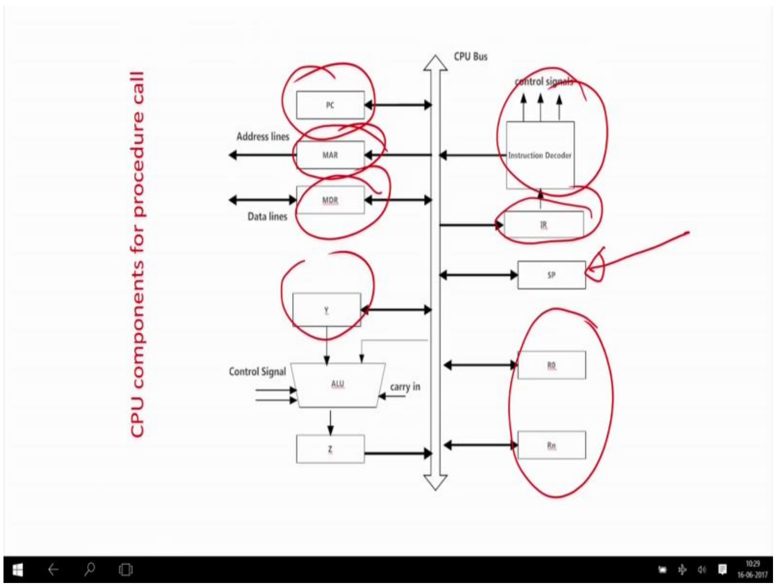
- Memory Address Register (MAR) and address bus: To transfer the address from PC or IR and back through address bus and access that address location in main memory.
- Memory Data Register (MDR) and data bus : To transfer the data from main memory location to CPU
- System Bus: For all interconnections 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.

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 call, 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 fetch 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 I need a variable or 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 the 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 a 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 main memory for the last element that was pushed 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 from the memory will be fetched and it will be fed back to the memory data register and from the memory data register 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.