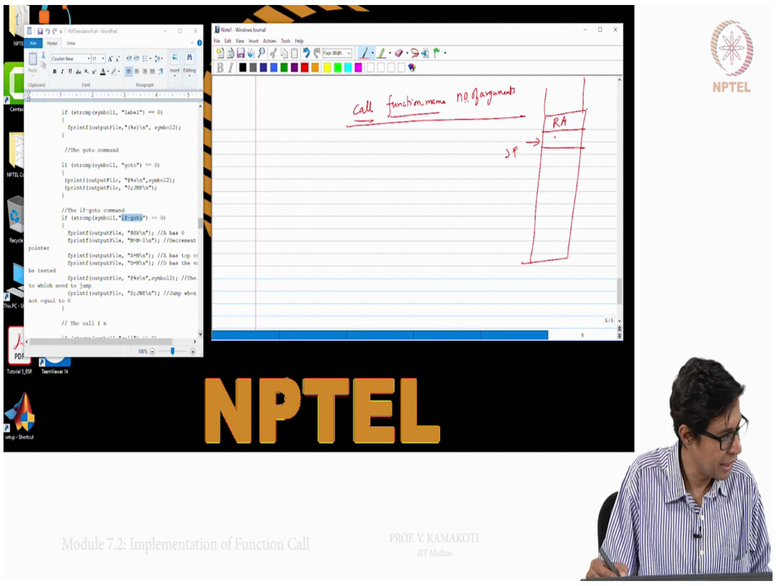
Foundations to Computer Systems Design
Professor V. Kamakoti
Department of Computer Science and Engineering
Indian Institute of Technology Madras
Module 7.2
Implementation of Function Call

So welcome to module 7.2. In this module 7.2 we will now talk about implementation of the function call commands of the virtual machine. So there are three function call commands and that we have to see in detail. Now how they are getting translated, etc. Now there is a call function name and it will tell you number of arguments, okay.

So apparently what should happen is in the stack when I do a call as we saw in the last module, I need to basically push a return address, right, because after this call finishes, it has to come back to some return address so I have to push that return address in the stack. So this is a stack. Now when the call executes, some function name number function of, so I need to put a return address first. So a stack pointer now points here.

(Refer Slide Time: 01:40)

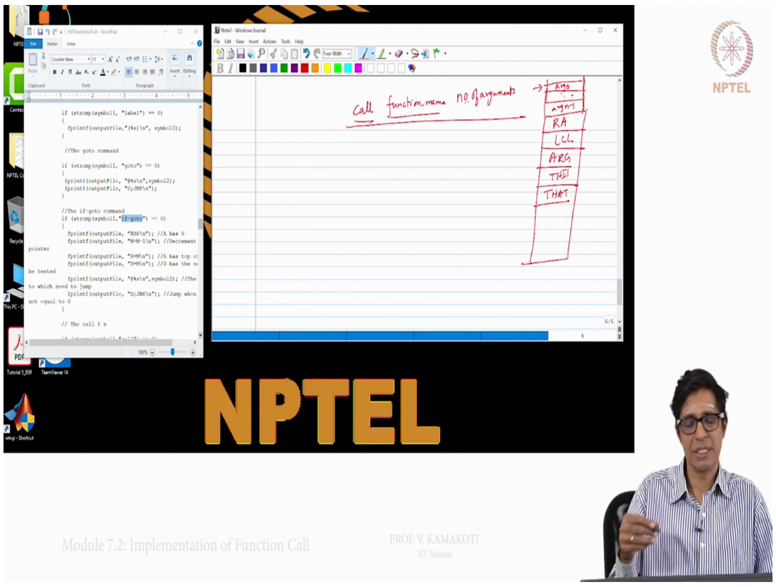

```
if (strcmp(symbol1, "label") == 0)
{
  fprintf(outputFile,"(%s)\n", symbol2);
}

 //The goto command

if (strcmp(symbol1, "goto") == 0)
{
 fprintf(outputFile, "@%s\n",symbol2);
 fprintf(outputFile, "0;JMP\n");
}

//The if-goto command
if (strcmp(symbol1,"if-goto") == 0)
{
  fprintf(outputFile, "@SP\n"); //A has 0
  fprintf(outputFile, "M=M-1\n"); //Decrement
pointer
  fprintf(outputFile, "A=M\n"); //A has top o
  fprintf(outputFile, "D=M\n"); //D has the v
be tested
  fprintf(outputFile, "@%s\n",symbol2); //The
to which need to jump
  fprintf(outputFile, "D;JNE\n"); //Jump when
not equal to 0
}

// The call f n
```

call function_name no. of arguments
RA
SP
NPTEL
Module 7.2: Implementation of Function Call
PROF. V. KAMAKOTI
IIT Madras

Then I have to store as I mentioned here the LCL, the ARG, then the THIS and THAT. These are all basically pushed into the stack. This is LCL of the calling function, ARG of the calling function, THIS and THAT of the calling function. These are all the base addresses what you see in the data RAM 1, 2, 3, 4.

All these things we push here, right? Now after this before I execute a call I also assume that all the arguments are actually let us say there are K arguments, argument 0 or n argument to argument n minus 1 is already pushed by the calling function, right?

(Refer Slide Time: 02:35)



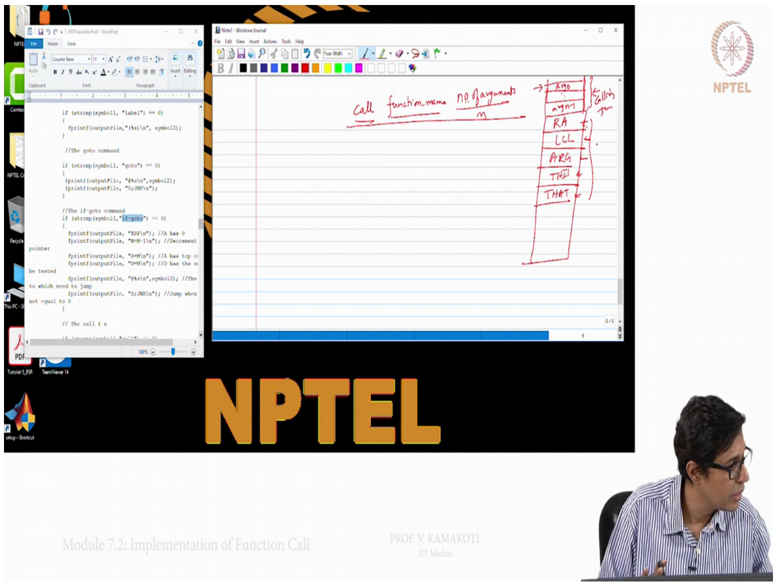
```
        if (strcmp(symbol1, "label") == 0)
        {
          fprintf(outputFile,"(%s)\n", symbol2);
        }

         //The goto command

        if (strcmp(symbol1, "goto") == 0)
        {
         fprintf(outputFile, "@%s\n",symbol2);
         fprintf(outputFile, "0;JMP\n");
        }

        //The if-goto command
        if (strcmp(symbol1,"if-goto") == 0)
        {
          fprintf(outputFile, "@SP\n"); //A has 0
          fprintf(outputFile, "M=M-1\n"); //Decrement
pointer
          fprintf(outputFile, "A=M\n"); //A has top o
          fprintf(outputFile, "D=M\n"); //D has the v
be tested
          fprintf(outputFile, "@%s\n",symbol2); //The
to which need to jump
          fprintf(outputFile, "D;JNE\n"); //Jump when
not equal to 0
        }

        // The call f n
```

call function_name no. of arguments
arg0
argn-1
RA
LCL
ARG
THIS
THAT
NPTEL
Module 7.2: Implementation of Function Call
PROF V. KAMAKOTI
IIT Madras

So argument 0 to argument n minus 1, totally n arguments, this is n, are already pushed by the calling function. So when I say I am calling say multiply something, I need two values A, B. This will be already pushed by the calling function. It will push this into the stack and then it will call multiply. So that assumption that I am making before call all arguments are available on the top of the stack. So these are the n arguments which are pushed by the calling function. Then only it is executing call.

So your stack has the n arguments available when the call is executing. Your stack point just on the top of the stack if you go on the first n locations on top you have all the n arguments available. Then when call executes, first return address is put in, then the local of that call is put in, ARG of the calling function is put in, THIS and THAT of the calling function because the calling function will also have arguments.

Refer Slide Time: 03:52)



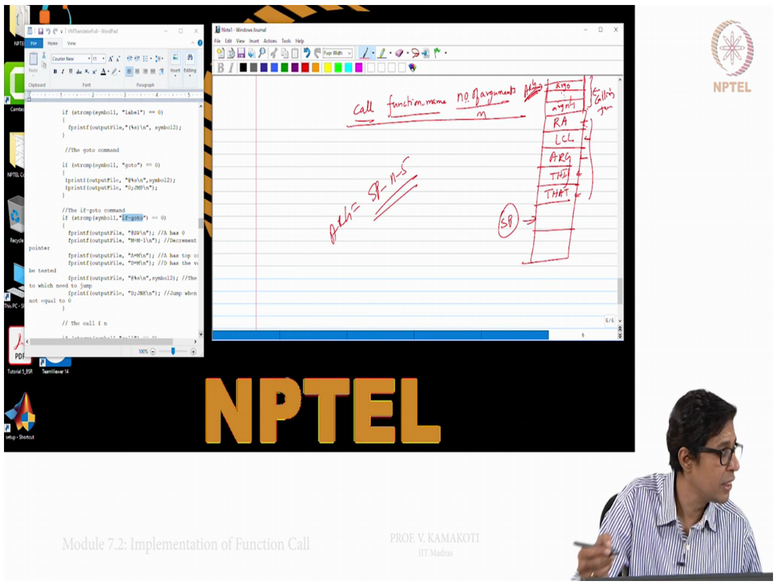
```
        if (strcmp(symbol1, "label") == 0)
        {
          fprintf(outputFile,"(%s)\n", symbol2);
        }

         //The goto command

        if (strcmp(symbol1, "goto") == 0)
        {
         fprintf(outputFile, "@%s\n",symbol2);
         fprintf(outputFile, "0;JMP\n");
        }

        //The if-goto command
        if (strcmp(symbol1,"if-goto") == 0)
        {
          fprintf(outputFile, "@SP\n"); //A has 0
          fprintf(outputFile, "M=M-1\n"); //Decrement
pointer
          fprintf(outputFile, "A=M\n"); //A has top o
          fprintf(outputFile, "D=M\n"); //D has the va
be tested
          fprintf(outputFile, "@%s\n",symbol2); //The
to which need to jump
          fprintf(outputFile, "D;JNE\n"); //Jump when
not equal to 0
        }

        // The call f n
```

call function_name no. of arguments
n
arg0
argn-1
RA
LCL
ARG
THIS
THAT
Calling fun
NPTEL
Module 7.2: Implementation of Function Call
PROF. V. KAMAKOTI
IIT Madras

Now we have to find a new ARG. Where is the new ARG? The new ARG, so currently SP is here. The new ARG will be here. For the called function the ARG will be starting at ARG 0 and that is nothing but SP. This is the current value of SP minus n. There are n elements minus 5. So if I subtract minus n minus 5, if I subtract n and 5 from SP then I know this is ARG. So ARG will be SP minus n arguments minus 5 that you are seeing.

So the new ARG of the called function will start at this point. This is correct, right, because these are the arguments passed by the calling function. These are the arguments of the called function.

(Refer Slide Time: 04:46)

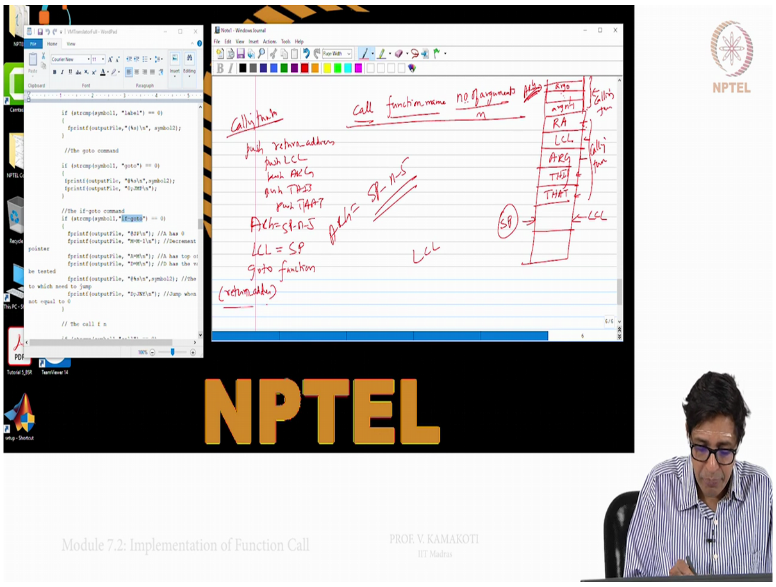


```
        if (strcmp(symbol1, "label") == 0)
        {
          fprintf(outputFile,"(%s)\n", symbol2);
        }

         //The goto command

        if (strcmp(symbol1, "goto") == 0)
        {
         fprintf(outputFile, "@%s\n",symbol2);
         fprintf(outputFile, "0;JMP\n");
        }

        //The if-goto command
        if (strcmp(symbol1,"if-goto") == 0)
        {
          fprintf(outputFile, "@SP\n"); //A has 0
          fprintf(outputFile, "M=M-1\n"); //Decrement
pointer
          fprintf(outputFile, "A=M\n"); //A has top o
          fprintf(outputFile, "D=M\n"); //D has the va
be tested
          fprintf(outputFile, "@%s\n",symbol2); //The
to which need to jump
          fprintf(outputFile, "D;JNE\n"); //Jump when
not equal to 0
        }

        // The call f n
```

And your new local, whatever you have stored here is for the calling function. The new local is nothing but your SP. So this is your new local, right? And then what we need to do, right, so these are all the steps I need to do. First I need to push a return address. This all happens in the calling function. In the calling function I have to push a return address and I have to create that label for the return address also.

Then I have to push LCL, I have to push ARG, I have to push THIS, I have to push THAT, right? Now I have to reset ARG as SP minus n minus 5. I have to make the new local as current SP and then I have to say go to the function. And your return address would be here. So after that function finishes when it returns it will come back to this return address.

(Refer Slide Time: 06:24)



```
if (strcmp(symbol1, "label") == 0)
{
  fprintf(outputFile,"(%s)\n", symbol2);
}

 //The goto command

if (strcmp(symbol1, "goto") == 0)
{
 fprintf(outputFile, "@%s\n",symbol2);
 fprintf(outputFile, "0;JMP\n");
}

//The if-goto command
if (strcmp(symbol1,"if-goto") == 0)
{
  fprintf(outputFile, "@SP\n"); //A has 0
  fprintf(outputFile, "M=M-1\n"); //Decrement
pointer
  fprintf(outputFile, "A=M\n"); //A has top o
  fprintf(outputFile, "D=M\n"); //D has the v
be tested
  fprintf(outputFile, "@%s\n",symbol2); //The
to which need to jump
  fprintf(outputFile, "D;JNE\n"); //Jump when
not equal to 0
}

// The call f n
```

Calling function

push return_address
push LCL
push ARG
push THIS
push THAT
ARG = SP-n-5
LCL = SP
goto function
(return_address)

call function_name no. of arguments
n

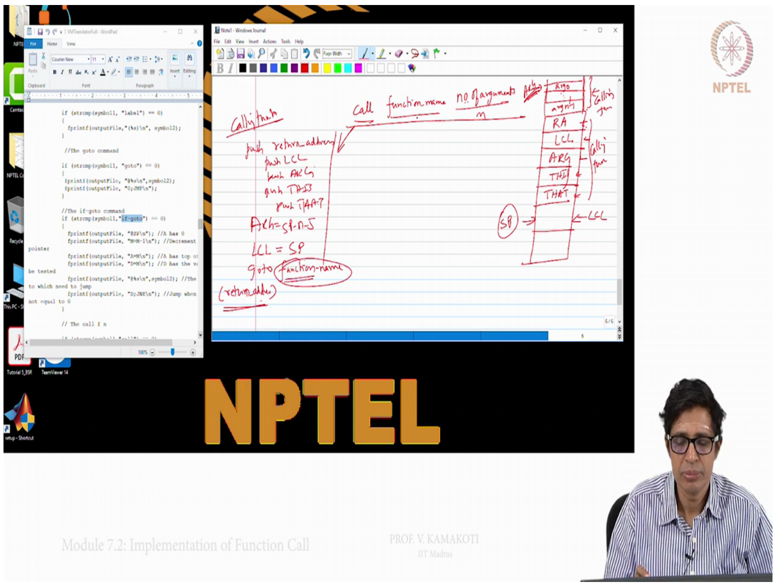
ARG = SP-n-5

LCL

So these are all whatever the 1, 2, 3, 4, 5, 6, 7, 8 different instructions that we need to and of course 1 label, so totally 9 things that we need to take care when there is a call. I hope you also understood why I am pushing the LCL, why I am pushing the ARG, THIS and THAT of the current function. So calling function does all these things.

So this code is the translation of your call and this call will be in the calling function so this code will be in the calling function of the translated version, right? So the calling function is responsible for saving all its context because once the called function finishes it will retrieve it back, right? So this is how calling function works. And it just go to function name, whatever.

(Refer Slide Time: 07:30)

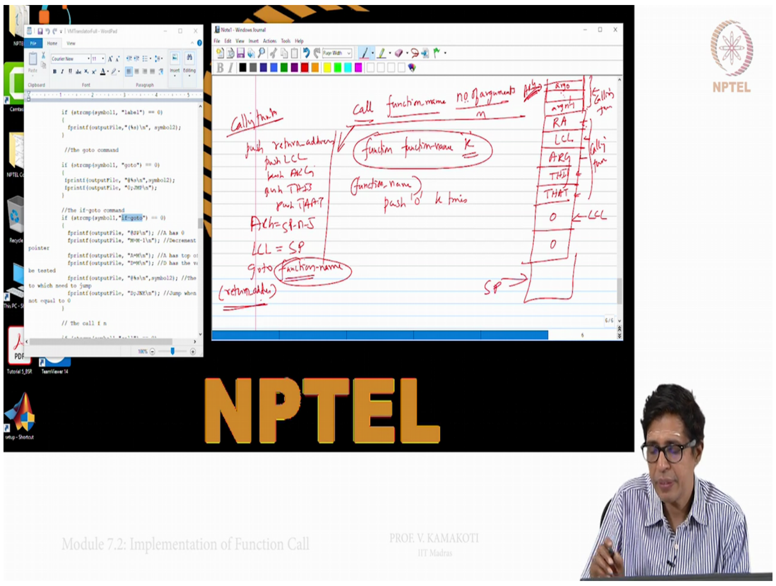


```
if (strcmp(symbol1, "label") == 0)
{
  fprintf(outputFile,"(%s)\n", symbol2);
}

 //The goto command

if (strcmp(symbol1, "goto") == 0)
{
 fprintf(outputFile, "@%s\n",symbol2);
 fprintf(outputFile, "0;JMP\n");
}

//The if-goto command
if (strcmp(symbol1,"if-goto") == 0)
{
  fprintf(outputFile, "@SP\n"); //A has 0
  fprintf(outputFile, "M=M-1\n"); //Decrement
pointer
  fprintf(outputFile, "A=M\n"); //A has top o
  fprintf(outputFile, "D=M\n"); //D has the v
be tested
  fprintf(outputFile, "@%s\n",symbol2); //The
to which need to jump
  fprintf(outputFile, "D;JNE\n"); //Jump when
not equal to 0
}

// The call f n
```

So how do I declare a function? I said function name and there are K local variables for this. So anybody who wants to execute this function who calls this function they will execute this go to that function name, right? So the entry point to this function will always be the name of that function. So I can say function name within parenthesis.

So if I say go to function name, essentially I go and jump on to this function. So the assembler will take care of this, right? Because it will go to statement that we have already seen, okay. And then what we do is now we just say push 0 K times. Suppose this is function with two arguments I will push 0 0 and SP will come here.

(Refer Slide Time: 08:38)

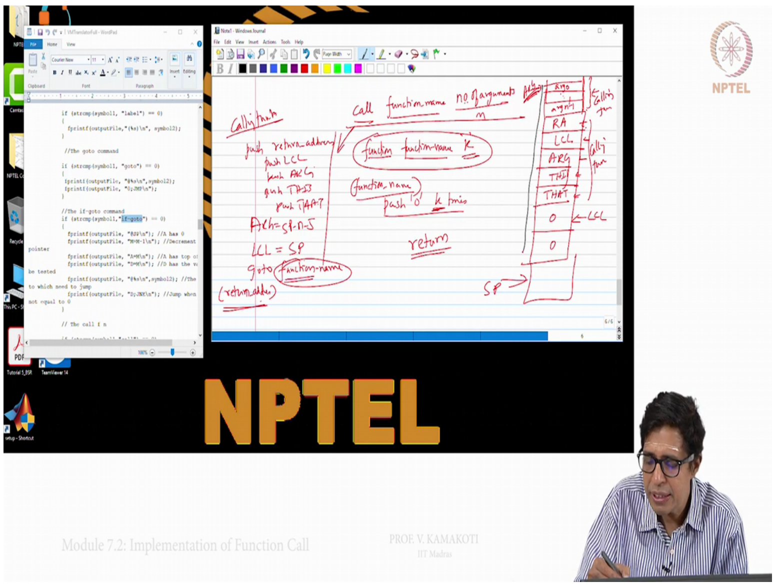


```
if (strcmp(symbol1, "label") == 0)
{
  fprintf(outputFile,"(%s)\n", symbol2);
}

 //The goto command

if (strcmp(symbol1, "goto") == 0)
{
 fprintf(outputFile, "@%s\n",symbol2);
 fprintf(outputFile, "0;JMP\n");
}

//The if-goto command
if (strcmp(symbol1,"if-goto") == 0)
{
  fprintf(outputFile, "@SP\n"); //A has 0
  fprintf(outputFile, "M=M-1\n"); //Decrement
pointer
  fprintf(outputFile, "A=M\n"); //A has top o
  fprintf(outputFile, "D=M\n"); //D has the v
be tested
  fprintf(outputFile, "@%s\n",symbol2); //The
to which need to jump
  fprintf(outputFile, "D;JNE\n"); //Jump when
not equal to 0
}

// The call f n
```

What is K? This is the local variables and all the variables she is setting it to 0. All the variables is set to 0 by the function. So the moment I start executing, when I see this function name K command, immediately I replace it with the function name within parenthesis loop because anybody wants to come to this function will go to function name. So that label should be there as function name.

The first thing I do there is to initialise all the K local variables and then I keep translating whatever is inside but these two are very important, right? Now this is done. Now what should return do, the last one? So once I return as I had shown in the previous module 7.1, the value that I am returning should be on the top of the stack, right? So in this case when the called function returns all these things become unnecessary. All these are useless.

(Refer Slide Time: 10:03)



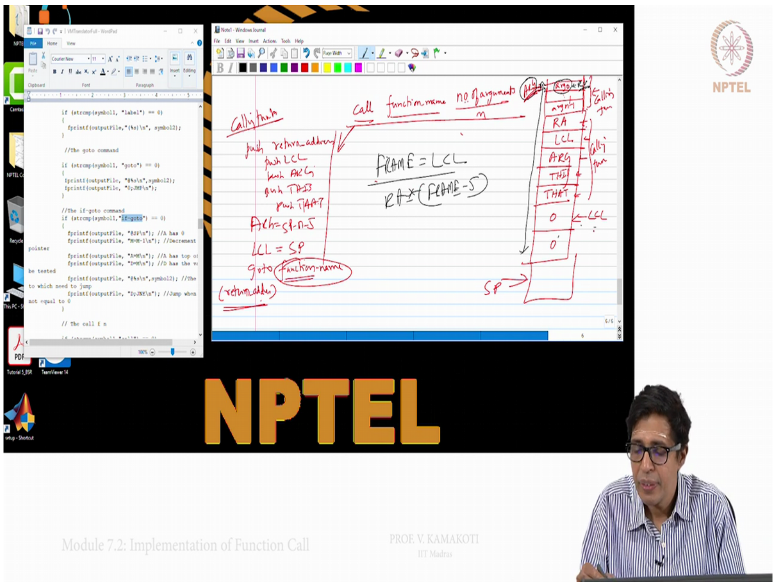
```
if (strcmp(symbol1, "label") == 0)
{
  fprintf(outputFile,"(%s)\n", symbol2);
}

 //The goto command

if (strcmp(symbol1, "goto") == 0)
{
 fprintf(outputFile, "@%s\n",symbol2);
 fprintf(outputFile, "0;JMP\n");
}

//The if-goto command
if (strcmp(symbol1,"if-goto") == 0)
{
  fprintf(outputFile, "@SP\n"); //A has 0
  fprintf(outputFile, "M=M-1\n"); //Decrement
pointer
  fprintf(outputFile, "A=M\n"); //A has top o
  fprintf(outputFile, "D=M\n"); //D has the v
be tested
  fprintf(outputFile, "@%s\n",symbol2); //The
to which need to jump
  fprintf(outputFile, "D;JNE\n"); //Jump when
not equal to 0
}

// The call f n
```

So the return value we will go and save it in ARG 0, right? In ARG, so wherever ARG is pointing this is where the return value will be put and then you return. So then you pop this out so the stack pointer will be pointing to this. So we will exactly go back to the state where all the calling function will go back to the return address. And when the calling function starts re-executing the code after the called function finishes, on the top of the stack it should see the return value.

So we will go and write the return value wherever ARG 0 was stored because that is the first thing. After that it is all of the calling function, right? So the called function populated at the stack this much and all these things go off and I will have only one reminiscent which is the return value, right? So how should the return work?

The return will be in the path of that called function. So when I want to return I will just say your frame as LCL. Let us say your frame as LCLC. Now the return address will be content of frame minus 5.

(Refer Slide Time: 11:45)

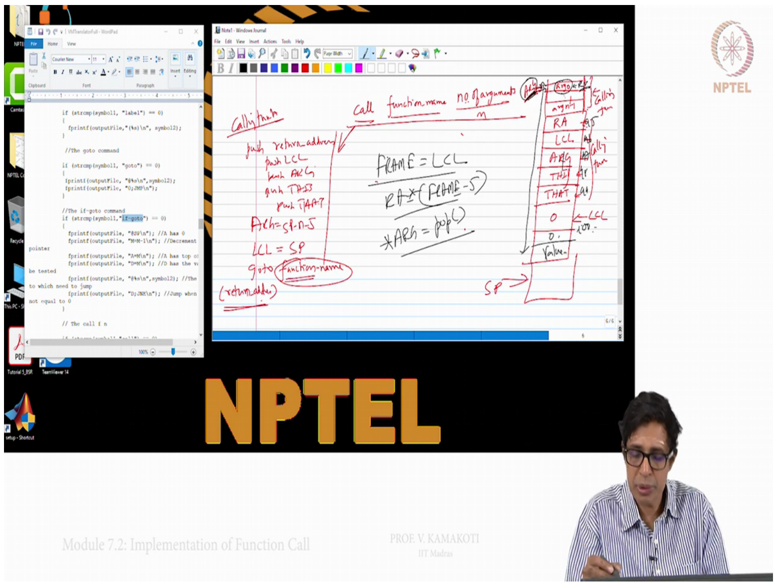


```
        if (strcmp(symbol1, "label") == 0)
        {
          fprintf(outputFile,"(%s)\n", symbol2);
        }

         //The goto command

        if (strcmp(symbol1, "goto") == 0)
        {
         fprintf(outputFile, "@%s\n",symbol2);
         fprintf(outputFile, "0;JMP\n");
        }

        //The if-goto command
        if (strcmp(symbol1,"if-goto") == 0)
        {
          fprintf(outputFile, "@SP\n"); //A has 0
          fprintf(outputFile, "M=M-1\n"); //Decrement
pointer
          fprintf(outputFile, "A=M\n"); //A has top o
          fprintf(outputFile, "D=M\n"); //D has the v
be tested
          fprintf(outputFile, "@%s\n",symbol2); //The
to which need to jump
          fprintf(outputFile, "D;JNE\n"); //Jump when
not equal to 0
        }

        // The call f n
```

So let me say this is 100, so this is 99, 98, 97, 96, 95. The return address is always from the local minus 5 always, right, because local starts here and then there are 5. So local minus 5 will be your return address, right? Now what you do is your stack would have completed all the computation so this will be say somewhere it will have the last value. So somewhere this is the stack. This will be the value that it has computed, right? Now you pop this value and put it in ARG 0.

Your pop this value because the entire stack now is going after the called function finishes executing this execution this whole stack goes off. So you pop this value and put it at ARG 0. That is what you mean by star of ARG, this is the ARG of the called function, is equal to the pop. So this is the third instruction that you need to execute.

(Refer Slide Time: 13:14)

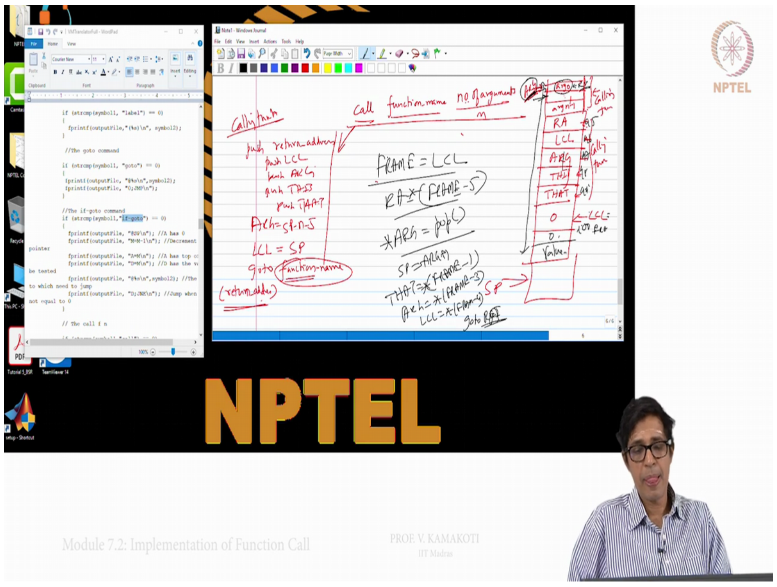


```
        if (strcmp(symbol1, "label") == 0)
        {
          fprintf(outputFile,"(%s)\n", symbol2);
        }

         //The goto command

        if (strcmp(symbol1, "goto") == 0)
        {
         fprintf(outputFile, "@%s\n",symbol2);
         fprintf(outputFile, "0;JMP\n");
        }

        //The if-goto command
        if (strcmp(symbol1,"if-goto") == 0)
        {
          fprintf(outputFile, "@SP\n"); //A has 0
          fprintf(outputFile, "M=M-1\n"); //Decrement
pointer
          fprintf(outputFile, "A=M\n"); //A has top o
          fprintf(outputFile, "D=M\n"); //D has the v
be tested
          fprintf(outputFile, "@%s\n",symbol2); //The
to which need to jump
          fprintf(outputFile, "D;JNE\n"); //Jump when
not equal to 0
        }

        // The call f n
```

So you pop the stack and store it in wherever ARG 0 is, right? Then what is SP? Now your stack should now point to ARG plus 1, so SP will be ARG plus 1. If I move SP to ARG plus 1 then the whole thing goes off, right, because this is for some function that has already executed. Now I have to retrieve back. So what is THAT? I have to retrieve back the THAT, THIS, ARG, LCL of the calling function, right? So THAT is nothing but my frame minus 1. Frame is LCL, right? Frame minus 1. Content of frame minus 1.

Frame and LCL are same so that is content of frame minus 1. Similarly this would be content of frame minus 2. Of course ARG we have done. Your ARG will be content of frame minus 3. Similarly this will be content of frame minus 2. Your LCL will be content of frame minus 4. And then we say go to the return address RA which is already content of frame minus 1.

(Refer Slide Time: 14:44)



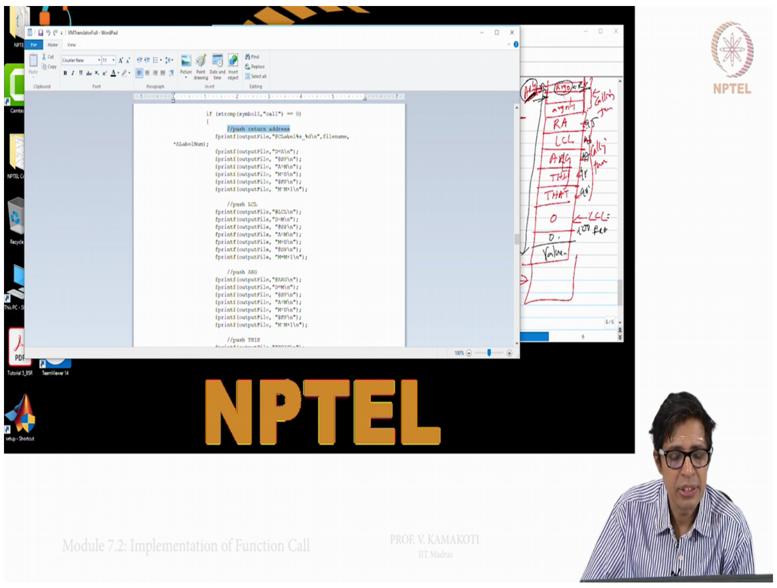
```
if (strcmp(symbol1, "label") == 0)
{
  fprintf(outputFile,"(%s)\n", symbol2);
}

 //The goto command

if (strcmp(symbol1, "goto") == 0)
{
 fprintf(outputFile, "@%s\n",symbol2);
 fprintf(outputFile, "0;JMP\n");
}

//The if-goto command
if (strcmp(symbol1,"if-goto") == 0)
{
  fprintf(outputFile, "@SP\n"); //A has 0
  fprintf(outputFile, "M=M-1\n"); //Decrement
pointer
  fprintf(outputFile, "A=M\n"); //A has top of
  fprintf(outputFile, "D=M\n"); //D has the va
be tested
  fprintf(outputFile, "@%s\n",symbol2); //The
to which need to jump
  fprintf(outputFile, "D;JNE\n"); //Jump when
not equal to 0
}

// The call f n
```

So we exactly execute in this sequence. So this is what the return address. So what will the return do? Return basically returns back to the called function and while going back it restores back the LCL, ARG, THIS and THAT of your calling function so that calling function start executing at the same point where it left.

And also note that the return value will be on the top of the stack when the calling function gets back to control. So it can pop from the stack and take the value that the called function has actually generated, okay, right? So one of the thing is that this should be executed in this sequence. So first frame is LCL, RA is this and stuff. So these are the sort of 9 instructions that we execute, right? So this is done.

So this is how return basically works, okay. So these three are the call, return and function. So let me just give you a brief demo of how this is going to work. So the moment I see a call I pushed a return address. So this is the thing, I push a return address here.

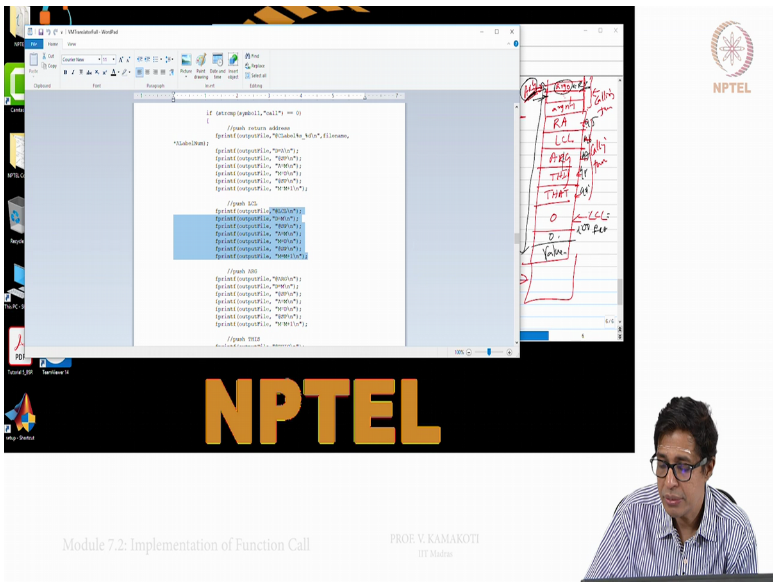
(Refer Slide Time: 16:22)



```
if (strcmp(symbol1,"call") == 0)
{
    //push return address
   fprintf(outputFile,"@CLabel%s_%d\n",filename,
*ALabelNum);
   fprintf(outputFile,"D=A\n");
   fprintf(outputFile, "@SP\n");
   fprintf(outputFile, "A=M\n");
   fprintf(outputFile, "M=D\n");
   fprintf(outputFile, "@SP\n");
   fprintf(outputFile, "M=M+1\n");

    //push LCL
   fprintf(outputFile,"@LCL\n");
   fprintf(outputFile,"D=M\n");
   fprintf(outputFile, "@SP\n");
   fprintf(outputFile, "A=M\n");
   fprintf(outputFile, "M=D\n");
   fprintf(outputFile, "@SP\n");
   fprintf(outputFile, "M=M+1\n");

    //push ARG
   fprintf(outputFile,"@ARG\n");
   fprintf(outputFile,"D=M\n");
   fprintf(outputFile, "@SP\n");
   fprintf(outputFile, "A=M\n");
   fprintf(outputFile, "M=D\n");
   fprintf(outputFile, "@SP\n");
   fprintf(outputFile, "M=M+1\n");

    //push THIS
```

So there is a label that I generate and I basically push it into the stack. So this is very simple. So there is a label and that label I have to push in the stack so I just push that label as at A. I put it into D. Now I go and push it into the stack. Similarly, I will push LCL. This is the code for pushing LCL.

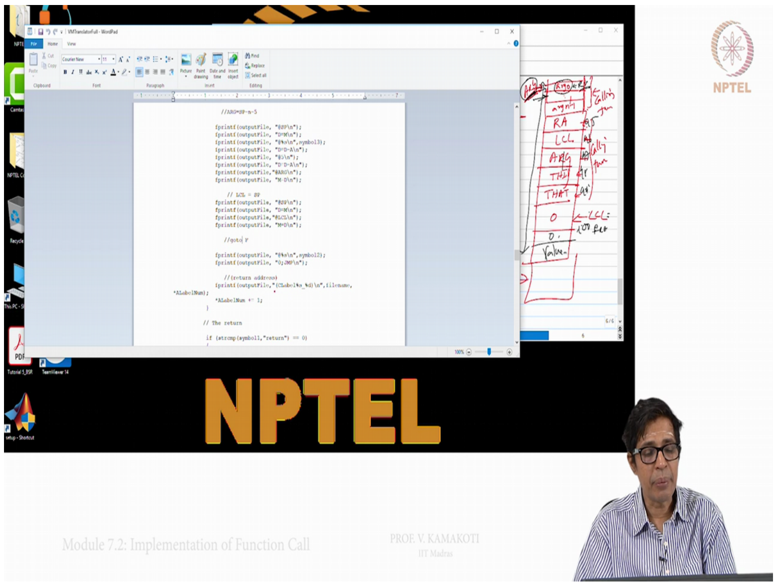
(Refer Slide Time: 16:54)



```
if (strcmp(symbol1,"call") == 0)
{
    //push return address
   fprintf(outputFile,"@CLabel%s_%d\n",filename,
*ALabelNum);
   fprintf(outputFile,"D=A\n");
   fprintf(outputFile, "@SP\n");
   fprintf(outputFile, "A=M\n");
   fprintf(outputFile, "M=D\n");
   fprintf(outputFile, "@SP\n");
   fprintf(outputFile, "M=M+1\n");

    //push LCL
   fprintf(outputFile,"@LCL\n");
   fprintf(outputFile,"D=M\n");
   fprintf(outputFile, "@SP\n");
   fprintf(outputFile, "A=M\n");
   fprintf(outputFile, "M=D\n");
   fprintf(outputFile, "@SP\n");
   fprintf(outputFile, "M=M+1\n");

    //push ARG
   fprintf(outputFile,"@ARG\n");
   fprintf(outputFile,"D=M\n");
   fprintf(outputFile, "@SP\n");
   fprintf(outputFile, "A=M\n");
   fprintf(outputFile, "M=D\n");
   fprintf(outputFile, "@SP\n");
   fprintf(outputFile, "M=M+1\n");

    //push THIS
```

I push ARG, I push THIS, I push THAT, then I compute ARG as SP minus n minus 5. Then I said LCL is equal to SP for the called function and then go to that function F.

(Refer Slide Time: 17:25)



```
            //ARG=SP-n-5

          fprintf(outputFile, "@SP\n");
          fprintf(outputFile, "D=M\n");
          fprintf(outputFile, "@%s\n",symbol3);
          fprintf(outputFile, "D=D-A\n");
          fprintf(outputFile, "@5\n");
          fprintf(outputFile, "D=D-A\n");
          fprintf(outputFile,"@ARG\n");
          fprintf(outputFile, "M=D\n");

            // LCL = SP
          fprintf(outputFile, "@SP\n");
          fprintf(outputFile, "D=M\n");
          fprintf(outputFile,"@LCL\n");
          fprintf(outputFile, "M=D\n");

            //goto F

          fprintf(outputFile, "@%s\n",symbol2);
          fprintf(outputFile, "0;JMP\n");

            //(return address)
          fprintf(outputFile,"(CLabel%s_%d)\n",filename,
*ALabelNum);
          *ALabelNum += 1;
        }

      // The return

        if (strcmp(symbol1,"return") == 0)
```

So this is how we implement and then we also generate a label for the return address. So this is all part of your calling function. Similarly, when we do the return so rate is star frame minus 5, okay. So the same thing, this is frame. So I also do star ARG equal to pop, SP equal to ARG plus 1, THAT equal to frame minus 1 and so on, right? So this is the return, right?

So this is how we implement call and return and function declaration which are the three important commands of your function call for the virtual machine, right? So this also explains how the whole thing works. So when a calling function calls, its responsibility is to push all the arguments and execute that statement call which in turn essentially translates to pushing a return address and saving all the context like the local, argument, and THIS and THAT of that.

And setting of the argument and local segments for the called function and then jump to the call function. So when the call function starts executing it will first go and initialise all the local parameters as K. It will have its own LCL, etc. So those are all already set. So it starts working on that. At the end of this it executes a return wherein all the local, argument and THIS and THAT of your previous calling function is set back and then you return back.

So when the calling function comes to called function it sets all the parameters for it and then transfers control to the call function. Similarly, when the called function returns back to the calling function it resets all the parameters as per the calling function and then it transfers to control back. So this is how the three program and the function call commands work in the virtual machine.

Now we will go on to the next module where we will show some examples of execution of this virtual machine commands and then we will proceed to go and do the project 8 and we will see what are the things involved in that. Okay, thank you.