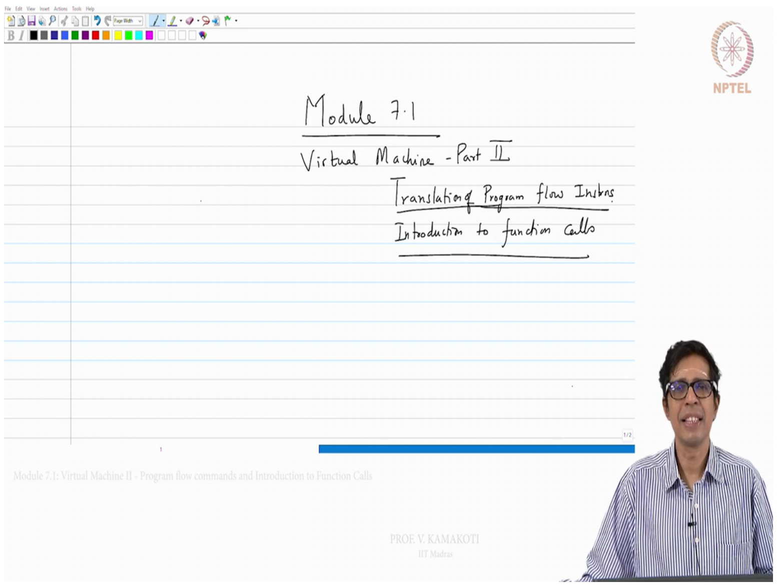
Foundations to Computer Systems Design
Professor V. Kamakoti
Department of Computer Science and Engineering
Indian Institute of Technology Madras
Module 7.1
Virtual Machine II - Program flow commands and Introduction to Function Calls

So welcome to module 7.1 and in this module we will be talking about virtual machine - the second part. So in the last module we did talk about two classes of instructions, the stack charismatic instruction and then the memory access instructions. And we also did a demo of the project 7. So in this module we will go in to the next two classes of instruction namely the program flow instruction and the function calls.

(Refer Slide Time: 00:44)



Module 7.1

Virtual Machine - Part II

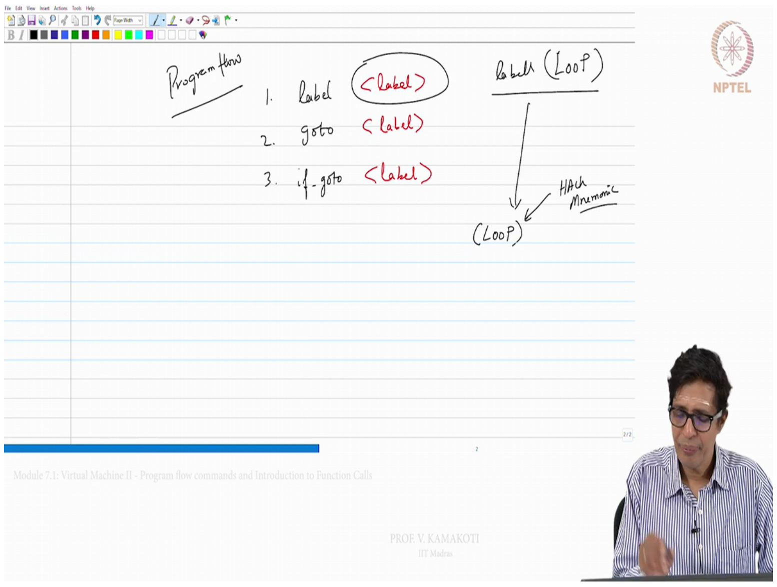
Translating Program flow Instrns

Introduction to function Calls

There are three programs flow instructions and three function call instructions as a part of your stack based virtual machine and what will be covering now is all those three in this entire module and we will also be finishing of project 8 by the end of this module. Now first the easiest of the lot are what we call as the program flow instructions, right?

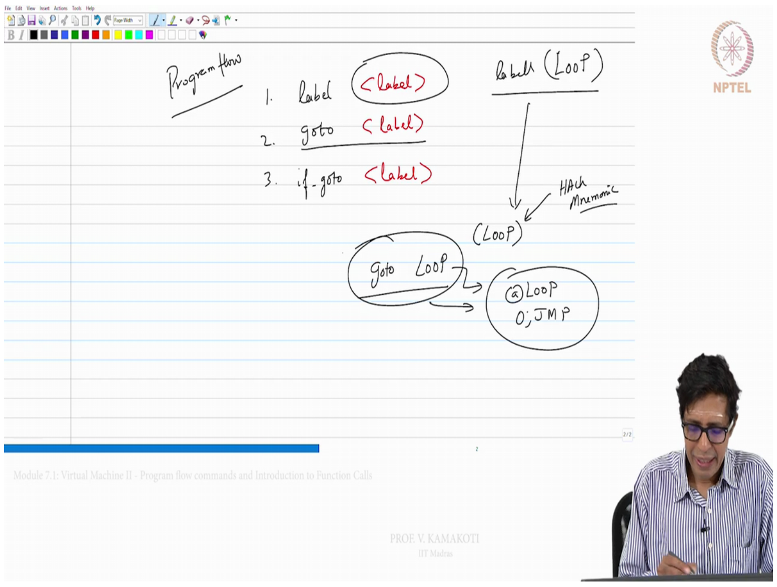
So the program flow instructions basically have three numbers, one is label and they give a label for example label loop, right? So how do we translate this? This is very straight forward. We have to just replace it by loop. This is the hack mnemonic and the assembler will take care of resolving this. It will treat like a symbol and symbol table, etc.

(Refer Slide Time: 01:43)



So the next thing is go to label. Go to say some loop now or go to loop. So how we read this? So we should say at loop. It is an unconditional statement so we say at loop and then say 0 colon jump. This is the actual translation of this go to loop statement, right?

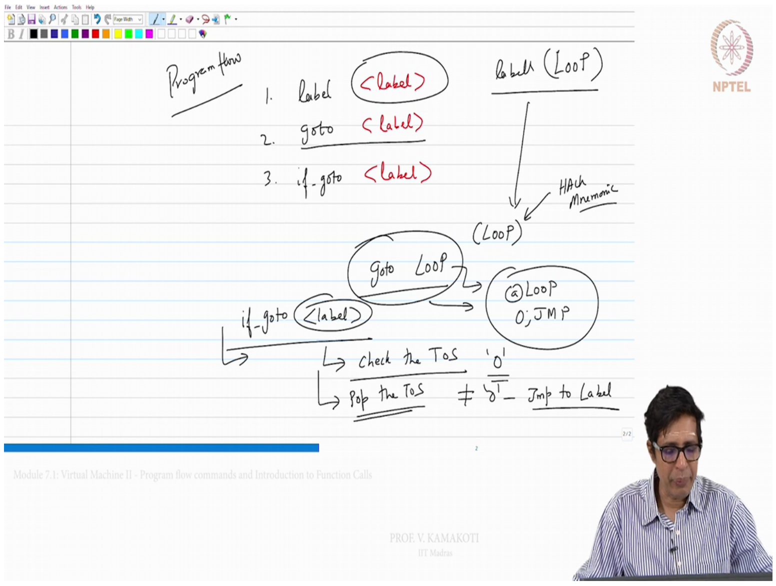
(Refer Slide Time: 02:13)



And then if underscore go to some label. So this is just slightly interesting stuff. What it does is this will check the top of the stack. If the top of the stack is having 0 then it will do nothing. If the top of the stack is not equal to 0 then it will jump to the label, whatever label you have given here. So this will jump to the label if the top of the stack is not equal to 0. The top of the stack is equal to 0 then it just proceeds to the next instruction.

So this is the conditional statement, right? If the top of the stack is non zero then it is going to jump to the label. So this is if underscore go to statement. And importantly it will also pop the top of the stack, okay, right?

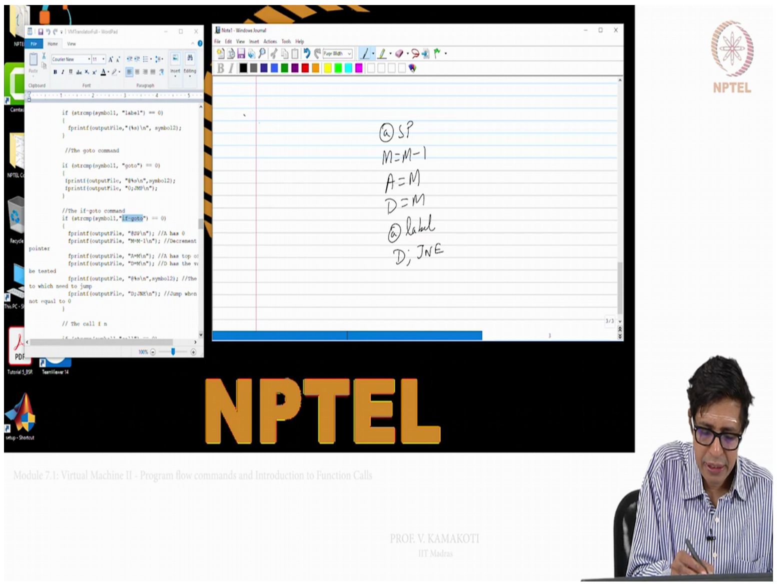
(Refer Slide Time: 03:20)



So this can be very easily implemented. Let me just go through the implementation. So for this which we have done as a part of project 8, so as you see here, right? As you see here if it is just a label we are just printing it as a semicolon followed by that label. If it is go to statement, go to some symbol so we are putting that symbol here at and doing 0 colon jump. If it is not underscore it is if hyphen go to then what we do?

So what are the commands that are here? We will just copy these commands and then explain you in detail. For if hyphen go to, just copy this. So at SP M equal to M minus 1, A equal to M, D equal to M, at whatever label, D semicolon J N E and that is it. Yes, that is it.

(Refer Slide Time: 05:02)



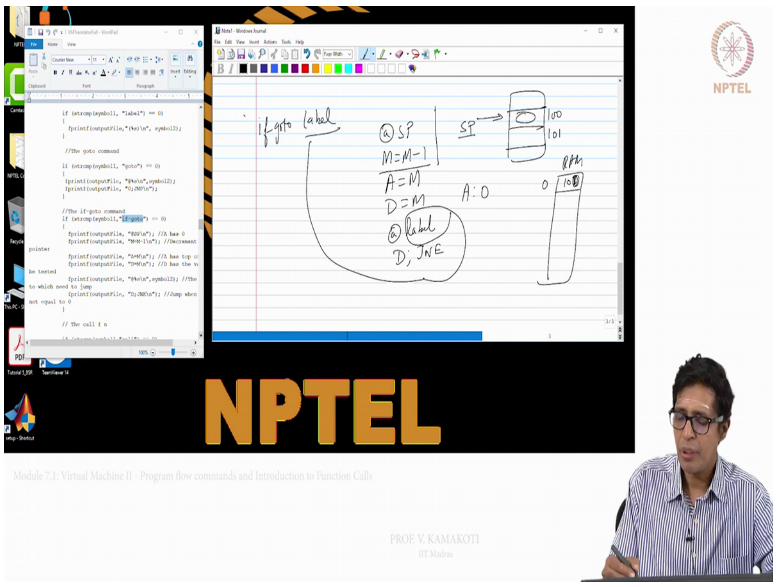
```
    if (strcmp(symbol1, "label") == 0)
    {
      fprintf(outputFile,"(%s)\n", symbol2);
    }

     //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
```

@SP
M=M-1
A=M
D=M
@label
D; JNE
NPTEL
Module 7.1: Virtual Machine II - Program flow commands and Introduction to Function Calls
PROF. V. KAMAKOTI
IIT Madras

So this is what is if hyphen go to some label. The same label is copied here. So what it does at SP? At SP, so the stack pointer will be pointing to the empty location. There will be some location on here. I have to check if this location, say this is 101. I have to check if 100 is non zero. If it is non zero, I need to jump. If it is 0 I need to go to the next instruction.

So there is SP in our RAM. SP is in 0 so 0 will be having, this is RAM 0, 0 will be pointing to 101, right? So at SP means A will get 0. Then M equal to M minus 1, so M of 0 is equal to M of 0 minus 1 so this 101 will become 100, right? So now your stack pointer will now start pointing. Anyway we are going to pop this out so stack pointer will be pointing to this 100.

(Refer Slide Time: 06:05)

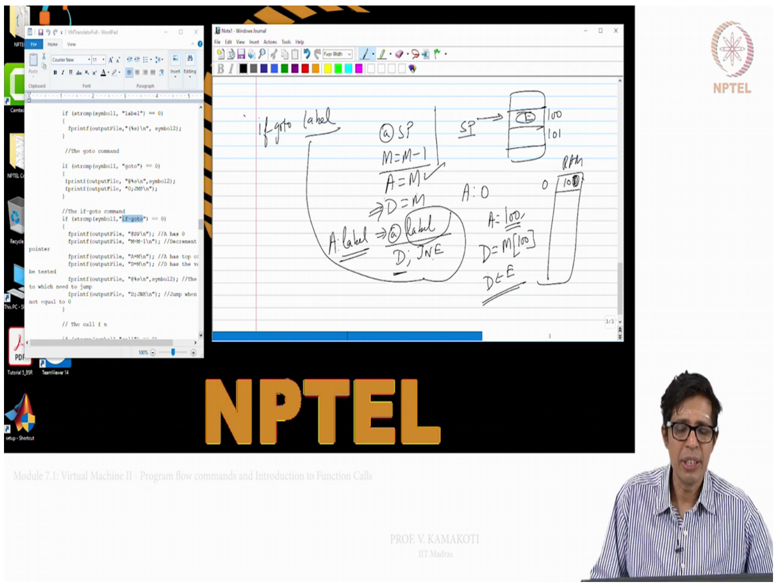


```
    if (strcmp(symbol1, "label") == 0)
    {
      fprintf(outputFile,"(%s)\n", symbol2);
    }

     //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
```

Now A equal to M of 0. So A equal to M of 100 which will be that value, right? A is equal to M of 0 so A will get 100. A will get 100 and D is equal to, this is what? Now D equal to M so D equal to M of 100 now. So and this D will get the value so whatever value here some element E. So D will get the value E here, right? To repeat my SP has now become 100. After that I say A equal to M, that means A equal to M of 0. M of 0 contains 100 so A becomes 100.

The next statement is D equal to M. D will be D equal to M of 100 now. Whatever is there in the A register is what is used. D equal to M of 100 so D will actually get an element. Now I said at label whatever this label, so A will get this label now at this instruction. Now if D is not equal to 0, jump to whatever is stored in label so it will jump. Otherwise it will continue.

(Refer Slide Time: 07:22)

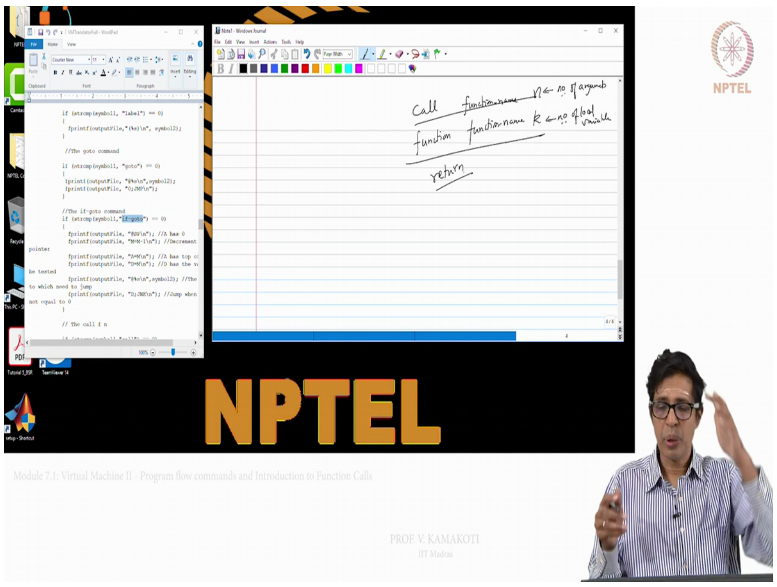


```
        if (strcmp(symbol1, "label") == 0)
        {
          fprintf(outputFile,"(%s)\n", symbol2);
        }

         //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 this is how if hyphen go to label is basically implemented. So what we have seen is there are three program flow instructions and all these three program flow instructions are very trivial and this can be implemented like this. This is again a small revision if you have forgotten how to write equivalent assembly code. This is a small example of how we can code some ideas in to hack assembly, right? Okay, now the next set of instructions that we need to do is about function calls, right?

So there are three instructions in function calls namely the call, function name and number of arguments. We call it as number of arguments that you are passing. Function, this is the function declaration, function name and in the book this is k and this is n. Okay, we will follow the same thing here. N is the number of arguments in the call, k is the number of local variables that we are going to use, right? So these two are there and of course there is something called the return, okay.

(Refer Slide Time: 08:45)



```
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
```

Call functionname $n \leftarrow$ no. of arguments

function functionname $k \leftarrow$ no. of local variable

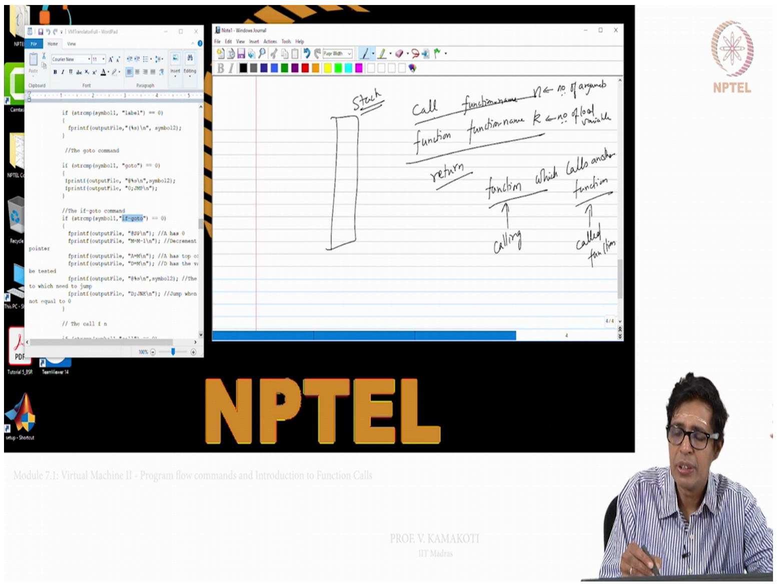
return

NPTEL
Module 7.1: Virtual Machine II - Program flow commands and Introduction to Function Calls
PROF. V. KAMAKOTI
IIT Madras

Now to understand and appreciate how this call function and return works it is very important that we should also understand how programming languages handle function calls. So we will now take an example of function calls are handled in see the same thing for many of the programming language or almost all of the programming languages. So the programming language actually uses something called a stack. The system has a stack, right?

So there is a function which calls another function. We have seen many examples of this. You would have seen many example for this in your preliminary course itself. So there is a function which calls another function. This function is called the calling function and this is calling some other function that fellow is called the called function.

(Refer Slide Time: 09:57)



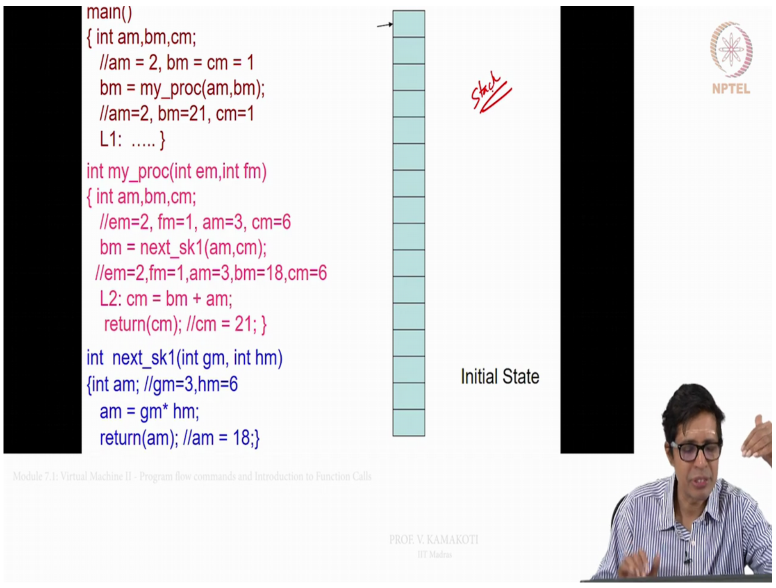
```
if (strcmp(symbol1, "label") == 0)
{
  fprintf(outputFile,"(%s)\n", symbol2);
}

 //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
```

Stack
Call functionname n ← no. of arguments
function functionname k ← no. of local variable
return
function which calls another function
calling
called function
NPTEL
Module 7.1: Virtual Machine II - Program flow commands and Introduction to Function Calls
PROF. V. KAMAKOTI
IIT Madras

Now let us look at this command call. Call is responsible for a transition from the calling function to the called function. This call is responsible and for the transition from the called function back to the calling function the return is responsible, right? So when you want to implement call and return as a part of your translation, we need to understand how the calling function transfer control to the called function and how called function returns back to the calling function, right?

Now before proceeding further we will actually give an example of taking some code in C and show how these things are managed and what do you mean by transferring control from calling function to called function and what it means to return back the control. And all these things you will see through a very simple example, right? What you see on the right hand side is a stack. Why do we use a stack? The thing will become clear in a while.

(Refer Slide Time: 11:26)



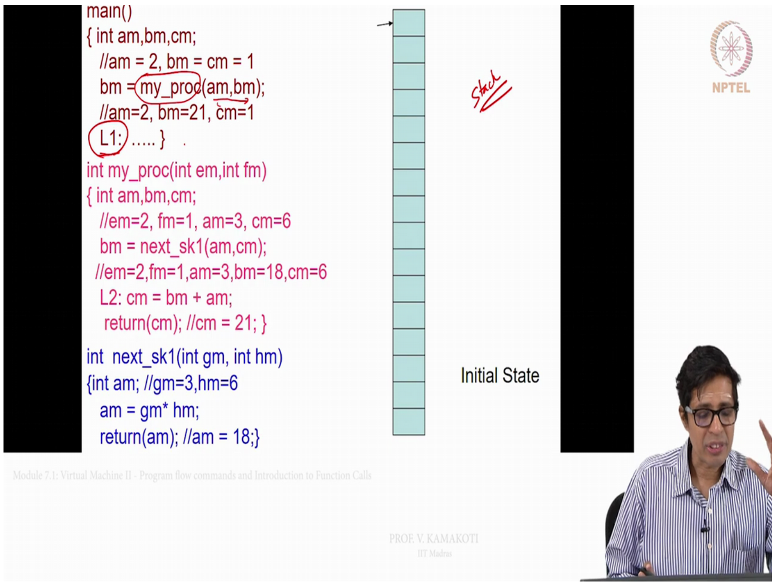
```
main()
{ int am,bm,cm;
  //am = 2, bm = cm = 1
  bm = my_proc(am,bm);
  //am=2, bm=21, cm=1
  L1: ..... }
int my_proc(int em,int fm)
{ int am,bm,cm;
  //em=2, fm=1, am=3, cm=6
  bm = next_sk1(am,cm);
  //em=2,fm=1,am=3,bm=18,cm=6
  L2: cm = bm + am;
   return(cm); //cm = 21; }
int  next_sk1(int gm, int hm)
{int am; //gm=3,hm=6
  am = gm* hm;
  return(am); //am = 18;}
```

Stack
Initial State
NPTEL
Module 7.1: Virtual Machine II - Program flow commands and Introduction to Function Calls
PROF. V. KAMAKOTI
IIT Madras

Now let us see main. So it is a C program so there is main. The main actually has certain local variables am, bm and cm and now main is basically calling my proc, right? This is a function that it is calling. So main is the calling function. My proc is the called function. To my proc this basically sends certain arguments. My proc returns a value which it assigns to main. So there are some arguments passed to my proc and there is a return value that my proc returns that comes back to main.

And after my proc finishes its execution and returns a value it should start again back with L1. So there is a return address. So what happens is, the calling function basically makes a call after pushing the arguments into the stack. It provides the arguments and then it calls my proc. After my proc finishes, my proc returns a value. After that it starts exactly at the point where it left. So after calling my proc it comes here, after my proc returns it comes and starts again from L1, okay, right?

(Refer Slide Time: 12:53)



```
main()
{ int am,bm,cm;
  //am = 2, bm = cm = 1
  bm = my_proc(am,bm);
  //am=2, bm=21, cm=1
L1: ..... }
int my_proc(int em,int fm)
{ int am,bm,cm;
  //em=2, fm=1, am=3, cm=6
  bm = next_sk1(am,cm);
 //em=2,fm=1,am=3,bm=18,cm=6
  L2: cm = bm + am;
   return(cm); //cm = 21; }
int  next_sk1(int gm, int hm)
{int am; //gm=3,hm=6
  am = gm* hm;
  return(am); //am = 18;}
```

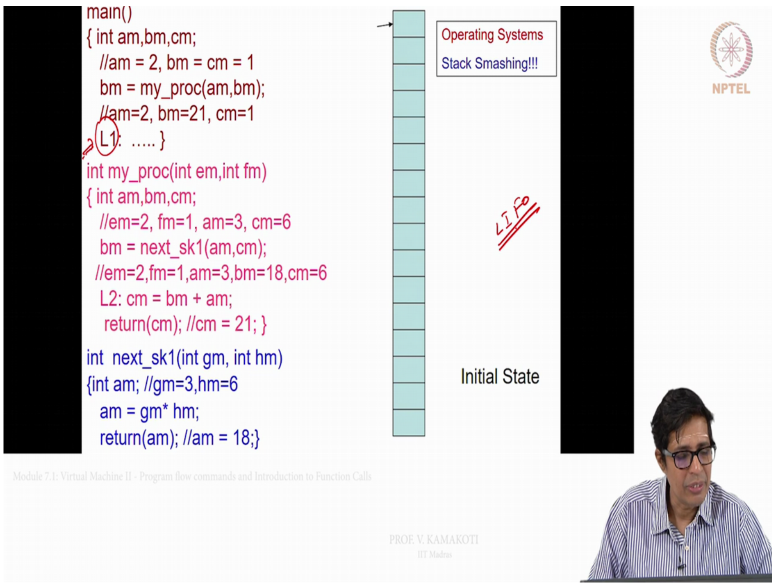
Stack
Initial State
NPTEL
Module 7.1: Virtual Machine II - Program flow commands and Introduction to Function Calls
PROF. V. KAMAKOTI
IIT Madras

So as far as main is concerned it has certain local variables and it has certain arguments that it is providing for my proc. Now my proc basically takes those arguments. It has its own local variables. It does set a computation, it finishes and returns back to L1, right? Returns back to L1 because main has called. Now when my proc is executing now you see it is calling another function called next sk1 for which it is providing parameters arguments and what happens is now next sk1 executes and it gives back an answer back to my proc and it resumes its execution for my proc at L2.

So main starts executing then it goes to my proc, my proc starts executing then it goes to next sk1, next sk1 does the execution, it finishes, its returns a value and starts executing from L2, again now my proc completes its execution and returns back to L1 and then main start executing again from L1. So who finished first? Main did not finish, my proc did not finish first, my proc in turn called the next sk1.

Next sk1 finished first and then it was my proc and then probably its main, okay. So the last in is the guy who finish the first. And that is how the stack also works, right? So stack also works on what we call as the last in first out policy and that is why stack is being used for all these computations, right?

(Refer Slide Time: 14:51)



```
main()
{ int am,bm,cm;
  //am = 2, bm = cm = 1
  bm = my_proc(am,bm);
  //am=2, bm=21, cm=1
  L1: ..... }
int my_proc(int em,int fm)
{ int am,bm,cm;
  //em=2, fm=1, am=3, cm=6
  bm = next_sk1(am,cm);
 //em=2,fm=1,am=3,bm=18,cm=6
  L2: cm = bm + am;
   return(cm); //cm = 21; }
int  next_sk1(int gm, int hm)
{int am; //gm=3,hm=6
  am = gm* hm;
  return(am); //am = 18;}
```

Operating Systems
Stack Smashing!!!

Module 7.1: Virtual Machine II - Program flow commands and Introduction to Function Calls

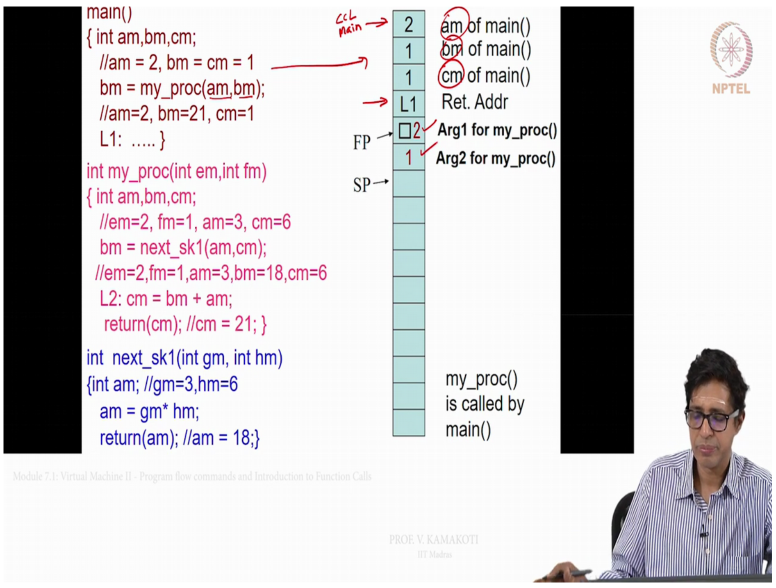
PROF. V. KAMAKOTI
IIT Madras

Now what happens? Initially the stack is empty. Now when main starts executing there is a space allocated for the local variables of main namely am, bm and cm.

(15:00)

So we are assigning am equal to 2, bm equal to 1 and cm equal to 1. So these are the local variables of main, right? Now when main calls my proc there is a return address and then it puts in certain arguments for this. So what are the arguments? So it is basically passing am and bm where am is 2 and bm is 1. So it passes on 2 and 1, right?

(Refer Slide Time: 15:56)



```
main()
{ int am,bm,cm;
  //am = 2, bm = cm = 1
  bm = my_proc(am,bm);
  //am=2, bm=21, cm=1
  L1: ..... }
int my_proc(int em,int fm)
{ int am,bm,cm;
  //em=2, fm=1, am=3, cm=6
  bm = next_sk1(am,cm);
 //em=2,fm=1,am=3,bm=18,cm=6
  L2: cm = bm + am;
   return(cm); //cm = 21; }
int  next_sk1(int gm, int hm)
{int am; //gm=3,hm=6
  am = gm* hm;
  return(am); //am = 18;}
```

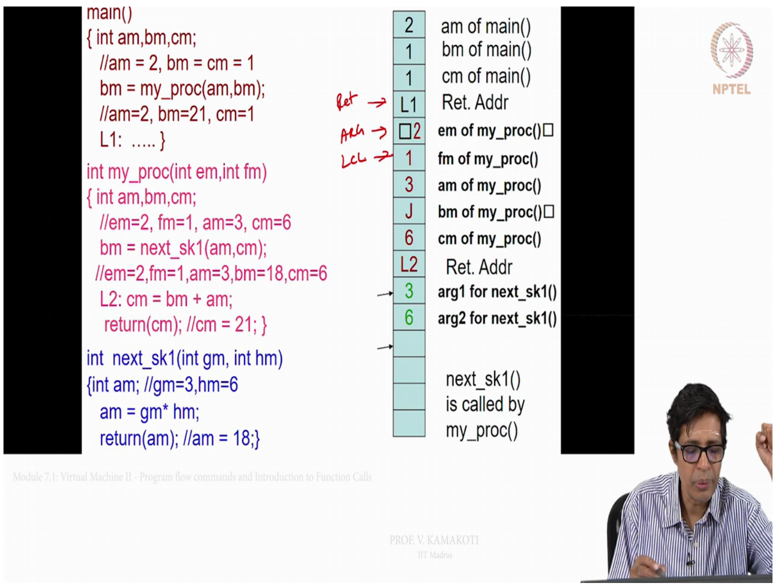
Module 7.1: Virtual Machine II - Program flow commands and Introduction to Function Calls

PROF. V. KAMAKOTI
IIT Madras

So when this call is executing, it stores the return address on the stack and then it basically passes the arguments 2 and 1, right? So this is now the argument section for my proc, let me say mp. This is the argument section for mp. And this is the return address for mp, right? And your stack pointer is here. This is what has happened when the call actually happens. Now what has happened when my proc starts executing, it now has three local variables.

So, when my proc starts executing there will be three local variables for my proc. So this will be your return address, this will be your argument and this will be your local address for my proc.

(Refer Slide Time: 17:06)



```
main()
{ int am,bm,cm;
  //am = 2, bm = cm = 1
  bm = my_proc(am,bm);
  //am=2, bm=21, cm=1
  L1:  ..... }
int my_proc(int em,int fm)
{ int am,bm,cm;
  //em=2, fm=1, am=3, cm=6
  bm = next_sk1(am,cm);
 //em=2,fm=1,am=3,bm=18,cm=6
  L2: cm = bm + am;
   return(cm); //cm = 21; }
int  next_sk1(int gm, int hm)
{int am; //gm=3,hm=6
  am = gm* hm;
  return(am); //am = 18;}
```

2	am of main()
1	bm of main()
1	cm of main()
L1	Ret. Addr
□2	em of my_proc()□
1	fm of my_proc()
3	am of my_proc()
J	bm of my_proc()□
6	cm of my_proc()
L2	Ret. Addr
3	arg1 for next_sk1()
6	arg2 for next_sk1()

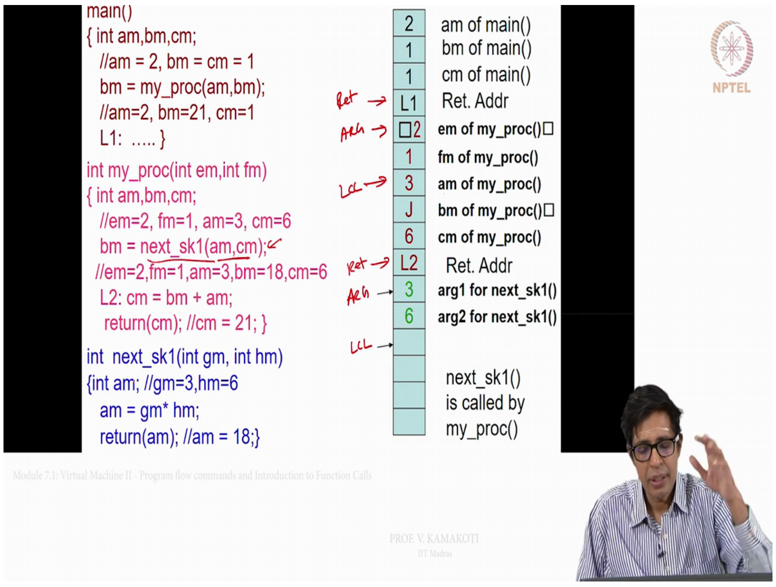
next_sk1() is called by my_proc()

Ret
ARG
LCL
NPTEL
Module 7.1: Virtual Machine II - Program flow commands and Introduction to Function Calls
PROF. V. KAMAKOTI
IIT Madras

So my proc will start executing. There will be a space allocated for fm. There will be a place allocated so this will be the local. There will be a space allocated for am, bm and cm of my proc. So this is the local, right? And then now my proc in turn now calls next sk1.

So this is the return address for that and it passes through parameter to arguments am and cm, am is 3 and cm is 6 when this is happening. So this will be ARG is 3 and 6 for this and then you will have basically what, LCL. This is the local for next sk1, right?

(Refer Slide Time: 18:07)



```
main()
{ int am,bm,cm;
  //am = 2, bm = cm = 1
  bm = my_proc(am,bm);
  //am=2, bm=21, cm=1
  L1: ..... }
int my_proc(int em,int fm)
{ int am,bm,cm;
  //em=2, fm=1, am=3, cm=6
  bm = next_sk1(am,cm);
 //em=2,fm=1,am=3,bm=18,cm=6
  L2: cm = bm + am;
   return(cm); //cm = 21; }
int next_sk1(int gm, int hm)
{int am; //gm=3,hm=6
  am = gm* hm;
  return(am); //am = 18;}
```

	2	am of main()
	1	bm of main()
	1	cm of main()
Ret →	L1	Ret. Addr
Arg →	□2	em of my_proc()□
	1	fm of my_proc()
LCL →	3	am of my_proc()
	J	bm of my_proc()□
	6	cm of my_proc()
Ret →	L2	Ret. Addr
Arg →	3	arg1 for next_sk1()
	6	arg2 for next_sk1()
LCL →		

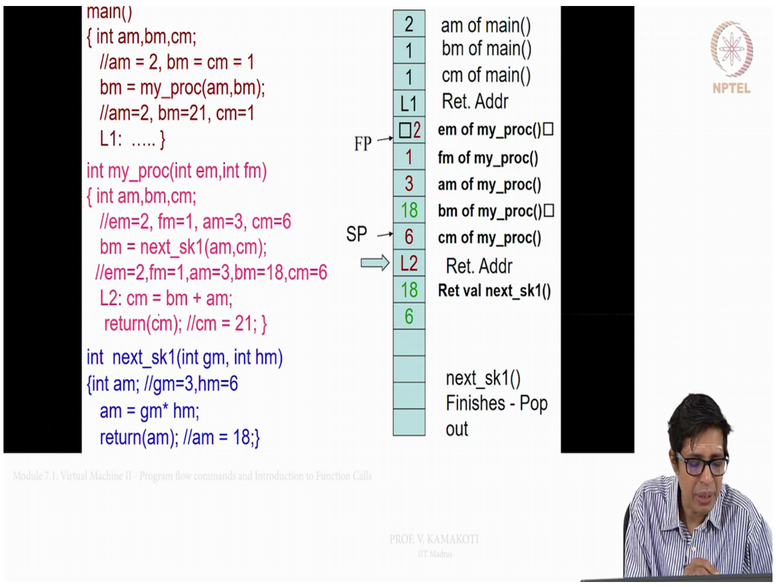
next_sk1() is called by my_proc()

NPTEL
Module 7.1: Virtual Machine II - Program flow commands and Introduction to Function Calls
PROF. V. KAMAKOTI
IIT Madras

Now next sk1 thus completes, it finishes, right? It finishes so what it will do? So that finished value it will go and store somewhere. So it goes and stores. So the finished value it goes and stores at this point. So now when it is finishing this is all not anymore necessary because the function finishes all the local variables, etc. will go. So it will go and store its value in this first of the argument.

So what happens is if you load here at this point you will get 18 next to L2. And then it will take the return address and it will go to return address. So it will start executing from L2. Which will start executing? My proc will start executing now.

(Refer Slide Time: 19:10)



```
main()
{ int am,bm,cm;
  //am = 2, bm = cm = 1
  bm = my_proc(am,bm);
  //am=2, bm=21, cm=1
  L1: ..... }
int my_proc(int em,int fm)
{ int am,bm,cm;
  //em=2, fm=1, am=3, cm=6
  bm = next_sk1(am,cm);
 //em=2,fm=1,am=3,bm=18,cm=6
  L2: cm = bm + am;
   return(cm); //cm = 21; }
int next_sk1(int gm, int hm)
{int am; //gm=3,hm=6
  am = gm* hm;
  return(am); //am = 18;}
```

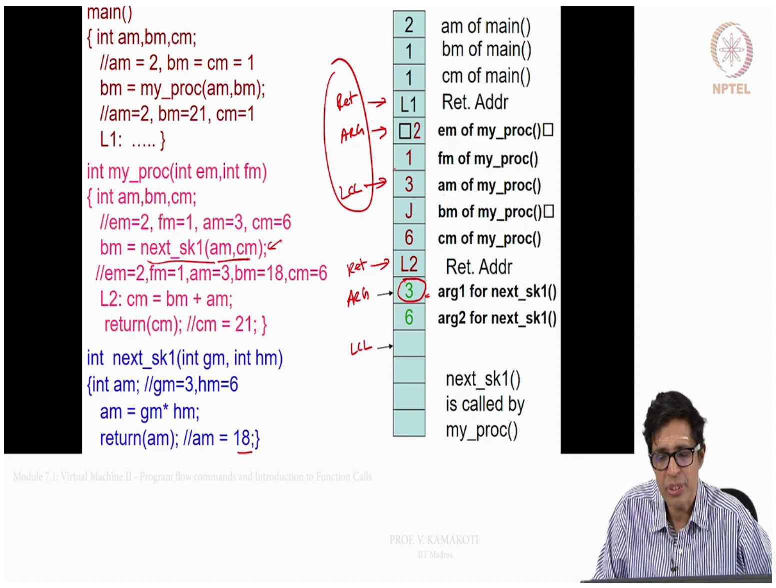
	2	am of main()
	1	bm of main()
	1	cm of main()
	L1	Ret. Addr
FP →	□2	em of my_proc()□
	1	fm of my_proc()
	3	am of my_proc()
	18	bm of my_proc()□
SP →	6	cm of my_proc()
⇨	L2	Ret. Addr
	18	Ret val next_sk1()
	6	

next_sk1() Finishes - Pop out

NPTEL
Module 7.1: Virtual Machine II - Program flow commands and Introduction to Function Calls
PROF. V. KAMAKOTI
IIT Madras

And when my proc starts executing what should happen? So let us just go back. So this was ARG and LCL for next sk1. Now when my proc start executing your return address, your ARG and LCL should now point to this sign. I need to go back to this stage, right? So then I will start executing my proc so I need to go back to this return ARG and LCL, right, because I may be using the argument again, but when I am using it for my proc I have to point to that. So this is basically called the context of your system.

(Refer Slide Time: 20:00)



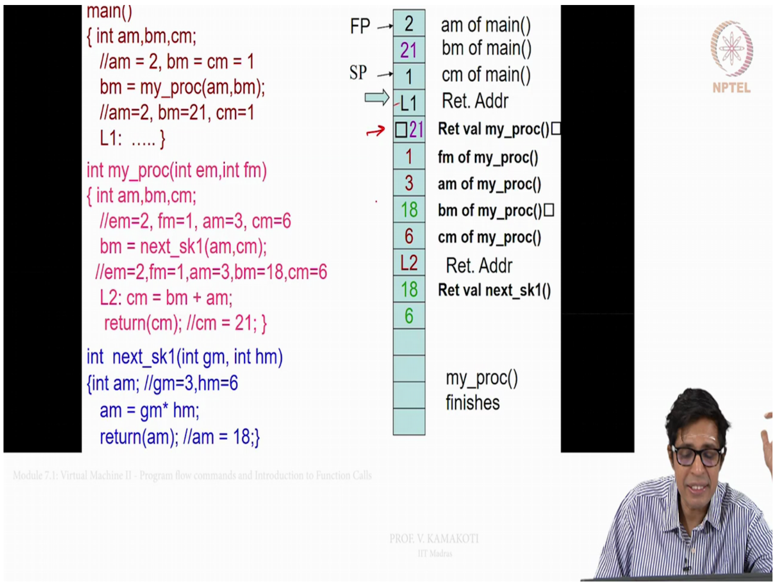
```
main()
{ int am,bm,cm;
  //am = 2, bm = cm = 1
  bm = my_proc(am,bm);
  //am=2, bm=21, cm=1
  L1:  ..... }
int my_proc(int em,int fm)
{ int am,bm,cm;
  //em=2, fm=1, am=3, cm=6
  bm = next_sk1(am,cm);
 //em=2,fm=1,am=3,bm=18,cm=6
  L2: cm = bm + am;
   return(cm); //cm = 21; }
int  next_sk1(int gm, int hm)
{int am; //gm=3,hm=6
  am = gm* hm;
  return(am); //am = 18;}
```

	2	am of main()
	1	bm of main()
	1	cm of main()
Ret →	L1	Ret. Addr
ARG →	□2	em of my_proc()□
	1	fm of my_proc()
LCL →	3	am of my_proc()
	J	bm of my_proc()□
	6	cm of my_proc()
Ret →	L2	Ret. Addr
ARG →	3	arg1 for next_sk1()
	6	arg2 for next_sk1()
LCL →		
		next_sk1()
		is called by
		my_proc()

NPTEL
Module 7.1: Virtual Machine II - Program flow commands and Introduction to Function Calls
PROF. V. KAMAKOTI
IIT Madras

The next my proc executes and it finishes and when it finishes whatever value it is going to return, it is going to return 21. That 21 it will go and put at this point if you notice just next to L1. So 21 is done. Now it will return to L1 and starts executing, okay, right? So this is how the function call is implemented.

(Refer Slide Time: 20:36)



```
main()
{ int am,bm,cm;
  //am = 2, bm = cm = 1
  bm = my_proc(am,bm);
  //am=2, bm=21, cm=1
  L1: ..... }
int my_proc(int em,int fm)
{ int am,bm,cm;
  //em=2, fm=1, am=3, cm=6
  bm = next_sk1(am,cm);
 //em=2,fm=1,am=3,bm=18,cm=6
  L2: cm = bm + am;
   return(cm); //cm = 21; }
int  next_sk1(int gm, int hm)
{int am; //gm=3,hm=6
  am = gm* hm;
  return(am); //am = 18;}
```

FP →	2	am of main()
	21	bm of main()
SP →	1	cm of main()
⇨	L1	Ret. Addr
→	21	Ret val my_proc()
	1	fm of my_proc()
	3	am of my_proc()
	18	bm of my_proc()
	6	cm of my_proc()
	L2	Ret. Addr
	18	Ret val next_sk1()
	6	
		my_proc() finishes

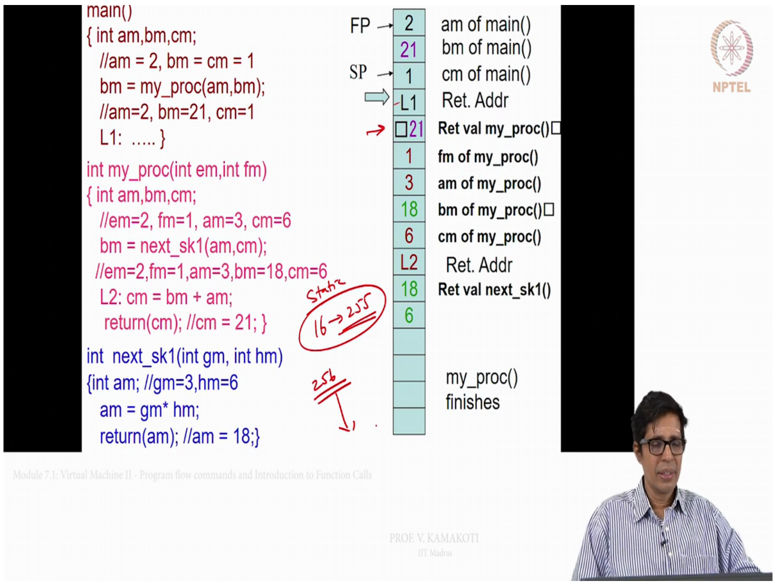
Module 7.1: Virtual Machine II - Program flow commands and Introduction to Function Calls

PROF. V. KAMAKOTI
IIT Madras

But one of the other thing that we need to keep in mind is for all the local variables of the function and all the arguments these are all basically loaded onto the stack, right? For certain elements which are going to be static, meaning they are not depending on the function, they need to exist across function boundaries.

Those are static variables which in our hack architecture will be allocated anywhere between RAM 16 to RAM 255 it will be allocated. But for all the local variables and other computations it will be allocated as part of the stack and your stack normal is starting from 256 to very larger address, right?

(Refer Slide Time: 21:26)



```
main()
{ int am,bm,cm;
  //am = 2, bm = cm = 1
  bm = my_proc(am,bm);
  //am=2, bm=21, cm=1
  L1: ..... }
int my_proc(int em,int fm)
{ int am,bm,cm;
  //em=2, fm=1, am=3, cm=6
  bm = next_sk1(am,cm);
 //em=2,fm=1,am=3,bm=18,cm=6
  L2: cm = bm + am;
   return(cm); //cm = 21; }
int  next_sk1(int gm, int hm)
{int am; //gm=3,hm=6
  am = gm* hm;
  return(am); //am = 18;}
```

Static
$16 \rightarrow 255$

256
$\downarrow$

FP →	2	am of main()
	21	bm of main()
SP →	1	cm of main()
⇨	L1	Ret. Addr
→	21	Ret val my_proc()
	1	fm of my_proc()
	3	am of my_proc()
	18	bm of my_proc()
	6	cm of my_proc()
	L2	Ret. Addr
	18	Ret val next_sk1()
	6	
		my_proc() finishes

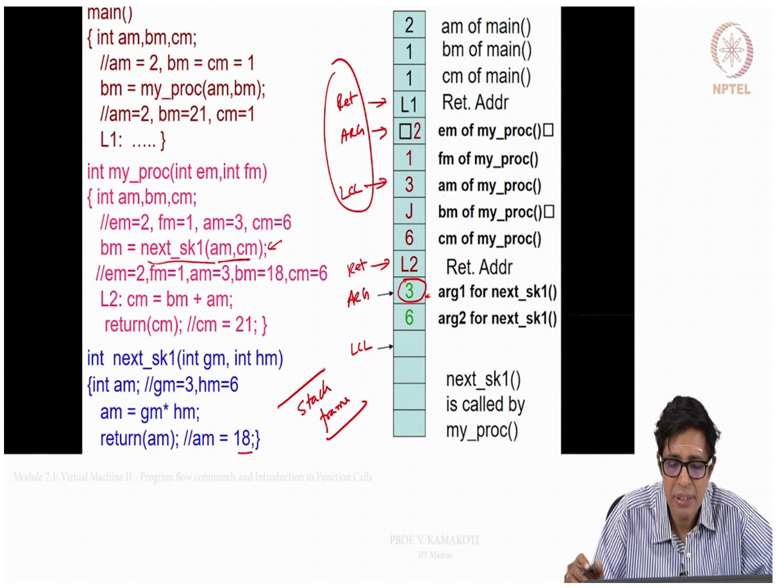
Module 7.1: Virtual Machine II - Program flow commands and Introduction to Function Calls

PROF. V. KAMAKOTI
IIT Madras

So when a function is executing what happens is there is a frame created. So let us just go back and see what has happened. When main starts executing there is a frame created where your local variables are there and when my proc starts executing again there is a frame which says where you want to return, what are your arguments, what are the local variables and so on.

So every time a function starts executing there is some amount of memory allocated to the function in the stack. That amount of memory that is allocated to the function in the stack is called the activation record of that function. It is the memory that is activating the function so this is called activation record or we also call it as a stack frame. This is the activation stack frame which is responsible for the execution of the function.

(Refer Slide Time: 22:32)



```
main()
{ int am,bm,cm;
  //am = 2, bm = cm = 1
  bm = my_proc(am,bm);
  //am=2, bm=21, cm=1
  L1: ..... }
int my_proc(int em,int fm)
{ int am,bm,cm;
  //em=2, fm=1, am=3, cm=6
  bm = next_sk1(am,cm);
 //em=2,fm=1,am=3,bm=18,cm=6
  L2: cm = bm + am;
   return(cm); //cm = 21; }
int  next_sk1(int gm, int hm)
{int am; //gm=3,hm=6
  am = gm* hm;
  return(am); //am = 18;}
```

2	am of main()
1	bm of main()
1	cm of main()
L1	Ret. Addr
2	em of my_proc()
1	fm of my_proc()
3	am of my_proc()
J	bm of my_proc()
6	cm of my_proc()
L2	Ret. Addr
3	arg1 for next_sk1()
6	arg2 for next_sk1()
	next_sk1() is called by my_proc()

Ret
ARG
LCL
Ret
ARG
LCL
Stack frame
NPTEL
Module 7.1: Virtual Machine II - Program flow commands and Introduction to Function Calls
PROF. V. KAMAKOTI
IIT Madras

Now another important thing is when main is executing there will be some space in the memory which is kept for its local variables, its arguments and the return address. When we move from main to my proc, for my proc the return argument and LCL are different. And when we move from my proc to next sk1 then again your return, argument, LCL are different.

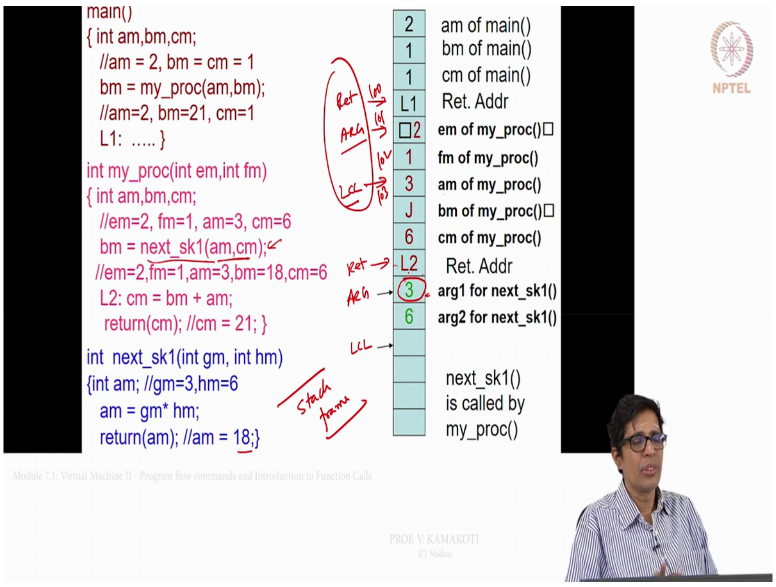
So as and when we move from one function to another, the return address, the arguments where they are stored on the stack and LCL changes. So when we go back, this is when I am calling, when the called function is finished and I am going back to the calling now what should be stored in ARG and LCL? I hope you remember what is ARG and LCL. They are there in the memory RAM 1 and RAM 2 which points to the arguments and the local variables of that current function.

So when I am executing main, the ARG and LCL of main should be there. When I am executing my proc, the ARG and LCL of my proc should be there in RAM 1 and RAM 2. So when I go back from next sk1 to my proc, my original value of the ARG and LCL should be retrieved, right? So now that is called the context of the process and we need to retrieve it.

So when I am going from a calling function to a called function, not only we will push the arguments and we will keep some space for the local variable, we will also go and save along with the return address the start of your argument segment and start of your local segment for the calling function. So when I am calling from let us say my proc to next sk1 what did I save? I just saved a return address and I put the argument.

Along with this return address I will also say let this be 100, 101, 102, 103, right? So let it be like this, right? So I will also go and save the Ret, ARG and LCL of my calling function so that when I complete I can basically retrieve that and start executing, right? So that is very important.

(Refer Slide Time: 25:11)



```
main()
{ int am,bm,cm;
  //am = 2, bm = cm = 1
  bm = my_proc(am,bm);
  //am=2, bm=21, cm=1
  L1: ..... }
int my_proc(int em,int fm)
{ int am,bm,cm;
  //em=2, fm=1, am=3, cm=6
  bm = next_sk1(am,cm);
 //em=2,fm=1,am=3,bm=18,cm=6
  L2: cm = bm + am;
   return(cm); //cm = 21; }
int  next_sk1(int gm, int hm)
{int am; //gm=3,hm=6
  am = gm* hm;
  return(am); //am = 18;}
```

2	am of main()
1	bm of main()
1	cm of main()
L1	Ret. Addr
□2	em of my_proc()□
1	fm of my_proc()
3	am of my_proc()
J	bm of my_proc()□
6	cm of my_proc()
L2	Ret. Addr
3	arg1 for next_sk1()
6	arg2 for next_sk1()

next_sk1() is called by my_proc()

Ret
ARG
LCL
100
101
102
103
Ret
ARG
LCL
stack frame
NPTEL
Module 7.1: Virtual Machine II - Program flow commands and Introduction to Function Calls
PROF. V. KAMAKOTI
IIT Madras

So this is how function calls work and function calls use the stack, right? So the stack is basically used by the function calls to allocated memory for all its local variables and then the return address. And when something happens, it basically happens on the stack and when it completes its execution it goes back to the calling function.

And every time I am executing a function, its local variable, its arguments, the base of that local and ARG should be saved in the ARG and LCL space in RAM 1 and RAM 2. And that is possible if I have these ARG and LCL also saved in the stack so that when I move from one function to another function I can copy that ARG and LCL of that function into the RAM 1 and RAM 2 so that there is a seamless execution of this.

So this is in some essence trying to basically explain you how function calls work, right? And we will use this concept that we have learnt to implement the three functions that we are going to talk of namely the called, return and function. Thank you. We will now meet in model 7.2.