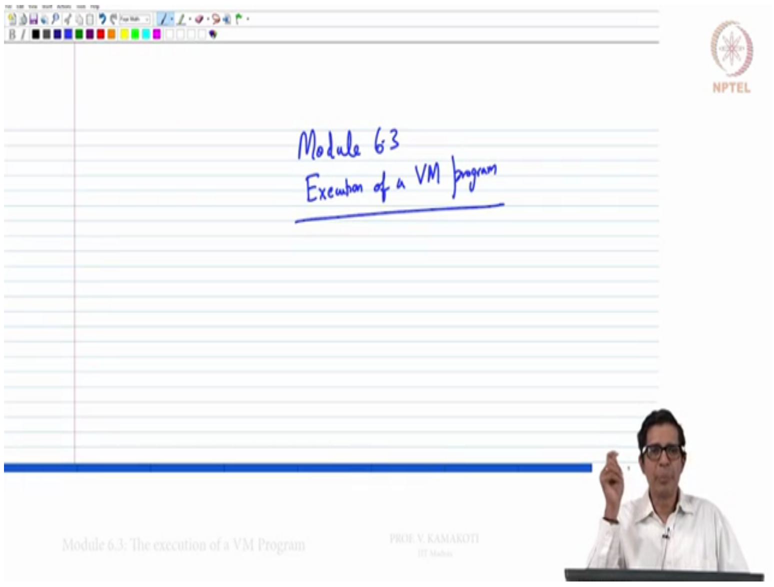
Foundations to Computer Systems Design
Professor V. Kamakoti
Department of Computer Science and Engineering
Indian Institute of Technology, Madras
Module 6.3
The execution of a VM Program

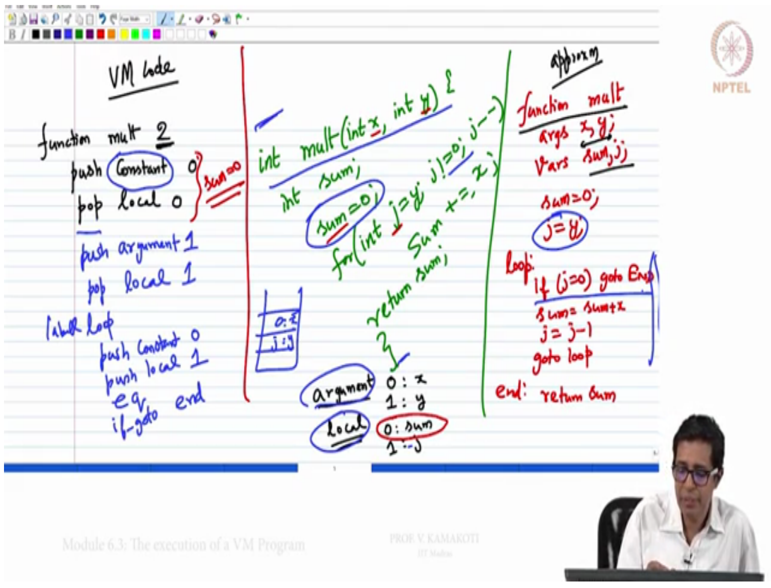
(Refer Slide Time: 0:16)



Module 6.3
Execution of a VM program
Module 6.3: The execution of a VM Program
PROF. V. KAMAKOTI
NPTEL

So welcome to module 6.3, in this module 6.3 we will do we will basically explain you the execution of a VM program, we will take an example very simple example of actually multiplying two numbers and then tell you how it gets converted to a VM program, how the C program is getting converted to a equivalent VM program, so when we understand that translation of how a C code is getting you know interpreted or compiled into a VM code and we will have very clear understanding of what each instruction is expected to deliver, what each instruction is expected to execute. So that complete understanding will come once we do this.

(Refer Slide Time: 1:18)


```
VM Code

function mult 2
    push constant 0
    pop local 0        } sum=0

    push argument 1
    pop local 1
label loop
    push constant 0
    push local 1
    eq
    if-goto end
```

```
int mult(int x, int y) {
    int sum;
    sum=0;
    for(int j=y; j!=0; j--)
        sum +=x;
    return sum;
}

argument  0 : x
          1 : y
local     0 : sum
          1 : j
```

```
approxm
function mult
    args x,y;
    vars sum,j;
    sum=0;
    j=y;
loop:
    if (j=0) goto End
    sum= sum+x
    j= j-1
    goto loop
end: return sum
```

So we take a very simple program which is written here, so I have int, mult int x comma int y so I want to multiply x and y and note that our (())(1:32) does not have a multi-player, right it only does addition. So how will I multiply x and y? I will add x, y times that is equivalent to x into y and that is given by int sum, sum equal to 0 for int j equal to y, j not equal to 0, j minus minus, sum equal to sum plus x return sum.

So this will basically this is a very simple C code that you see here in the middle in green and this basically multiplies x and y realizes the multiplication of x and y by adding x, y times. Now how will it, what is this equivalent of the C code now as I told you now there are two segments now there will be an argument segment which will have two entries, it is a memory segment stored somewhere in the memory say 1000 in 1000 there will be x and in 1001 there will be y.

Similarly there will be another local segments say it is stored in 2500, in 2500 it will have sum the local variables there are two local variables sum and j in (200) (())(2:47) 2000, in 2000 there will be sum and in 2001 there will be j, right. So this program that we are seeing as a C program basically has when you want to convert it to VM then we need to create segments one for argument, one for local and in that argument this is how the variables are basically mapped on to, okay.

So now what will be the approximate form of this VM, the approximate form of the VM for this the C routine is essentially function mult and then there are two arguments x and y and then there are two variables sum and j, sum equal to 0, j equal to y, loop if j is equal to 0 go to int otherwise sum equal to sum plus x, j equal to j minus 1 go back to loop, so you are seeing this happening, return sum.

So this whatever you are seeing on the right most side which I call it as approximate approximating the VM this whatever you see on the right hand side is much closer to the our VM instruction set, once I have this we can easily go and get translation of this approximate VM to the actual VM code.

Now what you will see is the actual VM code for this C routine and to understand the C routine when somebody is when the system is when the program is some other program is calling mult, it will create an argument segment in which there will be two entries and somebody is calling mult, so some other function is calling mult, so that function will be providing the values of x and y and that will be stored in arguments in the argument segment at 0 and 1 location, right and (the current) the function when it is executing will use the local segment for manipulating these sum and j which are local variables as far as this is concerned, right. So this is how this is the C code and this is the approximate VM.

Now we will do the VM code (())(5:13) right. Now first is function mult 2, right this is a function with 2 local variables so the equivalent VM code of this mult int x, int y will become function mult 2, then what we do? We push constant 0, so constant is another segment it is a virtual segment essentially say I can push constant 0. So the maximum I can push is 32767 because this is basically a 15 bit number, so push constant 0, right.

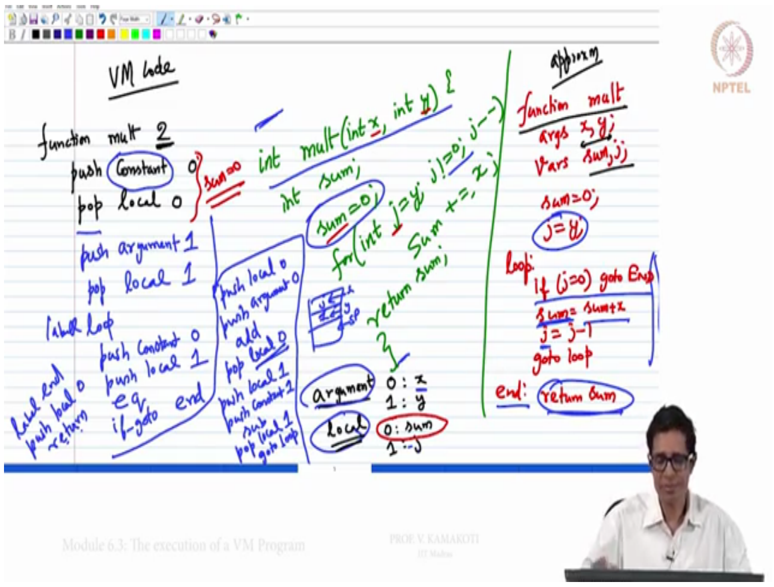
So what will happen on the top of the stack will be pushed into, when I pop the stack what will come out? 0 will come out, so when I say pop local 0, so in the local segment which is in the 0th location please look at here in the local segment 0, (sum is in the local location) sum is in the 0th location. So when I say pop local 0, right that means it will make sum is equal to 0 because I need to realize sum equal to 0, here and this 2 will totally help me realize sum equal to 0.

Now I will say push y, pop j and this will (sorry) what is push y? I want to make j equal to y, so I need to push y, what is push y? This is the first entry in your argument segment, 0th entry is x, first entry is 1. So push argument 1 that means it will go to the argument segment the frits location will be y, so it will push y and then I have to make j equal to y, so j is in local 1, so pop local 1, so your j actually becomes y.

So what are these four instructions that we have seen so far done, they have made sum equal to 0 and they have also made j equal to y. Now I am basically I am seeing this entry loop, so I need to say label loop, so this will create a label in the (())(7:55) so label loop. Now what I should say if j equal to 0, go to (())(8:07) so first I will say push constant 0, push local 1, what is push local 1? In the local segment which is the first entry? j is the first entry, 0 is the sum, 0th entry is sum, first entry is j.

So push local 1 means the value of j gets on to the stack, so what was j gets on to the stack. So what is now on the stack? I have 0 and I have the value of j, then I just say eq, note that eq is an arithmetic command, what will eq do? It will compare x and y, what is x? x is 0, y is j, it will compare x is equal to y and set the stack as 0 if x is not equal to y and 1 if x is equal to y. Now I say if go to end, that means if j is equal to 0 so this is done here, right.

(Refer Slide Time: 9:48)



Now the next one that we need to do is sum equal to sum plus x, how do you do that sum equal to sum plus x? Where is sum? It is in local segment 0, so I will just push local 0, so sum comes on top of the stack. Now I have to push argument 0, so x go to x argument 0 is x, right so x also goes on. Now I say add, now on top of the thing I have x plus sum and that should now go back to x, right.

So when I say (push local 0, x was in the stack), then I say push argument sorry push local 0, sum was in the stack, push argument 0, x was there. Now I say add, so I have now made sum plus x and this should be stored back in sum, right. So how do I store back it in sum? So again I say pop, where is the sum? Where is sum plus x? This is in the top of the stack, now I say pop local 0, so push local 0 will push x onto the stack, (push argument 0 will push) push local 0 will push sum onto the stack, push argument 0 will push x onto the stack. When I add this x plus sum will be stored on the top of the stack, when I pop x plus sum will come and that x plus sum will be stored in local of 0 on sum itself.

So sum will now get replaced by x plus sum and that is what we need to do, sum equal to sum plus x. Now I have to do j equal to j minus 1, where is j? j is in local so push local 1, right so then this means j gets loaded on the top of the stack this is the stack pointer j gets loaded. Now push constant 1, so 1 comes on the top of the stack and now (())(12:35) now I say subtract. Now when I say subtract, subtract x minus y, right. So x is j, y is 1, so I am doing (x minus j) j minus 1 here and this we need to, so now at the end of the sub j minus 1 will be on the top of the stack and now I have to assign it back to j, j equal to j minus 1.

So j minus 1 is on the top of the stack, now I have to assign it to j that means pop that value into local of 1 so this will take care of j equal to j minus 1 then I have go to loop so I will just say go to loop and there a label loop is already there and then what I am going to say here is there is end return sum, so this end essentially will translate it as label end, return sum is what? return sum is I have to push local 0 and return (sorry).

So return sum essentially means push the value onto local 0 which is sum and return, push the value of sum onto the stack and then return so this is how a C multiplication go essentially gets translated to our virtual machine code, right and now we have seen different segments, we have seen segment for argument, we have seen segment for local, we have also seen segment for constant like constant 0, etc and then the entire stack operation that is happening so with this.

So what we now see that we have seen some arithmetic logic operation here, we have seen certain memory access both push and pop on different varieties of segments and then we have also seen certain program flow instructions like label, loop, etc and then we are also and then go to loop etc and then we are also having some function calling commands like return and function. So these are some of the very interesting things that we are seeing, okay.

Now in the next module we will so hope you have an understanding of this so you can go through this lecture again to understand what is it and if you have some doubts you can certainly put back on the web on our discussion forum. Now what we will do in the subsequent two modules is that we will basically give some interesting examples of how we can transform certain VM code on to (())(16:30) was a power of the stack machines, that we will understand and then we will go ahead and look at the implementation of some of these concepts. So that will form the first basis of our virtual machine interpreter, thank you.