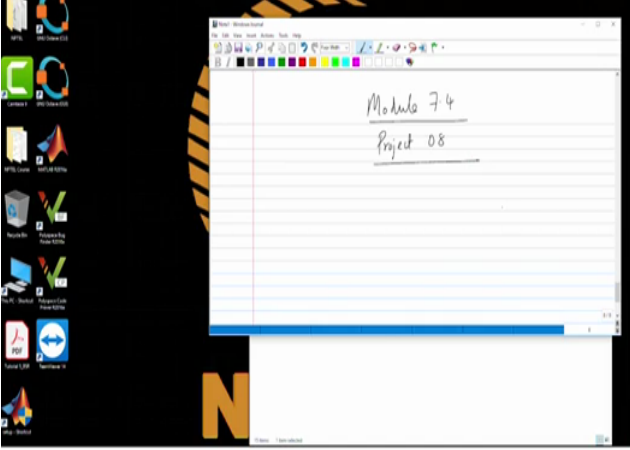


Foundations to Computer Systems Design
Professor V. Kamakoti
Department of Computer Science and Engineering
Indian Institute of Technology, Madras
Module 7.4

Project 08: Translation of Program Flow and Function Call to HACK Mnemonic

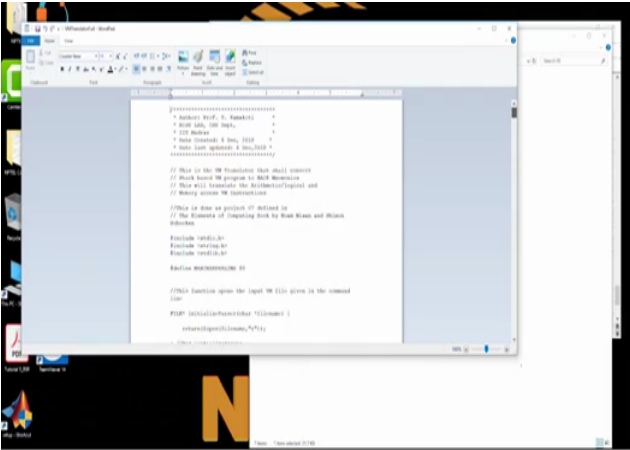

(Refer Slide Time: 0:24)



The slide displays a desktop environment with a Notepad window open. The window contains the handwritten text "Module 7.4" and "Project 08". The NPTEL logo is visible in the top right corner of the slide.

Module 7.4: Project 08: Translation of Program Flow and Function Call to HACK Mnemonic

PROF. V. KAMAKOTI
IIT Madras



The slide displays a code editor window showing C code for an emulator. The code includes comments and function definitions. The NPTEL logo is visible in the top right corner of the slide.

```
.....
+ Andrew Weil, V. Kamakoti
+ IIT Madras, IIT Madras
+ IIT Madras
+ Home: Bangalore, India, 560075
+ Email: kamakoti@iitmadras.ac.in
.....

// This is the HW Emulator that shall convert
// Your hand written program to HACK Mnemonic
// This will translate the Machine/Logical and
// Binary to the instructions

//This is how we project it defined as
// The Elements of computing that by Mark Weiser and Milton
// Weiser


#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>

//This function open the input file and give in the content
char *
FILE * fopen(const char * filename, const char * mode) {
    return fopen(filename, "r");
}

.....
```

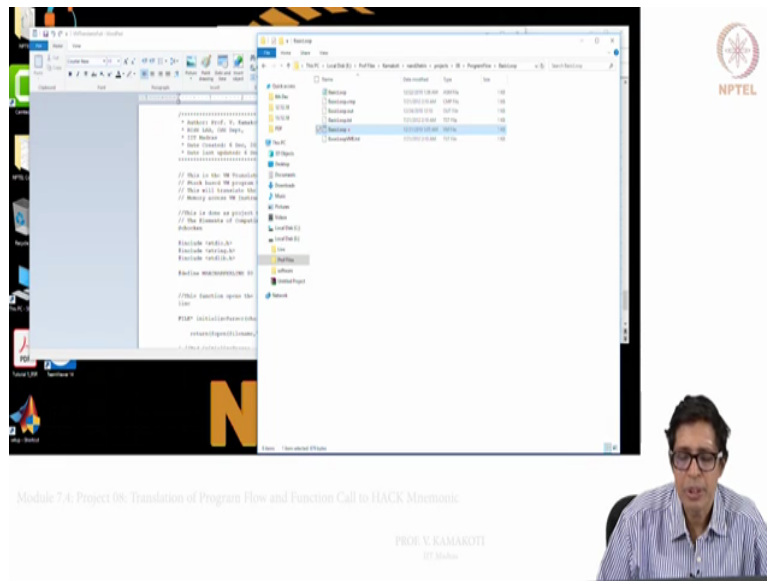
Module 7.4: Project 08: Translation of Program Flow and Function Call to HACK Mnemonic

PROF. V. KAMAKOTI
IIT Madras



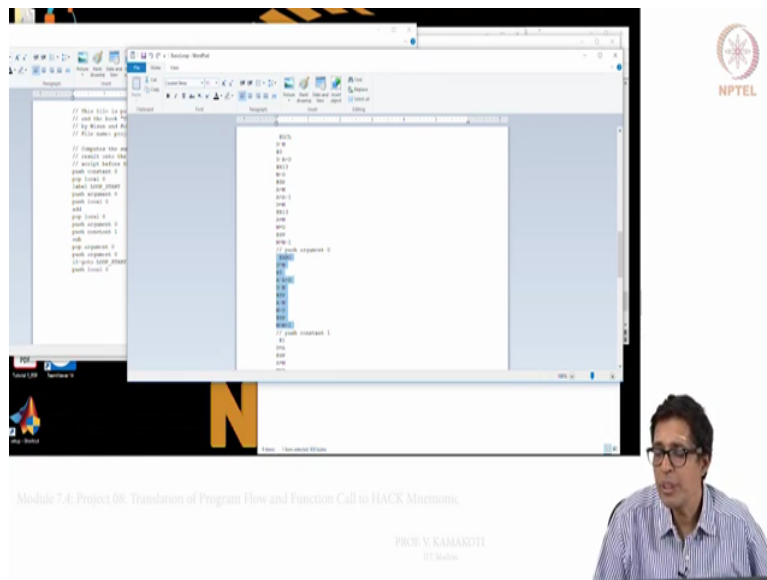
So welcome to module 7.4 in which we will basically do the project 08 and we will see how it is going to work. Now what we have done is we have created a file, we have written the complete emulator which you see here this is the entire emulator written, now this is I have written it in C, you can write it in C plus plus, you can compare this so you can get an executable out of this, right.

(Refer Slide Time: 0:54)



Module 7.4: Project 08: Translation of Program Flow and Function Call to HACK Mnemonic

PROF. V. KAMAROTI
IIT Madras



Module 7.4: Project 08: Translation of Program Flow and Function Call to HACK Mnemonic

PROF. V. KAMAROTI
IIT Madras

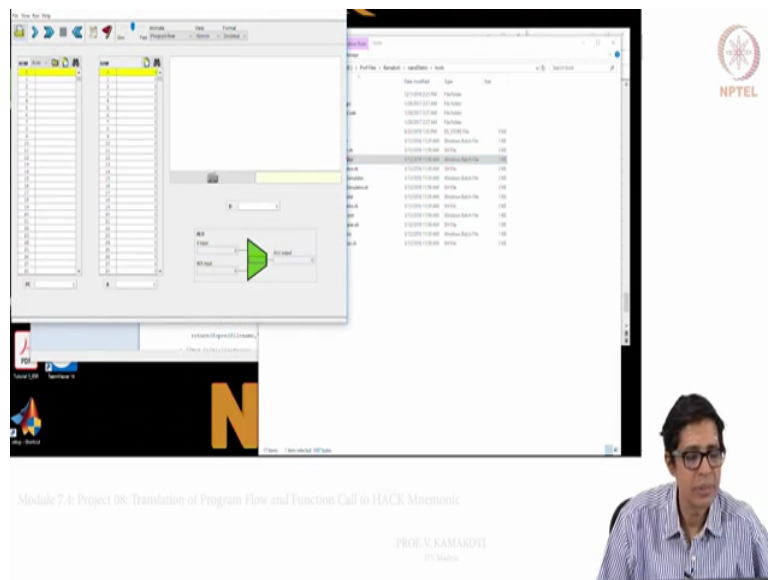
And what we need to do is now we go to these projects so we can do there are two so go to the (program loop) program flow there are basic loop now there is a VM file that you see here which is this basic loop dot VM, now you execute that program with this as the input basic loop dot VM. So it will generate the basic loop dot ASM file, so this is the VM file which you will open with notepad, this is the VM file and this is the and this is the corresponding ASM file this is the corresponding ASM file.

So I will put it side by side yeah, so push constant 0 so this is translated into set of 7 statements like pop, local 0 this is translated into a set of statements then add, right. So push constant 0, pop local 0, so every one of these statements will basically get executed, right

push constant 0, pop local 0, label loop start so it created a label, push argument 0 so at ARG, push local 0, etc.

So for every statement that you see basically you see the corresponding hack mnemonic, so this is the VM interpreter and this is how it is basically worked.

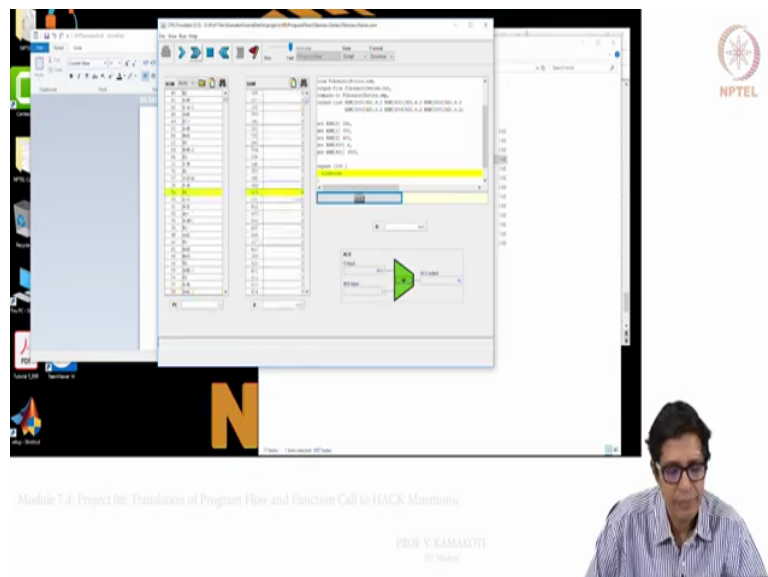
(Refer Slide Time: 2:50)



The screenshot displays the VM interpreter interface. On the left, there are two columns of assembly code. The right pane shows the current state of the VM, including the stack and memory. A video inset in the bottom right corner shows Prof. V. Kamakoti speaking. The NPTEL logo is visible in the top right corner.

Module 7.4: Project 08: Translation of Program Flow and Function Call to HACK Mnemonic

PROF. V. KAMAKOTI
IIT Madras



This screenshot shows the VM interpreter with a different view of the assembly code and VM state. The video inset of Prof. V. Kamakoti is present in the bottom right corner. The NPTEL logo is in the top right corner.

Module 7.4: Project 08: Translation of Program Flow and Function Call to HACK Mnemonic

PROF. V. KAMAKOTI
IIT Madras

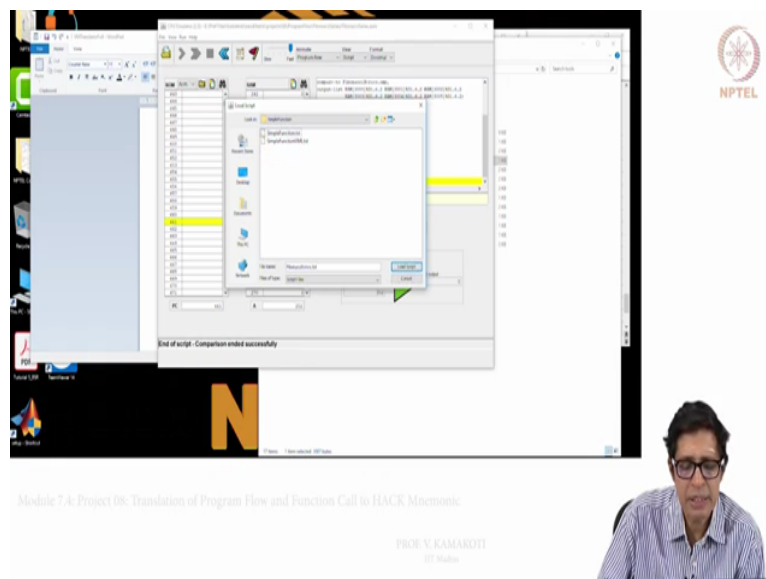
Now what we need to do at this point is basically now we will go back and in the tools we will basically invoke the CPU emulator, right. So in this CPU emulator basically now we will load the script so we will do everything now in terms of let us go to the you know project 8, now you can take all the program flow, we can load the basic loop that is a script corresponding to the basic loop dot test, okay.

We run this we can run this but this is very slow we can actually make it super-fast by making like this run for some 600 units of time and then you see here yeah comparison successful. So we will run one more to show you so we can just say load script we have done basic loop, so we can just say some program flow let us say Fibonacci series load script (())(4:36) then we will say run it with full speed.

So you can also single step and basically see at every stage what is happening. Now what is this script to do as you see while it is running, it is setting your RAM 0 to 256 it is setting, what is RAM 0? Your speed set to 256, your local argument is set to 300, then this is to 400, etc and then, right and in your stack it has already loaded in your argument it has already loaded 6 and 3000, so some of these initial settings it has and then we run the tool.

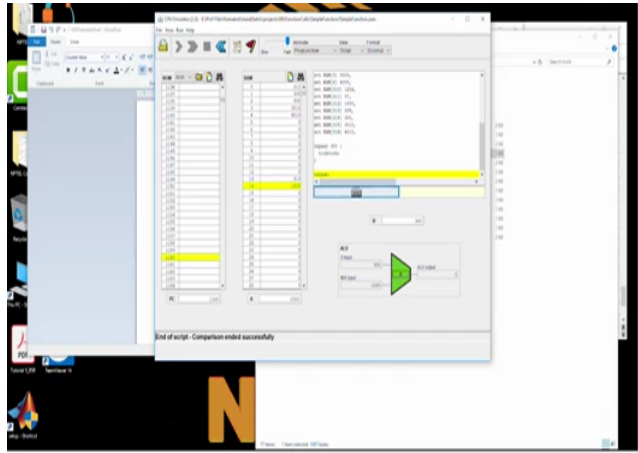
So initially setting up the stack, etc there is something which a script is doing for us which apparently the operating system will do at the (())(5:36) right we will see that as we will proceed.

(Refer Slide Time: 5:43)





Module 7.4: Project 08: Translation of Program Flow and Function Call to HACK Mnemonic

PROF. V. KAMAROTTI
IIT Madras



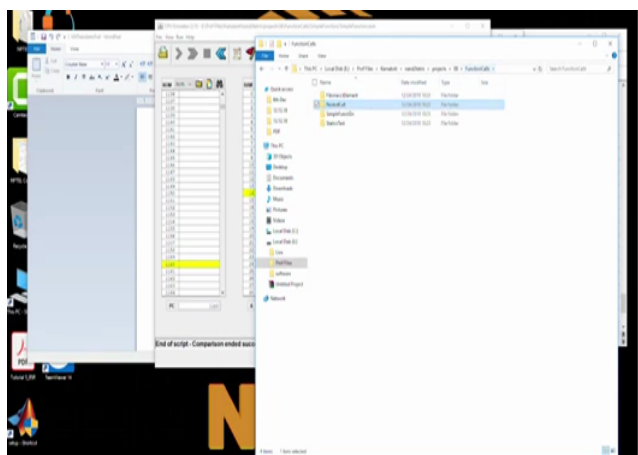
Module 7.4: Project 08: Translation of Program Flow and Function Call to HACK Mnemonic

PROF. V. KAMAROTTI
IIT Madras



Now this will be quite good like we have done for now we have so if you go to the 08 project there are program flow both the program flows have worked. Now there is a function called here also that is the simple function and nested call, if you look at the simple function there is a simple function dot test, we load script and then again just execute it, it will $()(6:04)$ right.

(Refer Slide Time: 6:24)



Module 7.4: Project 08: Translation of Program Flow and Function Call to HACK Mnemonic

PROF. V. KAMAROTTI
IIT Madras

```

// This file is part of www.nand2tetris.org
// and the book "Elements of Computer Systems"
// by Russ and Abraham, MIT Press
// File name: projects/08/Fibonacci.asm

// Compute the n'th element of the Fibonacci series,
// n is given as argument(), called by the Sys-Vish Function
// from the Sys-Vish File, which also passes the argument()
// parameter before this code starts running.

Function Main.Fibonacci()
push arguments()
push constant() 2
// check if not
// push of 2
// label of 2
push arguments()
// if not, return 1
return()
// if not, return 2
push arguments()
push constant() 1
call Main.Fibonacci() // compute fib(n-1)
push arguments()
push constant() 1
call Main.Fibonacci() // compute fib(n-2)
return() // return fib(n-1) + fib(n-2)

```



Module 7.4: Project 08: Translation of Program Flow and Function Call to HACK Mnemonic

PROF. V. KAMARUJI
IIT Madras



```

// This file is part of www.nand2tetris.org
// and the book "Elements of Computer Systems"
// by Russ and Abraham, MIT Press
// File name: projects/08/Fibonacci.asm

// Compute the n'th element of the Fibonacci series,
// n is given as argument(), called by the Sys-Vish Function
// from the Sys-Vish File, which
// parameter before this code starts
// running.

Function Main.Fibonacci()
push arguments()
push constant() 2
// check if not
// push of 2
// label of 2
push arguments()
// if not, return 1
return()
// if not, return 2
push arguments()
push constant() 1
call Main.Fibonacci() // compute fib(n-1)
push arguments()
push constant() 1
call Main.Fibonacci() // compute fib(n-2)
return() // return fib(n-1) + fib(n-2)

```



Module 7.4: Project 08: Translation of Program Flow and Function Call to HACK Mnemonic

PROF. V. KAMARUJI
IIT Madras



Now we will slowly move onto what we call as the challenge now is that we need to start looking at the nested call, let us look at the next function the nested call. So simple function at work let us go for it. In a nested call please note that the VM file is called sys dot VM, right when you will compile this it will give you a sys dot ASM which you rename it as nested call and run it, right in the previous (6:53) in all the previous projects as you see when I say Fibonacci sorry when I in the previous projects that we have executed say for example simple function the VM file simple function dot VM is already available which you will compile and it will give you simple function dot ASM as you see here, right.

But in this case just when you look at the nested call that function is called sys dot VM and so we have to rename it as nested call so that this test will work. Now this is this is also quite fine, but the nest thing that would happen is what really happens here is let us go to this

Fibonacci element, the Fibonacci element basically has sys dot VM, right and it has a main dot VM, so sys dot VM and the main dot VM let us so there are two VM files.

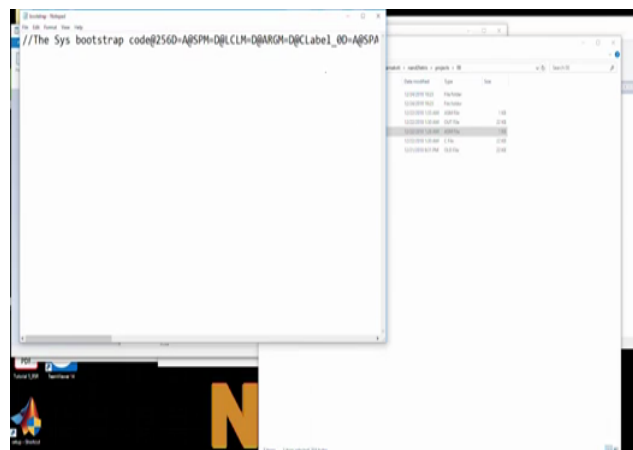
Now how are we going to compile this and what is the relationship between these two VM files? Okay, so let us see this how it is going to work. So this is the main file and this is the sys file. So normally the sys dot VM starts as you see here it is a 0 argument function, it pushes the constant 4 so basically finding the 4th Fibonacci element and then it calls main dot Fibonacci 1.

Now this is the function main dot Fibonacci, it has nil a local argument, so this is main dot Fibonacci 1 means it passes 1 argument to this and that argument is already pushed onto the stack, push constant 4 actually pushes onto the stack and it calls main dot Fibonacci and so main dot Fibonacci starts executing here and this is the main dot Fibonacci, okay and this is also a recursive function calls itself again and again.

But see there are two functions now one is sys dot VM and then other system main dot VM and actually sys dot VM has a function call sys dot init which should first execute and that will call main. So the interesting thing is that we need to basically separately and compile this separately and since this is the first to be executed the compiled version of this should be integrated with the compiled version of this to form an executable file and that is sometime called linking in the long run.

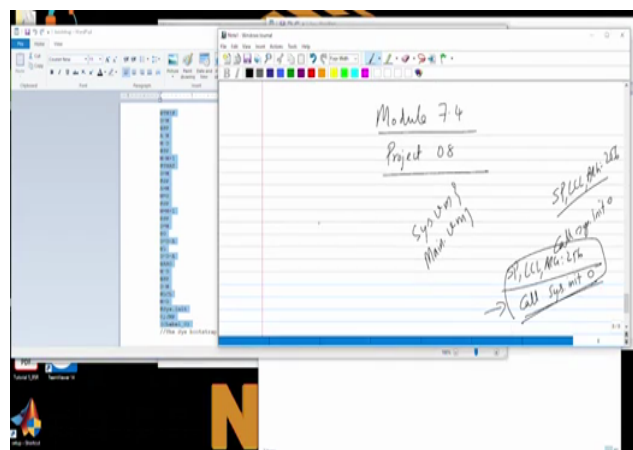
So we will talk about linkers and loaders and compilers but what is that it mean that you basically we have two different functions for example in your stdio.h has printf code and you will set somewhere and you link those things for execution, the same thing is happening here. So you compile this as a function then you compile this separately, then merge these two together to form the executable and that is essentially linking.

(Refer Slide Time: 10:28)



The screenshot shows a terminal window with the following text: `//The Sys bootstrap code@256D-A9SPH-D9CLM-D9ARGH-D9CLLabel_00-A9SPA`. To the right, a file explorer window displays a directory listing with columns for Name, Type, and Size. The files listed include `00000000`, `00000001`, `00000002`, `00000003`, `00000004`, `00000005`, `00000006`, `00000007`, `00000008`, `00000009`, `0000000A`, `0000000B`, `0000000C`, `0000000D`, `0000000E`, `0000000F`, `00000010`, `00000011`, `00000012`, `00000013`, `00000014`, `00000015`, `00000016`, `00000017`, `00000018`, `00000019`, `0000001A`, `0000001B`, `0000001C`, `0000001D`, `0000001E`, `0000001F`, `00000020`, `00000021`, `00000022`, `00000023`, `00000024`, `00000025`, `00000026`, `00000027`, `00000028`, `00000029`, `0000002A`, `0000002B`, `0000002C`, `0000002D`, `0000002E`, `0000002F`, `00000030`, `00000031`, `00000032`, `00000033`, `00000034`, `00000035`, `00000036`, `00000037`, `00000038`, `00000039`, `0000003A`, `0000003B`, `0000003C`, `0000003D`, `0000003E`, `0000003F`, `00000040`, `00000041`, `00000042`, `00000043`, `00000044`, `00000045`, `00000046`, `00000047`, `00000048`, `00000049`, `0000004A`, `0000004B`, `0000004C`, `0000004D`, `0000004E`, `0000004F`, `00000050`, `00000051`, `00000052`, `00000053`, `00000054`, `00000055`, `00000056`, `00000057`, `00000058`, `00000059`, `0000005A`, `0000005B`, `0000005C`, `0000005D`, `0000005E`, `0000005F`, `00000060`, `00000061`, `00000062`, `00000063`, `00000064`, `00000065`, `00000066`, `00000067`, `00000068`, `00000069`, `0000006A`, `0000006B`, `0000006C`, `0000006D`, `0000006E`, `0000006F`, `00000070`, `00000071`, `00000072`, `00000073`, `00000074`, `00000075`, `00000076`, `00000077`, `00000078`, `00000079`, `0000007A`, `0000007B`, `0000007C`, `0000007D`, `0000007E`, `0000007F`, `00000080`, `00000081`, `00000082`, `00000083`, `00000084`, `00000085`, `00000086`, `00000087`, `00000088`, `00000089`, `0000008A`, `0000008B`, `0000008C`, `0000008D`, `0000008E`, `0000008F`, `00000090`, `00000091`, `00000092`, `00000093`, `00000094`, `00000095`, `00000096`, `00000097`, `00000098`, `00000099`, `0000009A`, `0000009B`, `0000009C`, `0000009D`, `0000009E`, `0000009F`, `000000A0`, `000000A1`, `000000A2`, `000000A3`, `000000A4`, `000000A5`, `000000A6`, `000000A7`, `000000A8`, `000000A9`, `000000AA`, `000000AB`, `000000AC`, `000000AD`, `000000AE`, `000000AF`, `000000B0`, `000000B1`, `000000B2`, `000000B3`, `000000B4`, `000000B5`, `000000B6`, `000000B7`, `000000B8`, `000000B9`, `000000BA`, `000000BB`, `000000BC`, `000000BD`, `000000BE`, `000000BF`, `000000C0`, `000000C1`, `000000C2`, `000000C3`, `000000C4`, `000000C5`, `000000C6`, `000000C7`, `000000C8`, `000000C9`, `000000CA`, `000000CB`, `000000CC`, `000000CD`, `000000CE`, `000000CF`, `000000D0`, `000000D1`, `000000D2`, `000000D3`, `000000D4`, `000000D5`, `000000D6`, `000000D7`, `000000D8`, `000000D9`, `000000DA`, `000000DB`, `000000DC`, `000000DD`, `000000DE`, `000000DF`, `000000E0`, `000000E1`, `000000E2`, `000000E3`, `000000E4`, `000000E5`, `000000E6`, `000000E7`, `000000E8`, `000000E9`, `000000EA`, `000000EB`, `000000EC`, `000000ED`, `000000EE`, `000000EF`, `000000F0`, `000000F1`, `000000F2`, `000000F3`, `000000F4`, `000000F5`, `000000F6`, `000000F7`, `000000F8`, `000000F9`, `000000FA`, `000000FB`, `000000FC`, `000000FD`, `000000FE`, `000000FF`.

Module 7.4: Project 08: Translation of Program Flow and Function Call to HACK Mnemonic
PROF. V. KAMAROTTI
MIT 6.034



The screenshot shows a whiteboard with handwritten notes. The notes include: `Module 7.4`, `Project 08`, `Sys init 0`, `Main init 0`, `SPECIAL Mnemonic`, `SPECIAL Mnemonic`, `Call Sys init 0`. The terminal window shows assembly code.

Module 7.4: Project 08: Translation of Program Flow and Function Call to HACK Mnemonic
PROF. V. KAMAROTTI
MIT 6.034

Now let us see what is the executable is for this particular thing so we have finally created ASM file for this executable yeah so this has the function `sys dot init 0` and then it has the `main dot Fibonacci` calls the `main dot Fibonacci` and then the `main dot Fibonacci` code is this. So basically we have integrated these two codes together, in addition if you look at the script for this Fibonacci which we are going to basically use for this the script does not initialize anything here, it does not initialize the stack pointer or does not initialize anything here.

So we now assume that there should be when the compiler is compiling it should do the initialization also. For that purpose we need to so how does the normally how does the system works? The system works as follows basically it needs certain bootstrap code also, so

the bootstrap code looks like this, right. Now it sets the thing to stack pointer to 256 all your LCL, ARG, SP everything to 256, right.

And then it basically calls sys dot init, so the bootstrap code what does what supposedly the script was doing, we need to write that code again here the bootstrap code what it does it sets the stack pointer LCL, ARG everything to 256 this and that and it calls sys dot init always it call sys dot init 0. Now sys dot init starts executing, so the bootstrap code is actually responsible for setting up the stack and everything and then it will go and call sys dot init and then sys dot init will in turn call you know in this case it call Fibonacci, etc.

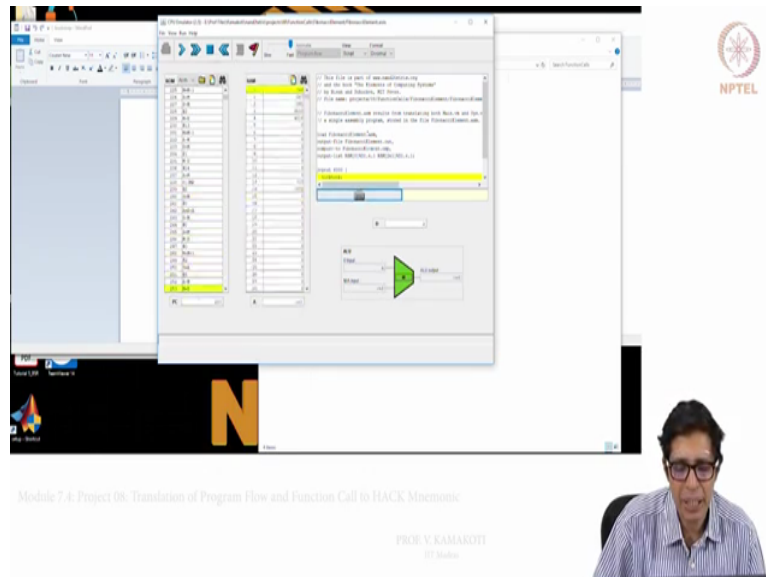
So what we need to do when I want to execute this Fibonacci project is you compile sys dot VM separately, you compile the main dot VM separately these should be separate files and then you also take this sys bootstrap this bootstrap would bootstrap is what we need to do is we make SP, LCL, ARG all these three as 256 and essentially get the assembly code for call sys dot init 0, so you know how to generate an assembly code for call, but you generate the assembly code for this and this is basically the bootstrap code that is what we have done here, we have made 256, made LCL, ARG everything LCL, ARG and SP as 256 that is what we set here and this entire thing at label 0 to this is basically equivalent to call sys dot init 0 and that is what we have done here.

So this bootstrap code in turn will call sys dot init, right this is at sys dot init 0 (0)(14:40) it will call sys dot init and sys dot init in turn will call Fibonacci and (0)(14:45) interesting these are Fibonacci code you will execute you said recursive code, so recursion is that I basically call myself, so that is so we can now see how this code is basically going to execute.

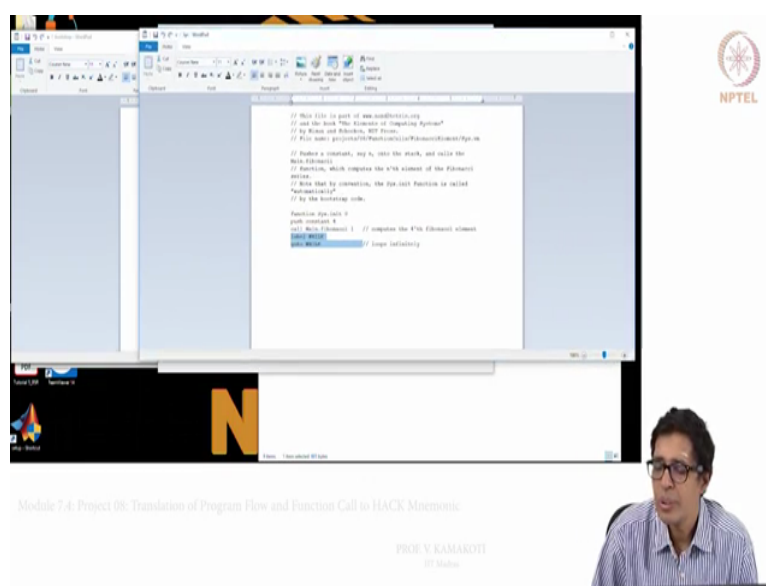
So in future for this Fibonacci and the static test there are two codes that we see here as a part of your program as a part of the project, right in the function calls you see now Fibonacci element and static test for these Fibonacci element and static test we basically have to compile all the individual files and then put them together and do, right and so we will just do for the Fibonacci element what we have done we compiled sys dot VM, we compiled you know the main dot VM and then there is a bootstrap ASM so when we compile the sys dot VM we got sys dot ASM, when we compile the main dot VM we got main dot ASM so you put and then we compile there is a bootstrap dot ASM so bootstrap dot ASM followed by sys dot whatever you got for sys dot init followed by whatever you got when you compile main dot VM.

So the three ASM files you basically (())(16:13) one after another and that will be your final executable, right. So this is what we have done here.

(Refer Slide Time: 16:24)



The screenshot shows a debugger window with assembly code. The code includes comments in Hindi and assembly instructions. A green arrow points to a specific instruction. The video title is "Module 7.4: Project 08: Translation of Program Flow and Function Call to HACK Mnemonic" and the speaker is "PROF. V. KAMAROTTI".



The screenshot shows a code editor with assembly code. The code includes comments in Hindi and assembly instructions. The video title is "Module 7.4: Project 08: Translation of Program Flow and Function Call to HACK Mnemonic" and the speaker is "PROF. V. KAMAROTTI".

So let us go and execute it as a part of your code, so I am just going to now load the script corresponding to the Fibonacci I have loaded the script, now we will just go and execute that yeah so this will go on for quite some time you have 6000 tick tocks so we need lot of patience to do that, why this was 6000 tick tocks had been given you can reduce it to say even 2000 or 3000, why it is given as 6000 because if you write very large code we need to atleast execute 6000 instructions but normally this code will finish of very very fast, right.

So this is how we go about doing this and right so the last of the exercise is so we have actually completed so this is just so if you see the sys dot main here, sys dot VM here sorry

this is the sys dot that is the label while and go to while so this will be basically rounding here and that is precociously what got translated and what you see here so this will take quite some time and it will execute and finally you will see that program executed correctly. So there is one more interesting thing that one more part of this project we will explain it in the next module and that is very very important, thank you.