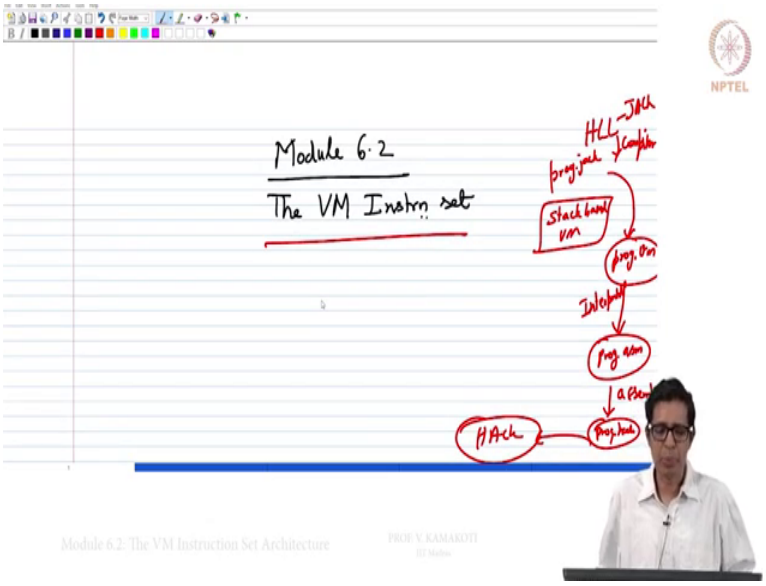
Foundations to Computer Systems Design
Professor V. Kamakoti
Department of Computer Science and Engineering
Indian Institute of Technology, Madras
Module 6.2
The VM Instruction Set Architecture

(Refer Slide Time: 00:17)

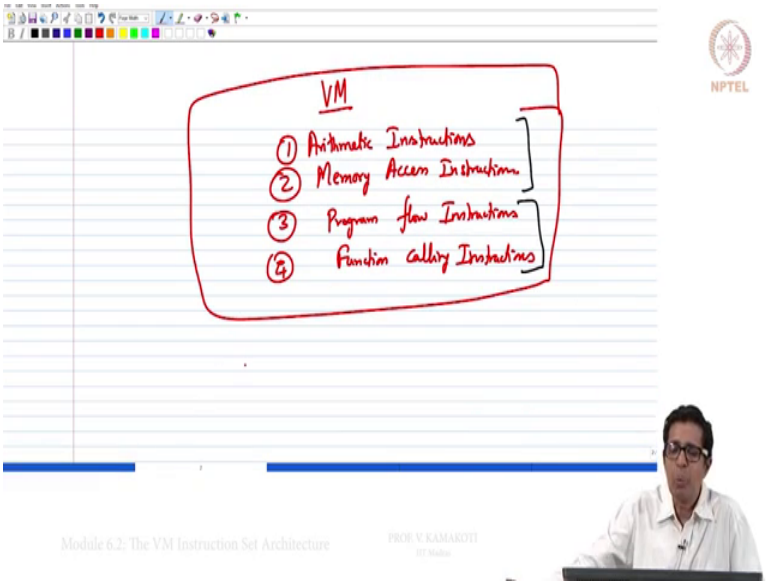


Welcome to module 6.2, in this module 6.2 will be talking about the VM Instruction Set. So just to sum up what we have done so far our understanding there is an high level language called Jack we are going to write a compiler later that so there is a prog dot jack which is in the high level language the compiler will translate it to prog dot VM. So what are the instruction in prog dot VM that is what we are going to talk about it. Now and the prog dot VM should now essentially get interpreted into prog dot asm that prog dot asm will be in the mnemonics of hack which w saw and then the assembler which we have already written will make it into binary which is prog dot hack and it will get executes.

So the prog dot jack becomes prog dot VM, prog dot VM becomes prog dot asm, prog dot asm becomes prog dot hack and prog dot hack gets executed on the hack. Now we know how prog dot asm gets converted to prog dot hack we also know the mnemonics of that asm. Now we will understand what are the instruction in the prog dot VM and how will it get translated so each

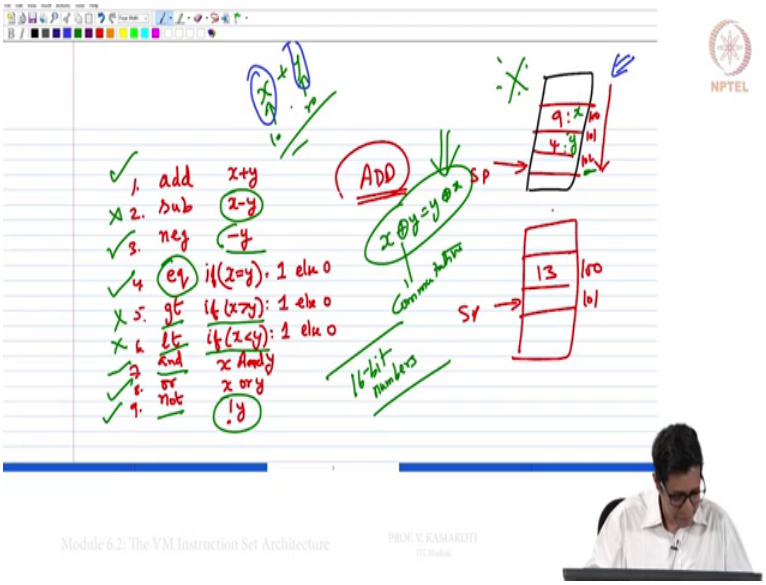
instruction what is the equivalent translation of that into the mnemonics of hack. So that is what we are going to discuss in this module 6.2 and subsequent module 6.3 etc.

(Refer Slide Time: 01:41)



Now what are the instruction that will be part your whatever machine there are four types of instructions namely arithmetic instructions, memory access instructions, program flow instruction, function calling instructions, so there are four types of instructions and we will see them one by one very quickly as a part of this module.

(Refer Slide Time: 02:04)



Now what are the arithmetic instructions there are nine types of arithmetic instructions that this whatever machine will support and note that this is a stack based virtual machine. So all the arithmetic instruction will work on the elements that are on the top of the stack.

For example, let us take this one which I am marking here I have 9 and 4 on top of the stack right I have 9 and 4 on top of the stack I said add so let the stack be stack pointer be pointing to 102 location, 101 and 100 has 9 and 4 when I say add what will the operations suppose to do? It needs to basically pop the top two elements of this stack add them and store it back in the stack and adjust the stack pointer. So what will happen if I say add here? 9 and 4 will popped out it will be added 13 then that 13 will be stored here on the 100 and your stack pointer now will point to 101. So this is what should happen.

Similarly when I say subtract x minus y when I ((()))(03:20) is a unary operation so I will just if I will take y and minus y I do if x is equal to y so EQ is the instruction if x and y are equal then I need to store 1 else I need to store 0. Similarly if x is greater than y, x is less than y now AND of x and y, OR of x and y knot. Now when you see vey importantly 1 3 4 7 8 9 are commutative operation that means even if I swap y and x right I call x as y and y as x still the result will be the same right so 1 3 4 7 8 9 are commutative operation that means x operation y is you know equal to y operation x.

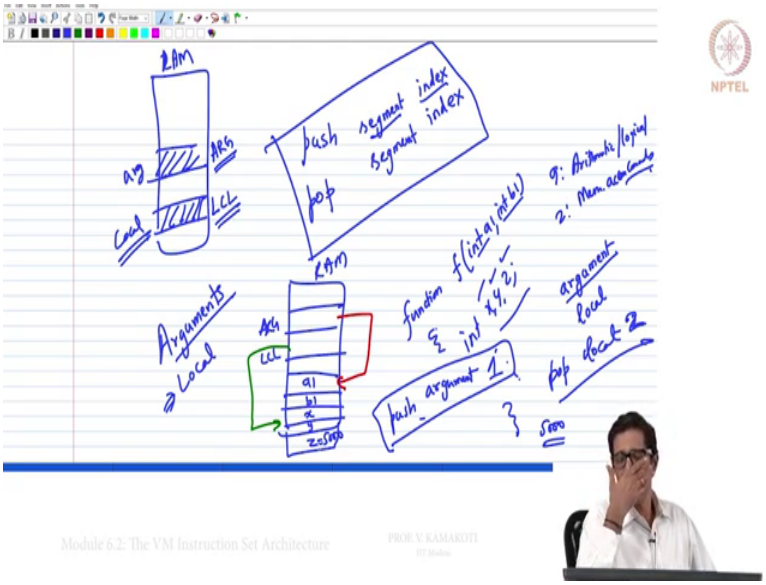
But note that for example, in 3 and 9 are unary operator so there is nothing like the second operator but look at second, fifth and sixth instruction namely subtract, greater than and less than these instructions are not commutative that means x minus y is not equal to y minus x. If I am comparing x and y (x) for greater than x greater than y is not the same as y greater than x or x less than y is not the same as y less than x right.

So atleast these three instructions are not commutative, so the mment I know that there are binary instruction which are not commutative binary in the sense I have more than one operand that is binary if there is only one operand s called unary operation. So in this case there are two unary operation namely (())(04:58) but all the other things are binary and if I have a binary operation which is not commutative I should know where is the left operand and where is the right operand please note here x and y, x is the left operand y is the right operand.

In the case of non-commutative operation this matters I have to get the correct left operand I need to get the correct y operand. Simple reason x minus y is not equal to y minus x right so this is an very-very important thing that we need to remember when we are talking about this arithmetic instruction so basically in the stack when I am working on a binary arithmetic instruction the first so stack pointer always points to your empty location the first after that before that is y and then is x. So always the right operand which is y so x operation y, y is on immediate top of the stack immediately after the stack pointer and x is beheaded.

So this we need to remember because we need to handle non-commutative operation as a part of our entire develplopment. So this is about what we call as the you know the memory access arithmetic and logical instructions right and one of the thing that we need to next talk about is memory access commands right.

(Refer Slide Time: 06:45)



So there are two memory access commands, one is PUSH and another is POP. Push means push into the stack, pop means pop out of the stack, push what into the stack? I need to push some element into the stack that element is given by segment and index. Similarly pop yes I will pop the top of the stack to where?

I need to specify where should I store the result, to some segment index, so there are two memory access commands as we saw we have 9 arithmetic commands, arithmetic logical and we have two memory access commands. So right now let us understand what is this whole segment in x, that something is new Push and Pop we have already seen only, what is segment what is index? Let s look at I am having a function now this function requires different types of memory, what do you mean by different types of memory? Or a very proper word is I need different portions of memory.

What are the different portions? As a function suppose I say function F of into a1, int b1 then I have (())(08:26) right. So I will be asking for int a1 and int a now what are the things that I have arguments ok then I have local variables right at least two of them that we could understand right so where are this arguments store? As far as the VM is concerned this arguments will be stored in a segment called argument right similarly the local variables will be stored in a segment called local. What is segment called now? Argument in this case argument is one memory segment which can hold two variables namely a1 and b1.

So argument will be so when the program is executing then this particular function is executing immediately the operating system will create one small piece of memory which can hold its two locations one which can hold a1 another which can hold b1 and that is the that is part of the argument for this function. Similarly the operating system will also create a memory with three locations 1, 2, 3 where that's called the local segment where in the 0^{th} location x will be stored, one location (())(10:07)ok right.

So I have said as this concern argument so you have that entire big run right data run so when the function starts executing there will be some space allocated for arguments some place allocated for (local) right. So when we are defining the symbol table right you remember we can go back to that exercise you will remember that you have used a variable called (HCL), LCL and another variable called ARG etc symbol name called ARG and LCL so this are basically you know those symbols. The LCL right this LCL is a location right it is a location LCL will store the start address of this local segment right and similarly the ARG will store the starting address of your argument segment right.

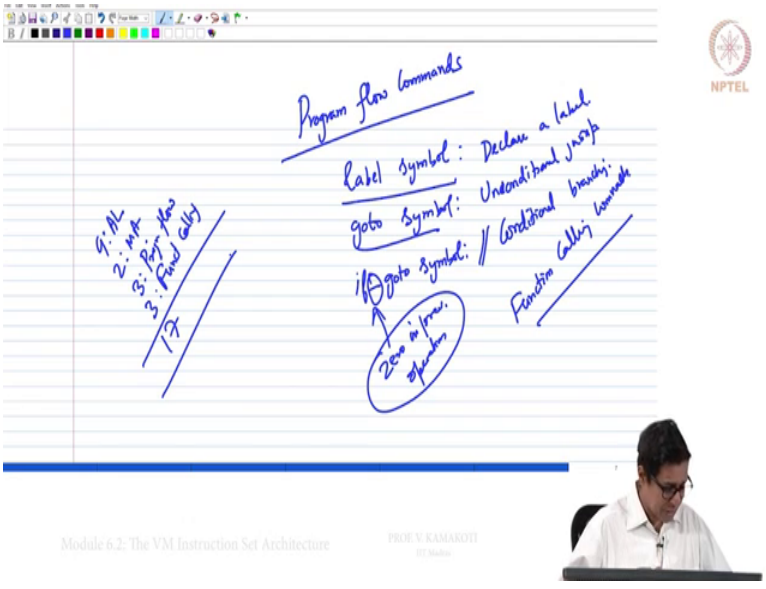
So when this function gets allocated so this is let us say this is the ramp inside that ramp you will have something called ARG and you will have something called local, this ARG will point to some location here, this local can point to some other location here then at this location please note that I would have created for ARG and here I will be storing the value of a1 and b1 and similarly local I will be storing the value of x, y and z right. So that is what I mean by a segment a segment is basically a piece of memory created by the operating system for the function to basically store some of its parameters and local variables right.

So there are many types of segments we will introduce this segments one by one later but as far as you understand now when a function is executing there are two segments that are necessary one is for storing arguments another is for storing local and this is how they are basically organized right. Now when I say push segment 0, suppose I say instead of push argument 0 means what? Go to the argument section, go to the 0^{th} index right go to the argument section that means you have reached this storage a1 right, now take the 0^{th} index that will be a1 and push argument 0 means we are pushing the value of a1 into the stack right.

Suppose I say push argument 1 that means go to this argument segment that is here, go to the first index that is b1 push argument 1 means push b value of b1 onto the stack right. So this is how so that is what we mean by segment and in (())(13:59). Similarly now I can go and say pop argument or say let us say pop local 2 right, so this will go to local this and it will go with plus 2 will go (())(14:22) so whatever is there on the will be suppose the top of the stack somewhere stack will be inside the ramp so somewhere let the top of the stack will storing 5000 when I say pop local 2 that means that value 5000 will get assign to e Z here.

Local is pointing from there you take the second location so that is the location of e z that we have allocated when I say pop local 2 and the pop of the stack as 5000 is basically get 5000 right. So there are nine arithmetic commands and logical commands and two memory access commands namely push segment index and pop segment index right. So this two are the two memory access command and there are atleast eight types of segments we will talk about that a little later but you do understand what is a segment here, I have just described it using a very simple function wherein I said when the function is executing you need atleast two segments one for storing the arguments and another for storing the local variables.

(Refer Slide Time: 16:53)



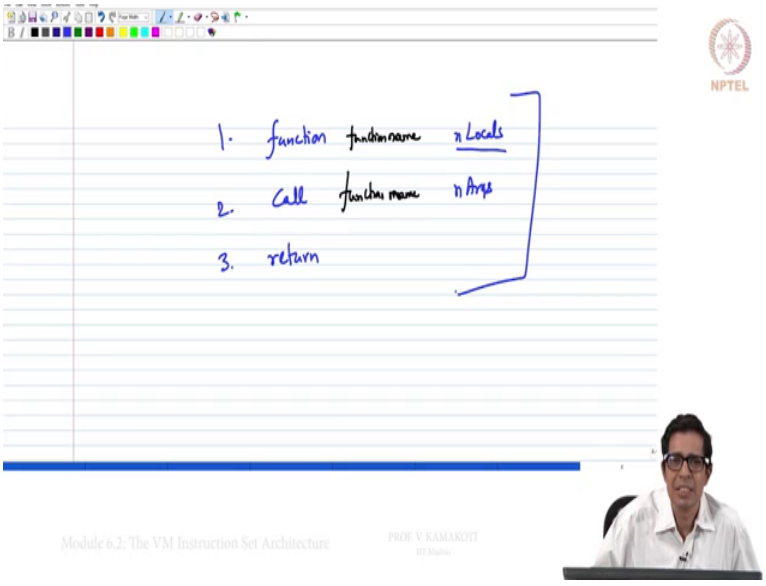
Then the next type of instruction that we are going to introduce here are basically program flow commands, they are very-very straight forward. The program flow commands are I have label, symbol, label is a keyword, label symbol this will declare a label right then there is something

called go to symbol this is nothing but an unconditional jump to the label given by the symbol and then if dash go to symbol, so this what you call as a conditional branching.

For example if the previous operation as yield the zero then it will go to for example I can go and say the previous operation is a zero just go just jump to (())(17:20). So this are program flow commands in the sense that they alter the they dictate the way in which the program basically gets executed ok and the last so we have again repeat 9 A arithmetic logical, 2 memory access 3 program flow and the last one that will be discussing is basically function calling programs, a function calling commands and we will have three of them, 3 function calling.

So total number of instructions in your virtual machine is 11 plus 6 ,17 instructions in your virtual machine (())(18:12) so if I know how to convert each of these 17 instructions into the hack mnemonic right then we have understood the translation from the virtual machine to a mnemonic part that is the very interesting easy thing. So now what are the function calling instructions?

(Refer Slide Time: 18:47)



There are three types, three function calling instructions, the first one is function, function name and n locals this basically tells you how many local variables are there, then I have call the first command will tell you how may local variables are there in the function.

For example in our previous case that we described here this had three local variables x, y and z then there is a call and I have to specify the function name and it has n Arg's how many arguments are there (())(19:49) and then ofcourse the third one is return, return from the call function. So there are three function calling commands right. So totally 17 commands and this know how to translate each one of these 17 commands into its equivalent mnemonics then the whole thing is then the entire translation process from VM to (())(20:16). So now what we need to understand is how we will translate from translate each of this things to VM and first and foremost we will I have given example of how to use this instructions will give some one case study of how to use this instructions and then we can go ahead and translate them into the mnemonic of hack right.

So in the next module will take one example and see through how it basically works right, so I want you to remember these 17 instructions that are 9 (())(20:56) instructions 2 memory access instructions, 3 program control instructions program flow instructions and 3 function calling instructions. So we will just have an understanding of this and in the next module we will immediately see an example program how it is getting translated.