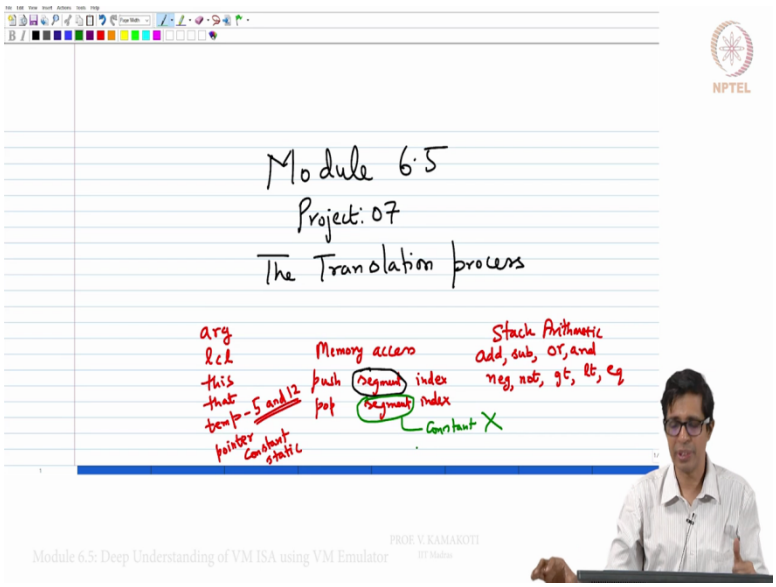
Foundations to Computer Systems Design
Professor V. Kamakoti
Department of Computer Science Engineering
Indian Institute of Technology Madras
Lecture 01
Module 6.5
Project 7
Deep Understanding of VM ISA using VM Emulator

So welcome to module 6.5 and in this module we will be explaining Project 7 which is essentially the translation process, the translation from the virtual machine instructions to the pneumonic of hack, right. We have seen a set of examples and set of you know a basic idea of how this would do but now what we will do is to give you are very concrete explanation of this entire process right. So as a part of Project 7 we will be basically handling 2 instructions right, stack arithmetic instructions which are those 9 instructions add, sub, OR, AND (()) (1:09) neg, NOT, then greater than, less than and equal to, we have basically seen how these operations work.

(Refer Slide Time: 2:49)



And then we will also be handling the memory call the memory access functions which is push of segment index and pop segment index. And we have also seen there are 8 different segments, the segments that will hold the arguments, the segments that will hold the local variables then two general-purpose segments this and that and then there we have a temporary segment which is told in the rank between 5 and 12 so there are 8 locations in the temporary segment and then we have pointers to this and that we have pointers segment and then of course constant. So the segment here in the case of push can take any one of these

one, 2, 3, 4, 5, 6 and constants okay so and static of course, anyone of these 8 values this particular segment can take.

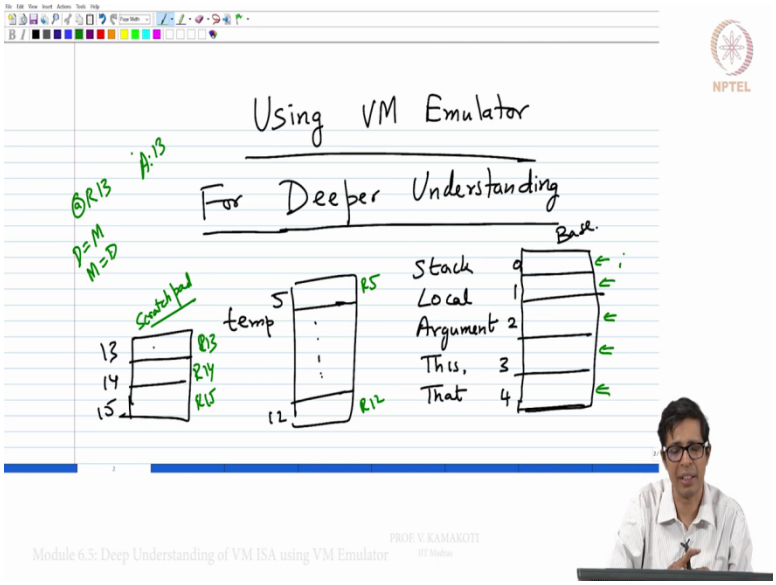
So what it push means is that the value from the segment, whichever segment + index need to be pushed into the stack. Pop is, from the top of the stack remove an element and put it into the value given by the segment, so the address pointed out by the segment index we have to take from the stack and put it here. Now this segment in the case of pop we cannot have it as constant because I cannot take a value and put it in a constant right. So for pop, constant does not exist but for push all the 8 exists 8 different segments exist and for the pop there are 7 segments except constant. And so these are the type of legal instructions that we will see in your vm file and every such instruction we have to translate it into the corresponding hack pneumonic and that is all our Project 7, is a very easy project, it hardly takes one hour for you to complete.

I will tell you how to go about it and then you can take it forward and this time this is exactly the time where we talk about compilation one important thing is that when we surveyed many of the education curriculum especially in the case of compilers, back end, compiler has 2 parts; one thing is you have a language, you do a syntactic analysis, you do a passing then you actually generate an intermediate representation and from there you do something called back end which is basically you translate that code into the assembly language, the real translation process, the assembly language of the target architecture.

What we are doing here is essentially the part of the back end, where we are taking the virtual machine code which is actually intermediate representation. So that is the original programming language where the compiler will convert it into that intermediate representation, now we are taking that intermediate representation and getting onto the pneumonic so the back end process is basically done here and this is the point where we need to study about compiler optimization and so many things and many curriculums actually do not have big emphasis or give lesser emphasis for the back end.

It is very important that we learn back end because back end is one that is going to become extremely important especially when we now talk of varieties of processes coming up and specifically targeting different types of applications, etc, back end becomes extremely crucial. So this for particular projects 7 is very-very important for any aspiring computer scientist or computer engineer okay, so let us give our full effort and see how we go about this particular problem, right.

(Refer Slide Time: 5:44)



Now let us understand the VM instruction in its full glory right, so we have been talking about that with some pieces, I am sure you will have a vague idea, now I want to concretise that idea. Whatever you have it should be crystal clear of what every instruction supposed to do, and to aim this so there is a tool that is given as a part of the software which is called the VM emulator, the virtual machine emulator now we will use the VM emulator and basically tell you how things will work.

Now we have already explained that when a particular program starts executing and functions starts executing, the function will have different types of variables like we have local variables which is stored in the local segment, arguments that are passed which is stored in the argument segment and then it will ask for some extra space (())(6:33) and those 2 segments are this and that and then of course for doing all the arithmetic computation this being a stack-based Virtual machine, we will use a stack. Now there are 5 different areas of memory what we call as segments of memory where the values of the local variables, arguments, this, that and sacked all this would be stored.

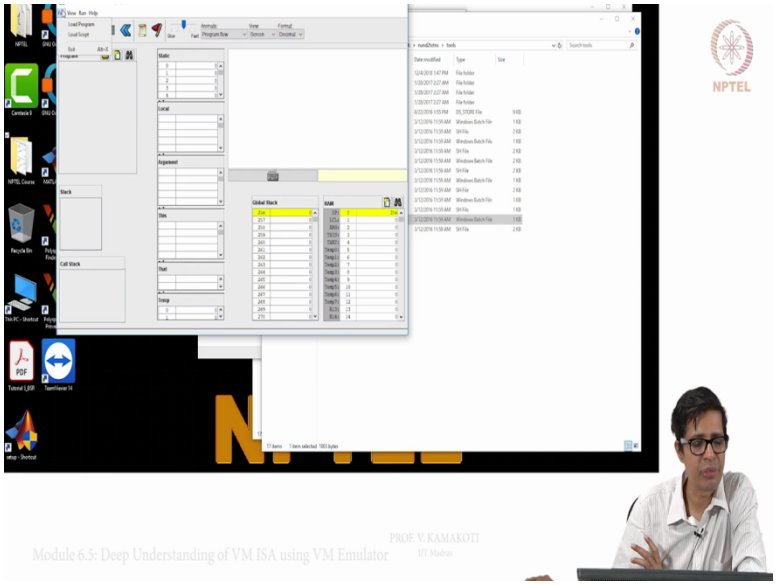
Now when the architects are executing, it needs to know where the stack starts is the base address right, and where the local segment starts, where the argument starts, where the This if at all I am using it where it starts. So those based addresses of this pack segment, local segment argument This That are stored in your data RAM on the location 0, 1, 2, 3 and 4. So please this stores the base address of the stack right, and then there is a temporary segment which we can again use in addition to this and that the compiler can use the can use the temporary segment. The temporary segment of memory has 8 locations starting from 5 to 12

in the RAM in the data RAM, you have instantiated that data RAM and in the data RAM this 5 to 12 will be for temporary.

And this 13, 14 and 15 are again registers or temporary we call it as scratchpad memory, which is again free anyone can use it is just a scratchpad so we can just use it again okay. So essentially, to have a… So this is how you understand this whole thing right, so I have these segments, the based addresses are stored in 0 to 4 then I have a temporary segment which has 8 locations, I can store any data and retrieve it, the temporary data segment is between 5 and 12, 8 of them and 13, 14, 15 are still scratchpad which still I can use it, right. And when you actually did the assembler, you have assigned this as you know R 5 so from here right R 0, so R 5 to R 12, this is R 13, R 14, R 15, if you go to a single table you would have matched to 13, R 14 to 14, R 15 to 15 right.

So if I want to access this location so how do I access R 13 if I want? I just pray at R 13, the moment I say R 13 your A at register will get 13 because at R 13 in your single table it is mapped down to 13. Now I say M so M is M of A so this whatever value in this 13 will come out, so if I say D = M then I can actually hear the value that is stored in 13. If I say M equal to D then I can write whatever is there on the D register onto the same right, all these operations I can do. So this is one simple understanding that we need to have before we start using the emulator. Now we will start using the emulator and run from interesting code here.

(Refer Slide Time: 10:00)

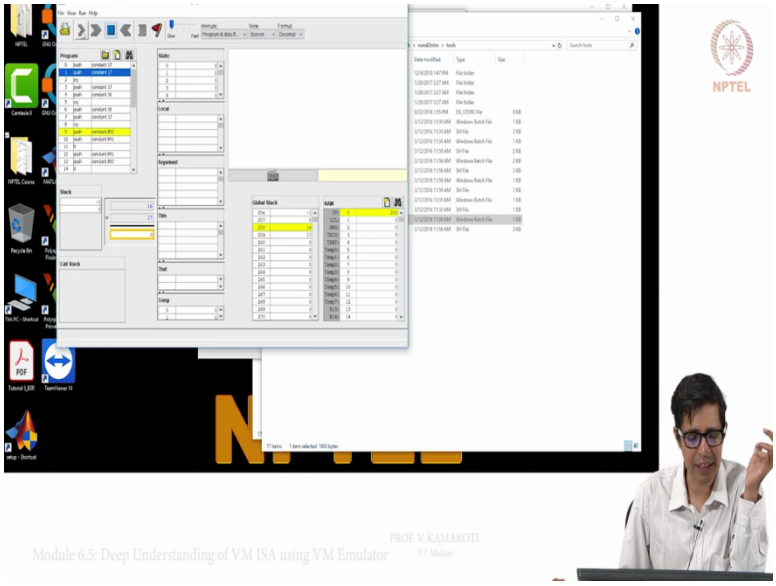


So how do you invoke the emulator, right? So I will rotate here I will again start from the beginning how do we go and import the emulator okay. So go to your 92 Tetris directory, go

to your tools, there is VM emulator dot windows back file, if you are using Linux you can use dot sh, in Windows use this VM emulator. Okay, your emulator has come up, now let us take some programs here, so load a program right, so I can load a program yeah so when I say push constant 17, 17 gets pushed into the stack and this is 0 is direct 257, the stack now stack segment no points to 257, so 17 is pushed into the stack. Again I push another 17 so 17 so stack now becomes 258 stack pointer becomes 258, it is in RAM 0 again.

Now I am doing EQ, what will EQ do? It will compare whatever is there on the top of the stack, namely 257 and 256 and if they are equal then it will pop them out so you now see 256 as – 1, what is - 1? 11111 that is so now let us do this EQ, so from the stack as you see both the 17 are matched and the answer is - 1 and the - 1 goes back to the stack pointer. Now the stack has - 1 alone and your stack pointer now is in 257 right, so this is what basically happens here. Now I am now trying to push 17 now to so 257 will again get 17, now 258 will get 16, now we are now going to compare EQ, then that means 257 and 258 will be compared and if they are equal, they are not equal so essentially now we have to get in 257 we have to get 0 that is false because you pop these 2 values and then compare and then whatever is the result we store here.

(Refer Slide Time: 13:28)



Now we do this, so first 16 is bought down that is why then 17 is bought down, they are compared for equal to, they are not 0 now 0 goes back and stays in the stack right. Now again let us do this, now your stack is at 258 now 16 is pushed so 258 will get 16, now 17 is pushed so 259 will get 17, stack pointer now is 260. Now again I am comparing EQ, now you will

say 17 goes up, 16 goes up, 16 is XCR in this case, answer is 0 the 0 is pushed so now your stack pointer is 259 because 256, 257, 258 are full.

Now I push 892, I push 891, now I say less than again so less than we have to be careful, please note that X is 892, Y is 891 so I am comparing 892 with 891 for less than. Obviously 892 is not less than 891 and so you should get 0 and that 0 will now go and stay in 259 and the stack pointer should become 260 after this operation so let us see that. So 891 is Y, 892 is X we have already explained that so the answer is 0 and that goes there and the stack pointer now is 260. Now again I am pushing 891 into 260, 892 into 261 now I am doing again less than here so 892, 891, yes 891 is less than 892 so the stack should get - 1 so stack gets - 1 and we are at 892.

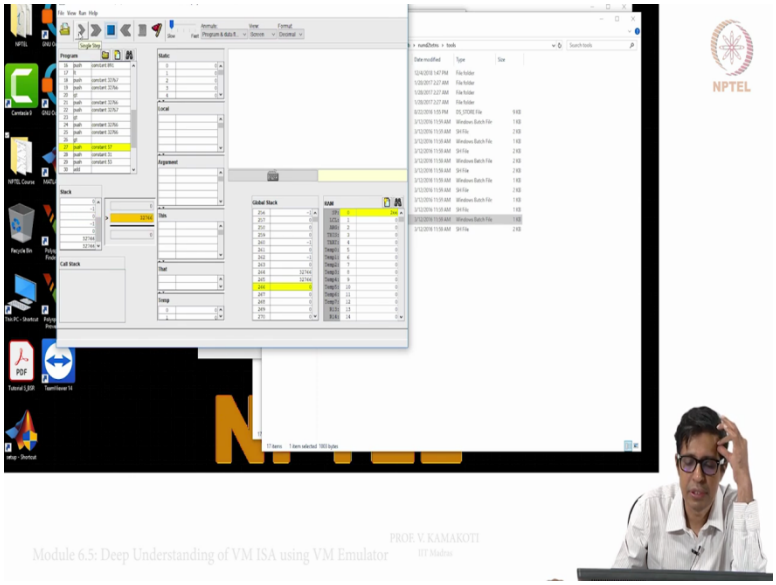
Now again we do the same thing, 891, 891 now less than we compare… 891 is not less than 891 and so I need to get a 0 and the stack now becomes 262. Now I compare 32767 with 32766 for greater than yes… Y is 32766, X is 32767. 32767 is indeed greater than 32766 so the answer is - 1, - 1 is true 11111 the decimal 2's complement of that and then your stack now should point to 263. So 263 is now having 32766 but is part of the previous computation so when I could out of the stack, when I pop something I am not going to erase that, I am just reducing my stack pointer right. And the next time so if I am pushing now again I am pushing through it 32766 so 32766 again comes here right and 32767 is loaded, now we are saying greater than now obviously 32766 is not greater than 32767 so I get a 0 there.

So 263 becomes 0, now the stack is 264 but originally the stack was 264th location was used for storing the 32767, we have not removed that 32767 right. So something that was used in the past is still left over in the stack right, what is that that when popping means I reduce my stack pointer so the fresh thing like I am now going to write 32766, that 32766 replaces this 32767 as you see here but the original 32767 was not removed right. Now this is one major issue when we come to information security right, so when you take advanced process in information security what would happen is that something that you have written in the stack, when you pop out of the stack, you do not even it like we are not erasing it here.

And since you did not erase it, some other process which is trying to access make get the same stack and they will see all these values that we have used which are not erased and that is how some of the critical information like passwords, etc, can leak out so this is one very important thing so when you actually write a compiler is a secure compiler right that secure

compiler should say whenever I pop you should make it 0 and then only pop so this 32767 should not have been there when the moment I pop.

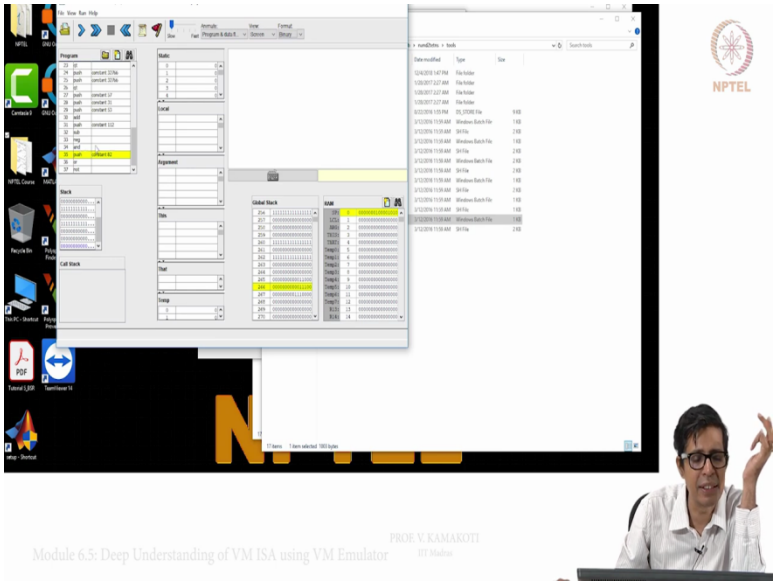
(Refer Slide Time: 18:02)



So in this operation also we will see 32766 is loaded into 265, now I am saying greater than so obviously 32766 you can see the animation that is happening very nice one right. So 264 became 0 because 32766 is not greater than 32766 so that operation got over but still that 32766 is still there which is used in the previous operation though now I am going to replace it with 57 now right but it is still there. So this is one thing, when you write a secure compiler it should not be there you should make it 0 otherwise the next process, this information will be available in the memory and some other process which will use the memory later can basically get out the critical information. Suppose you stored passwords instead of numbers here then it can go out, so this is very important clue that you should keep in mind right.

Now let us go and see, now I am going to push 57 inside yes, then I am pushing 31 then I am pushing 53, these are all constant push. I am going to do add which will add 31 and 53, 84 and it will make 266 as 84 and my stack should become now 267 right, so this is how it will go… sorry… 53 + 31, 84 now goes back into the stack. Note that that 53 is still here, this is what 84 is the answer, now I am pushing constant 112, now I is subtracting 84 from 112 yeah 112 from 84, X is 84, Y is 112 always this is index. So when I do subtract, 112 is Y, 84 is X, 84 – 112 is - 28 so - 28 will take the position of 84 and the stack pointer now is 267.

(Refer Slide Time: 21:22)

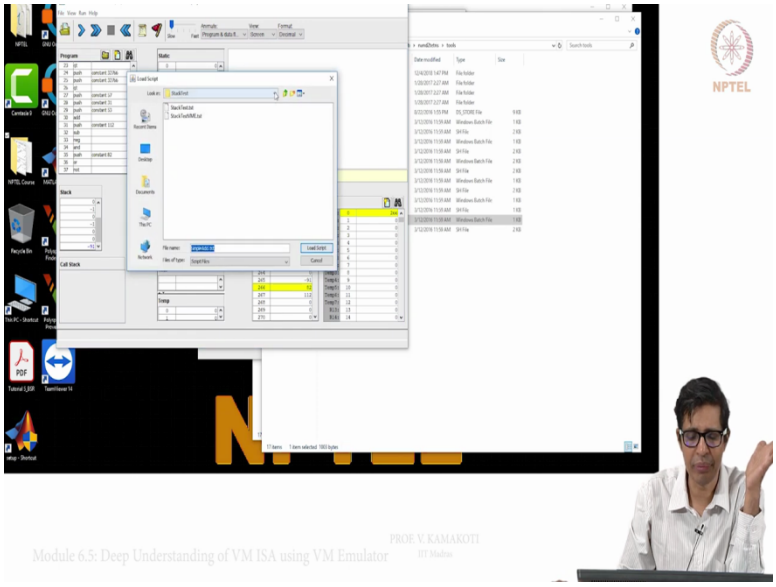


Now I do negate, what will negate do? It will go and make the element on the top, what is there on the top of the stack? 267 is the stack pointer, always a stack pointer points to the location where I can enter something fresh, the top of the stack is one less than that stack pointer again you should remember that so - 28 on the top of the stack, now I do a negate and it should become 28, so -28 comes here it is a unary operation, I negate and then 28 goes back to the stack. Now I am pushing constant 82, now I am doing an AND so I am taking 57 and 28 and we are getting 82. 57 AND 28 so for doing this AND probably we can use it as binary then we could have appreciated this, I do not know whether we can backtrack this let me see, I cannot backtrack, so 57 and 82 got AND so that it became so let us go back, we will explain the OR.

Now I am pushing 82 inside to the stack yes your stack is now 267, now I am going to do an OR operation, so let me just do binary. So what will OR do? It will take the top 2 fellows and do an OR of this okay so let us do how it is going to… 266 and this has gone and so it is 90 and we have got this 1011010 so this is the OR right. Now again I change (())(22:11) now I am going to do a NOT of 82 that is interesting, so I just do NOT of 82 so I should get all 11111110 okay so I just do a NOT so this becomes - 91 in 2's complement arithmetic okay and that is it, this is how these things work. So when I push, what it means to push something? go to your RAM location 0 that is at P means your RAM location is 0. Read the content of RAM location 0 that will give you the stack pointer that address will be the stack pointer.

And if I am doing a binary operation, go to the one address below it get Y one more address below that, get X, do the operation. We will say unary operation, go to that point and then after that reduce the stack pointer that means go back to the 0 and decrement its content by 1 that is what you mean by reducing the stack pointer, so this is what you do in the case of binary operations right. In the case of you know push operation what you do? You go to the stack pointer, whatever is the memory location go to that memory location and write whatever you want into it and then go back to the 0^{th} location and increment this value say from 266 to 267 right, so these are some of the things that we are getting and this understanding you need to get out of this.

(Refer Slide Time: 23:58)



Now let us do some more of things, so we have done stack test, let us do say we will go up let us do memory access, this is very interesting. So let us go… So this is the program again note that your stack segment is initialized to 256 let us go one by one, push constant M in O what will happen? So pop local 0, the local segment starts at 0 that 0 + 0 is 0 because what is the base of the local segment? It is 0, that 0 + 0 is 0 so you pop the value of the stack that is from 10 onto 0 + 0 = 0 so your stack pointer now should become 10 that is what should happen, understand? So local is the segment, its base address is 0, I add 0 to that that is 0 so that means that the address to which I need to pop his 0 and what I will pop in so now your stack pointer should become 10, let us see whether it is happening.

Now what are we understanding? See I have to, when the system boots up, the operating system we have to allocate some space for the local segment. Everything I made 0 so this obviously should not work right, so let us reload this program again and then we should also

look at the script okay, now this is something very interesting look at the script and there you will see what are the things that will be set. So my RAM 0 basically is 256 for the stack, now I will go and set my local segment as 300, we have to set the local segment as 300 and my argument segment as 400, everything cannot be 0 so I have to allocate memory when the program starts executing, I need to locate memory for that function, I mean so the operating system has to allocate memory for that function for storing the different segments otherwise things will not work and that is what I have demonstrated here, right.

So I make this as 3000 and that as some 3010 okay so this is something that we need to load okay. So what I have done here is as follows, the stack is 256, this particular program is actually going to read from the local segment there are some local variable so for the local variables I want to allocate space so I say let your local segment start at 300 so anything that I want to write local segment will be from 300 and your argument will start at 400, your This will start at 3000 and your That will start at 3010 right and Temp always will start from 6552 12 okay. Now let us reload this program, I reload the program, we have loaded the program now let us go inside all these things, this should be 300, this should be 400, this should be 3000, this should be 3010 okay and now let us start executing.

Push constant 10, so 10 gets pushed into the stack, pop local 0 so local is a 300 right local is a 300 so 300 is the base for local + 0 so the address to which I need to pop is 300, in the 300th location I need to pop the value, so the 300th location will become 10 and your SP will again become 256 after this, let us see whether it happens, right. As you here, yes your 300 location became 10 and your SP has become 256, let us see what will happen at the 300 memory location, we can go down here and we can see that the 300th memory location has become 10 so this is how it popped.

So what does pop to? First it will go, it will find with segment you want that is local, it will take the base address of that local, add with that index, it will find the address to which I need to pop that is in this case 300, now it will go to SP right, it will get the stack pointer value, it will go one below that stack pointer value, whatever value is there it will now go and put to the calculated address in this case 300, so then goes to that address and now I decrement the stack pointer, now stack pointer is now 256 stack is actually empty now. Now I am pushing constant 21 so 256 gets 21, now 256 will get 22, now I want to pop this to argument 2, so where does argument starts?

Go to RAM location 2 and you find that address 400, 400 + 2 because you said pop argument 2 so 402 now I need to go to 402 and 402 I have to pop 22 and your SP will now again become 257, so your 402 will get 22 and your SP becomes 257, let us see that. So in your argument 22 come and it become 257, now if I go and see 402 right if I see 402 you get 82 here. Similarly, pop argument 1 so in 401 you will now get 21, 401 you should get 21 and your stack now has again become empty that is it again points to 256, you pushed 2 constants and you popped.

Now I say pop this 6, whereas this goes to location 3 again in the assembler symbol table we have marked this as 3, right go to that 3 so that will be 3000 + 6 = 3006, 3006 should get the value 36 now, we have actually pushed constant 36 there now 3006 should get the value 36, now let us see. So we have only till… this is a global stack (())(32:05) so 3006, be careful in just scrolling this, I can just so I am just giving you so that scrolling will become extremely complex like it becomes slow now I can go and say so 3006 gets 36, I can use this search option okay.

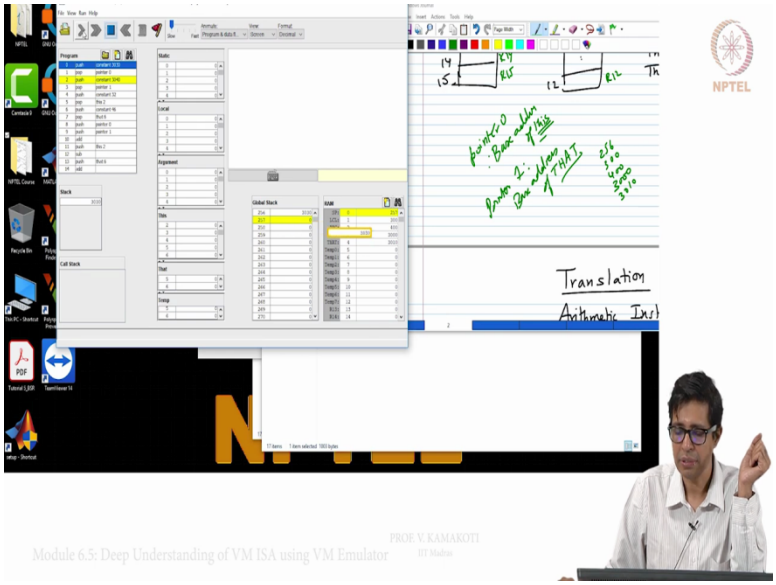
Now again push constant 42, 45 now pop dot 5 so that is at 3010 + 5 = 3015 you pop 45, so 3015 should get 45 and 3012 should get 42 because you do pop that 5 and pop that 2 let us do that. So how did you calculate this 3015 because that in the assembler single table is 4 so go to that address 3010 add 5, 3015 then you pop to that and… So let us go and search for 3015 and 3010… 45 and you also see 3012 getting 42 yeah… Now let us do the next one pop temp 6, Temp is an array as you see temp will always start from 5 so now that 510 which is on the top of your stack should go to. The base address of temp is always 5, 5 + 6 is 11 so into the 11^{th} location 510 should go on, let us see whether it is going, so on the 11^{th} location in the RAM you get 510 right and then the stack actually gets stopped.

So the difference between the other things that we have seen like pop that, pop argument, pop local is I go to the location, from there I get the base address and then I add that base address to the index and then push for pop to that from that. But here when I say temp, the base address is already given so that is always 5 for temp so I just add 5 with that 6 so I need not do additional memory access to find the base address, base address is by default 5 right. Now push local 0, local is 300 so what is there in local 0 that is 300 we have 10, in location 300 we have 10 that 10 will now get pushed into the stack and the stack will now become 256, let us see whether it is happening… that is 3010 that + 5 is 3015 so that in 3015 there is 0 right,

sorry 3015 currently has 45, now that 45 has to get you know pushed onto this stack right, pushed that + 5, 45 essentially has to get pushed up right, 45 gets pushed up.

Now we need to do add so this 45 + 10 should become 55 and the stack pointer should become 257 now, let us see that is happening… yes. Now I will say push argument 1, argument 1 is 21, argument one is 401 right 401 has 21 the base address for R is 400 so 21 essentially comes here. Now I do was abstraction of X - Y so this is X is 256, Y is 257 right, so 55 - 21 should essentially become 34. This segment is total 3000 so I have to push this value 34 to 3000 + 6 = 3006 sorry whatever is there in 3006 which is 36 will now get pushed to a stack, 3006 had 36 right and again I am pushing the same this + 6 another 36 I am pushing. Now I am abstracting these 2, first Y, second Y, I am adding these two I am getting 72, again I push it back into the stack, now I am subtracting these 2; 34 – 72 that will give a me – 38.

(Refer Slide Time: 42:11)

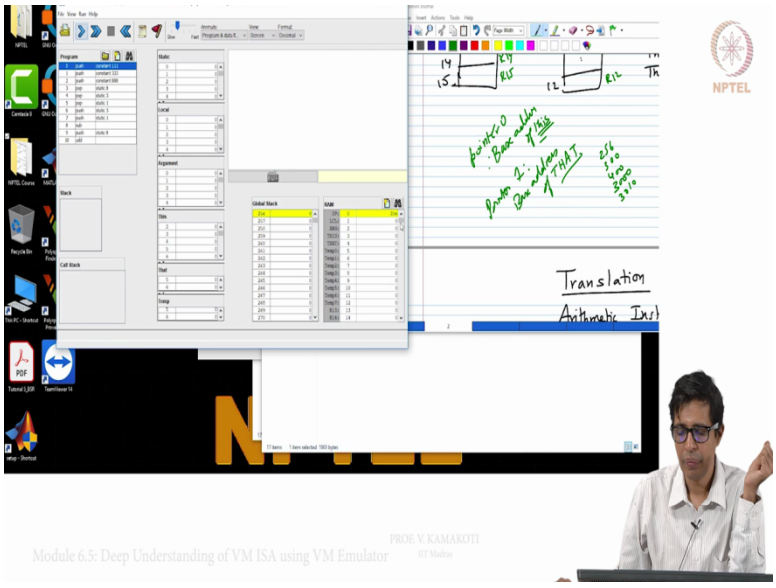


Now I am pushing temp of 6, temp of 6 is 510, please note here you are seeing that 510. We are adding that – 38 + 510, 472 should come back and that is end of this code. So I am going through this very slowly because we try and get to understand how the intellectual machine works. We have to do two more codes here, so please bear with me for this, this is pointer test, so we will again set this up this as 300, this has 400, 3000, 3010, okay. Now push constant 3030 it is gone in, your stack pointer has become… Pop this to pointer 0 so what is pointer 0? Pointer 0 is the value of the base address of this, please understand this we have been explaining in detail earlier but we will concretize that explanation.

Pointer 0 is nothing but base address of this segment, pointer 1 is nothing but base address of that segment okay. So when I say pop pointer 0 that means whatever is there on the top of the stack that is 3030 should become the base address of this, the base address of this is stored in 3 so 3 location 3 should now become 3030, let us see if it is going to happen… Yes look at the animation, yes it is going to become 3030 and the stack is now popped off. Now push 4040, now if I say pop pointer 1, the base address of that should become 3040 let us say it is going to happen yes that becomes 3040. Now pushed constant 32, pop this to what is this now? 3030 + 2 is 3032 into location 3032 write 32 so 32 will be written into 3032, so now we can see search for 3032 you will get 32 okay.

Push constant 46 , push pointer 0, push constant and pop that 6 right, pop that 6 will become 46 that is 3040 so 3046 will get 46 yes. Now we will say, push pointer 0, pointer 0 is the base address now 3030 will be now pushed into the stack, 3030 will get pushed into the stack. Push pointer 1 3040 will be… Pointer one is the base address of that that means pointer 1 is always 4, pointer 0 is always 3, so whatever is the content of RAM location 3 will go to the stack. Now I will add this so this will become 6070, now the stack is now having only one element now push this to, what is this? This 2 has 32, this is 3030, this too is 3032 and it has 32 right so 3032 will now go into the stack.

(Refer Slide Time: 46:15)



Now I will subtract this so this will now become 6038 and push that 6 that is now 3040 so 3046 I will push, whatever is there in 3046 I will push, in 3046 we actually have 46 that 6 we have 46 as you see okay so that 46 will now get pushed. Now we add these two, this will give you 84 6084 and that is it okay. And the last program that we will feel before the wind up, if

you are feeling bored but still bear with me but please understand this because anything tough will come with a pinch of salt, now you need to have lots of patience to understand and take it forward so static is very-very important.

So where will the static segment start? Your assembler normally starts assigning variables from 16, if you remember if it finds some variable it will start, everything else is in the stack, what will be normally variables if they are actually local variables, they will be in the local, if they are arguments they will be in the arguments, other than local, argument, etc, the other variables that are not there or not map to local and argument, they are all going to be static variables and their assembler for all the variables that it sees right, it starts locating locations from 16, etc right.

So **so** in this case also we will assume that so let us see what is going to happen in this case, first we are pushing 111 then 333, then 888 now I said pop static 8, so where will static 8 do, let us see what is going to happen. So basically this should have gone to location **23** it is going to 24. 16 so it is that assigning from 16, 15 will be static 0 so 8 variables so it goes to 20 so all the static variables you start assigning from 16 onwards okay so **so** essentially this goes there. So when we are doing the translation, we will just leave it as that variable name some will give a variable name because automatically the assembler will do this allocation, so when we do the translation we need not do the allocation fall static variable, we will just say some variable name and automatically the assembler will do assign this say 16 to this thing right.

Similarly, pop static 3 so 333 should go to 19 yes 19 got 333, similarly static push static 3 so 333 the pop static 1 so 17 got 111 right and now push static 3 so 333 should get into the stack and push static 1 so 111 should go there and then I do subtraction so 111, 333 is X remember that so 222 should be on top of the stack, and top of the stack again goes to 257 because we popped those 2 values and push this. Now push static 8 it is 888, now I do an add and it should give me 1010 yeah good.

So what you have understood so far in this thing, thanks for your patience I think we have now understood very clearly how these stack arithmetic and memory instructions work right, now we will quickly go into now we are going to translate these instructions.