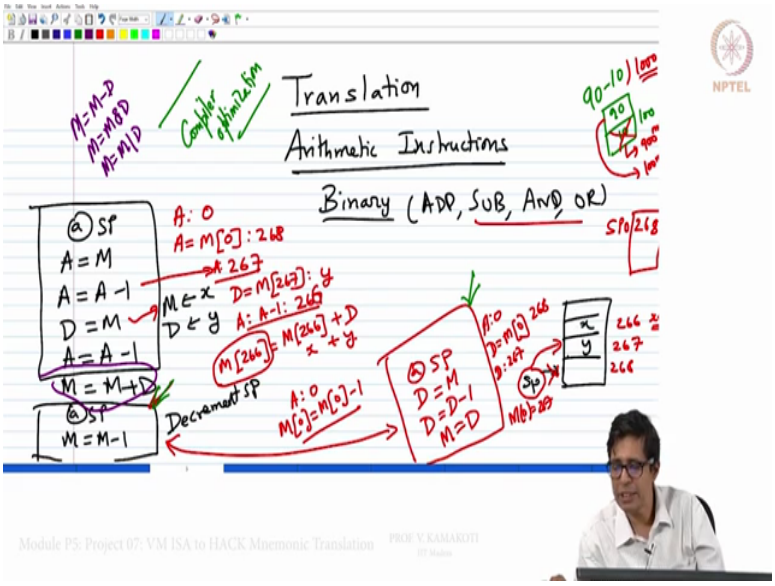
Foundations to Computer Systems Design
Professor V. Kamakoti
Department of Computer Science and Engineering
Indian Institute of Technology, Madras
Module P5
Project 07: VM ISA to HACK Mnemonic Translation

(Refer Slide Time: 0:16)



So how do I translate so this is how this is stack pointer and this is x and y, if I get an arithmetic instruction like ADD, SUB, AND, OR first I will do at SP, SP will give me the address of the stack pointer. So let me say SP is say 268, this is RAM location 0 and that is 268, when I say at SP your A register will get 268, when I say A equal to M, A will be A of M of 268 sorry A equal to M sorry when I say at SP A will get 0, when I say A equal to M the A will be M of 0, so A will get 268.

Now I say A equal to A minus 1, so this will become 267, now D equal to M, M of A which is M of 267 this will be now whatever y. Now A equal to A minus 1, so A becomes 266 because A has already 267, now 266, now I now say M of 266 is equal to, what is M of 266? So 268 is SP, so this is 268, 267 is y, 266, M of 266 is equal to M of 266 plus D this is x plus y, right and this will be stored at 266.

So 266 now will get x plus y, again I do at SP, A is 0, I do M is equal to M minus 1 so M of 0 is equal to M of 0 minus 1 so your stack pointer now will become 267, so this will now start pointing to, so what I have done I have retrieved x, I have retrieved y, first I retrieved y then x for that I went to the first I up went to the stack pointer, got the value of the stack pointer into

my A register, decremented it, copied the value of (D x) y on to D that is D equal to M and then decremented my A register that will now point to x, now I added that memory with this x plus y I get that, then again I went back to my stack pointer A 0 and decremented it so the (()) (3:18).

Suppose instead of doing at I can say at SP, A equal to M, A equal to A minus 1, right. The way to decrement SP can be like I can say at SP, so your A is becoming 0, D equal to M of 0, so let us say SP now is 267, so D will become so what generally it was 268, right to start with so SP will become 268. Now I will say D equal to D minus 1, so D becomes 267. Now I will say M is equal to 3, so M of 0 is equal to 267.

So this is also another way of decrementing your SP, so let us compare these two now, I prefer to do this one because there are only two instructions, if I execute this then this will be 4 instructions. So by having 4 instructions I am going to spend 4 cycles to decrement SP while I can finish this off in 2 cycles. So my program is 50 percent faster with respect to this code.

And the second thing is when we go in for embedded devices today one of the important thing is that my memory has to be less in size, the amount of memory taken by my program has to be less in size, right why should the memory be less in size because these embedded devices specially in the IOT internet of things they are going to have they are going to be small small devices running with battery, memory consumes lot of power, right if I have a system with large GB's of memory it consumes more power.

So these small small systems that we throw say for at different places say for temperatures in (())(5:47) stuff like that these are going to be very small devices they are going to consume very less power. So your memory has to be very less and if my program that I am compiling is large it cannot fit into that memory, then we have a big issue. So always your compiler should generate especially for embedded system should generate small amount of code, your compiled code should have as less instructions as possible.

So in that context also this is this makes lot of sense, right I would like to have this (())(6:24) this way of decrementing pointer rather than you know this way of doing this rather than doing (())(6:31). So I get 50 percent reduction in storage and also 50 percent reduction in execution time by doing this is what we mean by compiler optimization this is what we mean by compiler optimization, right.

And very importantly the other path that we need to keep in mind is that okay, this particular ADD this makes sense this 4 to 2 suppose this ADD is executed 1 million times by using this decrement I will be spending 4 million instructions by doing this decrement 2 million instructions, so the saving is 2 million instructions if this particular ADD is going to execute million times.

Now how can an ADD execute million times? Obviously if it is in a loop, right we have written loops, right in the assembly program if it is in the loop of course it will now a single ADD is in a loop of 1 million iterations it will execute million times and every times it executes this increment that you see on the right hand side is 4 and that is 2, so you spend 4 million versus 2 million.

So that is also something that we need to, even though per instruction if I am just removing one instruction per translation that essentially can translate to millions of instructions being saved or millions of execution cycles being saved, right. In this context I should also talk to you about this what we call as 90-10 rule, so I have a program say which has 100 instructions there will be a core set of 10 instructions and suppose this program is taking 1000 units of time, there will be a core set of 10 instructions which will use 900 units of time, the remaining 19 instructions say 900 milliseconds and the remaining 90 instructions will take only 100 milliseconds.

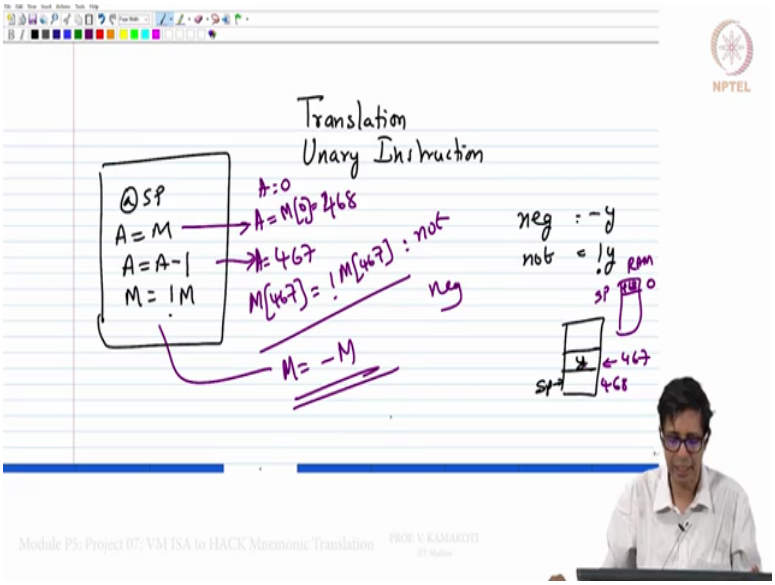
Why these 10 instructions alone take 900 milliseconds? These 10 instructions are in a loop, so always 90 percent of your program will take 10 percent of your time, 10 percent of your program 10 percent of the instructions in a program will execute for 90 percent of your time, this is an empirical if you ask me for an analytical proof I do not have but this is an empirically validated software engineering TRM software engineering rule it is called a 90-10 rule.

So 90 percent of your program will take only 10 percent of time, 10 percent will take 90. If this ADD is normally these arithmetic operations will be in that 10 percent code and that will take lot of time. So optimizing I am just basically giving you a (())(9:42) of the iceberg of what we mean by compiler optimization and this is one such at the backend.

Now we know how to do for ADD, SUB, AND, OR the three other operations are very easy. So instead of this M is equal to M plus D, right if you write M is equal to M minus D same code with M is equal to M minus D will take care of sub M equal to M AND D will take care

of AND, M equal to M OR D will take care of OR, right. So this is how the binary operations get translated.

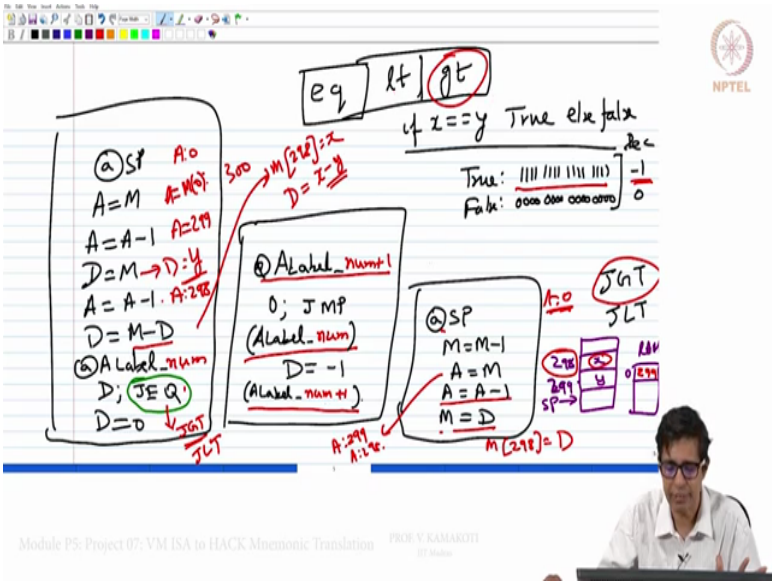
(Refer Slide Time: 10:24)



Now let us go to the next one Unary translation, we have done it for neg so again let SP be some 468, so 468, this is RAM 0, so this is 467, very quickly at SP means A becomes 0, now A equal to M of 0 that is the second instruction A is equal to M of 0 that becomes say 468, A equal to A minus 1, that is 467, M of 467 is not of M of 467 that is done, so the y it will do a bitwise not of this.

Now the stack need not be (())(11:15) need not be decremented just I am taking y and I am popping y and I am making negation (of y) not of y and I am writing it back so this works, so the stack pointer remains the same there is nothing so I am pushing, popping and again pushing it back so in that thing, so this is thus just these 4 instructions but if it this is this will work for not, if it is for neg just write M is equal to minus M that is all, this is for the unary translation.

(Refer Slide Time: 11:50)



Now for equal to, less than and greater than again like this is RAM let SP be say 300 for a reason, so this will be (300) 299 and 298. So at SP, so A actually A becomes here, A equal to M of 0 which is 300, A equal to 299, D gets the value of y, D equal to M of 299 D is M of 299 which is y, now A becomes 298 so M of 298 is x, right M of 298 is x. So what I am doing here is D equal to x minus original D had y here, so D equal to x minus y, I am subtracting y from x, okay.

Now I have a at label underscore num, now whatever I put here in you know red for underscore num so we will be having several such (eq) equal to, less than, greater than instruction and we have to generate unique labels, so we make something like A label which nobody else will use and this num will be a running parameter, so every time we encounter a eq a new num will be generated and it will be incremented by 2 because we will use two labels per eq instruction, similarly for less than and greater than so num will be (())(14:04).

So initially it will be A label 0, for the first time eq is coming, A label 0 and I will also use A label 1 here next time eq comes it will become A label 2 and 3, 4 and 5 and so on and either eq or less than or greater than any comparison operation comes this num will get incremented by 2, so you will get unique labels because in the assembly code I cannot have duplicate labels.

So I will just (())(14:32) at label num whatever, D; JEQ if D is equal to 0 I jump to at label num that is here and if x is actually equal to D equal to 0 means x is equal to y, if x is equal to y then I should be true so I put D equal to minus 1 minus 1 in 2's complement is all 1 which

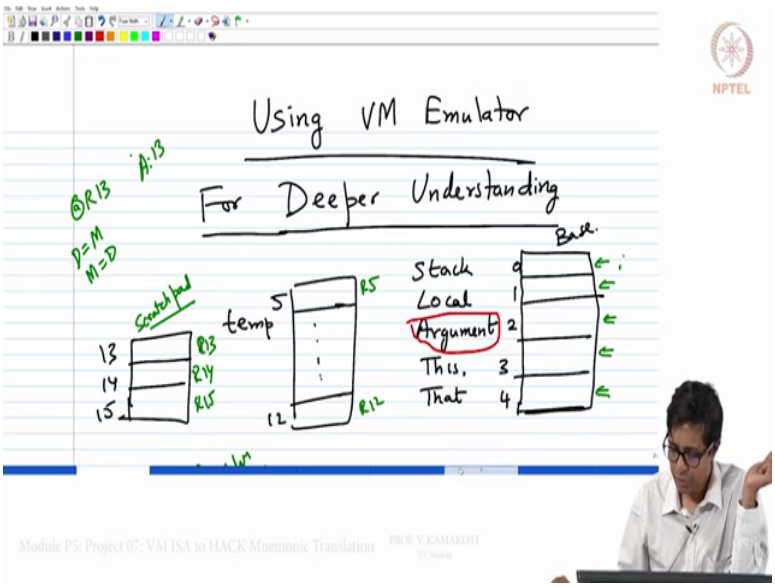
is in our case it is true and when D equal to otherwise I make D equal to 0 and at label num plus 1 and I jump 0 colon jump and this is at label num plus 1, so two labels are there here.

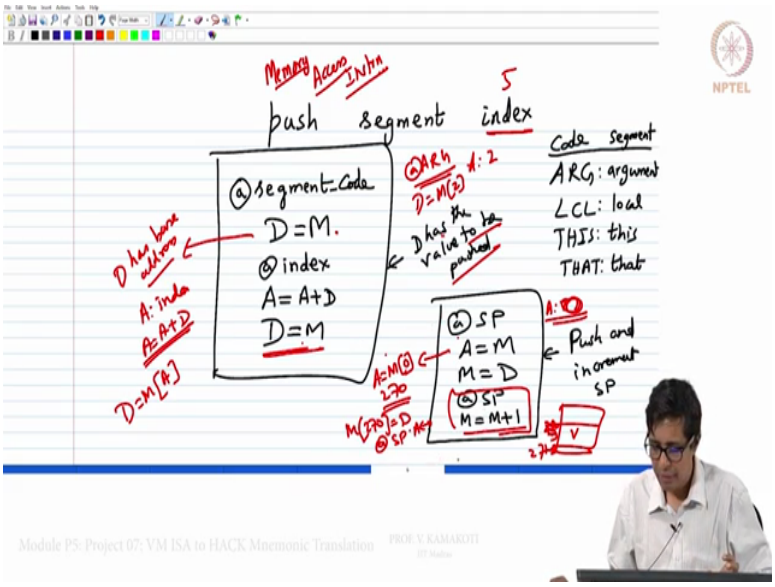
So what will happen is when D is equal to 0 that means x is equal to y, I will jump to this num make D equal to minus 1 and come out, otherwise I will make D equal to 0 and I will jump by passing this D equal to minus 1 to this at A label num plus 1, right. So what will happen this is what happens, so at the end of these two exercises my D will have the result of the operation that is if x is equal to y D will have true which is minus 1 and (if it is false) if x is not equal to y it will be 0.

Now what I need to do that value of D I need to push into the stack. So how I do? Again at SP so A gets 0, M equal to M minus 1, I reduce the stack, right now I have taken two opponents out and I am (())(16:05) so your stack pointer will now become 299 and A equal to M, right so A equal to M of so A is currently 0 A equal to M of 0, so A now gets 299 again A equal to A minus 1 that is A becomes 298, right 298 on 298 M equal to D so M of 298 is equal to whatever value of D true or false that will go here, okay that is all so this is how eq.

So if I want to do LT or greater than instead of JEQ, I can put JGT then this will become x greater than y implementing gt if I put JLT then this will be J less than y (sorry) this will be J less than y, so this is how I implemented these three operations the important thing is I am assigning labels and a care has to be taken that whenever you assign labels it has to be unique and that is this is how you do EQ, LT and GT.

(Refer Slide Time: 17:38)





Now the last two instructions this is the so now we go on to the memory access instructions. So let us look at so push segment index, so what are the segments? We have argument, local, this, that, first these four these four is the case first and four most I need to get a segment code because push segment index means take the base address of the segment add the index to it and to that whatever is there in that location take it and push it into the stack that is what, right.

So this particular first part of the code will generate the will get the whatever I need to push that will be available in the D register what I just say suppose I say push argument index I have to say at ARG, at ARG will give me the base address of the argument segment. Now that D so A will be so at ARG so A will now become ARG stored somewhere, right 0, 1 or 2 whatever.

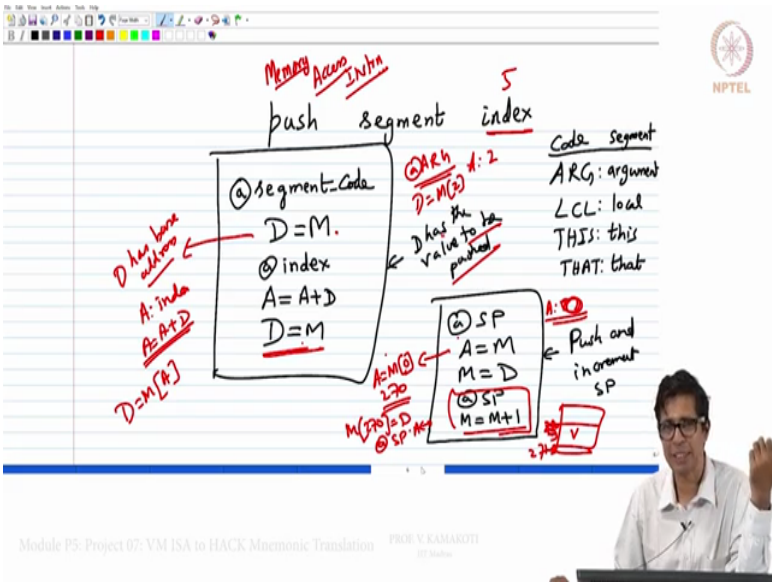
So A will get that 2 here, right A will get argument is 2, argument is 2 so A will get that 2 here, D equal to M of 2, so D will get the base address of the segment D has base address of the segment whatever segment you are asking for, now this code is valid for argument, local, this, that, these are the four segments that we can give first. Then at index so this is just take this index so which is say if it is 5 or something so that so A will have that index value.

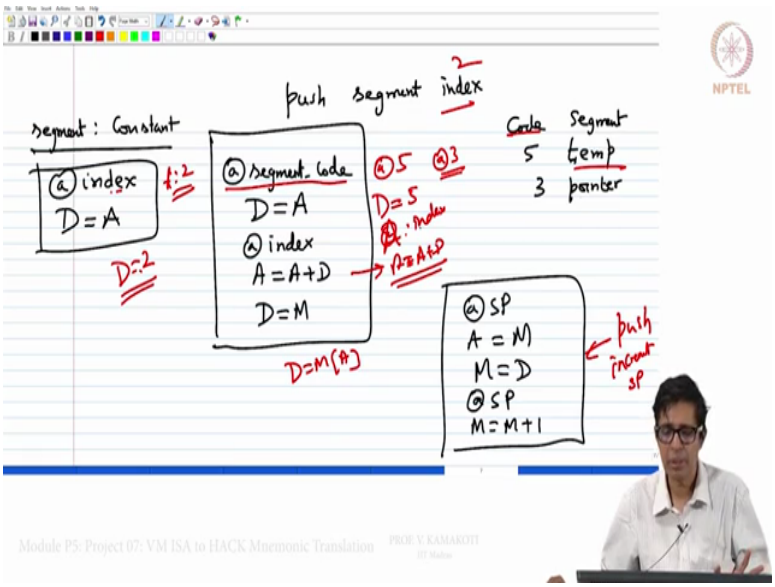
Now I am adding A equal to A plus D, so this will give me the exact address from which I need to take the variable because base plus that index so D equal to M of A, So D will at this end of this D will has the value that needs to be pushed, then now this is very simple at SP so this is a stack pointer so this will say let the stack pointer be you know pointing to 270, so (A will get 270) A will get 0 so A equal to M this will make at this point A gets 0, so A equal to

M of 0 now this will get 270, M of 270 is equal to D, D has the value that needs to be pushed that will go here.

Again I will do at SP, so A gets 0, I do M equal to M plus 1, so this will become 271, so this value gets here and your SP now becomes 271. So this is how I increment the SP, push (()) (20:48). So this is true for argument, local, this and that.

(Refer Slide Time: 21:00)





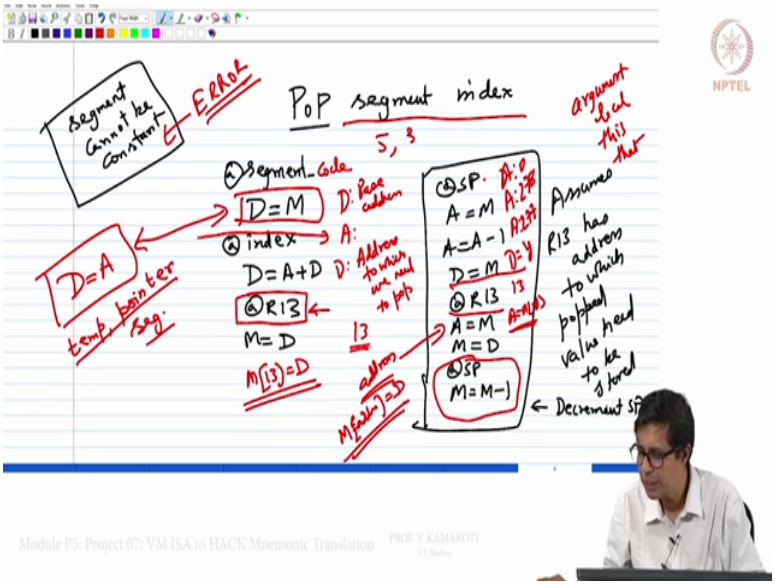
For temp we know that the base address is 5, so the difference between other segments and temp is already explained when we want to have the base address I have to go to some memory location and get the base address in the case of argument, local, this and that, I have to go to location 0, 1, 1, 2, 3, 4 and get at. But in the case of temp we know that the base addresses always starts at 5.

So I will just say at segment code so when we say temp the segment code will be 5, this is the code if I say pointer always see your pointer, right pointer actually points to the base address of this which is 3, so always pointer will start with 3, right because the address of the base of this is stored at 3, we have already seen in the emulation at segment. So if you are saying so we so it will come at 5 or at 3. So your D will be 5, or 3 depending on this.

Now I have an index, so this is 2 or something in the case of pointer it can be 0 or 1, right so that index so A will have the index. Now A equal to A plus D will give me the address where I need to address from which I need to take the value, so now D equal to M of A. So at the end of the story D basically has the value that needs to be pushed and this is exactly I push it into the stack the same code the same thing I push the value into the stack and I increment of that stack pointer.

If it is a constant, right just I say at index D equal to A because D now has the value that I need to push this, if I say constant 2, I need to push 2 into the stack, so I will put at 2 so basically A will get the value 2 and D equal to A so D gets the value 2 and so D has the value that needs to be pushed again I push and increment the stack pointer. So this is how push will work for different segments. So we have covered all the segments here argument, local, this, that, that four of them and then temp, pointer and that, right.

(Refer Slide Time: 23:36)



Now the last one that is pending now is we have still to talk about static, okay we will talk about static a little later, okay. Now let us now talk about PoP segment index, so again what we get here is for this so this is whatever the segment cannot be a constant here, right I cannot

pop into your constant, right, I can only push a constant pop into a constant. So I need to generate an error for that, okay I need to generate an error for that so this is at segment code.

So for the remaining things, right the argument, local, etc this, that, this is the code. So we will say at ARG or at LCL or at this and at that so we will get that 1, 2, 3, 4 and D equal to M, so D basically has the base address of the segment. Now at index so A as the index so D actually has the address to which we need to pop, the popped value has to go to the address so D has that address.

This address now I say at R13, as I told you R13, R14, R15 are temporary storages which we can use, right they had temporary storages so I just use at R13 here. So this will give me 13 RAM 13 in which I am basically storing. So M of 13 is equal to D, M of 13 is equal to D so the value of this address to which we need to pop that address to which I need to pop is stored in M of 13, so at this. So R13 has the address to which the popped value need to be stored that is there.

Now what we do? Now we pop the value at SP A equal to M, so this will give you so let the stack pointer be at 298, so this is A will get 0 and A equal to M of A so A will get now say let us say 278, A is 278. Now I need to pop, right I become A is 277 and D is equal to M of 277, so whatever I need to pop, this is let us say I am popping that this is the value to be popped is now at D, right.

Now at R13, so this will be 13, A equal to M of A now A will get because R13 has the address to which we need to pop now A will now have the address to which we need to pop. So M of A and D has the value that is already popped out so M is equal to M of that address is equal to D, right so the popped value goes to that particular location, right. Now again at SP again A becomes 0, I decrement the stack pointer using this, right.

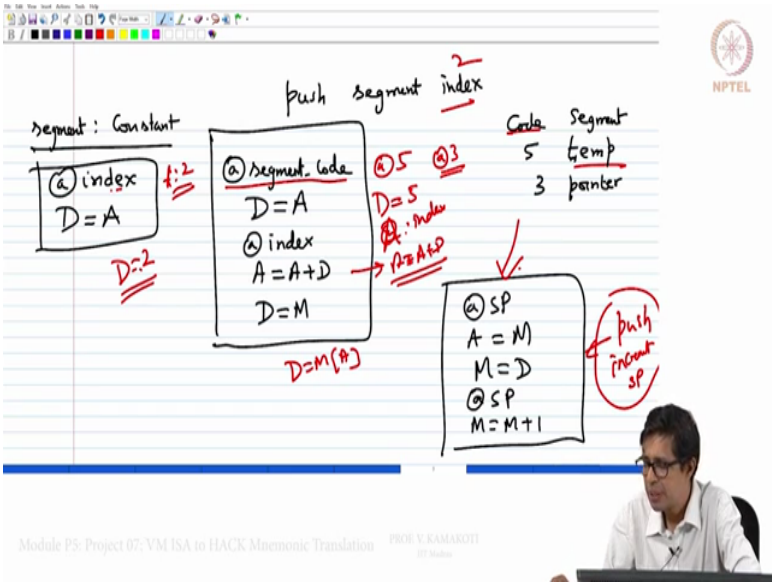
So what I am doing here? I am getting the value of the stack pointer, I am decrementing it by 1 to get the value that needs to be popped that value I am storing it in D, I go back to R13 at R13 and I retrieve the address into my A register again to which I need to store this popped value that is currently in D and I store that value into that particular memory location and I decrement by that.

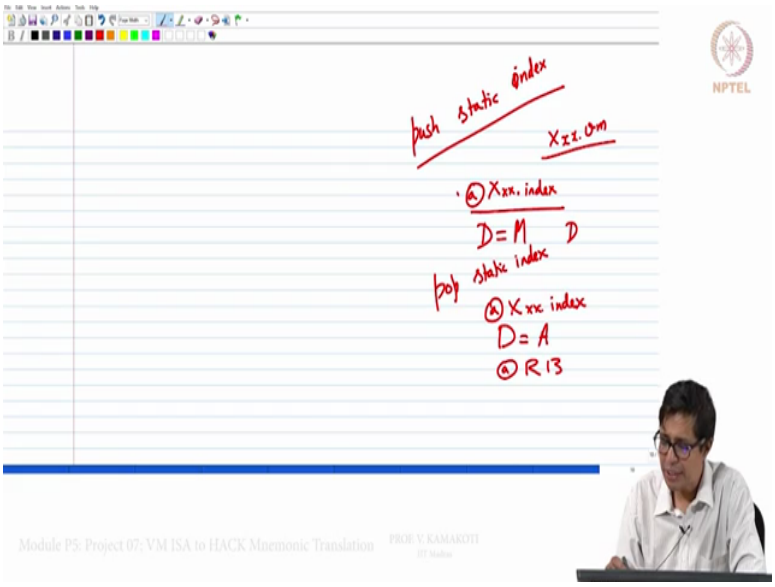
So this is what we do for pop segment, in the case of the segment being argument, local, this and that if the (())(27:41) this is temp 1 pointer then instead of doing D equal to M, I need to do D equal to A, right because in the case of temp and pointer you will get temp the segment

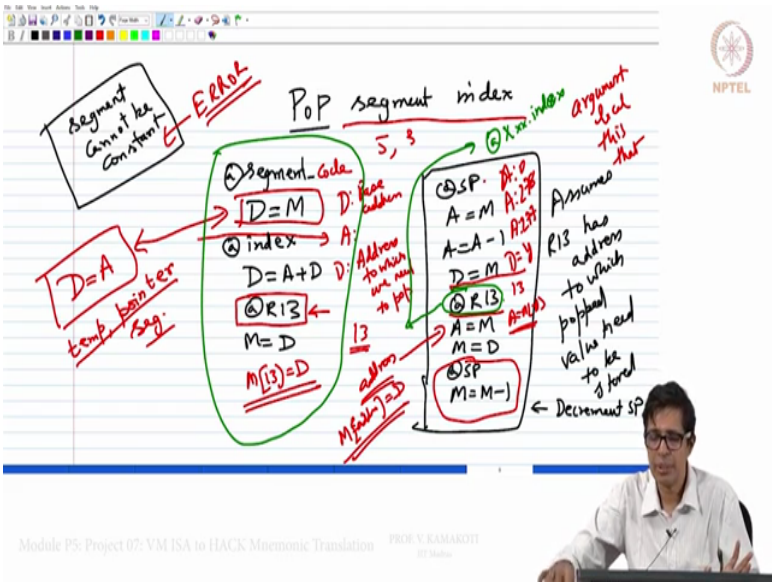
code will be 5 and the pointer code will be 3 so D will now get the these needs the base address it will be 5 or 3 depending upon the base.

So if this segment pop segment is going to be temp per pointer then instead of D equal to M, I will have D equal to A. Ultimately at this end of this particular two instructions D should have the base address and that is basically 5 or 3 in the case of temp per pointer respectively. So this is how we work with pop.

(Refer Slide Time: 28:38)







So one of the thing that we need to basically look at is what will happen if I say push static some 0 or whatever index, all the static variables will be assigned automatically memory by the assembler. So let your program be Xxx dot vm we will just say at we will define the static variable as Xxx dot say index, right this is the variable and now your D will become M, right D equal to M, so at Xxx dot index will be the address of that variable (())(29:45) will automatically D equal to M.

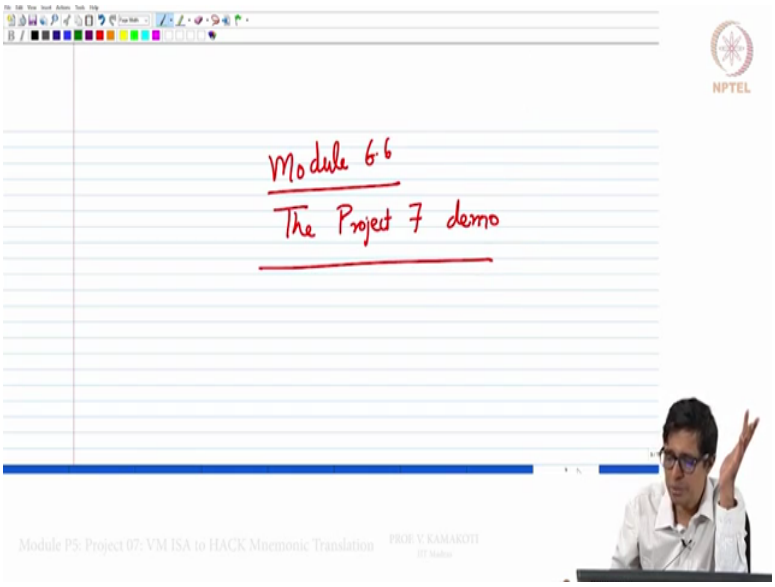
So D will have the value that needs to be pushed, so when once D has the value that needs to be pushed, right the moment D has the value that needs to be pushed then everything else then I just push an increment (())(30:01) I assume that D has the value to be pushed, right. Similarly, so then I have to just use this code at whatever code I am marking here at SP to push it that is all.

So but on the other hand in the case of pop so but suppose in the case of pop we have pop static index so again I put at Xxx dot index D equal to A because we need the address, right Xxx dot index so D equal to A and then at R13 and then we can so actually this is in the case of pop we can just see what will happen here, let us go to the pop, we can do much more better than this.

So all see when we looked at pop segment index, I am marking in green here all this was to calculate the address and all this was to pop to that address and here I assume that R13 has the address to which we need to pop. So instead of R13 here which I am using here, I can basically say at instead of this I can say at Xxx dot index. So if your pop this segment is static then basically I can basically go and say this is at Xxx dot index.

So rather than, right if it is static I can just go and say this is Xxx dot index, I need not have this part of the code at all because at Xxx dot index will give you the address here so we just have this part to go ahead. So this is how we do the push and pop so this is sort of a very large module that we are done so far I think the longest module but please you know understand in this full aspect we have made an honest attempt to give you the complete understanding of how the whole arithmetic and the memory access instructions work and this is extremely important because this is the first introduction to backend of compiler and I want you to enjoy this and do, right.

(Refer Slide Time: 33:11)



Now we will go into the next module 6.6 where we will do a demo of the project 7, thank you.