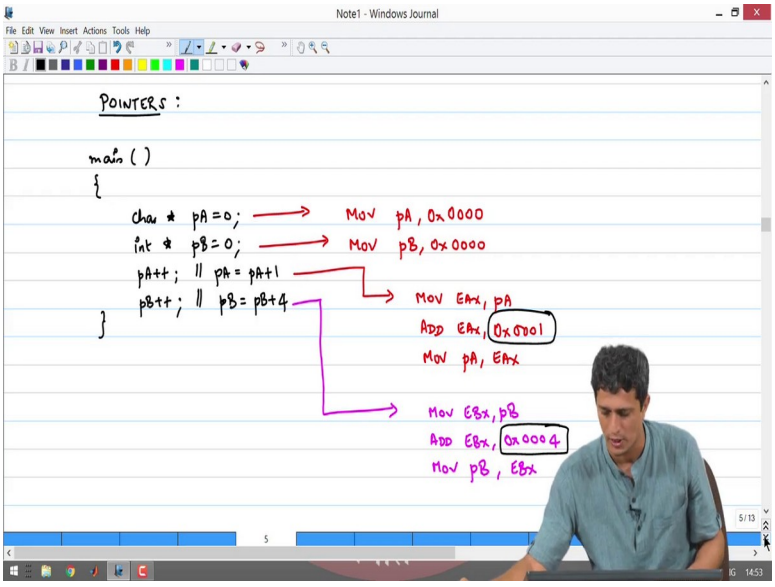
C Programming and Assembly language
Prof. Janakiraman Viraraghavan
Department of Electrical Engineering
Indian Institute of Technology, Madras

Lecture - 08

Welcome back to this course on C Programming and Assembly language. So, we were in module 2 discussing inline assembly and C programming. So, let us proceed with our discussion and in this lecture we will look at the topic of pointer arithmetic at an assembly language level .

(Refer Slide Time: 00:29)



So, pointers so, let us start with a simple C example. So, let us consider a very simple example here, I have char star pointer pA equal to 0 and I have an int star pointer pB equal to 0. So, what we want to understand is when we do pA plus plus and when we do pB plus plus what is the difference at an assembly language level .

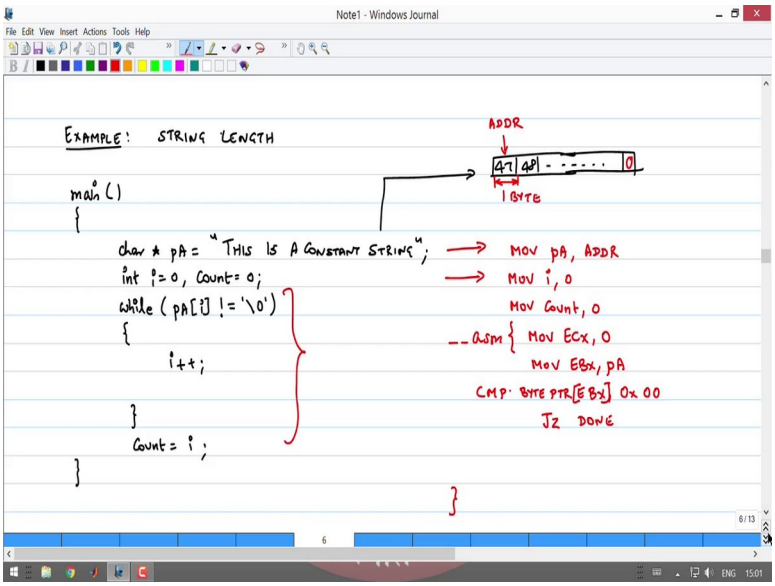
So, let us look at the assembly translation of this program first. So, since we have char star you know its just a data type , there is a variable pA and that is being initialized to 0 . So, if you look at this, this will simply translate to MOV pA comma 0 x 0 0 0 0 right how many over 0's that are and if you look this it would translate to MOV pB comma 0 x 0 0 0 0 . So now, the point is I am doing a pA plus plus . So, what does this instruction

actually do in C? It is pA equals pA plus 1 and on the other hand when you do pB plus plus it will actually be pB plus 4 .

So, why is this? Because, the character pointer is a pointer to a set of characters and each character occupies a single byte in memory. On the other hand the integer pointer pB is a pointer to a set of integers and each integer occupies 4 bytes in memory . So, therefore, when I go ahead and perform this particular operation, this is what would effectively happen. You will have MOV EAX comma pA ADD EAX comma 1 or 0 x 0 0 0 1. And, then I would do MOV pA comma EAX. Similarly, if I look at the translation of this instruction it would be MOV EBX comma pB ADD EBX comma 0 x 0 0 0 4.

And finally, I MOV it back into my pB. So, this value that gets added is very much dependent on the data type and when you are dealing with pointers and pointer arithmetic specifically you will always find that the addition happens depending on the size of the data that is being stored. So, let us look at some of these examples of you know pointers and how to deal with pointers at an assembly level .

(Refer Slide Time: 04:47)



So, let us look at I want to write an assembly program to calculate STRING LENGTH of some given string . So, let us look at both the C implementation as well as the assembly implementation. So, main and I have char star pA, this is some constant string THIS IS and what I actually want to do is, I want to find the number of characters in the string. And, we all know that a string is stored as a sequence of ASCII numbers and its

terminated by a backslash 0. So, if you look at how this particular string is stored in memory, it would be a sequence of bytes which would basically look like 47 48.

These do not necessarily translate to the correct ASCII values, but this is the typical ASCII value of alphabets and so on. So, you have a whole bunch of them and after you are done with the final character you have a value 0 that is stored in that . So, if you look at this particular data size is 1 byte. So, the idea is to simply scan through the string and count the number of bytes until you hit a 0 so, that is what we want to do. So, in C what would you do? You would basically you know have an integer i equals 0 and count equals 0 .

This is where I am going to count my variable and i is just an index variable. So, what you do you basically say that while pA of i is not equal to backslash 0 you just simply do what? i plus plus and count plus plus. So, it will simply scan through each of these things, find out what the count is, I mean find out what the stored character is; if it is not backslash 0 it will just keep incrementing both the index and the counter . So, oh typically here the variable i is and count seem to be redundant, you can just do with one of them .

So, typically you could even do away with this and you just in the end of this you just say count equals i. So, if you go ahead and translate each of these statements in an un optimized manner you will get a whole lot of instructions which is very inefficient. So, let us look at the optimized you know assembly implementation of this straightaway ok. So, what does this char star pA equal to you know something mean? It basically says that this is a fixed string that is stored in data somewhere.

So, this we will translate so, let us assume that this guy is stored in some location ADDR starting address is ADDR. So, this will effectively translate to MOV pA comma ADDR, you know whatever that constant number is that constant address is. So, then you have these two instructions which basically is simply initializing i and so, MOV i comma 0 MOV comma 0 . So now, instead of going and just blindly translating I am going to take this block and do an inline assembly implementation of the block.

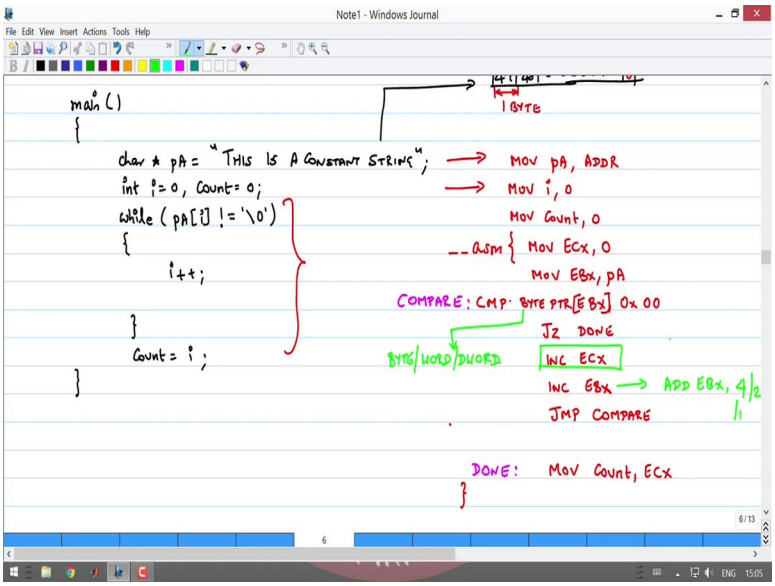
So, what am I going to do here? I am going to simply say that I need to scan character by character and keep counting until I hit a backslash 0. So therefore, what do I need to do? I need a counter register so, I will MOV ECX comma 0, then I will MOV the address

that is there into some EBX comma pA . And, what am I going to do? I am going to compare and see if the contents pointed to by EBX is 0 or not. So, I will do a compare EBX comma 0 x 0 0 0 0 .

Now, here it is important to know we know what we are comparing against. So, the 0 x 0 0 4 0s is wrong since each of this characters occupy only 1 byte what you have to look for is not a word, but a byte. So, you have to look for 0 x 0 0 and the prefix that you will add here is going to be word point byte pointer , byte pointer pointed two by EBX. If this is 0 0 0 then essentially I am done ; so, this is jump on 0 flag to DONE . DONE is basically some address in the code segment, I will show you where that label will come in the end.

Now, if I do not encounter a backslash 0 which means that my compare instruction has not resulted in 0, it means there is a valid character and I need to continue counting. So, therefore, what do I do is when the condition fails so, jump on 0 to done is going to go to another location . So, let me just scroll down a little here yeah so, let me take this off.

(Refer Slide Time: 12:29)



So, my DONE is somewhere here ok. So, we will fill in what the instruction has to be a DONE a little later. Now, if the jump on 0 condition does not happen which means that there is a valid character I need to continue counting. So therefore, I will continue counting by incrementing my ECX right and also incrementing my EBX. So, if my jump on 0 condition is not satisfied, it means there is a valid character and I need to continue

counting. And therefore, I will unconditionally jump back to my compare part which is basically so, my let me put the labels in a different colors.

So, if the jump on 0 does occur, it means I am done and the string length has been calculated and the value is available now in my ECX register. And therefore, if I want to calculate the you know put the value in count, then all I have to do is MOV; let us use the red here MOV comma ECX . So, what we have done here effectively is just simply scan the array for a backslash 0 and counted till we encountered a backslash 0. So, the important thing to note here is while it is for me to use this increment operation out here on ECX, I may not be able to use the INC EBX always.

Because, INC increments the count, the value of the register exactly by 1, but suppose this where an integer array instead of a character array then I have to replace this with an appropriate instruction which is basically ADD EBX comma maybe 4 or 2 or whatever, 2 slash 1. And, also instead of this byte pointer I might have to simply put BYTE WORD slash DWORD. And, even the constant that I am going to search for is now going to be either an 8 bit number, 16 bit number or a 32 bit number. So, with that we have the looked at the assembly level changes that come in when we deal with pointers.

(Refer Slide Time: 15:37)