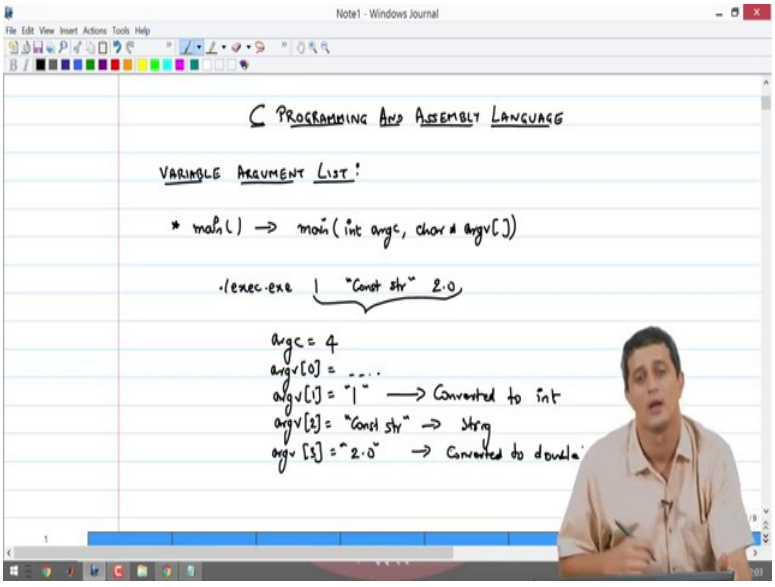
Programming and Assembly language
Prof. Janakiraman viraraghavan
Department of Electrical Engineering
Indian Institute of Technology, Madras

Lecture - 17 part a

Welcome back to this course on C Programming and Assembly language. So, in the last module, we looked at how C program is translated to its assembly output and we also saw how local variables are accessed parameters are accessed and so on. So, in module 4 we shall look at certain examples and look at the effect of translating an a C program to its assembly equivalent in the way that was discussed in module 3 . So, in particular I would like to discuss some performance impact .

For example, recursion versus loops, then I would like to discuss certain optimization that can be used to speed up C programs, by using specific hardware instructions right. Then I would also like to look at the implementation of a function like print f which has a variable argument list. So, in this lecture let us start with the implementation of print f .

(Refer Slide Time: 01:13)



And specifically I am going to discuss Variable Argument List how do you handle such variable arguments in a C program .

So, before we start let us let me point out that there are two kinds of variable argument list functions that are encountered in C ; one is a function like main and the function

declaration of main is as follows . You have main int arg c and char star of arg v . Now, how does this handle variable number of arguments? So, when you invoke the executable , you could invoke it with variable number of arguments. For example, I could call this executable . Let me just call it as dot slash e x e c dot dot e x e in windows and I could pass arguments as one may be const string and maybe one more floating point number .

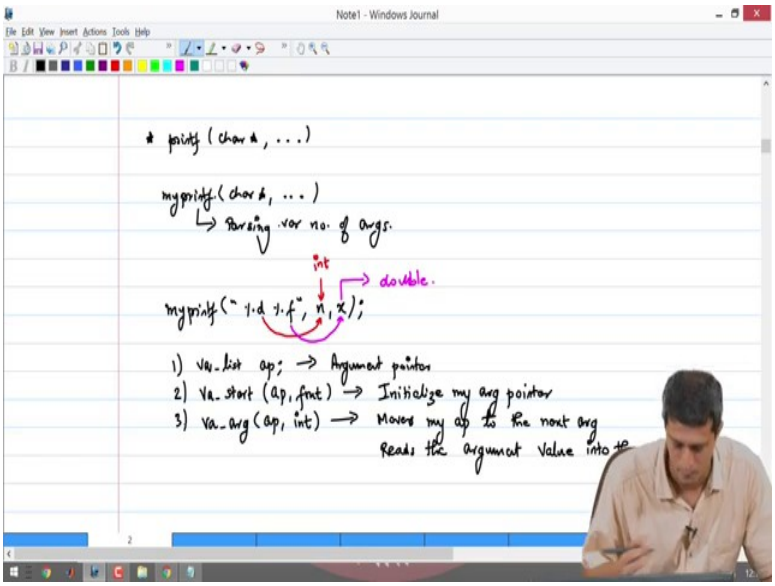
So, if you look at the call to this executable and which in turn is actually translated to the call to main, it has 3 arguments here . Alternatively, I could call this without any argument . Nevertheless, the way main handles variable arguments is by treating all of these arguments as character strings . So, affectively what will happen is when main is invoked the operating system knows that there are these many parameters to it.

And therefore, it will invoke main with arg c ; arg c equal to 4 actually because the now the executable main itself happens to be one of the arguments and arg v of 0 will be one will be the executable name let me erase this.

This is some special thing arg v of 1 will be string 1; arg v of 2 will be this constant string; arg v of 3 will be 2.0. So, if as a developer I want to use these arguments in my function , in main what I need to do is to parse this strings by going through all the arguments that are parsed which is given by arg c . So, I go to the ith argument and convert that to either an integer or a double because I have to assign that up front by the way . I need to assign that up front and I will just convert this string.

So, this is converted to int and this is of course, just a string and this is converted to double. So, the owners of actually parsing this string and converting it to the data types that I need in my program is on the developer. So, strictly this is a variable argument list function only in terms of the number of arguments . I cannot parse arbitrary data types to it and expect that the function and handle these data types.

(Refer Slide Time: 05:42)

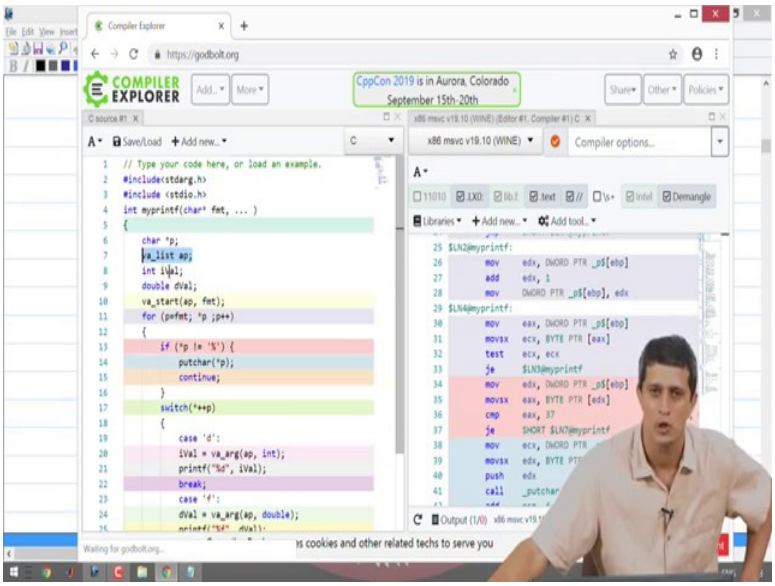


On the other hand, it is very interesting to note a function like print f . Printf basically has this declaration char star and then, it says dot dot dot. What this dot dot dot means is that I can pass in an integer or a double or any number of arguments into this function .

Now, to understand how this function works the best thing to do is to look at an implementation . So, we are going to look at the implementation that is given in Kernighan and Ritchie and they have a definition, they have a function called my print f which has been written . So, the function my print f essentially takes exactly the same arguments char star and then dot dot dot and the only job that my print f is doing is parsing variable number of arguments.

Internally, when I want to print out the value, I still invoke the original printf function . So, latest look at the implementation first I will take you through the C implementation and then, we will get into the assembly implementation of some specific functions .

(Refer Slide Time: 07:15)



So, what you see here is let me close this output window. What you see here is the my printf function which has been reproduced from Kernighan and Ritchie exactly as it is and it takes a char star of fmt and then, as I mentioned there is a dot dot dot .

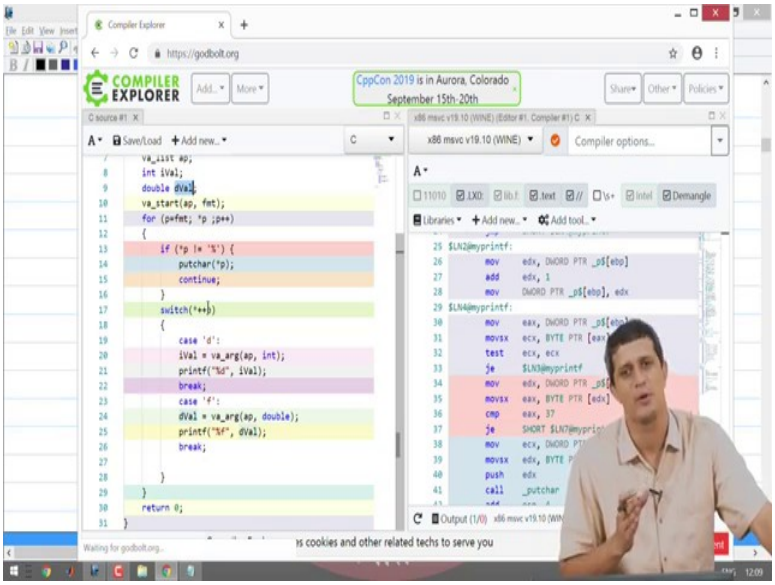
So, what does this function do? How do I invoke this function? It is going to be a exactly like my print f. So, I will say my print f percent d percent f comma n comma x; assuming n and x is an integer and double respectively . So, what I need to do is to basically parse this character string that has been given to the function and then, interpret if the following argument is an integer or a double or so on .

So, for example, this percent d will tell me that this n is an integer. Now, on the other hand, this percent f will tell me that this x is a double . So, what this function is doing is quite simple because there are certain internal functions that are used. So, if you abstract out those internal functions, then implementation is pretty simple. But our job in this lecture is to go into the assembly implementation of those internal functions as well.

So, you start with having a char star of p , some general pointer and you have a argument pointer a p which is of the data type v a list . So, the first thing that you need in order to implement this function or to handle variable arguments is v a underscore list and a p. So, what is this is basically my argument pointer .

Now, for example, in this program I am just assuming that I can parse only an integer and a double. So, I am going to handle only percent d and percent f in this example. The program can be easily augmented to handle any other data in a very similar manner . So, to obtain the value of the integer and the double, I am going to use iVal for integer and dVal for double.

(Refer Slide Time: 10:17)



So, then I have another macro which is invoked called va start . So, what I am I doing? I have another special function called va start of I am going to call it with ap comma fmt. So, this is going to initialize my argument pointer . We will see why we need this argument pointer by the way. So, after that what we are going to do is simply start parsing the character string fmt that is been parsed to this function.

Now, if there is no percent at all, there is no percent d, percent f, percent c. Then, all I have to do is simply print that string as it is . So, that is what we are doing in this for loop and this first if condition. So, for p equal to fmt basically says that initialize the dummy pointer p to point to the start of fmt and then, you check if you are reaching back slash 0; if not, you keep incrementing the character pointer .

So, inside we have an if condition, where we say that if star p is not a percent ; that means, the character that is being parsed is not a percentage, just print that character as it is. Put char of star p and then, you continue from the loop because we do not have to deal with any extra argument that has been passed to this function.

On the other hand, if it is not the case , if I come out of this if and I do not go through continue, then it means that p or the point the character there is a percentage. In which case I need to check what the data type is; whether it is a percent d or a percent f in this example or I need to check for other data types as well in a general print f implementation.

So, I am doing a switch star plus plus p which means I am incrementing p and then, checking for that particular data type and if it is d, which means it is an integer I do something. If it is an f, then I need to do something else.

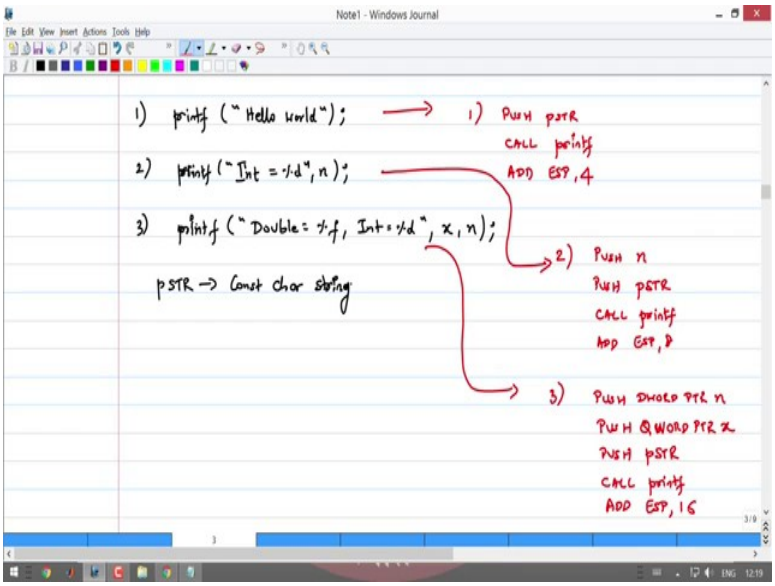
So, what is it that we are doing when we are invoke if we encounter a percent d. What we are saying is we are calling this macro again va arg of a p comma int. So, that is the third special function that we need; va arg of a p comma int. This function does two things. It moves my argument pointer to the next argument and reads the argument value into the variable. So, in this case I am saying iVal is equal to va arg of ap comma int .

Because I know it is a percent d, I need to handle the next argumentative that is being passed after char star as an integer. And therefore, I know how much I have to move by argument pointer in order to point to the next argument and also in the process I know how many bytes I have to read out of memory into my particular variable.

So, here iVal is an integer and the argument that has been passed after the char star will be treated as an integer. If is percent f, then the argument that has been passed after the char star will be treated as a double. Now, you just go through in a loop until you are done, until you are done with the entire print f string and then, you return from that function.

So, the implementation is pretty simple. Given the abstraction of these three functions ; 1, 2, 3 and so, you have v a list va start and a va arg that is being invoked repeatedly . So, our job is now to look at how I can implement this in assembly language so that I can actually process these variable number of arguments that are passed to this function.

(Refer Slide Time: 15:39)



So, let us look at three examples . Example 1, I am going to invoke printf without any you know format string. So, I am without any percent d, percent f nothing. So, I will say “Hello world” and 2nd one I am going to call it with 1 integer equals percent d comma n and the 3rd one I am going to say Int equals percent d or maybe we should do it the other way around. Because I will double equals percent f Int equals percent d comma x comma n; where, x is a double and n is an integer.

So, first let us look at what the stack implementation or what the assembly implementation of these calls will be . So, I am also assuming that P S T R is my constant character string that points to either “Hello world” or this particular string or this particular string right. So, this is my const character string . So, this particular implementation here is going to get translated as PUSH P S T R . I am going to leave out the d word pointer and the fact that P S T R is actually evp minus 4 or something like that .

And then, call printf and because this is c declaration function printf, these stack cleanup is going to happen after I return from this function. So, after I return I will say ADD ESP comma 4. Why 4? Because pointer is 4 bytes long , any pointer is 4 bytes long and therefore, I am going to say add ESP comma 4 there is only one argument that is been pushed onto stack and therefore, I will add ESP comma 4.

On the other hand, so this is implementation 1. Implementation 2 will look as follows. Now, I am going to call it with an integer. So, I will do a PUSH n PUSH pSTR remember pSTR, now points to this string Int equal to percent d and then, call printf. After I am done I will now come back and do ADD ESP comma 8. Why 8? Because I pushed 1 integer and 1 character string.

The 3rd implementation is now going to look as follows . I will specify the exact data types its important here. I will say DWORD pointer of n let us let me just say n and PUSH the variable x is a double and therefore, it is going to be 8 bytes long and therefore, it is a Q WORD

. So, I will say Q WORD pointer x; then, I am going to PUSH pSTR and call printf and once I return from this function, I am going to clean up this stack by ADDing ESP comma 8 this is 4 plus 8 plus 4.

So, I will do 16. So, as far as the calling goes there are 3 different implementations that have been shown here for the c call c calls that have been made on the left. As I discussed in the earlier lecture, pushing extra parameters on to the stack is never a problem as long as the stack clean up happens outside and that is exactly what happened happening here.

What definitely needs to be passed to printf is the character array which is pSTR if you do not call it with, if you call printf without that pSTR then definitely you will get a compiler error it will not even compile . On the other hand, as long as you pass pSTR, you are free to pass many more parameters to it and there will be no error or there will be no functionality error also when the program gets executed.

The question now is how do we go ahead and implement these two special functions va start and va arg; v a list is just a declaration and hence, we do not have to worry about it from an assembly point of view. In the next lecture, we will focus on the implementation of va start and va arg.

(Refer Slide Time: 21:53)