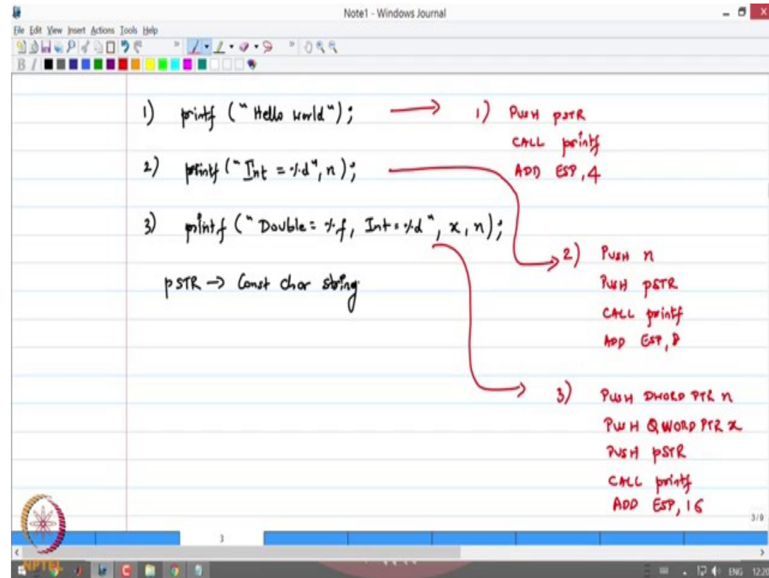
C Programming and Assembly language

Prof. Janakiraman Viraraghavan
Department of Electrical Engineering
Indian Institution of Technology, Madras

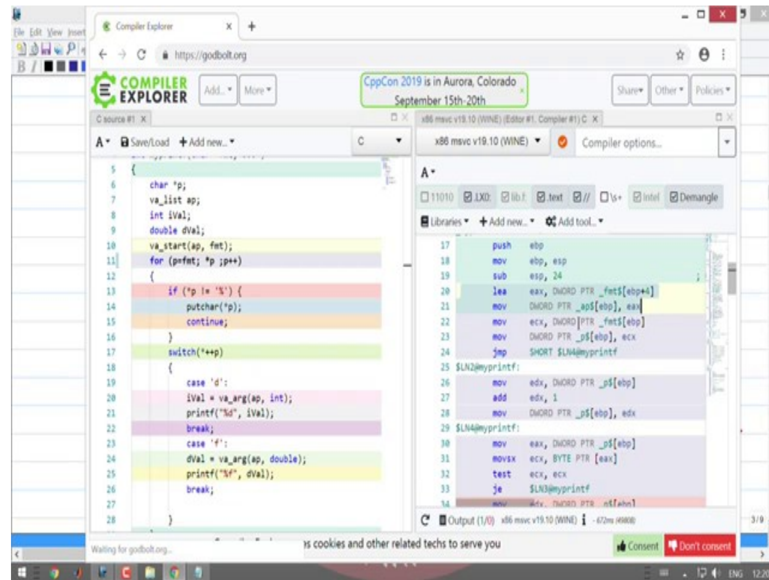
Lecture - 17 Part b

(Refer Slide Time: 00:16)



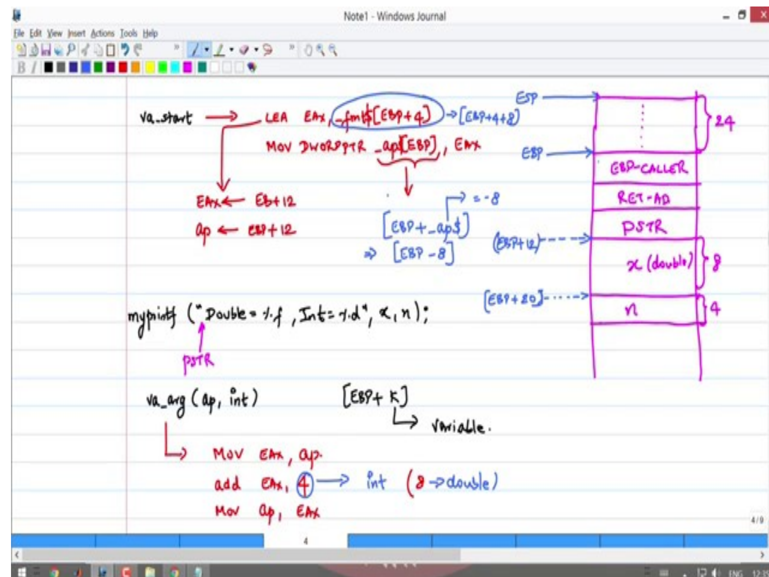
Welcome back to this course on CP and Assembly language. In the last lecture we looked at how we could pass variable number of arguments to printf and yet have the function execute without any error. In this lecture we will now look at how to implement printf and particularly we will look at the implementation of two functions va start and va arg.

(Refer Slide Time: 00:39)



Instead of looking at trying to write this from scratch we will just look at what implementation we have here, we look at the assembly output of the two functions. So, you have the va start and that translates to these two functions , let me write the down separately, so that we can yeah.

(Refer Slide Time: 01:06)



So, va start , what does this translate to? It translates LEA, I will tell you what that instruction is right LEA, EAX comma underscore fmt of EBP plus 4 and then I am going to simply MOV DWORD PTR underscore ap of EBP comma EAX .

So, before we proceed let me clarify what this particular thing means this essentially is simply going to translate to I think there is a dollar also yeah fmt dollar and ap dollar, this simply is going to translate to contents of EBP plus underscore ap dollar . So, this is just DWORD pointer of EBP plus underscore ap dollar. Now, what is underscore ap dollar that is a constant that is defined on top of the program here and you see that that is nothing, but EBP a underscore ap dollar is minus 8.

So, this guy is equal to minus 8 and therefore, this is nothing but contents of EBP minus 8 that is how you are suppose to interpret this particular nimonic . So, all you have to do is take that constant and just add it into the contents of EBP plus something .

Now, let us look at what this function is doing . So, for that let us look at the picture of the stack when we invoke function, when we invoke printf in the following way . I am going to say double equals percent f int equals percent d and x comma n .

So, what would happen? We said that the assembly implementation is push n push x and then push PSTR . So, the stack is going to look like this . So, I will push 4 bytes which is n on to stack , then I am going to push 8 bytes which is x remember x is a double and this is 8 bytes this is 4 bytes. Then I am going to push PSTR the constant string which is pointing to that this is PSTR is this particular string and then I am going to call printf or I am going to call the function my printf which is ok.

So, what will happen is the return address right will essentially get stored. So, you will basically invoke this and say return address and the moment you enter the function my printf the prologue will get executed which is basically push EBP of the collar auto stack MOV EBP to sp and then subtract ESP with some value . So, what happens is EBP of the caller will get pushed on to stack and after this my EBP will be made to point here. So, this we already saw in the earlier lectures and of course, my ESP will be pointing to some location here . So, in this example it says subtract ESP comma 24.

So, this distance would be 24 . So, now, let us look at what this LEA EAX comma underscore fmt of EBP plus 4 s. So, what does this particular thing translate to first of all let us go and look at what underscore fmt dollar is it is 8. So, this is nothing, but contents of EBP plus 4 plus 8 which is EBP plus 12 .

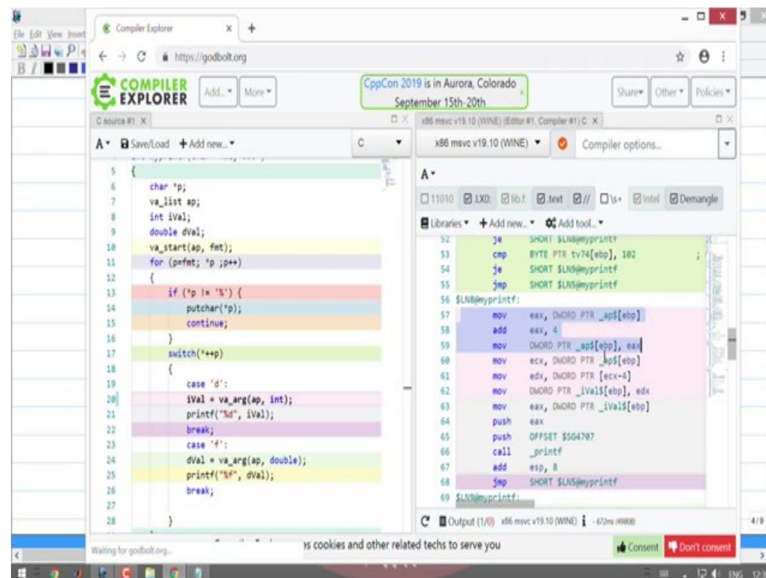
So, if this is EBP this is EBP plus 4 plus 8 plus 12. So, this is where my argument pointer is getting initialized two . So, this is EBP plus 12. So, why are we initializing it there because you need to initialize it to after the char star pointer the argument start after the char star pointer. So, you are going to initialize it to something after that . Now, what does this LEA, EAX you know comma something do? It loads the effective address on to this register which means that this particular thing is load EAX and it is a address.

So, the contents of will go and whatever address remains will get loaded into the EAX register, you will simply get loaded with EBP plus 12 that is what the load effective address instruction does. And the next instruction MOV DWORD pointer comma ap you know underscore ap or VBP is to simply assign ap with the value that has been calculated, EBP plus 12. So, now, my argument pointer variable has got initialized to the right value; so that I can start reading out these values as and when I encounter them.

So, remember now upfront I do not know if the data that is pointed 2 by EBP plus 12 is an integer or a double or whatever that needs to be inferred only from the string by parsing it and figuring out if its a percent d or a percent f. So, now, the next thing that we want to do is to look at , how you would implement va arg macro . So, we are basically saying we are going to call va arg of ap comma int let us say .

So, what is this telling us? It is basically telling us that the data pointed 2 by ap is an integer; that means, the next argument will be available after 4 bytes and that is why if you look at the implementation of this particular function here you will see that let me just get that. So, you look for the colour which is iVal equal to va you know arg v of this yeah.

(Refer Slide Time: 10:05)



So, you have three things that are being done here and we will come to what those three instructions are . Normally, if I wanted to access a parameter that is being passed, I would have simply accessed the function parameter as contents of EBP plus some constant.

The constant usually is known upfront at compile time and therefore, it is just a register indirect addressing with offset that is used to access that variable. Unfortunately here the k is now a variable because it depends on what the string is and how the function has been invoked you know as given in the three examples here that I just showed you . So, therefore, I cannot do this register indirect addressing directly because this k is now a variable and therefore, I need to evaluate what this offset is.

So, how do you evaluate the offset you just MOV the variable into a register add the offset that you want and then MOV that new value back into the argument pointer. So, if you look at the three instructions that are being implemented here you have MOV. So, what is this translate to its, MOV EAX comma ap I am not going to put the d word pointer of underscore ap basically this particular thing simply refers to the variable underscore I mean ap .

I am just moving ap into EAX, then I am adding EAX comma 4. Why 4? This 4 is because I know it is an integer and then I am moving the value back into my ap; MOV ap comma EAX. So, what was being done using a simple register indirect addressing

now has to be done in a slightly more complicated way by invoking some registers and ALU operations that is how, and its quite obvious then that is only way you can handle this variable arguments and variable data types when they are passed to these functions .

So, after suppose it where an int then what would happen is after these instructions my ap would be pointing 4 bytes away. Of course, in this case because it is a percent f that we encounter first note that the, let me just show you that . So, yeah you will note that because if the argument that was passed was a percent f which is a double then you will do exactly the similar set of instructions of moving ap into a register then you will add ecx comma 8 .

So, if I do 8; if I do 8 then that would be a double . So, in this example I have d o u b l e equal to which basically gets printed using this portion of the program there is no person that has been encountered yet, then you encounter the first percentage and then you check what the type specifier is it happens to be f. And therefore, you now try to access the data from EBP plus 12 as a double and that is what you are doing here. So, after these three instructions MOV EAX comma ap add EAX comma eight and MOV ap comma EAX my ap would be forced to point here it would have been offset by EBP plus 20 .

So, remember we are still not read the value of out into our variable we have only MOV the argument pointer to the next argument successfully. In order to implement the functionality of reading that particular argument into the variable the following three instructions are executed.

(Refer Slide Time: 15:25)

MOV ECX comma ap and then you do let us look at the case of the integer. So, that it is easier yeah and then you MOV EDX DWORD pointer of ECX minus 4 ok. Then you do MOV into iVal which is DWORD pointer of iVal comma EDX . Now, of course, if this happens to be a double instead of an integer, then all the DWORD pointers would become a Q word Q slash D and the instead of 4 I would say 8 slash 4.

So, what we are doing is very simple the argument pointer by the way because of the proceeding setup instructions has already gone to the next argument. So, therefore, in order to bring it back to read the double or the integer which is the current argument I need to do this subtraction operation; this is being done because ap has moved to next argument in the previous step .

So, this is what we are essentially doing is we are just calculating the new offset depending on the character string that was passed going to that location reading out either an integer or a double or whatever it is, and then where getting that value into whatever variable we need in order to access in order to execute this function accordingly .

At this juncture I would also like to point out that instead of an integer, if you tried accessing a character it is not necessary that you will see you know a single byte being pushed on to stack. This is because the arguments that are passed and all the memory alignment is done to 4 bytes in some compilers right all the memory accesses are aligned

to 4 bytes. And therefore, even a single byte would actually be you know sign extended to 4 bytes and then pushed on to stack and when you access the data of course, you will access it as a byte pointer instead of a DWORD pointer .

So, I will leave this as an exercise for you to implement the percent c case, and see what happens when you call it with a character instead of an integer . I like to conclude this lecture by pointing out that it is you know we discuss that arguments are pushed onto stack from right to left. So, the question really is should I push char or arguments only from right to left or can I also do it from left to right; .

So, in some sense I am asking is this just a convention and was it just a you know arbitrary choice the compilers made that they push parameters from right to left, it turns out that the answer is no this is not an arbitrary choice you have no choice, but to push it from right to left and why is that let us look at again the you know simple variable argument list passing . If I want to do print f of percent d comma n , then what I will do is we will push. So, let me just draw the picture of the stack here I am going to push n onto stack, then I want to push the constant string on to stack, then the return address, then EBP; EBP of caller is pushed onto stack and then my base pointer is made two point here.

So, now in order to implement this variable argument list I need to access the char star right. So, the declaration of printf, if is char star command dot dot dot; so, this char star I should be able to access irrespective of whether I pass other arguments or not. Now, if the parameters are pushed right to left, then the stack gets filled as follows and PSTR will always be EBP plus 8; contents of EBP plus 8.

So, there is no issue I know exactly where this PSTR should be. Now, on the other hand if I chose to push the parameters from left to right, then look at what would happen in two cases, printf percent d; percent d comma n comma n let me just have this kind of a call this is pushing right to left.

Now, what happens if I pushed left to right, then what would happen is I would first push PSTR, then I would push n, then I would push n again, then I would call printf which means the return address and this EBP of caller because the pro log has to be executed like that and then EBP is here.

Now, in order to access PSTR right in this case it would end up being at EBP plus 4, 8, 12, 16 . On the other hand if I had only a single integer that was pushed onto stack which is basically case 1 here 1, 2 . If I look at the stack picture pushing left to right for case 1, then PSTR would appear now at EBP plus 12. So, what happens, if I put the parameters left to right in a variable argument list kind of a case, the constant character string will have different addresses with different calls.

On the other hand if I push the parameters right to left then it is evident that my character string PSTR is always EBP plus 8 whatever happens. It is only the other variables that have variable addresses which is perfectly fine because depending on PSTR I know how to actually access those other variables . In conclusion you have no choice, but to push the parameters from right to left as we have discussed in our lectures here.

(Refer Slide Time: 24:15)

Topics Covered

- **Implementing printf()**
  - C implementation
- **Assembly implementation of**
  - va_start()
  - va_arg()