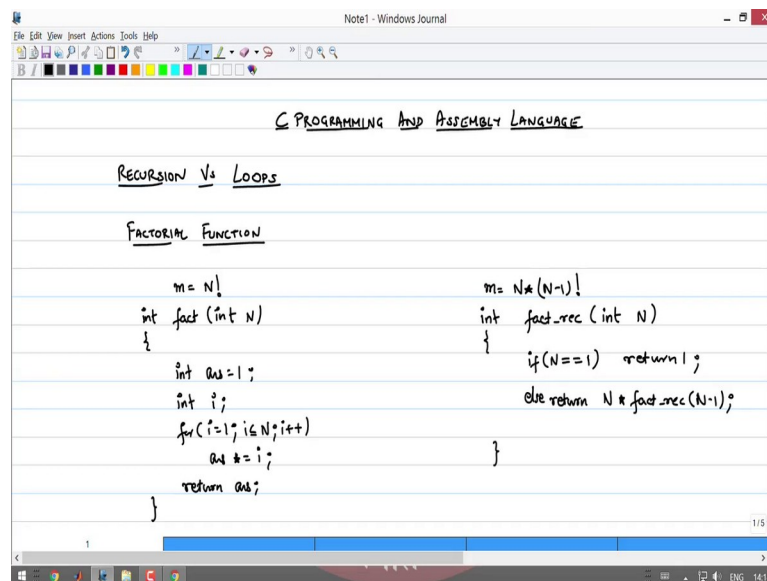


**C Programming and Assembly language**  
**Prof. Janakiraman Viraraghavan**  
**Department of Electrical Engineering**  
**Indian Institute of Technology, Madras**

**Lecture - 18**

Welcome back to this course on C Programming and Assembly language. We are in module 4 and in the last couple of lectures we looked at implementing a function like printf that handled one variable number of arguments and two variable kinds of data types as well as function parameters to which are passed to that function. So now, moving on with our analysis of because of the assembly translation that happens in the certain way that we have studied that till module 3, what is the impact on the high levels programming language .

(Refer Slide Time: 00:49)



The screenshot shows a Notepad window titled "Note1 - Windows Journal" with the following handwritten text:

C PROGRAMMING AND ASSEMBLY LANGUAGE

RECURSION Vs LOOPS

FACTORIAL FUNCTION

$m = N!$ <pre>int fact (int N) {     int ans = 1;     int i;     for (i=1; i&lt;=N; i++)         ans *= i;     return ans; }</pre>	$m = N * (N-1)!$ <pre>int fact_rec (int N) {     if (N==1) return 1;     else return N * fact_rec(N-1); }</pre>
---------------------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------

So, in that regard in this lecture we are going to discuss the following recursion versus loops. So, anyone who is worked with C programming or C plus plus till now would have use recursion at some point or the other. And, recursion is programming concept which is very used; is a very useful tool because, it just allows us to think of the problem in a very hierarchical manner. And, all you have to do is call the function same function again and again with fewer and fewer arguments as we go along right and the only thing is to ensure that you have one exit criterion so, that you do not go into an infinite loop .

So, let us look at simple example first of all of what recursion and is and how the same thing could be implemented in a loop.

So, let us look at implementing a factorial function. So, what is this let us say for example, I want to evaluate  $m$  equals  $N$  factorial . So, if I were to implement this using a loop then this is how it would happen `int fact of int N` and I would say `int answer equals to 1`. I create other loop variable `i` and simply say for `i equal to 1`, `i less than or equal to N` `i plus plus and answer into equals i` and then `return answer` .

On other hand if I able to implement the same concept using recursion then I would simply do the following, I would say `m is equal to N into N minus 1 factorial`. So, all I have to do is `int fact underscore recursion of int N` and what we are going to do is if or let us just go ahead and implement the factorial as it is first. We will just say that `return N into fact underscore recursion of N minus 1`; of course, the problem here is if you start with a positive integer then at some point you will hit 0.

And then you will hit minus 1 minus 2 and you will go on and on and on, there is no end to this recursion here. So therefore, you have to give us stopping criterion and that is very obvious if `N is equal to is equal to 1` , then we say `return 1 else return N into fact of N minus 1` . So, this is not a very complex program. So, let us go ahead and look at amongst these two implementations which one is bound to be faster especially for large numbers like large `N`. So, let us take the recursion example first of all .

(Refer Slide Time: 04:51)

```
Recursion:
int fact_rec(int N)
{
    if(N==1) return 1;
    else return N * fact_rec(N-1);
}

OBJECT CODE
PUSH EBP.
MOV EBP, ESP
SUB ESP, 0x40
MOV EAX, 1 // INIT EAX=1
MOV EBX, [EBP+8] // N=[EBP+8]
DEC EBX
JZ RETURN_1
PUSH EBX // PASSING (N-1)
CALL fact_rec
MOV ECX, [EBP+8] // ECX ← N
IMUL ECX // EAX ← N * fact(N)
RETURN_1: RET
```

And I am going to translate this to its assembly equivalent . So, I will say in fact underscore recursion of int N and I will say if N equal to 1 return 1 else return N into fact underscore recursion of N minus 1 . And, instead of doing an unoptimized assembly translation I am going to go ahead and implement this directly in assembly language . So, let us translate this to the assembly output right OBJECT CODE . So, first of all the bracket here would get translated to the prologue .

And what is the prologue? It is first PUSH EBP and then MOV my ESP whatever the ESP value is MOV that into EBP and then subtract ESP comma some constant , let us call it 0 x 4 0. Now, this is my prologue, then I go ahead and implement this particular functionality . How will this code get translated, remember I am not doing a line by line translation I am implementing this directly in assembly now . So, what do I have to do? I have to first take the value of N which let us say is contents of EBP plus 4 right or EBP plus 8.

So, what do I do? I am simply going to say MOV into EBX ok, contents of EBP plus 8 . And my why 8? Because, N is stored in EBP plus 8 , N is going to be contents of EBP plus 8. So, I am simply getting the value of N into my BX register then I am going to subtract or I will just decrement, I do not know N have to subtract I will decrement EBX. In the process if EBX were 1 and I decremented it and then it set the 0 flag and I will say jump on 0 to RETURN underscore 1, RETURN underscore 1 is a label.

So, I will put that somewhere down here , RETURN underscore 1 . What do I have to do in this place? All I have to do is to simply MOV EAX because I if I have to return the value 1 from the function I have to load EAX with the value 1 and simply say RETURN. So, why do not we go ahead and initialize our EAX here comma 1 . So, this is help INIT EAX equal to 1, basically I am saying the return value is initialized to 1 already. So, if I want to come here I am just going to say RET because, when I come here EAX has already has a value 1 INIT .

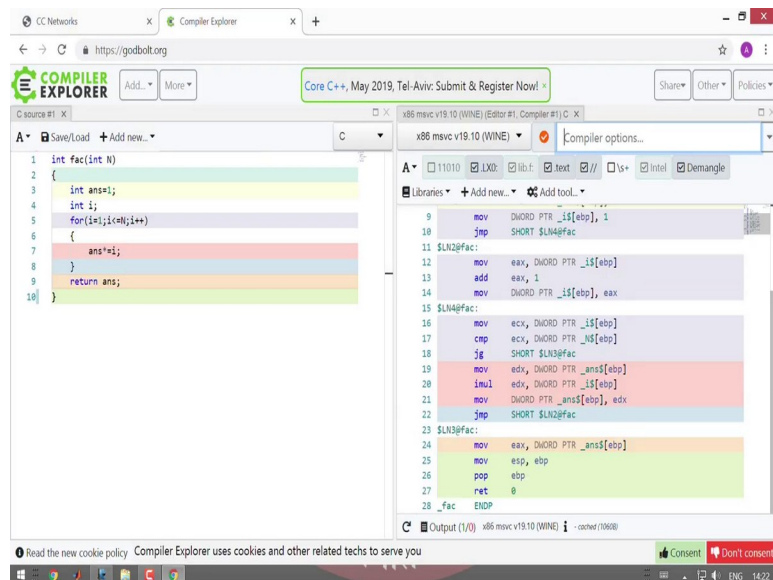
Now, suppose the operation of decrementing EBX by 1 did not result in 0 and let us assume that only a positive number is being passed to this function , we are not going to check for that condition here. Then it means that I need to implement the other part which is basically this portion here , now this portion will translate to a call as shown here . So, what is that I need to simply multiply the value of N with a call to the same

function with an argument  $N - 1$ . So therefore, what do I do, already my EBX should have the value  $N - 1$ .

So, if I just say `PUSH EBX` this should do because,  $N$  minus it already has the value of  $N - 1$  right. This is basically `PASSING N - 1` argument and then I will say `CALL fact underscore recursion`. So, let me do one thing, let me move this guy a little below here this `RETURN 1` I will move here and this label is called `RETURN underscore 1`. So, I am just going to call `fact underscore recursion` here and when I return from this function EAX should have the return value. And therefore, all I need to do is to multiply this EAX with the value of  $N$ .

So, what do I do? I simply will maybe yeah I will just `MOV ECX comma N` or I will say `EBP plus 8` and then I am going to say `i MUL of ECX`. So, what is this doing? It is essentially loading ECX is getting the value of  $N$  here and this step is simply saying EAX is  $N * \text{fact of } N - 1$ . So, this is the optimized assembly output of this particular recursion call.

(Refer Slide Time: 12:35)



The screenshot shows the Compiler Explorer interface. On the left, the C source code for a factorial function is displayed:

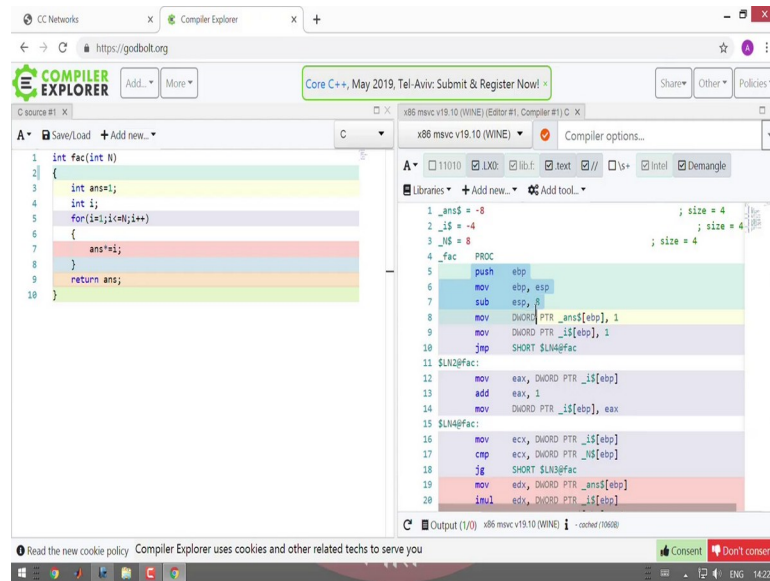
```
1 int fac(int N)
2 {
3     int ans=1;
4     int i;
5     for(i=1;i<=N;i++)
6     {
7         ans*=i;
8     }
9     return ans;
10 }
```

On the right, the assembly code for the same function is shown, with instructions corresponding to the C code:

```
9   mov     DWORD PTR _i$[ebp], 1
10  jmp     SHORT $LN4@fac
11 $LN2@fac:
12  mov     eax, DWORD PTR _i$[ebp]
13  add     eax, 1
14  mov     DWORD PTR _i$[ebp], eax
15 $LN4@fac:
16  mov     ecx, DWORD PTR _i$[ebp]
17  cmp     ecx, DWORD PTR _N$[ebp]
18  jg     SHORT $LN3@fac
19  mov     edx, DWORD PTR _ans$[ebp]
20  imul   edx, DWORD PTR _i$[ebp]
21  mov     DWORD PTR _ans$[ebp], edx
22  jmp     SHORT $LN2@fac
23 $LN3@fac:
24  mov     eax, DWORD PTR _ans$[ebp]
25  mov     esp, ebp
26  pop     ebp
27  ret     0
28 _fac   ENDP
```

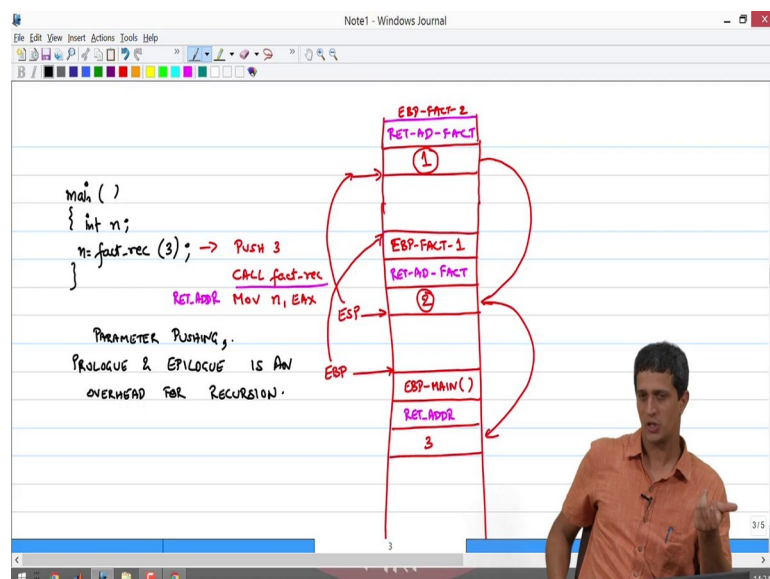
Now, on the other hand if you go ahead and look at the implementation of the factorial function for loop right, it is the C implementation is shown on the left right what I wrote earlier. And, if you look at the assembly translation it is not very different.

(Refer Slide Time: 12:53)



So, this is again you know the prologue, then this is just initializing some variables and then I have a couple of MOV's and I am doing a jump and I am doing a multiply and so on. So, if you look at the assembly output of this loop implementation, it is not very different from what we had in our recursion call. You have a compare, you have a jump and then you are, you know doing some multiplications and so on. However, the difference lies in the number of instructions that get executed eventually and this will become evident when we look at the stack more closely. So, let us look at the picture of the stack when I go ahead and call this function in the main.

(Refer Slide Time: 13:43)



I am going to say fact underscore recursion of let me call 3. So, if I look at the picture of the stack here, just look at the number of operations that are involved when we are implementing a function using recursion. So, if I want to call fact underscore rec of 3 right, this will simply translate to push 3 CALL fact underscore rec of 3 . No there is no of 3, it is just calling the function and after I returned here let us say I have a variable N equal to . So, I will say int n, then I will simply have to out here after I return from the function I will MOV into N the value of EAX, this is the assembly translation .

So, if you look at this the value 3 gets push on to stack and then you are calling fact underscore recursion and this is my return address, let me assign a variable a label here. So, this return address will get pushed onto stack when you implement the call fact underscore recursion when you call this function, this particular return address gets pushed onto stack . And, then you go into the implementation of the fact underscore recursion, as we saw there is a prologue that gets executed . So, the EBP of main will get pushed onto stack. Then what am I going to do? I want to move my EBP move my EBP to where ESP was .

So, my EBP will point here now , then I am going to subtract some local variable space, this will be my ESP . This is the first call of fact underscore recursion, now you check if the value of N is 1 well it isn't. So, then I am going to call the function again . So, and I am going to call the function with the argument 2 this time . So, I am going to push on to stack the argument 2 and when you come here you will have the return address of factorial function that is defined there . So, if you look at this guy its essentially what we are doing is we are implementing this particular call now .

(Refer Slide Time: 17:13)

```

}
if(N==1) return 1;
else return N * fact_rec(N-1);
}

PUSH EBP;
MOV EBP, ESP;
SUB ESP, 0x40;
MOV EAX, 1; // INIT EAX=1
MOV EBX, [EBP+8] // N=[EBP+8]
DEC EBX
JZ RETURN_1
PUSH EBX // PASSING (N-1)
CALL fact_rec;
RET-AD-FACT: MOV ECX, [EBP+8] // ECX←N
MUL ECX // EAX←N*fact(N)
RETURN_1:
EPILOGUE → {ADD ESP, 0x40}
RET {POP EBP}

```

So, this will be my return address fact, this is the address it has to return to after finishing the call . So, that will get pushed onto stack here , then again my EBP of FACT underscore recursion for the first call will get pushed onto stack EBP FACT I will just say first call , that will get pushed on to stack . And, then you are going to move my EBP into this particular point here right and then ESP will again get move offset here somewhere. Now, again you come into this function and then you check if the value of N that has been passed to this function.

Now, what is the value of N? It is this value 2, you check if that is 1 well that is not 1 and therefore, you again decrement this value and then call the recursive function again with the value 1 this time. So, what do you do? You again push on to stack so, this is my thing, I am going to push on to stack the value 1 and again I am going to call it with the particular return address. And, then I am now going to again move my push on stack this EBP FACT 2 . And, then I am going to repeat this process again and again of course, this time the argument that has been passed to this factorial function is 1.

And therefore, I will simply do a return of 1 which means EAX will get loaded with 1 and then I will execute the epilogue. By the way in the previous case here , if you look at this page here what I am not shown is the implementation of the epilogue. What is the epilogue? It is basically simply going to undo these operations before I return. So, out here I have to execute the epilogue and then do a return without undoing the prologue I

cannot get out of the function. And therefore, the label RETURN underscore 1 has to point to the epilogue. And what is the epilogue? It is nothing, but ADD ESP comma 0 x 4 0 and then POP EBP and then the return instruction will happen accordingly .

So now, you come here and then you see that 1 the argument N is equal to 1 and therefore, I just need to do a return of 1. So, you will move into EAX the value of 1 and then you will execute the epilogue. And, then come back to this particular stack here right, you will return from here, come here then return from there come here. So, what we say is ultimately the operation that has to be performed is a very simple multiplication . But, in order to do this single instruction of iMUL with another register, what we are in turn doing is executing the prologue and epilogue for every call of this function . So, conclusion is the prologue and epilogue is an overhead for recursion.

So, on the other hand if you look at the loop implementation, it is actually very straight forward because we are doing just what is necessary. In the sense we are multiplying an accumulating this product over the loop and we are not having to call any function in this process . So therefore, there is no prologue epilogue overhead when we implement the same function in a loop . So, and remember the prologue and epilogue is not a very simple operation because, all of these are memory operations. They involve the stack , you have to push the EBP on to stack then you have to subtract the ESP with some value which is not too bad .

Then you are during the call you have to push the return address on to stack, then you have to pass parameters again ; each parameter that I am passing has to be done through a particular push. so, effectively apart from the prologue and the epilogue there is also a parameter pushing . So, prologue and epilogue I will also add parameter pushing, prologue yeah parameter pushing, prologue and epilogue is an overhead for recursion which is not in there in loops. And therefore, it is a common practice to take a program that has been implemented using recursion and convert it to a loop, if performance is extremely critical .

So, these are some implications that we have at an assembly level and we are able to appreciate them very well because, we understand exactly what happens at the lowest level in machine language . So, in the next lecture we will look at more such examples of



speeding up programs or performance analysis of certain assembly implementations for certain special functions.

(Refer Slide Time: 24:33)

### Topics Covered

- **Recursion vs Loops**
- **Factorial implementation**
  - Analysis of a recursion implementation
  - Comparison with a loop implementation