

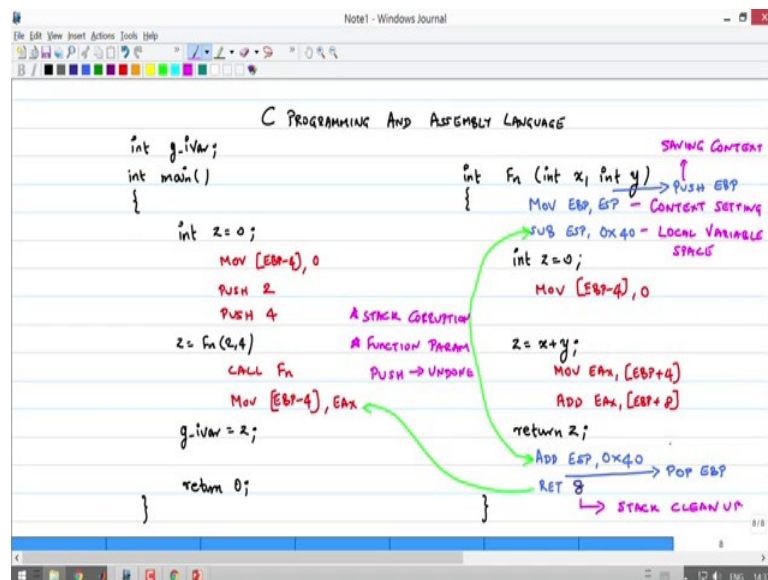
C Programming and Assembly language
Prof. Janakiraman Viraraghavan
Department of Electrical Engineering
Indian Institute of Technology, Madras

Lecture - 14

Welcome back to this course on C Programming and Assembly Language we are in module 3. Where we are discussing how we can translate a given C program into its assembly output. So, by and we are trying to compile the code here, which means we are trying to do task 2 that we alluded to in an earlier discussion.

So, last lecture we stopped at this point, where we translated the C program into its first order assembly output, which means we just translated all the ALU operations into their corresponding unoptimized compiled output. And, we just saw what happens if the program were executed as is and what instructions needed to be added. And, we saw that there were two problems that we encountered one is stack corruption and the other is returning to a wrong address from the main.

(Refer Slide Time: 01:09)



So, let us just quickly refresh ourselves on what we had discussed there. So, the function Fn which basically took two parameters x and y got translated to the following assembly output, where EBP minus 4 is the local variable z in the function. Note that that same local variable, same address EBP minus 4 is used even to access the local variable z in

main . Of course, the fact that both have the same name is not related to this discussion right, but all local variables are accessed in exactly the same way . And, you know z equal to x plus y translates to some simple MOV and ADD operations. So, the instructions in red are the unoptimized assembly output, which is a very obvious thing that from what we have discussed till now.

The instructions shown in blue here are the ones that we had to ADD in order to do certain things . For example, the first instruction `MOV EBP comma ESP` was to basically MOV the base pointer to this stack pointer . So, why was this done? This was essentially done to ensure that the context of the called function is being set correctly , this is context setting . So, the EBP which was actually pointing to the local variable space of main before this is now being made to point to the stack ESP, which is basically the new local variable space of the function Fn .

And, next we had to add another instruction `SUB ESP comma 0 x 4 0` and this was mainly to allocate local variable space. So, that any stack operation like PUSH or POP does not overwrite the local variables that we have been using in the function Fn . So, then we went ahead and executed the body of this function, which are basically the instructions in red. And, before we exited this function we had to undo all these operations right especially what we had done on the stack pointer. Otherwise, we would return to a wrong value when the RET instruction is now called.

So, therefore, the `SUB ESP minus 0 x 4 0` had to be undone so, if you see this instruction is basically a pair. So, whatever you do with ESP in that instruction you have to undo here. Now, if you had more local variables, then the compiler would have basically done `SUB ESP minus some larger value`, and out here before you return it would `ADD ESP comma the exactly the same value` . So, this occurs in pairs and this is obvious because any operation that you do on the stack pointer has to be undone before you exit that function, otherwise you will be in trouble. And, after that we simply did a return .

And, in spite of this we had two problems one was stack corruption . So, after this return the our instruction pointer would simply go back all the way to this point . After the return it would go back to this `MOV EBP minus 4` you know contents of comma EA x, because that is the instruction after call Fn.

When we return from the function Fn to main unfortunately we have not been able to reset the context of the local variable space back to that of main. So, before we went into Fn we were able to set the context to the new function by doing a MOV EBP comma ESP. Unfortunately now we cannot reset that and, why is that because before we went into this function Fn, we did not care to save the value of EBP that it had originally. So, therefore, before returning from Fn there is no way that we can restore this value .

(Refer Slide Time: 05:55)

The slide displays the following assembly code and stack diagram:

```

CODE SEGMENT - Function - main()
.
    int z = 0;
C100 MOV [EBP-4], 0
    z = Fn(2,4);
C101 PUSH 0x00000004
C102 PUSH 0x00000002
C103 Call C200
C104 MOV [EBP-4], EAX
    g_iVar = z;
C105 MOV [g_iVar], EAX
C106 RET
    
```

The stack diagram shows the following state:

- EBP points to address C104, which contains 0x00000006.
- ESP points to address 0x00000002, which contains 0x00000002.
- Address 0x00000004 contains 0x00000004.
- Address 0x00000000 contains 0x00000000 and is labeled 'Local var Z'.


Then, we looked at an other problem where we went ahead and try to execute the function main . And, the problem is the stack pointer ESP is now pointing to the 2 0 x 0 0 0 2 and what is that, that is essentially a value that we pushed on to stack . So, as I mentioned earlier any operation that you do with the stack pointer has to be undone before you come out of that function and that is the exact problem that we are facing even now .

So, what are the problems? The problems essentially are stack corruption and the other problem is the function parameter PUSH it has to be undone. So, this is what we want to achieve and we need to add some more instructions out here in the blue region. So, that we are able to overcome this problem. And, now we will look at what those instructions should be .

(Refer Slide Time: 07:21)

Compile the C Program

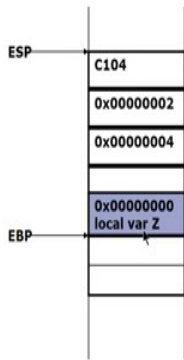
Contd..



CODE SEGMENT - Function - main()

```
int z = 0;
C100 MOV [EBP-4], 0
      z = Fn(2,4);
C101 PUSH 0x00000004
C102 PUSH 0x00000002
C103 Call C200
C104 MOV [EBP-4], EAX
      g_iVar = z;
C105 MOV [g_iVar], EAX
```

STACK SEGMENT



| | |
|-----|---------------------------|
| ESP | C104 |
| | 0x00000002 |
| | 0x00000004 |
| | |
| EBP | 0x00000000 local var Z |
| | |
| | |


So, let us again go through the execution of this function main . And, as usual the stack pointer then the base pointer are pointing at some arbitrary values that we do not know right now. And, we do MOV EBP minus 4 so, assuming that the stack pointer and base pointer have been initialized correctly. It is going to point to the local variable space of main.

So, therefore, the local variable Z in main gets set to 0. Now, you PUSH 4 you PUSH 2 these are there is no difference from what we did earlier, and then we call the function which is at the address C 200. So, the return address C 104 is pushed on to stack and you MOV into the function Fn.

(Refer Slide Time: 08:07)

Compile the C Program

Contd..




CODE SEGMENT - Function - Fn()

```
C200 PUSH EBP
C202 MOV EBP, ESP
C203 SUB ESP, 0x40
      int z=0;
C204 MOV [EBP-4], 0
      z = x + y
C205 MOV EAX, [EBP+8]
C206 ADD EAX, [EBP+12]
C207 MOV [EBP-4], EAX
      return z;
C208 ADD ESP, 0x40
C209 POP EBP
C20A RET 8 ←
```

STACK SEGMENT

| | |
|-----|---------------------------|
| | Local variable space |
| Z | 0x00000006 |
| | EBP - main() |
| ESP | C104 |
| | 0x00000002 |
| | 0x00000004 |
| | |
| EBP | 0x00000000 local var Z |



So, now before we proceed with anything else in this function what we need to do is to simply save the EBP value somewhere. So, that we can restore the context of main before we return from this function Fn. So, where do you store it of course, you cannot use a register as usual, it has to be stored somewhere turns out that the stack is the best place again to store this.

So, before you do anything you PUSH the current value of EBP onto stack. So, what gets pushed onto stack is actually the EBP of main now . Now, I do the same old operations that we had discussed earlier MOV EBP comma ESP which is context setting, SUB ESP comma 0 x 4 0 which is allocating local variable space, then the function body executes. Note now that because I have done an extra PUSH of EBP in the first instruction, the local the function parameters x and y have now got to be accessed as contents of EBP plus 8 because it is 4 bytes here 4 bytes here and then EBP plus 8 and EBP plus 12.

Earlier it was EBP plus 4 and plus EBP plus 8. So, now, we need to add this change this addressing a little bit . Now, you continue executing the body of the function. Now, before you return from the function you have to undo all that you did. So, you ADD ESP comma 0 x 4 0 which is the local variable space, POP the top of stack right, which is basically storing the EBP of main back to the EBP register.

So, this instruction POP EBP now ensures that my EBP has moved back to the calling functions local variable space, which is the; that of main in our case here. So, that solves one of our problems which is the stack corruption problem.

Now, we still need to undo the function parameter pushing that we did, because now when you continue from this function and go back to main, ESP will still point to the wrong value. That is where the RET N instruction comes to our rescue. If you remember I had pointed out that RET N not only pops the top of stack into the instruction pointer to resume, you know execution from where it left off, but it also adds that N to the stack pointer.

So, the purpose of this is exactly for undoing the function parameter pushing onto stack. So, note that just because we pushed two variables on to stack before coming into the function. It does not mean that we have to only POP these two values out to undo this operation; we can also simply add something to the stack pointer ESP and undo this operation. So, what you do is, you do a RET 8.

(Refer Slide Time: 11:23)

CODE SEGMENT - Function - main()

```
int z = 0;
C100 MOV [EBP-4], 0
z = Fn(2,4);
C101 PUSH 0x00000004
C102 PUSH 0x00000002
C103 Call C200
C104 MOV [EBP-4], EAX
g_iVar = z;
C105 MOV [g_iVar], EAX
C106 Epilogue ←
```

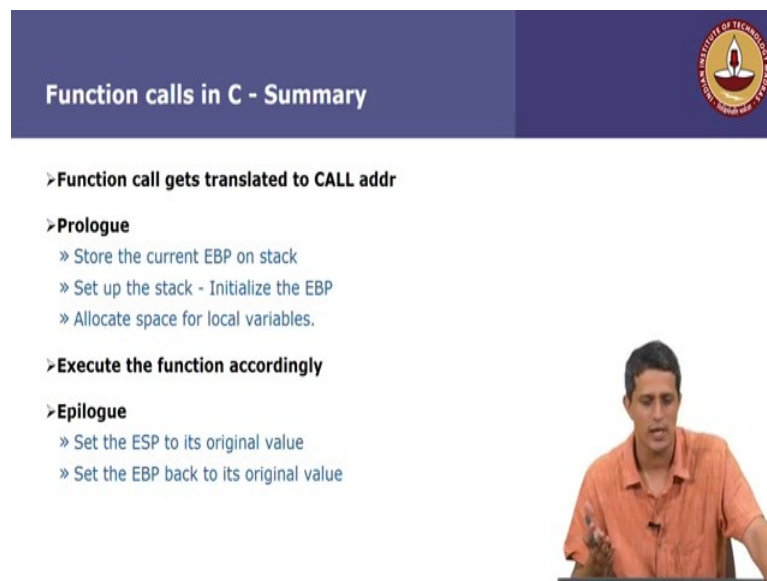
STACK SEGMENT

| |
|------------|
| |
| |
| 0x00000006 |
| C104 |
| 0x00000002 |
| 0x00000004 |
| ESP |
| 0x00000006 |
| EBP |

And, what does RET 8 do, it does two things; one is pops C 104 and ensures that the instruction pointer has been loaded with the return address, which is C 104, and in the same instruction also adds the value 8 to ESP.

Why 8, because I passed 2 4 byte integers to this function. If I had instead passed 3 4 byte integers to this function, I would have done RET 12. Now, like the earlier example if we had three integers and one character, then I would have done RET 13 the ESP would have got added with 13. Now, I go ahead and continue this particular execute body of the function , the body of main continue executing that and I can exit from my function gracefully.

(Refer Slide Time: 12:25)



The slide is titled "Function calls in C - Summary" and features a logo of the Indian Institute of Technology Bombay in the top right corner. The content is organized into a list of steps:

- **Function call gets translated to CALL addr**
- **Prologue**
 - » Store the current EBP on stack
 - » Set up the stack - Initialize the EBP
 - » Allocate space for local variables.
- **Execute the function accordingly**
- **Epilogue**
 - » Set the ESP to its original value
 - » Set the EBP back to its original value

A video inset in the bottom right corner shows a man in an orange shirt speaking.

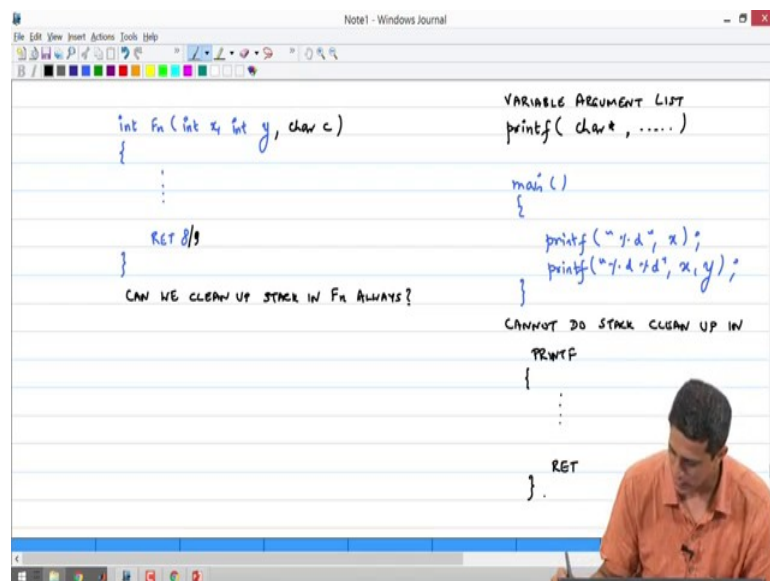
So, in summary the function gets called translated to, you know a function call gets translated to a call address . This is something that we already knew from microprocessor programming, but what we did not know is that we had to do a prologue and an epilogue. What is a prologue? It is exactly those three instructions that we added to store the current EBP on stack, which is storing the current context, set up setting up the stack or initializing the EBP , or this is setting up the context of the function Fn and allocating local variable space.

These are the three tasks that happen in the prologue of a function. Then you execute the function according to its whatever the instructions are and before you exit that function you have to undo this operations of prologue and that is known as an epilogue . So, what is the epilogue it is basically setting the ESP to it is original value and setting the EBP back to it is original value before exiting that function . So, how did we fix these particular problems we went ahead and fixed it in the following way.

So, before we let me add that instruction here. Before we did anything in the function Fn we said we will PUSH EBP onto stack . And, before we return from this function we were going to simply POP the value back onto stack . So, these were the two corrections that we needed in order to execute our function Fn in a consistent manner and go back to main without causing any context switching problems .

And, of course, to handle the function parameter pushing on to stack we had to do this particular operation let me write that in a different color . So, what did this do this is stack cleanup . And, PUSH EBP is saving the context , here is a comment here this is saving context . The next instruction move EBP comma ESP is setting the context for the function Fn the PUSH EBP on to stack is saving the current context of the calling function onto stack . So, now, that brings us to a very interesting question which is in this function Fn I had two parameters int x comma int y and, therefore, I did RET 8 .

(Refer Slide Time: 16:23)



So, let us look at this function in particular int y I did the usual things and let us just focus on the RET instruction here . I am going to do RET 8. Now, suppose here instead I had int x comma y comma char c then I would do 9. Now, if I had another integer then I would do return 13 and so on .

So, the question is can I always do the stack cleanup in my function Fn itself . So, that is the question that we want to answer clean up stack in Fn always. So, what other kind of function are we used to. For example, let us look at this function called printf . The

signature of this function is char star comma dot dot dot. So, what is this dot dot dot mean, it means it is a variable argument list function. So, this is variable argument list.

So, for example, if I wanted to call this you know in my main, then I could call it in the following way printf some string percent d comma some integer x or I could call it as percent d percent d comma x comma y . So, depending on what my character string is and how many persons I have in that I need to pass as many parameters to this function.

And, therefore, if you look at the implementation of printf evidently it is not possible for me to do the stack cleanup by doing RET 8 or 9 or 13 or whatever, because the function printf does not know how many parameters it is going to receive up front .

So, you cannot do stack clean up in printf . So, if you look at the function printf and you look at the implementation, it will have whatever the body is, but the return instruction will simply be RET ok. This because it does not know how many parameters is going to receive up front. However, what the compiler does know is how many parameters the calling function is going to pass to printf .

For example, in this main I have two instances of printf , I am calling printf in two cases; one is I am passing only x as an argument for the percent d, in the second case I am passing x and y to 2 percent d's . So, therefore, these two calls will get translated slightly differently. So, let us look at how this thing would happen? Right.

(Refer Slide Time: 21:07)

The screenshot shows a Windows Journal window with handwritten assembly code for two printf calls. The first call is `printf("%d", x);` with assembly instructions: `PUSH x`, `PUSH PTR`, `CALL printf`, and `ADD ESP, 8`. The second call is `printf("%d %d", x, y);` with assembly instructions: `PUSH y`, `PUSH x`, `PUSH PTR`, `CALL printf`, and `ADD ESP, 12`. A man in an orange shirt is visible in the bottom right corner of the screenshot.

So, I have the first printf percent d comma x. How will this get translated to assembly, essentially you will do PUSH whatever x is then you will do PUSH whatever this constant string is right, this is a constant string. So, you will PUSH PTR . So, somewhere in memory PTR is a constant string is this particular string and then you do call printf. Now, when you return from this function what you know is you have pushed 4 bytes an integer and you have pushed a character pointer , which also is 4 bytes. And, therefore, out here you do what is known as ADD ESP comma 8.

Now, on the other hand the same printf when it is called as percent d percent d x comma y. Will get translated as PUSH y, PUSH x, PUSH another constant string let me call it as PTR 2 is going to be percent d percent d in memory PTR 2 . And, call printf and when you return from printf now you got to do ADD ESP comma 12.

So, the difference between our earlier implementation where we did RET N and doing this add ESP comma 8 or 12 explicitly is in principle the same thing , we are splitting the job of returning and adding a constant to the ESP in two instructions in this case . So, here we are pushing the onus of doing the stack clean up on the calling function , because it is only the calling function that knows exactly how many parameters are being passed to that particular function .


So, this calling convention and stack cleanup are sort of related and you know you have a particular calling convention as they call it in a windows programming , which is underscore underscore cdecl .

(Refer Slide Time: 24:31)

Calling conventions

- `__cdecl`
 - » Default calling convention of C functions
 - » Needed for variable argument list
 - » Caller cleans the stack - ADD ESP, N instruction
- `__stdcall`
 - » Faster than the `__cdecl` call.
 - » Callee cleans the stack - RET N instruction

Contd.....



Where; what this means is that, it is by default this is the calling convention in C programs by the way . In C functions it is needed for variable argument list like printf and the caller calling function is going to clean the stack by doing add ESP comma N instruction. Underscore stdcall is of course, it is faster I will tell you for what reason, where the callee or the function itself does it is own stack clean up . And, this can be done only when the arguments are fixed and known at compile time itself .

Of course the stdcall is a little faster than the cdecl call because the RET N instruction does two things in one shot, popping the top of stack into the instruction pointer and adding ESP comma N. Whereas, in cdecl these two things happen separately and therefore, it is marginally slower. Of course, when we go into assembly level programming some of these marginal differences also can make a significant impact our programming time. And therefore, it is important to know this and pick the right one appropriately.

Yeah. So, in the next lecture we will look at what happens when we look at a C plus plus program at assembly language level.

(Refer Slide Time: 26:07)

Topics Covered

- **Setting a function context**
 - Prologue
- **Resetting the function context**
 - Epilogue
- **Stack cleanup**
- **Calling conventions**
 - `__cdecl`
 - `__stdcall`