

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

Lecture - 12

(Refer Slide Time: 00:15)

C PROGRAMMING & ASSEMBLY LANGUAGE

```

int givar = 5;
void main()
{
    int z = 0;
    z = fn(2, 4);
    givar = z;
}
F1.C
    
```

DESIRED FEATURES

- * LOCAL VARIABLES
- * SCOPE OF LOCAL VAR
- * PASS AS MANY PARAMETERS AS I WANT

```

int fn(int x, int y)
{
    int z = 0;
    z = x + y;
    return z;
}
F2.C
    
```

FUNCTION PARAM PASSING

CALL FN_ADDR ~~MOV EBX, 4~~ PUSH 4 ~~MOV ECX, 4~~ PUSH 2


← FUNCTION PARAMETERS
 Mov x, EBX
 Mov y, ECX

MOV Z, 0
 z = x + y;
 MOV EBX, x
 ADD ECX, y
 MOV Z, EBX
 return z;
 RET

Welcome back to this course on C Programming and Assembly language. So, we are in module 3 and we now are ready to get into the heart of this course, which is our ability to compile a given C Program into assembly language right.


(Refer Slide Time: 00:29)

Compile the C program!!




```
int g_iVar = 5;
void main()
{
    int z=0;
    z = Fn(2,4);
    g_iVar = z;
}

int Fn(int x, int y)
{
    int z=0;
    z = x+ y
    return z;
}
```



So, let us start with the simple example the task that is given to us is to compile the following C program into the respective object code or the assembly output . So, when I say compile I mean the task 2 that I discussed last time , which is to just generate the assembly output for this particular file without worrying about if what the other files contain and so on .

So, what do I have here I have int g underscore iVar which is a global variable the g underscore indicates that it is a global variable. Then, I have the void main I need a main for any C executable which basically has a local variable int z equal to 0, then z is basically the return value of function Fn and I am passing 2 parameters to it 2 comma 4 . And, then I am assigning the global variable g underscore I Var to z here and I am done with that function. What is the function Fn do? It is a function that takes 2 integers x and y as parameters, returns an integer and that does simply does the addition of the 2 parameters and returns the value z .

So, there are many interesting things that we first of all note here. And, by the end of this lecture, what we are going to do we are going to put down a desirable set of features that we need. And, we have to ensure that these features are retain when we translate these to assembly language . So, we need to pick up the right assembly translation that enables this feature . So, the first thing we note is both the main and the function F n have this

variable called `z`; so, which means that the variables `z` can be used in the same function, in different functions with the same name.

This is particularly useful because imagine if this is in file `F 1 dot C` and this is in file `F 2 dot C`. Remember that when we compile we do not even know what the content of `F 2 dot C` is. So, to keep track of whether these variable `z` are used even in that file is impossible. So, therefore, we need a way in which variable names can be local to a function. So, `global Var` for example, is not local it is a global variable; that means, across all the files I have to ensure that, I do not have a repetition of that name. And therefore, the concept of local variable is very good.

These are my desired features. So, the next thing is that unlike the `Malloc` and stuff like that. The developer really does not worry about where you know this variable `z` is stored or how much memory has to be allocated, whether it is going to crash with another location and so on. So, all the developer is worried about is that when you enter the function `F n` the local variables `z` in that function is alive. And, when you are done with that particular function when you return from that function the local variable `z` of that function is gone. That is the key idea here only then I can use the same variable across 2 places without worrying about any clashes.

So, the which means that the scope of variables of local variables is also very useful. And, by scope I mean that this local variable `z` here is available only when this function `Fn` is active and is being executed. So, the moment the instruction pointer returns back to the function `main`, that variable `z` should be gone. And, it should not be accessible anymore, that is the idea and that is the desired feature which is known as scope of a local variable. Now, you have you know the other things that you see are being used here are something known as function parameters, these parameters.

So, we need to make sure that our assembly code generation takes care of all these aspects very well. So, let us start with a very naive and an unoptimized assembly output. So, we already know you know what happens. For example, when we when we say that `int z equal to 0`, we know that this translates to `MOV z comma 0`. We will come to where you know what `z` is, where `z` should be stored and all that either in this lecture or at most by the end of the next lecture.

Now, the next thing that I am doing is I am calling a function and passing 2 parameters to it . So, the function F n of 2 comma 4 is simply going to result in a assembly instruction called call and that is going to be a F n underscore address. Now, what this exact address is and where this code for the function F n goes and sits will be determined only by the linker at linking time . So, we are going to assign some label to this address called F n address for now we will not worry about where this exact address is at this point of time t.

But, before I call this function remember that when I call this function the programmer execution control goes into that particular function . Now, I need to pass 2 function parameters to it right. So, there are 2 parameters are 2 and 4 ok. So, what I need to do is I need to put some assembly code that allows me to pass function parameters . So, this you know how we pass these parameters and how they are used in the function are closely tied with each other. So, let us look at a both of them together.

Now, let us look at the compilation of this F 2 dot C, the function Fn is sitting in an another filed called F 2 dot c. So, this MOV in z comma 0 , e is equal to 0 is again MOV z comma 0. So, note that the 2 instructions are exactly the same , these instructions are exactly the same between my 2 functions main and Fn .

Now, I want to do z is equal to x plus y , which means that I am going to MOV into let us say EAX comma x I MOV x into EAX, then add EAX comma y, low case y and MOV Z comma EAX . When I do a return of z, what I really need to do is simply call the RET, note that all the red mnemonics that I am writing are actual assembly instructions. All, I need to now do is to decide what I have to put here? You know what instructions I have to put to pass parameters from one function to another.

Now, because I am passing only 2 parameters, it may be natural for us to think. Maybe I could simply pass these two parameters through registers . For example, this instruction could simply be MOV EBX comma 2 MOV ECX comma 4 . And, then I am going to go into that particular function Fn, where I need to access x and y . So, now, it is possible for example, here where you know before I start this function, if I simply say MOV x comma EBX or MOV y comma ECX.

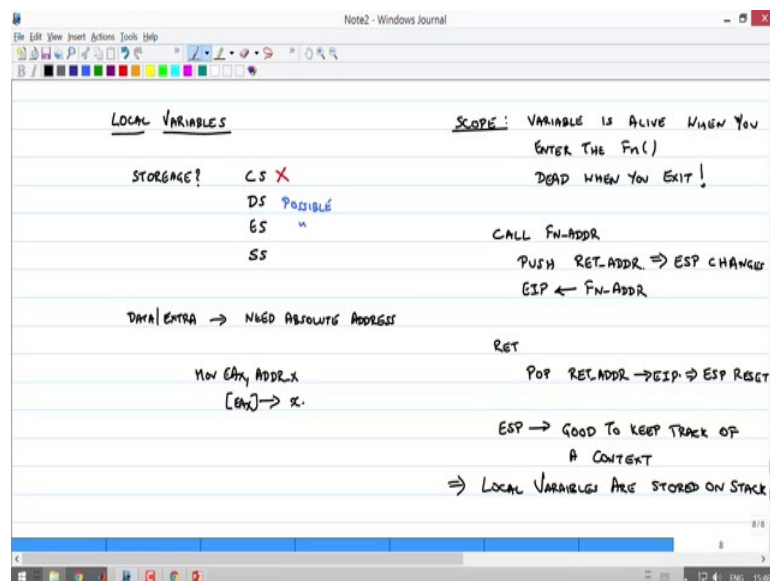
Then it is quite evident that the function parameters 2 and 4 have indeed got passed to this particular function it is possible . So, what I am saying is that before you enter and

execute anything in that function Fn, use store these values of EBX and ECX into your variables x and y . Maybe effectively become like some local variable right wherever they are going to get stored and you are free up the registers EBX and ECX, this is one legitimate way of doing it.

However, the problem very clearly is that I cannot assume that I am going to pass only 2 parameters or 4 parameters or only as many parameters as there are registers that are available in the microprocessor . So, an other desirable feature is I should be able to pass as many parameters as I want into a function .

So, the assembly translation that we do should allow us to pass as many parameters as we want into a function . So, therefore, this straightaway rules out depending on only the registers. So, the question is what else can I do? So, let us go ahead and look at what we do with local variables first and then we will see that function parameters are not any different .

(Refer Slide Time: 13:21)



So let us go ahead and look at local variables . So, local variables essentially you know we have to ensure that, the scope is maintained where in the moment you enter a function that local variable is alive and the moment you exit that function the local variable is gone . So, the question is where do I store local variables?.

So, there are not many places that you can store these variables, there are only 4 segments that we have right. So, we have the code segment, the data segment, the extra segment, , and stack segment. So, if I try to store my local variables in the code segment it really does not make any sense, because if your instruction pointer happens to come to the location then it will get interpreted as an instruction.

So, therefore, straight away I do not think it is advisable to store anything in the code segment which is related to data. So, one possibility is to store the local variables either in data segment or extra segment . So, this is a possibility right. Now, let us look at the requirement that we had that we discussed previously, where we said that we wanted we like the idea of a local variable . And, we like the concept of a scope. Now, the problem is if you put something into the data segment or the extra segment, then the only way you can access these is by an absolute address .

So, if you want to access something in the data segment, data slash extra segment absolute address . So, basically I am going to say that you know I have to MOV an address MOV into some MOV EAX comma address of x . And, then I could say that the contents of EAX right refers to the local variable x. Similarly, I could have another local variable ADDR you know, another address ADDR underscore y and that could you know refer to the local variable y.

So, now again the problem is that when I have to do this, I cannot use these registers EAX EBX and all that stuff to keep track of these addresses . Because, I need to use EAX, EBX, ECX all the general purpose registers for my ALU instruction. So, that is again ruled out. So, then the question is where do I put these local variables. So, it turns out that the stack is a good place to do it, but let us look at an intuition or an intuitive reason of why the stack is the right place to put these local variables in .

So, we want to have the concept of scope . What is the definition of scope? The variable is alive when you enter the function . And, dead when you exit or it is no longer accessible the moment you exit that function Fn . So, now, let us look at you know what is it that happens, when I enter a function. How do you enter a function you enter a function by doing what is known as a call of FN underscore ADDR .

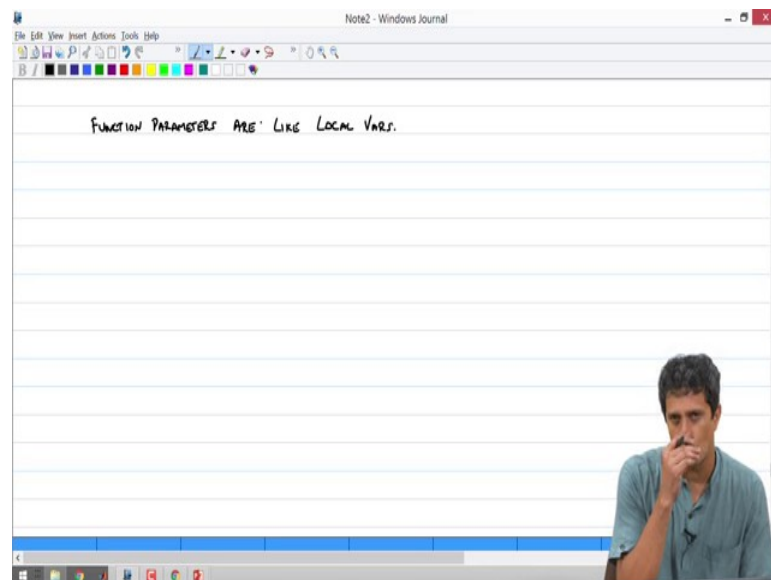
So, what is this instruction do it basically loads the of course, it first pushes the return address onto stack, then it loads the EIP with the function underscore ADDR, which

means control is transferred to that particular location for execution. So, when my EIP moves to that Fn address and starts executing, I want my local variable of that function to become active . So, the question now is I need something to set a context for my new function, when I enter and the context to be reset when I exit that particular function .

So, how do you exit that particular function you do a RET and what does RET do it does a pop of the return address into to the EIP right, it is going to simply pop the return address into EIP. So, what we notice here is implicitly when you call a function through the call instruction the stack pointer ESP changes , this implies ESP and when you return ESP is reset. So, every time a function is called a push and pop operation happens implicitly altering the stack pointer ESP.

So, therefore, there is a certain context that is set for us when we enter a function in the ESP and the contacts gets reset when we leave that function through the return instruction. Therefore, ESP is a good to keep track of a context and this is what we are going to exploit in order to store local variables. And, this is the reason that variables are stored on stack.

(Refer Slide Time: 22:27)



So, now it turns out that function parameters are also not very different from local variables . So, therefore, even function parameters are like local variables. And, therefore, these function parameters are also going to be stored on the stack before you call a particular function. Therefore, if you go back to the example that we discussed

earlier instead of moving data on 2 registers before you call that function, what you would do is to simply push 4 push 2 onto stack .

Because, now I can do any number of pushes which means that I can pass any number of parameters this way to a particular function. So, in the next lecture we will look at how you actually push these parameters onto stack before you call a function. And, how exactly we are going to access local variables from the stack and what instructions are necessary in order to consistently go from one function to the other and come back to the original calling function without causing any disruption.

(Refer Slide Time: 24:10)

Topics Covered

- **Desirable features**
 - Local variables
 - Function scope
 - Passing as many parameters
- **Storing local variables**