

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

Lecture - 13

Welcome back to this course on C Programming in an Assembly Language we are in module 3.

(Refer Slide Time: 00:21)

C PROGRAMMING & ASSEMBLY LANGUAGE

```

int givar = 5;
void main ()
{
    int z = 0;
    MOV Z, 0
    (FUNCTION PARAM PASSING)
    z = Fn(2, 4);
    CALL FN-ADDR
    givar = z;
}
    
```

DESIRED FEATURES

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

```

int Fn(int x, int y) ← FUNCTION PARAMETERS
{
    int z = 0;
    MOV Z, 0
    z = x + y;
    MOV EAX, x
    ADD EAX, y
    MOV Z, EAX
    return z;
    RET
}
    
```

(FUNCTION PARAM PASSING)

```

CALL FN-ADDR
PUSH 4
PUSH 2
MOV EAX, 4
    
```

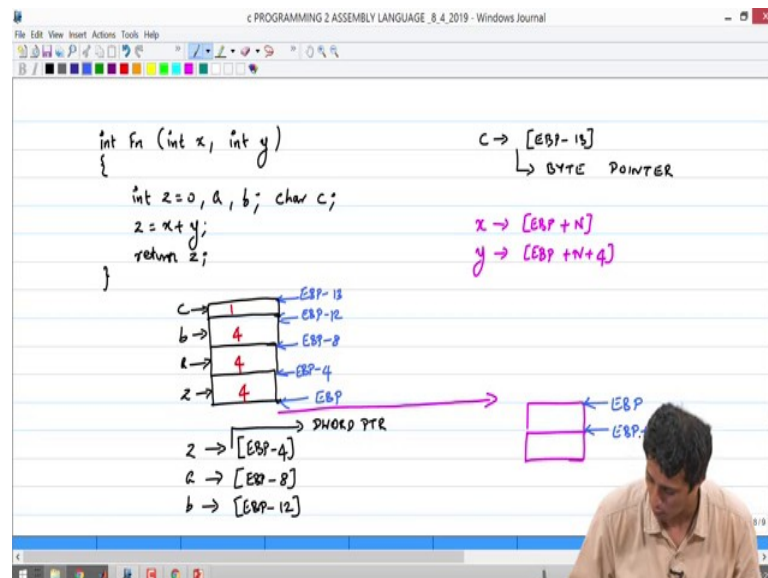
Last lecture we summarized, what the desired features of our compilation process should be. So, we essentially summarized that local variables is a good thing, you do not want to be tied down to using unique name for every variable. So, the concept of a local variable being local to a function is a very good thing. The scope of a local variable is also a very good thing because the moment you enter it is a life and the moment you leave it is gone.

So, that is also a very useful thing. And, the other thing we discussed is because we have a limited number of register in the microprocessor, we need to be able to pass function parameters as well in some other way other than through register. So, essentially we might want to pass as many parameters as needed and for this we also discussed last time that, it is probably best to push this parameters onto stack and pass function parameters as well through the stack into the called function.

So, in this lecture before we go ahead and look at how, a C program gets executed and, what more instructions we need to add. Let us quickly understand, how local variables and these function parameters could be accessed. So, to remind you we had this example where we had `int g_1` where which is a global variable, we had a main function and the main had a local variable `int z=0` and which translated to `move z, 0`. Then, there was a call function which 2 parameters 2 comma 4 and we said we are going to; we are going to push these onto stack using these particular instructions .

We said we will pass them on to the function Fn by pushing them onto stack using `push 4` and `push 2`. So, we are going from right to left. And, then the function called gets translated using the call `Fn_address` right. Similarly, the function Fn had another local variable called `z` as shown here right, which is basically the same name and we had to do some addition, and the function parameters `x` and `y` are now coming in as variables that are being pushed onto the stack. And, of course, once we are done with the function Fn we do a return as shown here. So, now the question is how do I access these local variables? How do I access `z`? How do I access function parameters `x` and `y` in Fn.

(Refer Slide Time: 03:09)



So, let us look at our function Fn which is as follows `int Fn of int x comma int y` . And, there is a local variable `int z = 0` and `z = x + y` and I am doing return of `z`. So now, given that we have established the fact that `x` and `y` and `z` have to be stored on stack , the question is how do I access this? So, the compiler for example, when it sees this function

Fn it knows that there are only fixed number of local variables. For example, in this function there is only 1 local variable called z so therefore, what all that the compiler has to do is to allocate 4 bytes on the stack . Now, here for example, if I had another variable you know maybe a and b . And, maybe I also add to this another character char c , then the compiler looks at this program and says there are 3 integers z a and b.

And, there is 1 character . So, what it would do is it would simply go ahead and allocate 4 bytes of memory for each z,a and b. So, this would be my z, this would be my a and this would be my b. And, of course, now there is a character C. So, instead of 4 bytes it would simply allocate only 1 byte of memory here. So, this is 1 byte ,4 bytes ,4 bytes and 4 bytes. These of course, cannot be accessed in a last int first out manner, because you need to be able to do random alu operations on these.

And therefore, the best register that is that can be used to access these variables is my EBP. So, let us assume that my base pointer register is pointing here somewhere EBP. Then, it is quite obvious that this is EBP -4, this guy is EBP -8, this is EBP -12, and the final address is EBP -13 . So, what would the addresses of z a b and c be. So, the variable z is going to be accessed as contents of EBP-4. The variable a would be accessed as contents of EBP -8, b would be contents of EBP -12. And, C would now be contents of EBP-13, note here because z ,a and b are integers these have to be D word pointers.


So, out here you would have to add D word pointer. And, because C is a character it is only 1 byte I would have to add here byte pointer. So, now, how are x and y accessed. So, if you go back to the function translation that we did before we called Fn underscore address push 4 and push 2 happened right. So, the stack pointer is getting altered through the push 4 push 2 and the call function address and then the local variables are getting assigned here .

So, it is very clear that the function parameters that are being passed will be on the other side of the stack pointer or if we initialize the base pointer correctly it will be on the other side of the base pointer therefore, parameter x and y . So, let me use a different color for function parameters x would be EBP plus you know some number let me call it N, y would be EBP+N +4, because x is also an integer. And, therefore, that is going to occupy about 4 bytes of information .


So, what actually happens is out here if I just continue my stack from here x would be assigned 4 bytes here and y would be assigned 4 bytes here. So, if you look at the base pointer address this would be EBP for example, EBP plus 4. So, with that we are in a position to go ahead and do our preliminary compiler output generation and then we will see what other instruction needs to be added ok.

(Refer Slide Time: 09:41)

Parameters, Local and Global variables




- Before a function is called parameters are pushed onto stack
- Parameters are accessed by $[EBP + n]$
- Local variables are accessed by $[EBP - n]$
- Integers are returned in EAX
- Global variables are accessed by direct address values



So, this is what we had discussed it is a quick summary before a function is called parameters are pushed onto stack. Parameters are accessed function parameters are accessed as contents of EBP plus some offset, local variables are accessed as EBP minus some offset of contents of that . So, we will come to the other points a little later.


(Refer Slide Time: 10:01)

Compile the C program Contd..



```
void main()
{
    int z=0;
    MOV z, 0
    z = Fn(2,4);
    PUSH 0x00000004
    PUSH 0x00000002
    CALL Fn
    MOV z, EAX
    g_iVal = z;
    MOV [g_iVar], EAX
}

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



So, here is an animation that we are going to go through now. In order to understand what other instructions are needed in order to complete this compilation process successfully. So, this is what we had discussed the `int z equal to 0` becomes moves at comma 0 `int function of 2 comma 4` gets translated as `push 4 push 2` and then `call Fn` right.

So, the address `Fn` is whatever the function address is at linking time that will get plugged in here . After, we return the return value let us assume it is going to be in `EAX`. So, that brings us to an interesting point, while we pushed function parameters and local variables on stack. They there the return values from a function can be only 1 value, you can only return 1 value and therefore, you do not have to return through the stack.

So, we choose to return through the register `EAX` by default. So, if you are trying to `int a` return an integer from function `Fn`. The function `Fn` will load the return value into `EAX` and after you return from the function `Fn` out here you just move the `EAX` value into your variable and you are done.

Global variables of course, `g_iVal` is a global variable, you have no choice, but to access them through an absolute address. Now, similarly the function `Fn` with `int x comma int y` gets translated like this `MOV z comma 0 z equal to x plus y` gets translated to these `MOV` and `add` instructions. And, once you are done with the function you just do a `return z` ok. Note that here we are not doing a `return N` at this point, we will come you know

where that is needed a little later. And, also note that that return N has nothing to do with the actual return value that is in z .

(Refer Slide Time: 11:59)

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_jVar = z;
C105 MOV [g_jVar], EAX
.
.

STACK SEGMENT

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

So, now let us go ahead and look at an animation where we look at all the segments; the code segment, the stack segment, the data segment all at once, to see how this function is going to get executed in memory. So, the int the function main has a local variable and I already showed you that the local variables are accessed as EBP-4, because there is only 1 variable now, there is only EBP minus 4 contents of that is my z .


So, let us assume that this particular program was translated in such a way, that the code segment address started in C 1 0 0. These are arbitrary addresses that I am putting I just need a label in order for me to refer to certain addresses. So, I am assuming that the move EBP minus 4 contents of comma 0 is C 100, then there is C 101 where I am doing the push C 102 is again the push of the second function parameter, and then I am calling the function Fn which I am assuming sits in C 200 again it is an arbitrary address.

So, before we start also let us look at the stack segment I am assuming that the stack segment ESP and EBP are pointing at some arbitrary locations, we will see how to initialize them before we start the function main as well . So, we execute the first instruction the arrow that is going to move here is the instruction pointer EIP . So, it has read the instructions from C 1 0 0 which is move contents of EBP minus 4 to 0 . And, you will see that wherever EBP is EBP-4 that location gets initialized to 0, that is the

local variable z. Now, I do a push 0 x you know how many our 0 4. Obviously, this is going to alter my stack pointer, it will move up or it will get subtracted by 4 by 4 values and the integer will get loaded onto the stack.

Similarly, the second push is going to do something very similar and the stack pointer moves up further. Now, when I try to do a call of C 200 the microprocessor as we discussed this earlier knows during the decode phase of this call instruction, that the next instruction where it has to resume from after returning is C 104. So, therefore, before it transfers the control of the instruction pointer to C 200, it will push on to stack the return address which is C 104. So, note that the stack pointer has implicitly moved by a large amount now because of various push and call instructions.

(Refer Slide Time: 15:15)

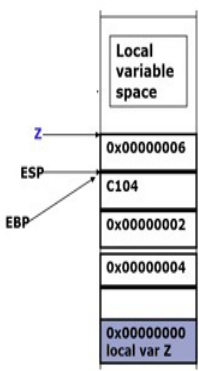
Compile the C Program
Contd..


CODE SEGMENT - Function - Fn()

```

C200  MOV EBP, ESP
C201  SUB ESP, 0x40
      int z=0;
C202  MOV [EBP-4], 0
      z = x + y
C203  MOV EAX, [EBP+4]
C204  ADD EAX, [EBP+8]
C205  MOV [EBP-4], EAX
      return z;
C206  ADD ESP, 0x40
C206  RET
          
```

STACK SEGMENT



Local variable space	
Z	0x00000006
ESP	C104
EBP	0x00000002
	0x00000004
	0x00000000
	local var Z

Now, the instruction pointer is loaded with the address C 200 and we are assuming that our function Fn is sitting in this location C 200. So, now let us look at what happens here. First of all as I already told you any local variable will be accessed as EBP minus 4. So, clearly even before we start execution in this function Fn our EBP is currently pointing somewhere in the local variable space of main. So, in order to bring the EBP to the right context, I first need to move my EBP to the value of ESP.

So, that is the first thing or a first instruction that gets added and something that we did not discuss yet only when we go through this execution process we realize we need to do it. So, EBP is now moved to ESP, and now I need to allocate what is known as the local

variable space. So, I earlier alluded to the fact that local variables are accessed as EBP minus 4 and so on. But, what would happen if I just did a push or a call somewhere here. Because, my ESP is pointing here, if I now did a push it would overwrite my local variable space that is meant for the local variables of the function Fn.

So, in order to allocate local variable space I first need to move my stack pointer far away. So, that there is no clash with my local variables that are being allocated here. Now, how does the compiler know that $0 \times 4 \times 0$ that is 64 bytes is enough well it knows it because the compiler knows exactly how many local variables there are in the function. So, at compile time you can calculate how much of space you need and subtract the stack ESP with an appropriate value.

So, while we claim that the developer does not need to worry about allocating local variable space, which is a true statement. It does not mean that the compiler need not do that the compiler does it by this particular instruction `sub ESP, 0x40`, which means the local variable space is now not going to be affected by any push or pop, because the top of stack is now pointing to after the local variable space. So, now, I go ahead and execute my function. Remember that this function Fn also had a local variable z, which was going to be accessed by contents of EBP-4 note that even in main the contents of EBP-4 referred to the local variables z even there.

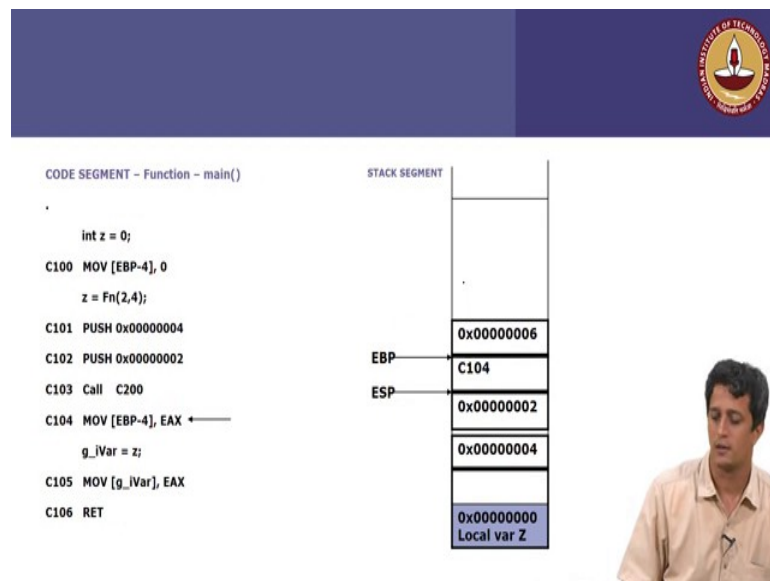
But, here I am now able to access a different variable, because I have moved my EBP to ESP as my first instruction. So, in some sense the context for local variables is set by this precise instruction `move EBP, ESP`. So, when we space when we say that local variables have a context they have a scope that scope is set by this instruction of moving the base pointer to point to this stack pointer even before you enter that function. So, you go ahead and execute the instruction `move z, 0` is move contents of EBP-4, 0, then I am going to do the addition operation of x and y.

Note that x is nothing, but a function parameter which as I told you is going to be accessed as contents of EBP plus some offset. So, I am going to access it as EBP +4. Why EBP +4, because I had to push on to stack the return address when I did a call. So, therefore, the EBP is now actually sitting after the return address which is C 104. Therefore, the local the function parameter 2 and 4 can be accessed only after that which is EBP +4 and EBP +8.

So, I go ahead and execute my program. So, EAX is loaded with the local variable x, then I am going to do the addition operation and after that the contents of EAX will store my result of the summation $x + y$, that needs to be moved into the local variable z which is nothing, but EBP -4 contents of EBP -4. And, therefore, you see that that location now gets the result of my addition . And, now I am done with my program earlier the returned z simply did only one thing for us it called the RET instruction. Unfortunately if you now do the ret as is the top of the stack is now pointing to some random location, because we did local variable space allocation using the `sub ESP, 0 x 4 0`.

We need to undo this operation in order to return and get the stack pointer to the right value . So, what do we do? We have to for every operation on ESP; you have to do the counter operation on ESP before you return otherwise you will have a random returning problem. So, what you do is you add to ESP `0 x 4 0` and bring ESP back to where it was when you returned , when you came into the function sorry.

(Refer Slide Time: 21:49)



So, now you do a return and it simply gets back to C 104, remember that my stack pointer is now pointing to that the top of stack is pointing to the data which is C 104. So, when you do a return the top of stack is now popped into the instruction pointer, which is C 104 as indicated by this arrow and the stack pointer is now incremented to come back to the original location .

So, after returning from the function Fn, I now need to load the return value which is x + y which was going to be stored in EAX into my local variable z, which is addressed by EBP - 4 ok. Now, the issue is clearly EBP is now pointing to the local variable space of Fn. Why is that because we have not brought our EBP back to our original value . Before, we enter the main EBP was pointing here to this local variable z which means that when I did EBP - 4 it would point to this local variable z. Unfortunately, because I have done a call and I moved my EBP to the local variable space of the new function, I am now accessing the local variable space of Fn and not that of main.

We have a bigger problem. In fact, because we do not even know what our original EBP was before we altered our EBP in function Fn. So, that makes it a bigger problem because we cannot even reset our EBP to come back here now . Now, is that all well it turns out that there is another problem .

(Refer Slide Time: 23:49)

The slide displays the following assembly code and stack state:

```

CODE SEGMENT - Function - main()
.
    int z = 0;
C100 MOV [EBP-4], 0
C104 MOV [EBP-4], EAX
      g_iVar = z;
C105 MOV [g_iVar], EAX
C106 RET
    
```

The stack segment shows the following memory addresses and values:

0x00000006
C104
0x00000002
0x00000004
0x00000000
Local var Z

A text box highlights the stack corruption: "Stack corruption!!!! You have accessed the stack of the function 'Fn()'".

So, which means now you have basically done a stack corruption, in the sense you have accessed the stack of function Fn even after returning from that function. So, let us see you know what happens as I execute this program further remember that the global variable is set by absolute addressing right. So, EAX is moved into that. And, now when I do a return what will happen remember that my stack pointer is pointing here ESP. So, when I do a return the top of stack is going to be popped into EIP and execution will continue from there.

(Refer Slide Time: 24:35)

CODE SEGMENT - Function - main()

```
int z = 0;
C100 MOV [EBP-4], 0
z = Fn(2,4);
C101 Call C200
C104 MOV [EBP-4], EAX
g_iVar = z;
C105 MOV [g_iVar], EAX
C106 RET
```

STACK SEGMENT

0x00000006
C104
0x00000002
0x00000004
0x00000000 Local var Z

ESP

Your computer will now REBOOT!!!!

So, if I do that; obviously, the return address is not $0 \times 0 \ 0 \ 0 \ 2$. So, what have we done in the process of calling this function, we have pushed on to stack 2 variables, 2 with values 2 and 4. And, we have forgotten to undo this stack operation before we did a return. And, this is what we are going to correct in the next lecture .

(Refer Slide Time: 25:11)

Topics Covered

- Passing function parameters
- Address of local variables
- Allocating local variable space