

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

Lecture - 09

(Refer Slide Time: 00:15)

C PROGRAMMING AND ASSEMBLY LANGUAGE

SWAPPING VARIABLES

```
main ( )
{
    int x=2, y=3;
    int temp;
    temp = x;
    x = y;
    y = temp;
}
```

```
--asm {
    PUSH x
    PUSH y
    POP x
    POP y
}
```

ESP

x

y

Welcome back to this course on C Programming and Assembly language we are in module 2, where we have discussed various examples of doing inline assembly implementations of arithmetic logical instructions some loops, some pointer arithmetic and so on. So, let us continue with our discussion and in this lecture we will particularly discuss some very interesting operation of Swapping two-variables in C programming, that is quite extensively discussed in you know various techniques are available for swapping variables you know with the use of temporary variables without the use of temporary variables and so on.

So, by itself it is a very interesting problem and once you throw in the inline assembly option as well then it makes it even more interesting. So, in this lecture we will look at Swapping two variables. So, let us start with the vanilla C implementation that we know of right. So, I have two variables x equal to 2, y equal to 3 and I want to swap these two variables let us say using a temporary variable . So, what do we do, we just simply say temp equal to x move the value of x into a temp , then we say that x can take the value y and then y can simply take the value temp, .

So, the problem with this kind of an implementation is that you are going to introduce a lot of assembly instructions in order to perform this particular operation. So, to simplify that we can always look at more optimized assembly; inline assembly operations that will help solve this problem. So, let us look at what the simplest assembly implementation is to swap two integers. So, we are assuming that these are double words or 32 bits in length. So, what do we do I am going to now simply take this part of the program and I am going to replace it with some inline assembly implementation, asm.

So, instead of dealing with so many variables what we are actually doing is we need another variable temp which has a particular address and we are trying to move data in there and you know and so on. So, is there any other simpler way right in which we can do this in assembly where we do not have to deal with an other temporary variable. So, the answer is yes and how do we do this we simply use the stack. So, a stack is as we discussed it is a last in first out operation. So, if I perform two instructions PUSH x, PUSH y, then what gets moved onto stack is first the value of x then the value of y and this is my stack pointer, this is where my top of stack pointer is.

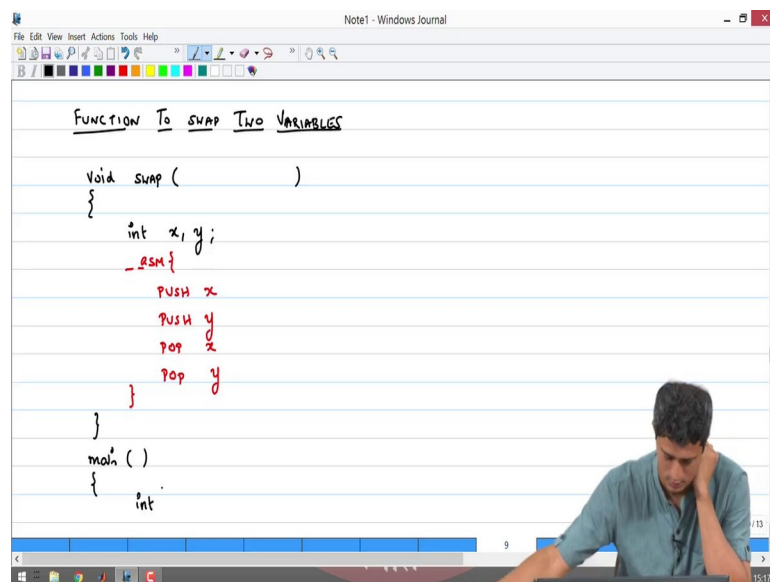
Now, if I simply go ahead and POP in exactly the same order, POP x POP sorry POP y, then what would happen is the value of y which basically sitting on the top of the stack would simply come into the x here and my stack pointer would then move below. And when I do this next POP the value of x will simply get popped into y, so this will move here and this will move here. So, you can directly see that if you choose to go into some assembly implementations, it is very easy to solve certain problems. Note here that this does not mean that we are using any you know lesser memory locations. In fact, we might be using one extra memory location.

So, if you look at the implementation on the left, there is one temporary variable temp which basically is some memory location and we are simply dealing with that temporary memory location. So, what we are doing here is we have x which is going into temp, then x takes on the value y and then y takes on the value that was in temp, so is what we have done. So, we have actually introduced one extra memory location and done a couple of moves in order to SWAP x and y. What we have done here on the other hand is simply used two extra memory locations on the stack and with four simple instructions

we were able to and we are not dealing with any registers or any such complicated thing here.

So, if you look at the assembly instruction of the code on the left right, assembly translation as this it would involve moving data into some registers as well which we are completely avoiding in the inline assembly implementation on the . We simply use four instructions PUSH x, PUSH y, POP x and POP y in order to swap these two values. So, that is you know swapping two integers in a particular function. Now let us assume that I want to do this operation more often , so what I want to do is I want to write a function to SWAP two variables.

(Refer Slide Time: 07:03)



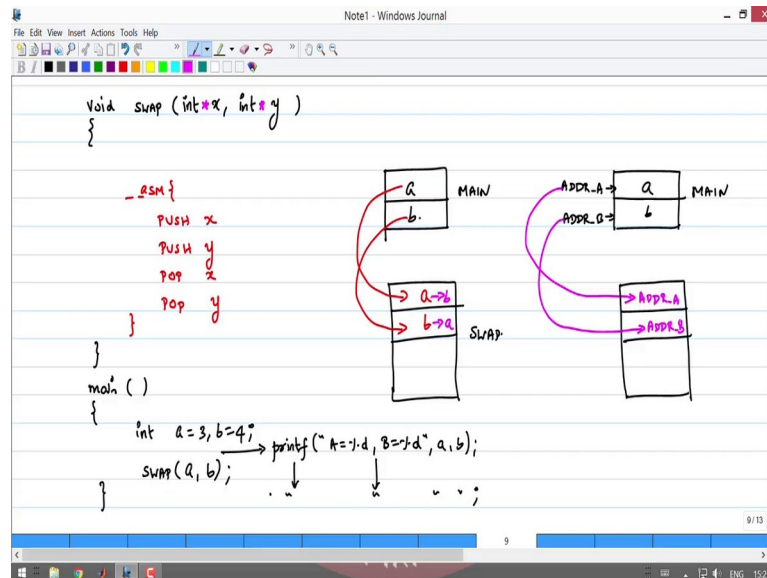
```
FUNCTION To SWAP TWO VARIABLES

void swap (      )
{
    int x, y;
    asm {
        PUSH x
        PUSH y
        POP x
        POP y
    }
}

main ( )
{
    int
```

So, let us first look at the assembly C implementation ; so, I want to say void SWAP of something I am not going to fill in the arguments for now. Let us say that I have int x and int y sitting here and then I go ahead and do the swapping routine in assembly language right, PUSH x PUSH y POP x POP y. Now what happens here is I have the main program which is sitting here and I want to call this function.

(Refer Slide Time: 08:33)



So, I have int a equals 3 and b equals 4 and I want to call this function SWAP of a comma b ok. So, let us assume that instead of int x and y being local variables in my function, I am going to call so I am going to put this as function arguments int x comma int y. So, what happens is a and b are sitting somewhere in memory so, this is my a and this is my b and I am now trying to call this function to SWAP a and b remember that I want to SWAP a and b in main. So, let us assume that I have two print statements printf A equals percent d, B equals percent d, a comma b and then I call this same print statement again after calling the SWAP function.

So, the intended output of this program is that it should print 3 and 4 first and after calling swap it should print 4 and 3. Unfortunately if the function is declared exactly as shown here where int x and int y are the arguments that we are going to pass that inspite of performing the, or calling the function swap both these print statements would print exactly the same value 3 and 4. So, the reason for that is that when I call this function swap, what is happening is a copy of a and b are being sent into my function swap.

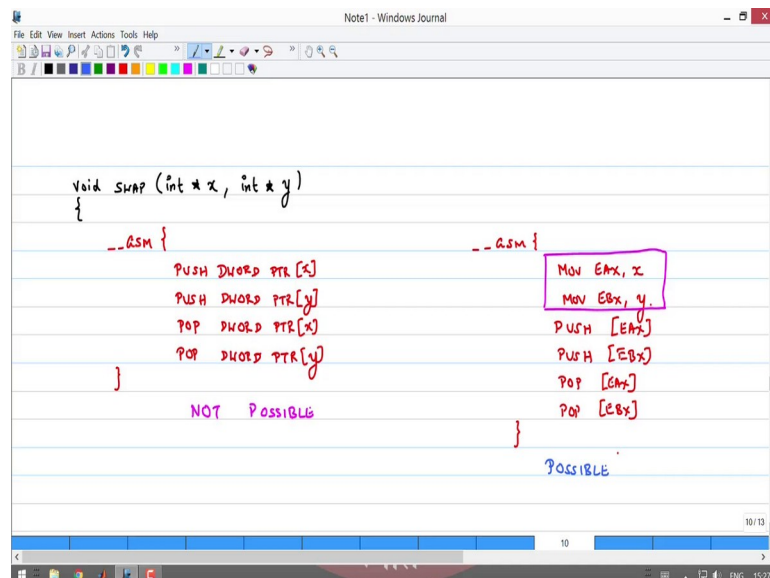
So, what happens is swap has some memory space and this a will get copied here this b will get copied here so and the function executes the instructions in swap right which is basically PUSH x, PUSH y, POP x, POP y and what happens is a and b get swapped in this space b becomes a ok. So, if you look at this, this is main this is my SWAP, will we will come to a more physical understanding in a physical picture of this memory a little

later in the course right. I am just driving the concept of why we need a pointer to these integers in order to actually have a function to SWAP two variables as opposed to just two integers directly.

So, when I come back to my main and print there is nothing that has happened in a and b right. So therefore, it is not possible for me to go ahead and implement this swap function exactly as it is. So, the correction that I need in order to go ahead and implement this is I need to call these pointers, I need to have these as addresses as opposed to direct variables. So, what happens in that case is that in main I have a and b with addresses ADDR or maybe we will redraw this picture here .

So, I have my MAIN here a and b and I have address ADDR underscore A, ADDR underscore B as you know their addresses. What I want to do is I want to pass into this particular function not the values of a and b, but the addresses of a and b. So, what I want to do is I want to make a copy of these particular addresses into this location, so this will get ADDR underscore B . Now in order to implement this particular swap with pointers I need to make some modification of my program here, so we will look at what that modification is in the rest of the lecture.

(Refer Slide Time: 13:57)



```
void swap (int * x, int * y)
{
  --asm {
    PUSH DWORD PTR [x]
    PUSH DWORD PTR [y]
    POP  DWORD PTR [x]
    POP  DWORD PTR [y]
  }
  NOT POSSIBLE

  --asm {
    MOV EAX, x
    MOV EBX, y
    PUSH [EAX]
    PUSH [EBX]
    POP [EAX]
    POP [EBX]
  }
  POSSIBLE
}
```

So, I have a void SWAP of int star x int star y. So, if I were to go ahead and directly do an assembly implementation what do I need to do. So, these are addresses that I have . So, what do I need to do, I need to simply SWAP the contents pointed to by ADDR

underscore A and ADDR underscore B. So, if I perform a SWAP on the contents of these addresses then I would have been successful in swapping the two variables in the main function itself.

So, the question is can I go ahead and do this indirect addressing in the following manner, PUSH contents of x and because these are integer pointers let me actually explicitly write that D WORD PTR of x PUSH y I will do POP D WORD PTR of x POP y. So, technically this should have been because what I am saying here is PUSH the contents pointed to by x onto stack then PUSH the contents pointed to by y onto stack right and then POP in the same order. The question is this possible and the answer is no this is not possible, that is going to be the discussion for the rest of this course and to understand why that is not possible; we need to go and figure out how these variables x y star x star y are all stored in memory.

We need to physically understand what it means to say that these variables are actually stored on stack, how they are addressed and then we can answer the question why I cannot do this kind of an indirect addressing. So, I need to do indirect addressing but I need to do it with a small modification which is as shown here, so unfortunately this is not possible.

So, what is possible the following is possible, I will MOV EAX comma x MOV EBX comma y and then I will do PUSH I am leaving out the D WORD PTR here, I am going to say EAX EBX POP POP EBX. Really if you look at these two implementations they are not very different except for a fact that I had to add these two instructions in order to make the indirect addressing possible, so this is possible and this is correct.

So the discussion in module three will revolve around how x and y are stored in memory. Until this point we have treated all these variables x y a b integers characters all these are abstract entities we have really not worried about where they are stored, how they are stored and how they are accessed. We just said that they could be read from or they could be written into how we did not worry about. Now unless we worry about them we will not be able to answer this particular question and the focus of the rest of the course we will be only to worry about this particular aspect.

(Refer Slide Time: 18:58)

Topics Covered

- **Swapping Variables**
 - C implementation with temporary variables
 - Inline assembly implementation
 - Function to swap variables