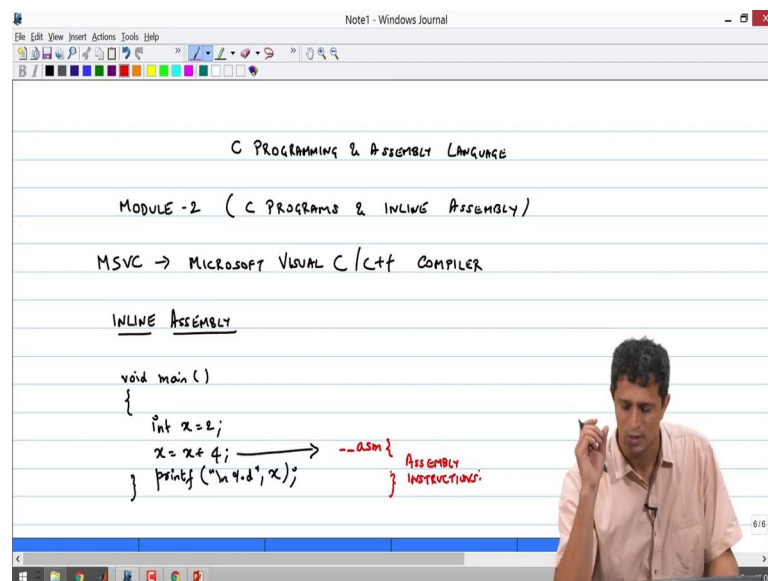


**C Programming and Assembly language**  
**Prof. Janakiraman Viraraghavan**  
**Department of Electrical Engineering**  
**Indian Institute of Technology, Madras**

**Lecture - 06**

So, welcome back to this C Programming and Assembly language course. In the last module we discussed in reasonable detail the various instructions from the 886 architecture that are relevant to C programming, we discussed a few examples and hopefully the assignments would have reinforced many of those concepts. So now, we move on to module 2 which essentially deals with the C programming and inline assembly.

(Refer Slide Time: 00:43)

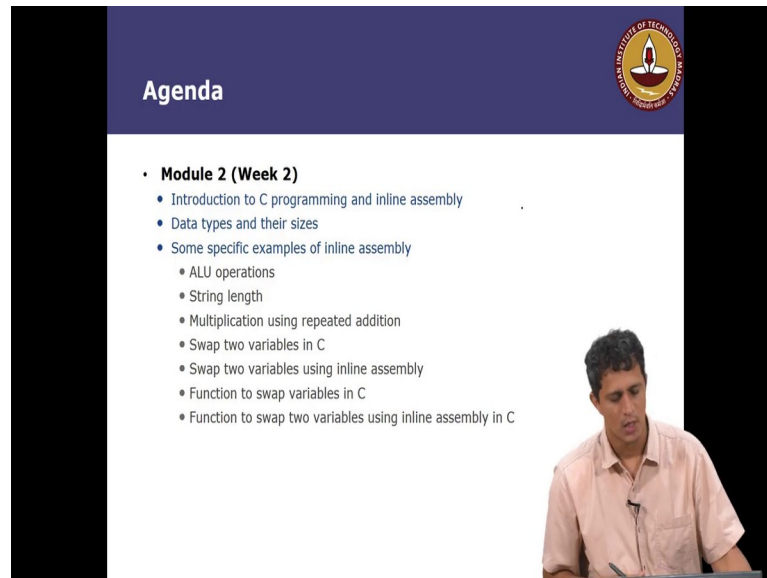


So, primarily we are going to deal with C programs inline assembly. So, what is inline assembly? Inline assembly is nothing, but a simple way of moving from a C program to assembly language and then coming back to a C program.

So, essentially it's a way in which you can intersperse assembly instructions in between a high level C like program, by the way before I proceed here I have to mention that like I said in my introductory class that I assume that the viewers and the students of this course are already familiar with C programming. And I am not going to go into any detail not even as much detail as I did for the assembly language of 886, I am not going

to go into any such detail of the C programming syntax or the functionality or you know any such thing. I am going to primarily deal only with inline assembly language examples and thereby reinforce some of the C programming concepts. So, you can always go back and refer Kernighan and Ritchie in case you are in doubt of any of these C syntax or the C functionalities.

(Refer Slide Time: 02:19)



**Agenda**

- **Module 2 (Week 2)**
  - Introduction to C programming and inline assembly
  - Data types and their sizes
  - Some specific examples of inline assembly
    - ALU operations
    - String length
    - Multiplication using repeated addition
    - Swap two variables in C
    - Swap two variables using inline assembly
    - Function to swap variables in C
    - Function to swap two variables using inline assembly in C

So, what are we going to do in module 2? So, we will primarily deal with C programming inline assembly then so, we will talk about some of the data types and their sizes you know just to make sure that we tie it well with our discussion on the micro process that we had in module 1.

Then we will look at some very specific examples. So, this module will be primarily run based on some examples, but the examples have been carefully chosen to drive home a certain concept which can be modified later or will become much clearer as we go on with the course. So, we will look at some ALU operations, string length function, multiplication, then we look at swapping of two functions of two variables in a function in some detail so, there are various ways in which you can do it in C.

So, we will look at how you can actually do this better with assembly language how would you do this if you were to do it using a function instead of swapping variables within within that scope. So, these are some nitty gritty details and some details that get driven home when we discuss these particular examples we will come to it a little later.

So, first of all before I proceed into concrete examples let me also state that there are different kinds of compilers there is a GCC compiler, there is a turbo C compiler, there is a Microsoft visual C compiler and so on, there are many compilers for C and C plus plus. So, which one do I pick so? I have picked technically should not matter which one you pick, because they are all ultimately they all implement the same functionality, but I have picked a compiler which essentially allows this inline assembly coding most easily .

So, to do inline assembly programming you need to follow a certain syntax certain compilers like GCC, the syntax is pretty complex and is not worth it for us because you do not want to get stuck in the syntax of these operations rather than actually understanding the concept. So therefore, I have picked the MSVC compiler , this is the Microsoft visual C compiler, C or C++ compiler does not matter. So, let us quickly get into you know what inline assembly is all about.

So, let us assume that we have this function void main int x equals to and I m going to do you know x is equal to x plus 4, printf slash n percent d comma x. So, let us assume that I want to translate this particular instruction alone into assembly, I want to do only this instruction in assembly. I want to leave the rest of the syntax as it is, which means that the rest of my function has to be interpreted as a C program which means I can follow the syntax of C programming there, only certain key instructions which I want to speed up I want to move to assembly language.

So, it turns out that it is very much possible and in Microsoft visual C or visual C plus plus that is achieved simply by putting this directive called underscore underscore asm and in flower bracket you go ahead and write any assembly instructions. So, you can put in any assembly instructions here, now the beauty is that the variable names x y and z whatever we have as variables can continue to be used inside this assembly directive flower braces that we have put. And we can also use all the registers and other assembly instructions as it is from the assembly instruction set that we have.

(Refer Slide Time: 07:23)

The screenshot shows a Notepad window titled 'Note1 - Windows Journal'. The main content is handwritten code and a table. The code is as follows:

```
x = x + 2;
--asm { // x = x + 2;
    mov eax, x // EAX ← x
    add eax, 0x0002 // EAX ← EAX + 2
    mov x, eax // x ← EAX
}
printf("\n %d", x); // 4
```

Red annotations include a circle around 'x = x + 2;', a bracket on the right side of the assembly block labeled 'INLINE ASSEMBLY.', and arrows pointing from the C code to the assembly instructions.

To the right of the code is a table titled 'DATA TYPES IN C':

DATA TYPES IN C	
BYTE OF DATA	→ CHAR
WORD OF DATA	→ SHORT INT / INT
DWORD OF DATA	→ LONG INT / INTEGER

A person is visible in the bottom right corner of the Notepad window, appearing to be speaking.

So, let us look at a very concrete example first . So, let us assume that I want to do this x equal to x plus 2 operation in assembly . So, what do you do? So, I will write underscore asm bracket open the instruction I want to do in comments I am going to put it here is x plus 2. So, what do you do? You just say move EAX comma x.

Now, what this x is in terms of microprocessor data and microprocessor registers we will look at later let us not worry about it now, for now assume that x is a variable and is accessible inside this in inline assembly portion as well. So, then what do I do? I do add EAX comma 0 x 0 0 0 2 . So, what is it done? It has simply moved EAX the register it has got the value x here, it has done EAX EAX plus 2 this value has not yet got replaced into x.

Therefore, I need to execute another instruction which is move x comma or rather small x EAX. So, this executes the instruction where x will eventually get the value EAX. So, now, if I go ahead and do my printf percent d comma x, what gets printed here is of course, it depends on what x was initialized 2. So, x was initialized to 2 here .

So therefore, this will simply print 4 for you here. So, notice how we simply translated only this instruction x equal to x plus 2 into a particular assembly block, this is known as inline. So, what we are going to do in this module is to reinforce the assembly language instructions that we learnt in module 1, through inline assembly C programming. So, that

way we sort of cover some concepts of C as well as reinforce the instruction set that we learnt in module 1.

So, with regard to that let us look at our first example, write an assembly program to evaluate the following expressions . So, let us assume all variables are 32 bit integers so that brings us to an interesting discussion, saying what are the typical data types that we will use in C. So, typically these remember that the logical memory map allows us to deal with multiples of bytes.

So; obviously, the smallest unit that we can deal with is going to be a byte of data. So, therefore, byte of data so, this is what is known as a character CHAR in C. Now you could also deal with a word of data depending on you know whether you are dealing with an older processor, a new processor or with what kind of compiler this could either be a SHORT integer or an integer itself.

Now, we could also deal with the D WORD of data, again depending on the compiler this could be a LONG INT or it could be an INTEGER. So, the main point is that because the registers and the logical address mapping is going to deal with multiples of bytes or words or D words the data types in C are also mapped to similar sizes and that is what we see in all our compilers across various kinds of processors.

(Refer Slide Time: 12:47)

### Exercise 1



➤Write an assembly program to evaluate the following expression. (All variables are 32 bit integers)

- »  $EAX = x * y + a - b$
- »  $EBX = (x \wedge y) | (a \& b)$



So, here this example let us assume that all variables are 32 bit integers and we want to perform the operation  $x$  into  $y$  plus  $a$  minus  $b$  and load it into EAX register. We want to do  $x$  x or  $y$  or  $a$  and  $b$  and load it into the EBX register ok. So, let us start of by writing our inline assembly.

(Refer Slide Time: 13:17)

```

void main()
{
    int x=2, y=3, a=4, b=5;
    // EAX ← x*y+a-b
    // EBX ← (x*y) | (a&b);

    --asm {
        MOV EAX, x; // EAX ← x
        MUL y; // [EDX:EAX] ← x*y
        ADD EAX, a; // EAX ← x*y+a
        SUB EAX, b; // EAX ← x*y+a-b
    }

    // EBX = (x*y) | (a&b);
}

```

Handwritten annotations in red:

- Red bracket on the left side of the assembly block labeled "INLINE CODE 1".
- Red bracket on the right side of the assembly block labeled "INLINE CODE 2".

Let us void main going forward I may not always write this void main. So, let s assume that int x equals 2, y equals 3, a equals 4, b equals 5 and what is the operation that we want to perform, EAX should get loaded with x into y plus a minus b. So, let us go ahead and do this thing in assembly language underscore underscore asm. So, what do we do, we first load x into EAX, remember that when we want to do multiplication EAX is an implicit register and because this is a 32 bit integer that we are talking about we have to deal with EAX and not just a x.

So, therefore, MOV EAX comma I will just call it x, then I am going to do a MUL y. So, what is this instruction done for us, it is simply loaded EAX with the value x. What is this operation done, it has loaded EDX and EAX together the 64 bit number as x into y. Now, I go ahead and add remember that now EAX has my answer there . So, I go ahead and add a to it sorry ADD EAX comma a and finally, I subtract EAX comma b. So, of course, this particular program will work only if the higher EDX happens to be 0 . So, it is an interesting exercise I leave it to you to figure out how to modify this, if EDX also happens to be a non 0 number because of the multiplication.

So, what are we doing here, EAX is simply now  $x$  into  $y$  plus  $a$  and EAX eventually is  $x$  into  $y$  plus  $a$  minus  $b$ . So, I can close this bracket and this concludes our arithmetic operation that we wanted to do  $x$  into  $y$  plus  $a$  minus  $b$  and we are loading the result eventually just in EAX we are not interested in getting this to any other variable.

So, now let us look at the logical operation we wanted to implement EBX equals  $x$  x or  $y$  or  $a$  and  $b$ , remember all these are bitwise operations. So, therefore, I again go ahead open my underscore underscore asm block I load EBX with  $x$ , then XOR EBX with  $y$ , then I have to know because there is a bracket I have to do the  $a$  and  $b$  very carefully I MOV ECX with  $a$ , then I do an AND of ECX with  $b$  and then I do an OR of EBX comma ECX.

So, this operation is EBX will get  $x$ , this operation is EBX is  $x$  x or  $y$ , this operation is ECX gets  $a$ , this is ECX is  $a$  and  $b$  bitwise and the final operation is EBX equals  $x$  x or  $y$  or  $a$  and  $b$ . So, I can close this and of course, I eventually I can close my C function as well out here. So, there are 2 blocks that we have introduced into the C programming syntax to just illustrate the concept of inline assembly, this is 1 inline INLINE CODE 1, 2. So, with that now let us move on to another interesting example, which is you know where we are going to reinforce the concept of jumped instructions and loops in assembly language.

(Refer Slide Time: 20:01)

## Exercise 2



➤ Write an assembly program to evaluate the expression " $z = x * y$ " using

» Repeated addition

» MUL instruction

➤ Write an assembly program to calculate the string length of a constant string

We want to write an assembly program to evaluate the expression “z equal to x into y” using repeated addition.

(Refer Slide Time: 20:13)

```

z = x * y; // Repeated Addition
x & y 16 bit short ints
main ( )
{
    shortint x = 2, y = 3;
    int z = 0;
    // z = x * y;
    --asm {
        XOR EAX, EAX // EAX ← 0
        MOV ECX, y // ECX ← Counter (y)
        MULT: ADD EAX, x // EAX ← EAX + x
        DEC ECX // ECX ← ECX - 1
        → JNZ MULT
    }
    MOV Z, EAX // z = x * y.
    printf("x * y = %d", z);
}

```

So, what do we want to perform, we want to perform z equals x into y using repeated Addition . So, let us assume that x and y are 16 bit short integers right. So, we go ahead x equals 2 y equals 3 and let us say int z equals 0 and of course, the intended operation is z equals x times y. This is just one statement in C programming, but we want to now break this down into a repeated addition operation, how would you do this in assembly just to drive home the point of doing a branching and looping operation in assembly language.

So, this is the instruction that I want to convert. So, therefore, I will now introduce my underscore underscore asm block here. So, repeated addition is just adding x y times . So, what do you do? You first have to clear some register where they are going to add this any number of times. So therefore, I do x or EAX comma EAX . So, this is EAX equals 0 irrespective of what EAX was, the value of at the end of this XOR operation is just 0.

So, I am clearing the register and I am going to MOV ECX with y , this is my counter I am going to add x that many times with my eventual register. So, what I do is, I start adding now EAX comma x after this I need to decrement ECX because I have now added it once I need to decrement ECX remember that on decrementing ECX the 0 flag



may or may not be set. So, you need to do this operation of decrementing or adding  $x$  to itself as many times as the value of ECX does not go to 0.

So, therefore, here you have to do a jump on no 0 to this address here there is let us call this maybe I will write it in a different color to indicate that this is a label or an address. So, I am going to jump to this label called MULT. So, here what are we doing, we are simply loading the COUNTER ECX is which is nothing, but  $y$ .

Then add EAX with  $x$  I am adding  $x$  equals or plus  $x$  and this is ECX becomes ECX minus 1 and as long as it is not 0 you keep adding this  $x$  to itself. So, when the for example,  $y$  is 3 here so, after 3 counts  $y$  will come down to 0 and that is the condition when the instruction instead of looping all the way back to MULT will actually proceed which means, when it hits 0 the 0 flag will be set. And therefore, jump or no 0 will not be satisfied and the instruction will proceed to the next step and where I am ready to load my final answer into  $z$ .

So, therefore, here MOV  $z$  comma EAX so, when you come here  $z$  will be  $x$  into  $y$ . So, again I finish my inline assembly block and if you want you can do a printf here of percent  $d$  which is  $z$  value and you will see that the answer is 6. So, here we have illustrated apart from all the ALU operations we have also shown how you can exploit the Jump on NO Zero operation to loop back to a particular address depending on a particular condition.

So, in the next lecture we will look at some more examples of a string length and so on to reinforce certain other assembly instructions that we studied in module 1.

(Refer Slide Time: 26:43)

## Topics Covered

- **Inline assembly code**
- **Examples**
  - Addition
  - Multiple arithmetic operations
  - Multiple logical operations
  - Multiplication using repeated addition
    - Jump on No Zero - JNZ