C Programming and Assembly language

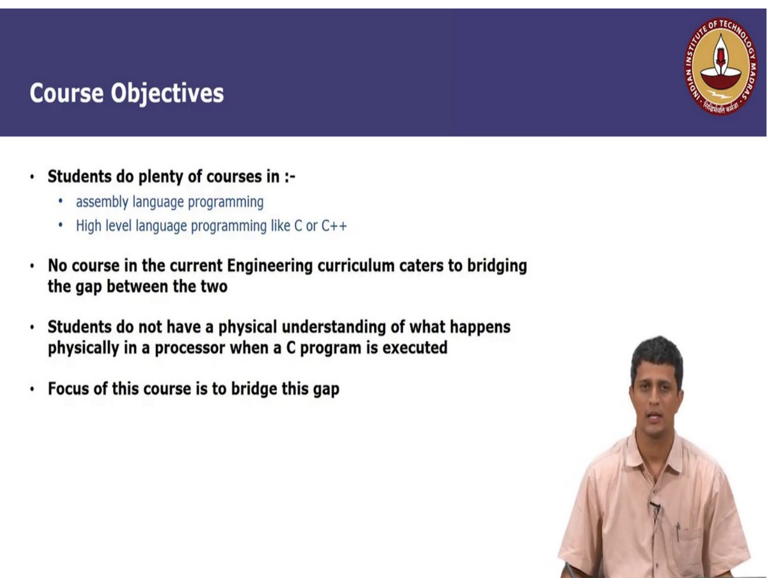
Prof. Janakiraman Viraraghavan
Electrical Engineering Department
Indian Institute of Technology, Madras

Lecture - 01

So, welcome to this course on C Programming and Assembly language. So, at the outset let me state that; lot of students do plenty of courses in C programming and other high level languages. And also they do many courses on assembly language programming, on various micro process and microcontrollers. What is really missing in the current curriculum is a bridging gap between these two languages. That is to say students do not have a physical understanding of how a C program or a high level language is actually executed in a microprocessor. And people do not understand what exactly happens in various segments of the memory, when a program is being executed.

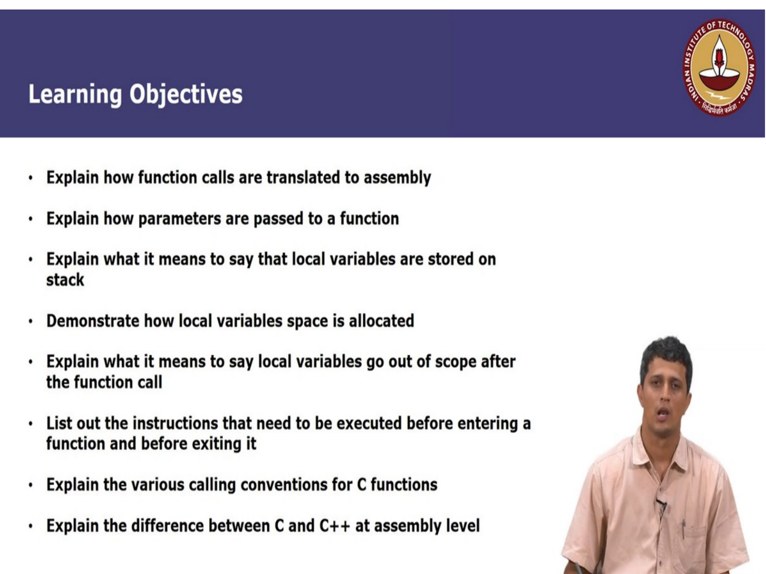
The idea of this course is to bridge, this very gap. For example, when students are asked where local variables are stored in C? The answer comes very quickly, that it is stored on a stack. And then asked, what kind of scope do, variables in a function have, they call it a local scope or a global scope unfortunately there is no understanding of, what physically these things mean. In this course, we hope to address these issues.

(Refer Slide Time: 01:27)



So, the course objectives really is to ensure that, we are able to show to students, what exactly happens when a C program is executed in the microprocessor by way of animations and some simple examples.

(Refer Slide Time: 01:47)



Moving on to the learning objectives of this course, which by the way is what a student should be able to do, on completing this course. The following are the learning objectives, you should be able to explain how function calls are translated to assembly language, explain how parameters are passed to function, explain what it means to say that local variables are stored on stack, demonstrate how local variable space is allocated by a compiler .

This is a week, you at least have to be able to show one way, in which this can be done. Explain what it means to say local variables go out of scope after a function call, list out the instructions that need to be executed before entering and before exiting a particular function in C.

Then we will move on to be able to explain, what various calling conventions are in C functions right, for example, you have a function which has a variable argument list, as opposed to a fixed argument list, how do these functions differ in assembly language and that is where calling conventions actually come in. Then you should be able to explain simple difference between C and C++ at assembly level. C++ is a very powerful programming language that offers object oriented programming.

But the key point to note is that at assembly level it is not very different from C. There is very little change that you need to incorporate when you move from compiling a C program to compiling a C++ program and we will demonstrate this by way of an example again.

(Refer Slide Time: 03:23)

You should also be able to exploit certain hardware instructions to speed up C functions and then you should also be able to explain why recursion is not a great idea for performance. Typically you will see that, recursion though is a very powerful way of programming and conceptualizing an idea, it is not necessarily the most efficient way of coding or programming it in a microprocessor.

(Refer Slide Time: 03:53)

References and Pre-requisites

References:

- "The C Programming Language" by BRIAN W. KERNIGHAN and DENNIS M. RITCHIE, Second Edition
- "The INTEL Microprocessors – Architecture, Programming and Interfacing", by Barry B. Brey 8th Edition

Pre-requisites: Students should be comfortable with

- C programming – Focus will be on Inline Assembly
- Assembly language programming – Not necessarily the INTEL x86

We will look at the reasons for that in this course. So, the references and prerequisites , so, there like I said there are plenty of courses that students do in C programming, students do in microprocessor assembly language programming. But for C programming you can typically refer any book, but of course, there is nothing better than the Bible called the C programming language by Kernighan and Ritchie, the second addition. As far as a microprocessor assembly language goes, what I am focusing on this course can be explained with any microprocessor.

And it may be slightly different from microprocessor to microprocessor, depending on the assembly instructions that are available. But unless, we actually freeze on a particular architecture or on a particular assembly language, it is not going to be possible to get a physical understanding of what actually happens at the lowest level. For this reason, I decided to adopt the Intel microprocessors architecture and programming for this course. The reason is when you compile a program on a normal desktop, which is based on an Intel microprocessor, which is mostly the case, you will see that the assembly instructions are exactly the, what I am referring to in this course.

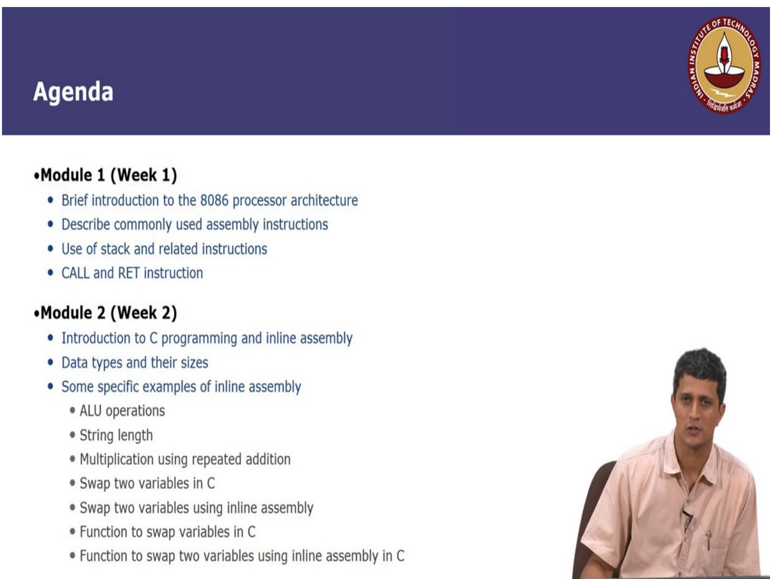
So, for the Intel microprocessor architecture and programming there is again no better book then very Barry Brey's book on Intel microprocessors architecture and programming. So, I referred the second edition, but there is nothing different even if you

refer the 8th edition, which is the latest book available. As far as prerequisites go , I expect that students are already comfortable with C programming .

So, I will not be going into any details of C programming here. For example, what a function is, what an array is, what variables are, I am not going to go into any of those details. I am assuming that you already know, how to program, simple programs in the language C. However, I will focus on something called inline assembly which again can be put in any high level language. But I will look at it in the context of C programming. For assembly language programming, I am assuming that students have worked on assembly language programming of some microprocessor, be it arm or you know the any other microprocessor.

So, I just expect familiarity with assembly type of instructions. I am not assuming that you know the Intel architecture or the Intel assembly instruction set already. So, I will spend some time describing these instructions. So, that you get familiarity, but I am not going to go into gory details of how these instructions are you know executed or you know the need for these instructions. I am not going to go into any such detail. I am going to give only some working familiarity with the assembly language of the Intel architecture.

(Refer Slide Time: 06:49)



So, the agenda for this course is, I have broken this four week course into four modules. Module 1 which will be covered in the first week, will be a brief introduction to the 8086

processor architecture and then describing commonly used assembly instructions right. Here when I say commonly used assembly instructions, I mean the assembly instructions that you will typically encounter when you translate a C program into assembly language. There are numerous other instructions that are, there in the x 8 6 architecture of Intel , which can be used for hardware programming and so, many other purposes.

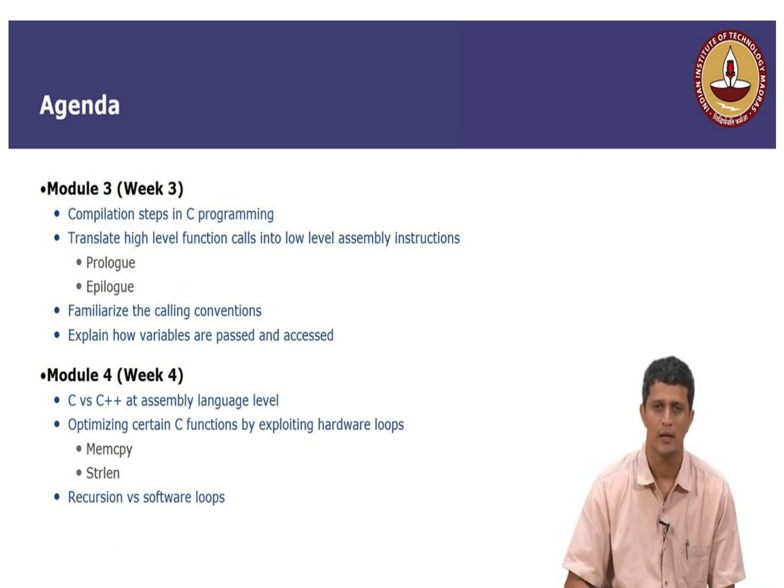
But that is not the focus of this course. I will be only dealing with a subset of the Intel x 8 6 instructions, that apply to C programming. Then we will look at the use of stack and related instructions. And finally, I will look at the call and return instructions in some detail. Again all these instructions are typically used in any microprocessor, the idea is to just get you in, to the Intel style of coding these instructions. In the second module which is in week 2, I will give an introduction to C programming and inline assembly.

So, Inline assembly is a way in which we can actually intersperse some assembly instructions within a C program. So, you have the syntax of a C program and there is a certain way in which, I can actually switch from a high level programming language into a low level programming language, like assembly language, do certain instructions and then come back to my high level programming language again. Then, I look at the data types and their sizes which are typically used in C and relate them to, you know what we do in a microprocessor. Then I look at some specific examples, of inline assembly. And these examples have been chosen specifically to drive home certain advantages, which we will see in later modules.

So, I look at ALU operations, the string length operation, multiplication using repeated addition, swapping two variables in C, this is a very interesting exercise that all students do in C programming, where you know by using a temporary variable, without using a temporary variable and so on right. So, we look at that example. Then we will look at swapping two variables using inline assembly language. Later we will move on to a function to swap two variables in C right and we will also look at swapping two variables using inline assembly in C.

So, they have various flavors of these swapping functions that, I would like to look at and each example will drive home a certain advantage and a certain concept, which we will cover and come to at a later point in the week 3.

(Refer Slide Time: 09:45)



Module 3 which we will cover in week 3 is essentially the main focus of this course. The idea is to take a given C program and simply compile that into low level assembly language.

So, the idea of introducing inline assembly is that, at least the basic instructions which are the arithmetic and logical related instructions can be translated pretty trivially, as we will see. However to actually translate the entire function needs certain knowledge of what exactly happens and there is something called a prologue and epilogue that has to be executed for each function. A prologue is a set of instructions that are executed, before you enter into a function and an epilogue is a set of instructions that are executed before you exit that function.

So, the idea is to drive home the need for a prologue and an epilogue and to even derive what instructions actually need to be there, by way of an example and animation. In the animation, I will ; give you an exact idea of what happens at assembly language level in the code segment, in the data segment, in the stack segment, all simultaneously, right. And with those examples, we will see what happens if this prologue and epilogue were not there., what is the exact need for these instructions, .

So, then I will move to the calling conventions as, I briefly mentioned earlier and then we will look at how variables are passed and accessed. So, these are the key points that

we will touch on in week 3. Module 4 which is the final module of this course is where we are actually going to compare C and C++ at assembly language level, right.

So, we will take an example of a structure in C and a class in C++ and then we will find that there is actually not much of difference other than the, you know how in C plus plus you have certain restricted access, while in C you do not, right. So, what do these things actually mean at assembly language level is, the going to be the focus of this first discussion in module 4. Then we will look at optimizing certain C functions by exploiting hardware loops right.

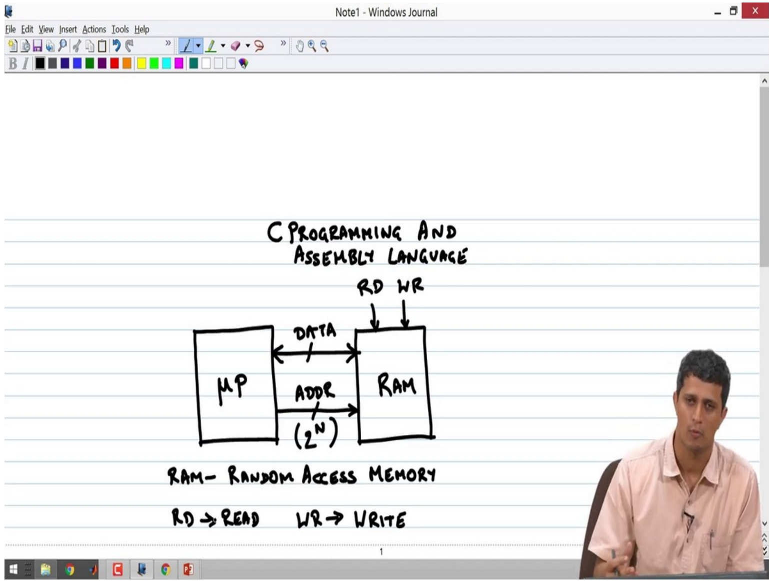
So, for example, memcpy, string length and some maybe a few other instructions or functions we will also look at, . The idea of discussing this, is to show you that it is not a good idea to actually code, certain basic functions like string length. For example, using loops and you know the regular style of coding that we do. You need to exploit certain instructions that the hardware gives you in order to speed up this process and that is what you will see in any library that is provided to you.

So, if you as well actually disassemble that code and look at the assembly implementation of a string length function, the heart of the function will be this hardware instruction, which is well used to exploit loops at a hardware level and not at a software level. We will also discuss, why it is a good idea to sort of convert recursion into software loops, people do this on a regular basis and we will look at, why that is the case.

(Refer Slide Time: 13:23)

So, with that let us move into module 1, right. So, the idea here is to first give you a brief introduction into the 8086 processor architecture, describe the commonly used assembly instructions, use of stack and related instructions and then focus on the call and return instruction. So, with that let us enter the world of microprocessor programming.

(Refer Slide Time: 13:49)



So, the microprocessor is actually very sophisticated chip that can do many things for us. I am going to abstract out then necessary definition and the necessary details that we need, in order to proceed with the learning objectives of this course. So, a

microprocessor can simply be defined as a black box that can do certain computations, . So, we will call this mu P and this microprocessor talks to what is known as a random access memory, .

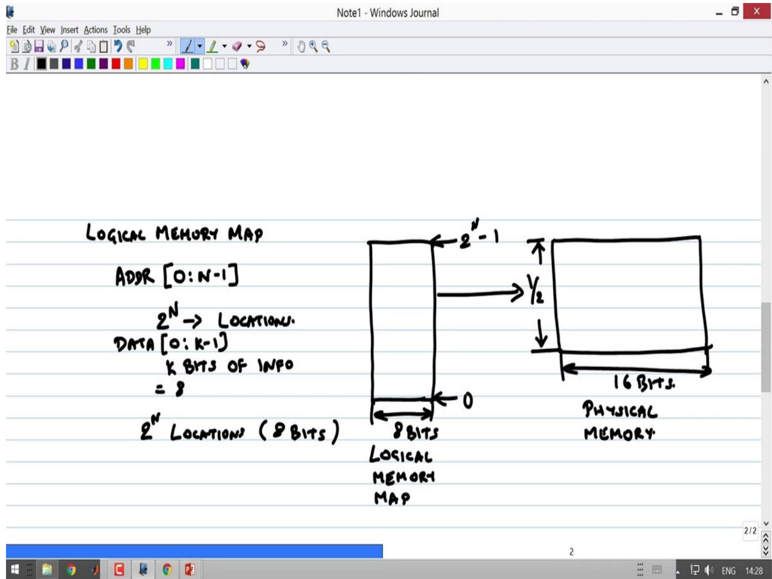
So, a RAM is as we all know, is Random Access Memory. So, a microprocessor has what is known as a data bus, right and the line here indicates that this could be more than one bit it could typically be 8 bits, 16 bits, 32, 64, . And this data bus is actually going to be bidirectional in nature which means that, microprocessor can send data to the memory, or it can receive data from the memory.

So, this is my data. The microprocessor also has what is known as an address bus. This again is a multi bit number, depending on the size of the memory and how much memory it can access, it is that many bits will be assigned. Typically it is going to be 2 power N, if N is the number of address bits, then the number of locations that it can logically address is 2 power N. So, the memory also has two other control signals called the read and the write.

So, as far as we are concerned a memory is again a black box, which takes data and addresses inputs and two control signals called read and write, right. So, the RD is read and WR is write. So, the black box called memory here, if you issue a read command, which means that you are going to actually make the read command go high for a short while, and you present an address to it, than that particular location will be read out and placed on the data bus, . Similarly if you assert the right command and you actually present an address to the microprocessor, then this would simply write the data that is available on the data bus, into that particular location in the memory .

So, that is all a memory is and that is all we care for in this course, . For example, a memory could be an SRAM or a DRAM or various other kinds of memories are there non-volatile, volatile, I am not going into all that. I am just going to call a memory as a random access memory in this case, which has the features that I just described. So, the other thing that we need to abstract out is what is known as a logical memory map. So, what we want to look at is a Logical Memory Map.

(Refer Slide Time: 18:03)



So, what is a logical memory map? So, I have an address, which can be about N bits. So, the address bits are a 0 to a N minus 1, and these could actually refer to 2 power N locations. And in each location, I could write K bits of data. So, my data bus could be about 0 to K-1, right. So, it has about K bits of info. So, the idea here is, we are trying to separate out the physical memory implementation from a logical addressing implementation.

So, by logical addressing, what I mean is that there are 2 power N locations and in each location, I would say for example, I could say that we write 8 bits of information, for example, this k could be like 8 bits . So, there are 2 power N locations, each 8 bits long, this is the logical address which means that location 0 to 2 power N minus 1 each of it is, actually going to be 8 bits in length.

So, this is my location 0, this is my location 2 power N- 1, right and each of this is going to be 8 bits. This is called a logical memory map. This does not mean that, the physical memory has an 8 bits data bus, connecting to the microprocessor, it could be 32 bits or it could be 16 bits. For example, I could say that my physical map is only half of these locations and each location I can write 16 bits, and this is half of this, this is my physical memory. As far as the program is as concerned, you do not have to worry about the physical memory arrangement, all you have to worry about is a logical memory map,

which basically says that, if you are given an address with N bits you can write or read 8 bits from that.

(Refer Slide Time: 21:49)

Topics Covered

- Course summary and learning objectives
- References and pre-requisites
- Microprocessor and memory
- Logical and Physical memory