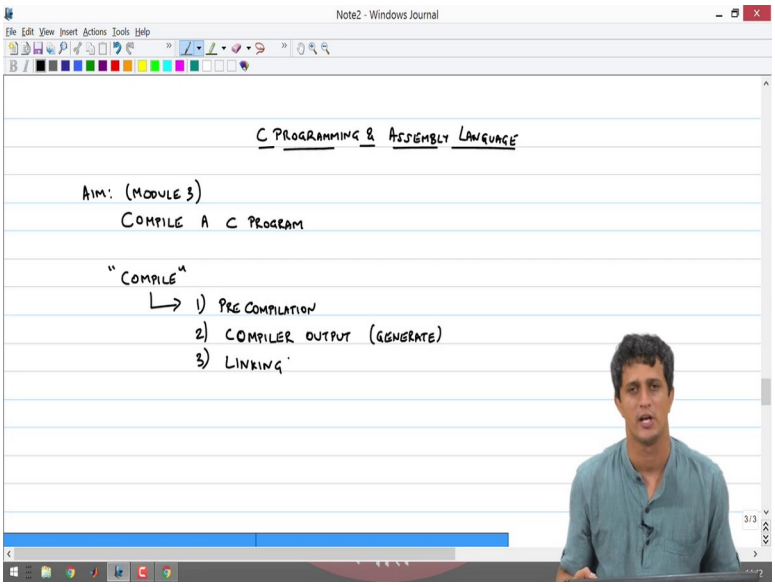
C Programming and Assembly language
Prof. Janakiraman Viraraghavan
Department of Electrical Engineering
Indian Institute of Technology, Madras

Lecture - 11

Welcome back to this course on C Programming and Assembly language. We now ready to move into module 3 of this course. In module 1, we looked at a lot of assembly instructions and some detail to understand the x 86 instruction set. In module 2, we looked at the concept of inline assembly and a way to generate some unoptimized assembly output, right, which basically was pretty straightforward if you know what the instruction set of the particular processor is.

(Refer Slide Time: 00:51)

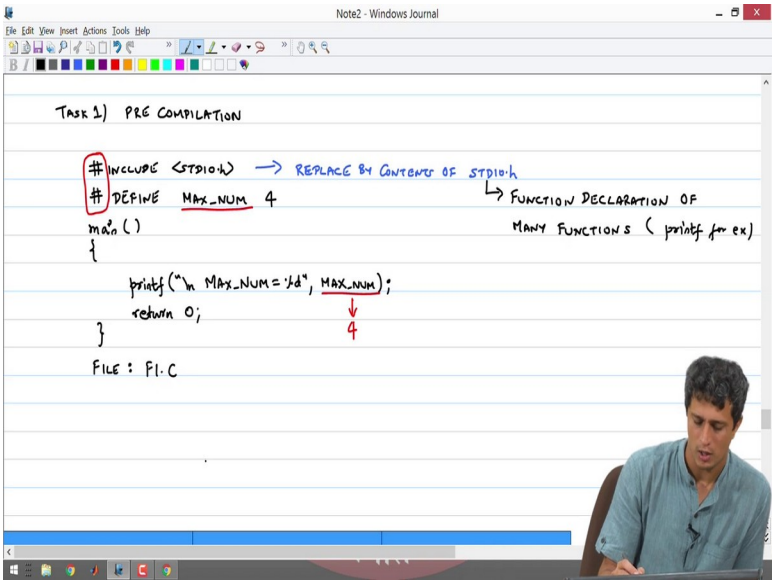


The aim of module 3 is to simply be able to compile a given C program into its assembly output, . So, the aim for is compile a C program, . So, the word compile is sort of used in a very generic manner, ok. So, I think before we jump into the actual task of compiling a C program into its particular assembly output it is important for us to understand what the phases of compilation are, .

So, there are three main tasks that are performed when we say we are trying to compile a particular program, . So, when we say this term is used very loosely. We actually mean that we are going to do the following three tasks which is precompilation. Second task is

the actual compiler output, . I will not use the word compile as it is here, I will call it the compiler output, has to be generated, . So, this, and the third one is the job of linking the compiler outputs that we generated in the previous step.

(Refer Slide Time: 02:57)



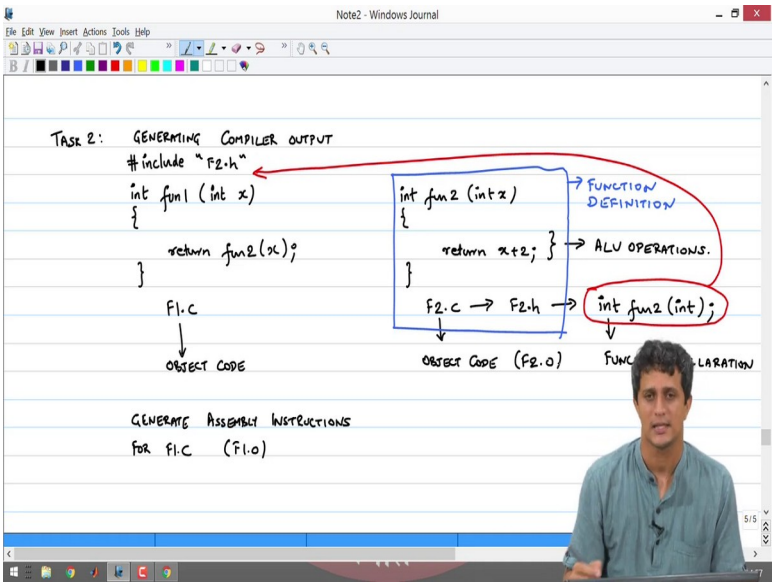
So, let us start with the pre compilation step. Task 1 is; so, what does this do? As the name suggests before you compile, you are going to process this file and generate a few things. So, what are the few things? So, for example, if you do hash INCLUDE STDIO dot h, ok, of course, this has to be in lowercase, cannot be in uppercase. And let us say you do a hash DEFINE MAX underscore NUM as 4, ok. Then am going to define a function main that is simply going to printf percent d comma sorry MAX underscore NUM, and I will do return 0.

So, this is I am going to place it in a file called F 1 dot c, ok this is my file. So, what does precompilation do? Precompilation essentially deals with all of these hash tags, . So, when you do a hash INCLUDE STDIO dot h, what we are saying is simply take the contents of the file STDIO dot h and replace it in place of this line. So, this one will simply get replace by contents of STDIO dot h, which means that if you look at this intermediate output before the actual file gets compiled then you will see a lot of you know information that has been added and that is primarily. Because all your header files that have been included have simply got replaced in place of these hash INCLUDE statements. So, what does the hash DEFINE MAX NUM equal to 4 do?

So, this simply does a find and replace of this particular keyword MAX NUM and replaces it by whatever is to the right of that, . So, deliberately I have put this MAX NUM inside the quotes as well when I do a printf in my main, . What the pre compiler does is, it replaces only this guy by 4, . And of course, you must also realize that now when I do printf, there is no definition of printf in my code, printf is not a function that I am writing, so therefore, it has to be defined elsewhere.

Now, where is it defined? It is defined in some other file, but all the compiler cares for during the compilation phase is for what is known as a function declaration. So, this STDIO dot h has the function declaration of, ok, of many functions you know including printf, printf for example. With that let us now move on to our second task which is basically the process of generating the compiler output, .

(Refer Slide Time: 07:49)



So, here for example, let me take two examples here, let me take two files, where I have int function 1 of int x and I am going to say return function 2 of x, . This is my file 1 dot c, . I am going to now have another file which is int fun 2 of int x again and I will do return x plus 2 and this is sitting in F 2 dot c.

So, the idea here, of generating the compiler output is to simply take each individual C file and generate the assembly output for whatever is in that file, . So, what we need to do is to simply convert this to what is known as object code, . So, what we are going to focus on in module 3 is actually only on task 2, and some of task 1 also will be covered.

Remember that the compiler explorer that we discussed earlier in module 2 does only task 1 and task 2. We will see what task three is a little later,.

So, now the problem is when I want to generate this the assembly output for this particular object code for this ; for this function F 1 and fun 2. We will see what complications there are in the process. So, if I want to do for example, do this return x plus 2, , so all I have to do is really do some sort of an ALU calculation, if I want to translate this to its assembly code this will be some ALU operations, . And this is pretty easy you just have to for example, move it into a register and then add 2 to it, and then call the return, right. You have to put the result in ax that is the only catch here, which we will discuss a little later.

Now, in order to translate the output of F 1 dot c, and generate the compiler output for the same, there is a small complication here because we really do not know what fun 2 is in this place, because fun 2 is not defined in my this file. What I need to do now is to tell the compiler, in F 1 dot c itself what fun 2 is,. And therefore, what you do is for if every dot c file you have what is known as a F 2 dot h file a dot h file, a dot h file. And what does this file contain? It simply contains the following, it just says int fun 2 of int. It just contains the function declaration.

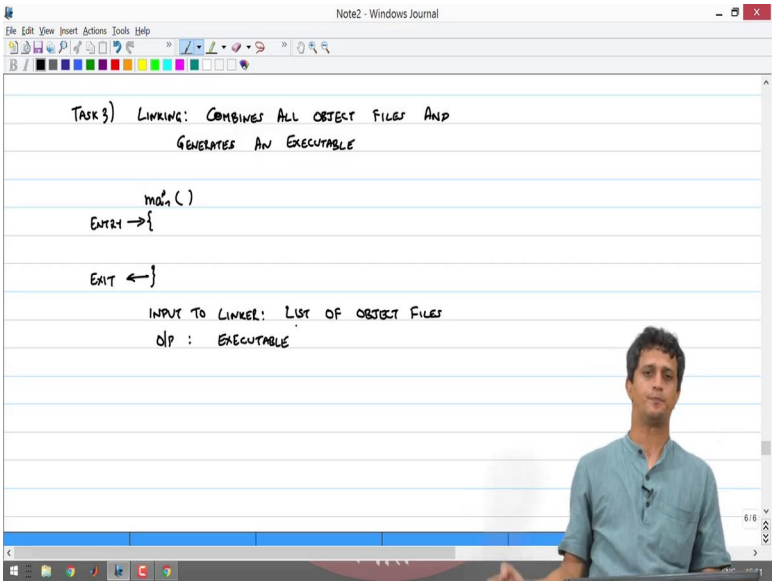
So, therefore, if I want to be able to generate the object code for F 1 dot c, without worrying about what is an F 2 dot c, then I need to add the function declaration of fun 2 and that is achieved by doing this hash include, it is F 2 dot h, . So, now what happens? The precompiler takes care of simply putting the contents of this file which is basically this statement into this line here, and then we are now looking to compile this particular C code. So, what happens is when you see if encounter the function fun 2, right the function call fun 2 of x, it knows that there is a function fun 2 which takes an integer and returns an integer and therefore, the function call is correct it will just translated to the corresponding assembly code, .

So, the job of a compiler output is to simply generate the assembly instructions for that particular file without worrying about what the contents of the other files are, . For example, when I compiled fun 1, and gave a function declaration of int fun 2 of int, , it does not care if there is a function definition for fun 2. It does not care if this particular definition exists or not, . This is the function definition. So, F 1 dot c does not care for or

does not even know if a function definition exists or not, that is taken care off in the third task which is known as linking, .

So, the objective of generating the compiler output is to assembly instructions for F 1 dot c, and the output is called F 1 dot o and object file, . Similarly, the compiler when you compile F 2 dot c will generate and other object code called F 2 dot o.

(Refer Slide Time: 15:53)



So, now the third task which is basically linking is going to put all of these files together and see that a consistent executable can be made. So, what does the process of linking do? It simply combines all object files and generates.

So, now if F 1 dot c for example, had the function declaration of the function 2, but F 2 dot c was not compiled and was not linked to the object file here, then you would get what is known as a linker error. On the other hand, if the function F 1 dot c, let us go back to that page. On the other hand, if the file F 1 dot c did not have the hash include F 2 dot h which is basically the function declaration of F 2 of function 2 then you will get what is known as a compiler error, . So, once we understand the difference between these two errors we will be able to debug our code and fix the errors very very easily, .

Now, the linker also looks for what is known as the main, . So, if I want to write a particular C program it is mandatory that I have a function called main, it could be void or int does not matter, that exists in my code, . What is the job of this main? The

operating system is now running on the microprocessor, and if I want to run my executable then the entry point to my function is the entrance that is defined by main. So, this entry point and this is my exit point for my function, . So, which implies that the operating system hands over control to your program at the entrance of main and when it exits main control is handed over back to the operating system.

So, the linker checks for the existence of this function which is a keyword called main and it will give you an error if there is no such function, . It will also link your function calls to all the other object code that has been generated, . So, the linker essentially it takes, , input to linker is basically the list of object files, and the output is an executable, .

So, in this course we are going to focus mainly on task 1 and task 2. Actually, mostly on task 2, we are not even going to worry about task 1, . So, the job of the compiler now, I am using the word compiler very specifically because as I mentioned earlier when we say we compile a C code, its very loosely used to talk of all these three tasks together precompile, compile and link, . But going forward when I say we are compiling a code it means that we are specifically doing task 2, assuming that task 1 has been already completed successfully.

Of course, what I have really not mentioned here is, if there is any syntax error and stuff like that you cannot write any garbage and get the code to compile. So, all those syntax errors are also caught during the compilation phases, but that is not the focus of our course. We are assuming that, we are writing syntactically correct code and our job is now to generate the assembly output for that particular file which is now going to be consistent when we link.

(Refer Slide Time: 20:52)

Topics Covered

- **Compiling a C program – Rudimentary introduction:**
  - Pre-compilation
  - Compilation
  - Linking