

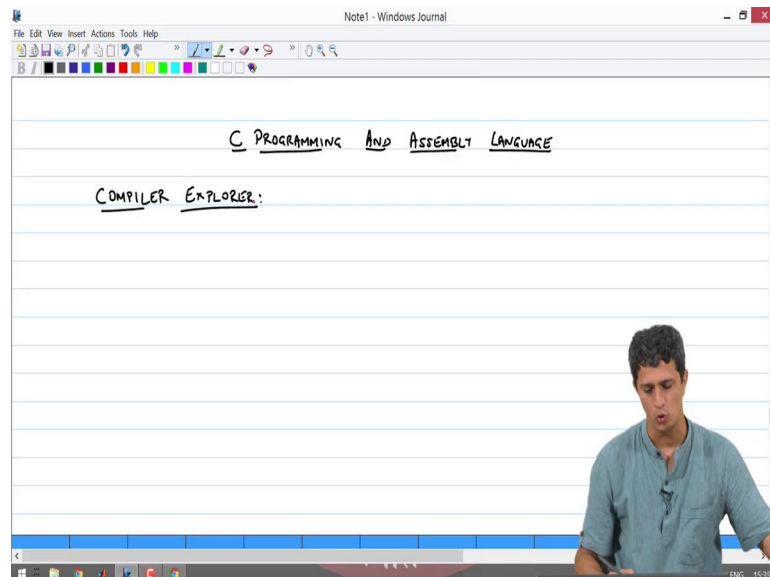
C Programming and Assembly language
Prof. Janakiraman Viraraghavan
Department of Electrical Engineering
Indian Institute of Technology, Madras

Lecture - 10

Welcome back to this course on C Programming and Assembly language. So, in the last couple of lectures in module 2, we have been looking at various examples of C programming and inline assembly. So, we also looked at optimize compiler output versus unoptimized compiler output, you know pointer arithmetic and you know swapping variables and so on.

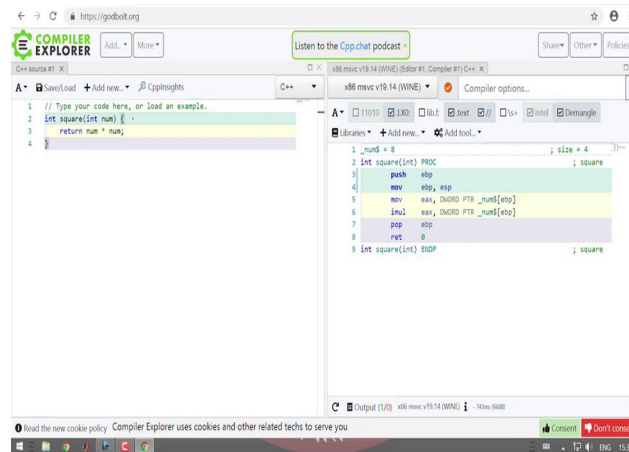
So, in this lecture, I will introduce you to a tool wherein you can actually play around with C programs and assembly language in order to understand how you know the compiler output changes when you make various; you know changes either in compiler options or in the programming code.

(Refer Slide Time: 00:57)



So, the tool that we are going to use is basically something called a COMPLER EXPLORER. This is an open source tool that is available for everyone to use.

(Refer Slide Time: 01:19)

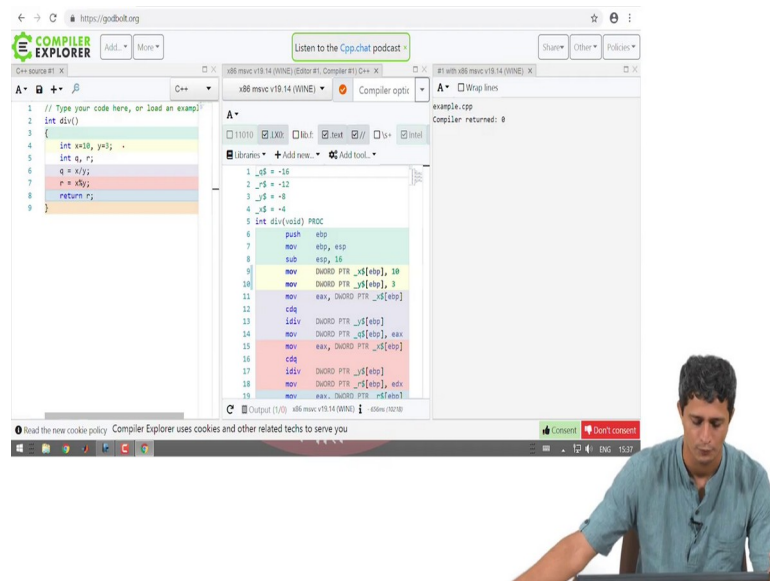


And, if you open this website called <https://godbolt.org> what you would see is a screen like this . So, on the left you can go ahead and type your C programming example that you want to deal with or you could even load an example file that you want and on the right what you see is the corresponding compiler output for this particular file.

So, before we proceed with deal you know doing some examples, I would like to tell you that we have been specifically dealing with the msvc compiler because the interspersing assembly inline instructions is the easiest and we have been dealing specifically with the x86 code. So, when you do this, you will find that you know the instructions come out in a more familiar fashion and they are also easy to sort of understand and debug.

So, let us go ahead and you know start keying in a few examples that we discussed here . Let us for example, discuss that you know division example that we spoke about .

(Refer Slide Time: 02:33)



So, I have int x equal to 10 y and then y equal to 3 and int q and r write quotient and remainder q is equal to x by y and r is equal to x percentage y . So, what does this tell you? It tells you straight away that there is a compiler failure and therefore, you just go ahead and click this output here. And you will see that you know, it tells you that there is some error here what is the error they must return a particular value .

So, what is the reason for that because I said int div which is a function. So, let me just arbitrarily choose to return r here . So, what you see on the left is now a color coded version of your code . For example, in cream you have this int x equal to 10, y equal to 3 and on the right hand side you have some other you know particular code and you have the corresponding colors appearing here as well. So, what this means is that if you take the cream color code int x equal to 10, y equal to 3; it translates to assembly instructions as shown here.

So, do not worry about this you know DWORD pointer underscore x you know dollar ebp; basically just assume that this is the variable x and this is the variable y that is all we are dealing with . And if you have DWORD pointer underscore q you know, dollar of ebp that is the variable q and the other one is the variable r. So, do not worry about this complicated notation, we will exactly come to this point in a couple of lectures and we will we will tell you why we need to do all of this .

So, coming back to our code the cream color code now simply translates to `mov x comma 10, mov y comma 3` . And if you place your pointer over this instruction, it actually tells you exactly what that instruction does at assembly language level.

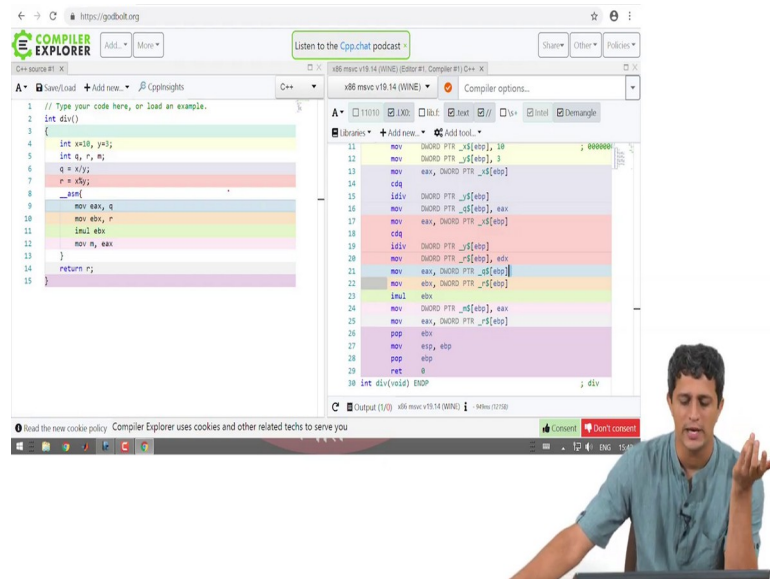
So, this is done by a person called Mad God Bolt whose done a wonderful job at creating this website precisely for this reason. You can actually play around with C code on the left hand side and immediately see for yourself, what the output is at an assembly level. You can do this even with other compilers you know on your desktop, but you got to go through many steps in order to do it is not so easy . So, let me close this window so that we get more space here .

So, if you look at now the next instruction `q equal to x by y` it, translates to a couple of instructions here which we already discussed earlier which is basically moving the value of `x` into `eax`. Then there is one more thing which they do here which is `CDQ` , it doubles the size of the operand. So, I leave that as an assignment for you to figure out what the use of that particular instruction is .

Then you do an `idiv` of `y` and then you would move the value of `eax` into the variable `q`. Next you come to the pink highlight which is basically `r equal to x percent y` that again translates to four similar instructions again which you know is a `mov` a `cdq` and `idv` and a `mov` again. So, clearly this is nothing, but the unoptimized code that we saw. What you see on top the green is basically corresponds to the open braces of the function and the last you know, `color` basically points to the closing braces of the function.

The interesting point to notice even those things translate to certain assembly instructions and in this course, we will tell you what those instructions should be and why they are; the way they are today . So, you can now go ahead and do you know more changes in this .

(Refer Slide Time: 07:09)

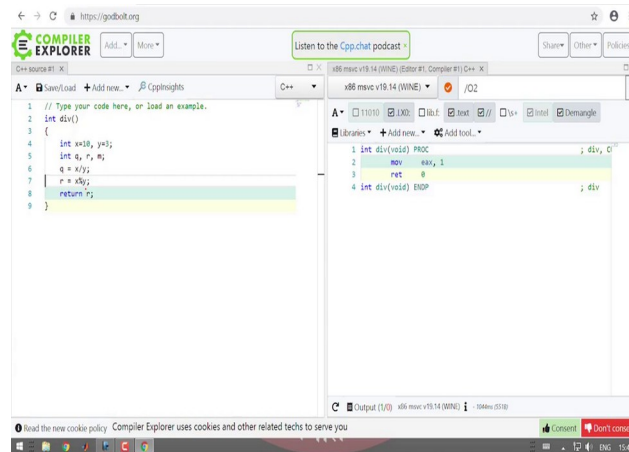


For example, I could start doing inline assembly code here; asm bracket right and for example, I can do `mov eax comma the quotient` and then I can do `and mov ebx comma the remainder ebx comma ;` otherwise you get a compiler error immediately and then I can do an `imul` of `ebx`. And the answer is now going to be in `eax` and therefore, I can now choose to do something with that answer. For example, I could create another variable here as `r` and you know `m`; I just call it `m` and I will move this particular answer `mov eax comma m`. So, I have to do it the other way; I go to move into `m` `eax`.

So, you can see that when you now you know move to that, I can put in some inline assembly code here and the color coding again is still valid. You can see the corresponding code on the right hand side also the blue `mov eax comma q` is basically this instruction here `mov ebx comma r` is particular is this instruction `imul` of `ebx` and so on. So, this is the unoptimized assembly code.

So, it will be interesting to see what happens when you actually try and optimize this particular assembly code. So, let us look at maximum you know optimization. So, what happens here is you know ; let us remove this assembly part for now. So, what you see is it is quite amusing.

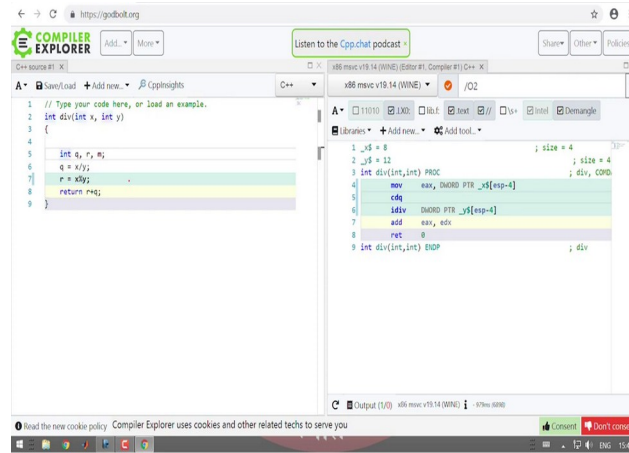
(Refer Slide Time: 09:35)



What the compiler has done is it has simply optimized all the instructions out of this function and why is that, because none of these variables, you know though you are doing some computation; you are not using them in any particular way . And it is also interesting to note that even though we are returning the value r, you do see that what has happened is you have moved a constant into eax and just done a return 0 .

So, what does this mean? So, let us for example, replace this with 4, then what you see is that the mov eax actually becomes the value that is moved into eax is now 2. So, what the compiler is doing is because these are constant values 10 and 4 and I am not dependent on inputs to the program the compiler is actually evaluating this remainder and quotient in a hard coded way. So, it is doing 10 by 4 and taking the remainder which is 2 and pushing that into eax . So, that is all it is doing. Now in order to optimum; if you want to remove this degenerate case, all you have to do is make int x and y as arguments to this function .

(Refer Slide Time: 10:55)

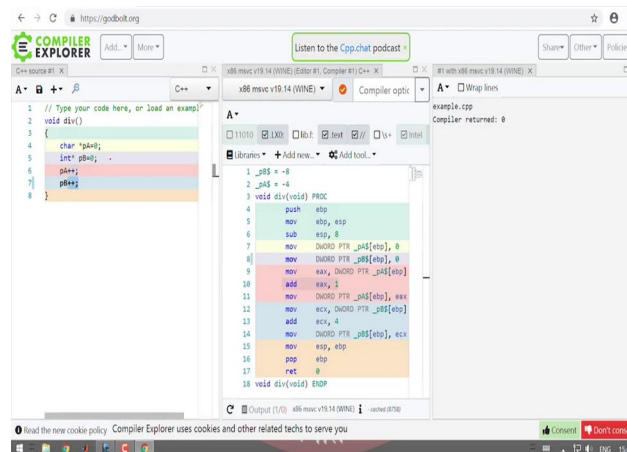


So, then you see that x and y are now variables and then you are going to move them into eax then you do a cdq and you do an $idiv$ of y . So, now, you note that just like the example, we discussed earlier the optimized implementation of this particular program does only one division called $idiv$. It has stopped moving data into various registers again and again; it has calculated the x by y quotient and remainder in one shot and loaded the result into r here of course, .

So, if I for example, did return r plus q , then it would simply add eax and edx ; basically eax has the quotient edx has the remainder, you add these to the value is in eax and you are done. This is return 0 has nothing to do at assembly level; when I say return 0, it has nothing to do with the return of r plus q that I am doing here. We will come to that at a later point in our discussion.

So, I urge you to go ahead and play around with this software trying to code various programs and, see what happens when small things change from one place to another. So, let us look at one more example, here I am going to now talk about the pointer arithmetic example that we did.

(Refer Slide Time: 12:45)



The screenshot shows the Compiler Explorer interface. On the left, the source code is:

```
1 // Type your code here, or load an example
2 void div()
3 {
4     char *pA;
5     int* pB;
6     pA++;
7     pB++;
8 }
```

 On the right, the assembly output is:

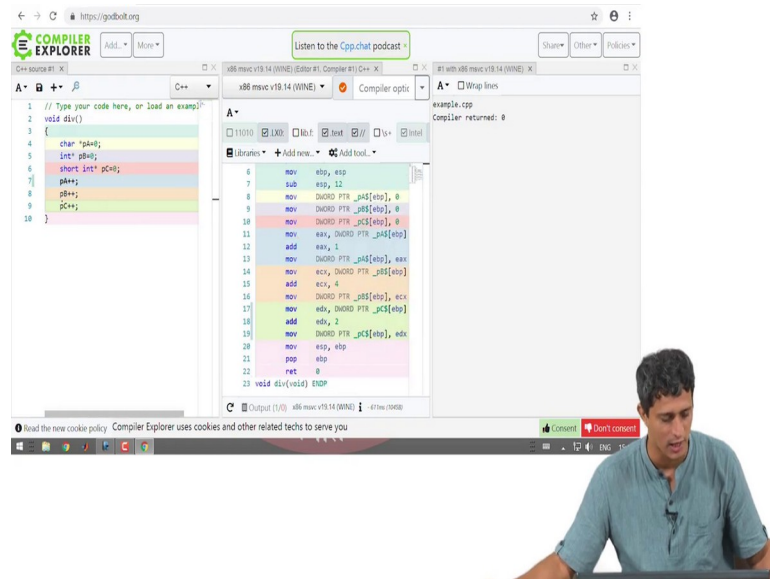
```
1  _pB = -8
2  _pA = -4
3  void div(void) PROC
4      push    ebp
5      mov     ebp, esp
6      sub     esp, 8
7      mov     DWORD PTR _pA[ebp], 0
8      mov     DWORD PTR _pB[ebp], 0
9      mov     ecx, DWORD PTR _pA[ebp]
10     add     ecx, 1
11     mov     DWORD PTR _pA[ebp], ecx
12     mov     ecx, DWORD PTR _pB[ebp]
13     add     ecx, 4
14     mov     DWORD PTR _pB[ebp], ecx
15     mov     esp, ebp
16     pop     ebp
17     ret     0
18 void div(void) ENDP
```



So, what do you do? So, let us look at the pointer arithmetic char star pA equals 0, int star pB equals 0 pA plus plus, pB plus plus So, there is a failure because of a compilation problem; div again must return a value. So, I will just make this function void for now . Yeah and of course, the optimized output is nothing because there is nothing to be done in this program.

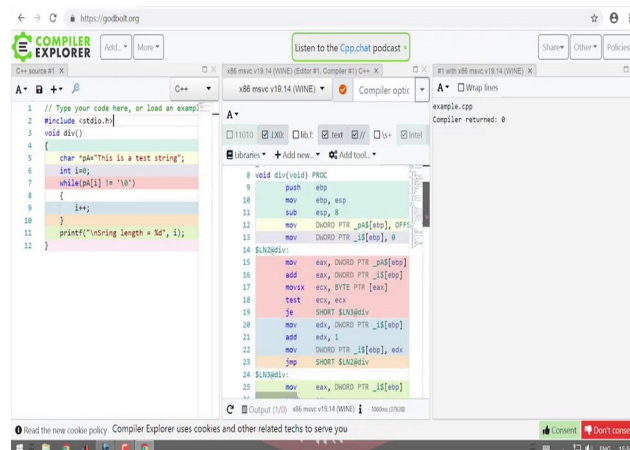
So, let us remove the optimization option here right, in the compiler option and what you see is that when I do pA plus plus the pink line on the left; the corresponding pink on the right is moving the value of you know ebp whatever pA into eax and adding 1 to it add eax comma 1 . And when you do pB plus plus it is moving the value into ecx and adding 4 to it .

(Refer Slide Time: 13:59)



Now, you could even try you know unsigned or short int star pC equals 0 and then do pC plus plus. So, what you see here is the green code which is pC plus plus, the addition is now a value of 2 because the pointer pC is pointing to a short int whose data width is 16 bits and not 32 . So, let us now go ahead and try the last example that we did which was basically the string length we will implement that in C directly.

(Refer Slide Time: 14:45)

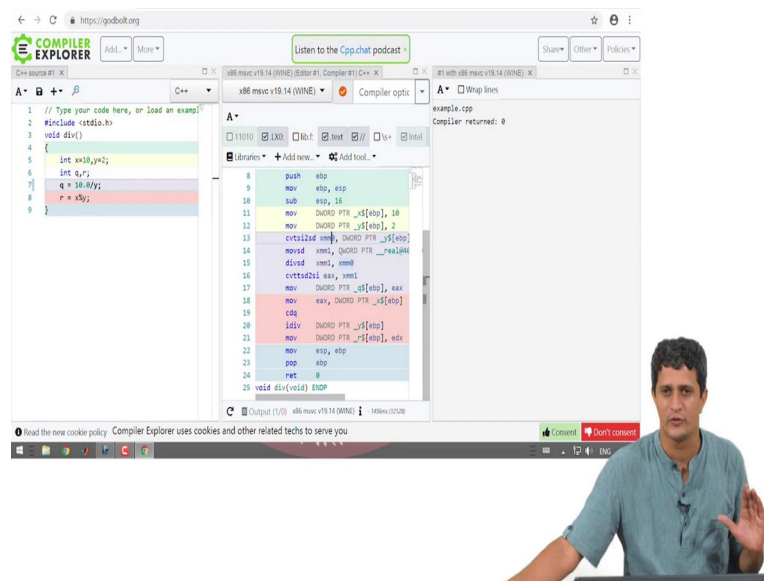


So, I have char star p test A equals; this is a test test string and while pA int i equals 0 while pA of i not equal to backslash 0, i plus plus and then I am saying printf quotient d

is i. Yeah and; obviously, it is going to say that printf is not found. So, unless you include hash hash include stdio dot h, it is not going to be able to find that particular function for you. The moment you include it, it will go ahead and compile the code. So, what you see now is that we implemented this string length function in a very compact way, but now you will see that when the C program is translated to assembly as is it gets translated to a whole lot of instructions and that is why it makes sense to optimize a lot of these C programs with sm in line assembly.

But remember that you should be able to do better than the optimized compiler output, you should not compare yourself with this unoptimized output because; obviously, we will be better than that. So, there are certain times when you can do even better than the optimized output and that is the single most useful value of being able to do inline assembly program.

(Refer Slide Time: 16:57)



So, the last example that I would like to discuss and wind up today's lecture is let us just go back to our int x equal to 10 and y equal to 2 and I will do div; I will do a simple division which is basically int q equal to q comma r and I will say q is equal to let me say 10 divided by y and r is equal to x percent y. So, what you see here is that q equal to 10 by y the gray line gets translated to an idiv operation. You move the value 10 into eax then you divide by y and the quotient is going to be int eax and you move that value into the variable q.

On the other hand; if I simply moved or changed this 10 to 10 dot 0 then what you see is a completely different output and this is what I had alluded to earlier where I had said that if you do floating point arithmetic, the processor sends these instructions to be evaluated outside into a coprocessor. So, what you see here is no longer and a simple idiv, it starts invoking some xmn0, xmn 1 and all that which are not even part of this processor. So, for the purpose of this course, we will only deal with integer arithmetic operations.

So, that we do not have to worry about these complex operations now. The concepts do not change whether you are doing floating point operations or you are doing integer operations, it only becomes more cumbersome when you deal with floating point numbers and therefore, we will avoid them in this course.

(Refer Slide Time: 18:55)

Topics Covered

- **Demo for using Compiler Explorer website**
- **Compiler Explorer**
 - <https://godbolt.org>
 - Author: Matt Godbolt
 - Open source free tool
 - Excellent to understand compiler output