

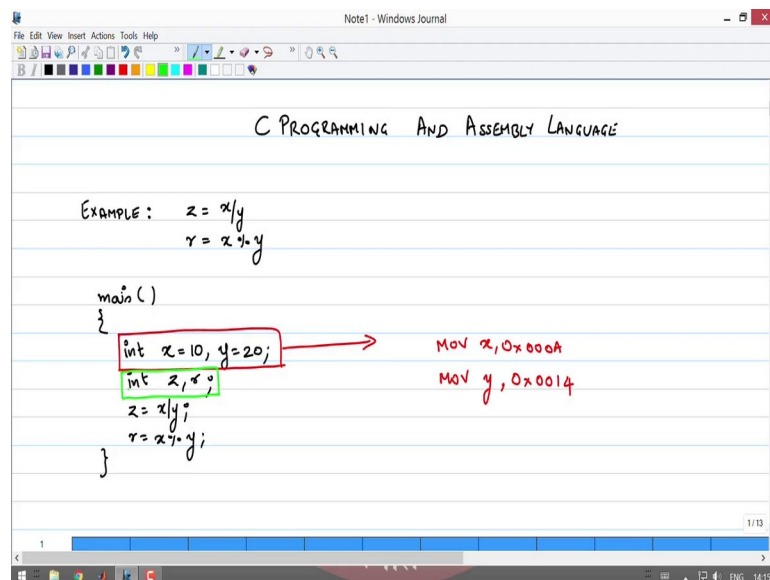
C Programming and Assembly language
Prof. Janakiraman Viraraghavan
Department of Electrical Engineering
Indian Institute of Technology, Madras

Lecture - 07

Welcome back to this course on C Programming and Assembly language. We are currently in module 2 which deals with C programming and inline assembly. In the last couple of lectures we discussed some elementary way of interspersing inline assembly instructions with C programs and we are discussed how to for example, evaluator and arithmetic expression in terms of variables and put them into registers. Then we later on implemented a multiplication program in assembly language using looping , using the jump on no zero kind of instructions.

So, let us processed in that direction and in this lecture, we will discuss some ideas on converting C programs to assembly language both in an optimized and un-optimized manner.

(Refer Slide Time: 01:03)



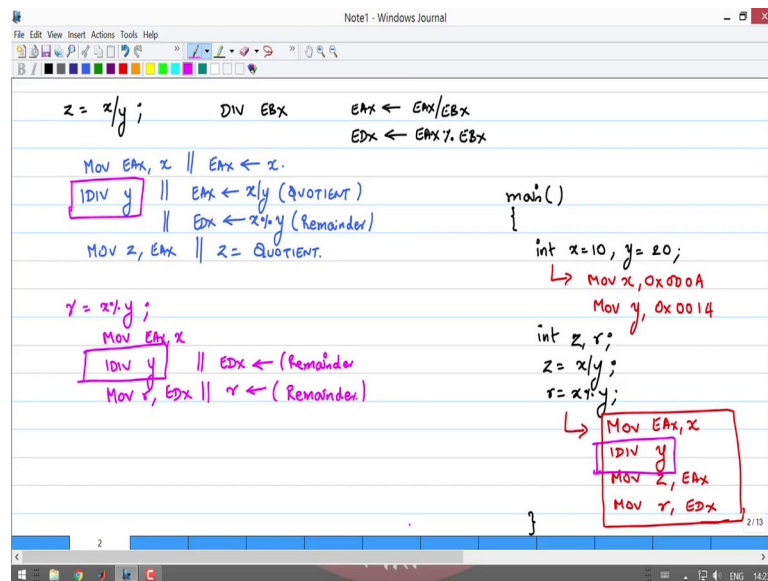
So, let us start with a very simple example. I want to divide two integers and put the value x by y in the variable z and the remainder x percent y into a variable called r and I want to implement this in a particular C program. So, now, let us look at the C implementation first and then we can go ahead and look at the assembly implementation.

So, the C implementation is as follows main and I have int x let us say equal to 10, y equal to 20 then int z comma r. So, how I do this? I would simply say z is equal to x by y and r is equal to x percent y. Of course, you could choose to print these values after this program, but let us leave out those aspects for now. So, let us first look at a very elementary way of implementing or translating this program into assembly language without any optimization.

So, therefore, we have we look at the following statements there is int x equal 10 and y equal to 20. So, this is basically a declaration of x and y as integers which are, is going to be 4 bytes ; 32 bits long, but we are also initializing these variables with values of 10 and 20. So, if you look at the assembly instruction, then this particular instruction would get translated into the following which is basically MOV x comma 0 x you know 000A . It is a 32 bit number, but I am leaving out the trailing zeros here and MOV y comma 0 x 00 so, 16 plus 4 yeah; this is basically 20 int hexadecimal.

Next there is a second statement which is basically int z comma r. So, here we are not initializing any of these instructions with or variables with any value and therefore, this will really not translate into any particular assembly implementation. Next we move on to the statement z equal to x by y .

(Refer Slide Time: 04:31)



So, we have the statement z is equal to x by y and both are integers. In order for the microprocessor to perform a division instruction just like multiplication, you have to use the EAX register implicitly .

So, just to remind you of what happens, you have to you can say `DIV EBX`, then what would happen is EAX would get the quotient by EBX and EDX would get the remainder EAX remainder EBX. So, in order to perform a division operation, I need to setup certain registers first. So, therefore, I go ahead and first `MOV EAX` with x right EAX simply gets the value x . Now I can go ahead and perform a division instruction on this EAX right. So, I can say `DIV` of y whatever this y is let us not worry about it for now ; it is an abstract entity for now where I have a variable in which I can read from or write to some particular value .

And because these are integers, note that the instruction that actually gets implemented in assembly has to be the instruction called `IDIV` which is basically integer division or signed division that has to be done . So, in the end of this particular operation, what would have happened is EAX would have got x by y QUOTIENT and EDX would have got x percent y remainder. So, now, I need to go ahead and initialize my z with the quotient and therefore, all I have to do is `MOV z comma EAX`; z is equal to QUOTIENT.

So, these an `MOV` on to the next assembly instruction the; next C instruction which is z is equal to x percent y . So, here again you have to do and other division operation in order to obtain the quotient and sorry this is not z ; this is r is equal to x percent y . So, note that this division operation or the remainder operation in C could have been done with any two other variables. So, typically if you look at an unoptimized compiler output, it would not worry about if these registers have changed or not from one C instruction to the next. And therefore, it would go ahead and setup this division much the same way as it did in the previous C instruction.

So, it would repeat exactly the same thing `EAX comma x` , then it would do `IDIV y` and the result of this is that EDX will now have the instruction or the value of the remainder. Now on finishing this instruction all I have to do is `MOV r comma EDX`. So, this is nothing, but the variable r getting the remainder.

So, what you notice here is that there is a lot of redundancy in the instructions that are generated for an unoptimized compiler output. For example, you see very clearly that when I finish my operation here EAX and you know load EAX with x and IDIV with y, EDX has already been evaluated and hence there is no real need for me to go ahead and perform this second division operation. So, that is the crux of unoptimized compiler output and optimized compiler output. So, if we were to go ahead and do this particular seek take translate this particular C program into assembly, then this is what we would have done .

So, we have int x equal to 10, y equal to 20 . This would simply translate to two instructions MOV x comma 0x00 MOV y comma 0x0014 , and then there is the next instruction which is int z comma r which basically translates to nothing in assembly. Then the z is equal to x by y and r is equal to x percent y you would simply translate to the following instructions which is MOV EAX comma x IDIV y, MOV z comma EAX, MOV r comma EDX.

And so, what we are effectively doing is we have only one set of division operation that is being done as shown here. On the other hand, the unoptimized compiler output has two division operations that are being done and the reason for that is, you are just taking line by line and just translating it blindly into the necessary assembly that will result in the correct output. We are not trying to keep track of the contents of the registers between two instructions.

Let us now consider another example in order to look at this concept of optimized and unoptimized compiler output.

(Refer Slide Time: 12:35)

```
main()
{
    int a, b, c, d;
    int x=10, y=5;
    a=x+y;
    b=a-y;
    c=b*y;
    d=c/y; // Quotient
}
```

```
Mov x, 0x000A
Mov y, 0x0005
Mov Eax, x // Eax ← x
Add Eax, y // Eax ← x+y
Mov a, Eax // a ← x+y
Mov Ebx, a // Ebx ← a
Sub Ebx, y // Ebx ← a-y
Mov b, Ebx // b ← a-y
Mov Ebx, b
Imul y
Mov c, Eax
```

So, let us consider this C program main have int a, b, c, d and int x equals 10 and y equals 5. So, let us say I want to do the following a equals x plus y, b equals a minus y, c equals b into y and d equals c by y this is the quotient by the way ; just to remind you this is the because we are talking about integer arithmetic here.

So, let us look at an unoptimized compiler output first which basically is simply going to take line by line of the C program and just translate it to its corresponding assembly output . So, you take this statement because there is an assignment happening. This will simply translate to MOV x comma 0x000A and this the other one will translate to MOV y comma 0x0005.

So, now let us look at the first evaluation that we want to do which is basically a equals x plus y. I want to now look at the assembly output of this block. So, in order to perform any ALU operation, I need to MOV certain operands into registers. So, therefore, let me arbitrarily pick EAX as my register and I am going to MOV EAX comma x . So, what is this instruction do? EAX simply gets the value x and then I am going to perform an addition on this which is ADD EAX comma y.

So, what is this two EAX is simply going to be EAX or that already has x so, it will be x plus y. Now I need to MOV this sum into my variable a so, I will do MOV a comma EAX. So, a will get x plus y here. Now let us look at the next C instruction which is b equal to a minus y. So, I want to just blindly translate this into some setup assembly

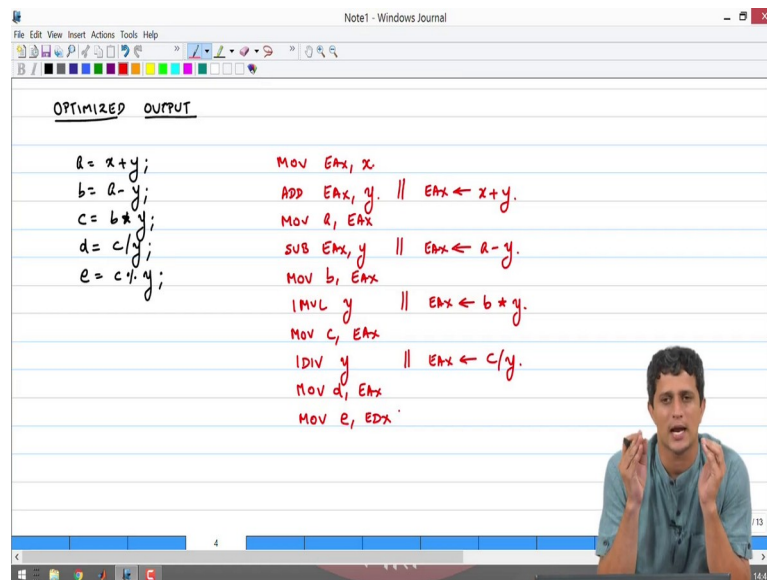
instructions which are correct. So, what do we do? We MOV again this instruction; this time let me pick $b = EBX - a$ and then I will directly do $a - EBX$ and then MOV $b = EBX$.

So, what am I doing here? I am loading EBX with a , I am loading EBX with $a - y$ and here I am saying $b = a - y$. Similarly if you go ahead and translate the next instruction which is basically $c = b - y$; I mean I have to store one of the operand in EAX. So, therefore, I will say MOV EAX b and I will simply say IMUL y . And in the end of this instruction, I could simply say; let me just put some space here; MOV $c = EAX$.

Similarly, you can do it for the fourth division operation as well. So, the key point here is that if we just blindly translate every C instruction into assembly, what you will get is a very unoptimized output in the sense that there are lot of MOV instructions which are severely redundant in this process. For example, if you look at this particular instruction MOV EBX a , it is totally unnecessary because EAX already has the right value in the previous step.

So, if you look at this guy, the previous step already has the value of a in EAX. So, without loading anything into EBX, you can simply subtract directly from EAX and obtained the value of b . Now similarly if you look at this particular operation now, it is totally redundant because we are unnecessarily loading the value of b again into this EAX register and so on. So, let us now look at what a what an optimized output is lie by considering what the contents of the register a was in the previous instruction.

(Refer Slide Time: 19:07)



```
OPTIMIZED OUTPUT

a = x + y;      MOV EAX, x
b = a - y;      ADD EAX, y  || EAX ← x + y.
c = b * y;      MOV E, EAX
d = c / y;      SUB EAX, y  || EAX ← a - y.
e = c % y;      MOV b, EAX
                IMUL y   || EAX ← b * y.
                MOV c, EAX
                IDIV y   || EAX ← c / y.
                MOV d, EAX
                MOV e, EDI
```

So, we will now look at optimized output. So, what I want to do is x plus y , b equals a minus y , c equals b into y and d equals c by y . So, what I am going to do is because the multiplication and division typically needs EAX I am going to simply deal only with EAX as my primary operand throughout. So, what do I do? I first `MOV EAX, x` and I simply do and `ADD EAX, y`. So, this is EAX is x plus y . Now since my EAX already has the sum x plus y or the variable a , I do not need to `MOV` it explicitly into another register before starting the next instruction. So, what I will just do here is, I will just `MOV` my EAX into the variable a .

But EAX still has the sum x plus y and therefore, now I can simply go ahead and do subtract EAX comma y . So, what does this do? It simply does EAX is a minus y and now I just have to `MOV` this result into the variable b . Now similarly I go ahead and perform my multiplication operation directly here on let me call it `IMUL` of y . So, what is this? This is EAX is what it has the value b into y and of course, I `MOV` the value from EAX into the variable c .

Now, I go ahead and perform my division as well because the value of c is already in EAX now; `IDIV y` which implies EAX now has the value c by y and of course, I `MOV` the quotient into the variable d and the quotient is available in EAX . Now note that I could for example, add another thing which is c percent y which implies without adding any new instruction, I can simply go ahead and let me not call this d ; I will call this e

another variable and I am I will simply add another instruction here; MOV e comma EDX.

So, what you see here is we have got rid of so many redundant MOV operations in order to optimize this assembly code. So, you must realize that unless you use a compiler in a particular optimized mode, it may not be possible to always get the most optimized and quickest instruction set in the assembly level.

(Refer Slide Time: 23:27)

Topics Covered

- **Un-optimized compiler output**
- **Optimized compiler output**
 - Keeping track of register values across instructions