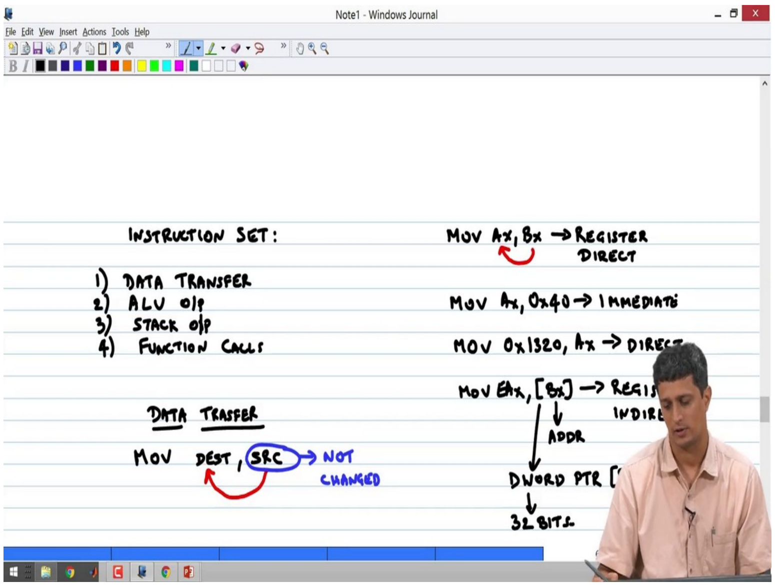
C Programming and Assembly language

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

Lecture - 03

So, welcome back to this course on C Programming and Assembly language. So, with that we introduced all the necessary registers in order for the microprocessor to go ahead and do the fetch, decode and execute operation again and again. So, with that let us move on to what are the typical instructions and how are certain instructions executed in x 86 microprocessor, . So, the instruction set is what we are going to look at in this lecture.

(Refer Slide Time: 00:42)



INSTRUCTION SET:

1) DATA TRANSFER
2) ALU O/P
3) STACK O/P
4) FUNCTION CALLS

DATA TRASFER

MOV DEST, SRC → NOT CHANGED

MOV AX, BX → REGISTER DIRECT

MOV AX, 0x40 → IMMEDIATE

MOV 0x1320, AX → DIREC

MOV EAX, [BX] → REGIS INDIRE

ADDR

DWORD PTR [

32 BITS

So, there are different kinds of instructions that one would want to execute when we do this final execute operation, . The third step of every instruction which is the fetch, decode and execute. So, one could be that you want to just move data. So, you could do data transfer, . Then you could do what is known as a an ALU operation, , arithmetic or a logical ALU related operation. Then you could do for example a stack operation, which effectively is still a sort of a data transfer, stack operation. You could call or return a function, you want to do, .

So, the word function as used in C is not the same as used in assembly language, there therefore, maybe this is better to call it as a subroutine, . A function in assembly language is called a subroutine. So, there are many more such instructions which we will not go in detail, we will only look at the related operations as I mentioned earlier that are useful for us in C programming. So, let us start with the data transfer.

So, the idea is to actually you know let us go back to our fetch, decode and execute loop, . So, the first step might be for us to actually fetch some data, . It could be from a memory of course, that for the instruction pointer it is very clear it has to be from the memory, and that is a data transfer from the external memory into the microprocessor. But you could also do in a data transfer inside the microprocessor. So, in general, data transfer instructions are actually denoted by MOV destination, destination comma source. So, this is again very specific. This format is very specific to the Intel architecture. It can be different in other microprocessor, but does not matter.

So, data from the source is moved to the destination, and the source is unaltered, . So, this remains not, it does not change. . So, you may want to for example move data from a one register to an other register, right. You may want to say MOV Ax comma Bx, which means that I am actually moving my data from register Bx into Ax which is 16 bits long, and Bx is going to be unaltered in the process. So, the direction of data movement is right to left in the Intel architecture, .

So, before the go forward note that I have already started using some key words like a MOV and so on, . So, these keywords actually are known as mnemonics in assembly language. So, of course, ultimately in the microprocessor everything is stored in as bytes and bits, and information is decoded out of that. Mnemonics are only sort of a slightly high level translation for us to understand and process easily. What is the difference between a mnemonic and a high level language programming statement?

So, the fundamental difference is a high level language program statement can be foremore sophisticated, you can do more complex things of course, but a high level programming language statement can translate to multiple machine level instructions. On the other hand, a mnemonic in assembly language strictly translates to only one machine code even at the microprocessor level.

So, though this is still an abstraction a slightly high level language for us, the only difference is that it has a one to one mapping with the machine code that is eventually executed in the microprocessor, unlike a high level language statement, . So, this is only one example that we wanted to deal with you move data from one register to another register. Of course, this could be MOV Cx comma Dx or whatever, .

So, I could also move data, I could move a constant into my register which is given by you know MOV Ax comma 0x40 which means the hexadecimal 0x40 is loaded into my Ax register, . So, this is called register direct addressing. This is called immediate addressing, which means that the actual data that I want to load into a register is available immediately after my app code, right. It is part of the machine instruction that is loaded into memory.

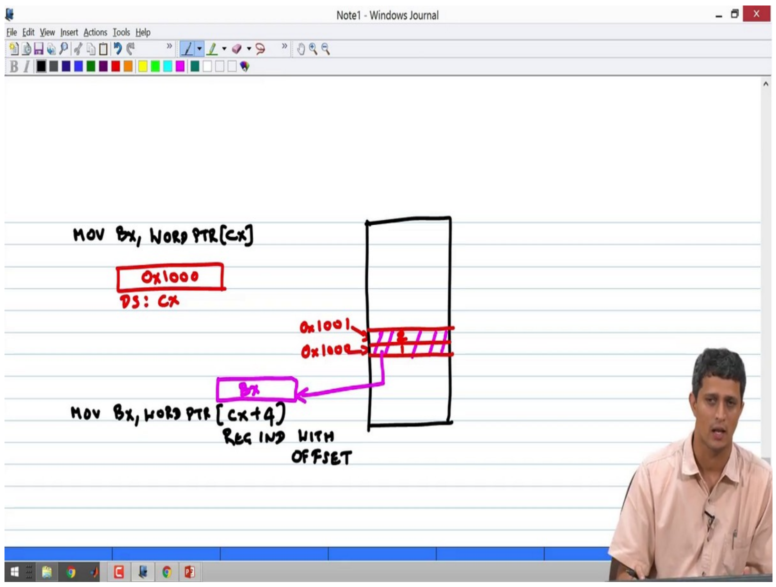
Then I could, for example, load contents from the memory, . So, I could for example, say I want to move the contents that is in location or the contents of Ax into location 1320, 0 x hexadecimal 1320. I want to move the contents of Ax. So, this is called direct addressing. What this mean is that contents of Ax are actually loaded into the memory location given by 0x 1320, right this is the hexadecimal number and that is the address and that location basically two 16 bits starting at 0x1320 will be loaded with the contents of Ax. So, this is called direct addressing.

And I could do register indirect addressing which basically means that the address that I want to deal with, in memory is actually sitting in a register and I am going to you know load the contents of a microprocessor register into that particular address. For example, I can say MOV Ax comma right Bx. What does this do? It treats Bx as an address and whatever data is present in that particular address will be loaded into my Ax register, .

So, now, it bring us to an interesting point, . Like I said previously logical address we always say we are going to deal in chunks of 8 bits, . But is it that I always want to deal with 8 bits. Well no, I might want to deal with 8 bits at one time, I might want to deal with 16 bits in sometime and maybe 32 bits at other times. So, we need a way to actually tell the microprocessor that I need you to fetch not just 8 bits, but I need you to fetch 16 bits or even 32 bits from starting at that location, . So, that is actually given by a specification which is you know instead of just Bx I would say DWORD pointer Bx. What is this mean? It means that whatever is in Bx will be treated as an address and the next 4 locations DWORD stands for double word, right which means that this is 32 bits.

So, whatever is in this location that is pointed to by Bx, I will load 4 of those locations into my of course, I cannot have Ax then I should have EAx because I need 32 bits, . So, whatever is in those 4 locations starting at the location pointed to by Bx, 4 of those bytes will be loaded into my EAX register in some particular order. That order let us not go into the details it does not matter to us, . The main thing I want to convey here is that we may want to deal with a byte, a word or a DWORD, or a double word.

(Refer Slide Time: 11:35)



So, if I want to load only 16 bits, for example, if I want to load only 16 bits from an address that is pointed to by Cx, into register Bx. I would say MOV Bx comma word pointer Cx, . What does this mean? It means that whatever is in Cx, let us look at the concrete example here I have Cx register, there is some let us say that the data let us gives a concrete value 0x1000, , and this is my Cx register.

I want to load 16 bits of information from the memory, this is my memory and this is my location t 0x1000, . So, this is byte 1 and the other one is byte 2 which is basically location 0x1001. So, when I say MOV word pointer contents of Cx, it means whatever is in Cx that location will actually be accessed and two of those bytes will be moved into Bx, . So, the contents here will be moved to my Bx register.

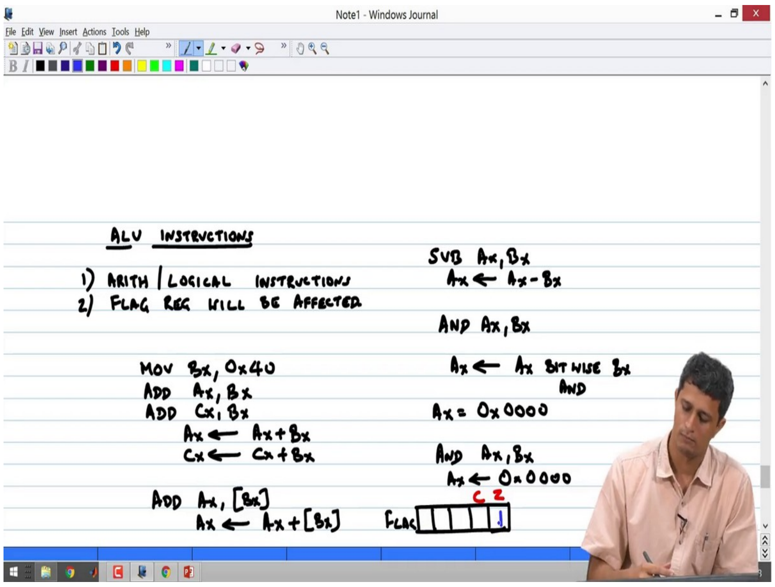
Also note that when I say that contents pointed to by Cx I am referring to the memory which means that I also need a segment register to tell me which segment to access this data from, Cx only gives me the offset and therefore, by default Cx you know all these general purpose registers refer to the data segment. So, this is actually not just Cx, it is going to be pointed to as data segment:Cx, which means it is an offset given by the value Cx from data segment value that is given in the ds register, .

So, again you can look up the book for more details, but the general idea is that you can move either data between registers or you can move data between you know you can move immediate data to a register, you can directly access the memory or you can indirectly access the memory through a register. So, this is register indirect addressing. If I want to go back here I would call this as register indirect.

Now, I can actually do more than this. I can actually do some computations also. For example, I could Mov data into Bx, wordpointer(Cx+ 4) I can specify an offset now . So, you should not confuse that Cx plus 4 is actually evaluated using an other setup assembly instructions. For example, it is not that I load some value into Cx and then do an arithmetic addition using the add instruction of 4 and then calculate what that new address is and then do this calculation. All of these is enabled in hardware, .

So, this is register indirect, register indirect with offset, . You can actually do more than this you can even do certain complicated things like you know do a scaled value 4 times Cx+4 and so on. I urge you to refer the book, but as far as we are concerned for this course it is sufficient to know these 4 addressing schemes which is basically register direct, immediate, direct addressing, and register indirect with offset. So, these 4 modes of addressing will cover the needs for this course. So, with that we can move on to the next class of instructions which are basically the ALU instructions, right.

(Refer Slide Time: 16:45)



ALU INSTRUCTIONS

1) ARITH / LOGICAL INSTRUCTIONS
2) FLAG REG WILL BE AFFECTED

```
MOV Bx, 0x40
ADD Ax, Bx
ADD Cx, Bx
    Ax ← Ax+Bx
    Cx ← Cx+Bx

ADD Ax, [Bx]
    Ax ← Ax+[Bx]

SUB Ax, Bx
  Ax ← Ax-Bx

AND Ax, Bx

  Ax ← Ax BIT WISE AND Bx

Ax = 0x0000

AND Ax, Bx
  Ax ← 0x0000
```

So, as the name suggest here the idea is to do arithmetic and logical instructions. So, what are the kind of instructions that you do? You do arithmetic or logical instructions. The key thing is that with every instruction that is executed here the flag register will be affected.

So, for example I might want to do addition of two simple numbers, . So, I can load for example, MOV my Bx register with a number 0x40 and then I would like to ADD this to my Ax register, so Ax, Bx. So, what is this do? As a name suggests, it is going to do an addition it is going to replace Ax register with Ax+ Bx, .

So, it is not necessary that Ax always has to be an operant to the ADD instruction, you could have any two register registers doing this addition for you, . So, for example, I could even do ADD Cx, Bx which implies Cx will get Cx+Bx, . And remember here I could extend this addressing scheme that I spoke about previously, even to this addition operation as well. For example, I could do ADD Ax comma contents pointed to by Bx, right which means that whatever data sits in Bx, will be brought on into the microprocessor and added with Ax, .

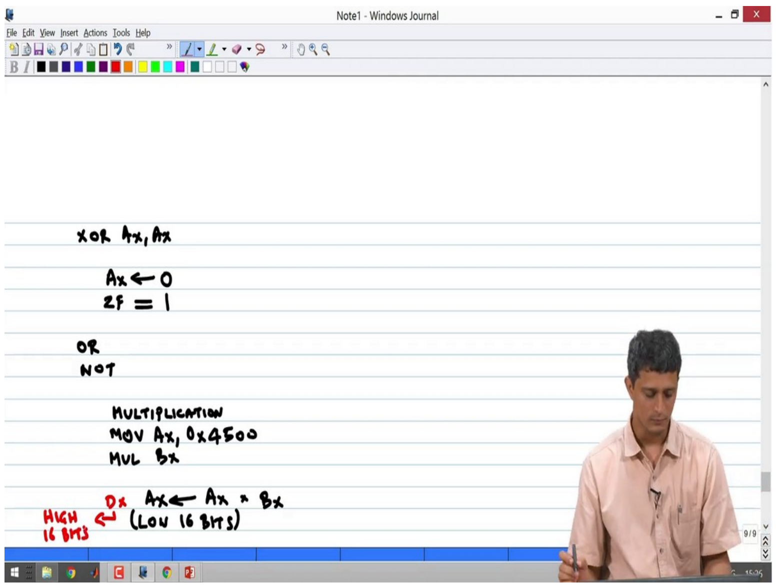
So, what does this mean? This is Ax gets replaced with Ax plus contents pointed to by Bx, . So, you could do this various addressing schemes here as well and get similar results. So, I am not going into the gory details the by way of some examples and assignments we will cover the necessary details that we need out of these instructions.

So, let us look at for example, you know you could also do subtraction, Ax comma Bx, which is basically Ax is replaced by Ax- Bx, . You could do logical bitwise XOR, right or an AND operation let us say, AND Ax comma Bx. What does this mean? It means that Ax is simply replaced with a bitwise and BIT WISE AND with Bx, .

So, let us look at what happens to the flag register for example in this process. If Bx happens to be 0, or Ax happens to be 0 in this AND instruction then the result of this AND Ax Bx is going to be a 0, . Let us assume that Ax happen to have this number 4, 16 bit hexadecimal all 0s. That means, that the AND operation Ax comma Bx, if you execute this irrespective of what is there in Bx, Ax will now have 0. So, when this instruction is executed and the result happens to be 0 the flag register will actually set the 0 bit to high which means in the flag register, there are various features that you can you used to indicate which like I mentioned the 0, the carry and so on.

Let us assume that this actually refers to the 0 bit this is the carry and so on. Then this particular bit will be set to 1; that is what it means, that is what it means to say that the flag is affected with very single arithmetic or logical operation. So, what it means is that I can now make some sort of a decision based on this flag register and we will come to that later which is basically known as branching and looping instructions. So, we will come to that.

(Refer Slide Time: 22:42)



XOR Ax, Ax

Ax ← 0
ZF = 1

OR
NOT

MULTIPLICATION
MOV Ax, 0x4500
MUL Bx

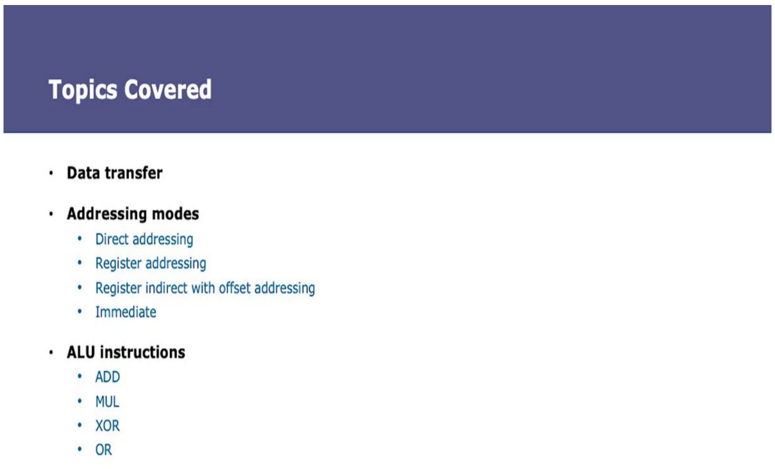
Dx Ax ← Ax * Bx
HIGH 16 BITS ← (LOW 16 BITS)

Similarly, you know if I wanted to clear any register then I would simply do XOR Ax comma Ax. This is the simplest and fastest way to clear any register, because it what is what it says is that any irrespective of what Ax had before executing this instruction if you XOR with itself bitwise XOR, the result has to be 0 which implies that after this instruction Ax will necessarily become 0. And of course, the 0 flag is set.

So, there are other logical operations that you can do you like the OR, which again is bitwise or operation. So, you can do XOR, you can do XNOR, then you can do the NOT, logical NOT operation which means you are just inverting, a particular register. And on the arithmetic side you can do the addition, subtraction, you can also do MULTIPLICATION. Unlike the addition or subtraction operation where you are free to give any two registers, the multiplication works only with Ax as one of the operants. So, if you want to multiply two numbers, you have to move one of the operants into Ax the other operant into any other register and then perform it.

So, if I want to multiply a Ax and Bx, then I would have to MOV Ax comma let us say I am going to load 0x4500, 16 bit number, I want to now multiply with Bx. So, this here the second operant is implicitly assumed to be the A register. So, you can multiply with Bx and the higher order result, , when you multiply two numbers; obviously, 16 bit numbers you are not going to get another 16 bit number, you need more than 16 bits. So, the higher 16 bits is made available in the Bx register. So, if you look at this what it basically does is Ax will be Ax* Bx and Ax stores the low 16 bits, Dx will store the higher 16 bits.

(Refer Slide Time: 25:53)



Topics Covered

- **Data transfer**
- **Addressing modes**
  - Direct addressing
  - Register addressing
  - Register indirect with offset addressing
  - Immediate
- **ALU instructions**
  - ADD
  - MUL
  - XOR
  - OR