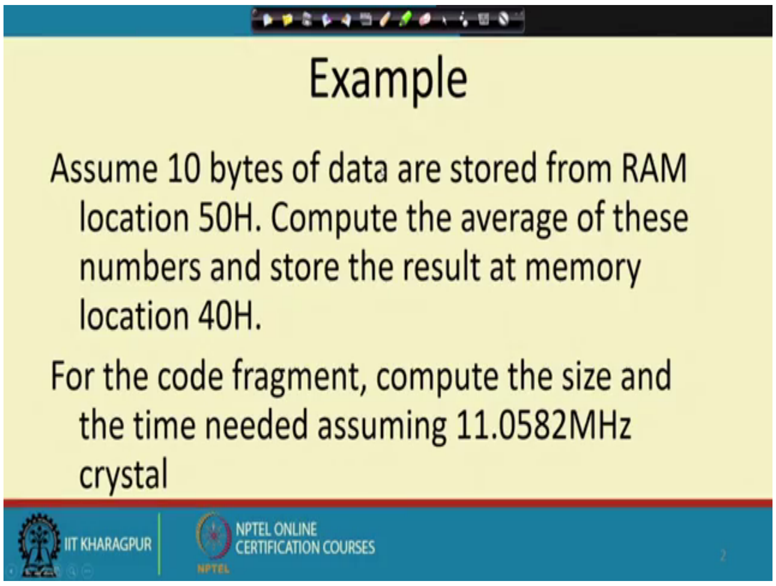
Microprocessors and Microcontrollers

Prof. Santanu Chattopadhyay
Department of Electrical & Electronics and Communication Engineering
Indian Institute of Technology, Kharagpur

Lecture – 36
8051 Programming Examples

So, next we will look into a few programming examples for this 8051 system. So, 8051 microcontroller we have seen that it; it can do many instructions and it can we can use it for designing many embedded applications. So, we will see some programming examples of 8051 to get a better understanding of how this system can be made into use.

(Refer Slide Time: 00:43)



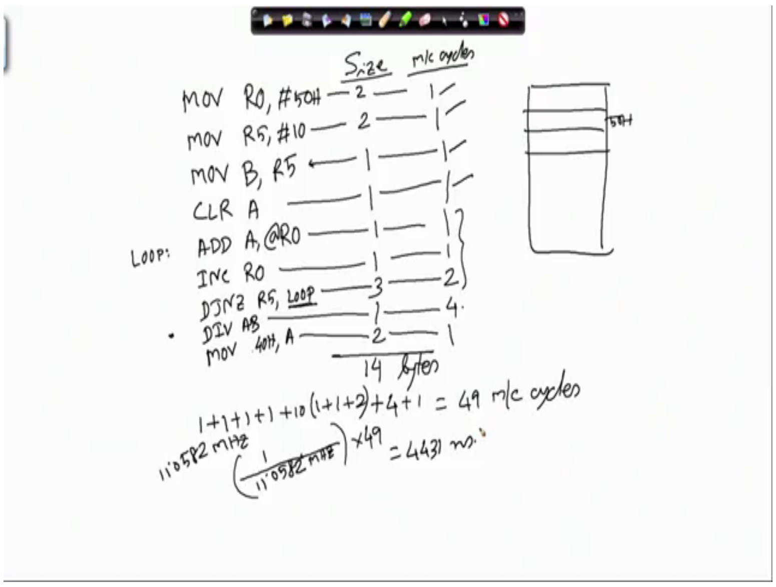
Example

Assume 10 bytes of data are stored from RAM location 50H. Compute the average of these numbers and store the result at memory location 40H.

For the code fragment, compute the size and the time needed assuming 11.0582MHz crystal

So, first example that we will look into is the like this that we assume that there are 10 bytes of data that are stored from RAM location 50 h. And then we want to compute the average of these numbers and store the result in the memory location 40 h. And then for the code fragment will try to see what is the size of the program and what is the time needed for its execution; assuming that the crystal that we have is 11.0582 megahertz ok. So, the program that will try to write is like this. So, it will be having this 50 it will be having this 50 h number.

(Refer Slide Time: 01:32)



```
                      Size   m/c cycles
      MOV R0, #50H  —  2  —  1
      mov R5, #10   —  2  —  1
      mov B, R5     —  1  —  1
      CLR A         —  1  —  1
LOOP: ADD A, @R0    —  1  —  1
      INC R0        —  1  —  1
      DJNZ R5, LOOP —  3  —  2
      DIV AB        —  1  —  4
      mov 40H, A    —  2  —  1
                      14 bytes
```

$1+1+1+1+10(1+1+2)+4+1 = 49$ m/c cycles

11.0592 MHz

$\left(\frac{1}{11.0592\text{ MHz}}\right) \times 49 = 4.431$ ms

So, that it is the memory location from the RAM location 50 h the numbers are stored; from RAM location 50 h onwards the numbers are stored for this is 50 h. So, what we do first in the R0 register we move the value 50 h because this R0 will be using as the index to access this 50 that this memory location. So, first in the R0 register we move in the value 50 h and there are 10 numbers and that count to you move into the register R5. So, MOV R5 comma hash 10. So, since this is a decimal number 10. So, you do not put a h there. So, just write hash 10.

Then we say like MOV B comma R5 just to keep a count that it is in the B register we are we will be using that value for implementing and all; then with count value for averaging purpose. So, this will be necessary; so, you just store this in the B register then this accumulator register will be accumulating the sum. So, for that purpose we clear this A register; so, that it is cleared now.

Then I have to add this successive values to the A register. So, for that purpose; so, I can use this instruction add A comma at the rate R0. So, whatever be the content of the first location that will be added with A register and A register content is previously 0. So, with that if R0 is added to it will be the first number will get added then I need to R0 should be incremented.

So, that it points to the next register next number increment R0 between point to the next register next data. And then in R5 we have got that count 10 ok; so, the following just do a decrement jump not 0 decrement jump not 0 R5 comma; so, a loop a loop is this instruction where I am getting it from the memory location and adding it at comma at the rate R0; so, this instruction there is the loop.

So, once we are at this point; so, our job is over as far as the addition is concerned. The next thing that I have to do is to compute the sum of the average of these numbers and in the B register I already have a kept this account 10 ok. So, I have to just divide it a by B; A resistor has got the sum B register has got the number of numbers. So, I said DIV AB ok.

The DIV AB instruction; so, this will be having this a divided by B and then the value that I get in the A registered is the quotient that is the average value. And that average value you can move to the memory location say 40 h comma A. So, in the memory location 40 h we are moving the sorry this is not hash should not be there because that in that case this we mean that this is an immediate value. So, here trying to say that this is the memory address; so, MOV 40 h 40 h comma A. So, this can be used for moving the value of A onto memory location 40 h.

Now, so this program does this now if I want to compute what is the size of this program in terms of number of bytes what is the total size of this program. So, the first instruction; so, this takes 2 bytes. So, for that purpose you need to consult the manual of 8051, but essentially one thing we should understand that every instruction there will be an opcode which will be taking 1 byte and another byte will be required for holding the immediate value.

So, that is this has got this is 2 byte. So, this is also 2 byte then the next one MOV B comma R5 this is1 byte here A is also1 byte MOV A comma at the rate R0. So, this is also 1 byte; so, this is also 1 byte then this DJNZ instructions. So, this is a bit complex because apart from this op-code I have to tell this loop address also and loop address since this is a 16 bit value. So, it will be a 3 byte instruction. So, one will be holding the op code for DJNZ R5 and the other 2 bytes this they will hold the address of this loop.

So, there is this level loop; so, that address is 3 byte 2 byte. So, total instruction is 3 byte then this DIV AB is again1 byte and this instruction is 2 byte because it has got and intermediate value 40. So, that will take1 byte and this move A; so, that is taking 1 byte if you sum them up then this number turns out to be 14. So, total size of this program is 14 bytes fine.

The next that you may be interested to calculate is what is the speed; what is the time required for executing this instruction? So, for that purpose you need to find out the time needed for executing each of these instructions. So, if you consult the manual then you can find that the machine cycles needed for various instructions for the first instruction. So, this is this will take 1 machine cycle this is also take 1 machine cycle.

So, this is also take 1 machine cycle; so, this DIV AB it takes 4 machine cycle and apart from that. So, this and these DJNZ; so, this requires 2 machine cycles apart from that if you look into the manual of 8051, you will find that all others they will take only 1 machine cycles all of them take 1 machine cycle.

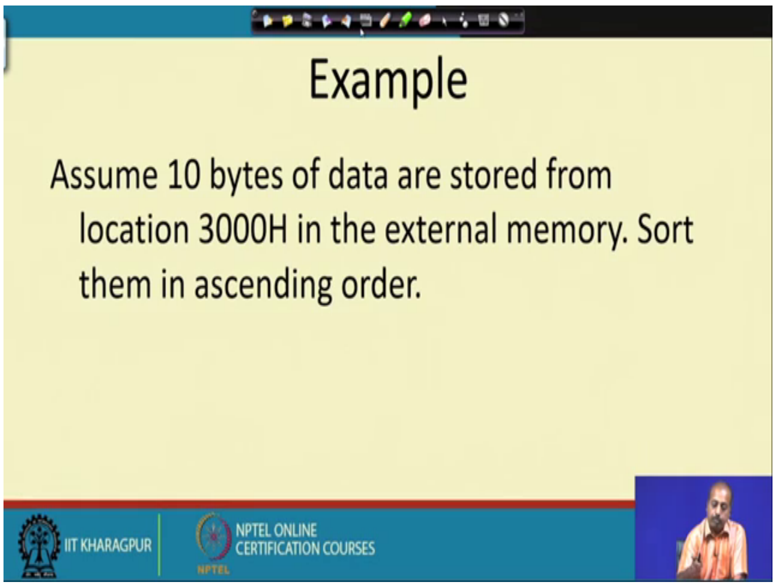
So, if you are not just trying to compute what is the total number of machine cycles needed. So, it is 1 plus; so, this is one then this one 1 plus 1 then this one then this one. Now the next instructions like 1, 1 and 2; so, they are repeated 10 times. So, next is 10 into 1 plus 1 plus 2 then it takes 4 machine cycles this one and this takes 1 machine cycles; then if you sum them then this whole thing that turns out to be 49 machine cycle it is 49 machine cycles. And as said that that clock frequency is taken pro to be 11; 0582 megahertz.

So, 1 machine cycle is 1 machine cycle is 1 upon 11.0582; so this 1 point 1 upon 11.0582 megahertz. So, this number multiplied by 49 such clock cycles; so, this time turns out to be if you calculate. So, this is about 4431 nanosecond.

So, this way in an assembly language program; so, you can compute how much memory it will take and how much time it will take? So, that is a very well; so, that is the very strong point of assembly language programming because here you can you can tell exactly how much time will be needed for executive the program and how much space the program will take whereas, if you are writing some program in high level language; so, it is you cannot tell it correctly.

So, I have to first compile the program and it depends on the compiler like how much time it will take for doing those things ok; for how the code will be translated and all that. So, that is why for getting the most predictable operation; so, we should go for this assembly language programming. So, next we will look into another example program; so, which is slightly more complex.

(Refer Slide Time: 10:35)



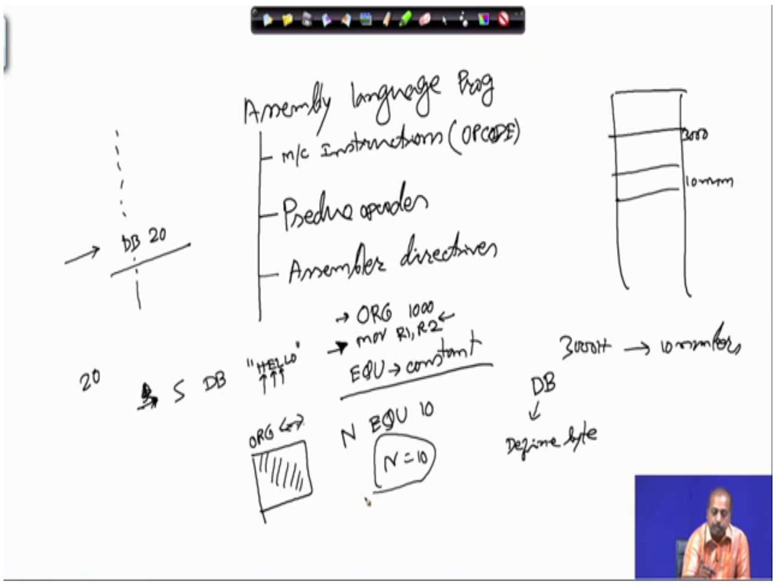
Example

Assume 10 bytes of data are stored from location 3000H in the external memory. Sort them in ascending order.

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES

So, I assume that there are 10 bytes of data stored from memory location 3000 h in the external memory and we want to sort them in ascending order ok. So, we want to do that now how to do this type of program?

(Refer Slide Time: 11:00)



So, for doing this type of programs; so, so it is starting at memory location 3000 h from memory location 3000 h, the number are stored and there are 10 such numbers. So, it is said that there are 10 such numbers stored 10 numbers ok.

So, first of all for writing this type of complete programs; so, will do it like this. So, we will introduce some special notations which are known as assembler directives ok. So, while writing any assembly language program let me just introduce just those things first like if you look into any assembly language program then you can find several components in it assembly language program. So, you can find that there are components in the first component first type of things that you can find out are the machine instructions.

So, this is actually the instructions that that processor supports like MOV ADD SUB JUMP. So, all those is the they are the machine instructions. So, apart from that you will find some more and they are often called op codes there they are the operation codes. So, this that is called op code or machine op code then you can find another type of entries which are known as pseudo op codes. So, peuso op code they mean that they are actually not for execution, but they are required for making the program structure better. And if they are useful for defining particularly the variables like say this particular case. So, you want to have this thing that from location 3000 h from location 3000 h I will have 10 numbers as I said.

So, you can if you want to initialize this thing. So, that the file executing this program we are assuming that those 3 locations 3000 h the values are already there ok. So, you can do that by putting a one particular pseudo op code which is known as D B D B stands for define byte. So, if this defined byte, so if you know in your program somewhere if you write like say DB 20. So, what it means what the assembler will do is that whatever be the address at this point at that address it will put the value 20.

So, that what is the meaning of it is up to the user definitely, but it will put the value there. So, that way if you have to define a string; so, you can define a string like say I can say like S DB hello. So, I can define a sting like this. So, what the assembler will do whatever you the address at which it find this finds it finds that it should be allocated to. So, it will; so, in the first character in first location it will put H second location it will put a third location it will put L; so, it will do like that. So, it will initialize a portion of memory with this particular string ok; so these are the pseudo op code. So, we will be using them and another category of words that you can find they corresponds assembler directives.

So, this assembler directives are to for telling assembler what to do. what the user is looking for what to do? Like one assembler directive is as is say O R G. So, if ORG is followed by some address. So, if I say ORG 1000; that means, after that if I have MOV instruction MOV say R1 comma R 2; that means, this instruction the code for this I want to put at memory location 1000. So, when the instruction the program will be loaded this instruction will be loaded into the memory location 1000.

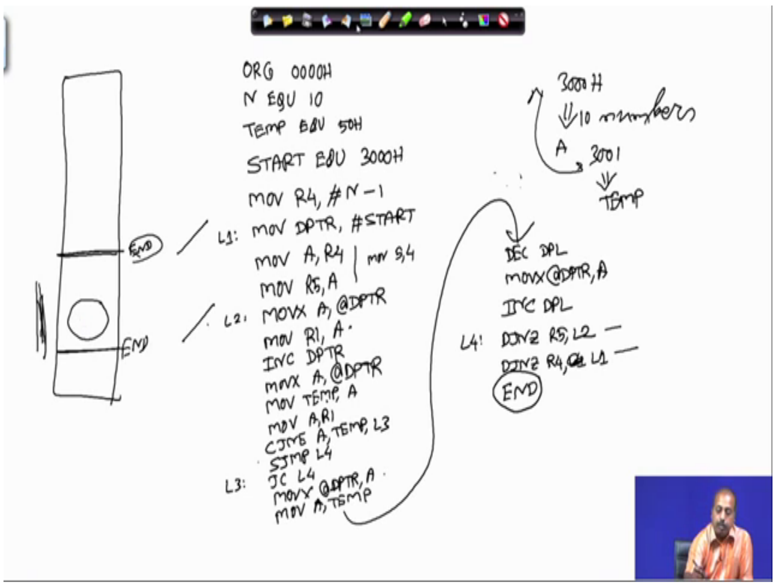
So, if you know the address at which your program should be loaded. So, you can put this org statement in the in your program to tell that the your program translated code should assumed that the origin of the code is that particular address; this helps in the jump instructions and all because we need to the actual target address.

So, if the org is there at the beginning; so, you know that their program the first statement of the program is starting from that address. Or sometimes we have seen that we have got interrupt service routines; now interrupt service routines they are they have to be loaded from the particular address. So, before writing that interrupt service routine. So, you can put an ORG statement here. So, that way and whatever we may target address here that value can be stated here.

So, that this interrupt service routine when it translated; so, it is put into the exact interrupt vector or the ISR address whatever is required for that. So, that way we have got assembler directives and there are many such assembler directives say. So, one of them is org and another one that you have is E QU equality. So, this is something like some constant definitions; so, some constant definitions.

So, you can define something like in high level programs you have like C C plus plus you are writing like hash define X 5. So, 5 the x is a constant whose value is 5; so, here also you. So, I can have this EQU type of definition. So, like say I can have say N EQU 10. So, N N is a constant whose value is 10; so,. So, unlike this D B; this N is not assigned any memory location. So, in the assemble code you will never find this N having the as a location, but where ever in the program this N will occur. So, it will be replaced by 10 in that program.

(Refer Slide Time: 11:36)



```
      ORG 0000H
      N EQU 10
      TEMP EQU 50H
      START EQU 3000H
      mov R4, #N-1
L1:   mov DPTR, #START
      mov A, R4   | mov 5,4
      mov R5, A
L2:   MOVX A, @DPTR
      mov R1, A
      INC DPTR
      movx A, @DPTR
      mov TEMP, A
      mov A, R1
      CJNE A, TEMP, L3
      SJMP L4
L3:   JC L4
      movx @DPTR, A
      mov A, TEMP
      DEC DPL
      movx @DPTR, A
      INC DPL
L4:   DJNZ R5, L2
      DJNZ R4, L1
      END
```

So, with this we will be trying to write the code for this the program where from 3 from location 3000 h from location 3000 h; I have got 10 numbers stored and I want to sort those 10 numbers. So, I assume that since the program is there is no other program in the system the; my program originates at address 0 0 0 0 h.

So, if you know that if you know some different address in your system your RAM maybe from some different address. So, then you can put some other value there; then this N I take it as a constant which is 10. So, that later on if tomorrow I have to solve 20 number; so, I can just change this value from 10 to 20 ok. So, I do not need to change the entire program.

I take another temporary location which I called 10 for doing this exchange basically and I take the location 50 h for that purpose ok. So, this 50 h is a temporary RAM location which I will be using as a scratch pad for exchanging the variables if the exchanging the data items if they are out of order then. So, the start address is stored to be 3000; so, I take another constant start which is starting to EQU 3000 h. So, this is that they can write in terms of starts tomorrow if I have to sort an array starting at address 4000. So, I just need to change this point in the program and rest of the program will be made unaltered.

Then I take a counter like MOV R4 hash N minus 1; so, N is ten. So, I I take this will evaluate to 9; so you see that this statement is actually the statement like MOV R4 comma 9, but if I write 9 here. So, if tomorrow if I am asked to sort for 20 numbers; I have to change this point also in the program, but if I otherwise I can just change this number 10 to 20 and this part will be working as it is; so, there is no problem with that.

So, we will be following that philosophy MOV R4 comma N minus 1; then the DPTR should be set to that start address the DPTR is having the start address is 3000 h is coming to the DPTR. Then we MOV this R4 value to A for and then move this move this A value to R5.

So, this is done to just MOV R5 0 MOV this R5 A to R5; so, that we can do some more comparisons. And then I get the first number in the A register; so, MOV MOV X it MOV X; A comma at the rate DPTR. So, from the DPTR the; so, from the location 3000, so the value is coming to the A register. Actually these two statements; so, they are actually moving this R4 value to R5. So, I cannot directly MOV R4 to R5; so, I have done it like this.

So, you can of course, replace this two pair by putting a statement like MOV 5 comma 4. So, that can be done, but it is not done here they have a two separate statements. So, we can have this MOV X A comma at the rate DPTR. Then we do like MOV R1 comma A; so, the value of A is moved to R1 then INC DPTR. So, that now it is pointing to the next entry 3001 and then MOV X; A comma at the rate DPTR.

So, the next number it will be coming to the A register and then we save that value into a temporary location; in a temporary location 50 h we save that value. And then in R1 we had saved that the first value that we had read in R1, then we have read about the next value into the moved down to the temporary register. Now, from R1 I take it back to the A register so, that I can do a comparison. So MOV A comma R1; so now, the situation is the value that we had at 3000 is in is in A register and value that is there at 3001 is in the is in the memory location temporary. So, I can do a comparison CJNE instructions can be used CJNE Compare Jump or Not Equal A comma temp comma L 3 ok.

So, L 3 will come it will be jumping to otherwise I have to jump otherwise the numbers are they are not same. So, if they are not same there it is jumping to L 3 then it will otherwise it is make a jump to L 4. And this L 3 it will be it will be checking for that some carry is generated. So, if the numbers are same it if the numbers are same then I do not need to interchange. So, it will be it will it will come to this is SJMP L 4. So, it will be jumping over to this otherwise I do not need to change this thing.

So then after that I MOV X MOV X at the rate DPTR comma A; so, it is it is, so this A register for content was bigger so, that is now moved to the L 4 the to the memory location 3000; so A is greater than temporary. So, now, I need to exchange; so, these are first step of that exchange then MOV A comma temp MOV A comma temp and then we have to go back in the DPTR. So, we do a decrement of these DPL register.

So we cannot decrement this DPTR register because it that is 16 bit; so, you have to do DEC DPL and then MOV X at the rate DPTR comma A. So, the temp value has been moved to A register; so, this temp values has been moved to A register; so this MOV X instruction. So, this will take that temp value to the previous value to the previous value that is the 3000.

So, between 3000 and 3001; so, it will do an exchange; then again I do INC DPL and then in L. So, that is the composition of one, one comparison and exchange; so, I have to see whether I am done with the comparison. So, this is this is done by this D J NZ R5 comma L 2, where L 2 is that instruction this instruction is L 2.

So, I need to take the decrement R5 and if it is not 0 decrement jump no nom 0 R5. So, it will come to this L 2 so, that I can take the next number from the memory; DJNZ R5 L 2 and this will be L 2. And so, this will be completing one loop; so on top of that you should have another that is by R4. So, DJNZ R4 DJNZ R4 to L 1; DJNZ R4 L 1 where this L 1 is where L 1 is the location of this one; so, L 1 is this one.

So, this is starting the next iteration of the outsider. So, for comparison you know there for the sorting you know that we should you have to loops and those two loops they will be operating in a nested fashion. So, we create two loops here one it in L 1 another L 2; so that way it goes; so, this way we can have this program developed ok.

So, if you just trace through the program; so, you will see that this is actually doing the is actually doing the sorting. So, after this; so, this will be repeating till R5 R5 becomes 0 this is this will be repeating till R4 becomes 0. So, after this we have this one and then we have got another statement to end a program. So, which is called END; so, this is also another assembler directive.

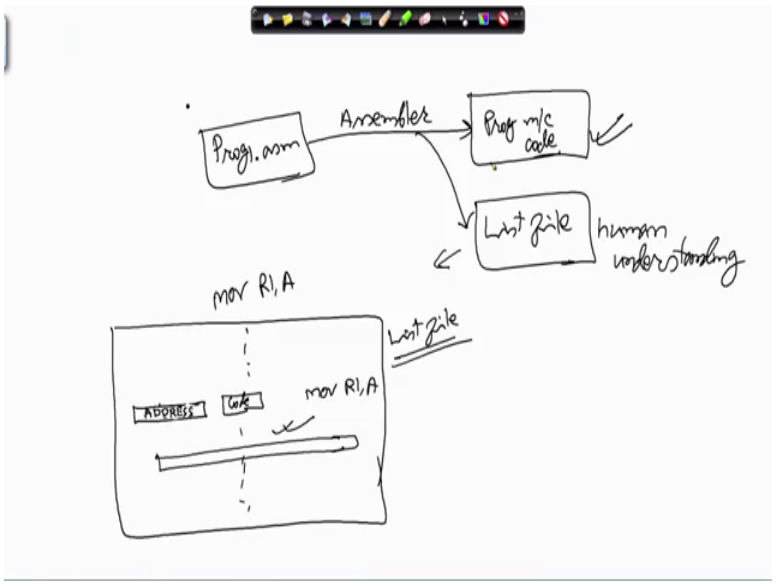
So, this is telling that there is translation process will be ending at this point. So, I should not do I do I do is there no more translation be required. So, this is useful because many times what happens is that in the development phase of the program. So, we write a many routine and we do not want to really put them into our final code, but if there is something happening in the final code, then maybe for debugging purpose we need to leave up those codes again.

So, for that job; so, this end directive can be useful like when I am say this is my is this is a complete program that I have written then maybe finally, I see that I need to the I need only this much of program for the correct operation of the system. So, at this point I put an end in my assembly language file so, that this part is not translated.

So, this part is not translated into machine code, but I do not believe this part because later on what can happen is that there is some bug that is that is come up or some augmentation has to be done then this routine will have will become part of routine ok. So, in that case I can just shift this end point to here just shift this endpoint to here. So, that I do not have to just be I do not have to readjust everything if I do not have to rewrite the program together.

So, this way we can have this we can have this program translated into machine code and have selected portion can be taken. Now when you are doing this translation like the program that we have we are writing. So, they are the assembly language program.

(Refer Slide Time: 29:09)



May be the name of the program is a program 1. So, this is this normally has an extension of a s m Prog 1 dot asm. So, when this is passed through the assembly process the assembly process of the assembler. So, it produces the number of output one one one output is definitely the executable version. So, this program translated code the machine code another output it produces; so, which is known as the list file. So, this program code output; so, this is for the processor to understand, but this list file to this is for human understanding. And in this list file; so, we have got this the program statements and their corresponding codes.

So, if I have got a statement like MOV say R1 comma A; then what will happen is that in the list file. So, you will first the corresponding address will be there at which this statement is being put the address will be there then the code the corresponding code that is generated for doing this thing operation. So, for this instruction that code will be there and then the instruction will also be written here. So, this way for the entire program all the statements they will be shown, they will be written one after the other that their addresses will be shown and their addresses will be shown and also there code will be shown.

So, this helps in understanding whether the program has been translated the program has been written correctly or we can understand whether the program has been translated correct; whatever we are looking for whether that has been done or not.

And also if there is some error in the Trans assembly process then they can be flashed. So, this error messages maybe somewhere here; so, we understand that while translating this statement there was an error. So, we can rectify this error in the asm file and again give it for assembly this particular file is called the list file. So, I would suggest that whenever you are doing assembly language programs so, you would take help of some assembler because that is the languages are complex.

So, you take help of assembler and do look into the list file that is produced apart from this code. So, code will not understand anything because that is a binary file, but this list file is a text files; we can have a look into this text file and see what is there in the translated version of the code. And if there is some error then of course, we can see the other and then go back and correct the assembly language file.