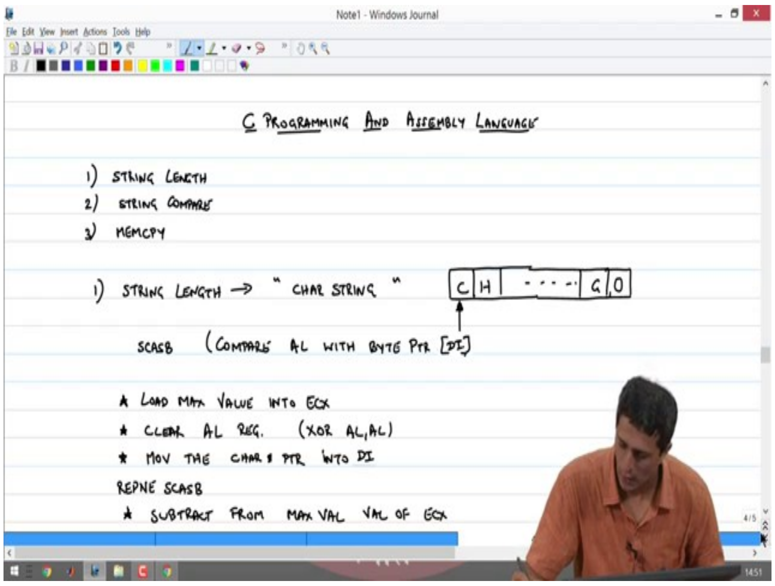
C Programming and Assembly language
Prof. Janakiraman Viraraghavan
Department of Electrical Engineering
Indian Institute of Technology, Madras

Lecture - 19

Welcome back to this course on C Programming and Assembly language. We are in module 4 where we are looking at the impact of assembly translations on performance . So, in the last few lectures we looked at handling you know variable argument list like printf, then we also looked at the performance impact of recursion why it is slow and why loops are faster.

So, continuing on those lines in this lecture we will look at accelerating certain functions in C using or by exploiting, certain hardware loops which we introduced earlier in the instruction set of 8086 .

(Refer Slide Time: 00:55)



C PROGRAMMING AND ASSEMBLY LANGUAGE

1) STRING LENGTH
2) STRING COMPARE
3) MEMCPY

1) STRING LENGTH → " CHAR STRING "

C	H	- - - -	G	0

SCASB (COMPARE AL WITH BYTE PTR [DI]

- LOAD MAX VALUE INTO ECX
- CLEAR AL REG. (XOR AL,AL)
- MOV THE CHAR * PTR INTO DI

REPNE SCASB
- SUBTRACT FROM MAX VAL VAL OF ECX

So, we are going to look at the implementation of the following functions. String length, string compare and we look at memcpy . I am not going to go through and put down the exact assembly instructions here, what I am going to do I am going to discuss the key instructions that are needed and just tell you the pseudo code that is needed to get this assembly program going .

I will leave it as an assignment for you to go ahead and complete this assembly programs in actual detail. So, let us start with string length . String length what you have to do? You have a character string stored like this and this in memory is effectively going to be stored 1 byte per character and will get terminated using the backslash 0. So, this will be C H all that you know string and then you will have the number 0 being stored in the last character or the last byte and that is the terminating byte .

So, effectively if you want to implement this particular function in C, you just start by pointing your char star pointer here check if it is backslash 0 and if not keep preceding and you keep counting till you hit the backlash 0 . Of course, this kind of a loop implementation in C is going to be extremely slow and therefore, its advised that you do not go ahead and implement your own string length or string compare or such key functions .

Standard libraries are provided to you, because those libraries have been implemented using the right hardware instructions, they are bound to be much faster than your own implementation in terms of loops or recursion . So, what would those libraries have used is the question . So, the key instruction that we want to use for this particular implementation is this instruction called SCAS byte . So, all I need to do is simply go ahead and look for my backslash 0 . So, SCAS byte what does it do? It basically compares the AL with byte pointer that is pointed to by the source index .

And if it matches then it is just going to go head and stop , and you can also provide a counter in ECX so, that you do not exceed and you do not keep comparing for ever if there is no match . So, what you have to do? You are simply going to load max value into ECX. So, you just imply you know load this with the highest positive value that you have and then what I have to do is, I am going to load the pattern that I want to search for in AL.

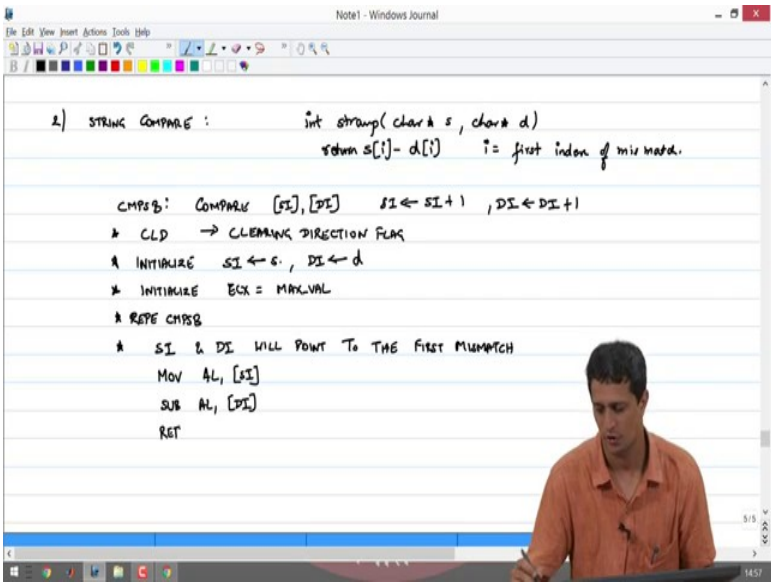
So, I am going to search for backslash 0. So, I will just clear AL register . These are the steps; of course, this can be implemented using XOR AL comma AL . Then I need to point my string to the right position . So, therefore, I am going to MOV my SI with I will MOV char star pointer into destination index, with that I have initialized all the variables that I need and then I will go ahead and implement this function called SCASB . So, what is it going to do, it is going to compare AL with byte pointer of DI .

If I wanted to do a word pointer out say compare AX with word pointer or EAX with a D word pointer and so, on . Now of course, this SCASB will only do it for one particular value and then increment or decrement my source index by 1 4 or 1 2 or 4 as we discussed earlier . So, therefore, you also have to use repeat right as long as it is not equal . So, you keep repeating this as long as it is not equal and in the process your decrementing ECX by 1 every time and when you reach 0 or you have found a match the program will stop .

So, the problem here is that, in order to get the particular string length right after a match has been found, I need to subtract from ECX the max value that has stored initially because remember that the ECX counter is not counting up, it is counting down from some MAX value that I had initialized it with. And therefore, to get string length I will do subtract from MAX VAL the value of ECX. So, if you go ahead and do this operation then at the end of you know the subtraction what will remain in that register is the string length of this particular string.

So, now let us go ahead and do the, an accelerated implementation for string compare .

(Refer Slide Time: 08:27)



2) STRING COMPARE : int strcmp(char * s , char * d)
return s[i] - d[i]   i = first index of mismatch.

CMPSB: COMPARE [SI], [DI]   SI ← SI + 1 , DI ← DI + 1

- CLD → CLEARING DIRECTION FLAG
- INITIALIZE SI ← s , DI ← d
- INITIALIZE ECX = MAX_VAL
- REPE CMPSB
- SI & DI WILL POINT TO THE FIRST MISMATCH

```
MOV AL, [SI]
SUB AL, [DI]
RET
```

So, what is string compare supposed to do? So, if you look at the signature of this function it would be int strcmp of char star let me write that clearly; char star source, char star destination . So, when I compare two strings what I want to do it, as long as the comparison is correct you keep going, but when there is the mismatch you return the difference of the source minus the index .

So, you want to return the s of i minus d of i value as key subtraction where i is equal to first index of mismatch that is what I want to return. Now, how do you go head and implement the same thing in hardware by exploiting a hardware loop? So, the hardware instruction that needs to be exploited here is CMPSB, which is compare byte. I am going to compare byte by byte and keep preceding until I you know find the mismatch . So, as long as they are equal you keep proceeding with the comparison and when the mismatch happens, you return saying you know this has failed.

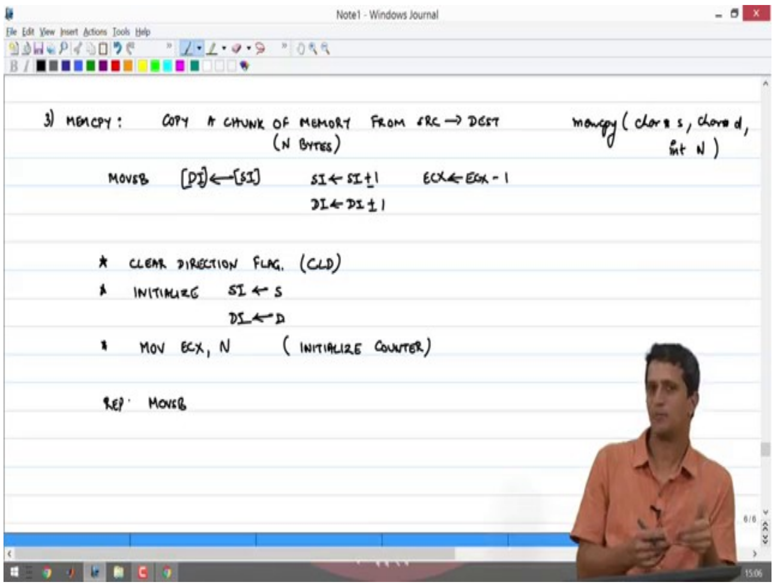
So, what you do? Again I need to for example, load initialize, source index with s then destination index with the char star pointer d, then I will initialize my ECX equal to sum MAXVAL and then now I am going to invoke this CMPSB. So, what does CMPSB do? It is going to compare contents of SI with contents of DI right and then SI will be either incremented or decremented by 1, 2 or 4 depending on whether it is a you know ah byte or a word, but in this case since its CMPSB I am not going to worry about the word or the de word .

And of course, I will also do a CLD before I start which is Clearing Direction Flag . In this case I will not even worry about the minus sign, auto increment is all I am worrying about here and DI will also become DI plus 1. But of course, if I want now   compare across many such characters, then I need to put this in a loop and therefore, I will do REPE repeat as long as there equal you keep preceding and on the first mismatch you just stop .

You just stop here and once you are done with this particular instruction SI and DI will point to the first mismatch. So, all I have to do now is in order to implement the string compare function right which is to subtract the first mismatch value and return that guy out, I will simply MOV into some register AL contents of SI, and subtract AL from contents of DI or the other way round contents of DI from AL and then I just call return .

So, this is how the string compare function can be accelerated in hardware by exploiting the CMPSB instruction.

(Refer Slide Time: 14:13)



3) MEMCPY : COPY A CHUNK OF MEMORY FROM SRC $\rightarrow$ DEST (N BYTES)

memcpy (char * s, char * d, int N)

MOVSB $[DI] \leftarrow [SI]$ $SI \leftarrow SI \pm 1$ $ECX \leftarrow ECX - 1$

$DI \leftarrow DI \pm 1$

* CLEAR DIRECTION FLAG. (CLD)
* INITIALIZE $SI \leftarrow S$
  $DI \leftarrow D$
* MOV ECX, N (INITIALIZE COUNTER)

REP MOVSB

So, the last instruction that I wish to discuss the function that I wish to discuss is the memcpy. As I mentioned earlier I am not going into the detailed implementation, I am only outlining the steps involved as a pseudo code and I am also highlighting the key instruction that can be exploited to speed up this process . So, of course, you have to check for many things whether the memory is valid all of that the function will still do.

So, that is why if you go and look at an implementation in a standard library and you look at the assembly output, it will be lot more than just these set of instructions that I explained earlier , but those are not the heart of this function . So, memcpy what does memcpy do? I want to copy a chunk of memory sorry from source to destination. I want to copy let us say n bytes of information right, chunk of memory is N bytes. So, the key instruction that has to be used here is MOVS byte; MOVS byte essentially simply does the following, I am going to MOV from contents of source index I will move it to the destination index and after this operation I will do SI equals SI plus 1 or minus 1 or DI not or and DI equal to DI plus minus 1.

Plus minus is decided by the direction flag . So, what you do here? You again Clear Direction flag CLD then you initialize SI and let me write down you know what this thing is . So, this will be memcpy of I will call it char star s for now d comma N; int N, initialize SI to S then DI to D and then I am going to load MOV into ECX which is the counter the value of N , this is basically . And now I will go ahead and implement this function or I will call this instruction MOVS byte.

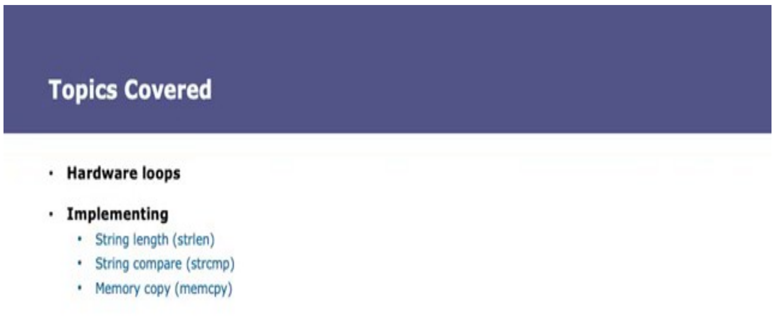
So, unfortunately this MOVS byte will do the operation only for one particular byte right, you implement MOVS byte then it basically either auto increments or decrements SI and DI and then decrements ECX by 1. So, in order to do this again and again until ECX has gone down to 0, you need to have this prefix which is of course, not dependent on any condition, but it is an unconditional prefix REP MOVS byte . So, since MOVS byte does not affect any flag register, this REP is going to just do it as long as ECX is not 0 .

Unlike the earlier prefix that if that we used, where we did REP ne or REP e; here the REP does not use any flag register to check the condition after MOVS byte has been executed. Of course, because MOVS byte is just a data movement instruction no flag is going to be affected in the process and therefore, the only way that you can stop this loop is to get ECX down to 0. So, this way you can implement the memcopy using MOVS byte and the REP prefix unconditional repeat prefix and it can be used to speed up the memcopy function significantly .

So, with that we conclude the discussion on accelerating certain string functions and of course, its these are just examples; three examples that are specified here, you can go back and look at other examples as well in order to understand if those functions have been spread up at an assembly level using hardware loops or not.

Thank you.

(Refer Slide Time: 19:49)



Topics Covered

- **Hardware loops**
- **Implementing**
  - String length (strlen)
  - String compare (strcmp)
  - Memory copy (memcpy)