Assembly Language Lab 8

String Primitive Instructions

String primitive instructions are highly optimized instructions to work on strings or integer arrays. These instructions are MOVSx, CMPSx, SCASx, STOSx, LODSx where x in each instruction name is replaced by b, w or d to make the instruction works with byte, word or double-word arrays respectively.

These instructions use memory operands pointed by the ESI register or EDI register or both. Used register(s) is automatically incremented or decremented after instruction execution based on the Direction flag (a bit in the EFLAGS register). If the direction flag is cleared (DF = 0), registers are incremented. Otherwise (DF = 1), they are decremented. Direction flag can be set by `std` instruction, and cleared by `cld` instruction

They often used with repeat prefix which allows them to be automatically repeated using ECX as a counter. So, you can process an entire array or string using only single instruction. The following repeat prefixes are used:

Prefix	Description
REP	Repeat while ECX > 0
REPE/REPZ	Repeat while ECX > 0 and zero flag is set
REPNE/REPNZ	Repeat while ECX > 0 and zero flag is clear

When repeat prefix is used, the sequence of its operation is as follows:

1. Test repeat condition(s) [ECX > 0 and zero flag status with REPE and REPNE]
2. If test success, perform the instruction (in addition to incrementing or decrementing ESI or EDI or both). Otherwise, skip the entire instruction.

MOVSB / MOVSW / MOVSD Instructions

These instructions copy data pointed by ESI to the memory location pointed by EDI. They are used with repeat prefixes to copy arrays and strings.

Example 1: Copy an Array

This example uses string primitive instructions to copy items from double word array to another

```
include irvine32.inc
.data
source dword 5 dup(10)
target dword 5 dup(?)
```

```
.code
main proc
     cld                 ;clear direction flag. Move forward
     mov ecx, lengthof source
     mov esi, offset source
     mov edi, offset target

     rep movsd

     exit
main  endp
end main
```

Note that after `rep movsd` is executed the ESI and EDI are pointing to the next item after the end of `source` and `target` arrays respectively.

CMPSB / CMPSW / CMPSD

These instructions compare a memory operand pointed by ESI (source) with the memory operand pointed by EDI (destination). They actually subtract target from source and set flags according to the result like CMP instruction. In the following sample code, we find that source is less than target, so when JA is executed the condition fails and the conditional jump is not taken.

```
  .DATA
    SOURCE DWORD 1234h
    TARGET DWORD 5678h

  .CODE
    MOV ESI, OFFSET SOURCE
    MOV EDI, OFFSET TARGET
    CMPSD     ;like cmp [ESI], [EDI] except cmp can
              ;not take two memory operands
    JA l1
    JMP l2
```

When these instructions are used with REPE/REPNE repeat prefixes to compare items from two arrays and stop comparing on the first mismatch. Conditional jumps can be used later on to decide if the source array is greater or less than the destination.

Example 2a: Compare two strings

This example uses primitive string instructions to compare two strings assuming that both strings have the same size.

```
include irvine32.inc
.data

source byte 'ameen' target byte 'ayman'

strSmall byte "source is smaller",0
strGreat byte "source is greater",0
strEqual byte "both strings are equal",0

.code
main proc

     cld                        ;clear the direction flag
     mov esi, offset source
     mov edi, offset target
     mov ecx, lengthof source

     repe cmpsb                 ;compare the 2 arrays and stop at the
                                ;first mismatch

     jb source_smaller          ;check last char in source and last
     ja source_greater          ;char in dest
     mov edx, offset strEqual
     jmp done

source_smaller:
     mov edx, offset strSmall
     jmp done

source_greater:
     mov edx, offset strGreat

done:
     call writestring
     call Crlf
     exit
main endp
end main
```

Example 2b: Compare two strings (with different sizes)

This example uses primitive string instructions to compare two strings with different sizes.

```
include irvine32.inc

.data
```

```
source byte 'ameen'
target byte 'ameena'

strSmall byte "source is smaller",0
strGreat byte "source is greater",0
strEqual byte "both strings are equal",0

.code
main proc

     cld                              ;clear the direction flag
     mov esi, offset source
     mov edi, offset target
     mov ecx, lengthof source
     mov eax, lengthof target         ;EAX = length of target string
     .if ecx > eax
          mov ecx, eax
     .endif

     repe cmpsb                  ;compare the 2 arrays and stop at the
                                 ;first mismatch

     jb source_smaller           ;check last char in source and
     ja source_greater           ;last char in dest


     .if eax < lengthof source        ;if both chars are equal then
          jmp source_greater          ;check the length of each one
     .elseif eax > lengthof source
          jmp source_smaller
     .else
          mov edx, offset strEqual
          jmp done
     .endif

source_smaller:
     mov edx, offset strSmall
     jmp done

source_greater:
     mov edx, offset strGreat

done:
     call writestring
     call Crlf
     exit
main endp
end main
```

SCASB / SCASW / SCASD Instructions

Compare a value in AL/AX/EAX to a byte, word, or double word addressed by EDI respectively. They are useful when looking for a single value in an array or a string.

Example 3: Sequential Search

This example uses string primitive instructions to search in a string for a given character.

```
include irvine32.inc

.data
str1 byte "hello world",0
.code
main proc
     cld
     mov edi, offset
     str1 mov al, 'w'
     mov ecx, lengthof str1

     repne scasb         ;scan string till 'w' is found
     jne not_found       ;'w' is not found, jump to not_found label

     ;Otherwise, ecx has the index of that character but
     reversed ;So, make eax = lengthof(str1) - ecx - 1
     mov eax, lengthof str1 -
     1 sub eax, ecx
     jmp done

not_found:
     mov eax, -1

done: ;eax has the index of the char OR -1 if not found call
     writeint

     exit
main endp
end main
```

STOSB / STOSW / STOSD Instructions

These instructions store the content of AL/AX/EAX into memory at the offset pointed to by EDI. They are useful for filling all elements of a string or array with a single value.

Example 4: Fill an array

This example uses string primitive instructions to initialize an array by a specific value.

```
include irvine32.inc
.data
arr1 word 20 dup (?)

.code
main proc
     cld
     mov edi, offset arr1
     mov ax, 5
     mov ecx, lengthof arr1

     rep stosw           ;set all items in arr1 by AX value

     exit
main endp
end main
```

LODSB / LODSW / LODSD Instructions

These instructions load a byte, word, or double word from memory at ESI into AL/AX/EAX respectively. REP prefix is rarely used with them as each new value loaded into accumulator overwrites its previous contents.

Example 5: Array Multiplier

This example uses string primitive instruction to multiply an array by a integer value.

```
include irvine32.inc

.data
arr1 dword 1, 2, 3, 4, 5
multiplier dword 10

.code
main proc
     cld
     mov esi, offset arr1     ;initialize ESI for LODSD instruction
     mov edi, esi             ;initialize EDI for STOSD instruction
     mov ecx, lengthof arr1
l1:
     lodsd                    ;mov eax, [esi]
     mul multiplier           ;edx:eax = eax * multiplier
     stosd                    ;mov [edi], eax
```

```
    loop l1

    exit
main endp
end main
```

Two Dimensional Arrays

Multi-dimensional arrays in assembly are defined as single dimensional arrays. For example, a 2D array can be defined as 1D array where rows are concatenated in a single row. Accessing items in 2D arrays can be done by indirect memory addressing mode using a base register and an index register like [ebx+esi]. In such addressing mode, a base register (i.e. ebx) should contain the row offset while an index register (i.e. esi) should contain the column offset.

Example 6: Sum a column

This example sums values in a column in a 2D integer array. Its idea is to iteratively make ESI register points to the first item in each row, then calculate the offset of the item in the targeted column and keep it in EBX register. Using base-index addressing mode (i.e. [ESI+EBX]), we can access the targeted item.

```
include irvine32.inc
.data
table1    dword 1, 2, 3,  4
          dword  5, 6, 7,  8
          dword  9, 10, 11, 12
;the same as: table1 dword 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12

ncol dword 4

.code
SumCol proto table: PTR DWORD, nrow:DWORD, ncol:DWORD,
icol:DWORD main proc
     invoke SumCol, offset table1, 3, 4, 3

     call
     writeint
     call CrLf
     exit
main endp

SumCol proc table: PTR DWORD, nrows:DWORD, ncols:DWORD,
     icol:DWORD push esi
```

```
        push
        ecx
        push
        ebx
        push
        edx

        mov esi, table ;ESI: points to the begining of array (1st row)
        mov ecx, nrows ;ECX: # of rows
        mov edx, ncols
        shl edx, 2     ;EDX: # of bytes in each row (ncols*4)
        mov ebx, icol  ;EBX: index of target column
        mov eax, 0     ;EAX: accmulator for the result

l1:
        add eax, [esi+4*ebx]
        add esi, edx              ;ESI: points to the next row
        loop l1

        pop edx
        pop ebx
        pop ecx
        pop esi

        ret
SumCol endp
end main
```

Assignment #5 :

1. Write a procedure named asm_strCat that concatenates a source string to the end of a target string. Use string primitive instructions when is possible.

2. Write a procedure named asm_strFind that searches for a string inside a target string and returns the first matching position if any. Use string primitive instructions when is possible.

3. Write a procedure named asm_strRemove that removes n characters from a string. This procedure should take a pointer to a string, the position in the string where the character are to be removed and number of characters to be removed n. Use string primitive instructions when is possible.