Seminar 3

Little-endian representation of numbers in the memory. Conditional and unconditional jumps.

String operations

Contents

1. Little-endian representation in the memory and non-destructive conversions

In the computer memory (not on the CPU registers!) numbers represented on more than 1 byte are represented in little-endian format (i.e. the least significant byte of the representation is placed at the smallest memory address).

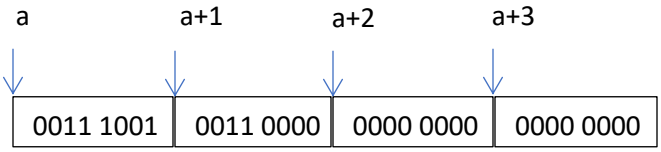
We also said in the previous seminar that numbers are represented in the memory and on the CPU registers in the unsigned representation or the signed representation. And now we are saying numbers are represented in little-endian format in the memory. So which is it?

Well, every number in assembly is represented on 1, 2, 4 or 8 bytes (depending on the context in which it is declared; if we have this declaration: "a dw 2", then 2 will be represented on 2 bytes). It is first represented in binary using signed or unsigned representation. If the number is represented on more than 1 byte and in the memory (not on the CPU registers), the bytes of the signed or unsigned representation are then placed in the memory in little-endian format.

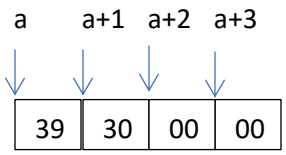
Example:

```
a  DD  0

….

mov dword [a], 12345
```

In the above instruction, the number 12345 is positive so it will be represented in the unsigned representation on 4 bytes as 0000 0000 0000 0000 0011 0000 0011 1001. Let's write this binary number in the hexadecimal base (as we know from seminar 1 that a group of 4 bits is a hexadecimal digit): 00003039h. This double word number will then be represented in memory in little-endian representation on 4 bytes as:



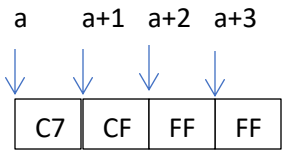
The memory addresses: “a”, “a+1”, “a+2”, “a+3” represent the memory addresses of the 4 bytes of variable “a”; each address is the address of 1 byte. The byte from memory address “a” is the least significant byte of the number, the byte from memory address “a+1” is the next least significant byte of the number and so on. We usually use the hexadecimal base when we represent a number in the memory (because there are less digits) and we also know that 4 bits is a hexadecimal digit. The above little-endian representation is:



If we consider the next example:

```
mov dword [a], -12345
```

we first represent -12345 in the signed representation as 1111 1111 1111 1111 1100 1111 1100 0111 and we consider this sequence on 32 bits in hexadecimal as FFFFCFC7h so this will be represented in the memory in little-endian format as:

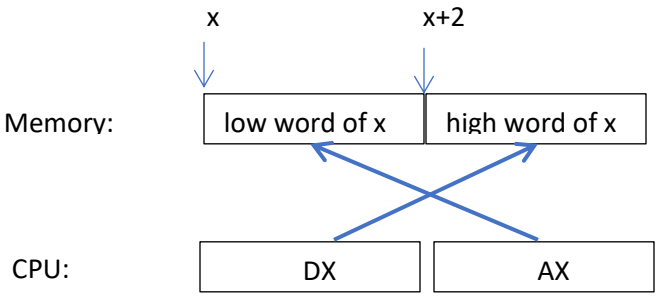


The little-endian representation is important when we want to do “non-destructive conversions” like: refer to the low word of a double word from the memory, refer to the high byte of a word from the memory, etc.

Ex. 1. Write a program that computes the expression: x:=a*b+c*d where all numbers are unsigned and represented on a word.

```
bits 32
global start
extern exit
import exit msvcrt.dll
```

```
segment data use32 class=data
        a dw 2
        b dw 5
        c dw 10
        d dw 40
        x dd 0
segment code use32 class=code
start:
        mov ax, [a]
        mul word [b]            ; DX:AX := AX * b = 2*5 = 10
        ; we save the doubleword DX:AX into x
```




```
        mov word [x], ax        ; save AX into the low word of "x" (i.e. the word from the address "x")
        mov word [x+2], dx      ; save DX into the high word of "x" (i.e. the word from the address "x+2")

        mov ax, [c]
        mul word [d]                    ; DX:AX := AX*d = c*d = 400

        ; add DX:AX to x
        add word [x], ax        ; we first add AX to the low word of "x"
                                ; if there is an overflow, the CF (Carry Flag) will be set to 1
                                ; otherwise CF will be set to zero
        adc word [x+2], dx      ; add DX to the high word of "x", but add also the CF
        ; call exit(0)
        push dword 0
        call [exit]
```

New instructions:

ADC – Add with Carry

adc dest, source => dest:=dest+source+CF

SBB – Substract with borrow

sbb dest, source => dest:=dest+source-CF

2. Conditional and unconditional jump instructions

Conditional jump instructions are just like the "IF" instruction in a high-level programming language. In assembly language, an "IF" is composed of 2 instructions: the compare instruction and the conditional jump instruction.

The compare instruction:

cmp a, b ; performs a non-distructive *sub a, b* (meaning that *a* does not change) and sets the flags accordingly

A conditional jump instruction checks the value of specific flags and depending on the value, performs a "jump in the program" (i.e. moves the execution to a different part of the program – a part of the program is identified by a label). A conditional jump instruction makes sense and should follow a **cmp** instruction.

Jump Instructions are composed from a few key words:

- E = equal;
- N = not;
- J = jump
- A = above; B = below ; for unsigned numbers
- G = greater L = less ; for signed numbers

Conditional jump instructions that consider the numbers unsigned

jb label ; (Jump if below) jumps to label if a < b (CF = 1)

jbe label ; (Jump if below or equal) jumps to label if a ≤ b (CF = 1 or ZF = 1)

jnb label ; (Jump if not below) jumps to label if a ≥ b

jnbe label ; (Jump if not below or equal) jumps to label if a > b

ja label ; (Jump if above) jumps to label if a > b (CF = 0 & ZF = 0)

jae label ; (Jump if above or equal) jumps to label if a ≥ b

jna label ; (Jump if not above) jumps to label if a ≤ b

jnae label ; (Jump if not above or equal) jumps to label if a < b

The *a* and *b* values from above are the *a* and *b* operands of the **cmp** instruction that was issued before the conditional jump instruction.

Conditional jump instructions that consider the numbers signed

jl label ; (Jump if less) jumps to label if a < b (SF ≠ OF)

jle label ; (Jump if less or equal) jumps to label if a a ≤ b

jnl label ; (Jump if not less) jumps to label if a a ≥ b

jnle label ; (Jump if not less or equal) jumps to label if a > b

jg label ; (Jump if greater) jumps to label if a > b (SF = 0 & ZF = 0)

jge label ; (Jump if greater or equal) jumps to label if a a ≥ b (SF = OF)

jng label ; (Jump if not greater) jumps to label if a a ≤ b

jnge label ; (Jump if not greater or equal) jumps to label if a < b

The *a* and *b* values from above are the *a* and *b* operands of the **cmp** instruction that was issued before the conditional jump instruction.

Conditional jump instructions that consider one flag

jc label ;Jump if CF=1 (carry flag)

jnc label

je label ;Jump if ZF=1 (zero flag)

jz label ;Jump if ZF=1

jne label

jnz label

js label ;Jump if SF = 1 (sign flag)

jns label

jp label ;Jump if PF = 1 (parity flag)

jnp label

jo label ;Jump if OF =1 (overflow flag)

jno label

Conditional jump instructions that check the value of CX or ECX

Jcxz label ;Jump if CX = 0

Jecxz label ;Jump if ECX = 0

Unconditional jump instruction

jmp label ; always jumps to the specified label

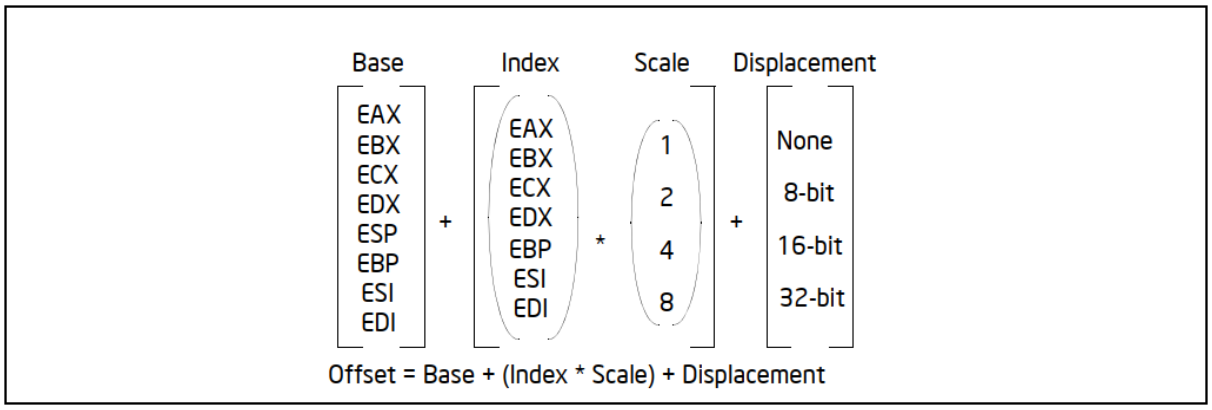
3. Working with strings of bytes

Memory addresses and offsets

In order to work with strings of bytes/words/dwords we need to know more about memory addresses. So far, we have used expressions like:

mov ax, [a]

where *a* is a variable and the instruction above moves a word value from the memory starting at the memory address "*a*". We have seen so far that a variable name is just a constant address – the address of that variable in the memory. To be more precise, a variable name is a constant offset. A full address specification comes in the form of two numbers: **segment_selector : offset**. A **segment_selector** is a 16-bit number and specifies a pointer in a segment descriptor table (GDT – global descriptor table) which defines a memory zone. For today's seminar you do not really need to understand what a memory segment is (you will find more details about memory segments and memory management at the course), just know that a memory segment is a continuous memory zone and its address is already stored in the appropriate segment register (CS, DS, ES or SS) by the operating system before your program starts and you are not allowed to change it. The **offset** is a 32-bit number which specifies a pointer inside the segment specified by the **segment_selector**. The name of a variable, like I have said before, represents just the offset of that variable (the segment to which this offset refers to is the data segment whose starting address is already placed in the DS register by the operating system). The full format of an offset includes 4 quantities : a base, an index, a scale and a displacement and is presented below in the following figure:



In an offset specification any combination of the above 4 quantities can appear (inside "[ ]"), including each component individually. The displacement is just a constant number. Below you can find some examples of offset specifications (as the second operand of the instruction):

mov ax, [a] *; only displacement*

mov ax, [eax] *; only base or index*

mov ax, [a+eax+ebx] *; base, index and displacement*

mov ax, [eax+eax+a+2] *; base, index and displacement*

*mov ax, [a+4+ebx*2]* *; index, scale (i.e. 2) and displacement*

*mov ax, [eax + ebx*4 + 20]; base, index, scale (i.e. 4) and displacement*

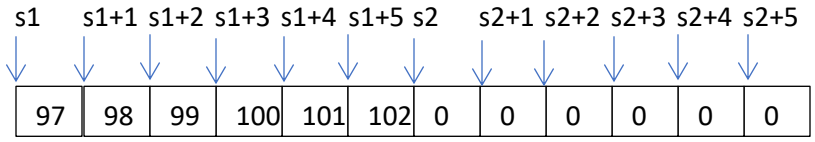
Ex.2. Being given a string of bytes containing lowercase letters, build a new string of bytes containing the corresponding uppercase letters.

Method 1:

```
bits 32
global start
extern exit
import exit msvcrt.dll
```

```
segment data use32 class=data
    s1 db 'abcdef'
    lenS1 equ $-s1
    s2 times lenS1 db 0
```

; the data segment looks in the memory like this:



; the top row in the above figures represents offsets. For example, in OllyDebugger, the first byte ; from the data segment and thus the offset of *s1* is always 0x00401000. The offset of variable *s2* ; is equal to the offset of *s1* plus 6 (i.e. 0x00401006). The bottom row represents the actual

; values from the memory in base 10 (97 is the ASCII code of 'a', 98 is the ASCII code of 'b', 99 ; is the ASCII code of 'c' …). The bytes from string s2 are initialized with zero.

; *lenS1 equ $-s1* defines a constant (equ = constant, no memory space is reserved)

; $ is the location counter, the current offset in the data segment up to the code line in which it

; appears. So, before the code line *lenS1 equ $-s1* , there were 6 bytes generated in the string *s1*

; so the current offset is $=s1+6. *s1* in the above instruction is just the offset of variable *s1*, so

; len1 = $ - s1 = s1+6 – s1 = 6 bytes (the length in bytes of string *s1*).

; *s2 times lenS1 db 0* defines a variable s2 which contains lenS1=6 bytes, all initialized with

; the value zero.

```
segment code use32 class=code
start:
    ; we solve this problem by considerring in ESI the current index in string s1 and string s2
    ; and we do a loop with lenS1=6 iterations and at each iteration we move the byte s1[ESI]
    ; into s2[ESI], after we change it to an uppercase letter. So, in this loop, ESI will have the
   ; values: 0, 1, 2, 3, 4, 5.
```

```
    mov esi, 0

    repeat:
        mov al, [s1+esi]    ; AL <- the byte from the offset s1+esi
        sub al, 'a' -'A'    ; obtain the corresponding uppercase letter in AL
        mov [s2+esi], al   ; AL -> the byte from the offset s2+esi

        inc esi             ; esi:=esi+1; move to the next index in strings s1 and s2
        cmp esi, lenS1
        jb repeat           ; IF (esi < lenS1) jump to repeat,
                            ; otherwise continue below


    push dword 0
    call [exit]
```

Method 2:

```
bits 32
global start
extern exit
import exit msvcrt.dll

segment data use32 class=data
    s1 db 'abcdef'
    lenS1 equ $-s1
    s2 times lenS1 db 0

segment code use32 class=code
start:
    ; this time we solve this problem by considerring in ESI the offset of the current byte from
```

; string s1 and in EDI the offset of the current byte from string s2. We do a loop with
; lenS1=6 iterations and at each iteration we move the byte from offset ESI into the byte
; from offset EDI, after we change it to an uppercase letter. So, in this loop, ESI will have
; the values: s1+0, s1+1, s1+2, s1+3, s1+4, s1+5 and EDI will have the values: s1+6, s1+7,
; s1+8, s1+9, s1+10, s1+11.

```
mov esi, s1      ; initialize esi
mov edi, s2      ; initialize edi
mov ecx, lenS1   ; ecx will store the number of iterations in the loop

repeat:
    mov al, [esi]          ; AL <- the byte from the offset s1+esi
    sub al, 'a' -'A'       ; obtain the corresponding uppercase letter in AL
    mov [edi], al          ; AL -> the byte from the offset s2+edi

    inc esi      ; esi:=esi+1; move to the next byte in strings s1
    inc edi      ; edi:=edi+1; move to the next byte in strings s2
    dec ecx      ; exc:=ecx-1
    cmp ecx, 0   ; IF (ecx > 0) jump to repeat
    jb repeat    ; otherwise exit the loop

push dword 0
call [exit]
```

Instructions for loops

LOOP label

- Continues the loop as long as the value of ECX ≠ 0
- It is equivalent to these 3 instructions:
  - ; dec ecx
  - ; cmp ecx, 0
  - ; ja label

LOOPE / LOOPZ

- Continues the loop as long as the value of ECX ≠ 0 and ZF = 1
- So the cycle terminating either if ECX = 0 or if ZF = 0.

LOOPNE / LOOPNZ

- Continues the loop as long as the value of ECX ≠ 0 and ZF = 0
- So the cycle terminating either if ECX = 0 or if ZF = 1.

Method 3:

```
bits 32
global start
extern exit
import exit msvcrt.dll
segment data use32 class=data
    s1 db 'abcdef'
    lenS1 equ $-s1
    s2 times lenS1 db 0
segment code use32 class=code
start:
    mov esi, s1      ; initialize esi
    mov edi, s2      ; initialize edi
    mov ecx, lenS1   ; ecx will store the number of iterations in the loop
    jecxz finished
    repeat:
        mov al, [esi]          ; AL <- the byte from the offset s1+esi
        sub al, 'a' -'A'       ; obtain the corresponding uppercase letter in AL
        mov [edi], al          ; AL -> the byte from the offset s2+edi

        inc esi      ; esi:=esi+1; move to the next byte in strings s1
        inc edi      ; edi:=edi+1; move to the next byte in strings s2
        loop repeat
    finished:
    push dword 0
    call [exit]
```

Ex.3. How many times is the loop executed?

```
MOV ECX,5
repeat:
DEC ECX
LOOP repeat
```

Ex.4. How many times is the loop executed?

```
MOV EBX, 0
MOV ECX,0
repeat:
INC EBX
LOOP repeat
```