**IV. Bitwise operations. Conditional and unconditional jumps. String operations**

**IV.1. Bitwise operations**

Bitwise operations are operations that are executed for each bit (i.e. bit-by-bit). We have: logical bitwise operations, bit shift operations and rotate operations.

Bitwise logical operations

We consider that a bit with the value 1 represents a logical value of TRUE and a bit with the value 0 represents the logical value FALSE. Having established these conventions, we can describe the well-known logic tables for the logical operations: AND, OR, XOR and NOT.

|  |  |  |
| --- | --- | --- |
| **AND** | 0 | 1 |
| 0 | **0** | **0** |
| 1 | **0** | **1** |

|  |  |  |
| --- | --- | --- |
| **OR** | 0 | 1 |
| 0 | **0** | **1** |
| 1 | **1** | **1** |

|  |  |  |
| --- | --- | --- |
| **XOR** | 0 | 1 |
| 0 | **0** | **1** |
| 1 | **1** | **0** |

|  |  |
| --- | --- |
| **NOT** |  |
| 0 | **1** |
| 1 | **0** |

The IA-32 assembly instructions that perform these logical bitwise operations are detailed below:

**and** *operand1, operand2*

**or** *operand1, operand2*

**xor** *operand1, operand2*

**not** *operand1*

where *operand1* and *operand2* are either registers, memory references (i.e. variables) or constants (*operand1* can not be a constant!) both of the same size/type: byte, word, doubleword.

Each single bit of *operand2* is and/or/xor with each corresponding bit from *operand1*. The *not* operation is a unary operation, so the bits of *operand1* are modified directly.

Examples:

Let the *AL register* be 1111 0000b. For each instruction below, AL is considered to have the initial value 1111 0000b.

*and al, 0011 1100b* => *AL:=0011 0000b*

*or al, 0011 1100b* => *AL:=1111 1100b*

*xor al, 1010 1010b* => *AL:=0101 1010b*

*not al* => *AL:=0000 1111b*

Shift and rotate operations

*shl a, n* : (Shift Logic Left) moves the bits of *a* with *n* positions to the left; *n* bits from

the left side are lost and *n* zero bits are added to the right side

*shr a, n* : (Shift Logic Right) moves the bits of *a* with *n* positions to the right; *n* bits from

right side are lost and *n* zero bits are added to the right side

*sal a, n* : (Shift Arithmetic Left) identical to *shl*

*sar a, n* : (Shift Arithmetic Right) similar to *shr*, but the sign bit of *a* (not zeroes) is added

on the left side *n* times

In the above shift instructions, *a* can be a register or a memory reference (i.e. variable) on a byte, word or doubleword and *n* can be the register CL or a constant (smaller than 31).

Rotate operations are just like shifts, but the bit that exits the bit configuration on one side enters the bit configuration on the other side (it is not lost like in the case of shifts!).

*rol a, n* : (Rotate Left) rotate the bits of *a* with *n* positions to the left

*ror a, n* : (Rotate Left) rotate the bits of *a* with *n* positions to the right

*rcl a, n* : (Rotate with Carry Flag to the Left) rotate the bits obtained from concatenating

CF and the bits of *a* with *n* positions to the left

*rcr a, n* : (Rotate with Carry Flag to the Right) rotate the bits obtained from concatenating

the bits of *a* and CF with *n* positions to the right.

In the above rotate instructions, *a* can be a register or a memory reference (i.e. variable) on a byte, word or doubleword and *n* can be the register CL or a constant (smaller than 31).

Examples:

Let the *AL register* be 1111 0000b. Before each instruction below, AL is considered to have the initial value 1111 0000b.

shl al, 1 => AL:=1110 0000b

shr al, 2 => AL:=0011 1100b

sal al, 1 => AL:=1110 0000b

sar al, 2 => AL:=1111 1100b

rol al, 1 => AL:=1110 0001b

ror al, 1 => AL:=0111 1000b

**Observation:** After every instruction from above, the last bit that exists the bit configuration is always stored in CF also. For example if 1000 0000b is rotated with 1 position to the left the result will be 0000 0001b and CF=1.

**Observation:** No matter what the value *x* is (either 0 or 1), we have the following rules:

|  |
| --- |
| x OR 0 = x |
| x OR 1 = 1 |
| x AND 0 = 0 |
| x AND 1 = x |

Ex.1. Being given a byte A, construct a new byte B in the following way:

* bits 0-2 of B should be equal to bits 0-2 of A
* bits 3-4 of B should be set to 1
* bits 5-7 of B should be equal to bits 2-4 of A

bits 32

global start

extern exit

import exit msvcrt.dll

segment data use32 class=data

a db 11110101b

b db 0

segment code use32 class=code

start:

; bits 0-2 of B should be equal to bits 0-2 of A

mov al, [a] ; AL:=1111 0101b

and al, 0000 0111b ; AL:=0000 0101b (we isolate bits 0-2 of AL

; we leave the bits 0-2 of AL unchanged and set the

; other bits to zero

or [b], al ; b:=0000 0101b

; bits 3-4 of B should be set to 1

or byte [b], 0001 1000b ; we set the bits 3 and 4 of B to one and

; leave the other bits unchanged

; b:=0001 1101b

; bits 5-7 of B should be equal to bits 2-4 of A

mov al, [a] ; AL:=1111 0101b

shl al, 3 ; shift with 3 position to the left so that bits 2-4

; arrive on positions 5-7

and al, 1110 0000b ; AL:=1110 0000b (we isolate bits 5-7 of AL

; we leave the bits 5-7 of AL unchanged and set the

; other bits to zero

or [b], al ; b:=1111 1101b

push dword 0

call [exit]

**IV.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

The conditional jump instructions check the value of specific flags and depending on this 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.

Conditional jump instructions that consider the numbers unsigned

jb label : (Jump if below) jumps to label if a<b

jbe label : (Jump if below or equal) jumps to label if a<=b

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

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

jle label : (Jump if less or equal) jumps to label if a<=b

jnl label : (Jump if not less) jumps to label if 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

jge label : (Jump if greater or equal) jumps to label if a>=b

jng label : (Jump if not greater) jumps to label if 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.

Unconditional jump instruction

**jmp** label : always jumps to the specified label

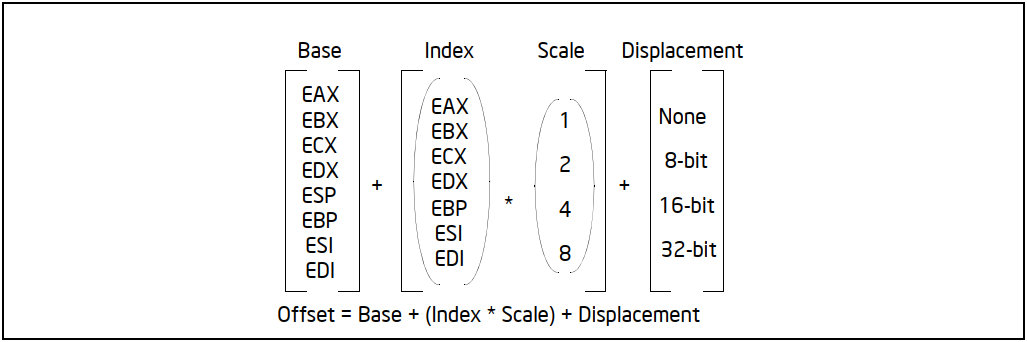
**IV.3 Working with strings of bytes**

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 don 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*

Ex1. Being given a string of bytes containing lowercase letters, build a new string of bytes containing the corresponding uppercase letters.

**Variant 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:

97

98

101

100

102

0

0

0

0

0

0

99

s1

s1+1

s1+4

s1+3

s1+5

s2+5

s2+4

s2+3

s2+2

s2+1

s2

s1+2

; 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]

**Variant 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]

**Variant 3:**

We solve the problem using string instructions.

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

mov edi, s2

mov ecx, lenS1

cld

repeat:

lodsb ; mov al, [esi] + inc esi

sub al, 'a' -'A'

stosb ; mov [edi], al + inc edi

loop repeat ; is equivalent to these 3 instructions:

; dec ecx

; cmp ecx, lenS1

; jb repeat

push dword 0

call [exit]

The string instructions have all default operands and they work in the following pattern: they do something with the current element of the string(s) and they move to the next element in the string(s). In order to work with string instructions, we must initially:

* set the offset of the source string in ESI (the source string is the one we do not modify)
* set the offset of the destination string in EDI (the destination string is the one we modify)
* set the parsing direction (rom. directia de parcurgere) of strings; if the Direction Flag DF=0 strings are parsed from left to right and if DF=1 strings are parsed from right to left

Some string instructions work only with the source string, some others work only with the destination string and some others work with both.

**String instructions for data transfer**

(Load String of Bytes)

**1. LODSB** AL🡨 <DS:ESI>

if DF=0 inc(ESI) else dec(ESI)

(Load String of Words)

**2. LODSW** AX🡨 <DS:ESI>

if DF=0 ESI🡨ESI+2 else ESI🡨ESI-2

(Store String of Bytes)

**3. STOSB** <ES:EDI>🡨 AL

if DF=0 inc(EDI) else dec(EDI)

(Store String of Words)

**4. STOSW** <ES:EDI>🡨 AX

if DF=0 EDI🡨EDI+2 else EDI🡨EDI-2

(Move String of Bytes)

**5. MOVSB** <ES:EDI>🡨 <DS:ESI>

if DF=0 {inc(ESI); inc(EDI)} else {dec(ESI); dec(EDI)}

(Move String of Words)

**6. MOVSW** <ES:EDI>🡨 <DS:ESI>

if DF=0 {ESI🡨ESI+2; EDI🡨EDI+2} else {ESI🡨ESI-2; EDI🡨EDI-2}

**String instructions for data comparisons**

(Scan String of Bytes)

**7. SCASB** CMP AL, <ES:EDI>

if DF=0 inc(EDI) else dec(EDI)

(Scan String of Words)

**8.** **SCASW** CMP AX, <ES:EDI>

if DF=0 EDI🡨EDI+2 else EDI🡨EDI-2

(Compare String of Bytes)

**9.** **CMPSB** CMP <DS:ESI>, <ES:EDI>

if DF=0 {inc(ESI); inc(EDI)} else {dec(ESI); dec(EDI)}

(Compare String of Words)

**10.** **CMPSW** CMP <DS:ESI>, <ES:EDI>

if DF=0 {ESI🡨ESI+2; EDI🡨EDI+2}

else {ESI🡨ESI-2; EDI🡨EDI-2}

There also exist instructions LODSD, STOSD, MOVSD, SCASD, CMPSD that work with strings of doublewords, use EAX and always increment/decrement ESI and EDI by 4 bytes.