Extra material: **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]