ASC structure exercitii 3

31 ian 2020

a) explicati functionarea si efectul fiecarei dintre urm. instructiuni

1) lea eax, [6+esp]

The instruction is correct
The offset of the operand is calculated with base(esp) + constant(6)
The instruction loads in eax the address from esp+6

2) mov eax, 6+esp

SYNTAX ERROR - invalid operand type

3) movsx ax, [6+esp]

The instruction is correct. It loads in ax the value from the address 6+esp. If the value is less than a word, it is completed with the sign bit

;movsx ax, byte[x] = { mov al, byte[x]
                      { cbw

4) mov ebp, [6+ebp*2]

The instruction is correct. The operand offset is calculated with base(ebp), index(ebp) and constant(ebp)
It loads in ebp the value at 6+ebp*2 constant

5) mov [6+ebp*2], 12

SYNTAX ERROR - operation size not specified
it should be mov byte/word/dword [6+ebp*2], 12

6) mov ebp, [ebx+esp]

The instruction works

The offset formula: [esp] + [ebx]

↓ base ↓ index

7) movsx [6+esp], eax

SYNTAX ERROR: invalid combination of opcode and operands

movsx requires ~~2 registers as operands~~ a register destination, not memory

8) mov [6+esp*2], eax

~~This is ok. It loads the value from eax in the memory at 6+esp*2~~

~~offset formula~~

SYNTAX ERROR - invalid effective address

esp can't be index

9) mov [6+ebp*2], [6+esp]

SYNTAX ERROR! - operand size not specified

10) movzx eax, [6+ebp*2]

SYNTAX ERROR - operand size not specified

In order for movzx to work, the source should be less than the destination register, so it can extend it

b) Se dă urm. secvență de instrucțiuni asm

push edx / push eax / pop edx / xor dh, dh / shl edx, 16 / dec / ror edx, 16 / add edx, ebx / push edx / pop esi / lodsb / pop edx

push edx; loads edx on stack
push eax; loads eax on stack
pop edx; loads in edx the last value from the stack (eax)
xor dh, dh; initializes dh with 0
shl edx/16; moves ax in the higher part of edx
clc ; clears carry flag (CF=0)
rcr edx, 16; rotates edx with 16 bits, firstly, it puts the cf, and the last value remains in CF
CF=0 and ax is the higher part of edx
add edx, ebx; adds ebx in edx
push edx; loads edx in stak
pop esi; esi = edx practic
lodsb ; the byte from the address esi is loaded in al
pop edx; cleans the stack

~~mov al,~~

Final form (even if it is not in one single instruction):
and eax, 0FFh
mov al, [ebx + eax]

4 feb 2022

III

a1) 2 exp de instrucțiuni asm diferite pt care avem operanzi explicați sau fără de dim. diferite. Explicați

movsx ax, bl

-> ax and bl are explicit operands
-> they have different dimensions: ax - word, bl - byte
-> the instruction extends 8 bit bl to 16 bit ax by filling ah with the sign bit

movzX EAX, CX
-> the same explaination

a2) - " — amlii operanzi implicit să fie de dim. diferite

CBW
-> al, ax - implicit operands, they have different dimensions:
al - byte, ax, word
-> converts the signed byte from al into a word, filling ah with the sign bit

lodsb
-> al, esi - implicit operands, they have different dimensions
al - byte, esi - dword
-> loads into al the byte from the address <DS: ESI>

a3) ← " — amlii operanzi explicit să fie din memorie
It doesn't exist

a4) — " - amlii operanzi implicit să fie din memorie

MOVSB
-> implicit operands: the address <DS: ESI> and <DS: EDI>
-> it stores the byte from the address <DS: ESI> to the address <ES: EDI>

MOVSW
-> implicit operands: the same from MOVSB
-> it stores the word — " —

a5) - " — an op explicit și unul implicit de dimensiuni diferite

div bl
-> ax, bl : ax is the implicit operand, bl is the explicit one, ax - hword, bl - byte
-> it divides ax by bl, result is: quotient in al, rem - in ah

push ax ! you can't do push al
it works only 16, 32 bits

a6) — " — un singur operand implicit, să nu fie din memorie

CLC
- implicit operand: Carry flag
- the instruction clears the CF (CF=0)

STD
- implicit operand: Direction flag
- the instruction sets the DF (DF=1)

a7) — " — un singur operand optional

RET
-> Return from the current function / subroutine
-> with optional operand: RET 4
-> Return and cleans up the stack

a8) — " — un operand explicit și unul implicit de aceleași dimensiuni

MUL bx
-> the implicit operand: ax
-> the explicit operand: bx
-> they have the same dimension

imul bx
-> the implicit operand: ax
-> the explicit operand: bx
-> the instruction executes a multiplication between ax and bx on signed interpretation

a9) — " — 2 operanzi expliciti și unul implicit

ADC eax, ebx

-> implicit operand: CF
-> explicit operands: eax, ebx
-> the value of the carry flag is added into the sum of the two operands

SBB EDX, EBX
-> implicit: CF
-> Explicit: EDX, EBX
-> the value of the carry flag is substracted from (dest-source)

a10) — " — un singur operand implicit din memorie

RET
the implicit operand: the address on the stack
-> the instruction returns the address on the stack

28 January 2022

Which is the result of each of the above operations and instruction? Detail every line, every introduced value in b10 and b10, signed and unsigned. Overflow concept discussed

a) mov ax, 0100h
mov bx, 1000+10b
idiv bl

mov ax, 0100h

-> the instruction loads in ax register the value 0100h

• both signed and unsigned interpretations are the same for 0100h, because it is at the intersection of the unsigned interp. with the signed interp. represented on 2 bytes

0100h = 256

mov bx, 1000 + 10b

1000 + 10b = 1002

1002 : 16 = 62 r. 10 = A
62 : 16 = 3 r. 14 = E
3 : 16 = 0 r. 3

1002 = 3EAh

• represented on 2 bytes (word): 03EAh
• both signed and unsigned interp. are the same

idiv bl

• divides ax by bl on signed interpretation
ax on unsigned = 0100h
bl on unsigned = EAh on unsigned = 1110 1010b
We should do the 2's complement

100h -
EAh
16h ← BL on signed

100h : 16h = B r. E

Because the quotient is a negative number, we should do its 2's complement to represent it

100 -
B
F5

In conclusion, for idiv bl, the quotient is F5, represented on 1 byte on AL, and the reminder is E, represented on 1 byte in Ah

b) mov ah, 0cdh
mov al, 0eb h
add ah, al

mov ah, 0cdh

loads the value 0cdh into ah

0CDh : 11001101

on signed interpretation: the sign bit is 1, so we compute its 2's complement

100-
CD
33h = 51d

on signed: 205d

mov al, 0eb h

loads the value 0ebh into al

0ebh: 1110 1011b

on unsigned: 240d

on signed: 100h-
eb h
15h = -21d

add ah, al

- performs addition of ah and al
destination: ah

how it works on unsigned:

11001101b
11101011b
1 10111000

There is a transport digit, so CF = 1 (we have overflow at signed interpretation

How it works on signed:

11001101b +	-51 +
11101011b	-21
1|10111000b	-72

10111000b - negative number

~: 0100 0111 +
0000 0001
01001000 = 48h = 72

c) mov ax, 1010h
mov bx, 1111b
mul bl

mov ax, 1010h

loads in ax the value 1010h

1010h = 0001 0000 0001 0000b

The sign bit is 0, so both signed and unsigned interpretations are the same

1010h = 4112d

mov bx, 1111b

loads in bx the value 1111b

in bx it is represented on 2 bytes

= 0000 0000 0000 1111b

signed and unsigned interpretations are the same:

00 0Fh = 16

mul bl

performs a multiplication between bl and al

destination of the result: ax

10h.
Fh
00F0h

In multiplication we never have overflow

a) mov dh, 200
mov ch, 62h
sub dh, ch

mov dh, 200

loads in dh the value 200
200 = C8h
C8h = 1100 1000b
· the sign bit is 1 so it has 2 interpretations
· unsigned is C8h
· signed: 100h
C8h
38h = -56

mov ch, 62h

loads in ch the value 62h
62h = 0110 0010b
· sign bit is 0, so both signed and unsigned interp. are the same
62h = 98

sub dh, ch

· computes dh - ch
· destination: dh

Here it works on unsigned:

1100 1000b -	200 -
0110 0010b	98
0110 0110b	102

= 66h = 102

It gives the correct result => CF = 0

(CF = 1 if we need significant digit borrowing for performing the substraction)

How it works on signed

1100 1000b -
0110 0010b
0110 0110b

We can't substract a pos. number from a neg. number and obtain a pos. number, so OF=1

27 january 2023

The following 5 ASM code sequences are given:

i) ii) iii) iv) are like the ones from the last

v) xor eax, eax
lea ebx, [esi]
xlat

a) Detail the complete effect of every source line
b) Replace the last instruction with another (also with no explicit operands) such that the output effect on al register is the same

a) xor eax, eax

fills the eax register only with 0 bits
forces the content of EAX to 0 by executing the logical operation XOR on its bits

lea ebx, [esi]

loads the effective address of the operand into the ebx register

After executing this line: ebx = esi

xlat

The effect of xlat: al = the byte found at the memory address [DS:EBX+AL]

b) In conclusion, this sequence loads in al the byte at the memory address [DS:ESI]

The last instruction can be replaced with lodsb,

because this is exactly this instruction's role

25 jan 2019

Present a classification of the following 14 instructions in a number of categories based on the identical effect on the EBX register (explained and justified)

1) lea ebx, [ebx+6] ; loads in ebx the effective address from ebx+6
2) lea ebx, [bx+6]; loads in ebx the address from bx+6
3) lea bx, [bx+6]; loads in bx the address from bx+6
4) lea bx, [ebx+6]; loads in bx the address from ebx+6
5) mov ebx, ebx+6 ; SYNTAX ERROR: invalid operand type
6) mov ebx, [ebx+6]; loads in ebx the dword in memory starting at ebx+6
7) movzx ebx, [ebx+6]; SYNTAX ERROR: the dim. of [ebx+6] should be mentioned
8) movzx ebx, [bx+6]; SYNTAX ERROR — " — [bx+6] — " —
9) add bx, 6; movzx ebx, bx ; adds 6 to bx, movzx ebx, bx is a comment, it's not executed
10) mov [ebx], dword [bx+6]; SYNTAX ERROR:
11) add ebx, 6; adds 6 to ebx
12) add bx, 6 ; adds 6 to bx
13) push [ebx+6]; pop ebx; SYNTAX ERROR: operation size not specified
14) xchg ebx, [ebx+6] ; exchanges the value from ebx with the value from ebx+6

Categories:

I Instructions that loads in bx the address from ebx+6: 4
II —" — bx+6: 3
III —" — ebx —" — ebx+6: 1, 14
IV —" — ebx —" — bx+6: 2
V —" — adds at ebx the value 6: 9, 11, 12
VI —" — loads in ebx the dword in memories: 6
VII SYNTAX ERRORS: 5, 7, 8, 10, 13

12