***x db -128,128,* -1** **y dd -128** **z dw 128** **...** **mov ax,word ptr x+1** **not ah** **neg al** **mov bx, word ptr x+2** **xchg bh,bl** **neg bx** **cbw** **xor ax,bx** **xor bx,ax** **sub ax,bx** **les di,y+1** **inc di** **push ds** **pop es** **mov ch,es:[di]** **inc di** **mov dx,es:[di]** **mov ax,-129** **cbw** **and ax,-1** **cwd** **shl ax,2** **sar al,2** **or ax,dx** **div al**	**-128=0x80, 128=0x80, -1 = 0xFF** **in ordinea in care se declara (DB)** **-128=0x80 in memorie |80 FF FF FF| se pun in memorie conform regulii little-endian = byte-ul cel mai putin semnificativ se afla la adresa cea mai mica** **conversie de pe 8 pe 32 de biti** **directiva dd aloca in memorie 32 de biti** **128 = 0x0080, memorie |80 00| conform Little endian se acola 16 biti (word) pentru reprezentarea in memorie** **ce e aia ptr?? :))) chestia aia e folosita doar de debugger pentru reprezentarea interna** **ptr = pointer** **oricum instructiunea nu e corecta** **pentru ca x+1 nu este referinta la memorie** **corect: mov ax, word PTR [x+1]** **Atentie, asta nu merge in nasm (cred, verific acum)** **merge doar mov ax, Word [x+1] (word optional)** **!! warning: `ptr' is not a NASM keyword [-w+ptr]** **neg ah ← neaga continutul lui AH cu semn. Echivalent cu (not ah, inc ah)** **de unde stiu ce este in ax? (nu poti sti pentru ca este eroare la instructiunea precedenta, poti presupune ca acolo este 0 sau valoarea lui ax daca instructiunea aia ar fi fost corecta :)))** ***x db -128,128, -1*** ***mov ax, [x+1] => ax = 0xff80, ah=ff*** ***ceva de genul***

	ah = 01h **not ah ; ah = FEh** **neg ah ; ah = -1 = FFh**

x db -128,128, -1

-128=0x80, 128=0x80, -1 = 0xFF
in ordinea in care se declara (DB)

y dd -128

-128=0x80 in memorie |80 FF FF FF| se pun in memorie conform regulii little-endian = byte-ul cel mai putin semnificativ se afla la adresa cea mai mica
conversie de pe 8 pe 32 de biti
directiva dd aloca in memorie 32 de biti

z dw 128

128 = 0x0080, memorie |80 00| conform Little endian
se acola 16 biti (word) pentru reprezentarea in memorie

>>>>>

mov ax,word ptr x+1

ce e aia ptr?? :))) chestia aia e folosita doar de debugger pentru reprezentarea interna
ptr = pointer
oricum instructiunea nu e corecta
pentru ca x+1 nu este referinta la memorie
corect: mov ax, word PTR [x+1]
Atentie, asta nu merge in nasm (cred, verific acum)
merge doar mov ax, Word [x+1] (word optional)

!! warning: `ptr' is not a NASM keyword [-w+ptr]

not ah

neaga toti bitii din registrul ah

neg al

neg ah ← neaga continutul lui AH cu semn. Echivalent cu (not ah, inc ah)

x db -128,128, -1
mov ax, [x+1] => ax = 0xff80, ah=ff
ceva de genul

ah = 01h
not ah ; ah = FEh
neg ah ; ah = -1 = FFh

mov bx, word ptr x+2

incorrect (acelasi motiv ca mai sus). Corect: mov bx, [x+2]: incarca in reg. BX word-ul de la DS:[x+2]

xchg bh,bl

interschimba valorile din cei doi registrii bh, bl

neg bx

neaga toti bitii din bx

cbw

converteste cu semn byte-ul din registrul AL intr-un word in registrul AX

xor ax,bx

realizeaza operatia xor (sau exclusiv) intre ax si bx, pune rezultatul in ax. **Seteaza ZF, PF, SF deci tre sa spun si asta :)))) Pentru completitudine, da**

completitudine n-am mai auzit de la logica :)))) Cu flagurile poti sa scoti puncte. La toate operatiile aritmetice. Daca stii valorile din registri, specifici direct ce si cum, daca nu o lasi teoretic.

xor bx,ax

0^1 = 1, 1^0 = 1, 0^0 = 0, 1^1 = 0

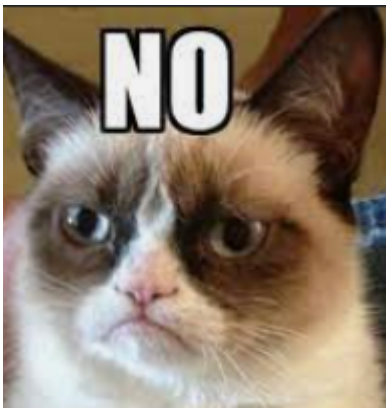
sub ax,bx

ax = ax - bx, se seteaza OF, CF, PF, AF, ZF,

les di,y+1

ce o mai fi asta :)))

PFFF LEA?? [“Load far pointer”] n-am faucet asa ceva. Next :))



inc di

Incrementeaza partea low a lui EDI (16 biti). DI=DI+1, seteaza flagurile CF, OF, AF, PF, ZF, SF

push ds

exista ds??:))) DS:EDI. DS/CS/ES/SS registrii de segment.

Se plaseaza valoarea din DS pe stiva, apoi ESP-=2.

pop es

es preia ultimul word din stiva, adica selectorul de segment al lui ds

selectorul de segment este de 16 de biti si este definit si furnizat de catre sistemul de operare in urma procesului de paginare/segmentare

(nu ai de unde sa stii asta! pop es poate pune in es orice valoare, fie ea selector de segment sau nu). Imagine: push word 0x0002, pop es.

Sa mai ai grija:

mov es, ax ; merge, aici in es se incarca valoarea de la ax (16 biti) OK

mov es, [x] ; merge, aici in es se incarca valoarea de la adresa furnizata de x (16b) da

mov es, 2 ; nu merge; de ce nu merge:(((Invalid opcode or operands) n-ai voie sa pui constante in ES. E ca la mul/div. Nu merge mul 10. ci mov al, 10. mul al

nici in DS? nici un registru de segment

mov ch,es:[di]

Incarca in CH —--- NU MERGE IN 32 DE BITI! offset-ul are 16 biti. Probabil merge pe programarea sub 16 biti.

inc di

incrementeaza di + EFLAGS?? adica INC face adunare, afecteaza si flagurile.

mov al, FFh
inc al ; ZF = 1, CF = 1 etc :)) aaa, am intelss ms

mov dx,es:[di]

incarca in dx valoarea de la adresa es:[di], es selectorul de segment, di offsetul

[Syntax error]

unde le invat pe asteaaaaa, ne-a zis asa ceva?

Nu cred ca o sa pice asa ceva. Asta era valabil pe 16 biti, dar noua nu ne-a povestit asa ceva. NASM da eroare daca scrii asta

mov ax,-129

pornind de la 129 = 128+1 = 2^7 + 2^0 = 1000 0001 = 81h

complementul sau este 7Fh

ah al

deci pe 16 biti este: FF 7Fh

cbw

converteste byteul al cu semn in word in AX
al = 7Fh
ax = 007Fh
se extinde bitul de semn care in cazul nostru este 0 in registrul ah

and ax,-1

-1 = FFFFh

ax = 007Fh

dupa and va fi ax = 007Fh

Poti adauga ca efect, masca de biti 0xFFFF nu schimba valoarea operandului pe care este aplicata. (sper ca am calculat bine) :))

cwd

Conversie cu semn de la AX la DX:AX. Se copiaza cel mai semnificativ bit al lui AX in fiecare pozitie a lui DX (conversie distructiva)

DX:AX = 0000 007Fh (sper ca am calculat bine pana acum) Presupunand ca ax era 0x007f atunci e ok :))

shl ax,2

Shift left logic cu 2 pozitii in AX. Are acelasi efect ca inmultirea lui AX cu 4. Ultimul bit shiftat se va regasi in carry flag. e.g.
mov ax, 0xC010
shl ax, 2 ; AX = (1)(1)00_0000_0001_0000 << 2 = 0000_0000_0100_0000, CF = 1
ax = 0000 0000 0111 1111 << 2 = 0000 0001 1111 1100

sar al,2

shiftare aritmetica la dreapta cu 2(%32) pozitii la dreapta, ultimul bit iesit in dreapta se pastreaza in CF si bitii din stanga se completeaza cu bitul de semn
al = 1111_1100>>2 (aritm) = 1111_1111 = 0FFh

or ax,dx

Se face sau logic intre DX si AX, rezultatul apare in AX. Flaguri afectate: ZF, PF, SF
ax = 00FFh
orice valoare ar fi in dx, or nu are niciun efect pentru al** pentru ca al este plin de 1**
deci ax = xyFFh
Aici are, pentru ca se schimba ax! Dar da, or ax, dx nu face decat sa copieze dx in ax pt ca ax = ffff? nu este 00ffh? AA ba da, deci pica toata explicatia

div al

AL = AX/AL

AH = AX%AL

deci ax = xyFFh

Ok si atunci AL=FFh = 255 unsigned.

este destul de mare incat sa nu fie overflow (zici?)

asaaa!yeeee

0xFFFF/0xFF = 65535 / 255 = 257 :)))))) overflow

Cat timp partea high e destul de mica e ok

wowowo

stai sa procesez

deci in registru de 8 biti am ca intervale admisibile

unsigned [0,255]

signed [-128, 127]

16 biti

unsigned [0, 65535]

signed [-32768, 32767]

cum in al am 255, nu?

in ax cel mai mare numar posibil poate fi FFFF adica 65535

65535/255 = ai scris mai sus de aici deci am inteles

Exact. Deci daca imparti 65535 la orice iti poate da overflow :)) (fara semn)

Dar oricum FEFF / FF = 65259/255 = 255 rest ceva, deci nu mai da overflow daca scazi partea cu 1 =)) Sansele aici ca AH sa fie fix FF sunt minuscule, deci poti sa iti asumi (doar in acest context) ca s-ar putea sa nu fie overflow =))

Deci Daca ai n biti

unsigned [0, 2^n-1]

signed [-2^(n-1), 2^(n-1)-1]

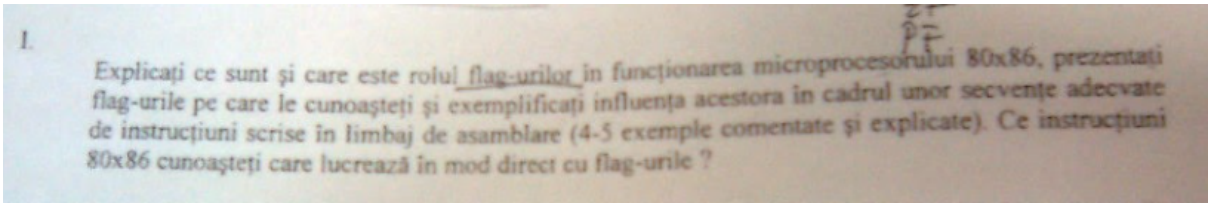
operatiile principale si ce flaguri se seteaza

flaguri: OF, DF, IF, TF, SF, ZF, AF, PF, CF

instructiune	flags
ADD SUB ADC SBB	PF, ZF, SF OF, CF
(I)MUL (I)DIV	OF,CF in conditii speciale (OF=CF=1 la depasire spatiu operanzi la inmultire), nedefinite la divide overflow

	La impartire, CF, OF, SF, ZF, AF, and PF flags are undefined.
conversiile au efect asupra flagurilor? **CBW, CWD(E), CDQ**	**Flags Affected None.**
shitari si rotiri	**SF, PF, ZF + CF care retine ultimul bit afectat**
and, or, xor, not	**SF, PF, ZF**
instructiuni de salt	**nu, astea doar folosesc flagurile**
operatii pe bytes/words/dd/qd **load** **store** **scan** **mov** **cmp (usurel)**	**ce fel de operatii? La siruri?** **Tot ce este echivalent cu MOV <ceva>, <altceva> NU MODIFICA FLAGURI** **Ce este echivalent cu CMP, modifica flaguri (face scaderea aia fictiva)**
CMP, TEST	**SF, PF, ZF** **AF, CF si OF doar la CMP**
functii de apel?	**nu, in schimb modifica resursele volatile (eax, care mai erau:)))** **(eax, ecx, edx, eflags parca)** **asta n-are treaba cu flagurile tho** **call in sine nu afecteaza flagurile SI NICI REGISTRII, DOAR STIVA!! (si ESP-ul)** **functia pe care o apeleaza poate face ce vrea cu flagurile =))**

instructiunile care se ocupa de flaguri (in cerinta aia de la examen) ce vrei sa spui??



1.

Explicați ce sunt și care este rolul flag-urilor în funcționarea microprocesorului 80x86, prezentați flag-urile pe care le cunoașteți și exemplificați influența acestora în cadrul unor secvențe adecvate de instrucțiuni scrise în limbaj de asamblare (4-5 exemple comentate și explicate). Ce instrucțiuni 80x86 cunoașteți care lucrează în mod direct cu flag-urile ?

aaaaa ok. Ce vag formulata e :))

Flagurile sa zicem ca le scriem si asa. La instructiuni nu e prea mult de zis

cld, std - seteaza DF la 0 sau 1 si afecteaza felul in care sirurile sunt parcurse prin instructiuni de tipul movsb & stuff (daca DF=0, ESI/EDI+=1, faca DS=1, ESI/EDI-=1)

clc, cmc, stc - seteaza CF la 0, 1 sau il complementeaza
cli, sti (n-avem treaba cu ele) - flaguri de intreupere, inaccesibile pe 32+ biti

rotiri cu CF, adc, sbb lucreaza in mod direct cu CF (+1)

Jumpurile evident lucreaza cu flaguri???? Pai in functie de ele iau decizia aia

Sau nu se pune??AAAAA

si mai erau niste insturcituni SETcc

si PUSHF/POPF de mentionat sigur? ca stiu ca intrebase cineva si spunea ca sunt echivalente pe NASM pe care lucram noi

pushfd = pushf (Okk nu stiam asta, in documentatie scrie altceva =))))stiu :)))

si ca au gresit ei in teorie

ce este EFLAGS register? cum pui toate flagurile intr-un registru? sau flagurile sunt intr-un registru? EFLAGS e un registru in care unui flag ii este rezervat un singur bit. Nu poate fi modificat decat prin pushf/popf nu nu poate fi accesat direct

cum pui toate flagurile intr-un registru?

pushf
pop EAX

are sens da

deci

mov, pop, push nu modifica flaguri (Exact)

xchg? - NU

xlat e un fel de mov deci nu cred - NICI

si nici LEA

la fel si cmov - asta ce face? :)) mov daca o conditie este indeplinita

acum va si eu in tabel astea:)))

aaaa ok. Asta merge la instructiuni care folosesc flaguri

SETcc ne-a dat un exemplu cu asa ceva? Cica ceva de genul

setz AL; AL=1 daca ZF=1, AL=0 otherwise

practic muta valoarea din flag intr-un byte.

EXISTA!!!

LAHF load register AH from EFLAGS copiaza indicatorii SF, ZF, AF, PF, CF

din registrul de flaguri in bitii 7, 6, 4, 2, 0 ai registrului AH

continutul bitilor 5, 3, 1 este nedefinit

indicatorii nu sunt afectati in urma acestei operatii de transfer.

pushf(d) modifica flagurile? sau doar la transfera?

din cate tin minte le modifica

pushf parca nu modifica, doar popf

Am verificat cu Olly, nu modifica =))

nu stiu :)) dar eu am scris-o de mana nici macar nu e in documentatie Doamne fereste :)) Poate nu da asa ceva

si daca nu era destul…

SAHF(store register AH into Flags) - chiar trebuie sa ne incarcam capul cu astea? =))) pai ori facem facultate calumea ori nu Corect :))

O depasire este o conditie/situatie matematica ce exprima faptul ca rezultatul unei operatii nu a incaput in spatiul rezervat acestuia.

la nivelul procesorului si a limbajului de asamblare o depasire este o conditie/situatie matematica ce exprima faptul ca rezultatul UOE:

- **nu a incaput in spatiu rezervat acestuia**
- **acest rezultat nu apartine intervalului de reprezentare admisibil**
- **operatia efectuata este un nonsens matematic in respectiva interpretare**

(cu semn sau fara semn) si nu poate fi astfel acceptata drept o operatia matematica corecta