
**Tehnička škola Mihajlo Pupin
Bijeljina**



RAČUNARI - assembler za i8086
Zbirka zadataka

prof Lukić Nebojša dipl.ing.el.

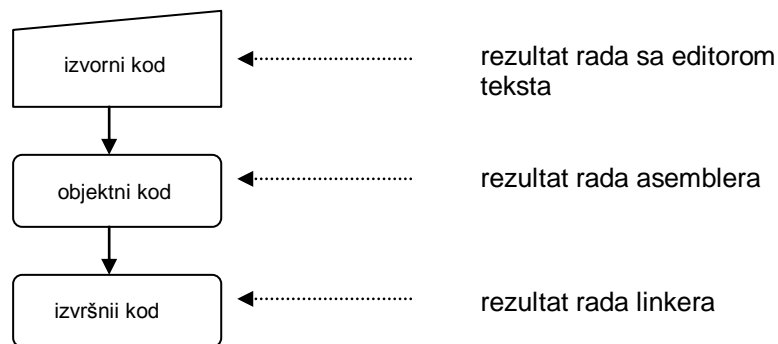
UVOD

Programiranje na simboličkom mašinskom jeziku, popularnije na asembleru je disciplina koja pored uobičajenog znanja iz programiranja, zahteva i poznavanje arhitekture računara za kojeg se piše program. Pre pisanja programa - potrebno je upoznati se sa programerskim pogledom na procesor tj na registre procesora koji su dostupni programeru, zatim sa načinima adresiranja za dotični procesor, sa načinom komunikacije sa memorijom, organizacijom ulaza-izlaza itd.

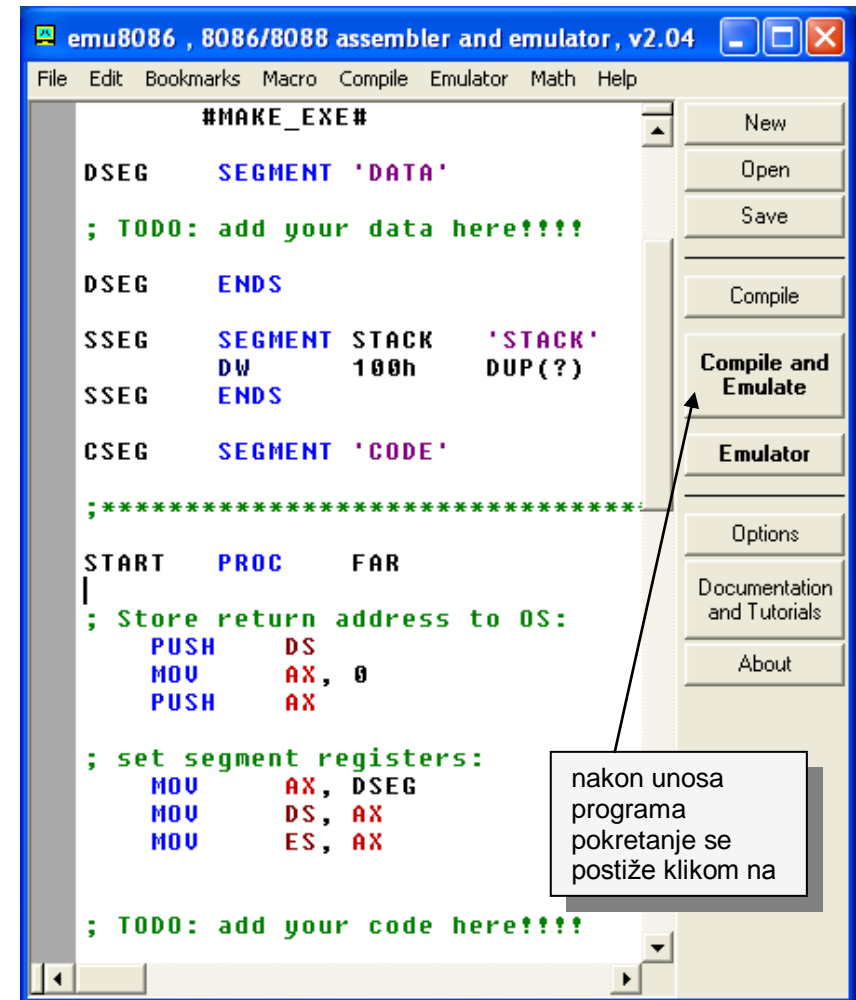
Za prethodno nabrojane teme može se koristiti literatura različitih autora koja pokriva arhitekturu 16 bitnih procesora, sa posebnim naglaskom na Intel 8086 za koji se u okviru predmeta Računari u 4. razredu izvode vežbe.

Pre pristupanja programiranju potrebno je takođe biti upoznat sa instrukcionim setom za ovaj procesor - vrstama instrukcija (aritmetičke, logičke, ...) i njihovom sintaksom. Jednom rečju potrebno je prethodno dobro upoznati teorijsku osnovu a tek zatim pristupiti programiranju.

Ovde ćemo se još jednom podsetiti da je asembler prevodilac sa simboličkog mašinskog jezika na mašinski jezik (jezik nula i jedinica). Proces koji na kraju treba da proizvede izvršni programski fajl uprošćeno teče ovako:



Sa prethodne slike se vidi da je za rad sa asemblerom potrebno raspolagati sa nekim od tekst editora, sa asemblerskim prevodiocem (obično neki od makroassemblera Microsoft-a ili turboassemblera Borland-a), i programom za linkovanje. Pošto je vrlo retko da program iz prve radi bez greške potrebno je imati i program za otklanjanje grešaka tzv debager.



S obzirom da je prvenstveni cilj edukacioni to ćemo umesto opisanog realnog procesa, raditi sa programom za emulaciju asemblerskog programiranja koji u integrisanoj formi poseduje sve prethodno nabrojane programe.

STRUKTURA ASEBLERSKOG PROGRAMA

Zbog segmentiranog pristupa memoriji struktura programa je takva da se direktivama `SEGMENT` i `ENDS` u okviru programa posebno definišu segmenti:

- podataka ,
- steka (magacina) i
- koda.

Evo primera kostura programa koji se dobije kao File->New->EXE Teplate:

TITLE 8086 Code Template (for EXE file)

```
; AUTHOR      emu8086
; DATE        ?
; VERSION     1.00
; FILE        ?.ASM
```

```
; 8086 Code Template
```

```
; Directive to make EXE output:
; #MAKE_EXE#
```

```
DSEG SEGMENT 'DATA'
```

```
; TODO: add your data here!
```

```
DSEG ENDS
```

segment
podataka

```
SSEG SEGMENT STACK 'STACK'
      DW 100h DUP(?)
SSEG ENDS
```

segment
steka

```
CSEG SEGMENT 'CODE'
```

```
START PROC FAR
```

```
; Store return address to OS:
```

```
PUSH DS
MOV AX, 0
PUSH AX
```

```
; set segment registers:
```

```
MOV AX, DSEG
MOV DS, AX
MOV ES, AX
```

```
; TODO: add your code here!
```

```
; return to operating system:
```

```
RET
START ENDP
```

```
CSEG ENDS
```

```
END START ; set entry point.
```

kod
(programski)
segment

Kao što se u gornjem primeru vidi program se sastoji od samo jedne procedure koja se zove `START`. Ona je ujedno i glavna procedura, tj ona sa kojom započinje izvršenje programa. Koja procedura će biti

glavna zavisi od toga šta (tačnije - ime koje procedure) se navede iza poslednje direktive u programu END (set entry point).

Znači pravilo za definiciju segmenta je da se upotrebi par direktiva SEGMENT - ENDS uz zadavanje nekog imena. Npr ovako:

```
podaci SEGMENT
    brojac DB ?
    suma DW ?
    poruka DB ' Kraj programa D / N ? ','$'
podaci ENDS
```

Glavna procedura mora na svom početku imati nekoliko instrukcija koje su zadužene za postavljanje adrese povratka nakon izvršenja programa i za inicijalizaciju segmentnih registara CS,DS,ES i SS. U gornjem primeru :

```
; Store return address to OS:
PUSH DS
MOV AX, 0
PUSH AX

; set segment registers:
MOV AX, DSEG
MOV DS, AX
MOV ES, AX
```

Treba reći da nema značaja da li se piše velikim ili malim slovima. Ono što je navedeno iza znaka " ; " je komentar i ne utiče na izvršenje programa.

Gornji "kostur" programa se može koristiti kao polazna podloga za pisanje programa. Ono što treba ugraditi jeste definicija podataka na mestu označenom sa:

; TODO: add your data here!

i upisivanje odgovarajućeg koda na mestu označenom sa:

; TODO: add your code here!

INT 21H

U okviru programa koristićemo često poziv int 21h. To je poziv upućen operativnom sistemu koji kontroliše sve uređaje da za potrebe našeg programa učita znak sa tastature ili pak da ispiše string na monitoru. Šta tačno zahtevamo određeno je sadržajem registra AH u trenutku poziva prekida (interapta) int 21h. Ako je naprimer u programu napisano:

```
MOV AH, 9
LEA DX, poruka
INT 21H
```

to znači da hoćemo da se na monitoru ispiše string **poruka**.

ZAKLJUČAK

U nastavku će rešenja zadataka biti data u obliku potrebne definicije podataka i odgovarajućim kodom. Zadaci će biti takvi da kreću od onih koji se mogu realizovati prostom sekvencom instrukcija, preko onih za koje su potrebni programski skokovi, petlje do onih za koje ćemo koristiti potprograme (u vidu procedura).

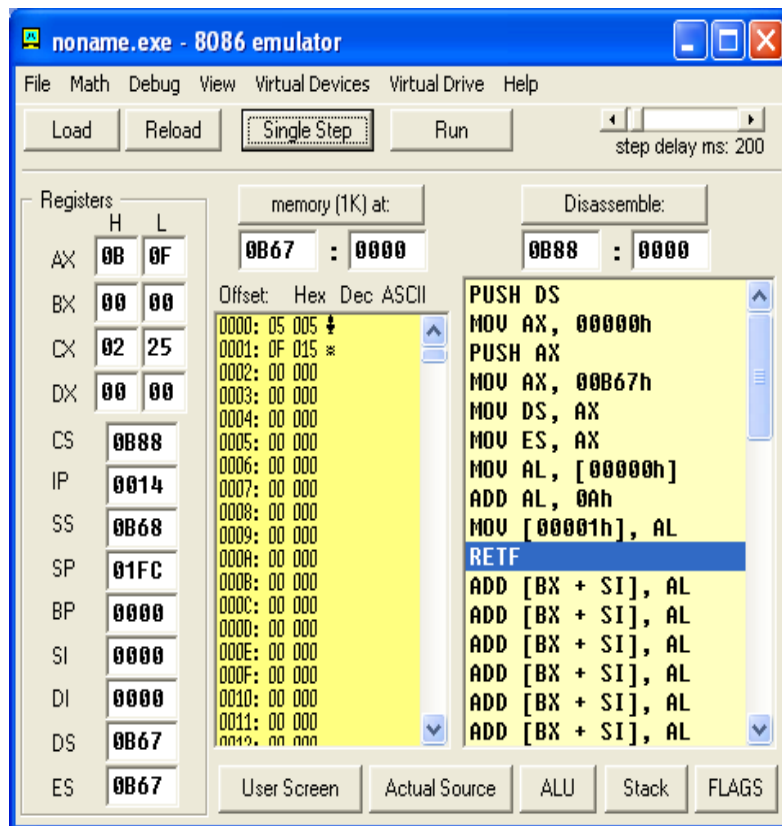
Da bi se postigao maksimalan efekat preporučljivo je samostalno raditi zadatke pogotovo što je redak slučaj da dva programera napišu vrlo sličan program. Jedino u nedostaku ideje naravno nije loše pogledati dato rešenje. Pošto je ovo zamišljeno kao pomoć u prvim koracima assemblyskog programiranja algoritmi u rešenjima često neće biti optimalni (ni sa stanovišta brzine izvršavanja ni sa stanovišta utroška resursa).

Autor

1. Napisati program kojim će se promenljivoj broj_1 dodati 10 a zatim rezultat upisati u promenljivu broj_2.

```
data SEGMENT
    broj_1 DB 5
    broj_2 DB ?
data ENDS
```

```
MOV AL, broj_1
ADD AL, 10
MOV broj_2, AL
```



Kada se pokrene izvršavanje programa onda se pojavljuje prozor kao na slici. Da bi se najbolje pratio rad programa potrebno je uzastopno klikati na taster **Single Step**. U ovom prozoru se vidi trenutni sadržaj svih registara, memorije (s tim što je potrebno da ako hoćemo da gledamo u deo memorije gde su naše promenljive, sadržaj registra DS bude upisan u ćeliju ispod tastera **memory (1K) at:**, kao na primeru sa slike koji odgovara zadatku 1), programskog koda, a u toku izvršenja otvoriće se po potrebi i prozor koji "glumi" monitor. Ako hoćemo možemo otvoriti i prozor koji npr pokazuje sadržaj ALU-a.

No da se vratimo našem zadatku. Ako pažljivije pogledate sadržaj memorije videćete da je na adresi 0000 (to je offset tj pomeraj u odnosu na segmentnu adresu segmenta podataka 0B67) sadržaj: 05 (Hex) tj 005 (Dec) tj nekakav simbol (ASCII). Šta to znači? To znači da je za računar naša promenljiva broj_1 ustvari lokacija na adresi 0B67:0000. Kako je naša promenljiva 1 bajt, već na adresi 0B67:0001 se nalazi lokacija druge promenljive tj našeg broj_2. Ako se pažljivo pogleda sadržaj u memoriji je 0F (Hex) tj 15 (Dec), što i treba da bude jer je $5+10=15$. Takođe se vidi da je u registru AL ostao rezultat sabiranja.

Jednom rečju ovaj prozor predstavlja ustvari debager.

2. Napisati program kojim će se izračunavati sledeći izraz $rez = br1 + br2 - br3$.

```
data SEGMENT
    br1 DB 3
    br2 DB 5
    br3 DB 2
    rez DB ?
data ENDS
```

```
MOV AL, br1
MOV BL, br2
ADD AL, BL
MOV BL, br3
```

```
SUB AL , BL
MOV rez , AL
```

3. Napisati program kojim će se stringovima str1 i str2 na prva dva mesta promeniti upisani znakovi (prva dva iz str1 na mesto prva dva iz str2 i obrnuto)

```
data SEGMENT
    str1 DB 'jabuka$'
    str2 DB 'kruska$'
data ENDS
```

```
MOV AL , str1
MOV AH , str1+1
MOV BL , str2
MOV BH , str2+1
MOV str1 , BL
MOV str1+1 , BH
MOV str2 , AL
MOV str2+1 , AH
```

4. Napisati program kojim se određuje da li je godina prestupna ili ne i ispisuje odgovarajuću poruku.

```
; istina sve jedinice , laž sve nule
TRUE EQU 0FFh
FALSE EQU 0
```

```
data SEGMENT
    ; zadavanje godine
    godina DW 1984
```

```
jeste DB 'Godina je prestupna !$'
nije DB 'Godina nije prestupna !$'
```

```
; logičke varijable
```

```
log1 DB ?
log2 DB ?
log3 DB ?
data ENDS
```

```
MOV log1 , TRUE
MOV log2 , TRUE
MOV log3 , TRUE
```

```
XOR DX , DX
MOV AX , godina
MOV BX , 4
DIV BX
CMP DX , 0
JE dalje1
MOV log1 , FALSE
```

```
dalje1: MOV AX , godina
MOV BX , 100
DIV BX
CMP DX , 0
JNE dalje2
MOV log2 , FALSE
```

```
dalje2: XOR DX , DX
MOV AX , godina
MOV BX , 400
DIV BX
CMP DX , 0
JE dalje3
MOV log3 , FALSE
```

```
dalje3: MOV AL , log1
MOV BL , log2
AND AL , BL
MOV BL , log3
OR AL , BL
```

; ako je konačan odgovor laž (netačno, nula)
JZ ne

MOV AH , 9
LEA DX , jeste
INT 21H
JMP kraj

ne: MOV AH , 9
LEA DX , nije
INT 21H
kraj: NOP

5. Modifikovati prethodni program tako da se godina može uneti sa tastature.

;unos hiljada
MOV AH , 1
INT 21h
SUB AL , 48
MOV BX , 1000
MUL BX
ADD godina , AX

;unos stotina
MOV AH , 1
INT 21h
SUB AL , 48
MOV BL , 100
MUL BL
ADD godina , AX

;unos desetina
MOV AH , 1
INT 21h
SUB AL , 48
MOV BL , 10
MUL BL
ADD godina , AX

;unos jedinica
MOV AH , 1
INT 21h
SUB AL , 48
ADD godina , AX

Napomena: prethodni program se može daleko elegantnije napisati korišćenjem petlje ali ćemo to ostaviti za kasnije. U prethodnom kodu se pojavljuje naredba **sub al,48**. Razlog je sledeći - naime kada se očitava tastatura od strane prekida int 21h, ASCII kod pritisnutog tastera se upiše u registar AL. Tako npr ako se pritisne taster sa oznakom 1 u registru AL će biti sadržaj 49 (dekadno odnosno 31 hexa). Kako bi smo u registru AL imali upravo onu brojnu vrednost koju smo birali sa tastature neophodno je izvršiti popravak tj oduzeti vrednost 48 (ASCII kod nule).

6. Prikazati naziv dana u zavisnosti od pritisnutog tastera od 1 do 7 . Npr za unos 1 prikazati "ponedeljak".

```
DSEG  SEGMENT 'DATA'
      ulaz  db 'broj 1-7 : $'
      novired db 10,13,$'
      greska db '* broj nije od 1 do 7 *$'
      dan1  db ' Ponedeljak$'
      dan2  db ' Utorak$'
      dan3  db ' Sreda$'
      dan4  db ' Cetvrtak$'
      dan5  db ' Petak$'
      dan6  db ' Subota$'
      dan7  db ' Nedelja$'
DSEG  ENDS
```

MOV AH , 9
LEA DX , ulaz
INT 21h
MOV AH , 1

```

INT 21h
SUB AL , 48
MOV BL , AL
MOV AH , 9
LEA DX , novired
INT 21h

MOV AL , BL
case:  MOV AH , 9
      c1: CMP AL , 1
          JNE c2
          LEA DX , dan1
          JMP end_case
      c2: CMP AL , 2
          JNE c3
          LEA DX , dan2
          JMP end_case
      c3: CMP AL , 3
          JNE c4
          LEA DX , dan3
          JMP end_case
      c4: CMP AL , 4
          JNE c5
          LEA DX , dan4
          JMP end_case
      c5: CMP AL , 5
          JNE c6
          LEA DX , dan5
          JMP end_case
      c6: CMP AL , 6
          JNE c7
          LEA DX , dan6
          JMP end_case
      c7: CMP AL , 7
          JNE else
          LEA DX , dan7
          JMP end_case
      else: LEA DX , greska
end_case: INT 21h

```

7. Uneti dva jednocifrena pozitivna broja , a zatim jedan od znakova " + - * / " . U zavisnosti od unetog znaka primeniti odgovarajuću operaciju nad dva uneta broja i prikazati rezultat.

```

BR1 EQU BL
BR2 EQU BH
ZNAK EQU CL

DSEG SEGMENT 'DATA'
    ulazbr db 10,13,'broj: $'
    ulazzn db 10,13,'znak: $'
    izlaz  db 10,13,'rezultat: $'
    r1     db '0'
    r2     db '0$'
DSEG ENDS

```

```

; unos prvog broja
MOV AH , 9
LEA DX , ulazbr
INT 21h
MOV AH , 1
INT 21h
SUB AL , 48
MOV BR1 , AL

```

```

; unos drugog broja
MOV AH , 9
LEA DX , ulazbr
INT 21h
MOV AH , 1
INT 21h
SUB AL , 48
MOV BR2 , AL

```



```

; unos znaka
MOV AH , 9
LEA DX , ulazn
INT 21h
MOV AH , 1
INT 21h
MOV ZNAK , AL

case:  XOR AX , AX
      sab: CMP ZNAK , '+'
          JNE odu
          ADD BR1 , BR2
          CALL sredi
          JMP end_case
      odu: CMP ZNAK , '-'
          JNE mno
          SUB BR1 , BR2
          CALL sredi
          JMP end_case
      mno: CMP ZNAK , '*'
          JNE del
          MOV AL , BR1
          MUL BR2
          MOV BR1 , AL
          CALL sredi
          JMP end_case
      del: CMP ZNAK , '/'
          JNE end_case
          MOV AL , BR1
          DIV BR2
          MOV BR1 , AL
          CALL sredi
end_case: NOP

MOV AH , 9
LEA DX , izlaz
INT 21h
LEA DX , r1
INT 21h

```

```

; dosadasnji kod pripada glavnoj proceduri.
; u nastavku se nalazi kod jos jedne procedure
; kojom se priprema rezultat za prikaz na izlaznoj
; jedinici (monitoru)

```

```

sredi PROC NEAR
; u BR1 se nalazi rezultat no mozda je
; on dvocifren

```

```

XOR AX , AX
MOV BR2 , 10
MOV AL , BR1
DIV BR2
; u AL je prva cifra
ADD AL , 48
MOV r1 , AL
; u AH je druga cifra
ADD AH , 48
MOV r2 , AH
RET
sredi ENDP

```

8. Napisati program kojim se unosi do devet pozitivnih jednocifrenih brojeva (ili manje ako se umesto broja samo pritisne taster ENTER) i određuje koliko je od unetih brojeva parno a koliko je neparno. Prikazati odgovarajuće poruke.

```

DSEG  SEGMENT 'DATA'
      par  db 0,'$'
      nepar db 0,'$'
      pp   db 10,13,'Broj parnih je : $'
      pn   db 10,13,'Broj neparnih je : $'
      novired db 10,13,'$'
DSEG  ENDS

```

```

NOVI:    MOV CL , 0
        CMP CL , 9
        JE KRAJ
        INC CL
        MOV AH , 9
        LEA DX , NOVIREN
        INT 21H
        MOV AH , 1
        INT 21H
        CMP AL , 13
        JE KRAJ
        SUB AL , 48
        SHR AL , 1
        JC NEPARNI
        INC PAR
        JMP DALJE
NEPARNI: INC NEPAR
DALJE:   JMP NOVI

KRAJ:    ADD PAR , 48
        ADD NEPAR , 48
        MOV AH , 9
        LEA DX , PP
        INT 21H
        LEA DX , PAR
        INT 21H
        LEA DX , PN
        INT 21H
        LEA DX , NEPAR
        INT 21H

```

U ovom zadatku se vidi da je za rešenje upotrebljena petlja kao kontrolna struktura. Registar CL je upotrebljen kao kontrolna promenljiva koja uzima vrednosti od 1 do 9 . Petlja je izvedena sa ispitivanjem na početku pa ona predstavlja ekvivalent petlji tipa FOR..TO...NEXT. Unutar gornjeg rešenja postoji još jedan mehanizam za završetak petlje (prekid petlje a da se ne izvrši svih 9 iteracija) , a to je iz uslova zadatka - ako se pritisne samo ENTER a ne broj.

Očigledno je iz prethodnih par zadataka da kontrolne strukture tipa IF....THEN...ELSE...ENDIF , CASE..OF.....ENDCASE , WHILE.....WEND , REPEAT.....UNTIL , FOR.....TO....NEXT i slične, realizujemo upotrebom instrukcija uslovnog programskog skoka kojima prethodi instrukcija CMP za poređenje dva data sadržaja.

Prilikom rada sa programskim skokovima treba uvek imati na umu jednu važnu činjenicu, ato je da **u programu nije dozvoljeno postojanje dve iste labele.** To znači da se u programu ne mogu na dva (ili više) mesta sa leve strane naći oznake (labele) , naprimer


```

KRAJ: .....
      .....
      .....
      .....
KRAJ: .....

```

Sledeći savet za upotrebu naredbi uslovnog programskog skoka je da broj programskih linija koje se preskaču bude minimalan radi lakšeg praćenja programskog koda - jednom rečju težiti ka strukturiranoj tehnici programiranja.

U nastavku ćemo videti kako se neke od osnovnih kontrolnih programskih struktura iz viših programskih jezika mogu realizovati u asemblerskom programiranju. (ovde treba reći da noviji makroasembleri imaju ugrađene neke od ovih struktura tako da se mogu koristiti kao što se koriste u višim programskim jezicima)

 IF...(var1=var2)...THEN.....ENDIF

```

      IF : CMP  var1 , var2
          JNE  ENDIF
THEN : .....
      .....
ENDIF : NOP

```


Labela može postojati i ako se nigde u programu ne vrši skok na nju, ako je to potrebno iz nekog razloga programeru. (prilikom prevođenja programa prevodilac će samo prijaviti WARNING ERROR).

 *IF...(var1=var2)...THEN.....ELSE.....ENDIF*

```

IF : CMP  var1 , var2
    JNE  ELSE
THEN : .....
      .....
      JMP ENDIF
ELSE : .....
      .....
ENDIF : NOP


```

 *CASE..var...OF.....ELSE....ENDCASE*

```

CASE : NOP
C1 : CMP var , var_case1
    JNE C2
    .....
    .....
    JMP ENDSACE
C2 : CMP var , var_case2
    JNE C3
    .....
    .....
    JMP ENDSACE
C3 : CMP var , var_case3
    JNE ELSE
    .....
    .....
    JMP ENDSACE
ELSE : .....
      .....
ENDCASE : NOP

```

 *WHILE..(var1>var2)..DO.....WEND*

```

WHILE : CMP var1 , var2
        JNG WEND
DO : .....
      .....
      .....
        JMP WHILE
WEND : NOP

```

 *REPEAT.....UNTIL..(var1<var2)..ENDREPAET*

```

REPEAT : .....
        .....
        .....
UNTIL : CMP var1 , var2
        JGE REPEAT
ENDREPEAT : NOP

```



FOR...var1=var2..TO..var3..STEP..var4.....NEXT..ENDFOR

```

MOV var1 , var2
FOR : CMP var1 , var3
    JG ENDFOR
    .....
    .....
    .....
STEP : ADD var1 , var4
NEXT : JMP FOR
ENDFOR : NOP

```

Gornji primeri su napisani tako da bi se što lakše uočila ista funkcionalnost assemblerkog koda sa konkretnom kontrolnom strukturom. Jasno je naravno da u gornjim primerima ima labela koje nemaju nikakav značaj i mogu se jednostavno izbaciti.

Osim opisanih načina za izvođenje petlje postoji i mogućnost da se iskoristi naredba LOOP :

Decrease CX, jump to label if CX not zero.

Algorithm:

```
CX = CX - 1
if CX <> 0 then
    jump
else
    no jump, continue
```

Example:

```
include 'emu8086.inc'
```

```
MOV CX, 5
label1:
PRINTN 'loop!'
LOOP label1
```

Gore je navedeno objašnjenje naredbe LOOP iz Helpa emulatora. Ono što je iz gornjeg primera možda nejasno je linija koja glasi PRINTN 'loop!' .

Radi se o takozvanom MAKRO-u koji se nalazi u fajlu 'emu8086.inc'. Mehanizam makro-a je sledeći - na mestu u programu gde se nalazi njegovo ime prilikom prevođenja programa ugrađuje se kod njegove definicije (u ovom slučaju iz fajla emu8086.inc). Znači nešto slično potprogramu.

Inače ovaj makro < PRINTN *string* > radi prikaz stringa na monitoru i prelazi u novi red.

9. Sabrati prvih 50 prirodnih brojeva i rezultat prikazati sa odgovarajućim tekstom.

```
DSEG SEGMENT 'DATA'
    tekst db 10,13,'Suma je : $'
DSEG ENDS
```

```
; organizacija petlje upotrebom naredbe LOOP
MOV CX , 50
MOV AX , 0
MOV BX , 1
NEXT: ADD AX , BX
      INC BX
      LOOP NEXT
```

```
; određivanje cifara rezultata uz pomoć steka
; najvažnija cifra biće na vrhu steka i ona treba
; prva da se prikaže
MOV DL , 10
MOV BX , 0
PUSH BX
SLED: CMP AL , 0
      JE REZ
      DIV DL
      ; korekcija prema ASCII tabeli
      ADD AH , 48
      PUSH AX
      ; ostatak od prethodnog deljenja više ne treba
      XOR AH , AH
      JMP SLED
```

```
REZ: MOV AH , 9
      LEA DX , tekst
      INT 21H
```

```
; iskoristićemo funkciju AH=2 interapta 21h
; koja prikazuje znak čiji se ASCII kod nalazi
; u registru DL
```

```

        MOV AH , 2
        POP BX
PISI:   CMP BX , 0
        JE KRAJ
        MOV DL , BH
        INT 21h
        POP BX
        JMP PISI
KRAJ:   NOP

```

Sa stekom mora da se radi u formatu reči (WORD) i zato je kod naredbi PUSH i POP naveden registar **BX**. Generalno treba voditi računa o tome da se u programu ne dese greške uzrokovane nejednakošću dužine operanada (TYPE MISMATCH), npr ovako

```

PUSH BL ;
ADD AX , CL ;
CMP CX , AH i slično.

```

U nastavku ćemo se pozabaviti klasičnim zadacima sa petljama i nizovima (vektorima) kao što su redukcija, kompresija, rotacija, ekspanzija i sortiranje vektora.

10. Uneti niz jednocifrenih brojeva od 5 članova a zatim odrediti sumu, proizvod (a to je faktorijel broja 5 ako se unose brojevi od 1 do 5), minimum i maksimum i prikazati odgovarajuće poruke.

```

DSEG   SEGMENT 'DATA'
        niz db 5 dup(?)

```

```

        sum db ?,?,'$'
        pro db ?,?,?,'$'
        min db ?, '$'
        max db ?, '$'

```

```

        p_sum db 10,13,'Suma je : $'
        p_pro db 10,13,'Proizvod je : $'
        p_min db 10,13,'Minimum je : $'

```

```

        p_max db 10,13,'Maksimum je : $'

```

```

        p_ulaz db 'Uneti clanove niza',10,13,'$'
DSEG   ENDS

```

```

        MOV AH , 9
        LEA DX , P_ULAZ
        INT 21H

```

; unos clanova niza

```

        MOV CX , 5
        MOV SI , 0
LAB1:   MOV AH , 1
        INT 21H
        SUB AL , 48
        MOV NIZ[SI] , AL
        INC SI
        MOV AH , 2
        MOV DL , ''
        INT 21H
        LOOP LAB1

```

; odredjivanje rezultata

```

        MOV SI , 0
        MOV AX , 1
        MOV BL , 0
        MOV DH , NIZ[SI]
        MOV DL , NIZ[SI]
        MOV CX , 5
LAB2:   ADD BL , NIZ[SI]
        MUL NIZ[SI]
        CMP NIZ[SI] , DH
        JL LAB3
        MOV DH , NIZ[SI]
LAB3:   CMP NIZ[SI] , DL
        JGE LAB4

```

```

        MOV DL , NIZ[SI]
LAB4: INC SI
        LOOP LAB2

        ADD DL , 48
        MOV MIN , DL
        ADD DH , 48
        MOV MAX , DH

        ; priprema cifara rezultata za
        ; proizvod i sumu

        ; ocekujemo trocifreni rezultat za
        ; proizvod

        MOV CX , 3
        MOV SI , 2
        MOV DL , 10
LAB5: DIV DL
        ADD AH , 48
        MOV PRO[SI] , AH
        XOR AH , AH
        DEC SI
        LOOP LAB5

        ; ocekujemo dvocifreni rezultat za
        ; sumu
        MOV AX , BX
        MOV CX , 2
        MOV SI , 1
LAB6: DIV DL
        ADD AH , 48
        MOV SUM[SI] , AH
        XOR AH , AH
        DEC SI
        LOOP LAB6

        ; prikaz rezultata

```

```

        MOV AH , 9
        LEA DX , P_SUM
        INT 21H
        LEA DX , SUM
        INT 21H

        LEA DX , P_PRO
        INT 21H
        LEA DX , PRO
        INT 21H

        LEA DX , P_MIN
        INT 21H
        LEA DX , MIN
        INT 21H

        LEA DX , P_MAX
        INT 21H
        LEA DX , MAX
        INT 21H

```

11. Uneti niz od N<10 jednocifrenih brojeva a zatim iz niza izbaciti brojeve koji su neparni. Prikazati novi niz.

```

DSEG  SEGMENT 'DATA'
        n      dw ?
        niz    db 9 dup(?)

        p_n    db 'Broj clanova niza : $'
        p_niz  db 10,13,'Uneti niz : ',10,13,'$'
        p_rez  db 10,13,'Rezultat : ',10,13,'$'
DSEG  ENDS

        MOV AH , 9
        LEA DX , P_N
        INT 21H

```

```

MOV AH , 1
INT 21H
SUB AL , 48
XOR AH , AH
MOV N , AX

MOV AH , 9
LEA DX , P_NIZ
INT 21H

MOV SI , 0
MOV CX , N
LAB1:MOV AH , 1
INT 21H
MOV NIZ[SI] , AL
INC SI
MOV AH , 2
MOV DL , ''
INT 21H
LOOP LAB1

MOV DI , 0
MOV CX , N
MOV SI , 0
LAB2:MOV BL , NIZ[SI]
SHR BL , 1
JC LAB3
MOV AL , NIZ[SI]
MOV NIZ[DI] , AL
INC DI
LAB3:INC SI
LOOP LAB2

MOV N , DI

CMP N , 0
JE LAB5

MOV AH , 9

```

```

LEA DX , P_REZ
INT 21H

MOV CX , N
MOV SI , 0
MOV AH , 2
LAB4:MOV DL , NIZ[SI]
INT 21H
MOV DL , ''
INT 21H
INC SI
LOOP LAB4

LAB5:NOP

```

12. Uneti niz od N<10 brojeva , a zatim izvršiti rotiranje članova niza oko simetralnog člana. Prikazati rezultatni niz.

```

DSEG  SEGMENT 'DATA'
      n  dw ?
      niz db 9 dup(?)

      p_n  db 'Broj članova niza : $'
      p_niz db 10,13,'Uneti niz : ',10,13,$'
      p_rez db 10,13,'Rezultat : ',10,13,$'
DSEG  ENDS

```

```

MOV AH , 9
LEA DX , P_N
INT 21H

MOV AH , 1
INT 21H
SUB AL , 48
XOR AH , AH

```

```

MOV N , AX

MOV AH , 9
LEA DX , P_NIZ
INT 21H

MOV SI , 0
MOV CX , N
LAB1:MOV AH , 1
INT 21H
MOV NIZ[SI] , AL
INC SI
MOV AH , 2
MOV DL , ''
INT 21H
LOOP LAB1

MOV AX , N
MOV BL , 2
DIV BL
XOR AH , AH

MOV CX , AX
MOV SI , 0
MOV DI , N
DEC DI
LAB2:MOV DL , NIZ[SI]
MOV DH , NIZ[DI]
MOV NIZ[SI] , DH
MOV NIZ[DI] , DL
INC SI
DEC DI
LOOP LAB2

MOV AH , 9
LEA DX , P_REZ
INT 21H

MOV SI , 0
MOV AH , 2

```

```

LAB3:MOV DL , NIZ[SI]
INT 21H
MOV DL , ''
INT 21H
INC SI
CMP SI , N
JL LAB3

```

13. Uneti niz od $N < 9$ brojeva , a zatim izvršiti umetanje broja 0 na k-tu poziciju u nizu ($0 < k < N-1$). Prikazati rezultatni niz.

```

DSEG  SEGMENT 'DATA'
n      dw ?
k      dw ?
niz    db 9 dup(?)

p_n    db 'Broj clanova niza (< 9): $'
p_k    db 10,13,'Pozicija za umetanje : $'
p_niz  db 10,13,'Uneti niz : ',10,13,$'
p_rez  db 10,13,'Rezultat : ',10,13,$'
DSEG  ENDS

```

```

MOV AH , 9
LEA DX , P_N
INT 21H

```

```

MOV AH , 1
INT 21H
SUB AL , 48
XOR AH , AH
MOV N , AX

```

```

MOV AH , 9
LEA DX , P_K
INT 21H

```



```
MOV AH , 1
INT 21H
SUB AL , 48
DEC AL
XOR AH , AH
MOV K , AX

MOV AH , 9
LEA DX , P_NIZ
INT 21H

MOV SI , 0
MOV CX , N
LAB1:MOV AH , 1
INT 21H
MOV NIZ[SI] , AL
INC SI
MOV AH , 2
MOV DL , ''
INT 21H
LOOP LAB1

MOV SI , N
DEC SI

LAB2:MOV DL , NIZ[SI]
MOV NIZ[SI+1] , DL
DEC SI
CMP SI , K
JGE LAB2
INC SI
MOV NIZ[SI] , 48

MOV AH , 9
LEA DX , P_REZ
INT 21H

MOV SI , 0
```

```
MOV AH , 2
LAB3:MOV DL , NIZ[SI]
INT 21H
MOV DL , ''
INT 21H
INC SI
CMP SI , N
JLE LAB3
```

14. Uneti niz od N < 10 brojeva , a zatim izvršiti sortiranje u opadajućem redosledu. Prikazati rezultatni niz.

```
DSEG  SEGMENT 'DATA'
n      dw ?
niz    db 9 dup(?)

p_n     db 'Broj clanova niza : $'
p_niz  db 10,13,'Uneti niz : ',10,13,$'
p_rez  db 10,13,'Rezultat : ',10,13,$'
DSEG  ENDS
```

```
MOV AH , 9
LEA DX , P_N
INT 21H

MOV AH , 1
INT 21H
SUB AL , 48
DEC AL
XOR AH , AH
MOV N , AX
```

```

MOV AH , 9
LEA DX , P_NIZ
INT 21H

MOV SI , 0
LAB1:MOV AH , 1
INT 21H
MOV NIZ[SI] , AL
INC SI
MOV AH , 2
MOV DL , ''
INT 21H
CMP SI , N
JLE LAB1

; uredjivanje niza

MOV SI , 0
LAB2:MOV SI , N
JE EXT1
MOV DI , SI      ; ugrađeni algoritam za
INC DI           ; za sortiranje je
LAB3:MOV DI , N   ; for i=1 to n-1
JG EXT2          ;   for j=i+1 to n
MOV AH , NIZ[SI] ;   if niz[i]<niz[j]
MOV AL , NIZ[DI] ;       swap(niz[i],niz[j])
CMP AH , AL
JNL NOSW
MOV NIZ[SI] , AL
MOV NIZ[DI] , AH
NOSW:INC DI
JMP LAB3
EXT2:INC SI
JMP LAB2
EXT1:NOP

MOV AH , 9
LEA DX , P_REZ
INT 21H

```

```

MOV SI , 0
MOV AH , 2
LAB4:MOV DL , NIZ[SI]
INT 21H
MOV DL , ''
INT 21H
INC SI
CMP SI , N
JLE LAB4

```

15. Napisati program za izračunavanje i prikaz tabele faktoriijela prvih 8 prirodnih brojeva. Faktorijele računati rekursivnim postupkom.

```

DSEG  SEGMENT 'DATA'
        poruka db 'Rekursivno racunanje faktoriijela$'
        dodatak db '! = $'
DSEG  ENDS

```

```

LEA DX , PORUKA
CALL PRIKAZ
CALL NOVIREN
MOV BX , 8
CALL FAKT

```

```

FAKT PROC
        CMP BX , 0 ; da li je brojač 0 ?
        JLE ISPIS ; ako da - izlaz
        PUSH BX   ; pamti se broj preko steka
        DEC BX    ; umanjeње za 1 i ponovo
        CALL FAKT ; poziv procedure

```

```

POP BX          ; vraća se broj iz steka
MOV AX , BX     ; upisuje se u ax i
MUL SI          ; množi sa (si)
MOV SI , AX     ; rezultat množenja ide u si
MOV AX , BX     ; broj ponovo ide u ax i
CALL IZLAZ      ; stampa se
MOV AX , SI     ; faktorijel broja ide u ax i
CALL STAMPA     ; štampa se
CALL NOVIRED
RET
ISPIS:MOV SI , 1 ; granični uslov za rekurziju
RET
FAKT ENDP

```

```

STAMPA PROC
MOV DX , 0
MOV BX , 10
DIV BX
PUSH DX
CMP AX , 0
JZ UPIS
CALL STAMPA
UPIS:POP DX
ADD DL , 48
CALL DISPLEJ
RET
STAMPA ENDP

```

```

IZLAZ PROC
MOV AX , BX
CALL STAMPA
LEA DX , DODATAK
CALL PRIKAZ
RET
IZLAZ ENDP

```

```

PRIKAZ PROC
MOV AH , 9
INT 21H
RET

```

```

PRIKAZ ENDP

```

```

NOVIRED PROC
MOV DL , 10
CALL DISPLEJ
MOV DL , 13
CALL DISPLEJ
RET
NOVIRED ENDP

```

```

DISPLEJ PROC
MOV AH , 2
INT 21H
RET

```

```

DISPLEJ ENDP

```

Napomena: u ovom zadatku se rekurzija koristi u dve procedure - FAKT i STAMPA. Uslov da bi se mogla primeniti rekurzija je postojanje kontrole nad memorijom tipa Steka. Pošto je to dostupno u asemblerskom programiranju to je moguće koristiti rekurzivne procedure.

16. Napisati program za grafički prikaz funkcije $y = k * x$. Koeficijent k treba da bude $1 / q$, gde se vrednost q unosi, a vrednost nezavisno promenljive x se kreće u intervalu od 1 do 15. Koristiti makro *GOTOXY col , row* (za pozicioniranje kursora) iz biblioteke *emu8086.inc*.

```

DSEG SEGMENT 'DATA'
    znak db '*'
    vl   db '|'
    hl   db '-'

    koef db ?
    p_n  db 'Funkcija y=k*x$'
    p_u  db 'Vrednost koeficijenta k=1/$'

```

```
naslov db 'G R A F$'  
naslov1 db 'F U N K C I J E$'  
DSEG ENDS
```

```
; naslov  
;=====  
gotoxy 41 , 5  
mov ah , 9  
lea dx , naslov  
int 21h  
gotoxy 38 , 7  
lea dx , naslov1  
int 21h
```

```
; horizontalna osa  
;=====  
mov cx , 40  
mov ah , 2  
mov dl , hl  
gotoxy 10 , 15  
l1: int 21h  
loop l1  
gotoxy 49 , 16  
mov dl , 'x'  
int 21h
```

```
; vertikalna osa  
;=====  
mov cx , 15  
mov ah , 2  
mov dl , vl  
mov bl , 1  
gotoxy 10 , bl  
l2: int 21h  
inc bl  
gotoxy 10 , bl
```

```
loop l2  
gotoxy 8 , 1  
mov dl , 'y'  
int 21h
```

```
call ulaz
```

```
; funkcija  
;=====  
mov bl , 15  
mov bh , 10  
mov dl , znak  
mov cx , 15  
gotoxy bh , bl  
l3: int 21h  
inc bh  
call vr_fn  
gotoxy bh , bl  
loop l3
```

```
vr_fn proc near  
push ax  
push dx  
mov al , bh  
sub al , 10  
mov dl , 1  
mul dl  
mov dl , koef  
div dl  
mov bl , 15  
sub bl , al  
pop dx  
pop ax  
ret  
vr_fn endp
```

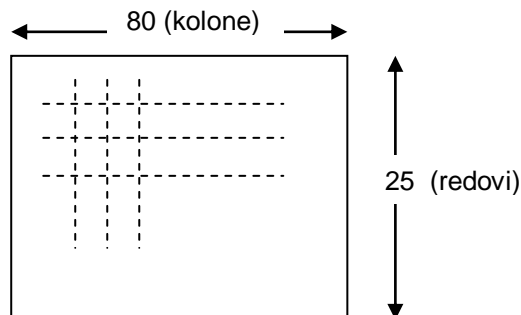
```
ulaz proc near  
push ax
```

```

push dx
gotoxy 10 , 18
mov ah , 9
lea dx , p_n
int 21h
gotoxy 10 , 20
lea dx , p_u
int 21h
mov ah , 1
int 21h
sub al , 48
mov koef , al
pop dx
pop ax
ret
ulaz endp

```

Napomena: U tekstualnom režimu rada monitor se posmatra kao matrica ćelija u kojima se može naći neki od karaktera (slova, brojevi, specijalni znaci). Svaka ćelija određena je sa dve vrednosti - kolonom u kojoj se nalazi i redom u kom se nalazi. Broj redova je 25 , a broj kolona je 80. Ćelija određena parom (0,0) nalazi se u gornjem levom uglu, a ćelija (24,79) u donjem desnom uglu.



Makro gotoxy omogućava da se znak ili prvi od više znakova (ako je to string) pojavi na onom mestu na monitoru koje je određeno vrednostima koje se navedu kao parametri poziva makro-a, npr

gotoxy 10,10 će proizvesti da se znak pojavi na monitoru u preseku 10-te kolone i 10-og reda.

17. Napisati program za približno izračunavanje kvadratnog korena. Broj čiji se koren traži upisati u segmentu podataka a rezultat zapisati u varijablu "kor". Pratiti korak po korak izvršenje programa i proveriti rezultat uvidom u sadržaj memorije.

```

DSEG  SEGMENT 'DATA'
        broj dw 80
        kor db ?
DSEG  ENDS

```

```

MOV KOR , 1
MOV BL , 1
XOR BH , BH
PON: MOV AX , BX
      MUL KOR
      CMP AX , BROJ
      JAE KRAJ
      INC BL
      INC KOR
      JMP PON
KRAJ:NOP

```

ZADACI ZA VEŽBANJE

1. Napisati program za sabiranje niza jednobajtnih brojeva. Broj članova niza i članovi niza su zadati u segmentu podataka.
2. Napisati program za izračunavanje:

$$S = \sum_{k=1}^n K^k x^{2k}$$

gde su x i n dati jednobajtni brojevi.

3. Dati su prirodni brojevi n i k i niz celih brojeva x_1, x_2, \dots, x_n . Napisati program za određivanje k najvećih članova datog niza. ($k < n$).
4. Dato je 20 celih brojeva. Napisati program za formiranje 2 niza čiji su elementi:
 - (1. niz) $x_1, x_{11}, x_3, x_{13}, \dots, x_9, x_{19}$
 - (2. niz) $x_{12}, x_2, x_{14}, x_4, \dots, x_{20}, x_{10}$
5. Napisati program za određivanje n-tog člana Fibonačijevog niza:

$$u_0 = 0, \quad u_1 = 1, \quad u_n = u_{n-1} + u_{n-2} \quad (n = 2, 3, 4, \dots)$$
6. Napisati program za formiranje niza Y na sledeći način:

$$y_i = \begin{cases} \frac{x_i + x_{n-i+1}}{2}, & n = 2k \\ x_{2i+1}, & n = 2k+1 \end{cases}$$

od elemenata niza x_1, x_2, \dots, x_n . Deljenje izvesti kao celobrojno.

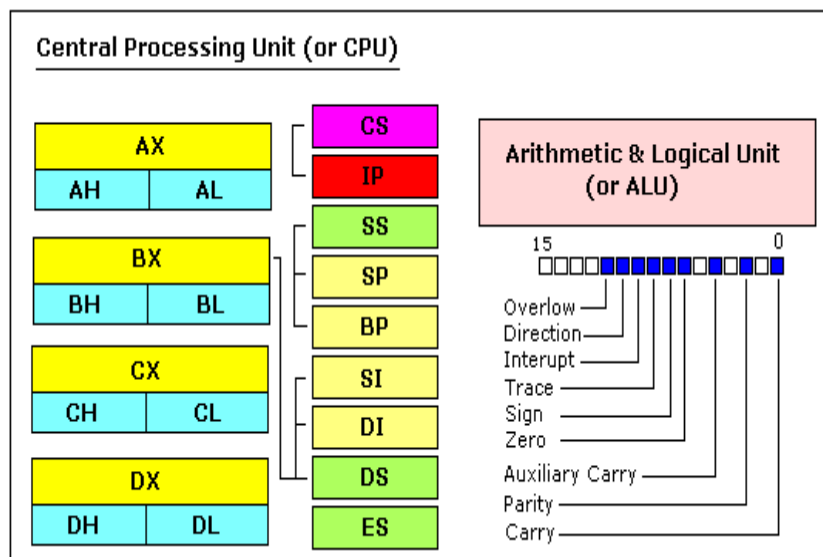
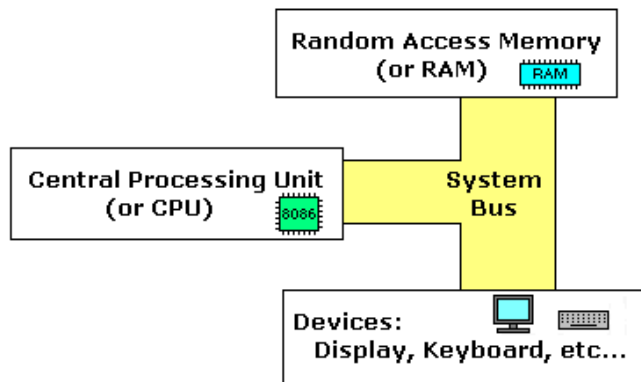
7. Za dati prirodni broj N i niz celih brojeva x_1, x_2, \dots, x_n , napisati program za izračunavanje:

$$\sum_{1 \leq i < j \leq n} |x_i - x_j|$$

8. Dat je prirodni broj k i k celih brojeva a_1, a_2, \dots, a_k . Neka je M najveći, a m najmanji u datom nizu. Napisati program za pronalaženje svih celih brojeva iz intervala (m, M) koji ne pripadaju datom nizu celih brojeva.
9. Dati su celi brojevi a_1, a_2, \dots, a_{10} . Ako važi: $b_1 = a_1, \dots, b_{10} = a_{10}$ i $b_k = b_{k-1} + b_{k-2} + \dots + b_{k-10}$ za $k = 11, 12, \dots, m$ (m je dati prirodni broj), izračunati članove niza b.
10. Dat je prirodan broj N. Napisati program za određivanje prirodnog broja iz intervala [1, N] koji ima najveći broj faktora.
11. Dat je kvadrat [(0,0), (0,10), (10,10), (10,0)] i tačka M(x,y) gde su x i y dati brojevi. Napisati program za određivanje kvadrata minimalnog rastojanja date tačke od ruba datog kvadrata.
12. Dati su prirodan broj n i kvadratna matrica A formata nxn. Matrica A je zapisana u memoriji u obliku jednodimenzionog niza po kolonama. Napisati program za prikaz te matrice u originalnom obliku.
13. Uneti nisku u memoriju završivši je znakom <CR>. Napisati program za ispitivanje da li je data niska palindrom.
14. Uneti nisku u memoriju završivši je znakom <CR>. Napisati program za određivanje broja blanko znakova.
15. Uneti nisku u memoriju završivši je znakom <CR>. Napisati program za određivanje broja samoglasnika.
16. Uneti nisku u memoriju završivši je znakom <CR>. Napisati program za određivanje broja reči.

PRILOG A

Arhitektura CPU-a



8086 CPU ima 8 registara opšte namene :

- **AX** - the accumulator register (divided into **AH / AL**).
- **BX** - the base address register (divided into **BH / BL**).
- **CX** - the count register (divided into **CH / CL**).
- **DX** - the data register (divided into **DH / DL**).
- **SI** - source index register.
- **DI** - destination index register.
- **BP** - base pointer.
- **SP** - stack pointer.

Segmentni registri :

- **CS** - points at the segment containing the current program.
- **DS** - generally points at segment where variables are defined.
- **ES** - extra segment register, it's up to a coder to define its usage.
- **SS** - points at the segment containing the stack.

Segment registri imaju specijalnu namenu - pokazuju na blokove memorije kojima se može prići.

Segment registri deluju zajedno sa registrima opšte namene da bi se omogućio pristup memoriji. Npr ako želimo prići memoriji na fizičkoj adresi **12345h** (hexadecimal), trebalo bi da sadržaj registara bude **DS = 1230h** i **SI = 0045h**. Ovako se može pristupiti mnogo većem broju lokacija nego upotrebom samo jednog 16-bitnog registra. CPU izračunava fizičku adresu tako što pomnoži sadržaj segment registra sa 10h i na to doda vrednost iz registra opšte namene ($1230h * 10h + 45h = 12345h$):

```

  12300
+ 0045
-----
 12345

```

Adresa formirana sa 2 registra zove se **effective address**. Po default-u **BX**, **SI** i **DI** registri rade sa **DS** segment registrom;

BP i **SP** rade sa **SS** segment registrom. Ostali registri opšte namene NE MOGU formirati efektivnu adresu!

BX može formirati efektivnu adresu, ali **BH** i **BL** ne mogu!

Registri specijalne namene :

- **IP** - the instruction pointer.
- **Flags Register** - determines the current state of the processor. (određuju tekuće stanje procesora)

IP registar uvek radi zajedno sa **CS** segment registrom i pokazuje na tekuće izvršavanu instrukciju. **Flags Register** se menja automatski posle operacija CPU-a, što omogućava da se utvrdi tip rezultata i uslovi za transfer kontrole na druge delove programa. Generalno ne može se prići ovim registrima direktno.

Pristup memoriji

Za pristup memoriji možemo koristiti ova četiri registra: **BX**, **SI**, **DI**, **BP**.

Kombinovanjem ovih registra unutar [] simbola, možemo prići različitim memorijskim lokacijama. Sledeće kombinacije su podržane (addressing modes (načini adresiranja)):

[BX + SI] [BX + DI] [BP + SI] [BP + DI]	[SI] [DI] d16 (variable offset only) [BX]	[BX + SI] + d8 [BX + DI] + d8 [BP + SI] + d8 [BP + DI] + d8
[SI] + d8 [DI] + d8 [BP] + d8 [BX] + d8	[BX + SI] + d16 [BX + DI] + d16 [BP + SI] + d16 [BP + DI] + d16	[SI] + d16 [DI] + d16 [BP] + d16 [BX] + d16

d8 - stays for 8 bit displacement (8-mo bitni pomeraj).

d16 - stays for 16 bit displacement (16-to bitni pomeraj).

Displacement (razmeštaj, pomeraj) može biti neposredna vrednost ili offset varijable, ili pak oboje. Do kompajlera je da sračuna jedinstvenu neposrednu vrednost.

Displacement može biti unutar ili spolja [] simbola, sa istim efektom.

Displacement je **signed** (označena) vrednost, pa može biti i pozitivne i negativne vrednosti.

Npr, pretpostavimo **DS = 100**, **BX = 30**, **SI = 70**. Sledeći adresni način:

[BX + SI] + 25

je od strane CPU-a preračunat u sledeću fizičku adresu:

100 * 16 + 30 + 70 + 25 = 1725.

Po default-u **DS** segment registar se koristi za sve modove izuzev onih sa **BP** registrom, za koje se koristi **SS** segment registar.

Postoji lagan način da se zapamte sve moguće kombinacije uz pomoć tabele:

BX	SI	+ disp
BP	DI	

Validne kombinacije se dobiju ako se uzme jedna stavka iz svake kolone ili nijedna. Npr **BX** i **BP** nikad ne idu zajedno. **SI** i **DI** takođe. Sledeći primer je ispravan: **[BX+5]**.

☞ Vrednost u segment registru (CS, DS, SS, ES) se naziva "**segment**", a vrednost iz registara (BX, SI, DI, BP) se naziva "**offset**". Kada DS sadrži **1234h** a SI sadrži **7890h** to se može zabeležiti kao **1234:7890**. Fizička adresa će biti $1234h * 10h + 7890h = 19BD0h$.

Da bi se kompajler obavestio na šta će pokazivati pokazivač, koriste se sledeći prefiksi :

BYTE PTR - for byte.

WORD PTR - for word (two bytes).

Npr:

BYTE PTR [BX] ; byte access.

or

WORD PTR [BX] ; word access.

Variable (promenljive)

Varijabla je memorijska lokacija. Za programera je mnogo lakše da se vrednost pohrani u varijablu nazvanu npr "**var1**" nego da tu vrednost "proziva" preko adrese npr 5A73:235B, pogotovo ako program ima više varijabli.

Sintaksa za deklaraciju varijabli:

name **DB** value

name **DW** value

DB - Definiše Byte.

DW - Definiše Word.

name - kombinacija slova i brojeva započeta slovom.

value - numerička vrednost (hexadecimal, binary, or decimal), ili "?" za varijable koje nisu inicijalizovane.

Nizovi

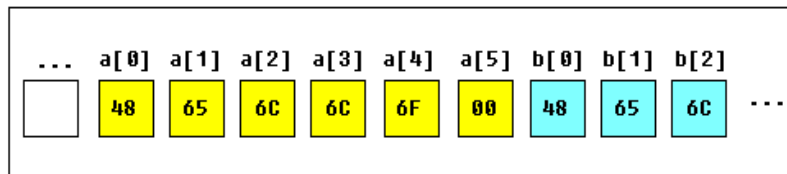
Nizovi se mogu dočarati kao lanci varijabli.

Evo par primera za definiciju niza:

a DB 48h, 65h, 6Ch, 6Ch, 6Fh, 00h

b DB 'Hello', 0

b je egzaktna kopija niza *a* ; kada kompajler vidi string unutar navoda on to automatski konvertuje u set bajtova. Sledeći grafik pokazuje deo memorije gde se nalaze ovi nizovi:



Bilo kojoj vrednosti u nizu se pristupa upotrebom uglastih zagrada, npr:

```
MOV AL, a[3]
```

Takođe se mogu koristiti registri **BX, SI, DI, BP**, npr:

```
MOV SI, 3
```

```
MOV AL, a[SI]
```

Ako je potrebno deklarirati veliki niz koristimo **DUP** operator. Sintaksa za **DUP** je:

number DUP (value(s))

number - broj duplikata (konstantna vrednost).

value - izraz koji će DUP duplicirati.

Npr:

```
c DB 5 DUP(9)
```

je isto što i:

```
c DB 9, 9, 9, 9, 9
```

Npr:

```
d DB 5 DUP(1, 2)
```

je isto što i:

```
d DB 1, 2, 1, 2, 1, 2, 1, 2, 1, 2
```

Naravno, može se koristiti **DW** umesto **DB** ako je potrebno pohranjivati vrednosti veće od 255, ili manje od -128. **DW** se ne može koristiti za deklaraciju stringova!

Adresa varijable

LEA (Load Effective Address) instrukcija i alternativa **OFFSET** operator mogu se koristiti da se dobije adresni pomeraj varijable.

Npr:

```
MOV AL, VAR1 ; vrednost VAR1 se kopira u AL.
```

```
LEA BX, VAR1 ; adresa VAR1 se upisuje u BX.
```

```
MOV BYTE PTR [BX], 44h ; sadržaj VAR1 postaje 44h.
```

Drugi primer, koji koristi **OFFSET** umesto **LEA**:

```
MOV AL, VAR1 ; vrednost VAR1 se kopira u AL
```

```
MOV BX, OFFSET VAR1 ; adresa VAR1 se upisuje u BX.
```

```
MOV BYTE PTR [BX], 44h ; sadržaj VAR1 postaje 44h.
```

Oba primera imaju isto dejstvo.

Konstante

Za definisanje konstanti koristimo direktivu **EQU** :

name EQU < any expression >

Npr:

```
k EQU 5
MOV AX, k
```

isto dejstvo ima kod:

```
MOV AX, 5
```

Prekidi (Interapti)

Interapti se mogu posmatrati kao funkcije. Ove funkcije olakšavaju programiranje, tako što se ne mora pisati kod za neke osnovne radnje kao što je prikaz karaktera na monitoru - jednostavno se pozove interapt koji će to uraditi. Ovakve funkcije zovemo **software interrupts**.

Interpti se takođe dešavaju i kao servis različitim hardverskim zahtevima, a takve zovemo **hardware**

interrupts. Kao programeri zainteresovani smo za softverske prekide.

Da bi pozvali prekid koristimo **INT** instrkciju, sa vrlo prostom sintksom:

INT value

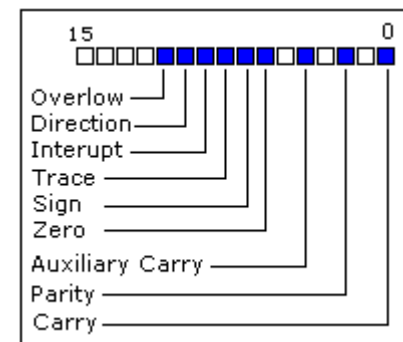
gde je **value** broj između 0 i 255 (0 do 0FFh).

Savki interapt može imati pod-funkcije. Za specifikaciju pod-funkcije koristi se **AH** registar.

Pregled interapta dat je u prilogu B.

Aritmetičke i logičke instrukcije

Aritmetičke i logčke instrukcije menjaju sadržaj status registra (odnosno **Flag-ova**)



Postoji 16 bitova, koje zovemo **flag**-ovima (zastavica , marker):

- **Carry Flag (CF)** - setovan na **1** kada se desi **unsigned overflow**. Npr bajtu vrednosti **255** dodamo **1** (rezultat nije u opsegu 0..255).
- **Zero Flag (ZF)** - setovan na **1** kada je rezultat **zero** (nula).
- **Sign Flag (SF)** - setovan na **1** kada je rezultat negativan. Vrednost ovog flega je *most significant bit*.
- **Overflow Flag (OF)** - setovan na **1** kada se desi **signed overflow**. Npr, kada saberemo bajtove **100 + 50** (rezultat nije u opsegu -128..127).
- **Parity Flag (PF)** - setovan na **1** kada je paran broj jedinica u rezultatu. Analizira se samo 8 donjih bitova za word!
- **Auxiliary Flag (AF)** - setovan na **1** kada se desi **unsigned overflow** za niža 4 bita.
- **Interrupt enable Flag (IF)** - kada je setovan na 1 CPU reaguje na interpte iz spoljnih uređaja.
- **Direction Flag (DF)** - kada je **0** - obrada je završena u smeru forward, kada je **1** obrada je završena u smeru backward.

Postoje 3 grupe ovih instrukcija.



Prva grupa: ADD, SUB, CMP, AND, TEST, OR, XOR

Podržani su sledeći tipovi operandi:

REG , memory
memory , REG
REG , REG
memory , immediate
REG , immediate

REG: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.

memory: [BX], [BX+SI+7], variable, itd...

immediate: 5, -24, 3Fh, 10001101b, itd...

Rezultat je upisan u prvi operand. **CMP** i **TEST** ne upisuju rezultat nigde.

Flagovi koji se menjaju zbog ovih instrukcija su:

CF, ZF, SF, OF, PF, AF.

- **ADD** - dodaj drugi operand prvom.
- **SUB** - oduzmi drugi operand od prvog.
- **CMP** - oduzmi drugi operand od prvog.
- **AND** - logičko I između svih bita oba operanda
- **TEST** - kao logičko I.
- **OR** - logičko ILI između svih bita oba operanda.s **1**.
- **XOR** - logičko ekskluzivno ILI između svih bita oba operanda.

**Druge grupa:** MUL, IMUL, DIV, IDIV

Podržani su sledeći tipovi operandata:

REG
memory

REG: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.

memory: [BX], [BX+SI+7], variable, itd...

MUL i **IMUL** instrukcije menjaju sadržaj flagova:

CF, OF

DIV i **IDIV** ne menjaju flegove.

- **MUL** - neoznačeno množenje:
operand je **byte**:
 $AX = AL * \text{operand}$.
operand je **word**:
 $(DX\ AX) = AX * \text{operand}$.
- **IMUL** - označeno množenje:
operand je **byte**:
 $AX = AL * \text{operand}$.
operand je **word**:
 $(DX\ AX) = AX * \text{operand}$.
- **DIV** - neoznačeno deljenje:
operand je **byte**:
 $AL = AX / \text{operand}$
 $AH = \text{ostatak (modul)}$.
operand je **word**:
 $AX = (DX\ AX) / \text{operand}$
 $DX = \text{ostatak (modul)}$.
- **IDIV** - označeno deljenje:

operand je **byte**:

$AL = AX / \text{operand}$

$AH = \text{ostatak (modul)}$.

operand je **word**:

$AX = (DX\ AX) / \text{operand}$

$DX = \text{ostatak (modul)}$.

**Treća grupa:** INC, DEC, NOT, NEG

Podržani su sledeći tipovi operandata:

REG
memory

REG: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.

memory: [BX], [BX+SI+7], variable, itd...

INC, DEC menjaju flegove:

ZF, SF, OF, PF, AF.

NOT ne menja flegove!

NEG menja flegove:

CF, ZF, SF, OF, PF, AF.

- **NOT** - negira svaki bit operanda.
- **NEG** - operand pravi negativnim (drugi komplement).

Kontrola programskog toka**Bezuslovni skokovi**Sintaksa **JMP** instrukcije:JMP label

Deklaracija *label* u programu, se postiže navođenjem njenog imena praćenog sa ":", a ime ne može početi brojem, npr:

label1:

label2:

a:

Labela se može koristiti na dva načina ako u primeru:

x1:

MOV AX, 1

x2: MOV AX, 2

Primer **JMP** instrukcije:

```
MOV  AX, 5
MOV  BX, 2
JMP  calc
back: JMP stop
calc:
ADD  AX, BX
JMP  back
stop:
RET
END
```

JMP može vršiti skok i unapred i unazad. Skok može biti bilo gde u okviru tekućeg kod segmenta (65,535 bajtova).

**Kratki uslovni skokovi**

Instrukcije uslovnog skoka se izvršavaju samo ako je ispunjen uslov. Podeljene su u tri grupe, prva grupa testira jedan fleg,

druga upoređuje brojeve kao označene, a treća upoređuje brojeve kao neoznačene.

Instrukcije skoka koje testiraju jedan fleg

Instrukcija	Opis	Uslov	Inverzna instrukcija
JZ , JE	Jump if Zero (Equal).	ZF = 1	JNZ, JNE
JC , JB, JNAE	Jump if Carry (Below, Not Above Equal).	CF = 1	JNC, JNB, JAE
JS	Jump if Sign.	SF = 1	JNS
JO	Jump if Overflow.	OF = 1	JNO
JPE, JP	Jump if Parity Even.	PF = 1	JPO
JNZ , JNE	Jump if Not Zero (Not Equal).	ZF = 0	JZ, JE
JNC , JNB, JAE	Jump if Not Carry (Not Below, Above Equal).	CF = 0	JC, JB, JNAE
JNS	Jump if Not Sign.	SF = 0	JS
JNO	Jump if Not Overflow.	OF = 0	JO
JPO, JNP	Jump if Parity Odd (No Parity).	PF = 0	JPE, JP

Ima instrukcija koje rade istu stvar, ali je programeru nekad lakše da zbog logike upotrebi npr JE a nekad JZ.

Instrukcije skoka za označene brojeve

Instrukcija	Opis	Uslov	Inverzna instrukcija
JE , JZ	Jump if Equal (=). Jump if Zero.	ZF = 1	JNE, JNZ
JNE , JNZ	Jump if Not Equal (<>). Jump if Not Zero.	ZF = 0	JE, JZ
JG , JNLE	Jump if Greater (>). Jump if Not Less or Equal (not <=).	ZF = 0 and SF = OF	JNG, JLE
JL , JNGE	Jump if Less (<). Jump if Not Greater or Equal (not >=).	SF <> OF	JNL, JGE
JGE , JNL	Jump if Greater or Equal (>=). Jump if Not Less (not <).	SF = OF	JNGE, JL
JLE , JNG	Jump if Less or Equal (<=). Jump if Not Greater (not >).	ZF = 1 or SF <> OF	JNLE, JG

Instrukcije skoka za neoznačene brojeve

Instrukcija	Opis	Uslov	Inverzna instrukcija
JE , JZ	Jump if Equal (=). Jump if Zero.	ZF = 1	JNE, JNZ
JNE , JNZ	Jump if Not Equal (<>). Jump if Not Zero.	ZF = 0	JE, JZ
JA , JNBE	Jump if Above (>). Jump if Not Below or Equal (not <=).	CF = 0 and ZF = 0	JNA, JBE
JB , JNAE, JC	Jump if Below (<). Jump if Not Above or Equal (not >=). Jump if Carry.	CF = 1	JNB, JAE, JNC
JAE , JNB, JNC	Jump if Above or Equal (>=). Jump if Not Below (not <). Jump if Not Carry.	CF = 0	JNAE, JB
JBE , JNA	Jump if Below or Equal (<=). Jump if Not Above (not >).	CF = 1 or ZF = 1	JNBE, JA

Kada je potrebno upoređivati numeričke vrednosti koristi se **CMP**.

Naprimjer:

```
MOV AL , var1
MOV BL , var2
CMP AL , BL
JE dalje
```

dalje:

```
.....
.....
.....
.....
```

Sve instrukcije uslovnog skoka imaju ograničenje - skok može biti najviše **127** bajtova napred odnosno **128** bajtova nazad.

No ovo ograničenje se može zaobići upotrebom trika:

- Upotrebiti inverznu instrukciju, skočiti na *label_x*.
- Upotrebiti **JMP** instrukciju za skok na željenu lokaciju.
- Definirati *label_x*: neposredno posle **JMP** instrukcije.

label_x: - može biti bilo koja labela.

```
include emu8086.inc

ORG 100h

MOV AL, 25 ; set AL to 25.
MOV BL, 10 ; set BL to 10.

CMP AL, BL ; compare AL - BL.

JNE not_equal ; jump if AL <> BL (ZF = 0).
JMP equal
not_equal:

; pretpostavimo da ovde
; postoji kod koji asembliran
; ima više od 127 bajtova...

PUTC 'N' ; if it gets here, then AL <> BL,
JMP stop ; so print 'N', and jump to stop.

equal: ; if gets here,
PUTC 'Y' ; then AL = BL, so print 'Y'.

stop:

RET ; gets here no matter what.

END
```

Evo primera:

Procedure

Procedura je deo koda koji može bit pozvan iz programa da bi izveo neki specifični zadatak. Procedure prave program više strukturiranim i lakšim za razumevanje. Po pravilu procedura vraća kontrolu na isto mesto u programu odakle je pozvana.

Sintaksa za deklaraciju procedure je:

```

name PROC
    ; kod
    ; procedure ...

    RET
name ENDP

```

name - je naziv procedure.

RET je instrukcija za povratak u program iz procedure.

PROC i **ENDP** su direktive, potrebne pri asembliranju.

Za poziv procedure koristi se **CALL** instrukcija.

Evo primera:

```

ORG 100h
CALL m1
MOV AX, 2
RET          ; povratak u OS

m1 PROC
    MOV BX, 5
    RET
m1 ENDP
END

```

Postoji nekoliko načina za prenos parametara proceduri, a najlakši način je da se koriste registri.

Primer koji koristi proceduru da odštampa poruku *Hello World!*:

```

ORG 100h
LEA SI, msg    ; prenesi adresu msg u SI.
CALL print_me
RET            ; povratak u OS
; procedura štampa string, koji treba da bude
; završen nulom,
; string adresa treba da bude u SI.
print_me PROC
    next_char:
        CMP b.[SI], 0
        JE stop

        MOV AL, [SI]
        MOV AH, 0Eh
        INT 10h

        ADD SI, 1
        JMP next_char ; go back, and type another char.
    stop:
        RET
print_me ENDP

msg DB 'Hello World!', 0 ; null terminated string.
END

```

"**b.**" - prefiks ispred [SI] znači da hoćemo da uporedimo bajtove, a ne reči. Da je trebalo teči stavili bi smo "**w.**" prefiks. Kada je jedan od operanada registar to nije potrebno.

Stek (Magacin)

Stek je deo memorije za čuvanje privremenih podataka. Stek se koristi kod naredbe **CALL** da sačuva adresu povratka iz procedure; **RET** naredba uzima ovu vrednost sa steka. Ista stvar se događa kada se **INT** naredba upotrebi; ona pohrani u steku fleg registar, code segment i offset. **IRET** naredba se koristi za povratak iz prekida.

Stek možemo koristiti za čuvanje i drugih podataka, a za rad sa stekom koriste se naredbe:

PUSH - upisuje 16-to bitnu vrednost na vrh steka.

POP - čita 16-to bitnu vrednost sa vrha steka.

Sintaksa za **PUSH** instrukciju:

PUSH REG
 PUSH SREG
 PUSH memory
 PUSH immediate

REG: AX, BX, CX, DX, DI, SI, BP, SP.

SREG: DS, ES, SS, CS.

memory: [BX], [BX+SI+7], 16 bit variable, etc...

immediate: 5, -24, 3Fh, 10001101b, etc...

Sintaksa za **POP** instrukciju:

POP REG
 POP SREG
 POP memory

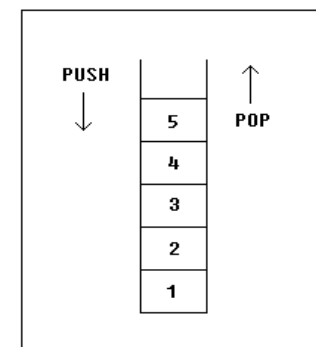
REG: AX, BX, CX, DX, DI, SI, BP, SP.

SREG: DS, ES, SS, (except CS).

memory: [BX], [BX+SI+7], 16 bit variable, etc...

PUSH neposredno radi samo na 80186 CPU i novijim

Stek radi po **LIFO** (Last In First Out) algoritmu:



Vrlo je važno uraditi jednak broj **PUSH-ova** i **POP-ova**, u suprotnom stek će biti *corrupted* (narušena ispravnost

podataka) i neće biti moguće vratiti se u OS.

PUSH i **POP** naredbe su pogotovo korisne zbog ograničenja u broju registara opšte namene, pa koristimo sledeći trik:

- pohranimo originalnu vrednost registra u stek
- upotrebimo registar za traženu aktivnost
- vratimo originalnu vrednost u registar sa steka

Evo primera:

```
ORG 100h

MOV AX, 1234h
PUSH AX
; .....
; sada se AX može upotrebiti za
; druge svrhe
; .....

POP AX

RET

END
```

Stek memorija je definisana **SS** (Stack Segment) registrom, i **SP** (Stack Pointer) registrom. Operativni sistem postavlja ove registre na startu programa.

PUSH source obavlja sledeće:

- Oduzima **2** od **SP** registra.
- Zapisuje vrednost **source** na adresu **SS:SP**.

POP destination obavlja sledeće:

- Zapisuje vrednost sa adrese **SS:SP** u **destination**.
- Dodaje **2 SP** registru.

Tekuća adresa na koju ukazuje **SS:SP** zove se **the top of the stack (vrh steka)**.

Makroi

Makroi su slični procedurama, ali oni postoje samo dok program nije kompajlovan. Posle toga svi makroi su zamenjeni sa realnim instrukcijama. Ako se makro deklariše ali se nigde ne pozove u programu, kompajler će ga prosto zanemariti.

```
Makro definicija:

name MACRO
    [parameters,...]

    <instructions>

ENDM
```

Za razliku od procedura, makroi treba da budu definisani pre koda koji će ga koristiti, naprimer:

```

MyMacro MACRO p1, p2, p3
    MOV AX, p1
    MOV BX, p2
    MOV CX, p3
ENDM

ORG 100h
MyMacro 1, 2, 3
MyMacro 4, 5, DX
RET

```

Gornji kod se proširuje u:

```

MOV AX, 00001h
MOV BX, 00002h
MOV CX, 00003h
MOV AX, 00004h
MOV BX, 00005h
MOV CX, DX

```

☐ Nekoliko važnih činjenica o **makroima** i **procedurama**:

- ✓ Kada se koriste procedure koristi se **CALL** instrukcija, naprimer:

```
CALL MyProc
```

- ✓ Kada se koristi makro, samo se navodi njegovo ime. Naprimer:

```
MyMacro
```

- ✓ Procedura je locirana na nekoj specifičnoj adresi u memoriji, i ako se koristi ista procedura 100 puta, CPU će uvek preneti kontrolu na ovaj deo memorije. Kontrola će biti vraćena nazad **RET** naredbom. **Stek** se koristi za čuvanje povratne adrese. **CALL** instrukcija zauzima oko 3 bajta, pa veličina izlaznog izvršnog fajla raste beznačajno, bez obzira koliko puta pozivali proceduru.

- ✓ Makro se razvija direktno u programskom kodu. Ako se isti makro pozove 100 puta, kompajler će 100 puta razviti makro, čime se značajno uvećava veličina izlaznog fajla. Naime svaki put kada se makro pozove naredbe koje čine njegovu definiciju se insertuju u program.
- ✓ Treba koristiti **stek** ili registar opšte namene za prenos parametara u proceduru.
- ✓ Za prenos parametara makrou, oni se prosto navode iza imena makroa. Naprimer:

```
MyMacro 1, 2, 3
```

- ✓ Oznaka kraja makroa se postiže jednostavnom direktivom **ENDM**.
- ✓ Oznaka kraja procedure se izvodi navođenjem imena procedure, a zatim **ENDP** direktivom.

Makroi su direktno insertovani u kod, pa stoga dolazi do greške ako u telu makroa postoji labela jer će se onda ona pojaviti dva ili više puta u programu što nije dozvoljeno. Da bi se izbegao ovaj problem, treba koristiti **LOCAL** direktivu praćenu sa imenima varijabli, labela ili procedura. Naprimer:

```

MyMacro2 MACRO
    LOCAL label1, label2
    CMP AX, 2
    JE label1
    CMP AX, 3
    JE label2
label1:
    INC AX
label2:
    ADD AX, 2
ENDM
ORG 100h
MyMacro2
MyMacro2
RET

```

PRILOG B

Lista podržanih prekida

INT 10h / AH = 00h - set video mode.

input:

AL = desired video mode.

These video modes are supported:

00h - Text mode 40x25, 16 colors, 8 pages.

03h - Text mode 80x25, 16 colors, 8 pages.

INT 10h / AH = 01h - set text-mode cursor shape.

input:

CH = cursor start line (bits 0-4) and options (bits 5-7).

CL = bottom cursor line (bits 0-4).

When bits 6-5 of CH are set to **00**, the cursor is visible, to hide a cursor set these bits to **01** (this CH value will hide a cursor: 28h - 00101000b). Bit 7 should always be zero.

INT 10h / AH = 02h - set cursor position.

input:

DH = row.

DL = column.

BH = page number (0..7).

INT 10h / AH = 03h - get cursor position and size.

input:

BH = page number.

return:

DH = row.

DL = column.

CH = cursor start line.

CL = cursor bottom line.

INT 10h / AH = 05h - select active video page.

input:

AL = new page number (0..7).

the activated page is displayed.

INT 10h / AH = 06h - scroll up window.

INT 10h / AH = 07h - scroll down window.

input:

AL = number of lines by which to scroll (00h = clear entire window).

BH = [attribute](#) used to write blank lines at bottom of window.

CH, CL = row, column of window's upper left corner.

DH, DL = row, column of window's lower right corner.

INT 10h / AH = 08h - read character and [attribute](#) at cursor position.

input:

BH = page number.

return:

AH = [attribute](#).

AL = character.

INT 10h / AH = 09h - write character and [attribute](#) at cursor position.

input:

AL = character to display.

BH = page number.

BL = [attribute](#).

CX = number of times to write character.

INT 10h / AH = 0Ah - write character only at cursor position.

input:

AL = character to display.

BH = page number.

CX = number of times to write character.

INT 10h / AH = 0Eh - teletype output.

input:

AL = character to write.

This function displays a character on the screen, advancing the cursor and scrolling the screen as necessary. The printing is always done to current active page.

INT 10h / AH = 13h - write string.

input:

AL = write mode:

bit 0: update cursor after writing;

bit 1: string contains [attributes](#).

BH = page number.

BL = [attribute](#) if string contains only characters (bit 1 of AL is zero).

CX = number of characters in string (attributes are not counted).

DL,DH = column, row at which to start writing.

ES:BP points to string to be printed.

INT 10h / AX = 1003h - toggle intensity/blinking.

input:

BL = write mode:

0: enable intensive colors.

1: enable blinking (not supported by emulator!).

BH = 0 (to avoid problems on some adapters).

Bit color table:

Character attribute is 8 bit value, low 4 bits set foreground color, high 4 bits set background color. Background blinking not supported.

HEX	BIN	COLOR
0	0000	black
1	0001	blue
2	0010	green
3	0011	cyan
4	0100	red
5	0101	magenta
6	0110	brown
7	0111	light gray
8	1000	dark gray
9	1001	light blue
A	1010	light green
B	1011	light cyan
C	1100	light red
D	1101	light magenta
E	1110	yellow
F	1111	white

INT 11h - get BIOS equipment list.*return:*

AX = BIOS equipment list word, actually this call returns the contents of the word at 0040h:0010h.

Currently this function can be used to determine the number of installed number of floppy disk drives.

Bit fields for BIOS-detected installed hardware:

Bit(s)	Description
15-14	number of parallel devices.
13	not supported.
12	game port installed.
11-9	number of serial devices.
8	reserved.
7-6	number of floppy disk drives (minus 1): 00 single floppy disk; 01 two floppy disks; 10 three floppy disks; 11 four floppy disks.
5-4	initial video mode: 00 EGA,VGA,PGA, or other with on-board video
BIOS;	
	01 40x25 CGA color; 10 80x25 CGA color (emulator default); 11 80x25 mono text.
3	not supported.
2	not supported.
1	math coprocessor installed.
0	set when booted from floppy (always set by emulator).

INT 12h - get memory size.*return:*

AX = kilobytes of contiguous memory starting at absolute address 00000h, this call returns the contents of the word at 0040h:0013h.

INT 13h / AH = 00h - reset disk system, (currently this call doesn't do anything).

INT 13h / AH = 02h - read disk sectors into memory.

INT 13h / AH = 03h - write disk sectors.

input:

AL = number of sectors to read/write (must be nonzero)
CH = cylinder number (0..79).
CL = sector number (1..18).
DH = head number (0..1).
DL = drive number (0..3, depends on quantity of FLOPPY_? files).
ES:BX points to data buffer.

return:

CF set on error.
CF clear if successful.
AH = status (0 - if successful).
AL = number of sectors transferred.

Note: each sector has **512** bytes.

INT 15h / AH = 86h - BIOS wait function.

input:

CX:DX = interval in microseconds

return:

CF clear if successful (wait interval elapsed),
CF set on error or when wait function is already in progress.

Note:

the resolution of the wait period is 977 microseconds on many systems, emu8086 uses 1000 microseconds period.

INT 16h / AH = 00h - get keystroke from keyboard (no echo).

return:

AH = BIOS scan code.
AL = ASCII character.
 (if a keystroke is present, it is removed from the keyboard buffer).

INT 16h / AH = 01h - check for keystroke in keyboard buffer.

return:

ZF = 1 if keystroke is not available.
ZF = 0 if keystroke available.
AH = BIOS scan code.
AL = ASCII character.
 (if a keystroke is present, it is not removed from the keyboard buffer).

INT 19h - system reboot.

Usually, the BIOS will try to read sector 1, head 0, track 0 from drive A: to 0000h:7C00h. Emulator just

stops the execution, to boot from floppy drive select from the menu: **'Virtual Drive' -> 'Boot from Floppy'**

INT 1Ah / AH = 00h - get system time.

return:

CX:DX = number of clock ticks since midnight.
AL = midnight counter, advanced each time midnight passes.

Notes:

There are approximately **18.20648** clock ticks per second, and **1800B0h** per 24 hours.
AL is not set by emulator yet!

MS-DOS can not be loaded completely in emulator yet, so I made an emulation for some basic DOS interrupts also:

INT 20h - exit to operating system.

INT 21h / AH=09h - output of a string at DS:DX.

INT 21h / AH=0Ah - input of a string to DS:DX, first byte is buffer size, second byte is number of chars actually read.

INT 21h / AH=4Ch - exit to operating system.

INT 21h / AH=01h - read character from standard input, with echo, result is stored in AL.

INT 21h / AH=02h - write character to standard output, DL = character to write, after execution AL = DL.