Praktikum iz predmeta
ORGANIZACIJA RAČUNARA

Uvod u asemblerski jezik

Zbirka rešenih zadataka

Školska 2017/2018. godina

1. Rešeni zadaci u programskom jeziku Asembler

1. Napisati program koji vrši sabiranje brojeva u dvostrukoj preciznosti. Program kao ulaze koristi 32-bitne brojeve AXBX i CXDX, a izlaz se nalazi u broju AXBX. Ukoliko dođe do prekoračenja, vrednost registra SI postaviti na 1, a u suprotnom na 0.

```
data_seg      SEGMENT
    ; Definicija praznog segmenta podataka
data_seg      ENDS

stack_seg      SEGMENT STACK
    ; Definicija praznog stek segmenta
stack_seg      ENDS

code_seg      SEGMENT
        ; Postavljanje vrednosti u segmentne registre
        ASSUME cs:code_seg, ss:stack_seg, ds:data_seg;

        ; Vrednosti koje se sabiraju
        mov ax,34
        mov bx,2
        mov cx,3
        mov dx,4

        ; Brisemo SI
        xor si,si

        ; Sabiranje
        add bx,dx ; bx := bx + dx
        adc ax,cx ; ax := ax + dx + Carry flag

        jo prekoracenje
        jmp   kraj

        prekoracenje: mov si,1
        kraj: jmp kraj

code_seg ENDS
END
```

Objašnjenje: registri opšte namene mogu da sadrže samo 16-bitne brojeve koji pripadaju intervalu [0..65535]. Da bi radili sa većim brojevima koristimo dva 16-bitna broja i njihovim spajanjem dobijamo 32-bitni broj koji pripada intervalu $[0..\sim 4*10^9]$. Kada želimo da saberemo dva 32-bitna broja, prvo moramo da saberemo njihove donje polovine (BX i DX), a zatim i njihove gornje polovine (AX i CX). Prilikom sabiranja donjih polovina naredbom ADD, može doći do prenosa (Carry flag) koji se dodaje prilikom sabiranja gornjih polovina broja naredbom ADC. Ukoliko prilikom sabiranja dodje do prekoračenja (Overflow flag), tada naredbom JO (jump if overflow) skačemo na labelu *prekoračenje* i tamo postavljamo SI na 1. U suprotnom, registar SI zadržava vrdnost 0 i naredbom JMP (jump) skačemo na labelu kraj gde se vrtimo u beskonačnoj petlji.

2. Napisati program koji vrši predstavljanje broja -2 u računaru i vrši njegovo sabiranje sa brojem 2.

```
code_seg SEGMENT
        ; Postavljanje vrednosti u segmentne registre
        ASSUME cs:code_seg, ss:stack_seg, ds:data_seg;

        ; Ovde stavljamo pozitivan i negativan broj
        mov ax, 2
        mov bx, -2
        ; Sabiramo ta dva broja
        add ax, bx

        ; Ovde je isti broj kao malopre, ali zapisan drugacije
        mov cx, 2
        mov dx, 0FFFEh

        ; Sabiramo ta dva broja
        add cx, dx

kraj:   jmp kraj

code_seg ENDS
         END
```

Objašnjenje: registri opšte namene mogu da sadrže samo 16-bitne brojeve koji pripadaju intervalu [0..65535]. Da bi koristili broj -2, umesto broja -2 koristi se njegov drugi (potpuni) komplement. Drugi komplement 16-bitnog broja X je broj koji u zbiru sa brojem X daje broj 2^{16}, tj. to je vrednost 2^{16}-X. Kako broj 2^{16} nije moguće koristiti, koristi se formula da je drugi komplement broja X jednak 2^{16}-1-X+1, tj. 65535-X+1. Vrednost 65535-X naziva se prvi (potpuni) komplement broja X i dobija se tako što se sve vrednosti 1 u binarnoj reprezentaciji broja X zamene sa 0, a sve vrednosti 0 zamene sa 1. Nakon računanja prvog komplementa, ovoj vrednosti dodje se broj 1 i tako dobija negativna vrednost broja X.

Broj 0FFFEh predstavlja drugi komplement broja -2 zapisan u osnovi 16, tj. broj 65534. U osnovi 16, svaki broj počinje sa cifrom, pa zato pišemo 0 na početku i završava se sa oznakom h. Cifre F i E predstavljaju brojeve 15 i 14.

3. Napisati program koji demonstrira tri načina adresiranja. U prvom načinu prikazati pristup segmentu podataka preko labele sabiranjem brojeva *broj1* i *broj2*. U drugom načinu demonstrirati bazno relativno adresiranje i pristupiti lokaciji broj 2 broja *broj2*. U trećem načinu, demonstrirati direktno indeksno adresiranje sabiranjem elemenata niza *niz*.

```
data_seg      SEGMENT
    broj1   dw  17
    broj2   dw  3
    niz     db  1,2,3,4,5,6,7,0 ; definicija niza brojeva tipa bajt
data_seg      ENDS

code_seg SEGMENT
        ; Postavljanje vrednosti u segmentne registre
        ASSUME cs:code_seg, ds:data_seg;

        ; Podesavamo DATA segment
start:  mov dx, offset data_seg
        mov ds, dx

        ; Koriscenje labele za pristup memoriji, sabiramo broj1 i broj2
        mov ax, broj1
        add ax, broj2
        mov broj1, ax
```

```
        ; Bazno relativno - zapravo cemo upisati u broj2 jer je BX=2
        mov bx, 2
        mov broj1[BX], 255

        ; Direktno indeksno - pomocu labela[SI]
        ; suma clanova niza dok se ne dodje do 0, rezultat u dx
        mov si, 0 ; pocetni index niza
        mov ax, 0 ; resetujemo ah (tj. ceo ax)
        mov dx, 0 ; pocetni zbir postavljamo na 0

petlja:
        mov al, niz[si]  ; citaj clan niza – jedan bajt (zato je AL)
        cmp ax, 0        ; da li je al=0? (to nam je oznaka kraja niza)
        je kraj          ; ako jeste, idi na kraj
        add dx, ax       ; dodajemo broj na zbir u dx-u
        inc si           ; povecavamo index si
        jmp petlja

kraj:   jmp kraj
        end start   ; pocinje se od labele start
```

Objašnjenje: Procesor i8086 nije dozvoljavao naredbe u kojima postoje dva pristupa memoriji preko labela, kao ni još neke kombinacije registara i labela. Zbog toga, da bi sabrali promenljive *broj1* i *broj2* prvo se vrednost jedne promenljive stavi u registar, u ovom slučaju AX, zatim na vrednost tog registra dodamo vrednost promenljive *broj2* i nakon toga zbir kopiramo u *broj1*.

Kako je svaka od promenljivih *broj1* i *broj2* velika 2 bajta (tip dw), pristup memoriji 2 bajta udaljenoj od početka broja *broj1* je ustvari pristup promenljivoj *broj2*. Na ovaj način, umesto da pristupimo promenljivoj *broj1* i tamo stavimo vrednost 255, ova vrednost je postavljena u promenljivu *broj2*.

Prilikom korišćenja direktnog indeksnog adresiranja, memoriji se pristupa sa labela[si] ili labela[di]. Prilikom sabiranja elemenata niza *niz*, koristićemo registar SI da pristupimo elementima niza, a zbir ćemo stavljati u registar DX. Kako će registar AX sadržati elemente niza koji su veliki jedan bajt (tip db), a AX je velik dva bajta, registar AH (ili ceo AX) se prethodno mora postaviti na 0. Prilikom obrade niza, uzimaju se elementi niza počevši od nultog elementa (SI=0), stavljaju u AH (tj. u AX) i dodaju na sumu DX, koja je prethodno postavljena na vrednost 0. Ukoliko se naiđe na graničnik (element 0) tada se prekida sabiranje i prelazi se na beskonačnu petlju na kraju programa. Ukoliko element nije nula, nastavlja se sabiranje na labeli *petlja.*

Procedure

Procedure su podprogrami koji obezbeđuju da se deo koda može pozivati odredjen broj puta bez ponovnog ispisivanja celog koda. Procedura se deklariše na sledeći način:

```
ImeProc PROC
        ; telo procedure
ImeProc ENDP
```

Procedure se pišu na početku kodnog segmenta, a da bi se izvršile, potrebno ih je pozvati naredbom CALL:

```
CALL imeproc
```

Pri pozivu procedure, vrednost IP registra se smesta na stek, a u IP registar se smešta ofsetna adresa procedure. Nakon poziva, procesor uzima adresu procedure iz IP registra, tj. adresu njene prve naredbe i počinje sa izvršavanjem procedure. Stara vrednost IP registra koja je smeštena na stek naziva se povratna adresa i velika je 2 bajta. U slučaju udaljenih poziva (far), adresa je velika 4 bajta jer osim IP registra, na stek se smesta i segmentni registar. Kod običnih poziva, uvek se koristi CS registar kao segmentni registar.

Svaka procedura se završava sa RET naredbom, koja može da ima parametar N. Pri nailasku na RET naredbu, procesor uzima povratnu adresu sa vrha steka, smešta je u odgovarajuće registre (IP registar i po potrebi segmentni registar) i nastavlja sa daljim radom. Kako je povratna adresa u stvari adresa prve naredbe posle CALL naredbe, program nastavlja da izvršava naredbe posle poziva procedure.
Procedure u asembleru nemaju parametre, ali ih je moguće simulirati na 3 načina, korišćenjem globalnih promenljivih, korišćenjem registara i korišćenjem steka. Ukoliko se koristi stek, za pristup podacima se koristi BP registar, a moguće je iskoristiti i vrednost SP registra koji pokazuje na vrh steka. Kako se na vrhu steka uvek nalazi povratna adresa, ove podatke je potrebno preskočiti i uzeti podatke ispod povratne adrese. Ukoliko je povratna adresa velika 2 bajta, parametri procedure se nalaze 2 bajta ispod površine steka, tj. na adresi SP+2. Ukoliko smo pre poziva procedure na stek stavili N bajtova, ovu količinu memorije na steku možemo osloboditi korišćenjem RET naredbe, tako što ćemo joj proslediti broj N kao parametar.

Primer 1: procedura koja sabira registre AX, BX i CX i rezultat smešta u registar AX

```
saberi3 proc
      add ax, bx
      add ax, cx
      ret
saberi3 endp

Primer 2: deo programa koji koristi proceduru:
      mov ax, 3
      mov bx, 5
      mov cx, 7
      call saberi3
```

Makroi

Makroi su podprogrami koji obezbeđuju da se deo koda može pozivati određen broj puta bez ponovnog ispisivanja celog koda od strane programera. Makroi se deklarišu na sledeći način:

Imemakroa MACRO parametri
; telo makroa
ENDM

Pri pozivu makroa, navodi se samo njegovo ime i argumenti:

ImeMakroa argumenti

Postoji nekoliko razlika između makroa i procedura. Prva razlika predstavlja postojanje parametara u korišćenju makroa, što kod procedura nije bilo moguće. Ipak, najveća razlika između makroa i procedura jeste način njihovog funkcionisanja. Pri korišćenju makroa, u postupku kompajliranja prvo se startuje makro predprocesor koji sve pojave poziva makroa prvo zameni sa telom makroa, pri čemu sve parametre makroa zameni sa odgovarajućim argumentima. Ukoliko makro ima na primer N naredbi i poziva se K puta, tada se svaki od K poziva zameni sa svih N naredni pa dobijeni kod ima N*K naredbi.

Primer 1: makro koji sabira parametre A, B i C i rezultat smešta u parametar A

```
saberi3 macro a b c
      add a, b
      add a, c
      endm
Primer 2: program koji koristi makro
      mov ax, 3
      mov bx, 5
      mov cx, 7
      saberi3 ax bx cx
      mov dx, 4
      saberi3 dx bx cx
```

```
Nakon makro predprocesora, dobija se sledeći kod:
     mov ax, 3
     mov bx, 5
     mov cx, 7
     add ax, bx
     add ax, cx
     mov dx, 4
     add dx, bx
     add dx, cx
```

Pri odabiru da li će se koristiti procedure ili makroi, treba obratiti pažnju na sledeće osobine:

1. Makroi su brži za izvršavanje jer nemaju naredbe CALL i RET, ali se zato troši više memorije za kod
2. Procedure su sporije za izvršavanje jer koriste naredbe CALL i RET, ali zato ne dolazi do dupliranja koda

4. Napisati program koji broj zapisan u obliku stringa u sistemu sa osnovom 10 konvertuje u tip podataka word. Nakon toga, broj tipa word konvertovati ponovo u string u osnovu 10. Koristiti procedure.

Prekidi (Interapti)

Prekidi predstavljaju male "procedure" koje imaju mogućnost da urade neke male osnovne operacije. Na računaru može da postoji najviše 256 prekida, a mogu da imaju i svoje podprekide (podprocedure). Svaki prekid ima svoj redni broj, a pozivaju se naredbom INT N, gde N predstavlja redni broj prekida. U registar AH stavlja se broj podprekida.

Da bi se prekidu prosledili parametri, najčešće se koriste registri opšte namene. Takođe, rezultati određenih prekida se smeštaju u za to unapred određene registre. Ukoliko se u nekom od tih registara nalazi korisna informacija, ona se mora prvo negde skloniti, a zatim vratiti nakon izvršenog prekida. Za pamćenje registara najčešće se koristi stek.

Prekid 16, tj. 10h, karakterističan je za BIOS (Basic Input Output System) koji se nalazi ugrađen najčešće u čipu na matičnoj ploči računara. Dva karakteristična podprekida prekida 10h su podprekidi 2 i 3 koji služe za postavljanje i čitanje pozicije kursora u tekstualnom režimu rada. U ovom režimu, ekran se posmatra kao matrica dimenzija 80x25 koji sadrži znakove.

Podprekid	Opis	Ulaz	Izlaz
AH=02	Postavljanje kursora	BH – broj video strane (0) DH – broj vrste DL – broj kolone	
AH=03	Čitanje pozicije kursora	BH – broj video strane	DH – broj vrste DL – broj kolone CH – početna linija CL – krajnja linija kursora

Tabela: Prekid BIOS-a 10h

Da bi se kursor postavio na neku poziciju na ekranu, u registar AH se stavlja podprekid 2, u registar BH redni broj video stranice (najčešće 0), a u registre DH i DL redni broj vrste i kolone. Nakon naredne INT 10h, kursor se pomera na traženu poziciju.

Prilikom čitanja pozicije kursora na ekranu, u registar AH potrebno je staviti podprekid 3, a u registar BH redni broj video stranice. Nakon poziva naredbe INT 10h, u registrima DH i DL će se naći redni broj vrste i kolone, a u regstrima CH i CL veličina, tj oblik, kursora.

```
3.
```

Primer 1: Napisati proceduru koja prebacuje ispis na ekranu u novi red:

```
novired proc
    push ax
```

```
        push bx
        push cx
        push dx
        mov ah,03
        mov bh,0
        int 10h
        inc dh
        mov dl,0
        mov ah,02
        int 10h
        pop dx
        pop cx
        pop bx
        pop ax
        ret
    novired endp
```

Procedura *novired* prvo pamti na steku sadržaj svih registara koji će biti korišćeni. Nakon toga, vrši čitanje pozicije kursora na ekranu, da bi znala u gde treba da pređe. Kako se početak sledećeg reda nalazi u koloni 0, u registar DL se postavlja 0, a u redni broj vrste DH postavlja se pročitani redni broj vrste uvećan za 1 (inc DH, 1). Nakon ovoga, vrši se vraćanje vrednosti registara i procedura završava sa radom.

Prekid 33, tj. 21h, karakterističan je za operativni sistem DOS (Disk Operating System), a koristi se i pod operativnim sistem Windows. Neki od podprekida nalaze se u tabeli ispod, a njihovo objašnjenje biće dato kroz primere:

Podprekid	Opis	Ulaz	Izlaz
AH=01	Ulaz znaka sa izlazom		AL – ucitani znak
AH=02	Izlaz jednog znaka	DL –znak	
AH=08	Ulaz znaka bez izlaza		AL – ucitani znak
AH=09	Ispis stringa	DS:DX – adresa prvog znaka	
AH=0Ah	Unos stringa	DS:DX – adresa početka bafera	
AX=4c02h	Prekid programa		

Tabela: Prekid DOS-a 21h

Primer 2: Napisati makro koji učitava jedan znak sa tastature i prikazuje ga na ekranu.

```
read macro c
    push ax
    mov ah, 01
    int 21h
    mov c, al
    pop ax
endm
```

Da bi se učitao znak, koristi se podprekid 1 prekida 21h. Posle učitavanja, znak se nalazi u registru AL, a zatim se prebacuje u parametar C.

Primer 3: Napisati makro koji ispisuje jedan znak na ekran.

```
write macro c
    push ax
    push dx
    mov ah, 02
    mov dl, c
    int 21h
```

```
    pop dx
    pop ax
endm
```

Da bi se ispisao znak na ekran, koristi se podprekid 2 prekida 21H, pri čemu se znak koji treba prikazati stavlja u registar DL. Znak se uvek ispisuje na tekuću poziciju kursora, a poziciju je moguće promeniti korišćenjem prekida 10h.

Primer 4: Napisati makro koji učitava jedan znak sa tastature i NE prikazuje ga na ekranu.

```
readkey macro c
    push ax
    mov ah, 08
    int 21h
    mov c, al
    pop ax
endm
```

Da bi se učitao znak bez prikazivanja na ekran, koristi se podprekid 8 prekida 21h. Posle učitavanja, znak se nalazi u registru AL, a zatim se prebacuje u parametar C. Ovaj podprekid može biti primenljiv za učitavanje šifre ili pri programiranju igrica.

Takođe, moguće je kreirati i varijantu ovog makroa u kojoj se samo čeka na pritisak tastera, bez da se učitana vrednost negde smesti. Za ovako nešto, može se koristiti makro *keypress*:

```
keyPress macro
    push ax
    mov ah, 08
    int 21h
    pop ax
endm
```

Primer 5: Napisati makro koji ispisuje string na ekran.

```
writeString macro s
    push ax
    push dx
    mov dx, offset s
    mov ah, 09
    int 21h
    pop dx
    pop ax
endm
```

Pri ispisu stringa, koristi se podprekid 9 prekida 21h. Pre poziva interapta, u registar DX potrebno je staviti ofsetnu adresu stringa koji treba ispisati, a string se mora završiti znakom ‘$’.

Primer 6: Napisati proceduru koja učitava string sa tastature.

```
readString proc
    push ax
    push bx
    push cx
    push dx
    push si
    mov bp, sp
    mov dx, [bp+12]
    mov bx, dx
    mov ax, [bp+14]
    mov byte [bx] ,al
```

```
    mov ah, 0Ah
    int 21h
    mov si, dx
    mov cl, [si+1]
    mov ch, 0
kopiraj:
    mov al, [si+2]
    mov [si], al
    inc si
    loop kopiraj
    mov [si], '$'
    pop si
    pop dx
    pop cx
    pop bx
    pop ax
    ret
readString endp
```

Pri učitavanju stringa podprekidom 0Ah, umesto stringa koristi se formatizovani bafer. Ovaj bafer u nultom bajtu mora da sadrži maksimalnu dužinu stringa koji može da bude učitan, a nakon učitavanja u prvom bajtu se nalazi dužina učitanog stringa. Ovako učitan string ne odgovara definiciji stringa koja je do sada korišćena, upravo zbog ta dva bajta na početku. Zbog toga, nakon učitavanja stringa, svi učitani karakteri se pomeraju za dva bajta u levo, i na kraju se dopiše znak '$'.

Na početku procedure, prvo se svi registri koji će biti korišćeni stavljaju na stek. Nakon toga, u registar DX se stavlja parametar koji predstavlja ofsetnu adresu stringa (preskaču se 10 bajtova za registre AX, BX, CX, DX i SI i 2 za povratnu adresu), a broj dozvoljenih znakova za učitavanje se stavlja na početak tog stringa. Nakon učitavanja stringa (naredba int 21h), u registar CL se stavlja broj učitanih znakova, a zatim se loop petljom taj broj učitanih znakova pomera za dva mesta u levo. Kada se završi pomeranje, na kraj stringa se postavlja znak '$' i u korišćene registre se vraćaju stare vrednosti.

Primer 7: Napisati makro kojim se prekid program u asembleru.

```
krajPrograma macro
    mov ax, 4c02h
    int 21h
endm
```

Prekid programa vrši se pozivom podprekida 4ch prekida 21h. Nakon ovog prekida kontrola programa se vraća operativnom sistemu (DOS-u), a parametar 02 koji je stavljen u registar AL označava da je program završen bez problema. Ukoliko je potrebno vratiti podatak da je program prekinut zbog neke greške, u registar AL se stavlja redni broj greške.

4. Napisati program koji učitava brojeve M i N i ispisuje tablicu množenja dimenzije MxN

```
data segment
    poruka1 db "Unesite broj N: $"
    strM db "        "
    M dw 0
    poruka2 db "Unesite broj N: $"
    strN db "        "
    N dw 0
    poruka3 db "Tablica mnozenje izgleda ovako: $"
    strBroj db '       '
    Broj dw 0
    poruka4 db "Pritisnite neki taster...$"
ends
; ovde ide definicija stek segmenta
code segment
```

```
; ovde idu procedure i makroi
start:
    ASSUME cs: code, ss:stek
    mov ax, data
    mov ds, ax

    call novired
    writeString poruka1
    push 6
    push offset strM
    call readString
    push offset strM
    push offset M
    call strtoint
    call novired
    writeString poruka2
    push 6
    push offset strN
    call readString
    push offset strN
    push offset N
    call strtoint
    call novired
    writeString poruka3
    call novired

    mov si, 1
    mov cx, N
petlja3:
    mov di, 1
    push cx
    mov cx, M
petlja4:
    mov ax, si
    mul di
    mov Broj, ax
    push Broj
    push offset strBroj
    call inttostr
    writeString strBroj
    Write ' '
    add di,1
    loop petlja4

    pop cx
    add si, 1
    call NoviRed
    loop petlja3
    keypress
    krajPrograma
    ends
end start
```

Na početku programa, prvo se unose brojevi M i N. Ova dva broja se prvo učitavaju u stringove *strM* i *strN*, a zatim se procedurom *strtoint* prebacuju u brojeve. Nakon toga, računanje tablice množenja i njen ispis vrši se unutar dvostruke loop petlje.

Kako loop petlja uvek koristi isključivo CX registar, javlja se problem dvostrukog korišćenja registra CX, kao brojača u spoljnoj petlji (*petlja3*) i kao brojača u unutrašnjoj petlji (*petlja4*). Da bi se izbegao konflikt, pre pokretanja unutrašnje petlje, brojač CX ćemo staviti na stek, a nakon završetka ove petlje uzeti sa steka njegovu staru vrednost. Inicijalna vrednost za brojač CX u spoljnoj je *N* jer imamo N vrsta, a u unutrašnjoj petlji *M* jer imamo M kolona. Kako brojač CX

broji od M ka 1, tj. od N ka 1, a za računanje matrice su potrebne obrnute vrednosti brojača, uvešćemo dodatne brojače za ove petlje, i to SI za spoljašnju i DI za unutrašnju petlju. Početne vrednosti za ove brojače su 0.

Nakon inicijalizacije SI i CX za spoljnu petlju, prelazi se na čuvanje registra CX i inicijalizaciju vrednosti DI i CX za drugu petlju. Unutar druge petlje, vrednost brojača SI se stavlja u registar AX, a nakon toga ta vrednost se pomnoži sa registrom DI. Posle množenja, u registru se nalazi proizvod dva brojača koji treba ispisati na ekran. Prilikom ispisa, proizvod se stavlja u promenljivu *Broj*, konvertuje u string *strBroj* i ispisuje na ekran, nakon čega se ispisuje i jedan razmak. Na kraju unutrašnje petlje potrebno je povećati vrednost brojača DI, dok se vrednost brojača CX automatski koriguje.
Nakon završetka unutrašnje petlje, vraća se vrednost brojača CX i spoljna petlja nastavlja sa radom. Slično kao i kod prethodne petlje, brojač SI mora da se poveća za 1. Nakon toga, štampa se jedan prazan red da bi se na ekranu dobio izgled matrice, i prelazi se na sledeću iteraciju spoljne petlje.

5. Napisati program koji učitava niz od 10 brojeva i ispisuje najmanji broj u učitanom nizu.

```
data segment
    poruka1 db "Unesite broj : $"
    strN db "        "
    N dw 0
    Niz dw 10 dup(?)
    poruka2 db "Minimum unetog niza je: $"
    strMin db "        "
    Min dw 0
    poruka3 db "Pritisnite neki taster...$"
ends
; ovde ide definicija stek segmenta
code segment
; ovde idu definicije procedura i makroa
start:
    ASSUME cs: code, ss:stek
    mov ax, data
    mov ds, ax
    mov cx, 10
    mov si, 0

unos:
    call novired
    writeString poruka1
    push 6
    push offset strN
    call readString
    push offset strN
    push offset N
    call strtoint
    mov ax, N
    mov Niz[si], ax
    add si, 2
    loop unos

    mov si, 0
    mov ax, Niz[si]
    mov min, ax
    mov cx, 9
petlja:
    add si, 2
    mov ax, Niz[si]
    cmp min, ax
    jl dalje
    mov min, ax
dalje:
    loop petlja
```

```
    push Min
    push offset strMin
    call inttostr
    call novired
    writeString poruka2
    writeString strMin
    call novired
    writeString poruka3
    keypress
    krajPrograma
ends
end start
```

Prilikom definicije niza (*Niz dw 10 dup(?)*) navodi se kojeg će tipa podata biti elementi niza (*dw*), broj elemenata niza (*10*) kao i inicijalna vrednost elemenata niza (opcija *dup(?)*). Oznaka *?* označava da je vrednost elemenata niza na početku programa nepoznata, a ukoliko se umesto *?* upiše npr. 0 to znači da će svi elementi niza biti inicijalizovani na 0.

Kod učitavanja niza koristi se loop petlja koja ponavlja učitavanje jednog broja i njegovo smeštanje u niz ukupno 10 puta. Svaki put, učita se broj N u obliku stringa, konvertuje u promenljivu N i smesti u niz na poziciji SI. Nakon toga, SI se poveća za 2 kako bi dobio poziciju sledećeg elementa u nizu, jer su elementi niza veliki 2 bajta.

Da bi se pronašao minimum, za inicijalnu vrednost minimuma (promenljiva *min*) uzima se vrednost nultog elementa niza. Nakon toga, preostalih 9 elemenata niza se upoređuju sa postojećim minimumom i ukoliko se pronađe element koji je manji od minimuma, smešta se u promenljivu *min*. Na kraju programa, vrednost promenljive *min* se štampa na ekran.

6. Napisati program koji učitava dva stringa i leksikografski ih poredi. Program treba da ispiše odgovarajuću poruku.

```
data segment
  poruka1 db 'Unesite prvi string: $'
  poruka2 db 'Unesite drugi string: $'
  string1 db '                      '
  string2 db '                      '
  porukaisti db 'Stringovi su isti$'
  porukamanji db 'Prvi string je manji$'
  porukaveci db 'Prvi string je veci$'
ends
; ovde ide definicija stek segmenta
code segment
; ovde idu definicije procedura i makroa
start:
    ASSUME cs: code, ss:stek
    mov ax, data
    mov ds, ax

    call novired
    writeString poruka1
    push 20
    push offset string1
    call readString
    call novired
    writeString poruka2
    push 20
    push offset string2
    call readString
    call novired

    xor si, si
petlja:
    mov al, string1[si];
    mov ah, string2[si];
```

```
    cmp al, ah
    jl manji
    jg veci
    cmp al, '$'
    je isti
    inc si
    jmp petlja
manji:
     writeString porukamanji
    jmp kraj
veci:
    writeString porukaveci
    jmp kraj
isti:
    writeString porukaisti
kraj:
    keypress
    krajPrograma
ends
end start
```

Na početku programa, učitavaju se stringovi *string1* i *string2*, ne duži od 20 karaktera koje je potrebno porediti. Kako se uvek porede karakteri koji se nalaze na istim pozicijama u stringu, dovoljno je koristiti samo jedan brojač (SI) koji će se koristiti kao pozicija znakova u oba stringa koje treba porediti. Na početku algoritma, ovaj brojač se postavlja na 0 (naredba *xor si, si*).

Prilikom poređenja, u registre AL i AH se postavljaju karakteri iz stringova koje treba uporediti. Ukoliko je karakter koji je uzet iz stringa *string1* manji, tada je i *string1* manji, ispisuje se odgovarajuća poruka i završava se program. Ukoliko je ovaj karakter veći, tada je i string *string1* veći, ispisuje se odgovarajuća poruka i završava program. Ukoliko nije ništa od ovoga, tada su karakteri jednaki i postoje dve mogućnosti. Ukoliko su oba jednaka znaku '$', tada su stringovi isti, ispisuje se odgovarajuća poruka i program završava sa radom, a ukoliko nisu, prelazi se na sledeće karaktere i algoritam se ponavlja za sledeće karaktere.

Ukoliko su oba karaktera jednaka, tada je dovoljno proveriti samo za jedan od njih da li je jednak sa '$', a provera drugog je suvišna. Ukoliko je neki od stringova kraći od drugog, tada se u jednom trenutku sa kraja ovog stringa uzima karakter '$' i on se poredi sa sledećim karakterom u drugom stringu. Kako karakter '$' ima redni broj 36 u ascii tabeli i manji je od svih cifara i slova, ovaj string će biti biti manji. Ukoliko bi u stringu bio dozvoljeno korišćenje znakova koji se nalaze pre znaka '$' (npr. karakter space), tada bi ovaj slučaj trebalo ispitati posebno...

7. Napisati program koji učitava dva broja N i M i rekurzivnom procedurom računa NZD(N, M).

```
data segment
    poruka1 db "Unesite broj M: $"
    strM db "        "
    M dw 0
    poruka2 db "Unesite broj N: $"
    strN db "        "
    N dw 0
    poruka3 db "NZD za unete brojeve je: $"
    strNZD db "        "
    NZD dw 0
    poruka4 db "Pritisnite neki taster...$"
ends
; ovde ide definicija stek segmenta
code segment
; ovde idu definicije procedura i makroa
NZDRek proc
    push ax
    push bx
    push bp
```

```
    mov bp, sp
    mov ax, [bp+8]
    mov bx, [bp+10]
    cmp ax, bx
    je NZDKraj

    jg vece
    sub bx, ax
    jmp nastavi
vece:
    sub ax, bx
nastavi:
    push ax
    push ax
    push bx
    call NZDRek
    pop ax

NZDKraj:
    mov [bp+12], ax
    pop bp
    pop bx
    pop ax
    ret 4
NZDRek endp
start:
    ASSUME cs: code, ss:stek
    mov ax, data
    mov ds, ax

    call novired
    writeString poruka1

    push 6
    push offset strM
    call readString

    push offset strM
    push offset M
    call strtoint

    call novired
    writeString poruka2
    push 6
    push offset strN
    call readString

    push offset strN
    push offset N
    call strtoint

    push NZD
    push M
    push N
    call NZDRek
    pop NZD

    push NZD
    push offset strNZD
    call inttostr

    call novired
```

```
    writeString poruka3
    writeString strNZD

    call novired
    writeString poruka4
    keypress
    krajPrograma

ends
end start
```

Na početku programa, učitavaju se brojevi N i M u obliku stringa i konvertuju u promenljive M i N. Nakon toga na stek se prvo stavlja jedno prazno mesto koje će služiti da se unutar procedure u njega stavi rezultat, a zatim se na stek stavljaju parametri M i N. Nakon završetka procedure *NZDRek*, na vrhu steka (na mestu koje smo ostavili za rezultat) će se naći najveći zajednički delitelja za ove brojeve. Ova vrednost se skida sa steka i stavlja u promenljivu NZD, zatim se konvertuje u string i ispisuje na ekran.

Pre startovanja procedure *NZDRek*, na steku su se našli mesto za rezultat, kao i dva broja koji predstavljaju parametre procedure. Kako se pri startovanju procedure na stek stavlja povratna adresa (registar IP), a nakon startovanja se stavljaju i registri AX, BX i BP, to znači da će se parametri procedure nalaziti 8, tj. 10 bajtova ispod površine steka. Ovi parametri se uzimaju sa steka i procedura kreće sa radom.

Ukoliko su oba parametra jednaka, prelazi se na kraj procedure gde se vrednost jednog od njih stavlja na stek na unapred predvidjeno mesto za rezultat, koje je u ovom slučaju 12 bajtova ispod površine steka. Nakon toga, vraćaju se stare vrednosti registara i procedura završava sa radim, pri čemu se vrši i automatsko skidanje parametara sa steka (*ret 4*). Mesto na steku u kojem se nalazi rezultat ostaje na steku i ono se ne briše.

Ukoliko parametri nisu jednaki, tada se od većeg parametra oduzima manji i prelazi na rekurzivni poziv. Pri rekurzivnom pozivu, na stek se prvo stavlja prazno mesto u koje će biti upisan rezultat, a zatim i parametri procedure. Nakon rekurzivog poziva, rezultat se skida sa steka u registar AX i dobijena vrednost se stavlja u unapred pripremljeno mesto na steku za rezultat.

8. Napisati program koji učitava broj N i rekurzivnom procedurom računa faktorijel unetog broja.

```
data segment
    poruka1 db "Unesite broj N: $"
    strN db "        "
    N dw 0
    poruka2 db "Faktorijel unetog broja je: $"
    strFakt db "        "
    Fakt dw 0
    poruka3 db "Pritisnite neki taster...$"
ends
; ovde ide definicija stek segmenta
code segment
; ovde idu definicije procedura i makroa
Faktorijel proc
    push ax
    push bx
    push dx
    push bp
    mov bp, sp
    mov ax, [bp+10]

    cmp ax, 0
    jne Nastavi
    mov ax, 1
    mov [bp+12], ax
    jmp krajFakt
```

```
Nastavi:
    mov bx, ax
    sub bx, 1
    push ax
    push bx
    call Faktorijel
    pop bx
    mul bx
    mov [bp+12], ax

krajFakt:
    pop bp
    pop dx
    pop bx
    pop ax
    ret 2
Faktorijel endp
start:
    ASSUME cs: code, ss:stek
    mov ax, data
    mov ds, ax
    call novired
    writeString poruka1
    push 6
    push offset strN
    call readString
    push offset strN
    push offset N
    call strtoint

    push Fakt
    push N
    call Faktorijel
    pop Fakt

    push Fakt
    push offset strFakt
    call inttostr
    call novired
    writeString poruka2
    writeString strFakt
    call novired
    writeString poruka3
    keypress
    krajPrograma
ends
end start
```

Na početku programa, učitava se broj N u obliku stringa i konvertuje u promenljivu *N*. Nakon toga na stek se prvo stavlja jedno prazno mesto koje će služiti da se unutar procedure u njega stavi rezultat, a zatim se na stek stavlja parametar N. Nakon izvršavanja procedure *Faktorijel*, na vrhu steka (na mestu koje smo ostavili za rezultat) će se naći faktorijel broja N. Ova vrednost se skida sa steka i stavlja u promenljivu *Fakt*, zatim se konvertuje u string i ispisuje na ekran.

Pre startovanja procedure *Faktorijel*, na steku su se našli mesto za rezultat kao i broj koji predstavlja parametar procedure. Kako se pri startovanju procedure na stek stavlja povratna adresa (registar IP), a nakon startovanja se stavljaju i registri AX, BX, DX i BP, to znači da će se parametar procedure nalaziti 10 bajtova ispod površine steka. Parametar se uzima sa steka, stavlja u registar AX i procedura kreće sa radom.

Ukoliko je parametar jednak broju 0, rezultat procedure treba da bude vrednost 1. Ova vrednost se stavlja u registar AX i iz ovog registra smešta na stek u unapred predvidjeno mesto za rezultat procedure. Ukoliko parametar nije 0, tada se u registar BX smešta vrednost parametra umenjena za 1 i prelazi na rekurzivni poziv. Pre rekurzivnog poziva, na stek se

stavljaju mesto na koje treba staviti rezultat i parametar koji se nalazi u registru BX. Nakon rekurzivnog poziva, sa steka se skida vrednost faktorijala broja BX, koja se skida sa steka, stavlja u registar BX i množi sa registraom AX u kojem se nalazi vrednost parametra za koji tražimo faktorijel. Nakon ovoga, u registru AX će se naći rezultata procedure i ovaj rezultat se stavlja na stek na unapred predvidjeno mesto 12 bajtova ispod površine steka. Nakon ovoga, vraćaju se stare vrednosti registara i procedura se završava narednom *ret 2* koja automatski skida i parametar sa steka, a na njegovom vrhu ostavlja vrednost procedure.

Pristup ekranskoj memoriji u tekstualnom režimu rada

Ekranska memorija računara koristi se za pamćenje slike koja se prikazuje na monitoru računara. Postoje dva načina na koja može da se koristi ova memorija: u grafičkom ili tekstualnom režimu rada. Kada se koristi u grafičkom režimu rada, ova memorija može da se posmatra kao matrica čiji elementi pamte vrednost boje za svaki piksel na ekranu. Za razliku od ovog režima, u tekstualnom režimu rada ova memorija se posmatra kao matrica koja sadrži podatke o znakovima (najčešće iz ascii tabele) koji se nalaze prikazani na ekranu.

Najčešći grafički mod u tekstualnom režimu bio je mod koji je ekran posmatrao kao matricu koja sadrži 80x25 znakova na ekranu, pri čemu su za pamćenje svakog znaka bila korišćena dva bajta, jedan za redni broj znaka u ascii tabeli, a drugi za boju znaka i pozadine. U ovom režimu, na ekranu je moglo da se prikaže 2000 znakova, a za njihovo pamćenje bilo je potrebno 4000 bajtova memorije.

Boja znaka je u sebi sadržala 3 podatka: boju samog znaka BZ od 0..15, boju pozadine BP od 0..7 i da li znak treba da treperi T. Kako bajt za boju sadrži 8 bitova, prvi bit koristio se za pamćenje vrednosti T, sledeća tri bita za BP, a poslednja četiri bita za BZ. Ukupna boja mogla je tada da se računa po formuli BOJA=T*128+BP*16+BZ. Za boju pozadine BZ pamtile su se vrednosti za svaku od tri osnovne boje RGB (crvena, zelena i plava) i svaka od ovih osnovnih boja mogla je da se uključi i isključi. Sve ostale boje dobijale su se kao njihove kombinacije. Za boju znaka koristila se IRGB paleta, pri čemu je vrednost I predstavljala intenzitet, a ako je on bio uključen dobijala se svetlija nijansa boje.

Ekranska memorija je bila popunjena redom. Bajt na adresi 0 koristio se za pamćenje znaka na poziciji (0, 0), a bajt na adresi 1 za njegovu boju. Bajt na adresi 2 koristi se za pamćenje znaka na poziciji (1, 0), a bajt na adresi 3 za njegovu boju itd.. Za pamćenje jednog teksta koristilo se ukupno 160 bajtova memorije. U ovom režimu rada, segmentna adresa memorije bila je fiksinara na adresi 0B800h.

Ukoliko posmatramo određeni znak na poziciji (X, Y), ofsetna adresa u ekranskoj memoriji na kojoj se nalazio ovaj znak mogla je da se dobije po formuli ADRESA=Y*160+X*2, dok je adresa boje za ovaj znak bila za jedan veća. Za pristup ekranskoj memoriji koristićemo ES registar koji će imati vrednost 0B800h, dok ćemo ofsetnu adresu memorije drćati u registru BX. Nakon što izračunamo adresu određenog znaka u memoriji, toj lokaciji pristupamo sa ES:[BX]. Na primer, ukoliko želimo da postavimo znak '*' na ekran, to moćemo da uradimo sa naredbom MOV ES: [BX], '*', a ukoliko želimo da pročitamo znak sa ekrana u registar AL, to možemo da radimo naredbom MOV AL, ES: [BX].

Da bi olakšali rad sa ekranskom memorijom, kreiraćemo određeni skup promenljivih, makroa i procedura koje će pamtiti neke određene podatke ili moći da urade neke osnovne funkcije. Objasnićemo prvo podatke koji će biti korišćeni u ostatku obve skripte. Neka je dat sledeći segment podataka:

```
data segment
     pozX dw ?
     pozY dw ?
     adresa dw ?
     sirina dw ?
     visina dw ?
     boja db ?
     znak db ?
data ends
```

Promenljive *pozX* i *pozY* koristićemo za pamćenje (X, Y) pozicije trenutnog znaka, dok će promenljiva *adresa* sadržati ofsetnu adresu tog znaka u ekranskoj memoriji. Polja *sirina* i *visina* sadržaće maksimalnu širinu i visinu ekrana, dok će polje *boja* sadržati vrednost tekuće boje sa kojom se radi. Polje znak koristićemo kao pomoćnu prmenljivu u makrou *readString*.

Makro *initGraph* koristi se za inicijalizaciju ES registra, kao i svih ovih promenljivih. ES registar se postavlja na vrednost 0B800h, a za vrednost tekućeg polja uzima se polje (0, 0) koje se nalazi na adresi ofsetnoj 0. Širina i visina ekrana postavljaju se na vrednosti 80 i 25. Za početnu vrednost boje uzima se boja 7 koja predstavlja sivu boju na crnoj podlozi.

```
macro initGraph
     push ax
     mov ax, 0B800h
     mov es, ax
     mov pozX, 0
     mov pozY, 0
     mov adresa, 0
     mov sirina, 80
     mov visina, 25
     mov boja, 7
     pop ax
endm
```

Da bi se pomerili na neko polje na ekranu koristi se makro *setXY* koje nas pomera na polje definisano parametrima makroa. Ove vrednosti se smeštaju u promenljive *pozX* i *pozY*, a nakon toga vrši se izračunavanje adrese tog znaka u ekranskoj memoriji po formuli adresa = pozY*160+pozX*2.

```
macro setXY x y
     push ax
     push bx
     push dx
     mov pozX, x
     mov pozY, y
     mov ax, pozY
     mov bx, sirina
     shl bx, 1
     mul bx
     mov bx, pozX
     shl bx, 1
     add ax, bx
     mov adresa, ax
     pop dx
     pop bx
     pop ax
endm
```

Postavljanje radne boje može se uraditi makroom *setColor* koji kao parametar ima boju koju treba postaviti. Treba obratiti pažnju da parametar ovog makroa ne moće da bude druga promenljiva, jer bi u tom slučaju koristili dve promenljive u istoj naredni što na procesoru i8086 noje bilo moguće.

```
macro setColor b
     mov boja, b
endm
```

Makro W*rite* se koristi za ispis jednog znaka na ekran na tekuću poziciju definisanu sa *pozX* i *pozY*, odnosno adresom *adresa*. Na ovu adresu postavlja se znak *c* koji treba prikazati na ekranu, dok se na sledeću adresu smešta tekuća boja. Boju prvo moramo da stavimo u neki pomoćni registar iz kojeg njenu vrednost postavljamo na ekran.

```
macro Write c
     push bx
     push dx
     mov bx, adresa
     mov es:[bx], c
     mov dl, boja
     mov es:[bx+1], dl
```

```
    pop dx
    pop bx
endm
```

Kako se ispis stringa na ekran svodi na postavljanje svih znakova stringa u ekransku memoriju, za potrebe ovako nečega kreiraćemo makro writeString, koji kao parametar ima string *str* koji treba ispisati.

U ovom makrou koristiće se dve lokalne labele *petlja* i *kraj*, koje su lokalne samo za ovaj makro. Korišćenjem direktive LOCAL izbegava se višestruko korišćenje istih labela pri višestrukoj upotrebi makroa. Registar SI koristiće se kao index u stringu pomoću kojeg ćemo uzimati znak po znak i prikazivati ga na ekranu. Kao početni znak uzima se nulti znak stringa, i on se smesta na adresu definisanu u promenljivoj *adresa*. Iz stringa se uzimaju redom znakovi i smeštaju u registar AL, a iz registra AL se zajedno sa bojom stavljaju na ekran. Ukoliko se naidje na znak '$', prelazi se na kraj ispisa, a tekuće pozicija se postavlja iza ispisanog stringa, tj. vrednost promenljive *pozX* se povećava za dužinu stringa.

```
writeString macro str
    LOCAL petlja, kraj
    push ax
    push bx
    push si
    mov si, 0
    mov ah, boja
    mov bx, adresa
petlja:
    mov al, str[si]
    cmp al, '$'
    je kraj
    mov es:[bx], al
    mov es:[bx+1], ah
    add bx, 2
    add si, 1
    jmp petlja
kraj:
    mov ax, pozX
    add ax, si
    mov bx, pozY
    setXY ax bx
    pop si
    pop bx
    pop ax
endm
```

Da bi se obrisao ekran, dovoljno je u ekransku memoriju za ascii vrednost svih 2000 znakova postaviti vrednost *space*, a za boju boju sa rednim brojem 7. Makro *clrScreen* popunjavanje ekranske memorije kreće od adrese 0 koja se stavlja u registar BX, i zatim popunjava vrednosti za svih 2000 znakova.

```
macro clrScreen
   LOCAL petlja
   push bx
   push cx
   mov bx, 0
   mov cx, 2000
petlja:
   mov es:[bx], ' '
   mov es:[bx+1], 7
   add bx, 2
   loop petlja
   pop cx
   pop bx
endm
```

Makro *writeLn* koristi se za prelazak u novi red. Za ovako nešto, potrebno je prvo povećati vrednost promenljive *pozY* za 1, a vrednost promenljive *pozX* staviti na 0. Nakon toga, makroom *setXY* vršimo pozicioniranje na ekranu, kao i računanje adrese tekućeg znaka

```
writeLn proc
     push ax
     push bx
     mov bx, pozY
     add bx, 1
     mov ax, 0
     setXY ax, bx
     pop bx
     pop ax
     ret
writeLn endp
```

Osim makroa *writeString*, redefinisaćemo i makro *readString* tako da i on koristi direktan pristup ekranskoj memoriji da prikazuje znakove koji su učitani. Registar Si ćemo koristiti kao brojač unetih slova, i zato on na početku ima vrednost 0. U registrima CX i DX pamtićemo poziciju tekućeg znaka na ekranu, a u BX njegovu adresu.

Kao dozvoljene znakove pri unosu podrazumevaćemo znakove *enter* kojim se završava unos, *backspace* kojim se briše jedan uneti znak, kao i sve znakove koji su vidljivi na ekranu. Nakon učitavanja jednog znaka makroom *readKey* učitani znak se smešta u promenljivu *znak*. Ukoliko je ovaj znak jednak enteru (karakter 13), prelazi se na kraj učitavanja i postavlja sa karakter '$' na kraj stringa. Ukoliko je znak bio backspace (karakter 8) i broj unetih znakova nije bio 0, tada se pomeramo za jedan znak levo na ekranu, prebrišemo ga na ekranu sa znakom ' ' i smanjimo broj unetih znakova. Ukoliko je bio neki od vidljivih karaktera, on se smesta u string, prikazuje na ekranu makroom write, a zatim se pozicija tekućeg znaka na ekranu pomera u desno za 1 i poveća broj unetih znakova. Nakon ovoga, prelazi se ponovo na učitavanje sledećeg znaka.

```
readString macro str
     LOCAL unos, nastavi, kraj
     push ax
     push bx
     push cx
     push dx
     push si
     mov si, 0
     mov bx, adresa
     mov cx, pozX
     mov dx, pozY
unos:
     readKey znak
     cmp znak, 13
     je kraj
     cmp znak, 8
     jne nastavi
     cmp si, 0
     je unos
     sub cx, 1
     setXY cx dx
     write ' '
     dec si
     jmp unos
nastavi:
     mov al, znak
     mov str[si], al
     write al
     add cx, 1
     setXY cx, dx
     inc si
     jmp unos
```

```
kraj:
     mov str[si], '$'
     pop si
     pop dx
     pop cx
     pop bx
     pop ax
endm
```

9. Napisati program koji korišćenjem ekranske memorije učitava koordinate pravougaonika na ekranu kao i redni broj boje, i iscrtava taj pravougaonik u učitanoj boji.

```
data segment
     ; ovde idu promenljive potrebne za rad sa ekranskom memorijom
     porukaX1 db 'Unesite X1: $'
     porukaY1 db 'Unesite Y1: $'
     porukaX2 db 'Unesite X1: $'
     porukaY2 db 'Unesite Y1: $'
     porukaBoja db 'Unesite boju: $'
     strBroj db '            '
     broj dw ?
     pozX1 dw ?
     pozY1 dw ?
     pozX2 dw ?
     pozY2 dw ?
     bojaPravougaonika db 13
      poruka db 'Pritisnite neki taster...$'
data ends
; ovde ide stek segment

code segment
; ovde idu definicije koriscenih procedura i makroa

nacrtajPravougaonik macro x1 y1 x2 y2
    LOCAL petlja1, petlja2
    push ax
    push bx
    push cx
    push dx
    push si

    mov ax, x1
    mov bx, y1
    setXY ax bx
    mov bx, adresa

    mov al, 178
    mov ah, boja

    mov dx, x2
    sub dx, x1
    add dx, 1

    mov cx, y2
    sub cx, y1
    add cx, 1

petlja1:
    push cx
    mov cx, dx
    mov si, 0
petlja2:
```

```
        mov es:[bx+si], al
        mov es:[bx+si+1], ah
        add si, 2
        loop petlja2

        pop cx
        add bx, 160
        loop petlja1

        pop si
        pop dx
        pop cx
        pop bx
        pop ax
    endm
```

Pri crtanju pravougaonika, prvo se pozicioniramo na gornje levo teme (x1, y1) i u registar BX stavimo adresu tog polja. U registar AL se stavlja redni broj karaktera koji će se koristiti pri iscrtavanju pravougaonika (karakter 178, popunjen pravougaonik), a u registar AH tekuća boja. U registar DX smešta se broj znakova u jednoj vrsti koje treba iscrtati (X2-X1+1), a u CX broj vrsta koje treba iscrtati (Y2-Y1+1). Iscrtavanje pravougaonika vrši se korišćenjem dvostruke loop petlje, a nakon svake unutrašnje petlje vrednost registra BX povećava se za 160 kako bi sadržao adresu prvoh znaka u sledećem redu.

U glavom delu programa, potrebno je prvo inicijalizovati sve registre i promenljive koje se koriste, kao i učitati vrednosti x1, y1, x2, y2 i boju u koju će biti oboje pravougaonik. Pri učitavanju svakog od ovih brojeva, prvo se ispisuje odgovarajuća poruka, zatim se učitava broj u obliku stringa u promenljivu *strBroj*, a nakon toga se se string konvertuje u odgovarajući broj. Jedini izuzetak od ovoga jeste učitavanje boje jer je ona predstavljena kao tip podataka byte, a procedura *strtoint* vraća rezultat tipa word. Da bi se ovaj problem prevazišao, boja se prvo smešta u promenljivu tipa *word*, a zatim u registar AX, nakon čega se uzima vrednost AL i postavlja za tekuću boju.

Kada su učitani svi podaci, vrši se brisanje ekrana makroom *clrScreen* kao i iscrtavanje pravougaonika makroom *nacrtajPravougaonik*. Nakon toga, vrednost tekuće boje se vraća na 7, i ispisuje poruka da se pritisne neki taster. Nakon pritiska tastera, program završava sa radom.

```
start:
     assume cs:code, ss:stek
     mov ax, data
     mov ds, ax

     initGraph

     writeString porukaX1
     readString strBroj
     push offset strBroj
     push offset pozX1
     call strtoint
     call writeLn

     writeString porukaY1
     readString strBroj
     push offset strBroj
     push offset pozY1
     call strtoint
     call writeLn

     writeString porukaX2
     readString strBroj
     push offset strBroj
     push offset pozX2
     call strtoint
```

```
        call writeLn

        writeString porukaY2
        readString strBroj
        push offset strBroj
        push offset pozY2
        call strtoint
        call writeLn

        writeString porukaBoja
        readString strBroj
        push offset strBroj
        push offset broj
        call strtoint
        mov ax, broj
        setColor al

        clrScreen
        nacrtajPravougaonik pozX1 pozY1 pozX2 pozY2

        setXY 0 24
        setColor 7
        writeString poruka
        keyPress
        krajPrograma
    code ends
    end start
```

10. Napisati program koji simulira igricu „Snake“.

```
data segment
     pozX db ?
     pozY db ?
     sirina dw ?
     visina dw ?
     adresa dw ?
     boja db ?
     znak db 178          ; popunjen pravougaonik
     glavaX db 4          ; pocetna pozicija "zmije"
     glavaY db 4
     smer db ?
     poruka db 'Pritisnite neki taster...$'
data ends
; Deficijija stek segmenta
stek segment stack
     dw 128 dup(?)
stek ends

code segment
; Postavljanje pocetnih vrednosti promenljivih
macro initGraph
     push ax
     mov ax, 0B800h
     mov es, ax
     mov pozX, 0
     mov pozY, 0
     mov sirina, 80
     mov visina, 25
     mov adresa, 0
     mov boja, 7
```

```
     pop ax
endm
; Postavljanje tekuce pozicije na poziciju (x, y)
macro setXY x y
     push ax
     push dx
     mov pozX, x
     mov pozY, y

     mov dx, sirina
     shl dx, 1
     mov ax, dx
     mov ah, pozY
     mul ah
     mov dl, pozX
     shl dl, 1
     add ax, dx

     mov adresa, ax
     pop dx
     pop ax
endm
; Postavljanje tekuce boje
macro setColor b
     mov boja, b
endm
; Ispis stringa na ekran
writeString macro str
    LOCAL petlja, kraj
    push ax
    push bx
    push si
    mov si, 0
    mov ah, boja
    mov bx, adresa
petlja:
    mov al, str[si]
    cmp al, '$'
    je kraj
    mov es:[bx], al
    mov es:[bx+1], ah
    add bx, 2
    add si, 1
    jmp petlja
kraj:

    mov ax, si
    add al, pozX
    mov ah, pozY
    setXY al ah
    pop si
    pop bx
    pop ax
endm
; Ucitavanje znaka bez prikaza i memorisanja
keyPress macro
    push ax
    mov ah, 08
    int 21h
    pop ax
endm
; Ucitavanje znaka bez prikaza
```

```
readkey macro c
    push ax
    mov ah, 08
    int 21h
    mov c, al
    pop ax
endm
; Ispis znaka na tekucu poziciju
macro Write c
     push bx
     push dx
     mov bx, adresa
     mov es:[bx], c
     mov dl, boja
     mov es:[bx+1], dl
     pop dx
     pop bx
endm
; Kraj programa
krajPrograma macro
    mov ax, 4c02h
    int 21h
endm
; Brisanje ekrana
macro clrScreen
   LOCAL petlja
   push bx
   push cx
   mov bx, 0
   mov cx, 2000
petlja:
   mov es:[bx], ' '
   mov es:[bx+1], 7
   add bx, 2
   loop petlja
   pop cx
   pop bx
endm


; Crtanje okvira
nacrtajOkvir proc
    push ax
    push bx
    push cx
    push si
    mov bx, 0
    mov si, 3840
    mov cx, 80
    mov al, znak
    mov ah, boja
    petljaHor:
       mov es:[bx], al
       mov es:[bx+1], ah
       mov es:[bx+si], al
       mov es:[bx+si+1], ah
       add bx, 2
    loop petljaHor
    mov cx, 23
    mov bx, 160
    mov si, 158
    petljaVer:
```

```
        mov es:[bx], al
        mov es:[bx+1], ah
        mov es:[bx+si], al
        mov es:[bx+si+1], ah
        add bx, 160
    loop petljaVer
    pop si
    pop cx
    pop bx
    pop ax
    ret
nacrtajOkvir endp

start:
     ; postavljanje segmentnih registara
     assume cs:code, ss:stek
     mov ax, data
     mov ds, ax
     ; inicijalizacija grafike
     initGraph
     ; Postavljanje okvira koji se iscrtava
     setColor 5
     call nacrtajOkvir

     setColor 1
     mov al, glavaX
     mov ah, glavaY
     setXY al ah
     mov al, znak
     write al
     petlja:
       readKey smer
       cmp smer, '2'
       je dole
       cmp smer, '4'
       je levo
       cmp smer, '6'
       je desno
       cmp smer, '8'
       je gore
       jmp kraj

       dole:
       add glavaY, 1
       jmp dalje

       levo:
       sub glavaX, 1
       jmp dalje

       desno:
       add glavaX, 1
       jmp dalje

       gore:
       sub glavaY, 1
       jmp dalje


       dalje:
       mov al, glavaX
       mov ah, glavaY
```

```
        setXY al ah
        mov bx, adresa
        mov al, es:[bx]
        cmp al, znak
        je kraj
        mov al, znak
        write al

    loop petlja

    kraj:
    ; Kraj programa
    setXY 1 23
    setColor 14
    writeString poruka
    keyPress
    krajPrograma
code ends
end start
```