LABORATORIJSKA VEŽBA 1

Korisničko upustvo za korišćenje DOS-a

Uvod

Skraćenica DOS potiče od *Disk Operating System* pod kojim se podrazumeva skup softverskih sredstava za spregu, tj. interfejs, izmedju korisnika i računara. Računar kao mašina, bez prihvatljivog interfejsa, korisniku ne pruža mnogo mogućnosti za korišćenje. Uz odgovarajuću spregu na višem nivou komunikacije, računar omogućuje efikasnu i raznovrsnu upotrebu. Vezu izmedju herdvera računara i potreba i zahteva korisnika uspostavlja skup programa (softver) koji naziva se operativni sistem. Ovaj skup obuhvata rezidentne programe (programi ugradjeni u računarski sistem) koji olakšavaju rad i približuju korisnika mašini.
Nakon uključenja, personalni računar (PC) testira priključeni hardver i po izdavanju odgovarajućih poruka, ispisuje takozvani *prompt* (**A:>** ili **C:>**). *Prompt* označava fizičku memorijsku jedinicu na kojoj se nalaze programi (flopi disk - A, ili hard-disk - C), kao i to da je računar spreman da prihvati komandu. Operativni sistem DOS prepoznaje odredjeni broj tzv. **internih** komandi (programi koji ih realizuju su neprekidno u memoriji - rezidentni programi) i po potrebi sa hard-diska učitava i izvršava programe koji realizuju tzv. **eksterne** komande. Osim toga, DOS omogućava direktno startovanje programa pozivom njihovog imena. Otkucani tekst nakon *prompt*-a računar tumači kao naredbu (komandu). Računar proverava otkucani tekst tražeći odgovarajući program sa tim imenom prvo medju internim, a ako ga tamo ne nadje, i medju eksternim programima (naredbama). Provera internog skupa naredbi vrši se proverom odgovarajućeg dela unutrašnje memorije, a provera eksternih komandi obavlja se proverom memorije na hard-disku. Osim flopi diska A i hard-diska C, PC-računar može imati drajv flopi diska B, particije na hard disku koje nose oznaku D, E, F itd., ili fizički drugi hard disk (sa oznakom D). Prelaz na pomenute resurse obavlja se sa B:, D:, E: itd..

S obzirom da se sa PC-jem komunicira preko tastature, u uputstvu su prezentirane njene najvažnije mogućnosti. Tastatura sadrži grupe tastera. Najveća grupa su tasteri slova. Iznad njih su funkcijski tasteri obeleženi sa F1 do F12. Njihova namena vezana je za komercijalne programe sa kojima se dobija uputstvo o korišćenju funkcionalnih tastera. Što se tiče DOS-a, funkcijski tasteri se malo koriste. Ove dve grupe čine osnovnu tastaturu. Na desnoj strani je numeriča tastatura od 17 tastera, koji služe za unos brojeva i druge funkcije (strelice, *Home*, *End*, *Insert*, *PgUp* itd.) što je omogućeno prethodnim izborom funkcije (*NumLock*). Izmedju osnovne i numeričke tastature postoje tri grupe tastera. Prvu grupu čine tasteri: *PrintScreen*/*SysRq*, *ScrollLock*, *Pause*/*Break*. Drugu grupu čine tasteri: *Insert*, *Home*, *PageUp*, *Delete*, *PageDown*, a treću grupu tasteri strelica, koji služe za kretanje po ekranu. Ove dirke oslobadjaju numeričku tastaturu od kontrolnih funkcija. Pojedinačne funkcije nekih tastera su: *Shift* - obezbedjuje kucanje velikih slova, *Caps Lock* - sva slova su velika (*Shift* bi sada dao mala slova), *Ctrl* - svakom tasteru daje još jednu funkciju, npr. još 12 funkcija za funkcijske tastere, *Alt* - omogućava unošenje ASCII kodova koji nisu pristupačni na tastaturi (držanje *Alt* i kucanje broja koji predstavlja kod ASCII znaka), *Backspace* - (strelica ulevo) briše poslednji znak u komandnoj liniji, *Esc* - u komercijalnim programima uglavnom označava povratak u prethodni meni ili kraj rada, dok u DOS-u ima značenje brisanja komandne linije, *Num Lock* - izbor načina korišćenja numeričke tastature, *Print Screen* - sadržaj ekrana ispisuje na štampač. Ako je uneta pogrešna naredba, komandna linija se može ispraviti njenim pozivom pritiskom na F3 i kretanjem strelicama.

DOS i datoteke

Rad pod operativnim sistemom DOS svodi se na kucanje komandi i startovanje programa. Programi koji se startuju smešteni su u fajlove tj. datoteke (*file*). Svaka datoteka ima svoje ime. Ime datoteke sastoji se iz dva dela: **naziva** i **tipa**. Naziv je bilo koja reč od najviše osam slova, brojki ili specijalnih znakova, pri čemu slovo mora biti na prvom mestu. Tip datoteke se sastoji od jednog, dva ili tri slova. Tip datoteke tj. ekstenzija predstavlja vrstu datoteke, a njegov značaj se najbolje sagledava u primeru programa koji je u izvornom obliku zapamćen pod imenom PROGRAM.PAS. Posle prevodjenja u izvršni oblik, u memoriji se nalazi i datoteka sa istim imenom PROGRAM ali sa oznakom za tip EXE, što znači da je u njoj smeštena izvršna verzija programa. Tip, dakle, eliminiše potrebu za davanjem različitih imena fajlovima koji logički imaju isti sadržaj. Primeri usvojenih tipova su: PAS, FOR, BAT, C, EXE, DOC, LIB, BMP, PCX itd. Za naziv datoteke treba koristiti strateška imena. To su imena koja podsećaju na njegov sadržaj.

Operativni sistem DOS podržava i džoker znake. To su znaci kojima se zamenjuje deo imena datoteke. Znak ? zamanjuje jedan znak imena, a * zamanjuje grupu znakova. Ograničenje pri ovome je da * ne može biti navedena na početku ili umetnuta unutar imena.

Kreiranje datoteka DOS-a najjednostavnije se obavlja ne sledeći način:

COPY CON DATIME.DAT

Kada se unese poslednja linija, pritisne se taster F6, a zatim *ENTER*.

Organizacija diska

Svaki memorijski medijum, a time i hard-disk, u logičkom smislu, organizovan je kao skup hijerarhijski organizovanih celina - direktorijuma. Svaki direktorijum ima svoje ime koje mora biti jednoznačno, a poželjno je da, kao i kod datoteka, ima strateško ime. Direktorijumi predstavljaju logičke celine u kojima se čuvaju datoteke tj. fajlovi. Najslikovitija prezentacija direktorijuma je u obliku stabla. Sa aspekta datoteka, može se reći da svaka datoteka ima ime koje se sastoji iz tri dela: ime direktorijuma, naziv i tip. Na ovaj način omogućena je bolja logika dodeljivanja imena i eksplicitnost naziva datoteka na novou direktorijuma. Hijerarhijska struktura je prihvatljiv način organizacije diska ali, isto tako, omogućuje efikasnije asocijativno adresiranje.

Periferijska oprema

Svaki računar je opremljen tastaturom, ekranom, i masovnom memorijom. Kod PC računara masovna memorija svodi se na hard-, CD- i flopi-disk. Svi uredjaji su imenovani u cilju komunikacije sa računarom. Za širi spektar primene, računar je opremljen i drugim uredjajima kao što su: štampači, ploteri, merni uredjaji, razvojna oprema i dr. Konstruktori DOS-a obezbedili su konzistentno imenovanje svih uredjaja. Naime, svaki od uredjaja dobio je naziv koji se sastoji od nekoliko slova, a završava se obaveznom dvotačkom (:). Spisak uredjaja koje DOS prepoznaje je: AUX: prvi serijski port (kao COM1), COM1: prvi serijski port, COM2: drugi serijski port, COM3: treći serijski port, COM4: četvrti serijski port, CON: ekran i tastatura, LPT1: prvi paralelni port, LPT2: drugi paralelni port, LPT3: treći paralelni port, PRN: paralelni štampač (kao LPT1), NUL: fiktivni uredjaj.

U situaciji kada računar otkaže, tj. blokira sa radom (usled startovnja nekog neispravnog ili neispravno instaliranog programa ili iz bilo kog drugog razloga), da bi se situacija otklonila prvo treba pokušati pritiskom na *Ctrl+Break* (pokušati i *Ctrl+C*), što će obično prekinuti program i ispisati *prompt*. Ako više pritisakana na *Ctrl+Break* ne da rezultate, treba resetovati računar. Softverski se to čini pritiskom na *Ctrl+Alt+Delete*, pri čemu se startovanje računara odvija bez testiranja hardvera i memorije. Ako ni ovakav reset ne može otkloniti blokadu, pribegava se

resetu posebnim prekidačem (hardverski reset). Resetovanje treba izbegavati zbog gubitka podataka koji se nalaze u unutrašnjoj memoriji računara a nisu zapamćeni na hard-disku.

Najčešće korišćene komande DOS-a

BACKUP - čuvanje sadržaja datoteka

Format naredbe:

BACKUP izvor odrediste [/S][/M][F], izvor - ime datoteke koju treba sačuvati, odrediste - ime diska na koji se podaci prenose, /S - prenose se sadžaji svih potdirektorijuma ispod tekućeg, /M - prenos samo modifik. datoteka, /A- odredišna disketa se prethodno formatira.

Primer: BACKUP C:\ A: /S - kompletan sadržaj diska prebacuje se na diskete

CHDIR (CD) - promena radnog direktorijuma

Format naredbe:

CHDIR [[disk:]direktorijum], ili:

CD [[disk:]direktorijum], disk - ime diska na koji se prelazi, direktorijum - ime direktorijuma na koji se prelazi.

Primeri: CD D:POM - prelezak na disk D i direktorijum POM,
CD.. - povratak na prethodni direktorijum.

CLS - brisanje ekrana

Format naredbe:

CLS

COPY - kopiranje datoteka

Format naredbe:

COPY [/A|/B] izvor [/A|/B] odrediše[/A| / B|][/V], izvor - datoteka (ili ceo put do datoteke sa njenim imenom) koja se kopira, /A - datoteka sa tekstom, /B - binarna datoteka, odredište - mesto na kome se kopira datoteka, /V - vrši se automatska verifikacija pri kopiranju.

Primer: COPY PROBA1.DAT C:\POM

DEL - brisanje datoteka

Format naredbe:

DEL datoteka [/P] - datoteka je ime datoteke koja se briše, /P - opcioni parametar koji služi za potvrdjivanje brisanja svake datoteke.

DELTREE - brisanje stabla direktorijuma

Format naredbe:

DELTREE dir [/Y], dir - naziv direktorijuma koji se uklanja, /Y - ne postavlja se kontrolno pitanje pre brisanja (ova naredba karakteristična je samo za DOS 6.0).

DIR - podaci o datotekama

Format naredbe:

DIR [ime][/P][/W][/A[:]atribut][/O:redosled][/S][/B][L][/C], ime - naziv diska, direktorijuma (ako se izostavi u pitanju je radni) i/ili fajla, pri čemu se mogu koristiti džoker znaci, /P - ispisivanje se privremeno prekida kada se ekran napuni, /W - ispisivanje samo imena datoteka u više stubaca, atribut - atributi datoteka koje se ispisuju, redosled - označava redosled kojim se datoteke prikazuju (po kestenziji, fizičkom redosl., i sl), /S - ispis sadržaja i svih poddirektorijuma, /B - u svakom redu se ispisuje samo naziv i ekstenzija datoteke tj. direktorijuma, /L - svi podaci se ispisuju samo malim slovima, /C - stepen kompresije datoteka za podrazumevanu veličinu klastera.

DISKCOPY - kopiranje diskete

Format naredbe:
DISKCOPY [disk1:][disk2:][/1][/V], /1 - kopira se samo jedna strana diskete, /V - pri kopiranju se vrši verifikacija.
Primeri: DISKCOPY A: A: - kopiranje disketa sa naizmaničnim stavljanjem u drajv,
DISKCOPY A: B: - kopiranje diskete iz drajva A na disketu iz drajva B.

ERASE - brisanje datoteka
Format naredbe:
ERASE imedat, imedat - ime datoteke koja se želi obrisati. Dejstvo je isto kao kod naredbe DEL.

FORMAT - formatiranje diska **!!!!!!!!! NIKAKO NE PROBATI NA VEŽBAMA!!!!!!!!**
Format naredbe:
FORMAT drajv: [/S | /B][/1][/4][/8][/V[:ime]][/F:kapacitet][/T:yy/N:xx][/U][/Q], - drajv - oznaka disk jedinice, /S - na disketu se prenose i sistemski fajlovi, /B - ostavlja se mesto za sistemske fajlove ali se oni ne prenose, /1 - formatiranje samo jedne strane diskete, /4 - formatiranje diskete od 360K na drajvu od 1.2M, /8 - formatiranje diskete sa 8 sektora po traci (ne koristi se odavno), /V - imenovanje diska, /F - definisanje kapaciteta diska u Kb, /T:yy/N:xx - odredjivanje broja traka i sektora pom traci, /U - nepovratno formatiranje, /Q - samo brisanje datoteka dok formatirana disketa ostaje pod starim formatom.
Primeri: FORMAT A:
FORMAT A: /S

HELP - detaljan opis MS-DOS-a
Format naredbe:
HELP [/B][/G][/H][/NOHI][tema], tema - komanda DOS-a čiji se opis traži, /B - isključenje boje na kolor karticama, /G - ubrzano osvežavanje bojena CGA karticama, /H - prikaz max. broja linija na ekranu, /NOHI - slova se ne naglašavaju pri ispisu. Primer: HELP DIR - detaljan opis naredbe DIR.

MKDIR - kreiranje direktorijuma
Format naredbe:
MKDIR direktorijum ili
MD direktorijum, direktorijum - ime direktorijuma koji se kreira.

MOVE - prenos datoteke u drugi direktorijum
Format naredbe:
MOVE izvor [,izvor...] odrediše, izvor - ime datoteke ili grupe datoteka koje treba preneti ili direktorijuma čije se ime menja, odrediste - ime direktorijuma u koji se datoteke premeštaju.
Primer: MOVE PISMO1.TXT, PISMO2.DOC D:\PISMA

PATH - direktorijumi u kojima treba tražiti komandu
Format naredbe:
PATH [dir[;...]], dir - direktorijum u kome se traži naredba.
Primer: PATH C:\DOS;C:\SYS;C:\ - ako se naredba ne pronadje na tekućem direktorijumu traži se na direktorijumu DOS; ako je i tamo nema traži se na direktorijumu SYS; ukoliko nije nadjena ni tamo, traži se na root-direktorijumu, tj. na C:\

PRINT - štampanje datoteka u pozadini
Format naredbe:
PRINT[/D:uredjaj][/B:bufsiz][/U:ticks1][/M:ticks2][/S:ticks3][/Q:maxfiles][/T][datoteka[...]][/C][/P], uredjaj - uredjaj na kome se štampa (LPT1, LPT2, LPT3, COM1, COM2, COM3, COM4,a podrazumeva se PRN tj LPT1), bufsiz - veličina bafera koji se koristi, ticks1 - vreme koje računar čeka ako se štampač ne odaziva (izmedju 1 i 255), ticks2 - vreme koje se daje za glavni program

(za prednje izvršenje - podrazumeva se 2), ticks3 - koliko se puta u sek. prekida glavni program, maxfiles - maksimalan broj datoteka koje mogu da čekaju u red za izvršenje (podrazumeva se 10), /T uklanja sve datoteke iz reda i štampanje tekuće odmah, datoteka - ime datoteke koja se štampa, /P - stavljanje datoteke u red za štampanje, /C - poništavanje zahteva da se datoteka štampa.

RENAME (REN) - promena imena datoteke
Format naredbe:
RENAME [disk:direktorijum\] ime1 ime2 ili
REN [disk:direktorijum\] ime1 ime2 .
Primer: REN \TEXT*.DOC *.TXT

RESTORE - vraćanje datoteka sačuvanih sa BACKUP
Format naredbe:
RESTORE izvor: odrediste
[/S][/P][/B:datum1][/A:datum2][/E:vreme1][/L:vreme2][/M][/N][/D], izvor - ime diska na kojem su sačuvane datoteke, /S - prenose se i sadržaji svih potdirektorijuma ispod odredišta, /P - prenos sa kontrol. pitanjem računara, datum1 - zamenjuje samo datoteke koje su kreirane ili menjane pre specificiranog datuma, vreme2 - zamena datoteka koje su kreirane ili menjane posle odr. datuma, /M - zamenjuje samo datoteke koje su modifikovane, /N - zamenjuje samo datoteke koje ne postoje na odredišnom disku, /D - prikaz spiska datoteka koje bi se vratile na disk.
Primeri: RESTORE A: C:\ ili RESTORE A: C:*.*
RESTORE A: C:\TEXT\KORESP /P

RMDIR (RD) - uklanjanje direktorijuma
Format naredbe:
RMDIR direktorijum ili
RD direktorijum, direktorijum - prazan direktorijum koji se uklanja.

TREE - ispisivanje stabla direktorijuma
Format naredbe:
TREE [disk:][/F][/A], /F - prikazuju se i imena datoteka, /A - umesto grafičkih koriste se ASCII znakovi.
Primer: TREE C: /F LPT1 - ispisuje na štampaču komletno stablo direktorijuma hard-diska (sve direktorijume i sve datoteke u njima).

TYPE - ispisivanje sadržaja datoteke
Format naredbe:
TYPE ime_dat
Primer: TYPE \TEXT\KORESP\PISMO2.TXT

UNDELETE - oporavak obrisanih datoteka
Format naredbe:
UNDELETE [datoteka][/LIST|ALL|PURGE[disk]|/STATUS|/LOAD|/UNLOAD|/S[disk]|/Tdisk [-elem]][/DOS|/DT|/DS], datoteka - ime obrisane datoteke koju treba vratiti, /LIST - ispis imena datoteka koje bi se mogle vratiti, /ALL - vraćanje svih datoteka koje se mogu vratiti, /DOS - vraća samo datoteke koje je DOS markirao kao obrisane, /DS - vraća samo datoteke koje je UNDELETE/LOAD/DS markirao kao obrisane, /LOAD - instalira rezudentni deo UNDELETE-a, koji će ubuduće pratiti brisanje datoteka, /UNLOAD - uklanja rezidentni deo UNDELETE-a iz memorije, /PURGE - briše SENTRY direktorijum na zadatom ili tekućem disku, /STATUS - podaci o trenutno aktivnom rezidentnom UNDELETE-u, /S - aktivira SENTRY zaštitu na zadatom disku.

UNFORMAT - oporavak formatirane diskete/diska

Format naredbe:
UNFORMAT disk: [/L][/TEST][/P], disk - oznaka diska koji se oporavlja, /L - ispis datoteka i direktorijuma koji bi mogli da se oporave, /P - ispisuje sve poruke na štampač povezan na LPT1.

VER - obeveštava o verziji DOS-a
Format naredbe:
VER [/R], /R - ispisuje interni broj verzije i obaveštava da li je DOS u expanded memoriji.

XCOPY - selektivno kopiranje datoteka
Format naredbe:
XCOPY izvor [odredište][/A][/M][/D:datum][/P][/S][/E][/V][/W], izvor - ime datoteke ili datoteka čiji sadržaj treba kopirati, odredište - ime diska ili direktorijuma u koji se datoteke prenose, /A - kopiranje samo datoteka sa setovanim atributom a, /M - isto kao /A, ali se atribut posle kopiranja resetuje, datum - prenose se datoteke kreirane samo posle odr. datuma, /P - selektivno kopiranje, /S - prenose se sadržaji svih potdirektorijuma ispod tekućeg, /E - kreira direktorijume na odredišnom disku, /V - automatska verifikacija prenosa, /W - pre početka kopiranja računar zahteva novu disk-jedinicu.

Batch datoteke

Pored interaktivnog rada, DOS ima mogućnost i za programsko izvršavanje svojih komandi. Obrada u kojoj sistem izvršava naredbe iz datoteke, a ne sa tastature, naziva se *batch obrada*. Pored naredbi DOS-a u batch datotekama sreću se i sledeće naredbe: REM, PAUSE, ECHO, GOTO, IF, FOR I SHIFT.

Kreiranje *batch* datoteke

Jedan od načina, ujedno i najbrži, za kreiranje *batch* datoteke je sledeći:

```
COPY CON DATIME.BAT.
```

Kada se unese poslednja linija, pritisne se taster F6, a zatim ENTER.

Naredbe karakteristične za *batch* obradu

REM

Batch naredba REM omogućuje prikaz poruka na ekranu u toku *batch* obrade. Format:

REM [*poruka*]

gde je *poruka* - reč dužine do 123 karaktera (znaka).

PAUSE

PAUSE privremeno prekida obradu sve dok od korisnika ne dobije potvrdu o nastavku rada, pritiskom na bilo koji taster, ili, ukoliko se nastavak ne želi, pritiskom na CTRL+BREAK ili CTRL+C.

ECHO

Naredbom ECHO se dozvoljava ili zabranjuje prikaz imena naredbe DOS-a koja se trenutno izvršava iz batch datoteke. Format:

ECHO [ON/OFF/*poruka*],

gde je ON dozvola prikaza imena, OFF zabrana, a *poruka* - poruka na ekranu koja se vidi pri izvršavanju. Za ECHO bez ON, OFF ili poruke, aktuelan je status koji je prethodno naveden (default vrednost je ON).

GOTO

Naredba GOTO obezbedjuje mehanizam za grananje u *batch* datoteci. Format:

GOTO *labela* ,
gde je labela oznaka od koje se nastavlja izvršavanje DOS naredbi u *batch* datoteci.

IF
Za podršku uslovnoj obradi u batch datotekama koristi se naredba IF. Format:
IF [NOT] *uslov* DOS_naredba ,
gde je *uslov* logički uslov koji može biti: broj ERRORLEVEL-a, EXIST spec_datoteke ili string1==string2. Broj ERRORLEVEL-a tumači se kao "istina" kada u prethodnom programu postoji greška, i predstavlja logički broj greške.

FOR
Naredbom FOR obezbedjen je mehanizam za ponavljanje obrade u *batch* proceduri. Format:
FOR %%*promenljiva* IN [*skup*] DO DOS_naredba ,
%%*promenljiva* predstavlja promenljivu koja uzima vrednost svih članova *skupa* sekvencijalno, dok *skup* predstavlja kolekciju imena datoteka koje se proveravaju. Primer:
FOR %%F IN (AUTOEXEC.BAT CONFIG.SYS TEST.BAT) DO TYPE %%F
Na ovaj način je omogućen prikaz sadržaja svih datoteka navedenih u zagradi, onim redosledom kojim su navedene.

Parametri batch datoteka
Pri pozivu *batch* datoteka mogu se navesti i parametri, ukoliko u obradi postoji potreba za njima. Parametri se koriste kako u DOS naredbama *batch* datoteke, tako i u specifičnim naredbama *batch* datoteka.
SHIFT
Naredbom *SHIFT* postiže se mogućnost navodjenja više od deset parametara koliko dozvoljava poziv. Posle svake iteracije (ili pre) naredbom *SHIFT* se niz parametara pomera ulevo za jednu poziciju.

Zadatak: Upoznati se sa osnovama DOS operativnog sistema i isprobati navedene primere naredbi. Izveštaj treba da sadrži opise i rezultate izvršenja pojedinih naredbi.
Proučiti dato uputsvo za DOS. Posebnu pažnju obratiti na primere naredbi iz uputsva, a konkretne zadatke obaviti po uzoru na njih.
Podgupa 1 Po uključivanju računara proučiti stablo direktorijuma i datoteka. U stablu direktorijuma naći direktorijum VEŽBE i pozicionirati se na njega. Kreirati potdirektorijum GRUPA1. U novokreiranom direktorijumu kreirati dve datoteke PROBA1.DAT i PROBA2.DAT koje će sadržati neki tekst. Kreirati pod-direktorijum POMOCNI i prebaciti u njega jednu od datoteka (drugu samo kopirati) iz direktorijuma GRUPA1. Preimenovati datoteke u direktorijumu POMOCNI tako da budu sa ekstenzijom TXT. Videti sadržaj direktorijuma. Videti sadržaj svake datoteke. Obrisati datoteke iz direktorijuma POMOCNI. Obrisati direktorijum POMO]NI. Vratiti se na "root" direktorijum.
Podgrupa 2 Po uključenju, utvrditi verziju DOS-a. Prikazati stablo direktorijuma. Kreirati direktorijum GRUPA2 kao potdirektorijum direktorijuma VEŽBE. Sve datoteke sa ekstenzijom TXT iz direktorijuma VEŽBE kopirati u direktorijum GRUPA2. Pregledati sadžaj direktorijuma GRUPA2. Kreirati datoteku PRVA.DAT sa nekim tekstualnim sadržajem. Prikazati sadržaj ove datoteke. Obrisati datoteku PRVA.DAT, a zatim pokušati povratak obrisane datoteke. Preneti sve datoteke iz direktorijuma GRUPA2 u direktorijum VEŽBE. Obrisati direktorijum GRUPA2. Proveriti stablo direktorijuma. Vratiti se na ROOT direktorijum.
Podgrupa 3 Pregledati *root* direktorijum. Pregledati stablo direktorijuma. Pozicionirati se na direktorijum VEŽBE. Pogledati sadržaj ovog direktorijuma. Prikazati sadržaj datoteke PROBA.DAT. Kreirati potdirektorijum GRUPA3. Prebaciti datoteku PROBA.DAT iz direktorijuma VEŽBE u direktorijum GRUPA3. Kreirati novu datoteku PROBA.TXT sa nekim

tekstom. Kopirati ovu datoteku u direktorijum VEŽBE i obrisati je u direktorijumu GRUPA3. Proveriti sadržaj direktorijuma VEŽBE i GRUPA3. Preimenovati datoteku PROBA.TXT u PRB.DAT. Vratiti se na *root* direktorijum.
Napomena: Izveštaj o radu treba da sadrži detaljan opis rada u smislu svake poruke na računaru, naredbe korisnika i rezultata izvršene naredbe. Navesti eventualne poruke o greškama i objasniti razloge pojavljivanja.

Pitanja

1. Koja se naredba DOS-a koristi za formatiranje diskete u disk drajv jedinici B kao 720k, 3½ inčni disk nakon formatiranja treba da sadrži vrednost labele DATA6 a i takodje sistemske fajlove.
2. Šta se dešava kada se otkuca FORMAT /? u DOS komandnoj liniji?
3. Šta se dešava kada se otkuca HELP FORMAT u DOS komandnoj liniji?
4. Koji direktorijum na video displeju prikazuje naredba DIR A: ?
5. Za šta se koristi komanda DIR /W /P ?
6. Koji se fajlovi prikazuju naredbom DIR /AH ?
7. Šta se dešava kada se otkuca naredba DIR > TEST.DIR ?
8. Prikazati komande DOS-a koje su potrebne za (a) pozicioniranje na drajv A, (b) kreiranje direktorijuma sa nazivom TEST u root direktorijumu drajva A.
9. Direktorijum se kreira u root direktorijumu, poddirektorijum se kreira u direktorijumu. Da li je ovaj iskaz tačan?
10. U DOS komandnoj liniji je prikazano A:\TEMP\TEST\SEVEN\ , objasniti šta znači svaki deo u ovom iskazu.
11. Šta će se kopirati nakon izvršenja naredbe COPY A*.* B:*.* ?
12. Šta će se kopirati nakon izvršenja naredbe COPY A:\TEMP\ABC?????.??? B:TEST\?
13. Šta će se izvršiti kada se otkuca naredba COPY *.* u *root* direktorijumu?
14. Šta može da zameni džoker karakter * u naredbi?
15. Šta može da zameni džoker karakter ? u naredbi?
16. Sta se uklanja ili briše komandom DEL B:\TEMP.* ?
17. Formulisati komandu DOS-a koja će obrisati sve fajlove sa ekstenzijom .BAK u direktorijumu TEMP na disk drajv jedinici B.

LABORATORIJSKA VEŽBA 2

DOS: autoexec.bat i config.sys fajlovi

Uvod

Fajl pod nazivom CONFIG.SYS prisutan je kod gotovo svih računarskih sistema. Njegova uloga ogleda se u obavljanju konfigurisanja personalnog računara za korektan rad. Glavna namena fajla CONFIG.SYS jeste punjenje memorije pobudnim programima, odnosno drajverima, pre pokretanja operativnog sistema DOS. Drajveri su programi koji upravljaju radom U-I uredjaja kao što je miš, a takodje i softverske komponente kao što je *high* memorija i *extended* memorija.
Fajl AUTOEXEC.BAT je opcioni u softverskoj konfiguraciji, medjutim, uobičajeno je da se on nalazi u root direktorijumu gotovo svakog personalnog računara. Osnovna funkcija ovog fajla ogleda se u izvršavanju komandi za postavljanje (*setup*) sistema. Jedan od najčešće pokretanih

programa od strane fajla AUTOEXEC.BAT jeste program za keširanje diska (*disc cache program*), ali takodje su to i iskazi koji opisuju put i druge parametre korišćene od aplikacija.

U ovoj vežbi razmatra se sadržaj kako CONFIG.SYS tako i AUTOEXEC.BAT fajla sa ciljem da se sagleda regularna konfiguracija za rad sistema. Takodje, kroz odredjene ograničene aktivnosti, razmatraju se tipične komande koje se sreću u ovim fajlovima.

Predmet rada

1. Opis sadržaja fajlova CONFIG.SYS i AUTOEXEC.BAT;
2. Korišćenje tektualnog editora EDIT za pisanje jednostavnih fajlova kao što su CONFIG.SYS i AUTOEXEC.BAT;
3. Razvoj jednostavnog CONFIG.SYS fajla kojim se omogućuje rad sistema sa sistemskog diska;
4. Uvežbavanje komandi fajla AUTOEXEC.BAT koje se aktiviraju kod uspostavljanja i pokretanja sistema;
5. Kreiranje flopi-diska koji sadrži CONFIG.SYS i AUTOEXEC.BAT i koji se, u neželjenim okolnostima, može koristiti kao sistemski disk.

Postupak rada

Kao i kod vežbe 1, i u ovoj vežbi je neophodan flopi-disk. U ovoj vežbi flopi-disk je formatiran kao sistemski i sistem se podiže sa flopi-diska. Disk koji oformimo u ovoj vežbi moći će se u neželjenim okolnostima koristiti kao sistemski disk.

KORAK 1: Umetnuti flopi disk u drajv A ili B i formatirati ga korišćenjemm sledeće komande koja smešta sistemske fajlove na disk:

FORMAT A: /S

Ovom komandom se disk formatira i sistemski fajlovi smeštaju na *root* direktorijumu. Naglasimo da su za formatiranje flopi diska kao sistemskog neophodni sistemski fajlovi na *hard*-disku. Verifikovati prisustvo ovih fajlova na *hard*-disku korišćenjem DIR komande, odnosno proverom da li se fajl COMMAND.COM nalazi u *root* direktorijumu. Takodje, probati DIR komandu sa opcijom /A kako bi se videli i nevidljivi sistemski fajlovi.

Nakon ovih aktivnosti, isključiti računar i sačekati nekoliko sekundi. Posle toga uključiti računar. Proveriti da li je flopi disk u drajvu A kako bi se sistem uspostavio ("podigao") sa flopi-diska (diskete). Alternativni način restartovanja sistema jeste pritiskom na *restart* taster ili držanjem dirki Ctrl i Alt i pritiskom na Del (CTRL-ALT-DEL kombinacija). Prvi način restartovanja– isključivanjem napajanja, naziva se i hladni restart sistema iz razloga što je sistem bio ostao bez napajanja. Kod pritiska tastera restart ili kod pritiska dirki CNTRL-ALT-DEL, radi se o tzv. vru}em restartu jer ni jednog momenta sistem nije ostao bez napajanja.

Nakon što se *hard*-disk pokrene, a to sledi u nekoliko narednih trenutaka, trebalo bi da na ekranu vidimo *prompt* datuma (poruka koja ukazuje na datum) (videti primer 2-1). Pritisnuti dirku *Enter* kako bi zadržali ponudjeni datum, ili uneti novi datum. Nakon toga vidimo *prompt* vremena. Kao i kod ponudjenog datuma, i u ovom slučaju prihvatamo ponudjeno vreme ili unosimo novo. Konačno, nakon poruke o pokretanju DOS operativnog sistema i oznake verzije, pojavljuje se i komandni DOS *prompt* (poruka). Personalni računar sada radi sa DOS-om koji je napunjen u memoriji sa flopi diska u drajvu A. Treba napomenuti da niti jedan od korisničkih drajverskih programa neće biti pokrenut (za pobudu miša, i dr.) kao što je to bio slučaj kod uobičajenog rada i korišćenja CONFIG.SYS i AUTOEXEC.BAT fajlova. Na ovaj način računar radi na način karakterističan za slučaj nepredvidjenih okolnosti (*emergency manner*), odnosno pokrenut sa flopi diskete u koraku 1. Ovako formirani flopi disk čuva se za situacije kada dodje do fizičkog oštećenja *hard*-diska i kada nije moguće podići sistem sa njega. Takodje, fajlovi sa sistemskog

flopi diska koriste se kod reparacije starog hard-diska ili kod zamene hard-diskova kada na novom ne postoji operativni sistem.

Primer 2-1

Current date is Man 12-06-2000
Enter new date (mm-dd-yy):

Sada kada imate validan disk koji se koristi za podizanje sistema izvaditi disketu iz drajva A i restartovati sistem ga sa *hard*-disk drajva ili iz mreže. Trebalo bi da se vide sve uobičajene poruke i informacije kao i pre.

Komanda EDIT

Za potrebe kreiranja ili modifikacije nekog CONFIG.SYS ili AUTOEXEC.BAT fajla, u ovom delu vežbe opisana je način korišćenja programa EDIT. Program EDIT može se naći u DOS direktorijumu. Naglasimo da program EDIT koristi QBASIC u toku izvršavanja, tako da je neophodno da i QBASIC postoji na ovom direktorijumu istovremeno.
Program EDIT poziva se iz komandne linije DOS-a kucanjem EDIT a zatim i naziva programa koji se želi kreirati ili modifikovati.

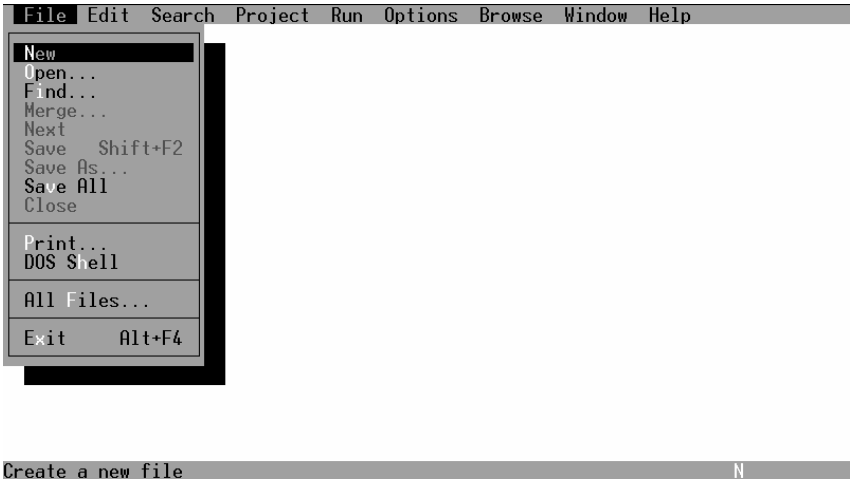
KORAK 2: Ubaciti u drajv jedinicu A flopi-disketu kreiranu u koraku 1 i otkucati sledeću komandu u komandnom *promptu* (poruci):

EDIT A:\TEST.DOC

Ovom komandom u memoriji PC mašine se upisuje program EDIT i kreira fajl na disketi drajva A pod nazivom TEST.DOC. Sada se na ekranu može videti poruka koja se odnosi na DOS `Survival Guide`. Ako ne vidi sadržaj ekrana editora tada treba pokušati sa eksplicitnim pozicioniranjem programa u komandnoj liniji:

C:\DOS\EDIT A:TEST.DOC

Većina sistema koristi iskaz PATH za navodjenje putokaza programa koji se startuju iz komandne linije, tako da nije potrebno navoditi ceo putokaz, posebno za programe iz DOS direktorijuma. Izgled ekrana editora (videti sliku 2-1) sadrži meni na vrhu, pri čemu možemo birati izmedju sledećih opcija: *File*, *Edit*, *Search*, *Options*, i *Help* podmenija. Ovim opcijama se pristupa putem miša ili pritiskom na ALT dirku i podvučenog slova iz naziva podmenija. U svakom slučaju, pojavljuje se padajući meni sa novim podopcijama. Da bi se selektovao neki od "padajućih" podopcija menija, pritisne se na levi taster miša u trenutku kada je obeležen željeni izbor opcije sa kursorom miša iznad njega ili kretanjem gore-dole uz pomoć dirki ↑ i ↓. Alternativno, želejna podopcija se može birati izabrati pritiskom dirke ALT i označenog slova u nazivu podopcije. Meni *File* omogućava da program bude snimljen, napunjen (*load*-ovan), odnosno da se iz programa izadje izborom *Exit* podopcije. Meni *Edit* omogućuje obavljanje funkcije odsecanja (*Cut*), kopiranja (*Copy*) i umetanja, tj. lepljenja (*Paste*) markiranog/zapamćenog teksta. *Search* služi kod pretraživanje reči ili niza karaktera u tekstu. Meni *Options* omogućava izmenu boje ekrana, i/ili pretaživanje putokaza *Help* fajla, EDIT programa itd. Konačno, *Help* meni pruža podršu u vidu *on-line help*-a u slučaju da se ne snalazite dobro u radu sa programom. Istu funkciju alternativno obavlja i F1 funkcijski taster.



Slika 2-1. Izgled ekrana editora.

KORAK 3: Sada kada ste u editorskom programu treba pokušati sa kucanjem kratkih jednostavnih linija informacija koje se vide na video displeju. Naglasimo da se u donjem desnom uglu uvek nalazi broj tekuće linije i pozicije karaktera u liniji u kojoj je kursor. Forma broja je XXXXX:YYY, gde je XXXXX broj linije, a YYY broj pozicije u liniji.
Nakon kucanja odredjene informacije, selektujete *File* meni i birajte X (*Exit*) kako bi ste izašli iz programa. Komanda *Exit* uvek pita da li želite da snimite fajl pre izlaska. Treba odgovoriti sa *Yes* selekcijom odgovarajućeg *box*-a ili kucanjem Y za *Yes* ili N za *Not* - ukoliko ne želimte da fajl bude zapamćen na *hard*-disku. Na ovaj način će fajl pod nazivom TEST.DOC biti sačuvan na tekućem diretkorijumu. Pozvati ponovo EDIT koristeći EDIT TEST.DOC. Ukoliko ne možete zatvoriti fajl pošto ste u direktorijumu koji jedino dozvoljava operacije čitanja, izabraćete *File* i *SaveAs* funkciju kojom se može snimati fajl i na drugom direktorijumu ili pod drugim imenom.

Fajl CONFIG.SYS

Fajl CONFIG.SYS konfiguriše personalni računar kod koga je instaliran operativni sistem DOS tako da omogućava korektan rad sistema. Kao što je već ranije istaknuto, CONFIG.SYS puni sistem fajlovima i pokretačkim programima (drajverima) za upravljanje mišem, memorijom proširenja i dr.

KORAK 4: Pozicionirati se na *root* direktorijum drajva. To je drajv C, ili, ukoliko je sistem podignut iz mreže, to je drajv F. Sada otkucati sledeću liniju u DOS-ovom komandnom *promptu*:

EDIT CONFIG.SYS

Ova akcija će uzrokovati prikaz tekućeg sadržaja fajla CONFIG.SYS na ekranu video displeja. Nemojte menjati sadržinu ovog fajla. Ukoliko ste u u mrežnom okruženju (Novel mreža pod DOS-om), to nikako nećete moći, medjutim, ako je sistem zasnovan na *hard*-disku a na mašini je instaliran DOS operativni sistem, izmene sadržaja fajla CONFIG.SYS mogu biti opasne, pa zbog toga bi trebalo da ovaj fajl bude zaštićen od izmena tako što bi imao atribut *read-only*.
Sadržaj fajla CONFIG.SYS se neznatno menja od sistema do sistema, ali tipična sadržina mogla bi se predstaviti kao u Primeru 2-2, za slučaj da niste u mogućnosti da joj pristupite na svom računaru.

Primer 2-2

DEVICE=C:\DOS\HIMEM.SYS

```
DEVICE=C:\DOS\EMM386.EXE NOEMS
BUFFERS=10,0
FILES=30
DOS=UMB
LASTDRIVE=H
DEVICEHIGH /L:1,832 =C:\DOS\SETVER.EXE
SHELL=C:\DOSCOMMAND.COM C:\DOS\ /E:512 /P
DOS=HIGH
DEVICEHIGH /L:1,39488 =C:\DOS\DBLSPACE.SYS /MOVE
DEVICEHIGH /L:1,33152 =C:\DOS\RAMBIOS.SYS
DEVICEHIGH /L:1,14464 =C:\DOS\IMOUSE.SYS /S75
DEVICEHIGH /L:1,10816 =H:\SCANMAN\HHSCAND.SYS /A=280/I=9/D=3
```

Fajl u prethodnom primeru sadrži neke najuobičajenije elemente. Prve dve linije pune memoriju drajverima uredjaja HIMEM.SYS i EMM386.EXE. Drajver HIMEM.SYS omogućuje da DOS pristupa oblasti memorije koja je locirana u višem delu adresnog prostora. Drajver EMM386.EXE omogućuje da memorija proširenja puni memorijske blokove koji su locirani u gornjem delu memorijskog prostora obezbedjujući na taj način većeg prostora u memoriji PC mašine za potrebe DOS aplikacija. Opcija NOEMS specificira to da EMM386.EXE ne snabdeva DOS memorijom proširenja čiji se rad može emulirati. Ove dve linije trebalo bi da budu prisutne uz druge specifičnije drajvere kod većine sistema. Parametri specificirani uz BUFFERS i FILES adekvatni su za većinu sistema osim ako neka specifična aplikacija ne zahteva veći broj bafera i fajlova kod izvršavanja. Preostale linije pune ostale drajvere u višu (*high*) memorijsku oblast instaliranu u DOS-u, pri čemu pune ovu oblast memorije u toku korišćenja memorijskih blokova koji su locirani na višim memorijskim adresama (*Upper Memory Blocks* - DOS=UMB). Komanda SHELL, uz pomoć COMMAND.COM, čini komandnu liniju procesorskog sistema (/p) i dodeljuje 512 bajtova za DOS-ovo okruženje.

KORAK 5: Ubaciti sistemski flopi-disk kreiran u Koraku 1 u odgovarajući drajv i promeniti aktuelni drajv kucanjem A: nakon toga, otkucati narednu liniju za pokretanje EDIT programa i upisati CONFIG.SYS:

```
EDIT A:\CONFIG.SYS
```

Na ovaj način se na ekranu pojavljuje sa aspekta editovanja prazan (blanko) ekran iz razloga što je i sadržina fajla takva. Zatim, kreirati sledeću liniju pomoću SHELL komande:

```
SHELL=A:\COMMAND.COM A:\ /E:256 /P
```

Sada snimiti ovaj sadržaj na disku na drajvu A izborom *File* i *eXit* funkcije. Sada imate kreiran sistemski disk sa jednostavnim CONFIG.SYS fajlom. Prednost što postavljamo permanentanim COMMAND.COM je ta što uvek kada se pristupa nekoj internoj komandi DOS-a (na primer DIR), računar ne pristupa disku. Permanentni COMMAND.COM fajl koji sadrži ove interne komande, nalazi se u memoriji i iz DOS-a mu se pristupa direktno. Ako fajl COMMAND.COM nije permanentan u memoriji, kod poziva svake interne i eksterne DOS-ove komande, vrši se obraćanje *hard*-disku, što rezultuje sporijim radom sistema.

Pokušajte da koristite komandu TYPE sa ciljem da se prikaže sadržaj upravo kreiranog CONFIG.SYS fajla. Naglasimo da TYPE komanda predstavlja jedan efikasan način za prikaz kraćih fajlova bez upotrebe editora. Ukoliko pak želte da prikažete neki duži fajl uz pomoć komande TYPE, tada treba pokušati da upotrebite kanalisano sprovodjenje komande MORE, odnosno, | MORE koje se navodi nakon naziva fajla. Na primer, za prikaz dugog fajla pod nazivom TEST.TXT otkucati: TYPE TEST.TXT | MORE.

Fajl AUTOEXEC.BAT

Fajlu AUTOEXEC.BAT se, od strane DOS-a, pristupa nakon pristupa fajlu CONFIG.SYS. Osnovna svrha toga je automatski izvršenje pojedinačnih ili čitavih nizova DOS-ovih komandi, pre pojave *prompta* komandne linije. Sadržina fajla AUTOEXEC.BAT neznatno se menja od sistema do sistema pre svega zbog raznovrsnosti aplikacija koje se na njima izvršavaju. Neke standardne linije fajla AUTOEXEC.BAT date su u Primeru 2-3.

Primer 2-3

```
@echo off
PATH C:\DOS
SET TEMP=C:\TEMP\
C:\DOS\DOSKEY
C:\DOS\DOSKEY D=DIR /P
```

Iako je Primer 2-3 fajla AUTOEXEC.BAT jednostavan, on realno ukazuje na nekoliko komandi koje se uobičajeno sreću kod većine AUTOEXEC.BAT fajlova. Iskaz PATH usmerava DOS na putokaz pretraživanja uvek kada se u komandnoj liniji navodi ime nekog programa a ono nije pronadjeno u tekućem direktorijumu. U ovom primeru putokaz pretraživanja je postavljen na C:\DOS. Iskaz SET omogućuje da se dodele konkretne vrednosti promenljivama koje opisuju radno okruženje. U konkretnom primeru, promenljivoj TEMP dodeljen je putokaz do direktorijuma C:\TEMP. Program pod nazivom DOSKEY puni se u memoriju kao pomoćni editor radi proširenja komandne linije. Poslednja komanda nalaže izvršenje DIR komande uvek kada se otkuca samo D, što znači da radi kao makro.

KORAK 6: Ako je na raspolaganju, iskopirati program DOSKEY na *root* direktorijum flopi diska u drajvu A bilo sa hard diska ili iz mrežne DOS particije.

```
COPY C:\DOS\DOSKEY.COM A:
```

Sada izvršiti EDIT program sa ciljem da se generiše jedan AUTOEXEC.BAT fajl na vašem sistemskom disku u drajvu A:

```
EDIT A:\ AUTOEXEC.BAT
```

Kao što je napomenuto ranije, pojavljuje se prazan ekran. Uneti sledeće linije:

```
@echo off
a:\DOSKEY.COM
A:\DOSKEY.COM D=DIR /P
```

Sledeće što treba uraditi jeste snimiti ovaj fajl i izaći iz EDIT programa. Dok se flopi disk još nalazi u drajvu, isprobati CNTL-ALT-DEL i uveriti se u njegovu funkcionalnost. Na ovaj način vaš sistem se podiže sa diskete. Proveru rada komande iz AUTOEXEC.BAT možete sprovesti kucanjem D. Trebalo bi da se na ekranu vide direktorijumi i fajlovi u prikazu kao kod interne DOS komande DIR /P. Pri izvršavanju komandi iz AUTOEXEC.BAT nije se mogao videti prikaz komandnih linija zbog toga što je to zabranjeno uz pomoć @ECHO OFF. Simbol @ sprečava prikaz i same komande ECHO OFF koja pak, zabranjuje prikaz komandi koje slede.

Na ovaj način formiran sistemski flopi disk može poslužiti u bilo kojoj situaciji kada je sistem na *hard*-disku nedostupan zbog oštećenja diska, ili kada jednostavno želimo da izbegnemo podizanje sistema sa *hard*-diska. Preporuka je čuvati ovaj disk i ne menjati mu sadržaj.

Pitanja

1. U koraku 1 vežbe kreirali ste sistemski disk. Šta se prikazuje na ekranu nakon prompta datuma?
2. Koristeci program EDIT kreirati na disku A fajl pod nazivom MYFILE.TXT. Smestiti podatke o vašem imenu i adresi u ovaj fajl. Sada ponovo pozvati editor navodeci naziv ovog fajla. Da li se vaše ime i adresa vide na ekranu?
3. Opisati postupak "otsecanja" i "lepljenja" teksta u editorskom programu EDIT.
4. U čemu je razlika izmedu COPY i CUT funkcija koje se mogu pronaci u Edit meniju programa EDIT?
5. Zbog čega je mnogo efikasnije otkucati EDIT A:\FILENAME, za razliku od EDIT, ako se želi editovanje fajla FILENAME?
6. Koje se opcije nalaze u meniju Options programa EDIT?
7. Koja je namena fajla CONFIG.SYS?
8. Koja je namena drajvera HIMEM.SYS koji se loaduje iz CONFIG.SYS fajla?
9. Koji element fajla CONFIG.SYS predvida korišcenje memorijskih blokova smeštenih na višim adresama za potrebe DOS aplikacija?
10. Koja je namena komande SHELL u fajlu CONFIG.SYS?
11. Zbog cega je važno da program COMMAND.COM ucinimo stalno prisutnim u memoriji (permanentnim)?
12. Koja je namena fajla AUTOEXEC.BAT?
13. Koju funkciju obezbeduje komanda @ECHO u fajlu AUTOEXEC.BAT?
14. Koje nove mogucnosti pruža korišcenje programa DOSKEY.COM u DOS-u?
15. Šta ce se dogoditi ako pozovemo makro DOSKEY.COM E=DEL *.BAK?
16. Šta se dešava pritiskom i držanjem CNTL i ALT dirki i pritiskom na DEL dirku?
17. Koji se simbol koristi za kanalisano sprovodenje komandi DOS-a?
18. Na koji nacin se komanda MORE koristi u kombinaciji sa TYPE komandom?

LABORATORIJSKA VEŽBA 3

Batch programi

Uvod

Batch fajlovi ili *batch* programi kako se najčešće nazivaju, omogućavaju kreiranje programa koji upravljaju radom operativnog sistema DOS. Jedan tipičan primer *batch* programa je bio prikazan u Laboratorijskoj vežbi 2 pod nazivom AUTOEXEC:BAT fajl. Naglasimo da svi *batch* fajlovi moraju koristiti ekstenziju .BAT kako bi se od strane DOS-a interpretirali kao *batch* fajlovi. Ova Laboratorijska vežba prikazuje detalje koji se odnose na kreiranje *batch* fajla sa ciljem da se automatizuje DOS-ov proces a takodje oslobodo operater od dugotrajnog unošenja komandi u DOS linijama.

Predmet rada

1. Napisati *batch* program koji upravlja radom PC mašine.
2. Koristiti ECHO komandu radi prikazivanja niza karaktera na video displeju.

3. Koristiti IF, IF ERRORLEVEL, i GOTO radi modifikacije toka izvršenja *batch* programa.
4. Napisati *batch* program koji koristi izmenjive parametare.
5. Koristi CHOICE komandu u *batch* fajlu da bi se očitao izbor tokom izvršavanja *batch* programa.

Postupak rada

Batch programi su moćno sredstvo za automatizovanje procesa kod PC mašine. Kroz nekoliko koraka, u ovoj Laboratorijskoj vežbi, kreiraćemo *batch* progam.
Primer 3-1 pokazuje tipičan *batch* fajl koji se koristi za formatiranje 3 ½ inčne flopi diskete u formatu DSDD (*double sized double densit*) sa kapacitetom memorisanja od 720 k ili HD (*high density*) sa kapacitetom 1.44 M na disk drajv jedinici A(B). Uočimo na koji način ovaj *batch* fajl koristi IF ERRORLEVEL, GOTO, i CHOICE iskaze da bi ispitali željeni format diskete. Takodje obratimo pažnju na način korišćenje iskaza ECHO da bi se na video displeju pokazala poruka kao i iskaza ECHO. koji se koristi za prikaz blanko karaktera na video displeju.
Iskaz CHOICE prikazuje zahtev "Tvoj izbor" i očekuje da primi A ili B kao odgovor koji se unese sa tastature. Ako se otkuca slovo B, ERRORLEVEL je 2 a ako se otkuca slovo A ERRORLEVEL je 1. Važno je da se prvo uvek testira najviši ERRORLEVEL broj zbog toga što iskaz IF testira prvo uslov veći od ili jednak. Iskaz REM je napomena koja se ne izvršava ali se koristi da bliže pojasni odredjeni deo *batch* fajla. Labele u *batch* fajlu uvek počinju sa (:) i označavaju mesto grananja unutar tog fajla.

Primer 3-1

```
@ECHO OFF
REM     Batch fajl koji formatira disketu u disk jedinici A
ECHO.
ECHO A    Formatiranje diskete u disk drajv jedinici A kapaciteta 720 k
ECHO B    Formatiranje diskete u disk drajv jedinici A kapaciteta 1.44 M
ECHO.
CHOICE  /C:AB Tvoj izbor
IF ERRORLEVEL 2 GOTO HD
IF ERRORLEVEL 1 GOTO DSDD
:HD
ECHO.
FORMAT A: -F:1440 /U
GOTO END
:DSDD
ECHO.
FORMAT A: /F:720 /U
:END
```

KORAK 1: Koristiti program EDIT da bi se kreirao i uredio *batch* fajl koji se naziva FORM.BAT. Prekopirati sadržaj Primera 3-1 u *batch* fajl i snimiti ga. Zatim, pokrenuti ovaj *batch* fajl tako što će se te otkucati FORM u DOS komandnoj liniji.

U predvidjenom prostoru, kopirati isključivo ono što se vidi na video displeju pre odabiranja A ili B sa tastature.

KORAK 2: Sada nakon što su neke od *batch* komandi razumljive, koristiti EDIT kako bi kreirali *batch* fajl koji se naziva SELECT.BAT. Ovaj fajl prikazuje meni iz Primera 3-2. Vaš *batch* program mora takodje koristiti komandu CHOICE da bi obezbedili korektni izbor od 1 do 4 i za slučaj da je IF ERRORLEVEL korektan formatirajte disketu iz disk drajv jedinice A ili B kao DSDD ili HD respektivno.

Primer 3-2

```
Formatiranje 3 ½ flopi diskete u

1 – Disk drajvu A kao HD
2 – Disk drajvu B kao DSDD
3 – Disk drajvu B kao HD
4 – Disk drajvu B kao DSDD

Unesite izbor (1,2,3,4):
```

Izmenjivi parametri se često koriste kod *batch* fajlova. Simbol %1 se koristi za predstavlljanje prvog parametara nakon unošenja imena *batch* programa u DOS komandnoj liniji, simbol %2 se koristi za prestavu drugog parametara, itd. Simbol %0 se odnosi na komandu. Primer 3-3 prikazuje listing jednog kratakog *batch* fajla koji se naziva RO.BAT a koristi da modifikuje *read-only* (sadržaj fajla se može samo pročitati dok njegovo modifikovanje nije dozvoljeno) osobinu (atribut) svih fajlova u direktorijumu. Komanda RO C:\DATA ON menja osobinu svih fajlova u direktorijumu C:\DATA u osobinu *read-only*. Komanda RO C:\DATA OFF menja osobinu svih fajlova u direktorijumu C:\DATA u osobinu *read-write* (sadržaj fajla se može modifikovati tj. fajl se može čitati a takodje i na njemu upisivati). Primetiti kako se simbol %2 koristi da bi se promenio parametar ON ili OFF iz komandne linije. Takodje obratiti pažnju kako se dobija ime fajla koristeći simbol %1.

Primer 3-3

```
@ECHO OFF
CD %1

IF “%2” ==”ON” GOTO ON
IF “%2” ==”OFF” GOTO OFF

ECHO.
ECHO Izbor mora biti ON ili OFF
ECHO.
GOTO END

:ON
ATTRIB +R*.*
GOTO END

:OFF
ATTRIB -R*.*

:END
```

KORAK 3: Koristeći program EDIT, napisati *batch* program nazvan FORMS:BAT koji koristi izmenjivi parametar radi formatiranja diskete u disk drajv jedinici A ili B kao 3 ½ DSDD ili 3 ½

HD disketu. Koristiti izamenjivi parametar %1 za disk drajv jedinicu A ili B a %2 za tip formatiranja DSDD ili HD. Sintaksa komandne linije za ovaj program je neka od sledećih:

FORMS A HD
FORMS A DSDD
FORMS B HD
FORMS B DSDD

Pitanja

1. Testirajti @ECHO OFF i @ECHO ON da bi uočio efekte na vido displeju koji su posledica izvršenja *batch* programa. Kakvo efekat i kakav tip prikaza na video displeju uzrokuje svaka od komandi u *batch* programu.
2. Komanda CHOICE koristi bilo koji parametar, ali uvek vraća _____ kao prvi parametar, _____ kao drugi parametar, itd.
3. Šta se prikazuje sledećom komandom:

CHOICE /C:XYZ Koje slovo

4. Objasniti kako IF ERRORLEVEL komanda testira vrednost ERRORLEVEL
5. Kojim karakterom moraju početi sve labele koje se koriste sa iskazom GOTO?
6. Šta će na video displeju prikazati sledeći *batch* program ako je on nazvan RO.BAT?

poruku
@ECHO
ili poruku
ECHO %0.

7. Napisati iskaz IF koji se grana na labeli :TEMP za slučaj da drugi parametar (%2) je "WHAT"
8. Kakav se efekat ostvaruje izvršenjem iskaza ECHO.?
9. Kakav se efekat ostvaruje izvršenjem iskaza REM u *batch* fajlu?
10. Kreirati *batch* program koji prikazuje sledeći meni i izvršava komande koje su prikazane u tom meniju.

1 – Normalni prikaz direktorijuma
2 – Prošireni prikaz direktorijuma
3 – Proširoki prikaz direktorijuma sa pauzom

Unesiti izbor (1,2,3):

LABORATORIJSKA VEŽBA 4

Korišćenje programa Debug

Uvod

Program DEBUG je sastavni deo DOS-a namenjen za pisanje, izmenu i korigovanje asemblerskih programa za mikroprocesore familije 80x86. Osim što omogućuje izmene programa, DEBUG poseduje i kompletan skup komandi kojima se može prikazati sadržaj memorije ili registara kao i izvršavati program - u celosti ili koračno.

Predmet rada

1. Korišćenje debagera za pregled sadržaja memorije i prikaz sadžaja bilo kog registra.
2. Pisanje kratkih sekvenci u asemblerskom jeziku korišćenjem DEBUG mini-asembler programa.
3. Koračno izvršavanje programa korišćenjem funkcije T (Trace).
4. Disasembliranje programa iz memorije, korišćenjem funkcije U (Unassemble).
5. Debagiranje programa korišćenjem funkcije G (Go) sa prekidnim tačkama.

Postupak rada

Prvi deo ove Laboratorijske vežbe je informativan iz razloga što on predvidja listu komandi u DEBUG-u koji ilustruju, objašnjavaju korišćenje neke DEBUG komande. Tabela 4-1 prikazuje listu DEBUG komandi sa nekoliko primera svake komande pri njenom najčešem korišćenju. Treba naglasiti da se lista ovih komandi prikazuje kada se u DEBUG-u unese karakter ?.

Komanda	Primer	Opis
A	A1000:1200	Asemblira program počev od segmentne adrese 1000H i ofset adrese 1200H (fizička adresa 11200H)
	A100	Asemblira program počev od adrese u tekućem kodnom segmentu sa ofset adresom 100H
C	C100,1FF,400	Uporedjuje sadržaj bloka memorije (od 100H do 1FFH) sa blokom memorije koji počinje od adrese 400H. Podudarnost se ne prikazuje na displeju.
D	D1000,1100	Prikazuje sadžaj memorijskih lokacija počev od 1000H do 1100H u heksadecimalnom i ASCII formatu
	D1200	Prikazuje sadžaj memorijskih lokacija u tekućem segmentu podataka počev od ofset-adrese 1200H
E	E2000:1000	Unosi podatak u obliku heksadekadnog bajta u segment na adresi 2000H sa ofsetom 1000H (fizička adresa 21000H)
F	F100,200,22	Puni memorijske lokacije od 100H do 200H podatkom 22H
G	G=100,104	Izvršava program počev od ofset adrese 100H u tekućem kodnom segmentu sa zaustavljanjem na ofset adresi 104H (break point)
H	H200,100	Nad podacima 200H i 100H preduzima heksadecimalne aritmetičke operacije sabiranja i oduzimanja. Prikazuje zbir i razliku (300H i 100H).
I	I20	Prihvata podatak sa ulazno-izlaznog porta broj 20H
L	L	Puni, tj loaduje fajl čije je ime najpre specificirano komandom N
M	M100,200,500	Pemera adržaj lokacija od 100H do 200H u memorijski prostor počev od 500H
N	N C:\FILEX.TXT	Specificira ime fajla koji će se napuniti u memoriju komandom L ili upisati iz memorije u fajl korišćenjem komande W
O	O20,10	Šalje podatak 10H na ulazno-izlazni port broj 20H
P	P=1000	Predvidja izvršenje od adrese 1000H i vraća upravljanje debageru posle instrukcije ili potprograma pozvanog sa adrese 1000H
Q	Q	Završetak rada i povratak u DOS
R	R	Prikazuje sadržaj svih registara
	RCS	Prikazuje sadržaj registra kodnog segmenta (CS)
S	S100,200,12	Pretražuje memoriju počev od 100H pa do 200H i podatak 12H. Prikazuje adresu podatka.

T	T3	Koračno izvršava sledeće tri instrukcije uz prikaz instrukcija i sadržaja registara.
U	U100 U100,120	Disasemblira program koji počinje u kodnom segmentu od lokacije 100H Disasemblira sekvencu u kodnom segmentu koja počinje od adrese 100H, a završava na adresi 120H
W	W	Upisuje sadržaj memorije u fajl specificiran komandom N. Naglasimo da BX:CX sadži broj bajtova koji će se upisati u fajl
XA	XA3	Alocira 3 stranice veličine po 16K expanded memorije ako je instaliraran EMM386.EXE ili neki drugi menadžer expanded memorije
XD	XD1	Dealocira memoriju korišćenjem handle 1
XM	XM 1,2,3	Preslikava expanded memoriju korišćenjem lokalne stranice 1 u fizičku stranicu 2, korišćenjem handle-a broj 3
XS	XS	Prikazuje statusne informacije *expanded* memorije

Tabela 4-1. DEBUG komande

Komande D (*Display*) i E (*Enter*)

Komanda D koristi se za prikaz sadržaja memorije u heksadekadnoj i ASCII formi. Parametri koji idu uz D komandu specificiraju početnu lokaciju i, ako je neophodno, završnu lokaciju oblasti memorije koja se prikazuje.
Primeri:

-D100,400
-D1200:0000

KORAK 1: Pozvati program DEBUG iz DOS-ovog komandnog *prompta*. *Prompt* DEBUG programa je znak minus (-). Otkucati komandu:

-D0010:0000

Na ekranu je prikazana memorija počev od adrese 0010H i ofseta 0000H, tj. fizičke adrese 00100H+0000H=00100H. Fizička adrea formira se proširenjem segmentne adrese (u ovom lsučaju to je 0010H) cifrom 0 i heksa sabiranjem sa ofset adresom. U predvidjenom praznom prostoru napisati prvu liniju koja je prikazana ovom komandom.

KORAK 2: Otkucati komandu D uz DEBUG *prompt*, bez ikakvih parametara. Napisati u predvidjenom prostoru šta se dešava pri tome.
Na ovaj način moguće je prikazati sadržaj bilo kojeg dela memorije iz prostora prvog megabajta.

KORAK 3: Otkucati E1000:0000 uz DEBUG komandni *prompt*. Program DEBUG je obavešten da će podaci koje unosimo biti smešteni počev od memorijske lokacije 10000H (1000:0000). Otkucati komandu:

-E1000:0000

Komanda E ispisuje sadržaj memorijske lokacije i čeka novi podatak ili blanko za prelaz na sledeću memorijsku lokaciju.Taster Enter označava kraj unosa podataka. Uneti vrednosti 12H, 13H i 14H u tri sukcesivne lokacije, počev od adrese 10000H.

KORAK 4: Radi provere prethodnog unosa, otkucati D1000:0000 za prikaz sadržaja memorije.

Komande A (*Assemble*) i U (*Unassemble*)

Komanda A pristupa mini-asembler programu okviru DEBUG programa. Naziv "mini-asembler" koristi se zbog toga što ovaj program nema sve karakteristike pravog asemblerskog prevodioca. Primera radi, nema labela za memorijske lokacije ili programske skokove. Jedini način pristupa nekoj memorijskoj lokaciji ili adresi, kod korišćenja komande A, jeste navodjenje heksadecimalne adrese. U sledećem primeru demonstrirana je upotreba ove komande:

```
-A1000:0000
```

Program DEBUG prikazuje adresu 1000:0000 i čeka unos simboličke instrukcije. Pretpostavimo da treba brojeve 12H, 13H i 14H smestiti u memoriju, kao u prethodnom primeru komandom E. To se postiže na sledeći način:

```
-A1000:0000
1000:0000    DB  12,13,14
1000:0003
-D1000:0000, 0002
1000:0000    12  13  14
```

Pretpostavimo da je u memoriju unet program koji ispisuje poruku: *Zdravo kolege*. Da bi se niz karaktera prikazao na displeju, koristi se DOS funkcijski poziv broj 9. Ova funkcija zahteva da registarski par DS:DX ukazuje na niz karaktera i da je vrednost u AH registru jednaka 09H, pre nego se izvrši instrukcija INT 21H koja znači pristup DOS funkcijama. Takodje, naglasimo da niz karaktera mora da se završava sa $. U primeru koji sledi niz karaktera smeštamo od adrese 1000:0000, a program od adrese 1000:0000:

```
-A1000:0100
1000:0000    DB  'Zdravo kolege$'
1000:0111
-A1000:0000
1000:0000    MOV     AX,1000
1000:0003    MOV     DS, AX
1000:0005    MOV     AH, 9
1000:0007    MOV     DX, 100
1000:000A    INT 21
1000:000C    INT 3
-
```

KORAK 5: Uz pomoć komande U (*Unassemble*) proveriti da li je program smešten u memoriju, tj. prikazati ga uz pomoć sledeće DEBUG komande:

```
U1000:0000
```

Ako je program korektno smešten u memoriju, moguće je izvršavati ga.

Komanda G (*Go*)

Komanda G izvršava program sa bilo koje početne lokacije. Na primer, G=1000:0000 komanda ukazuje DEBUG programu da izvrši program koji počinje na lokaciji 1000:0000. Znak =mora da prethodi adresi početka. Ukoliko je neophodna i tačka prekida (adresa završetka programa), ona se navodi kao drugi parametar. Komanda G dozvoljava do deset adresa tačaka prekida.

KORAK 6: Izvršiti prethodni program sledećom G komandom:

```
G=1000:0000, 000C
```

DEBUG će izvršiti program od adrese 1000:0000 pri čemu će se izvršenje okončati na adresi 000C. Sadržaj svih registara prikazan je u trenutku kada je programska sekvenca stala u tački

prekida. Alternativni način završavanja programa jeste završna instrukcija INT 3 na kraju programa (*break point command*).

Prikaz registara komandom R (Register) i T (Trace)

Komanda R može da se koristi u bilo kom trenutku za vreme rada DEBUG programa sa ciljem da se prikaže sadržaj svih registara ili nekog registra pojedinačno. Ukoliko se prikazuje jedan registar, njegov sadržaj može se menjati unosom nove vrednosti.

KORAK 7: Otkucati R uz DEBUG *prompt* i opisati tačno šta se pojavljuje na ekranu.
Sadržaj marker registra prikazan je na kraju druge linije u obliku skraćenica sa dva slova za svaki marker. U tabeli koja sledi dati su markeri i njihov kôd:

marker	obrisan	postavljen
Zero	NZ (not zero)	ZR (zero)
Carry	NC (no carry)	CY (carry)
Sign	PL (plus)	NG (negative)
AuxCarry	NA (no aux carry)	AC (aux carry)
Parity	PO (parity odd)	PE (parity even)
Overflow	NV (no overflow)	OV (overflow)
Direction	UP (increment)	DN (decrement)
Interrupt	DI (disabled)	EI (enabled)

Tabela 4-2. *Flag* kôdovi

U prethodnom primeru sadržaj CS:IP modifikuje se uz pomoć RCS i RIP komandi DEBUG programa. To je vrlo važno za komandu W. Ime programa zadaje se komandom N. Dužina programa (počev od 0100H) smešta se uvek u BX:CX. Zatim se vrši upis. Upozorenje: ne specificirati nikakve parametre uz komandu W, kako se ne bi desilo oštećenje sistemskih fajlova na disku.

Primer programa

Sada kada vam je većina komandi DEBUG programa razumljiva, treba napisati program koji prikazuje vaše ime na video displeju i prelazi na izvršenje programa u *debug* režim rada kada je preko tastature otkucan upravljački karakter C (03H), ili prikazuje vaše ime ako je pritisnuta bilo koja druga dirka. Da bi se kreirao ovaj program potrebno je koristiti funkcijski poziv DOS INT 21H, funkcija 0AH, (videti Laboratorijsku vežbu 8) kako bi se prikazaolo vaše ime i DOS funkcijski broj AH = 06H ili AH = 07H (bez efekta ehoa) kako bi se očitao podatak otkucan na tastauri i istovremeno proverilo dali je otkucan karakter za povratak C (03H).
Kreirajte ovaj program i unesite ga u PC mašine koristeći komandu A za potrebe asembliranja i komandu G za potrebe izvršenja programa. Ako želite da odštampate listing programa otkucajte U u DEBUG *promptu* koristeći početnu i krajnju adresu programa. Na primer, ako program počinje sa adrese 1000:0000 i završava se na adresi 1000:0030, treba otkucati komandu U1000:0000,0030 u DEBUG *promptu*. Pre nego što pritisnete *enter* dirke treba pritisnuti *control* P da bi se preusmerio izlaz ka štampaču. Nakon što prvo *control* P preda podatke ka štampaču i monitoru, drugo *control* P vraća izlaz na monitor. Nakon toga DEBUG *prompt* vrati tip *control* P i *enter* i zaustavlja štampanje.

Pamćenje fajlova

Mada se DEBUG ne koristi za ozbiljan rad koji manipuliše sa asemblerom, može se javiti situacija kada postoji potreba da se zapamti neki jednostavan program napisan uz pomoć DEBUG-a.

Da bi ste mgli da zapamtie vaš rad, program mora biti napisan kao .COM fajl. Fajl .COM mora početi sa ofset adrese 0100H. To znači da prva naredba u ovom programu mora početi sa ofset adrese 0100H kako bi se isti uspešno izvršavao. Ime fajla, kako je to definisano sa N (*name*) komandom za debagiranje mora da sadrži ekstenziju fajla .COM u protivnom program se ne može izvršavati. Obim programa u ovom slučaju je ograničen na ne više od jednog segmenta za sve podatke i program.

Na primer, ako kreirani program počinje sa adrese 1000:0100 i završava se na adresi 1000:021CH, on se može zapamtiti pomoću komande W (*write*) nakon što je ime već generisano komandom N. Primer 4-4 pokazuje niz koraka koji se koristi za pamćenje ovog programa pod imenom MYPROG.COM u *root* direktorijumu disk drajv jedinice C.

Primer 4-4

```
-RCS 2000-06-02
:1000
-RIP  1002
:0100
-NC:\MYPROG.COM
-RBX  0200
:0000
-RCX  0200
:RCX  0200
:011C
-W
```

U ovom primeru sadržaj registarskog para CF:IP se modifikuje korišćenjem komande RCS i RIP DEBUG komandama. Veoma je važno da ovi registri adresiraju prvu instrukciju u programu u bilo kom segmentu sa ofset adrese 0100H koja je smeštena u registu IP. Ime programa se pamti komandom N. Konačno dužina programa (računajući od adrese 0100H) se smešta u registarski par BX:CX pre nego što se koristi komanda W kako bi se zapamtio sadržaj tog fajla. Treba pri ovome biti veoma obazriv kako se ne bi specificirao neki nevalidan parametar za komandu W, jer u tom slučaju može doći do oštećenja informacije na disku tako što će se vršiti dopisivanje preko već postojećih sistemskih fajlova.

Izlaz u DOS

Za izlazak u DOS i povratak upravljanja DOS komandnom procesoru, treba otkucati Q uz DEBUG *prompt*. Na taj način se završava program DEBUG i pojavljuje DOS *prompt*.

Pitanja

1. Formulisati DEBUG komandu koja prikazuje sadžaj memorijskih lokacija od 12000H do 12023H na displeju.
2. Komanda D1000:0300 ispisuje sadržaj memorije u heksadekadnom formatu, počev od neke fizičke adrese. Koja je to adresa?
3. Upisati podatak 3AH u memorijsku lokaciju 12300H koristeći komandu E.
4. Sa koje memorijske lokacije počinje asembliranje kada se unese komanda A1000:0100?
5. Uneti komandu G kojom će se izvršavati program od adrese: segmenta - 1000H, ofseta 0200H, (1000:0200) sa završetkom na lokaciji 0300H (1000:0300).
6. Šta se postiže instrukcijom INT 3.
7. Ako je AH=09H, DS:DX=1000:3000 i izvrši se INT 21, šta se postiže?
8. Šta se izvršava sledećim programom ako se unese i izvrši iz DEBUG-a?

```
MOV     AH,6
```

```
MOV   DL,41H
INT   21H
INT 3
```

9. Kako se program štampa u DEBUG-u?
10. Opisati kako se program u heksadecimalnom formatu prenosi na štampač od lokacije F0000H do FFFFFH.
11. Koliko se prekidnih tačaka može uneti korišćenjem komande G?
12. Napisati program koji prikazuje vaše ime, adresu i broj telefona, na displeju.
13. Koji su parametri neophodni za izvršenje instrukcije snimanja programa u memoriji komandom W?
14. Sledeći program prikazuje informaciju o PC mašini. Unesi program sa ofset adrese 0100H i pomoću komande W snimi ga u root direktorijumu kao C:\MZPROG.COM.

```
        -A1000:0100
MOV   AX,CS
MOV   DS,AX
MOV   AH,9
MOV   DX,120
INT 21H
        -A1000:0120
DB 'RACUNAR JE VLASNISTVO EL. FAK.$'
```

15. Koja je namena komande Q?

LABORATORIJSKA VEŽBA 5

Uvod u Programmer's WorkBench i Code View

Uvod

Programiranje PC mašine na makroasemblerskom jeziku korišćenjem Microsoft-ovog programskog paketa MASM verzije 6.X olakšano je upotrebom programa *Programmer's WorkBench* (PWB), *QuickHelp* i *CodeView*. PWB je editor koji koristi skup opcija za upravljanje procesom linkovanja asemblerskih programa što razvoj programa čini gotovo automatizovanim. Program *QuickHelp* olakšava pristup helpu koji se odnosi na asemblerske instrukcije, DOS INT 21H funkcijske pozive kao i BIOS funkcijske pozive. Program *CodeView* je unapredjena verzija DEBUG-a koji omogućava debagiranje programa u simboličkoj formi. Ovi programi su deo MASM pragramskog paketa.

Predmet rada

1. Upoznavanje sa Microsoft-ovim MASM asemblerskim programom.
2. Upotreba PWB-a za kreiranje i asembliranje asemblerskog programa.
3. Upotreba *CodeView*-a za debagiranje i izvršavanje programa na asemblerskom jeziku.
4. Upotreba *QuickHelp*-a za pomoć kod instrukcija i asemblera.

Postupak rada

Microsoft-ov MASM asemblerski paket kao konpletno okruženje potrebno za programiranje na asemblerskom jeziku sastoji se iz četri osnovna dela:

1. asembler i linker;
2. *QuickHelp* i *online help*;
3. PWB (*Programmer's Work Bench*);
4. *CodeView*.

Kod Microsoft-ove verzije 6.1 asembler i linker su integrisani kao jedinstveni program nazvan ML.EXE. Ovaj asembler ima sve osobine kao i prethodne i potpuno je komplementaran sa ranijim programima. Program ML.EXE zamenjuje kod ranijih verzija zasebne programe MASM.EXE (asembler) i LINK.EXE (linker). Programu ML.EXE se pristupa preko PWB-a a ne preko DOS komandne linije.
Online help-u se pristupa preko DOS komandne linije sa komandom QH.EXE (*QuickHelp*) ili često preko PWB-a. Kada radimo u okruženju PWB-a pomoć možemo dobiti pritiskom na taster F. Online Help osobina daje kompletna objašnjenja koja se tiču skupa instrukcija, asemblera i njegovih osobina, PWB-a i *CodeView*-a. Online prikazuje sve DOS INT 21H funkcijske pozive kao i BIOS funkcije koje su raspoložive programeru.
Programu PWB se pristupa preko DOS komandne linije ukucavanjem komande PWB. Program PWB koordinira asembliranje programskog zadatka tako što je omogućen pristup editoru, *online help*-u, asembleru i *CodeView*-u radi debagiranja.
CodeView prikazuje na ekranu program u simboličkoj formi i omogućava izvršenje instrukcija u realnom vremenu, u grupama instrukcija (princip prekidnih tačaka), kompletnog programa. Debagiranje se izvršava prikazivanjem sadržaja registara i memorijskih segmenata na ekranu u različitim prozorima. Ukažimo da miš nije neophodan za upotrebu PWB paketa, ali se preporučuje.

Korišćenje PWB-a

Početni izgled ekrana nakon aktiviranja PWB-a prikazan je na slici 5-1. Duž vrha ekrana je meni bar koji sadrži 9 padajućih menija pomoću kojih se pristupa različitim delovima PWB-a. Kao kod svih Microsoft-ovih programa za aktiviranje padajućih menija upotrebljava se miš. Ukoliko miš nije dostupan aktiviranje se vrši istovremenim pritiskom ALT tastera i slova kojim počinje ime menija.

Slika 5-1 Glavni ekran PWB-a

KORAK 1: Aktivirajte program PWB ukucavanjem PWB u DOS komandnoj liniji. Po aktiviranju PWB-a, selektujte *File* padajući meni mišem ili ALT+F kombinacijom tastera. Zatim selektujte *New* za početak novog fajla. Sada ste spremni za početak unošenja asemblerskog programa. Ukucajte program čiji je listing dat u Primeru 5-1.

Primer 5-1

```
CODE   SEGMENT  'code'          ;početak novog CODE segmenta
ASSUME  CS:CODE         ;identifikuj CODE kao kodni segment
MAIN   PROC   FAR               ;početak MAIN procedure
MOV    AH,1             ;čitanje dirke sa efektom ehoa
INT    21H
PUSH   AX               ;čuvanje ASCII kôda u magacinu
MOV    AH,2             ;prikazivanje znaka '=' na ekranu
MOV    DL,'='
INT    21H
POP    AX               ;vraćanje ASCII koda iz magacina
CALL   DISP             ;poziv potprog. za znaka na ekranu
MOV    AX,4C00H         ;prelazak na DOS
INT    21H
MAIN   ENDP                     ;kraj MAIN procedure
DISP   PROC   NEAR          ;početak NEAR procedure
PUSH   AX           ;stavljanje sadržaja AX u magacin
MOV    CL,4
ROR    AL,CL        ;rotiranje u desno za CL broj puta
CALL   CONV         ;konvertuj i prikaži MS cifru
POP    AX           ;vraćanje ASCII koda iz magacina
CALL   CONV         ;konvertuj i prikaži LS cifru
RET
DISP   ENDP                 ;kraj DISP procedure

   CONV   PROC   NEAR           ;početak proc. za konvertov. i prikaziv. na ekranu
   AND   AL,0FH         ;maskiranje višeg dela bajta
   ADD   AL,30H         ;konvertovanje u ASCII znak
   CMP   AL,3AH
   JB    CONV1          ;ukoliko je ASCII znak 0-9
   ADD   AL,7           ;ukoliko je od A do F
  CONV1: MOV   DL,AL            ;prikazivanje sadržaja AL
   MOV   AH,2
   INT   21H
   RET
CONV   ENDP                 ;kraj CONV procedure
```

```
CODE    ENDS                    ;kraj kodnog segmenta
END     MAIN                    ;kraj programa
```

Ovaj program će biti upotrebljen kroz većinu vežbi za učenje korišćenja PWB-a i njegovih karakteristika. Pre unošenja programa, poželjno je dati ime programu. Za imenovanje programa selektujte *Save* iz *File* padajućeg menija. Unesite željeno ime fajla sa ekstenzijom .ASM za izvorni fajl u asemblerskom jeziku. Pošto su ime i put direktorijuma gde žemimo da snimimo fajl odabrani, odaberite OK kao potvrda novog imena fajla. Nakon ovog PWB program vraća upravljanje editorskom programu tako da se program može uneti.

Program u Primeru 5-1 je dokumentovan komentarima (delovi programskih linija koji počinju sa "; "). Ovaj program je smešten u jedan segment nazvan CODE. Programski paket *CodeView*-a koristićemo kasnije u ovoj vežbi za objašnjenje primera. Zbog toga iza direktive SEGMENT sledi reč 'code' koja ukazuje *CodeView*-u da je ovaj segment upravo kôdni segment. Direktiva 'Code' takodje omogućava *CodeView*-u da prikaže kodni segment u simboličkoj formi kao što je to prikazano u Primeru 5-1. Reč 'data' se upotrebljava iz istog razloga da bi prikazali segment podataka u simboličkoj formi uz pomoć *CodeView*-a. Analizirani program iz Primera 5-1 koristi tri DOS INT 21H funkcijska poziva radi prisupa osobinama DOS-a:

1. Funkcija AH=1 čita tastaturu i sa efektom ehoa i vraća ASCII kôdirani znak u registar AL.
2. Funkcija AH=2 prikazuje ASCII kôdirani znak iz registra DL na video displej.
3. Funkcija AH=4CH omogućava prelazak na DOS na kraju procedure main. Ukažimo da AH=00H ukazuje na to da ne postoji greška kada se iz programa koga izvršavamo prelazi na DOS. Postavljena vrednost u AL se može ispitati u *batch* fajlu pomoću komande IF-ERROR LEVEL.

Ovaj primer takodje ilustruje dekompoziciju i prikazivanje ASCII kôdiranog znaka unetog sa tastature u heksadecimalnom formatu. Procedura CONV konvertuje viša četri bita registra AL u ASCII kôdirani karakter za brojeve od 0 do 9 i slova od A do F. Potom se ovi karakteri prikazuju na video displeju.

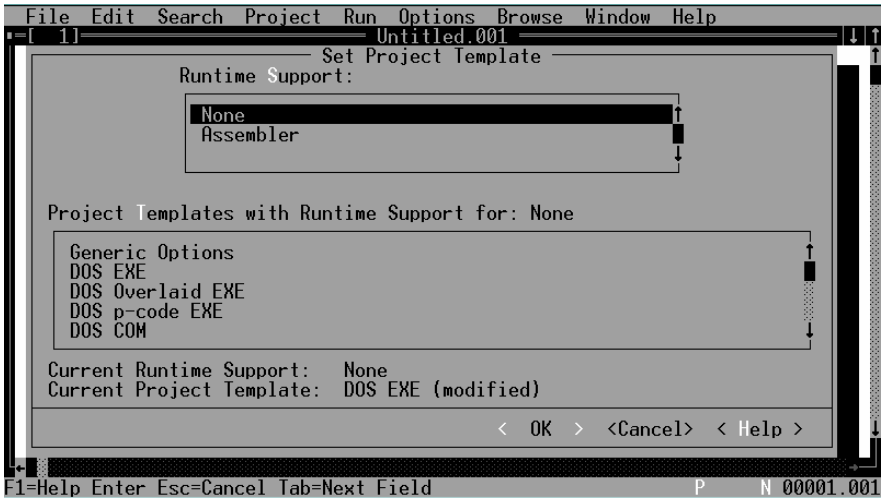
KORAK 2: Pošto je program iz Primera 5-1 unešen u editor PWB-a, PWB mora biti konfigurisan da asemblira programa u željenom formatu, npr. .EXE, .COM itd. Konfigurisanje se ostvaruje pomoću *Option* padajućeg menija. Selektujte meni *Options* i odaberite *Project Templates* kao ulaz. Zatim odaberite *Set Project Template* kao ulaz iz sledećeg direktorijuma. (Slika 5-2 ilustruje *Set Project Template* dijalog box.)
Sekcija *Runtime Support* omogućava izbor izmedju *None* ili *Assembler*. Uobičajeni izbor je *None* jer većina programa ne zahteva *Runtime Support* iz posebnih, izdvojenih *Runtime* biblioteka. *Runtime* biblioteke su predvidjene za kombinovanje programa na asembleru i višim programskim jezicima (Pascal, C/C++) ili kod Windows programiranja. Selektujte zatim DOS.EXE opciju za generisanje DOS.EXE izvršnog fajla za asembler i linker. Kada su *None* i DOS.EXE selektovani odaberite OK sa dna prozora kako potvrdu unšenih opcija.

KORAK 3: Sada, pošto je project definisan, selektujte *Build Options* u *Options* meniju. Ovo odredjuje tip programa razvijenog asemblerom i *builder* programom. Postoje dve *build* opcije, jedna selektuje izvršnu verziju a druga verziju za debagiranje. Odaberite *Debug Option*.

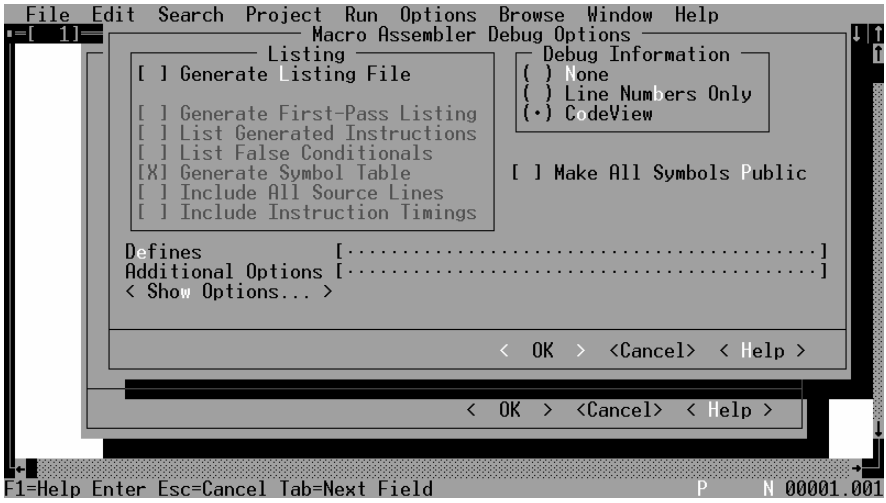
KORAK 4: Selektujte zatim *Language Options* iz *Options* menija. Potom selektujte *Set Debug Options* meni. (Slika 5-3 prikazuje *Macro Assembler Options* meni.) Treba naznačiti da se *warnings* (upozorenja) ne moraju tretirati kao greške.
U većini slučajeva postoji potreba da se selektuje opcija *Generate Listing File*-a iz ove liste, ukoliko to već nije selektovano. Ostale opcije uključuju tajminge instrukcija, kôd izvornih linija itd. Osobina *Generate Listing File* generiše .EXE fajl i .LST (*listing*) fajl. *Listing* fajl sadrži

izvorni program i objektni program u okviru istog listinga i često se koristi kao dokumentacija programa.

KORAK 5: Sada kada je PWB konfigurisan, projekat se može kreirati. Da bi kreirali .EXE fajl selektujte *Project* meni i odaberite *Build*. Posle nekoliko trenutaka sistem će kreirati program iz Primera 5-1 i pitati vas da li želite da ga izvršite (pokrenete kreirani program) ili debagirate. Možete prihvatiti i upozorenje u vezi magacina. Ono se može ignorisati ukoliko asemblirani program ne koristi više od 128 bajtova magacinskog prostora. Naravno, program iz primera upotrebljava mnogo manje memorijskog prostora za potrebe magacina.
Na ovaj način kreirali smo .EXE fajl kao i .LST fajl. Odaberite *Cancel* opciju za povratak u PWB. Pogledajte sada asemblirani fajl selektovanjem *Open* ispod *File* menija. Otvorite .LST fajl. Generisani LST fajl je prikazan u Primeru 5-2, i treba da bude identičan onom koji se vidi displeju.



Slika 5-2 Meni za postavljanje Project Template-a



Slika 5-3 Meni Macro Assembler Debug Options

Primer 5-2

```
Microsoft (R) Macro Assembler Version 6.11    04/03/98    11:23:45
0000        CODE    SEGMENT 'code'          ;početak novog CODE segmenta
ASSUME CS:CODE      ;identifikuj CODE kao kodni segment CS
0000        MAIN    PROC    FAR             ;početak MAIN procedure
0000 B4 01          MOV     AH,1            ;čitanje dirke sa ehom
0002 CD 21          INT     21H
0004 50             PUSH    AX              ;čuvanje ASCII koda u magacinu
0005 B4 02          MOV     AH,2            ;prikazivanje znaka '=' na ekranu
0007 B2 3D          MOV     DL,'='
0009 CD 21          INT     21H
000B 58             POP     AX          ;vraćanje ASCII koda iz magacina
000C E8 0005        CALL    DISP        ;poziv  potprograma za prikaz znaka na ekranu
000F B8 4C00        MOV     AX,4C00H    ;prelazak na DOS
0012 CD 21          INT     21H
0014        MAIN    ENDP                ;kraj MAIN procedure

0014        DISP    PROC    NEAR        ;početak NEAR procedure
0014 50             PUSH    AX          ;stavljanje sadržaja AX u magacin
0015 B1 04          MOV     CL,4
0017 D2 C8          ROR     AL,CL       ;rotiranje u desno za CL broj puta
0019 E8 0005        CALL    CONV        ;konvertuj i prikaži MS cifru
001C 58             POP     AX          ;vraćanje ASCII koda iz magacina
001D E8 0001        CALL    CONV        ;konvertuj i prikaži LS cifru
0020 C3             RET
0021        DISP    ENDP                ;kraj DISP procedure

0021        CONV    PROC    NEAR        ;početak proc. za konvert. i prikaz. na ekr.
0021 24 0F          AND     AL,0FH      ;maskiranje višeg dela bajta
0023 04 30          ADD     AL,30H      ;konvertovanje u ASCII znak
0025 3C 3A          CMP     AL,3AH
0027 72 02          JB      CONV1       ;ukoliko je ASCII znak 0-9
0029 04 07          ADD     AL,7        ;ukoliko je od A do F
002B        CONV1:
002B 8A D0          MOV     DL,AL       ;prikazivanje sadržaja AL
002D B4 02          MOV     AH,2
002F CD 21          INT     21H
0031 C3             RET
0032        CONV    ENDP                ;kraj CONV procedure
0032        CODE    ENDS                ;kraj kodnog segmenta
```

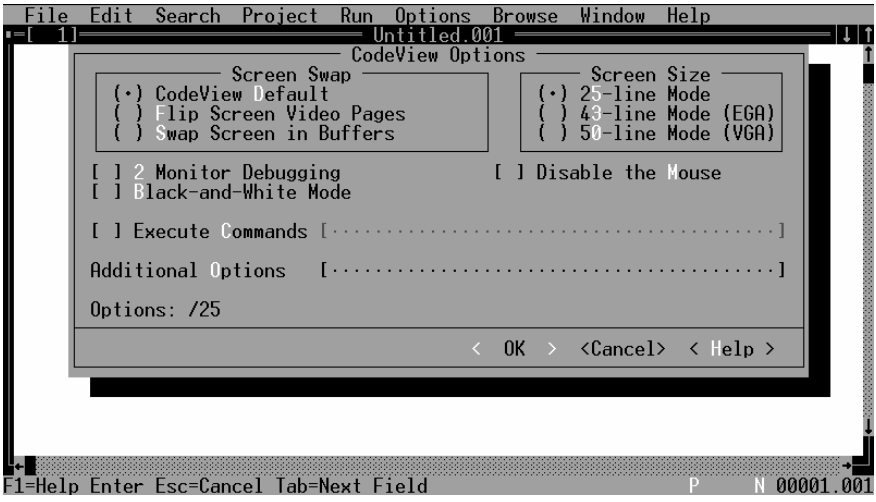
```
END    MAIN            ;kraj programa
```

```
Microsoft (R) Macro Assembler Version 6.11        04/03/98  11:23:45
test.asm                Symbols 2-1
Segments and Groups:
    N a m e      Size      Length    Align     Combine     Class
    CODE         16 Bit    0032      Para      Private     'CODE'
Procedures, parameters and locals:
    N a m e          Type        Value       Attr
    CONV             P Near      0021        CODE Length=0011 Public
    CONV1            L Near      002B        CODE
    DISP             P Near      0014        CODE Length=000D Public
    MAIN             P Far       0000        CODE Length=0014 Public
Symbols:
    N a m e          Type        Value       Attr
    0 Warnings
    0 Errors
```

KORAK 6: Izvršite ovaj program slektovanjem RUN u *Run* meniju. Ukucajte broj 6, dobijeni rezultat je = _____. Izvršite program ponovo i ukucajte slovo D. Rezultat = _____.

Upotreba *CODEVIEW*-a

Premda ovi programi ne mogu imati greške, i rade savršeno, sledeći deo ove vežbe prikazuje kako se koristi program *CodeView*. Programu *CodeView* se pristupa iz *Run* menija selektovanjem *CodeView*, ali pre ovog poželjno je selektovati *Option* meni za konfigurisanje *CodeView*-a, vidi sliku 5-4. Konfiguracija će zavisiti od broja linija prikazanih na vašem monitoru, ali pouzdano možete selektovati 50 linija.

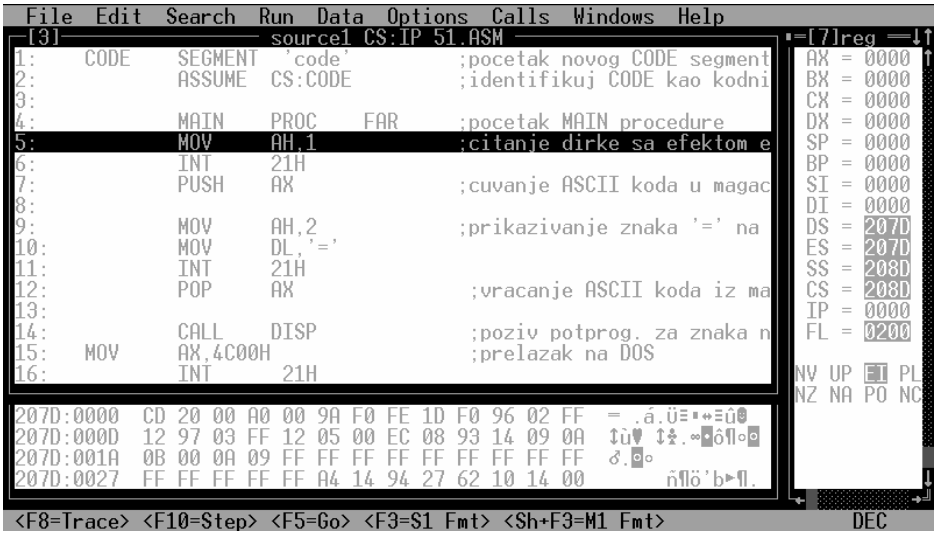


Slika 5-4 Meni *CodeView Options*

KORAK 7: Sada, pošto je konfigurisan *CodeView*, selektujte *CodeView* sa *Debug* ispod *Run* menija. Na slici 5-5 prikazan je *CodeView* gde se vidi simbolička forma programa iz Primera 5-1.

Program *CodeView* omogućava da se program izvršava korak po korak ili po sekvencama instrukcija dok se za to vreme dinamički prikazuje sadržaj registara i memorijskih lokacija na prozoru za nihov prikaz. *CodeView* sadrži sledeće važne osobine koje se upotrebljavaju za debagiranje programa:

1. Za izvršavanje jedog programskog koraka u programu sa tekuće lokacije (osvetljeni red) pritisnite taster F10.
2. Za izvršavanje čitavog programa sa tekuće lokacije pritisnite taster F5.
3. Za povratak u video displej pritisnite taster F4. Uočimo da se višestrukim pritiskanjem F4 prikazuje ekran *CodeView* i video displej ili izlazni ekran.
4. *Run* meni omogućava da se program restartuje sa ili bez komandnih linijskih parametara.



Slika 5-5 Na video displeju *CodeView* pokazuje okvire prozora za *source*, *register* i *memory*

KORAK 8: Pošto se nalazite *CodeView*-u i imate prikazan program na video displeju, pritisnite taster F5 da bi ste ga izvršili. Naglasimo da će vam možda biti potreban taster F4 da bi se promenio sadrža izlaznog ekrana radi unošenja ASCII znaka sa tastature. Odaberite OK i restartuje program ispod *Run* menija. Ovog puta izvršite program korak po korak koristeći taster F10. Ukoliko se sadržaj registara ne prikazuje, odaberite *Windows* meni i selektujte registre. Ovim je omogućeno da se prikažu sadržaji svih registra u svakom pojedinačnom koraku u toku izvršenja programa.

Pitanja

1. Šta se ukucava u DOS komandnoj liniji da bi se pristupilo PWB-u?
2. Kako se novi program unosi koristeći PWB?
3. Koji se *Option* meni koristi pri konfigurisanju PWB-a?
4. Koja je svrha *Set Debug Options* menija?
5. Koju ekstenziju moraju imati svi fajlovi koji se trebaju asemblirati?
6. Šta opcija *Build* predstavlja u *Project* meniju?
7. Kako se pristupa *CodeView*-u iz PWB-a ?
8. Kako su registri prikazuju u *CodeView*-u?
9. Kako se program izvršava u *CodeView*-u?

10. Kako se program izvršava korak po korak u *CodeView*-u?
11. Pritiskom kojeg funkcijskog tastera se selektuje izlazni video ekran u *CodeView*?
12. Koji tip informacija se može prikazati kod *CodeView*-a?

LABORATORIJSKA VEŽBA 6

Uvod u programiranje na asemblerskom jeziku 80x86

Uvod

Ova Laboratorijska vežba nas uvodi u programiranje na asemblerskom jeziku. Tokom ove laboratorijske vežbe ukazaćemo na osnovne U/I komponente (tastatura, monitor) i detaljno objasniti korišćenje DOS INT 21H funkcijskih poziva, kao i BIOS INT poziva koji se koriste za pristup tastaturi i za video prikaz.

Predmet rada

1. Pokazati kako izgleda pristup tastaturi koristeći se:
a) DOS funkcijskim pozivom (INT 21H)
b) BIOS funkcijskim pozivom (INT 16H).
2. Pokazati kako se pristupa video displeju u tekstualnom načinu rada:
a) DOS funkcijskim pozivom (INT 21H)
b) BIOS funkcijskim pozivom (INT 10H).
3. Ukazati na razliku izmedju funkcija koje se odnose na rad sa tastaturom u režimu rada sa ehom i bez efekta ehoa .
4. Pokazati kako se vrši pristup displeju u tekstualnom načinu rada preko memorije.

Postupak rada

Prvi deo ove vežbe pokazuje način na koji se koriste DOS INT 21H funkcijski pozivi: 02H, 06H i 09H radi prikazivanja informacije na video displeju programiranjem na asemblerskom jeziku.

DOS funkcije 02H i 06H za rad sa video displejom

Obe DOS funkcije 02H i 06H imaju gotovo identičan efekat u pogledu prikazivanja ASCII kodiranih podataka na video displeju. Jedina razlika je da se izvršavanje funkcije 02H može prekinuti sa *control-break*, dok se funkcija 06H ne može prekinuti u izvršenju. Da bi prikazali jedan ASCII kodirani znak na video displeju koji će se locirati na tekućoj poziciji kursora, neophodno je uraditi sledeće:

1. Napuniti registar AH na vrednost 02H ili 06H koristaći se jednom od sledećih instrukcija MOV AH,02H ili MOV AH,06H.

2. Napuniti registar DL ASCII kôdiranim znakom koji se treba prikazati. Na primer, da bi prikazali broj 2 koristi se instrukcija MOV DL,32H (32H je ASCII kôd za broj 2), ili MOV DL,'2'
3. Izvršiti DOS funkcijski poziv. Nakon MOV instrukcije sledi INT 21H.

Dakle, programska sekvenca ima oblik:

```
MOV   AH,XXH      ;XX je broj funkcijskog poziva poziva npr. 02H,06H itd.
MOV   DL,'Y'      ;Y je znak koji treba prikazati npr. 'a','B', '4' itd.
INT   21H         ;aktiviranje izvršenja funkcijskog poziva
```

Kratak program iz Primera 6-1 ilustruje prikazivanje znakova ABC na video displeju. Ovo je kompletan program koji se može uneti korišćenjem PWB-a, nakon toga treba ga asemblirati i izvršiti. U ovom primeru koristi se potpuna definicija segmenata koja je detaljno pbjašnjena u Laboratorijskoj vežbi 7. (Segment podataka se ovde nalazi u okviru kodnog segmenta).

Primer 6-1

```
CODE  SEGMENT 'code'        ;identifikuje početak kodnog segmenta
ASSUME CS: CODE       ;dodeljuje ime MAIN kôdnom segmentu
MAIN  PROC  FAR
MOV   AH,02H
MOV   DL,'A'          ;prikaz slova A
INT   21H
MOV   DL,'B'          ;prikaz slova B
INT   21H
MOV   DL,'C'          ;prikaz slova C
INT   21H
MOV   AX,4C00H        ;prelazak na DOS
INT   21H
MAIN  ENDP                  ;kraj procedure MAIN
CODE  ENDS                  ;kraj programskog segmenta
      END   MAIN
```

Uočimo u ovom primeru da je u registar AH smeštena samo jedanput vrednost 02H na početku programa. Naredni pozivi za prikazivanje ASCII znaka na video displeju ne moraju ponovo da pune vrednost 02H u registar AH, jer DOS čuva jednom smeštenu vrednost u AH. Treba imati u vidu da će se sadržaj registra AL menjati, dok se sadržaj ostalih registara neće menjati.

KORAK 1: Koristeći PWB, unesite program iz Primera 6-1 i proverite da li radi kao što je predvidjeno. Sada, prepravite ovaj program i prikažite reč ELEF na video displeju. Ne zaboravite da je sadržaj registra DL rezervisan od strane DOS INT 21H funkcijskog poziva (u registru DL se mora smestiti tekući znak za prikaz).

Prikaz niza znakova korišćenjem DOS funkcije 09H

U mnogim slučajevima potrebno je istovremeno prikazati više od jednog znaka na video displeju. Za ovo se koristi funkcija 09H. Da bi prikazali niz znakova na ekranu neophodno je da kraj teksta bude markiran sa dolarskim znakom $, korake koje treba preduzeti za ovo su:

1. Napuniti registar AH vrednošću 09H instrukcijom MOV AH,09H.
2. Smestiti adresu segmenta i ofseta niza znakova u registarski par DS:DX. Podrazumevamo da DS adresira segment podataka, tada se ofset adresa niza NIZ1 definiše sledećom instrukcijom: MOV DX,OFFSET NIZ1.

3. Sledi instrukcija INT 21H, koja će prikazati NIZ1 na video displeju.

Primer 6-2 predstavlja program koji prikazuje niz znakova na video ekranu. Treba uočiti da je u kôdnom segmentu smešten niz znakova kome se pristupa punjenjem registra DS adresom kôdnog segmenta. Takodje treba uočiti kako se direktiva DB (*define byte*) koristi za definisanje obima podataka u memoriji počev od adrese NIZ1. Naglasimo da se ispred i iza niza znakova nalazi *carriage return* (13)(povratak kursora na levu marginu) i *line feed* (10)(pomeranje kursora na novu liniju). Niz znakova se završava znakom dolara kao što je specifirano. Važno je da se ASCII znakovi nalaze izmedju apostrofa (‘ ‘).

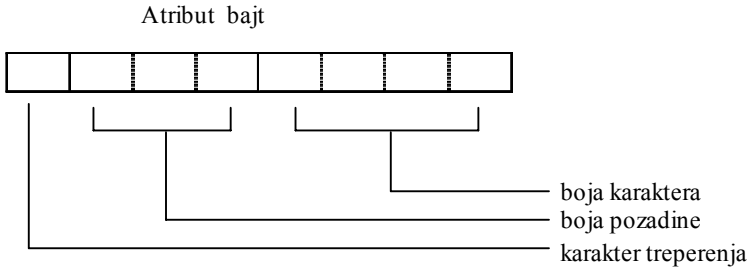
Primer 6-2

```
CODE  SEGMENT 'code'            ;početak programskog segmenta
ASSUME   CS:CODE
POR1  DB 13,10,10,'Ja sam student El Fakulteta',13,10,'$'
      MAIN  PROC  FAR           ;početak procedure MAIN
      MOV   AX,CS               ;inicijalizacija segmenta podataka
      MOV   DS,AX
MOV   AH,9                  ;priprema za prikaz znaka
MOV   DX,OFFSET POR1        ;smeštanje u DX ofset adrese
INT   21H
MOV   AH, 4CH               ;povratak u DOS
INT   21H
MAIN  ENDP                        ;kraj procedure MAIN
CODE  ENDS                        ;kraj programskog segmenta
      END   MAIN
```

KORAK 2: Unesite program iz Primera 6-2 i kreirajte izvršni fajl koristeći PWB. Izvršavanjem ovog programa videčete: ’Ja sam student El Fakulteta’ na ekranu. Modifikujte ovaj program za prikazivanje vašeg imena (prva linija), ulice i broja (druga linija) i grada (sledeća linija). Izvršite ovaj program da bi ste proverili njegov ispravan rad.

Direktan pristup video memoriji za prikaz teksta na video displeju

Mada nije preporučljiv, direktan pristup video memoriji on se koristi u specijalnim slučajevima kada je potrebno obrisati trenutni prikaz video displjeja. Tekst na video memoriji počinje sa lokacije B800:0000 za ubičajene grafičke kartice tipa VGA 25 x 80 kada se koristi u tekstualnom režimu rada. Ako se koristi drugi režim rada, ovaj način možda neće funkcionisati korektno. Karakter na gornjem levom uglu koji se prikazuje u tekstualnom režimu rada predstavljen je na memorijskoj lokaciji B8000:0000 i B800:0001. Naredni karakter je smešten na memorijskoj lokaciji B800:0002 i B800:0003. Svaki znak zauzima dve memorijske lokacije. Prva memorijska lokacija sadrži ASCII kodirani znak koji se prikazuje, a druga atribut bajt koji opisuje boju znaka, boju pozadine i da li znak treperi, blinka. Slika 6-1 prikazuju sadržaj atribut bajta kao i raspoložive boje.



kôd	Boja
0000	Crna
0001	Plava
0010	Zelena
0011	Cijan (maslinasta)
0100	Crvena
0101	Magenta (Purpurna)
0110	Braon
0111	Bela
1000	Siva
1001	Nezasićeno (Pastelno) plava
1010	Nezasićeno zelena
1011	Nezasićeno cijan
1100	Nezasićeno crvena
1101	Nezasićeno purpurna
1110	Nezasićeno Žuta
1111	Jako bela

Slika 6-1 Atribut bajt i vrednosti boja. Naglasimo da boja karaktera može biti bilo koja od 16 raspoloživih boja, dok boja pozadine može biti samo iz opsega boja od 000 do 111.

Primer 6-3 ilustruje smeštanje ASCII kodiranog blanko znaka (20H) u svaku poziciju na ekranu (atribut bajt je: beli znak, crna pozadina i bez treperenja). Uočimo da MOV AX,3E20H istovremeno smešta ASCII kôd blanko znaka u registar AL i atribut bajt u registar AH. Instrukcija STOSW smešta sadržaj registra AX u memorijsku lokaciju koja pripada ekstra segmentu i adresira se preko registarskog para ES:DI i ES:DI+2.

Primer 6-3

```
CODE   SEGMENT 'code'
       ASSUME CS:CODE
MAIN   PROC  FAR            ;početak procedure MAIN
       CLD                  ;selektovanje autoinkrementiranja
MOV    AX,0B800H    ;smeštanje početne adrese video memorije u registar AX
MOV    ES,AX
MOV    DI,0         ;ofset je 0 (ES:DI=0B800:0000)
MOV    CX,25*80     ;25x80 je broj pozicija na ekranu
```

```
MOV    AX,3E20H      ;karakter treperenja i atribut bajta
;cijan pozadina, nezasićeno žuti karakter
REP    STOSW         ;ponovi STOSW 2000 puta
MOV    AX,4C00H      ;povratak u DOS
INT    21H
MAIN   ENDP                  ;kraj procedure MAIN
CODE   ENDS
       END    MAIN
```

KORAK 3: Modifikujte Primer 6-3 tako da smešta broj 3 (ASCII 33H) u svaku lokaciju na ekranu koristeći atribut koji koristi trepereću crveni broj 3 na zelenoj pozadini. Kreirajte izvršni fajl i izvršite ga. Treba imati na umu da će ekran nastaviti da treperi sve dok se neki drugi program ne izvrši ili se CLS (*clear screen*) ne otkuca u DOS komandnu liniju.

Korišćenje BIOS INT 10H za pristup video displeju

BIOS (osnovni U/I sistem) se koristi za prikaz video podataka, ali mnogo češće za upravljanje pozicijom kursora. DOS INT 21H funkcije ne omogućavaju pristup kursoru. Bez mogućnosti upravljanja pozicijom kursora teško je napisati program koji prikazuje neku informaciju na proizvoljnoj lokaciji na video ekranu.

Ova vežba pokazuje upravljanje pozicijom kursora preko INT 10H, ali ne prikazuje i mnoge druge pogodnosti koje ovaj funkcijski poziv nudi. Pozicija kursora je definisana upotrebom INT 10H funkcije AH=02H, prema algoritmu koji sledi:

1. Napuniti vrednošću 02H registar AH da bi se selektovalo kretanje kursora: MOV AH,02H
2. Smestite broj strane 00H u registar BH pomoću MOV BH,0. Uočite da prikazi mogu da čine do 8 strana u nekim načinima rada, ali se, praktično, retko koristi bilo koja strana osim nulte.
3. Smestite u registar DH broj linije.
4. Smestite u registar DL broj kolone.
5. Unesite instrukciju INT 10H posle MOV instrukcija.

KORAK 4: Modifikujte program prikazan u primeru 4-3 tako da ne samo da briše ekran, nego i postavlja kursor na liniju 0 i kolonu 0 (gornja, krajnja levo pozicija na ekranu). Unesite ovaj novi program i proverite da li radi kao DOS-ova CLS komanda.

Čitanje dirki koristeći INT 21H funkcije 01H i 07H

Pošto smo savladali sve detalje koji se odnose na prikaz informacija na video ekranu i na pozicioniranje kursora, sada izučavamo prikaz podataka unetih tastaturom. Ukazaćemo kako se koriste funkcijski pozivi 01H i 06H za čitanje podataka sa tastature.

Funkcija 01H čita pritisnutu dirku i sa efektom ehoa prikazuje znak na ekranu. Funkcija 07H takodje čita dirku, ali je ne prikazuje na ekranu. Koju ćemo funkciju koristiti zavisi od efekta koji želimo postići. U svakom slučaju da bi pročitali jedan znak sa tastature smestite u AH vrednost 01H ili 07H i zatim izvršite INT 21H za čitanje tastature. ASCII kôd znaka se smešta u registar AL. Uočimo da, ako se radi o dodatnim dirkama (funkcijske dirke, *home* itd.), sadržaj registra AL je 00H. Ukoliko je pritisnuta dodatna dirka, sadržaj AL bi trebalo uporediti sa 00H, a zatim se drugom INT 21H instrukcijom dobija kôd dodatnog ASCII znaka.

Primer 6-3 prikazuje kako se računar koristi kao pisaća mašina. Da bi pomerili kursor u novi red morate pritisnuti *ENTER* (*carriage return*) ili *CONTROL+M* za kojim sledi *CONTROL+J* (*line feed*). Ovaj program sadrži izlazak iz petlje kada se pritisne @.

KORAK 5: Unesite program 6-4 i izvršite ga. Kada se pritisne @ program se vraća na DOS. Ukažimo da se dirka @ ovim programom prikazuje na video ekranu. Modifikujte program tako da se znak @ ne prikazuje po pritisku. (Predlog: upotrebite funkciju 07H i 02H.)

Primer 6-4

```
CODE   SEGMENT 'code'      ;početak programskog segmenta
ASSUME CS:CODE
MAIN   PROC  FAR
MOV    AH,1          ;čita dirku sa efektom eha
       INT   21H
       CMP   AL,'@'        ;provera za izlazak iz petlje
       JNE   MAIN          ;ako nije '@' idi na početak procedure MAIN
MOV    AX,4C00H      ;prelazak na DOS
       INT   21H
MAIN   ENDP                ;kraj procedure MAIN
CODE   ENDS                ;kraj kodnog segmenta
       END   MAIN          ;kraj fajla
```

Korišćenje INT 16H za čitanje pritisnute dirke sa tastature

Ponekad je neophodno izbeći upotrebu DOS-a za čitanje tastature. Ovo se postiže korišćenjem BIOS INT 16H poziva. Da bi čitali dirku bez efekta eha, neophodno je ponoviti sledeće korake:

1. Smestite 00H ili 10H u registar AH. Ukoliko je korišćena funkcija 00H, ova funkcija čita bilo koju tastaturu, ali je prvenstveno namenjena tastaturi sa 88 dirki (stara PC mašina tzv. *PC Junior*). Funkcija 10H čita novije tastature koje imaju 101/102 dirke.
2. Izvršite INT 16H da bi pročitali tastaturu. Uočimo da ova kombinacija obavlja identičnu funkciju kao DOS INT 21H funkcijski poziv 07H. Nema eha tj. ne pojavljuje se znak na video ekranu. ASCII kodirani znak se smešta u AL, a *scan* kôd (kôd analize) tastature, u registar AH.

KORAK 6: Modifikujte program kreiran u KORAKU 5 tako da se koristi INT 16H za čitanje dirke umesto DOS funkcije 07H.

Pitanja

1. Koje informacije prikazuju na ekranu monitora DOS INT 21H funkcijski pozivi?
2. Koja je razlika izmedju DOS INT 21H funkcija 02H i 06H?
3. U koji registar je smešten ASCII kôd znaka kod DOS funkcije 06H?
4. Napišite program koji ispisuje na ekranu poruku *Ja se zovem <ime>*.
5. Za direktno prikazivanje podatka u tekstualnom načinu rada memorije pristupljeno je lokacijama _____.
6. Koja je svrha atribut bajta kada se direktno obraćamo video memoriji u tekstualnom načinu rada memorije?
7. Napišite sekvencu instrukcija koja pomera kursor na poziciju: linija 6 i kolona 5.
8. Koje DOS funkcije se koriste za čitanje tastature?
9. Koji se BIOS prekid (*interrupt*) koristi za čitanje tastature?

LABORATORIJSKA VEŽBA 7

Korišćenje modela i definicija potpunih segmenata za asemblerski jezik

Uvod

Cilj ove vežbe je da ukaže na razlike koje postoje kod kreiranja preograma na asemblerskom jeziku mikropreocsersora 80x86 za slučaj kada se koristi forma potpune definicije segmenata (*full segment definition*) i forma pojednostavljene definiocije segmenata (*simplifierd segment definition*). Uobičajena praksa kod programaera koji koriste Microsoftov MASM asembler je da se pojednostavljena definicija segmenta alternativno naziva i *model*. Obično modele koriste manje iskusni programeri na asemblerskom jeziku dok potpunu definiciju segmenta oni koji su vičniji. Obe varijante kreiranja programa sagledaćemo kroz ovu vežbu.
Programi napisani na asemblerskom programu mikroprocesora 80x86 koriste potpunu definiciju segmenata tada je uobičajeno da registar CS pokazuje na baznu adresu kôdnog (programskog) segmenta, a registri DS, ES i SS na bazne adrese segmenata podataka, ekstra segmenta, i segmenta magacina, respektivno.
Svaka linija u programu na asemblerskom jeziku mora da pripada jednom od ovih segmenata. Zadnjih godina novi metod korišćenja segmentata je uveden od strane Microsoft-ovog asemblera MASM 5.0 i naviše i Borland-ovog asemblera TASM 1.0 i naviše, kao i od strane drugih kompatibilnih asemblera. Ovaj metod se naziva format sa pojednostavljenom definicijom, a njegova suština se sastoji u tome što koristi tri jednostavne direktive: .CODE, .DATA i .STACK, koje odgovaraju registrima CS, DS i SS respektivno. Koristeći ove direktive korišćenje direktiva SEGMENT i ENDS se čini nepotrebnim. U suštini pojednostavljena definicija segmenta je lakša za razumevanje i korišćenje, posebno za početnike.
Pre nego što programer odluči da koristi odgovarajući model tj. pojednostavljenu definiciju segmenata neophodno je da donese odluku o izaboru memorijskog modela za program koji on kreira. Uglavnom se koriste sledeći memorijski modeli:

Model	Opis
TINY	Program i podaci su smešteni u jedan segment, kôdni segment veličine 64k. Programi u modelu TINY su uglavnom pisani u formatu fajla .COM, što znači da se program mora početi od memorijske lokacije 0100H. Ovaj model se uglavnom koristi kod programa koji koristi jedan segment.
SMALL	Svi podaci su smešteni u jedan segment veličine 64k bajta a svi programi su smešteni u drugi segment veličine 64k bajta. Ovo dozvoljava da se pristup svim programima vrši sa kratkim skokovima (*near jump*) i pozivima (*near call*).

MEDIUM	Podaci su smešteni u jedan segment veličine 64k bajta, program se smešta u više od jednog segmenta svaki obima 64k bajta.
COMPACT	Podaci su smešteni u više od jednog segmenta 64k bajta, program se smešta u više od jednog segmenta svaki obima 64k bajta. Suprotano od modela MEDIUM
LARGE	Podaci su smešteni u više od jednog segmenta 64k bajta,. program se smešta u više od jednog segmenta svaki obima 64k bajta. Veći broj segmenata za program i podatke
HUGE	Isto kao i za LARGE, ali se dozvoljava da segment podataka bude veći od 64K bajta.
FLAT	Ne postoje segmenti, za program i podatke se koriste 32-bitne adrese. Važi samo za *Protected mode*.

Tabela 7-1. Memorijski modeli koje prepoznaje Microsoftov MASM program.

Primer potpune i pojednostavljene definicije segmenta

```
;Potpuna definicija segmenta
;----------segment magacina----------
SMAG  SEGMENT
      DB 64 DUP (?)
SMAG  ENDS
;---------segment podataka------------
SPOD      SEGMENT
POD_1     DW 1234h
POD_2     DW 5678h
REZ       DW ?
SPOD      ENDS
;---------programski segment----------
SPROG     SEGMENT
SABERI    PROC     FAR
          ASSUME   CS:SPOD
          MOV   AX,SPOD
          MOV   DS,AX
- - -
SABERI    ENDP
SPROG     ENDS
          END   SABERI
```

```
;Pojednostavljena def. segmenta
          .MODEL    SMAL
          .STACK    64
;
;
;-----------------------------------------------
          .DATA
POD_1     DW      1234h
POD_2     DW      5678h
REZ       DW      ?
;
;-----------------------------------------------
          .CODE
SABERI:   MOV   AX,@DATA
          MOV   DS,AX
          . . .
          . . .
          . . .
          . . .
          . . .
          END   SABERI
```

Predmet rada

1. Korišćenje modela kod pisanja asemblerskih programa.
2. Korišćenje potpunih segmentnih definicija kod pisanja asembleskih programa.

Postupak rada

U prvom delu Laboratorijske vežbe manipulisaćemo sa korišćenjem modela. Model je od strane firme Microsoft kreiran za rad programera početnika. Na prvi pogled model može izgledati kao dobra ideja ali u suštini njegovo korišćenje može prouzrokovati veliki broj problema kod kreiranja složenijih asemblerskih programa. U slučaju kada je model odabran kao način rada za kreiranje programa korisnik treba da pazi i da bude svestan trebamo paziti i biti svesni da dotična verzija programa možda neće korektno raditi kada se izvršava na drugim verzijama asemblera.

Model

Kada se koristi model prva iskaz na semblerskom programu mora biti .MODEL. ovaj iskaz prati veličina modela (kakva je TINY, SMALL, MEDIUM i dr.) adabrana od strane tog asemblerskog programa.

Tablea 7-2. Standardne .MODEL naredbe

Model	Directiva	Ime	Poravnanje	Komb.	Klasa	Group-a
TINY	.CODE	_TEXT	Word	PUBLIC	'CODE'	DGROUP
	.FARDATA	FAR_DATA	Para	Private	'FAR_DATA'	
	.FARDATA?	FAR_BSS	Para	Private	'FAR_BSS'	
	.DATA	_DATA	Word	PUBLIC	'DATA'	DGROUP
	.CONST	CONST	Word	PUBLIC	'CONST'	DGROUP
	.DATA?	_BSS	Word	Public	'BSS'	DGROUP
SMALL	.CODE	_TEXT	Word	PUBLIC	'CODE'	
	.FARDATA	FAR_DATA	Para	Private	'FAR_DATA'	
	.FARDATA?	FAR_BSS	Para	Private	'FAR_BSS'	
	.DATA	_DATA	Word	PUBLIC	'DATA'	DGROUP
	.CONST	CONST	Word	PUBLIC	'CONST'	DGROUP
	.DATA?	_BSS	Word	PUBLIC	'BSS'	DGROUP
	.STACK	STACK	Para	STACK	'STACK'	DOROUP
MEDIUM	.CODE	name_TEXT	Word	PUBLIC	'CODE'	
	.FARDATA	FAR_DATA	Para	Private	'FAR_DATA'	
	.FARDATA?	FAR_BSS	Para	Private	'FAR_BSS'	
	.DATA	_DATA	Word	PUBLIC	'DATA'	DGROUP
	.CONST	CONST	Word	PUBLIC	'CONST'	DGROUP
	.DATA?	_BSS	Word	PUBLIC	'BSS'	DGROUP
	.STACK	STACK	Para	STACK	'STACK'	DGROUP
COMPACT	.CODE	_TEXT	Word	PUBLIC	'CODE'	
	.FARDATA	FAR_DATA	Para	Private	'FAR_DATA'	
	.FARDATA?	FAR_BSS	Para	Private	'FAR_BSS'	
	.DATA	_DATA	Word	PUBLIC	'DATA'	DGROUP
	.CONST	CONST	Word	PUBLIC	'CONST'	DGROUP
	.DATA?	_BSS	Word	PUBLIC	'BSS'	DGROUP
	.STACK	STACK	Para	STACK	'STACK'	DGROUP
LARGE	.CODE	naine_TEXT	Word	PUBLIC	'CODE'	
OR	.FARDATA	FAR_DATA	Para	Private	'FAR_DATA'	
HUGE	.FARDATA?	FAR_BSS	Para	Private	'FAR_BSS'	
	.DATA	_DATA	Word	PUBLIC	'DATA'	DGROUP
	.CONST	CONST	Word	PUBLIC	'CONST'	DGROUP
	.DATA?	_BSS	Word	PUBLIC	'BSS'	DGROUP
	.STACK	STACK	Para	STACK	'STACK'	DGROUP
FLAT	.CODE	_TEXT	Dword	PUBLIC	'CODE'	
		.FARDATA	_DATA	Dword	PUBLIC	'DATA'
		.FARDATA?	_BSS	Dword	PUBLIC	'BSS'
		.DATA	_DATA	Dword	PUBLIC	'DATA'
		.CONST	.CONST	Dword	PUBLIC	'CONST'
		.DATA?	_BSS	Dword	PUBLIC	'BSS'

		.STACK	STACK	Dword	PUBLIC	'STACK'

Primer 7-1 prikazuje jedan tipičan program koji koristi TINY model radi očitavanja pritisnute dirke sa tastature i čuvanje ASCII kôda u memorijsku oblast koja se naziva ARRAY. Program završava kada se na tastaturi pritisne dirka *enter* (*cariage return*). Primetimo da program koristi .CODE direktivu da bi ukazao na početak programa a .DATA direktivu da bi ukazao na početak segmenta podataka. U programu se korisi .STARTUP direktiva kako bi se ukazalo na početnu adresu programa a takodje i napunili ostali registri na odgovarajuće vrednosti koje su u skladu sa izabranim modelom. Direktiva .EXIT 0 se koristi za povratak na DOS sa povratnom vrednošću ERRORLEVEL čija je vrednost 0. Primetimo da zadnja komanda END ne prati adresu jer je direktiva .STARTUP ta koja postavlja postavlja početnu adresu. U konkretnom slučaju .LISTALL direktiva se koristi za listanje svih kodnih linija generisanih od početka .STARTUP do .EXIT. Sobzirom da program koristi TINY model (jedinstvani segment), podaci postoje u kôdnom segmentu i pristupa im se pomoću CS: segmentno dopisnog prefiksa.

Primer 7-1

```
.MODEL TINY                          ;odabran model TINY
.STACK
.CODE                                ;početak programa
.LISTALL                             ;listanje celog programa
      .STARTUP                       ;umetanje STARTUP dijaloga
      MOV   DI,OFFSET ARRAY          ;adresa početka memorijske oblasi ARRAY
REPS: MOV   AH,1                     ;čitanje podatka sa tastature sa efektom ehoa
      INT   21H
      MOV   CS:[DI],AL       ;pamćenje kôda pritisnute dirke
      INC   DI               ;adresa narednog elementa memorijske oblasti
      CMP   AL,13            ;testiranje da li je pritisnuta dirka enter
      JNE   REPS             ;ponoviti telo petlje dok se ne pritisne dirka enter
      .EXIT 0                ;prelazak na DOS
ARRAY DB 256 DUP (?)         ;oblast memorije koja je rezervisana za ARRAY
      END
```

Primer 7-2. prikazuje isti program kao i Primera 7-1, ali umesto .ASM fajla generisan je .LST fajl kod koga se mogu uočiti efekti direktiva. Zvezdica na početku programa ukazuje na kôd koji asembler dodaje po automatizmu (odredjuje da program počne sa memorijske lokacije 0100H) jer je u konkretnom slučaju korišćen model MODEL TINY.

Primer 7-2

```
                  .MODEL TINY          ;izabran model TINY
0000              .CODE                ;početak programa
                  .LISTALL             ;prikaz svih kodnih linija
                  .STARTUP             ;umetanje STARTUP dijaloga
0100              *@Startup:
0100 BF 0114 R        MOV  DI,OFFSET ARRAY     ;adresa ARRAY
0103 B4 01      REPS: MOV  AH,1                ;čitanje podatka sa
0105 CD 21            INT  21H                 ;tastature sa efektom ehoa
0107 2E: 88 05        MOV  CS:[DI],AL          ;čuvanje koda dirke
O10A 47               INC  DI                  ;adresiranje sledećeg
                                               ;memorijskog elementa
```

```
O10B 3C 0D          CMP  AL,13      ;testiranje da li je pritisnuta enter dirka
O10D 75 F4          JNE  REPS       ;ponoviti telo petlje dok se ne pritisne enter
                  .EXIT 0           ;izlaz u DOS
O10F B8 4C00 *      mov  ax,04C00h
0112 CD 21   *      int  021h

0114 0100 [    ARRAY  DB 256 DUP (?)  ;oblast memorije gde se čuva ARRAY
       00
             ]
                END
```

U primeru 7-3 prikazan je listing fajl programa iz primera 7-1 za slučaj kada se koristi MODEL SMAL. Uočimo razliku izmedju Modela TINY i SMALL u iskazu .STARTUP. Kod MODEL-a SMAL .STARTUP puni registar DS na adresu segmenta podataka ali takodje rezerviše prostor na vrhu segmenta podataka za potrebe magacina.

Primer 7-3

```
          .MODEL SMALL                  ;odabiranje SMALL modela
0000      .CODE                         ;početak programa
          .LISTALL                      ;prikaz svih kodnih linija
          .STARTUP                      ;umetanje STARTUP dijaloga
0000      *@Startup:

0000 BA ---- R      *     mov   dx,DGROUP
0003 8E DA          *     mov   ds,dx
0005 8C D3          *     mov   bx,ss
0007 2B DA          *     sub   bx,dx
0009 DI E3          *     shl   bx,001h
000B DI E3          *     shl   bx,001h
000D DI E3          *     shl   bx,001h
000F DI E3          *     shl   bx,001h
0011 FA             *     cli
0012 8E D2                mov   ss,dx
0014 03 E3                add   sp,bx
0016 FB                   sti

0017 BF 0000 R        MOV  DI,OFFSET ARRAY     ;adresa ARRAY
001A B4 01    REPS:   MOV  AH,1                ;čitanje podatka sa
001C CD 21            INT  21H                 ;tastature sa efektom ehoa
001E 88 05            MOV  [DI],AL             ;čuvanje kôda dirke
0020 47               INC  DI                  ;adresiranje sledećeg
                                                   ;memorijskog elementa
0021 3C OD            CMP  AL, 13      ;testiranje da li je pritisnuta enter dirka
0023 75 F5            JNE  REPS        ;ponoviti telo petlje dok se ne pritisne enter
                      .EXIT  0         ;izlaz u DOS
0025 B8 4C00          mov ax,04C00h
0028 CD 21            int 021h
```

```
0000                DATA
0000 0100[          ARRAY DB 256 DUP (?) ;oblast memorije gde se čuva ARRAY
00
    ]
                    END
```

Drugi modeli kao što je MEDIUM koriste .STACK direktivu praćenu koju prati broj bajtova koji se rezervišu za potrebe magacina. Ako se .STAC direktivom ni jedan bajt ne dodeli za potrebe magacina tada se inicijalno za potrebe magacina dodeljuje prostor od 1,024 bajt lokacija.

KORAK1: Kreirati program koji prikazuje vaše ime na video dipleju koristeći TINY model.

Potpuno definisani segmenti

Potpuno definisani segmenti prilikom kreiranja zahtevaju više unošenje teksta sobzirom da asembler ne koristi .STARTUP direktivu, a takodje je i veći stepen upravljanja nad procesom kreiranja programa prepušten je programeru. Kod potpuno definisanih segmenata direktiva SEGMENT uvek se nalazi na početku segmenta a direktiva ENDS na kraju segmenta. Primer 7-4 prikazuje program koji odgovara programu iz Primera 7-1 samo sada kreiran korišćenjem potpune definicije segmenata. Naglasimo da direktiva ORG 100H definiše da početak programa bude na 100H. Takodje primetimo da je obim programa skoro isti. Ako planirate da unesete i izvršavate ovaj program proverite da li ste definisali karakterističan šablon za asembliranje .COM fajla (početna adresa je 100H).

Primer 7-4

```
0000    CODE   SEGMENT 'code'     ;označavanje početka kôdnog segmenta
        ASSUME CS:CODE            ;usvajamo da CS adresira kôdni segment CODE
        ORG    100H               ;definisanje da početak programa bude na
                                  ;adresi 100H
0100 BF 01 14 R  START:MOV DI,OFFSET ARRAY    ;adresa ARRAY
0103 B4 01       REPS: MOV AH,1               ;čitanje podatka sa tastature
                                              ;uz korišćenje efekta eho
0105 CD 21       INT   21H
0107 2E: 88 05   MOV   CS:[DI],AL    ;pamćenje kôda pritisnute dirke
010A 47          INC   DI            ;adresiranje narednog memorijskog elementa
010B 3C OD       CMP   AL,13         ;testiranje da li je pritisnuta enter dirka
010D 75 F4       JNE   REPS          ;ponoviti telo petlje dok se ne pritisne enter
01OF BS 4C00     MOV   AX,4C00H      ;izlaz u DOS
0112 CD 21       INT   21H
0114 0100 [      ARRAY DB 256 DUP(?) ;oblast memorije gde se čuva ARRAY
    00
    ]
0214       CODE  ENDS                ;označavanje kraja kôdnog segmenta
           END   START               ;označavanje kraja programa
```

Kad god se koriste potpuno definisani segmenti, preporučuje se da kreirate glavni program kao proceduru tipa FAR. Ova mogućnost dozvoljava da se kreirani program poziva u okviru drugih programa ako za to postoji potreba. Takodje ne sme se ispustiti iz vida da je potrebno ‘code’ pridružiti direktivi SEGMENT što obezbedjuje definisanje kôdnog segmenta. Ovo omogućava debageru, *CodeView,* da simbloički prikaže i debagira program.

Primer 7-5 prikazuje jedan kratak program koji koristi tehniku potpuno definisanih segmenata da bi definisao magacin i kôdni segment. Kod većine programa ovakva definicija segmenta se veoma često zahteva. Naglasimo takodje da se i drugi segmenti mogu definisati i koristiti u toku kreiranja programa umetanjem ASSUME direktiva na svakom mestu gde se segment menja. ASSUME direktiva uzrokuje da asembler automatski umeće segmentno dopisni prefiks kada je to potrebno a umetanje se obavlja indirektnim adresiranjem pomoću pokazivača ili indeksnog registra. Program pirikazan u Primeru 7-5 čita vreme DOS-a i prikazije ga kao XX:YY gde je XX broj sati (0-23) a YY broj minuta (0-59).
Primetimo da procedura DISP konvertuje broj sati ili munuta u BCD broj korišćenjem AAM instrukcije. AAM deli AX registar sa 10 i smešta rezultat u AH i AL. AH sadrzi količnik, a AL ostatak. Ako je vrednost AX=000EH, rezultat će biti AH=01H i AL=04H. Zatim DISP procedura dodaje 3030H nad sadržajem registra AX kako bi da bi konvertovala BCD cifre u ASCII kod radi prikazivanja na displeju.

Primer 7-5

```
STACK SEGMENT  STACK                    ;početak segmenta magacina
      DW 256 DUP (?)                    ;rezervisanje 256 reči za magacin
STACK ENDS                              ;kraj segmenta magacina
CODE  SEGMENT  'code'                   ;počatk programskog segmenta
      ASSUME CS:CODE,SS:STACK           ;usvajamo da CS adresira kôdni
                                ;segment CODE
      MAIN  PROC FAR                    ;početak procedure MAIN
            MOV   AH,2CH                ;dobijane vremena iz DOS-a
            INT   21H
            MOV   AL,CH                 ;dobijanje sati
            CALL  DISP                  ;pribavljanje informacije o vremenu iz
                                ;DOS-a
            MOV   AH,2                  ;prikazivanje dvotačke
            MOV   DL,':'
            INT   21H
            MOV   AL,CL                 ;izbavljanje broja minuta
            CALL  DISP                  ;prikazivanje broja minuta
            MOV   AX,4C00H              ;prelaz na DOS
            INT   21H
      MAIN  ENDP                        ;kraj procedure MAIN
      DISP  PROC  NEAR                  ;počtak procedure DISP
            XOR   AH,0                  ;AH =0
            AAM                         ;konvertovanje AX u BCD
            ADD   AX,3030H              ;konvertovanje AX u ASCII
            PUSH  AX
            MOV   DL,AH                 ;prikazivanje cifre
            MOV   AH,2
            INT   21H
            POP   DX
            INT   21H                   ;prikazivanje cifre
```

```
        RET
DISP    ENDP                    ;kraj procedure DISP
CODE    ENDS                    ;kraja kodnog segmenta
        END     MAIN            ;kraj programa MAIN
```

KORAK 2: Prepisati program iz Primera 7-5 tako da isti koristi za prikazivanje broja sati u dvanestočasovnom formatu umesto dvadesetčetvoročasovnom formatu. Asemblirati ovaj program i pokreniti ga.

KORAK 3: Prepisati program iz Primera 7-5 tako da prikazuje broj sati, minuta i sekundi (HH:MM:SS) na video displeju. Za više detalja pogledajte DOS INT 21H, funkcija 2CH. Asemblirati ovaj program i pokreniti ga.

Pitanja

1. Koja je razlika izmedju TINY i SMALL programskog modela?
2. Koja je svrha .EXIT direktive kada se kod programiranja koriste modeli?
3. Kada se koristi sa programskim modelima koji se zadatak dodeljuje direktivi .STARTUP?
4. Kada se koristi iskaz .LISTALL koja se informacija generiše od strane asemblera u listing fajlu (.LST)?
5. Koji tip direktive se koristi da definiše oblast magacina ili segment magacina?
6. Kada se koristi potpuna definicija segmenta koja se oblast definife iskazom SEGMENT?
7. Koja asemblerska direktiva označava kraj segmenta kada se koristi potpuna definicija segmenta?
8. Kojom asemblerskom direktivom se označava procedura?
9. Koja je svrha ENDP iskaza?
10. Kod potpune definicije segmenata iskaz END prati odgovarajuća labela. Koja se informacija identifikuje ovom labelom?
11. Koja je svrha ASSUME direktive u asemblerskom programu kada se koristi potpuna definicija segmenta?
12. Koja je svrha reči ‘code’ koja sledi nakon SEGMENT iskaza kod asembleskog jezika koji koristi potpunu definicijeu segmenata?

LABORATORIJSKA VEŽBA 8

Korišćenje dopunske DOS INT 21H naredbe

Uvod

Prethodne Laboratorijske vežbe su ilustrovale nekoliko DOS INT 21H funkcijskih poziva ili komandi. Ova Laboratorijska vežba ilustruje neke dodatne DOS INT 21H komande. U ovoj Laboratorijskoj vežbi ukazaćemo kako se DOS INT 21H funkcijski poziv koristi za čitanje podataka komandne linije sa tastature, a takodje ilustrirati i kako se za potrebe programoma pristupa parametaru koji pripada komandnol liniji DOS-a.

Predmet rada

1. Koristiti DOS INT 21H, funkcija 0AH, da bi se pročitala informaciona linija sa tastature i sačuvala u memoriji.
2. Koristiti DOS INT 21H, funkcija 30H, da bi se dobio broj DOS verzije i prikazati ga na displeju.
3. Koristiti PSP (*Program segment Prefix*) da bi se pročitali i obradili parametar dobavljen iz DOS komandne linije.

Postupak rada

Prvi deo Laboratorijske vežbe odnosi se na čitanje niza karaktera iz DOS komandne linije korišćenjem DOS INT 21H, funkcija 0AH. Ova tehnika se često koristi jer veliki broj aplikacija zahteva unošenje niza znakova podataka preko tastature.

Čitanje nizova karaktera (znakova) koji su uneti preko tastature

Kod odredjenih aplikacijama često treba da se pročita niz karaktera sa tastature. Korišće DOS INT 21H, funkcija 0AH, obezbedjuje jednostavni metod pristupa nizu karaktera kod prihvatanjea niza karaktera sa tastature kao i njihovo memorisanje. Takodje ova funkcija dozvoljava korišćenje "←*backspace* " dirke sa tastature u slučaju kada se briše pogrešno otkucan podatak. Način korišćenja DOS INT 21H funkcija 0AH je sledeći:

1. Postaviti 0H u AH registar, instrukcijom MOV AH, 0AH.
2. Napuniti par registra DS:DX vrednošću adrese bafera tastature koju čini bazna adresa segmenta (čuva se u DS) i ofset adresa (čuva se u DX). Bafer tastature sadrži memorijsku oblast u kojoj se čuvaju podaci uneti preko tastature, bajt koji označava maksimalni broj karaktera koji se može smestiti u bafer i bajt koji ukazuje na aktuelni stvarni broj karaktera koji je memorisan u baferu.
3. Izvršiti INT 21H instrukciju da bi se pročitao niz karaktera sa tastature. Povratak se dešava kada je otkucan enter, ili se otkuca željeni (zadati) broj karaktera.

U Primeru 8-1 prikazan je kratak program koji se koristi za čitanje jedne linije podataka sa tastature i memorisanje podatka u memoriji koja se naziva BUF. Prvi bajt bafera puni se na maksimalni broj karaktera koji se može smestiti u jednoj liniji i drugi bajt korišćenjem INT 21H, funkcija 0AH, kada se pritisne dirka enter puni se na aktuelni broj karaktera unetih u toj lniji. Naglasimo da je enter dirka smeštena u memoriji kao kôd 0DH.

Primer 8-1

```
CODE   SEGMENT   'code'          ;početak segmenta CODE
       ASSUME CS:CODE            ;usvajamo da CS adresira kôdni segment CODE
       BUF    DB  10             ;broj karaktera koji se čita
```

```
                DB   ?                ;popunjen od strane DOS sa aktuelnim brojem
                DB   10 DUP (?)       ;rezervisanje 10 bajtova za niz
        MAIN    PROC    FAR                     ;početak MAIN procedure
                MOV     AX,CS                   ;DS = CS
                MOV     DS,AX
                MOV     AH,0AH                  ;čitanje i upis niza u BUF
                MOV     DX,OFFSET BUF
                INT     21H
                 MOV    AX,4C00H                ;izlaz u DOS
                INT     21H
        MAIN    ENDP                            ;kraj procedure MAIN
CODE    ENDS                                    ;kraj segmenta CODE
        END     MAIN                            ;kraj programa
```

KORAK 1: Unesite, asemblirajte i izvršite program iz Primera 8-1. Zatim otkucajte ABC a nakon toga jedanput backspace (blanko). Šta će se dešavati kada je pritisnuta backspace dirka?

KORAK 2: Otkucajte enter kako bi izašli iz programa u koraku 1. Ponovo izvršite program iz Primera 8-1. Ovog puta otkucajte 1234567890. Da li ova linija može biti uneta a za slučaj da ne može, šta je moguće otkucati?
U koraku 2 smo naučili da ako je 10 smešteno u prvom bajtu BUF, program će dozvoljavati da se unese 9 karaktera i enter dirka. Ako želimo da vidimo sadržaj BUF-a treba koristiti, *CodeView*, enter dirka (0DH) je smeštena kao poslednji bajt, BUF-a dok povratni broj ne sadrži kod 0DH.

KORAK 3: Kreirati program koji čita jednu liniju sa tastature uz korišćenje INT 21H, funkcija 0AH, a zatim prikazati sadržaj linije korišćenjem INT 21H, funkcija 09H. Prisetimo se Laboratorijske vežbe 6, koja se odnosila na prikazivnjae niza karaktera, i naglasimo da se niz mora završiti sa '$'. Program koga kreirate mora pristupiti kôdu 0DH i promeniti ga u 24H ili u '$' zank.

Korišćenje DOS INT 21H, funkcija 30H

I pored toga što se ne koristi često u programaima, DOS INT 21H, funkcija 30H vraća tekući broj DOS verzije. Da bi se dobavio broj DOS verzije potrebno je:

1. Smestiti 30H u AH pomoću instrukcije MOV AH,30H.
2. Izvršiti INT 21H instrukciju kako bi se dobavio broj DOS verzije. Pri ovome u AH se smešta glavna DOS verzija a u AL sporedna verzija. Na primer broj DOS verzije 6.21 postavlja AH=06H a AL=15H (decimalno 21).

Kao što smo već ukazali u Laboratorijskoj vežbi 6 instrukcija AAM se koristi za konverziju brojeva iz binarnog u BCD foramt. AAM instrukcija deli AX sa 10 sa ciljem da dobije BCD rezultat do 99. Na primer ako AX sadrži 0015H, AAM će postaviti AX=0201H, što odgovara 21 u BCD formatu.

KORAK 4: Kreirati program za čitanje broja DOS verzije korišćenjem funkcije 30H, zatim ga prikazati na video displeju (kao 6.21 za DOS 6.21 ili bilo koju drugu verziju koja stoji instalirana na vašem računaru). Za prikaz ASCII znaka postaviti u AH=2, u DL=ASCII znak, a nakontoga pozvati INT 21H.

Čitanje parametara komandne linije

Komandnoj liniji DOS-a se pristupa preko PSP (*program segment prefix*). PSP čini informacija od 256 bajtova koja se pridružujuje na početku svakog programa. U PSP-u se čuva (vidi sliku 8-1) sadržaj DOS-ove komandne linije kao i druga informacija. PSP sadrži obilje informacija o DOS-ovom okruženju koje je dostupno programu. Kada program počinje da se izvršava segmentna adresa PSP-a se nalazi u DS i ES. Da bi se pristupilo PSP-u, treba smesti željenu ofset adresu u odgovarajući memoriski pokazivač a zatim adresirati segment podataka ili ekstra segment. Informacija koja se čuvaju u PSP je korisna za odredjene aktivnosti. Segmentna adresa nezauzete memorijske oblasti, kao i memorije od kraja programa do lokacije 9FFFFH, se može naći na ofset adresi 02H. Dostupne su takodje adrese DOS-ovih rutina za obradu grešaka kakve su: INT 22H, INT23H i INT24H.
Najvažnije je to što PSP sadrži DOS komandnu liniju. Ako se program zove FROG, a otkuca se FROG /? u DOS komandnoj liniji kako bi se ovaj program izvršio, tada će se parametar /? smestiti na početku na ofset adresi 81H. Naglasimo da komandna linija može biti dužine do 127 bajtova. Takodje naglasimo da se blanko pre /? smešta na ofset lokaciji 81H, a karakter "/" je smešten na ofset lokaciji 82H, dok se "?" je smešta na 83H, a enter (0DH) je na adresi adresi 84H. Na ofset adresi 80H čuva se dužina komandne linije (u našem slučaju je 03H (0DH se ne broji)).

	Ofset
INT 20h	00H
Sledeća slobodna adresa segmenta	02H
Rezervisano	04H
Daleki CALL do DOS	05H
Prekidna adresa (INT22H)	0AH
CTRL-BREAK adresa (INT 23H)	0EH
Adresa kritične greške (INT 24H)	12H
Rezervisano	16H
Adresa okolone segmenta	2CH
Rezervisano	2EH
INT 21H	50H
Dalek RET	52H
Rezervisano	53H
FCB 1 (16 bajta)	5CH
FCB 2 (20 bajta)	6CH
Dužina komandne linije	80H
DOS komandna linija	81H
. . .	
	FFH

Slika 8-1. Pripajanje PSP (program segment prefiks) na početku programa

Ostale dve oblasti takodje sadrže informaciju o komandnoj liniji: FCB 1 i FCB 2. Ove oblasti se nazivaju fajl kontrolni blok i popunjavaju se iz DOS-a sa prve dve reči koje koje se pojavljuju u DOS komandnoj liniji. Svaka reč je ispisana velikim slovom a za slučaj da ne postoje ove reči tada se oblast popunjava sa 12 blanko karaktera, dok se ostatak postavlja na 00H. Naglasimo da se /? se neće pojaviti u FCB 1, dok će DOS jedino smestiti ASCII kodirane alfanumeriške podatke u ove dve oblasti.
Primer 8-2 pokazuje jedan kratak program pomoću koga se vrši pristup u prikaz komandne linije na video displeju. Ovaj primer je kratak iz razloga što se adresa segmenata podataka PSP-a nalazi

na početku programa. Uočimo kako se vrši konverzija komandne linije niza koji završava karakterom enter (0DH) u niz karaktera koji završava sa &.

Primer 8-2

```
CODE  SEGMENT 'code'          ;početk kôdnog segmenta CODE
   ASSUME  CS:CODE            ;usvajamo da CS adresira kôdni segment CODE
   MAIN  PROC  FAR
      MOV   SI,80H            ;dobavljane broja o dužini komandne linije
      MOV   AL,[SI]
      INC   SI                ;adresa komandne linije
      XOR   AH,AH
      ADD   SI,AX             ;adresa 0DH (enter)
      MOV   BYTE PTR [SI],'$'
      MOV   AH,9              ;prikazivanje komandne linije
      MOV   DX,81H
      INT   21H
      MOV   AX,4C00H          ;prelazak na DOS
      INT   21H
MAIN  ENDP
CODE  ENDS                    ;kraj kôdnog segmenta
      END   MAIN              ;kraj programa
```

KORAK 5: Uneti, asemblirati i izvršiti program iz Primera 8-1. Da li ovaj program prikazuje blanko na memorijskoj lokaciji 81H? Ako prikazuje, opišite bilo koju zahtevanu promenu da bi eliminisali blanko iz prikazane linije.
Kao što je pomenuto u ranijim Laboratorijskim vežbama prilikom korišćenja DOS većina komandi prikazuje *help* uvek kada je u komandnol liniji otkucano /?. Karakteri /? se mogu dobaviti iz komandne linije preko komandno linijskog bafera koji počinje na memorijskoj lokaciji 81H. Ostali parametri koji su potrebni programu se mogu uneti na sličan način.

KORAK 6: Kreirati program koji proverava komandnu liniju na nailazak /? i prikazuje primer pomoćnu, *help* liniju (HELP FOR THIS COMANT) jedino ako se /? pojavi na komandnoj liniji nakon imena vašeg programa. Naglasimo da ovim /? karakterima mogu prethoditi znaci tipa blanka (20H) ili znaci tipa tabular (09H) pri čemu trebate ove karaktere treba preskočite kada se proverava sadržaj komandne linije.

Pitanja

1. Opisati kako se čita niz karaktera sa tastature korišćenjem INT 21H, funkcija 0AH.
2. Da li se DOS INT 21H, funkcija 0AH može koristiti za čitanje svih podataka sa tastature?
3. Da li broj koji se vraća od strane DOS INT 21H, funkcija 0AH sadrži kôd enter dirke (0DH)?
4. Kada DOS INT 21H, funkcija 30H čita broj DOS verzije, u kom registru se nalazi glavni broj verzije DOS-a?

5. Opisati kako AAM instrukcija konvertuje binarni u BCD broj, sve dok je broj manji od 1100011?
6. Gde su smešteni parametri iz komandne linije kada su pribavljeni iz DOS komandne linije?
7. Šta je FCB 1 i FCB 2 u PSP-u i koje informacije oni sadrže?
8. Šta se uvek nalazi na ofset adresi 81H u PSP-u?
9. Uobičajeno je da se preskoče vodeći blanko i tab karakteri kada se testira komandna linija. Na koji način se to može ostvariti? (koristite sekvencu instrukcija da bi ste ilustrovali odgovor)
10. Napiasti sekvencu instrukcija koja menja segment i ofset adrese koje se čuvaju u INT 22H, INT 23H i INT 24H lokacija u PSP-u tako da one adresiraju FAR RET instrukciju na ofset adresi 52H u PSP-u.

LABORATORIJSKA VEŽBA 9

Pisanje makro naredbi

Uvod

I pored toga što nisu ključne za programiranje, makro naredbe (sekvence) rasterećuju programera od višestrukog prepisivanja identičnih sekvenci instrukcija i omogućavaju umetanje dodatnih instrukcija u programu. Umetanje instrukcija se takodje ostvaruje korišćenjem procedura i bibliotečkih fajlova. Makroi dozvoljavaju kreiranje sopstvenih pseudojezičkih instrukcijskih sekvenci instrukcija koje se često koriste u programima. Makro je specijalno kreirana instrukcija

po želji korisnika a može da sadrži bilo koji broj instrukcija i da koristi do 32 parametara. Makro postaje nova instrukcija u programu koja može biti smeštema na disku kao fajl sa i umetati je se u svim programima u kojima to programer želi.

Predmet rada

1. Kreirati makro naredbu za programske zadatke opšteg tipa kao što su čitanje kôda pritisnute dirke sa tastature ili prikazivanje podataka na video displeju.
2. Pokazati kako se vrši prenos parametara kod pozivanja makro naredbi.
3. Organizovati makro naredbe u biblioteci makroa nazvan *include* fajl.
4. Iskoristiti lokalne promenjive u makro naredbi da bi se pristupilo lokalnim podacima i adresama grananja (*jump addresses*).

Postupak rada

Makro naredbe se kreiraju pomoću asemblerskih iskaza MACRO i ENDM. Iskaz MACRO ukazuje na početak makro sekvence, a iskaz ENDM označava kraj. MACRO-u se pridružuje ime makroa i veći broj parametara koji se prenose makrou. Ovim parametrima manipulišu instrukcije koje se nalaze izmedju iskaza MACRO i ENDM.
Primer 9-1 pokazuje jednostavnu makro naredbu koja se koristi za prikaz jednog karaktera na video displeju. Primetimo kako se iskazi MACRO i ENDM koriste i kako se prenose parametari makrou. Komentari u okviru makro naredbi koriste dupli simbol tačka – zarez (“;;”) na mestima gde je uobičajeno koristiti samo jedan simbol (“;”). Važno je uočiti da ime makro naredbe mora biti specifično odabrano kako ne bi kod programera stvorilo konfuziju sa korišćenjem nekih od. Prvo slovo labele može početi sa slovom (A-Z) ili bilo kojim od sledeća 4 karaktera: @, $, _, ili ?. U ovom tekstu mi ćemo koristiti _ (podvučeno) da bi smo ukazali da jeto makro. Microsoft preporučuje da se koristi @ kao simbol za narednoa proširenja programa. Znak $ je rezervisan da predstavlja programski brojač kod korišćenje karaktera znak pitanja “?” izgleda nepodesno.

Primer 9-1

```
_DISP MAKRO P1              ;početak makroa _DISP
   MOV   AH,2               ;koristi funkciju 2 da bi prikazao P1
   MOV   DL,P1
   INT   21H
ENDM                        ;kraj makroa _DISP
```

Da bi ukazali na princip _DISP makroa, kreiran je program MAIN koji poziva _DISP makro (vidi Primer 9-2) radi prikazivanja različitih podataka na video displeju. Uočimo razliku u izmedju izvornog iz Primera 9-2 (a) i asembliranog programa iz Primeru 9-2 (b). Svim iskazima koji se umeću korišćenjem makroa prethode brojevi (1, 2, itd.) sa ciljem da ukažu da je iskaz tipa makro iskaz i da bi se ukazalo na nivo ugneždjenja. Kod ove ove Laboratorijske vežbe nema ugnježdenih makroa tako da se jedinica nalazi na levoj strani svake proširene (umetnute) makro instrukcije.

Primer 9-2(a)

```
STACK SEGMENT STACK
      DW  256 DUP(?)
STACK ENDS

CODE   SEGMENT     'code'          ;početak kôdnog segmenta
```

```
      ASSUME  CS:CODE,SS:STACK      ;usvajamo da CS adresira kôdni segment
                                    ;CODE
      _DISP    MACRO  P1           ;početak makroa _DISP
               MOV    AH,2         ;koristi funkciju 2 za prikaz P1
               MOV    DL,P1
               INT    21H
               ENDM                ;kraj makroa _DISP
      MAIN     PROC   FAR          ;početak glavnog programa
               _DISP  'A'          ;prikaz slova A
               _DISP  30H          ;prikaz broja 0
               MOV    BL,'Z'       ;punjenje BL slovom Z
               _DISP  BL           ;prikaz sadržaja BL
               MOV    AX,4C00H     ;povratak u DOS
               INT    21H
      MAIN     ENDP
CODE  ENDS
      END      MAIN
```

Primer 9-2 (b)

```
0000        CODE  SEGMENT 'code'     ;početak kôdnog segmenta
            ASSUME CS:CODE           ;usvajamo da CS adresira kôdni segment
                                     ;CODE
            _DISP MACRO  P1          ;početak makroa _DISP
                  MOV   AH,2         ;koristi funkciju 2 za prikaz P1
                  MOV   DL,P1
                  INT   21H
                  ENDM               ;kraj makroa _DISP
0000        MAIN  PROC  FAR          ;početak programa MAIN
                  _DISP 'A'          ;prikaz slova A
0000 B4 02  1     MOV   AH,2
0002 B2 41  1     MOV   DL,'A'
0004 CD 21  1     INT   21H
                  _DISP 30H              ;prikaz broja 0
0006 B4 02  1     MOV   AH,2
0008 B2 30  1     MOV   Dl,30H
000A CD 21  1     INT   21H
000C B3 5A        MOV   BL,'Z'           ;punjenje BL slovom Z
                  _DISP BL               ;prikaz sadržaja BL
000E B4 02  1     MOV   AH,2
0010 CD 21  1     INT   21H
0014 B8 4C00      MOV   AX,4C00H         ;prelazak na DOS
0017 CD 21        INT   21H
0019        MAIN  ENDP
```

```
0019  CODE ENDS
          END   MAIN
```

KORAK 1: Modifikovati program prikazan u Primeru 9-2 tako da prikazuje vaše ime na video displeju. Koristi _DISP makro naredbi radi prikazivanja svakog slova na video displeju.

Makroi se često pridružuju u bibliotekama makroa koji se nazivaju *include* (pridruženi) fajlovi, a pune se u objektni program za vreme asembliranja. *Include* fajl je ASCII tekstualni fajl kreiran sa PWB-om, ali nije asembliran. U većini slučajeva ime fajla sa ekstenzijom .INC označava fajl koji se pridružuje. Kada program poziva fajl koji se pridružuje (umeće) INCLUDE iskaz treba smestiti na početku programa zajedno sa imenom makro fajla .

Primer 9-3 pokazuje sadržaj fajla nazvan MACS.INC koji sadrži četri makroa koji se koriste prilikom kreiranja programa. Dva makroa su namenjena za prikaz podatka na video displeju, jedan se koristi za čitanje kôda pritisnute dirke sa tastature, a četvrti za prelazak na DOS. Naglasimo da ovaj fajl nije asembliran, tj. on je samo ASCII tekstualni fajl koji sadrži makroe.

Primer 9-3

```
_DISP MACRO   P1            ;prikazivanje P1 korišćenjem makroa
      MOV  AH,2
      MOV  DL,P1
      INT  21H
      ENDM

_KEY  MACRO          ;čitanje podatka sa tastature sa efektom ehoa korišćenjem makroa
      MOV  AH,1
      INT  21H
      ENDM

_STRING MACRO WHERE         ;prikazivanje niza DS:DX korišćenjem makroa
MOV AH,9
MOV DX,OFFSET WHERE
INT   21H
ENDM

_EXIT MACRO                 ;povratak u DOS korišćenjem makroa
MOV  AX,4C00H
INT  21H
ENDM
```

KORAK 2: Memorisati makro prikazan u Primeru 9-3 u fajlu koji se naziva MACS.INC, na bilo kom direktorijumu. Sada kreirati program čiji je listing prikazan u Primeru 9-4, asemblirati ga i izvršiti. U Primeru 9-4 se koristi MASC.INC fajl kako bi se pristupilo video displeju, tastaturi, i peršlo na DOS. Navedte korektan broj diska i put do direktorijuma umetanjem iskaza INCLUDE u Primeru 9-4.

Primer 9-4

```
CODE SEGMENT 'code'          ;početak kôdnog segmenta
     ASSUME CS:CODE          ;usvajamo da CS adresira kôdni segment CODE
     INCLUDE MACS.INC        ;pridruživanje makro fajla
MESI  DB  13,10,10,'Otkucati bilo koju dirku za nastavak: $'
```

```
     MAIN PROC FAR          ;početak procedure MAIN
         MOV AX,CS          ;DS = CS
         MOV DS,AX
         _STRING MESI       ;prikaz MESI
         _KEY               ;čitanje dirke
         _DISP 13           ;prikaz povratka
         _DISP 10           ;prikaz novog reda
          _EXIT
     MAIN ENDP
CODE ENDS
      END MAIN
```

Uočimo da se razumljivost Primera 9-4 povećava zbog korišćenja makroa. Sadržaj registara CS i DS je identičan što znači da se kôdni segment i segment podataka preklapaju. Preklapanje se ostvaruje kopiranjem sadržaja CS u DS.

KORAK 3: Kreirati program koji koristi makro *include* (priključne) fajlove iz Primera 9-3 a koristi se za prikaz ASCII sadržaja memorijskih lokacijaa počev od adrese F800:0000 sve do i uključujući adresu F800:00FF. Treba naglasiti da se ovaj efekat ostvaruje adresiranjem segmenta F800 pomoću DS registrom a nakon toga adresiranjem svake lokacije preko indirektnog registra kakav je SI. Tipičan primer je instrukcija, MOV AL,[SI] ili kada se makro _DISP [SI] koristi direktno. Takodje naglasimo da _DISP [SI] adresira bajtovsko organizovanu memoriju zbog načina na koji je napisan _DISP makro. Prikazati nekoliko kombinacija *carige return* (13) / *line feed* (10) pre nego što se prikažu sadržaji memorije koristeći makro STRING.

Korišćenje lokalnih promenjivih kod makroa

Kod velikog broja makroa grananje može biti izvedeno unutar makroa. Ovo zahteva korišćenje lokalne promenjive, koja se definiše naredbom LOCAL. Takav primer je prikazan u makrou iz Primera 9-5. Ovaj makro se koristi da prihvati kôd pritisnute dirke sa tastature bez efekta ehoa. Makro se završava sa posatljanjem na nulu markera prenosa CF = 0 kada se prihvati kôd od standardnog ASCII karaktera ili postavljenem markera prenosa CF = 1 kada se prihvati kôd proširenog ASCII znaka. Proširenim ASCII karakterima dodeljuju se kodovi od 80H do FFH.

Primer 9-5

```
_KEYS MACRO    LO,HI       ;početak makroa
      LOCAL  KI,K2,K3      ;identifikacija LOCAL labele
K1:
      MOV    BP,L0-1       ;iniciraj BP na LO-1
K2:
      INC    BP            ;inkrementiraj BP
      CMP    BP,HI+1       ;test da li je iznad HI
      JE     KI            ;ako je ispod HI
      MOV    AH,6          ;pribavi kôd dirke, bez ehoa
      MOV    DL,-1
      INT    21H
      JE     K2            ;ako nije pritisnuta dirka
```

```
        OR      AL,0            ;test za proširenI kôd
        JNE     K3              ;regularni ASCII kôd
        INT     21H             ;dobavljanje proširenog ASCII kôda
        STC                     ;postavljanje markera prenosa
K3:
        ENDM
```

U ovoj Laboratorijskoj vežbi lokalne promenjive su definisane kao K1, K2 i K3. Veoma je važno da je bilo koja promenjiva u makrou definisana kao lokalna. Ovo dozvoljava asembleru da umetne vlastite promenjive, koje su jedinstvene za svaku lokalnu promenjivu, uvek kada se poziva makro. Promenjive koje umetne asembler su numerisane i zamenjuju lokalne promenjive.

KORAK 4: Kreirati program koji koristi makro (smestiti makro u svom makro *include* fajlu) u iz Primera 9-5 koji prihvata i prikazuje slučajne brojeve koji se nalaze u opsegu od 1 do 9. Program treba da prikaže poruku sledećeg sadržaja Pritisnuti bilo koje taster da bi se prikazai bilo koji proizvoljni broj: pre nego što se pozove _KEYS makro. Nakon povratka iz _KEYS, u registar BP će se nalaziti slučajan broj iz opsega od 1 do 9 koji se konvertuje u ASCII kôdiran broj (31H - 39H) dodavanjem broja 30H. Taj broj se tada prikazuje pozivom _DISP makroa koji je prikazan u Primeru 9-3. Ponoviti ovaj program dok se ne pritisne na tastaturi Ctrl + C (ASCII 03H). Da bi otkucali Ctrl + C treba držati pritisnutu dirku Ctrl a dodatno pritisnuti C dirku.

Pitanja

1. Kojim iskazima u asemblerskom jeziku se opisuju makro sekvence?
2. Kako se vrši prenos parametrara kod makro sekvenci?
3. Gde se može pamtiti makro sekvenca kada se poziva u okviru programa?
4. Koja je svrha LOCAL diretive?
5. Koja je svrha INCLUDE diretive?
6. Kreirati makro sekvencu koja prikazuje niz karaktera na bilo kojoj memorijskoj lokaciji. Ovo zahteva da kako ofset tako i segmentna adresa proslede makrou kao parametri.
7. Kreirati makro sekvencu kojaom će se identifikovati da li je kreirani program vaš. Ovaj makro mora da prikaže vaše ime, verziju i ostale informacije koje identifikuju softver kao vaše vlasništvo.
8. Kreirati makro koji se naziva _CRLF koji će prikazati *carriage return* (0DH) i *line feed* kombinaciju (0CH). Treba pri kreiranju biti vispren i koristiti _DISP makro u ovom makrou. Ova tehnika se naziva gnježdjenje. Naglasimo da se _DISP makro mora pojavljivati pre nego što se počne koristiti _CRLF makro.
9. Na koji način u okviru makro programa se koristi komentar?
10. Kreirati makro program koji prikazuje niz karaktera koji se završava nulom (00H) umesto uobičajenog dolar znaka (24H). Segment i ofset adresa su parametri koji se prosledjuju u ovom makrou?

LABORATORIJSKA VEŽBA 10

Konverzija podataka

Uvod

U prethodnim Laboratorijskim vežbama samo smo prikazivali ASCII karaktere na video displeju ili smo pribavljali kôd pritisnute dirke sa tastature. Kod najvećeg broja slučajeva ovakav rad je adekvatan, ali se ponekad javlja potreba da se pribave numerički podaci kodirani u formi decimalnih ili heksadecimalnih podataka. Ova Laboratorijska vežba ukazuje na način korišćenja decimalnih i heksadecimalnih podataka u programima i pokazuje na način kreiranja procedura ili makroa koje se koriste za konverziju i prikazivanje ovih tipova podataka. Ova vežba takodje pokazuje kako se broj proizvoljne osnove može pribaviti i prikazati.

Predmet rada

1. Pročitati heksadecimalni podatak sa tastature i asemblirajti ga kao bajt ili reč.
2. Pročitaj decimalni podatak sa tastature i asemblirajti ga kao reč.
3. Prikazati heksadecimalni podatak na video displeju.
4. Prikazati decimalni podatak na video displeju.

Postupak rada

Postoji više načina za konvertuju brojeva iz decimalne brojne osnove u neku proizvoljnu brojnu osnovu. U toku ove Laboratorijske vežbe ukazaćemo kako se vrši konverzija broja proizvoljne brojne osnove u decimalni broj. Program koji je ovde napisan je uopšten i u velikoj meri fleksibilan.

Prikazivanje broja proizvoljne osnove

Osnovna pretpostavka koja se odnosi na prikazivanje podataka bilo koje brojne osnove bazira se na operaciji deljenja. Ako se binarni broj deli sa 10, ostatak (količnik) deljenja se pamti kao značajna cifra, tada ostatak mora biti broj koji se nalazi u opsegu od 0 do 9. Sa druge strane ako se broj podeli sa 8, ostatak mora biti izmedju 0 i 7. Zbog ovoga ostatak će biti druge brojne osnove u odnosu na ulazni broj koji je bio brojne osnove 2. Da bi konvertovali binarni broj u broj druge brojne osnove, treba koristiti sledeći algoritam:

1. Podeliti broj koji se konvertuje željenom brojnom osnovom.
2. Sačuvati ostatak kao značajanu cifru rezultata.
3. Ponoviti korake 1 i 2 dok rezultantni količnik ne bude nula.
4. Prikažati ostatake kao cifre rezultata, naglasimo da je prvi ostatak najmanje značajna cifra, dok je zadnji ostatak najveća značajna cifra.

Kod Primera 10-1 primenjen je ovaj algoritam u formi makro sekvence (može biti i procedura ako je potrebno), koja konvertuje binarni sadržaj AX-a u broj proizvoljne osnove koji se prikazuje na video displeju. Ovaj makro program pretpostavlja da svaki broj iznad brojne osnove 10 koristi slova od A do Z za one cifre koje su veće od 9. ovaj program obezbedjuje konverziju i prikaz iz binarnog broja u broj bilo koje brojne osnove od 2 do 37 (šta više i prikazivanje binarnog broja za slučaj da je izabrana brojna osnova 2).

Primer 10-1

```
_NUM  MACRO RADIX          ;prikazivanje brojne osnove koja se čuva u AX
   LOCAL    N1,N2,N3
   PUSH     BX             ;pamćenje u magacinu stanja registra
   PUSH     DX
   MOV      BX,RADIX
   PUSH     BX             ;ukazivanje na kraj broja
N1:
   MOV      DX,0           ;DX = 0
   DIV      BX             ;deljenje para registara DX:AX brojnom osnovom
   ADD      DL,30H         ;konverzija u ASCII
   .IF      DL > 39H
            ADD DL,7
   .ENDIF
   PUSH     DX             ;pamćenje stanja ostataka u magacinu
   CMP      AX,0
   JNE      N1             ;granaj se kada količnik nije nula
   MOV      AH,2
N2:
   POP      DX             ;pribavljanje iz magacina vrednosti cifre
   CMP      DX,BX
```

```
    JE      N3                  ;gramnaj se ako je konverzija završena
    INT       21H
    JMP       N2                ;ponoviti dok se ne završi
N3:
    POP       DX                ;ponovno obnavljanje registara iz magacina
    POP       BX
    ENDM
```

Smestiti ovaj makro u makro fajl MACS.INC ili koristiti PROC i ENDP iskaze da bi kreirale proceduru. Kada se obavi ova aktivnost, makrou se može pristupiti od strane bilo kog drugog programa. Uočimo kako iskazi .IF i .ENDIF testiraju DL da bi sabrali dodatnih 7 sa sadržajem registra DL kada je vrednost registra DL veća od 39H (ASCII kôdirano 9). Na primer, ako je DL=3AH, tada se sadržaju registra dodaje 7 i dobija se 41H (ASCII kôdirano A). Ovo je potrebno uraditi kada je brojni sistem veći od osnove 10 kako bi se taj broj se mogao prikazati ovim makroom. Da bi se prikazao sadržaj AX-a osnove 10 treba koristiti makro _NUM 10. Da bi se prikazao sadržaj AX-a u osnovi 16 treba koristititi makro _NUM 16, i tako dalje.

KORAK 1: Koristeći makro sekvencu prikazanu u Primeru 10-1, kreirati program koji puni AX sa 1A34H i prikazuje njegov sadržaj u heksadecimalnoj, decimalnoj, oktalnoj, i binarnoj formi. Vrednost prikazana u heksadecimalnoj = _____, decimalnoj = _____, oktalnoj = _____, i binarnoj = _____, formi. Prikazati svaku od ovih vrednosti u razdvojenim linijama. Naglasimo da ovaj makro (_NUM) ne pamti sadržaje AX-a na način kako ih prikazuje.

KORAK 2: Sa ciljem da pokažemo da ovaj program radi korektno treba kreirati program koji prikazije heksadecimalni sadržaj memorijske oblasti koja počinje od lokacije F800:0000 do lokacije F800:000F. Koristiti makro kreiran iz Primera 10-1. Programer treba biti svestan da se po jedan blanko znak nalazi izmedju svakog broja koga treba prikazivati.

Da li ovaj makro može biti korišćen za pregled memorije kao kada se koristi D (*dump*) naredba kod DEBUG-u?

Drugi način prikazivanja heksadecimalnog podatka dat je Primeru 10-2. Ovaj makro se obično koristiti da prikaže heksadecimalne podatke isped kojih se nalaze vodeće nule. Na primer, u koraku 2 smo uočili da makro _NUM 16 potiskuje vodeće nule tako da se 0AH prikazuje kao A. Vodeća nula ne prikazuje se. Makro _HEXAL čiji je sadržaj izlistan u Primeru 10-2, prikazije podatak koji se čuva u registru AL sa dve cifre u heksadecimalnom formatu bez istisnutih vodećih nula.

Primer 10-2

```
_HEXAL   MACRO                  ;prikazati AL kao heksadecimalni broj
         PUSH    AX
         SHR     AL,1           ;pozicijonirati levu cifru
         SHR     AL,1
         SHR     AL,1
         SHR     AL,1
         ADD     AL,30H         ;konvertovanje u ASCII
         .IF     AL>39H
         ADD     AL,7
         .ENDIF
```

```
DISP    AL              ;koristi DISP makro
POP     AX
AND     AL,0FH
ADD     AL,30H
.IF     AL > 39H
 ADD     AL,7
.ENDIF
DISP    AL
ENDM
```

KORAK 3: Prepisati program iz koraka 2 tako da koristi makro izlistan u Primeru 10-2 umesto makroa _NUM16 iz Primera 10-1. Izvršiti ovaj program i porediti prikazani izraz na CRT displeju sa onim koji se dobija iz koraka 2.

Čitanje podatka bilo koje brojne osnove

Da bi se pribavio podatak sa tastature zadat u bilo kojoj brojnoj osnovi i konvertovali ga u binarni, neophodno je da pomnožite osnovu broja kako bi generisali binarni rezultat. Kada se podatak pribavi sa tastature, 30H se oduzima od brojeva dok se 37H oduzima od slova da bi se generisao binarni ekvivalent otkucanog kôda. Da bi se asemblirao binarni kôd dirke treba koristiti sledeći algoritam:

1. Postaviti rezultat na nulu.
2. Pribavite kôd pritisnute dirke sa tastature, proverite da li je to kôd velikog slova, a a ko je slovo i opseg vrednosti koda je korektan 30H od broja (0-9) a 37H od slova (A-Z). Ako je kôd pritisnute dirke blanko, zarez, tab ili enter završavite makro ili proceduru.
3. Pomnožiti rezultat, obrisan u koraku 1, željenom brojnom osnovom.
4. Dodati binarnu vrednost dirke ovom rezultatu i ponoviti korake 2 i 3.

Zbog iznosa posla koji je potreban da bi ovaj makro obavio kako bi pribavio broj bilo koje brojne osnove, ova aktivnost može izgledati duža od aktivnosti ostalih makroa koji su izloženi u ovom priručniku za Laboratorijske vežbe. Kao i kod svih ostalih makroa, moguće je koristiti proceduru umesto makroa. Primer 10-3 prikazuje listing makroa _RNUM, koji koristi jedan parametar za preneos željene brojne osnove makrou. Uočimo kako ova makro sekvenca izbacuje sve neželjne ili nevažeće kôdove pritisnutih dirki. Takodje uočimo kako se više puta koriste .IF i .ENDIF iskazi kako bi konvertovali mala slova u velika slova; proverili da li se pribavio kôd dirke blanko, zarez, tab ili enter; proverio da li se pribavio kôd dirke za broj ili slovo; i istestiralo da bi se ustanovilo da li uneti broj ili slovo pripada odredjenom (korektonom) opsegu brojeva. Uočimo takodje AND (&&), OR (||), i jednako (= =) operatori koriste u ovom makrou. Makro takodje koristi DOS INT 21H, funkciju 07H pribavi kôd pritisnute dirke bez efekta ehoa tako da se nevežeći karakteri mogu lako izbrisati.

Primer 10-3

```
_RNUM MACRO  RADIX              ;pribavljanje bilo koje brojne osnove u AX
      LOCAL  RI,R2
      PUSH   DX
      PUSH   BX
      PUSH   BP
      MOV    BX,RADIX
```

```
      MOV    BP,0                  ;obrisati rezultat
R1:
      MOV    AH,7                  ;pribavljanje kôda dirke bez efekta ehoa
      INT    21H
      .IF    AL==',' || AL==' ' || AL==9 || AL==13
      JMP    R2
      .ENDIF 1
      .IF    AL>='a' && AL<='z'
      SUB    AL,20H
      .ENDIF
      .IF    AL>91 && AL<'A'
      IMP    RI
      .ENDIF
      .IF    AL<'0' && AL>'Z'
      JMP    RI
      .ENDIF
      MOV    DL,AL                 ;zapamti ASCII
      SUB    AL,30H

      .IF    AL > 9
      SUB    AL,7
      .ENDIF
      .IF    AL>=BL
      JMP    Rl
      .ENDIF
      MOV    AH,0
      PUSH   AX
      DISP   DL                    ;eho dirka
      XCHG   BP,AX
      MUL    BX
      XCHG   BP,AX
      POP    AX
      ADD    BP,AX
      IMP    Rl
R2:
      DISP   AL                    ;eho ograničavač
      MOV    AX,BP                 ;dobijanje rezultata
      POP    BP
      POP    BX
      POP    DX
      ENDM
```

KORAK 4: Koristeći makro iz Primera 10-3, kreirati program koji pribavlja kôd broja pritisnute dirke sa tastature u decimalnom formatu i prikazuje ga kao decimalni, heksadecimalni, oktalni ili binarni. Koristi makro kreiran u Programu 10-3 (NUM) da bi prikazali broj pribavljen od strane makroa _RNUM 10 za sve pomenute brojne osnove.

KORAK 5: Kreirati program koji prihvata dva heksadecimalna broja (_RNUM 16) i prikazuje decimalnu sumu (_NUM 10). Kod pisanja programa znak plus i znak jednako tretirati kao ograničavače, (delimitere) a zatim promeniti .IF iskaz u _RNUM modifikovanjem .IF AL = =',' || … iskaza sa .IF AL = ='+' || AL = = '=' || AL = =',' … iskazom. Ovo dozvoljava da se znak plus i znak jednako koriste da ukažu na kraj broja ili da se koriste kao dopuna za blanko, tab, zarez ili enter dirki. Format ovog programskog prikaza se pojavljuje u Primeru 10-4, on mora ponavljati sabiranje dok se ne otkuca kôd dirke zarez.

Primer 10-4

1A + 20 = 58
3C1 + 3A = 1019

Pitanja

1. Objasni kako funkcioniše instrukcija .IF AL = =10H.
2. Zašto se u .IF iskazu koristi operator ||?
3. Zašto se u .IF iskazu koristi operator &&?
4. Koji se relacioni operator koristi za 'veće od' ili 'jednako' ?
5. Kreirati program koji koji koristi macro iz ove Laboratorijske vežbe da bi se pročitao sa tastature binarni podatak i prikazao ga u formatu brojne osnove 11.
6. Kreirati program koji pribavlja (čita sa tastature) decimalni broj 123 i prikazuje ga u formatu brojne osnove 20. Koji je rezultat u osnovi 20?
7. Koja je svrha .IF AL >='a' && AL <= 'z' iskaza u Primeru 10-3 ?
8. Koja je svrha .IF AL >='9' && AL <= 'A' iskaza u Primeru 10-3 ?
9. Koja je svrha .IF AL >= BL iskaza u Primeru 10-3 ?
10. Koja je svrha = = operatora kada se koristi sa .IF iskazom ?

LABORATORIJSKA VEŽBA 11

Lookup tabele

Uvod

Lookup tabele (tabele pretraživanja) se koriste za konverziju podataka, ali takodje i za pristup podacima koji su zadati u tabelarnoj formi. Podaci zadati u tabelarnoj formi mogu biti numerički i alfabetski podci, nizovi karaktera kao i podaci u binarnoj formi. Ova Laboratorijska vežba koristi *lookup* tabele da bi ilustrovala kako se vrši konverzija podataka iz jednog kôda u drugi a takodje i da lookup preslika jedan niz znakova u drugi. Pored toga ova Laboratorijska vežba ilustruje kako se pomoću tabele grananja (*multiple jump table*) pristupa različitim delovima programa.

Predmet rada

1. Koristi *lookup* tabelu da bi konvertovali podatke iz jednog numeričkog kôda u drugi.
2. Locirati i prikazati niz karaktera koji se pamti u *lookup* tabeli.
3. Za odredjene adrese grananja (*jump*) ili poziva (*call*) koristi *lookup* tabelu da bi odredio adresu grananja ili poziva.

Postupak rada

Lookup tabela je grupa podataka tako organizivana da lako pristupa njenim elementima a da pri tome nije nephodno vršiti njihovo pretraživanje. Na primer u *lookup* tabeli se može čuvati informacija kao što je 7-segmentni kôd za numerički prikaz ili se može čuvati informacija u EBCDIC kôdu (engl. *extended binary coded decimal interchange code*) koji se konvertuje u ASCII kôd pomoću *lookup* tabelue Nijedan od ovih kôdova ne može biti konvertovan us pomoć neke numeričke tehnike kao što je slučaj kod konverzije podataka iz ASCII kôda u BCD kôd (vrši se oduzimanjem 30H).

Pretpostavimo da se alfabetski podaci umesto u ASCII moraju kodirati nekim drugim kôdom. U tom slučaju *lookup* tabela se može koristiti za čuvanje šifriranih kôdova. Ova *lookup* tabela kao i

procedura koja konvertuje ASCII znake u šifrirane kôdove prikazano je u Primeru 11-1. Naglasimo da ASCII kôdiran karakter koji se čuva u registru AL se konvertuje u odgovarajuću adresu elementa koji se čuva u tabeli korišćenjem XLAT instrukcije. XLAT instrukcija sabira sadržaj registra AL sa sadržajem registra BL, čuva reyultat u BL da bi se na ovaj način dobila adresa u tabeli. Ovom instrukcijom se zatim vrši prenos kopije podatka sa te adrese u registar AL. Ovaj program koristi dve kodovane tabele: UTAB kodira velika slova a LTAB tabela kodira mala slova. Slova koja se čuvaju u tabeli se mogu promeniti za različite šifrirane kodove. Kod drugih govornih područja podaci se mogu kodirati drugačije. Ovaj primer može jedino biti praktičan kod pisanja anagrama ili slovnih ukrštenica iz novina. XLAT instrukcija koristi segmentno dopisni prefiks radi promenu inicijalno dodeljenog segmenta (DS).

Primer 11-1

```
UTAB    DB    'MNBVCXZLKJHGFDSAPOIUYTREWQ'
LTAB    DB    'bgtnhymjukilopvfrcdexswzaq'

ENCRP   PROC   NEAR              ;šifriraj AL u odgovarajući kôd u UTAB i LTAB
        .IF    AL>='A' && AL<='Z'
        MOV    BX,OFFSET UTAB
        XLAT   CS:UTAB
        RET
        .ENDIF
        .IF    AL>='a' && AL<='z'
        MOV    BX,OFFSET LTAB
        XLAT   CS:LTAB
       .ENDIF
        RET
ENCRP   ENDP
```

KORAK 1: Kreirati *lookup* tabelu koja sadrži velika slova od A do Z i kreirati proceduru koja testira AL za velika slova. Kreirati proceduru koja ispituje AL i ako sadrži malo slovo koristi *lookup* tabelu da bi ga konvertovali u veliko slovo. Zatim napisati program koji pribavlja kod pritisnute dirke sa tastature korišćenjem DOS INT 21H, funkcija 7 (bez efekta ehoa) i konvertati mala slova u velika slova i velika slova u mala slova uz pomoć već kreirane procedure i *lookup* tabele.

Lookup tabela koju smo koristili do sada koristi instrukciju XLAT da bi konvertovala jedan 8-bitni kôd u drugi. Pretpostavimo da *lookup* tabela konvertuje jedan 8-bitni kôd u drugi 16-bitni kôd. Takva tabela je oblika kao u Primeru 11-2. Ova *lookup* tabela sadrži kvadrate brojeva izmedju 0 i 25. Ova Laboratorijska vežba takodje pokazuje listing procedure koja kvadrira broj u AL i vraća rezultat u AX korišćenjem *lookup* tabele. Treba naglasiti da se operacija kvadrirenja može izvsti korišćenjem instrukcije množenja. Tehnika koja se zasniva na korišćenju *lookup* tabele je vrlo brza i zbog toga se koristi kod većine aplikacija koje su zasnovane na mikrokontrolerima.

Primer 11-2

```
STAB   DW    0,1,4,9,16,25,36,49,64,81,100
       DW    121,144.169,196,225,256,289,324,361,400
       DW    441,484,529,576,625
SQAL   PROC    NEAR
       MOV    AH,0               ;AH = 0
       MOV    SI,OFFSET STAB     ;adresiranje STAB
```

```
        ADD    AX,AX                ;udvostruči AX da bi adresirao elem. tipa reči
        ADD     SI,AX               ;sabrati AX sa SI kako bi se generisala adresa
        MOV     AX,CS:[Sl]          ;odredjivanjeodredjivanje kvadrata od AL
        RET
SQAL    ENDP
```

KORAK 2: Kreirati *lookup* tabelu koja sadrži brojeve od 0 do 9 stepenovane na četvrti. Ilustracije radi broj 2 na četvrti stepen je 2*2*2*2 ili 16. Koristeći makro napisanog u Laboratorijskoj vežbi 10 (_NUM 10) da bi prikazao podatke u decimalnom formatu, kreirati program koji prikazuje brojeve od 0 do 9 stepenovane na četvti. Prikaz mora biti tako urašen da sadrži brojeve od 0 do 9 u levoj koloni sa znakom jednakosti i stepenovanim brojem na četvrti.

Još jedna veoma korisna *lookup* tabela je ona koja sadrži ASCII kôdirani niz znakova. Primer 11-3 prikazuje *lookup* tabelu koja sadrži ASCII kôdiran niz znakova koji se odnose na dane u nedelji. Program iz Primera 11-3 takodje čini i procedura koja čita podatke iz DOS-a i prikazuje dane u nedelji. Uočimo kako se u *lookup* tabeli (DTAB) čuvaju adrese znakovnih nizova od Nedelje do Subote.

Primer 11-3

```
CODE    SEGMENT   'code'                ;označavanje početka kôdnog segmenta
        ASSUME    CS:CODE
DTAB    DW       D0,Dl,D2,D3,D4,D5,D6
D0      DB       'Nedelja$'
D1      DB       'Ponedeljak$'
D2      DB       'Utorak$'
D3      DB       'Sreda$'
D4      DB       'Cetvrtak$'
D5      DB       'Petak$'
D6      DB       'Subota$'
        MAIN     PROC FAR
           MOV    AX,CS                        ;DS = CS
           MOV    DS,AX
           MOV    AH,2AH                       ;dobijanje datuma
           MOV    SI,OFFSET DTAB
           INT    21H
           MOV    AH,0
           ADD    AX,AX
           ADD    SI,AX
           MOV    DX,[SI]                     ;dobijanje adresu niza
           MOV    AH,9                         ;prikazivanje dana u nedelji
           INT    21H
           MOV    AX,4C00H                     ;povratak u DOS
           INT    21H
        MAIN ENDP
CODE    ENDS
        END MAIN
```

KORAK 3: Unesite program iz Primera 11-3, asemblirajte ga i izvršite. Videćete današnji dan prikazan na video displeju.

KORAK 4: Da bi se dobio dan u nedelji, DOS INT 21H, funkcija 2AH takodje vraća i podatak koji je to mesec. Modifikovati program u Primera 11-3 tako da prikazuje dane u nedelji i mesece u obliku Ponedeljak, Mart. Ovo zahteva uvošenje *lookup* tabele u kojoj se čuvaju imena za 12 meseci. Treba biti obazriv, jer je prvi mesec vraćen funkcijom 2AH kao 1 a ne kao 0. Drugi način korišćenja *lookup* tabele bazira se na pamćenju adresa umesto niza karaktera u tabeli. Ostale adrese mogu biti korišćene sa JMP ili CALL instrukcijom radi pristupa različitim delovima programa ili različitim procedurama.

Program izlistan u Primeru 11-4 pita da li korisnik želi povratak u DOS ili nastavi da postavlja isto pitanje. Lokaciji grananja u ovom programu se pristupa preko *lookup* tabele koja sadrži početnu adresu programa ili mesta u programu gde se vrši povratak u DOS. Uočimo kako se koristi iskaz .IF radi testiranja da li je vadjeći znak postavljan 0 ili 1. Takodje uočimo da labele MAIN1 i EXIT prate dve dvotačke (::) umesto jedna dvotačka (:). Labela je lokalna ako je prati jedna dvotačka a globalna je ako je prate dve dvotačke. U konkretnom slučaju labela MAIN2: je lokalna samo u MAIN proceduri a nije u segmentu CODE. Iz ovog razloga labele MAIN1 i EXIT su globalne tako da se mogu koristiti iz CODE segmenta da bi se pristupilo memorijskim lokacijama MAIN1 i EXIT u iskazu LTAB.

Primer 11-4

```
CODE   SEGMENT  'code'          ;označavanje početka kodnog segmenta
       ASSUME   CS:CODE        ;usvajamo da CS adresira kôdni segm. CODE
LTAB   DW   MAIN1,EXIT
MES    DB   13,10,'Uneti 0 ako zelite  nastavak, ili 1 ako zelite povratak u DOS: $'
   MAIN  PROC FAR              ;početak procedure MAIN
       MOV    AX,CS            ;DS = CS
       MOV    DS,AX
MAIN1:
       MOV    AH,9             ;prikazati poruku MES
       MOV    DX,OFFSET MES
       INT    21H
MAIN2:
       MOV    AH,7             ;čitanje bez efekta ehoa
       INT    21H
       .IF    AL<'0' || AL>'1'
       JMP MAIN2
       .ENDIF
       SUB    AL,30H           ;konverzija u binarni
       MOV    BX,OFFSET LTAB
       MOV    AH,0
       ADD    AX,AX            ;udvustručiti AX
       ADD    BX,AX
       JMP    WORD PTR [BX]
EXIT:
       MOV    AX,4C00H         ;povratak u DOS
       INT    21H
   MAIN  ENDP
CODE ENDS
   END   MAIN
```

KORAK 5: Koristeći *lookup* tabelu, kreirati program koji prikaziuje vaše ime kada se otkuca 1, adresu kada se otkuca 2, i broj telefona kada se otkuca 3. Ako je otkucana nula program se vraća u DOS. Nakon prikazivanja vašeg imena, adrese i telefona program treba ponovo da pokaže meni. Treba imati u vidu da program pristupa četiri različitim potprogramima kako bi izvršio sva četri zadatka. Ako je otkucan bilo koj drugi karakter tada se on mora ignorisati. Zahtev za unos podatka i meni koji se prikazuje u programu mora biti sledećeg oblika.

Glavni meni
0 – povratak u DOS
1 – ime
2 – adresa
3 – broj telefona
Unosite vaš izbor:

Lookup tabeli skokova (*jump lookup table*) se pristupa pomoću instrukcije JMP WORD PTR [BX] u Primeru 11-4. Isti ovakav tip programa se može modifikovati da poziva različite procedure tako što će se JMP instrukcija zameniti CALL instrukcijom. Ako se koristi CALL instrukcija, nakon izvršenja potprograma upravljanje programamom se vraća na instrukciju koja u programu neposredno sledi iza instrukcije CALL. Ovo nije slučaj kod izvršenja instrukcije JMP.

Sve tehnike koje koriste lookup tabele a bile su prikazane do sada koristile su direktno preslikavanje (*lookup*). Naredni deo ove Laboratorijske vežbe pokazije princip rada koji se bazira na pretraživačima tabela. Pretraživač tabela se uvodi korišćenjem bilo SCAS instrukcije (pretraživanje bajta, reči ili duple reči) bilo CMPS instrukcije (pretraživanje polje bajtova). Tip pretraživanja zavisi od podataka koji su memorisani u tabeli.

Pretpostavimo da podatak kodiran u Primeru 11-1 mora biti dekodiran. Ovo se može obaviti korišćenjem SCASB instrukcije kako bi se vršilo pretraživanje kroz tabela velikih i malih slova i odredila (locirana) adresa elementa. Kada je lokacija odredjena, relativna pozicija elementa u tabeli se nalazi u ragistru DI. Ponovnim pozivom SCASB-a pretražuje se polje memorije u ekstra segmentu adresiranom od strane DI registra. Vidi proceduru iz Primera 11-5 koja dekodira (dešifrira) informaciju kodiranu (šifriranu) u Primeru 11-1.

Primer 11-5

```
UTAB  DB  'MN VCXZLKJHGFDSAPOIUYTREWQ'
LTAB  DB  'bgtnhymjukilopvfrcdcxswzaq'
DCRP  PROC  NEAR                  ;dešifriranje AL iz kôda u UTAB i LTAB
      PUSH  ES
      PUSH  CX
      PUSH  DI
      MOV   CX,CS
      MOV   ES,CX
      CLD                         ;odabira se autoinkrementiranje
      MOV   CX,0FFFFH             ;punjenje brojača na maksimalnu vrednost
      .IF   AL>='A' && AL<='Z'
      MOV   DI,OFFSET UTAB
      REPNE SCASB
      SUB   DI,OFFSET UTAB
      ADD   DI,40H                ;skala do velikih slova
      MOV   AX,DI
      .ENDIF
```

```
        POP    DI
        POP    CX
        POP    ES
        RET
DCRP    ENDP
```

U ovom primeru koristi se instrukcija SCASB da bi se pretražila tabela. Instrukcija SCASB modifikuje sadržaj registra DI koji pokaziuje na bajt koji sledi zanak koji nalazi u tabeli. Uočimo kako se broj u DI konvertuje radi pristupa velikim ili malim slovima oduzimanjem početne adrese tabele a zatim dodavanjem 40H ili 60H vrednosti koja se čuva u registru DI. Takoše uočimo kako se koristi REPNE prefikiks radi ponavljanja instrukcije SCASB sve dok je rezulat instrukcije različit u odnosu na rezultat komparacije koja se obavlja operacijom SCASB.

KORAK 6: Koristiti instrukciju SCASB (SCASB adresira podatak u ekstra segmentu adresiran registrom DI), da bi kreirali program koji pretražuje sadržaj memorije počev od memorijiske lokacije F800:0000 radi slova M. Program treba da prikaže ofset adresu ove lokacije korišćenjem NUM 16 makroa koji je već objašnjen u Laboratorijskoj vežbi 10. Ofset adresa je _____.

Pitanja

1. Koristiti instrukciju XLAT kod kreiranja procedure koja je zasnovana na korišćenju *lookup* tabele. Lookup tabela se koristi za konverziju sadrćaja registra AL (0-F) u ASCII-kôdirane zanake za brojeve od 0 do 9 i velika sova od A do F.
2. Koristiti instrukciju XLAT kod kreiranja procedure bazirane na korišćenu *lookup* tabele. *Lookup* tabela se koristi za konverziju ASCII kôdiranih znakova 30H-39H i 41H-46H u brojeve od 0 do 9 i velika slova od A do F. Konvertovani podatak smestiti u registar AL. Naglasimo da ova tabela sadrži i veći broj neiskorišćenih memorijskih lokacija koje se mogu premostiti korišćenjem direktivre DUP kako bi se pamtili duplikat vrednosti.
3. Korišćenjem tehnike objašnjene Primeru 11-2, kreirati proceduru baziranu na korišćenju *lookup* tabele. procedura se koristi za konverziju sadržaja registra AL iz heksadecimalne u ASCII-kôdirane cifre koje se smeštaju u tabeli. na primer, prvi ulaz (elemenat) tabele, koji odgovara AL = 00H treba da čuva vrednost 30H, drugi ulaz koji odgovara AL = 01H treba da čuva vrednost 3031H, itd. Imajući u vidu da je obim ove tabele veliki u tabeli čuvati samo ASCII vrednosti koje odgovaraju heksadecimalnima vrednostima od 00H do 0FH.
4. Koja je svrha i koju aktivnost obavlja instrukcija MOV AC,CS[SI] prikazana u Primeru 11-2.
5. Korišćenjem tehnike prikazane u Primeru 11-3, kreirati program koji prikazuje 8 različitih faza karakterističnih za slučaj kada se pritisne bilo koja dirka sa tastature. Napomenma da se slučajan broj, koji generiše broj izmedju 0 i 7 a čuva se u registru CL, dobija sledećom procedurom:

```
RAND    PROC    NEAR
        INCCL
        MOV     AH,L
        MOV     DL,-1
        INT21H
        JZ   RAND
        AND     CL,7
        RET
RAND    ENDP
```

6. Koja je svrha iskaza .IF AL<'0' || AL>'1' u Primeru 11-4?
7. Objasniti kako u Primeru 11-4 funkcioniše instrukcija JMP WORD PTR [BX] ?
8. Koristeći instrukciju SCASB kreirati proceduru koja analizira memorijski prostor u kome se čuva (smešten je) niz znakova adresiran registarskim parom ES:DI. Naglasimo da je dužina niza kraća od 256 bajtova.
9. Objasniti kako se u Primeru 11-5 konvertuje šifrirano veliko slovo B koje se čuva u registru AL u vrednost 43H koja odgovara C velikom slovu.
10. Ukazati na namenu prefikasa REPNE koji se koristi sa SCASB instrukcijom u Primeru 11-5?

LABORATORIJSKA VEŽBA 12

Korišćenje fajlova na disku

Uvod

Najveći broj programa prilikom rada zahteva korišćenje fajlova na disku, kako bi se podaci memorisali ili pročtali. Ova Laboratorijska vežba pokazuje kako se kreiraju, čitaju, pišu, pridružuju i zatvaraju fajlovi na disku. Pored ostalogcilj ove vežbe je da pokaže i kako se vrši rukovanje fajlovima, koji se koriste za čitanje i unos podataka u računar (uz korišćenje tastature i video displeja). Fajlovima na disku se prvo pristupa njihovim imenom kako bi se odredio način rukovanja nad tim fajlovima. Fajlovima se pristupa na flopi disk jedinici ili hard disk jedinici na bilo kojem direktorijumu ili poddirektorijumu.

Predmet rada

1. Kreirati fajl na disku i upisati podatke u njemu koristeći asemblerski jezik i DOS INT 21H funkcijski poziv.
2. Pročitati bilo koji deo fajla na disku korišćenjem funkcije pokazivača pomeraja u okviru fajla koja je dostupna iz DOS-a.
3. Pridružiti fajlu nove podatke korišćenjem pokazivača na pomeraj u okviru fajla i funkcije upisa koja je dostupna u DOS-u.

Postupak rada

Fajl na disku u formatu koji podržava operativni sistem DOS mora biti otvoren pre nego što se isti pročita ili se u njemu upisuje a zatim se isti zatvori kada je pristup na njemu završen. Ako se javi potreba za novim fajlom, on se prvo kreira i istovremeno otvara radi upisa, a nakon završetka upisa taj fajl se zatvara. kod personalnog računara da bi se obavili ovi zadaci koriste se funkije tipične za, DOS INT 21H .
Primer 12-1 prikazuje listing makroa kojim se kreira i pamti fajl kojim se manipuliše sa memorijskom lokacijom u kojoj se čuva podatak obima reči. Makro _CREAT takodje sadrži parametar koji ukazuje na ofset adresu segmenata podataka ASCII kodiranog fajl imena. Ofset adresa, ime fajla i lokacija fajla kojim se manipuliše moraju se čuvati u segmentu podataka koji se adresira registrom DS. Naglasimo da DOS vraća lokaciju fajla sa kojim se manipuliše u registar AX.

Primer 12-1

```
_CREAT   MACRO HANDLE,FNAME          ;kreiranje FNAME, pamćenje HANDLE
```

```
MOV    AH,3CH                ;kreiranje broja funkcije
XOR    CX,CX                 ;čitanje/upis
MOV    DX,OFFSET FNAME
INT    21H
MOV    HANDLE,AX             ;pamćenje HANDLE
ENDM
```

KORAK 1: Dodati ovaj makro makro fajlu MACS.INC na flopi disk jedinici. Na ovaj način će se brzo kreirati disk fajl.
Kada je jednom fajl kreiran, on najčešće sadrži i podatke koji su upisani u njemu. Primer 12-2 pokazuje makro program koji upisuje podatke u fajl. Ovaj primer takodje pokazuje makro koji zatvara fajl. Ovi makroi takodje koriste HANDLE kao fajl koji se pamti od strane makroa da bi se pristupilo fajlu. Ovaj metod omogućava simultano korišćenje vićeg broja fajlova. Makro za upis sadrži parametar koji ukazuje na broj bajtova koji se upisuje (1 do 64) i ofset lokaciju bafera koja čuva informaciju gdee treba upisivati. Naglasimo da manipulisanje fajlom kao i bafer moraju biti locirani u okviru segmenta podataika kako bi mogli da se koriste od strane ovih makroa.

Primer 12-2

```
_WRITE   MACRO HANDLE,BUFFER,COUNT
   MOV   AH,40H                    ;funkcija upisa u fajl
   MOV   BX,HANDLE
   MOV   CX,COUNT
   MOV   DX,OFFSET BUFFER
   INT   21H
   ENDM
_CLOSE   MACRO HANDLE
   MOV   AH,3EH                    ;funkcija zatvaranja fajla
   MOV   BX,HANDLE
   INT   21H
   ENDM
```

Sva tri makroa predstavljena u dosadašnjem toku ove Laboratorijske vežbe detektuju greške. Ako se javi greška, marker prenosa = 1 ukazuje na pojavu greške. Imajući u vidu da uzrok i tip greške mogu biti različite , nijedan od ovih makroa ne manipuliše uzrocima koji su doveli do greške. Zadatak je programa da pozove makro koji manipuliše greškama pomoću instrukcije JC (*jump if carry*) iza koje sledi poziv makroa. detaljna nformacija koja uklazuje na prirodu greške može se dobiti pozivom INT 21H, funkcija 59H. Ova funkcija ukazuje na akcije koje treba preuzeti kada se javi greška na osnovu broja koji se vraća u BL. Za više detalja videti funkciju 59H.

KORAK 2: Dodati makroe za upis i zatvaranje u vaš makro fajl da bi kasnije koristili ovaj makro Laboratorijskoj vežbi.

Primer 12-3 prikazuje kratak program koji kreira fajl sa flopi diska A u *root* direktorijumu a nakon toga ga zatvara. Naglasimo da ovaj program koristi jedan segment. Registar segmenta podataka DS se puni adresom kôdnog segmenta CS da bi postavio segment podataka na vrh kôdnog segmenta.

Primer 12-3

```
CODE SEGMENT  'code'          ;označavanje početka CODE segmenta
   ASSUME CS:CODE             ;usvajamo da CS adresira kôdni seg. CODE
   INCLUDE MACS.INC           ;označavanje macro fajla
```

```
   HAN1   DW ?
   NAME1 DB 'A:\MYFILE.TST',0
   MAIN   PROC   FAR                ;označavanje početka procedure MAIN
      MOV    AX,CS                  ;DS = CS
      MOV    DS,AX
      _CREAT    HAN1,NAME1          ;kreiranje MYFILE.TST
      _CLOSE    HAN1                ;zatvaranje MYFILE.TST
      _EXIT
   MAIN   ENDP                      ;označavanje kraja procedure MAIN
CODE ENDS                           ;označavanje kraja segmenta CODE
END  MAIN                           ;označavanje kraja i početka fajla
```

Uočimo da se ime fajla ovom programu čuva kao ASCII-Z niz. Niz ASCII-Z se uvek završava znakom (karakterom) nula ili *null* umesto znakom *carriage return* (0DH), *line feed* (0AH), ili dolarskim znakom $. Kod svih DOS fajl funkcija ime fajla se čuva kao ASCII-Z niz. Takodje uočimo da se podaci obima reči memorišu (zapisuju) pre podataka obima bajt. Ovo je važno zbog mikroprocesora jer se podaci obima reči čuvaju u memoriji kao reči. Ako je prisutan podatak obima dupla reč, isti }e se deklarisati pre podataka obima reč. Mikroprocesor izvršava i koristi podatke u bilo kom redosledu, ali ne tako efikasno kao kada su podaci uredjeni tako da su prvo oni obima dupla reč, nakon toga oni obima reč i na kraju podaci obima bajt. Za poravnanje podataka se koristi direktiva ALIGN. Na primer, ALIGN 4 poravnjava podatke na obim duple reči. Ovaj program ne detektuje bilo kakvu grešku, ali pre izvršenja programa korisnik prvo treba proveriti da li je formatirana disketa u disk jedinici A.

KORAK 3: Unesite i asemblirajte program iz Primera 12-3 i izvršite ga. Naglasimo da je potrebno uvrsti putokaz u iskazu INCLUDE ako se fajl MASC.INC ne nalazi u tekućem inicijano radnom ili tekuće zatečenom direktorijumu. Postaviti flopi disk u disk jedinicu A i izvršiti program. Direktorijum disk jedinice A će prikazati fajl MYPROG.TST obima nula bajtova podataka nakon izvršenja.

KORAK 4: Nakon što su dostupni makroi za kreiranje (_CREAT), upisivanje (_WRITE) i zatvaranje fajla (_CLOSE), kreirati program koji će na *root* direktorijumu disk jedinice A kreirati fajl FROG.JMP. Smesti pet bajtova podataka (48H, 65H, 6CH, 6CH i 6FH) u ovaj fajl i nakon ovoga zatvoriti ga. Podaci moraju biti smešteni u segmentu podataka korišćenjem bilo direktive DB ili direktive BYTE. Izvršiti ovaj program a zatim proveri da li flopi disk jedinica A sadrži fajl. A _____ se prikazuje kada se otkuca TYPE A:\FROG.JMP u DOS komandnoj liniji. Sada nakon što je fajl kreiran i zapamćen ukazaćemo na efekat funkcija čitanje i otvaranje fajla. Naglasimo da kada je fajl kreiran uz pomoć DOS INT 21H, funkcija 3CH ne vrši se provera da bi se ustanovilo da li isti zapisan tj. da li on već postoji na disku. Postojeći fajl istog imena biće izbrisan. Ako se ovakav princip rada ne svidja manipulatoru tada treba koristiti DOS INT 21H, funkcija 5BH. Funkcija 5BH ima isti efekat kao i funkcija 3CH osim što prvo proverava da li je fajl prisutan i ako jeste postavlja prenos = 1 bez da obriše postojeći fajl.
Primer 12-4 pokazuje _READ i _OPEN makroe koji se koriste za čitanje bajtova (obima do 64K bajova bez prekida) podataka iz fajla i otvaranje postojećeg fajla radi operacija čitanje ili upis. kao i u prethodnim slučajevima ovi makroi koriste parametar HANDLE koji specificira način manipulisanja sa fajlom. Makro _OPEN uključuje parametar imena fajla (FNAM) kojim se adresira ASCII-Z niz pomoću registra DS radi specifikacije imena fajla. Makro _READ koristi BUFFER i COUNT parametre koji prenose ofset adresu memorijeske oblasti segmenta podataka kako bi se pročitao podatak iz fajla kao i broj bajtova (1 do 64K) koji se čitaju iz tog fajla.

Primer 12-4

```
_OPEN  MACRO  HANDLE,FNAM      ;otvaranje fajla FNAM, pamćenje HANDLE
   MOV   AX,3DO2H             ;čitanje/upis fajl
   MOV   DX,OFFSET FNAM
   INT   21H
   MOV   HANDLE,AX            ;pamćenje HANDLE
   ENDM
_READ  MACRO  HANDLE,BUFFER,COUNT      ;čitanje fajla u BUFFER
       MOV   AH,3FH
       MOV   BX,HANDLE
       MOV   CX,COUNT
       MOV   DX,OFFSET BUFFER
       INT   21H
       ENDM
```

KORAK 5: Korišćenjem makroa _OPEN i _READ, kreirati program koji čita fajl FROG.JMP kreiran u KORAKU 4 i prikazuje sadržaj na video displeju korišćenjem DOS INT 21H, funkcija 09H. Ne treba ispustiti iz vida da ova funkcija zahteva da se niz karaktera mora završiti karakterom 24H ($).

Sada počto fajl FROG.JMP sadrži podatke možemo njemu pridružiti neke dodatne podatke. Kada se fajl otvori pokazivač fajla adresira prvi bajt tog fajla. Kada se fajl otvori a puisuju se novi podaci, tada se često vrši ponovno prepisvanje tekućeg sadržaja tog fajla. Da bi se ovo izberglo, koristi se poziv DOS INT 21H, funkcija 42H kojim se to pre nego što se upiče nova informacija u fajlu pokazivač fajla pomera na kraj fajla. Primer 12-5 prikazuje makro listing makroa _ENDF, koji pomera pokazivač fajla na kraj fajla. Kao i u ostalim makro fajlovima, koristi se HANDLE kako bi se identifikovao fajl.

Primer 12-5

```
_ENDF MACRO HANDLE    ;pomeranje pokazivača na kraj fajla
   MOV   AX,4202H     :pomeranje funkcije pokazivača fajla
   XOR   CX,CX
   XOR   DX,DX
   MOV   BX,HANDLE
   INT   21H
   ENDM
```

Pokazivač fajla obavlja tri različita tipa pomeranja koja se odnose na fajl. (1) ako je registar AL = 0, pokazivač u okviru fajla se pomera od početka fajla pa nadalje; (2) ako je AL = 1, pokazivač u okviru fajla se pomera u odnosu na njegovu tekuću poziciju; i (3) ako je AL = 2 (kao u Primeru 12-5) pokazivač u okviru fajla se premešta na kraj fajla. Broj bajtova za koji pomera pokazivač u okviru fajlova odredjen je 32-bitnim brojem koji se čuva u registru CX (MS 16 bitova) i DX (LS 16 bitova). Naglasimo da makro u Primeru 12-5 počinje od broja nula koji se smešta u registrima CX i DX a pomera pokazivač u okviru fajla sa kraja fajla. Ovaj makro pomera pokazivač prema kraju fajla tako da se tom fajlu može pridružiti nova informacija.

KORAK 6: Kreirati program koji otvara FROG.JMP, postavlja pokazivač fajla na kraj fajla i pridružuje fajlu blanko znak nakon koga sledi vaše ime. Pozvati, asemblirati i izvršiti ovaj program. Ako iz DOS –ove komandne linije izdate komandu TYPE A:\FROG.JMP, računar će otpozdraviti sa Hello za slučaj da program funkcioniče korektno.

KORAK 7: Kreirati program koji modifikuje podatke u FROG.JMP i smešta reč there izmedju reči Hello i vašeg imena. Kreirati fajl koji se naziva FROG.TST i unesite ovu novu informaciju u

fajl FROG.TST. Nemojte menjati sadržaj fajla FROG.JMP. Pozvati, asemblirati i izvršiti ovaj program a zatim komandom TYPE iz DOS komandne linije proveriti njegovu korektnost.
Pozvati i izvršiti sledeći program (Primer 12-6) kako bi se kreirao falj podataka za naredni deo ove Laboratorijske vežbe. Ovaj program kreira fajl koji se naziva TEST.DAT a sadrži informaciju od 128k (131,072) bajtova. Naglasimo da putokaz mora biti specificiran za fajl TEST.DAT a takodje i za makro fajl MASC.INC.

Primer 12-6

```
CODE SEGMENT   'code'          ;označavanje početka CODE segmenta
   ASSUME   CS:CODE           ;usvajamo da CS adresira kôdni seg. CODE
   INCLUDE  MACS.INC          ;umetanje makro fajla
   HAN   DW ?                 ;fajl HANDLE
   NAM   DB 'TEST.DAT',0      ;ime fajla, dodati i putokaz
   BUF   DB 256 DUP(?)        ;bafer podataka
   MAIN  PROC  FAR            ;početak procedure MAIN
      MOV   AX,CS             ;DS = CS
      MOV   DS,AX
      MOV   ES,AX             ;ES = CS
      CLD                     ;selekovanje autoinkrementiranja
      _CREAT  HAN,NAM         ;kreiranje fajla TEST.DAT
      MOV   BP,512            ;potreban broj od 256 bajtova
MAIN1:
      MOV   AX,BP             ;generisanje 256 bajtova podataka za test
      MOV   CX,256            ;postavljanje brojača
      MOV   DI,OFFSET BUF
      REP   STOSB
      _WRITE  HAN,BUF,256     ;upis 256 bajtova
      DEC   BP
      JNZ   MAIN1             ;ponavljanje 512 puta
      _CLOSE  HAN             ;zatvaranje fajla TEST.DAT
      _EXIT                   ;povratak u DOS
   MAIN  ENDP                 ;kraj MAIN
CODE   ENDS                   ;kraj CODE
   END   MAIN
```

KORAK 8: Kreirati program koji pretražuje fajl TEST.DAT za bilo koji deo obima 256 bajta koji sadrži 41H u svakoj lokaciji. Kreirati novi fajl koji se naziva TEST.DET gde će se smestiti delovi iz fajla TEST.DAT koji sadrže podatak 41H. Uočimo da se kod čitanja fajla pomoću makroa _READ, broj bajtova koji se čita vraća se u AX. Ova mogućnost se koristi u programu da bi se odredilo kada se pojavi kraj fajla TEST.DAT. Pozvati, asemblirati i izvršiti program. Da bi se videlo da li on funkcioniše korektno, koristiti komandu TYPE iz DOS-ove komandne linije kako bi se prikazao sadržaj fajla TEST.DET. Koja će se funkcija prikazati na video displeju?
Dok smo diskutirali o manipulacijama nad fajlovima, pomenuli smo da je direktan pristup konzoli računara dostupan preko fajla HANDLE. Tabela 12-1 pokazuje dozvoljene manipulacije nad fajlom koje se koriste kod DOS-a.

Tabela 12-1. Standardne manipulacije nad fajlovima kod DOS-a

Tip aktivnosti	**Funkcija**
0000H	čitanje tastature (CON:)
0001H	prikazivanje na video displeju (CON:)
0002H	prikazivanje greške (CON:)
0003H	korišćenje COM1 port (AUX:)
0004H	koričćenje porta štamača (PRN:)

Makro predstavljen u ovoj Laboratorijskoj vežbi se koristi da adresira video displej (_WRITE) ili tastaturu (_READ) ako je odgovarajuća aktivnost specificirana makroom. Ovi makroi mogu takodje adresirtati COM port (broj porta se može promeniti DOS komandom).
Primer 12-7 prikazuje na video displeju listing kratkog programa koji koristi HANDLE (0001H) za prikaz pomoću makroa _WRITE. Naglasimo da makro _WRITE u ovom programu koristi manipulaciju tipa 1 (tip aktivnosti 0001H) nad fajlom kako bi prikazao sadržaj MES. Takodje uočimo kako se odredjuje broj bajtova koji se upisuje koristeći razliku izmedju DUMMY i MES.

Primer 12-7

```
CODE  SEGMENT  'code'              ;označavanje početka segmenta CODE
ASSUME   CS:CODE             ;usvajamo da CS adresira kôdni seg. CODE
INCLUDE  MACS.INC
      MES    DB  13,10,'Ovo je interesantno!'
      DUMMY  DB  ?
      MAIN  PROC   FAR            ;početak procedure MAIN
              MOV   AX,CS
              MOV   DS,AX
              _WRITE 1,MES,DUMMY-MES
              _EXIT
      MAIN   ENDP
CODE  ENDS
      END    MAIN
```

KORAK 9: Koristeći makro _WRITE kreirati program koji po vašem izboru prikazuje dve linije informacije na video displeju. Za slučaj da je dotupan štampač dostupan na LPT1 portu, promeniti HANDLE na 4 i ponovo pokušati izvršenje programa za štampanje.

Pitanja

1. Opisati opcije koje su dostupne preko registra CX kada se pozove DOS INT 21H funkcija (3CH).
2. Da li se DOS-ova funkcija za kreiranje fajlova može koristiti da bi se kreirao novi direktorijum? Ako može kako se to radi?
3. Objasni zašto se fajl mora zatvoriti, naročito nakon operacije upisa.
4. Kako se više od 64k bajta podataka može smestiti u fajl?
5. Šta je to ASCII-Z niz?
6. Kako se otkriva greška pomoću DOS-ovim funkcija za manipulisanje fajlovima?
7. Šta je to pokazivač fajla?
8. Napisati makro koji se naziva _TOPF a pomera pokazivač fajla na početak fajla.
9. Napiši program koji kreira dva nova fajla (UP.DAT i DOWN.DAT). Napisati 02H u svaki od 256 bajtova UP.DAT fajlu i 03H u svaki od 256 bajtova fajla DOWN.DAT.

10. Koristeći fajl koji manipuliše sa tastaturom, kreirati makro koji čita jedan karakter sa tastature. Nazvati10 novi makro _KEYF.

LABORATORIJSKA VEŽBA 13

Pristup video memoriji - VGA kontroler

Uvod

U ovoj Laboratorijskoj vežbi posebna pažnja biće posvećena načinu korišćenja registara u okviru funkcija koje se odnose na rad VGA kontrolera, kao i poziva BIOS INT 10H softversko prekidno uslužne rutune radi pristupa video memoriji u grafičkom 12H režimu rada. U režimu rada 12H VGA kartica prikazuje sliku rezolucije 640 x 480 sa 16 boja. Dostupni su i drugi režimi rada, ali oni nisu objašnjeni u ovoj Laboratorijskoj vežbi. i pored toga što je cilja ove vežbe da posluži kao primer, ona ilustruje tehnike koje se mogu koristiti ya postiyanje većih rezolucija koje opet nisu trenutno od interesa za ovu Laboratorijsku vežbu.

Predmet rada

1. Korišćenje grafičkog režima 640 x 480 sa 16 boja radi prikaza bilo koje boje na bilo koju poziciju tačke na ekranu video displeju.
2. Korišćenje režima rada 12H radi podele ekrana na 53 linije sa po 80 karaktera po liniji tako da se mogu prikazivati blokovi boja.
3. Prikaz teksta u grafičkom režimu 640 x 480 tačaka sa 16 boja, bez izmene boje pozadine.

Postupak rada

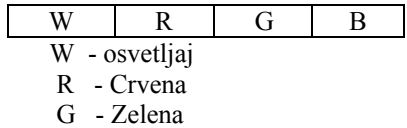
Prvi deo ove Laboratroijske vežbe odnosi se na grafički režim rezolucije 640 x 480 tačaka i 16 boja.

Grašički režim 640 x 480 sa 16 boja za prikaz

Ovaj režim rada karakteriše se kôdom 12H, a zahteva da u PC konfiguraciji bude uključena i video kartica tipa VGA (*Video Graphic Array*).
U jednoj od prethodnih vežbi bavili smo se direktnim pristupom video memoriji tekstualnog sadržaja u režimu rada 3. To je režim rada koji DOS uobičajeno koristi za svoj prikaz. U toj vežbi naučili smo da je video memorija locirana počev od adrese B800:0000 pa do B800:FFFF i da sadrži ASCII kôdirane podatke (odnosi se na informaciju) kao i atribut podatke (odnosi se na efekte prikaza) za potrebe video prikaza. Režim rada sa 16 boja i sa rezolucijom 640x480 tačaka (nazivaju se osnovni elementi slike ili *piksel*-ima) je znatno kompleksniji dok je video meorija locirana počev od adrese A000:0000 pa sve do adrese A000:FFFF. Uz malo računice može se pokazati da je za prikaz 16 boja dovoljno 4 bita pri rezoluciji od 640 x 480 (307.200 tačaka) memorijskom sistemu potrebno 1.228.800 bitova, tj. 153.600 bajtova. Na raspolaganju je, medjutim, dostupan samo prostor od 64 kB.
Da bi se obezbedio pristup ovolikom iznosu memorije, proizvodjači video adaptera standardizovali su režim rada 12H tako da se video memoriji pristupa korišćenjem principa bit-ravni. Bit-ravan je linearni deo memorije koji sadrži po jedan od 4 bita koij su potrebni za prikaz 16 boja. Svaka bit-ravan zauzima memorijski prostor od 307.200 bitova, koji se čuvaju u memoriju I zauzimaju prostor od 38.400 bajtova. Prostor od 64 kB u segmentu A000 je dovoljan za adresiranje samo jedne bit-ravani u datom trenutku. Bit-ravan je jocirana počev od adrese

A000:0000 pa do adrese A000:95FF. Lokacija A000:0000 odnosi se na osam tačaka koje se nalaze na krajnjoj gornjoj levoj pozticiji ekrana, a lokacija A000:95FF na osam tačaka pozicioniranih na krajnje donjoj desnoj strani video displeja. Postoje četiri ravni ili memorijskih banaka izmedju kojih postoji preklapanje u ovom adresnom prostoru a koriste se za prikaz svake tačke na video displeju sa rezolucijom od 640 x 480 u 16 boja. Naglasimo da su kodovi boja tako formirani da se bit koji je krajnji levo odnosi na atribut 'osvetljaj', a ostali bitovi predstavljaju crvenu, plavu i zelenu boju, respektivno (Slika 13.1).

kôd	**boja**
0000	Crna
0001	Plava
0010	Zelena
0011	Cijan (maslinasta)
0100	Crvena
0101	Magenta (Purpurna)
0110	Braon
0111	Bela
1000	Siva
1001	Nezasićeno (Pastelno) plava
1010	Nezasićeno zelena
1011	Nezasićeno cijan
1100	Nezasićeno crvena
1101	Nezasićeno purpurna
1110	Nezasićeno žuta
1111	Jako bela



Slika 13-1. Bit oblici boja raspoloživih kod VGA, režim rada 12H

Pristup memoriji se ostvaruje u sledećim koracima:

1. Pročitati bajt koga treba promeniti.
2. Selektovati onaj bit ili bitove u okviru bajta koga/koje treba promeniti pomoću vrednosti indeksa 8 adresnog registra VGA kartice upisom 8 na U/I port 3CEH. Zatim, napuniti u registar AL bit oblik koga treba promeniti (jedinica na odgovarajućoj bit ppoziciji ukazuje da se taj bit menja) i upisati poslati ovaj podata na port 3CFH tj. u registar maske bitova (BMR).

```
MOV     DX,3CEH     ;selektovanje adresnog registra VGA kartice
MOV     AL,8        ;index 8
OUT     DX,AL       ;postavljanje indexa 8 na U/I port
MOV     DX,3CFH     ;selektovanje registar maske bitova
MOV     AL,80H      ;smeštanje maske u registar AL
OUT     DX,AL       ;selektovanje nižih bitova (u ovom primeru 80H)
```

3. Nakon ovoga, u registru mape maske (MMR) postaviti sve bitove maske na 1111 na vrednost ofset sekvence 2 i upisati boju 0 u VGA karticu kako bi se postavila boja na crno. Bitovi maske selektuju bit ravni koje treba menjati menjati. Ako su sve bit-ravni selektovani i ako je upisana boja 0, tada će se sve četri bit-ravni biti postavljene na 0.

```
MOV     DX,3C4H     ;selektovanje registra sekvenci VGA kartice
MOV     AL,2        ;vrednost indexa jednaka 2
OUT     DX,AL       ;selektovanje index 2
MOV     DX,3CSH     ;adresiranje MMR-a (map mask register)
MOV     AL,OFH      ;postavljanje maske registra na 1111 binarno
```

```
        OUT     DX,AL
```

;dalje, upisati binarnu vrednost 00H u selektovanu lokaciju video memorije

4. Postaviti željeni broj boje u MMP i upisati FFH u video memoriju. Na ovaj način se postavlja logička jedinica u samo jednoj selektovanoj bit-ravni čime su ostvareni uslovi za upis nove boje tačke koju treba prikazati na video displeju.

```
        MOV  AL,3
    OUT  DX,AL
```

;dalje, upisati FFH u selektovanu memorijjsku lokaciju video memorije

Prikaz jednog elementa slike u režimu rada12H

Prikaz informacije u režimu rada 12H nije jednostavan zadatak, kao što se to i moglo videti iz prethodnog primera. Primer 13-1 prestavlja listing programa koji prikazuje jednu nezasićeno-zelenu tačku u gornjem levom uglu ekrana u režimu rada 12H. Takodje, bira se režim rada 12H uz pomoć INT 10H, a zatim, nakon toga se čeka na pritisak bilo koje dirke na tastaturi, i na kraju resetuje video režim rada na 3. Nakon startovanja programa, primećuje se tačka u gornjem levom uglu koja je nezasićeno-zelena ali je veoma mala i jedva primetna.

Primer 13-1

```
CODE    SEGMENT  'code'
        ASSUME   CS:CODE
        INCLUDE  MAS.INC
MAIN    PROC FAR
        MOV AX,12H                  ;izbor režima rada 12H
        INT 10H
MOV AX,0A000H                       ;adresiranje segmenta video memorije
MOV DS,AX
MOV SI,0                            ;adresiranje prvog bajta
MOV AL,[SI]                         ;čitanje prvog bajta
MOV DX,3CEH                         ;selekcija VGA registra adrese
MOV AL,8                            ;indeks 8
OUT DX,AL
MOV DX,3CFH                         ;selekcija registra bit maske
MOV AL,80H                          ;selekcija krajnjeg levog bita
OUT DX,AL                           ;izbor bita
MOV DX,3C4H                         ;registar sekvence adresa
MOV AL,2                            ;indeks 2
OUT DX,AL
MOV DX,3C5H                         ;adresa MMR-a
MOV AL,0FH                          ;selekcija svih ravni
OUT DX,AL
        MOV BYTE PTR [SI],0         ;upis nule radi postavljanja crne boje
        MOV AL,0AH                  ;postavljenje boje na nezasićeno-zelenu
        OUT DX,AL                   ;postavljanje boje
        MOV BYTE PTR [SI],0FFH      ;upis boje
        MOV AH,7                    ;pauza, očekuje se pritisak dirke sa tastat.
        INT 21H
```

```
        MOV AX,3                    ;ponovo postaviti rezim rada na 3
        INT 10H
        _EXIT                       ;prelaz na DOS
       MAIN ENDP
CODE   ENDS
       END MAIN
```

Očigledno je na osnovu dužine listinga prethodnog programa iz Primera 13-1 da je za prikaz tačke na video displeju u režimu rada 12H neophodno kreirati makro naredbu. Jedan takav makro nazivan _DOT prikazan je u Primeru 13-2 (Pridružiti ovaj makro vašem markou MACS.INC). Ovaj makro koristi tri parametra: COLOR, X i Y. Parametar X odredjuje poziciju duž ekrana (od 0 do 639), Y je vertikalna pozicija (od 0 do 479), a COLOR (od 0 do 15) definiše boju tačke.

Primer 13-2

```
_DOT MACRO COLOR,X,Y        ;makro za crtanje tacke (rezim 12H)
   PUSH  DS                 ;čuvanje registara
   PUSH  CX
   PUSH  DX
   PUSH  SI
   MOV   AL,COLOR           ;postavljanje boje
   PUSH  AX
   MOV   AX,0A00H           ;adresiranje video segmenta
   MOV   DS,AX
   MOV   AX,Y
   MOV   SI,640
   MUL   SI                 ;odredjivanje pozicije Y
   ADD   AX,X               ;odredjivanje XY pozicije
   ADC   DX,0
   PUSH  AX
   AND   AL,7               ;dobavljanje vrednosti za bit-maskiranje
   MOV   CL,AL
   POP   AX
   MOV   SI,8               ;deljenje sa 8 da bi odredili poziciju bajta
   DIV   SI
   MOV   SI,AX              ;zapamtiti adresu tacke u SI
   MOV   AL,[SI]            ;čitanje video lokacije
   MOV   AH,80H             ;postavljanje bita na krajnje levoj poziciji na ekranu
   SHR   AH,CL
   MOV   DX,3CEH                ;selekcija registra adrese VGA
   MOV   AL,8                   ;indeksa 8
   OUT   DX,AX                  ;i izlazne bit-maske
   MOV   DX,3C4H                ;adresira registra sekvence
   MOV   AX,0F02H               ;selekcija svih boja
   OUT   DX,AX
   INC   DX
   MOV   BYTE PTR [SI],0        ;upis nule radi postavljanja crne boje
   POP   AX                     ;pribavljanje boje
   OUT   DX,AL                  ;postavljanje boje
```

```
   MOV   BYTE PTR [SI],0FFH    ;upis boje
   POP   SI                    ;obnavljanje sadržaja registara
   POP   DX
   POP   CX
   POP   DS
ENDM
```

KORAK 1: Zapamtiti makro _DOT u makro fajlu MACS.INC. Kreirati program koji koristi makro _DOT radi prikaza tačaka na sledećim pozicijama na displeju: Nezasićeno-plavu duž početnih triju linija video displeja, Nezasićeno-crvenu duž zadnje tri linije video displeja, jednu žutu tačku na poziciji X=320 i Y=220. Uneti, asemblirati i izvršiti program.

Prikaz blokova boja u režimu rada 12H

Iz razloga što makro _DOT u datom trenutku crta samo jednu tačku, proces crtanja celog ekrana ili većih delova ekrana može trajati veoma dugo. Da bi se ubrzao proces prikazivanja velikih obojenih oblasti, ekran se može podeliti na blokove čime se ubrzava proces bojenja velikih površina. Pretpostavimo da se 640 tačaka duž ekrana deli na 80 pozicija, pri čemu svaka pozicija odgovara jednoj bajt poziciji u video memoriji. Pretpostavimo, takodje, da su 480 linija ekrana podelejne u 53 linije sa po 9 analizatorskih linija u grupi i sa po 3 neiskorišćene linije na dnu ekrana (to odgovara prikazu 53 linije tekstualnog sadržaja). Za velike oblasti, ekran odgovara prikazu od 80 x 53 blokova.
Makro prikazan u Primeru 13-3 pod nazivom _BLOCK deli ekran na blokove dimenzije 80x53. Kao i kod makroa _DOT, _BLOCK koristi tri parametra: COLOR (0-15), X (0-79) i Y (0-52).

Primer 13-3

```
_BLOCK  MACRO  COLOR,X,Y          ;prikaz bloka
      LOCAL BI
      PUSH  DS                    ;pamćenje stanja registara
      PUSH  CX
      PUSH  DX
      PUSH  SI
      MOV   AL,COLOR              ;pribavljanje boje
      PUSH  AX
      MOV   AX,0A000H             ;adresiranje segmenta video memorije
      MOV   DS,AX
      MOV   AX,Y                  ;odredjivanje linije kod raster prikaza
      MOV   SI,80*9
      MUL   SI
      ADD   AX,X                  ;bajt ofseta adrese
      MOV   SI,AX
      MOV   AX,0FF08H
      MOV   DX,3CEH
      OUT   DX,AX                 ;dozvola rada svim bitovima
      MOV   DX,3C4H
      MOV   CX,9                  ;punjenje brojača
BI:   MOV   AL,[SI]
      MOV   AX,0F02H
      OUT   DX,AX
      INC   DX
```

```
        MOV   BYTE PTR [SI],0
        POP   AX
        PUSH  AX
        OUT   DX,AL
        MOV   BYTE PTR [SI],0FFH
        DEC   DX
        ADD   SI,80
        LOOP  BI
        POP   AX                        ;obnavljanje stanja registara
        POP   SI
        POP   DX
        POP   CX
        POP   DS
ENDM
```

KORAK 2: Koristeći makro _BLOCK kreiran u Primeru 13-3, napisati program koji prikazuje ekranu u boji nezasićeno-purpurnu boju. Uočimo brzinu promene boje na ekranu pri izvršavanju programa.

KORAK 3: Kreirati program koji koristi makro _BLOCK za prikaz belog pravougaonika duž dna ekrana, širine dva bloka. U ostatku ekrana prikazati površinu obojenu cijan bojom.

Prikaz teksta u režimu rada 12H

Izlaz iz programa na video displeju sličan je kod većine ekrana koji su u komercijalnoj upotrebi. Da bi se prikazao tekst na ekranu, potrebno je koristiti funkcijske DOS pozive koji uvek prikazuju bela slova na crnoj pozadini. Srećom, video BIOS ROM sadrži nekoliko različitih skupova karaktera koji se mogu koristiti za prikaz podataka na grafičkom displeju. Ako koristimo 80 karaktera, u 53 linija prikaza generisanih od strane makroa _BLOCK, možemo selektovati skup karaktera veličine 8 x 8 iz BIOS ROM-a. Karakter set (skup) 8 x 8 je obima jednog bajta i visine 8 linija. Ekran je podeljen u blokove visine 9 linija, što znači da postoji po jedna blanko linija radi razdvajanja karaktera koji se prikazuju u susednim redovima.
Da bi se odredila lokacija skupa karaktera 8 x 8 iz ROM-a potrebno je pozvati BIOS-ov funkcijski INT 10H sa atributom 1130H predat u registru AX. Ako koristimo sekvencu koja sledi, početna adresa skupa karaktera biće vraćena u registarskom parou ES:BP:

```
MOV AX,1130H        ;čitanje informacije iz ROM-a
MOV    BH,3             ;povratna adresa seta karaktera veličine 8 x 8
INT    10H
```

Početne adrese ostalih skupova karaktera prikazane su u tabeli 13-1.

Tabela 13-1 Početne adrese skupova karaktera u ROM-u

BH	Skup karaktera (znakova)
02H	8 (širina) x 14 (visina)
03H	8 x 8 (standardni)
04H	8 x 8 prošireni skup karaktera (ASCII 80H-FFH)
05H	9 x 14 alternativni
06H	8 x 16
07H	8 x 16 alternativni

Kada je adresa skupa karaktera poznata, bit oblici koji predstavlaju karakter skup koriste se za prikaz bilo kog karaktera na grafičkom displeju koristeći bilo koju boju. Primer 13-4 pokazuje

makro koji prikazuje bilo koji standardni ASCII karakter koji je smešten u AL na video displeju rezolucije 640 x 480 sa 16 boja. Makro pretpostavlja da je displej organizovan sa 80 karaktera po liniji i 53 linija po prikazu. Naglasimo da ovaj makro ne menja boju pozadine i ne prikazuje poziciju kursora. Parametari koji se prenosi makrou su: boja karaktera (0 – 15), broj linije u slici (vrsta) i pozicija karaktera u iniji.

Primer 13-4

```
_CHAR MACRO COLOR,LINE,ROW        ;prikaz karaktera
LOCAL       CL
PUSH        DS                    ;sačuvaj stanje registra
PUSH        ES
PUSH        SI
PUSH        BP
PUSH        BX
PUSH        EX
PUSH        DX
MOV         BL,COLOR              ;sačuvati boju
PUSH        BX
MOV         BX,ROW
PUSH        BX                    ;sačuvati vrste
MOV         BX,LINE               ;sačuvati liniju
PUSH        BX
PUSH        AX                    ;sačuvati ASCII kôda
MOV         AX,1130H              ;dobaviti skup 8 x 8
MOV         BH,3
INT         10H
POP         AX                    ;dobaviti ASCII kôda
      XOR   AH,AH
      SHL   AX,1                  ;množenje sa 8
      SHL   AX,1
      SHL   AX,1
      ADD   BP,AX                 ;indek karaktera u ROM-u
      MOV   AX,0A000H             ;adresa video segmenta
      MOV   DS,AX
      POP   AX                    ;dobavljenje broja linije
      MOV   SI,8019
      MUL   SI
      MOV   SI,AX                 ;generisanje raster adrese
      POP   AX                    ;dobavljenje broja vrste
      ADD   SI,AX
      MOV   CX,8                  ;postavljanje brojača na 8 vrsta
Cl:   MOV   DX,3CEH               ;adresa bit-mask registara
      MOV   AI,8                  ;postavljanje indeksa 8
      MOV   Alf,ES:[BP]           ;dobavljanje karaktera u vrsti
      INC   BP                    ;pokazivanje na narednu vrstu
      OUT   DX,AX
      MOV   DX,3C4H               ;adresa map mask registera
      MOV   AX,0F02H
```

```
        OUT   DX,AX                  ;izbor svih ravni za prikaz
        INC   DX
        MOV   AL,[SI]                ;čitanje podatka
        MOV   BYTE PTR [SI],0        ;upis blanko
        POP   AX                     ;dobavlanje boje
        PUSH  AX
        OUT   DX,AL                  ;upis boje
        MOV   BYTE PTR [SI],0FFH
        ADD   SI,80                  ;adresa naredne vrste u raster prikazu
        LOOP  C1                     ;ponovi 8 puta
        POP   AX                     ;obnavljanje stanja registara
        POP   DX
        POP   CX
        POP   BX
        POP   BP
        POP   SI
        POP   ES
        POP   DS
ENDM
```

Uočiti kako se koristi registar za maskiranje bitova da bise logičkom operacijom manipulisalo nad stanjem bitova ASCII kodiranog karaktera. Na ovaj način je omogućeno da pojedinačnom karakteru lociran na odredjenoj poziciji ekrana može menjati boje. Karakter se pojavljuje bez promene boje pozadine.

KORAK4: Koristeći program napisan u koraku 3, dodati sekciju koja koristi _CHAR za prikaz Vašeg imena na sredini početne linije u slovima žute boje (0EH), kao i reči *Main Menu* crnom bojom na sredini druge linije. Važno je kreirati proceduru koja koristi makro _CHAR za prikaz niza karaktera. Na ovaj način će se značajno smenjiti posao potreban da se razvije ovaj deo programa.

Pitanja

1. Koliki memorijski prostor se koristi za prikaz u grafičkom režimu sa rezolucijom 640 x 480 tačaka i 16 boja?
2. Šta je to bitska ravan i kako se ona implementia kod video displeja rezolucije 640 x 480 i 16 boja?
3. Kako se selektuje video režim rada 12H?
4. Koja je namena registra za maskiranje bitova?
5. Kako se adresira registara za maskiranje bitova?
6. Koja je namena registra mape maskiranja?
7. Kako se adresira registar mape maskiranja?
8. Boja se mora najpre posatviti na _____ pre nego se upiše nova boja.
9. Kreirati program koji koristi makro _DOT za crtanje neprekidne crvene linije debljine dve linije u raster prikazu na dnu ekrana kod video prikaza.
10. Kreirati program koji koristi makro _BLOCK za bojenje oblasti širine dve i četiri karakter-linije na dnu i vrhu ekrana, respektivno. Oblasti su obojene cijan bojom, a ostatak ekrana belo.

LABORATORIJSKA VEŽBA 14

Korišćenje miša

Uvod

Jedan od novijih U/I uredjaja koji se pojavio na tržištu računara je miš. Miš je U/I uredjaj koji u odredjenim situacijama zamenjuje korišćenje tastature kod grafičko i tekstualno orjentisanih programa. *Trackball* je jedna od varijanti miša koja se ponaša skoro identično kao i miš a razlika se sastoji u tome što je *trackball* fiksiran na jednom mestu a miš je pokretan.
Ova Laboratorijska vežba pokazuje kako se manipulisanje mišom ugradjuje u razne aplikacije koje se beziraju na pozivanju makroa (ovi makroi dozvoljavaju rad miša), a takodje, i identifikuju poziciju miša na video displeju.

Predmet rada

1. Napisati makro koji detektuje prisustvo miša i omogućava njegovo korišćenje kod raznih aplikacija.
2. Napisati makro koji prati poziciju miša i ukazuje da li je njegova dirka pritisnuta.
3. Koricenje miša kod jednostavnih programa.

Postupak rada

U ovoj Laboratorijskoj vežbi testira se prisustvo miša, čita njegova pozicija i testiraju tasteri na mišu. Jedanput kreirana, ova Laboratorijska vežba dozvoljava pozivanje programa za rad miša od strane bilo kog drugog programa.

Testiranje prisustva miša?

Mišu se pristupa izvršenjem instrukcije INT 33H. Pre nego što se miš može koristi od strane nekog programa potrebno je proveriti da li je miš pravilno provezan i da li je u funkciji. Da bi detektovli prisustvo miša i osposobili ga za rad neophodno je preduzeti sledeće korake:

1. Testirati prekidni vektor 33H da bi se ustanovilo da li on sadrži broj koji je različit od nule. Nula se uglavnom upisuje na lokaciji vektor broja u vektorskoj tabeli u slučaju da pokrtetački program miša (drajver miša) nije instaliran u sistem.
2. Ako vektor nije nula, proveriti da li vektor pokazuje na instrukciju IRET čime se ustanovljava da je vektor neiskorišćen.
3. Ako vektor nije nula i ako ne pokazuje na instrukciju IRET (CFH), smestiti nulu u registar AX a zatim izvršiti funkcijski poziv, INT 33H. Za slučaj da je nula vraćena kao vrednost u registar AX, tada nema miša, miš nije instaliran, u suprotnom miš je prisutan.

Makro je idealan za proveru prisustva miša. Takav jedan makro (_MP) se pojavljuje u Primeru 14-1. Makro _MP detektuje prisustvo miša i izvršava se postavljanjem bita prenosa C = 0 u slučaju kada je miš prisutan. Ako miš nije prisutan, prikazuje se poruka i postavlja bit prenosa C = 1.

Primer 14-1

```
_MP   MACRO                    ;da li je miš prisutan?
```

```
   LOCAL  M1,M2,M3
   PUSH   ES                      ;pamćenje registra ES u magacinu
   MOV    AX,3533H                ;čitanje vektora 33H
   INT    21H
   MOV    AX,ES
   OR     AX,BX                   ;test da li je ES:BX = 0
   JZ     M2                      ;ako nije završi makro
   CMP    BYTE PTR [BX],0CFH
   JZ     M2                      ;ako nije završi makro
   MOV    AX,0                    ;aktiviraj miš
   INT    33H
   OR     AX,AX
   JZ     M2                      ;miš nije prisutan
   CLC                            ;miš je prisutan, prenos = 0
   JMP    M3
M1  DB    13,10,'*** MIS NIJE PRISUTAN ***$'
M2:
   PUSH   DS
   MOV    AX,CS
   MOV    DS,AX
   _STRING  M1                    ;prikazati da nema miša
   POP    DS
   STC
M3:
   POP    ES
ENDM
```

KORAK 1: Kreirati program koji testira prisustvo miša (koristi makro _MP) i zatim se vraća u DOS (koristiti makro _EXIT). Drajveru miša se može dozvoliti ili zabraniti rad iz DOS-ove komandne linije tako što se otkuca ime drajvera miša nakon čega sledi ON ili OFF kako bi se rad miša dozvolio ili zabranio. (Proveriti prisustvo miša i korektnu sintaksu vašeg drajvera miša tako što ćete otkucati MOUSE /? da bi ustanovili da li je miš ON ili OFF). Pokrenuti program kreiran u ovom koraku aktiviranjem ili deaktiviranjem miša. Kada je miš deaktiviran pojaviće se poruka *** MIS NIJE PRISUTAN ***

Aktiviranje miša

Analiziraćemo sada program koji testira prisustvo miša. Samim tim što je miš prisutan to ne znači da se on može i koristiti u aplikacijama. Program za testiranje miša ima ON i OFF funkciju a aktivira kursor miša ili strelicu. Kursor miša ili strelica se aktivira pozivom INT 33H, funkcije 01H a deaktivira pozivom INT 33H, funkcija 02H. Ni jedna od ovih funkcija ne vraća informaciju pozivnoj proceduri. Primer 14-2 prikazuje listing dva makroa koji aktiviraju (_MON) ili deaktiviraju kursor miša (_MOFF).

Zbog čega je potrebno aktiviranje ili deaktiviranje kursora miša? Rad kursora miša je obično zabranjen sve se dok drajver miša ne akrivira. Makro _MON prikazuje kursor miša. Ako je kursor miša aktiviran i podaci se prikazuju na video displeju, računar prikazuje kopiju strelice (pokazivača) miša na ekranu. Prikazati 4 objekta na ekranu i videti prikaz 4 strelica miša. Da bi izbegao ovaj problem, uvek deaktivirati strelicu miša pre ažuriranja video informacije na ekranu a zatim ga je aktivirati pre nego što se ažuriranje završi.

Primer 14-2

```
_MON  MACRO            ;aktiviranje kursora miša
      MOV   AX,1
      INT   33H
      ENDM

_MOFF MACRO            ;deaktiviranje kursora miša
      MOV   AX,2
      INT   33H
      ENDM
```

KORAK 2: Modifikovati program napisan u Primeru 14-2 tako da se kursor miša aktivira (_MON) pre povratka na DOS. Strelica miša je sada u centru ekrana. Ukoliko se miš pomera, i strelica će se pomerati ali neće biti dostupna informacija o mišu.

Praćenje aktivnosti tastera miša

Da bi se u aplikacijama koristio miš, mora se pratiti pozicija i stanje tastera miša. Izvršenjem poziva INT 33H funkcijki broj 5 (odnosi se na poziv miša), vraća informaciju o poziciji tastera koji se odnosi na zadnji pritisak na taster kada se pozove parametri koji se prenose nalaze se u registrima AX i BX a oni su AX= 5 i BX = dirka koja se testira. Ako je registar BX = 0 testira se levi taster, ako je registar BX = 1 testira se desni taster. Ako postoji miš sa tri tastera pri BX = 4 testira se srednje taster. Pri povratku iz funkcije 5, u registru AX se čuva AX = stanje tastera. Vrednost u registru AX pokazuje da li je taster tekuće pritisnut. Bit pozicija 0 je postavljena na 1 ako je pritisnut levi taster, bit pozicija 1 je postavljena na 1 ako je pritisnut desni taster i bit pozicija 2 je postavljena na 1 kada je pritisnut srednji taster. Registar BX sadrži broj koliko puta je bio pritisnut taj taster od trenutka kada je zadnji put ta funkcija bila pozvana. Registri CX (hotizontalni pokazivač) i DX (vertikalni pokazivač) čuvaju poziciju strelice miša u trenutku zadnjeg pritiska tastera. Videti Primer 14-3 makroa (_MBUT) koji proverava da li je pritisnut taster miša.

Primer 14-3

```
_MBUT MACRO NUM        ;čitanje tastera
      MOV   AX,5       ;NUM = 0 za levi taster
      MOV   BX,NUM     ;NUM = 1 za desni taster
      INT   33H        ;NUM = 4 za srednji taster
      ENDM
```

KORAK 3: Kreirati program koji prikazuje reč LEFT ako je pritisnut levi taster a RIGHT ako je pritisnut desni taster. Izaći iz programa ako registar AX pokazuje da su istovremeno pritisnuti i levi i desni taster. Deaktivirati pokazivač miša pre prikazivanja LEFT ili RIGHT a aktivirati ga ponovo nakon toga.

Praćenje položaja miša

Informacija o horizontalom i verikalnom položaju miša se prati u registrima CX (horizontalni) i DX (vertikalni), pri čemu se počinje od nule. To znači da ako je strelica miša u gornjem levom položaju tada se u registre CX i DX vraća (postavlja) vrednost nula. Ako ste u tekstualnom

režimu rada 80 x 25 (mode 3, standardni DOS mod), vrednost koja se vraća u registar CX se nalazi u opsegu od 0 do 632 i vrednosti koja se vraća u registar DX se nalazi u opsegu od 0 do 192. Koraci su u inkerementima od po 8. To znači da kada je miš lociran u liniji 1 na poziciji karaktera 3 tada se u registre CX i DX vraća sledeća informacija CX = 8 i DX = 24. Opseg ovih vrednosti kao i iznos koraka se menjaju a zavise od režima rada i obima (veličine) karaktera koji se prikazuju na video displeju.
Funkcija 5 jedino vraća poziciju kursora koja se odnosi na zadnji pritisak tastera. Ponekad je potrebno znati poziciju strelice miša u realnom vremenu. Funkcija miša broj 3 čita u realnom vremenu status miša u pokretu. Primer 14-4 pokazuje status miša koji se čita u pokretu. Na kraju makroa _MFLY, registar BX čuva status tastera (bit = 0 levi, bit = 1 desni i bit 2 = srednji), registar CX sadrži horizontalnu poziciju i DX vertikalnu poziciju strelice miša.

Primer 14-4

```
_MFLY  MACRO          ;čitanje statusa miša
       MOV  AX,3
       INT  33H
       ENDM
```

KORAK 4: Kreirati program koristeći makro _MFLY koji detektuje kada je kursor miša pozicioniran na koordinatama 40,40. Ako je u tom trenutku pritisnut levi taster poslati zujalici zvučni signal čiji je ASCII kôd (07H). Ako su istovremeno pritisnuti i levi i desni taster miša izaći iz programa.

Miš u grafičkom režimu rada

Miš takodje može da funkcioniše i u grafičkom režimu rada. Primer 14-5 prikazuje listing programa koji prebacuje video adapter u režimu rada 12H (640 x 480, 16 boja) i prikazuje strelicu miša. Prikaz informacije na displeju se vrši sve do trenutka dok nisu instovremeno pritisnuti i levi i desni taster, tj. kada se gde program vraća video u mod 3 i prelazi na DOS.

Primer 14-5

```
CODE   SEGMENT  'code'          ;početak kôdnog segmenta
   ASSUME CS:CODE              ;usvajamo da CS adresira kôdni segm. CODE
   INCLUDE  MACS.INC
   MAIN  PROC  FAR             ;početak procedure MAIN
      MOV   AX,12H             ;prelaz u mod 12H
      INT   10H
      _MP                      ;test za prisustvo miša
      JC   MAIN2               ;granaj se miš nije prisutan
      _MON                     ;aktiviranje dozvole rada za strelicu miša
MAIN1:
      _MFLY                    ;čitanje statusa miša
CMP   BX,3                ;testiranje da li je istovremeno pritisnut levi i
;desni taster miša
      JNE   MAIN1          ;ako nije istovremeno pritisnut levi i desni taster miša
MAIN2:
```

```
        _MOFF                   ;deaktiviranje strelice miša
        MOV    AX,3             ;prelaz u mod 3
        INT    10H
        _EXIT                   ;povtatak u DOS
MAIN    ENDP                  ;kraj glavne procedure
CODE    ENDS                    ;kraj kodnog segmenta
    END    MAIN                 ;kraj programa
```

KORAK 5: Pozvati program iz Primera 14-5 i izvršti ga. Nakon toga uporediti poziciju strelice miša generisane u grafičkom režimu rada (mod 12H) sa pozicijom strelice miša generisane u ranijim programima.
Naredni deo ove Laboratorijske vežbe koristi BIOS INT 10H za pomeranje pozicije kursora miša na ekranu. Kursor se upravlja upisom vrednosti 2 u registar AH. Broj linije se smešta u registar DH a pozicija karaktera se čuva u registru DL. Registar BH mora biti postavljen na logičkoj nuli kako bi se selektovala stranica 0 koja po definiciji predstavlja početnu stranicu video prikaza. Nakon što su registri napunjeni, izvršava se instrukcija INT 10H. Primer 14-6 prikazuje niz instrukcija koje se koriste za pomeranje kursora na liniju 0 (gornja linija) i karakter poziciju 0 (krajnji levi karakter).

Primer 14-6

```
    MOV    AH,2          ;selektovanje funkcije 2
    MOV    BH,0          ;selektovanje stranice 0
    MOV    DX,0          ;gornji levi ugao
    INT    10H
```

KORAK 6: Koristeći poziciju kursora iz Primeru 14-6, kreirati program koji postavlja displej u video režim rada 12H a zatim prikazuje vertikalnu i horizontalnu poziciju strelice miša na ekranu. Prikazati vertikalnu poziciju linije 0 na levoj margini i horizontalnu poziciju linije 1 na levoj margini. Koristiti makro _NUM (prikazati AX u decimalnoj notaciji) da bi prikazao ove položaje. Napomenimo da zbog brzine kojom se vrši prikaz, svrsishodnije je prikazivati samo one pozicije kod kojih je došlo do promene položaja kao što se ostvaruje čitanjem od strane makroa _MFLY. Ako su istovremeno pritisnuti levi i desni taster miša tada se izlazi iz programa.

Pitanja

1. Koja INT instrukcija se koristi da bi se pristupilo drajveru miša?
2. Opisati kako se ispituje drajver miša da bi se ispitalo njegovo prisustvo?
3. Koja je svrha makroa _MON i _MOFF kreiranih u ovoj Laboratorijskoj vežbi?
4. Zašto strelica miša mora biti isključena pre pokazivanja podataka na video ekranu?
5. Koja je razlika izmedju drajverske (pokretačke) funkcije miša 3 i 5?
6. Kada se testira status miša koji bit registra BX (funkcija 3) ili registra AX (funkcija 5) odgovara levom tasteru miša?
7. Ako je ugradjen srednji taster miša, koji bit pokazuje da je on pritisnut?
8. Vrednosti koje se nalaze u registrima CX i DX na koji tip informacije ukazuju pozivnom programu kada se pozove pokretački program miša pomoću funkcije 3?
9. Koja se funkcija miša koristi da testira dvostruki pritisak tastera?

10. Kreirati makro _MMID koji testira drajver miša da bi se odredilo da li je pritisnut srednji taster miša. Završiti ovaj makro sa prenosom = 1 ako je taster pritisnut i prenosom = 0 kada taster nije pritisnut.

LABORATORIJSKA VEŽBA 15

Obrada prekida

Uvod

Struktura prekida kod personalnih računara ima dominantnu ulogu na način izvršenja ulazno izlaznih zadataka. Da bi se korektno obavila ulazno izlazna aktivnost kod PC mašina, za programera je neophodno da dobro razume strukturu prekida. Ova Laboratorijska vežba ukazuje na tehnike koje se koriste kod prihvatanja zahteva za prekid i načina povezivanja (njihov odnos) sa prekidnim vektorom. Standardni U/I uredjaji kod PC mašina koji iniciraju uahteve za prekid su tajmer i tastatura. Stanje tajmera testira se preko zahteva za prekid na neke vremenske dogadjaje koji treba da se dese dok se sa druge strane tastatura testira radi dešifriranja koda pritisnute dirke od PC mašine. Obrada prekida kod tastature se koristi da bi se instalirale urgentne dirke (Ctrl +, Alt +) ili da bi se promenila funkcija dirke.

Predmet rada

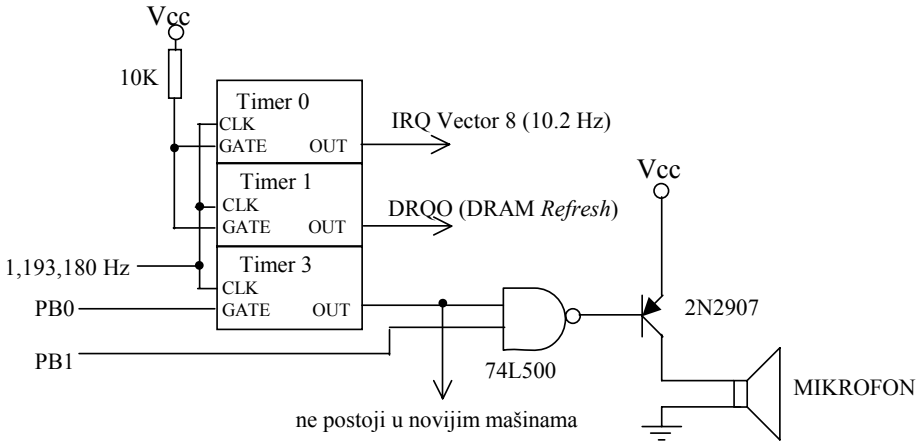
1. Korišćenje tajmer-prekidnog vektora broj 8.
2. Korišćenje tajmera kao generatora audio signala.
3. Korišćenje tajmera i miša radi automatskog testiranja pozicije miša sa učestanošću od 18 puta u sekundi.

Postupak rada

Prvi deo ove Laboratorijske vežbe se odnosi na način korišćenja i programiranja tajmera. Tajmer je programabilni brojač po modulu *n* koji deli fiksnu ulaznu frekfenciju bilo kojim celim brojem izmedju 2 i 64k. Kao rezultat se dobija signal koji se predaje zvučniku.

Tajmer 8253/8254

Slika 15-1 prikazuje internu strukturu tajmera 8253/8254. Tajmer čine tri nezavisne celine (dela) koje se koriste za praćenje različitih vremenskih dogadjaja kod personalnih računara. Treba napomenuti da korisnik (student) ne sme programirati ili menjati režime rada tajmera 0 i 1. Tajmer 0 generiše instrukciju INT 8 18.2 puta u sekundi. Ovaj prekid se koristi kao časovnik u okviru računara. Menjajući ovo vreme kao efekat imaćemo da sistemski sat ne radi korektno. Tajmer 1 se koristi da bi se kontrolisalo vreme izmedju dva ciklusa osvežavanja dinamičke RAM memorije. Ako se ovaj preiod menja funkcionisanje memorije se može poremetiti (osvežavanje se neće dešavati na vreme). Tajmer 2 se koristi za generisanje signala promenljive frekvence kojim se pobudjuje zvučnik (*buzzer*).
Svaki tajmer (brojač u okviru čipa) ima po 2 ulaza i 1 izlaz. Ulazi su CLK (takt) i GATE. Signal personalnog računara od 1,193,180 Hz predstavlja taktni ulaz CLK. Ovaj signal je generisan u računaru a dobija se tako što se njegova taktna pobuda od 4.77272 MHz deli sa faktorom 4. Taktini ulaz CLK tajmera je isti za sve verzije PC računara uključujući i one tipa Pentium. Ulaz GATE kontroliše da li tajmer broji ili ne. Ulazi GATE tajmera 0 i 1 su uvek priključeni na logičku jedinicu tako da oni uvek broje. Tajmer 2 se kontroliše od strane drugiog U/I uredjaja, unutar PC-a, koji se naziva PIA (*parallel interface adapter*). Naglasimo takodje da PIA mora biti programirana kako bi upravljala radom tajmera 2. Tajmer i PIA su u suštini programibilne periferije koje se ekvivalentno ponašaju kao U/I uredjaji. Tajmeru su dodeljene U/I lokacije 40H-43H, dok su PIA dodeljene U/I port lokacije 60H-63H.



Slika 15-1. Kolo zvučnika povezano na tajmer kod PC mašine

(Čipu PIA 8255 su dodeljene U/I port adrese 60H – 63H a čipu Tajmer/brojač dogadjaja 8253 U/I port adrese 40H – 43H)

Iz razloga što jedino smemo programirati tajmer 2, koristićemo U/I port adresu 42H da bi pristupili brojačkom regisrtu tajmera 2 i U/I port adresu 43H da bi pristupili upravljačkom registru tajmera. Brojački registar se puni na broj koji odredjuje faktor deljenja tajmera. Na primer, ako je brojač tajmera 2 programiran za deljenje sa 100, izlaz tajmera će biti 1,193,180 Hz / 100 ili 119,318 Hz. Brojač tajmera je 16-bitni tako da faktor deljenja može biti bilo koji broj izmedju 2 i 65,535. U/I port adresa 43H koja je dodeljena upravljačom registaru tajmera. Shodno tome sadržaj registra AL mora biti odredjen u saglasnosti sa formatom upravljačke reči definisane na Slici 15-2. Izvršenjem instrukcije OUT 43H,AL programira se tajmer 2 za režim rada 3.

Da bi programirali tajmer 2 (ne treba pokušavati da se programira bilo koji drugi tajmer) treba postaviti 10 na bit pozicije SC1 i SC0 upravlčačkog registra kako bi se selektovao tajmer 2. Pored toga treba postaviti 11 na bit pozivije RW1 i RW0 kako bi programirali svih 16 bitova tajmera 2. Većina brojača radi binarno tako da bit BCD treba postaviti na 0. Bitovi M2-M0 definišu način rada tajmera. Dozvoljeni načini rada tj. modovi su od 000 do 101. Pogledati Sliku 15-3 za postavljanje režima rada brojača.

Kod većine slučajeva režim rada 3 (generisanje neprekidne povorke pravougaonih impulsa) se obično koristi za programiranje tajmera 2 za potrebe pobude zvučnika kod PC mašine. Ostali režimi rada se mogu takodje koristiti, ali se u tom slučaju ne može generisti audio signal. Ako se generiše sintetizovana muzika ili govor, može se koristiti režim rada 1 radi generisanja modulisanog signala kojim se vrši pobuda zvučnika PC mašine. U ovoj Laboratorijkoj vežbi koristićemo isključivo režim rada 3.

Pretpostavimo da je audio signal frekfencije 800 Hz generisan za pobudu zvučnika. Ovaj zadatak se izvršava tako što se prvo odredjuje faktor deljenja a nakon toga se programira upravljački registr čija je port adresa (43H) i na kraju brojački registr čija je port adreasa (42H). Primer 15-1 pokazuje kako se programira tajmer 2 radi generisanja tonskog signala frekfence 800Hz.

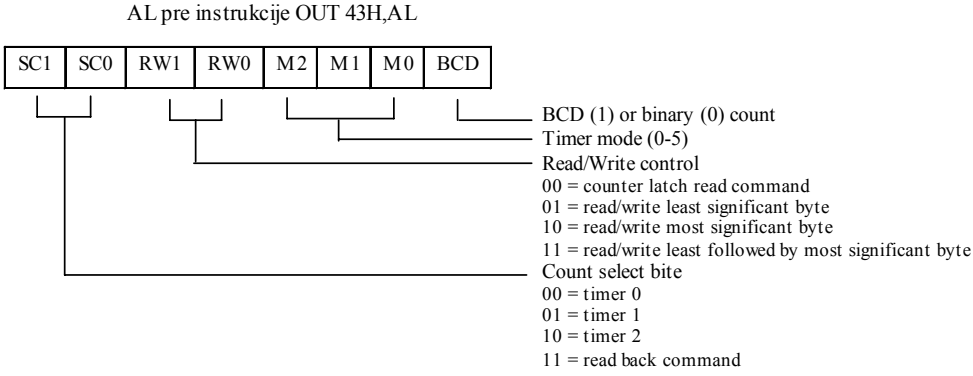
Primer 15-1

```
MOV    AL,0B6H          ;programiranje komandnog registra
OUT    43H,AL           ;tajmer 2, režim rada 3
```

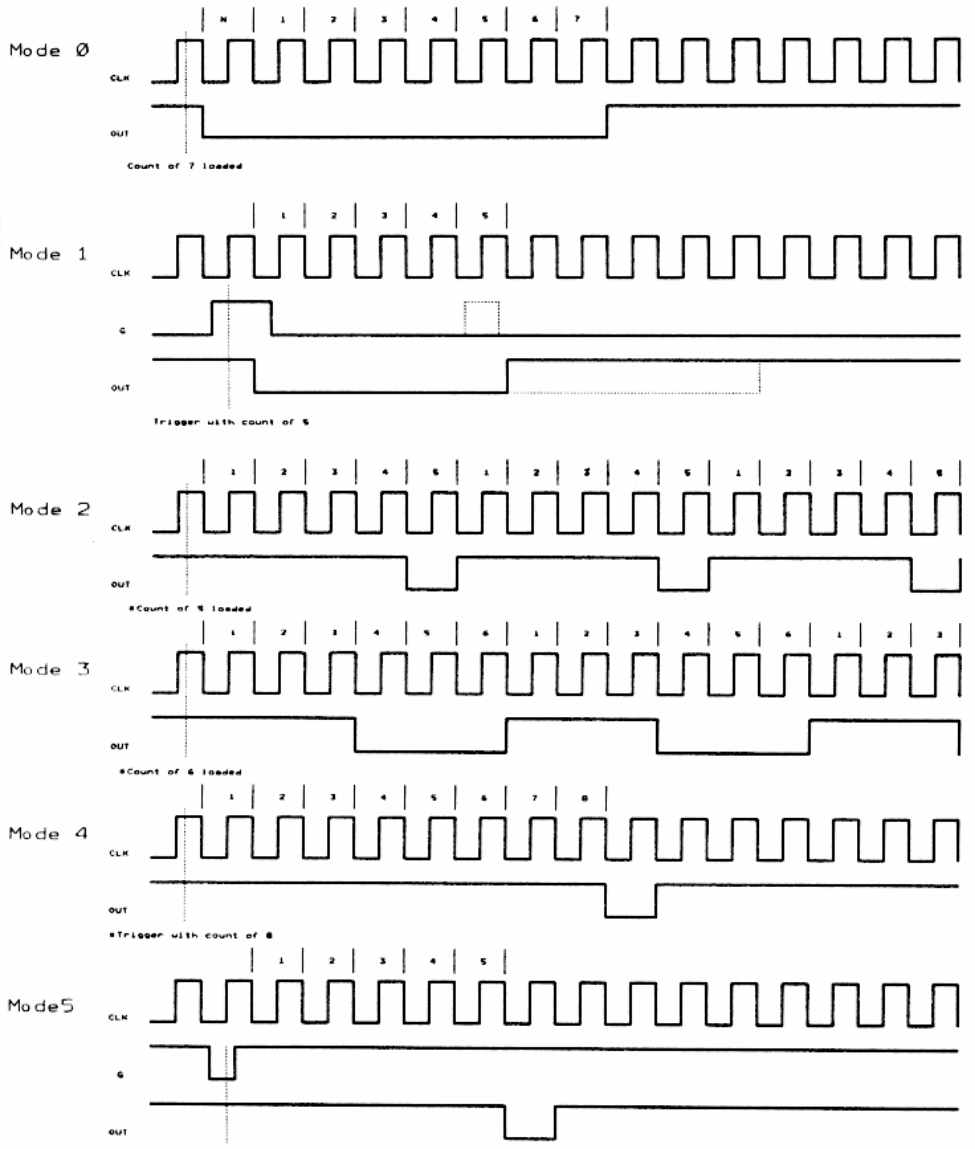
```
MOV   DX,12H          ;punjenje DX:AX sa 1,193,180
MOV   AX,34DCH
MOV   BX,800          ;punjenje BX sa 800
DIV   BX              ;odredjivanje broja deljenja

OUT   42H,AL          ;programiran brojač za tajmer 2
MOV   AL,AH
OUT   42H,AL
```

Uočimo kako se decimalna vrednost 1,193,180 smešta u registarski par DX:AX (ekvivalentna heksadecimalna vrednost je 1234DCH). Ova vrednost se deli sa 800 da bi se dobio broj koji će predstavljati osnovu deljenja brojača. Dobijena 16 bitna vrednost se zatim upisuje na adresi porta 42H iz dva pokušaja, prvo se upisuje sadržaj registra AL a zatim sadržaj registra AH kako bi se odredila početna vrednost 16-binog brojača.
Ovom sekvencom programira se brojač ali se ne generiše audio signal na zvučniku. Zvučnik je povezan na izlaz tajmera preko logičkog NI kola dok je GATE ulaz tajmera 2 povezan na PIA na portu B bit PB0. Da bi se dozvolio rad zvučnika logička jedinica se dovodi na ulaz GATE Tajmera 2 a takodje i na ulaz drugog NI kola (PB1 = 1) ovo se izvršava programiranjem PIA kako bi na oba izlazna pina porta B (PB0 = PB1 = 1). Adresa porta B čipa PIA 8255 je 61H. Primer 15-2 pokazuje sekvencu instrukcija koja postavlja izlaze porta B PB0 i PB1 na logičke jedinice kako bi se aktivirao zvučnik. Naglasimo da je neophodno prvo pročitati stanje porta B kako bi se modifikovalo stanje na izlaznim pinovima PB0 i PB1. Korisnik ne sme upisivati direktno vrednost 3 u port B jer se ostali pinovi porta B koriste u okviru PS mašine za druge potrebe.



Slika 15-2. Sadržaj upravljačkog registra tajmera (adresa upravljačkog porta u ovom primeru je 43H)



Slika 15-3. Načini rada kod programabilnog tajmera/brojača dogadjaja
Napomena: Mode = režim rada

Primer 15-2

```
IN   AL,61H          ;čitanje PIA, port B
OR   AL,3            ;postavljanje na 11 dva LS bita porta B
OUT  61H,AL          ;upis u PIA, port B
```

Generisanje tonskog signala za pobudu zvučnika

Primer 15-3 pokazuje makro _SPON koji se koristi za programiranje tajmera radi generisanja tonskog signala i pobudjivanje zvučnika. Primer takodje prikazuje makro _SPOFF koji zabranjuje rad zvučnika.

Primer 15-3

```
_SPON MACRO TONE          ;pobudjivanje zvučnika
   PUSH  DX
   PUSH  BX
   PUSH  AX
   MOV   AL,0B6H          ;odredjivanje vrednosti na koju treba postaviti brojač
   OUT   43H,AL
   MOV   BX,TONE          ;pronalaženje brojača
   MOV   DX,12H
   MOV   AX,34DCH
   DIV   BX
   OUT   42H,AL           ;programiranje tajmera
   MOV   AL,AH
   OUT   42H,AL
   IN    AL,61H           ;dozvola rada zvučnika
   OR    AL,3
   OUT   61H,AL
   POP   AX
   POP   BX
   POP   DX
   ENDM

_SPOFF   MACRO
   PUSH  AH               ;zabrana rada zvučnika
   IN    AL,61H
   AND   AL,0FCH
   OUT   61H,AL
   POP   AX
   ENDM
```

KORAK 1: Koristeći makroe _SPON i _SPOFF, kreirani u Primeru 15-3, napisati program koji pobudjuje zvučnik tonskom frekfencijom od 1000Hz a nakon toga tonskom frekfencijom od 500Hz. Imajući u vidu da personalni računar izvršava instrukcije veoma brzo, neophodno je umetnuti vremensko kašnjenje nakon svakog poziva makroa _SPON tako da se generisani ton može čuti. Na ovaj način generiše se načešće vremensko kašnjenje promenjive dužine trajanja, iz razloga što vreme kašnjenja zavisi od pobudne taktne frekfence PC mašine koja je od jednog do drugog modela različita.

```
          MOV   CX,0FFFFH
LABELE:   LOOP  LABELE
```

Vremensko kašnjenje koje smo formirali zavisi od taktne frekfencije tako da je kod većine aplikacija ovaj metod nepouzdan. Tačnije vremensko kašnjenje dobija se korišćenjem prekida koji se javlja u ritmu sistemskog takta za slučaj kada ovaj zahtev za prekid se generiše Tajmerom

0. Da bi pristupili vrednosti na osnovu koje se generiše zahtev za prekid treba pristupiti podatku tipa dupla reč koji se čuva u memoriji na lokaciji 0000:046C. Broj koji se čuva na ovoj lokaciji inkrementira se 18.2 pita u sekundi. On ne sadrži bilo koju specifičnu vrednost nego je jedino predvidjen za korišćenje kod generisanja vremenskih kašnjenja. Jedna od metoda korišćenja ovog broja sastoji se u tome da se da se broj prvo čita, zatim mu se dodaje neka vrednost, nakon toga se smešta u neku drugu grupu registara, a zatim se čeka dok se broj iz memorije ne izjednači sa brojem koji se čuva u registrima. Ovaj metog dozvoljava da se vreme kašnjenja inkrementira sa frekfencijom od 18.2 Hz. Takodje je, pomoću ovog memorijskog brojača, moguće odrediti dužinu trajanja nekog programa. Primer 15-4 pokazuje proceduru koja koristi broj dobavljen iz memorije radi generisanja vremenaskog kašnjenja koje se odredjuje kao AX / 18.2. Drugim rečima, ako AX u trenutku poziva procedure sadrži vrednost 18 tada će se povratak desiti za otprilike 1s. Slično, ako je registar AX postavljen na vrednost AX = 182 povratak iz procedure desiće se nakon 10 sekundi.

Primer 15-4

```
TIMED PROC  NEAR              ;procedura vremenskog kašnjenja
   PUSH  ES                   ;pamćenje stanja registara u magacin
   PUSH  CX
   MOV   CX,0
   MOV   ES,CX                ;adresa segmenta 0000
   XOR   CX,CX
   ADD   AX,ES:[46CH]         ;adresa tekućeg broja
   ADC   CX,ES:[46EH]
TIMED1:
   PUSH  CX
   CMP   AX,ES:[46CH]
   SBB   CX,ES:[46EH]
   POP   CX
   JNC   TIMED1               ;čekanje dok brojač stigne do kraja brojanja
   POP   CX
   POP   ES
   RET
TIMED ENDP
```

KORAK 2: Korišćenjem procedure prikazane u Primeru 15-4 i makroa _SPON i _SPOFF, kreirati program koji pobudjuje zvučnik sa tonskom frekfencijom od 1200 Hz u trajanju od jedne sekunde a zatim tonskom frekfencijom od 700 Hz u trajanju od ½ sekunde.
Procedura kreirana u Primeru 15-4 pogodna je za korišćenje kod velikog broja aplikacija. Ipak treba naglasiti da postoji jedan nedostatak. Naime, zadatak računara je da generiše samo vremensko kašnjenje. Kod nekih aplikacija ova varijanta je prihvatljiva ali kod drugih ovakav način rada može prouzrokovati veći broj problema. Bolji način da se postigne vremensko kašnjenje je da se vremensko kašnjenje obradjuje prekidom INT 8.

Obrada prekida

Obrada prekida zahteva poziv DOS INT 21H, funkcije 25H i 35H. Funkcija 25H upisuje novu adresu u prekidni vektor kako bi se instalirala nova prekidna uslužna rutina dok funkcija 35H čita tekući vektor pomoću koga se pristupa tekućoj prekidno uslužnoj rutini. U praksi, kod instaliranja prekidne uslužne rutine, funkcijom 35H čita se tekuća adresa prekida dok se funkcijom 25H instalira nova adresa prekidne rutine. Ovo dozvoljava staroj prekidnoj uslužnoj rutini da se izvršava iz nove.
Pretpostavimo da vreme za koje se preko zvučnika čuje ton iniciran programski mora biti odredjeno, ali pri tome ne želimo da generišemo ton dok računar očekuje prekid zbog isteka vremenskog kašnjenja. Nova prekidna procedura za vremensko kašnjenje (VEC8) prikazana je u Primeru 15-5. Ona je deo programa koji pobudjuje zvučnik tonskom frekfencijom od 1000 Hz u trajanju od 1 sekunde a nakon toga tonskim signalom frekfencije 500 Hz u trajanju ½ sekunde.

Primer 15-5

```
CODE  SEGMENT 'code'          ;označavanje početka kôdnog segmenta CODE
   ASSUME  CS:CODE            ;usvajamo da CS adresira kôdni segment CODE
   INCLUDE MACS.INC
   ADD8  DD ?                 ;stara adresa za INT 8
   CNT   DW 0                 ;vrednost koja odredjuje kašnjenje
   TONE  DW 0                 ;marker insteklo kašnjenje za  INT 8
   BUSY  DB 0                 ;kašnjenje isteklo
   DBUSY DB 0
   MAIN  PROC  FAR            ;početak procedure MAIN
      MOV   AX,CS             ;DS=CS
      MOV   DS,AX
      MOV   AX,3508H          ;dobavljanje tekuće adrese prekida
      INT   21H
      MOV   WORD PTR ADD8,BX
      MOV   WORD PTR ADD8+2,ES
      MOV   AX,2508H          ;pamćenje nove adrese prekida
      MOV   DX,OFFSET VEC8
      INT   21H               ;instaliranje VEC8
      MOV   CNT,18
      MOV   TONE,1000         ;pobudjivanje zvučnika tonskom frekfencijom
; 1000 Hz u trajanju od 1 sekunde
;obaviti bilo koje druge operacije koje se zahtevaju od strane programa ali pre nego što se
;ponovo pobudi zvučnik ili pre nego što se izadje iz programa proveriti TONE i DBUSY da ;bi se
ustanovilo da li je vremensko kašnjenje isteklo i da li je zvučnik nepobudjen

   MAIN1:
      CMP   TONE,0
      JNE   MAIN1
      CMP   DBUSY,0           ;testiranje da li je kašnjenje isteklo
      JNE   MAIN1             ;granaj se ako je isteklo

      MOV   CNT,9             ;pobuda zvučnika tonoskom frekfencijom
```

```
; od 500 Hz u trajanju od ½ sekunde
      MOV TONE,500
   MAIN2:
      CMP   TONE,0
      JNE   MAIN2
      CMP   DBUSY,0
      JNE   MAIN2
      MOV   DX,WORD PTR ADD8
      MOV   AX,WORD PTR ADD8+2
      MOV   DS,AX
      MOV   AX,2508H
      INT   21H                  ;deaktiviraj prekid 8
      _EXIT                      ;povratak na DOS
   MAIN  ENDP

   VEC8  PROC  FAR               ;novi prekid INT 8
      CMP   BUSY,0               ;testiraj da li je isteklo kašnjenje
      JE    VEC81                ;granaj se ako nije
      JMP   CS:ADD8              ;standardni INT 8
   VEC81:
      MOV   BUSY,1               ;pokaži INT 8 ako je zauzet
      CMP   DBUSY,0              ;da li je vreme kašnjenja aktivno?
      JE    VEC83
      PUSHF
      CALL  CS:ADD8
      STI
      DEC   CNT
      JNZ   VEC82
      _SPOFF
      MOV   DBUSY,0
   VEC82:
      MOV   BUSY,0
      IRET
   VEC83:
      CMP   TONE,0
      JNE   VEC84
      MOV   BUSY,0
      JMP   CS:ADD8
   VEC84:
      MOV   DBUSY,1
      PUSHF
      CALL  CS:ADD8
      STI
      _SPON TONE
      MOV   TONE,0
      MOV   BUSY,0
      IRET
   VEC8  ENDP
```

```
CODE  ENDS
   END   MAIN
```

Kao što se može videti ovo je veoma važan deo programa zbog načina kako se prekid instalira i briše iz memorije. Prvi deo programa izbavlja tekući prekidni vektor 8 i smešta adresu tekuće prekidne rutine obima dupla reč na adresi ADD8. Uočimo kako se ova aktivnost obavlja DOS-ovim sistemskim pozivom INT 21, funkcija 35H.
Zatim se nova prekidna uslužna rutina koja se naziva VEC8 instalira pozivom DOS INT 21H, funkcija 25H. Obratiti pažnju na to kako se adresa VEC8 smešta u registar DX i način na koji je izvedeno da registar DS adresira kodni segment (početak programa).
Nakon što je prekidno uslužna rutina instalirana, ona se na dalje može pozivati (to se vidi u narednom koraku). Da bi se pobudio zvučnik tonskom frekfencijom od 1000 Hz u trajanju od jedne sekunde, treba upisati 18 u CNT (brojač) a vrednost 1000 upisati u memorijsku lokaciju TONE. Ove dve vrednosti se moraju upisivati redosledno. Nakon programiranja brojača i generatora tona, moguće je da se izvršava bilo koja druga operacija jer prekidna uslužna rutina (VEC 8) vodi računa o tajmeru kao i vremenu za koje će zvučnik generisati ton.
Pre nego što se generator tona i brojač mogu ponovo programirati ili pre nego što što se izadje iz programa, vrednosti koje se čuvaju u TONE i DBUSY se moraju testirati. Obe vrednosti moraju biti postavljene na nulu pre nego što se deo programa za pobudu zvučnika može ponovo programirati ili pre nego što program završi.
Prekidno uslužna rutina (VEC8) koristi skup koga čine dva pokazivača kako bi se ona izvršavala. Marker BUSY pokazuje da VEC8 aktivno izvršava operacije dok marker DBUSY pokazuje da je vreme odbrojavanje vremena dodeljeno rutini vremenskog kašnjenja u toku. Takodje uočimo da se kod VEC8 možemo uveriti da se prvi prekid izvršava što je moguće pre. On se izvršava izvršenjem instrukcije JMP CS:ADD8 u nekim slučajevima ili u drugim izvršenjem instrukcija PUSHF i CALL CS:ADD8. Instrukcija JMP CS:ADD8 se ne vraća na VEC8. Instrukcija PUSH iza koje sledi CALL CS:ADD8 emulira hardverski prekid, koji se vraća prekidnoj uslužnoj rutini VEC8 nakon izvršenja prekida. Uočimo da se prekidna uslužna rutina završava bilo instrukcijom JMP CS:ADD8 bilo instrukcijom IRET. Instrukcija IRET je specijalna instrukcija koja se koristi za povratak iz prekida. Razlikuje so od normalnog povratka po tome što prilikom dobavljanja povratne adrese iz magacina obnavlja se istovremeno i marker registar.

KORAK 3: Uneti, asemblirati i izvršiti program iz Primera 15-5 i videti da se zvučnik pobudjuje dva puta.

KORAK 4: Sada kada program prekidno uslužne rutine VEC8 se izvršava i funkcioniše korektno promeniti proceduru MAIN iz Primera 15-5 tako da ona pribavlja informaciju iz memorije od 20 različitih tonskih signala i 20 puta pobudjuje zvučnik.

Pitanja

1. Tajmer 2 se programira preko dva U/I porta, koje su adrese tih portova?
2. Kada se govori o pobudi zvučnika PC mašine ukazati na to koja je uloga čipa 8255 (PIA)?
3. Kako se odredjuje broj koga treba upisati u tajmer 2 da bi se na zvučniku generisao ton frekfencije 1200Hz?
4. Kakva je namena podatka tipa dupla reč koji se čuva u memorijskoj lokaciji 0000:046C?

5. Objasniti koji se registri koriste i kako sistemski poziv DOS INT 21H, funkcija 35H čita prekidni vektor.
6. Objasniti koji se registri koriste i kako sistemski poziv DOS INT 21H, funkcija 25H menja prekidni vektor.
7. Koja je razlika postoji izmedju instrukcije JMP CS:ADD8 i instrukcije CALL CS:ADD8 koje se koriste u Primeru 15-5?
8. Ako je BUSY marker = 1, VEC8 prekidno uslužna rutina je ...
9. Ako je DBUSY marker = 1, VEC8 prekidno uslužna rutina je ...
10. Objasniti kako se prekidno uslužna rutina VEC8 uklanja iz prekidne vektor tabele u Primeru 15-5.

LABORATORIJSKA VEŽBA 16

Program Terminate and Stay Resident

Uvod

Program tipa *Terminate and Stay Residant* nakon izvršenja i dalje ostaju u meoriji. TSR program obično se instalira kao prekidno uslužna rutina koja obavlja neku korisnu funkciju. U ovoj vežbi prikazana je instalacija TSR programa koji se koristi kao prekidno uslužna rutina i *hot-key* sekvenca. Kod većine aplikacija TSR se aktivira bilo prekidnim signalom sa tajmera (INT 8) bilo odredjenim *hot-key* tasterom. U vežbi će biti pokazano kako se aktivira TSR program signalom generisanim od strane brojača tajmera, kao i od strane *hot-key* tastera.

Predmet rada

1. Instalacija prekidno uslužnih rutina koje postaju rezidentne.
2. Instalacija prekidno uslužne rutine tipa TSR koja koristi signal sa internog časovnika, odnosno INT 8.
3. Instalacija prekidno uslužne rutine tipa TSR koja analizira tastaturu i reaguje na odredjeni kôd pritisnute dirke sa tastature (*hot-key*).

Postupak rada

Instaliranje rezidentnih TSR programa se vrši gotovo na identičan način kao i instaliranje prekidno uslužnih rutina. Razlika se ogleda u tome što nakon instalacije prekidno uslužne rutine izlazak iz programa i povratak u DOS se ne vrši uz pomoć poziva INT 21H i funkcije AH=4CH. U ovom slučaju sa INT 21H funkcja 31H se vrši povratak na DOS sa tom razlikom što program koji se izvršavao ostaje rezidentan, tj. on je TSR tipa.

Instaliranje prekidno uslužne rutine tipa TSR

Primer 16-1 prikazuje kratku sekvencu instrukcija pomoću koje se program VEC8 instalira kao nova prekidno uslužna rutina INT 8, a zatim se pokazuje na koji način se iz ove rutine izlazi i kako rutina VEC8 ostaje rezidentna u memoriji.

Primer 16-1

```
;VEC8 i parametar ADD8 moraju se pojaviti pre dela programa SAVE
SAVE: MOV   AX,CS                 ;DS = CS
      MOV   DS,AX
      MOV   AX,3508H              ;pribavljanje INT 8 adresu
      MOV   WORD PTR ADD8,BX
      MOV   WORD PTR ADD8+2,ES
      MOV   AX,2508H              ;instaliranje VEC8
      MOV   DX,OFFSET VEC8
      INT   21H
      MOV   DX,OFFSET SAVE
      SHR   DX, 1
      SHR   DX, 1
      SHR   DX, 1
      SHR   DX, 1                 ;odredjivanje broja paragrafa
      INC   DX
      MOV   AX,3100H
      INT   21H                   ;povratak na DOS
```

U ovom primeru program VEC8 se instalira na isti način kao i u prethodnoj Laboratorijskoj vežbi, ali za razliku od standardnog izlaska za izlazak se koristi funkcija 31H. Funkcija 31H zahteva da u registar DX smesti broj paragrafa (paragraf predstavlja 16 bajtova) koji definiše obim

rezidentnog dela memorije. Uočimo kako se ofset adresa SAVE pomera 4 mesta udesno da bi sadržaj DX bio podeljen sa 16, a zatim se inkrementira sa ciljem da se dobije broj paragrafa. Pretpostavimo da je ofset adresa SAVE smeštena na lokaciji 017AH. To znači da se program za obradu prekida VEC8 kao i parametri koji se pojavljuju pre SAVE smeštaju u memoriju od ofset adrese 0000H do 017AH. Ova oblast memorije sadrži 18H paragrafa. Ako je DX napunjen sa 017AH (ofset adresa SAVE) i pomeren udesno za 4 mesta, on postaje 0017H, a to je prvi parametar. Izvršenjem instrukcija INC DX sadržaj registra DX prima vrednost 0018H pa se taj broj prenosi funkcije 31H kao parametar koji ukazije na broj paragrafa.

Korišćenje sistemskog vremena iz TSR-a

Sobzirom da je TSR softver veoma kompleksan, u ovoj vežbi biće prikazan samo deo TSR softvera koji će biti modifikovan a ne i kompletno njegovu strukturu pisan kompletno. Prvi primer TSR programa prikazan je u Primeru 16-2. To je TSR program koji čita sistemsko vreme korišćenjem BIOS-ov umesto DOS-ov sistemsko-funkcijski poziv. Za rad sistema veoma je važno da TSR koristi BIOS a ne DOS funkcijski poziv. Primer 16-2 instalira poziv za prekid u prekidnom vektoru broj 8, tj. prekid koji se inicira od internog časovnika, testira vreme i generiše po jedan jedan zvučni signal svakog minuta.

Informacija o vremenu se pribavlja iz BIOS-a punjenjem AH sa 2 pre poziva instrukcije INT 1AH. Na ovaj način broj časova se vraća u CH, broj minuta u CL a broj sekundi u DH. Naglasimo da su sve ove vrednosti predstavljene u BCD kôdu. Program pamti broj minuta u trenutku instalacije u memorijsku lokaciju MIN. Na pojavu svakog taktnog impulsa generisan od strane internog časovnika, poziva se VEC8, zatim se čita vreme sa sistemskog časovnika koristeći poziv INT 1AH i uporedjuje se to vreme sa vremenom koje se čuva u MIN. Ako je došlo do promene, to znači da je istekao jedan minut, tako da se ažurira MIN i aktivira zvučni signal. Program generiše po jedan zvučni signal svakog minuta sve dok se računar ne restartuje ili isključi.

Primer 16-2

```
CODE  SEGMENT'code'          ;ukazuje na početak kôdnog segmenta
   ASSUME   CS:CODE          ;usvajamo da CS adresira kôdni segm. CODE
   INCLUDE  MACS.INC
   ORG   100H                ;karakteristična adresa početka .COM program
 START:  JMP SETUP           ;instalacija VEC8
         ALIGN 4             ;izbor granice u duple reči
         ADD8  DD ?          ;mesto za čuvanje stare INT 8 adrese
         MIN   DB ?          ;broj minuta
         BUSY  DB 0          ;marker zauzeća VEC8
         DBUSY DB 0          ;marker da je kašnjenje u toku
         DCNT  DB 0          ;brojač koji odredjuje kašnjenje
VEC8   PROC   FAR            ;nova rutina INT 8 za obradu prekida
         CMP   BUSY,0        ;da li je VEC8 zauzet?
         JE    VEC81         ;granaj se ako nije
         JMP   CS:ADD8       ;izvršiti stari INT 8
 VEC81:  MOV   BUSY,1        ;prikaz da je VEC8 zauzet
         PUSHF
         CALL  CS:ADD8       ;pozovi stari INT 8
         STI                 ;dozvola prekida
         CMP   DBUSY,0       ;da je li brojac koji odredjuje kašnjenje zauzet
         JE    VEC83         ;ako nije
```

```
          DEC   DCNT              ;dekrementira brojača koji odredjuje kašnjenje
          JNE   VEC82             ;ako nije nula
          _SPOFF                  ;isključiti pobudu zvučnika
          MOV   DBUSY,0           ;briše se marker brojača koji odr. kašnj. VEC8
VEC82:    MOV   BUSY,0            ;briše se marker o zauzeću VEC8
          IRET
 VEC83:   PUSH  AX                ;pamti se stanje registara
          PUSH  CX
          PUSH  DX
          MOV   AH,2              ;pribavlja se informacija o vremenu
          INT   1AH
          CMP   CL,MIN            ;provera da li došlo do promene stanja minuta
          JE    VEC84             ;ako nije došlo do promene
          MOV   MIN,CL            ;meni MIN
          MOV   DCNT,3            ;generisati ton u trajanju od 1/6s
          _SPON  1000             ;postaviti tonsku frekfencu na 1000 Hz
          MOV   DBUSY,1           ;postaviti marker da je kašnjenje u toku
 VEC84:   POP   DX
          POP   CX
          POP   AX
          MOV   BUSY,0            ;obrisati marker o zauzeću VEC8
          IRET
VEC8  ENDP

 SETUP:   MOV    AX,CS                    ;DS = CS
   MOV    DS,AX
   MOV    AX,3508H                 ;dobaviti staru adrersu za INT 8
   INT    21H
   MOV    WORD PTR ADD8,BX
   MOV    WORD PTR ADD8+2,ES
   MOV    AH,2
   INT    1AH                      ;dobaviti podatke o vremenu
   MOV    MIN,CL                   ;zapamtiti minute u MIN
   MOV    AX,2508H                 ;instaliranje VEC8
   MOV    DX,OFFSET VEC8
   INT    21H
   MOV    DX,OFFSET SETUP
          SHR   DX,1
          SHR   DX,1
          SHR   DX,1
          SHR   DX,1
          INC   DX
          MOV   AX,3100H                ;učiniti da program bude tipa TSR
          INT   21H
CODE  ENDS                              ;kraj kodnog segmenta
   END   START                          ;kraj programa
```

KORAK 1: Uneti, asemblirati i izvršiti program iz Primera 16-2. Pri tome treba podesiti opcije u PWB meniju tako da se program asemblira kao .COM fajl. Iz razloga što fajl tipa .COM zauzima manje memorije od odgovarajućeg fajla tipa .EXE, kod kreiranja TSR programa, preporučuje se korišćenje formata .COM. Po izvršenju ovog programa, trebalo bi da se čuje kratak zvučni signal sa zvucnika PC-ja jedput svake minute.

KORAK 2: Modifikovati program iz koraka 1 tako da se zvuk čuje jednom svakog sata. Takodje, promeniti ton sa 1000Hz na 700Hz tako da ovaj ton zvuči drugačije od ostalih sistemsko generisanih tonova.

Prekidna rutina za opsluživanje tastature

Prekidni program za obradu podataka koji se prihvataju sa tastature kao i TSR rezidentni program često koriste INT 9 radi detekcije pristupa specijalnih dirki ili kombinacije dirki koji se nazivaju *hot-key*. Prekidno uslužna rutina INT 9 poziva se uvek kada se pritisne bilo koja dirka tastature. Ako se na lokaciji prekidnog vektora 9 smesti adresa prekidno uslužne rutine 9 tada se vrši analizira tastature i prihvata informacija koja ukazuje na to koji yadatak treba obaviti ako je pritisnuta jedna dirka, a koji ako je pritisnuto više njih a odnose se na kombinaciju dirki. Program ya obradu prekida INT 9 obično smešta kôd pritisnute dirke u red čekanja radi kasnijeg preuzimanje od strane DOS-ovih ili BIOS-ovih funkcijskih poziva. Kada se podatak sa tastature prihvati on se prosledjuje prekidnom programu, a nakon obrade, odbacuje. *Hot* tasteri se često koriste za aktiviranje programa koji ostaju rezidentni u memoriji.

Primer 16-3 ukazuje na jedan kratak INT 9 prekidni program koji testira da li je pritisnuta dirka koja kôdira veliko slova Q a briše ga ako je taj znak otkucan. Ovo naravno nije praktični primer, medjutim, ilustrativno ukazuje na mogućnosti obrade. Kodovi tipa *Scancode* generisani od strane sa tastature prihvataju se sa interfejsa tastarue izvršenjem instrukcije IN AL,60H, čime se u stvari pristupa portu A odgovarajuće PIA komponente. Nakon izvršenja ove instrukcije u registru AL nalazi se kopija kôda dirke koji se još uvek ne briše od strane sa interfejsa. Nakon što se *csancod* pribavi u registar AL, on se testira na sadržaj dirke Q (10H). Ukoliko se ustanovi da je to kôd 10H, u daljem toku se testira da li je pritisnuta dirka *Shift* kako bi se utvrdilo da je to kôd velikog slova. To se postiže na sledeći način. Informacija o dirci *Shift* smešta se u memorijsku lokaciju koja se nalazi na adresi 0000:0417 (Slika 16-1). Ako je pritisnut leva ili desna *Shift* dirka (dva LS bita memorijse reči su postavljene na 1), a za slučaj da registar AL sadrži 10H, sa sigurnošću se može reći da je pritisnuta dirka velikog slova Q. Ako ni jedan od pomenutih bita nije postavljen, pritisnuta je dirka malog slova Q (*q*).

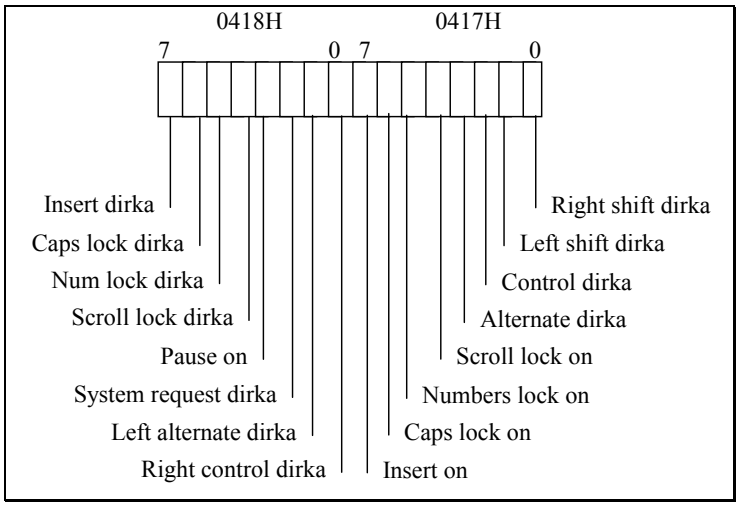
Primer 16-3

```
CODE SEGMENT 'code'              ;početk kôdnog segmenta CODE
   ASSUME CS:CODE          ;usvajamo da CS adresira kôdni segm. CODE
          ORG   100H              ;za COM programe
 START:   JMP  SETUP              ;instalacija programa VEC9
          ALIGN 4                 ;izbor memorijskog poravnanja tipa dupla reč
          ADD9   DD   ?           ;mesto za adresu starog INT 9

VEC9     PROC    FAR             ;nova INT 9 prekidna rutina
STI                        ;dozvola prekida
          PUSH  AX                ;sačuvaj AX
          IN    AL,60H            ;čita scancode sa tastature
          CMP   AL,10H            ;testiranje za dirku Q
          JNE   VEC91             ;ako nije Q dirka
          PUSH  ES
```

```
         MOV   AX,0
         MOV   ES,AX            ;segment sa adresom 0000
         MOV   AL,ES:[17H]      ;pribavlja se bajt sa adrese 417H
         POP   ES
         AND   AL,3             ;maskiranje svih bitova osim za Shift
         JNZ   VEC92            ;ako je pritisnuta Shift dirka
VEC91:   POP   AX
         JMP   CS:ADD9          ;izvrsavaj uobičajeni prekidni program
VEC92:   CLI                    ;zabrana prekida
         IN    AL,61H           ;odbacuje kôd dirke
         OR    AL,80H
         OUT   61H,AL           ;slanje impulsa tastaturi
         AND   AL,7FH
         OUT   61H,AL
         MOV   AL,20H           ;reset kontrolera prekida
         OUT   20H,AL
         STI                    ;dozvola prekida
         POP   AX
         IRET
VEC9  ENDP

SETUP:   MOV   AX,CS                    ;DS = CS
         MOV   DS,AX
         MOV   AX,3509H                 ;pribavlja staru adresu za INT 9
         INT   21H
         MOV   WORD PTR ADD9,BX
         MOV   WORD PTR ADD9+2,ES
         MOV   AX,2509H                 ;instalira se VEC9
         MOV   DX,OFFSET VEC9
         INT   21H
         MOV   DX,OFFSET SETUP
         SHR   DX, 1
         SHR   DX, 1
         SHR   DX, 1
         SHR   DX, 1
         INC   DX
         MOV   AX,3100H                 ;upisati rezidentni program TSR
         INT   21H
CODE ENDS                               ;kraj kôdnog segmenta
   END   START                          ;kraj programa
```



Slika 16-1 Binarni sadržaj statusne reči tastature na lokacijama 0000:0417H i 0000:0418H

KORAK 3: Uneti asemblirati i izvršiti program iz Primera 16-3. Prethodno, uveriti se da je PWB konfigurisan tako da generiše .COM fajl. Da li je nakon izvršavanja ovog programa moguće pritisnuti taster Q? _____
Ovaj program se izvršava u pozadini a glavni zadatak je da proveri da li je pritisnut kôd dirke Q. Jedini način da se izadje iz ovog programa je da se obavi restartovanje računara.

KORAK 4: Modifikovati program iz Primera 16-3 tako da detektuje pritisnutu dirku za malo slovo t. Asemblirati i izvršiti ovako izmenjeni program.

KORAK 5: Modifikovati program iz Primera 16-3 tako da detektuje dirku F1 i pobuditi zvučnik da generiše signal uvek kada je pritisnuta dirka F1.

KORAK 6: Modifikovati program iz Primera 16-3 tako da detektuje bilo koju funkcionu dirku (F1 do F10). Kreirati program tako da F1 generiše zvučni signal frekfencije 1000Hz, F2 900Hz, F3 800Hz, itd.

Pitanja

1. Koja je svrha DOS-ovog INT 21H poziva, funkcija 31H?
2. Šta predstavlja paragraf?
3. Koliko dugo ostaje TSR zapamćen u memoriji?
4. Kako bi se mogao modifikovati program iz Primera 16-2 da bi se po jedanput pobudio zvučnik tonskim signalom svakih pola sata a po dva puta na svaki sat?
5. Koja je namena instrukcije IN AL,60H?
6. Može li se prekidno uslužna rutina definisana vektor brojem 9 i odgovarajućom tabelom pretraživanja iskoristiti za preraspodelu kôdova svih dirki na tastaturi?
7. Koja je namena memorijske reči na adresi 0000:0417H?
8. U čemu je razlika izmedju *scancode* kôda i ASCII kôda?
9. Može li se kreirati takav prekidni program koji će detektovati pritisnutu kombinaciju dirki ALT, desni Shift, i Z dirka?
10. Mogu li se programi iz Primera 16-2 i 16-3 kombinovati tako da ALT F1 aktivira zvučni tonski signal, a ALT F2 deaktivira tonski signal?

LABORATORIJSKA VEŽBA 17

Programiranje sa aritmetičkim koprocesorom

Uvod

Aritmetički koprocesor kao komponenta (*off chip*) se pridruživao ranijim mikroprocesorima iz familije 80x86. Danas je aritmetički deo integrisan u okviru čipa (*on chip*) kakav je slučaj kod 80486DX i Pentium, PentumPro itd. To znači da je on postao značajno sredstvo, alat koji se koristi kod kreiranja sistemskih i korisničkih programa pa je zbog toga za programaera od izuzetne važnosti da razume njegov rad. Ova Laboratorijska vežba pokazuje način korišćenja aritmetičkog koprocesora kod nekih jednostavnih aplikacija. Naglasimo da je u ovoj Laboratorijskoj veži minimalno potreban mikroprocesor tipa 80386 ili neki noviji. Takodje moguće je koristiti i različite tipove emulatora za koprocesor 80387.

Predmet rada

1. Koristi aritmetički koprocesor i njegovu magacinsku arhitekturu da bi se kreirali programi.
2. Detektovati greške koristeći statusni registar koprocesora.
3. Korišćenje makroa koji je namenjen da obavi prikaz podataka na video displeju u formatu pokretni zarez.
4. Korišćenje podataka u pokretnom zarezu za izračunavanje.

Postupak rada

Aritmetički koprocesor izvršava sve aritmetičke operacije u formatu brojeva pokretnog zareza. Podaci izmedju memorije i koprocesora se mogu prenositi koristeći format pokretnog zareza, format celobrojnih vrednoasti (*integer*) ili u BCD formatu.

Definisanje podataka za koprocesor

Formati koji se koriste za prezentacviju brojeva u pokretnom zareza mogu biti tipa: jednostuka preciznost (32-bita), dvostruka preciznost (64-bita) i proširena preciznost (80-bitova); formati tipa *integer* su : 16-, 32-, i 64-bita; BCD format se uvek memoriše kao 18-cifarski BCD *integer* celi broj a obima je 80-bitova. Detaljan opis sadržaja ovih formata nije predstavljen ovde jer se najčešće koriste standardne direktive za njihovo definisanje. Primer 17-1 pokazuje kako se smeštaju podaci u ovim formatima u memoriji korišćenjem atandardnih asemblerskih direktiva.

Primer 17-1

```
                ;integer brojevi obima reč (16-bitova)
0000 0138           DATA1   DW    312
0002 FF16           DATA2   DW    -234
                ;
                ;integer brojevi obima dupla reč (32-bita)
0004 5BA0           DATA3   DW    23456
0006 A476           DATA4   DW    -23434
                ;
                ;integer brojevi obima četvorostruke reči (64-bita)
0008                DATA5   DQ    123456789
    000000004995FE85
0010                DATA6   DQ    -1
```

```
    FFFFFFFFFFFFFFFF
                ;
                ;integer BCD brojevi
0018                DATA7   DT    1234
    0000000000000001234
                ;
                ;broj u pokretnom zarezu u jednostrukoj preciznosti
002C 42F6AE14       DATA9   DD    123.34
0030  BF800000       DATA10  DD    -1.0
                ;
                ;broj u pokretnom zarezu u dvostrukoj preciznosti
0034                DATA11  DQ    123.34
    405ED5C28F5C28F6
003C                DATA12   DQ   -200.0
    C069000000000000
                ;
                ;broj u pokretnom zarezu u proširenoj preciznosti
0044                DATA13   DT   123.34
    4005F6AE14
```

Naglasimo da su *integer* i BCD brojevi definisani bez decimalnog zareza a da brojevi u pokretnom zarezu moraju sadržati decimalni zarez. Takodje naglasimo da se može koristiti eksponencijalna forma, na primer 1E2 odgovara vrednosti ($1*10^2 = 100$)

Korišćenje koprocesora kod rešavanja jednostavnih problema

Primer 17-2 prikazuje kratak program koji se koristi za odredjivanje površine pravougaonika dimenzije L * W i prikazuje rešenje korišćenjem makroa _NUM 10. Dužina i širina pravougaonika se čuvaju u memoriji kao *integer* konstante (tipa celobrojna vrednost). Naglasimo da prva linija ovog primera koristi .387 komutator (*switch*) kako bi pristupio aritmetičkom koprocesoru 80387. Ako koristite mikroprocesor 80287 koristite postavite *switch* .287. Kada koristite mikroprocesor 80486DX upotrebite *switch* .487 a kada imate Pentium koristite *switch* .586.

Primer 17-2

```
.387
CODE  SEGMENT  'code'        ;označavanje početka kôdnog segmenta CODE
   ASSUME  CS:CODE          ;usvajamo da CS adresira kôdni segment CODE
   INCLUDE  MACS.INC
   L  DW 12                 ;dužina
   W  DW 14                 ;širina
   AREA  DW ?               ;polje
   MAIN  PROC  FAR
      FILD  L               ;punjenje celog broja L
      FIMUL W               ;množenje sa celim brojem W
      FISTP AREA            ;polje smeštanja i izbavljanja
      MOV   AX,AREA         ;dobijanje polja
      _NUM  10              ;prikaz polja
      _EXIT                 ;povratak u DOS
```

```
MAIN  ENDP
CODE  ENDS                    ;kraj kodnog segmenta
   END   MAIN                 ;kraj programa
```

KORAK 1: Uneti, asemblirati i izvršiti program prikazan u Primeru 17-2. Koji se broj prikazuje na video displeju?

Mada je program prikazan u Primeru 17-2 kratak i manipuliše sa celobrojnim vrednostima (*integers*) on pokazuje kako se programira koprocesor. Prva instrukcija koprocesora FILD L puni koprocesor sa L. Druga instrukcija množi sadržaj vrha magacina sa *integer* vrednošću W i zatim smešta rezultat AREA na vrh magacina. Zadnja instrukcija koprocesora FISTP smešta kopiju sadržaja vrha magacina u memorijsku lokaciju AREA i izbavlja tu vrednost sa vrha magacina. Naglasimo da podaci u koprocesoru ostaju kao što su i bili i pre izvršenja ove sekvence (ostaju nepromenjeni). Važno je da se uvek izbavi podatak sa vrha magacina na kraju operacije.

Korišćenje makroa radi prikazivanja podataka u pokretnom zarezu

Makro _NUM 10 prikazuje sadržaj registra AX kao decimalni ceo (*integer*) broj. U programima koji koriste koprocesor, ovaj makro se koristi za prikaz podataka u pokretnom zarezu. Takav jedan makro nazvan _FNUM se javlja u Primeru 17-3. Parametri koji se prenose makrou _FNUM se nalaze u MEMS, u koji se čuva adresa broja u pokretnom zarezu u jednostrukoj ili dvostrukoj preciznosti, kao i vrednost PL koja odgovara broju cifara iza zareza koje treba prikazati. Ovaj način specifikacije broja je neophodan kako bi se isti korektno prikazao na video displeju. Kada je PL = 0, makro prikazuje samo njegov celobrojni deo.

Primer 17-3

```
_FNUM  MACRO   MEMS,PL               ;prikaz podataka u pokretnom zarezu
LOCAL
  T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,N1,N2,N3,N4,N5,E1
      PUSH   AX
      PUSH   BX
      PUSH   CX
      PUSH   DX                     ;koristi se od strane makroa _DISP
      PUSH   SI
      FSTCW  N2                     ;pamćenje upravljačke reči
      MOV    N1,0E00H
      FLDCW  N1                     ;punjenje nove upravljačke reči
      FLD    MEMS                   ;dobavljanje broja
      FBSTP  N3                     ;pamćenje celobrojnog dela vrednosti
      MOV    SI,OFFSET N3+8
      MOV    CX,9
      MOV    BL,1                   ;anulirati uticaj markera
      CMP    BYTE PTR CS:[SI+1],80H
      JNE    T1                     ;ako je pozitivan
      _DISP  '-'
T1:   MOV    AL,CS:[SI]             ;prikaz cifre
      DEC    SI
      MOV    BH,AL
      SHR    AL,4
      JNZ    T2
      CMP    BL,1
```

```
        JE      T3
T2:     MOV     BL,0                    ;prikaz cifre
        ADD     AL,30H
        _DISP   AL
T3:     AND     BH,0FH
        JNZ     T4
        CMP     BL,1
        JE      T5
T4:     MOV     BL,0
        ADD     BH,30H
        _DISP   BH
T5:     LOOP    T1
        CMP     BL,1
        JNE     T6
        _DISP   '0'
T6:     MOV     CX,PL
        CMP     CX,0
        JE      E1                      ;granaj se ako je PL=0
        FLD     MEMS                    ;odredjivanje razlomljenog dela
        FBLD    N3
        FSUB
        FABS                            ;učiniti broj pozitivanim
        FTST                            ;testiraj ako ne postoji razlomljen deo
        FSTSW   AX
        AND     AX,4500H
        CMP     AX,4000H
        JNE     T7                      ;granaj se ako je razlomljeni deo
        FISTP   N1                      ;brisanje magacina
        JMP     E1
T7:     _DISP   '.'                     ;prikaz decimalne tačke
        FLD     N4                      ;dobijanje faktora zaokruživanja
        MOV    CX,PL                    ;dobijanje broja mesta
T8:     FIDIV   N5                      ;zaokruživanje razlomljenog dela
        LOOP    T8
        FADD
        MOV    CX,PL                    ;odredjivanje broja cifrara
        MOV     SI,OFFSET  N3
T9:     FIMUL   N5
        FIST    N1
        MOV     AX,N1
        MOV     CS:[SI],AL
        INC     SI
        FISUB   N1
        LOOP    T9
        FISTP   N1                      ;brisanje magacina
        MOV     CX,PL                   ;brisanje vodećih nula
T10:    DEC     SI
        CMP     BYTE  PTR  CS:[SI+1],0
        JNE     T11
```

```
      LOOP   T10
T11:  MOV    BYTE  PTR  CS:[SI+1],10
      MOV    SI,OFFSET  N3
T12:  MOV    AL,CS:[SI]
      CMP    AL,10
      JE     E1
      INC    SI
      ADD    AL,30H
      _DISP  AL
      JMP    T12
 N1   DW  ?                          ;privremena konstanta
 N2   DW  ?                          ;upravljački registar
 N3   DT  ?                          ;celobrojna vrednost (integer) broj
      DT  ?                          ;dodatni broj mesta za razlomljeni deo
 N4   DQ  0.5                        ;faktor zaokruživanja
 N5   DW  10                         ;celobrojna vrednost 10
E1:   FLDCW  N2                      ;obnavljanje upravljačke reči
      POP    SI
      POP    DX
      POP    CX
      POP    BX
      POP    AX
      ENDM
```

Makro _FNUM je prilično kompleksan program prvenstveno zbog aktivnosti koje on obavlja. Prvi deo ovog makroa izrečunava celobrojni deo broja u pokretnim zarezu na taj način što ga zaokružuje i smešta u memoriji kao BCD celobrojnu vrednost koristeći instrukciju FBSTP. Ako se koristi neka druga tehnika zaukruživanja tada će broj biti zaokružen, kao u Primeru 12-5.
Nakon što je celobrojna vrednost (maksimum do 18 cifara) dobijena iz broja, isti se prikazuje bez vodećih nula koje se nalaze na početku broja. Ako je broj cifara koji treba prikazati iza zareza nula (PL = 0), ili ne postoji razlomljeni deo tada se makro završava prikazivanjem celobrojne vrednosti.
Ostatak broja iza zareza se zaokružuje i konvertuje u BCD koristeći sekvencu uzastopnih množenja sa 10. Bilo koja nula koja se nalazi na kraju broja ne prikazuje se. Tako na primer, broj kao što je 123.234 kada se pozove makro _FNUM DATA1,5 se prikazuje kao 123.234.

KORAK 2: Priključiti makro _FNUM *include* fajlu, asemblirati ga i izvršiti test programa čiji je listing dat u Primeru 17-4. Na video displeju prikazaće se –100.023 pod uslovom da su svi podaci korektno uneti. Pokušajte da smestite drugi broj u DATA1, kako bi testirali ovaj makro. Naglasimo da je prikaz broja ograničen na 18 cifara sa leve strane zareza i na 19 cifara sa desne.

Primer 17-4

```
.386                                 ;koristiti mikroprocesor 80386
.387                                 ;koristi koprocesor 80387
CODE  SEGMENT USE16  'code'          ;označavanje početka CODE segmenta
   ASSUME   CS:CODE                  ;usvajamo da CS adresira kôdni segm. CODE
   INCLUDE  MACS.INC
   DATA1  DQ   -100.023
   MAIN   PROC  FAR
```

```
      _DISP  13
      _DISP  10
      _FNUM  DATA1,5
      _DISP  13
      _DISP  10
      _EXIT                    ;izlaz u DOS
   MAIN  ENDP
CODE  ENDS                     ;kraj kôdnog segmenta
   END   MAIN                  ;kraj programa
```

KORAK 3: Sada sobzirom da raspolažemo sa makroom koji se koreisti za prikazivanje podataka u pokretnom zarezu, kreirati program koji izračunava površinu kruga ($P = r^2\pi$) kada se radijus menja u opsegu od 10.0 cm do 16.0 cm sa korakom od ½ cm. Prikaz rezultata organizovati tabelarno tako da se u levoj koloni prikazuje radijus a u desnoj površina kruga za taj radijus. Broj kolone numerisati. Napomenimo da se π kao konstanta generiše unutar koprocesora izvršenjem instrukcije FLDPI.

KORAK 4: Kreirati program koji izračunava vrednost funkcije sinus u opsegu od 0^o do 90^o sa korakom od 1^o. Izračunate vrednosti prikazati tabelarno. Pošto je kreirana tabela velika prikazati za dobijene rezultate samo uglove koji su celobrojni umnožak za 10^o (tj. 10^o 20^o 30^o ...).
Da bi izvršio ovaj zadatak koristiti instrukciju FSIN. Instrukcija FSIN odredjuje sinus ugla datog u radijanima. Dobijeni rezultat je lociran na vrhu magacina. Naglasimo da je $360^o = 2\ \pi$ rad, ili $1^o = \pi / 180$ rad.

Pitanja

1. Pokazati kako se sledeći decimalni brojevi smeštaju kao brojevi u pokretnom zarezu u prezentaciji duple preciznosti:
   a) –123
   b) 23.3
   c) –0.0000345
2. Pokazati kako se sledeći decimalni brojevi smeštaju kao BCD brojevi.
   a) 123 b) -312
3. Koja je svrha *switch* .487 na početku programa?
4. Koja je svrha *switch* .387 na početku programa?
5. Zabog čega se javlja direktiva USE16 u Primeru 17-4?
6. Makro _FNUM koristi instrukciju FBSTP. Koju aktivnost obavlja ova instrukcija?
7. Makro _FNUM koristi instrukciju FIMUL. Koju aktivnost obavlja ova instrukcija?
8. Objasniti kako makro _FNUM prikazuje celobrojni deo broja u pokretnom zarezu.
9. Objasniti kako makro _FNUM zaokružuje razlomljeni deo broja u pokretnom zarezu.
10. Objasniti kako makro _FNUM prikazuje mantisu (frakciju) broja u pokretnom zarezu.

LABORATORIJSKA VEŽBA 18

Korišćenje instrukcija kod miroprocesora 80386 – Pentium

Uvod

Personalni računari tipa 80386, 80486, Pentium, PentiumPro, ... su napredne 32-bitne verzije ranijih 8086/8088, 80186/80188 i 80286 računara. Glavne razlike ogledaju se u implementaciji registara obima 32-bita i odredjenih instrukcija koje ne postoje kod ranijih 16-bitnih verzija. U ovoj Laboratorijskoj vežbi objašnjeno je kako se programiraju novije verzije mikroprocesora iz familije x86 korišćenjem Microsoftovog MASM asemblera.

Predmet rada

1. Kreirati program koji koristi 32-bitne registre kakvih ima implementirano kod 80386, 80486, Pentium, … mikroprocesora.
2. Kreirati neku jednostavnu aplikaciju koristeći 32-bitne instrukcije.
3. Razumevanje razlike izmedju 16- i 32-bitnih instrukcija kod 80386, 80486 i Pentium mikroprocesora.

Postupak rada

Programiranje novijih mikroprocesora iz famije x86 je u osnovi isto kao i programiranje ranijih mikroračunara. Glavna razlika ogleda se u tome jer su nam sada dostupni 32-bitni registri pa ne postoji potreba da se naglasi asembleru sa kojim tipom mikroprocesora radimo (80386, 80486, Pentium …).

Korišćenje asemblera kod 80386, 80486, Pentium

Postoje dve stvari koje se moraju obaviti da bi se koristio 80486 mikroprocesor sa njegovim asemblerom. Kao prvo direktiva (.486) mora se nalaziti na početku programa i to u prvoj liniji dok se direktivama USE16 ili USE32 mora identifikovati da li kôdni segment koristi 16- ili 32-bitni instrukcijski režim rada. Tekuće su sve DOS i veliki broj Windows aplikacija napisane za 16-bitni instrukcijski režim rada. Današnji operativni sistemi, sa druge strane, standardno koriste 32-bitni instrukcijski režim rada.

16-bitni instrukcijski režim rada po definiciji pristupa 8 - i 16 - bitnim registrima na isti način kako je to bilo kod 16–bitnih mikroprocesora tipa 8086/8088 itd. Kada je kod mikroprocesora 80386/486/Pentium.... specificiran 32–bitni registar ili memorijska lokacija tada se u okviru instrukcije, mora koristiti prefiks koji će ukazati mikroprocesoru u kom režimu rada će izvršiti instrukciju (16–bitni ili 32–bitni). Direktiva USE16 je ta koja po automatizmu obaveštava asembler da umetne prefikse za 16–bitni rad. U 32–bitnom instrukcijskom režimu rada, mikroprocesor po definiciji adresira 8– i 32–bitne registre. Prelazak na korišćenje 16–bitnih registara ili 16–binih memorijskih lokacija zahteva upotrebu prefiksa. Kakav je slučaj bio sa direktivom USE16, isti zaključak važi i za direktivu USE32 koja umeće odgovarajuće prefikse u trenutku komutacije mikroprocesora iz jednog režima rada u drugi.

Primer 18-1 pokazuje jedan asemblirani program, ali pri tome ne prikazuje listing makro ekspanzija a koristi instrukcije mikroprocesora 80386/486/Pentium ... kao i aritmetički koprocesor radi prikaza osam brojeva na video displeju.

Primer 18-1

```
          .386                              ;koristi se 80386 mikroprocesora
          .387                              ;koristi se 80387 mikroprocesora
0000   CODE  SEGMENT  USE16 'code'          ;označav. početka kôdnog segmenta
              ASSUME   CS:CODE              ;označavanje CODE kao CS
              INCLUDE  MACS.INC
0000 404EF9DB       NUM       DD 3.234
```

```
0004 00000000          DATA1    DD ?
0008                   MAIN  PROC    FAR
0008 B8 000A                 MOV   CX,10          ;punjenje brojača
000B 66| 2E: A1  0000 R      MOV   EAX,NUM
0010 66| 2E: A3  0004 R      MOV   DATA1,EAX
0015                  MAIN1:
                             _FNUM    DATA1,3
                             _DISP    13
                             _DISP    10
014D 2E: D9 06 0004          FLD      DATA1
0152 D9 EB                   FLDPI
0154 DE C9                   FMUL
0156 2E: D9 1E 0004 R        FSTP     DATA1
015B 49                      DEC      CX
015C 0F 85 FEB5              JNZ      MAIN1
                             _EXIT                ;povratak u DOS
0165                   MAIN  ENDP
0165   CODE   ENDS                                ;kraj kôdnog segmenta
              END    MAIN                         ;kraj programa
```

Ovaj program pribavlja vrednost promenjive sa lokacije NUM (3.234) i prikazuje je. Nakon njenog prikaza iz DATA1, program množi ovaj broj sa *n* i prikazuje novi rezultat. Ova aktivnost se ponavlja deset puta. Obratimo pažnju na način na koji se instrukcije MOV EAX,NUM i MOV DATA1,EAX kôdiraju. Prvi broj koji se smešta u memoriji (66|) predstavlja prefiks koji se dopisuje ispred instrukcije i definiše obim registra sa kojim instrukcija manipuliše. On informiše mikroprocesor 80386, koji radi u 16-bitnom instrukcijskom režimu rada, da je obim registra sa kojim instrukcija treba da manipuliše 32 bita. Takodje treba obratiti pažnju da ove dve instrukcije adresiraju podatke koji se nalaze u kôdnom segmentu tako da kodiranje instrukcije uključuje korišćenje para CS: segmentno dopisni prefiks (2E:). Ako se pogledju ostale instrukcije koje adresiraju DATA1, može se takodje uočiti prisustvo segmentno dopisnog prefiksa. Jedna dodatna razlika se javlhja u listingu programa a ta je što instrukcija JNZ MAIN1 koristi specifikaciju adrese koja se odnosi na bliski (*near*) skok. Kod miroprocesora 80386/486/Pentium … instrukcije uslovnog grananja mogu da koriste bliske adrese grananja, ili tradicionalne kratke adrese grananja. Kod ovog primera se koristi specifikacija bliskih adrese grananja jer je razdaljina grananja suviše velika za kratke adrese.

KORAK 1: Uneti, asemblirati, i izvršiti program prikazan u Primeru 18-1. Nakon što je program pokrenut, prikazati 10 brojeva na video displeju.
1._____ 2._____ 3._____ 4._____ 5._____ 6._____ 7._____ 8._____ 9._____ 10._____

KORAK 2: Modifikovati program iz Primera 18-1 tako da za prezentaciju brojeva u pokretnom zarezu koristi format duple preciznosti. U ovom slučaju za potrebe definicije promenjivih NUM i DATA1 koristiti direktivu DQ umesto DD i promeniti program da kopira svih 64-bita promenjive NUM u promenjivu DATA1. Da bi se adresirali svih 64-bitova promenjive NUM potrebno je koristiti DWORD PTR NUM kako bi se adresirala prva četri bajta promenjive NUM a zatim koristiti DWORD PTR NUM + 4 kako bi se adresirala zadnja četri bajta promenjive NUM.

KORAK 3: Uneti, asemblirti, i izvršiti modifikovani program, a nakon toga zapisati 10 brojeva koji se prikazuju na vido displeju.

1._____ 2._____ 3._____ 4._____ 5._____ 6._____ 7._____ 8._____ 9._____ 10._____

KORAK 4: Porediti 10 brojeva prikazanih u KORAKU 1 sa 10 brojeva prikazanaih u KORAKU 3. Ukazati na njihove razlike.

Adresiranje sa umnožavanjem indeksa

Jedan važan način adresiranja koji se javlja počev od mikroprocesora 80386 pa nadalje naziva se adresiranje sa umnožavanjem indeksa (*scaled-index addressing*). Adresiranje sa umnožavanjem indeksa dozvoljava da se strukturama podataka tipa polja koje se smeštaju u memoriji pristupa indeksnim registrom. Format ovog načina adresiranja zahteva korišćenje dva registra kako bi se adresirala memorija. U jedanom registaru se čuva bazna adresa plolja podataka a u drugom se čuva indeks. Tipičan primer je instrukcija MOV EAX,[EBX+4*ECX]. Ako registar EBX sadrži početnu adresu memorijskog polja a registar ECX sadrži broj elemenata, tada ova instrukcija može da adresira bilo koje 32-bitno polje podataka. Na primer, ako registar ECX sadrži 2, tada memorijska adresa kojoj se pristupa oderdjuje se sadržajem registra EBX (u ovom registru se čuva početna adresa polja) plus 8 (4 x 2). Prvi elemenat polja (broj 0) se čuva u bajt lokacijama koje počinju od adrese 0 do 3, drugi element (broj 1) se čuva u bajt lokacijama čije se adrese nalaze u opsegu od 4 do 7, treći elemenat (broj 2) se smešta u bajt lokacije čije se adrese nalaze u opsegu od 8 do C, itd. Treba naglasiti da indeks 2 generiše 8 jer se kod ovog adresnog načina rada koristi faktor umnožavanja 4.
Primer 18-2 prikazjue kako se adresni način rada sa umnožavanjem indeksa koristi da prikaže 5 brojeva koji su locirani u memorijskoj oblasti (polje) koje se naziva ARRAY.

Primer 18-2

```
.386                          ;koristi se 80386 mikroprocesor
.387                          ;koristi se 80387 koprocesor
CODE  SEGMENT USE16  'code'   ;označavanje početka kôdnog segmenta CODE
   ASSUME   CS:CODE           ;usvajamo da CS adresira kôdni segm. CODE
   INCLUDE  MACS.INC
   ARRAY    DD  6.29                    ;element 0
            DD  33.234                  ;element 1
            DD  -234.2                  ;element 2
            DD  23.4                    ;element 3
            DD  1.2E-1                  ;element 4
   TEMP     DD  ?
   MAIN     PROC  FAR
            MOV   ECX,5                 ;broj = 5
            MOV   EBX,OFFSET  ARRAY-4
    MAIN1:  MOV   EAX,CS:[EBX+4*ECX]
            MOV   TEMP,EAX
            _FNUM TEMP,3
            _DISP 13
            _DISP 10
            DEC   ECX
            JNZ   MAIN1
            _EXIT                       ;povratak u DOS
   MAIN     ENDP
CODE  ENDS                              ;kraj kôdnog segmenta
```

```
END     MAIN                                    ;kraj programa
```

KORAK 5: Uneti, asemblirati i izvršiti program iz Primera 18-2. Kada je program izvršen prvo se prikazuje sadržaj elemenata polja _____ (koji je to sadržaj).

KORAK 6: Modifikovati program iz Primera 18-2 tako da prikazuje krajnji elemenat polja kao prvi. Na primer, ako se u KORAKU 5 prikazuje prvo elemenat 0, program treba da prikaže prvo element 4.

KORAK 7: Modifikovati program iz Primera 18-2 tako da prikazuje brojeve u formatu duple preciznosti (DQ).

KORAK 8: Kreirati program koji izračunava reaktansu kalema po obrascu (XL = 2*pi*f*L). Gde vrednost *f* odgovara frekvenci pobudnog sinusnog signala a *L* se odnosi na induktivnost kalema. Postupak izračunavanja ovog izraza je standardan. Program mora izračunati *XL*:

A. Za vrednosti *f* od 100Hz do 1000Hz u koracima po jedan Hz. Odrediti vrednost *XL* prezentirane u formi brojeva u pokretnom zarezu tipa obična preciznost. Izračunati vrednosti za *XL* kada je *L*= 0.01 i kada je za *L* = 0.001. Memorisati rezultate u dva različita polja, po jedno polje za svaku od vrednosti *L*.

B. Zatim, sabrati vrednost iz jednog polja i vrednost iz drugog polja i zatim smestiti rezultat u treće polje.

C. Na kraju, prikazati rezultate sume koje se nalaze u posegu od 60 do 62. Ove sume moraju se prikazati zajedno sa vrednošću frekvencije *f* za koje su izračunate.

Pitanja

1. Koji je inicijalno dodeljen obim registara sa kojim manipulišu mikroprcesori 80386, 80486, Pentium ... kada rade u 16- bitnom instrukcijskom režimu rada?
2. Koja je namena prefiksa kojim se odredjuje obim registra a koja prefiksa koji definiše obim adrese?
3. Koje se asemblerske direktive koriste za selekciju 16- i 32- bitnog instrukcijskog načina rada kod mikroprocesora 80386, 80486, Pentium ...?
4. Koji mikroprocesor koristi direktivu .586 ?
5. Napisati instrukciju kojom se puni registar EAX sadržajem memorijske lokacije obima reč locirane u kôdnom segmentu na lokaciji adresiramoj registrom EDX.
6. Objasniti razliku instrukcija uslovnog grananja kojim se specificira kratki i blizak skok.
7. Koji su faktori umnožavanja (skaliranja) dozvoljeni kod indeksnog adresiranja sa umnožavanjem?
8. Ako je CX = 1000, EDX = 00001000H, i ECX = 00000001H, koja se memorijska lokacija adresira instrukcijom ADD EAX,CS:[EDX+2*ECX] ?
9. Zašto se ne koristi instrukcija LOOPD u Primeru 18-1 i 18-2 ?
10. Napisati program koji generiše tabelu kvadratnih korena za brojeve u opsegu 1-99.

LABORATORIJSKA VEŽBA 19

Projektni zadaci

Uvod

Ova vežba zamišljena je kao skup problema koje treba rešavati pisanjem odgovarajućih programa. Pisanjem programa za jedan ili više problema značajno će se unaprediti vaša veština programiranja. Veliki broj stručnjaka smatraja da je najbolji način za usavršavanje znanja u vezi asemblerskog- ili programiranja na nekom višem programskom jeziku, rad na konkretnim problemima uz korišćenje stečenog znanja i iskustva na uradjenim primerima.

Predmet rada

1. Značajno unaprediti veštinu programiranja na asemblerskom jeziku kroz rad na nezavisnim projektima.
2. Proučavanje i istraživanje u domenu korišćenja DOS-ovih i BIOS funkcija u cilju rešavanja konkretnih problema.

Postupak rada

Projekat 1

Prvi problem vezan za programiranje tiče se grafike i predstavlja nastavak rada u odgovarajućoj prethodno uradjenoj vežbi. Takodje, za rešavanje problema koristi se i *floating-point* koprocesor, konkretno, u cilju rešavanja jedne jednačine. Makroi kao što su _DOT, _BLOCK i _CHAR prikazani u jednoj od prethodnih vežbi, trebaju biti od značajne koristi u toku rešavanja ovog problema.

Napisati program koji rešava jednaćinu *sin2x+cos2x* i iscrtava rezultat na beloj pozadini video displeja crnom linijom. Uvrstiti u prikaz i naslov problema uokviren cijan okvirom preko cele širine ekrana. Uneti različite vrednosti za sinus i kosinus tako da se na ekranu iscrtavaju krive za dve periode periodičnog signala (720°). Amplituda (visina slike) mora biti skalirana tako da ispunjava ekran po vertikali.

Projekat 2

U toku rada produbljuje se i proširuje znanje koje se odnosi na DOS funkcijske pozive koji su u vezi sa grafičkim prikazom. Takodje, neophodno je koristiti i DOS funkcijske pozive radi pristupa direktorijumu ili nekom disk-drajvu.

Kreirati program koji prikazuje sadržaj bilo kog direktorijuma na video displeju. Direktorijum se mora prikazati na ekranu i mora sadržati datum kreiranja i veličinu izraženu u bajtovima za svaku stavku (fajl) u direktorijumu. . U suštini, ovaj program treba da prikazuje sadržaj direktorijuma isto kao što to čini DOS (komanda DIR). Razlika treba da bude ta da vaš program prikazuje crna slova i brojeve na beloj pozadini u rezoluciji 640x480, sa 16 boja odnosno u režimu 12H.

Projekat 3

Ovaj projekat ukazuje na jedan primer jedne računarske igre koja koristi grafičke mogućnosti PC računara.

Kreirati program koji sumulira rad mašine za izdavanje karata. Mašina za izdavanje karata koristi specijalne ASCII karaktere za tip karte (tref=05H, pik=06H, herc=03H i karo=04H). Na displeju mora biti iscrtan okvir (23H). Makro _CHAR će prikazivati svaki od ovih kôdova kao tip karte i okrvir. Program treba istovremeno da prikaže po četiri karte koje se konstantno menjaju sve dok se ne pritisne dirka "blanko". Nakon toga se promene se zaustavljaju i prikazuje se dobitak ili gubitak kao i količina akumuliranog kredita (inicijalno 0 din). Za svaku novu igru ulog je jedan dinar. Iznosi gubitaka i dobitaka su.

100 din za četiri iste karte;
25 din za četiri karte različitog znaka u bilo kojoj kombinaciji boje;
25 din za bilo koje tri iste karte;
10 din za bilo koje dve iste;

2 din za bilo koji herc u poslednjoj poziciji desno.
Projekat bi se mogao usavršiti tako da prikazuje uvećane karte, odn., oznake za vrstu karte i okvir. To se može postići korišćenjem tehnika sličnih onima u _BLOCK makrou.

Projekat 4

U programima se veoma često koristi i miš. U ovom projektu primarne aktivnosti odnosiće se na rad sa mišem.
Kreirati program koji koristi miš i grafički režim 12H. Program treba da prikazuje paletu od 16 boja na dnu ekrana. Takodje, treba da omogućava da se u posebno predvidjenom i oivičenom crnom okviru iznad palete, kretanjem miša mogu crtati oblici sa prethodno izabranom bojom. Okvir za crtanje je dimenzija 400x300 tačaka i oivičen je belom linijom debljine 3 tačke. Levi taster miša koristi se za selekciju boje iz palete, a desni za crtanje u predvidjenom okviru i to držanjem pritisnutim ovaj taster i kretanjem miša.

Projekat 5

Pošto se prekidni programi pojavljuju i koriste kod velikog broja aplikacija, u ovom projektu se posvećuje više pažnje uvežbavanju ovih procedura i korišćenju TSR programa.
Kreirati TSR program koji nadgleda pojavu zahteva za prekid generisanog od strane internog časovnika (INT 8) a koji pobudjuje zvučnik signalom na polovini intervala izmedju dva cela sata, a dvaput isprekidanim tonskim signalom u trenutku najave celog sata. Koristiti INT 9 za prihvatanje pritisnute dirke, dozvoliti generisanje tonskog signala ALT B komandom i zabraniti generisanje tonskog signala ALT J komandom. Takodje, obezbediti mogućnost da /ON i /OFF komande ili direktive iz DOS-ove komandne linije omoguće promenu statusa kôda *hot*-dirke koja se prihvata sa tastatre. Na primer, /ON ALT-D treba da postavi tj. aktivira *hot*-dirku u kôd ALT-D. Omogućiti i prihvatanje kôda dirke ALT (*alternate*), LSHF (*left shift*), RSHF (*right shift*) i CNTL (*control*) kako bi bili kôdovi za *hot*-dirku.

Projekat 6

Iako na raspolaganju imamo puno softvera za rad sa bazama podataka, u okviru rada na ovom potprojektu biće instalirana i korišćena jedna baza podataka za skromnije potrebe.
Napisati program koji kreira fajl na disku sa slučajnim pristupom. Sadržaj fajla neka bude 4000 zapisa sa po 256 bajtova svaki. To je dovoljno da bi bilo smešteno na jedan HD flopy disk od 3,5 inča. Konfigurisati zapise tako da svaki sadrži sledeće informacije:
Polje 1: (jedna reč) broj ID zapisa;
Polje 2: (80 bajtova) ASCII polje koje sadrži naziv audio CD-a ili video kasete;
Polje 3: (40 bajtova) ASCII polje koje sadrži naziv izvodjača ili glumaca;
Polje 4: (1 bajt) numerički kôd koji se odnosi na rejting popularnosti filma; kod audio kaseta ovo polje se ne koristi;
Polje 5: (dva bajta) dužina zapisa u minutama ili sekundama (za audio CD-e);
Polje 6: (133 bajta) prostor za komentar koji se popunjava po želji.
Program koji radi sa ovakvim zapisima treba da omogućava dodavanje novog zapisa, brisanje zapisa i listanje sadržaja baze podataka u bilo kojem polju.

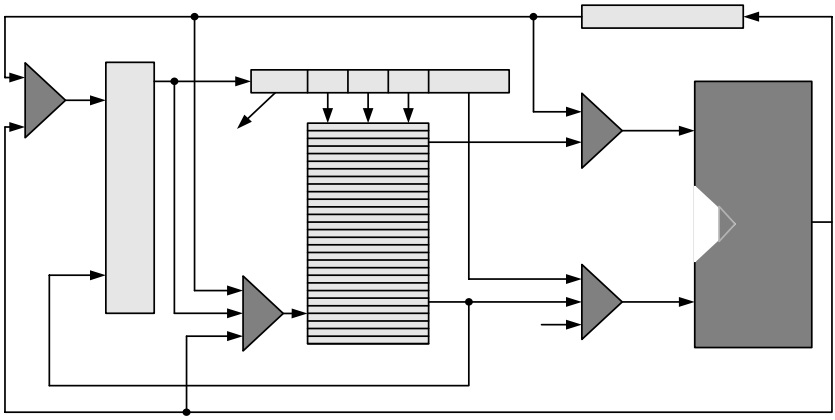
LABORATORIJSKA VEŽBA 20

Uputstvo za rad sa simulatorom za MIPS procesore - PCSpim

Uvod

Upoznavanje sa arhitekturom: MIPS procesori su tipični predstavnici RISC arhitektura. Svaki student, pre nego {to po~ne da kreira programe na asemblerskom jeziku, treba dobro da poznaje MIPS arhitekturu. Arhitektura ra~unara se defini{e: a) registrima koji su dostupni (vidljivi) programeru na asemblerskom jeziku; b) skupom instrukcija; c) adresnim na~inima rada; i d) tipovima podataka.

Pri savladavanju materije veoma je korisno imati sliku o stazi podataka kojom se opisuju klju~ne komponente i karakteristike MIPS arhitekture. Jedan pojednostavljeni dijagram MIPS-ove staze podataka prikazan je na slici 20-1.



Slika 20-1 Dijagram pojednostavljene staze podataka procesora MIPS

Osnovne funkcionalne komponente MIPS arhitekture su:

- upravlja~ka jedinica (*control unit - CU*)
- registarsko polje (*register file - RF*)
- aritmeti~ko-logi~ka jedinica (*ALU*)
- programski broja~ (*PC*)
- memorija (*MEM*)
- instrukcioni registar (*IR*)

Medjusobno povezivanje svih funkcionalnih komponenata, sa izuzetkom CU-a, ostvaruje se preko magistrala. Magistrala je, u su{tini, skup elektri~nih provodnika preko kojih se prenose razli~iti skupovi binarnih vrednosti. Najve}i broj magistrala kod MIPS arhitekture je obima 32 bita.

Detaljan opis principa rada i funkcije svake od pomenutih funkcionalnih komponenata studentima je dostupan u knjizi **RISC, CISC i DSP procesori** (pogl. 4 i 5) i **Zbirci zadataka za mikroporcesore i mikroračunare**. Na ovom mestu, radi kontinuiteta u pra}enju i savladavanju materije koja se odnosi na odgovaraju}e ve`be, ukaza}emo na neke klju~ne operativne karakteristike koje se odnose na kori{}enje MIPS procesora:

memorija

adresa

kont
log

1) **RF polje** ~ine 32 registra, svaki obima 32 bita. Usvojena konvencija kojom se specificira koji }e se od registara za koju namenu koristiti data je na sllici 20-2.

Registar	Broj	Kori{}enje
zero	0	Konstanta 0
at	1	Rezervisan za potrebe asemblera
v0 v1	2 3	Koriste se za prenos povratnih vrednosti iz poziva funkcija / poziva potprograma
a0 a1 a2 a3	4 5 6 7	Koriste se za prenos argumenata funkcijama (potprogramima)
t0 t1 t2 t3 t4 t5 t6 t7	8 9 10 11 12 13 14 15	Registri za privremeno promenljive (Ove registre koristi (menja) pozivni program (recimo, glavni), a po pravilu, ne bi trebalo da ih koriste (menjaju) pozvani programi (procedure ili funkcije))
s0 s1 s2 s3 s4 s5 s6 s7	16 17 18 19 20 21 22 23	Registri za pam}enje privremeno promenljivih (Registri koje pozvani program koristi za memorisanje, pozvana funkcija mora da ih zapamti i obnovi)
t8 t9	24 25	Registri za privremeno promenljive (Ove registre koristi (menja) pozivni program (recimo, glavni), a po pravilu, ne bi trebalo da ih koriste (menja) pozvani programi (procedure ili funkcije))
k0 k1	26 27	Rezervisani za kernel OS-a
gp	28	Pokaziva~ na globalnu memorijsku oblast
sp	29	Pokaziva~ magacina (*stack pointer*)
fp	30	Pokaziva~ okvira
ra	31	Povratna adresa za funkcijske pozive

Slika 20-2 Imenovanje i na~in kori{}enja registara RF polja

2) **Memoriju** treba shvatiti kao linearno polje registara pri ~emu se u svakoj lokaciji ~uva informacija obima re~, koja kod procesora MIPS iznosi 32 bita. Svaka lokacija u memoriji specificira se 32-bitnom memorijskom adresom.

3) Sve **instrukcije** kod MIPS arhitekture su obima 32 bita, a to zna~i da se PC nakon pribavljanja svake instrukcije inkrementira za 4.

4) Instrukcije mikroprocesora MIPS dele se na slede}e tri grupe:
   a) **regularne instrukcije za manipulisanje sa *integer* vrednostima**
   b**) makro-instrukcije** ili tzv. pseudoinstrukcije
   c) **regularne instrukcije za manipulisanje sa *floating-point* vrednostima**

Na slici 20-3 prikazan je skup regularnih (aktuelnih) instrukcija za manipulisanje sa *integer* vrednostima.

Ime	Sintaksa	Prostor/ Vreme
Add	add Rd,Rs,Rt	1/1
Add Immediate	addi Rd,Rs,Imm	1/1
Add Immediate Unsigned	addiu Rd,Rs,Imm	1/1
Add Unsigned	addu Rd,Rs,Rt	1/1
And	and Rd,Rs,Rt	1/1
And Immediate	andi Rd,Rs,Imm	1/1
Branch if Equal	beq Rs,Rt,Label	1/1
Branch if Greater Than or Equal to Zero	bgez Rs,Label	1/1
Branch if Greater Than or Equal to Zero and Link	bgezal Rs,Label	1/1
Branch if Greater Than Zero	bgtz Rs,Label	1/1
Branch if Less Than or Equal to Zero	blez Rs,Label	1/1
Branch if Less Than Zero and Link	bltzal Rs,Label	1/1
Branch if Less Than Zero	bltz Rs,Label	1/1
Branch if Not Equal	bne Rs,Rt,Label	1/1
Divide	div Rs,Rt	1/38
Divide Unsigned	divu Rs,Rt	1/38
Jump	j Label	1/1
Jump and Link	jal Label	1/1
Jump and Link Register	jalr Rd,Rs	1/1
Jump Register	jr Rs	1/1
Load Byte	lb Rt,offset(Rs)	1/1
Load Byte Unsigned	lbu Rt,offset(Rs)	1/1
Load Halfword	lh Rt,offset(Rs)	1/1
Load Halfword Unsigned	lhu Rt,offset(Rs)	1/1
Load Upper Immediate	lui Rt,Imm	1/1
Load Word	lw Rt,offset(Rs)	1/1
Load Word Left	lwl Rt,offset(Rs)	1/1
Load Word Right	lwr Rt,offset(Rs)	1/1
Move From Coprocessor 0	mfc0 Rd,Cs	1/1
Move From High	mfhi Rd	1/1
Move From Low	mflo Rd	1/1
Move To Coprocessor 0	mtc0 Rt,Cd	1/1
Move To High	mthi Rs	1/1
Move To Low	mtlo Rs	1/1

Multiply	mult Rs,Rt	1/32
Multiply Unsigned	multu Rs,Rt	1/32
Nor	nor Rd,Rs,Rt	1/1
Or	or Rd,Rs,Rt	1/1
Or Immediate	ori Rd,Rs,Imm	1/1
Return From Exception	rfe	1/1
Store Byte	sb Rt,offset(Rs)	1/1
Store Halfword	sh Rt,offset(Rs)	1/1
Shift Left Logical	sll Rd,Rt,sa	1/1
Shift Left Logical Variable	sllv Rd,Rt,Rs	1/1
Set on Less Than	slt Rd,Rt,Rs	1/1
Set on Less Than Immediate	slti Rd,Rt,Imm	1/1
Set on Less Than Immediate Unsigned	sltiu Rd,Rt,Imm	1/1
Set on Less Than Unsigned	sltu Rd,Rt,Rs	1/1
Shift Right Arithmetic	sra Rd,Rt,sa	1/1
Shift Right Arithmetic Variable	srav Rd,Rt,Rs	1/1
Shift Right Logical	srl Rd,Rt,sa	1/1
Shift Right Logical Variable	srlv Rd,Rt,Rs	1/1
Subtract	sub Rd,Rt,Rs	1/1
Subtract Unsigned	subu Rd,Rt,Rs	1/1
Store Word	sw Rt,offset(Rs)	1/1
Store Word Left	swl Rt,offset(Rs)	1/1
Store Word Right	swr Rt,offset(Rs)	1/1
System Call	syscall	1/1
Exclusive Or	xor Rd,Rt,Rs	1/1
Exclusive Or Immediate	xori Rd,Rt,Imm	1/1

Slika 20-3 Skup regularnih instrukcija za manipulisanje sa ***integer*** vrednostima

MIPS asembler sadr`i i skup makro (takodje nazvane sinteti~ke ili pseudo) instrukcija. Svaki put kada programer specificira makro instrukciju asembler, da bi ostvario ovaj zadatak, zamenjuje je odgovaraju}im skupom koga ~ine, u zavisnosti od tipa makro instrukcije, od jedne do nekoliko aktuelnih MIPS instrukcija. Na slici 20-4 prikazana je lista makro instrukcija MIPS procesora.

Ime	Sintaksa	Prostor/ Vreme
Absolute Value	abs Rd,Rs	3/3
Branch if Equal ti Zero	beqz Rs,Label	1/1
Branch if Greater Than or Equal	bge Rs,Rt,Label	2/2
Branch if Greater Than or Equal Unsigned	bgeu Rs,Rt,Label	2/2
Branch if Greater Than	bgt Rs,Rt,Label	2/2
Branch if Greater Than Unsigned	bgtu Rs,Rt,Label	2/2
Branch if Less Than or Equal	ble Rs,Rt,Label	2/2

Branch if Less Than or Equal Unsigned	bleu Rs,Rt,Label	2/2
Branch if Less Than	blt Rs,Rt,Label	2/2
Branch if Less Than Unsigned	bltu Rs,Rt,Label	2/2
Branch if Not Equal to Zero	bnez Rs,Label	1/1
Branch Unconditional	b Label	1/1
Divide	div Rd,Rs,Rt	4/41
Divide Unsigned	divu Rd,Rs,Rt	4/41
Load Address	la Rd,Label	2/2
Load Immediate	li Rd,value	2/2
Move	move Rd,Rs	1/1
Multiply	mul Rd,Rs,Rt	2/23
Multiply (with overflow exception)	mulo Rd,Rs,Rt	7/37
Multiply Unsigned (with overflow exception)	mulou Rd,Rs,Rt	5/35
Negate	neg Rd,Rs	1/1
Negate Unsigned	negu Rd,Rs	1/1
Nop	nop	1/1
Not	not Rd,Rs	1/1
Remainder Unsigned	remu Rd,Rs,Rt	4/40
Rotate Left Variable	rol Rd,Rs,Rt	4/4
Rotate Right Variable	ror Rd,Rs,Rt	4/4
Remainder	rem Rd,Rs,Rt	4/40
Rotate Left Constant	rol Rd,Rs,sa	3/3
Rotate Right Constant	ror Rd,Rs,sa	3/3
Set if Equal	seq Rd,Rs,Rt	4/4
Set if Greater Than or Equal	sge Rd,Rs,Rt	4/4
Set if Greater Than or Equal Unsigned	sgeu Rd,Rs,Rt	4/4
Set if Greater Than	sgt Rd,Rs,Rt	1/1
Set if Greater Than Unsigned	sgtu Rd,Rs,Rt	1/1
Set if Less Than or Equal	sle Rd,Rs,Rt	4/4
Set if Less Than or Equal Unsigned	sleu Rd,Rs,Rt	4/4
Set if Not Equal	sne Rd,Rs,Rt	4/4
Unaligned Load Halfword Unsigned	ulhu Rd,n(Rs)	4/4
Unaligned Load Halfword	ulh Rd,n(Rs)	4/4
Unaligned Load Word	ulw Rd,n(Rs)	2/2
Unaligned Store Halfword	ush Rd,n(Rs)	3/3
Unaligned Store Word	usw Rd,n(Rs)	2/2

Slika 20-4 Skup makro instrukcija

Kori{}enjem makro instrukcija pojednostavljuje se zadatak pisanja koda (programa) na asemblerskom jeziku, pa je ~esta praksa da se programeri ohrabre da koriste makro instrukcije.

Skup regularnih instrukcija koje se koriste za manipulisanje sa ***floating-point*** vrednostima prikazan je na slici 20-5.

Ime	Sintaksa
Absolute value double	abs.d Fd,Fs
Absolute value single	abs.s Fd,Fs
Add double	add.s Fd,Fs,Ft
Add single	add.d Fd,Fs,Ft
Branch if floating-point status flag is true	bclt label
Branch if floating-point status flag is false	bclf label
Compare and set flag if equal double	c.eq.d Fs,Ft
Compare and set flag if equal single	c.eq.s Fs,Ft
Compare and set flag if less than or equal double	c.le.d Fs,Ft
Compare and set flag if less than or equal single	c.le.s Fs,Ft
Compare and set flag if less than double	c.lt.d Fs,Ft
Compare and set flag if less than single	c.lt.s Fs,Ft
Convert single to double	cvt.d.s Fd,Fs
Convert integer to double	cvt.d.w Fd,Rs
Convert double to single	cvt.s.d Fd,Fs
Convert integer to single	cvt.s.w Fd,Rs
Convert double to integer	cvt.w.d Rd,Fs
Convert single to integer	cvt.w.s Rd,Fs
Divide double	div.d Fd,Fs,Ft
Divide single	div.s Fd,Fs,Ft
Load double (macro instruction)	l.d Fd,address
Load single (macro instruction)	l.s Fd,address
Load word into coprocessor 1	lwc1 Fd,offset(Rs)
Move double	mov.d Fd,Fs
Move single	mov.s Fd,Fs
Move from coprocessor 1	mdc1 Rd,Fs
Move double from coprocessor 1 (Macro Inst.)	mdc1.d Rd,Fs
Move to coprocessor 1	mtc1 Fd,Rs
Multiply double	mul.d Fd,Fs,Ft
Multiply single	mul.s Fd,Fs,Ft
Negate double	neg.d Fd,Fs
Negate single	neg.s Fd,Fs
Store double (macro instruction)	s.d Ft,address
Store single (macro instruction)	s.s Ft,address
Store word into coprocessor 1	swc1 Fd,offset(Rs)

Subtract double	sub.d Fd,Fs,Ft
Subtract single	sub.s Fd,Fs,Ft

Slika 20-5 Skup regularnih instrukcija koje se koriste za manipulisanje sa *floating-point* vrednostima

Napomena: Fd, Fs i Ft se odnose na *floating-point* registre koji su sastavni deo RF-FP polja koprocesora za numeri~ka izra~unavanja (numeri~ki koprocesor nije prikazan na slici 20-1). Asembler mikroprocesora MIPS kada je u pitanju imenovanje ovih registara koristi notaciju $fn gde se slovo f odnosi na FP registar a n na broj registra. Ukupno postoje 32 *floating-point* registra.

5) Lista **asemblerskih direktiva** prikazana je na slici 20-6.

.align n	Pode{avanje po~etne adrese narednog podatka na granici 2^n bajtova. Na primer, .align 2 pode{ava narednu vrednost na granici re~i, .align 0 isklju~uje automatsko pode{avanje .half, .word, .float i .double direktiva sve do naredne .data ili .kdata direktive
.ascii string*	Sme{ta niz u memoriju, ali ga ne zavr{ava znakom *zero*
.asciiz string*	Sme{ta niz u memoriju i zavr{ava ga znakom *zero*
.byte b1,...,bn	Sme{ta *n* 8-bitnih vrednosti u sukcesivne bajt memorijske lokacije
.data <addr>	Uzastopni podaci se memori{u u segmentu podataka. Ako postoji opcioni argument *addr*, uzastopni podaci se memori{u po~ev{i od adrese *addr*. Na primer: .data 0x00008000
.double d1,...,dn	Sme{ta *n floating-point* brojeva duple preciznosti u sukcesivne memorijske lokacije.
.extern Symb size	Deklari{e da }e podatak sme{ten na adresi Symb biti obima *size* bajtova i da je to globalna labela. Ova direktiva dozvoljava asembleru da memori{e podatak u delu segmenta podataka kome se efikasno pristupa preko registra $gp.
.float f1,...,fn	Memori{e *n* floating-point brojeva jednostruke preciznosti u sukcesivne memorijske lokacije
.globl Symb	Deklari{e da labela Symb bude globalna i da se mo`e referencirati (njoj obra}ati) od strane drugih fajlova
.half h1,...,hn	Memori{e *n* 16-bitnih veli~ina u sukcesivne memorijske lokacije tipa polure~i (polure~ = 16 bitova = 2

	bajta).
.kdata <add>	Naredni podaci se memori{u u segmentu podataka tipa kernel. Ako je prisutan opcioni argument ***addr*** naredni podaci se memori{u po~ev od adrese ***addr***.
.ktext <addr>	Uzastopni podaci se sme{taju u text (programskom) kernel segmentu. Kod SPIM-a ovi podaci mogu biti samo instrukcije ili re~i. Ako je prisutan opcioni argument ***addr*** naredni podaci se memori{u po~ev od adrese ***addr*** (npr. .ktext 0x80000080).
.space n	Dodeljuje ***n*** bajtova blanko (***space***) u teku}em segmentu (kod SPIM-a to mora biti segment podataka).
.text <addr>	Uzastopne stavke se sme{taju u korisni~ki programski (***text***) segment. Kod SPIM-a ove stavke mogu biti samo instrukcije ili re~i (videti direktivu .word). Ako je prisutan opcioni argument ***addr*** naredne stavke se memori{u po~ev od adrese ***addr*** (tj., .data 0x00400000).
.word w1,...,wn	Sme{ta ***n*** 32-bitnih veli~ina u sukcesivne memorijske lokacije tipa re~i.
.word w:n	Sme{ta 32-bitnu vrednost ***w*** u ***n*** sukcesivnih memorijskih re~i.

Slika 20-6 Lista asemblerskih direktiva

Napomene: *Nizovi su zatvoreni u duplim navodnicima ("). Za specijalne karaktere u nizu va`i C konvencija: newline: \n, tab: \t, quote: \". Opkodovi instrukcija su rezervisane re~i i ne mogu se koristiti kao labele. Labele moraju da se pojavljuju na po~etku linije a iza nje sledi ":". ASCII kod "back space", tj. blanko, ne podr`ava se od strane SPIM simulatora. Po definiciji se koriste brojevi brojne osnove 10. Ako brojevima prethodi 0x, oni se interpretiraju kao heksadecimalni. Saglasno tome, 256 i 0x100 ozna~ava istu vrednost.*

Kao {to se mo`e uo~iti sve asemblerske direktive identifikuje po~etni simbol "." kao na primer .align n, .asciiz string*, itd. Ra~unar ne izvr{ava direktive u toku rada programa. One se koriste od strane asemblera za formiranje odredjenih struktura podataka pre izvr{enja programa i od velike su mu pomo}i u fazi prevodjenja programa.

6) **Sistemsko ulazno-izlazni servisi** napisani su, u formi decimalnih U/I funkcija, od strane programera SPIM simulatora sa ciljem da olak{aju kreiranje programa na asemblerskom jeziku. Pristup ovim funkcijama ostvaruje se generisanjem softverskih izuzetaka. Poziv izuzetka se vr{i naredbom syscall. Postoje 10 razli~itih sistemskih servisa koji su prikazani na slici 20-7.

Servis	Kod u	Argument(i)	Rezultat(i)

	$v0		
Print Integer	1	$a0 = broj koji }e se {tampati	
Print Float	2	$f12 = broj koji }e se {tampati	
Print Double	3	$12 = broj koji }e se {tampati	
Print String	4	$a0 = adresa niza u memoriji	
Read Integer	5		broj koji se vra}a u $v0
Read Float	6		broj koji se vra}a u $f0
Read Double	7		broj koji se vra}a u $f0
Read String	8	$a0 = adresa ulaznog bafera u memoriji $a1 = veli~ina bafera (n)	
Sbrk	9	$a0 = iznos	
Exit	10		

Slika 20-7 Sistemsko ulazno-izlazni servisi

Sistemski poziv ***Read Integer*** ~ita celokupnu ulaznu liniju unetu preko tastature sve dok ne naidje na informaciju o novoj liniji. Karakteri koji slede nakon zadnje cifre decimalnog broja se ignori{u. Read String ima istu semantiku kao i Unix bibliote~ka rutina ***fgets***. Ona u~itava do ***n-1*** karaktera u bafer i zavr{ava niz sa nultim bajtom. Ako postoji manje od ***n-1*** karaktera u teku}oj liniji ***Read String*** u~itava podatke do karaktera nove linije i ponovo zavr{ava niz sa nultim karakterom. ***Read String*** }e prikazati na terminalu niz karaktera koji se nalaze u memoriji po~ev od likacije na koju ukazuje adresa sme{tena u registru $a0. [tampanje }e se zaustaviti kada se detektuje nulti karakter u nizu. ***Sbrk*** vra}a pokaziva~ bloka memorije koji sadr`i ***n*** dodatnih bajtova. ***Exit*** zavr{ava izvr{enje korisni~kog programa i vra}a upravljanje operativnom sistemu.
Pre poziva sistemskog ulazno-izlaznog servisa potrebno je da se registar $v0 postavi na vrednost odredjenog poziva, a prenos argumenata i rezultata funkcija prikazan je na slici 20-7.

Upoznavanje sa simulatorom SPIM: Sa ciljem da se upotpuni saznanje o arhitekturi ovih procesora u smislu asemblerskog jezika, odnosno skupa naredbi, načina adresiranja, skupa registara i meorijske organizacije, potrebno je najpre proučiti softversko sredstvo u vidu emulatora kojim se omogućava prividno izvr{avanje i praćenje toka programa, odnosno debagiranje, na bilo kojoj mašini koja se zasniva na mikroprocesoru koji nije iz MIPS RISC familije. Kroz ovu vežbu student treba da usvoji

`postupak i pravila rada sa programskim simulatorom za MIPS procesore - programom SPIM.`

SPIM simulator postoji u više verzija. Jedna od njih pod imenom *SPIM* jeste program koji se pokreće iz DOS-ove komandne linije i zahteva jedino alfanumerički displej za prikaz. Ovaj program radi poput većine programa ovog tipa: naime, korisnik otkuca liniju teksta, pritisnete taster <*enter*> i SPIM izvršava komandu. Nešto "humanija" verzija koja se izvršava pod operativnim sistemom X-Windows, kao varijantom Unix-a, naziva se *xspim*. Osnovna prednost ogleda se u tome što se, u ovom slučaju, zahteva bit-mapirani displej za prikaz. Ova verzija je znatno jednostavnija za učenje i korišćenje zbog činjenice da su njene komande uvek vidljive na ekranu. Takodje, registri mašine se kontinualno prikazuju nakon "izvršenja" svake instrukcije. Sledeća varijanta SPIM simulatora kompatibilna je operativnim sistemima Windows 3.1, Windows95 i WindowsNT. U ovoj vežbi, iz razloga što nam je trenutno najšire dostupan WindowsXP operativni sistem, predmet rada odnosiće se na varijantu simulatora pod nazivom PCSpim.

Predmet rada

Usvojiti postupak i pravila rada sa programskim simulatorom za MIPS procesore - programom SPIM.

Postupak rada

KORAK 1: Za pokretanje programa PCSpim za Windows, biramo programsku ikonu *PCSpim for Windows* slično kao kod startovanja bilo kog Windows programa. Na primer, u Windows XP, možete koristiti i *Start→Programs→PCSpim for Windows→PCSpim for Windows* iz Windows XP task bar-a. Kod Windows 3.1, selektujete aplikaciju iz *File Manager*-a.
Kada se PCSpim startuje, na ekranu se pojavljuje aplikacioni prozor kao na slici 20-8. *Aplikacioni prozor* programa se sastoji iz četiri dela:

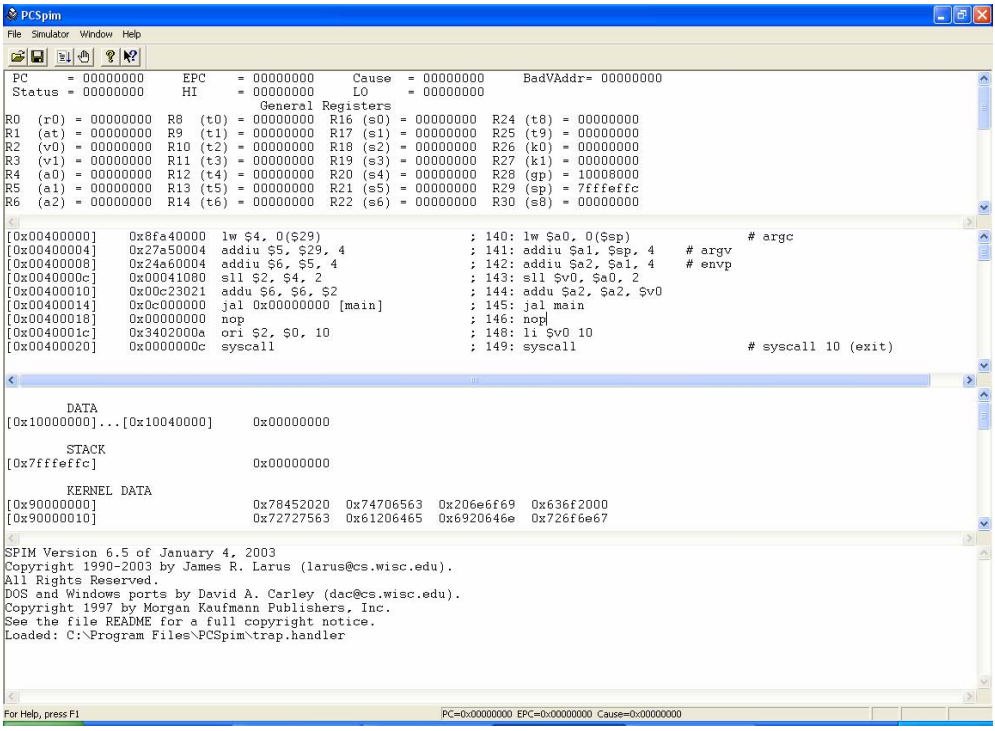
1. Sekcija na vrhu predstavlja *menu-bar*. *Menu-bar* omogućava selekciju *File* operacije, postavljanje konfiguracije - opcijom *Simulator*, selekciju načina i vrste prikaza - opcijom *Windows*, kao i dobijanje pomoćnih informacija - opcijom *Help*.
2. Sledeća sekcija ispod *menu-bar*-a je *toolbar*. *Toolbar* omogućuje brzi pristup putem miša do svih alata koje koristi PCSpim.
3. Najveća sekcija u sredini ekrana predstavlja sekciju prikaza. Postoje četiri različita prikaza: *Registers*, *Text Segment*, *Data Segment* i *Messages*. Da bi se promenio način prikaza selektuje se u meni-baru: *Windows→Til*e. Kada se program izvršava prvi put, svi prikazi su "prazni". U daljem tekstu opisan je svaki prozor prikaza.

- *Register* prozor prikazuje vrednosti svih registara CPU i MPU jedinica procesora MIPS;
- *Text Segment* prozor prikazuje instrukcije iz korisničkog programa, kao i sistemski kôd koji se puni (loaduje) uvek kada se PCSpim izvršava;
- *Data Segment* prozor ukazuje na podatke sa kojima manipuliše korisnički program, kako u programskoj memoriji, tako i u memoriji magacina.
- *Messages* prozor služi za ispis poruka PCSpim-a. Na primer, poruka o greškama, izveštaj o izvršavanju instrukcije, i sl.

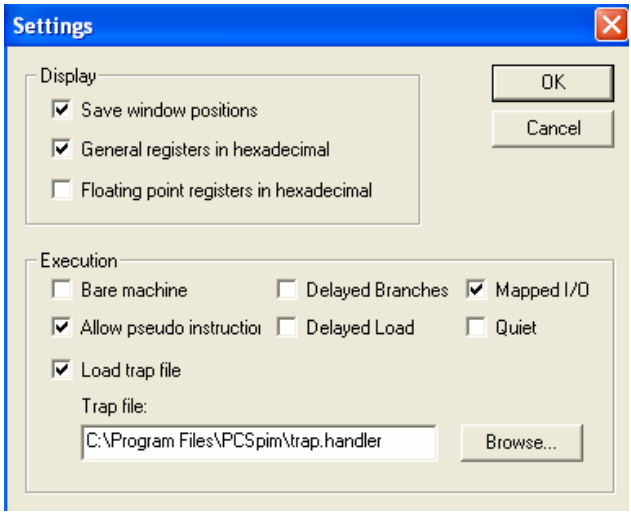
4. Sekcija *Status bar* nalazi se na dnu velikog prozora. Ova sekcija sadrži informacije o tekućim aktivnostima i statusu simulatora.

PCSpim pruža i druge mogućnosti koje se ipak redje koriste. Kada korisnik postane verziraniji u korišćenju PCSpim-a, dodatne informacije o nekim naprednijim opcijama korišćenja mogu se dobiti iz *online help*-a. To se postiže selekcijom *Help→Help_topics* u *menu-bar*-u.

KORAK 2: PCSpim poseduje grafički interfejs za uvid u tekuću konfiguraciju (svojstva) simulatora (slika 20-9). Za bilo koji korisnički program koji se loaduje, vrlo je važno postaviti korektnu konfiguraciju okruženja simulatora. PCSpim odredjuje kako loadovati i kako izvršavati program, zato su moguće greške u izvršenju ako konfiguracija okoline nije korektna. Ako konfiguracija okoline nije korektna i program ne može da se loaduje korektno, PCSpim omogućuje da postavite konfiguraciju okoline i reloadujete vaš program. Kada se *PCSpim* pokrene, ne moramo da unosimo bilo kakve parametre (oni se inače unose preko komandne linije). Medjutim, trebalo bi proveriti konfiguraciju simulatora bilo na *PCSpim*-ovom *status bar*-u, bilo kroz postavljanje *dialog box*-a pre loadovanja vašeg asemblerskog programa. U cilju provere/promene PCSpim okruženja (*settings*), u simulatorovom *dialog box*-u treba selektovati *Simulator→Settings* iz *menu bar*-a.



Slika 20-8. Korisnički interfejs PCSpim-a: glavni prozor



Slika 20-9. Dialog box programa PCSpim za konfigurisanje okruženja simulatora

Ukoliko, nakon loadovanja korisničkog programa, postoji potreba za izmenom PCSpim konfuguracije, program se, nakon takve izmene, treba ponovo loadovati (reloadovati) izborom *Simulator→Reload* iz *menu bar*-a.

U tekstu koji sledi biće opisana funkcija svih stavki iz konfiguracionog *dialog box*-a sa slike 20-9. Većina funkcija slična je kao kod *SPIM*-a, i odgovaraju verziji interfejsa programa koji nema grafičku podršku (verzija za komandnu liniju).

Display

Ovom opcijom može se selektovati forma prikaza sadržaja registara - heksadecimalna ili dekadna notacija. Ako su u *check-box*-u selektovani registri opšte namene i *floating point* registri, pojaviće se tzv. *check-marker* i sadržaj registara biće prikazan u heksadecimalnoj notaciji.

Save window positions

Kada se selektuje ova opcija, PCSpim će zapamtiti poziciju svojih prozora pri izlasku i obnoviće ih na istim lokacijama kod sledećeg startovanja PCSpim-a.

Bare machine

Ako je ova opcija selektovana, moguće je simulirati samo tzv. *bare* MIPS mašinu , tj. asemblerske sekvence bez pseudoinstrukcija i/ili dodatnih načina adresiranja koje omogućuje napredniji asembler.

Allow pseudo instructions

Asembliranje pseudoinstrukcija u programu dozvoljava se selekcijom opcije *allow pseudo instructions*. U protivnom, kada ova opcija nije selektovana, asembler ne prepoznaje pseudoinstrukcije.

Load trap file

Kada je ova opcija selektovana loaduju se standardni rukovaoc izuzecima (*exception handler*) i tzv. *startup* kôd. Kod pojave izuzetka, *SPIM* se grana na lokaciju 80000080h, koja mora da sadrži kôd za opsluživanje izuzetka. Tako|e, rukovaoc izuzetkom tipa *trap* sadrži *startup* kôd koji poziva rutinu *main*. Bez *startup* rutine, *SPIM* počinje izvršenje od instrukcije označene sa *__start*. Po

default mehanizmu, *trap* fajl se isporučuje sa *PCSpim*-om, medjutim, korišćenjem *Browse* tastera može se izabrati i neki drugi fajl.

Mapped I/O

Ako je ova opcija selektovana, dozvoljen je memorijski-preslikan ulaz/izlaz (U/I). Programi koji koriste SPIM-ove sistemske pozive (*syscalls*) za čitanje sa terminala, još uvek ne mogu koristiti memorijski-preslikan ulaz/izlaz.

Quiet

Kada je dozvoljena ova opcija, PCSpim neće dozvoljavati ispis poruka u posebnom prozoru pri pojavi izuzeteka. U protivnom, kada nije selektovana ova opcija, poruka o pojavi izuzeteka se ispisuje.

Delayed Branches i **Delayed Load**

Kada su ove opcije selektovane, kôd napisan na asemblerskom jeziku izvršavaće se na protočnoj implementaciji MIPS arhitekture.

KORAK 3: U cilju loadovanja korisničkog asemblerskog programa, najpre treba izabrati opciju *Open* iz *toolbar*-a. Alternativno tome, možemo u *menu bar*-u selektovati: *File→Open*. Poseban *dialog box* za otvaranje fajla javlja se kod selekcije odgovarajućeg asemblerskog fajla. Selektovati željeni fajl i pritisnuti na taster *Open* u *dialog box*-u. Ako konfiguraciono okruženje simulatora nije korektno postavljeno i fajl se ne može učitati, *PCSpim* će pružiti mogućnost za promenu konfiguracije okoline i automatski će izvršiti *reload* fajla.
U svakom trenutku, za slučaj da ste se predomislili, možete izabrati taster *Cancel*, na šta će PCSpim ukloniti *dialog box*. Kada učitamo asemblerski fajl, *dialog box* se uklanja i pojavljuju se prozori sa prikazom instrukcija i podataka. Ako to nije slučaj, treba promeniti izgled ekrana izborom *Windows→Til*e iz *menu bar*-a. Nakon ovih operacija, na ekranu bi trebalo da se vidi korisnički program u prozoru *Text segment*-a.
Svaka instrukcija u teksualnom segmentu prikazana je u liniji sličnoj sledećoj:

[0x00400000] 0x8fa40000 `lw $4,0($29);` 89: `lw $a0,0($sp)`

Prvi broj u liniji, smešten izmedju uglastih zagrada, jeste heksadecimalna memorijska adresa instrukcije. Drugi broj predstavlja heksadecimalni kôd instrukcije. Treća stavka jeste mnemonički opis instrukcije. Sve što sledi nakon tačke i zareza, jeste aktuelna linija iz vašeg fajla koja generiše instrukciju. Broj 89 je broj linije u tom fajlu. Ponekad, nema nikakve linije nakon tačke i zareza. To znači da je instrukciju generisao *SPIM* u procesu prevodjenja pseudoinstrukcije.
Za izvršavanje korisničkog programa, treba izabrati *Go* taster u *toolbar*-u. Alternativno, možete selektovati *Simulator→Go* iz *menu bar*-a. Korisnički program počeće da se izvršava. Za zaustavljanje izvršavanja programa, selektujte *Simulator→Break* iz *menu-bar*-a. To se može postići i kucanjem *Control-C* u trenutku kada je *PCSpim* aplikacioni prozor u prvom planu (*foreground*). Pri tome, pojavljuje se *dialog-box* koji pita da li želite da se izvršavanje nastavi. Selektovati *No* za prekid izvršavanja. Pre bilo kakve druge aktivnosti, možete pogledati na memoriju i registre kako bi se uverili u to šta program radi. Kada ste sigurni u to šta je program do tog trenutka uradio, možete nastaviti izvršavanje selekcijom *Simulator→Continue* ili završiti program sa *Simulator→Break* iz *menu-bar* -a.
Ako korisnički program čita sa terminala ili upisuje u terminal, *PCSpim* otvara novi prozor koji se naziva konzola (*console*). Za otvaranje *Console* treba selektovati *Window* meni i izabrati opciju *Console*. Svi karakteri koje program upisuje prikazuju se u ovom prozoru, a sve što program treba da čita sa terminala, mora biti prethodno upisano u ovaj prozor.

Pretpostaviti da korisnički program ne radi korektno onako kao što se očekuje. U tom slučaju *SPIM* poseduje dve mogućnosti debagiranja. Prva i najuobičajenija jeste koračno izvršavanje. Na ovaj način je omogućeno da se program izvršava tako da se u jednom trenutku izvršava jedna instrukcija. U tom smislu, selektovati *Simulator→Single_Step* za izvršenje samo jedne instrukcije. To isto postiže se i funkcijskim tasterom F10. Svaki put pravi se jedan korak u korisničkom programu, prikaz se ažurira i upravljanje opet vraća korisniku. Možete takodje izabrati i broj instrukcija koje će se izvršavati u jednom koraku. Za ovu mogućnost, selektovati *Simulator→Multiple_Step* kako bi izmenili osnovni korak od jedne instrukcije. U posebnom *dialog box* -u, koji se pri tome pojavljuje, selektovati broj instrukcija.

Šta činiti ako korisnički program dobro radi u dužem vremenskom intervalu na početku, a pre pojave logičke greške, tj. *bug*-a? I u tom slučaju možete koračno izvršavati program. Medjutim, to može potrajati prilično dugo. Bolja alternativa jeste korišćenje prekidnih tačaka. U prekidnoj tački *PCSpim* staje sa izvršavanjem programa odmah nakon završetka tekuće instrukcije. Za postavljanje prekidnih tačaka selektovati *Simulator→Breakpoints* iz *menu-bar* -a. *PCSpim* program prikazuje prozor *dialog box* -a sa dva segmenta. Viši segment je za unos adrese prekidne tačke, dok je drugi, niži segment zadužen za prikaz liste aktivnih prekidnih tačaka. Otkucati u prvom segmentu adresu instrukcije u kojoj želite da program stane sa izvršavanjem. Ili, ako instrukcija ima globalnu labelu (oznaku), dovoljno je samo otkucati ime ove oznake. Označene prekidne tačke su posebno pogodan način za zaustavljanje toka programa. Da bi postavili prekidne tačke, izaberati taster *Add*. Po završetku dodavanja prekidnih tačaka, birati opciju *Close* za izlaz iz *dialog box* -a. Nakon toga možete izvršavati program.

Kada simulator izvršava instrukcije prekidnih tačaka, *PCSpim* prikazuje *dialog box* sa adresom instrukcije i pita da li želite da nastavite izvršavanje. Na izbor *Yes* tastera program dalje nastavlja izvršavanje, a izborom tastera *No* program se zaustavlja. Kada želite da uklonite prekidnu tačku, birajte *Simulator→Breakpoints* iz *menu-bar* -a, kliknete na adresu koju uklanjate, i zatim birate taster *Remov*e.

Jednokoračno izvršavanje i umetanje prekidnih tačaka najverovatnije će pomoći u brzom nalaženju grešaka u korisničkom programu. Sada se postavlja pitanje kako pronadjene greške ukloniti. To se postiže u editoru u kome je izvorno unet korisnički program. Nakon izmena u ovom fajlu, u PCSpim simulatoru treba reloadovati program sa ekstenzijom *.s*. U tom smislu, biramo *Simulator→Reload<filename>* iz *menu bar* -a. To prouzrokuje brisanje memorije i registara PCSpim-a, i povratak procesora u stanje u koje je bio kada je prvi put startovan. Kada se simulator reinicijalizuje on loaduje najskorije korišćeni *.s* fajl.

KORAK 4: U nekim situacijama treba promeniti konfiguraciju okruženja simulatora iz komandne linije. Windows verzija SPIM-a prihvata sledeće opcije iz komandne linije:

-bare Simulira tzv. *bare* MIPS mašinu - mašinu bez pseudoinstrukcija i dodatnih adresnih režima koje predvidja assembler.

-trap Loaduje standardni *exception handler* i startup kôd. Po inicijalnom mehanizmu (*default*-u) je aktivan.

-noquiet Štampa poruku kada se pojavi izuzetak. Po inicijalnom mehanizmu (*default*-u) je aktivan.

-quiet Ne štampa poruke kod pojave izuzetaka.

-nomapped_io Zabranjuje mogućnost memorijski-preslikanog ulaza/izlaza. Po inicijalnom mehanizmu (*default*-u) je aktivan.

-mapped_io Dozvoljava mogućnost memorijski-preslikanog ulaza/izlaza. Programi koji koriste SPIM sistemske pozive za čitanje terminala još uvek ne mogu koristiti memorijski-preslikan ulaz/izlaz.

-file Loaduje i izvršava asemblerski program *a.asm*.

-execute Loaduje i izvršava asemblerski program u MIPS izvršnom fajlu *a.out*. Ova komanda može se primeniti samo kod izvršavanja SPIM-a na MIPS-zasnovanim mašinama.

-s <seg> size Postavlja inicijalnu veličinu memorijskog segmenta *seg* na veličinu od *size* bajtova. Memorijski segmenti nose sledeća imena: *text, data, stack, ktext* i *kdata*. Segment *text* sadrži instrukcije programa. Segment *data* sadrži podatke koje koristi program. Segment *stack* predstavlja magacin aktivan u toku izvršenja programa (*runtime stack*). Osim izvršavanja vašeg programa, SPIM izvršava i sistemski kôd koji upravlja prekidima i izuzecima. Ovaj kôd nalazi se u odvojenom delu adresnog prostora koji se naziva *kerne*l. Segment *ktext* sadrži instrukcije ovog kôda, dok segment *kdata* čuva podatke koje koristi pomenuti kôd. Pošto sistemski kôd koristi isti magacin kao i korisnički kôd, ne postoji *kstack* segment. Na primer, par argumenata *-sdata* 2000000 kreiraju korisnički segment veličine 2,000,000 bajtova.

-l <seg> size Postavlja ograničenje u rastu meorijskog segmenta do veličine od *size* bajtova. Memorijski segmenti koji mogu da rastu su *data, stack* i *kdata*.

-pseudo Omogućuje da ulazni asemblerski kod može da sadrži pseudoinstrukcije. Po inicijalnom mehanizmu (*default*-u) je aktivan.

-nopseudo Omogućava da ulazni asemblerski kôd ne može da sadrži pseudoinstrukcije.

-notrap Zabranjuje loadovanje fajla sa *trap*- programom. Ovaj program upravlja izuzecima. Kada se pojavi neki izuzetak, *SPIM* skače na lokaciju 80000080h, na kojoj mora da se nalazi program za opsluživanje izuzetka. Osim *trap*-programa, ovaj fajl sadrži i *startup* program koji poziva glavnu rutinu *main*. Bez *startup* rutine, *SPIM* počinje izvršavanje programa počev od instrukcije sa labelom *__start*.

KORAK 5: Slede}a programska sekvenca se koristi za izra~unavanje povr{ine kruga, p, ~iji je polupre~nik r = 10 cm

```
   .data                    # definicija segmenta podataka
r: .word 10                 #  na  adresi  r  nalazi  se  vrednost
polupre~nika kruga
p: .word 0                  # na adresu p treba upisati vrednost
povr{ine kruga
   . text                   # po~etak programskog segmenta
main:
   la $a0,r                 # adresa polupre~nika kruga sme{ta se
u $a0
   lw $t8,0($a0)            #  vrednost  polupre~nika  sme{ta  se  u
$t8
   li $t0,314156            #  punjenje  konstante  π  pomno`ene  sa
100 000
   mult $t8,$t8             # kvadriranje
   mflo $t1                 # LS 32 bita proizvoda sme{ta se u $t1
```

```
   mult $t1,$t0            # mno`enje skaliranom konstantom π
   mflo $s0                # LS 32 bita proizvoda sme{ta se u $s0
   li $t1,100000           # punjenje $t1 faktorom 100 000
   div $s0,$t1             # deljenje skaliranim faktorom
   mflo $s0                # 32-bitni koli~nik sme{ta se u $s0
   la $a1, p               #  adresa povr{ine kruga sme{ta se u
$a1
   sw $s0,0($a1)           #  vrednost povr{ine sme{ta se u
lokaciju p
   li $v0,10               # priprema za poziv sistemske funkcije
10
   syscall                 # poziv funkcije i izlaz iz programa
```

Da bi ste prošli kroz korake pisanja i izvršavanja PCSpim programa, najpre trebate uneti datu programsku sekvencu u tekst-editoru *Notepad* ili *MIPSter*. Kreirani izvorni programski fajl na asemblerskom jeziku snimiti pod imenom *primer.s*. U *Simulator* meniju pod stavkom *Settings* proveriti da li su isključene opcije *Delayed-branch* i *Delayed-load*. Ove opcije treba isključiti da bi se program izvršavao na standardnoj MIPS arhitekturi. Zatim u *File* meniju selektovati opciju *Open* ili samo kliknuti na prvu ikonu u *tool bar*-u. Selektovati *All Files* za *Files of type* i u~itati (loadovati) u simulator fajl kreiran na asemblerskom jeziku. U *Messages* prozoru pojavljuje se poruka o uspešnosti loadovanja fajla. U *Register* prozoru pratiti promenu sadr`aja registara RF polja i memorijskih lokacija nakon izvr{enja svake instrukcije.

KORAK 6: Usvojimo da *A* predstavlja vektor koga čini 100 elemenata, a *g*=1500 i *h*=1900 su promenljive. Takodje, usvojimo da se promenljive *g* i *h* prihvataju u registre *$s1* i *$s2*, bazna adresa vektora *A* prihvata u registar *$s3*, a rezultat se smešta na lokaciju promenljive *g*. U tekst-editoru *Notepad* ili *MIPSter* uneti najpre deklaraciju svih podataka na sledeći način:

```
.data
a: .word 0x00001234 0x00005678 0x00009abc 0x0000def0 0x12340000
.word 0x56780000 0x9abc0000 0xdef00000 0x00123400 0x00567800
.word 0x009abc00
g: .word 1500
h: .word 1900
```

Sledeći iskaz programskog jezika *C*:

```
   g=h+A[8];
```

preveden na asemblerski jezik mikroprocesora MIPS, ima oblik:

```
main:
   lw   $s3,a($zero)       # a[0]→$s3
   lw   $s2,h($zero)       # h→$s2
   lw   $t0,8*4($s3)       # a8→$t0
   add  $s1,$s2,$t0        # g = $s1 = h + a[8]
   sw   $s1,g($zero)       # g→G
   j    $ra
```

U istom fajlu, nakon podataka, uneti direktivu `.globl main`, a zatim i sekvencu počev od labele `main`. Zapamtiti program kao *pr_1.s*, na direktorijumu *PCSpim*. Zatim učitati asemblerski

program u simulator PCSpim. Izvršavati program korački (korišćenjem tastera F10). Uočiti efekat izvršenja svake instrukcije i dodati opis u izvorni fajl, u obliku komentara. Koje su instrukcije tipa pseudoinstrukcija? Kolika je heksadekadna vrednost rezultata?

Pitanja

1. Kakvo je sredstvo simulator SPIM, koja je njegova namena i koje mogućnosti ima?
2. Koje su opcije menija File?
3. Koje su opcije menija Windows?
4. Čemu služi Simulator meni?
5. Koje karakteristike okruženja podešavamo u Settings podmeniju?
6. Čemu služi fajl trap.handler?
7. Opisati liniju prikaza u svakom prozoru glavnog prikaza (kako je to učinjeno sa linijom Text segmenta).
8. Objasniti ulogu prekidnih tačaka u procesu debagiranja.
9. Na koji način otklanjamo bagove u korisnickom programu, nakon njihove detekcije?
10. Na koji način navodimo opcije programa PCSpim u komandnoj liniji?
11. Opisati način kreiranja asemblerskog fajla, kao i način njegovog loadovanja u simulator SPIM.

LABORATORIJSKA VEŽBA 21

Skup instrukcija mikroprocesora MIPS, pseudoinstrukcije

Uvod

Za ovu laboratorijsku vežbu neophodno je koristiti stečeno iskustvo vezano za rad sa programskim simulatorom PCSpim iz prethodne vežbe. Takodje, potrebno je i odredjeno poznavanje osnovnog skupa instrukcija za mikroprocesor MIPS, kratko opisanih u prethodnoj ve`bi, a detaljno datih u knjizi **RISC, CISC i DSP procesori** (pogl. 4 i 5) i **Zbirci zadataka za mikroporcesore i mikroračunare**.

Kroz unos i izvršavanje kraćih sekvenci na asemblerskom jeziku za MIPS, student u praksi potvrdjuje poznavanje mehanizama izvršenja instrukcija kroz efekat njihovog izvršavanja. Takodje, koristi se pojam ***pseudoinstrukcija*** i ističe značaj korišćenja ovog koncepta. Na primeru konkretnog problema poredjenja dve strukture podataka tipa vektor, student se upoznaje sa osnovnom konvencijom pisanja asemblerskih programa.

Upoznavanje sa pseudoinstrukcijama

MIPS asembler koristi skup makro (takodje nazvane sinteti~ke ili pseudo) instrukcije. Svaki put kada programer specificira makro instrukciju, asembler, da bi obavio zadatak, zamenjuje je skupom aktuelnih MIPS-ovih instrukcija. U ve`bi 20 definisan je skup makro instrukcija procesora MIPS. Da bi ukazali na na~in kori{}enja pseudoinstrukcija kao i njihovo prevodjenje u aktuelne instrukcije analizira}emo slede}e primere:

Primer 21-1:

Neka je data slede}a pseudoinstrukcija

```
   abs $s0,$t8
```

Ova pseudoinstrukcija od strane MIPS-ovog asemblera prevodi se u slede}e tri aktuelne instrukcije:

```
      addu $s0,$zero,$t8
      bgez $t8,pozitivan
      sub $s0,$zero,$t8
pozitivan:
```

Primer 21-2:

Slede}i pseudokod tipa aritmeti~ki izraz

```
   $s0=srt($a0*$a0+$a1*$a1)
```

gde je srt bibliote~ka funkcija kvadratni koren, prevodi se od strane asemblera u slede}u sekvencu instrukcija:

```
      mult $a0,$a0
      mflo $t0
      mult $a1,$a1
      mflo $t1
      add $a0,$t0,$t1
      jal srt
      move $s0,$v0
```

Predmet rada

Upoznati se sa osnovnim instrukcijama i funkcionalnim sekvencama instrukcija koje odgovaraju iskazima na višem programskom jeziku. Sagledati koncepciju celovitog programa, odnosno, način deklarisanja podataka (*data segment*) i programa (*text segment*).

Postupak rada

KORAK 1: Programeri, kod kreiranja programa, ~esto koriste kontrolnu strukturu tipa "if (uslov) then *do* (ovaj deo kôda) else *do* (ovaj deo kôda)". Ovaj HLL iskaz, za jedan konkretan slu~aj, ima slede}i oblik:

```
   if ($t8<0) then
      {$s0=0-$t8
      $t1=$t1+1}
   else
      {$s0=$t8
      $t2=$t2+1}
```

Prethodni iskaz se mo`e transformisati u kôd na asemblerskom jeziku mikroprocesora MIPS na slede}i na~in:

```
   bgez $t8,else        # ako je $t8 ve}e od ili jednako 0,
grananje na else
   sub $s0,$zero,$t8    # u $s0 se upisuje negativna vrednost
registra $t8
   addi $t1,$t1,1       # inkrementiranje $t1
   b next
else:
   mov $s0,$t8          # u $s0 se kopira $t8
```

```
   addi $t2,$t2,1       # inkrementiranje $t2
next:
```

U tekst-editoru *Notepad* ili *MIPSter* uneti potrebne naredbe i direktive za kreiranje izvornog programa koji koristi datu sekvencu asemblerskih instrukcija. Zapamtiti program kao *primer21-1.s*, na direktorijumu *PCSpim*. Zatim ga u~itati u simulator i pratiti promenu sadr`aja registara RF polja i memorijskih lokacija nakon izvr{enja svake instrukcije.

KORAK 2: Programeri koriste i kontrolnu strukturu "while (uslov) do (ovaj deo kôda). Slede}i HLL iskaz, za jedan konkretan slu~aj,

```
   while ($a1<$a2) do
      {$a1=$a1+1
      $a2=$a2-1}
```

transformi{e se u kôd na asemblerskom jeziku mikroprocesora MIPS na slede}i na~in:

```
while:
   bgeu $a1,$a2,done    # ako je $a1>=$a2 grananje na labelu done
   addi $a1,$a1,1       # $a1=$a1+1
   addi $a2,$a2,-1      # $a2=$a2-1
   b while              # grananje na labelu while
done:
```

U tekst-editoru *Notepad* ili *MIPSter* uneti potrebne naredbe i direktive za kreiranje izvornog programa koji koristi datu sekvencu asemblerskih instrukcija. Zapamtiti program kao *primer21-2.s*, na direktorijumu *PCSpim*. Zatim ga u~itati u simulator i pratiti promenu sadr`aja registara RF polja i memorijskih lokacija nakon izvr{enja svake instrukcije.

KORAK 3: for petlja je vrlo korisna kontrolna struktura. Slede}i HLL iskaz

```
   $a0=0;
   for ($t0=10; $t0>0; $t0=$t0-1) do
      {$a0=$a0+$t0}
```

mo`e se transformisati u kôd na asemblerskom jeziku mikroprocesora MIPS na slede}i na~in:

```
   li $a0,0             # $a0=0
   li $t0,10            # Inicijaliziranje broja~a petlje na 10
loop:
   add $a0,$a0,$t0
   addi $t0,$t0,-1      # Dekrementiranje broja~a petlje
   bgtz $t0,loop        # ako je $t>0 grananje na loop
done:
```

U tekst-editoru *Notepad* ili *MIPSter* uneti potrebne naredbe i direktive za kreiranje izvornog programa koji koristi datu sekvencu asemblerskih instrukcija. Zapamtiti program kao *primer21-3.s*, na direktorijumu *PCSpim*. Zatim ga u~itati u simulator i pratiti promenu sadr`aja registara RF polja i memorijskih lokacija nakon izvr{enja svake instrukcije.

KORAK 4: Kontrolna struktura switch predstavljena je slede}im iskazima:

```
      $s0=32;
top:  cout << "Input a value from 1 to 3"
      cin >> $v0
      switch ($v0)
         {case(1):{$s0=$s0<<1; break;}
          case(2):{$s0=$s0<<2; break;}
          case(3):{$s0=$s0<<3; break;}
         default: goto top;}
   cout << $s0
```

Data struktura se mo`e transformisati u kôd na asemblerskom jeziku mikroprocesora MIPS na slede}i na~in:

```
            .data
            .align   2
jumptable:  .word    top, case1, case 2, case3
prompt:     .asciiz  "\n\n Unos vrednosti od 1 do 3:"
            .text
top:
         li    $v0,4             # Kod za {tampanje niza
         la    $a0,prompt
         syscall
         li    $v0,5             #   Kod   za   ~itanje   integer
vrednosti
         syscall
         blez  $v0,top           # Difoltno za manje od jedan
         li    $t3,3
         bgt   $v0,$t3,top       # Difoltno za ve}e od tri
         la    $a1,jumptable     # U~itavanje adrese jumptable
         sll   $t0,$v0,2         #   Izra~unavanje   ofset   re~i
(mno`enje sa 4)
         add   $t1,$a1,$t0       # Izra~unavanje adrese
         addi  $t0,$t0,-1        # Dekrementiranje broja~a petlje

         bgtz  $t0,loop          # ako je $t0>0 grananje na loop
done:
```

U tekst-editoru *Notepad* ili *MIPSter* uneti potrebne naredbe i direktive za kreiranje izvornog programa koji koristi datu sekvencu asemblerskih instrukcija. Zapamtiti program kao *primer21-4.s*, na direktorijumu *PCSpim*. Zatim ga u~itati u simulator i pratiti promenu sadr`aja registara RF polja i memorijskih lokacija nakon izvr{enja svake instrukcije.

KORAK 5: Koriste}i isti tekst-editor kao u koraku 1, uneti navedeni asemblerski program za MIPS kojim se vrši poredjenje dva *integer* vektora. Koliko vrednosti je predvidjeno za testiranje u svakom vektoru? U kom obliku i gde se upisuje rezultat komparacije? Izvršavati program korač no i dopuniti izvorni fajl komentarima sa efektom izvršenja svake instrukcije.

```
   .text
   .globl main
main:
subu $sp,$sp,32
sw   $ra,20($sp)
sw   $fp,16($sp)
addu $fp,$sp,32
lw   $a0,size
la   $a1,array1
la   $a2,array2
jal  compare
lw   $ra,20($sp)
lw   $fp,16($sp)
addu $sp,$sp,32
j    $ra
```

a0 = duzina vektora
a1 = startna adresa vektora 1
a2 = startna adresa vektora 2

```
compare:
subu  $sp,$sp,32
sw    $ra,20($sp)
sw    $fp,16($sp)
addiu $fp,$sp,32
loop:
beq   $a0,$0,done
lw    $t0,0($a1)
lw    $t1,0($a2)
bne   $t0,$t1,no
addiu $a1,$a1,4
addiu $a2,$a2,4
addi  $a0,$a0,-1
b     loop
no:
ori  $v0,$0,1
b    return
done:
ori  $v0,$0,0
return:
lw   $31,20($sp)
lw   $fp,16($sp)
addu $sp,$sp,32
j    $31
.data
size:   .word 5
array1: .word 1 2 3 4 5
array2: .word 1 2 3 4 5
```

KORAK 6: Kao što smo već pomenuli, odredjeni skup asemblerskih instrukcija predstavlja proširenje osnovnog skupa instrukcija, odnosno tzv. pseudoinstrukcije. Od strane asemblerskog

prevodioca pseudoinstrukcije se najpre predstavljaju u formi kraćih sekvenci osnovnog skupa instrukcija, pa tek nakon toga prevode u mašinski kôd. Koristeći opise osnovnih instrukcija asemblera, odrediti kojim se sekvencama zamenjuju sledeće instrukcije:

a) *abs,* a) *mul,* c) *neg,* d) *rem,* e) *rol,* f) *seq,* g) *bgeu,* h) *bgt, i* i) *ld*.

Proveriti da li PCSpim podržava ove pseudoinstrukcije. Za one koje podržava, uporediti sa vašim rešenjem.

Pitanja

1. Objasniti pojam bare code mašina.
2. Kojom direktivom pocinje segment podataka, a kojom segment programa?
3. Odrediti kljucnu instrukciju u primeru pr_1.asm kojom se pribavlja ciljni clan vektora. Koji je to, po redosledu, element vektora?
4. Nacrtati dijagram toka primera za komparaciju dva vektora. Naznaciti labele iz programa na samom dijagramu.
5. Uociti i istaci sve pseudoinstrukcije u datim primerima.
6. Koriste}i definicije instrukcija datih u ve`bi 20 prevesti svaki od slede}ih pseudokod izraza u MIPS asemblersku sekvencu:

(a) `t3 = t4+t5-t6;`
(b) `s3 = t2/(s1-54321);`
(c) `sp = sp-16;`
(d) `cout << t3;`
(e) `cin >> t0;`
(f) `a0 = &array;`
(g) `t8 = Mem(a0);`
(h) `Mem(a0+16) = 32768;`
(i) `cout << "Hello World";`
(j) `if (t0<0) then t7=0-t0 else t7=t0;`
(k) `while (t0!=0) { s1=s1+t0; t2=t2+4; t0=Mem(t2) };`
(l) `for (t1=99; t1>0; t1=t1-1) v0=v0+t1;`
(m) `t0 = 2147483647 - 2147483648;`
(n) `s0 = -1*s0;`
(o) `s1 = s1*a0;`
(p) `s2 = srt(s0`^{2}`+56)/a3;`
(q) `s3 = s1-s2/s3;`
(r) `s4 = s4*8;`
(s) `s5 = 7*s5;`

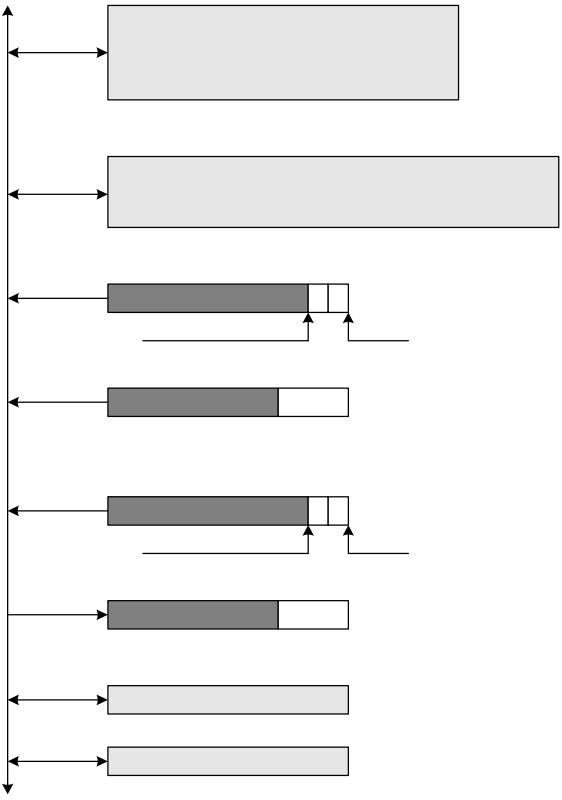
LABORATORIJSKA VEŽBA 22

Memorijsko-preslikani ulaz-izlaz

Uvod

Mikroprocesor MIPS komunicira sa ulazno-izlaznim (U/I) podsistemom koriste}i tehniku memorijsko-preslikani U/I. Uobi~ajeno je da kod projektovanja mikrora~unarskih sistema baziranih na memorijsko-preslikanom U/I podsistemu projektant bude taj koji rasporedjuje i dodeljuje prostor kako memorijskom tako i U/I podsistemu. PCSpim simulator je projektovan na taj na~in da je memoriji dodeljen prostor po~ev od lokacije 0x00000000 pa sve do 0xfffeffff, a U/I podsistemu adrese po~ev od 0xffff0000 pa sve do 0xffffffff. To zna~i da kod MIPS arhitekture, bilo koja Load ili Store instrukcija ~ija je

efektivna adresa ve}a ili jednaka adresi 0xffff0000 ne}e mo}i da pristupi glavnoj memoriji. Ove adrese su rezervisane za pristup registrima koji pripadaju U/I uredjajima. Povezivanje U/I kontrolera na U/I magistralu obavlja se kako je to prikazano na slici 22-1.



Slika 22-1 Ulazno-izlazna magistrala kod MIPS-a

Svakom U/I registru pridru`ena je odgovaraju}a adresno dekoderska logika. Operacija ~itanje (upis) obavlja se na slede}i na~in: Mikroprocesor postavlja na magistralu va`e}u adresu registra U/I kontrolera kome `eli da pristupi. Nakon toga, u zavisnosti od toga da li se obavlja operacija Load ili Store generi{e se upravlja~ki signal ***Read*** ili ***Write*** iza ~ega sledi prenos podataka.

Memorijsko-preslikani U/I podsistem u okviru PCSpim-a

U okviru PCSpim simulatora izvedena je simulirana verzija memorijsko-preslikanog tastaturnog kontrolera i memorijsko-preslikanog displej kontrolera.
Koriste}i PCSpim simulator studenti }e ste}i iskustvo u pisanju kôda koji se odnosi na prenos karaktera, jedan za drugim, preko fizi~kih U/I uredjaja. Kôd koji komunicira sa fizi~kim uredjajem na ovom nivou naziva se **drajver**.

Brzina prenosa podataka preko U/I uredjaja je znatno sporija u odnosu na brzinu sa kojom mikroprocesor MIPS izvr{ava instrukcije. Zbog ovoga je neophodno uskladiti rad CPU-a sa jedne i U/I podsistema sa druge strane, {to se izvodi ubacivanjem ka{njenja.
Kao {to se vidi sa slike 22-1 postoje dva registra koji su pridru`eni tastaturi i dva registra koji su pridru`eni displeju. Kod realnih sistema postoje i drugi U/I uredjaji kakvi su DMA disk kontroler i DMA Ethernet kontroler koji se povezuju na U/I magistralu, ali u konkretnom slu~aju oni nisu deo PCSpim simulatora.

Komuniciranje sa kontrolerom tastature

Dva 32-bitna registra koji su pridru`eni tastaturi nazivamo ***Receiver-control*** i ***Receiver-data***. Ovim registrima dodeljene su adrese 0xffff0000 i 0xffff0004, respektivno. Komuniciranje sa tastaturom metodom ***polling*** ostvaruje se testiranjem stanja LS bita (***ready*** bit) ***Receiver-control*** registra i ~itanjem LS 8-bitova ***Receiver-data*** registra. Kada se na tastaturi pritisne dirka odgovaraju}i 8-bitni ASCII kod koji odgovara simbolu dirke le~uje se u ***Receiver-data*** registar, a ***ready*** bit se postavlja na "1".
Slede}a programska sekvenca prikazuje odgovaraju}i MIPS kôd koji se odnosi na memorijsko-preslikani pristup registrima kontrolera tastature.

```
      Li $a3,0xffff0000    # bazna adresa memorijsko-preslikanog
terminala
ckready:
      Lw $t1,0($a3)        # ~itanje Receiver-control registra
      Andi $t1,$t1,1       # izdvajanje bita ready
      Beqz $t1,ckready     #  ako nije pritisnuta dirka go to
ckready
      Lw $t0,4($a3)        # pribavi karakter sa tastature
```

Bazna adresa memorijsko-preslikanog U/I prostora se upisuje u registar $a3. Tri instrukcije koje ~ine petlju kod implementirane ***polling*** tehnike prenosa koriste se za testiranje stanja bita ***ready***. Kada je ***ready*** bit postavljen na "1", ASCII kôd iz ***Receiver-data*** registra puni se u $t0 izvr{enjem instrukcije Lw.Nakon ove sekvence, sa programske ta~ke gledi{ta, sledi prenos podataka u odgovaraju}i prihvatni bafer (ovaj detalj nije prikazan u programu), a zatim sledi ponovno testiranje stanja bita ***ready*** ako se `eli primiti novi karakter.
Naglasimo da MIPS programer mo`e samo da ~ita podatak iz ***Receiver-data*** registra kao i samo da ~ita stanje bita ***ready*** u ***Receiver-control*** registru. Instrukcije koje upisuju u ove lokacije nemaju efekat (pravo pristupa radi upisa nije dozvoljeno).

Komuniciranje sa tastaturnim kontrolerom

Dva 32-bitna registra pridru`ena displeju nazivaju se *Transmitter-control* i *Transmitter-data*. Fizi~ke adrese koje su dodeljene ovim registrima su 0xffff0008 i 0xffff000c, respektivno. Komuniciranje sa displejom se ostvaruje testiranjem stanja LS bita (*ready* bit) *Transmitter-control* registra, a zatim pam}enjem ASCII koda LS 8 bitova *Transmitter-data* registra. Ne smemo upisati vrednost u *Transmitter-data* registar sve dok displej nije spreman da ga prihvati. Slede}i primer prikazuje MIPS kôd koji se odnosi na memorijsko-preslikani pristup registrima kod displej kontrolera.

```
   Li   $a3,0xffff0000        # bazna adresa memor.-preslikanog
terminala u $a3
XReady:
   Lw   $t1,8($a3)            # ~itanje Transmitter-control registra
   Andi $t1,$t1,1             # izdvoji bit ready
   Beqz $t1,XReady            # ako je bit ready≠0 go to XReady
   Sw   $t0,12($a3)           # po{alji karakter displeju
```

Kada displej kontroler detektuje da je karakter upisan u *Transmitter-data* registar, logika kontrolera postavlja bit *ready* na "0". MIPS programer mo`e samo da upisuje u *Transmitter-data* registar, i da ~ita *ready* bit iz *Transmitter-control* registra.

Sat realnog vremena

Da bi se re{ili neki od zadataka koji su sastavni deo ove ve`be, student mora, na simuliranom PCSpim okru`enju svog ra~unara, da odredi koliko se MIPS-ovih instrukcija izvr{ava u sekundi. U slede}oj programskoj sekvenci na labeli "broji_nanize", registar $s0 se puni na vrednost 2 500 000. U "waitloop" postoje dve instrukcije. To zna~i da pre nego {to se napusti petlja (waitloop) izvr{i}e se pet miliona instrukcija. Na primer, ako program raportira da je proteklo vreme 5s to zna~i da se u proseku svake sekunde izvr{ava 1 000 000 instrukcija. Struktura programa je slede}eg oblika:

```
# Funkcionalni opis: Raporti o proteklom vremenu svakih pet
sekundi u trajanju od jedne minute
   .data                      # deklaracija sekcije podataka
por:  .asciiz  "\n Proteklo vreme = "
   .text
main:                         # po~etak programske sekcije
   Li   $s1,0
broji_nanize:
   Li   $s0,2500000           # vremenski faktor
waitloop:
   Addi $s0,$s0,-1            # petlja ~ekanja
   Bnez $s0,waitloop
   Addi $s1,$s1,5
   Li   $v0,4                 # {tampanje poruke
```

```
   La   $a0,por
   Syscall                 # {tampanje iznosa
   Addi $t0,$s1,-60
   Bnez $t0,broji_nanize
   Li   $v0,10
   Syscall                 # izlaz
```

Za slu~aj da program, kada se izvr{ava na va{em ra~unaru, ne raportira ta~no proteklo vreme, tada je potrebno podesiti "vremenski faktor" i ponoviti proceduru.

Predmet rada

Upoznati se sa osnovnim instrukcijama i funkcionalnim sekvencama instrukcija koji se odnosi na prenos karaktera, jedan za drugim, preko fizi~kih U/I uredjaja. Sagledati koncepciju celovitog programa, odnosno, način deklarisanja podataka (*data segment*) i programa (*text segment*).

Postupak rada

KORAK 1: Napisati programsku sekvencu za predaju slede}eg ASCII niza "ABCDEF" displeju preko registra ***transmitter data***. Po~etna adresa ASCII niza u memoriji je PODACI. Programsku sekvencu zavr{iti funkcijskom naredbom syscall. Program otkucan u tekst editoru ***Notepad*** zapamtiti kao *primer22-1.s*, na direktorijumu *PCSpim*. Zatim ga u~itati u simulator i pratiti promenu sadr`aja registara RF polja i memorijskih lokacija nakon izvr{enja svake instrukcije.

KORAK 2: Napisati programsku sekvencu za predaju slede}eg ASCIIZ niza "ABCDEFZ" displeju preko registra ***transmitter data***. Po~etna adresa ASCIIZ niza u memoriji je PODACI_1. Programsku sekvencu zavr{iti funkcijskom naredbom syscall. Program otkucan u tekst editoru ***Notepad*** zapamtiti kao *primer22-2.s*, na direktorijumu *PCSpim*. Zatim ga u~itati u simulator i pratiti promenu sadr`aja registara RF polja i memorijskih lokacija nakon izvr{enja svake instrukcije.

KORAK 3: Koriste}i memorijsko preslikani U/I napisati funkciju koja }e obavljati isti zadatak kao i sistemska funkcija ***Print*** String. Uzeti da je adresa niza u memoriji NIZ1 i da je niz ograni~en do maksimalno 10 karaktera.

KORAK 4: Koriste}i memorijsko preslikani U/I napisati funkciju koja }e obavljati isti zadatak kao i sistemska funkcija ***Read String***. Uzeti da je adresa ulaznog bafera u memoriji NIZ2 i da je niz ograni~en do maksimalno 10 karaktera.

KORAK 5: Kreirati programsku sekvencu koja }e se na va{oj PC ma{ini koristiti za odbrojavanje vremena od jedne sekunde. Nakon isteka ovog vremena na displeju prikazati odgovaraju}u vrednost. Brojanje obaviti po modulu 60.

KORAK 6: Napisati MIPS program koji na svaku sekundu analizira tastaturu. Ako postoji karakter u ***Receiver-data*** registru poslati ga na ***Transmitter-data*** registar za displej.

Pitanja

1. Objasniti razliku izmedju memorijsko preslikanog i izdvojenog U/I-a.
2. Kako je od strane SPIM simulatora izvr{ena podela memorijskog prostora na U/I prostor i memoriski prostor?
3. Kako se naziva kod za komuniciranje sa fizi~kim uredjajem?
4. Kada bi vam se dodelila uloga projektanta koje bi adrese dodelili disk kontroleru, a koje Ethernet kontroleru?
5. Objasninti smisao i na~in kori{}enja petlji ~ekanja. Na koji na~in se procenjuje njihovo vreme izvr{enja?

LABORATORIJSKA VEŽBA 23

Protočna implementacija

Uvod

U prethodnim ve`bama analizirali smo kreiranje programa na asemblerskom jeziku za pojednostavljeni model MIPS arhitekture koji izvr{ava instrukcije strogo sekvencijalno. Da bi projektovali procesor koji }e raditi takore}i pet puta br`e u odnosu na pojednostavljeni model koristi se tehnika nazvana *proto~na obrada*. Ovu tehniku implementira}emo i na na{u MIPS arhitekturu. Kod petostepene proto~ne implementacije postoje pet razli~ita hardverska stepena od kojih svaki obavlja specificiranu aktivnost, a instrukciji je potrebno pet ciklusa da prodje kroz sve stepene. Funkcije koje obavljaju ova pet stepena su slede}e:

1. Pribavljanje instrukcija (*Instruction Fetch - IF*) – pribavlja se instrukcija iz ke{ memorije i puni u instrukcioni registar. Programski broja~ se inkrementira za 4.
2. Pribavljanje operanada (*Operand Fetch - OF*) – pribavljaju se vrednosti Rs i Rt iz RF polja. Ako se radi o instrukciji *Branch* i uslov grananja je ispunjen tada se u PC puni ciljlna adresa grananja.
3. Izvr{enje (*Execute - EX*) – ALU obavlja aritmeti~ku ili logi~ku funkciju i puni rezultantni registar, tj. stepen u kome see obavlja sabiranje radi izra~unavanja efektivne adrese *Load* i *Store* instrukcija.
4. Pristup memoriji (*Memory Access - MA*) – ako se obavlja instrukcija *Load* tada se ~ita podatak iz ke{a za podatke. Kada je instrukcija tipa *Store* tada se podatak upisuje u ke{. Ina~e, prenosi se rezultat u rezultantni registar prema stepenu *Write Back* (u ovoj fazi se ne vr{i upis u registar).

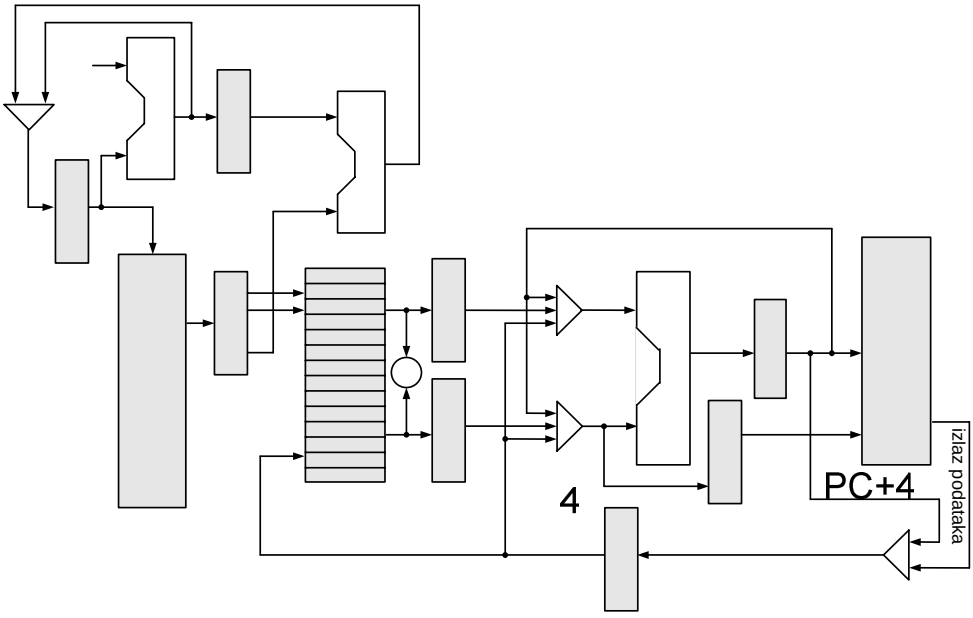
5. Upis rezultata (***Write Back - WB***) – sme{ta se vrednost u specificirani registar registarskog polja. Ako prva instrukcija upisuje rezultat u registarsko polje u toku stepena ***Write Back*** tada }e ~etvrta instrukcija proto~nog stepena mo}i simultano da ~ita podatak iz registarskog polja.

Protočna staza podataka

Dijagram petostepene staze podataka sa svim glavnim gradivnim blokovima prikazan je na slici 23-1. IR je proto~ni registar koji ~uva rezultate stepena IF. Registri Rs i Rt ~uvaju rezultate stepena OF. Registar ***rezultat*** ~uva rezultat stepena EX. ***WB*** registar ~uva podatke koji su pro~itani iz memorije u slu~aju kada je instrukcija tipa ***Load***, a za sve ostale instrukcije to je proto~ni registar koji ~uva vrednost koja je bila u rezultantnom registru na kraju prethodnog ciklusa. Upis u ***WB*** registar koji pripada registarskom polju vr{i se u toku prve polovine taktnog perioda, a ~itanje sadr`aja iz registara RF polja vr{i se u toku druge polovine. Time se obezbedjuje da instrukcija koja se nalazi u drugom stepenu proto~nog sistema mo`e da pro~ita novu vrednost u toku druge polovine istog taktnog ciklusa.

Hazardi po podacima

Termin hazard po podacima se odnosi na slede}u situaciju. Pretpostavimo da imamo tri sekvencijalne instrukcije ***x***, ***y*** i ***z*** koje ulaze u proto~ni sistem i neka je ***x*** prva instrukcija, iza nje sledi ***y*** i na kraju ***z***. Ako je rezultat izra~unat od strane instrukcije ***x*** potreban instrukcijama y i z, tada ka`emo da }e do}i do hazarda po podacima. Hardversko re{enje za ovaj problem se sastoji u uvodjenju puteva sa premo{}avanjem u okviru staze podataka ma{ine tako da ~ak i kada rezultati nisu jo{ upisani u RF polje (aktivnost u okviru stepena ***WB***), potrebna informacija }e se prosledjivati od rezultantnog registra, ili ***WB*** registra, prema ulazu ALU-a.



Slika 23-1 Dijagram petostepene staze podataka

Jedan tip hazarda ne mo`e se re{iti hardverom za premo{}avanje. To je situacija kada nakon instrukcije ***Load*** (lw ili lb) neposredno sledi instrukcija koja koristi vrednost koja se pribavlja iz memorije. Jedino re{enje za ovaj problem sastoji se u preuredjenju asemblerskog koda tako da instrukcija koja sledi ***Load*** nije instrukcija koja koristi vrednost koja se pribavlja iz memorije. Kod ove situacije re{enje se nalazi u ubacivanju slotova zaka{njenja. Ako postoje}e instrukcije u algoritmu ne mogu da se preurede da bi se re{io hazard po podacima, tada se odmah nakon instrukcije ***Load*** ubacuju operacije ***Nop***. Na ovaj na~in se obezbedjuje da se stanje ma{ine ne}e promeniti ako ***Nop*** prodje kroz proto~ni sistem, ali posti`emo da se ostvari dovoljno vremensko ka{njenje tako da naredna instrukcija u sekvenci ima va`e}u kopiju vrednosti koju treba da ~ita iz memorije a upisana je od strane instrukcije ***Load***.
Asembler prevodi ***Nop*** mnemonik u 0x00000000 {to je takodje binarni kod za instrukciju sll $0,$0,0.

Upravljački hazardi

Upravlja~ki hazardi prate izvr{enje instrukcije ***Branch*** ili ***Jump***. Kao {to smo prethodno napomenuli u toku prvog ciklusa vr{i se pribavljanje instrukcija. U toku drugog taktnog ciklusa ~itaju se vrednosti iz RF polja kako je to specificirano Rs i Rt poljima instrukcije ***Branch***, i u toku istog taktnog ciklusa vrednosti ova dva operanda se kompariraju kako bi se utvrdilo da li su oni jednaki ili nisu. Ako je uslov grananja ispunjen tada znakovno pro{irenom neposrednom polju, koje je deo ***Branch*** instrukcije

dodaje se vrednnost PC-a. Dok se obavlja ova aktivnost pribavlja se instrukcija koja u programskoj sekvenci sledi nakon ***Branch*** ili ***Jump***. Za slu~aj da je uslov grananja ispunjen, tada u toku tre}eg taktnog ciklusa, izra~unata ciljna adresa grananja koja se nalazi u PC-u bi}e iskori{}ena za pribavljanje naredne instrukcije. Ali dok se ova instrukcija pribavlja, instrukcija koja neposredno sledi ***Branch*** ili ***Jump*** nalazi}e se u drugom proto~nom stepenu. Imaju}i u vidu da grananje ne mo`e imati efekat sve dok se ne zavr{i procesiranje u drugom stepenu ka`emo da dolazi do efekta zaka{njenog grananja. U su{tini, ne postoji na`in za eliminisanje kontrolnih hazarda, tako da se re{enje ogleda u prepoznavanju efekta koga }e ***Branch*** ili ***Jump*** instrukcija imati nakon instrukcija koje u proto~noj obradi slede ***Branch*** ili ***Jump***. Ovo zna~i da programer mora da prepozna ovu ~injenicu i da preuredi program ili kod. Obi~no programe rpronalazi neku MIPS instrukciju u kodu koja nije zavisna od grananja i sme{ta je u slotu zaka{njenja (***delay slot***) koji sledi nakon instrukcije ***Branch***. Za slu~aj da postoje}e instrukcije u algoritmu ne mogu da se preurede kako bi se re{io kontrolni hazard, tada odmah nakon instrukcije ***Branch*** sledi operacija ***Nop***.
Najve}i broj komercijalnih MIPS asemblera detektuje hazarde jo{ na nivou izvornog asemblerskog koda. Ovi asembleri poku{avaju da preurede kod ako je to mogu}e, a za slu~aj da nije oni automatski insertuju ***Nop*** instrukcije kako bi se eliminisali hazardi. PCSpim asembler ne poseduje ove osobine tako da, kada se program PCSpim izvr{ava u proto~nom re`imu rada (***pipeline mode***), tada je zadatak programera da elimini{e kako hazarde po podacima tako i kontrolne hazarde.

PCSpim opcija za simuliranje protočne implementacije

PCSpim poseduje opciju da simulator radi u proto~nom re`imu rada. Ova opcija se uklju~uje selekcijom ***Settings*** i klikom na ***Delayed Branches*** i ***Delayed Load*** u padaju}em meniju. Izborom ove opcije programer sti~e iskustvo u pisanju asemblerskih programa koji se izvr{avaju na proto~noj implementaciji MIPS arhitekture. Treba uo~iti da u re`imu rada korak po korak svaka instrukcija nakon ***Branch*** ili ***Jump*** se uvek izvr{ava. Kod sistemskih poziva se ne uvodi ka{njenje tako da ove delove programa ne treba menjati.

Suma celobrojnih vrednosti

Analizira}emo sada jedan program (vidi sliku 23-2) koji se koristi za odredjivanje sume celobrojnih vrednosti od 1 do N, gde je N vrednost koja se unosi preko tastature. Ako poku{amo da izvr{imo program pomo}u PCSpim-a u proto~nom re`imu rada ustanovi}emo da on zavr{ava sa petljom koja se beskona~no puta izvr{ava. Pogledajmo sada ~etiri instrukcije koje po~inju na labeli Loop. Prime}uje se da instrukcija koja sledi nakon instrukcije ***Branch*** (bnez $v0, Loop) puni registar $v0 vredno{}u 4. Kada se PCSpim izvr{ava u proto~nom rezimu rada, ova

instrukcija }e se uvek izvr{avati jeer se ona nalazi u slotu zaka{njenja (*delay slot*). Zbog ovoga $v0 ne}e nikad biti 0 pa }e se program izvr{avati u petlji sve dok se ne javi prema{aj.
Uo~imo da samo dve instrukcije treba da se promene kako bi u~inili da ovaj program korektno funkcioni{e u proto~nom re`imu rada. Ako je mogu}e, uvek treba smestiti korisnu instrukciju u slotu zaka{njenja. S obzirom da je instrukcija *Branch* zavisna od rezultata koji se kreira dekrementiranjem $v0, mi ne mo`emo smestiti ovu instrukciju u slot zaka{njenja. Malim razmi{ljanjem zaklju~ujemo da mo`emo ostvariti ovo ako inicijaliziramo $v0 na vrednost N, pa zatim smestimo instrukciju koja akumulira sumu u slot zaka{njenja. Na slici 23-3 prikazana je varijanta izvr{nog dela programa koja korektno vr{i sumiranje celobrojnih vrednosti, dok deo programa koji se odnosi na definiciju segmenta podataka ostaje isti kao na slici 23-2.

```
   .data
prompt:   .asciiz "\n Uneti vrednost za N = "
rezultat: .asciiz "Zbir integer brojeva od 1 do N je "
cao:      .asciiz "\n **** Kraj - Prijatan dan ****"
          .globl main
          .text
main:
        li   $v0,4          # sistemski poziv za Print String
        la   $a0,prompt     # napuni adresu prompt-a u $a0
        syscall             # od{tampaj prompt poruku
        li   $v0,5          # sistemski poziv za Read Integer
        syscall             # u~itaj vrednost N u $v0
        blez $v0,kraj       # go to kraj if $v0<=0
        li   $t0,0          # postavi registar $t0 na 0
Loop:
        add  $t0,$t0,$v0    # suma integer-a u registar $t0
        addi $v0,$v0,-1     # sabiraj integer-e po~ev od najve}e
vrednosti
        bnez $v0,Loop       # go to Loop if $v0!=0
        li   $v0,4          # poziv sistemske funkcije za Print
String
        la   $a0,rezultat   # napuni adresu poruke u $a0
        syscall             # {tampaj niz
        li   $v0,1          # poziv sistemske funkcije za Print
Integer
        move $a0,$t0        # kopiraj vrednost koja se {tampa u
$a0
        syscall             # od{tampaj sumu integer-a
        b    main           # go to main
kraj:   li   $v0,4          # poziv sistemske funkcije za Print
String
        la   $a0,cao        # napuni adresu poruke u $a0
        syscall             # od{tampaj niz
```

```
        li   $v0,10         # zavr{i izvr{enje programa, i
        syscall             # povratak upravljanja sistemu
```

Slika 23-2 Program za odredjivanje sume celobrojnih vrednosti od 1 do N

```
.text   # Za protocnu implementaciju
main:
        li   $v0,4          #
        la   $a0,prompt     #
        syscall
        li   $v0,5          #
        syscall
        blez $v0,kraj       #
        move $t0,$v0        # *** uo~iti na ovom mestu izmenu
Loop:
        addi $v0,$v0,-1     #
        bnez $v0,Loop       #
        add  $t0,$t0,$v0    # *** uo~iti na ovom mestu izmenu
        li   $v0,4          #
        la   $a0,rezultat   #
        syscall
        li   $v0,1          #
        move $a0,$t0        #
        syscall
        b    main           #
kraj:   li   $v0,4          #
        la   $a0,cao        #
        syscall
        li   $v0,10         #
        syscall             #
```

Slika 23-3 Program za korektno odredjivanje sume celobrojnih vrednosti od 1 do N kada PCSpim radi u proto~nom re`imu rada

Funkcijski poziv u protočnom režimu rada

Na slici 23-4 prikazan je program koji odredjuje sumu pozitivnih i sumu negativnih vrednosti. Zatim je na slici 23-5 pokazano kako se taj program mora modifikovati da bi se korektno izvr{io na proto~noj implementaciji. Napravljena je samo jedna promena u glavnom programu. Jedan od argumenata koji je prene{en funkciji sume predstavlja obim vektora, i u konkretnom slu~aju iznosi ~etiri. Uo~imo da je instrukcija li $a1,4 sme{tena u slotu zaka{njenja nakon instrukcije jal suma. Bez sumnje, instrukcija za u~itavanje neposredne vrednosti je u proto~nom sistemu pre bilo koje instrukcije koja je pridru`ena funkcijom suma, tako da se parametar koji ukazuje na veli~inu vektora stvarno prenosi funkciji.

U okviru funkcije suma neophodno je izvr{iti odredjene modifikacije. Po~e}emo sa time gde treba smestiti instrukcije u slotovima zaka{njenja. Prebacivanje dve instrukcije na labelu negative predstavlja dibru osnovu za po~etak. Mo`emo takodje prebaciti instrukciju add $v0,$v0,$t0 koja neposredno prethodi instrukciji b Loop u njen slot zaka{njenja. Poslednja modifikacija ima efekat da dovede dve instrukcije grananja jednu do druge, a to je definitivno problem. Moramo da nadjemo neku instrukciju koju bi smestili u slot zaka{njenja nakon instrukcije bltz $t0,negative. Postoji samo jedan prihvatljiv kandidat: addi $a1,$a1,-1. Vidite li za{to? Mi trebamo da smestimo neku instrukciju u slot koji sledi instrukciju lw $t0,0($a0) zbog hazarda po podacima, a instrukcija addi $a0,$a0,4 je jedina instrukcija koju mo`emo smestiti u ovaj slot. Na kraju, naglasimo da se instrukcija nop ubacuje nakon instrukcije jr $ra jer naredna lokacija u memoriji mo`e da sadr`i bilo koju instrukciju, a sigurno ne `elimo da program izvr{ava neku proizvoljnu instrukciju svaki put kada se obavlja povratak (***return***) ka glavnom (***main***) programu. Ovaj program je prikazan na slici 23-5.

```
   .data
array:  .word  -4, 5, 8, -1
msg1:   .asciiz "\n Zbir pzitivnih vrednosti = "
msg2:   .asciiz "\n Zbir negativnih vrednosti = "
        .globl main
        .text
main:
        li   $v0,4          # sistemski poziv funkcije Print
String
        la   $a0,msg1       # napuni adresu msg1 u $a0
        syscall             # od{tampaj niz
        la   $a0,array      # inicijaliziraj adresu array (vektora
podataka)
        li   $a1,4          # napuni du`inu vektora u $a1
        jal  suma           # poziv funkcije suma
        move $a0,$v0        # suma pozit. vred. se vra}a u $v0 i
kopira u $v0
        li   $v0,1          # poziv sistemske funkcije za Print
Integer
        syscall             # od{tampaj sumu pozitivnih vrednosti
        li   $v0,4          # poziv sistemske funkcije za Print
String
        la   $a0,msg2       # napuni adresu msg2 u $a0
        syscall             # od{tampaj niz
        li   $v0,1          # poziv sistemske funkcije za Print
Integer
        move $a0,$v1        # suma neg. vred. se vra}a u $v1 i
kopira u $v0
        syscall             # od{tampaj sumu negativnih vrednosti
        li   $v0,10         # zavr{i izvr{enje programa i
```

```
        syscall             # povratak upravljanja sistemu
###################################
# $a0: Pokaziva~ na Array
# $a1: Broj elemenata
###################################
suma:   li    $v0,0             # inicijaliziraj $v0 na 0
        li    $v1,0             # inicijaliziraj $v1 na 0
loop:
        blez  $a1,return        # if $a1<=0 go to return
        addi  $a1,$a1,-1        # dekrementiraj broja~ petlje
        lw    $t0,0($a0)        # dobavi elem. vektora sa adrese
array u $t0
        addi  $a0,$a0,4         # inkrementiraj pokaziva~ na
slede}u re~
        bltz  $t0,negative      # if $t0<0 go to negative
        add   $v0,$v0,$t0       # dodaj sumi pozitivnih vrednosti
        b     loop              # granaj se na loop za drugu
iteraciju
negative:
        add   $v1,$v1,$t0       # dodaj sumi negativnih vrednosti
        b     loop              # granaj se na loop za drugu
iteraciju
return:
        jr   $ra                # povratak
```

Slika 23-4 Program za odredjivanje sume pozitivnih i sume negativnih vrednosti

```
        .globl main
        .text   # Promenjen za proto~nu implementaciju
main:
        li   $v0,4          #
        la   $a0,msg1       #
        syscall
        la   $a0,array      #
        jal  suma
        li   $a1,4          #**** ovde uo~iti modifikaciju
        jal  suma
        move $a0,$v0        #
        li   $v0,1          #
        syscall
        li   $v0,4
        la  $a0,msg2
        syscall
        li   $v0,1
        move $a0,$v1        #
        syscall
        li   $v0,10
        syscall
###################################
```

```
# $a0: Pokaziva~ na Array
# $a1: Broj elemenata
###################################
suma:   li   $v0,0
        li   $v1,0
loop:
        blez $a1,return     #
        lw   $t0,0($a0)
        addi $a0,$a0,4      #
        bltz $t0,negative   #
        addi $a1,$a1,-1     #**** ovde uo~iti modifikaciju
        b    loop           #
        add  $v0,$v0,$t0    #**** ovde uo~iti modifikaciju
negative:
        b    loop           #
        add  $v1,$v1,$t0    #**** ovde uo~iti modifikaciju
return:
        jr   $ra
        nop                 #**** ovde uo~iti modifikaciju
```

Slika 23-5 Program za korektno odredjivanje sume pozitivnih i sume negativnih vrednosti kada PCSpim radi u proto~nom re`imu rada

Primer gde se operacije Nop ne mogu izbeći

U odredjenim situacijama ne postoji način da se izbegne korišćenje Nop operacija. U ovom primeru ukazaćemo na ubacivanje tri *Nop* operacije u modifikovani kod. Da li to znači da će se modifikovani kod duže izvršavati? Odgovor je ne, jer za protočnu mašinu prosečna brzina izvršenja je skoro pet puta veća od brzine neprotočne. Dodatne Nop operacije, kada prolaze kroz protočni sistem, dovode do povećanja kašnjenja. U suštini one ne obavljaju koristan posao, tako da svaka dodatno uvedena Nop operacija imaće samo efekat na brzinu izvršenja programa od strane protočne mašine (a ne i na semantiku programa), i obično, njen efekat je veoma mali. Efektivna brzina izvršenja biće znatno manja od pet puta (za petostepeni protočni sistem) u odnosu na rad neprotočne mašine.

Funkcija MinMax čiji je kod prikazan na slici 6 koristi se za pretraživanje vektora 32-bitnih reči i vraćanje (*return*) minimalne i maksimalne vrednosti vektora. Veći broj modifikacija je neophodno obaviti da bi se ova funkcija izvršavala u protočnom režimu rada. Prva stvar koju treba da uradimo je da se pronadje neka instrukcija koja će se smestiti u slot zakašnjenja na kraju prvih pet instrukcija. Nakon pažljive analize, može se zaključiti da instrukcija `move $v1,$v0` je verovatno najbolji kandidat za umetanje u slot zakašnjenja. Naglasimo da ova instrukcija koristi vrednost koja se čita iz memorije. Bila bi greška da se premesti instrukcija `addiu $a0,$a0,4` u slot zakašnjenja jer ćemo završiti sa `move $v1,$v0` odmah nakon instrukcije *Load*, a ovo neće biti dobro jer podatak koji se čita iz memorije a smešta se u $v0 biće dostupan tek nakon još jednog taktnog ciklusa.

U narednom programskom bloku postoje pet instrukcija koje počinju na labeli `Loop`. U ovom kodnom bloku postoje dve *Branch* instrukcije, tako da je neophodno da se odrede instrukcije koje

treba smestiti u slot zakašnjenja za svaku instrukciju *Branch*. Moramo da nadjemo neke instrukcija koje nisu zavisne od vrednosti koje se čitaju iz memorije a smeštaju u *delay* slot. Mi smo prinudjeni da ubacimo Nop nakon prve *Branch* instrukcije i da promenimo redosled zadnje dve instrukcije u bloku.

Na kraju programa, kod labele `chk` postoje dve instrukcije. Prva instrukcija dekrementira brojač petlje a druga instrukcija testira dekrementiranu vrednost. U suštini, da bi se program korektno izvršavao, neophodno je ubaciti *Nop* u slot zakašnjenja ove *Branch* instrukcije. Ipak pažljivijom analizom može se nadji put da se eliminiše ova Nop operacija.

Kompajleri prevode HLL kod u kod na asemblerskom jeziku. Konačna faza u procesu prevodjenja naziva se generisanje koda. Svako ko kreira program za generisanje koda mora biti svestan specijalnih zahteva koji se odnose na protočnu implementaciju.

```
##############################################
# MinMax ($a0: Adresa, $a1: Broj elemenata, $v0: Minimum, $v1:
Maksimum)
##############################################
   .globl MinMax
   .text
MinMax:
   lw $v0,0($a0)
   addiu $a0,$a0,4
   move $v1,$v0
   addi $a1,$a1,-1
   blez $a1,ret
loop:
   lw $t0,0($a0)
   addi $a0,$a0,4
   bge $t0,$v0,next
   move $v0,$t0
   b chk
next:
   ble $t0,$v1,chk
   move $v1,$t0
chk:
   addi $a1,$a1,-1
   bnez $a1,loop
ret:
   jr $ra
```

Slika 23-6 Program za za pretraživanje vektora 32-bitnih reči i nalaženje minimalne i maksimalne vrednosti

```
.text              # Promenjen za proto~nu implementaciju
MinMax:
   lw $v0,0($a0)     #
   addiu $a0,$a0,4   #
   addi $a1,$a1,-1   #
   blez $a1,ret      #
   move $v1,$v0      #**** ovde uo~iti modifikaciju
loop:
   lw $t0,0($a0)     #
```

```
   addi $a0,$a0,4    #
   bge $t0,$v0,next  #
   nop               #**** ovde uo~iti modifikaciju
   b chk             #
   move $v0,$t0      #**** ovde uo~iti modifikaciju
next:
   ble $t0,$v1,chk   #
   move $v1,$t0      #
chk:
   addi $a1,$a1,-1   #
   bnez $a1,loop     #
   nop               #**** ovde uo~iti modifikaciju
ret:
   jr $ra            #
   nop               #**** ovde uo~iti modifikaciju
```

Slika 23-7 Modifikovani program za korektno pretraživanje vektora 32-bitnih reči i nalaženje minimalne i maksimalne vrednosti kada PCSpim radi u proto~nom re`imu rada

Predmet rada

Upoznati se sa načinom modifikacije osnovnih instrukcijama i funkcionalnih sekvenci instrukcija kako bi se one korektno izvršavale na simulatoru SPIM koji radi u protočnom režimu.

Postupak rada

KORAK 1: Pokazati na koji način treba modifikovati sledeću sekvencu instrukcija:

```
Add   $4,$5,$6
Beq   $1,$2,40
Lw    $3,300($0)
```

da bi ona korektno izvršavala na simulatoru SPIM koji radi u protočnom režimu.

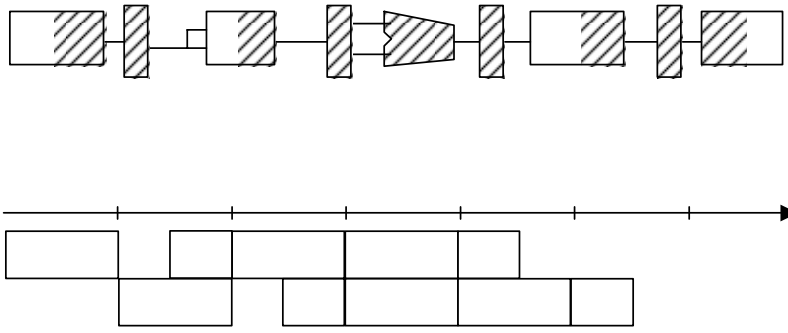
KORAK 2: Kreirati funkciju na asemblerskom jeziku mikroprocesora MIPS koja se koristi za pretra`ivanje niza X du`ine N 32-bitnih elemenata kako bi se odredilo koliki broj elemenata niza je deljiv sa 4. Adresu niza, ADR, preneti funkciji preko registra $a0, a broj elemenata niza, N, preko registra $a1. Rezultat funkcije vratiti preko registra $v0. Simulator PCSpim radi u proto~nom re`imu rada.

KORAK 3: a) Napisati funkciju na asemblerskom jeziku mikroprocesora MIPS koja }e vratiti N-ti element Fibonacci-evog niza. Vrednost N preneti funkciji preko registra $a0, a N-ti Fibonacci-ev broj E vratiti u registar $v0. Ako je N ve}e od 46 dolazi do prema{aja rezultata tako da je u tom slučaju potrebno vratiti vrednost 0 u registar $v0.
b) Prikazati sekvencu kojom se obavlja poziv funkcije i povratak 10-og elementa sekvence.

Napomena: Prvih nekoliko brojeva Fibonacci-eve sekvence su 0, 1, 1, 2, 3, 5, 8, ...
Simulator PCSpim radi u proto~nom re`imu rada.

Pitanja

1. Koje funkcije obavljaju stepeni IF, OF, EX, MA, WB kod petostepene proto~ne implementacije?
2. Objasniti na koji na~in se mogu re{iti hazardi po podacima?
3. Struktura protočne staze podataka prikazana je na slici 24-1a a način protočnog izvršenja instrukcije na slici 24-1b.



Slika 23-8. Struktura protočne staze podataka i protočno izvršenje

Razmotriti izvršenje programa koga čine 100 instrukcija tipa *Lw* (ne obraćati pažnju da li ovaj kôd ima smisla). Kod ovog kôda svaka instrukcija je zavisna od instrukcije koja joj prethodi. Koliki će biti CPI ako se program izvršava na protočnoj stazi podataka sa slike 23-8.

IM

L

a) strukture proto

0 1

IF Re

IF

Prevodjenje iskaza sa HLL-a na asemblerski jezik

Uvod

Usvoji}emo da PCSpim simulator radi u proto~nom re`imu rada (*pipelime mode*). Kao {to smo naglasili u prethodnoj ve`bi zbog postojanja strukturnih, upravlja~kih i zavisnosti po podacima dolazi do zastoja u radu sistema. Kao posledica zastoja ne mo`e se ostvariti efektivna propusnost od jedne instrukcije po taktnom ciklusu. Pametni kompilatori, u velikom broju slu~ajeva, uspe{no re{avaju konflilkte u radu procesora, na taj na~in {to automatski preuredjuju kod, u slotove zaka{njenja ubacuju Nop operacije, itd. PCSpim simulator nema izvedeno te mogu}nosti tako da najve}i deo optimizacije koda je zadatak koji treba da obavi programer, tj. student u toku izvodjenja ve`bi. Na ovaj na~in student sti~e dragoceno iskustvo koje je od va`nosti ne samo za pisanje optimalnog programa nego i za bolje upoznavanje principa rada arhitekture procesora MIPS.
Cilj ove ve`be je dvostruki: 1) da student ru~no (bez asistencije kompajlera) prevede iskaze sa vi{eg programskog jezika na asembler mikroprocesora MIPS, i b) da izvr{i optimizaciju prevedenih programskih sekvenci na asemblerski jezik kako bi minimizirao broj zastoja, a maksimizirao propusnost u izvr{enju instrukcija.

Opšte napomene

Obično, kompilator uspeva da popuni oko 50% *branch delay* slotova sa korisnim instrukcijama. Ako je protočnost dublja od 5 stepeni, tada je potrebno popuniti veći broj *branch delay* slotova, što je ponekad teško uraditi, tj. izvesti.
Kod odredjenih RISC arhitektura, instrukcijama tipa *Load*, *Store* i *Branch* potreban je i više od jedan takt da bi se izvršile. U slučaju punjenja (*load*) i čuvanja (*store*), kašnjenje se javlja zbog toga što je potrebno vreme da se pristupi memoriji, a u drugom slučaju (*branch*) dolazi do kašnjenje zbog pristupa instrukcijama koje su smeštene (nalaze se) na adresi grananja. Koncepti zakašnjenih punjenja i grananja dozvoljavaju da ako je instrukciji potreban više od jedan takt da bi se izvršila, procesor može da izvrši instrukciju koja sledi nakon instrukcije *Load* ili *Branch* sve dok se *Load* ili *Branch* ne završi. Ovo znači da kompilator ili programer treba da stremi (teži) da umetne instrukciju nakon instrukcije koja će zakasniti sa izvršenjem ali tako da izvršenje umetnute instrukcije ne utiče na rezultat instrukcije koja kasni sa izvršenjem. Obično se to izvodi umetanjem instrukcija tipa *Nop* na ta mesta, a efekat se zove slot zakašnjenja (*delayed slot*). Asistencija kompilatora kod ovakvih situacija je poželjna i korisna jer je programeru na asemblerskom jeziku teško da odredi da li instrukcija koja sledi nakon *Branch* biće izvršena ili ne, tj. da li će se grananje preduzeti ili ne.

Pojam princip lokalnosti možemo sagledati iz teorije keš memorija. Rad keš memorije u velikoj meri se zasniva na korišćenju sledećih triju principa lokalnosti:
1. *Vremenska lokalnost* (*temporal locality*) - informacija kojoj smo se od strane programa nedavno obratili sa velikom verovatnoćom će uskoro biti ponovo potrebna;
2. *Prostorna lokalnost* (*spatial locality*) - u skoroj budućnosti postoji velika verovatnoća da ćemo se ponovo obratiti delovima adresnog prostora bliskim tekućoj lokaciji obraćanja.

3. *Sekvencijalna lokalnost* (*sequential locality*) - specijalan slučaj prostorne lokalnosti kod koje će adresa narednog obraćanja biti neposredni naslednik tekućem obraćanju.
Pojam zavisnost (*dependance*) sagledaćemo poredjenjem protočnih i neprotočnih arhitektura. Kod neprotočnih mašina prvo se vrši pribavljanje zatim sledi izvršenje instrukcija, pri čemu se svaka instrukcija završi pre nego što počne naredna. To znači da bilo koja registarska ili memorijska vrednost koja se modifikuje od strane tekuće instrukcije biće dostupna instrukcijama koje slede. Ovo nije slučaj kod protočnih arhitektura, jer se u datom trenutku po nekoliko instrukcija može izvršavati od strane protočnog sistema Kod ovakve organizacije može da se desi sledeća situacija: Instrukcija koja modifikuje vrednost može da se desi da nije još ažurirala vrednost koja se čuva u registru ili memorijskoj lokaciji, a da instrukcija koja u programu sledi iza nje pokuša da pročita tu vrednost. Ovakav odnos izmedju dve instrukcije, kada vrednost generisana od strane jedne instrukcije nije još dostupna narednoj instrukciji je poznata kao *zavisnost* (*dependence*).
Protočni hazardi predstavljaju smetnje za potpuno iskorišćenje protočnog sistema. Problemi hazarda se mogu uspešno rešiti: a) zaustavljanjem rada protočnog sistema; b) premošćavanjem, tj. prosledjivanjem rezultata onim instrukcijama kojima su oni potrebni; c) ubacivanjem od strane kompilatora operacija tipa *Nop*; d) kombinacijom pomenutih tehnika.
Još neki značajni pojmovi koji se javljaju u toku rada su:
Slabo izražena vremenska lokalnost sa aspekta pristupa podacima znači da se pristup promenljivima vrši samo jedanput. Jako izražena vremenska lokalnost sa aspekta pristupa podacima znači da se pristup promenljivima vrši veoma često, tj. pristupi su tipa jedan za drugim. Slabo izračena prostorna lokalnost sa aspekta pristupa podacima znači da oni nisu grupisani (rasporedjeni) u strukture tipa polja nego su rasuti (razudjeni). Jako izražena prostorna lokalnost sa aspekta pristupa podacima znači da su oni grupisani u strukture tipa polja. Slabo izražena vremenska lokalnost kôda znači da ne postoje petlje, ili višestruko korišćenje istih programskih sekvenci. Jako izražena vremenska lokalnost kôda znači da postoje petlje, ili višestruko korišćenje istih programskih sekvenci. Slabo izražena prostorna lokalnost kôda znači da postoji veliki broj skokova koji su tipa "daleki" (*far*). Jako izražena prostorna lokalnost kôda znači da ne postoje instrukcije tipa *Branch/Jump*.

Predmet rada

1. Prevesti iskaze sa vi{eg programskog jezika na asembler mikroprocesora MIPS bez asistencije kompajlera (ru~no.
2. Izvr{iti optimizaciju prevedenih programskih sekvenci na asemblerski jezik kako bi minimizirao broj zastoja, a maksimizirala propusnost u izvr{enju instrukcija.

Postupak rada

KORAK 1: Za datu sekvencu na asemblerskom jeziku

```
Add   $4,$5,$6
Beq   $1,$2,40
Lw    $3,300($0)
```

i za model procesora koji se odnosi na proto~no izvr{enje proceniti da li }e do}i do zastoja u radu sistema i ukazati na na~ine kako se oni mogu re{iti (da li dolazi do zastoja i da li se mo`e preurediti redosled izvr{enja instrukcija).

KORAK 2: Neka je dat sledeći HLL iskaz

```
for (i=0; i<100; i++)
```

```
if (A[i]<50)
j = j+1;
else
k = k+1;
```

Napisati odgovarajući kôd na asemblerskom jeziku mikroprocesora MIPS i pokazati kako se mo`e optimizovati izvr{enje kada simulator radi u proto~nom re`imu rada.

KORAK 3: Za sledeći HLL iskaz

```
if k > 100
A:=k+1
else
A:=k-1
end if
```

Napisati odgovarajući kôd na asemblerskom jeziku mikroprocesora MIPS i pokazati kako se mo`e optimizovati izvr{enje kada simulator radi u proto~nom re`imu rada.

KORAK 4: Za sledeću petlju:

```
S:=0;
for k:=1 to 100 do
S:=S-1;
```

Napisati odgovarajući kôd na asemblerskom jeziku mikroprocesora MIPS i pokazati kako se mo`e optimizovati izvr{enje kada simulator radi u proto~nom re`imu rada.

KORAK 5: Procedura *swap* na programskom jeziku C se koristi da izvrši zamenu dve memorijske lokacije a sledećeg je oblika:

```
swap (int v[],int k)
[
int temp;
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
]
```

Odgovarajući kôd na asemblerskom jeziku mikroprocesora MIPS je oblika:

```
                        # u registru $t1 čuva se adresa od v[k]
Lw $t0,0($t1)           # registar $t0 (temp) = v[k]
Lw $t2,4($t1)           # registar $t2  = v[k+1]
Sw $t2,0($t1)           #  v[k] = registar $t2
Sw $t0,4($t1)           # v[k+1]  = registar $t0 (temp)
```

Locirati hazard u ovom programskom kôdu i preurediti instrukcije da bi se izbegao zastoj u protočnoj obradi.

KORAK 6: Na koji način treba modifikovati sledeću kôdnu sekvencu da bi se efikasno iskoristio slot zakašnjenja usled grananja (*delayed branch slot*)?

```
ponovi:  Lw    $2,100($3)
   Addi  $3,$3,4
```

```
        Beq   $3,$4,ponovi
```

Napisati odgovarajući kôd na asemblerskom jeziku mikroprocesora MIPS i pokazati kako se mo`e optimizovati izvr{enje kada simulator radi u proto~nom re`imu rada.

Pitanja

1. U čemu se RISC kompilator razlikuje od CISC kompilatora?
2. Ukazati na značenje sledećih pojmova:

a) paralelizam (*parallelism*)
b) protočnost (*pipelining*)
c) obrada sa preklapanjem (*overlapped processing*)

3. Identifikovati koncepte lokalnosti kod keš memorije.
4. Objasniti šta se podrazumeva pod pojmom *zavisnost*.
5. Kod protočne obrade postoje situacije kada se naredna instrukcija ne može izvršiti u sledećem taktnom intervalu. Ovakvi dogadjaji se nazivaju hazardi. Koji tipovi hazarda postoje?
6. Koje tehnike se koriste za rešavanje problema hazarda?
7. Sa aspekta pristupa podacima, za koji program kažemo da karakteriše:

	vremenska lokalnost	prostorna lokalnost
a)	slabo izražena	slabo izražena
b)	jako izražena	slabo izražena
c)	slabo izražena	jako izražena

8. Sa tačke gledišta pristupa instrukcijama, za koji program kažemo da karakteriše:

	vremenska lokalnost	prostorna lokalnost
a)	slabo izražena	slabo izražena
b)	jako izražena	slabo izražena
c)	slabo izražena	jako izražena

9. Kod skupa instrukcija mikroprocesora MIPS za instrukcije tipa *Branch* kažemo da su uvek zakašnjene (*delayed*). Objasniti značaj termina "zakašnjeno grananje" (*delayed branch*).
10. Objasniti pojmove zakašnjeno-punjenje i zakašnjeno-grananje (*delayed-load* i *delayed-branch*), a u kontekstu toga i slot zakašnjenja (*delay slot*).
11. U sledećoj kôdnoj sekvenci identifikovati sve zavisnosti po podacima.

```
        Add   $2,$5,$4
        Add   $4,$2,$5
        Sw    $5,100($2)
        Add   $3,$2,$4
```

Koje se zavisnosti se mogu eliminisati korišćenjem tehnike premošćavanja.

13. Posmatrajmo sledeći kôd

```
        for (i=0; i<20; i++)
      for (j=0; j<10; j++)
a[i] = a[i]*j
```

a) dati primer prostorne lokalnosti kod ovog kôda
b) dati primer vremenske lokalnosti kod ovog kôda

LABORATORIJSKA VEŽBA 25

Zavisnost po podacima, iterativna obrada, programske petlje, odmotavanje petlje

Uvod

Za ovu Laboratorijsku vežbu, osim iskustva u radu sa programskim simulatorom PCSpim iz prethodnih vežbanja, neophodno je proučiti osnovne koncepte programske implementacije iterativne obrade i programskih petlji, mašinski paralelizam (*Machine Level Paralelism*) i paralelizam na nivou instrukcija (*Instruction Level Paralelism - ILP*), protočne obrade kod RISC arhitektura, zavisnosti upravljačkog tipa, kao i koncept odmotavanja petlje izložen u kontekstu problematike superskalarnih procesora. Navedeno predznanje dostupno je u knjizi **RISC, CISC i DSP procesori** (pogl. 4 i 5) i **Zbirci zadataka za mikroprocesore i mikroračunare**. Nakon predvidjenog unosa, u fazi izvršavanja programa, aktivnosti studenta tiču se: identifikacije zavisnosti po podacima i procene vremena izvršavanja programa koji sadrže programske petlje. Saznanja koja se pri tome dobijaju rezultat su analize kôda programa u pogledu identifikacije zavisnosti po podacima, kao i optimalnosti dužine programa i vremena izvršenja.

Paralelizam instrukcija ili paralelizam na instrukcionom nivou se definiše kao mera količine instrukcija koje se mogu izvršavati istovremeno na jednoj superskalarnoj mašini neograničene širine (mašini sa neograničenim brojem hardverskih stepeni organizovanih u neograničenom broju funkcionalnih jedinica). Drugim rečima, to je mera paralelizma koji postoji u kôdu. Na primer, kôd:

```
for (i=0; i<=100; i=i+1) [a[i]=a[i]+1;]
```

sadrži značajan iznos paralelizma. Ako sagradimo mašinu sa 100 funkcionalnih jedinica i isto toliko memorijskih portova, takva koncepcija će nam pružiti ubrzanje od 100 puta. U većini slučajeva iznos ILP je obrnuto srazmeran količniku broja instrukcija koje dovode do zavisnosti i proceduralnih zavisnosti i broja ostalih instrukcija. Tako, možemo tvrditi da manji broj grananja i zavisnosti upravljačkog tipa povećavaju iznos ILP.

Mašinski paralelizam, sa druge strane, predstavlja iznos instrukcionog paralelizma koji, kao poboljšanje, nudi konkretna implementacija mašine.

Izmedju instrukcija koje se nalaze u nekoj od protočnih faza izvršenja mogu da postoje ograničenja tipa zavisnosti-po-podacima (*data dependencies*). Zavisnost-po-podacima izmedju instrukcija javljaju se zbog toga što instrukcije mogu da pristupaju (radi čitanja ili upisa) istoj lokaciji (memorijskoj ili registarskoj). Kada instrukcije pristupaju istoj lokaciji, kažemo da se javlja hazard jer postoji verovatnoća da redosled pristupa toj lokaciji ne bude korektan. Idealno posmatrano, izmedju instrukcija realno je očekivati da postoje samo ograničenja tipa prava zavisnost (*true dependence*). Ova ograničenja karakteristična su za hazarde *RAW* (*read-after-write*), jer instrukcija koja čita resurs (instrukcija tipa potrošač), može da pročita vrednost nakon što je instrukcija koja upisuje (instrukcija tipa proizvodjač) upisala rezultat. Pored pravih zavisnosti, posebno kod superskalarnih procesora, evidentne su i tzv. veštačke zavisnosti (*artificial* ažurira istu lokaciju, ali je pri ovome veoma bitno da se sačuva korektan *dependecies*).

Ove zavisnosti rezultat su *WAR* (*write after read*) i *WAW* (*write after write*) hazarda. *WAR* hazard se javlja u situacijama kada instrukcija treba da upiše novi rezultat u lokaciju, ali upis mora da se odloži sve dok prethodne instrukcije kojima je stara vrednost potrebna, ne pročitaju tu vrednost. *WAW* hazard se javlja kada veći broj instrukcija redosled ažuriranja.

Predmet rada

Upoznati se sa koncepcijom paralelne obrade na nivou instrukcije, koncepcijom iterativne obrade, problemima koji se tiču vremena izvršavanja i procene uslova grananja, kao jednim načinom za (delimično) otklanjanje uticaja zavisnosti po upravljanju - poznatog kao "odmotavanje petlje".

Postupak rada

KORAK 1: Uz pomoć tekst-editora kao u vežbi 20, uneti asemblerski program za MIPS kojim se izračunava vrednost iskaza na C-jeziku:

for (i=0; i<=1000; i=i+1) [a[i]=a[i]+c;]

Pretpostaviti da se adresa konstante *c* čuva u registru $s0, i da se početna adresa vektora a[i] nalazi u registru $a0. Da bi odredili bajt adresu sukcesivnih elemenata polja i proverili da li je kraj petlje, potrebne su nam konstante 4 i 4001. Usvojićemo da se one nalaze u memoriji na lokacijama *Konst4* i *Konst4001* u trenutku kada se vrši punjenje programa. Kod unosa programa predvideti deklaraciju segmenta programa (*.text*), kao i početnu labelu *main*. Jedna moguća asemblerska sekvenca koja predstavlja navedeni C-iskaz je:

```
            Add   $t0,$zero,$zero       # $t0 = 0
            Lw    $t1,0($s0)            # $t1 = c
            Lw    $t2,Konst4($zero)     # $t2 = 4
            Lw    $t3,Konst4001($zero)  # $t3 = 4001
Ponovi:     Add   $t4,$a0,$t0           # $t4 = adresa elementa a[i]
            Lw    $t5,0($t4)            # $t5 = a[i]
            Add   $t6,$t5,$t1           # $t6 = a[i]+ c
            Sw    $t6,0($t4)            # a[i]= a[i]+ c
            Add   $t0,$t0,$t2           # i = i + 4
            Slt   $t8,$t0,$t3           # $t8 = 1 if $t0 < 4001 tj. i < = 1000
            BNE   $t8,$zero,Ponovi      # go to Ponovi   if i < = 1000
```

Koliko se instrukcija obavi prilikom izvršenja ove sekvence? Koliko se obraćanja memoriji ukupno obavi? Može li se za navedenu sekvencu uvesti mašinski paralelizam i sa kojim stepenom? Kao posledica paralelizma na nivou instrukcije, kod protočnog izvršenja instrukcija, javljaju se zavisnosti po upravljanju i zavisnosti po podacima. Koja je instrukcija ključna za postojanje zavisnosti po upravljanju? Postoje li potencijalne zavisnosti po podacima izmedju instrukcija ove sekvence? Ako je odgovor potvrdan, identifikovati sve zavisnosti po podacima, a kod modifikovati tako da se one otklone.

KORAK 2: U Primeru iz koraka 1 realizovati kôd tako da se petlja razmota sa faktorom 2. U tom cilju, izmene u sekvenci iz Koraka 1 su sledeće:
Umesto konstante 4001 na lokaciji Konst4001, koristićemo konstantu 2001 na lokaciji Konst2001;
Nakon prve iteracije, u telu petlje navodimo i drugu iteraciju, i to: instrukcije `Add $t4,...`, pa do `Add $t0,...` umetnućemo izmedju ove poslednje i instrukcije `Slt`.

Da li razmotavanje petlje unosi dodatne zavisnosti po podacima, a za slu~aj da one postoje pokazati kako se mo`e modifikovati kod da bi se otklonile?

KORAK 3: Na osnovu prethodne dve sekvence, napisati treću koja izračunava iskaz:

for (i=0; i<=100; i=i+1) {a[i]=a[i]+a[i-1];}

Za čuvanje adrese elementa sa indeksom i-1 u svakoj iteraciji, uvodimo dodatni registar $t7. Umesto konstante c, registar $t1 sada čuva element a[i-1]. Uz pretpostavku da je a[-1]=0, jedno rešenje bilo bi:

```
            Add  $t0,$zero,$zero     # $t0 = 0
            Add  $t1,$zero,$zero     # $t1 = 0
            Lw   $t2,Konst4($zero)   # $t2 = 4
            Lw   $t3,Konst401($zero)# $t3 = 401
Ponovi:     Add  $t4,$a0,$t0         # $t4 = adresa elementa a[i]
            Lw   $t5,0($t4)          # $t5 = a[i]
            Add  $t6,$t5,$t1         # $t6 = a[i]+ a[i-1]
            Sw   $t6,0($t4)          # a[i]= a[i]+ a[i-1]
            Add  $t7,$t4,$zero       #$t7=adresa a[i]
            Lw   $t1,0($t7)          # $t1 = a[i]
            Add  $t0,$t0,$t2         # i = i + 4
            Slt  $t8,$t0,$t3         # $t8 = 1 if $t0 < 4001 tj. i < = 1000
            BNE  $t8,$zero,Ponovi    # go to Ponovi  if i < = 100
```

Da li sada postoje zavisnosti po podacima i koji su hazardi prisutni? Slično postupku iz prethodnog koraka, razmotati petlju faktorom 2. Uvodi li razmotavanje nove zavisnosti po podacima i koje? Za slu~aj da one postoje pokazati kako se mo`e modifikovati kod da bi se otklonile?

KORAK 4: Uneti asemblerski program za MIPS kojim se izračunava faktorijel broja 5 po iterativnom postupku. Program je dat u nastavku teksta. Na osnovu kôda i uz pomoć toka programa koji se može uočiti koračnim izvršavanjem, nacrtati dijagram toka programa i u njemu istaći početnu instrukciju petlje, telo petlje i tačku izlaska. Šta se obavlja u telu petlje (koji je rezultat svakog prolaska kroz petlju)? Koji su efekti instrukcije `jal fact` ? Ukazati na to šta čini drugu petlju u programu. Šta se obavlja i koji je efekat druge petlje? U kom registru je smešten konačni rezultat? Restartovati program i izvršavati ga koračno, brojeći pri tome instrukcije tipa `jal` i `j $31`. Koliko se instrukcija ukupno izvrši, ne računajući *startup* rutinu? Koristeći podatke koji se odnose na vreme izvršenja instrukcija, proceniti koliko će trajati izvršenje ovog programa.

KORAK 5: Modifikovati primer iz koraka 4 tako da se izračunava faktorijel broja 6. U primeru iz koraka 1 primeniti razmotavanje obe petlje sa faktorom 2. Izvršavati koračno tako dobijen program i odrediti koliko puta se izvršavaju instrukcije `jal` i `j $31`. Da li smanjenje broja ovih instrukcija utiče na ukupno vreme izvršenja programa i ako utiče, na koji način?

```
.data
str: .asciiz "faktorijel = "
.text
.globl main
main:
        subu $sp,$sp,32         #pomeranje vrha magacina za osam reci unapred
                                #(magacin raste smanjenjem adresa u sp i fp)
                                #time se rezervise prostor od osam reci
        sw $ra,20($sp)          #smestanje adrese povra  tka
                                #0($sp)  -trenutni vrh magacina, 4($sp)   -osma rec bloka,
                                #8($sp) -sedma,...,32($sp)  -prva
        sw $fp,16($sp)          #smestanje pokazivaca okvira na mesto pete reci bloka
        addu $fp,$sp,32         #pokazivac okvira (granice bloka)
        li $a0,5                #reg. a0 puni se vredno  scu 5
        la $t0, fact            #priprema registra t0 punjenjem adrese fact
        jalr $t0                #poziv procedure fact (adresa naredne instrukcije,
                                #a to je lw $ra,20($sp), pamti se u ra tj. u r31 )
        lw $ra,20($sp)          #nova vrednost ra postaje ona sacuvana u trecoj reci
                                #aktuelnog magacinskog bloka
        lw $fp,16($sp)          #vraca se pocetna vrednost pokazivacu okvira
        addu $sp,$sp,32         #vraca se pocetna vrednost pokazivacu magacina
        j $ra                   #povratak na adresu iz ra (u start   -up program)
fact:                           #procedura fact
        subu $sp,$sp ,32        #za proceduru fact rezervisemo novi blok
                                #od osam reci u magacinu
        sw $ra,20($sp)          #adresu povratka u pozivnu proceduru (na naredbu u main)
                                #smestamo na mesto cetvrte reci ovog bloka
        sw $fp,16($sp)          #pokazivac okvira smestamo na mesto pete r    eci
        addu $fp,$sp,32         #novi pokazivac okvira je osam reci posle vrha magacina
        li $2,1                 #inicijalna vrednost faktorijela je 1 i cuva se u v0
$L2:                            #ovo je petlja koja racuna faktorijel
        beq $a0,$0,$L1          #a0 je brojac i proverava se da li je stigao    do 0,
                                #ako jeste, izlaz iz petlja je labela $L1
        mul $2,$2,$a0           #ako nije, mnozi se a0 i trenutna vrednost v0
        subu $a0,$a0,1          #dekrementira se a0 (smanji se za 1)
```

Pitanja

1. Objasniti pojmove mašinski paralelizam (*Machine Level Paralelism*) i paralelizam na nivou instrukcija (*Instruction Level Paralelism*).
2. Kog su tipa hazardi RAW, WAR i WAW?
3. Šta se nastoji da eliminiše tehnikom razmotavanja petlje?
4. Koje su prednosti razmotavanja petlje, a koji su nedostaci ovog pristupa?
5. Ima li bitne razlike u broju izvršenih instrukcija u varijantama osnovnog programa programa sa razmotavanjem petlje?

LABORATORIJSKA VEŽBA 26

Rekurzivne procedure, funkcijski pozivi i interfejs sa korisnikom

Uvod

Kao i za prethodnu, i kod ove laboratorijske vežbe, osim iskustva u radu sa programskim simulatorom PCSpim iz prethodnih vežbanja, potrebno je proučiti osnovne koncepte organizacije i načina rada sa memorijskim lokacijama magacina i implementacije rekurzivne obrade kod MIPS procesora. Potrebno predznanje dostupno je u knjizi **RISC, CISC i DSP procesori** (pogl. 4 i 5), **Zbirci zadataka za mikroprocesore i mikroračunare** i **SPIM S20.** Nakon predvidjenog unosa, u fazi izvršavanja programa, aktivnosti studenta tiču se optimizacije kôda programa i razvoja interfejsa sa korisnikom uz korišćenje sistemskih poziva. Svi sistemski pozivi predstavljaju odredjene rutine u sistemskom (kernel) delu simulatora koji se učitava sa svakim korisničkim programom. Uz pomoć ovih procedura se pre svega obezbedjuje podrška interfejsu korisničkog programa i korisnika. Naglasimo da je potreba za ovakvom sistemskom podrškom rezultat činjenice da na PC mašini zasnovanoj na 80x86 mikroprocesoru i odgovarajućem sistemskom softveru ne postoji drugi način za implementaciju pomenutih funkcija. Sa druge strane, kod mikroprocesora MIPS familije, interfejs- i druge funkcije implementiraju se u samom opertivnom sistemu, odnosno kernelu, a mogu se pozivati posebno predvidjenom asemblerskom naredbom *syscall*.

U Tabeli 1 (ve~ba 20). prikazani su svi sistemski pozivi koji mogu da se pozivaju od strane korisničkog programa. Da bi se pozvao servis, u korisničkom programu treba učitati kôd sistemskog poziva u registar $v0 i argumente poziva u registre $a0, ..., $a3 (odnosno $f12 za vrednosti tipa *floating point*). Sistemska procedura vraća rezultat u registru $v0 (odnosno $f0 za rezultate tipa *floating point*).

Predmet rada

Upoznati se sa koncepcijom rekurzivne obrade, problemima koji se tiču rada sa magacinom i korišćenjem sistemskih poziva.

Postupak rada

KORAK 1: Uz pomoć tekst-editora kao u vežbi 20, uneti asemblerski program *fact.s* za MIPS kojim se izračunava vrednost faktorijela broja 5 po rekurzivnom postupku. Program je dat u nastavku teksta. Uz pomoć simulatora PCSpim, prateći efekte koračnog izvršavanja, nacrtati dijagram toka programa i u njemu istaći tačku rekurzivnog poziva, kao i proveru kriterijuma kraja rekurzije. U kom registru je smešten konačni rezultat?

```
.text
.globl main
main:
   subu $sp,$sp,32
   sw   $ra,20,($sp)
   sw   $fp,16,($sp)
   addu $fp,$sp,32
```

```
   li   $a0,5
   la   $t0, fact
   jalr $t0
   lw   $ra,20($sp)
   lw   $fp,16($sp)
   addu $sp,$sp,32
   j    $ra
fact:
   subu $sp,$sp,32
   sw   $ra,20($sp)
   sw   $fp,16($sp)
   addu $fp,$sp,32
   sw   $a0,0($fp)
   lw   $2,0($fp)
   bgtz $2,$L2
   li   $2,1
   j    $L1
$L2:
   lw   $3,0($fp)
   subu $2,$3,1
   move $a0,$2
   jal  fact
   lw   $3,0($fp)
   mul  $2,$2,$3
   addu $4,$0,$2
$L1:
   lw   $31,20($sp)
   lw   $fp,16($sp)
   addu $sp,$sp,32
   j    $31
```

KORAK 2:. U tekst-editoru, zatim, modifikovati program tako da se, nakon izvršenja, na konzoli prikazuje sledeća poruka tipa *string*: "faktorijel je", a zatim i broj tj.vrednost izračunatog faktorijela. Koristiti pri tome odgovarajuće sistemske pozive. Novi program neka se zove *fact_new.s*.

KORAK 3: Uz pomoć PCSpim-a i postavljanjem prekidne tačke na prvoj instrukciji rutine *fact_new*, verifikovati ispravnost izvršavanjem modifikovanog programa. U cilju efikasnije verifikacije, posmatrati memoriju magacina nakon svake prekidne tačke. Koliko reči zauzima svaki novi magacin?

Pitanja

1. Koje su osnovne karakteristike rekurzivne obrade?
2. Šta mora da sadrži svaka rekurzivna procedura?
3. Nacrtati principijelnu memorijsku mapu magacinskog okvira. (Koristiti preporuke iz konvencije pozivanja procedura - literatura)
4. U cemu je slicnost a u cemu razlika izmedu funkcijskih poziva DOS-a i BIOS-a kod mašina 80x86, i si1stemskih poziva kod MIPS-a?

LABORATORIJSKA VEŽBA 27

Protočni mikrokontroler

Uvod

Prolaskom kroz ve}i broj koraka analizira}emo postupak projektovanja jednog jednostavnog tro-stepenog proto~no organizovanog mikrokontrolera. Kroz diskusiju ukaza}emo na slede}e detalje:

a) definiciju skupa instrukcija mikrokontrolera,

b) definiciju arhitekture dizajna koriste}i dijagrame toka i tablice istinitosti,

c) definicije mikroarhitekturnog i interfejs modula.

Nakon uvodnih napomena, koje su u vezi sa ta~kama a), b) i c), u daljem tekstu bi}e opisani funkcionalni kôdovi u VHDL-u za slede}e gradivne blokove: ***predecode***, ***decode***, ***register file*** i ***execute***, i kona~no dat strukturni VHDL kôd za kompletan mikroprocesor.

Izlo`ena materija koja se odnosi na sintezu proto~no organizovanog mikrokontrolera bi}e prezentirana kroz organizaciju laboratorijskih ve`bi. U svakoj ve`bi bi}e dati detalji koji se odnose na funkciju odgovaraju}ih gradivnih blokova. Sastavni deo ve`bi bi}e kôdovi tipa ***testbench*** programa kojima se verifikuje ispravnost rada blokova.

Zadatak studenta je da na osnovu tajming dijagrama dobijenih postupkom simulacije verifikuje ispravnost modula. Ukupna materija koja se odnosi na sintezu mikrokontrolera bi}e prezentirana kroz tri ve`be. Prva ve`ba pokriva detalje koji se odnose na ***predecode*** i ***decode*** blokove, druga na ***register file*** i ***execute*** blokove, a tre}a pokriva kompletan mikrokontroler.

Definicija skupa instrukcija

Odluka koju na po~etku treba doneti odnosi se na definiciju skupa instrukcija koje mikrokontroler treba da izvr{va. Ograni~i}emo se na skup od osam instrukcija. Opis skupa instrukcija prikazan je na slici 27-1.

skup instrukcija	opis
MOVE <src1>,<dst>	kopiranje sadr‘aja <src1> u <dst>
ADD <src1>,<src2>,<dst>	sabiranje sadr‘aja <src1> sa sadr‘ajem <src2> i sme{tanje rezultata u <dst>
SUB <src1>,<src2>,<dst>	oduzimanje sadr‘aja <src1> od sadr‘aja <src2> i sme{tanje rezultata u <dst>
MUL <src1>,<src2>,<dst>	mno‘enje sadr‘aja <src1> sa sadr‘ajem <src2> i sme{tanje rezultata u <dst>

CJE <src1>,<src2>,<CODE>	upore|ivanje sadr‘aja <src1> sa sadr‘ajem <src2> i ako su jednaki grananje na instrukciju ispred koje stoji oznaka <CODE>
LOAD <vred>,<dst>	punjenje sadr‘aja <vred> u <dst>
READ <dst>	~itanje sadr‘aja <dst>i predaja podataka ka izlaznim pinovima
NOP	operacija bez efekta

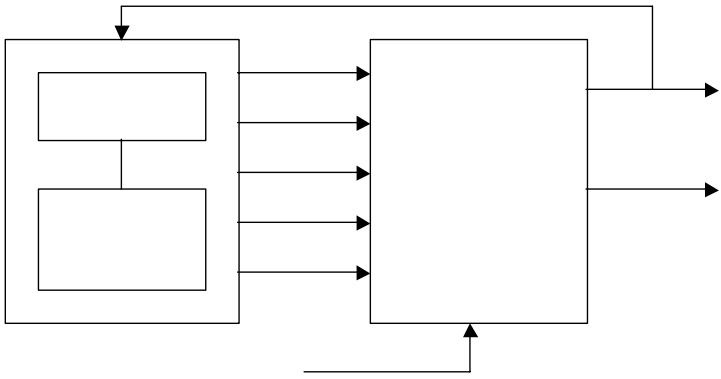
Slika 27-1. Definicija skupa naredbi

Napomena: <src1> - izvori{te 1; <src2> - izvori{te 2; <dst> - odredi{te; <vred> - vrednost

Za kodiranje opkôda instrukcije potrebna su tri ulazna pina (2^3= 8). Pro{irenje skupa instrukcija zahtevalo bi uvodjenje novih pinova.
U konkretnom slu~aju <src1>, <src2> i <dst> mogu da predstavljaju bilo koji od internih 32-bitnih registara reg0, reg1, ... , reg15.

Definicija arhitekture

Na osnovu skupa instrukcija definisa}emo interfejs mikrokontrolera. U cilju boljeg razumevanja problematike pojednostavi}emo strukturu mikrokontrolera, ali to }emo uraditi sa namerom da se jasnije sagleda kôd koji se mo‘e sintetizovati, a ne da se izu~ava teorija rada proto~nih sistema.
Kod arhitekture mikrokontrolera ne postoji *“register scoreboarding”* a takodje nije izveden pristup spoljnoj-memoriji. Za slu~aj da se zahteva pristup spoljnoj memoriji, arhitekturu je mogu}e pro{iriti uvodjenjem novih signala prema spoljnjem memorijskom modulu, a takodje i uvodjenjem novih instrukcija radi pristupa memoriji.
Za slu~aj da se zahteva implementacija *“scoreboarding”*-a tada svih {esnaest 32-bitnih registara treba pro{iriti na 33 bita, tj. po jedan dodatni bit u svakom od registara bi se koristio za manipulisanje sa *“scoreboarding”*-om.
Dodatna pretpostavka je ta da se instrukcije i podaci pribavljaju od strane mikrokontrolera preko eksternog ke{a za instrukcije i spoljne memorije.
Na slici 27-2 prikazan je na~in sprezanja mikrokontrolera. Mikrokontroler komunicira sa spoljnim instrukcionim-modulom ~iji je glavni zaadtak da puni (loaduje) instrukcije (sa podacima) u mikrokontroler. Izlaz *jump* je ulaz u instrukcioni-modul i signalizira instrukcionom-modulu da je potrebno izvr{iti grananje na deo kôda kada do grananja dodje (obavi se grananje).



Slika 27-2 Interfejs mikrokontrolera

Na slici 27-3 prikazan je opis svih ulaznih i izlaznih signala interfejsa mikrokontrolera.

pinovi	ulaz/izlaz	broj bitova	opis
inst	input	3	Instrukcija koja mikrokontroler treba da izvr{i 000 – MOVE 001 – ADD 010 – SUB 011 – MUL 100 – CJE 101 – LOAD 110 – READ 111 – NOP
source1	input	4	<source1>
source2	input	4	<source2>
destination	input	4	<destination>
data	input	32	ulazni podaci za LOAD instrukciju
clock	input	1	Mikrokontroler koristi ovaj clock ulaz da bi izvr{io proto~nu obradu
jump	output	1	Postavlja se na visoki nivo kada se CJE instrukcijom utvrdi da <source1> i <source2> imaju iste vrednosti
output	output	32	izlaz podatka za READ instrukciju

instrukcioni ke{

memorija za instrukcije

instrukcioni modul

Slika 27-3 Opis interfejs signala mikrokontrolera

Prezentacija svih 16 registra binarnim vrednostima za *src1*, *src2* i *dst* data je na slici 27-4.

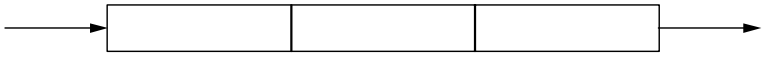
source1/source2/destination	registar	ime registra

0000	register 0	reg0
0001	register 1	reg1
0010	register 2	reg2
0011	register 3	reg3
0100	register 4	reg4
0101	register 5	reg5
0110	register 6	reg6
0111	register 7	reg7
1000	register 8	reg8
1001	register 9	reg9
1010	register 10	reg10
1011	register 11	reg11
1100	register 12	reg12
1101	register 13	reg13
1110	register 14	reg14
1111	register 15	reg15

Slika 27-4 Prezentacija 16 internih registara mikrokontrolera binarnim vrednostima

Definicija protočnog sistema

Nakon definicije arhitekturnih i interfejs detalja definisa}emo mikrokontroler kao 3-stepeni proto~ni sistem kod koga postoje slede}i stepeni: ***predecode***, ***decode*** i ***execute*** (vidi sliku 27-5).

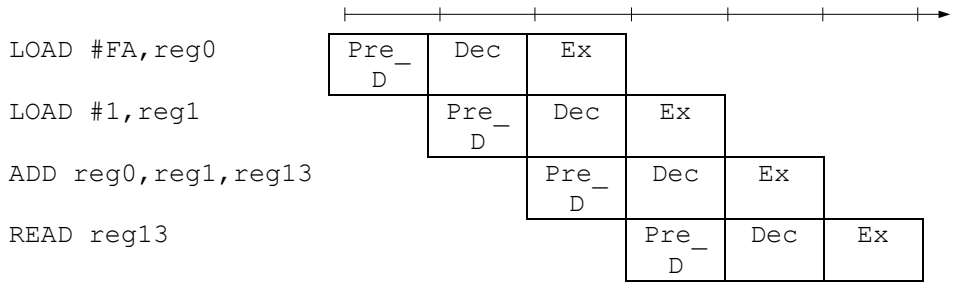


Slika 27-5 Proto~ni stepeni mikrokontrolera

Stepen ***predecode*** ostvaruje spregu (spre`e se) sa spoljnim instrukcionim-modulom. Na svojim ulazima ovaj stepen prihvata instrukcije i podatke od instrukcionog-modula.
Stepen ***decode*** dekodira ulazne pobudne signale koji se generi{u od strane instrukcionog-modula i prolaze (prenose se) kroz stepen ***predecode***. Ovaj stepen takodje dekodira i ulazne stimule. Izlazi ovog stepena se vode ka stepenu ***execute*** koji izvr{ava instrukcije.
Mikrokontroler je proto~no organizovani sistem, {to zna~i da se svakog taktnog intervala, apstrahuju}i hazarde, inicira i izvr{ava po jedna instrukcija. Dizajn mikrokontrolera je naravno ne{to slo`eniji za slu~aj kada se implementira tehnika premo{}avanja (***bypassing***). Za po~etak usvoji}emo da se od strane instrukcionog-modula predaje mikrokontroleru slede}i skup od ~etiri instrukcije:

```
LOAD  #FA, reg0
LOAD  #1, reg1
ADD   reg0, reg1, reg13
READ  reg13
```

Za izvr{enje ove ~etiri instrukcije potrebno je 6 taktnih intervala (vidi Sliku 27-6).



Slika 27-6 Proto~no izvr{enje instrukcija
Napomena: Pre_D - *predecode;* Dec - *decode* i Ex_D *execute*

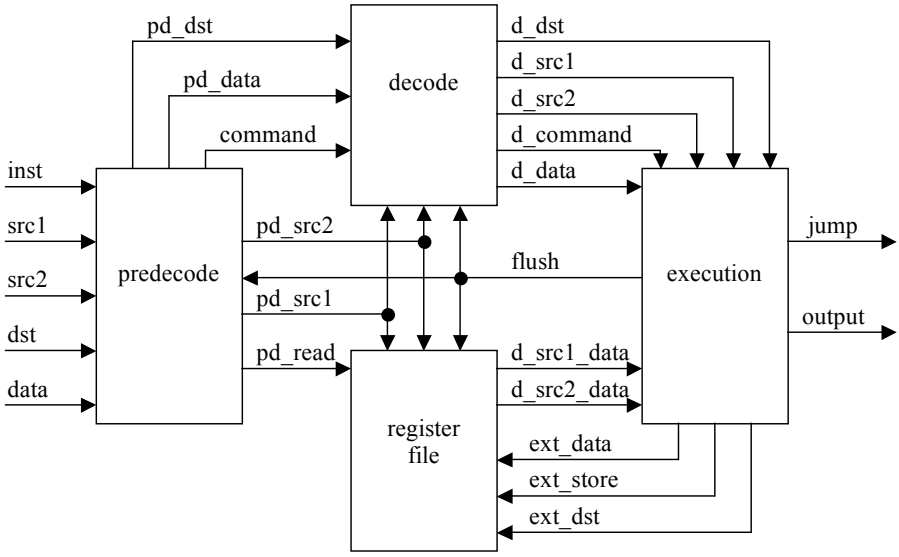
Definicija mikroarhitekture protočnog mikrokontrolera

Da bi se projektovala sva tri stepena proto~nog sistema neophodno je prvo sagledati mikro-arhitekturnu implementaciju dizajna. Pri ovome se mora strogo voditi ra~una o funkcionalnoj podeli interfejsa-za-signalizaciju kojim se ostvaruje sprega izmedju funkcionalnih blokova. Takodje treba imati u vidu da se razli~ite funkcije dizajna mogu grupisati u razli~ite particije pri ~emu svaka particija treba da obavlja odredjenu funkciju. Kada se za datu arhitekturu ostvari dobra funkcionalna particija tada ka`emo da se posti`u dobre (optimalne) performanse kao i efikasno iskori{}enje povr{ine silicijuma.

U konkretnom slu~aju podeli}emo funkciju mikrokontrolera na ~etiri razli~ita bloka: ***predecode*** - blok, ***decode*** - blok, ***register file*** - blok i ***execute*** - blok. Svaki od ovih blokova karakteri{e se sopstvenom funkcionalno{}u.

Na Slici 6 prikazan je na~in povezivanja blokova u okviru mikrokontrolera. Treba pri ovome ista}i da razvodjenje taktnog signala nije prikazano na slici 27-6 ali se implicitno podrazumeva da se taktni signal dovodi do svakog bloka.

Na Slici 27-6 treba uo~iti signal ***flush*** koji se generi{e od strane bloka ***execute*** a dovodi na ulaz blokova ***predecode***, ***decode*** i ***register file***. Signal ***flush*** se aktivira kod instrukcije grananja. Kada dodje do grananja sve teku}e instrukcije u svakom bloku moraju se poni{titi. Ovo }e obezbediti da se od strane instrukcionog-modula predaju nove instrukcije mikrokontroleru. Signal ***jump*** koji je izlaz bloka ***execute*** ali takodje i izlaz mikrokontrolera kojim se pobudjuje ulaz instrukcionog-modula. Ovaj signal se koristi od strane instrukcionog modula kao indikator da je do grananja do{lo, pri ~emu instrukcioni modul, nakon prihvatanja ovog signala, predaje mikrokontroleru novu instrukciju (ona koja se odnosi na adresu grananja).



Slika 27-7 Mikroarhitekturna definicija mikrokontrolera

Usvoji}emo takodje da instrukcioni modul je taj koji poseduje interna kola u kojima se ~uva trag o odredi{noj adresi grananja. Pre nego {to kreiramo kôd za svaki funkcionalni blok potrrebno je napisati VHDL ***package*** - fajl kojem }e se obra}ati (pozivati se) od strane svih funkcionalnih blokova.

Postupak rada

1. Pokrenuti program ***Active-HDL 3.6***.
2. U prozoru ***Getting Started*** selektovati opciju ***Create new design*** i kliknuti na OK.
3. U novootvorenom prozoru za naziv dizajna upisati ***ime_broj_indeksa*** (npr. goran_1234) i kliknuti na ***Next***.
4. Selektovati ***Create an empty design*** i kliknuti na ***Next***, a zatim na ***Finish***.
5. U prozoru ***Design Browser*** dva puta kliknuti na ***Add New File***. Za tip fajla izabrati VHDL ***Source Code*** i imenovati ga sa ***My_package***. Selektovati opciju ***Add To Design*** i kliknuti na OK.
6. U prozoru editora napisati kôd za VHDL ***package_file*** pridr`avaju}i se slede}eg:

➢ definisati ime paketa kao ***pipeline_package***
➢ definisati slede}e tipove:

- ***command_type*** sadr`i skup svih definisanih instrukcija (MOVE, ... ,NOP)
- ***register_type*** sadr`i skup svih registara (reg0, ... , reg15)
- ***array_size*** polje od 16 vektora tipa ***std_logic_vector*** veli~ine 32 bita

- definisati konstantu ZERO kao 32-bitni nulti vektor tipa *std_logic_vector*

7. Snimiti kreirani fajl komandom ***File -> Save***
8. U ***Design Browser*** prozoru kliknuti desnim tasterom mi{a na snimljeni fajl i izabrati opciju ***Compile***.

Napomena: Primer kako treba uraditi korak 6

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

PACKAGE pipeline_package IS
TYPE command_type IS (MOVE,ADD,SUB,MUL,CJE,LOAD,READ,NOP);
TYPE register_type IS
(reg0,reg1,reg2,reg3,reg4,reg5,reg6,reg7,reg8,reg9,reg10,
reg11,reg12,reg13,reg14,reg15);
TYPE array_size IS array (0 to 15) of std_logic_vector (31
 downto 0);
CONSTANT ZERO : std_logic_vector (31 downto 0) :=
(others=>'0');
END pipeline_packag;
```

9. Ukoliko je kompilacija izvr{ena korektno ispred fajla ***My_package.vhd*** u ***Design Browser*** prozoru pojavi}e se oznaka √. Ukoliko je do{lo do gre{ke u kompilaciji ispred imena fajla pojavi}e se oznaka X. Nakon ispravljanja gre{aka ponovo snimiti i kompajlirati fajl.

LABORATORIJSKA VEŽBA 28

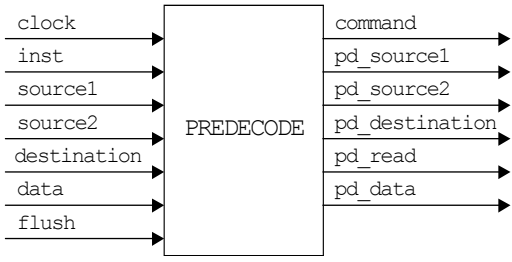
Predecode i *Decode* blokovi

Uvod

U ovoj ve`bi analizira}emo rad ***Predecode*** i ***Decode*** blokova. Ve`ba je organizovana na slede}i na~in:

Predecode blok

1. Predstavljeni su interfejs signali ***Predecode*** bloka (slika 28-1).
2. Opisani su interfejs signali ***Predecode*** bloka (slika 28-2).
3. Analizirati definicije signala datih na slikama 28-1 i 28-2.
4. Analizirati VHDL kôd, prikazan na slici 28-3, kojim se opisuje rad ***Predecode*** bloka.
5. Analizirati ***testbench*** program, prikazan na slici 28-4, kojim se verifikuje rad ***Predecode*** bloka.



Slika 28-1. Interfejs signali kod ***predecode*** bloka

ime signala	I/O	opis
clock	ulaz	ulaz taktnog impulsa
inst	ulaz	3-bitna instrukciona linija "000" = MOVE "001" = ADD "010" = SUB "011" = MUL "100" = CJE (compare and jump if equal) "101" = LOAD

		"110" = READ "111" = NOP (no operation)
source 1	ulaz	4-bitna linija preko koje se specificiraju registri koji }e se koristiti kao <source1> "0000" = reg0 "1000" = reg8 "0001" = reg1 "1001" = reg9 "0010" = reg2 "1010" = reg10 "0011" = reg3 "1011" = reg11 "0100" = reg4 "1100" = reg12 "0101" = reg5 "1101" = reg13 "0110" = reg6 "1110" = reg14 "0111" = reg7 "1111" = reg15
source 2	ulaz	4-bitna linija preko koje se specificiraju registri koji }e se koristiti kao <source2> "0000" = reg0 "1000" = reg8 "0001" = reg1 "1001" = reg9 "0010" = reg2 "1010" = reg10 "0011" = reg3 "1011" = reg11 "0100" = reg4 "1100" = reg12 "0101" = reg5 "1101" = reg13 "0110" = reg6 "1110" = reg14 "0111" = reg7 "1111" = reg15
destination	ulaz	4-bitna linija preko koje se specificiraju registri koji }e se koristiti kao <destination> "0000" = reg0 "1000" = reg8 "0001" = reg1 "1001" = reg9 "0010" = reg2 "1010" = reg10 "0011" = reg3

		"1011" = reg11 "0100" = reg4 "1100" = reg12 "0101" = reg5 "1101" = reg13 "0110" = reg6 "1110" = reg14 "0111" = reg7 "1111" = reg15
data	ulaz	32-bitna linija za ulaz podataka u toku LOAD instrukcije
flush	ulaz	postavljanje visokog nivoa u slu~aju grananja
command	izlaz	Instrukcija koja se prosledjuje ***decode*** bloku. Ona }e imati vrednost tipa ***command_type*** gde je ***command_type*** jedan od sled}ih tipova: MOVE, ADD, SUB, MUL, CJE, LOAD, READ i NOP.
pd_source1	izlaz	Odredjuje <source1> koji je tipa ***register_type*** i mo`e imati vrednosti reg0,reg1,reg2,reg3,reg4, reg5,reg6,reg7,reg8,reg9,reg10,reg11, reg12,reg13,reg14,reg15
pd_source2	izlaz	Odredjuje <source2> koji je tipa ***register_type*** i mo`e imati vrednosti reg0, reg1, reg2, reg3, reg4, reg5, reg6, reg7, reg8, reg9, reg10, reg11, reg12, reg13, reg14, reg15
pd_destination	izlaz	Odredjuje <destination> koji je tipa ***register_type*** i mo`e imati vrednosti reg0, reg1, reg2, reg3, reg4, reg5, reg6, reg7, reg8, reg9, reg10, reg11, reg12, reg13, reg14, reg15
pd_data	izlaz	32-bitna linija kojom se prosledjuju podaci do ***decode*** bloka.
pd_read	izlaz	Bit koji se postavlja na visoki nivo kako bi ukazao ***register file*** bloku da izvr{i ~itanje iz svojih internih registara <pd_source1> i <pd_source2>. Pro~itani podaci se postavljaju na izlaz kao <source1_data> i <source2_data>.

Slika 28-2 Opis interfejs signala ***predecode*** bloka

Postupak rada

1. U programu Active-HDL 3.6 otvoriti direktorijum ***ime_broj indeksa***.

2. Dva puta kliknuti na ***Add New Files*** a zatim na ***Add Existing File***. U direktorijumu C:\VEZBE\PMIC\ selektovati fajl ***predecode_ent.vhd*** i kliknuti na ***Add***. Nakon toga fajl }e biti prikazan u ***Design Browser*** prozoru pod istim imenom.
3. Dvostrukim klikom mi{a na fajl pogledati njegov izvorni kôd i proveriti da li se sla`e sa kôdom na Slici 3.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE WORK.pipeline_package.ALL;

ENTITY predecode_ent IS
PORT  (
   clock : IN std_logic;
inst : IN std_logic_vector (2 downto 0);
source1 : IN std_logic_vector (3 downto 0);
   source2 : IN std_logic_vector (3 downto 0);
   destination : IN std_logic_vector (3 downto 0);
   data : IN std_logic_vector (31 downto 0);
   flush : IN std_logic;
   command : OUT command_type;
   pd_source1 : OUT register_type;
   pd_source2 : OUT register_type;
   pd_destination : OUT register_type;
   pd_data : OUT std_logic_vector (31 downto 0);
   pd_read : OUT std_logic
   );
END predecode_ent;

ARCHITECTURE predecode_arch OF predecode_ent IS
BEGIN
PROCESS (clock,inst,source1,source2,destination,data, flush)
VARIABLE internal_command : command_type := NOP;
VARIABLE internal_source1 : register_type := reg0;
VARIABLE internal_source2 : register_type := reg0;
VARIABLE internal_destination : register_type := reg0;
   BEGIN
      IF (clock = '1' AND clock'EVENT) THEN
         IF (flush = '0') THEN
            CASE inst IS
               WHEN "000" =>
                  pd_read <= '1';
                  internal_command := MOVE;
               WHEN "001" =>
                  pd_read <= '1';
                  internal_command := ADD;
               WHEN "010" =>
                  pd_read <= '1';
                  internal_command := SUB;
               WHEN "011" =>
                  pd_read <= '1';
                  internal_command := MUL;
```

```
      WHEN "100" =>
         pd_read <= '1';
         internal_command := CJE;
      WHEN "101" =>
         pd_read <= '0';
         internal_command := LOAD;
      WHEN "110" =>
         pd_read <= '1';
         internal_command := READ;
      WHEN "111" =>
         pd_read <= '0';
         internal_command := NOP;
      WHEN OTHERS =>
         NULL;
   END CASE;
   CASE source1 IS
      WHEN "0000" =>
         internal_source1 := reg0;
      WHEN "0001" =>
         internal_source1 := reg1;
      WHEN "0010" =>
         internal_source1 := reg2;
      WHEN "0011" =>
         internal_source1 := reg3;
      WHEN "0100" =>
         internal_source1 := reg4;
      WHEN "0101" =>
         internal_source1 := reg5;
      WHEN "0110" =>
         internal_source1 := reg6;
      WHEN "0111" =>
         internal_source1 := reg7;
      WHEN "1000" =>
         internal_source1 := reg8;
      WHEN "1001" =>
         internal_source1 := reg9;
      WHEN "1010" =>
         internal_source1 := reg10;
      WHEN "1011" =>
         internal_source1 := reg11;
      WHEN "1100" =>
         internal_source1 := reg12;
      WHEN "1101" =>
         internal_source1 := reg13;
      WHEN "1110" =>
         internal_source1 := reg14;
      WHEN "1111" =>
         internal_source1 := reg15;
      WHEN OTHERS =>
         NULL;
```

```
END CASE;
CASE source2 IS
   WHEN "0000" =>
      internal_source2 := reg0;
   WHEN "0001" =>
      internal_source2 := reg1;
   WHEN "0010" =>
      internal_source2 := reg2;
   WHEN "0011" =>
      internal_source2 := reg3;
   WHEN "0100" =>
      internal_source2 := reg4;
   WHEN "0101" =>
      internal_source2 := reg5;
   WHEN "0110" =>
      internal_source2 := reg6;
   WHEN "0111" =>
      internal_source2 := reg7;
   WHEN "1000" =>
      internal_source2 := reg8;
   WHEN "1001" =>
      internal_source2 := reg9;
   WHEN "1010" =>
      internal_source2 := reg10;
   WHEN "1011" =>
      internal_source2 := reg11;
   WHEN "1100" =>
      internal_source2 := reg12;
   WHEN "1101" =>
      internal_source2 := reg13;
   WHEN "1110" =>
      internal_source2 := reg14;
   WHEN "1111" =>
      internal_source2 := reg15;
   WHEN OTHERS =>
      NULL;
END CASE;
CASE destination IS
   WHEN "0000" =>
      internal_destination := reg0;
   WHEN "0001" =>
      internal_destination := reg1;
   WHEN "0010" =>
      internal_destination := reg2;
   WHEN "0011" =>
      internal_destination := reg3;
   WHEN "0100" =>
      internal_destination := reg4;
   WHEN "0101" =>
      internal_destination := reg5;
```

```
               WHEN "0110" =>
                  internal_destination := reg6;
               WHEN "0111" =>
                  internal_destination := reg7;
               WHEN "1000" =>
                  internal_destination := reg8;
               WHEN "1001" =>
                  internal_destination := reg9;
               WHEN "1010" =>
                  internal_destination := reg10;
               WHEN "1011" =>
                  internal_destination := reg11;
               WHEN "1100" =>
                  internal_destination := reg12;
               WHEN "1101" =>
                  internal_destination := reg13;
               WHEN "1110" =>
                  internal_destination := reg14;
               WHEN "1111" =>
                  internal_destination := reg15;
               WHEN OTHERS =>
                  NULL;
            END CASE;
            IF (internal_command = LOAD) THEN
               pd_data <= data;
            ELSE
               pd_data <=
               (others => '0');
            END IF;
         ELSE
            pd_data <= (others => '0');
            internal_command := NOP;
            pd_read <= '0';
         END IF;
         command <= internal_command;
         pd_source1 <= internal_source1;
         pd_source2 <= internal_source2;
         pd_destination <= internal_destination;
      END IF;
   END PROCESS;
END predecode_arch;
```

Slika 28-3 VHDL kôd za blok ***predecode***

4. Desnim tasterom mi{a kliknuti na fajl ***predecode_ent.vhd*** u ***Design Browser*** prozoru, a zatim ga kompajlirati pritiskom na ***Compile,*** u padaju}em meniju. Ukoliko je kompilacija izvr{ena korektno ispred imena fajla ***predecode_ent.vhd*** u ***Design Browser*** prozoru pojavi}e se oznaka √. Ukoliko je do{lo do gre{ke u kompilaciji ispred imena fajla pojavi}e se oznaka X.

5. Dva puta kliknuti na ***Add New Files***, a zatim na ***Add Existing File***. U direktorijumu C:\VEZBE\PMIC\ selektovati fajl ***predecode_testbench.vhd*** a zatm kliknuti na ***Add***. Nakon toga }e se selektovani fajl pojaviti u ***Design Browser*** prozoru.
6. Dvostrukim klikom na fajl ***predecode_testbench.vhd*** pogledati njegov izvorni kôd u prozoru editora. Na Slici 28-4 dat je ***testbench*** program koji simulira rad ***predecode*** bloka pomo}u slede}e ~etiri instrukcije

```
Load  #4592fa83,reg0
      Load  #00000001,reg15
      Move  reg0,reg8
Add   reg0,reg15,reg1
Nop
```

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE WORK.pipeline_package.ALL;

ENTITY predecode_tb_ent IS
END predecode_tb_ent;

ARCHITECTURE predecode_tb_arch OF predecode_tb_ent IS
COMPONENT predecode_ent
PORT  (
   clock : IN std_logic;
inst : IN std_logic_vector (2 downto 0);
source1 : IN std_logic_vector (3 downto 0);
   source2 : IN std_logic_vector (3 downto 0);
   destination : IN std_logic_vector (3 downto 0);
   data : IN std_logic_vector (31 downto 0);
   flush : IN std_logic;
   command : OUT command_type;
   pd_source1 : OUT register_type;
   pd_source2 : OUT register_type;
   pd_destination : OUT register_type;
   pd_data : OUT std_logic_vector (31 downto 0);
   pd_read : OUT std_logic
   );
END COMPONENT;

SIGNAL data : std_logic_vector (31 downto 0);
SIGNAL source1 : std_logic_vector (3 downto 0);
SIGNAL source2 : std_logic_vector (3 downto 0);
SIGNAL clock : std_logic := '0';
SIGNAL inst : std_logic_vector (2 downto 0);
SIGNAL destination : std_logic_vector (3 downto 0);
SIGNAL flush : std_logic;
SIGNAL command : command_type;
SIGNAL pd_source1 : register_type;
SIGNAL pd_source2 : register_type;
SIGNAL pd_destination : register_type;
SIGNAL pd_data : std_logic_vector (31 downto 0);
```

```
SIGNAL pd_read : std_logic;
CONSTANT CYCLE : TIME := 50 ns;
BEGIN
DUT: predecode_ent port map (clock,inst,source1,source2,
destination,data,flush,command,pd_source1,pd_source2,
pd_destination,pd_data,pd_read);
clock <= NOT clock AFTER CYCLE/2;
PROCESS
BEGIN
-- default output set to 0
source1 <= "0000";
source2 <= "0000";
destination <= "0000";
data <= ZERO;
inst <= "111";
-- flush is 0, no flushing
flush <= '0';
-- load "4592fa83" into reg0 instruction
inst <= "101";
data <= "01000101100100101111101010000011";
destination <= "0000";
wait for CYCLE;
-- load "00000001" into reg15 instruction
inst <= "101";
data <= "00000000000000000000000000000001";
destination <= "1111";
wait for CYCLE;
-- mov reg0, reg8
inst <= "000";
source1 <= "0000";
destination <= "1000";
wait for CYCLE;
-- add reg0, reg15, reg1
   inst <= "001";
   source1 <= "0000";
   source2 <= "1111";
   destination <= "0001";
   wait for CYCLE;
   --no operation
   inst <= "111";
   wait for CYCLE;
   inst <= "111";
   wait for CYCLE;
END PROCESS;
END predecode_tb_arch;
CONFIGURATION predecode_tb_config OF predecode_tb_ent IS
```

```
   FOR predecode_tb_arch
      FOR ALL: predecode_ent
         USE entity work.predecode_ent(predecode_arch);
      END FOR;
   END FOR;
END predecode_tb_config;
```

Slika 28-4 *Testbench* program za *predecode* blok

7. Modifikovati kôd *testbench* fajla tako da *testbench* stimuli{e *predecode* blok slede}im instrukcijama:

Grupa 1:
```
Load #183FFA20, reg2
Load #0210AA18, reg4
Add reg2, reg4, reg8
Read reg8
Nop
```

Grupa 2:
```
Load #183FFA20, reg5
Load #0210AA18, reg3
Sub reg3, reg5, reg9
Read reg9
Move reg3, reg5
Read reg5
```

Grupa 3:
```
Load #99999999, reg0
Nop
Load #11111111, reg1
Mul reg0, reg1, reg2
Read reg0
Read reg2
```

Grupa 4:
```
Load #00000011, reg2
Move reg2, reg5
Load #00000001, reg3
Read reg5
Sub reg3, reg5, reg1
Read reg1
```

Grupa 5:
```
Load #AAAABBBB, reg0
Load #BBBBAAAA, reg1
Mul reg0, reg1, reg2
Nop
Read reg1
Read reg2
```

Grupa 6:
```
Load #00001111, reg0
```

Grupa 11:
```
_____________________
_____________________
_____________________
_____________________
_____________________
```

Grupa 12:
```
_____________________
_____________________
_____________________
_____________________
_____________________
```

Grupa 13:
```
_____________________
_____________________
_____________________
_____________________
_____________________
_____________________
```

Grupa 14:
```
_____________________
_____________________
_____________________
_____________________
_____________________
_____________________
```

Grupa 15:
```
_____________________
_____________________
_____________________
_____________________
_____________________
_____________________
```

Grupa 16:
```
_____________________
```

```
Read reg0                    _____________________
Load #1528FCDA, reg10        _____________________
Load #F8910004, reg11        _____________________
Add reg10, reg11, reg0       _____________________
```

Grupa 7:

```
Load #00000004, reg9         _____________________
Load #00001089, reg13        _____________________
Sub reg9, reg13, reg1        _____________________
Move reg9, reg13             _____________________
Read reg1                    _____________________
Read reg13                   _____________________
```

Grupa 17:

Grupa 8:

```
Load #98765432, reg0         _____________________
Read reg0                    _____________________
Load #12345678, reg1         _____________________
Read reg1                    _____________________
Sub reg1, reg0, reg3         _____________________
Read reg3                    _____________________
```

Grupa 18:

Grupa 9:

```
Load #87AC9310, reg8         _____________________
Load #88341502, reg1         _____________________
Add reg1, reg8, reg2         _____________________
Sub reg1, reg8, reg0         _____________________
Read reg0                    _____________________
Read reg2                    _____________________
```

Grupa 19:

Grupa 10:

```
Load #148CF9A7, reg3         _____________________
Nop                          _____________________
Mul reg3, reg3, reg3         _____________________
Load #71835942, reg2         _____________________
Add reg2, reg3, reg2         _____________________
Read reg2                    _____________________
```

Grupa 20:

8. Snimiti modifikovani fajl (***File → Save***), a zatim ga kompajlirati pritiskom na F11 (ili ***Compile***).
9. Ukoliko je kompilacija izvr{ena korektno ispred fajla ***predecode_testbench.vhd*** u ***Design Browser*** prozoru pojavi}e se oznaka √. Ukoliko je do{lo do gre{ke u kompilaciji ispred imena fajla pojavi}e se oznaka X, a u prozoru editora bi}e podvu~ene linije kôda koje nisu mogle biti kompajlirane. Postavljanjem kursora mi{a na podvu~enu liniju pojavljuje se prozor sa izve{tajem o gre{ci. Nakon ispravljanja gre{aka faj treba ponovo kompajlirati pritiskom na F11.
10. Nakon uspe{ne simulacije kliknuti na znak + ispred fajla ***predecode_testbench*** a zatim desnim tasterom mi{a na ***predecode_tb_ent*** (***predecode_tb_arch***). U padaju}em meniju izabrati

opciju ***Set as Top_Level***, kliknuti desnim tasterom mi{a i izabrati ***Simulation→ Initialize Simulation***.

11. Selektovanjem opcija ***File → New→ Waveform*** otvoriti ***waveform*** editor.

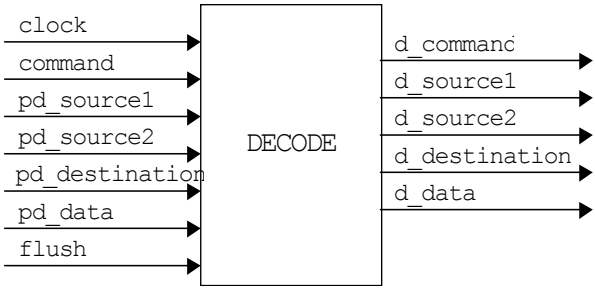
12. Iz ***Structure*** kartice ***Design Browser***-a levim tasterom mi{a prevu}i ***DUT: predecode_ent*** u prozor ***Waveform*** editor-a.

13. Pokrenuti simulaciju selektovanjem opcija ***Simulation→ Run Until...*** pri ~emu u polju za vreme trajanja simulacije treba upisati 500 ns.

14. Analizitati talasne oblike dobijene simulacijom i utvrditi da li ***predecode*** blok funkcioni{e ispravno.

Decode blok

1. Predstavljeni su interfejs signali ***Decode*** bloka (slika 28-5).
2. Opisani su interfejs signali ***Decode*** bloka (slika 28-6).
3. Analizirati definicije signala datih na slikama 28-5 i 28-6.
4. Analizirati VHDL kôd, prikazan na slici 28-7, kojim se opisuje rad ***Decode*** bloka.
5. Analizirati ***testbench*** program, prikazan na slici 28-8, kojim se verifikuje rad ***Decode*** bloka.



Slika 28-5 Interfejs signali bloka ***decode***

Ime signala	I/O	Opis
clock	ulaz	Taktni signal.
command	ulaz	Ulaz tipa ***command_type*** koji mo`e biti: MOVE, ADD, SUB, MUL, CJE, LOAD, READ i NOP.
pd_source1	ulaz	Ulaz tipa ***register_type*** koji mo`e biti neki od reg0,reg1,reg2,reg3,reg4,reg5,reg6, reg7,reg8,reg9,reg10,reg11,reg12, reg13,reg14,reg15 Odredjuje registar koji }e se koristiti kao <source1>.
pd_source2	ulaz	Ulaz tipa ***register_type*** koji mo`e biti neki od

		reg0,reg1,reg2,reg3,reg4,reg5,reg6, reg7,reg8,reg9,reg10,reg11,reg12, reg13,reg14,reg15 Odredjuje registar koji }e se koristiti kao <source2>.
pd_destination	ulaz	Ulaz tipa ***register_type*** koji mo`e biti neki od reg0,reg1,reg2,reg3,reg4,reg5,reg6, reg7,reg8,reg9,reg10,reg11,reg12, reg13,reg14,reg15 Odredjuje registar koji }e se koristiti kao <destination>.
pd_data	ulaz	32-bitna linija kojom se prosledjuju podaci izmedju ***predecode*** i ***decode*** bloka za vreme LOAD instrukcije.
flush	ulaz	Kada je na visokom nivou, ***predecode*** blok mora da udje u ***flush*** mod. Ovo se de{ava kada dodje do grananja.
d_command	izlaz	Prosledjivanje instrukcije u ***execute*** blok. Tip je ***command_type*** koji mo`e biti: MOVE, ADD, SUB, MUL, CJE, LOAD, READ i NOP.
d_destination	izlaz	Prosledjivanje informacije sa pd_destination iz ***predecode*** u ***execute*** blok.
d_source1	izlaz	Prosledjivanje informacije sa pd_source1 iz ***predecode*** bloka u ***execute*** blok.
d_source2	izlaz	Prosledjivanje informacije sa pd_source2 iz ***predecode*** bloka u ***execute*** blok.
d_data	izlaz	Prosledjivanje informacije sa pd_data iz ***predecode*** bloka u ***execute*** blok.

Slika 28-6 Opis interfejs signala bloka ***decode***

Postupak rada

1. U programu Active-HDL 3.6 otvoriti direktorijum ***ime_broj indeksa***.
2. Dva puta kliknuti na ***Add New Files*** a zatim na ***Add Existing File***. U direktorijumu C:\VEZBE\PMIC\ selektovati fajl ***decode_ent.vhd*** i kliknuti na ***Add***. Nakon toga fajl }e biti prikazan u ***Design Browser*** prozoru pod istim imenom.
3. Dvostrukim klikom mi{a na fajl pogledati njegov izvorni kôd i proveriti da li se sla`e sa kôdom na Slici 7.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
```

```
USE WORK.pipeline_package.ALL;
ENTITY decode_ent IS
PORT  (
   clock : IN std_logic;
   command : IN command_type;
pd_source1 : IN register_type;
   pd_source2 : IN register_type;
   pd_destination : IN register_type;
   pd_data : IN std_logic_vector (31 downto 0);
   flush : IN std_logic;
   d_command : OUT command_type;
d_destination : OUT register_type;
d_source1 : OUT register_type;
   d_source2 : OUT register_type;
d_data : OUT std_logic_vector (31 downto 0);
   );
END decode_ent;
ARCHITECTURE decode_arch OF decode_ent IS
BEGIN
PROCESS (clock, command, pd_source1, pd_source2,
  pd_destination, flush, pd_data)
BEGIN
      IF (clock = '1' AND clock'EVENT) THEN
         IF (flush = '0') THEN
            CASE command IS
               WHEN MOVE =>
            -- MOVE <source1>, <destination>
            -- <source2> is defaulted to reg0
            -- d_data defaulted to all zero
            -- since it is only used during
            -- LOAD
            d_source1 <= pd_source1;
                  d_source2 <= reg0;
                  d_destination <= pd_destination;
                  d_data <= ZERO;
               WHEN ADD =>
            -- ADD <source1>, <source2>, <destination>
            -- d_data default to all zeros
            d_source1 <= pd_source1;
                  d_source2 <= pd_source2;
                  d_destination <= pd_destination;
                  d_data <= ZERO;
               WHEN SUB =>
            -- SUB  <source1>, <source2>, <destination>
            -- d_data default to ZERO
            d_source1 <= pd_source1;
                  d_source2 <= pd_source2;
                  d_destination <= pd_destination;
                  d_data <= ZERO;
```

```
            WHEN MUL =>
         -- MUL  <source1>, <source2>, <destination>
         -- d_data again default to all zero
         d_source1 <= pd_source1;
               d_source2 <= pd_source2;
               d_destination <= pd_destination;
               d_data <= ZERO;
            WHEN CJE =>
         -- CJE   <source1>, <source2>, <destination>
         -- d_data default to all zero
         d_source1 <= pd_source1;
               d_source2 <= pd_source2;
               d_destination <= pd_destination;
               d_data <= ZERO;
            WHEN LOAD =>
         -- LOAD <value>, <destination>
         -- d_data passes the data from predecode
         -- block to execute block. <source1> and
         -- <source2> are defaulted to reg0 they
         -- are not used in this instruction.
               d_data <= pd_data;
         d_source1 <= reg0;
               d_source2 <= reg0;
               d_destination <= pd_destination;
            WHEN READ =>
         -- READ <destination>
         -- <source1> and <source2> defaults to
         -- reg0 as they are not used.
         -- d_data default to all zero.
         d_source1 <= reg0;
               d_source2 <= reg0;
               d_destination <= pd_destination;
               d_data <= ZERO;
            WHEN NOP =>
         -- no operation. all outputs to default
         -- value.
         d_source1 <= reg0;
               d_source2 <= reg0;
               d_destination <= reg0;
               d_data <= ZERO;
            WHEN OTHERS =>
               NULL;
         END CASE;
         d_command <= command;
      ELSE
   d_source1 <= reg0;
         d_source2 <= reg0;
         d_destination <= reg0;
         d_data <= ZERO;
         d_command <= NOP;
```

```
         END IF;
      END IF;
   END PROCESS;
END decode_arch;
```

Slika 28-7 VHDL kôd bloka ***decode***

4. Desnim tasterom mi{a kliknuti na fajl ***decode_ent.vhd*** u ***Design Browser*** prozoru, a zatim ga kompajlirati pritiskom na ***Compile,*** u padaju}em meniju. Ukoliko je kompilacija izvr{ena korektno ispred imena fajla ***decode_ent.vhd*** u ***Design Browser*** prozoru pojavi}e se oznaka √. Ukoliko je do{lo do gre{ke u kompilaciji ispred imena fajla pojavi}e se oznaka X.

5. Dva puta kliknuti na ***Add New Files***, a zatim na ***Add Existing File***. U direktorijumu C:\VEZBE\PMIC\ selektovati fajl ***decode_testbench.vhd*** a zatm kliknuti na ***Add***. Nakon toga }e se selektovani fajl pojaviti u ***Design Browser*** prozoru.

6. Dvostrukim klikom na fajl ***decode_testbench.vhd*** pogledati njegov izvorni kôd u prozoru editora. Na Slici 8 dat je ***testbench*** program koji simulira rad ***decode*** bloka pomo}u slede}e ~etiri instrukcije

```
Load #10051, reg0
Load #1, reg1
Add reg0, reg1, reg2
Sub   reg0, reg1, reg3
```

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE WORK.pipeline_package.ALL;

ENTITY decode_tb_ent IS
END decode_tb_ent;

ARCHITECTURE decode_tb_arch OF decode_tb_ent IS
COMPONENT decode_ent
PORT  (
   clock : IN std_logic;
   command : IN command_type;
pd_source1 : IN register_type;
   pd_source2 : IN register_type;
   pd_destination : IN register_type;
   pd_data : IN std_logic_vector (31 downto 0);
   flush : IN std_logic;
   d_command : OUT command_type;
d_destination : OUT register_type;
d_source1 : OUT register_type;
   d_source2 : OUT register_type;
d_data : OUT std_logic_vector (31 downto 0);
   );
END COMPONENT;

SIGNAL d_command : command_type;
SIGNAL d_destination : register_type;
```

```
SIGNAL d_source1 : register_type;
SIGNAL d_source2 : register_type;
SIGNAL pd_source1 : register_type;
SIGNAL pd_source2 : register_type;
SIGNAL clock : std_logic := '0';
SIGNAL pd_destination : register_type;
SIGNAL flush : std_logic;
SIGNAL command : command_type;
SIGNAL pd_data : std_logic_vector (31 downto 0);
SIGNAL d_data : std_logic_vector (31 downto 0);
CONSTANT CYCLE : TIME := 50 ns;
BEGIN
DUT: decode_ent port map (clock, command, pd_source1, pd_source2,
pd_destination, pd_data, flush, d_command,
d_destination, d_source1, d_source2, d_data);
clock <= NOT clock AFTER CYCLE/2;
PROCESS
BEGIN
flush <= '0';
wait for CYCLE;
-- load, #10051, reg0
command <= LOAD;
pd_destination <= reg0;
pd_data <= "00000000000000010000000001010001";
wait for CYCLE;
-- load #1, reg1
command <= LOAD;
pd_destination <= reg1;
pd_data <= "00000000000000000000000000000001";
wait for CYCLE;
-- add reg0, reg1, reg2
command <= ADD;
pd_source1 <= reg0;
pd_source2 <= reg1;
pd_destination <= reg2;
wait for CYCLE;
-- sub reg0, reg1, reg3
command <= SUB;
pd_source1 <= reg0;
pd_source2 <= reg1;
pd_destination <= reg3;
wait for CYCLE;
END PROCESS;
END decode_tb_arch;
CONFIGURATON decode_tb_config OF decode_tb_ent IS
   FOR decode_tb_arch
      FOR ALL : decode_ent
```

```
         USE ENTITY WORK. decode_ent (decode_arch);
      END FOR;
   END FOR;
END decode_tb_CONFIG;
```

Slika 28-8 *Testbench* program za *decode* blok

7. Modifikovati kôd *testbench* fajla tako da *testbench* stimuli{e *predecode* blok slede}im instrukcijama:

Grupa 1:
```
Load #183FFA20, reg2
Load #0210AA18, reg4
Add reg2, reg4, reg8
Read reg8
Nop
```

Grupa 2:
```
Load #183FFA20, reg5
Load #0210AA18, reg3
Sub reg3, reg5, reg9
Read reg9
Move reg3, reg5
Read reg5
```

Grupa 3:
```
Load #99999999, reg0
Nop
Load #11111111, reg1
Mul reg0, reg1, reg2
Read reg0
Read reg2
```

Grupa 4:
```
Load #00000011, reg2
Move reg2, reg5
Load #00000001, reg3
Read reg5
Sub reg3, reg5, reg1
Read reg1
```

Grupa 5:
```
Load #AAAABBBB, reg0
Load #BBBBAAAA, reg1
Mul reg0, reg1, reg2
Nop
Read reg1
Read reg2
```

Grupa 6:
```
Load #00001111, reg0
Read reg0
Load #1528FCDA, reg10
```

Grupa 11:
```
_____________________
_____________________
_____________________
_____________________
_____________________
```

Grupa 12:
```
_____________________
_____________________
_____________________
_____________________
_____________________
```

Grupa 13:
```
_____________________
_____________________
_____________________
_____________________
_____________________
```

Grupa 14:
```
_____________________
_____________________
_____________________
_____________________
_____________________
_____________________
```

Grupa 15:
```
_____________________
_____________________
_____________________
_____________________
_____________________
_____________________
```

Grupa 16:
```
_____________________
_____________________
_____________________
```

```
Load #F8910004, reg11      _____________________
Add reg10, reg11, reg0     _____________________
```

Grupa 7:
```
Load #00000004, reg9       _____________________
Load #00001089, reg13      _____________________
Sub reg9, reg13, reg1      _____________________
Move reg9, reg13           _____________________
Read reg1                  _____________________
Read reg13                 _____________________
```

Grupa 8:
```
Load #98765432, reg0       _____________________
Read reg0                  _____________________
Load #12345678, reg1       _____________________
Read reg1                  _____________________
Sub reg1, reg0, reg3       _____________________
Read reg3                  _____________________
```

Grupa 9:
```
Load #87AC9310, reg8       _____________________
Load #88341502, reg1       _____________________
Add reg1, reg8, reg2       _____________________
Sub reg1, reg8, reg0       _____________________
Read reg0                  _____________________
Read reg2                  _____________________
```

Grupa 10:
```
Load #148CF9A7, reg3       _____________________
Nop                        _____________________
Mul reg3, reg3, reg3       _____________________
Load #71835942, reg2       _____________________
Add reg2, reg3, reg2       _____________________
Read reg2                  _____________________
```

Grupa 17:
```
                           _____________________
                           _____________________
                           _____________________
                           _____________________
                           _____________________
                           _____________________
```

Grupa 18:
```
                           _____________________
                           _____________________
                           _____________________
                           _____________________
                           _____________________
                           _____________________
```

Grupa 19:
```
                           _____________________
                           _____________________
                           _____________________
                           _____________________
                           _____________________
                           _____________________
```

Grupa 20:
```
                           _____________________
                           _____________________
                           _____________________
                           _____________________
                           _____________________
                           _____________________
```

8. Snimiti modifikovani fajl (***File → Save***), a zatim ga kompajlirati pritiskom na F11 (ili ***Compile***).

9. Ukoliko je kompilacija izvr{ena korektno ispred fajla ***decode_testbench.vhd*** u ***Design Browser*** prozoru pojavi}e se oznaka √. Ukoliko je do{lo do gre{ke u kompilaciji ispred imena fajla pojavi}e se oznaka X, a u prozoru editora bi}e podvu~ene linije kôda koje nisu mogle biti kompajlirane. Postavljanjem kursora mi{a na podvu~enu liniju pojavljuje se prozor sa izve{tajem o gre{ci. Nakon ispravljanja gre{aka faj treba ponovo kompajlirati pritiskom na F11.

10. Nakon uspe{ne simulacije kliknuti na znak + ispred fajla ***decode_testbench*** a zatim desnim tasterom mi{a na ***decode_tb_ent*** (***decode_tb_arch***). U padaju}em meniju izabrati opciju ***Set as Top_Level***, kliknuti desnim tasterom mi{a i izabrati ***Simulation→ Initialize Simulation***.

11. Selektovanjem opcija ***File → New→ Waveform*** otvoriti ***waveform*** editor.
12. Iz ***Structure*** kartice ***Design Browser***-a levim tasterom mi{a prevu}i ***DUT: decode_ent*** u prozor ***Waveform*** editor-a.
13. Pokrenuti simulaciju selektovanjem opcija ***Simulation→ Run Until...*** pri ~emu u polju za vreme trajanja simulacije treba upisati 500 ns.
14. Analizitati talasne oblike dobijene simulacijom i utvrditi da li ***decode*** blok funkcioni{e ispravno.

LABORATORIJSKA VEŽBA 29

Register file i *Execute* blokovi

Uvod

U ovoj ve`bi analizira}emo rad ***Register-file*** i ***Execute*** blokova. Ve`ba je organizovana na slede}i na~in:

Register file blok

1. Predstavljeni su interfejs signali ***Register file*** bloka (Slika 29-1).
2. Opisani su interfejs signali ***Register file*** bloka (Slika 29-2).
3. Analizirati definicije signala datih na Slikama 29-1 i 29-2.
4. Analizirati VHDL kôd, prikazan na Slici 29-3, kojim se opisuje rad ***Register-file*** bloka.
5. Analizirati ***testbench*** program, prikazan na Slici 29-4, kojim se verifikuje rad ***Register-file*** bloka.

Slika 29-1 Interfejs signali kod ***register file*** bloka

ime signala	**I/O**	**opis**
clock	ulaz	signal takta
flush	ulaz	'1', ***register file*** blok
pd_read	ulaz	***execute*** blok MOVE, ADD, SUB, MUL, CJE ili READ. pd_source1 i pd_source2 ulazi pd_source1 i pd_source2 source1_data i source2_data izlazi.
pd_source1	ulaz	pd_source1 <source1>.
pd_source2	ulaz	pd_source2 <source2>.
ex_store	ulaz	'1', ***register file*** blok ex_data ex_destination.
ex_data	ulaz	***execute*** blok u ***register file*** blok
ex_destination	ulaz	***register file*** blok ex_data ex_store
source1_data	izlaz	<source1>
source2_data	izlaz	<source2>

Slika 29-2 Opis interfejs signala ***register-file*** bloka

Postupak rada

1. U programu Active-HDL 3.6 otvoriti direktorijum ***ime_broj indeksa***.
2. Dva puta kliknuti na ***Add New Files*** a zatim na ***Add Existing File***. U direktorijumu C:\VEZBE\PMIC\ selektovati fajl ***register_file_ent.vhd*** i kliknuti na ***Add***. Nakon toga fajl }e biti prikazan u ***Design Browser*** prozoru pod istim imenom.
3. Dvostrukim klikom mi{a na fajl pogledati njegov izvorni kôd i proveriti da li se sla`e sa kôdom na Slici 29-3.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE WORK.pipeline_package.ALL;

ENTITY decode_ent IS
PORT  (
pd_read : IN std_logic;
pd_source1 : IN register_type;
pd_source2 : IN register_type;
   clock : IN std_logic;
   flush : IN std_logic;
   ex_store : IN std_logic;
   ex_destination : IN register_type;
   ex_data : IN std_logic_vector (31 downto 0);
source1_data : OUT std_logic_vector (31 downto 0);
   source2_data : OUT std_logic_vector (31 downto 0);
   );
END register_file_ent;
```

```
ARCHITECTURE register_file_arch OF register_file_ent IS
SIGNAL reg : array_size;
BEGIN
PROCESS (clock, flush, pd_read, pd_source1, pd_source2)
BEGIN
      IF (clock = '1' AND clock'EVENT) THEN
         IF (flush = '0') THEN
            IF (pd_read = '1') THEN
               IF (pd_source1 /= ex_destination)
               THEN
                  CASE pd_source1 IS
                     WHEN reg0 =>
                        source1_data <= reg(0);
                     WHEN reg1 =>
                        source1_data <= reg(1);
                     WHEN reg2 =>
                        source1_data <= reg(2);
                     WHEN reg3 =>
                        source1_data <= reg(3);
                     WHEN reg4 =>
                        source1_data <= reg(4);
                     WHEN reg5 =>
                        source1_data <= reg(5);
                     WHEN reg6 =>
                        source1_data <= reg(6);
                     WHEN reg7 =>
                        source1_data <= reg(7);
                     WHEN reg8 =>
                        source1_data <= reg(8);
                     WHEN reg9 =>
                        source1_data <= reg(9);
                     WHEN reg10 =>
                        source1_data <= reg(10);
                     WHEN reg11 =>
                        source1_data <= reg(11);
                     WHEN reg12 =>
                        source1_data <= reg(12);
                     WHEN reg13 =>
                        source1_data <= reg(13);
                     WHEN reg14 =>
                        source1_data <= reg(14);
                     WHEN reg15 =>
                        source1_data <= reg(15);
                     WHEN OTHERS =>
                        NULL;
                  END CASE;
                     ELSE
                        source1_data <= ex_data;
                     END IF;
                  IF (pd_source2 /= ex_destination)
```

```
      THEN
         CASE pd_source1 IS
            WHEN reg0 =>
               source2_data <= reg(0);
            WHEN reg1 =>
               source2_data <= reg(1);
            WHEN reg2 =>
               source2_data <= reg(2);
            WHEN reg3 =>
               source2_data <= reg(3);
            WHEN reg4 =>
               source2_data <= reg(4);
            WHEN reg5 =>
               source2_data <= reg(5);
            WHEN reg6 =>
               source2_data <= reg(6);
            WHEN reg7 =>
               source2_data <= reg(7);
            WHEN reg8 =>
               source2_data <= reg(8);
            WHEN reg9 =>
               source2_data <= reg(9);
            WHEN reg10 =>
               source2_data <= reg(10);
            WHEN reg11 =>
               source2_data <= reg(11);
            WHEN reg12 =>
               source2_data <= reg(12);
            WHEN reg13 =>
               source2_data <= reg(13);
            WHEN reg14 =>
               source2_data <= reg(14);
            WHEN reg15 =>
               source2_data <= reg(15);
            WHEN OTHERS =>
               NULL;
         END CASE;
      ELSE
         source2_data <= ex_data;
      END IF;
   END IF;
   IF (ex_store = '1') THEN
      CASE ex_destination IS
         WHEN reg0 =>
            reg(0) <= ex_data;
         WHEN reg1 =>
            reg(1) <= ex_data;
         WHEN reg2 =>
            reg(2) <= ex_data;
         WHEN reg3 =>
```

```
                        reg(3) <= ex_data;
                     WHEN reg4 =>
                        reg(4) <= ex_data;
                     WHEN reg5 =>
                        reg(5) <= ex_data;
                     WHEN reg6 =>
                        reg(6) <= ex_data;
                     WHEN reg7 =>
                        reg(7) <= ex_data;
                     WHEN reg8 =>
                        reg(8) <= ex_data;
                     WHEN reg9 =>
                        reg(9) <= ex_data;
                     WHEN reg10 =>
                        reg(10) <= ex_data;
                     WHEN reg11 =>
                        reg(11) <= ex_data;
                     WHEN reg12 =>
                        reg(12) <= ex_data;
                     WHEN reg13 =>
                        reg(13) <= ex_data;
                     WHEN reg14 =>
                        reg(14) <= ex_data;
                     WHEN reg15 =>
                        reg(15) <= ex_data;
                     WHEN OTHERS =>
                        NULL;
                  END CASE;
               END IF;
            ELSE
               source1_data <= ZERO;
               source2_data <= ZERO;
            END IF;
         END IF;
      END PROCESS;
END register_file_arch;
```

Slika 29-3 VHDL kôd bloka *register file*

4. Desnim tasterom mi{a kliknuti na fajl ***register_file_ent.vhd*** u ***Design Browser*** prozoru, a zatim ga kompajlirati pritiskom na ***Compile,*** u padaju}em meniju. Ukoliko je kompilacija izvr{ena korektno ispred imena fajla ***register_file_ent.vhd*** u ***Design Browser*** prozoru pojavi}e se oznaka √. Ukoliko je do{lo do gre{ke u kompilaciji ispred imena fajla pojavi}e se oznaka X.
5. Dva puta kliknuti na ***Add New Files***, a zatim na ***Add Existing File***. U direktorijumu C:\VEZBE\PMIC\ selektovati fajl ***register_file_testbench.vhd*** a zatm kliknuti na ***Add***. Nakon toga }e se selektovani fajl pojaviti u ***Design Browser*** prozoru.
6. Dvostrukim klikom na fajl ***register_file_testbench.vhd*** pogledati njegov izvorni kôd u prozoru editora. Na Slici 29-4 dat

je ***testbench*** program koji simulira rad ***register file*** bloka pomo}u slede}e ~etiri instrukcije

```
Load #10051, reg0
      Load #1, reg1
      Add reg0, reg1, reg2
Sub reg0, reg1, reg3
```

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE WORK.pipeline_package.ALL;
```

```
ENTITY register_file_tb_ent IS
END register_file_tb_ent;
```

```
ARCHITECTURE register_file_tb_arch OF register_file_tb_ent IS
COMPONENT register_file_ent
PORT  (
   pd_read : IN std_logic;
pd_source1 : IN register_type;
   pd_source2 : IN register_type;
clock : IN std_logic;
flush : IN std_logic;
   ex_store : IN std_logic;
   ex_destination : IN register_type;
   ex_data : IN std_logic_vector (31 downto 0);
source1_data : OUT std_logic_vector (31 downto 0);
source2_data : OUT std_logic_vector (31 downto 0);
   );
END COMPONENT;
```

```
SIGNAL pd_read : std_logic;
SIGNAL pd_source1 : register_type;
SIGNAL pd_source2 : register_type;
SIGNAL clock : std_logic := '0';
SIGNAL flush : std_logic;
SIGNAL ex_store : std_logic;
SIGNAL ex_destination : register_type;
SIGNAL ex_data : std_logic_vector (31 downto 0);
SIGNAL source1_data : std_logic_vector (31 downto 0);
SIGNAL source2_data : std_logic_vector (31 downto 0);
```

```
CONSTANT CYCLE : TIME := 50 ns;
```

```
BEGIN
```

```
DUT: register_file_ent port map (pd_read, pd_source1, pd_source2,
clock, flush, ex_store, ex_destination, ex_data, source1_data,
source2_data);
```

```
clock <= NOT clock AFTER CYCLE/2;
```

```
PROCESS
BEGIN
```

-- set default to 0

```
flush <= '0';
pd_read <= '0';
```

```
pd_source1 <= reg0;
pd_source2 <= reg0;
ex_store <= '0';
ex_destination <= reg0;
ex_data <= ZERO;
wait for CYCLE;
-- load #10051, reg0
ex_store <= '1';
ex_destination <= reg0;
ex_data <= "00000000000000000000000001010001";
wait for CYCLE;
-- load #1, reg1
ex_store <= '1';
ex_destination <= reg1;
ex_data <= "00000000000000000000000000000001";
wait for CYCLE;
-- add reg0, reg1, reg2
-- reading values of reg0 and reg1
ex_store <= '0';
pd_read <= '1';
pd_source1 <= reg0;
pd_source2 <= reg1;
wait for CYCLE;
-- sub reg0, reg1, reg3
-- reading values of reg0 and reg1
ex_store <= '0';
pd_read <= '1';
pd_source1 <= reg0;
pd_source2 <= reg1;
wait for CYCLE;
END PROCESS;
END register_file_tb_arch;
CONFIGURATON register_file_tb_config OF register_file_tb_ent IS
   FOR register_file_tb_arch
      FOR ALL : register_file_ent
         USE ENTITY WORK. register_file_ent
      (register_file_arch);
      END FOR;
   END FOR;
END register_file_tb_config;
```

Slika 29-4 ***Testbench*** program za ***register file*** blok

7. Modifikovati kôd ***testbench*** fajla tako da ***testbench*** stimuli{e ***register file*** blok slede}im instrukcijama:

Grupa 1:	**Grupa 11:**
Load #183FFA20, reg2	_____
Load #0210AA18, reg4	_____

```
Add reg2, reg4, reg8
Read reg8
Nop
```

Grupa 2:

```
Load #183FFA20, reg5
Load #0210AA18, reg3
Sub reg3, reg5, reg9
Read reg9
Move reg3, reg5
Read reg5
```

Grupa 3:

```
Load #99999999, reg0
Nop
Load #11111111, reg1
Mul reg0, reg1, reg2
Read reg0
Read reg2
```

Grupa 4:

```
Load #00000011, reg2
Move reg2, reg5
Load #00000001, reg3
Read reg5
Sub reg3, reg5, reg1
Read reg1
```

Grupa 5:

```
Load #AAAABBBB, reg0
Load #BBBBAAAA, reg1
Mul reg0, reg1, reg2
Nop
Read reg1
Read reg2
```

Grupa 6:

```
Load #00001111, reg0
Read reg0
Load #1528FCDA, reg10
Load #F8910004, reg11
Add reg10, reg11, reg0
```

Grupa 7:

```
Load #00000004, reg9
Load #00001089, reg13
Sub reg9, reg13, reg1
Move reg9, reg13
Read reg1
Read reg13
```

Grupa 8:

```
Load #98765432, reg0
```

Grupa 12:

Grupa 13:

Grupa 14:

Grupa 15:

Grupa 16:

Grupa 17:

Grupa 18:

```
Read reg0                    _____________________
Load #12345678, reg1         _____________________
Read reg1                    _____________________
Sub reg1, reg0, reg3         _____________________
Read reg3                    _____________________
```

Grupa 9:

```
Load #87AC9310, reg8         _____________________
Load #88341502, reg1         _____________________
Add reg1, reg8, reg2         _____________________
Sub reg1, reg8, reg0         _____________________
Read reg0                    _____________________
Read reg2                    _____________________
```

Grupa 19:

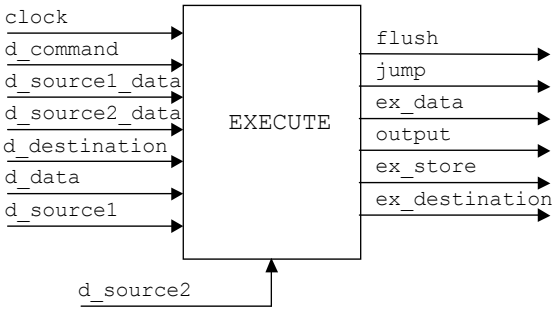
```
Load #148CF9A7, reg3         _____________________
Nop                          _____________________
Mul reg3, reg3, reg3         _____________________
Load #71835942, reg2         _____________________
Add reg2, reg3, reg2         _____________________
Read reg2                    _____________________
```

Grupa 20:

8. Snimiti modifikovani fajl (***File → Save***), a zatim ga kompajlirati pritiskom na F11 (ili ***Compile***).
9. Ukoliko je kompilacija izvr{ena korektno ispred fajla ***register_file_testbench.vhd*** u ***Design Browser*** prozoru pojavi}e se oznaka √. Ukoliko je do{lo do gre{ke u kompilaciji ispred imena fajla pojavi}e se oznaka X, a u prozoru editora bi}e podvu~ene linije kôda koje nisu mogle biti kompajlirane. Postavljanjem kursora mi{a na podvu~enu liniju pojavljuje se prozor sa izve{tajem o gre{ci. Nakon ispravljanja gre{aka faj treba ponovo kompajlirati pritiskom na F11.
10. Nakon uspe{ne simulacije kliknuti na znak + ispred fajla ***register_file_testbench*** a zatim desnim tasterom mi{a na ***register_file_tb_ent*** (***register_file_tb_arch***). U padaju}em meniju izabrati opciju ***Set as Top_Level***, kliknuti desnim tasterom mi{a i izabrati ***Simulation→ Initialize Simulation***.
11. Selektovanjem opcija ***File → New→ Waveform*** otvoriti ***waveform*** editor.
12. Iz ***Structure*** kartice ***Design Browser***-a levim tasterom mi{a prevu}i ***DUT: register_file_ent*** u prozor ***Waveform*** editor-a.
13. Pokrenuti simulaciju selektovanjem opcija ***Simulation→ Run Until...*** pri ~emu u polju za vreme trajanja simulacije treba upisati 500 ns.
14. Analizitati talasne oblike dobijene simulacijom i utvrditi da li ***register file*** blok funkcioni{e ispravno.

Execute blok

1. Predstavljeni su interfejs signali ***Execute*** bloka (Slika 29-5).
2. Opisani su interfejs signali ***Execute*** bloka (Slika 29-6).
3. Analizirati definicije signala datih na Slikama 29-5 i 29-6.
4. Analizirati VHDL kôd, prikazan na Slici 29-7, kojim se opisuje rad ***Execute*** bloka.
5. Analizirati ***testbench*** program, prikazan na Slici 29-8, kojim se verifikuje rad ***Execute*** bloka.



Slika 29-5 Interfejs signali bloka ***execute***

ime signala	**I/O**	**opis**
d_command	ulaz	command_type MOVE,ADD,SUB,MUL,CJE,LOAD,READ ili NOP.
d_source1_data		***register file*** blok ***execute*** blok ADD,SUB,MUL i CJE
d_source2_data		***register file*** blok ***execute*** blok ADD,SUB,MUL i CJE
d_destination		register_type reg0,reg1,reg2,reg3,reg4,reg5,reg6,reg7,reg8,reg9,reg10,reg11,reg12,reg13,reg14,reg15
d_source1		register_type reg0,reg1,reg2,reg3,reg4,reg5,reg6,reg7,reg8,reg9,reg10,reg11,reg12,reg13,reg14,reg15
d_source2		register_type reg0,reg1,reg2,reg3,reg4,reg5,reg6,reg7,reg8,reg9,reg10,reg11,reg12,reg13,reg14,reg15
d_data	ulaz	***decode*** blok ***execute*** blok
clock	ulaz	signal takta
flush	izlaz	'1',
jump		'1',

ex_data	izlaz	*execute* blok u *register file* blok
ex_store	ulaz	'1', *register file* blok ex_data ex_destination.
ex_destination	ulaz	register_type reg0,reg1,reg2,reg3,reg4,reg5,reg6, reg7,reg8,reg9,reg10,reg11,reg12, reg13,reg14,reg15 *register file* blok ex_data
output	izlaz	*execute* blok READ <destination>

Slika 29-6 Opis interfejs signala bloka *execute*

Postupak rada

1. U programu Active-HDL 3.6 otvoriti direktorijum *ime_broj indeksa*.
2. Dva puta kliknuti na *Add New Files* a zatim na *Add Existing File*. U direktorijumu C:\VEZBE\PMIC\ selektovati fajl *execute_ent.vhd* i kliknuti na *Add*. Nakon toga fajl }e biti prikazan u *Design Browser* prozoru pod istim imenom.
3. Dvostrukim klikom mi{a na fajl pogledati njegov izvorni kôd i proveriti da li se sla`e sa kôdom na Slici 29-7.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.std_logic_arith.ALL;
USE WORK.pipeline_package.ALL;

ENTITY execute_ent IS
PORT  (
   d_command : IN command_type;
   d_source1_data : IN std_logic_vector (31 downto 0);
d_source2_data : IN std_logic_vector (31 downto 0);
d_destination : IN register_type;
d_source1 : IN register_type;
   d_source2 : IN register_type;
d_data : IN std_logic_vector (31 downto 0);
clock : IN std_logic;
flush : OUT std_logic;
jump : OUT std_logic;
   ex_data : OUT std_logic_vector (31 downto 0);
ex_destination : OUT register_type;
ex_store : OUT std_logic;
   output : OUT std_logic_vector (31 downto 0)
   );
END execute_ent;

ARCHITECTURE execute_arch OF execute_ent IS

SIGNAL int_ex_destination : register_type := reg0;
SIGNAL int_ex_data : std_logic_vector (31 downto 0);
```

```
SIGNAL int_d_source1_data : std_logic_vector (31 downto 0);
SIGNAL int_d_source2_data : std_logic_vector (31 downto 0);
BEGIN
PROCESS (d_command, int_ex_destination, d_source1, d_source2,
int_d_source1_data, int_d_source2_data, d_source1_data,
d_source2_data, int_ex_data)
BEGIN
   IF (d_command = LOAD OR d_command = MOVE OR d_command = NOP)
THEN
      int_d_source1_data <= d_source1_data;
      int_d_source2_data <= d_source2_data;
   ELSE
      IF (int_ex_destination = d_source1) THEN
         int_d_source1_data <= int_ex_data;
         int_d_source2_data <= d_source2_data;
      ELSEIF (int_ex_destination = d_source2) THEN
         int_d_source2_data <= int_ex_data;
         int_d_source1_data <= d_source1_data;
      ELSE
         int_d_source1_data <= d_source1_data;
         int_d_source2_data <= d_source2_data;
      END IF;
   END IF;
END PROCESS;
PROCESS (clock, d_command, int_d_source1_data,
int_d_source2_data, d_destination, d_source1, d_source2,
int_ex_data, d_data)
BEGIN
   IF (clock = '1' AND clock'EVENT) THEN
      CASE d_command IS
         WHEN MOVE =>
            int_ex_data <= int_d_source1_data;
            int_ex_destination <= d_destination;
            ex_store <= '1';
            jump <= '0';
            output <= ZERO;
            flush <= '0';
         WHEN ADD =>
            -- both MSB of src1 and src2 cannot
            -- be '1', if it is then overflow:
            int_ex_data <= signed(int_d_source1_data) +
signed(int_d_source2_data);
            int_ex_destination <= d_destination;
            ex_store <= '1';
            jump <= '0';
            flush <= '0';
            output <= ZERO;
         WHEN SUB =>
```

```
            int_ex_data <= signed(int_d_source1_data) -
signed(int_d_source2_data);
            int_ex_destination <= d_destination;
            ex_store <= '1';
            jump <= '0';
            flush <= '0';
            output <= ZERO;
         WHEN MUL =>
            -- both src1 and src2 must be max
            -- 16 bits long
            -- if not will overflow
            -- excess of 16 bits is truncated
            int_ex_data <= signed(int_d_source1_data (15
       downto 0)) * signed(int_d_source2_data (15 downto 0));
            int_ex_destination <= d_destination;
            ex_store <= '1';
            jump <= '0';
            flush <= '0';
            output <= ZERO;
         WHEN CJE =>
            IF (int_d_source1_data = int_d_source2_data) THEN
                  jump <= '1';
                  flush <= '1';
            ELSE
                  jump <= '0';
                  flush <= '0';
            END IF;
               ex_store <= '0';
               output <= ZERO;
            WHEN LOAD =>
               int_ex_destination <= d_destination;
               int_ex_data <= d_data;
               ex_store <= '1';
               output <= ZERO;
               jump <= '0';
               flush <= '0';
            WHEN READ =>
               output <= int_d_source1_data;
               ex_store <= '0';
               flush <= '0';
               jump <= '0';
            WHEN NOP =>
               output <= ZERO;
               flush <= '0';
               jump <= '0';
               ex_store <= '0';
            WHEN OTHERS =>
               NULL;
         END CASE;
```

```
      END IF;
   END PROCESS;
   ex_data <= int_ex_data;
   ex_destination <= int_ex_destination;
END execute_arch;
```

Slika 29-7 VHDL kôd bloka ***execute***

7. Desnim tasterom mi{a kliknuti na fajl ***execute_ent.vhd*** u ***Design Browser*** prozoru, a zatim ga kompajlirati pritiskom na ***Compile,*** u padaju}em meniju. Ukoliko je kompilacija izvr{ena korektno ispred imena fajla ***execute_ent.vhd*** u ***Design Browser*** prozoru pojavi}e se oznaka √. Ukoliko je do{lo do gre{ke u kompilaciji ispred imena fajla pojavi}e se oznaka X.
8. Dva puta kliknuti na ***Add New Files***, a zatim na ***Add Existing File***. U direktorijumu C:\VEZBE\PMIC\ selektovati fajl ***execute_testbench.vhd*** a zatm kliknuti na ***Add***. Nakon toga }e se selektovani fajl pojaviti u ***Design Browser*** prozoru.
9. Dvostrukim klikom na fajl ***execute_testbench.vhd*** pogledati njegov izvorni kôd u prozoru editora. Na Slici 29-8 dat je ***testbench*** program koji simulira rad ***execute*** bloka pomo}u slede}e ~etiri instrukcije

```
Load #51, reg0
Load #1, reg1
Add reg0, reg1, reg2
Sub reg0, reg1, reg3
```

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE WORK.pipeline_package.ALL;
```

```
ENTITY execute_tb_ent IS
END execute_tb_ent;
```

```
ARCHITECTURE execute_tb_arch OF execute_tb_ent IS
COMPONENT execute_ent
PORT  (
   d_command : IN command_type;
d_source1_data : IN std_logic_vector (31 downto 0);
d_source2_data : IN std_logic_vector (31 downto 0);
d_destination : IN register_type;
d_source1 : IN register_type;
   d_source2 : IN register_type;
d_data : IN std_logic_vector (31 downto 0);
clock : IN std_logic;
flush : OUT std_logic;
jump : OUT std_logic;
ex_data : OUT std_logic_vector (31 downto 0);
ex_destination : OUT register_type;
ex_store : OUT std_logic;
   output : OUT std_logic_vector (31 downto 0));
END COMPONENT;
```

```
SIGNAL ex_data : std_logic_vector (31 downto 0);
SIGNAL ex_destination : register_type;
SIGNAL ex_store : std_logic;
SIGNAL d_command : command_type;
SIGNAL d_source1_data : std_logic_vector (31 downto 0);
SIGNAL d_source2_data : std_logic_vector (31 downto 0);
SIGNAL d_destination : register_type;
SIGNAL d_source1 : register_type;
SIGNAL d_source2 : register_type;
SIGNAL clock : std_logic := '0';
SIGNAL flush : std_logic;
SIGNAL jump : std_logic;
SIGNAL output : std_logic_vector (31 downto 0);
SIGNAL d_data : std_logic_vector (31 downto 0);
CONSTANT CYCLE : TIME := 50 ns;
BEGIN
DUT: execute_ent port map (d_command, d_source1_data,
d_source2_data, d_destination, d_source1, d_source2, d_data,
clock, flush, jump, ex_data, ex_destination, ex_store, output)
clock <= NOT clock AFTER CYCLE/2;
PROCESS
BEGIN
wait for CYCLE;
-- load #51, reg0
d_command <= LOAD;
d_destination <= reg0;
d_data <= "00000000000000000000000001010001";
wait for CYCLE;
-- load #1, reg1
d_command <= LOAD;
d_destination <= reg0;
d_data <= "00000000000000000000000000000001";
wait for CYCLE;
-- add reg0, reg1, reg2
d_command <= ADD;
d_source1 <= reg0;
d_source2 <= reg1;
d_destination <= reg2;
d_source1_data <= "00000000000000000000000001010001";
d_source2_data <= "00000000000000000000000000000001";
wait for CYCLE;
-- sub reg0, reg1, reg3
d_command <= SUB;
d_source1 <= reg0;
d_source2 <= reg1;
d_destination <= reg3;
d_source1_data <= "00000000000000000000000001010001";
```

```
d_source2_data <= "00000000000000000000000000000001";
wait for CYCLE;
END PROCESS;
END execute_tb_arch;
CONFIGURATON execute_tb_config OF execute_tb_ent IS
   FOR execute_tb_arch
      FOR ALL : execute_ent
         USE ENTITY WORK.execute_ent (execute_arch);
      END FOR;
   END FOR;
END execute_tb_config;
```

Slika 29-8 *Testbench* program za *execute* blok

15. Modifikovati kôd *testbench* fajla tako da *testbench* stimuli{e *execute* blok slede}im instrukcijama:

Grupa 1:
```
Load #183FFA20, reg2
Load #0210AA18, reg4
Add reg2, reg4, reg8
Read reg8
Nop
```

Grupa 2:
```
Load #183FFA20, reg5
Load #0210AA18, reg3
Sub reg3, reg5, reg9
Read reg9
Move reg3, reg5
Read reg5
```

Grupa 3:
```
Load #99999999, reg0
Nop
Load #11111111, reg1
Mul reg0, reg1, reg2
Read reg0
Read reg2
```

Grupa 4:
```
Load #00000011, reg2
Move reg2, reg5
Load #00000001, reg3
Read reg5
Sub reg3, reg5, reg1
Read reg1
```

Grupa 5:
```
Load #AAAABBBB, reg0
Load #BBBBAAAA, reg1
Mul reg0, reg1, reg2
```

Grupa 11:
```
_____________________
_____________________
_____________________
_____________________
_____________________
```

Grupa 12:
```
_____________________
_____________________
_____________________
_____________________
_____________________
_____________________
```

Grupa 13:
```
_____________________
_____________________
_____________________
_____________________
_____________________
_____________________
```

Grupa 14:
```
_____________________
_____________________
_____________________
_____________________
_____________________
_____________________
```

Grupa 15:
```
_____________________
_____________________
_____________________
```

```
Nop                                  _____________________
Read reg1                            _____________________
Read reg2                            _____________________
```

Grupa 6:

```
Load #00001111, reg0                 _____________________
Read reg0                            _____________________
Load #1528FCDA, reg10                _____________________
Load #F8910004, reg11                _____________________
Add reg10, reg11, reg0               _____________________
```

Grupa 7:

```
Load #00000004, reg9                 _____________________
Load #00001089, reg13                _____________________
Sub reg9, reg13, reg1                _____________________
Move reg9, reg13                     _____________________
Read reg1                            _____________________
Read reg13                           _____________________
```

Grupa 8:

```
Load #98765432, reg0                 _____________________
Read reg0                            _____________________
Load #12345678, reg1                 _____________________
Read reg1                            _____________________
Sub reg1, reg0, reg3                 _____________________
Read reg3                            _____________________
```

Grupa 9:

```
Load #87AC9310, reg8                 _____________________
Load #88341502, reg1                 _____________________
Add reg1, reg8, reg2                 _____________________
Sub reg1, reg8, reg0                 _____________________
Read reg0                            _____________________
Read reg2                            _____________________
```

Grupa 10:

```
Load #148CF9A7, reg3                 _____________________
Nop                                  _____________________
Mul reg3, reg3, reg3                 _____________________
Load #71835942, reg2                 _____________________
Add reg2, reg3, reg2                 _____________________
Read reg2                            _____________________
```

Grupa 16:

```
                                     _____________________
                                     _____________________
                                     _____________________
                                     _____________________
                                     _____________________
```

Grupa 17:

```
                                     _____________________
                                     _____________________
                                     _____________________
                                     _____________________
                                     _____________________
                                     _____________________
```

Grupa 18:

```
                                     _____________________
                                     _____________________
                                     _____________________
                                     _____________________
                                     _____________________
                                     _____________________
```

Grupa 19:

```
                                     _____________________
                                     _____________________
                                     _____________________
                                     _____________________
                                     _____________________
                                     _____________________
```

Grupa 20:

```
                                     _____________________
                                     _____________________
                                     _____________________
                                     _____________________
                                     _____________________
                                     _____________________
```

16. Snimiti modifikovani fajl (*File → Save*), a zatim ga kompajlirati pritiskom na F11 (ili *Compile*).

17. Ukoliko je kompilacija izvr{ena korektno ispred fajla *execute_testbench.vhd* u *Design Browser* prozoru pojavi}e se oznaka √. Ukoliko je do{lo do gre{ke u kompilaciji ispred imena fajla pojavi}e se oznaka X, a u prozoru editora bi}e podvu~ene linije kôda koje nisu mogle biti kompajlirane. Postavljanjem kursora mi{a na podvu~enu liniju pojavljuje se prozor sa izve{tajem o gre{ci.

Nakon ispravljanja gre{aka faj treba ponovo kompajlirati pritiskom na F11.

18. Nakon uspe{ne simulacije kliknuti na znak + ispred fajla ***execute_testbench*** a zatim desnim tasterom mi{a na ***execute_tb_ent*** (***execute_tb_arch***). U padaju}em meniju izabrati opciju ***Set as Top_Level,*** kliknuti desnim tasterom mi{a i izabrati ***Simulation→ Initialize Simulation***.

19. Selektovanjem opcija ***File → New→ Waveform*** otvoriti ***waveform*** editor.

20. Iz ***Structure*** kartice ***Design Browser***-a levim tasterom mi{a prevu}i ***DUT: execute_ent*** u prozor ***Waveform*** editor-a.

21. Pokrenuti simulaciju selektovanjem opcija ***Simulation→ Run Until...*** pri ~emu u polju za vreme trajanja simulacije treba upisati 500 ns.

22. Analizitati talasne oblike dobijene simulacijom i utvrditi da li ***execute*** blok funkcioni{e ispravno.

LABORATORIJSKA VEŽBA 30

Mikrokontroler

Uvod

U ovoj ve`bi analizira}emo rad proto~nog mikrokontrolera sastavljenog od blokova predecode, decode, register file i execute, kao na Slici 27-6 (Ve`ba broj 27). Ve`ba je organizovana na slede}i na~in:

1. Analizirati strukturni VHDL kôd, prikazan na Slici 30-1, kojim se opisuje rad proto~nog mikrokontrolera.
2. Analizirati ***testbench*** program, prikazan na Slici 30-2, kojim se verifikuje rad proto~nog mikrokontrolera.

Postupak rada

1. U programu Active-HDL 3.6 otvoriti direktorijum ***ime_broj indeksa***.
2. Dva puta kliknuti na ***Add New Files*** a zatim na ***Add Existing File***. U direktorijumu C:\VEZBE\PMIC\ selektovati fajl ***top.vhd*** i kliknuti na ***Add***. Nakon toga fajl }e biti prikazan u ***Design Browser*** prozoru pod istim imenom.
3. Dvostrukim klikom mi{a na fajl pogledati njegov izvorni kôd i proveriti da li se sla`e sa kôdom na Slici 30-1.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE WORK.pipeline_package.ALL;
ENTITY microc_ent IS
   PORT (
      clock : IN std_logic;
   inst : IN std_logic_vector (2 downto 0);
   source1 : IN std_logic_vector (3 downto 0);
      source2 : IN std_logic_vector (3 downto 0);
      destination : IN std_logic_vector (3 downto 0);
      data : IN std_logic_vector (31 downto 0);
      jump : OUT std_logic;
      output : OUT std_logic_vector (31 downto 0)
      );
END microc_ent;
ARCHITECTURE microc_arch OF microc_ent IS
COMPONENT execute_ent
PORT (
d_command : IN command_type;
   d_source1_data : IN std_logic_vector (31 downto 0);
   d_source2_data : IN std_logic_vector (31 downto 0);
   d_destination : IN register_type;
   d_source1 : IN register_type;
   d_source2 : IN register_type;
```

```
   d_data : IN std_logic_vector (31 downto 0);
   clock : IN std_logic;
   flush : OUT std_logic;
   jump : OUT std_logic;
   ex_data : OUT std_logic_vector (31 downto 0);
   ex_destination : OUT register_type;
   ex_store : OUT std_logic;
   output : OUT std_logic_vector (31 downto 0)
   );
END COMPONENT;
COMPONENT register_file_ent
PORT  (
pd_read : IN std_logic;
pd_source1 : IN register_type;
pd_source2 : IN register_type;
   clock : IN std_logic;
   flush : IN std_logic;
   ex_store : IN std_logic;
   ex_destination : IN register_type;
   ex_data : IN std_logic_vector (31 downto 0);
source1_data : OUT std_logic_vector (31 downto 0);
   source2_data : OUT std_logic_vector (31 downto 0)
   );
END COMPONENT;
COMPONENT decode_ent
PORT  (
   clock : IN std_logic;
   command : IN command_type;
pd_source1 : IN register_type;
   pd_source2 : IN register_type;
   pd_destination : IN register_type;
   pd_data : IN std_logic_vector (31 downto 0);
   flush : IN std_logic;
   d_command : OUT command_type;
d_destination : OUT register_type;
d_source1 : OUT register_type;
   d_source2 : OUT register_type;
d_data : OUT std_logic_vector (31 downto 0)
   );
END COMPONENT;
COMPONENT predecode_ent
PORT  (
   clock : IN std_logic;
inst : IN std_logic_vector (2 downto 0);
source1 : IN std_logic_vector (3 downto 0);
   source2 : IN std_logic_vector (3 downto 0);
   destination : IN std_logic_vector (3 downto 0);
   data : IN std_logic_vector (31 downto 0);
   flush : IN std_logic;
```

```
   command : OUT command_type;
   pd_source1 : OUT register_type;
   pd_source2 : OUT register_type;
   pd_destination : OUT register_type;
   pd_data : OUT std_logic_vector (31 downto 0);
   pd_read : OUT std_logic
   );
END COMPONENT;
SIGNAL sig_clock : std_logic;
SIGNAL sig_inst : std_logic_vector (2 downto 0);
SIGNAL sig_source1 : std_logic_vector (3 downto 0);
SIGNAL sig_source2 : std_logic_vector (3 downto 0);
SIGNAL sig_destination : std_logic_vector (3 downto 0);
SIGNAL sig_data : std_logic_vector (31 downto 0);
SIGNAL sig_jump : std_logic;
SIGNAL sig_output : std_logic_vector (31 downto 0);
SIGNAL sig_flush : std_logic;
SIGNAL sig_command : command_type;
SIGNAL sig_pd_source1 : register_type;
SIGNAL sig_pd_source2 : register_type;
SIGNAL sig_pd_destination : register_type;
SIGNAL sig_pd_data : std_logic_vector (31 downto 0);
SIGNAL sig_ex_data : std_logic_vector (31 downto 0);
SIGNAL sig_ex_destination : register_type;
SIGNAL sig_ex_store : std_logic;
SIGNAL sig_d_command : command_type;
SIGNAL sig_d_source1_data : std_logic_vector (31 downto 0);
SIGNAL sig_d_source2_data : std_logic_vector (31 downto 0);
SIGNAL sig_d_destination : register_type;
SIGNAL sig_d_source1 : register_type;
SIGNAL sig_d_source2 : register_type;
SIGNAL sig_pd_read : std_logic;
SIGNAL sig_d_data : std_logic_vector (31 downto 0);
BEGIN
DUT_predecode: predecode_ent PORT MAP (
                        clock => sig_clock,
                        inst => sig_inst,
                        source1 => sig_source1,
                        source2 => sig_source2,
                        destination => sig_destination,
data => sig_data,
flush => sig_flush,
command => sig_command,
pd_source1 => sig_pd_source1,
pd_source2 => sig_pd_source2, pd_destination =>
sig_pd_destination,
pd_data => sig_pd_data,
pd_read => sig_pd_read);
DUT_decode: decode_ent PORT MAP (
```

```
                clock => sig_clock,
                command => sig_command,
pd_source1 => sig_pd_source1,
pd_source2 => sig_pd_source2,
                pd_destination => sig_pd_destination,
                pd_data => sig_pd_data,
                flush => sig_flush,
                d_command => sig_d_command,
                d_destination => sig_d_destination,
d_source1 => sig_d_source1,
d_source2 => sig_d_source2,
                d_data => sig_d_data);
DUT_register_file: register_file_ent PORT MAP (
                pd_read => sig_pd_read,
pd_source1 => sig_pd_source1,
pd_source2 => sig_pd_source2,
clock => sig_clock,
                flush => sig_flush,
                ex_store => sig_ex_store,
ex_destination => sig_ex_destination,
                ex_data => sig_ex_data,
                source1_data => sig_d_source1_data,
                source2_data => sig_d_source2_data);
DUT_execute: execute_ent PORT MAP (
                d_command => sig_d_command,
                d_source1_data => sig_d_source1_data,
                d_source2_data => sig_d_source2_data);
                d_destination => sig_d_destination,
d_source1 => sig_d_source1,
d_source2 => sig_d_source2,
                d_data => sig_d_data,
                clock => sig_clock,
                flush => sig_flush,
                jump => sig_jump,
                ex_data => sig_ex_data,
                ex_destination => sig_ex_destination,
                ex_store => sig_ex_store,
                output => sig_output);
clock => sig_clock,
inst => sig_inst,
source1 => sig_source1,
source2 => sig_source2,
destination => sig_destination,
data => sig_data,
                jump <= sig_jump,
output <= sig_output);
END microc_arch;
CONFIGURATION microc_config OF microc_ent IS
FOR microc_arch
```

```
   FOR ALL: predecode_ent
      USE ENTITY WORK.predecode_ent(predecode_arch);
   END FOR;
FOR ALL: decode_ent
      USE ENTITY WORK.decode_ent(decode_arch);
   END FOR;
FOR ALL: register_file_ent
      USE ENTITY WORK.register_file_ent(register_file_arch);
   END FOR;
FOR ALL: execute_ent
      USE ENTITY WORK.execute_ent(execute_arch);
   END FOR;
END FOR;
END microc_config;
```

Slika 30-1 VHDL kôd za mikrokontroler

4. Desnim tasterom mi{a kliknuti na fajl ***top.vhd*** u ***Design Browser*** prozoru, a zatim ga kompajlirati pritiskom na ***Compile,*** u padaju}em meniju. Ukoliko je kompilacija izvr{ena korektno ispred imena fajla ***top.vhd*** u ***Design Browser*** prozoru pojavi}e se oznaka √. Ukoliko je do{lo do gre{ke u kompilaciji ispred imena fajla pojavi}e se oznaka X.
5. Dva puta kliknuti na ***Add New Files***, a zatim na ***Add Existing File***. U direktorijumu C:\VEZBE\PMIC\ selektovati fajl ***top_testbench.vhd*** a zatm kliknuti na ***Add***. Nakon toga }e se selektovani fajl pojaviti u ***Design Browser*** prozoru.
6. Dvostrukim klikom na fajl ***top_testbench.vhd*** pogledati njegov izvorni kôd u prozoru editora. Na Slici 30-2 dat je ***testbench*** program koji simulira rad ***mikrokontrolera*** pomo}u slede}e sekvence instrukcija

```
(1)                    LOAD 51, reg0
(2)                    LOAD 2, reg1
(3)                    ADD reg0, reg1, reg2
(4)                    SUB reg0, reg1, reg2
(5)                    MUL reg0, reg1, reg4
(6)                    MOVE reg1, reg5
(7)                    MOVE reg1, reg6
(8)                    CJE reg0, reg5
(9)                    CJE reg5, reg6
(10)                   CJE reg5, reg6
(11)                   CJE reg5, reg6
(12)                   READ reg0
(13)                   READ reg1
(14)                   READ reg2
(15)                   READ reg3
(16)                   READ reg4
(17)                   READ reg5
(18)                   READ reg6
```

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
ENTITY microc_tb_ent IS
   PORT (
      clock : IN std_logic;
   inst : IN std_logic_vector (2 downto 0);
   source1 : IN std_logic_vector (3 downto 0);
      source2 : IN std_logic_vector (3 downto 0);
      destination : IN std_logic_vector (3 downto 0);
      data : IN std_logic_vector (31 downto 0);
      jump : OUT std_logic;
      output : OUT std_logic_vector (31 downto 0)
      );
END microc_tb_ent;
ARCHITECTURE microc_tb_arch OF microc_tb_ent IS
COMPONENT microc_ent
PORT (
clock : IN std_logic;
inst : IN std_logic_vector (2 downto 0);
source1 : IN std_logic_vector (3 downto 0);
source2 : IN std_logic_vector (3 downto 0);
destination : IN std_logic_vector (3 downto 0);
   data : IN std_logic_vector (31 downto 0);
   jump : OUT std_logic;
   output : OUT std_logic_vector (31 downto 0)
   );
END COMPONENT;
SIGNAL sig_clock : std_logic := '0';
SIGNAL sig_inst : std_logic_vector (2 downto 0);
SIGNAL sig_source1 : std_logic_vector (3 downto 0);
SIGNAL sig_source2 : std_logic_vector (3 downto 0);
SIGNAL sig_destination : std_logic_vector (3 downto 0);
SIGNAL sig_data : std_logic_vector (31 downto 0);
SIGNAL sig_jump : std_logic;
SIGNAL sig_output : std_logic_vector (31 downto 0);
CONSTANT CYCLE : TIME := 50 ns;
CONSTANT ZERO : std_logic_vector (31 downto 0) :=
'00000000000000000000000000000000';
BEGIN
sig_clock <= NOT sig_clock AFTER CYCLE/2;
DUT_microc: microc_ent PORT MAP (
clock => sig_clock,
inst => sig_inst,
source1 => sig_source1,
source2 => sig_source2,
destination => sig_destination,
      data => sig_data,
jump => sig_jump,
```

```
output => sig_output);

PROCESS
BEGIN
        -- load #51, reg0
        sig_inst <= '101';
        sig_source1 <= '0000';
        sig_source2 <= '0000';
        sig_destination <= '0000';
        sig_data <= "00000000000000000000000001010001";
        wait for 2*CYCLE;
        -- load #02, reg1
        sig_inst <= '101';
        sig_source1 <= '0000';
        sig_source2 <= '0000';
        sig_destination <= '0001';
        sig_data <= "00000000000000000000000000000010";
        wait for CYCLE;
        -- add reg0, reg1, reg2
        sig_inst <= '001';
        sig_source1 <= '0000';
        sig_source2 <= '0001';
        sig_destination <= '0010';
        sig_data <= ZERO;
        wait for CYCLE;
        -- sub reg0, reg1, reg3
        sig_inst <= '010';
        sig_source1 <= '0000';
        sig_source2 <= '0001';
        sig_destination <= '0011';
        sig_data <= ZERO;
        wait for CYCLE;
        -- mul reg0, reg1, reg4
        sig_inst <= '011';
        sig_source1 <= '0000';
        sig_source2 <= '0001';
        sig_destination <= '0100';
        sig_data <= ZERO;
        wait for CYCLE;
        -- mov reg1, reg5
        sig_inst <= '000';
        sig_source1 <= '0001';
        sig_source2 <= '0000';
        sig_destination <= '0101';
        sig_data <= ZERO;
        wait for CYCLE;
        -- mov reg1, reg6
```

```
sig_inst <= '000';
sig_source1 <= '0001';
sig_source2 <= '0000';
sig_destination <= '0110';
sig_data <= ZERO;
wait for CYCLE;
-- cmp reg0, reg5
-- compare and not equal, jump
sig_inst <= '100';
sig_source1 <= '0101';
sig_source2 <= '0110';
sig_destination <= '0000';
sig_data <= ZERO;
wait for CYCLE;
-- penalty 2 clocks
wait for CYCLE;
wait for CYCLE;
-- read reg0
sig_inst <= '110';
sig_source1 <= '0000';
sig_source2 <= '0000';
sig_destination <= '0000';
sig_data <= ZERO;
wait for CYCLE;
-- read reg1
sig_inst <= '110';
sig_source1 <= '0001';
sig_source2 <= '0000';
sig_destination <= '0000';
sig_data <= ZERO;
wait for CYCLE;
-- read reg2
sig_inst <= '110';
sig_source1 <= '0010';
sig_source2 <= '0000';
sig_destination <= '0000';
sig_data <= ZERO;
wait for CYCLE;
-- read reg3
sig_inst <= '110';
sig_source1 <= '0011';
sig_source2 <= '0000';
sig_destination <= '0000';
sig_data <= ZERO;
wait for CYCLE;
-- read reg4
sig_inst <= '110';
sig_source1 <= '0100';
```

```
        sig_source2 <= '0000';
        sig_destination <= '0000';
        sig_data <= ZERO;
        wait for CYCLE;
        -- read reg5
        sig_inst <= '110';
        sig_source1 <= '0101';
        sig_source2 <= '0000';
        sig_destination <= '0000';
        sig_data <= ZERO;
        wait for CYCLE;
        -- read reg6
        sig_inst <= '110';
        sig_source1 <= '0110';
        sig_source2 <= '0000';
        sig_destination <= '0000';
        sig_data <= ZERO;
        wait for CYCLE;
END PROCESS;
END microc_tb_arch;
CONFIGURATION microc_tb_config OF microc_tb_ent IS
FOR microc_tb_arch
          FOR ALL: microc_ent
             USE ENTITY WORK. microc_ent(microc_arch);
          END FOR;
END FOR;
END microc_tb_config;
```

Slika 30-2 *Testbench* program za mikrokontroler

Uo~iti da se instrukcije prosledjuju kroz proto~nu strukturu mikrokontrolera na na~in prikazan u Tabeli 30-1.

Tabela 30-1

Ulazni stimulusi	PREDECODE	DECODE	EXECUTE
LOAD 51,reg0	LOAD 51,reg0		
LOAD 2,reg1	LOAD 2,reg1		
ADD reg0,reg1,reg2	ADD reg0,reg1,reg2		
SUB reg0,reg1,reg3	SUB reg0,reg1,reg3	ADD reg0,reg1,reg2	
MUL reg0,reg1,reg4	MUL reg0,reg1,reg4	SUB reg0,reg1,reg3	ADD reg0,reg1,reg2
MOVE reg1,reg5	MOVE reg1,reg5	MUL reg0,reg1,reg4	SUB reg0,reg1,reg3
MOVE reg1,reg6	MOVE reg1,reg6	MOVE reg1, reg5	MUL reg0,reg1,reg4
CJE reg0,reg5	CJE reg0,reg5	MOVE reg1,reg6	MOVE reg1,reg5
CJE reg5,reg6	CJE reg5,reg6	CJE reg0,reg5	MOVE reg1, reg6
CJE reg5,reg6	CJE reg5,reg6	CJE reg5,reg6	CJE reg0, reg5
CJE reg5,reg6	CJE reg5,reg6	CJE reg5,reg6	CJE reg5, reg6
READ reg0	NOP	NOP	NOP
READ reg1	NOP	NOP	NOP
READ reg2	READ reg2	NOP	NOP

READ reg3	READ reg3	READ reg2	NOP
READ reg4	READ reg4	READ reg3	READ reg2
READ reg5	READ reg5	READ reg4	READ reg3
READ reg6	READ reg6	READ reg5	READ reg4
		READ reg6	READ reg5
			READ reg6

Ulazni stimulusi za ***READ reg0*** i ***READ reg1*** ne}e biti izvr{eni od strane mikrokontrolera. Kod instrukcije ***CJE reg5, reg6,*** registri reg5 i reg6 imaju isti sadr`aj pa se izlazni signal ***jump*** postavlja na logi~ku '1'. Izlazni signal ***flush*** }e takodje biti postavljen na logi~ku '1'. Ovo }e prouzrokovati da poni{tavanje ***predecode, decode,*** i ***execute*** blokova.

Instrukciji ***NOP*** je potrebno dva taktna intervala da od ***predecode*** bloka stigne do ***execute*** bloka. Zbog toga }e dve instrukcije (***READ reg0*** i ***READ reg1***) ostati neizvr{ene od strane mikrokontrolera.

Na slici su prikazani talasni oblici dobijeni vremenskom simulacijom. Izlazni signal sa ***output*** linije mikrokontrolera uzima vrednosti 53, 4f, a2, 2 i 2. Ove vrednosti predstavljaju sadr`aje registara reg2, reg3. reg4, reg5, i reg6.

Sa slike se vidi da je ***jump*** postavljeno na logi~ku '1' pri taktnim signalima 11 i 12. Kada se ***jump*** ponovo postavi na logi~ku '0' pri taktnom signalu 13, mikrokontroleru su potrebna jo{ tri taktna signala da pro~ita sadr`aj registra reg2 i postavi vrednost na izlaz (***output***). Zbog ovoga mora da se plati cena od dva taktna impulsa.

Ovo je razlog zbog koga se na izlaz postavljaju samo sadr`aji registara reg2, reg3, reg4, reg5 i reg6 a ne i reg0 i reg1. Prve dve instrukcije nakon grananja se zbog grananja poni{tavaju iz proto~ne strukture.

7. Modifikovati kôd ***testbench*** fajla tako da ***testbench*** stimuli{e mikrokontroler slede}im instrukcijama:

Grupa 1:

```
Load #183FFA20, reg2
Load #0210AA18, reg4
Add reg2, reg4, reg8
Read reg8
Nop
```

Grupa 2:

```
Load #183FFA20, reg5
Load #0210AA18, reg3
Sub reg3, reg5, reg9
Read reg9
Move reg3, reg5
Read reg5
```

Grupa 3:

Grupa 11:

```
_____________________
_____________________
_____________________
_____________________
_____________________
```

Grupa 12:

```
_____________________
_____________________
_____________________
_____________________
_____________________
_____________________
```

Grupa 13:

```
Load #99999999, reg0
Nop
Load #11111111, reg1
Mul reg0, reg1, reg2
Read reg0
Read reg2
```

Grupa 4:

```
Load #00000011, reg2
Move reg2, reg5
Load #00000001, reg3
Read reg5
Sub reg3, reg5, reg1
Read reg1
```

Grupa 5:

```
Load #AAAABBBB, reg0
Load #BBBBAAAA, reg1
Mul reg0, reg1, reg2
Nop
Read reg1
Read reg2
```

Grupa 6:

```
Load #00001111, reg0
Read reg0
Load #1528FCDA, reg10
Load #F8910004, reg11
Add reg10, reg11, reg0
```

Grupa 7:

```
Load #00000004, reg9
Load #00001089, reg13
Sub reg9, reg13, reg1
Move reg9, reg13
Read reg1
Read reg13
```

Grupa 8:

```
Load #98765432, reg0
Read reg0
Load #12345678, reg1
Read reg1
Sub reg1, reg0, reg3
Read reg3
```

Grupa 9:

```
Load #87AC9310, reg8
Load #88341502, reg1
Add reg1, reg8, reg2
Sub reg1, reg8, reg0
Read reg0
Read reg2
```

Grupa 14:

Grupa 15:

Grupa 16:

Grupa 17:

Grupa 18:

Grupa 19:

Grupa 10:	**Grupa 20:**
Load #148CF9A7, reg3	_____
Nop	_____
Mul reg3, reg3, reg3	_____
Load #71835942, reg2	_____
Add reg2, reg3, reg2	_____
Read reg2	_____

8. Snimiti modifikovani fajl (***File → Save***), a zatim ga kompajlirati pritiskom na F11 (ili ***Compile***).
9. Ukoliko je kompilacija izvr{ena korektno ispred fajla ***top_testbench.vhd*** u ***Design Browser*** prozoru pojavi}e se oznaka √. Ukoliko je do{lo do gre{ke u kompilaciji ispred imena fajla pojavi}e se oznaka X, a u prozoru editora bi}e podvu~ene linije kôda koje nisu mogle biti kompajlirane. Postavljanjem kursora mi{a na podvu~enu liniju pojavljuje se prozor sa izve{tajem o gre{ci. Nakon ispravljanja gre{aka faj treba ponovo kompajlirati pritiskom na F11.
10. Nakon uspe{ne simulacije kliknuti na znak + ispred fajla ***top_testbench*** a zatim desnim tasterom mi{a na ***microc_tb_ent*** (***microc_tb_arch***). U padaju}em meniju izabrati opciju ***Set as Top_Level***, kliknuti desnim tasterom mi{a i izabrati ***Simulation→ Initialize Simulation***.
11. Selektovanjem opcija ***File → New→ Waveform*** otvoriti ***waveform*** editor.
12. Iz ***Structure*** kartice ***Design Browser***-a levim tasterom mi{a prevu}i ***DUT: microc_ent*** u prozor ***Waveform*** editor-a.
13. Pokrenuti simulaciju selektovanjem opcija ***Simulation→ Run Until...*** pri ~emu u polju za vreme trajanja simulacije treba upisati 500 ns.
14. Analizitati talasne oblike dobijene simulacijom i utvrditi da li mikrokontroler funkcioni{e ispravno.

Sadržaj