SADRŽAJ:

1 UVOD

Mikrokontroleri serije MCS-96 su poslednjih godina stekli izuzetnu popularnost. Poput svog osmobitnog prethodnika (serije MCS-31), ova serija postaje industrijski standard u mnogim granama industrije na zapadu. Najveću primenu nalazi u automobilskoj industriji i za upravljanje elektromotornim pogonima. Po podacima iz 1990. godine više od deset moliona mikrokontrolera ove familije se već tada nalazilo na putevina u automobilima različitih proizvođača. Pored toga po International Defence 12/90 ova serija je odobrena za primenu u vojnoj vazduhoplovnoj industriji SAD po standardu MIL-STD 17-50.

1996. godine, ova serija je predstavljala 52.4% prodaje od celokupnog tržišta šesnaesto-bitnih mikrokontrolera. Te godine proizvedeno je i **prodato** 263.9 miliona komada ove vrsre mikro-kontrolera (svih 16-bitnih). Po očekivanom trendu porasta prodaje, 1999. taj broj bi trebalo da iznosi 705.4 miliona primeraka (podaci po IN-STAT biltenu 1996.). Trend porasta verovatno nije ostvaren zbog naglog pada cena signal-procesora koji su poslednjih godina počeli u mnogim primenama da istiskuju klasične mikrokontrolere.

Zbog svoje arhitekture, velkog broja periferija u samom mikrokontroleru, brzog množenja i deljenja, oblast moguće primene je jako široka. Velike serije u kojima se proizvodi naglo su snizile cenu tako da se približila ceni boljih osmobitnih mikrokontrolera, dok je procesorska snaga neuporedivo veća.

Ovaj priručnik sadrži neophodne detalje za programiranje u asembleru koji su uslovljeni mikroračunarom. Opisani su tipovi podataka sa kojima mikrokontroleri ove serije mogu da rade, asemblerske instrukcije, direktive, kontrole, kao i načini adresiranja i indikatora stanja programa. Neki opisi su izostavljeni ili delimično uprošćeni radi ilustrovanja suštine. Kod izrade programa, preporučuje se i konsultovanje priručnika za konkretni asembler sa kojim se radi jer neki detalji variraju u različitim implementacijama.

2 PRINCIPI RADA ASEMBLERA

Asembler je softverska alatka (program koji se izvršava na računaru tipa PC) koji prevodi program iz teksta (čoveku razumljivog načina zapisa) u niz brojeva (način zapisa koji razume mikroprocesor). Taj niz brojeva se upisuje u memoriju tipa ROM i postaje program koji izvršava mikroprocesor.

Ovde treba jasno razlikovati program koji se izvršava na računaru tipa PC od programa koji se izvršava u mikroprocesoru. PC se koristi samo kao alatka za lakše pravljenje **mašinskog koda** i teoretski gledano, nije neophodan za razvoj programa (zaista, čisto teoretski).

Pojam mašinskog koda vezan je za niz **operacionih kodova**, praćenih odgovarajućim podacima, pri čemu je operacioni kôd šifra (broj) po kome mikroprocesor razlikuje jednu instrukciju od druge. To je, jednostavno, način kojim čovek saopštava mikroprocesoru šta treba da uradi. Tako je, na primer, operacioni kôd za instrukciju sabiranja 44h i kada prilikom čitanja memorije tipa ROM mikroprocesor pronađe ovaj kôd, on će znati da treba da sabira. Iza operacionog koda mora da dođe informacija o tome šta treba sabirati. Tu informaciju koja prati operacioni kôd, najčeće čine adrese memorijskih lokacija sa kojih se uzimaju podaci za sabiranje. Operacioni kôd zajedno sa ovom informacijom čini jednu **asemblersku instrukciju**. Asemblerske instrukcije su poređane jedna za drugom u memoriji tipa ROM. Mikroprocesor jedino za prvi podatak koji je pročitao iz ROM-a posle uključenja napajanja, zna da je operacioni kôd, a ne prateća informacija. Koliko bajtova zauzima prateća informacija zna se iz pročitanog operacionog koda pa se time zna i da je sledeći podatak u ROM-u opet operacioni kôd sledeće instrukcije.

Da programer ne bi morao da pamti i razmišlja o operacionim kodovima, on instrukcije piše pomoću **mnemonika**.

Mnemonik je simbolička zamena operacionog kôda (asember će svaki mnemonik pretvoriti u operacioni kôd). Mnemonici su (ili bar treba da budu) tako izabrani da asociraju na instrukciju koju mikroprocesor treba da izvrši. Tako je mnemonik za sabiranje **ADD** (add na engleskom znači saberi), i to je simbolična zamena za 44h (mora se priznati, lakša za pamćenje). Takođe postoji zamena za adrese podataka (pomenute u prethodnom pasusu) koji učestvuju u operacijama. Ako na primer, neka memorijska lokacija, sa adresom 23h služi kao brojač prolaza kroz petlju, mnogo je lakše uvesti simboličko ime za lokaciju 23h (na primer **BrojacProlaza**) i pamtiti njega umesto pamćenja adrese lokacije (23h). Ti simboli (koji zamenjuju adrese memorijkih lokacija) zovu se **promenljive**. Pored njih postoje i **konstante**. To su simboli koji zamenjuju podatke koji se u toku izvršavanja programa ne menjaju (to najčešće nisu adrese lokacija već borjevi). Simbolima se zamenjuju i adrese pojedinih instrukcija u ROM-u koje su od značaja programeru. Tako se adresa prve instrukcije u nekoj petlji može zameniti simbolom ("**Petlja:**", na primer), kako bi programer mogao lakše za napiše instrukciju za skok na početak petlje. Takvi simboli se nazivaju **labelama**. Da bi se razlikovale od drugih simbola, labele imaju iza simbola dvotačku.

Pored simbola, u tekstu koji piše programer, mogu se naći i **brojevi**, kada programer neke konstante ne želi da zameni simbolima. Brojevi se mogu napisati u jednom od četiri raspoloživa brojčana sistema (decimalnom, heksadecimalnom, oktalnom i binarnom).

Izrazi su kombinacije simbola i brojeva povezanie dozvoljenim računskim operacijama. Asember pre konačnog prevođenja mnemonika izračuna vrednost izraza i zameni ga brojem kad je god to mohuće. Ako zamenu ne izvrši asembler, izvršiće je druga alatka, **povezivač** (linker), ali svaki izraz će na kraju postati jedan broj.

Program se sastoji od niza asemblerskih linija. U svakoj od njih može da se nađe samo po jedna instrukcija asemblera. Asemblerske linije se unose kao svaki drugi tekst (pismo, na primer) u fajl. Fajl sa asemblerskim linijama naziva se **izvorni** (*source*) **fajl**, a njegov sadržaj, **izvorni kôd**. Izvorni fajl se smešta na disk i poziva se asembler (program) koji prevodi ovaj fajl.

ASEMBLER PREVODI TEKST NAPISAN NAMA RAZUMLJIVIM JEZIKOM (UZ KORIŠĆENJE MNEMONIKA, LABELA, PROMENLJIVIH I OSTALIH SIMBOLA) U MAŠINSKI KÔD. MAŠINSKI KÔD JE NIZ BROJEVA (OPERACIONIH KÔDOVA I PRATEĆIH INFORMACIJA) KOJI SU RAZUMLJIVI MIKROPROCESORU. ON SE UPISUJE U ROM KAO PROGRAM KOJI ĆE MIKROPROCESOR IZVRŠAVATI.

2.1 SIMBOLI

Simboli su kombinacije slova (velikih i malih), cifara, i znakova: donja crta (**_**) i znak pitanja (**?**). Asembler ne razlikuje velika slova od malih (Brojac je isto što i brojac ili bRoJaC). Prvi znak u simbolu ne sme biti cifra. Dužina simbola sme biti najviše 31 znak.

Prazn prostor (*blank*) ne sme biti deo simbola. Dakle, **brojac prolaza** nije regularan simbol, ali **brojac_prolaza** ili **BrojacProlaza** jesu.

Simboli mogu biti:

- **MNEMONICI** Oni su zamena za operacione kodove.
- **DIREKTIVE** Definišu simbole i programsku strukturu. Ne zamenjuju se operacionim kodom već služe za rezervisanje adresnog prostora za promenljive, definisanje konstanti i slično. To su komande namenjene **asembleru**, ne mikroprocesoru (vidi poglavlje 3)
- **ASEMBLERSKE KONTROLE** Namenjene su kontroli rada asemblera.
- **PROMENLJIVE** One su zamena za adrese u okviru memorije tipa RAM na kojima se nalaze podaci koji učestvuju u operacijama.
- **ASEMBLERSKE KONSTANTE** To su zamene za brojeve. Asembler će, jednostavno, svaki takav simbol zameniti brojem. U okviru programa, programer mora definisati svaki takav simbol (reći asembleru koji broj zamenjuje).
- **PROGRAMSKE KONSTANTE** To su simboli koji ponovo zamenjuju adrese, ali ovaj put, adrese u okviru memorije tipa ROM u kojima se nalaze nepromenljivi podaci.
- **LABELE** One zamenjuju adrese u memoriji tipa ROM na kojima se nalazi program.

! Pre zamene mnemonika operacionim kodovima, asembler sve simbole (koje može) zameni brojevima.

Gore navedena činjenica se vrlo često zaboravlja mada izgleda sama po sebi vrlo jasna i logična. Medjutim, mnoge nejasnoće i dileme kao čarobnim štapićem nestaju kada se programer seti da sam zameni simbole brojevima kako bi to asembler uradio. To ne znači da programer treba da razmišlja o svakoj zameni, već samo o onim gde, na primer, nije jasno koji će podatak biti iskorišćen u nekom načinu adresiranja. Kada se potpuno usvoje i shvate principi asemblera, vrlo je preporučljivo, što manje razmišljati o brojevima i što više koristiti simbole.

Simboli koje asembler ne može zameniti su oni koji u vreme asembliranja nisu dostupni (na primer, definisani u drugom modulu). Tada zamenu završava povezivač (linker).

2.1.1 Predefinisani simboli

Postoje unapred definisani simboli koje programer može koristiti kao sve druge koje je sam definisao. Vrednost tih simbola određuje asembler po unapred definisanim pravilima.

Takvi simboli su $, STACK, STACKSIZE, MODEL, MEMORY…

$** je simbol za asemblerski pokazivač adresa (*location counter*). Ovaj simbol je karakterističan samo za fazu asembliranja. On za svaku liniju u asembleru, ima vrednost adrese do koje je došao asembler na početku te linije. **$ nema nikakve dodirne tačke sa programskim brojačem (*program counter*).

Na primer, instrukcija **BR $**, je beskonačna petlja. Tu simbol $ označava adresu koju ima kôd ove linije u memoriji tipa ROM.

Primer:

```
Poruka:   dcb   Duzina, 'Primer stringa po Pskal standardu'
Duzina    equ   $ - Poruka - 1
```

Ovo je primer definisanja stringa po Paskal standardu. Po njemu, prvi podatak u stringu je dužina stinga u bajtima, pa tek iza toga dolazi prvo slovo stringa. Direktiva ***dcb*** *(vidi poglavlje 3.3) definiše programske konstante tipa BAJT. Instrukcije imaju sledeći smisao: Počev od adrese* ***Poruka*** *treba u memoriji tipa ROM upisati sledeće bajte: prvo bajt jednak simbolu* ***Duzina****, pa iza toga ASCII kôd slova* ***P****, pa ASCII kôd slova* ***r****, pa slova* ***i*** *…Vrednost simbola* ***Duzina*** *još uvek nije poznata, definisana je tek u sledećem redu. Ona je jednaka trenutnoj adresi, umanjenoj za vrednost simbola* ***Poruka*** *(a to je adresa od koje počinje string) i još za jedan. Kako je trenutna adresa (na početku prevođenja tog drugog reda) jednaka adresi do koje se došlo posle upisivanja svih slova stringa, to je pomenuta razlika jednaka upravo broju slova. Tako je izbegnuto da programer prebrojava slova u stringu da bi definisao prvi bajt stringa koji mora da sadrži podatak o dužini. Ako se promeni string, asembler će sam promeniti taj prvi podatak.*

Ostali predefinisani simboli primaju vrednosti količine slobodne ili zauzete memorije, broj slobodnih ili zauzetih mesta na steku i drugo. Njihove tačne definicije se menjaju u zavisnosti od konkretne implementacije asemblera (od proizvođača programa) i potrebno je konsultovati priručnik za konkretan asembler da bi se koristile. Inače, one se koriste relativno retko u odnosu na **$**.

2.1.2 Simboli koje generiše asembler

Asembler generiše lokalne simbole makroa. Svi ovi simboli počinju sa dva znaka pitanja. Na primer, **??00001:** je simbol koji je generisao asembler kao lokalnu labelu u nekom makrou. Asembler sam vodi računa da ne ponovi simbole koje on generiše. Programer, u principu, može da koristi ove simbole, ali se to u praksi gotovo i ne događa.

2.2 BROJEVI

U toku pisanja programa u izvorni kôd se mogu unositi brojevi. Oni mogu biti decimalni, heksadecimalni, oktalni, binarni i ASCII kodovi znakova.

	CIFRE	SUFIKS
• **DECIMALNI**	**0** do **9**	bez sufiksa, **d** ili **D**
• **HEKSADECIMALNI**	**0** do **9**, **a** do **f** ili **A** do **F**	**h** ili **H**
• **OKTALNI**	**0** do **7**	**o** ili **O** ili **q** ili **Q**
• **BINARNI**	**0** i **1**	**b** ili **B**

- **ASCII KODOVI** Jedno ili više slova (karaktera) unutar jednostrukih navodnika predstavljaju niz brojeva koji odgovaraju ASCII kodovima ovih slova. Na primer, **`A`** je isto što i **41h**, **`a`** je isto što i **61h**, a **`aBc`** odgovara nizu podataka **61h**, **42h**, **63h.** Kada učestvuju u aritmetičkim operacijama, broj slova je ograničen na dva.

Heksadecimalni brojevi moraju početi cifrom. Ako počinju slovom (A do F), moraju biti zapisani sa nulom kao prvom cifrom. Ovo pravilo je usvojeno da bi se, na primer, heksadecimalni broj **ABCD**h razlikovao od simbola **ABCDh**. Zato se broj zapisuje sa vodećom nulom **0ABCD**h.

Dozvoljena je upotreba brojeva između -65535 i +65535. Izuzetak je jedino direktiva za definisanje programske konstante tipa LONG (**dcl**, poglavlje 3.3) kada je moguće definisati broj u opsegu neoznačenog LONGa 0 do 2^{32}-1. Sve aritmetičke operacije se obavljaju po modulu 65536 i svi međurezultati se otsecaju na vrednosti koje se mogu zapisati sa 16 bita.

Primeri: *Brojevi **12**, **29d**, **27o**, **3A9h**, **3a9H**, **10001101b**, **0101B** su legalno zapisani.*
*Broj **FFh** nije legalno zapisan (treba **0FFh** ili **0ffh**).*
*Broj **123456** nije u redu jer je preko 65535.*
***`0Ah`** nije isto što i **`0ah`**, a nije isto ni sa **0Ah**. Prvi primer je zapis za niz brojeva **30h**, **41h**, **68h**. Drugi, za niz **30h**, **61h**, **68h**, a treći je zapis heksadecimalnog broja 0A (decimalno 10).*
*Broj **`0`** se razlikuje od broja **0**. Prvi iznosi 30h ili 48, decimalno.*

2.3 IZRAZI

Brojevi i simboli mogu učestvovati u formiranju izraza. U izrazima se mogu pojaviti aritmetičke i ločičke računske operacije koje se mogu klasifikovati u dve grupe:

- **UNARNE**, u kojima učestvuje samo jedan podatak (kao što je promena znaka, **NEG** ili računanje prvog komplementa - **NOT**)
- **BINARNE**, u kojima učestvuju dva podatka ili dva izraza.

! Pored brojeva ili simbola, ulogu operanada mogu igrati i izrazi.

Na primer, **LDB Ax+1, #NOT 01001111b** sadrži dva operanda sa izrazima. Prvi operand je definisan preko binarne operacije sabiranja simbola **Ax** i broja **1**, a drugi preko unarne operacije

NOT i broja **01001111b**. Da ne bi bilo zabune, dok se ne usvoji tehnika korišćenja izraza na mestima operanada, najlakše je proveriti kako će izgledati izraz pošto asembler zameni simbole brojevima. Na taj način lako se može ustanoviti da je prvi operand adresa registra **Ax** uvećana za 1, a drugi operand **#10110000b**.

U slučaju instrukcije **LD Ax, Ax+2** može se zaključiti da ona sadržaj registra Ax uvećava za dva (u ragistar Ax ubacuje staru vrednost uvećanu za dva), međutim to nije tačno. Zamenom simbola brojevima (vidi poglevlje 11) instrukcija postaje **LD 30h, 32h**, što znači da su oba operanda adrese registara (registarsko direktno adresiranje, poglavlje 11.2), pa je smisao instrukcije da u registar sa adresom Ax (adresa 30h) ubaci podatak sa adrese Ax+2 (adresa 32h).

Dozvoljeni su sledeći izrazi (dati po opadajućem prioritetu):

***, /, MOD, SHR, SHL**	množenje, celobrojno deljenje, ostatak deljenja, šiftovanje udesno, šiftovanje ulevo.
+, -	sabiranje, oduzimanje (i unarni plus ili minus).
EQ, NE, GT, GE, LT, LE	relacioni operatori za poređenje simbola: jednako, različito, veće od, veće ili jednako, menje od, manje ili jednako.
=, <>, >, >= , <, <=	simbolička predstava relacionih operataora (identično prethodnom redu.
NOT	logička inverzija (bit po bit). Unarna operacija
AND	logička I operacija (bit po bit).
OR, XOR	logička ILI, odnosno eks-ILI operacija (bit po bit).

! **Računske operacije u izrazima ne obavlja mikroprocesor, već asembler (PC)**

Asembler svim izrazima (kojima može) odredi vrednost i zameni ih brojevima pre prevođenja mnemonika. Važno je shvatiti da operotor **SHL** u izrazima nema baš nikakve veze sa asemblerskom instrukcijom **SHL**, osim što je zapisan na isti načina. Asembler ih razlikuje po položaju. Na primer, **LDB Al, #1 SHL 5** ubacuje u **Al** registar broj **00100000b** (broj 00000001b pomeren ulevo za pet mesta) i potpuno je ista kao instrukcija **LDB Al, #00100000b**, u kojoj nema ni traga šiftovanju ulevo.

Izrazi koje asembler ne može zameniti brojevima su oni koji sadrže neki simbol definisan u nekom drugom fajlu (eksternali). Za takve izraze, asembler samo rezerviše mesto, a zamenu obavlja povezivač (linker).

U izrazima se mogu naći brojevi, simboli i predefinisani simboli (kao što je $). Kada u izrazu učestvuje simbol, asembler koristi vrednost simbola. Ako je to oznaka registra, vrednost simbola jednaka je adresi tog registra, ne njegovom sadržaju. Na primer, **raz equ Cx-Ax** definiše novi simbol **raz**, jednak razlici adresa **Cx** i **Ax**, a ne razlici sadržaja. Najzad, u momentu asembliranja, kada se određuje vrednost simbola, niko ne može znati koliki će biti sadržaji registara. Isto pravilo važi i za labele (labela je simbolička zamena za adresu).

! U izrazima ne postoje načini adresiranja. Svi simboli zamenjuju se njihovim vrednostima pa se ovde ne upotrebljavaju prefiksi karakteristični za načine adresiranja (#, na primer).

Normalno, kada je operand u instrukciji dat preko izraza, ispred tog izraza mogu da stoje prefiksi za načine adresiranja. Na primer, **ADD Ax, #(Bx+8)**. Ovde je (Bx+8) izraz i on, kao operand, ima prefiks # što znači neposredno adresiranje tog operanda. Ako je registar Bx na adresi 32h, ova insrukcija dodaje sadržaju Ax registra broj 32h+8 (3Ah). Zagrade nisu neophodne u ovoj sintaksi.

! Kada se definiše promenljiva (na primer **Ax**) treba jasno razlikovati vrednost **promenljive Ax** (to je sadržaj registra) i vrednost **simbola Ax** (to je broj kojim će asembler zameniti simbol Ax, a to je adresa na kojoj se nalazi ovaj registar). Vrednost simbola je bitan parametar samo u toku asembliranja. To nije element izrade algoritama.

Svi izrazi uzimaju u obzir samo šesnaestobitne brojeve. Među-rezultati u toku izračunavanja izraza se takođe otsecaju na šesnaest bita.

2.4 MODULI I POVEZIVANJE

Kod složenijih programskih projekata postaje nepraktično da ceo izvorni kôd bude u jednom fajlu. Takođe se pojavljuje potreba da se neka rešenja (naročito potprogrami) koriste više puta u potpuno različitim projektima. Zbog ovih razloga asembler, poput viših programskih jezika, onogućava modularno programiranje.

Između izvornog i mašinskog koda, uvodi se još jedan prelazni stepen, nazvan **objektnim kodom**. Tako rezultat asembliranja izvornog koda, više nije mašinski već objektni kôd, a mašinski kôd se dobija povezivanjem više objektnih kodova. Programska alatka koja obavlja povezivanje naziva se **povezivačem** ili **linkerom**.

Ako je program podeljen u više izvornih fajlova, svaki od njih čini jedan **modul**. Jasno je da moduli moraju da koriste neke iste simbole (jer su deo jednog celovitog programa). Kako se moduli asembliraju nezavisno, ti zajednički simboli, koji smeju biti definisani u samo jednom od njih, u ostalim modulima predstavljaju problem (nisu definisani). Problem je rešen uvođenjem **globalnih** simbola, to jest simbola dostupnih svim modulima koji se povezuju. Modul koji definiše zajedničke simbole, mora naznačiti ove simbole kao globalne. Ostali moduli, koji ih koriste moraju naznačiti da ti simboli nisu definisani unutar modula (inače bi asembler prijavio grešku). U njima su ti simboli označeni kao **eksterni simboli**.

Postojanje eksternih simbola u izvornom kodu dovodi do situacije da asembler ne može zameniti sve simbole brojevima, niti izračunati one izraze u kojima učestvuju ovi simboli. To mora učiniti kasnije linker, prilikom povezivanja.

Povezivanje različitih modula u jednu celinu nameće još jednu potrebu. Naime, teško je unapred odredit opseg adresa gde programer želi da bude izvorni kôd modula kad ne zna sa čim će se sve taj modul povezivati, naročito ako se ranije asembliran modul povezuje sa drugima radi nekog potpuno novog projekta. Na primer, modul koji izračunava kvadratni koren ulaznog podatka, može se razviti (napisati, asemblirati, proveriti) jednom, a koristiti u najrazličitijim primenama.

Nepraktično bi bilo postaviti zahtev da taj modul mora da zauzme, na primer, adrese od 3000h do 3082h, tim pre što za tim **apsolutnim alociranjem kôda** ne postoji nikakva objektivna potreba. Potprogram može podjednako uspešno raditi i na drugim adresama, recimo od 6000h do 6082h ili bilo gde drugde u memorijskom prostoru predviđenom za ROM. Zbog toga se većina programa alocira **relativno** što znači da asembler pravi kôd koji nema tačno određeno mesto u ROM-u već konačnu poziciju mašinskog koda u ROM-u određuje linker.

Proces određivanja mesta u adresnom prostoru koje će zauzeti izvršni kôd ili deo izvršnog koda, naziva se **alokacija**.

Zavisno od mesta gde će biti alocirani, izvornoi kôd se deli na delove koji se alociraju kao celine. Ti delovi izvornog koda se nazivaju **segmenti**.

Iako je pojam segmenta u osnovi vezan za deo memorije koij zauzima mašinski kôd, isti taj pojam se preslikava i na odgovarajuće delove izvornog koda. Dakle, **segment**, kada se radi o izvornom kodu, se odnosi na grupu komandi koje će posle prevođenja zauzeti jedan celovit deo memorijskog prostora (memorijski segment).

Osnovni segmenti su programski segment (u ovoj seriji označen sa **CSEG**) i segment podataka (označen sa **DSEG**). Serija MCS96 ima i specijalan slučaj segmenta podataka, registarski segment (označen sa **RSEG**).

Postoje delovi izvornog programa (segmenti) čiji se kôd mora nalaziti na tačno određenim adresama. To je slučaj, na primer, sa vektorima prekida koje hardver mikrprocesora očekuje u memorijskm lokacijama na unapred definisanim adresama. Za takve segmente se kaže da su **apsolutno alocirani**. Ostali, koje linker može po volji premeštati (uz izvesna ograničenja) su **relativno alocirani**.

Linker posebno povezuje registarske, programske i segmente podataka. Svi registarski segmenti svih modula koje treba povezati, povezuju se "lepljenjem" jednog za drugi s tim što se prvo rasporede apsolutno alocirani segmenti pa se relativni alociraju redom koristeći prvi slobodan prostor dovoljne veličine. Isto se ponavlja i sa programskim i sa segmentom podataka.

ULOGA LINKERA JE DA POVEŽE OBJEKTNE KODOVE NEZAVISNO ASEMBLIRANIH MODULA I DA ONE SIMBOLE KOJE ASEMBLER NIJE MOGAO, ZAMENI BROJEVIMA. TAKOĐE LINKER PRERAČUNAVA SVE IZRAZE U KOJIMA SE POJAVLJUJU EKSTERNI SIMBOLI ILI SIMBOLI ČIJU VREDNOST ASEMBLER NE MOŽE DA ZNA ZBOG RELATIVNOG ALOCIRANJA KODA.

2.5 OPŠTI OBLIK ASEMBLERSKE LINIJE

Jedna linija izvornog koda, u opštem slučaju, ima sledeći oblik:

[labela**:**] [operacija] [operand1[**,** operand2[**,** …]]] [**;** komentar] <lf> [<cr>]

U zapisu, uglaste zagrade znače polje koje je opciono (može se izostaviti iz linije), a podebljani znaci su deo sintakse asemblera. Linija se sastoji od četiri polja i završetka linije.

- Prvo polje, **[labela:]** sadrži labelu. To je simbol koji je jednak adresi u adresnom prostoru od koje počinje mašinski kôd koji je rezultat asembliranja te linije. Polje nije obavezno, a ako postoji, iza labele mora da se nađe znak **:**.

- Drugo polje, **[operacija]** sadrži operaciju koja može biti asemblerska instrukcija, asemblerska direktiva ili kontrolna instrukcija. Asemblerska instrukcija je mnemonik koji se direktno zamenjuje operacinim kôdom, a direktive i kontrolne instrukcije nemaju direktne zamene operacinim kôdom već utiču na rezervisanje memorijskog prostora ili rad asemblera. Polje nije obavezno.
- Treće polje, **[operand1[, operand2[, …]]]** sadrži jedan ili više operanada odvojenih zarezima. Broj operanada zavisi od operacije. Polje nije obavezno. Rezultat prevođenja operanada su informacije koje prate operacioni kôd u memoriji tipa ROM.
- Četvrto polje, **[; komentar]** je proizvoljan tekst i namenjeno je komentaru koji olakšava razumevanje i praćenje programa. Polje nije obavezno, a ako postoji, komentar mora početi simbolom **;** .
- Završetak linije se sastoji od dva znaka, neobaveznog znaka za povratak kursora **<cr>** - ASCII - 0Dh (*carriage return*) i obaveznog znaka za novu liniju **<lf>** - ASCII - 0Ah (*line feed*). Programer **ne vidi završetak linije**. Njega automatski unosi editor kada programer pređe u novu liniju.

Polja su međusobno razdvojena jednim ili više praznih karaktera (*blank, space*). Ulogu odvajanja polja mogu igrati i TAB karakteri (proizvoljan broj ovih karaktera).

3 ASEMBLERSKE DIREKTIVE

Pod asemblerskim direktivama se podrazumevaju komande koje se ne prevode operacionim kodovima. Njihova namena je rezervisanje memorijskog prostora, definisanje simbola i konstanti i organizacija programa. Na osnovu namene, postoji nekoliko grupa direktiva.

- **SISTEMSKE** Ove direktive definišu modul i simbole koji se razmenjuju između modula. Tu spadaju direktiva **MODULE**, **PUBLIC**, **EXTRN** i **END**.
- **Za SEGMENTACIJU** Direktive iz ove grupe definišu segmente unutar izvornog koda i pomeraju asemblerski pokazivač adresa. Tu spadaju **CSEG**, **DSEG**, **RSEG**, **SSEG**, **OSEG** i **ORG**.
- **Za DEFINISANJE PROGRAMSKIH KONSTANTI** Pomoću ovih direktiva se definišu vrednosti programskih konstanti. To su veličine koje se upisuju u ROM zajedno sa kodom programa. Tu spadaju **DCB**, **DCW**, **DCL** i **DCR**
- **Za DEFINISANJE SIMBOLA** Direktive koje definišu vrednosti simbola **EQU** i **SET**.
- **Za USLOVNO ASEMBLIRANJE** Direktive pomoću kojih je moguće pojedine delove izvornog koda asemblirati a pojedine ne, zavisno od definisnog uslova. Uslov se definiše relacionim operatorima, manje, veće i drugi (vidi poglavlje 2.3), a na osnovu vrednosti simbola. Tu spadaju **IF**, **ELSE** i **ENDIF**.

Asemblerske direktive se unose na potpuno isti način kao i asemblerske instrukcije. Imaju svoje operande i svoju sintaksu i uklapaju se u opšti oblik asemblerske linije.

U domaćoj literaturi, asemblerske direktive nazivaju se još i **pseudo-asemblerskim** instrukcijama.

Sintaksa direktiva je zapisana standardnim načinom zapisa sintakse. Po tom standardu simboli imaju sledeće značenje:

{ } Unutar vitičastih zagrada se nalazi deo sintakse koji je obavezan.
[ ] Unutar ovih zagrada je deo koji nije obavezan.
| Znak za ili.
… Znak da se prethodni deo sintakse ponavlja.
Podebljani su znaci koji su elementi sintakse (unosi ih programer).

Na primer, sintaksa {**XX** | **YY** [**: aa**]} znači da se na tom mestu može pojaviti ili XX ili YY (ali jedan mora) i ako je to YY iza njega može da sledi aa odvojeno dvotačkom.

3.1 SISTEMSKE DIREKTIVE

3.1.1 MODULE i END direktive

Sintaksa direktive MODULE je sledeća:

Ime_modula **MODULE** [Atribut]

Gde je **Ime_modula** ime koje programer želi da dâ modulu. Ime mora da poštuje opšte zahteve za sve simbole.

Atribut može biti **MAIN** ili **STACKSIZE(n)**, pri čemu je **n** očekivani maksimalni broj bajtova koje modul može zauzeti na steku (mora biti paran). Atribut je opciono polje (ne mora da postoji). Ako ga nema podrazumeva se STAKSIZE(0).

Kada se koristi, ova direktiva mora biti prva u izvornom fajlu. Ona definiše ime modula. Ime modula je od značaja jedino kod izveštaja linkera i alatki za testiranje progtama.

! U instrukciji za poziv potprograma (CALL) **ne navodi se ime modula već labela**.

Ukoliko na početku fajla nema direktiva MODULE, asembler će modulu sam dodeliti ime izvornog fajla bez ekstenzije. Dakle, ime modula čiji je izvorni kôd u fajlu **proba.asm** bi bilo **proba**, osim ako se direktivom MODULE ne traži drugačije.

Atribut služi linkeru da napravi razliku između glavnog programa (atribut MAIN) i potprograma (STACKSIZE(n)). Međutim razlika je šisto akademska, linker očekuje da jedan od povezivanih programa bude glavni i ako takvog nema upozorava programera, međutim sve funkcioniše bez ikakvih problema i ako nema modula koji je proglašen glavnim.

Podatak o očekivanom zauzeću steka se unosi da bi linker mogao da upozori programera o mogućem prepunjavanju steka. Međutim, u kontrolerskim primenama ovaj mehanizam kontrole je gotovo neupotrebljiv zbog postojanja prekida (interapata) jer se u trenutku prevođenja ne može znati redosled pozivanja potprograma.

Primeri:

Koren MODULE STACKSIZE(8)

*Ova direktiva definiše ime modula **Koren**, i kaže linkeru da je taj modul potprogram od koga se ne očekuja da zauzme više od 8 bajta na steku, kada se pozove.*

Zad_4 MODULE

Ova direktiva kaže da je modul koji se nalazi u fajlu potprogram i da želimo da se u izveštajima pojavljuje pod imenom ZAD_4 i da ne nameravamo da koristimo konrolu linkera o zauzetosti steka.

Glavni MODULE MAIN

*Modul koga smo nazvali **Glavni** je glavni program.*

Direktiva **END** je poruka asembleru da je tekst izvornog koda završen. Ona se mora naći na kraju izvornog koda. Sav tekst iza direktive END neće biti uziman u ovzir.

Dakle, END je direktiva **asembleru** i nema nikakve veze sa nekakvin krajem glavnog programa mikroprocesora. Kraj glavnog programa ne postoji već glavni program mora da sadrži beskonačnu petlju osim ako se mikroprocesor zaustavlja zbog smanjenja potrošnje, ali direktiva END sa tim nema nikakve veze.

3.1.2 PUBLIC i EXTRN direktive

Sintaksa direktive EXTRN je sledeća:

EXTRN {Simbol [**:** tip_podatka]} [**,** …]

Gde je **Simbol** ime eksternog simbola koje mora da poštuje opšte zahteve za sve simbole.

Tip_podatka može biti **BYTE** ili **WORD** ili **LONG** ili **ENTRY** ili **REAL** ili **NULL.**

U jednoj liniji se kao eksternali može deklarisati više simbola. Uz EXTRN direktivu mora da stoji bar jedan simbol (nema smisla bez ijednog simbola u spisku). Simboli su razdvojeni zarezom, a svakome od njih se može opciono definisati tip podatka koji predstavljaju. Tipovi podataka su:

BYTE, **WORD**, **LONG**	Jednobajtni, dvobajtni i četvorobajtni podaci (bilo označeni bilo neoznačeni).
ENTRY	Podatak tipa ENTRY je adresa u okviru memorite tipa ROM koja pokazuje na neku asemblersku instrukciju. To je labela na koju može da skoči program prilikom izvršavanja. Labele ispred programskih konstanti nisu tipa ENTRY.
REAL	Četvorobajtni podatak, unet u asembler samo zbog mogućnosti povezivanja sa bibliotekom za rad sa realnim brojevima. Serija MCS96 podržava samo rad sa celim brojevima.
NULL	Ovaj tip objedinjuje sve ostale. On kaže asembleru i linkeru da je taj podatak običan broj i da ne vrši nikakvu proveru. Sve asemblerske konstante mogu biti prenete kao NULL podaci. To je tip koji se podrazumeva, ako se ne navede drugi.

Ako se ne navede očekivani tip podatka, simbol se tretira kao običan broj (tipa NULL) i ne vrše se nikakve provere prilikom povezivanja i asembliranja. Provere se odnose uglavnom na parnost adresa na kojima se nalaze i na to da pojedini tipovi podataka ne smeju da se pojaveu nekim segmentima. Na primer, ENTRY ne sme da se nađe u segmentu podataka, niti BYTE, WORD,

LONG, REAL smeju da se nađu u programskon segmentu. Ako se ne definiše tip podatka, ove provere neće biti pa programer neće biti upozoren na eventualnu slučajnu grešku. Inače, ako se odrekne ove provere, programer može bez problema sve podatke prenositi kao tipa NULL.

EXTRN direktiva kazuje asembleru da su simboli koji su navedeni iza nje **eksternali**. To znači da su definisani u nekom drugom modulu i da njih asembler ne može zameniti brojem niti preračunati izraze u kojima oni učestvuju. Taj posao se za ove simbole prepušta linkeru.

EXTRN direktiva mora biti u onom segmentu u kome se nalaze ti simboli u fajlu koji ih definiše. Na primer, ako su simboli **A** i **B** definisani u okviru registarskog segmenta u jednom fajlu, direktiva **EXTRN A, B** mora biti u registarskom segmentu svih onih fajlova koji koriste ove registre. Tako će asembler prilikom asembliranja tih fajlova znati da su A i B registri.

EXTRN direktiva sa podacima tipa ENTRY može se pojaviti samo u programskom segmentu (CSEG).

! Svaki simbol iz EXTRN direktive mora se pojaviti u PUBLIC direktivi nekog od fajlova sa kojima se povezuje. To proverava linker u fazi povezivanja. Asembler uopšte ne pokušava da pronađe definiciju simbola, čim je taj simbol deklarisan kao eksternal.

Primeri:

```
Rseg
Bx:        dsw   1                          ; lokalna promenljiva
EXTRN      Ax: word, dvaPi, Cl: byte        ; eksternali
```

Ovakav izgled registarskog segmenta deklariše jednu promenljivu Bx tipa REČ koju vidi samo taj modul. Pored toga simboli Ax, dvaPi i Cl su definisani kao eksternali. Asembler neće tražiti drugu definiciju ovih simbola, ali će linker pregledati sve PUBLIC direktive svih modula sa kojima se ovaj povezuje i ako ne nađe neki od ovih simbola u njima prijaziže grešku. Pored toga, linker će proveriti još i da li je Ax deklarisan kao podatak tipa REČ (na primer, Ax: dsw 1), a Cl kao podatak tipa bajt. Deklaraciju tipa promenljive linker će tražiti u onom fajlu u kome se nalazi PBLIC direktiva sa tim simbolom. Provera tipa za simbol dvaPi se neće vršiti jer je u EXTRN direktivi naveden bez naglašavanja tipa (NULL se podrazumeva) i taj simbol može biti definisan pomoću bilo koje direktive za definisanje simbola.

```
Cseg
EXTRN      KvadratniKoren:  entry
…
call       KvadratniKoren
```

*Ova direktiva u programskom segmentu obezeštava asembler da labela **KvadratniKoren** ne postoji u fajlu već da će se naći u nekom od fajlova sa kojim će se ovaj modul povezivati. U tom fajlu, ova labela mora biti u listi simbola PUBLIC direktive. Zahvaljujući ovoj EXTRN direktivi, asembler neće prijaviti grško kod **call** instrukcije, ali će adresu na koju treba skočiti ostaviti nepopunjenu. Tu adresu će kasnije upisati linker.*

Sintaksa direktive PUBLIC je sledeća:

PUBLIC {Simbol} [**,** …]

Gde je **Simbol** ime globalnog simbola koje mora da poštuje opšte zahteve za sve simbole.

U jednoj liniji moguće je definisati više gobalnih simbola. Kada ima više od jednog, simboli su međusobno odvojeni zarezima. U listi simbola iza PUBLIC direktive mora da bude bar jedan simbol inače direktiva nema smisla.

Direktiva PUBLIC označava simbole navedene u listi simbola kao **globale**. To znači da su ti simboli dostupni svim drugim modulima (u njima se mogu pojaviti kao eksternali). Svaki simbol ozvačen kao **global** more biti deklarisan (pomoću dsb, dsw, dsl, equ, kao labela ili pomoću neke druge direktive za definisanje simbola) u istom fajlu u kome se nalazi PUBLIC direktiva.

PUBLIC direktiva može se napisati na bilo kom mestu u izvornom kodu. Lep je običaj da se piše na početku programa ili segmenta.

Svi simboli označeni kao globali se mogu (ali i ne moraju) pojaviti kao eksternali u modulima sa kojima se modul koji sadrži PUBLIC direktivu, povezuje. Označavanje nekog simbola kao global, ne obavezuje na korišćenje tog simbola u drugim modulima.

Primer:

```
PUBLIC      Ax, Cl, dvaPi, KvadratniKoren
Rseg
        Ax:     dsw     1
        B:      dsw     1
        Cl:     dsb     1
        DvaPi   equ     1608            ; 6.28*256
Cseg
        KvadratniKoren:
        …
        end
```

Ovako bi trebalo da izgleda modul sa kojim bi se mogao povezati modul sa EXTRN direktivom iz primera u prethodnom poglavlju. PUBLIC direktiva označava simbole Ax, Cl, dvaPi i KvadratniKoren, deklarisane u ovom modulu, kao globale, to jest čini ih vidljivim i drugim modulima sa kojima će se ovaj povezati. Simbol B je lokalni i drugi moduli ga ne mogu koristiti.

3.2 DIREKTIVE ZA SEGMENTACIJU

U ovu grupu spadaju direktive za definisanje segmenata i pomeranje asemblerskog pokazivača adresa. Sve direktive za definisanje segmenata se mogu objediniti zajedničkom sintaksom:

{**CSEG	DSEG	RSEG	SSEG	OSEG}** [**REL	AT** Adresa]

Operand iza direktive je neobavezan. Ako postoji to može biti ključna reč **REL** koja označava da je segment koji se započinje relativno alociran ili reč **AT** koja obezbeđuje apsolutno alociranje segmenta na adresi **Adresa**. Adresa može biti i izraz ali je nužno da svi simboli koji u njemu učestvuju budu definisani u tom fajlu i da su apsolutni (mogu se zameniti brojem).

Ako operand izostavi, podrazumeva se REL

Ovom direktivom započinje novi segment i taj segment važi sve do naredne direktive za segmentaciju.

CSEG otpočinje programski (*code*) segment. U izvornom fajlu, programski segment čine one instrukcije i direktive koje se posle prevođenja alociraju u ROM. U okviru programskog segmenta se, dakle, nalaze asemblerske instrukcije ali i programske konstante koje se upisuju u ROM.

DSEG otpočinje segment podataka (*data*). U okviru tog segmenta se nalaze direktive za rezervaciju mesta (u memoriji tipa RAM) za promenljive. DSEG se odnosi samo na one promenljive za koje se prostor rezerviše van adresnog opsega 0 do 255 gde se nalaze registri.

RSEG otpočinje registarski segment. U okviru njega nalaze se direktive za rezervaciju mesta za promenljive u opsegu adresa 0 do 255 gde se nalaze registri. Registrima se pristupa kao svakoj drugoj memorijskoj lokaciji.

SSEG otpočinje segment u okviru koga se rezerviše mesto za stek. Više detalja o ovom segmentu mogu se pronaći u priručnicima za asembler. Ovaj segment se relativno retko upotrebljava u kontrolerskim primenama.

OSEG otpočinje segment preklapanja (*overlay*). Radi uštede mesta u adresnom prostoru registara, moguće je pomoćne registre u potprogramima koji se međusobno ne pozivaju, preklopiti (dodeliti im iste adrese). Korišćenje ove tehnike preklapanja nosi rizik veoma neprijatnih grešaka i koriste je samo programeri sa puno iskustva i samo onda kad je to neophodno. Više detalja o ovom segmentu mogu se pronaći u priručnicima za asembler.

Primeri*:*

CSEG at 2080h

Tipičan početak glavnog programa. Direktiva kaže da od te linije počinje programski segment i da kôd dobijen prevođenjem linija koje slede treba apsolutno alocirati počev od adrese 2080h. Asemblerski pokazivač adresa dobija vrednost 2080h

RSEG at 1Ah
Var1: dsw 1

Direktiva kaže da od te linije počinje registarski segment i da promenljive deklarisane od te linije na dalje treba da dobijaju adrese počev od 1Ah. Varijabla tipa REČ Var1, deklarisana iza RSEG direktive će zauzeti adrese 1Ah i 1Bh. Asemblerski pokazivač adresa dobija vrednost 1Ah, a iza druge (dsw) direktive, vrednost 1Ch.

RSEG
Var2: dsb 1

Direktiva RSEG otpočinje relativno alociran registarski segment (REL operand se podrazumeva). Asembler ne može znati na kojoj će se adresi nalaziti Var2. Zbog toga, ovaj simbol neće moći da bude zamenjen brojem u fazi asembliranja. Asembler će u objektnom kodu ostaviti nepopunjena sva mesta gde treba da se pojavi adresa Var2. Tek će linker u fazi povezivanja, popuniti ta mesta.

Da bi se zadržala sličnost sa drugim asemblerima, uvedena je direktiva za pomeranje asemblerskog pokazivača adresa ORG. Njena sintaksa je:

ORG Nova_adresa

Nova_adresa može biti izraz ali samo onaj koji se može zameniti brojem već u fazi asembliranja (ne sme da sadrži eksternale ni relativne simbole).

Direktiva pomera asemblerski pokazivač adresa na Nova_adresa. Pri tom ne menja segment u kome se nalazi. Na primer, u okviru programskog segmenta, direktiva **ORG 3000h** znači da će mašinski kôd programskih instrukcija iza ove linije biti apsolutno alociran počev od adrese 3000h. Direktiva je identična direktivi CSEG at 3000h.

3.3 DIREKTIVE ZA DEFINISANJE PROGRAMSKIH KONSTANTI

Sintaksa **DCB** direktive je sledeća:

[labela:] **DCB** {izraz | string} [**,** …]

Ova direktiva definiše niz bajtova koje treba smestiti u ROM, počev od trenutne vrednosti programskog brojača adresa. **Labela** ispred direktive je neobavezna. Ako postoji dobija vrednost asemblerskog pokazivača adresa.

Svaki operand DCB direktive može biti **izraz** ili **string**. Izraz može da bude i relativan, tada vrednost u ROM upisuje linker. String je niz slova (karaktera) uokviren jednostrukim navodnicima. Operandi su odvojeni zarezima.

Direktiva DCB se sme pojaviti jedino u programskom segmentu.

Primer:

```
Cseg at 3000h
Podaci:   DCB   Duz, 'Slova', 0Ah, (28 shr 1)-1, 'A', 00001111b
          Duz   equ 5
```

Počev od adrese 3000h u ROM će biti upisan sledeći niz bajtova: 5, 53h (ASCII kôd slova S),6Ch,6Fh,76h,61h, 0Ah, 13 to jest (28/2-1), 41h (ASCII kôd slova A), 0Fh. Simbol (labela) ***Podaci*** *dobiće vrednost 3000h.*

Sintaksa **DCW** direktive je sledeća:

[labela:] **DCW** {izraz} [**,** …]

Ova direktiva definiše niz dvobajtnih podataka koje treba smestiti u ROM, počev od trenutne vrednosti programskog brojača adresa. **Labela** ispred direktive je neobavezna. Ako postoji, dobija vrednost asemblerskog pokazivača adresa.

Izraz može da bude i relativan, tada vrednost u ROM upisuje linker. Operandi su odvojeni zarezima.

Direktiva DCW se sme pojaviti jedino u programskom segmentu.

Primer:

```
Cseg at 3000h
          broj   equ  5
Prvi:     DCW  5
Drugi:    DCW  1122h, Prvi, 0, (2*broj)-4
          DCW  Drugi
```

Počev od adrese 3000h u ROM će biti upisan sledeći niz bajtova: 5, 0 (prvo niži pa viši bajt podatka 5 predstavljenog kao REČ), zatim, na adresi 3002h: 22h, 11h, 00, 30h (ova dva bajta su niži i viši bajt 3000h, jer je to vrednost simbola ***Prvi****), 00, 00 (podatak 0 predstavljen kao šesnaestobitna veličina), 06, 00 (podatak 2*5-4, kao REČ). Time se završavaju podaci iz prve dve direktive. Treća direktiva upisuje podatak 3002h (vrednost simbola* ***Drugi****) na adresu do koje je*

*stigao asemblerski pokazivač adresa, a to je 300Ah. Znači, na adresi 300Ah i 300Bh će se nalaziti bajti: 02h i 30h. Simboli **Prvi** i **Drugi** će dobiti vrednosti 3000h i 3002h.*

! Operand u DCW direktivi može biti i labela

Ukoliko takva konstrukcija izgleda zbunjujuće, treba zamisliti kako će ta direktiva izgledati kad asembler zameni simbol brojem. Broj kojim će se zameniti labela je adresa kojoj je ta labela pridružena. Tada, direktiva **DCW 3002h** više sigurno nije problematična (poslednji red u prethodnom primeru). Činjenica da i labela može biti operand DCW direktive je naročito važna kad se radi o labelama u relativno alociranim programskim segmentima. Vrlo bi nepraktično bilo da programer mora da prevede program, pa da tek iz izveštaja linkera pronađe kojoj adresi je pridružena labela od interesa i tu adresu unese nazad u izvorni program. Potreba da neka konkretna adresa bude upisana u ROM je realna, na primer kod upisa vektora prekida. Tada je ova osobina DCW direktive pravo rešenje. Na primer, vektor prekida serijskog porta koji hardver očekuje na adresi 200Ah lako se definiše na sledeći način

```
Cseg  at  200Ah
DCW   Serijski_irq
```

Gde je **Serijski_irq** labela od koje počinje rutina za opsluživanje iprekida serijskog porta. Gde će linker smestiti tu rutinu, postaje sve jedno. Njena početna adresa (koja god da je) će biti na adresi 200Ah.

Sintaksa **DCL** direktive je sledeća:

[labela:] **DCL** {Long_konstanta} [**,** …]

Ova direktiva definiše niz četvorobajtnih podataka koje treba smestiti u ROM, počev od trenutne vrednosti programskog brojača adresa. **Labela** ispred direktive je neobavezna. Ako postoji, dobija vrednost asemblerskog pokazivača adresa.

Long_konstanta je konstanta (ne može biti izraz) u opsegu neoznačenog LONG podatka (0 do 2^{32}-1) Operandi su odvojeni zarezima.

Direktiva DCL se sme pojaviti jedino u programskom segmentu.

Primer:

```
Cseg at 3500h
Longovi:    DCL   5, 11223344h, 0
```

*Počev od adrese 3500h sadržaj memorije tipa ROM će biti sledeći: 5,0,0,0, 44h,33h,22h,11h, 0,0,0,0. Simbol (labela) **Longovi** dobija vrednost 3500h, iza ove direktive, vrednost asemblerskog pokazivača adresa je 350Ch*

U setu direktiva postoji i DCR direktiva za definisanje realnih (necelih, razlomljenih) brojeva, namenjena za rad bibliotekama koje sadrže funkcije za obradu podataka u pokretnom zarezu (*floating point*). Na primer:

```
Cseg
Pi:   DCR   3.14
```

Ova direktiva će u četiri memorijske lokacije ROM-a, počev od trenutne vrednosti asemblerskog pokazivača adresa upisati broj 3.14 upisan po IEE standardu za zapis brojeva pomoću pokretnog zareza (floating point). Kako serija MCS96 ne podržava rad sa takvim brojevima, ovde neće biti razmatrani detalji ovog zapisa. Moguće ih je pronaći u uputstvu za asembler ili priručniku za biblioteku funkcija u pokretnom zarezu.

3.4 DIREKTIVE ZA DEFINISANJE SIMBOLA

Sintaksa direktiva za definisanje simbola je sledeća:

Simbol {**EQU** | **SET**} izraz [**:** tip_podatka]

Tip_podataka može biti:

BYTE, **WORD**, **LONG**	Jednobajtni, dvobajtni i četvorobajtni podaci (bilo označeni bilo neoznačeni).
ENTRY	Podatak tipa ENTRY je adresa u okviru memorite tipa ROM koja pokazuje na neku asemblersku instrukciju. To je labela na koju može da skoči program prilikom izvršavanja. Labele ispred programskih konstanti nisu tipa ENTRY.
REAL	Četvorobajtni podatak, unet u asembler samo zbog mogućnosti povezivanja sa bibliotekom za rad sa realnim brojevima. Serija MCS96 podržava samo rad sa celim brojevima.
NULL	Ovaj tip objedinjuje sve ostale. On kaže asembleru i linkeru da je taj podatak običan broj i da ne vrši nikakvu proveru. Sve asemblerske konstante mogu biti prenete kao NULL podaci. To je tip koji se podrazumeva, ako se ne navede drugi.

Tip podatka je neobavezan, ako se ne navede, podrazumeva se NULL. Navođenje tipa omogućava asembleru i linkeru proveru alokacije podataka (parnost adresa za REČ i LONG podatke) i proveru konkretnog prenošenja simbola u druge module pomoću PUBLIC-EXTRN direktiva.

ENTRY tip se sme pojaviti jedino u programskom segmentu (CSEG), ostali svuda.

Ove direktive definišu navedeni **simbol**. Simbol se definiše na taj način čto mu se dodeljuje brojčana vrednost jednaka navedenom **izrazu.** Asembler (ili linker, ako ovaj nije u stanju) će sve simbole zameniti brojevima. Izraz može biti i relativan, sadržati simbole definisne tek kasnije u fajlu, ali ne sme sadržati eksternale. Za definiciju izraza i računskih radnji dozvoljenih u izrazima, vidi poglavlje 2.3.

EQU i SET se razlikuju jedino u tome što se vrednost simbola određena sa EQU ne sme kasnije menjati, dok simboli definisani sa SET, mogu biti kasnije predefinisani.

! Obratiti pažnju na to da se ovim direktivama definišu vrednosti **simbola**, ne sadržaji registara niti programske konstante koje se upisuju u ROM. Vrednost simbola je podatak bitan jedino u fazi asembliranja. Posle toga, simbol biva zamenjen brojem.

! EQU i SET direktive **ne pomeraju asemblerski pokazivač adresa**.

Primeri:

```
Maska       equ   00010100b
Br_bit      equ   5
Set_maska   equ   1 SHL Br_bit          ; ili 00100000b
Ah          equ   Ax+1 : byte
```

*Poslednji primer definiše simbol Ah tako da mu je vrednost jednaka vrednosti simbola Ax uvećanoj za jedan. Ako je Ax REČ promenljiva, onda je vrednost simbola Ax jednaka adresi te promenljive (na primer 30h). Viši bajt ove promenljive ima adresu 31h, pa se Ah može tretitrati kao viši bajt Ax. Da bi se asembleru i linkeru stavilo do znanja da taj simbol treba da predstavlja adresu BAJT podatka, a ne običan broj (kao ostali primeri) definiciji se dodaje tip (**:byte**).*

Obratiti pažnju da nema dvotačke iza imena simbola.

```
AA          SET   51
      …
AA          SET   AA+1
```

Simbol AA je najpre definisan kao broj 51, da bi kasnije njegova vrednost bila uvećana za jedan. Korišćenje EQU direktive u ovom primeru rezultovalo bi prijavom greške od strane asemblera. SET direktiva se koristi najviše u makroima.

3.5 DIREKTIVE ZA REZERVISANJE PROSTORA

Za svaku promenljivu koju želi da koristi, programer mora da rezerviše mesto u memoriji.

Navođenje tipa promenljive i rezervisanje memorijskog prostora za nju naziva se **deklaracija promenljive.**

Za deklaraciju promenljive služe četiri direktive čija je objedinjena sintaksa sledeća:

[Simbol:] {**DSB | DSW | DSL | DSR**} aps_izraz

Gde **Simbol** predstavlja ime promenljive koja se deklariše, a **aps_izraz** je apsolutni izraz (ne sme da sadrži relativne simbole ni eksternale).

Direktiva deklariše promenljivu sa imenom Simbol. Pri tom, **DSB** služi za deklaraciju jednobajtnih promenljivih, **DSW** za dvobajtne, a **DSL** i **DSR** za četvorobajtne promenljive. Apsolutni izraz, koji je operand ove direktive, određuje za koliko podataka željenog tipa programer želi da rezerviše mesto.

Asembler vodi računa o tome da dvobajtne promenljive moraju biti na parnim adresama, a četvorobajtne na adresama deljivim sa četiri, ukoliko su u registarskom prostoru (RSEG), ili na parnim ako se deklarišu segmentu podataka (DSEG).

Prilikom prevođenja ove direktive, asembler radi tri stvari:

- Koriguje parnost adrese ako treba (dodaje neiskorišćene bajte dok ne zadovolji zahtev)
- Dodeljuje simbolu trenutnu vrednost asemblerskog brojača adresa.
- Uvećava asemblerski brojač adresa na sledeći način:

$ ← $ + N * aps_izraz

Gde je **N**=1 za DSB, **N**=2 za DSW i **N**=4 za DSL i DSR

Kako je to u sintaksi naznačeno, prvo polje (koje sadrži simbol) nije obavezno. Ukoliko ne postoji, asembler će preskočiti dodeljivanje vrednosti simbolu, i jedino će uvećati asemblerski pokazivač adresa. To rezultuje rezervacijom "slepih" mesta u memorijskom prostoru (memorijske lokacije koje nemaju pridruženu simboličku adresu). Tim lokacijama se ipak može pristupiti pomoću apsolutne adrese (što se veoma retko primenjuje) ili pomoću simbolikčke adrese neke od prethodnih ili narednih promenljivih.

Treba primetiti da se simbol koji će biti vezan za promenljivu definiše na isti način kao da je labela (nalazi se u polju rezervisanom za labele i ima dvotačku iza simbola, vidi poglavlje 2.5). Tu je iskorišćen mehanizam dodele vrednosti asemblerskog pokazivača adresa labelama (ne baš previše dosledno) samo zato da se ne bi uvodio novi mehanizam dodele vrednosti simbolima koji su imena promenljivih. Da bi se razlikovao od labele, taj simbol automatski nosi i oznaku tipa podatka koji će biti u toj memorijskoj lokaciji. Tako, **DSB** automatski vezuje simbol za tip podatka BAJT, **DSW** za REČ, **DSL** za LONG.

Direktive za rezarvaciju prostora ne smeju se pojaviti u programskom segmentu (CSEG) jer tu nemaju smisla (ne može se rezervisati prostor za promenljivu u okviru memorije tipa ROM).
Primeri:

```
$ = ?     Rseg at 30h
$ = 30h          Bajt:       DSB   1
$ = 32h          Brojac:     DSW   1
$ = 34h                      DSW   1
$ = 36h          Brojevi:    DSW   5
$ = 40h          Poslednji:  DSB   1
$ = 41h
$ = 41h   Dseg   at 1000h
$ = 1000h        Longovi:    DSL   2
$ = 1008h        Polje:      DSB   500h
$ = 1508h        Aa:         DSB   1
$ = 1509h
```

*Počev od adrese 30h, u registarskom segmentu je sledeća slika: Promenljiva **Bajt**, tipa BAJT, zauzima adresu 30h. Adresa 31h je neiskorišćena. **Brojac** zauzima 32h i 33h (morao je da bude na parnoj adresi). Neimenovana (slepa) promenljiva zauzima 34h i 35h. Može joj se pristupiti kao **Brojac+2**. Sledi rezervacija mesta za 5 promenljivih tipa REČ, koje zauzimaju adrese 36h do 3Fh. Ovim promenljivim može da se pristupi preko simboličke adrese **Brojevi**, primenom indeksnog adresiranja (na primer, **ld Ax, Brojevi[Bx]**, vidi poglavlje 11). Poslednji deklarisan registar je namenjen podacima tipa BAJT i nalazi se na adresi 40h).*

*Segment podataka počinje na adresi 1000h. Dve promenljive za podatke tipa long zauzimaju adrese 1000h do 1007h. Drugoj promenljivoj se može pristupiti kao **Longovi+4**. **Polje**, namenjeno indeksnom pristupu (na primer, **ld Al, Polje[Bx]** gde sadržaj indeksnog registra Bx određuje redni broj podataka kome se pristupa, vidi poglavlje 11), zauzima adrese 1008h do 1507h. Na kraju, jednobajtna promenljiva Aa će zauzeti adresu 1008h. Sledeća BAJT promenljiva bi zauzela 1509h, a ako bi sledeća promenljiva bila namenjena REČ podatku, ona bi zauzela 150Ah i 150Bh*

U levom gornjem uglu ispred komande data je vrednost asemblerskog pokazivača (location counter) adresa pre prevođenja te direktive.

3.6 DIREKTIVE ZA USLOVNO ASEMBLIRANJE

Ponekad je potrebno više programa za različite projekte koji imaju zajedničke delove smestiti u isti izvodni fajl, pa jednostavnom promenom vrednosti nekog simbola, odrediti delove izvornog fajla koji će se asemblirati da bi se dobio modul za željeni projekat. Tako se može dobiti, na primer, na jasnoći dokumentacije. Takva potreba se nameće i u makroima kad zavisno od prenetog simbola treba ubaciti različite delove izvornog koda. Makroi su objašnjeni u priručniku za asembler.

Pomoću direktiva za uslovno asembliranje moguće je pre prevođenja isključiti pojedine delove izvornog koda u zavisnosti od uslova.

Sintaksa je sledeća:

```
IF uslov
  …          ; instrukcije koje se prevode ako je uslov ispunjen
[ELSE]
  …          ; instrukcije koje se prevode ako uslov nije ispunjen
ENDIF
```

Gde je **uslov** apsolutni izraz (ne sme da bure relativan niti da sarži eksternale). Po pravilu u izrazu figuriše neki od relacionih operatora za poređenje izraza (poglavlje 2.3). Ukoliko to nije slučaj da li je izraz istinit (uslov ispunjen) ili ne, odlučuje se na osnovo bita najmenje pozicione vrednosti.

IF…ENDIF konstrukcija može biti jedna u drugoj do 9 nivoa.

! Uslov određuje vrednost izraza. U izrazu figurišu **simboli** nikako sadržaji registara ili drugih promenljivih. Uslov se određuje prilikom asembliranja, a ne izvršavanja programa!

4 ASEMBLERSKE KONTROLE

Asemblerske kontrole su namenjene kontroli rada asemblera kao DOS programa. One su veza operativnog sistema i asemblera kao bilo kog drugog programa. Zbog toga i imaju jednu specifičnost. Pored toga što se mogu pojaviti u izvornom kodu programa, mogu se specificirati i u DOS komandnoj liniji.

Komandna linija u DOSu izgleda:

C:\ > **asm96.exe** ime_izvornog_fajla [spisak_kontrola]

Gde je **asm96.exe** ime izvršnog programa asemblera (staza do ovog fajla je izostavljena), **ime_izvornog_fajla** je ime fajla u kome se nalazi izvorni kôd, unet pomoću nekog standardnog editora teksta (***edit, ne,*** ili za ljubitelje Windows-a, ***notepad***), a **spisak_kontrola** niz željenih kontrola.

Kada se kontrole unose u izvorni kôd, linija mora početi znakom **$**. Ovo obezbeđuje da se razlikuju od asemblerskih direktiva i asemblerskih instrukcija. Dovoljno je da prvi znak u liniji bude $, iza toga se može navesti više kontrola u istoj liniji. Takva linija u izvornom kodu naziva se **kontrolnom linijom**. U kontrolnoj liniji ne mogu se nalaziti ni asemblerske instrukcije niti direktive. Iza liste kontrola može da se pojavi komentar, odvojen simbolom tačka-zarez.

! Lista asemblerskih kontrola u izvornom kodu počinju znakom **$**.

Sve direktive se dele u dve grupe:

- **PRIMARNE** koje se u izvornom kodu moraju pojaviti pre bilo koje asemblerske instrukcije ili direktive (takođe i pre bilo koje prazne linije) i koje se ne mogu menjati unutar izvornog koda. Takve kontrole važe za ceo izvorni kod, bilo da su navedene na njegovom početku ili u DOS komandnoj liniji prilikom pozivanja asemblera.
- **OPŠTE** koje se mogu pojaviti bilo gde u izvornom kodu i koje se mogu menjati. Kada su navedene u DOS komandnoj liniji važe za ceo izvorni fajl, osim ako se ista kontrola ne pojavi i unutar izvornog koda. Kontrola unutar izvornog koda poništava kontrolu iste vrste navedenu u komandnoj liniji DOSa.

Primarne kontrole izveštaja:

ERRORPRINT [**(**ime_fajla**)**]/**NOERRORPRINT**	**EP/NOEP**	**NOEP**
PAGELENGTH (n**)**	**PL**	**PL(60)**
PAGEWIDTH (n**)**	**PW**	**PW(120)**
PRINT [**(**ime_fajla**)**]/**NOPRINT**	**PR/NOPR**	**PR(**izvorni.LST**)**
SYMBOLS/NOSYMBOLS	**SB/NOSB**	**SB**
XREF/NOXREF	**XR/NOXR**	**NOXR**

Tabela 4.1 *Kontrole izveštaja, skraćeni oblik i podrazumevana vrednost*

Prethodna tabela daje pregled kontrola izveštaja listinga. Prva kontrola uključuje (EP) ili isključuje (NOEP) poseban izveštaj u kome su izlistane **samo greške** prilikom asembliranja. Ako se kontrola ne navede, ovakav izveštaj je isljućen, a ako se navede izveštaj o greškama se smešta u fajl **ime_fajla**. Ako se izostavi ovo ime, izveštaj se štampa na ekranu.

Sledeće dve kontrole definišu broj linija na stranici (**n**) i broj slova u liniji (**n**) prirkom pravljenja izveštaja asemblera. Podrazumevane vrednosti su 60 i 120, respektivno.

PRINT kontrola definiše ime fajla u kome će biti izveštaj asemblera, ili isključuje taj izveštaj. Ako se ne nevede, podrazumeva se da će izveštaj biti u fajlu čije se ime dobija kad se imenu izvornog fajla bez ekstenzije doda ekstenzija **.LST**. Na primer, ako je ime izvornog fajla **proba.asm**, izveštaj će biti u fajlu **proba.lst**. NOPRINT ne pravi izveštaj asemblera.

Poslednje dve kontrole određuju da li će se u okviru izveštaja pojaviti tabela simbola (**SB**) i pregled korišćenja simbola (**XR**) ili ne (**NOSB**, **NOXR**). Tabela simbola sadrži imena simbola sa vrednošću i tipom, a pregled korišćenja za svaki od njih daje informaciju u kojoj liniji izvornog koda je definisan i u kojim se sve linijama koristi. Iz tabele se vidi da se podrazumeva izveštaj sa tabelom simbola, ali bez pregleda korišćenja.

Primarne kontrole objekta:

DEBUG/NODEBUG	**DB/NODB**	**NODB**
OBJECT [**(**ime_fajla**)**]/**NOOBJECT**	**OJ/NOOJ**	**OJ**(izvorni.OBJ)

Tabela 4.2 *Kontrole objekta, skraćeni oblik i podrazumevana vrednost*

Ove dve kontrole određuju da li će tabela simbola biti uključena u objektni fajl i kako će se objektni fajl zvati. Podrazumevano ime se dobija kad se imenu izvornog fajla bez ekstenzije doda ekstenzija **.OBJ**. Na primer, ako je ime izvornog fajla **proba.asm**, objektni fajl će biti **proba.obj**.

Tabela simbola se uključuje u objektni fajl (DB) kada programer želi da u toku ispitivanja programa, promenljivima pristupa preko njihovih imena, umesto preko adresa. Ovu kontrolu (DB) treba uključiti dok traje razvoj ako se koristi neka od razvojnih alatki.

Opšte kontrole:

COND/NOCOND	**CO/NOCO**	**CO**
EJECT	**EJ**	**_**
GEN/NOGEN	**GE/NOGE**	**NOGE**
INCLUDE (ime_fajla**)**	**IC**	**_**
LIST/NOLIST	**LI/NOLI**	**LI**
SAVE/RESTORE	**SA/RS**	**_**
TITLE ('string'**)**	**TT**	**TT(**ime_modula**)**

Tabela 4.3 *Opšte kontrole, skraćeni oblik i podrazumevana vrednost*

Opšte kontrole se mogu naći bilo gde u izvornom fajlu, pri čemu svaka od njih važi do navođenja suprotne kontrole. Na primer, **LIST** važi do linije u kojoj je navedena kontrola **NOLIST**, koja opet važi do sledećeg navođenja **LIST** kontrole.

COND kontrola obezbeđuje da se u izveštaju asemblera pojave sve unete linije, nezavisno od toga što su neke isključene direktivana za uslovno asembliranje (poglavlje 3.6). **NOCOND** u izveštaj uključuje samo one linije koje su asemblirane.

EJECT kontrola generiše prelaz na novu stranicu u izveštaju.

GEN kontrola prikazuje razvijene makroe (umesto linije sa pozivom makroa, pojavljuje se kompletan izvorni kôd makroa, onakav kakav će biti asembliran), kao i prave instrukcije usvojene od asemblera umesto generičkih (poglavlje 10).

INCLUDE kontrola u izvorni kôd ubacuje kompletan tekst iz fajla čije ime je navedeno u zagradama. Ubacivanje se vrši pre početka asembliranja. Dobija se isto kao da je navedeni fajl pomoću editora uvačen u fajl koji se asemblira. Broj ubačenih fajlova nije ograničen, kao ni mesto gde se fajlovi ubacuju. Ova kontrola mora biti poslednja u listi ako se navodi zajedno sa drugim u istoj liniji.

NOLIST kontrola izbacuje iz izveštaja sve linije do sledeće pojave **LIST** kontrole.

SAVE kontrola omogućava asembleru da zapamti trenutno stanje opštih kontrola, i da to stanje rekonstruiše kontrolom **RESTORE**. Koristi se kad programer u nekom delu programa želi da promeni stanje neke od kontrola, pa da potom vrati sve kontrole u prethodno stanje, nezavisno od toga kakvo je to stanje bilo.

TT kontrola određuje naslov koji će se pojaviti u zaglavlju izveštaja asemblera na svakoj stranici. Ako se ne navede, to ime će biti jednako imenu modula.

Primeri:

Ako se kao prva linija u izvornom fajlu navede kontrola: ***$ ep****, na ekranu će se u toku asembliranja pojaviti spisak svih grešaka. U protivnom, programer mora da edituje* ***.lst*** *fajl da bi video dge se nalaze greške. Isti efekat se može dobiti i ako se u komandnoj liniji DOSa, iza imena fajla navede ova kontrola (bez znaka $).*

```
…                ; linije u kojima nisu razvijani makroi
$ GEN
…                ; linije u kojima su makroi razvijeni
$ NOGEN
…                ; linije u kojima ponovo nisu razvijani makroi
```

Svi makroi koji se pojave u linijama između GEN i NOGEN direktiva će biti razvijeni u izveštaju asemblera.

$ INCLUDE (8096.inc)

Ova konrola će iza linije ubaciti tekst fajla 8096.inc pa će se, tek onda, tako dobijeni novi izvorni fajl asemblirati.

$ EP NOLIST INCLUDE (8096.inc)

$ LIST

Čest početak programa. U prvoj liniji je uključen izveštaj za geške, i ubačen fajl 8096.inc, s tim da se tekst ovog fajla ne pojavljuje u izveštaju asemblera. Druga linija omogućava da se ostatak teksta pojavi u izveštaju.

5 TIPOVI PODATAKA

Asembler za mikroprocesore familije MSC-96 podržava različite tipove podataka kao i sami mikroprocesori. Tipovi podataka biće pisani veliki slovima. Tako "BAJT" označava neoznačeni osmobitni podatak dok je "bajt" oznaka osmobitnog dela bilo kakvog podatka.

5.1 BAJT

BAJT je neoznačen osmobitni podatak i **može primiti vrednost između 0 i 255.** Sve aritmetičke i logičke operacije sa podacima ovog tipa daju rezultat kao u aritmetici po modulu 256. Na primer, sabiranje brojeva 250 i 7 dje rezultat 1 zbog toga što zapis očekivaniog rezultata (257) zahteva više od osam bita. Binarno, broj 257 bi bio zapisan kao "1 0000001" pri čemu je prva jedinica na mestu bita broj osam ($257 = 2^8 + 2^0$). Kako je podatak tipa bajt zapisan sa samo osam binarnih cifara sa težinski koeficijentima 2^0 do 2^7, to deveta binarna cifra (koja je na mestu bita broj osam) jednostavno biva otsečena. Pri tom se ne dešava ništa dramatično, naredne operacije se najnormalnije odvijaju, osim što u mikrokontroleru postoji informacija da je deveti bit otsečen. Informacija postoji samo da naredne aritmetičke operacije i programer može, ali i ne mora, da je iskoristi. **Dakle, u aritmetici sa podacima tipa bajt, <u>250+7=1</u>** i o ovome treba voditi računa kada se koriste instrukcije koje rade sa ovim tipom podataka. Pokazaće se kasnije da je pod nekim okolnostima ovo tačan rezultat.

Logičke operacije se obavljaju bit po bit. Biti su numerisani od 0 do 7 pri čemu bit ima 0 najmanji težinski koeficijen (2^0) dok je sedmi bit najznačajniji *(most significifant bit)* sa težinskim koeficijentom 2^7. Podaci tipa BAJT mogu biti smešteni i na parnim i na neparnim adresama u okviru adresnog prostora.

5.2 REČ

REČ *(WORD)* je neoznačen šesnaestobitni podatak i **može primiti vrednost između 0 i 65535.** Sve aritmetičke i logičke operacije sa podacima ovog tipa daju rezultat kao u aritmetici po modulu 65536. Na primer, 65530+8=2. Logičke operacije se obavljaju bit po bit. Bitovi su numerisani od 0 do 15 pri čemu bit 0 ima najmanji težinski koeficijent (2^0), dok je petnaesti bit najznačajniji *(most significant bit)* i ima težinski koeficijent 2^{15}. Podaci tipa REČ moraju biti na parnim adresama u okviru adresnog prostora. Pri tome, manje značajan bajt se nalazi na nižoj (parnoj) adresi, a značajniji bajt na višoj (neparnoj) adresi. Adresa podatka tipa REČ je adresa

manje značajnog bajta. Operacije nad REČima koje su smeštene na neparnim adresama mogu dati pogrešne rezultate.

Termin "REČ" je usvojen u našoj terminologiji kao rezultat doslovnog prevođenja engleske reči *"WORD"* i nema nikakve veze sa osnovnim značenjem ovog termina u srpskom jeziku.

5.3 LONG

LONG je neoznačen tridesetdvobitni podatak **i može primiti vrednost između 0 i 4.249.967.295 (2^{32}-1).** Mikroprocesori familije MSC-96 direktno podržavaju samo operacije šiftovanja, množenja i deljenja sa podacima ovog tipa. Podaci tipa LONG moraju biti na parnim adresama, a ukoliko se nalaze u okviru registarskog prostora mikroprocesora, njihova adresa mora biti deljiva sa 4. Pri tome, najmanje značajan bajt se nalazi na najnižoj adresi, a najznačajniji bajt na najvišoj adresi. LONG se adresira pomoću adrese najmanje značajnog bajta. LONG se može posmatrati kao dve REČi koje su jedna za drugom u memorijskom prostoru.

Aritmetičke i logičke operacije koje nisu direktno podržane lako je izvesti pomoću operacija nad podacima tipa REČ.

5.4 OZNAČEN BAJT

OZNAČEN BAJT *(SHORT INTEGER)* je označeni osmobitni podatak i **može primiti vrednost između -128 i 127.** Aritmetičke operacije koje daju rezultat izvan ovog opsega postavljaju odgovarajuće indikatore u status registru mikroprocesora. Za zapisivanje negativnih brojeva, koristi se drugi komplement. Sve aritmetičke i logičke operacije sa podacima ovog tipa daju isti rezultat kao da se radi o podacima tipa BAJT. Podaci tipa OZNAČEN BAJT mogu biti smešteni i na parnim i na neparnim adresama u okviru adresnog prostora.

5.5 OZNAČENA REČ

OZNAČENA REČ *(INTEGER)* je označeni šesnaestobitni podatak **i može primiti vrednost između –32768 (-2^{15}) i 32767 (2^{15}-1).** Aritmetičke operacije koje daju rezultat izvan ovog opsega postavljaju odgovarajuće indikatore u status registru mikroprocesora. Za zapisivanje negativnih brojeva, koristi se drugi komplement. Sve aritmetičke i logičke operacije sa podacima ovog tipa daju isti rezultat kao da se radi o podacima tipa REČ. Podaci tipa OZNAČENA REČ mogu biti smešteni samo na parnim adresama u okviru adresnog prostora. Za njih važe ista pravila kao i za REČ.

5.6 OZNAČEN LONG

OZNAČEN LONG je označen tridesetdvobitni podatak i **može primiti vrednost između -2.147.483.648 (-2^{31}). i 2.147.483.647 (2^{31}-1).** Za zapisivanje negativnih brojeva, koristi se drugi komplement. Mikroprocesori familije MSC-96 direktno podržavaju samo operacije šiftovanja, množenja, deljenja i normalizovanja sa podacima ovog tipa. Podaci tipa OZNAČEN LONG moraju biti na parnim adresama, a ukoliko se nalaze u okviru registarskog prostora mikroprocesora, njihova adresa mora biti deljiva sa 4. Pri tome, najmanje značajan bajt se nalazi na najnižoj adresi, a

najznačajniji bajt na najvišoj adresi. OZNAČENI LONG se adresira pomoću adrese najmanje značajnog bajta.

Aritmetičke i logičke operacije koje nisu direktno podržane lako je izvesti pomoću operacija nad podacima tipa OZNAČENA REČ.

5.7 BIT

BIT je jednobitni podatak i može primiti vrednost 0 ili 1. BIT se koristi i u Bulovoj algebri sa vrednostima tačno *(true)* i netačno *(false)*. Pored mogućnosti operacija nad pojedinačnim bitovima kao delovima BAJTa ili REČi, arhitektura MSC-96 omogućava pristup bilo kom bitu u okviru registarskog prostora mikroprocesora.

5.8 REAL

MSC-96 arhitektura ne podržava direktno rad sa razlomljenim brojevima, ali postoji biblioteka programa za rad sa aritmetikom pokretne decimalne tačke *(floating point)*. Zbog toga u asembleru postoji mogućnost definisanja konstanti tipa REAL. Zapis je tridesetdvobitni pri čemu je bit najveće težine znak, sledećih 8 bita je eksponent uvećan za 127, a ostala 23 bita mantisa. Detaljni opis formata i korišćenja tipa REAL može se naći u uputstvu koje ide uz biblioteku.

6 OZNAČENI I NEOZNAČENI PODACI

Tipovi podataka opisani u prethodnom poglavlju mogu se klasirati na jednobitne, osmobitne, šesnaestobitne i tridesetdvobitne.

Jednobitni	**BIT**		
Osmobitni	**BAJT**	**OZNAČEN BAJT**	
Šesnaestobitni	**REČ**	**OZNAČENA REČ**	
Tridesetdvobitni	**LONG**	**OZNAČEN LONG**	**REAL**

Osmobitni podaci (BAJT i OZNAČEN BAJT) zauzimaju jedan bajt u memorijskom prostoru, šesnaestobitni (REČ i OZNAČENA REČ) dva bajta, a tridesetdvobitni (LONG, OZNAČEN LONG i REAL) po četiri bajta.

Zadržimo se kod osmobitnih (ili jednobajtnih) podataka. Treba primetiti da jednobajtni podatak može biti označen (OZNAČEN BAJT) ili neoznačen (BAJT).

Nameće se jednostavno pitanje: Kako mikroprocesor razlikuje označene od neoznačenih podataka? Odgovor je još jednostavniji: **NIKAKO !!!** Naime, sve operacije nad jednobajtnim podacima se izvode na identičan način i **mikroprocesor ni po čemu ne razlikuje podatke tipa BAJT od podataka tipa OZNAČEN BAJT.**

Može se postaviti i pitanje: “Da li je moguće na neki način saopštiti mikroprocesoru da li je podatak označen ili neoznačen?” Odgovor na ovo pitanje je ponovo negativan. **U asembleru ne postoji način da se određeni podatak deklariše kao označen ili neoznačen!** Asembler razlikuje podatke jedino po veličini i u asembleru je moguće jedino razlikovati jednobajtne (ili osmobitne),

dvobajtne (ili šesnaestobitne) i četvorobajtne (ili tridesetdvobitne). Podaci tipa BIT se tretiraju kao deo osmobitnog podatka.

Ako je to već tako (mikroprocesor ne razlikuje OZNAČEN BAJT i BAJT niti je moguće deklarisati ih različto), a tako je, **čemu onda dva tipa podataka? Ko pravi razliku?**

Razliku pravi **programer** i on je taj koji je dužan da vodi računa o tome koji jednobajtni podatak **želi** da bude tipa BAJT, a koji tipa OZNAČEN BAJT. Posle svake aritmetičke operacije, mikroprocesor definiše (postavlja ili briše) indikatore stanja (vidi poglavlje 13), a programer sâm mora da **tumači** rezultat na osnovu postavljenih indikatora. Postoje indikatori koji su značajni pri tumačenju rezultata operacija nad označenim podacima, a postoje i indikatori značajni za rad sa neoznačenim. Nezavisno od toga da li su ulazni podaci označeni ili ne, **obe grupe indikatora se postavljaju/brišu,** programer je taj koji treba da odluči **koje će indikatore koristiti za tumačenje rezultata**, a koje neće uzimati u obzir!

Istinsko razumevanje odgovora na prethodna pitanja je od izuzetne važnosti za pravilno programiranje bilo kakvih aritmetičkih operacija. Dodatna objašnjenja se mogu naći u poglavlju 13.

7 NEŠTO MALO O NEGATIVNIM BROJEVIMA

Pretpostavimo podatke zapisane sa tri bita. Svi zaključci će važiti i za osmobitne, šesnaestobitne i tridesetdvobitne podatke. Trobitni podaci su uzeti kao primer samo zbog jednostavnosti. Pomoću N bita moguće je predstaviti 2^N različitih podataka. Prem tome, pomoću tri bita moguće je prikazati ukupno 2^3=8 podataka. To može biti osam brojeva od 0 do 7, ali isto tako može biti i osam različitih boja, ili osam brojeva od 13 do 20. Ako želimo da radimo i sa pozitivnim i sa negativnim brojevina, a da pri tom zadržimo i prikazivanje nule, u obzir dolaze kombinacije od -4 do 3 ili od -3 do 4. Pogledajmo kombinacije bita, označene rimskim brojevima:

		Neoznačeni	Označeni (II komplement)
0	**000**	**0**	**0**
I	**001**	**1**	**1**
II	**010**	**2**	**2**
III	**011**	**3**	**3**
IV	**100**	**4**	**-4**
V	**101**	**5**	**-3**
VI	**110**	**6**	**-2**
VII	**111**	**7**	**-1**

Posmatrajmo, za početak, samo prve tri kolone ove tabele**.** Kada se radi isključivo sa pozitivnim brojevima (neoznačeni podaci), tu uglavnom, nema nedoumica. Kombinacija označena rednim brojem I odgovara binarnom kodu broja 1, kombinacija broj IV, odgovara binarnom kodu broja 6 i tako sa svakom od osam kombinacija. Time je treća kolona definisana. Postavlja se pitanje kako predstaviti negativne brojeve. Negativni broj je matematička fikcija. Na primer, -1 se može posmatrati kao broj koji sabran sa brojem 1 daje nulu. Pokušajmo među ovim kombinacijama da pronadjemo broj koji u zbiru sa 1 daje nulu. To je kombinacija VII:

0 0 1 **(I)**
+ 1 1 1 **(VII)**
1 0 0 0

Jedinca na mestu bita broj tri je četvrti bit i u sistemu sa trobitnim podacima će jednostavno biti otsečena. Prme tome binarnim sabiranjem kombinacija I i VII , dobija se nula, pa je zgodno kombinaciju VII proglasiti brojem –1. Na taj način su dobijeni i ostali negativni brojevi u četvrtoj koloni. Ovo predstavlja negativne brojeve u drugom komplementu. Pored ove osobine, drugi komplement ima još "plemenitih" osobina koje se mogu iskoristiti kod aritmetičkih operacija tako da je postao gotovo isključiva tehnika za predstavljanje negativnih brojeva. Drugi komplement se može dobiti dodavanjem jedinice na prvi komplement, a prvi komplement se dobija invertovanjem svih jedinica u nule i obrnuto. Tako broj –2 možemo dobiti polazeći od 2 na sledeći način:

(broj 2) **0 1 0** ➔ **1 0 1** (prvi komplement broja 2)

(prvi komplement broja 2) **1 0 1**
+ 1
1 1 0 (drugi komplement broja 2 = **–2**)

Drugi komplement se može dobiti instrukcijom **NEG**, a prvi, instrukcijom **NOT**

Opšti oblik zapisa **označenih** brojeva sa N bita bi bio sledeći:

$$-b_{N-1}\bullet 2^{N-1} + b_{N-1}\bullet 2^{N-1} + \ldots + b_1\bullet 2^1 + b_0\bullet 2^0$$

Radi poređenja, opšti oblik zapisa **neoznačenih** brojeva se razlikuje samo u prvom članu:

$$b_{N-1}\bullet 2^{N-1} + b_{N-1}\bullet 2^{N-1} + \ldots + b_1\bullet 2^1 + b_0\bullet 2^0$$

Za **označene** osmobitne podatke, taj opšti oblik postaje:

$$-b_7\bullet 2^7 + b_6\bullet 2^6 + \ldots + b_1\bullet 2^1 + b_0\bullet 2^0, \text{ ili}$$

$$-b_7\bullet 128 + b_6\bullet 64 + \ldots + b_1\bullet 2 + b_0 ,$$

gde su b_7 do b_0, binarne cifre (0 ili 1) osmobitnog podatka $b_7\ b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0$. Na primer, za broj 9 = 0000 1001b, cifre b_3 i b_0 su jedinice, ostale su nule.

Iz opšteg oblika označenih brojeva mogu da se sagledaju dve važne osobine ovog načina zapisa:

- Svi negativni brojevi imaju jedinicu kao najznačajniji bit. Međutim, nije dovoljno samo tu jedinicu pretvoriti u nulu da bi se dobio isti pozitivan broj. Ako pođemo od broja –3, u trobitnom sistemu 101, (broj V), menjanjem prve jedinice u nulo dobijamo 001 što jeste pozitivan broj, ali 1, a ne 3! Ova činjenica se često ispušta iz vida.
- Uopšte nije sve jedno kolika je dužina zapisa označenog broja. Naime, samo prvi član opšteg zapisa je negativan pa nije sve jedno da li se najznačajniji bit množi sa 2^7, ili na primer, sa 2^2. Tako se broj -3 u trobitnom sistemu zapisuje kao 101, ali u osmobitnom sistemu broj 101 nije -3, već +5. Broj -3 kao osmobitni podatak je 1111 1101b. Znači, kada se radi sa označenim brojevima mora se voditi računa o tome kolika je dužina podatka. To se u praksi svodi na to da zapis označenog broja kao BAJT podatak, nije isti kao zapis u formatu REČ ili LONG. Postoje

instrukcije koje prevode označeni BAJT u označenu REČ i označenu REČ u označen LONG. U tu svrhu, mogu se koristiti LDBSE i EXT ili EXTB.

8 NEŠTO MALO O ARITMETIČKIM OPERACIJAMA

Primeri u ovom poglavlju biće dati isključivo sa neposrednim i direktnim adresiranjem. Svi zaključci važe i za podatke do kojih se došlo drugim načinima adresiranja. Cilj ovog poglavlja je da objasni principe izvršavanja i primene aritmetičkih instrukcija, a načini adresiranja su obrađeni u drugom poglavlju.

U grupu aritmetičkih funkcija možemo ubrojati instrukcije za sabiranje, oduzimanje, množenje, deljenje, poređenje kao i neke instrukcije iz grupe "*read-modify-write*". Mikrorocesori serije MCS96 imaju realizovame instrukcije sabiranja sa dva i tri operanda kao i sabiranje sa dodavanjem C flega, i to za jednobajtne i dvobajtne podatke. Neka su registri Ax, Bx i Cx deklarisani kao podaci tipa WORD. (rseg segment isti kao u primerima za načine adresiranja).

```
Rseg at 30h             ; Počinje registarski segment. Adrese rezerviši počev od 30h
    Ax:   dsw   1;  dvobajtni registar Ax zauzima adrese 30h i 31h
                  ;   simbol Ax dobija vrednost 30h
    Bx:   dsw   1
    Cx:   dsw   1

    Ah  equ  Ax+1: byte; Viši bajt registra Ax ima adresu Ax uvećanu za 1
                       ;  Simbol Ah dobija vrednost 31h (30h+1)
    Al  equ  Ax: byte;   Niži bajt registra Ax ima istu adresu kao REČ registar Ax
                       ;  Simbol Al dobija vrednost 30h (istu kao simbol Ax)
    Bh  equ  Bx+1: byte
    Bl  equ  Bx: byte
```

8.1 SABIRANJE, INSTRUKCIJE

Instrukcija **ADD Ax, Bx** dodaje sadržaj dvobajtnog registra na adresi Bx (bajti 32h i 33h), trenutnom sadržaju bvobajtnog registra na adresi Ax (bajti 30h i 31h) i rezultat ostaje u registru Ax (bajti 30h i 31h). Često se u komentarima kaže skraćeno: "registru Ax se dodaje registar Bx". Pri tome se misli na njihove **sadržaje**. Simbol Ax, pri ovakvim deklaracijama u rseg, iznosi 30h i asembler će svuda gde nađe ovaj simbol, umesto njega ubaciti broj 30h. Slično, simbol Bx iznosi 32h, tako da je, uz ovakav rseg, instrukcija ADD Ax, Bx, u svemu potpuno identična instukciji **ADD 30h, 32h**.

Instrukcija **ADD Ax, Bx, Cx** dodaje sadržaj registra na adresi Bx (32h i 33h), trenutnom sadržaju registra na adresi Cx (34h i 35h) i rezultat ubacuje u u registar Ax (30h i 31h). Sadržaji u registrima Bx i Cx ostaju nepromanjeni.

Instrukcija **ADDB Ax, Bx** dodaje sadržaj **jednobajtnog** registra na adresi Bx (32h), trenutnom sadržaju **jednobajtnog** registra na adresi Ax (30h) i rezultat ostaje u **jednobajtnom** registru Ax (30h). To odgovara sabiranju nižih bajtova REČi u Ax i Bx pri čemu **viši bajttovi (registri na adresama 31h i 33h) ostaju nepromenjeni!** Nezavisno od toga što je registar na adresi Ax deklarisan kao REČ, ovakva operacija je legalna i često se pojavljuje u praksi. Da bi se povećala jasnoća programa poželjno je umesto ove instrukcije koristiti **ADDB Al, Bl** uz prethodno izjednačavanje Al sa Ax i Bl sa Bx (pomoću **Al equ Ax:byte** i **Bl equ Bx:byte**, vidi deklaraciju u poglavlju 11), iako obe generšu potpuno isti kôd.

! **VAŽNA NAPOMENA: Ne postoji mogućnost kombinovanja podataka različite dužine.** Ne može se sabirati podatak tipa BAJT sa podatkom tipa REČ. Postoje jedino instrukcije u kojima su **oba** sabirka dvobajtna (ili jednobajtna) i isti takav je i rezultat.

Ako se ukaže potreba sabiranja podatka tipa REČ i podatka tipa BAJT, podatak tipa BAJT mora se pre sabiranja pretvoriti u REČ. To pretvaranje zavisi od toga da li je podatak označen ili ne. Na primer, ako treba sabrati Ax sa Bl i ako su u tim registrima **označeni** podaci, Bl treba najpre prevesti u dvobajtni registar kao podatak tipa OZNAČENA REČ, pa tek onda sabirati:

```
LDBSE   Cx, Bl;   konverzija označenog bajta u označenu reč
ADD     Ax, Cx
```

Registar Cx je korišćen samo da bi se izbegla zabuna. Potpuno je ispravno koristiti instrukciju LDBSE Bx, Bl (ili jos bolje EXTB Bx), a potom sabrati Ax i Bx (slučaj označenih podataka).

Za slučaj neoznačenih podataka:

```
LDBZE   Cx, Bl;   konverzija neoznačenog bajta u neoznačenu reč
ADD     Ax, Cx
```

Postoji i instrukcija za sabiranje jednobajtnih podataka sa tri operanda, na primer, **ADDB Al, Bl, Cl**.

! Dozvoljeno je da operandi budu isti. Na primer, **ADD Ax, Ax** je legalna instrukcija koja udvostručava sadržaj u dvobajtnom registru Ax (uzgred, to je i najbrži način da se broj pomnoži sa dva). Ovo važi za sve instrukcije!

Postoje i unarne instrukcije sabiranja **INC** i **INCB** koje podatak uvećavaju za 1. Prva se odnosi na podatak tipa REČ, a druga na podatak tipa BAJT. Obe instrukcije su primenljive i na označene i na neoznačena podatke. Instrukcija **INC Ax** je po efektu (po postavljanju flegova i po rezultatu) identična instrukciji **ADD Ax, #1**, samo je nešto brža i manje zauzima prostora u memoriji.

8.1.1 Sabiranje sa prenosom

Instrukcija **ADDC Ax, Bx** dodaje sadržaj dvobajtnog registra na adresi Bx (bajti 32h i 33h) trenutnom sadržaju bvobajtnog registra na adresi Ax (bajti 30h i 31h), pa rezultat uveća za 1 ako je

C fleg pre ove instrukcije bio setovan. Ako je C fleg obrisan pre izvršavanja ove instrukcije, uvećavanja nema. Rezultat ostaje u registru Ax (bajti 30h i 31h). Treba imati u vidu da mnoge instrukcije utiču na C fleg, a da ADDC dodaje **trnutno** stanje C flega, znači ono koje je odredila poslednja instrukcija pre ADDC.

Postoji i instrukcija **ADDCB** koja sabira jednobajtne podatke i dodaje C fleg zbiru.

Sabiranje sa prenosom se koristi za rad sa višebajtnim podacima. Ako, na primer, želimo da pomoću ADDB (instrukcija namenjene podacima tipa BAJT) saberemo dvobajtne podatka, to treba uraditi tako što se niži bajti saberu pomoću ADDB, a viši pomoću ADDCB, da bi višem bajtu rezultata dodao eventualni prenos, nastao pri sabiranju nižih bajtova. Pri tom ADDCB mora biti neposredno iza prve. Na primer:

```
ADDB    Al, Bl   ; Dodaje niži bajt Bx nižem bajtu Ax. Generisaće prenos, ako ga ima
ADDCB Ah, Bh ; Dodaje viši bajt Bx višem bajtu Ax i prenos ako ga je bilo u prethodnoj ins.
```

Efekat ove dve instrukcije je identičan efektu instrukcije **ADD Ax, Bx**. Zbog toga, dok postoji dvobajtna instrukcija ADD, sabiranje pomoću kombinacije ADDB i ADDCB nema praktičnog smisla, ali ako sabiramo podatke tipa LONG (četvorobajtne), tada moramo upotrebiti dve instrukcije, prvo ADD za sabiranje nižih REČi, a posle ADDC za sabiranje viših REČi. Podaci koje sami kreiramo mogu biti, na primer, i osmobajtni. Oni bi se sabirali tako što se najniže reči saberu pomoću ADD, a zatim se pomoću tri ADDC instrukcije saberu ostale više reči.

Primer sabiranja sa prenosom:
Ako pomoću **dvobajtnog** sabiranja, sabiramo podatke 128 i 128 kombinacijom instrukcija iz prethodnog primera, to sabiranje bi izgledalo: **0000 0000 1000 0000b** + **0000 0000 1000 0000b** = **0000 0001 0000 0000b**, ili, razloženo na bajte:

viši bajt	niži bajt	
0000 0000	**1000 0000**	**128**
+ 0000 0000	**1000 0000**	**+ 128**
0000 0000	**1| 0000 0000**	**256**
+ 1		
0000 0001		

U primeru svetlija jedinica označava stanje C flega. Kada bi se i viši bajti sabirali bez dodavanja C flega, rezultat bi bio nula.

! Ne postoji sabiranje sa prenosom sa tri operanda, ni ADDC, ni ADDCB.

8.2 SABIRANJE, TIPOVI PODATAKA I FLEGOVI

Posmatrajmo dva slučaja sabiranja:

- U prvom slučaju, saberimo brojeve 1 i 253. Takvo sabiranje može da se izvrši nizom instrukcija: **LDB Al, #1; LDB Bl, #253; ADDB Al, Bl;** Rezultat koji očekujemo u registru Al je svakako 254. Ovo sabiranje označićemo rimskim brojem **I**.
- U drugom slučaju, saberimo brojeve 1 i -3. Takvo sabiranje može da se izvrši nizom instrukcija: **LDB Al, #1; LDB Bl, # -3; ADDB Al, Bl;** Rezultat koji očekujemo u registru Al je svakako -2. Ovo sabiranje označićemo rimskim brojem **II**.

Posmatrajmo sada binarne kodove ova dva sabiranja:

- **I** slučaj

0000 0001	**1**
+1111 1101	**+253**
1111 1110	**254**

- **II** slučaj

0000 0001	**1**
+1111 1101	**-3**
1111 1110	**-2**

Ako malo bolje pogledamo binarne oblike podataka (koje mikroprocesor jedino razume), primetićemo da su oba sabiranja identična, kako po ulaznim podacima, tako i po rezultatu. Rezultat je u oba slučaja 1111 1110b. Da li je to 254 ili -2? Odgovor je **DA**. Upravo tako, rezultat je i 254 i -2, kako ko voli. Tačnije, ako programer **želi** da ulazni podaci budu označeni (1-3), rezultat treba da tumači kao označen broj (a tada je 1111 1110b zaista -2), a ako **želi** da ulazni podaci budu neoznačeni (1+253), rezultat treba da tumači kao neoznačen broj (a tada je 1111 1110b zaista 254). O programerovim **željama**, mikroprocesor niti ima, niti može da ima pojma! Znači, jedini koji može da napravi razliku između slučaja I i II je **programer**.

Postavlja se pitanje koji će se flegovi setovati u primeru I, a koji u primeru II. Posmatraćemo samo fleg prekoračenja (V) i fleg koji pokazuje da je rezultat negativan (N fleg), ostali nisu problematični.

- I slučaj: Izgleda da ima prekoračenja (rezultat 254 je van opsega –128 do 127) i da bi V fleg trebalo da bude setovan. Rezultat je pozitivan (+254), pa izgleda da N fleg ne bi trebalo da bude postavljen. **N E T A Č N O !**
- II slučaj: Izgleda da nema prekoračenja (rezultat -2 je u opsegu -128 do 127) i da bi V fleg trebalo da bude obrisan. Rezultat je negativan (-2), pa izgleda da bi N fleg trebalo da bude postavljen. **T A Č N O !**

Međutim, teško je očekivati od mikroprocesora, ma kako verovali u njegove sposobnosti, da od istih ulaznih podataka, koji daju isti rezultat, različito postavlja flegove u zavisnosti od želje programera. Zbog toga će flegovi V i N biti **uvek** određeni uz pretpostavku da programer podatke tumači kao označene, znači, **kao u slučaju II** (vidi poglavlje 13). Ako programer radi sa neoznačenim brojevima, prekoračenje može da proveri testiranjem C flega, a ne testiranjem V flega

(V fleg tada nema nikakav smisao), ali ako radi sa označenim podacima, V fleg je taj koji nosi informaciju o prekoračenju, a ne C fleg. Informacija o tome da je broj negativan, kada programer radi samo sa pozitivnim brojevima nije potrebna, jer ni jedan podatak programer neće tumačiti kao negativan broj. Negativan rezultat posle oduzimanja dva neoznačena broja, biće naznačen prekoračenjem (opet pomoću C flega).

! VAŽNA NAPOMENA: Sve ulazne podatke (kao i izlazni) bi trebalo tumačiti na isti način, ili kao označene ili kao neoznačene. Kombinacije označenih i neoznačenih ulaznih podataka daju rezultate koje je teško interpretirati i takvo programiranje nosi veliku opasnost od grešaka koje mogu biti veoma neprijatne i problematične za otkrivanje.

! Sabiranje radi potpuno isto i za označene i neoznačene podatke i daje tačan **označeni** rezultat kada su ulazni podaci **označeni**, a tačan **neoznačeni** rezultat kada su ulazni podaci **neoznačeni**.

8.3 ODUZIMANJE

Oduzimanje se svodi na sabiranje uz prethodnu promanu znaka drugom operandu. Zbog toga, sve primedbe vezane za sabiranje, odnose se i na oduzimanje.

Instrukcije za oduzimanje su **SUB** i **SUBB**. Na primer, SUB Ax, Bx oduzima sadržaj dvobajtnog registra na adresi Bx (bajti 32h i 33h), od trenutnog sadržaja bvobajtnog registra na adresi Ax (bajti 30h i 31h) i rezultat ostaje u registru Ax (bajti 30h i 31h). Postoje varijante ovih instrukcija sa dva i tri operanda.

SUBC i **SUBCB** su instrukcije za oduzimanje sa pozajmicom. Na primer, **SUBC Ax, Bx** oduzima sadržaj dvobajtnog registra na adresi Bx (bajti 32h i 33h) od trenutnog sadržaja bvobajtnog registra na adresi Ax (bajti 30h i 31h), pa rezultat umanji za 1 ako je C fleg u pre ove instrukcije bio obrisan. Ako je C fleg bio setovan pre izvršavanja ove instrukcije, umanjivanja nema. Rezultat ostaje u registru Ax (bajti 30h i 31h).

Kako MCS96 nema poseban fleg pozajmice (B fleg) već se C fleg kod oduzimanja određuje tako da je **C = not B** (vidi poglavlje 13), to oduzimanje pozajmice od rezultata, znači oduzimanje komplementarne vrednosti C (jer je i **B = not C**).

Postoje i unarne instrukcije oduzimanja **DEC** i **DECB** koje podatak umanjuju za 1. Prva se odnosi na podatak tipa REČ, a druga na podatak tipa BAJT. Obe instrukcije su primenljive i na označene i na neoznačena podatke. Instrukcija **DEC Ax** je po efektu (po postavljanju flegova i po rezultatu) identična instrukciji **SUB Ax, #1**, samo je nešto brža i manje zauzima prostora u memoriji.

8.3.1 Poređenje, specijalni slučaj oduzimanja

Operacija veoma slična oduzimanju je poređenje. Poređenje dva podatka se vrši njihovim oduzimanjem i tumačenjem rezultata. Jasno je, da ako oduyimanje daje nulu, da su podaci jednaki, ako daje pozitivan rezultat, da je podatak od koga se oduzima veći, a ako je rezultat negativan, da je umanjenik manji od umanjioca. Tako odnos podataka koje poredimo možemo ustanoviti i ako samo znamo u koju od ove tri kategorije spada rezultat oduzimanja. Tačna vrednost razlike nam nije neophodna.

Postoje tri instrukcije za poređenje podataka zavisno od njihove dužine. **CMPB** poredi podatke tipa BAJT (i označen i neoznačen), **CMP** poredi podatke tipa REČ (i označena i neoznačena), i **CMPL** koji poredi podatke tipa LONG. Razlika instrukcije **CMP Ax, Bx** i insrukcije **SUB Ax, Bx** je jedino u tome što u slučaju CMP instrukcije registar Ax ostaje nepromanjen, rezultat oduzimanja ide u interni, korisniku nedostupni registar (jer se pretpostavlja da ga i ne interesuje). Sve ostalo je isto.

Cilj CMP instrukcija je **da postavi flegove kako bi se iza nje moglo da izvrši grananje** zavisno od odnosa podataka (da li su jednaki ili je jedan od njih veći od drugog). Zbog toga, po pravilu, iza CMP instukcije ide naka od instrukcija grananja koja tastira flegove i grrana program zavisno od njihovog stanja.

Kako se poređenje svodi na oduzimanje, to se ista CMP instrukcija koristi i ya označene i za neoznačene podatke, ali se instrukcije grananja, koje dolaze iza CMP, razlikuju za označene i neoznalene podatke. Ovde je data tabela grananja iza CMP u zavisnosti od željenog uslova i to za označene i neoznačene podatke. Zbog svog značaje tabela je ponovljena u dodatku, na kraju priručnika.

U tabeli je dato koju instrukciju grananja treba primeniti posle instrukcije **CMP A,B** (gde su A i B **označene** veličine), a koju posle **CMP U,V** (gde su U i V **neoznačene** veličine), da bi se dobilo naznačeno grananje.

CMP A, B A i B su **označeni** podaci		**CMP U, V** U i V su **neoznačeni** podaci	
JGE	skoči ako je **A ≥ B**	**JC**	skoči ako je **U ≥ V**
JGT	skoči ako je **A > B**	**JH**	skoči ako je **U > V**
JLE	skoči ako je **A ≤ B**	**JNH**	skoči ako je **U ≤ V**
JLT	skoči ako je **A < B**	**JNC**	skoči ako je **U < V**
JE	skoči ako je **A = B**	**JE**	skoči ako je **U = V**
JNE	skoči ako je **A ≠ B**	**JNE**	skoči ako je **U ≠ V**

Tabela 8.1 *Grananje posle poređenja*

! CMP samo postavlja flegove kao priprema za instrukciju grananja.

! CMP je isti i za označene i neoznačene podatke, instrukcije grananja se razlikuju.

! Postoji **CMPL** instrukcija koja poredi cele podatke tipa LONG. Ova činjenica se često ispušta iz vida. Instrukcija ima ograničenje da oba operanda moraju biti registri. Za ostale dve instrukcije poređenja važi da jedan od podataka može biti bilo gde u registarskom prostoru (drugi mora biti registar).

8.4 MNOŽENJE, DELJENJE

Da bi se ilustruvale ove aritmetičke operacije potrebno je registarski segment definisan u poglavlju 8 proširiti deklaracijom jednog četvorobajtnog registra. Ekvivalencije naznačene iza deklaracija ostaju nepromenjene.

```
Rseg at 30h          ; Počinje registarski segment. Adrese rezerviši počev od 30h
    Lon:  dsw  1; Četvorobajtni registar Lon zauzima adrese 30h, 31h, 32h i 33h
    Ax:   dsw  1;  Dvobajtni registar Ax zauzima adrese 34h i 35h
    Bx:   dsw  1;  Dvobajtni registar Bx zauzima adrese 36h i 37h
    Cx:   dsw  1;  Dvobajtni registar Cx zauzima adrese 38h i 39h
    Ah  equ  Ax+1: byte; Viši bajt registra Ax ima adresu Ax uvećanu za 1
    Al  equ  Ax: byte;   Niži bajt registra Ax ima istu adresu kao REČ registar Ax
    Bh  equ  Bx+1: byte
    Bl  equ  Bx: byte
```

Ovaj put simboli **Lon, Ax, Bx** i **Cx** dobijaju vrednosti 30h, 34h, 36h i 38h, respektivno. Tim brojevima će asembler, pre prevođenja, zameniti ove simbole. Ekvivalencije definišu simbole **Al** ≡ 34h, **Ah** ≡ 35h, **Bl** ≡ 36h, **Bh** ≡ 37h. Znak ≡ je upotrebljen da bi se naznačilo da je to vrednost **simbola**, a ne sadržaj tog registra.

*Insistiranje na adresama u komentarima je samo radi orjentacije i da bi se izbegle nejasnoće u fazi učenja principa rada mehanizama za dodelu adresa. Programeri **ne treba** da vode previše računa o adresama. Asembler i linker preuzimaju na sebe posao oko korektnog dodeljivanja adresa. Programeri treba da teže isključivo radu sa simboličkim imenima promenljivih. Komentari treba da se odnose na **smisao** promenljive, a nikako na njenu adresu!*

Za množenje postoje instrukcije:

- **MUL**, koja množi dva podataka tipa označena REČ (sa dva ili tri operanada);
- **MULB**, koja množi dva podataka tipa označen BAJT (sa dva ili tri operanada);
- **MULU**, koja množi dva podataka tipa neoznačena REČ (sa dva ili tri operanada);
- **MULUB**, koja množi dva podataka tipa neoznačen BAJT (sa dva ili tri operanada);

Programer treba da se opredeli za jednu od ovih instrukcija zavisno od veličine podataka i zavisno od toga da li radi sa označenim ili neoznačenim podacima.

Instrukcija **MUL Lon, Ax, Bx** množi sadržaj dvobajtnog registra na adresi Ax (bajti 34h i 35h), trenutnim sadržajem dvobajtnog registra na adresi Bx (bajti 36h i 37h) i rezultat smešta u

četvorobajtni registar Lon (bajti 30h, 31h, 32h, 33h). Pri pom se sadržaji oba ulazna registra tretiraju kao označeni brojevi. Izlazini podatak je proizvod ulaznih ako se tretira kao označen LONG.

Interesantno je prokomentarisati ovu istu instrukciju sa dva operanda, na primer, **MUL Lon, Ax**. Ova instrukcija množi sadržaj **niže reči registra Lon** (bajti 30h i 31h), trenutnim sadržajem dvobajtnog registra na adresi Ax (bajti 34h i 35h) i rezultat smešta u četvorobajtni registar Lon (bajti 30h, 31h, 32h, 33h). Viša reč registra Lon se ne uzima u obzir prilikom množenja! Znači, jedan označeni ulazni podatak treba ubaciti u Ax, a drugi u nižu reč LONG registra koji igra i ulogu izlaznog podatka. Svi podaci su označeni.

! **MUL Lon, Ax** ne množi LONG podatak Lon, već samo njegovu **nižu reč** podatkom u Ax. Slično, **MULB Cx, Bl** ne množi REČ podatak Cx, već samo njegov **niži bajt** podatkom u Bl, a šesnaestobitni označeni rezultat se smešta u Cx.

Slični komentari mogu se primeniti i na ostale instrukcije množenja.

! **Množenje ne utiče ni na jedan fleg**. Do prekoračenja po prirodi stvari ne može doći.

Primer:

```
LD      Ax, #4402h
LDB     Bl, # -1
MULB    Ax, Bl
```

Koliki će biti sadržaj registra Ax posle operacije množenja?

U ovom primeru namerno je napunjen i niži (sa 02h) i viši (sa 44h) bajt registra Ax. Međutim, od značaja je samo niži bajt. Viši bajt će biti zanemaren i množiće se brojevi -1 i 2. Rezultat će biti ubačen u **ceo** šenaestobitni registar Ax. Prema tome u registru Ax će biti rezultat množenja bajt podataka -1 i 2, predstavljen kao označena REČ (-2, to jest 0**FFFEh**).

Specijalan slučaj množenja je množenje konstantom oblika 2^N. Tada umesto množenja treba svakako upotrebiti šiftovanje nalevo za N mesta. Interesantno je napomenuti da množenje pomoću šiftovanja radi kako valja i za pozitivne i za negativne brojeve. Prekoračenje prilikom pomeranja je moguće, a flegovi koji to signaliziraju se uredno postavljaju (C fleg za neoznačene, a V fleg za označene podatke).

Operacija šiftovanja je daleko brža od množenja, pa zato uvek umesto **MULB Ax, #16** treba upotrebiti **SHL Ax, #4**, ili umesto **MULUB Bx, #128**, treba koristiti **SHL Bx, #7**. Nekad se čak i množenje sa 10 realizuje pomoću šiftovanja i sabiranja. Naime, $10 = 2^3 + 2^1$, pa je podatak moguće s jedne strane pomeriti za tri mesta ulevo, s druge strane za jedno mesto, i rezultate sabrati.

! Množenje konstantom oblika 2^N realizovati šiftovanjem nalevo.

U literaturi se često koristi zapis 16*16→32 koji bi opisao množenje kakvo realizuje instrukcija MUL (brojke ilustruju dužinu podatka u bitima). Shodno tome, MULB bi bio zapisan kao 8*8→16. Ne postoje množenja kombinacija podataka različitih dužina

! Ne postoje instrukcije za množenje 16*8→16, niti 32*16→32.

8.4.1 Deljenje

Za deljenje postoje instrukcije:

- **DIV**, koja deli podatak tipa označen LONG podatkom tipa označena REČ;
- **DIVB**, koja deli podatak tipa označena REČ podatkom tipa označen BAJT;
- **DIVU**, koja deli podatak tipa neoznačen LONG podatkom tipa neoznačena REČ;
- **DIVUB**, koja deli podatak tipa neoznačena REČ podatkom tipa neoznačen BAJT;

Prve dve dele označene, a druge dve, neoznačene podatke. Iza mnemonika idu dva operanda, prvi sadrži deljenik, a drugi delilac.

Instrukcija **DIV Lon, Ax** deli sadržaj četvorobajtnog registra na adresi Lon trenutnim sadržajem dvobajtnog registra na adresi Ax i **rezultat celobrojnog deljenja** smešta u nižu reč registara Lon (bajti 30h, i 31h), a u višu reč ovog registra (bajti 32h i 33h) ubacuje **ostatak prilikom deljenja**. Pri pom se sadržaji oba ulazna registra tretiraju kao označeni brojevi. Izlazini podatak je količnik ulaznih ako se tretira kao označena REČ.

! **Deljenje je celobrojno, sa otsecanjem. 17/9 = 1 !**

Slično prethodnom pasusu, instrukcija **DIVUB Ax, Bl** deli sadržaj dvobajtnog registra na adresi Ax trenutnim sadržajem jednobajtnog registra na adresi Bl i **rezultat celobrojnog deljenja** smešta u niži bajt registara Ax (bajt 34h), a u viši bajt ovog registra (bajt 35h) ubacuje **ostatak prilikom deljenja**. Pri pom se sadržaji oba ulazna registra tretiraju kao označeni brojevi. Izlazini podatak je količnik ulaznih ako se tretira kao označen BAJT.

O ostatku u višem bajtu (kod DIVB i DIVUB), odnosno višoj reči (kod DIV i DIVU), treba posebno voditi računa kad je deljenje u petlji i kad rezultat deljenja postaje ulazni podatak za sledeći prolaz kroz petlju. Tada registar koji sadrži ostatak treba obrisati po završetku svakog prolaska kako ne bi poremetio ulazni podatak za deljenje koje sledi.

Treba takođe voditi računa o tome da se, na primer, DIVB sme upotrebiti jedino dok je rezultat u okvirima označenog BAJTa, iako je prvi operand regisatr tipa REČ. Ako je rezultat van ovg opsega, postaviće se fleg prekoračenja kako je opisano u poglavlju 13.3. U slučaju prekoračenja, sadržaj rezultantnog registra je nedefinisan.

Koristeći opis pomenut kod množenja, deljenje DIV se može opisati kao 32/16→16, a DIVB kao 16/8→8 (brojke ilustruju dužinu podatka u bitima). Shodno tom zapisu, treba naglasiti da **ne postoji** deljenje 32/16→32, niti 16/8→16. Ova poslednja činjenica se često gubi iz vida pa se, na primer, koristi DIVB instrukcija da se podele brojevi 10000 i 2 jer i jedan i drugi ulazni podatak zadovoljavaju zahteve. Međutim, izlazni podatak ne može da stane u jednobajtni registar, pa će ovakvo deljenje dati nedefinisan rezultat i postaviti fleg prekoračenja (V fleg).

! Kada ima prekoračenja kod deljanja, dobijeni rezultat je **neupotrebljiv.**

Specijalan slučaj deljenja je deljenje konstantom oblika 2^N. Tada umesto deljenja treba svakako upotrebiti šiftovanje udesno za N mesta. Pri tom za označene podatke treba koristiti SHRA instrukciju (ili SHRAB ili SHRAL), a za neoznačene SHR ili SHRB ili SHRL

Operacija šiftovanja je daleko brža od deljenja, pa zato uvek umesto **DIVB Ax, #8** treba upotrebiti **SHRA Ax, #3**, a umesto **DIVUB Bx, #32**, treba koristiti **SHR Bx, #5**.

! Deljenje konstantom oblika 2^N realizovati šiftovanjem udesno

Primer1:

```
LD      Ax, #0048
LDB     Bl, # 5
DIVB    Ax, Bl
```

*Niži bajt Ax sadrži rezultat celobrojnog deljenja brojeva 48 i 5, a viši bajt Ax sadrži ostatak prilikom ovog deljenja. Prema tome, posle ovog skupa operacija, sadržaj Ax će biti **0309h**.*

Primer2:

```
LD      Ax, #10000
LDB     Bl, # 5
DIVB    Ax, Bl
```

*Posle ovakvog deljenja sadržaj Ax je **nedefinisan**, a V fleg setovan (vidi poglavlje 13.3).*

9 NEŠTO MALO O LOGIČKIM OPERACIJAMA

Serija MCS96 podržava binarne logičke operacije I (instrukcije AND i ANDB sa dva i tri operanda), ILI (instrukcije OR, ORB) i ekskluzivno ILI (instrukcije XOR i XORB) i unarne logičke operacije računanja prvog komplementa (instrukcije NOT i NOTB) i negiranja - menjanja znaka (NEG i NEGB).

Sve logičke operacije se obavljaju bit po bit. Znači da logička operacija nad podacima tipa REČ ustvari predstavlja šesnaest istovremenih logočkih operacija nad pojedinačnim bitima. Zbog toga, pošto se podaci ne trtiraju kao celina, besmislene su informacije o prekoračenjima, pa sve logičke operacije brišu flegove C i V (vidi poglavlje 13). Stanje flegova Z i N se definiše shodno pravilima opisanim u poglavlju 13.

9.1 LOGIČKA I OPERACIJA

Posmatrajmo tabelu istinitosti logičke I operacije:

Ulazi		Izlaz
0	0	0
0	1	0
1	0	0
1	1	1

Ulazi su ravnopravi, to jest, logička I operacija je simetrična (a I b = b I a).

Iz tabele istinitosti može se zaključiti najvažnija osobina logičke I operacije koja joj daje osobinu "kapije" (*gate*):

- Ako je jedan od ulaza nula, izlaz je nula, bez obzira na drugi ulaz.
- Ako je jedan od ulaza jedinica, izlaz je jednak drugom ulazu.

Jedan od ulaznih podataka se može posmatrati kao "ključ" kapije. Neka to bude prvi ulazni podatak u tabeli istinitosti. Kada je taj podatak jedinica (svetliji deo tabele), kapija je "otvorena" i drugi ulazni podatak jednostavno "prolazi" na izlaz. Kapija je "zaključana" kada je ključ na nuli i tada je izlaz kapije nula.

Zbog toga se za logičku I operaciju može reći da je kapija sa nultim izlazom u zaključanom stanju, i nulom kao ključem.

Ova osobina se koristi kada postoji potreba da se u podatku samo neki biti resetuju (da im se stanje svede na nulu), a da se pri tom ostali biti ne promene. Na primer, ako treba bit 1 u podatku koji se nalazi u registru Al svesti na nulu, a ne uticati na ostale bite, to se može izvesti pomoću instrukcije:

ANDB Al, #11111101b; obriši bit 1 u Al

Nezavisno od toga da li je bit 1 bio u stanju nule ili jedinice, posle ove operacije, njegovo stanje će biti nula jer je za taj bit kapija "zaključana". Svi ostali biti će ostati nepromenjeni jer je kapija "otvorena".

Podatak koji u ovom primeru učestvuje kao drugi operand često se naziva "maska". Ako bi bilo potrebno, na primer, obrisati viša četiri bita (b7, b6, b5 i b4), a ostale ne dirati, maska bi bila **#00001111b**. Maska može biti i sadržaj registra. Neka, registar Al sadrži redni broj bita (0 do 15) koji treba obrisati u podatku iz registra Cx a da se pri tom ne promene ostali biti. Ovaj zadatak se može rešiti sledećim skupom instrukcija:

LD Bx, #1111111111111110b; Maska za nulti bit
SHL Bx, Al; Pomeri ulevo sadžaj Bx za onoliko mesta koliki je sadržaj Al
AND Cx, Bx; Sadržaj registra Bx je maska za I operaciju

Obe varijante AND instrukcije postoje i sa dva i sa tri operanda. Instrukcija **AND Ax, Bx, Cx** radi sledeće: Bit 15 registra Ax postaje rezultat logičke I operacije bita 15 podatka u registrima Bx i Cx; Bit 14 registra Ax je razultat logičke I operacije bita 14 podatka u registrima Bx i Cx i tako dalje, sve do bita 0. Prva dva operanda moraju biti registri. Samo poslednji može biti adresiran na drugi način, na primer, **AND Al, Bl, #0FEh**.

! Brisanje pojedinačnih bita se vrši logičkom I operacijom.

! Maska za brisanje bita ima nule na mestima koja se brišu i ostale jedinice.

9.2 LOGIČKA ILI OPERACIJA

Posmatrajmo tabelu istinitosti logičke ILI operacije:

Ulazi		Izlaz
0	0	0
0	1	1
1	0	1
1	1	1

Ulazi su ravnopravi, to jest, logička ILI operacija je simetrična (a ILI b = b ILI a).

Iz tabele istinitosti može se zaključiti najvažnija osobina logičke ILI operacije koja joj daje osobinu "kapije" (*gate*):

- Ako je jedan od ulaza jedinica, izlaz je jedinica, bez obzira na drugi ulaz.
- Ako je jedan od ulaza nula, izlaz je jednak drugom ulazu.

Jedan od ulaznih podataka se može posmatrati kao "ključ" kapije. Neka to bude prvi ulazni podatak u tabeli istinitosti. Kada je taj podatak nula (svetliji deo tabele), kapija je "otvorena" i drugi ulazni podatak jednostavno "prolazi" na izlaz. Kapija je "zaključana" kada je ključ na jedinici i tada je izlaz kapije jedinica.

Zbog toga se za logičku ILI operaciju može reći da je kapija sa jediničnim izlazom u zaključanom stanju, i jedinicom kao ključem.

Ova osobina se koristi kada postoji potreba da se u podatku samo neki biti setuju (da im se stanje postavi na jedinicu), a da se pri tom ostali biti ne promene. Na primer, ako treba setovati bit 4 u podatku koji se nalazi u registru Al, a ne uticati na ostale bite, to se može izvesti pomoću instrukcije:

```
ORB    Al, #00010000b;          postavi bit 4 u Al
```

Nezavisno od toga da li je bit 4 bio u stanju nule ili jedinice, posle ove operacije, njegovo stanje će biti jedinica jer je za taj bit kapija "zaključana". Svi ostali biti će ostati nepromenjeni jer je kapija "otvorena".

Podatak koji u ovom primeru učestvuje kao drugi operand često se naziva "maska". Ako bi bilo potrebno, na primer, setovati nulti i sedmi bit (b7 i b0), a ostale ne dirati, maska bi bila **#10000001b**. Maska može biti i sadržaj registra. Neka, registar Al sadrži redni broj bita (0 do 15) koji treba setovati u podatku iz registra Cx a da se pri tom ne promene ostali biti. Ovaj zadatak se može rešiti sledećim skupom instrukcija:

```
LD     Bx, #0000000000000001b;     Maska za nulti bit
SHL    Bx, Al;          Pomeri ulevo sadžaj Bx za onoliko mesta koliki je sadržaj Al
OR     Cx, Bx;          Sadržaj registra Bx je maska za ILI operaciju
```

! Setovanje pojedinačnih bita se vrši logičkom ILI operacijom.

! Maska za setovanje bita ima jedinice na mestima koja se postavljaju, a ostale nule.

! Ne posroji OR instrukcija sa tri operanda.

9.3 LOGIČKA Ekskluzivno ILI OPERACIJA

Posmatrajmo tabelu istinitosti logičke Eks-ILI operacije:

Ulazi		Izlaz
0	**0**	**0**
0	**1**	**1**
1	**0**	**1**
1	**1**	**0**

Ulazi su ravnopravi, to jest, logička Eks-ILI operacija je simetrična (a Eks-ILI b = b Eks-ILI a).

Iz tabele istinitosti može se zaključiti najvažnija osobina logičke Eks-ILI operacije koja joj daje osobinu "uslovnog invertora":

- Ako je jedan od ulaza jedinica, izlaz je jednak invertovanom drugom ulazu.
- Ako je jedan od ulaza nula, izlaz je jednak drugom ulazu.

Jedan od ulaznih podataka se može posmatrati kao "uslov" invertora. Neka to bude prvi ulazni podatak u tabeli istinitosti. Kada je taj podatak jedinica (svetliji deo tabele) izlaz je invertovani drugi ulazni podatak. Kada "uslov" nije ispunjen (prvi podatak nula) izlaz je jednak drugom ulazu.

Znači, izlaz je jednak ili ulaznom podatku, ili invertovanom ulaznom podatku, zavisno od "uslova". Dok se "uslov" drži na nuli, izlazni podatak je jednak ulaznom, kada se "uslov" promeni na jedinicu, ova operacija postaje jednaka NOT operaciji

Ova osobina se koristi kada postoji potreba da u podatku samo neki biti promene stanje (ako su bili jedinica, da postanu nula i obrnuto), a da se pri tom ostali biti ne promene. Na primer, ako treba invertovati bit 2 u podatku koji se nalazi u registru Al, a ne uticati na ostale bite, to se može izvesti pomoću instrukcije:

XORB Al, #00000100b; invertuj bit 2 u Al

Ako je bit 2 bio u stanju nule, posle ove operacije, njegovo stanje će biti jedinica, ako je bio jedinica, postaće nula. Svi ostali biti će ostati nepromenjeni jer "uslov" za invertovanje nije ispunjen.

Podatak koji u ovom primeru učestvuje kao drugi operand često se naziva "maska". Ako bi bilo potrebno, na primer, invertovati nulti i drugi bit (b0 i b2), a ostale ne dirati, maska bi bila **#00000101b**. Maska može biti i sadržaj registra.

! Invertovanje pojedinačnih bita se vrši logičkom Eks-ILI operacijom.

! Maska za invertovanje bita ima jedinice na mestima koja se invertuju, a ostale nule.

! Ne posroji XOR instrukcija sa tri operanda.

10 INSTRUKCIJE GRANANJA I GENERIČKE INSTRUKCIJE

Za grananje programa postoje dve grupe instrukcija:

- **Instrukcije uslovnog skoka** iza kojih se program grana jedino ako su ispunjeni određeni uslovi.
- **Instrukcije bezuslovnog skoka** iza kojih se program svakako grana.

U grupu instrukcija uslovnog skoka spadaju instrukcije koje prouzrokuju grananje programa zavisno od stanja flegova iz PSW registra (vidi poglavlje 13) i one kod kojih je uslov za grananje stanje određenog bita iz nekog od regisatra u registarskom prostoru (adrese 0 do 255). Posle svake instrukcije uslovnog skoka, program može da se nastavi ili sledećom instrukcijom (kad uslov nije ispunjen), ili od labele koja je operand ovih instrukcija (kad je uslov ispunjen).

U grupu instrukcija bezuslovnog skoka spadaju instrukcije za skok na labelu, za poziv i povratak iz potprograma i instrukcija za posredni skok.

10.1 INSTRUKCIJE USLOVNOG SKOKA

Grananje na osnovu stanja flegova (bilo pojedinačnih, bilo njihoveih kombinacija) obezbeđuju sledeće instrukcije: JC, JNC, JE, JNE, JGE, JGT, JLE, JH, JNH, JV, JNV, JVT, JNVT, JST i JNST. Sledeća tabela ddefiniše uslove skoka ovih instrukcija:

INSTRUKCIJA	SKOČI AKO JE:	INSTRUKCIJA	SKOČI AKO JE:
JC	**C = 1**	**JNC**	**C = 0**
JE	**Z = 1**	**JNE**	**Z = 0**
JLT	**N = 1**	**JGE**	**N = 0**
JLE	**N = 1 ili Z = 1**	**JGT**	**N = 0 i Z = 0**
JH	**C = 1 i Z = 0**	**JNH**	**C = 0 ili Z = 1**
JV	**V = 1**	**JNV**	**V = 1**
JVT	**Vt = 1**	**JNVT**	**Vt = 1**
JST	**St = 1**	**JNST**	**St = 1**

Tabela 10.1 *Uslovi grananja instrukcija uslovnog skoka*

Poglavlja 13 i 14, kao i tabela data u poglavlju 8.3.1, sadrže dodatne informacije u vezi ovih instrukcija.

Za ilustraciju njihove sintakse i načina upotrebe, poslužiće nam primer JE instrukcije. Sve što važi za nju (normalno, osim uslova skoka), važi i za ostale iz ove grupe.

Sintaksa JE instrukcije je:

JE Labela

Instrukcija testira stanje Z flega. Ako je Z fleg setovan, program će se nastaviti počev od labele koja je jedini operand ove instrukcije. Ako je u trenutku izvršavanja Z fleg obrisan, program nastavlja sa radom izvršavanjem instrukcije koja sledi za ovom (nema skoka).

Treba primetiti da je operand **Labela**, što je simbolička predstava adrese operacionog koda instrukcije od koje će program nastaviti izvršavanje. Na primer:

```
DEC   Ax          ; Umanji sadržaj Ax registra
JE    Nula_je     ; Skoči ako je sadržaj Ax došao do nule
                  ; Deo programa koji će se izvršavati …
      …           ; … ukoliko Z fleg nije setovan (Ax nije nula)
      …
Nula_je:
                  ; Deo programa koji će se izvršavati
      …           ; … ukoliko Z fleg jeste setovan (Ax je nula)
      …
```

Labelu treba odabrati tako da asocira na funkciju koju taj deo programa obavlja. Dobar izbor labela umnogome može da olakša praćenje i razumevanje programa. S druge strane, zahtev da se labele ne smeju ponavljati, prava je noćna mora za programere velikih programa. Rešenje je upotreba makroa koji imaju lokalne labele ili deljanje programa na veći broj fajlova.

Sve instrukcije iz ove grupe imaju nedostatak što je arhitekturom mikroprocesora određeno da labela ne sme da bude udaljena više od 128 bajta pre ili 127 bajta iza instrukcije grananja. Normalno, ni od jednog programera se ne bi smelo očekivati da prebrojava bajte instrukcija, pa je asembler ponudio rešenje ovog problema.

Rešenje se sastoji u takozvanoj "**generičkoj instrukciji**" **BE**. Odgovarajuće generičke instrukcije postoje za sve instrukcije uslovnog grananja BC, BNC, BNE, BGE…Njihovi mnemonici se dobijaju iz originalnih zamenom slova **J** u slovo **B.**

Generičke insrukcije nemaju operacioni kôd već ih asembler, pre procesa prevođenja zamenjuje odgovarajućom instrukcijom (ili grupom instrukcija) koje imaju svoj operacioni kôd. U slučaju generičke instrukcije **BE**, ona je po funkciji jednaka **JE** instrukciji. Asembler će je pre prevođenja zameniti ovom instrukcijom (JE) ukoliko je labela dovoljno blizu. Međutim, ako je labela "predaleko", zameniće je sa dve instrukcije, prvom, sa suprotnim uslovom i drugom instrukcijom bezuslovnog skoka na udaljenu labelu.

```
…                                  …
BE    Labela        ➔              JNE   priv_lab
…                                     LJMP  Labela
                                   priv_lab:
                                   …
```

*Zamena generičke instrukcije **BE***

Priv_lab je privremena labela koju generiše sam asembler (nije dostupna programeru) da bi program preskočio instrukciju skoka na Labelu ukoliko Z fleg nije setovan. Ako je Z fleg setovan,

instrukcija JNE neće proizvesti granaje, pa će se izvršiti sledeća, a to je bezuslovni skok na Labelu čime je ostvarena funkcija BE instrukcije.

! Generičke instrukcije BE, BNE, BC, BNC, BGE, BGT… su po funkciji identične instrukcijama JE, JNE, JC, JNC, JGE, JGT …

Kao specijalan slučaj instrukcije iz ove grupe može se posmatrati instrukcija **DJNZ** i **DJNZW**. Generički ekvivalenti ovih instrukcija su instrukcija **DBNZ** i **DBNZW.** Ove instrukcije imaju dava operanda, prvi je registar (tipa BAJT za DJNZ, a REČ za DJNZW), a drugi je labela. Instrukcija radi dve operacije. Najpre umanjuje sadržaj registra za 1, a potom se grana na labelu (drugi operand) ako sadržaj registra nije dostigao nulu. Ako jeste, program se nastavlja sledećom instrukcijom. Na primer:

```
LD     Al, #50        ;  Inicijalizacija brojača (Al) na 50
Petlja:
   …                  ;−  Skup insrukcija koji će se ponoviti 50 puta
   …                  ; /
DJNZ  Al, Petlja
…                     ; Nastavak programa
```

Kao i kod ostalih generičkih instrukcija iz ove grupe, i **DBNZ** nema svoj operacioni kôd već je asembler pre prevođenja zamenjuje **DJNZ** instrukcijom ili grupom instrukcija zavisno od udaljenosti labele. Funkcionalno, instrukcije su identične.

Ovo je jedini mnemonik kod koga je Intel odstupio od principa da sufiks B znači operaciju nad jednobajtnim podacima, bez sufiksa - operaciju nad dvobajtnim, a sufiks L četvorobajtnu operaciju.

10.1.1 Grananje na osnovu stanja pojedinačnih bita

Veoma korisna grupa instrukcija, naročito za upravljačke primene gde se jako puno koriste jednobitne promenljive. U ovu grupu spadju dve instrukcije **JBS** i **JBC** kao njihovi generički ekvivalenti **BBS** i **BBC**. Sintaksa je sledeća:

```
JBS    Registar. BrBita, Labela
```

Gde je **Registar** adresa jednobajtnog registra čiji se bit testira, **BrBita**, redni broj bita koji se testira (0 do 7), a **Labela**, labela na koju će se program odgranati ukoliko je testirani bit jedinica. Ako je nula, program nastavlja sledećom instrukcijom.

Ograničenje da registar mora biti jednobajtni, i da labela ne sme biti udaljena više od ± 127 bajta, prevaziđeno je generičkom instrukcijom **BBS** kod koje prvi operand može biti i **četvorobajtni registar**, a BrBita može biti **0 do 31**. Asembler pre prevođenja sam koriguje adresu registra (na osnovu rednog broja testiranog bita) i ovu instrukciju zamenjuje instrukcijom **JBS**, ili odgovarajućom grupom instrukcija, kada je labela daleko.

Na primer:

```
BBS    Cx.10, Deseti_setovan
```

Ova instrukcija testira deseti bit podatka (tipa REČ) iz registra sa adresom Cx i grana se na labelu ***Deseti_setovan*** *ukoliko je taj bit setovan i biće zamenjena instrukcijom* ***JBS (Cx+1).2, Deseti_setovan****.*

Između prvog i drugog operanda može biti tačka, kao u primerima, a može biti i zarez, zbog kompatibilnosti sa sintaksom drugih instrukcija.

JBC instrukcija uzrokuje grananje programa kada je testirani bit nula. Sve ostalo je isto kao kod **JBS** instrukcije.

! Prvi operand JBS instrukcije mora biti adresiran registarski direktnim načinom.

! Drugi operand (BrBita) mora biti **konstanta**, ne može biti **sadržaj** nekog registra.

10.2 INSTRUKCIJE BEZUSLOVNOG GRANANJA

Ovo je grupa instrukcija koje služe za bezuslovni skok izvršavanja programa. One mogu biti za skok na labelu ili za prelazak i povratak iz potprograma. U prve spadaju:

```
SJMP    Labela
LJMP    Labela
BR      [Registar]
```

Prve dve obezbeđuju skok na blisku i daleku labelu, respektivno. Bliska labela ne sme da bude udaljena od tekuće instrukcije više od ±512 bajta. Daleka labela može biti bilo gde u adresnom prostoru. Njih objedinjuje generička instrukcija **BR Labela** koja nema svoj operacioni kôd, već je asembler pre prevođenja zamenjuje instrukcijom **SJMP** ili **LJMP** zavisno od udaljenosti labele.

Treća instrukcija je "instrukcija posrednog skoka". Registar, čija je adresa u uglastim zagradama, sadrži adresu na koju treba da skoči program. Tipičan primer korišćenja ove instrukcije je realizacija CASE instrukcije Paskala i C jezika (vidi poglavlje 11.7).

Primer:

```
LD    Ax, #Skoci_ovde      ; priprema
BR    [Ax]                 ; skok
```

Posle ove grupe instrukcija, program će se odgranati na labelu ***Skoci_ovde****. Ova labela mora da postoji negde u programu. Smisao ove instrukcije (za razliku od* ***LJMP Skoci_ovde****) je što je sada adresa na koju se skače u registru, pa se zbog toga može menjati u toku izvršavanja programa. Promena adrese uz LJMP instrukciju zahtevala bi ponovo prevođenje programa i nemoguća je dok program radi.*

Instrukacije za prelazak na potprogram su:

```
SCALL   Labela
LCALL   Labela
```

Ove instrukcije ostvaruju prelazak na potprogram koji počinje bliskom ili dalekom labelom, respektivno. Bliska labela ne sme da bude udaljena od tekuće instrukcije više od ±512 bajta. Daleka labela može biti bilo gde u adresnom prostoru. Njih objedinjuje generička instrukcija **CALL Labela** koja nema svoj operacioni kôd, već je asembler pre prevođenja zamenjuje instrukcijom **SCALL** ili **LCALL** zavisno od udaljenosti labele.

Primer:

```
CALL    Kvadrat
```

Ova instrukcija obezbeđuje prelazak na potprogram koji počinje **labelom** Kvadrat. Labela mora da postoji negde u programu!

! Operand CALL instrukcije je **labela**, ne ime modula, niti ima fajla!

Instrukcija za povratak iz potprograma je **RET**. Ona nema ni jedan operand.

! Instrukcija RET služi i za povratak iz interapta jer je u ovoj familiji mehanizam pančenja na stek isti i za prekide (interapte) i za potprograme.

11 NAČINI ADRESIRANJA

Svi primeri navedeni u ovom poglavlju podrazumevaće sledeći registarski segment:

```
rseg at 30h
Ax:   dsw   1
      Al    equ   Ax : byte        ; niži bajt registra Ax
      Ah    equ   Ax+1 : byte      ; viši bajt registra Ax
Bx:   dsw   1
      Bl    equ   Bx : byte
      Bh    equ   Bx+1 : byte
Cx:   dsw   1
      Cl    equ   Cx : byte
      Ch    equ   Cx+1 : byte
```

Poglavlje 2 opisije opšti oblik asemblerske linije. U ovom poglavlju će biti razmatrano polje operanada asemblerskih instrukcija i biće opisano kako su različite sintakse zapisa operanada povezane sa podatkom koji će učestvovati u operaciji.

Kada je u polju "operacija" neka asemblerska instrukcija, operandi treba da ukažu na podatke sa kojima ta instrukcija radi. Na primer, u instrukciji **ADD Ax, Bx**, operacija je sabiranje, a operandi (Ax i Bx) ukazuju na to koje podatke treba sabrati. U ovom slučaju, to su adrese registara čiji se sadržaji sabiraju. Ukoliko je registarski segment definisan kao na početku ovog poglavlja, prethodna instrukcija bi bila identična instrukciji **ADD 30h, 32h**, a simboli Ax i Bx su uvedeni jedino da bi programera resili briga vođenja računa o adresama.

Postavlja se pitanje kako je asembler znao da sadržaju registra na adresi 30h treba dodati **sadržaj** registra na adresi 32h, a ne, na primer, baš broj 32h? Nastavak ovog pitanja bi mogao da bude: "A kako treba zapisati drugi operand, ako želimo da sadržaju Ax dodamo baš broj 32h?" Odgovor leži u **načinu adresiranja** drugog operanda. Po dogovoru, ako operand ispred broja (ili simbola) nema nikakav znak, taj broj (ili simbol koji zamenjuje broj) označava **adresu registra** u kome se nalazi podatak.

Kako ni jedan, ni drugi operand (Ax i Bx, odnosno 30h i 32h) nisu imali nikakve znakove ispred sebe, to su se brojevi 30h i 32h (po dogovoru) odnosili na adrese podataka koji učestvuju u sabiranju. Takođe se podrazumeva da su te adrese u okviru registarskog prostora (<256). Ovakav način adresiranja zove se "**registarsko direktno adresiranje**".

Sa gledišta programera, postoje tri osnovna načina adresiranja koji će ovde samo biri pomenuti kroz primer, a kasnije će biti detaljnije objašnjeni u poglavlju 12.

U primerima koji slede, sadržaju registra Ax se dodje…

- **ADD Ax, Bx** … sadržaj registra na adresi 32h
- **ADD Ax, #32h** … baš broj 32h (neposredno adresiranje)
- **ADD Ax, [Bx]** … sadržaj memorijske lokacije čija je adresa u registru na adresi 32h. Ovakvo adresiranje se zove "posredno" jer operand nije ni podatak koji učestvuje, niti njegova adresa. Do podatka koji će se dodati registru Ax se dolazi posredno, tako što se iz registra na adresi 32h (Bx) očita **adresa**, pa se sa te adrese pokupi podatak za dodavanje.

Treba primetiti da se prethodna tri primera mogu potpuno ravnopravno zapisati kao **ADD Ax, 32h**; **ADD Ax, #32h** i **ADD Ax, [32h]**. Ove tri instrukcije se razlikuju samo po karakterističnim oznakama za neposredno (**#**) i posredno adresiranje (**uglaste zagrade**).

Iz prethodnih pasusa se može zaključiti da je način adresiranja vezan za svaki operand ponaosob. Tu međutim, postoji jedno ograničenje. Naime, ako operacija ima više operanada, svi operandi sem jednog moraju biti adresirani registarskim direktnim adresiranjem. Samo jedan od njih (po pravilu, poslednji), može biti adresiran nekim drugim načinom adresiranja. Način adresiranja tog operanda definiše način adresiranja instrukcije. Ako je on adresiran, na primer, neposredno, često za celu instrukciju kaže da je "neposredno adresirana" iako su svi ostali operandi (ukoliko ih ima), adresirani direktnim registarskim adresiranjem.

! Svi operandi sem jednog (po pravilu, poslednjeg) **moraju biti adrese registara!**

11.1 HARDVERSKI NAČINI ADRESIRANJA

Podacima se može pristupiti pomoću jednog od šest načina adresiranja koje podržavaju mikro-procesori serije MSC-96. To su sledeći načini:

- **REGISTARSKO DIREKTNO ADRESIRANJE**, osnovni tip adresiranja, operand je **adresa registra** u kome se nalazi podatak koji učestvuje u operaciji.
- **POSREDNO ADRESIRANJE**, operand je adresa registra u uglastim zagradama. Taj registar (u uglastim zagradama) sadrži adresu memorijske lokacije gde se nalazi podatak.
- **POSREDNO SA AUTOINKREMENTIRANJEM**, operand je adresa registra u uglastim zagradama iza čega stoji simbol +. Taj registar (u uglastim zagradama) sadrži adresu memorijske lokacije gde se nalazi podatak i on se po završenoj operaciji uvećava za jedan ili za dva, zavisno od toga da li je operacija nad jednobajtnim ili dvobajtnim podacima.
- **NEPOSREDNO ADRESIRANJE**, operand počinje znakom # iza koga sledi baš onaj podatak koji učestvuje u operaciji.

- **INDEKSNO BLISKO**, operand je adresa registra u uglastim zagradama ispred čega se nalazi jednobajtna konstanta nazvana ofset. Sadržaj registra u uglastim zagradama, uvećan za ofset, tretiran kao označen broj, formira adresu memorijske lokacije gde se nalazi podatak.
- **INDEKSNO DALEKO**, operand je adresa registra u uglastim zagradama ispred čega se nalazi dvobajtna konstanta (ofset). Sadržaj registra u uglastim zagradama, uvećan za ofset formira adresu memorijske lokacije gde se nalazi podatak.

Primer instrukcije ADD sa svim načinima adresiranja (način adresiranja cele instrukcije određuje poslednji operand, svi ostali su adresirani registarskim direktnim načinom adresiranja):

ADD Ax, Bx	**REGISTARSKO DIREKTNO**
ADD Ax, [Bx]	**POSREDNO**
ADD Ax, [Bx]+	**POSREDNO SA AUTOINKREMENTIRANJEM**
ADD Ax, #32h	**NEPOSREDNO**
ADD Ax, 15[Bx]	**INDEKSNO BLISKO**
ADD Ax, 1545[Bx]	**INDEKSNO DALEKO**

U ovom poglavlju biće opisano adresiranje onako kako ga obavlja i shvata sam mikroprocesor. Asembler donekle olakšava posao programeru birajući sam koji će od hardveskih načina adresiranja primeniti tako da je skup načina adresiranja sa gledišta programera uži od ovde opisanog skupa. Taj suženi skup se naziva softverskom načinima adresiranja. Za izradu programa koji nisu vremenski kritični, često uopšte nije bitno koji od hardverskih načina je asembler usvojio.

! Svaka instrukcija koja podržava ove načine adresiranja ima poseban operacioni kôd za svaki od njih.

Da bi se efikasno koristile mogućnosti koje ova arhitektura pruža, nužno je potpuno razumevanje načina adresiranja.

11.2 REGISTARSKO DIREKTNO ADRESIRANJE

Ovo adresiranje se koristi za direktan pristup promenljivim iz registarskog prostora mikroprocesora.

SINTAKSA: Programer na mesto operanda piše adresu registra u kome se nalazi podatak koji učestvuje u operaciji. Adresa se može upisati kao simbol koji će kasnije asembler zameniti odgovarajućim brojem ili kao broj (0 do 255). Zapis pomoću broja se u praksi gotovo i ne koristi.

SLIKA U MEMORIJI: Iza operacionog koda sledi po jedan bajt za svaki operand adresiran na ovaj na;in. Taj bajt je adresa registra (0 do 255) u kome se nalazi podatak. Adresa mora da poštuje pravila lociranja (parnost) u zavisnosti od tipa registra, to jest, mora biti paran broj za dvobajtne operacije, odnosno broj deljiv sa 4 za četvorobajtne. Zavisno od instrukcije najviše do tri operanda mogu biti adresirana na ovaj način.

Primeri:

ADD Ax, Bx, Cx ; ax := bx+cx

U ovom primeru, sva tri operanda su direktno adresirana. Prema tome, i Ax i Bx i Cx su adrese registara koji učestvuju u operaciji sabiranja, pri čemu je Ax adresa izlaznog registra (registra u kome će biti rezultat sabiranja posle operacije), a Bx i Cx su adrese registara u kojima se nalaze podaci koji se sabiraju. Znači, podatak iz registra sa adresom Bx se dodaje podatku iz registra sa adresom Cx i zbir se smešta u registar sa adresom Ax. Potpuno identična komanda, sintaksno ispravna, ali bez praktičnog značaja je ***ADD 30h, 32h, 34h****. Simboli Ax, Bx i Cx su uvedeni upravo zato da programer ne bi vodio računa o adresama, inače, dok je registarski segment kao na početku poglavlja, napisati Ax je isto što i napisati 32h (osim što se čitljivosti, jasnoće i promenljivosti programa tiče). Adrese Ax, Bx, i Cx moraju biti parni brojevi jer je ADD operacija nad podacima tipa REČ, što* ***dsw*** *direktiva automatski obezbeđuje.*

Slika ove komande u memoriji ie sledeća: ***44h, 34h, 32h, 30h****. Pri tom je 44h operacioni kôd za sabiranje sa tri registarski direkno adresirana operanda, a 34h, 32h i 30h su adrese registara. U memoriji, redosled adresa registara je suprotan od redosleda operanada. Poslednji operand daje adresu koja ide prva iza operacionog kôda.*

INCB Cl ; cl := cl+1

Ovde je jedini operand adresiran registarski direktnim načinom adresiranja, pa je Cl adresa registra koji sadrži podatak koji učestvuje u operaciji. Operacija je uvećavanje, što znači da se sadržar registra sa adrese Cl uvećava za jedan i rezultat ostaje u registru sa adresom Cl.

11.3 POSREDNO ADRESIRANJE

Pomoću ovog adresiranja može pristupiti podatku u bilo kojoj lokaciji iz adresnog prostora, uključujući i registarski prostor. Najčešće se koristi za preuzimanje podataka pomoću pokazivača (*pointer*).

SINTAKSA: Programer na mesto operanda piše adresu dvobajtnog registra u uglastim zagradama. U tom registru se nalazi adresa podataka koji učestvuje u operaciji. Registar u uglastim zagragdama se zove “Pokazivač”. Adresa registra (onog u uglastim zagradama) se može upisati kao simbol koji će kasnije asembler zameniti odgovarajućim brojem ili kao broj (0 do 255). Zapis pomoću broja se u praksi gotovo i ne koristi.

! Registar u uglastim zagradama **mora biti deklarisan kao REČ !**

SLIKA U MEMORIJI Kod ovog načina adresiranja adresa podatka koji učestvuje u operaciji je smeštena u dvobajtnom registru u registarskom prostoru. Osmobitna adresa tog registra je bajt koji sledi operacioni kod. Upisana adresa (sadržaj dvobajtnog registra) mora da poštuje pravila lociranja (u pogledu parnosti) zavisno od tipa podataka sa kojima se radi. Instrukcija sme da sadrži samo jedan posredno adresiran operand dok se ostali, ukoliko postoje, moraju adresirati direktnim rgistarskim načinom.

! Samo jedan (po pravilu, poslednji) operand može da bude posredno adresiran!

Primeri:

ADDB Al, [Cx] ; Al := Al + BAJT na adresi iz Cx

*U ovom primeru, drugi operand **[Cx]** je adresiran posredno. Cx igra ulogu pokazivača i mnogo prikladnije ime bi mu bilo POKAZ ili slično. Pre ove instrukcije u Cx treba ubaciti adresu podatka koji želimo da dodamo registru Al. Ta operacija se naziva "inicijalizacija pokazivača".*

*Ova operacija radi sledeće: Očita se sadržaj registra Cx i to postaje adresa memorijske lokacije sa koje se pokupi BAJT podatak i doda registru Al. Smisao ovakvog načina adresiranja je u tome da se sada pokazivač može "pomeriti" (promenom sadržaja Cx) i istom instrukcijom pokupiti podatak sa potpuno drugog mesta u memoriji. Na primer, šta bi predstavljao sadržaj registra Al ako se pokazivač inicijalizuje na 8000h (instrukcijom **LD Cx, #8000h**) i 100 puta u petlji ponove sledeće instukcije?*

ADDB Al, [Cx] *; Al := Al + BAJT na adresi iz Cx*

INC Cx *; Pomeri pokazivač na sledeći bajt*

Iza ovakve petlje, sadržaj registra Al bi predstavljao zbir 100 bajta, počev od adrese 8000h. Zbir je po modulu 256 (prenos se ne uzima u obzir).

LD Ax, [Ax] ; ax := REČ na adresi koja je bila u ax

Ova insrukcija može izgledati pomalo "egzotično", međutim, takva rešenja se relativno često mogu sresti u praksi. Sadržai registra sa adresom Ax je pokazivač na neku memorijsku lokaciju. Podatak iz te lokacije (na kojiu ukazuje sadržaj registra Ax) se prebacuje u Ax. Znači u Ax je pre operacije bila neka adresa, a posle operacije, tu se nalazi sadržaj sa te adrese.

U ovom primeru, sadržaj Ax pre operacije mora biti paran broj (jer je operacija LD nad dvobajtnim podacima). Programer sam mora da vodi računa o ovom uslovu, jer asembler ne može da se obezbedi od takvih grešaka (programer sam i upisuje sadržaj u Ax). U slučaju neparnog broja, rezultat operacije je nedefinisan.

pop [ax] ; sadržaj na adresi ax := REČ na adr sp
; sp := sp+2

Pomoću ovakve operacije se podatak sa vrha steka ubacuje u memorijsku kokaciju čija adresa je prethodno upisana u registar Ax. Ponovo, sadržaj registara Ax mora biti paran broj.

11.4 POSREDNO ADRESIRANJE SA AUTOINKREMENTIRANJEM

Razlika u odnosu na klasično posredno adresiranje je jedino u tome što se **posle** operacije pokazivač uvećava za jedan, dva ili četiri, zavisno od toga da li je operacija bila nad jednobajtnim, dvobajtnim ili četvorobajtnim podacima. Smisao ovog tipa adresiranja je da se pokazivač po obavljenoj operaciji automatski pomeri tako da ukazuje na sledeći podatak.

SINTAKSA: Programer na mesto operanda piše adresu dvobajtnog registra u uglastim zagradama i iza zatvorene zagrade zank +. U tom registru se nalazi adresa podataka koji učestvuje u operaciji. Registar u uglastim zagragdama se zove "Pokazivač". Nakon obavljene operacije, pokazivač (tačnije, sadržaj registra u zagradama) se uvećava za jedan, dva ili četiri, zavisno od toga da li je operacija bila nad jednobajtnim, dvobajtnim ili četvorobajtnim podacima. Adresa registra (onog u uglastim zagradama) se može upisati kao simbol koji će kasnije asembler zameniti odgovarajućim brojem ili kao broj (0 do 255). Zapis pomoću broja se u praksi gotovo i ne koristi.

! Registar u uglastim zagradama **mora biti deklarisan kao REČ !**

SLIKA U MEMORIJI Kod ovog načina adresiranja adresa podatka koji učestvuje u operaciji je smeštena u dvobajtnom registru u registarskom prostoru. Osmobitna adresa tog registra je bajt koji sledi operacioni kod. Upisana adresa (sadržaj dvobajtnog registra) mora da poštuje pravila lociranja (u pogledu parnosti) zavisno od tipa podataka sa kojima se radi. Instrukcija sme da sadrži samo jedan posredno adresiran operand dok se ostali, ukoliko postoje, moraju adresirati direktnim rgistarskim načinom.

! Samo jedan (po pravilu, poslednji) operand može da bude posredno adresiran!

Primeri:

```
LD    Bx, [Ax]+        ; Ax := REČ na adresi koja je u Ax
                       ; Ax := Ax+2
```

U registru Ax se pre operacije nalazi neka adresa. Dvobajtni podatak sa te adrese se prebacuje u registar sa adresom Bx, a potom se sadržaj registra Ax uveća za 2. Posle operacije pokazivač (registar Ax) ukazuje na prvi podatak tipa REČ iza onog koji je učestvovao u ovoj operaciji.

```
ADDB    Al, Bl, [Cx]+  ; Al := Bl + BAJT na adresi iz Cx
                       ; Cx := Cx+1
```

Cx je pokazivač. On sadrži adresu neke memorijske lokacije. Podatak tipa BAJT sa te memorijske lokacije se dodaje sadržaju registra Bl i rezultat se smešta u registar Al. Posle toga, pokazivač Cx (sadržaj registra Cx) se uvećava za 1.

11.5 NEPOSREDNO *(IMMEDIATE)* ADRESIRANJE

Ovaj tip adresiranja služi da se neke konstante unesu nepostredno kroz program. Normalno, to mogu biti samo podaci koji su stalni (oni koji se ne menjaju u toku izvršavanja programa).

SINTAKSA: Programer na mesto operanda piše znak # i iza njega neposredno podatak koji učestvuje u operaciji. Podatak se može upisati kao simbol koji će kasnije asembler zameniti odgovarajućim brojem ili kao broj. Za razliku od drugih načina adresiranja, ovde se u praksi ravnopravno pojavljuje i zapis pomoću broja.

SLIKA U MEMORIJI: Iza operacionog kôda instrukcije nalazi se podatak koji učestvuje u operaciji. Za jednobajtne operacije podatak se tretira kao osmobitna veličina, a za dvobajtne operacije, kao šesnaestobitna veličina. Asembler će prihvatiti osmobitne veličine od -256 do +255 i šesnaestobitne od -65536 do 65535, dok će za brojeve van tog opsega prijaviti grešku. Treba primetiti da ovi opsezi prevazilaze opsege označenih veličina. Ovo je zbog toga što asembler radi sa neoznačenim brojevima a znak minus tretira kao unarnu operaciju. Tako će, na primer, -255 bit tretiran kao -(+255) što je isto kao 1. Znači, #-255 će biti tretirano kao BAJT 00000001b. Ipak, sve aritmetičke operacije daju tačnu vrednost, razume se, po modullu 256 što važi za sve BAJT podatke.

Instrukcija sme da sadrži samo jedan neposredno adresiran operand dok se ostali, ukoliko postoje, moraju adresirati direktnim registarskim načinom.

! Samo jedan (po pravilu, poslednji) operand može da bude neposredno adresiran!

Primeri:

```
ADD   Ax, #9          ;Ax := Ax+9
```

*Sadržaju registra Ax (preciznije, registra sa adresom Ax) dodaje se 9. Uočiti razliku, instrukcija **ADD Ax, 9** bi sadržaju registra Ax dodala sadržaj registra sa adresom 9.*

```
Broj   EQU  –10
LDB    Al, #(Broj*4+27)
```

*U registar Al se ubacuje rezultat navedene operacije –10*4+27, to jest, –13. Obratiti pažnju na činjenicu da tu vrednost nije izračunao mikrokontroler, već **asembler** (PC) pre početka prevođenja. Instrukcija je potpuno ista instrukciji **LBD Al, #–13** i upravo tako će izgledati kada počne prevođenje mnemonika (**LBD 30h, #–13**). Koristeći dozvoljene računske radnje i simbole za preračunavanje konstanti moguće je povećati jasnoću i čitljivost programa. Spisak dozvoljenih aritmetičkih i logičkih operacija koje podržava asemblerski jezik, dat je u poglavlju o principima asemblera.*

```
Bit    equ    3
LDB    Bl, #(00000001b SHL Bit)
```

*Ilustracija preračunavanja izraza u asembleru. Ponovo, šiftovanje ulevo za tri mesta **ne radi mikroprocesor** već asembler. Sadržaj **Bl** će biti **00001000b**.*

! Dozvoljeno je, i poželjno, koristiti mnoge računske radnje i simbole da asembler umesto programera, pre prevođenja izračuna vrednosti pojedinih konstanti.

```
LD     Bx, #Ax        ;Bx := 0030h
```

*Kada asembler zameni Ax brojem, ova instrukcija postaće **LD Bx, #30**, koja je potpuno jasna. Medjutim, zapis kao u primeru ima veliki praktičan značaj kad je registarski segment relativno alociran, odnosno kad programer ne zna koju će adresu imati Ax. Ovaj primer ilustruje još jednu činjenicu. Iako je adresa Ax osmobitni podatak, on je proširen nulama do šesnaestobitnog jer je operacija (LD) nad dvobajtnim podacima.*

```
LD     Cx, #Labela
```

I ovaj primer pokazuje kako se u registar (sa adresom Cx) može ubaciti adresa neke progreamske linije, a da programer uopšte ne zna gde će se kôd te linije stvarno naći u memoriji. Podrazumeva se da Labela postoji u programu.

11.6 INDEKSNO BLISKO

Ovaj tip adresiranja služi uglavnom za pristup podacima oko podatka na koga ukazuje sadržaj indeksnog registra. Kada na primer, pristupamo tabeli bajtova, to se radi tako što se obrađuje jedan po jedan bajt. Bajtima se pristupa koristeći posredno adresiranje pri čemu je sadržaj pokazivača adresa podatka koji se trenutno obrađuje (tekući podatak). Ako je u tom slučaju, potrebno da se

preuzme prvi bajt iza tekućeg podatka, primeniće se ovo adresiranje sa ofsetom +1 i indeksnim registrom jednakim pokazivaču. Ofset je negativan za podatke ispred tekućeg

SINTAKSA: Operand u indeksnom adresiranju karakterišu dve veličine. Konstanta ispred uglastih zagrada i adresa dvobajtnog registra u uglastim zagradama. Konstanta ispred zagrada sa naziva "ofset", a registar unutar zagrada "indeksni registar". **Adresa podatka koji učestvuje u operaciji dobija se tako što se sadržaju indeksnog registra doda ofset, interpretiran kao označen BAJT**. Ofset kod indeksno bliskog adresiranja je **konstanta** izmeu -128 i +127. Adresa indeksnog registra (onog u uglastim zagradama) se može upisati kao simbol koji će kasnije asembler zameniti odgovarajućim brojem ili kao broj (0 do 255). Zapis pomoću broja se u praksi gotovo i ne koristi. Ofset se takođe može zapisati kao broj ili kao proizvoljno složen izraz u kome može biti i simbola i brojeva. Međutim, rezultat tog izraza je broj koji predstavlja ofset, a ne adresa registra čiji bi sadržaj igrao ulogu ofseta!

! Indeksni registar **mora biti deklarisan kao REČ !**

! **Ofset je KONSTANTA**. Na mestu ofseta se ne sme pisati adresa registra u nadi da će **sadržaj** tog registra poslužiti kao ofset. Indeksno adresiranje u kome je ofset promenljiva (sadržaj nekog registra) u seriji MCS 96 **ne postoji**. Kada je ofset dat kao sadržaj nekog registra, on mora najpre da se sabere sa sadržajem indeksnog registra i tako dobije pokazivač, pomoću koga se pristupa podatku posrednim adresiranjem. Vidi primer 3 u poglavlju 11.7.2.

SLIKA U MEMORIJI Kod ovog načina adresiranja, iza operacionog koda nalazi se jedan bajt sa adresom indeksnog registra i jedan bajt sa **vrednošću** ofseta koja se tretira kao ozbačen BAJT (-128 do +127). Znači **vrednost** ofseta je zapisana o memoriji tipa ROM, pa je to veličina za čije menjanje je nužno ponovno prevođenje programa i menjanje izvršnog kôda.

Instrukcija sme da sadrži samo jedan indeksno adresiran operand dok se ostali, ukoliko postoje, moraju adresirati direktnim rgistarskim načinom. Adresa podatka koji učestvuje u operaciji (sadržaj indeksnog registra + ofset) mora da poštuje pravila lociranja u pogledu parnosti.

Primeri:

```
LD    Ax, 12[Bx]        ; Ax := REČ na adresi (Bx+12)
```

Bx (registar sa adresom 32h) je indeksni registar. Njegov sadržaj uvećan za dvanaest daje adresu memorijske lokacije sa koje se podatak tipa REČ prebacuje u Ax registar. Obratiti požnju: sadržaj registra Bx se uvećava za 12, a ne za sadržaj registra sa adresom 12. Sadržaj Bx registra mora biti paran da bi se dobio ponovo paran broj posle dodavanja 12.

```
Ofs   equ   -123
LDB   Al, (Ofs+3)[Cx]   ; Al := BAJT na adresi (Cx-120)
```

U ovom primeru ofset je dat kao izraz (Ofs+3). Od sadržaja registra Cx oduzima se 120 da bi se dobila adresa. Sa te adrese se podatak tipa BAJT prebaci u Al registar. Sadržaj Cx pri tom ostaje nepromenjen. Oduzimanje se vrši u internom registru, bez uticaja na Cx.

! Kontra primer:

```
ADD   Ax, Cx[Bx]
```

Šta će biti rezultat ove operacije?

Ona je sintaksno ispravna, ali najverovatnije ne radi ono što je programer zamislio. Ovo je veoma česta greška kod korišćenja indeksnog adresiranja. Neka je sadržaj Cx registra 10h, a

sadržaj indeksnog registra Bx 1000h. Programer očekuje da će podatak koji se dodaje registru Ax biti očitan sa adrese 1010h. Međutim, ***to nije tako!***

Zamenimo samo simbol Cx brojem, onako kako to radi asembler sa svim simbolima pre prevođenja. Sa registraskim segmenton definisanim na početku poglavlja 11 instrukcija je jednaka ***ADD Ax, 34h[Bx]*** *. Odavde već postaje jasno da je ideja iz prethodnog pasusa pogrešna. Naime, podatak koji će se dodati registru Ax biće preuzet sa adrese koja se dobije kada se sadržaju indeksnog Bx registra (1000h) doda ofset, broj 32h. Dakle sa adrese 1032h, a ne 1010h!*

11.7 INDEKSNO DALEKO

Ovaj tip adresiranja razlikuje se od indeksnog bliskog jedino po tome što ofset može da bude šesnaestobitna (dvobajtna) konstanta. Ta razlika, na izgled beznačajna, potpuno menja oblast moguće primene ovog načina adresiranja. Sada ofset može biti i labela, pa se na taj način može pristupiti bilo kojoj lokaciji iz adresnog prostora.

Indeksno daleko adresiranje služi za pristup tabelama i drugim organizovanim grupama podataka, kao što su stringovi. Pri tom, uslov za korišćenje ovog tipa adresiranja je da je početna adresa podataka data kao **labela (dakle, konstantna)**. Kada je početna adresa data kao sadržaj nekog registra, ovaj način adresiranja je **neupotrebljiv**. Tada treba koristiti posredno adresiranje uz prethodno preračunavanje pokazivača (sabiranjem date početne adrese i indeksnog registra)

SINTAKSA: Operand u indeksnom adresiranju karakterišu dve veličine. Konstanta ispred uglastih zagrada i adresa dvobajtnog registra u uglastim zagradama. Konstanta ispred zagrada sa naziva "ofset", a registar unutar zagrada "indeksni registar". **Adresa podatka koji učestvuje u operaciji dobija se tako što se sadržaju indeksnog registra doda ofset.** Sabiranje je šesnaestobitno tako da ofset kod indeksno dalekog adresiranja može biti **konstanta** izmeu -65535 i +65535. Adresa indeksnog registra (onog u uglastim zagradama) se može upisati kao simbol koji će kasnije asembler zameniti odgovarajućim brojem ili kao broj (0 do 255). Zapis pomoću broja se u praksi gotovo i ne koristi. Ofset se takođe može zapisati kao broj ili kao proizvoljno složen izraz u kome može biti i simbola i brojeva. Međutim, rezultat tog izraza je broj koji predstavlja ofset, a ne adresa registra čiji bi sadržaj igrao ulogu ofseta! Kao ofset se često koristi labela definisana negde u programu, **ali nikad adresa registra**.

! Indeksni registar **mora biti deklarisan kao REČ !**

! **Ofset je KONSTANTA**. Na mestu ofseta se ne sme pisati adresa registra u nadi da će **sadržaj** tog registra poslužiti kao ofset. Indeksno adresiranje u kome je ofset promenljiva (sadržaj nekog registra) u seriji MCS 96 **ne postoji**. Kada je ofset dat kao sadržaj nekog registra, on mora najpre da se sabere sa sadržajem indeksnog registra i tako dobije pokazivač, pomoću koga se pristupa podatku posrednim adresiranjem. Vidi primer 3 u poglavlju 11.7.2.

SLIKA U MEMORIJI Kod ovog načina adresiranja, iza operacionog koda nalazi se jedan bajt sa adresom indeksnog registra i dva bajta sa **vrednošću** ofseta koja se tretira kao REČ. Znači, **vrednost** ofseta je zapisana o memoriji tipa ROM, pa je to veličina za čije menjanje je nužno ponovno prevođenje programa i menjanje izvršnog kôda.

Instrukcija sme da sadrži samo jedan indeksno adresiran operand dok se ostali, ukoliko postoje, moraju adresirati direktnim rgistarskim načinom. Adresa podatka koji učestvuje u operaciji (sadržaj indeksnog registra + ofset) mora da poštuje pravila lociranja u pogledu parnosti.

Daleko indeksno adresiranje zauzima jedan bajt više u memoriji u odnosu na indeksno blisko, pa je zbog toga bilo nužno da se operacioni kodovi ovih načina adresiranja razlikuju. Asembler sam, na osnovu vrednosti ofseta odlučuje koji će tip indeksnog adresiranja usvojiti tako da o tome programer ne mora mnogo voditi računa dok brzina ili memorijski prostor nisu kritični (daleko indeksno adresiranje proizvodi kôd koji je nešto sporiji i zauzima jedan bajt više).

Primeri:

```
CLR   Cx                    ; Inicijalizacija
LDB   Al, StringPoc[Cx]     ; Preuzmi pro slovo u stringu
```

Ove dve instrukcije obezbeđuju preuzimanje prvog slova iz stringa sa početnom adresom StringPoc. Podrezumeva se da je labela StringPoc definisana negde u programu. Indeksni registar Cx tada sadrži redni broj slova a ulogu ofseta igra labela koja ima verednost adrese prvog slova u stringu. Uvećavanjem Cx i ponavljanjem druge instrukcije pristupa se ostalim slovima. String može biti, na primer, definisan direktivom:

```
StringPoc:   dcb   `Ovo je string`, 0    ; string, definisan po C standardu
```

! Greška opisana u prethodnom poglavlju (vidi kontra primer) je još češća kod indeksnog dalekog adresiranja kada je početna adresa data kao sadržaj registra. Tada se, po pravilu, pokuša korišćenje tog registra (njegove adrese) kao ofset što je pogrešno (vidi kontra primer).

! Obratiti pažnju, **ne postoji indeksno adresiranje sa autoinkrementiranjem**. Znači, u sintaksi, znak + iza uglastih zagrada je dozvoljen samo kada nema ofseta ispred zagrada.

11.7.1 Specijalni slučajevi indeksnog dalekog adresiranja

- R0 adresiranje (specijalan slučaj dalekog indeksnog)

Po funkciji, ovo adresiranje je identično dalekom indeksnom adresiranju u kome ulogu indeksnog registra ima REČ registar R0. Ovaj registar zauzima prva dva bajta u adresnom prostoru (adresa 0000h), a sadržaj mu je fiksiran na nulu. Simbol R0 ili r0 je standardni simbol definisan od strane Intela. Normalno, programer može da koristi bilo koji drugi, ukoliko mu se ovaj ne sviđa. Sva pravila indeksnog dalekog adresiranje važe i za R0 adresiranje.

Primeri:

```
r0 equ 0
ADDB    Al, 123 [0]     ; Al := Al + BAJT u registru sa adresom 123
LD      Ax, 1234 [r0]   ; Ax := REČ na adresi 1234
LD      Ax, 1234 [0]    ; identično prethodnom
```

U ovim primerima je upotrebljeno daleko indeksno adresiranje s tim da r0 igra ulogu indeksnog registra. Smisao ovog tipa adresiranja je da ono zamni klasično direktno adresitanje. Naime, serija MCS96 nema klasično direktno adresiranja već samo registarsko direktno. Nemoguće je, na primer, preuzeti podatak u Ax registar iz memorijske lokacije sa adresom 1234. Sintaksa klasičnog direktnog adresiranja (koga većina mikroprocesora podržava), bila bi **LD Ax, 1234** ,

međutim, serija MSC96 očekuje adresu registra (0 do 255) kao operand direktnog adresiranja. Problem se rešava upravo R0 adresiranjem. Da bi zadržali kompatibilnost sa navikama programera, asembler dozvoljava i sintaksu direktnog adresiranja, s tim da će takvu instrukciju sam asembler, pre prevođenja zameniti odgovarajućom sa R0 adresiranjem

- SP adresiranje (specijalan slučaj dalekog indeksnog)

Po funkciji, ovo adresiranje je identično dalekom indeksnom adresiranju u kome ulogu indeksnog registra ima REČ registar na adresi 0018h. Ovaj registar pripada grupi regiatara specijalne nameme, a služi kao pokazivač steka. Ovaj registar (uobičajena oznaka je SP), automatski učestvuje u svim operacijama sa stekom. Sva pravila indeksnog dalekog adresiranja važe i za SP adresiranje.

Namena ovog tipa adresiranja je manipulacija sa podacima na steku.

Primeri:

```
SP equ 18h
PUSH    [SP]        ; duplira vrh steka
LD      Ax, 2 [SP]  ; Ax := pretposlednji podatak sa steka
```

U prvom primeru, pokazivač steka se najpre umanji za 2 (zbog instrukcije push) pa se na tu adresu smesti podatak koji se nalazi na adrei na koju je ukazivao SP ***pre*** *izvršavanja instrukcije. Tako se na vrh steka upisuje ponovo isti podatak koji je pre toga bio na vrhu.*

11.7.2 Ilustracija indeksnog dalekog adresiranja

Ovo poglavlje daje dva tipična primera korišćenja dalekog indeksnog adresiranja. Prvi primer je deo programa koji preuzima jedno po jedno slovo iz stringa upisanog u memoriji topa ROM i šalje ta slova na terminal. Potprogram za predaju slova nije napisan već je naznačen samo ulaz u potprogram. Preuzimanje podataka iz stringa rešeno je tako što je kao ofset dalekog indeksnog registra islorišćena labela početnog slova stringa, a kao indeksni registar iskorišćena promenljiva pod nazivom red_br čija namena je da sadrži redni broj slova u stringu (pocev od rednog broja 0)

Obratiti pažnju da je početna adresa stringa poznata i konstantna (definisana kao labela).

Primer 1, prenos stringa na terminal:

```
            rseg
            red_br:     dsw   1       ; redni broj slova u stringu
            slovo:      dsb   1       ; promenljiva koja se šalje terminalu

            cseg
            clr     red_br            ; inicijalizacija
            petlja:
                ldb     slovo,  PORUKA1 [index]
                cmpb    slovo, #0
                je      nastavak
                    inc     index
                    call    Predaj_slovo
            br    petlja
            nastavak:
            ; nastavak programa
            .
Predaj_slovo:
            ; potprogram koji varijablu slovo šalje na terminal
            .
            ret

            cseg at 5000h
PORUKA1:    dcb    'Ovo je prva poruka',0
PORUKA2:    dcb    'Ovo je druga poruka',0
            .
        end
```

U ovom primeu, počev od adrese 5000H, nalaze se, jedna za drugom, poruke koje treba preneti na terminal. Potprogrm Predaj_slovo (njegov kôd nije prikazan), predaje jedno slovo terminalu. Pomoću indeksnog registra (REČ-promenljiva **red_br)**, pristupa se jednom po jednom slovu sve dok se ne dođe do nul-karaktera (različito od nule '0'!), koji označava kraj poruke.

Primer 2, skok na labelu u zavisnosti od promenljive **i**:

```
            rseg
            i:      dsw    1        ; Promenljiva koja određuje gde će program
                                    ; … skočiti (mesto_0, mesto_1, ...mesto_i)
            ax:     dsw    1        ; Pomoćna promenljiva

            cseg
            .
            ; promenjivoj i se u ovom delu dodeljuje željena vrednost
            .
            add     indx, i, i                  ; indx := 2*i (adrese mesto_i su
                                                ; … dvobajtne)
            ld      ax, tab_MESTA [indx]        ; Sada ax pokazuje na željenu
                                                ; … labelu (mesto_i)
            br      [ax]                        ; Skok na labelu mesto_i

tab_MESTA:  dcw     mesto_0
            dcw     mesto_1
            dcw     mesto_2
            dcw     mesto_3
            .
            .
mesto_0:    .                       ;grana koja se izvršava kada je i=0
            .
            br      kraj
mesto_1:    .                       ;grana koja se izvršava kada je i=1
            .
            br      kraj
mesto_2:    .                       ;grana koja se izvršava kada je i=2
            .
            br      kraj
            .
kraj:       .                       ;  nastavak programa
            .
            end
```

U ovom primeru program nastavlja tok od labele mesto_0, mesto_1, mesto_2... zavisno od toga kolika je vrednost **i** (0,1,2...).

Promenljiva **i**, koja može da ima vrednost koliko imamo zapisa u tabeli tab_MESTA, se najpre množi sa dva jer su labele, tačnije adrese, u tabili dvobajtne veličine (dcw direktiva). Potom se tako pomnožena vrednost (promenljiva **indx**), zahvaljujući indeksnom adresiranju dodaje početnoj adresi tabele (tab_MESTA). Tako se dobije adresa (u okviru tabele) na kojoj se nalazi adresa mesta gde treba nastaviti program. Skok se obavlja preko posrednog adresiranja.

Ovo je često korišćen način grananja poznat u višim programskim jezicima kao CASE (paskal i c) ili sračunati GOTO u bejziku i fortranu.

Primer 3: Potprogram za prenos stringa na terminal, kome je ulazni podatak **Poc_adr**, registar tipa REČ i sadrži početnu adresu stringa koji treba preneti na terminal. Razlikuje se od primera 1 jedino po tome što početna adresa nije labela, već sadržaj registra. Ponovo, potprogram za predaju jednog slova neće biti razmatran.

U ovom potprogramu, početna adresa nije labela već se taj podatak nalazi u registru **Poc_adr**. Zbog toga nije moguće primeniti isti princip kao u prvom primeru. Preuzimanje slova iz stringa instrukcijom **LDB slovo, Poc_adr[red_br]**, bilo bi potpuno pogrešno (vidi poglavlje 11.7). Zbog toga će se stringu pristupiti na drugi način, preko pokazivača i posrednog adresiranja.

```
        Public   Str_term                ; Labela pomoću koje se ovaj potprogram poziva

        rseg
Pokaz:          dsw    1                 ; Pokazivač na tekuće slovo
slovo:          dsw    1                 ; Tekuće slovo
        Extrn  Poc_adr                   ; Ulazni podatak (deklarisan u glavnom programu)

        cseg
        Extrn  Predaj_slovo: entry       ; Potprogram deklarisan u glavnom programu
Str_term:                                ; Labela pomoću koje se ovaj potprogram poziva
            clr     Pokaz                ; Inicijalizacija …
            add     Pokaz, Poc_adr       ; … pokazivača, na prvo slovo u stringu
            petlja:
                ldb         slovo, [Pokaz]+
                cmpb        slovo, #0    ; Da li je kraj stringa ?
                je          gotovo       ; Ako jeste, gotovo
                    call        Predaj_slovo
            br      petlja
gotovo:
            ret
            end
```

Razlika u odnosu na primer 1 je jedino u tome što se ovde slovo preuzima posrednim adresiranjem sa autoinkrementiranjem. Zbog toga je izostalo i uvećavanje indeksnog registra instrukcijom **INC**. Inicijalizacija pokazivača na prvo slovo dobijena je sabiranjem rednog broja slova od koga treba početi (ovog puta nula) i početne adrese stringa. U ovom slučaju bilo je dovoljno i samo **ld Pokaz, Poc_adr**.

Deo glavnog programa u kome bi se pozvao ovaj potprogram bi mogao da izgleda:

```
        Public   Poc_adr

        Rseg
Poc_adr:        dsw    1                 ; ulazni podatak za potprogram Str_term

        Cseg
        Extrn  Str_term: entry           ; Potprogram deklarisan van glavnog programa
        .
        ld      Poc_adr, #PORUKA1        ; Priprema ulaznog podatka
        call    Str_term
        .
        cseg  at  5000h
PORUKA1:  dcb    'Ovo je prva poruka',0
PORUKA2:  dcb    'Ovo je druga poruka',0
        end
```

12 ASEMBLERSKI NAČINI ADRESIRANJA

Asembler olakšava programeru adresiranje operanda tako što sam bira vremenski najpovoljniji hardverski način adresiranja. Sa gledišta programera postoje samo tri načina adresiranja:

- NEPOSREDNO
- DIREKTNO
- INDIREKTNO

Neposredno *(immediate)* adresiranje se bira tako što se ispred operanda upiše znak #. Pri tom se asembler opredeljuje isključivo za hardversko neposredno adresiranje.

Direktno adresiranje se bira prostim upisivanjem simboličkog imena promenljive. Pri tom asembler bira između registarskog direktnog i R0 adresiranja. Ako se u trenutku asembliranja pouzdano zna da je promenljiva u registarskom prostoru, asembler će izabrati registarsko direktno adresiranje jer je vremenski optimalno. Da bi se to ispunilo, potrebno je da u okviru registarskog segmenta bude ili deklaracija promenljive (DSW, DSB...) ili direktiva EXTRN (poglavlje 3) kojom se promenljiva definiše kao eksternal. U svim ostalim slučajevima (naredba EXTRN van registarskog segmenta...), asembler bira R0 adresiranje kao jedino sigurno jer se pomoću njega može pristupiti bilo kojoj memorijskoj lokaciji uključujući i registarski prostor. To znači da, ukoliko EXTRN direktivu postavimo van registarskog segmenta, adresiranje promenljivih (onih koje ta instrukcija definiše) će biti preko R0 registra čak i ako su te promenljive registri (definisani u drugom modulu u okviru RSEG).

Indirektno adresiranje se bira upisivanjem željenog registra u uglaste zagrade i ofseta ispred uglastih zagrada. Kada asembler, u trenutku asembliranja, sa sigurnošću može da zna da je ofset nula bira hardversko posredno adresiranje, ukoliko je ofset u granicama -128 do 127 bira indeksno blisko, a u svim ostalim slučajevima bira indeksno daleko adresiranje kao jedino sigurno. Ostali slučajevi obuhvataju da se vrednost ofseta računa na osnovu izraza u kome učestvuju i konstante iz drugih modula ili premestive konstante (njihovu vrednost određuje tek linker). Čak i ako je konačna vrednost izraza (posle linkovanja) u granicama bliskog indeksnog adresiranja, ili nula, primeniće se indeksno daleko. Naime operacioni kod i broj bajtova se razlikuju kod bliskog i dalekog indeksnog adresiranja pa asembler mora da rezerviše određeni broj mesta u ROMu i upiše neki operacioni kod. Kasnije, linker može da menja sadržaje na rezervisanim mestima, ali ne i broj mesta. Takođe linker ne sme da menja operacioni kod instrukcija. Ovo je opšte pravilo.

13 INDIKATORI STANJA PROGRAMA

PSW (*Program Status Word*) registar sadrži indikatore stanja (flegove) koji se postavljaju iza aritmetičkih i logičkih operacija da bi definisali trenutno stanje u kome se nalazi program. Stanje podrazumeva da li je rezultat prethodne operacije bio pozitivan, negativan, nula, da li je imao prekoračenje i slično.

Termin “fleg”, iako tuđica, je odomaćen u žargonu programera. Usvojen (odobren) domaći termin (indikator stanja) je glomazan, može uneti zabunu, a i pored svega toga, nije preterano “domaći”. Kako je prvenstveni cilj ovog priručnika razumljivost, nadalje će biti korišćen termin “fleg”, uprkos ugrožavanju čistoće jezika.

Svakom flegu odgovara po jedan bit šesnaestobitne programske statusne reči PSW.

Raspored flegova u PSW registru je sledeći:

```
15 14 13  12  11 10  9  8     7  6  5  4  3  2  1  0
 Z  N  V  VT  C   x  I  ST     --- int. dozvole ---
```

Gde je x neiskorišćen fleg a “int. dozvole” predstavljaju pojedinačne dozvole za osam vrsta prekida (interapta) koje mikroprocesor podržava. Naime, pored globalne dozvole - I fleg, koji mora biti setovan da bi se **bilo koji** prekid opslužio, svaki od šesnaest maskirajućih prekida ima i svoju pojedinačnu, individualnu, dozvolu. Individualne dozvole osam od ovih šesnaest prekida, nalaze se u programskoj statusnoj reči od bita broj 0 do bita broj 7. Individualne dozvole ostalih prekida su u posebom registru. Individualna dozvola takođe mora biti setovana (pored globalne) da bi se taj prekid opslužio.

Interesantno je napomenuti da viši bajt programske statusne reči koji sadrži flegove (biti broj 8 do 15), **nije u adresnom prostoru** mikroprocesora – nema svoju adresu, tako da mu je **nemoguće pristupiti kao registru**, ni pomoću jedne vrste adresiranja. To znači da nije moguće direktno prebaciti stanje višeg bajta PSW u drugi registar, niti upisat bilo kakav podatak kao u osmobitni BAJT registar. Nižiem bajtu PSW, koji sadrži dozvole prekida, može se pristupiti na adresi 08h ili preko rezervisanog imena INT_MASK.

Ako je iz bilo kog razloga potrebno pristupiti višem bajtu PSW kao celini, to se može obaviti tako što se cela programska statusna reč PSW prvo smesti na stek instrukcijom PUSHF, pa zatim pročita sa steka u neki registar opšte namene tipa REČ (na primer, pomocu “POP Reg1”). Posle ovoga, u višem bajtu registra Reg1, nalazi se visi bajt PSW. U praksi se retko nameće potreba za takvom operacijom jer se svaki fleg može pojedinačno testirati, a postoje i komande za testiranje onih kombinacija flegova koje imaju određeno značenje. Stoga očitavanje bajta sastavljenog od osam pojedinačnih indikatora, od kojih svaki ima drugo značenje, nema previše smisla.

Cela PSW se može smestiti na stek instrukcijom PUSHF ili PUSHA. Ove dve istrukcije takođe **brišu sve flegove**.

PSW se može povući sa steka instrukcijom POPF ili POPA. Pri tom se stanje svih šesnaest bita PSW određuje na osnovu stanja odgovarajućih bita šesnaestobitnog podatka na vrhu steka. Taj podatak je, po pravilu, stanje PSW nekada ranije ubačeno na stek pomoću PUSHF ili PUSHA instrukcije.

Svi flegovi posle reseta su obrisani

13.1 FLEG NULE – Z fleg

Z - Fleg nule (*zero*), kada je setovan, pokazuje da je rezultat prethodne operacije bio nula, a kada je obrisan, da je rezultat prethodne operacije različit od nule. Ovaj fleg ima smisla i za označene i za neočnačene ulazne podatke.

Instrukcije ADDC i SUBC imaju specifično ponašanje u smislu postavljana ovog flega. One brišu Z fleg ako rezultat nije nula ali ga nikad ne postavljaju na jedinicu. Takvo rešenje je usvojeno jer se one najčešće koriste posle ADD ili SUB instrukcija za operacije sa dužim operandima. Da bi stanje Z flega odgovaralo rezultatu kompletne operacije poslednja instrukcija u nizu (po pravilu ADDC ili SUBC) ne sme postaviti Z fleg ako ga je neka od prethodnoh operacija obrisala jer da bi kompletan rezultat bio nula, potrebno je da je svaka od operacija u nizu dala rezultat nula.

Na Z fleg utiču sve instrukcije sabiranja, oduzimanja, sve logičke kao i sve instrukcije pomeranja (šiftovanja). Za detalje oko uticaja videti tabelu u prilogu.

Z fleg se može direktno testirati instrukcijama JE i JNE, a njegovo stanje utiče na instrukcije JH, JNH, JGT, i JLE, u kombinaciji sa drugim flegovima.

Primeri (operacije nad jednobajtnim podacima):

Posle sabiranja –12 i +12, Z fleg je je setovan.

Posle sabiranja –12 i +5, Z fleg je obrisan.

Posle sabiranja 255 i 1, Z fleg je je setovan, kao i posle sabiranja –1 i +1 (to je isto sabiranje, zar ne?).

Ako bi se umesto ADDB koristila operacija nad podacima tipa REČ (ADD), u poslednjem primeru sabiranja 255 i 1, Z fleg se ne bi postavio, i to sabiranje ne bi bilo isto kao sabiranje –1 i +1 (posle koga bi Z fleg bio setovan). Naime, u dvobajtnom svetu, 255 ima binarnu predstavu 0000 0000 1111 1111b, a broj –1 je 1111 1111 1111 1111b. U svetu jednobajtnih podataka ta dva broja (–1 i 255) su identični. O ovoj činjenici treba povesti računa!

13.2 NEGATIV FLEG – N fleg

N - Negativ fleg, kada je setovan, pokazuje da je rezultat prethodne operacije bio negativan pod uslovom da se ulazni podaci tretiraju kao **označeni brojevi**. Zbog toga, ovaj fleg ima svoje osnovno značenje **jedino posle operacija nad označenim brojevima!** Posle operacija nad neoznačenim brojevima, korišćenje ovog flega nema previše smisla osim kod operacija pomeranja (šiftovanja). Prilikom svih operacija šiftovanja, N fleg dobija vrednost najznačajnijeg bita rezultata. Ovo važi i kada je broj pomeranja nula.

N fleg ima korektnu vrednost čak i kada dođe do prekoračenja! Zbog ove osobine, tvrdnja da je stanje N flega identično stanju najznačajnijeg bita rezultata, važi samo kada nema prekoračenja.

Na N fleg utiču sve instrukcije sabiranja, oduzimanja, sve logičke kao i sve instrukcije pomeranja (šiftovanja). Za detalje oko uticaja videti tabelu u prilogu.

N fleg se može direktno testirati instrukcijama JGE i JLT, a njegovo stanje utiče i na instrukcije JGT, i JLE, u kombinaciji sa Z flegom.

Primeri (operacije nad jednobajtnim podacima):

Posle sabiranja –12 i +5, N fleg je setovan.

Posle sabiranja –12 i +15, N fleg je obrisan.

Posle sabiranja –12 i +12, N fleg je obrisan.

Posle sabiranja +125 i +4 rezultat (129) je van opsega OZNAČENOG BAJTA. Kako je binarni zapis rezultata (129 = $2^7 + 2^0$) 1000 0001b, to je najznačajniji bit (bit broj 7) logička jedinica, pa bi se moglo očekivati da N fleg bude setovan. Međutim, posle ovakvog sabiranja, N fleg će biti logička nula da bi ukazao da je prekoračenje bilo sa pozitivne strane (129 > 127). Fleg koji označava prekoračenje (V fleg) će se postaviti u ovom primeru.

Svi primeri podrazumevaju sabiranje nad jednobajtnim podacima (ADDB). Ako bi se umesto ADDB koristilo sabiranje nad REČima (ADD), N fleg bi imao isto stanje u svim primerima, ali V fleg u četvrtom primeru ne bi bio setovan.

Posle šiftovanja podatka 0010 0000 b za jedno mesto ulevo, N fleg je obrisan, a posle šiftovanja istog podatka za dva mesta ulevo, setovan, jer jedinica na mestu bita broj 5 dolazi na mesto bita broj 7 što je za operacije nad jednobajtnim podacima najznačajniji bit. Ovo važi jedino za operaciju šiftovanja podatka tipa BAJT (SHLB). Ako bi se isti podatak šiftovao dva puta operacijom SHL (šiftovanje podataka tipa REČ), fleg N bi bio nula jer je za dvobajtne podatke najznačajniji bit broj 15, a on ostaje nula u ovom primeru.

13.3 FLEG PREKORAČENJA – V fleg

V - Fleg prekoračenja (kada je setovan) pokazuje da je prethodna operacija dala rezultat izvan opsega označenih podataka dužine koja odgovara tipu opetacije. Za osmobitne (jednobajtne) operacije granice su označeni BAJTovi, za šesnaestobitne (dvobajtne) označene REČi, a za tridesetdvobitne (četvorobajtne), označeni LONGovi. **V fleg, dakle, nosi informaciju o prekoračenju jedino ako su ulazni podaci označeni brojevi!** Ovaj fleg nema previše smisla ukoliko programer želi da ulazne podatke tretira kao neoznačene brojeve. Informaciju o prekoračenju u slučaju neoznačenih ulaznih podataka nosi C fleg.

Važno je ponvo napomenuti da se stanje V flega **uvek** određuje na osnovu **označenih** ulaznih podataka, čak i onda kada ih programer tretira kao neoznačene. Na primer, posle jednobajtnog sabiranja 254 i 3, V fleg se NEĆE setovati (biće 0) iako je očekivani rezultat (257) izvan opsega (-128 do +127). Razlog tome je što prilikom određivanja stanja V flega, mikrorocesor tretira podatak 254 (1111 1101b) kao označen bajt, a on tada iznosi –2 (jer je 1111 1101b, binarni zapis za broj –2 u drugom komplementu). Prema tome, fleg prekoračenja se određuje za sabiranje –2 + 3, koje je potpuno legalno, i čiji je rezeltat (1) u pomenutom opsegu. Zbog toga, posle operacija sabiranja i oduzimanja nad neoznačenim brojevima, V fleg treba jednostavno ignorisati (ne koristiti) jer ne nosi nikakvu korisnu informaciju.

Za operacije šiftovanja ulevo (SHLB, SHL i SHLL), V fleg će se setovati ako se najznačajniji bit operanda promeni bilo kad u toku pmeranja. Promena najznačajnijeg bita znači

prekoračenje i ako se šiftovanje nalevo koristi za množenje sa 2^N. Ponovo, V fleg ima svoje osnovno značenje **samo za označene** ulazne podatke.

Fleg prekoračenja se postavlja prilikom deljenja pod sledećim uslovima:
- Ako je rezultat DIVUB deljenja veći od 255 (0FFh)
- Ako je rezultat DIVB deljenja veći od 127 (7Fh) ili manji od –127 (81h)
- Ako je rezultat DIVU deljenja veći od 65535 (0FFFh)
- Ako je rezultat DIV deljenja veći od 32767 (7FFFh) ili manji od –32767 (8001h)

Na V fleg utiču sve instrukcije sabiranja, oduzimanja, deljenja, sve logičke (brišu ga), kao i sve instrukcije pomeranja (šiftovanja). Za detalje oko uticaja videti tabelu u prilogu.

V fleg se može direktno testirati instrukcijama JV i JNV.

Primeri (operacije nad jednobajtnim podacima):

Posle sabiranja –12 i +15, V fleg je obrisan.

Posle sabiranja –12 i +12, V fleg je obrisan.

Posle sabiranja +125 i +4, V fleg je setovan

Posle sabiranja –125 i –4, V fleg je setovan.

Posle niza operacija ***ldb a,#254; ldb b,#3; addb a,b****; V fleg je* ***obrisan!***

Posle šiftovanja podatka 0010 0000 b za jedno mesto ulevo, V fleg je obrisan, jer najznačajniji bit ne menja vrednost. Posle šiftovanja istog podatka za dva mesta ulevo, V fleg je setovan, jer prilikom drugog čiftovanja najznačajniji bit menja vrednost, sa nule na jedinicu. Iz istog razloga, V fleg je setovan, i prilikom šiftovanja za tri ili više mesta ulevo. Ponovo, ako bi se isti podatak šiftovao operacijom SHL umesto SHLB, prekoračenja ne bi bilo jer bi tada bio merodavan bit broj 15, kao najznačainiji bit podataka tipa REČ.

Posle šiftovanja podatka –55 za dva mesta ulevo (množenje sa 4), V fleg je setovan, što je algebarski korektno.

Posle šiftovanja podatka +55 za dva mesta ulevo (množenje sa 4), V fleg je setovan, što je algebarski korektno jedino ako radimo sa označenim podacima. Ako je podatak 55 tretiran kao neoznačen broj posle množenja sa 4, rezultat je 220 što je još uvek u opsegu neoznačenog bajta (0 do 255) i indikacija prekoračenja bi unela zabunu, ako bi programer testirao V fleg.

13.4 TRAJNI FLEG PREKORAČENJA – VT fleg

VT - Trajni fleg prekoračenja se postavlja kad god se postavi V fleg a brišu ga jedino instrukcije CLRVT, JVT i JNVT. Ovaj fleg omogućava testiranje prekoračenja posle niza aritmetičkih instrukcija što je efikasnije od testiranja prekoračenja iza svake instrukcije.

Na VT fleg utiču sve instrukcije koje utiču i na V fleg, s tim da ga ni jedna od tih insrukcija ne može izbrisati. Za detalje oko uticaja videti tabelu u prilogu.

VT fleg se može obrisati instrukcijom CLRVT, a automatski se brise i posle testiranja.

VT fleg se može direktno testirati instrukcijama JVT i JNVT. Ove instrukcije takodje **brišu** VT fleg.

13.5 FLEG PRENOSA – C fleg

C - Fleg prenosa (*carry*) predstavlja najznačajniji deveti bit (bit broj 8) u osmobitnim operacijama odnosno sedamnaesti bit (bit broj 16) u šesnaestobitnim, ili tridesetreći (bit broj 32) u tridesetdvobitnim operacijama. Tačnije ovaj fleg dobija vrednost koju bi imao nepostojeći bit broj 8 u jednobajtnim operacijama (postojeći najznačajniji bit je bit broj 7)

Kod svih operacija šiftovanja, C fleg se postavlja na vrednost koju ima onaj bit koji "ispada" prilikom pomeranja (najmanje značajan bit podatka pri pomeranju nadesno, ili najznačajniji, pri pomeranju nalevo). Kada se vrši šiftovanje za jedno mesto udesno bit broj nula se gubi iz podatka (na njegovo nesto dolayi bit broj 1), ali se njegovo stanje prnosi u C fleg i programer ga može iskoristiti neposredno iza operacije šiftovanja. Na sličan način se ponaša i najznačajniji bit prilikom šiftovanja ulevo.

Prilikom sabiranja C fleg se postavlja ako ima prenosa sa najznačajnijeg bita (ako su najznačajniji biti oba sabiraka jedinice ili ako je jedan od njih jedinica, a postojao je prenos sa prethodnog bita). Prilikom oduzimanja, C fleg se postavlja ako nema pozajmice (*arithmetic borrow*), a briše ako pozajmice ima. Pozajmica postoji kada na mestu najznačajnijeg bita treba oduzeti jedinicu od nule. Neka rešenja mikrorocesora imaju poseban fleg pozajmice, obično označen sa B. Kod serije MCS96 C fleg objedinjuje oba flega tako što pri oduzimanju važi B = not C.

Zahvaljujući opisanim osobinama, C fleg može poslužiti kao indikator prekoračenja u operacijama sa neoznačenim podacima. Dakle, ukoliko se želi proveriti prekoračenje kada se radi sa **neoznačenim podacime, treba testirati C fleg, a ne V fleg!** Ovo važi za sve operacije sabiranja, oduzimanja, kao i šiftovanja u svrhu množenja sa 2^N.

Na C fleg utiču sve instrukcije sabiranja, oduzimanja, sve logičke (brišu ga), kao i sve instrukcije pomeranja (šiftovanja). Za detalje oko uticaja videti tabelu u prilogu.

C fleg se može direktno testirati instrukcijama JC i JNC, a utiče i na instrukcije JH i JNH u kombinacihji sa Z flegom.

C fleg se može obrisati instrukcijom CLRC i postaviti instrukcijom SETC.

Primeri (operacije nad jednobajtnim podacima):

Posle sabiranja podataka 1000 0000b i 0111 1111b, C fleg je obrisan.

Posle sabiranja podataka 1000 0000b i 1000 0000b, C fleg je setovan.

Posle sabiranja podataka 1100 0000b i 0100 0000b, C fleg je setovan.

Posle oduzimanja podatka 0100 0000b od 1100 0000b, C fleg je setovan jer nema pozajmice pri oduzimanju najznačajnijih bita (nepostojeći fleg B bi bio 0).

Posle oduzimanja podatka 1100 0000b od 0100 0000b, C fleg je obrisan jer ima pozajmice pri oduzimanju najznačajnijih bita (nepostojeći fleg B bi bio 1).

Posle šiftovanja podatka 0100 0001b ulevo za jedno mesto, C fleg je obrisan (jer je bit 7 podatka 0, a on prelazi u C fleg), a posle šiftovanja ovog istog udesno za jedno mesto, C fleg je setovana (jer je bit broj nula podatka 0, a on prelazi u C fleg).

13.6 GLOBALNA DOZVOLA PREKIDA I – Fleg

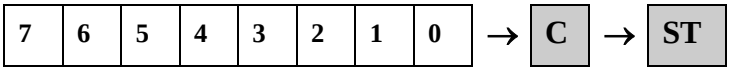
I – Fleg, kada je setovan, pokazuje da su interapti omogućeni. Može da se postavi i obriše programski (instrukcije EI i DI). Kada je obrisan, svi interapti sem TRAP i NMI su onemogućeni. Posle reseta fleg I je obrisan, sto znači da programer mora da setuje ovaj fleg (instrukcijom EI) ukoliko želi da koristi bilo koji maskirajući prekid (interapt). Maskirajući prekidi su svi sem TRAP i NMI.

Instrukcija PUSHF (kao i PUSHA) koja je, gotovo po pravilu, prva instrukcija u interapt rutini (rutina koja opslužuje prekid), brise sve flegove, pa i I fleg. To znači da programer mora da setuje ovaj fleg ukoliko želi da koristi mehanizam prekida u interapt rutini. Instrukcija PUSHF smešta na stek stanje svih flegova, pa i stanje I flega. Poslednja instrukcija u interapt rutini (opet, po pravilu POPF ili POPA) vraća sa steka stanje svih flegova, pa i stanje I flega, tako da po provratku iz interapt rutine stanje I flega je isto kao što je bilo pre ulaska u ovu rutinu.

13.7 OSTALI FLEGOVI

ST - Ovaj fleg (*sticky bit*, u bukvalnom prevodu, prilepak) se postavlja ili briše samo prilikom šiftovanja nadesno. Fleg dobija vrednost bita koji je pre šiftovanja bio u C flegu. Bit najmanje težinske vrednosti operanda prilikom prvog pomeranja nadesno prelazi u C fleg, a prilikom sledećeg pomeranja u ST fleg. Ovaj fleg se zajedno sa C flegom koristi prilikom zaokruživanja rezultata posle deljenja sa 2^N šiftovanjem.

Na sledećoj slici je primer šiftovanja udesno podatka tipa BAJT:



podatak koji se šiftuje

ST bit se može direktno testirati instrukcijama JST i JNST.

PSW.10, deseti bit programske statusne reči u novijim mikrokontrolerima, počev od 80c196KC pa nadalje, je iskorišćen kao dozvoila za novi tip opsluživanja periferija, sličan interaptima, pod nazivom "*Peripheral Transfer Server*", pa je oznaka ovog flega PTSE.

KARAKTERISTIKE PSW:

- SADRŽI KOLEKCIJU FLEGOVA U VIŠEM, I NEKE OD POJEDINAČNIH DOZVOLA PREKIDA, U NIŽEM BAJTU.
- VIŠEM BAJTU SE NE MOŽE PRISTUPITI NI JEDNIM NAČINOM ADRESIRANJA.
- INSTRUKCIJA PUSHF PAMTI PSW NA STEK I UBACUJE 00000000 00000000B U PSW.
- INSTRUKCIJA POPF UBACUJE U PSW ŠESNAESTOBITNI PODATAK SA VRHA STEKA.
- POSLE RESETA, STANJE PSW JE 00000000 00000000B.

14 OPIS INSTRUKCIJA

U ovom poglavlju dat je opis svih instrukcija koje podržavaju mikrokontroleri serije MSC-96. Za opis su korišćene sledeće oznake:

breg je oznaka operanda tipa BAJT (bilo označen ili neoznačen) u registarskom prostoru. Ovi operandi se mogu adresirati **jedino registarskim direktnim načinom**.

baop je oznaka operanda tipa BAJT (bilo označen ili neoznačen) koji može da se nalazi bilo gda u adresnom prostoru. Ovim operandima se može pristupiti pomoću bilo kog načina adresiranja.

wreg je oznaka operanda tipa REČ (bilo označena ili neoznačena) u registarskom prostoru. Ovi operandi se mogu adresirati **jedino registarskim direktnim načinom**.

waop je oznaka operanda tipa REČ (bilo označena ili neoznačena) koji može da se nalazi bilo gda u adresnom prostoru. Ovim operandima se može pristupiti pomoću bilo kog načina adresiranja.

Prefiks **S** (od *source*) označava ulazni podatak. Ako operacija ima dva ulazna podatka, oni su označeni sa S1 i S2.

Prefiks **D** (od *destination*) označava izlazni registar ili memorijsku lokaciju gde se upisuje rezultat operacije.

Prefiks **DS** označava registar u kome se nalazi ulazni podatak, ali koji istovremeno služi i kao izlazni registar (u koji se ubacuje rezultat operacije).

Instrukcije su poređane po abecednom redu. Iza opisa instrukcije sledi sintaksa naredbe u asembleru (podebljanim slovima). Na kraju je dato i dodatno objašnjenje ukoliko je potrebno, kao i uticaj te instrukcije na flegove u PSW registru.

Oznaka svaki od flegova (Z, N, C, V, Vt, St) na koji instrukcija utiče nalazi se iza opisa instrukcije. To znači da instrukcija ili postavlja, ili briše fleg čija je oznaka u spisku, u zavisnosti od rezultata operacije.

↑ iza oznake znači da instrukcija postavlja fleg u zavisnosti od rezultata operacije ali ga nikad ne briše.

↓ iza oznake znači da instrukcija briše fleg u zavisnosti od rezultata operacije ali ga nikad ne postavlja.

=0 iza oznake znači da instrukcija briše fleg

=1 iza oznake znači da instrukcija briše fleg

? iza oznake znači da je stanje flega posle instrukcije nedefinisano ili da je ponašanje instrukcije u odnosu na taj fleg specifično

Zbir dva operanda tipa REČ se smešta u izlazni registar (prvi operand).

ADD DSwreg, Swaop

Uticaj na flegove **Z, N, C, V, VT↑**

****

ADD (tri operanda) - saberi REČi

Zbir drugog i trećeg REČ operanda se smešta u izlazni registar (prvi operand).

ADD Dwreg, S1wreg, S2waop

Uticaj na flegove **Z, N, C, V, VT↑**

****

ADDB (dva operanda) - saberi BAJTe

Zbir dva operanda tipa BAJT se smešta u izlazni registar (prvi operand).

ADDB DSbreg, Sbaop

Uticaj na flegove **Z, N, C, V, VT↑**

****

ADDB (tri operanda) - saberi BAJTe

Zbir drugog i trećeg BAJT operanda se smešta u izlazni registar (prvi operand).

ADDB Dbreg, S1breg, S2baop

Uticaj na flegove **Z, N, C, V, VT↑**

****

ADDC - saberi REČi sa prenosom

Zbir dva operanda tipa REČ i flega prenosa (0 ili 1) se smešta u izlazni registar (prvi operand).

ADDC DSwreg, Swaop

Uticaj na flegove **Z↓, N, C, V, VT↑**

****

ADDCB - saberi BAJTe sa prenosom

Zbir dva operanda tipa REČ i flega prenosa (0 ili 1) se smešta u izlazni registar (prvi operand).

ADDCB DSwreg, Swaop

Uticaj na flegove **Z↓, N, C, V, VT↑**

****

AND (dva operanda) - logička I operacija nad REČima

Nad dva operanda tipa REČ se izvršava logička I opereacija, bit po bit. Rezultat ima logičku jedinicu samo na onim pozicijama na kojima oba operanda imaju jedinicu. Na svim ostalim pozicijama, dobijaju se nule. Rezultat se smešta u izlazni registar (prvi operand).

AND DSwreg, Swaop

Uticaj na flegove **Z, N, C=0, V=0**

****

AND (tri operanda) - logička I operacija nad REČima

Nad drugim i trećim operandom tipa REČ izvršava se logička I opereacija, bit po bit. Rezultat ima logičku jedinicu samo na onim pozicijama na kojima oba operanda imaju jedinicu. Na svim ostalim pozicijama, dobijaju se nule. Rezultat se smešta u izlazni registar (prvi operand).

AND Dwreg, S1wreg, S2waop

Uticaj na flegove **Z, N, C=0, V=0**

****

ANDB (dva operanda) - logička I operacija nad BAJTima

Nad dva operanda tipa BAJT se izvršava logička I opereacija, bit po bit. Rezultat ima logičku jedinicu samo na onim pozicijama na kojima oba operanda imaju jedinicu. Na svim ostalim pozicijama, dobijaju se nule. Rezultat se smešta u izlazni registar (prvi operand).

ANDB DSbreg, Sbaop

Uticaj na flegove **Z, N, C=0, V=0**

****

ANDB (tri operanda) - logička I operacija nad BAJTima

Nad drugim i trećim operandom tipa BAJT izvršava se logička I opereacija, bit po bit. Rezultat ima logičku jedinicu samo na onim pozicijama na kojima oba operanda imaju jedinicu. Na svim ostalim pozicijama, dobijaju se nule. Rezultat se smešta u izlazni registar (prvi operand).

ANDB Dbreg, S1breg, S2baop

Uticaj na flegove **Z, N, C=0, V=0**

****

BMOV - premeštanje bloka podataka

Ova instrukcija se koristi da premesti blok REČ podataka sa jednog mesta u memoriji na drugo. LONG (lreg) sadrži dva pokazivača (dve REČi). Prvi pokazivač (na nižoj adresi) ukazuje na mesto sa koga se uzimaju podaci, a drugi (na višoj adresi), na mesto gde se podaci smeštaju. REČ (wreg) sadrži broj REČi koje treba preneti. Blokovi mogu biti bilo gde u memorijskom prostoru ali se ne smeju preklapati. Instrukcija se ne može prekinuti interaptom dok se ne završi!

BMOV lreg, wreg

lreg: niža reč - pokazivač na ulazne podatke
viša reč - pokazivač na mesto gde se podaci smeštaju
wreg: broj podataka za prenos.

Uticaj na flegove -

****

BR - bezuslovni skok

Program skače na početnu labelu koja je operand u instrukciji. Insrukcija BR objedinjue LJMP i SJMP instrukciju. Ona ne postoji u instrukcijskom setu samog mikroračunara već je asembler zamenjuje ili SJMP ili LJMP instrukcijom zavisno od udaljenosti labele.

BR labela

Uticaj na flegove -

****

BR (indirektno) - indirektni skok

Posle ove instrukcije program se nastavlja od adrese koju sadrži operand (REČ registar).

BR [wreg]

Uticaj na flegove -

****

CALL - poziv potprograma

Trenutna vrednost programskog brojača se smešta na stek, pokazivač steka (registar 18h) se umanjuje za 2, a program skače na početnu adresu potprograma (labelu koja je operand u instrukciji). Insrukcija CALL objedinjue LCALL i SCALL instrukciju. Ona ne postoji u instrukcijskom setu samog mikroračunara već je asembler zamenjuje ili SCALL ili LCALL instrukcijom zavisno od udaljenosti labele

CALL labela

Uticaj na flegove -

****

CLR - obriši REČ

Vrednost operanda (REČ registar) postaje nula.

CLR wreg

Uticaj na flegove **Z=1, N=0, C=0, V=0**

****

CLRB - obriši BAJT

Vrednost operanda (BAJT registar) postaje nula.

CLR breg

Uticaj na flegove **Z=1, N=0, C=0, V=0**

****

CLRC - obriši fleg prenosa

Vrednost flega prenosa (C fleg) postaje nula.

CLRC

Uticaj na flegove **C=0**

****

CLRVT - obriši VT fleg

Vrednost trajnog flega prekoračenja (VT fleg) postaje nula.

CLRVT

Uticaj na flegove **VT=0**

****

CMP - uporedi REČi

Drugi REČ operand se oduzima od prvog REČ operanda i na osnovu rezultata postavljaju se flegovi u PSW. Rezultat oduzimanja ostaje u internom registru i nije dostupan korisniku. Fleg prenosa (C) se briše ako ima pozajmice prilikom oduzimanja, a postavlja ako pozajmice nema.

CMP S1wreg, S2waop

Flegovi se postavljaju na osnovu rezultata (S1 - S2).

Uticaj na flegove **Z, N, C, V, VT↑**

****

CMPB - uporedi BAJTe

Drugi BAJT operand se oduzima od prvog BAJT operanda i na osnovu rezultata postavljaju se flegovi u PSW. Rezultat oduzimanja ostaje u internom registru i nije dostupan korisniku. Fleg prenosa (C) se briše ako ima pozajmice prilikom oduzimanja, a postavlja ako pozajmice nema.

CMPB S1breg, S2baop

Flegovi se postavljaju na osnovu rezultata (S1 - S2).

Uticaj na flegove **Z, N, C, V, VT↑**

****

CMPL - uporedi LONGove

Drugi LONG operand se oduzima od prvog LONG operanda i na osnovu rezultata postavljaju se flegovi u PSW. Rezultat oduzimanja ostaje u internom registru i nije dostupan korisniku. Fleg prenosa (C) se briše ako ima pozajmice prilikom oduzimanja, a postavlja ako pozajmice nema.

CMPL S1lreg, S2lreg

Flegovi se postavljaju na osnovu rezultata (S1 - S2).
Oba operanda moraju biti **registri**. Može im se pristupiti jedino **registarskim direktnim** adresiranjem!

Uticaj na flegove **Z, N, C, V, VT↑**

****

DEC - umanji REČ

Vrednost REČ operanda se umanjuje za 1.

DEC DSwreg

Uticaj na flegove **Z, N, C, V, VT↑**

****

DECB - umanji BAJT

Vrednost BAJT operanda se umanjuje za 1.

DECB DSbreg

Uticaj na flegove **Z, N, C, V, VT↑**

****

DI - onemogući interapte

Onemogućuje sve interapte postavljajući I fleg (PSW.9) na nulu.

DI

Uticaj na flegove **I=0**

****

DIV - podeli označeni LONG označenom REČi

Deli označeni LONG (prvi operand) označenom REČi (drugi operand). Deljenje je celobrojno sa otsecanjem. Rezultat se smešta u nižu reč prvog operanda (reč na nižoj adresi), dok se u višu reč prvog operanda (reč na višoj adresi) smešta ostatak deljenja.

DIV DSlreg, Swaop

Niža reč DS postaje DS/S (celobrojni rezultat deljenja).
Viša reč DS postaje DS MOD S (ostatak deljenja).
Utiče samo na V i VT fleg !

Uticaj na flegove **V, VT↑**
****

DIVB - podeli označenu REČ označenim BAJTom

Deli označenu REČ (prvi operand) sa označenim BAJTom (drugi operand). Deljenje je celobrojno sa otsecanjem. Rezultat se smešta u niži bajt prvog operanda (bajt na nižoj adresi), dok se u viši bajt prvog operanda (bajt na višoj adresi) smešta ostatak deljenja.

DIVB DSwreg, Sbaop

Niži bajt DS postaje DS/S (celobrojni rezultat deljenja).
Viši bajt DS postaje DS MOD S (ostatak deljenja).
Utiče samo na V i VT fleg !

Uticaj na flegove **V, VT↑**
****

DIVU - podeli neoznačeni LONG neoznačenom REČi

Deli neoznačeni LONG (prvi operand) sa neoznačenom REČi (drugi operand). Deljenje je celobrojno sa otsecanjem. Rezultat se smešta u nižu reč prvog operanda (reč na nižoj adresi), dok se u višu reč prvog operanda (reč na višoj adresi) smešta ostatak deljenja.

DIVU DSlreg, Swaop

Niža reč DS postaje DS/S (celobrojni rezultat deljenja).
Viša reč DS postaje DS MOD S (ostatak deljenja).

Uticaj na flegove **V, VT↑**
****

DIVUB - podeli neoznačenu REČ neoznačenim BAJTom

Deli neoznačenu REČ (prvi operand) sa neoznačenim BAJTom (drugi operand). Deljenje je celobrojno sa otsecanjem. Rezultat se smešta u niži bajt prvog operanda (bajt na nižoj adresi), dok se u viši bajt prvog operanda (bajt na višoj adresi) smešta ostatak deljenja.

DIVUB DSwreg, Sbaop

Niži bajt DS postaje DS/S (celobrojni rezultat deljenja).
Viši bajt DS postaje DS MOD S (ostatak deljenja).

Uticaj na flegove **V, VT↑**

****

DJNZ (i DBNZ) - umanji BAJT i skoči ako nije nula

Umanjuje vrednost prvog BAJT operanda i ako rezultat nije nula skače ne labelu (drugi operand). Ako je rezultat nula, program se nastavlja izvršavanjm naredne instrukcije. Labela ne sme da bude više od 128 bajtova pre niti više od 127 bajtova posle DJNZ instrukcije. Ovo ograničenje ne važi za DBNZ instrukciju koja ne postoji u instrukcijskom setu samog mikroprocesora već je asembler, zavisno od udaljenosti labele, prevodi ili kao DJNZ ili kao nekoliko instrukcija sa istim efektom.

DBNZ breg, labela

Uticaj na flegove -

****

DJNZW (i DBNZW) - umanji REČ i skoči ako nije nula

Umanjuje vrednost prvog REČ operanda i ako rezultat nije nula skače ne labelu (drugi operand). Ako je rezultat nula, program se nastavlja izvršavanjm naredne instrukcije. Labela ne sme da bude više od 128 bajtova pre niti više od 127 bajtova posle DJNZW instrukcije. Ovo ograničenje ne važi za DBNZW instrukciju koja ne postoji u instrukcijskom setu samog mikroprocesora već je asembler, zavisno od udaljenosti labele, prevodi ili kao DJNZW ili kao nekoliko instrukcija sa istim efektom.

DBNZW wreg, labela

Uticaj na flegove -

****

EI - omogući interapte

Omogućuje interapte postavljajući I fleg (PSW.9) na jedinicu.

EI

Uticaj na flegove **I=1**

****

EXT - pretvara označenu REČ u označeni LONG

Ako je u reči na nižoj adresi operanda negativan broj (naznačajniji bit je 1), viša reč postaje 0FFFFh, u suprotnom (ako je najznačajniji bit niže reči operanda nula), viša reč postaje 0000. REČ koja treba da se pretvori u LONG, upisuje se u nižu reč LONG registra, a instrukcija EXT upiše odgovarajuću vrednost u višu reč.

EXT lreg

Uticaj na flegove **Z, N, C=0, V=0**

****

EXTB - pretvara označeni BAJT u označenu REČ

Ako je u bajtu na nižoj adresi operanda negativan broj (naznačajniji bit je 1), viši bat postaje 0FFh, u suprotnom (ako je najznačajniji bit nižeg bajta operanda nula), viši bajt postaje 00. BAJT koja treba da se pretvori u REČ, upisuje se u niži bajt REČ registra, a instrukcija EXTB upiše odgovarajuću vrednost u viši bajt.

EXTB wreg

Uticaj na flegove **Z, N, C=0, V=0**

****

INC - uvećaj REČ

Vrednost REČ operanda se uvećava za 1.

INC DSwreg

Uticaj na flegove **Z, N, C, V, VT↑**

****

INCB - uvećj BAJT

Vrednost BAJT operanda se uvećava za 1.

INCB DSbreg

Uticaj na flegove **Z, N, C, V, VT↑**

****

IDLPD - pređi u režim male potrošnje

Mikrokontroler prelazi u režim čekanja (*idle*) ako je operand 1 ili u režim male potrošnje (*powerdown*) ako je operand 2. Za sve drugi vrednosti operanda, mikrokontroler se resetuje.

IDLPD #broj

Uticaj na flegove ne utiče, osim u slučaju da izazove reset, tada se svi brišu

****

JBC (i BBC) - skoči ako je bit nula

Ako je bit čiji redni broj (0 do 7) definiše dugi operand u BAJTu (prvi operand) nula, skače ne labelu (treći operand). Ako taj bit nije nula, program se nastavlja izvršavanjm naredne instrukcije. Labela ne sme da bude više od 128 bajtova pre, niti više od 127 bajtova posle JBC instrukcije. Ovo ograničenje ne važi za BBC instrukciju koja ne postoji u instrukcijskom setu samog mikroprocesora već je asembler, zavisno od udaljenosti labele, prevodi ili kao BBC ili kao nekoliko instrukcija sa istim efektom. Takođe je u BBC instrukciji dozvoljeno da broj bita bude 0 do 31 a asembler će taj broj svesti na vrednost 0 do 7 i sam povećati adresu BAJTA u prvom operandu, ako je potrebno. Na taj način instrukcija se, sa gledišta programera, može posmatrati kao da je prvi operand bilo BAJT bilo REČ bilo LONG, a br_bita 0 do 31.

BBC breg, br_bita, labela

Br_bita 0 do 31. Ako je br_bita>7, br_bita postaje (br_bita MOD 8), a breg postaje (breg+br_bita/8). Na primer, ako je br_bita 9, asembler će povećati breg za 1 (9/8), a za br_bita uzeti 1 umesto 9 (1 je ostatak celobrojnog deljenja 9/8). Tako se pristupa bitu 1 u višem bajtu REČi na adresi u breg (prvi bajt do najmanje značajnog).

Uticaj na flegove -

****

JBS (i BBS) - skoči ako je bit jedinica

Ako je bit čiji redni broj (0 do 7) definiše dugi operand u BAJTu (prvi operand) jedinica, skače ne labelu (treći operand). Ako je taj bit nula, program se nastavlja izvršavanjm naredne instrukcije. Labela ne sme da bude više od 128 bajtova pre, niti više od 127 bajtova posle JBS instrukcije. Ovo ograničenje ne važi za BBS instrukciju koja ne postoji u instrukcijskom setu samog mikroprocesora već je asembler, zavisno od udaljenosti labele, prevodi ili kao BBS ili kao nekoliko instrukcija sa istim efektom. Takođe je u BBS instrukciji dozvoljeno da broj bita bude 0 do 31 a asembler će taj broj svesti na vrednost 0 do 7 i sam povećati adresu BAJTA u prvom operandu, ako je potrebno. Na taj način instrukcija se, sa gledišta programera, može posmatrati kao da je prvi operand LONG, a br_bita 0 do 31.

BBS breg, br_bita, labela

Br_bita 0 do 31. Ako je br_bita>7, br_bita postaje (br_bita MOD 8), a breg postaje (breg+br_bita/8). Na primer, ako je br_bita 21, asembler će povećati breg za 2 (21/8), a za br_bita uzeti 5 umesto 21 (5 je ostatak celobrojnog deljenja 21/8). Tako će se pristupiti bitu 5 trećeg bajta longa na adresi breg, što jeste bit 21 tog longa.

Uticaj na flegove -

****

JC (i BC) - skoči ako je fleg prenosa (C) jedinica

Ako je C fleg setovan (jedinica) skače ne labelu (operand). Ako je ovaj fleg nula, program se nastavlja izvršavanjm naredne instrukcije. Labela ne sme da bude više od 128 bajtova pre niti više od 127 bajtova posle JC instrukcije. Ovo ograničenje ne važi za BC instrukciju koja ne postoji u instrukcijskom setu samog mikroprocesora već je asembler, zavisno od udaljenosti labele, prevodi ili kao JC ili kao nekoliko instrukcija sa istim efektom.

JC labela

Instrukcija se može koristiti i za grananje posle poređenja (CMP) neoznaženih veličina. (BC labela) iza CMP U,V će preusmeriti izvršavanje programa na labelu ako ja U veće ili jednako V, gde se U i V tretiraju kao neoznačene veličine.

Uticaj na flegove -

****

JE (i BE) - skoči ako je fleg nule (Z) jedinica

Ako je Z fleg setovan (jedinica) skače ne labelu (operand). Ako je ovaj fleg nula, program se nastavlja izvršavanjm naredne instrukcije. Labela ne sme da bude više od 128 bajtova pre niti više od 127 bajtova posle JE instrukcije. Ovo ograničenje ne važi za BE instrukciju koja ne postoji u instrukcijskom setu samog mikroprocesora već je asembler, zavisno od udaljenosti labele, prevodi ili kao JE ili kao nekoliko instrukcija sa istim efektom.

JE labela

Instrukcija se može koristiti i za grananje posle poređenja (CMP) bilo neoznaženih, bilo označenih veličina. (BE labela) iza CMP A,B će preusmeriti izvršavanje programa na labelu ako ja A = B, gde A i B mogu biti i označene i neoznačene veličine.

Uticaj na flegove -

****

JGE (i BGE) - skoči ako je veće ili jednako (OZNAČENO)

Ako je N fleg nula skače ne labelu (operand). Ako je ovaj fleg setovan (jedinica), program se nastavlja izvršavanjm naredne instrukcije. Labela ne sme da bude više od 128 bajtova pre niti više od 127 bajtova posle JGE instrukcije. Ovo ograničenje ne važi za BGE instrukciju koja ne postoji u instrukcijskom setu samog mikroprocesora već je asembler, zavisno od udaljenosti labele, prevodi ili kao JGE ili kao nekoliko instrukcija sa istim efektom.

BGE labela

Instrukcija se koristi za grananje posle poređenja (CMP) označenih veličina.(BGE labela) iza CMP A,B će preusmeriti izvršavanje programa na labelu ako ja A veće ili jednako B, gde se A i B tretiraju kao označene veličine.

Uticaj na flegove -

****

JGT (i BGT) - skoči ako je veće (OZNAČENO)

Ako su i N fleg i Z fleg nule skače ne labelu (operand). Ako je bilo koji od flegova setovan (jedinica), program se nastavlja izvršavanjm naredne instrukcije. Labela ne sme da bude više od 128 bajtova pre niti više od 127 bajtova posle JGT instrukcije. Ovo ograničenje ne važi za BGT instrukciju koja ne postoji u instrukcijskom setu samog mikroprocesora već je asembler, zavisno od udaljenosti labele, prevodi ili kao JGT ili kao nekoliko instrukcija sa istim efektom.

BGT labela

Instrukcija se koristi za grananje posle poređenja (CMP) označenih veličina. (BGT labela) iza CMP A,B će preusmeriti izvršavanje programa na labelu ako ja A>B, gde se A i B tretiraju kao označene veličine.

Uticaj na flegove -

****

JH (i BH) - skoči ako je veće (NEOZNAČENO)

Ako je C fleg setovan (jedinica), a Z fleg nula skače ne labelu (operand). Ako je Z fleg jedinica **ili** C fleg nula program se nastavlja izvršavanjm naredne instrukcije. Labela ne sme da bude više od 128 bajtova pre niti više od 127 bajtova posle JH instrukcije. Ovo ograničenje ne važi za BH instrukciju koja ne postoji u instrukcijskom setu samog mikroprocesora već je asembler, zavisno od udaljenosti labele, prevodi ili kao JH ili kao nekoliko instrukcija sa istim efektom.

BH labela

Instrukcija se koristi za grananje posle poređenja (CMP) neoznaženih veličina. (BH labela) iza CMP U,V će preusmeriti izvršavanje programa na labelu ako ja U>V, gde se U i V tretiraju kao neoznačene veličine.

Uticaj na flegove -

****

JLE (i BLE) - skoči ako je manje ili jednako (OZNAČENO)

Ako je setovan bilo N fleg bilo Z fleg, skače ne labelu (operand). Ako su oba flega nule, program se nastavlja izvršavanjm naredne instrukcije. Labela ne sme da bude više od 128 bajtova pre niti više od 127 bajtova posle JLE instrukcije. Ovo ograničenje ne važi za BLE instrukciju koja ne postoji u instrukcijskom setu samog mikroprocesora već je asembler, zavisno od udaljenosti labele, prevodi ili kao JLE ili kao nekoliko instrukcija sa istim efektom.

BLE labela

Instrukcija se koristi za grananje posle poređenja (CMP) označenih veličina. (BLE labela) iza CMP A,B će preusmeriti izvršavanje programa na labelu ako ja A manje ili jednako B, gde se A i B tretiraju kao označene veličine.

Uticaj na flegove -

****

JLT (i BLT) - skoči ako je manje (OZNAČENO)

Ako je N fleg setovan (jedinica), skače ne labelu (operand). Ako je ovaj fleg nula, program se nastavlja izvršavanjm naredne instrukcije. Labela ne sme da bude više od 128 bajtova pre niti više od 127 bajtova posle JLT instrukcije. Ovo ograničenje ne važi za BLT instrukciju koja ne postoji u instrukcijskom setu samog mikroprocesora već je asembler, zavisno od udaljenosti labele, prevodi ili kao JLT ili kao nekoliko instrukcija sa istim efektom.

BLT labela

Instrukcija se koristi za grananje posle poređenja (CMP) označenih veličina.(BLT labela) iza CMP A,B će preusmeriti izvršavanje programa na labelu ako ja A < B, gde se A i B tretiraju kao označene veličine.

Uticaj na flegove -

****

JNC (i BNC) - skoči ako je fleg prenosa (C) nula

Ako je C fleg nula skače ne labelu (operand). Ako je ovaj fleg setovan (jedinica), program se nastavlja izvršavanjm naredne instrukcije. Labela ne sme da bude više od 128 bajtova pre niti više od 127 bajtova posle JNC instrukcije. Ovo ograničenje ne važi za BNC instrukciju koja ne postoji u instrukcijskom setu samog mikroprocesora već je asembler, zavisno od udaljenosti labele, prevodi ili kao JNC ili kao nekoliko instrukcija sa istim efektom.

BNC labela

Instrukcija se može koristiti i za grananje posle poređenja (CMP) neoznaženih veličina. (BNC labela) iza CMP U,V će preusmeriti izvršavanje programa na labelu ako ja U<V, gde se U i V tretiraju kao neoznačene veličine.

Uticaj na flegove -

****

JNE (i BNE) - skoči ako je fleg nule (Z) nula

Ako je Z fleg nula skače ne labelu (operand). Ako je ovaj fleg setovan (jedinica), program se nastavlja izvršavanjm naredne instrukcije. Labela ne sme da bude više od 128 bajtova pre niti više od 127 bajtova posle JNE instrukcije. Ovo ograničenje ne važi za BNE instrukciju koja ne postoji u instrukcijskom setu samog mikroprocesora već je asembler, zavisno od udaljenosti labele, prevodi ili kao JNE ili kao nekoliko instrukcija sa istim efektom.

BNE labela

Instrukcija se može koristiti i za grananje posle poređenja (CMP) bilo neoznaženih, bilo označenih veličina. (BNE labela) iza CMP A,B će preusmeriti izvršavanje programa na labelu ako ja A različito od B, gde A i B mogu biti i označene i neoznačene veličine.

Uticaj na flegove -

****

JNH (i BNH) - skoči ako je manje ili jednako (NEOZNAČENO)

Ako je Z fleg setovan (jedinica) **ili** C fleg nula, skače ne labelu (operand). Ako je Z fleg nula **i** C fleg jedinica, program se nastavlja izvršavanjm naredne instrukcije. Labela ne sme da bude više od 128 bajtova pre niti više od 127 bajtova posle JNH instrukcije. Ovo ograničenje ne važi za BNH instrukciju koja ne postoji u instrukcijskom setu samog mikroprocesora već je asembler, zavisno od udaljenosti labele, prevodi ili kao JNH ili kao nekoliko instrukcija sa istim efektom.

JNH labela

Instrukcija se koristi za grananje posle poređenja (CMP) neoznaženih veličina. (BNH labela) iza CMP U,V će preusmeriti izvršavanje programa na labelu ako ja U manje ili jednako V, gde se U i V tretiraju kao označene veličine.

Uticaj na flegove -

****

JNST (i BNST) - skoči ako je ST fleg prenosa nula

Ako je ST fleg nula skače ne labelu (operand). Ako je ovaj fleg jedinica, program se nastavlja izvršavanjm naredne instrukcije. Labela ne sme da bude više od 128 bajtova pre niti više od 127 bajtova posle JNST instrukcije. Ovo ograničenje ne važi za BNST instrukciju koja ne postoji u instrukcijskom setu samog mikroprocesora već je asembler, zavisno od udaljenosti labele, prevodi ili kao JNST ili kao nekoliko instrukcija sa istim efektom.

JNST labela

Uticaj na flegove -

****

JNV (i BNV) - skoči ako je fleg prekoračenja (V) nula

Ako je V fleg nula skače ne labelu (operand). Ako je ovaj fleg jedinica, program se nastavlja izvršavanjm naredne instrukcije. Labela ne sme da bude više od 128 bajtova pre niti više od 127 bajtova posle JNV instrukcije. Ovo ograničenje ne važi za BNV instrukciju koja ne postoji u instrukcijskom setu samog mikroprocesora već je asembler, zavisno od udaljenosti labele, prevodi ili kao JNV ili kao nekoliko instrukcija sa istim efektom.

BNV labela

Uticaj na flegove -

****

JNVT (i BNVT) - skoči ako je VT fleg nula

Ako je trajni fleg prekoračenja (VT fleg) nula, skače ne labelu (operand). Ako je ovaj fleg jedinica, program se nastavlja izvršavanjm naredne instrukcije. Labela ne sme da bude više od 128 bajtova pre niti više od 127 bajtova posle JNVT instrukcije. Ovo ograničenje ne važi za BNVT instrukciju koja ne postoji u instrukcijskom setu samog mikroprocesora već je asembler, zavisno od udaljenosti labele, prevodi ili kao JNVT ili kao nekoliko instrukcija sa istim efektom.

JNVT labela

Uticaj na flegove **VT=0**

****

JST (i BST) - skoči ako je ST fleg prenosa jedinica

Ako je ST fleg jedinica skače ne labelu (operand). Ako je ovaj fleg nula, program se nastavlja izvršavanjm naredne instrukcije. Labela ne sme da bude više od 128 bajtova pre niti više od 127 bajtova posle JST instrukcije. Ovo ograničenje ne važi za BST instrukciju koja ne postoji u instrukcijskom setu samog mikroprocesora već je asembler, zavisno od udaljenosti labele, prevodi ili kao JST ili kao nekoliko instrukcija sa istim efektom.

BST labela

Uticaj na flegove -

****

JV (i BV) - skoči ako je fleg prekoračenja (V) jedinica

Ako je V fleg setovan (jedinica) skače ne labelu (operand). Ako je ovaj fleg nula program se nastavlja izvršavanjm naredne instrukcije. Labela ne sme da bude više od 128 bajtova pre niti više od 127 bajtova posle JV instrukcije. Ovo ograničenje ne važi za BV instrukciju koja ne postoji u instrukcijskom setu samog mikroprocesora već je asembler, zavisno od udaljenosti labele, prevodi ili kao JV ili kao nekoliko instrukcija sa istim efektom.

BV labela

Uticaj na flegove -

****

JVT (i BVT) - skoči ako je VT fleg jedinica

Ako je trajni fleg prekoračenja (VT fleg) setovan (jedinica), skače ne labelu (operand). Ako je ovaj fleg nula program se nastavlja izvršavanjm naredne instrukcije. Labela ne sme da bude više od 128 bajtova pre niti više od 127 bajtova posle JVT instrukcije. Ovo ograničenje ne važi za BVT instrukciju koja ne postoji u instrukcijskom setu samog mikroprocesora već je asembler, zavisno od udaljenosti labele, prevodi ili kao JVT ili kao nekoliko instrukcija sa istim efektom.

JVT labela

Uticaj na flegove **VT=0**

****

LCALL - poziv dalekog potprograma

Trenutna vrednost programskog brojača se smešta na stek, pokazivač steka (registar 18h) se umanjuje za 2, a program skače na početnu adresu potprograma (labelu koja je operand u instrukciji). Insrukcija CALL objedinjue ovu i SCALL instrukciju.

LCALL labela

Uticaj na flegove -

****

LD - prebaci REČ u registar

Vrednost drugog REČ operanda se prebacuje u REČ registar (prvi operand).

LD Dwreg, Swaop

Uticaj na flegove -

****

LDB - prebaci BAJT u registar

Vrednost drugog BAJT operanda se prebacuje u BAJT registar (prvi operand).

LDB Dbreg, Sbaop

Uticaj na flegove -

****

LDBSE - prevedi označeni BAJT u označenu REČ

Vrednost drugog BAJT operanda se prebacuje u niži bajt REČ registra (prvi operand). Viši bajt (na višoj adresi) postaje 00 ako je prebačeni broj pozitivan, a 0FFh ako je broj negativan. Broj je negativan ako mu je najznačajniji bit jedinica, nezavisno od toga kako ga operator tretira (da li kao označen ili neoznačen bajt).

LDBSE Dwreg, Sbaop

Uticaj na flegove -

****

LDBZE - prevedi neoznačeni BAJT u neoznačenu REČ

Vrednost drugog BAJT operanda se prebacuje u niži bajt REČ registra (prvi operand). Viši bajt (na višoj adresi) postaje 00.

LDBZE Dwreg, Sbaop

Uticaj na flegove -

****

LJMP - bezuslovni daleki skok

Program skače na labelu koja je operand u instrukciji. Labela može biti bilo gde u adresnom prostoru. Postoji i insrukcija BR koja objedinjue LJMP i SJMP. NJe nema u instrukcijskom setu samog mikroračunara već je asembler zamenjuje ili SJMP ili LJMP instrukcijom zavisno od udaljenosti labele.

LJMP labela

Uticaj na flegove -

****

MUL (dva operanda) - pomnoži označene REČi

Proizvod označene reči na nižoj adresi prvog LONG operanda i označene REČi (drugi operand) smešta se u LONG registar koji je prvi operand u instrukciji.

MUL DSlreg, Swaop

Long DS postaje S*DS, gde su S i DS označene REČi.

Uticaj na flegove **ST?**

****

MUL (tri operanda) - pomnoži označene REČi

Proizvod označene REČi (drugi operand) i označene REČi (treći operand) smešta se u LONG registar koji je prvi operand u instrukciji.

MUL Dlreg, S1wreg, S2waop

Long D postaje S1*S2, gde su S1 i S2 označene REČi.

Uticaj na flegove **ST?**

****

MULB (dva operanda) - pomnoži označene BAJTOVE

Proizvod označenog bajta na nižoj adresi prvog REČ operanda i označenog BAJTa (drugi operand) smešta se u REČ registar koji je prvi operand u instrukciji.

MULB DSwreg, Sbaop

REČ DS postaje S*DS, gde su S i DS označeni BAJTOVI.

Uticaj na flegove **ST?**

****

MULB (tri operanda) - pomnoži označene BAJTOVE

Proizvod označenog BAJTa (drugi operand) i označenog BAJTa (treći operand) smešta se u REČ registar koji je prvi operand u instrukciji.

MULB Dwreg, S1breg, S2baop

REČ D postaje S1*S2, gde su S1 i S2 označeni BAJTOVI.

Uticaj na flegove **ST?**

MULU (dva operanda) - pomnoži neoznačene REČi

Proizvod neoznačene reči na nižoj adresi prvog LONG operanda i neoznačene REČi (drugi operand) smešta se u LONG registar koji je prvi operand u instrukciji.

MULU DSlreg, Swaop

Long DS postaje S*DS, gde su S i DS neoznačene REČi.

Uticaj na flegove **ST?**

****

MULU (tri operanda) - pomnoži neoznačene REČi

Proizvod neoznačene REČi (drugi operand) i neoznačene REČi (treći operand) smešta se u LONG registar koji je prvi operand u instrukciji.

MULU Dlreg, S1wreg, S2waop

Long D postaje S1*S2, gde su S1 i S2 neoznačene REČi.

Uticaj na flegove **ST?**

****

MULUB (dva operanda) - pomnoži neoznačene BAJTOVE

Proizvod neoznačenog bajta na nižoj adresi prvog REČ operanda i neoznačenog BAJTa (drugi operand) smešta se u REČ registar koji je prvi operand u instrukciji.

MULUB DSwreg, Sbaop

REČ DS postaje S*DS, gde su S i DS neoznačeni BAJTOVI.

Uticaj na flegove **ST?**

****

MULUB (tri operanda) - pomnoži neoznačene BAJTOVE

Proizvod neoznačenog BAJTa (drugi operand) i neoznačenog BAJTa (treći operand) smešta se u REČ registar koji je prvi operand u instrukciji.

MULUB Dwreg, S1breg, S2baop

REČ D postaje S1*S2, gde su S1 i S2 neoznačeni BAJTOVI.

Uticaj na flegove **ST?**

****

NEG - promeni znak označenoj REČi

Vrednost REČ registra koji je operand u instrukciji se negira. Podrazumeva se da je REČ označena.

NEG DSwreg

Uticaj na flegove **Z, N, C, V, VT↑**

****

NEGB - promeni znak označenom BAJTu

Vrednost BAJT registra koji je operand u instrukciji se negira. Podrazumeva se da je BAJT označen.

NEG DSbreg

Uticaj na flegove **Z, N, C, V, VT↑**

****

NOP - bez operacije

Ne radi ništa. Program se nastavlja sledećom insrukcijom.

NOP

Uticaj na flegove **-**

****

NORML - normalizuj označeni LONG

Označeni long (prvi operand) se šiftuje ulevo sve dok najznačajniji bit ne postane 1. Ako je najznačajniji bit nula i posle 31 pomeraja, šiftovanje se prekida i setuje se Z fleg. Broj pomeraja se smešta u BAJT registar koji je drugi operand u instrukciji.

NORML DSlreg, breg

Uticaj na flegove **Z, N?, C=0**

****

NOT - prvi komplement REČi

Vrednost REČ registra (operand) se komplementira. Bit po bit svaka nula se zamenjuje jedinicom i obrnuto.

NOT DSwreg

Uticaj na flegove **Z, N, C=0, V=0**

****

NOTB - prvi komplement BAJTa

Vrednost BAJT registra (operand) se komplementira. Bit po bit svaka nula se zamenjuje jedinicom i obrnuto.

NOT DSbreg

Uticaj na flegove **Z, N, C=0, V=0**

****

OR - logička ILI operacija nad REČima

Nad dva operanda tipa REČ se izvršava logička ILI opereacija, bit po bit. Rezultat ima logičku nulu samo na onim pozicijama na kojima oba operanda imaju nulu. Na svim ostalim pozicijama, na kojima je makar u jednom operandu jedinica, dobijaju se jedinice. Rezultat se smešta u izlazni registar (prvi operand).

OR DSwreg, Swaop

Uticaj na flegove **Z, N, C=0, V=0**

****

ORB - logička ILI operacija nad BAJTima

Nad dva operanda tipa BAJT se izvršava logička ILI opereacija, bit po bit. Rezultat ima logičku nulu samo na onim pozicijama na kojima oba operanda imaju nulu. Na svim ostalim pozicijama, na kojima je makar u jednom operandu jedinica, dobijaju se jedinice. Rezultat se smešta u izlazni BAJT registar (prvi operand).

ORB DSbreg, Sbaop

Uticaj na flegove **Z, N, C=0, V=0**

****

POP - prebaci REČ sa vrha steka

REČ sa vrha steka (poslednja koja je upisana na stek) se prebacuje u REČ koja je operand u instrukciji. REČ u koju se upisuje može biti bilo gde u adresnom prostoru. Pokazivač steka (registar na 18h) se uvećava za 2.

POP Dwaop

Uticaj na flegove **-**

****

POPA - prebaci sa vrha steka flegove i sve maske prekida

Sa vrha steka se očitavaju flegovi, WSR i sve interapt maske. Pokazivač steka (registar na 18h) se uvećava za 4. Interapt je hadverski onemogućen posle ove instrukcije sve do kompletiranja sledeće. Ovo znači da se garantuje izvršenje i naredbe koja sledi, pre bilo kog interapta. Time se izbegava mogućnost da se interapt desi između POPA i RET instrukcije.

POPA

Instrukcija se koristi kada su PSW, WSR i preostale interapt maske ubačene na stek pomoću PUSHA instrukcije.

Uticaj na flegove **Z, N, C, V, VT, ST**

****

POPF - prebaci sa vrha steka flegove i maske prekida

Sa vrha steka se očitavaju flegovi, i interapt maske iz PSW registra. Pokazivač steka (registar na 18h) se uvećava za 2. Interapt je hadverski onemogućen posle ove instrukcije sve do kompletiranja sledeće. Ovo znači da se garantuje izvršenje i naredbe koja sledi, pre bilo kog interapta. Time se izbegava mogućnost da se interapt desi između POPF i RET instrukcije.

POPF

Instrukcija se koristi kada je PSW registar ubačen na stek pomoću PUSHF instrukcije

Uticaj na flegove **Z, N, C, V, VT, ST**

****

PUSH - prebaci REČ na stek

Pokazivač steka se umanjuje za 2. REČ operand se ubacuje na vrh steka (u lokaciju na koju ukazuje pokazivač steka posle umanjivanja).

PUSH Swaop

Uticaj na flegove -

****

PUSHA - ubaci na stek PSW, WSR i sve interapt maske

Instrukcija ubacuje na stek PSW registar, WSR i preostale interapt maske. Pokazivač steka (registar 18h) se umanjuje za 4. Svi flegovi iz PSW registra se brišu (uključujući i I fleg). Svi interapti se maskiraju.

PUSHA

Uticaj na flegove **Z=0, N=0, C=0, V=0, VT=0, ST=0**

****

PUSHF - ubaci na stek PSW registar

Instrukcija ubacuje na stek PSW registar, WSR i preostale interapt maske. Pokazivač steka (registar 18h) se umanjuje za 2. Svi flegovi iz PSW registra se brišu (uključujući i I fleg) kao i bajt koji sadrži interapt maske (deo PSW).

PUSHF

Uticaj na flegove **Z=0, N=0, C=0, V=0, VT=0, ST=0**

****

RET - vrati se na glavni program

Program skače na adresu koju očita sa vrha steka. Pokazivač steka (registar 18h) se uvećava za 2.

RET

Instrukcija se koristi za povratak iz potprograma i interapt rutine.

Uticaj na flegove -

****

RST - resetuj mikroračunar

Ceo PSW registar se briše, program startuje od 2080h. Svi interni registri zauzimaju inicijalno stanje.

RST

Uticaj na flegove **Z=0, N=0, C=0, V=0, VT=0, ST=0**

****

SCALL - poziv bliskog potprograma

Trenutna vrednost programskog brojača se smešta na stek, pokazivač steka (registar 18h) se umanjuje za 2, a program skače na početnu adresu potprograma (labelu koja je operand u instrukciji). Insrukcija CALL objedinjue ovu i LCALL instrukciju.

SCALL labela

Uticaj na flegove -

****

SETC - postavi fleg prenosa na 1

Vrednost flega prenosa (C fleg) postaje jedinca

SETC

Uticaj na flegove **C=1**

****

SHL - šiftuj REČ ulevo

Prvi REČ operand se šiftuje ulevo onoliko puta koliko iznosi drugi operand. Broj šiftovanja može biti neposredno upisana vrednost u opsegu #0 do #15 ili sadržaj BAJT registra. Najvredniji bit (petnaesti) prilikom pomeranja ulevo prelazi u C fleg. Na mesto najmanje značajnog bita dolazi nula. Uticaj na Nfleg je specifičan, vidi opis N flega (poglavlje 13.2).

SHL DSwreg, #broj

ili

SHL DSwreg, breg

Ova se instrukcija može **koristiti za množenje sa 2^N**. Pri tom je N broj šiftovanja

Uticaj na flegove **Z, N?, C, V, VT↑**

****

SHLB - šiftuj BAJT ulevo

Prvi BAJT operand se šiftuje ulevo onoliko puta koliko iznosi drugi operand. Broj šiftovanja može biti neposredno upisana vrednost u opsegu #0 do #7 ili sadržaj BAJT registra. Najznačajniji bit (broj sedam) prilikom pomeranja ulevo prelazi u C fleg. Na mesto najmanje značajnog bita dolazi nula. Uticaj na N fleg je specifičan, vidi opis N flega (poglavlje 13.2) .

SHLB DSbreg, #broj

ili

SHLB DSbreg, breg

Ova se instrukcija može **koristiti za množenje sa 2^N**. Pri tom je N broj šiftovanja

Uticaj na flegove **Z, N?, C, V, VT↑**

****

SHLL - šiftuj LONG ulevo

Prvi LONG operand se šiftuje ulevo onoliko puta koliko iznosi drugi operand. Broj šiftovanja može biti neposredno upisana vrednost u opsegu #0 do #31 ili sadržaj BAJT registra. Najznačajniji bit (broj trideset jedan) prilikom pomeranja ulevo prelazi u C fleg. Na mesto najmanje značajnog bita dolazi nula. Uticaj na N fleg je specifičan, vidi opis N flega (poglavlje 13.2).

SHLL DSlreg, #broj

ili

SHLL DSlreg, breg

Ova se instrukcija može **koristiti za množenje sa 2^N**. Pri tom je N broj šiftovanja.

Uticaj na flegove **Z, N?, C, V, VT↑**

****

SHR - šiftuj REČ udesno (logički)

Prvi REČ operand se šiftuje udesno onoliko puta koliko iznosi drugi operand. Broj šiftovanja može biti neposredno upisana vrednost u opsegu #0 do #15 ili sadržaj BAJT registra. Najmanje značajan bit prilikom pomeranja udesno prelazi u C fleg. Na mesto najznačajnijeg bita (bit broj petnaest) dolazi nula.

SHR DSwreg, #broj

ili

SHR DSwreg, breg

Ova se instrukcija može koristiti za deljenje **neoznačene REČi sa 2^N**. Pri tom je N broj šiftovanja.

Uticaj na flegove **Z, N=0, C, V=0, ST**

****

SHRA - šiftuj REČ udesno (aritmetički)

Prvi REČ operand se šiftuje udesno onoliko puta koliko iznosi drugi operand. Broj šiftovanja može biti neposredno upisana vrednost u opsegu #0 do #15 ili sadržaj BAJT registra. Najmanje značajan bit prilikom pomeranja udesno prelazi u C fleg. Na mesto najznačajnijeg bita (broj petnaest) dolazi ili nula, ukoliko je taj bit pre šiftovanja bio nula, ili jedinica (ako je pre šifovanja bio jedinica).

SHRA DSwreg, #broj

ili

SHRA DSwreg, breg

Ova se instrukcija može koristiti za deljenje **označene REČi sa 2^N**. Pri tom je N broj šiftovanja.

Uticaj na flegove **Z, N, C, V=0, ST**

****

SHRAB - šiftuj BAJT udesno (aritmetički)

Prvi BAJT operand se šiftuje udesno onoliko puta koliko iznosi drugi operand. Broj šiftovanja može biti neposredno upisana vrednost u opsegu #0 do #7 ili sadržaj BAJT registra. Najmanje značajan bit prilikom pomeranja udesno prelazi u C fleg. Na mesto najznačajnijeg bita (broj sedam) dolazi ili nula, ukoliko je taj bit pre šiftovanja bio nula, ili jedinica (ako je pre šifovanja bio jedinica).

SHRAB DSbreg, #broj

ili

SHRAB DSbreg, breg

Ova se instrukcija može koristiti za deljenje **označenog BAJTa sa sa 2^N**. Pri tom je N broj šiftovanja.

Uticaj na flegove **Z, N, C, V=0, ST**

****

SHRAL - šiftuj LONG udesno (aritmetički)

Prvi LONG operand se šiftuje udesno onoliko puta koliko iznosi drugi operand. Broj šiftovanja može biti neposredno upisana vrednost u opsegu #0 do #31 ili sadržaj BAJT registra. Najmanje značajan bit prilikom pomeranja udesno prelazi u C fleg. Na mesto najznačajnijeg bita (bit btoj 31) dolazi ili nula, ukoliko je taj bit pre šiftovanja bio nula, ili jedinica (ako je pre šifovanja bio jedinica).

SHRAL DSlreg, #broj

ili

SHRAL DSlreg, breg

Ova se instrukcija može koristiti za deljenje **označenog LONGa sa sa 2^N**. Pri tom je N broj šiftovanja.

Uticaj na flegove **Z, N, C, V=0, ST**

****

SHRB - šiftuj BAJT udesno (logički)

Prvi BAJT operand se šiftuje udesno onoliko puta koliko iznosi drugi operand. Broj šiftovanja može biti neposredno upisana vrednost u opsegu #0 do #7 ili sadržaj BAJT registra. Najmanje značajan bit prilikom pomeranja udesno prelazi u C fleg. Na mesto najznačajnijeg bita (bit broj 7) dolazi nula.

SHRB DSbreg, #broj

ili

SHRB DSbreg, breg

Ova se instrukcija može koristiti za deljenje **neoznačenog BAJTa sa sa 2^N**. Pri tom je N broj šiftovanja.

Uticaj na flegove **Z, N=0, C, V=0, ST**

****

SHRL - šiftuj LONG udesno (logički)

Prvi LONG operand se šiftuje udesno onoliko puta koliko iznosi drugi operand. Broj šiftovanja može biti neposredno upisana vrednost u opsegu #0 do #31 ili sadržaj BAJT registra. Najmanje značajan bit prilikom pomeranja udesno prelazi u C fleg. Na mesto najznačajnijeg bita (bit broj 31) dolazi nula.

SHRL DSlreg, #broj

ili

SHRL DSlreg, breg

Ova se instrukcija može koristiti za deljenje **neoznačenog LONGa sa sa 2^N**. Pri tom je N broj šiftovanja.

Uticaj na flegove **Z, N=0, C, V=0, ST**

****

SJMP - bezuslovni bliski skok

Program skače na labelu koja je operand u instrukciji. Labela može biti bilo gde u adresnom prostoru. Postoji i insrukcija BR koja objedinjue LJMP i SJMP. NJe nema u instrukcijskom setu samog mikroračunara već je asembler zamenjuje ili SJMP ili LJMP instrukcijom zavisno od udaljenosti labele.

SJMP labela

Uticaj na flegove -

****

SKIP - bez operacije

Ne radi ništa. Program se nastavlja sledećom insrukcijom. Instrukcija ima isto dejstvo kao i NOP samo što zauaima dva bajta u romu.

SKIP

Uticaj na flegove -

****

ST - prebaci REČ registar u memoriju

Sadržaj REČ registra koji je prvi operand se prebacuje u REČ koja je drugi operand.

ST Swreg, Dwaop

Uticaj na flegove -

****

STB - prebaci BAJT registar u memoriju

Sadržaj BAJT registra koji je prvi operand se prebacuje u BAJT koja je drugi operand.

STB Sbreg, Dbaop

Uticaj na flegove -

****

SUB (dva operanda) - oduzmi REČi

Razlika dva operanda tipa REČ se smešta u izlazni registar (prvi operand).

SUB DSwreg, Swaop

Uticaj na flegove **Z, N, C, V, VT↑**

****

SUB (tri operanda) - oduzmi REČi

Razlika drugog i trećeg REČ operanda se smešta u izlazni registar (prvi operand).

SUB Dwreg, S1wreg, S2waop

Uticaj na flegove **Z, N, C, V, VT↑**

****

SUBB (dva operanda) - oduzmi BAJTe

Razlika dva operanda tipa BAJT se smešta u izlazni registar (prvi operand).

SUBB DSbreg, Sbaop

Uticaj na flegove **Z, N, C, V, VT↑**

****

SUBB (tri operanda) - oduzmi BAJTe

Razlika drugog i trećeg BAJT operanda se smešta u izlazni registar (prvi operand).

SUBB Dbreg, S1breg, S2baop

Uticaj na flegove **Z, N, C, V, VT↑**

****

SUBC - oduzmi REČi sa pozajmicom

Od prvog operanda tipa REČ oduzima se drugi operanda tipa REČ. Razlika se umanjuje za 1 ako je C fleg pre ove instrukcije bio obrisan. Izlazni registar je prvi operand. Fleg prenosa (C) dobija komplementarnu vrednost od pozajmice (*borrow*). Ako ima pozajmice prilikom oduzimanja C je nula, ako nema jedinica.

SUBC DSwreg, Swaop

Uticaj na flegove **Z↓, N, C, V, VT↑**

****

SUBCB - oduzmi BAJTe sa pozajmicom

Od prvog operanda tipa BAJT oduzima se drugi operanda tipa BAJT. Razlika se umanjuje za 1 ako je C fleg pre ove instrukcije bio obrisan. Izlazni registar je prvi operand. Fleg prenosa (C) dobija komplementarnu vrednost od pozajmice (*borrow*). Ako ima pozajmice prilikom oduzimanja C je nula, ako nema jedinica.

SUBCB DSbreg, Sbaop

Uticaj na flegove **Z↓, N, C, V, VT↑**

****

TRAP - poziv softverskog prekida

Ova instrukcija je softverski poziv interapta sa vektorom na adresi 2010h. Drugi interapti se neće opslužiti do završetka insrukcije koja sledi neposredno za ovom. Stanje I flega (PSW.9) ne utiče na izvršavanje TRAP instrukcije.

TRAP

Instrukciju najčešće koriste razvojne alatke pa zbog toga je neke verzije asemblara uopšte nemaju kao mnemonik.

Uticaj na flegove -

****

XOR - logička ekskluzivno-ILI operacija nad REČima

Nad dva operanda tipa REČ se izvršava logička ekskluzivno-ILI opereacija, bit po bit. Rezultat ima logičku nulu na onim pozicijama na kojima oba operanda imaju istu vrednost. Na ostalim pozicijama (na kojima je u jednom operandu jedinica, a u drugom nula), dobijaju se jedinice. Rezultat se smešta u izlazni registar (prvi operand).

XOR DSwreg, Swaop

Uticaj na flegove **Z, N, C=0, V=0**

****

XORB - logička ekskluzivno-ILI operacija nad BAJTima

Nad dva operanda tipa BAJT se izvršava logička ekskluzivno-ILI opereacija, bit po bit. Rezultat ima logičku nulu na onim pozicijama na kojima oba operanda imaju istu vrednost. Na ostalim pozicijama (na kojima je u jednom operandu jedinica, a u drugom nula), dobijaju se jedinice. Rezultat se smešta u izlazni registar (prvi operand).

XORB DSbreg, Sbaop

Uticaj na flegove **Z, N, C=0, V=0**

****

15 DODACI

A. TABELA SKOKOVA POSLE POREĐENJA

Posle instrukcije CMP A, B gde su A i B **OZNAČENE** veličine:

JGE	skoči ako je **A ≥ B**
JGT	skoči ako je **A > B**
JLE	skoči ako je **A ≤ B**
JLT	skoči ako je **A < B**
JE	skoči ako je **A = B**
JNE	skoči ako je **A ≠ B**

Posle instrukcije CMP U, V gde su U i V **NEOZNAČENE** veličine:

JC	skoči ako je **U ≥ V**
JH	skoči ako je **U > V**
JNH	skoči ako je **U ≤ V**
JNC	skoči ako je **U < V**
JE	skoči ako je **U = V**
JNE	skoči ako je **U ≠ V**

B. SINTAKSE ASEMBLERSKIH DIREKTIVA I KONTROLA

Asemblerske direktive:

Ime_modula **MODULE** [Atribut]
Atribut = { **MAIN** | **STACKSIZE(**n**)**}

EXTRN {Simbol [**:** tip_podatka]} [**,** …]
Tip_podatka = { **BYTE** | **WORD** | **LONG** | **ENTRY** | **REAL** | **NULL** }

PUBLIC {Simbol} [**,** …]
{**CSEG** | **DSEG** | **RSEG** | **SSEG** | **OSEG}** [**REL** | **AT** Adresa]

ORG Nova_adresa

[labela:] **DCB** {izraz | string} [**,** …]

[labela:] **DCW** {izraz} [**,** …]

[labela:] **DCL** {Long_konstanta} [**,** …]

Simbol {**EQU** | **SET**} izraz [**:** tip_podatka]
Tip_podatka = { **BYTE** | **WORD** | **LONG** | **ENTRY** | **REAL** | **NULL** }

[Simbol:] {**DSB** | **DSW** | **DSL** | **DSR}** aps_izraz

IF uslov
uslov= aps_izraz
[**ELSE**]
ENDIF

Primarne kontrole izveštaja:

ERRORPRINT[**(**ime_fajla**)**]/**NOERRORPRINT**	**EP/NOEP**	**NOEP**
PAGELENGTH (n**)**	**PL**	**PL(60)**
PAGEWIDTH (n**)**	**PW**	**PW(120)**
PRINT [**(**ime_fajla**)**]/**NOPRINT**	**PR/NOPR**	**PR(**izvorni.LST**)**
SYMBOLS/NOSYMBOLS	**SB/NOSB**	**SB**
XREF/NOXREF	**XR/NOXR**	**NOXR**

Primarne kontrole objekta:

DEBUG/NODEBUG	**DB/NODB**	**NODB**
OBJECT [**(**ime_fajla**)**]/**NOOBJECT**	**OJ/NOOJ**	**OJ**(izvorni.OBJ)

Opšte kontrole:

COND/NOCOND	**CO/NOCO**	**CO**
EJECT	**EJ**	**_**
GEN/NOGEN	**GE/NOGE**	**NOGE**
INCLUDE (ime_fajla**)**	**IC**	**_**
LIST/NOLIST	**LI/NOLI**	**LI**
SAVE/RESTORE	**SA/RS**	**_**
TITLE ('string'**)**	**TT**	**TT(**ime_modula**)**

C. KRATAK PREGLED INSTRUKCIJA

U prilogu je data kopija pregleda svih instrukcija iz priručnika. Prva kolona je mnemonik instrukcije. Sufiks B iza mnemonika znači da je operacija osmobitna. Druga kolona prikazuje broj operanada koji slede mnemonik. Skraćen opis instrukcije je u trećoj koloni. Oznake **D** i **B** predstavljaju operande koji moraju biti u registarskom prostoru i kojima se može pristupiti jedino direktnim registarskim načinom adresiranja. **A** je operand koji može biti bilo gde u adresnom prostoru i može mu se pristupiti bilo kojim načinom adresiranja. Pretposlednja kolona pokazuje na koje flegove iz PSW deluje instrukcija. Značenje oznaka je sledeće:

✓ znači da instrukcija ili postavlja, ili briše fleg, u zavisnosti od rezultata operacije.

↑ znači da instrukcija postavlja fleg u zavisnosti od rezultata operacije ali ga nikad ne briše.

↓ znači da instrukcija briše fleg u zavisnosti od rezultata operacije ali ga nikad ne postavlja.

- znači da instrukcija ne utiče na fleg

0 znači da instrukcija briše fleg

1 znači da instrukcija briše fleg

? znači da je stanje flega posle instrukcije nedefinisano ili da je ponašanje instrukcije u odnosu na taj fleg specifično

Primedbe u poslednjoj koloni su sledeće:

1. Ako se mnemonik završava sa B operacija je nad bajtima, u suprotnom nad šesnaestobitnim operandom. Operandi D, B i A moraju da zadovolje pravila lociranja u pogledu parnosti zavisno od tipa instrukcije. D i B su lokacije u registarskom prostoru; A može biti bilo gde u memoriji.
3. D,D+2 su uzastopne REČi u adresnom prostoru.
4. D,D+1 su uzastopne BAJTi u adresnom prostoru; D je na parnoj adresi.
5. Pretvara BAJT u REČ
6. Ofset je označeni broj (u drugom komplementu)
7. Specificirani bit je jedan od 2048 bita registarskog prostora
8. Sufiks L označava operacije nad tridesetdvobitnim podacima
9. Resetuje mikroračunar i postavlja RESET pin na nulu. Softver startuje od 2080h.
10. Asembler ne prihvata ovaj mnemonik
11. I = Dozvola prekida (PSW.9)

D. UTICAJ INSTRUKCIJA NA FLEGOVE

	Z	N	C	V	VT	ST
ADD, ADDB	✓	✓	✓	✓	↑	--
ADDC, ADDCB	↓	✓	✓	✓	↑	--
AND, ANDB	✓	✓	**0**	**0**	--	--
BMOV, BMOVI*	--	--	--	--	--	--
BR (Indirect)	--	--	--	--	--	--
CLR, CLRB	**1**	**0**	**0**	**0**	--	--
CLRC	--	--	**0**	--	--	--
CLRVT	--	--	--	--	**0**	--
CMP, CMPB	✓	✓	✓	✓	↑	--
CMPL	✓	✓	✓	✓	↑	--
DEC, DECB	✓	✓	✓	✓	↑	--
DI	--	--	--	--	--	--
DIV, DIVB, DIVU, DIVUB	--	--	--	✓	↑	--
DJNZ, DJNZW	--	--	--	--	--	--
DPTS	--	--	--	--	--	--
EI	--	--	--	--	--	--
EPTS	--	--	--	--	--	--
EXT, EXTB	✓	✓	**0**	**0**	--	--
IDLPD						
Ispravni operand	--	--	--	--	--	--
Neispravni operand	**0**	**0**	**0**	**0**	**0**	**0**
INC	✓	✓	✓	✓	↑	--
INCB	✓	✓	✓	✓	↑	--
JBC, JBS, JC, JE, JGE, JGT, JH, JLE, JLT, JNC, JNE, JNH, JNST	--	--	--	--	--	--
JNV	--	--	--	--	--	--
JNVT	--	--	--	--	**0**	--
JST, JV	--	--	--	--	--	--
JVT	--	--	--	--	**0**	--
LCALL, LD, LDB, LDBSE, LDBZE	--	--	--	--	--	--
LJMP	--	--	--	--	--	**?**
MUL, MULB, MULU, MULUB	--	--	--	--	--	**?**
NEG, NEGB	✓	✓	✓	✓	↑	--
NOP	--	--	--	--	--	--
NORML	✓	**?**	**0**	--	--	--
NOT, NOTB	✓	✓	**0**	**0**	--	--
OR, ORB	✓	✓	**0**	**0**	--	--
POP	--	--	--	--	--	--
POPA, POPF	✓	✓	✓	✓	✓	✓
PUSH	--	--	--	--	--	--
PUSHA, PUSHF	**0**	**0**	**0**	**0**	**0**	**0**
RET	--	--	--	--	--	--
RST	**0**	**0**	**0**	**0**	**0**	**0**
SCALL	--	--	--	--	--	--
SETC	--	--	**1**	--	--	--
SHL, SHLB, SHLL	✓	**?**	✓	✓	↑	--
SHR	✓	**0**	✓	**0**	--	✓
SHRA, SHRAB, SHRAL	✓	✓	✓	**0**	--	✓

	Z	N	C	V	VT	ST
SHRB, SHRL	✓	**0**	✓	**0**	--	✓
SJMP	--	--	--	--	--	--
SKIP	--	--	--	--	--	--
ST, STB	--	--	--	--	--	--
SUB, SUBB	✓	✓	✓	✓	↑	--
SUBC, SUBCB	↓	✓	✓	✓	↑	--
TIJMP*	--	--	--	--	--	--
TRAP	--	--	--	--	--	--
XCH*, XCHB*	--	--	--	--	--	--
XOR, XORB	✓	✓	**0**	**0**	--	--

* Instrukcije koje su dodate seriji MCS96 počev od modela 80c196KC pa nadalje