Priručnik za asemblersko programiranje na procesorima iz familije Intel 80x86

- osnovni kurs -

Autor:

Miodrag Milanović

Novi Sad, 2000

1. Uvod

Za razvoj asemblerskih programa potrebne alatke su asembler, linker i dibager. U ovom poglavlju su navedene osnovne informacije potrebne za korišćenje turbo asemblera, turbo linkera i turbo dibagera (proizvodi firme Borland).

1.1.1. Asembler

Turbo Assembler se poziva iz komandne linije operativnog sistema komandom: **TASM ime_datoteke**. Podrazumevana ekstenzija datoteke je **asm**. Na ovaj način smo startovali prevodilac, koji je generisao datoteku sa imenom ime_datoteke.obj. Ovako kreirane objektne datoteke ne sadrže informacije neophodne za rad sa dibagerom. Da bi se takve datoteke dobile potrebno je prevodilac pozvati komandom **TASM /zi ime_datoteke**.

1.1.2. Linker

Turbo Linker se poziva komandom: **TLINK /v ime_datoteke**. U komandi se može navesti i veći broj objektnih datoteka. Podrazumevana ekstenzija je **obj**. Opcija koja je navedena uz ime programa, služi da se u generisanoj izvršnoj datoteci nalaze i infomacije potrebne za rad dibagera. Generisana izvršna datoteka će dobiti ime po prvoj navedenoj objektnoj datoteci i imaće ekstenziju EXE. Kao rezultat se formira još jedna datoteka, sa ekstenzijom MAP, u kojoj se nalaze informacije o veličini pojedinih segmenata.

1.1.3. Dibager

Startovanje Turbo Debugger-a se izvodi naredbom **TD**, a uz nju se može i navesti naziv izvršne datoteke koju želimo analizirati.

<table>
<tr><td>1</td><td>2</td><td>3</td></tr>
<tr><td>5</td><td colspan="2">4</td></tr>
</table>

Turbo dibager ima puno opcija. U ovom priručniku navešćemo samo one komande, koje su nam najčešće potrebne u našem radu, i koje ćemo redovno koristiti. Opise preostalih komandi korisnik može dobiti pritiskom na taster F1, kada dođe do odgovarajuće opcije u meniju.

Na predhodnoj skici se nalazi izgled ekrana koji se dobija prilikom startovanja dibagera, i to bez parametara. Ukoliko se navede ime izvršne datoteke, a mi se nalazimo u direktorijumu u kome se nalazi izvorna datoteka programa, tada će ona biti učitana. Njen prozor se tada može dobiti odabirom opcije **CPU** iz **View** menija

Kao što se na skici može videti, radni prostor je podeljen na pet manjih celina, između njih se prelazi tasterom **TAB**.

Prvi deo sadrži uporedni ispis asemblerskih instrukcija i njihovih heksadecimalnih mašinskih kodova. Kroz taj deo je moguće kretati se strelicama, a direktnim unošenjem sa tastature asemblerskih naredbi moguće je menjati program "u letu". Izvršavanje programa se može vršiti instrukciju po instrukciju, što se čini tasterom **F7**, ili do označene linije, pomoću tastera **F4**, ili kompletnog programa sa **F9**. Ukoliko prilikom izvršenja instrukciju po instrukciju, ne želite da ulazite u procedure, tada koristite **F8**. Postoje naravno i mnoge druge komande, ali nam one nisu za početak toliko interesantne.

Drugi deo sadrži informacije vezane za registre procesora, kroz čiju listu se krećemo kursorskim tasterima. Naravno, postoje komande za izmenu sadržaja registara, koje se postižu kombinacijama tastera. **Ctrl-C** omogućuje da se pojavi dijalog za unos nove vrednosti registra. **Ctrl-I** uvećava sadržaj registra za jedan, dok ga **Ctrl-D** umanjuje za jedan. Pošto je dovođenje registra na nulu česta operacija, postoji i kombinacija za njeno aktiviranje: **Ctrl-Z**. Funkcija i značenje svakog registra ponaosob biće objašnjeni u narednom poglavlju.

Tu su naravno i bitovi status registra, koje sadrži treći deo prozora. Kroz njihovu listu se takođe krećemo kursorskim tasterima, a pritiskom na **ENTER** menjamo njihovo stanje. O tome koja je funkcija svakog od bitova status registra biće više reči u narednom poglavlju.

Pošto procesor ima internu podršku za stek, u četvrtom delu možemo pratiti trenutno stanje steka, i eventualno vršiti njegove izmene pomoću kombinacije **Ctrl-C**.

U preostalom petom delu nalazi se prikaz sadržaja memorije. Moguće je prikazati bilo koji blok memorije, ali je nama naravno najvažniji segment podataka. Pomoću **Ctrl-G** unosimo adresu od koje želimo da započne ispis. Tu ćemo najčešće upisivati DS:0. Prikaz je za početak u bajtovima, mada se pomoću

Ctrl-D može promeniti na bilo koji standardan zapis celoih ili brojeva u pokretnom zarezu, koji se koriste u asembleru.

Od interesantnih opcija koje nudi ovaj dibager, a i većina drugih, je postavljanje prekidnih tačaka. Ovde se to čini tasterom **F2**, a linija na kojoj je postavljena će biti označena crvenom bojom. Mi možemo, naravno, postaviti veći broj prekidnih tačaka a kada, prilikom izvršavanja, procesor naiđe na obeležene instrukcije, izvršavanje će biti zaustavljeno. Uklanjanje prekidnih tačaka se vrši na isti način kao i njihovo postavljanje.

Turbo dibager je simbolički dibager. Ukoliko su programi prevedeni na predhodno navedeni način, tada turbo dibager omogućuje praćenje stanja svih promenljivih pod onim imenima koje smo im dali prilikom pisanja izvornog koda. Ova mogućnost daleko olakšava testiranje programa, i nalaženje grešaka u njemu. Pomoću opcije **Variables** iz **Views** menija, moguće je dobiti pregled svih promenljivih, i obaviti njihove izmene. Slično tome je uz pomoć opcije **Watches**, iz istog menija, moguće menjati i pratiti vrednosti onih promenljivih koje mi odaberemo.

Preporuka je da se čitalac zadrži u ispitivanju mogućnosti ovog dibagera jedno vreme, jer bolje poznavanje njegovih mogućnosti u mnogome olakšava razvoj asemblerskih programa.

2. Procesor 8086

2.1. Uvod

Familiju procesora Intel 80x86 sačinjavaju procesori 8086, 80286, 80386, 80486 i Pentium. Prisutna je kompatibilnost u nazad, te procesori 80286, 80386, 80486, kao i procesori iz Pentium serije, imaju kao zajednički podskup instrukcije koje čine repertoar instrukcija procesora Intel 8086. U ovom materijalu će biti opisan upravo taj osnovni podskup.

2.2. Registri

Procesor Intel 8086 poseduje 14 registara. Od toga su 4 opšte namene, 2 pointera, 2 indeksna registra, programski brojač, 4 segmentna registra i jedan status registar. Treba reći da je ovaj set registara takođe prisutan i u svim sledećim generacijama ove Intel-ove familije procesora.

2.2.1. Registri opšte namene

Registri opšte namene imaju najširi dijapazoj upotrebe, ali svaki od njih ima i neke specifičnosti. Jedna stvar koja je specifična samo za njih je da se mogu posmatrati ili kao 16-bitni ili kao par 8-bitnih registara.

15 8	7 0	
AH	**AL**	**AX**
BH	**BL**	**BX**
CH	**CL**	**CX**
DH	**DL**	**DX**

AX i njegov niži deo **AL** se često nazivaju i akumulator i predstavljaju podrazumevani argument nekih instrukcija, na primer u BCD aritmetici i kod instrukcija ulaza i izlaza. Sve instrukcije koje koriste **AX** kao operand zauzimaju manji prostor u memoriji nego one koje rade sa drugim registrima.

BX (Base Register) se koristi kao bazni registar prilikom pristupa memoriji, naravno kada se radi o nekoj od baznih metoda adresiranja.

CX (Counter Register) se koristi kao brojač, tj. služi za smeštanje trenutne vrednosti brojača. Implicitno se umanjuje prilikom izvršavanja instrukcija za rad sa stringovima i petljama. Takođe se koristi, doduše njegov niži deo **CL**, u istrukcijama za rotiranje i pomeranje, gde se u njega smešta operand koji predstavlja broj ponavljanja date operacije nad argumentom.

Registar **DX** se implicitno koristi u aritmetičkim operacijama množenja i delenja, a takođe se koristi i za smeštanje adrese porta kod ulazno-izlaznih instrukcija.

2.2.2. Pointerski registri

Pointerski registri služe za pristup podacima koji se nalaze u stek segmentu, a dozvoljena je i njihova upotreba u atimetičko-logičkim operacijama.

SP	Stack Pointer
BP	Base Poitner

SP je pokazivač na vrh steka, i svaka instrukcija koja radi sa stekom ga modifikuje. Uz njega se podrazumeva segmentni registar **SS** (Stack Segment) prilikom formiranja apsolutne adrese. Treba izbegavati "ručne" izmene ovog registra, u slučaju da niste sigurni šta se tom prilikom dešava i šta se trenutno nalazi na steku.

BP se obično koristi za pristup podacima koji se nalaze na steku i to bez izmene steka. To je veoma korisno, jer na taj način možemo podatke prenete potprogramu koristiti u proizvoljnom redosledu, nezavisnom od redosleda postavljanja argumenata na stek. Na steku se obično nalaze i lokalne promenljive, koje pripadaju datom potprogramu i njima se pristupa na isti način.

2.2.3. Indeksni registri

Indeksni registri se koriste implicitno prilikom rada sa stringovima. Međutim, oni se mogu koristiti i kao registi opšte namene u aritmetričkim i logičkim operacijama.

SI	Source Index
DI	Destination I.

SI (Source Index) predstavlja indeks unutar izvornog stringa. Njegova vrednost se menja u zavisnosti od flega smera (Direction).

DI (Destination Index) igra sličnu ulogu i prestavlja indeks unutar odredišnog stringa. Njegova vrednost se takođe implicitno menja u zavisnosti od flega smera kao i **SI**.

2.2.4. Programski brojač

Programski brojač se kod ovog procesora zove **IP** (Instruction Pointer).

IP	Instr. Pointer

On pokazuje na narednu instukciju koju treba da izvrši procesor. Njegovu vrednost uvećava sam procesor prilikom izvršenja instukcije, osim u slučajevima skoka i poziva potprograma, kada mi zadajemo adresu na koju će se postaviti programski brojač. Do promena programskog brojača dolazi kada se desi prekid i kada procesor prelazi na izvršenje prekidne rutine.

2.2.5. Segmentni registri

Segmentni registri omogućavaju segmentnu oragnizaciju memorije procesora. Svaki od ovih segmentnih registara definiše blok od 64 KB memorije.

CS	Code Seg.
DS	Data Seg.
SS	Stack Seg.
ES	Extra Seg.

CS govori u kom bloku memorije se nalazi izvršni kod programa. U toku faze dobavljanja instrukcije pristupa se isključivo lokacijama koje se nalaze u ovom segmentu. Sadržaj CS registra se smešta na stek prilikom poziva potprograma samo u slučaju kada se potprogram nalazi u drugom segmentu (**far**).

DS pokazuje na blok memorije namenjen smeštanju podataka. Sve instrukcije, osim onih za rad sa stekom, pristupaju ovom segmentu (izuzev ako se to drugačije ne navede).

SS definiše stek segment i samim tim se koristi implicitno u svim operacijama za rad sa stekom, odnosno kada se adresa formira sa registrima **SP** i **BP**, čija funkcija je predhodno navedena.

ES je dodatni segment podataka, i implicitno se koristi za rad sa stringovima.

2.2.6. Status registar

Status registar je 16-bitni registar koji nam daje više informacija o rezultatu poslednje izvršene instrukcije. Prazna mesta označavaju rezervisane bite, čiji je sadržaj obično nula.

				O	D	I	T	S	Z		A		P		C

Funkcije bitova status registra su sledeće :

- **O** - Overflow, Ovaj bit označava da je došlo do prekoračenje opsega prilikom izvršenja aritmetičke operacije, koja se odnosi na rad sa označenim brojevima
- **D** - Direction, Ovaj bit se koristi u operacijama za rad sa stringovima i pokazuje smer kretanja unutar stringa, 0 - ka višim, 1 - ka nižim memorijskim adresama.
- **I** - Interrupt Enable, Ako je ovaj bit postavljen procesor će reagovati na spoljašnje prekide, a ako nije, reagovaće jedino na NMI (nemaskirajući prekid).
- **T** - Trap, Ovaj bit se koristi za izvršavanje programa instrukciju po intrukciju. Ako je postavljen, posle svake instrukcije će biti generisan INT 3.
- **S** - Sign, Ovaj bit sadrži znak rezultata predhodne instrukcije, zapravo kopiju najvišeg bita rezultata. Pretpostavka je da se radi o označenom broju.
- **Z** - Zero, Ovaj bit nam govori da je rezultat poslednje operacije jednak nuli, odnosno, ako se radi o poređenju, da su operandi jednaki.
- **A** - Auxiliary Carry, Ovaj bit predstavlja bit prenosa odnosno pozajmice sa tećeg bita, pri radu u BCD aritmetici.
- **P** - Parity, Ovaj je bit pariteta, koji će biti postavljen u slučaju da je broj jedinica u rezultatu predhodne instrukcije paran.
- **C** - Carry, Ovo je bit prenosa sa najvišeg mesta posle izvršavanja aritmetičkih operacija; On se koristi i kod operacija pomeranja i rotiranja.

2.3. Metode adresiranja

2.3.1. Neposredno adresiranje

Ovo je najjednostavnija metoda adresiranja. Kao što joj ime kaže, radi se o neposrednoj dodeli vrednosti. Kada se instrukcija, koja koristi ovo adresiranje, prevede u mašinski oblik, neposredno iza koda operacije nalaziće se zadata konstanta.

```
mov     ah,34h
mov     ax,4563h
```

Za 16-bitne konstante, važno je naglasiti da se u memoriju prvo smešta njen niži, a zatim viši bajt.

2.3.2. Registarsko adresiranje

Ovo je jedna od najjednostavnijih metoda adresiranja i svodi se na preuzimanje sadržaja registra i njegovog smeštanja u odredište. Primeri :

```
mov     ax,bx
mov     si,dx
mov     al,dl
mov     ds,ax
mov     ax,cs
```

2.3.3. Registarsko indirektno adresiranje

Sadržaj registra, koji je naveden između uglastih zagrada, tumači se kao adresa na kojoj se nalaze podaci. Primeri:

```
mov     al,[bx]
mov     al,[bp]
mov     al,[si]
mov     al,[di]
```

Treba naglasiti da se prilikom upotrebe registara BX, SI i DI za računanje efektivne adrese koristi segmentni registar DS, dok se za BP koristi SS. Eksplicitnim naviđenjem segmentnih registara moguće je uticati na računanje efektivne vrednosti, kao u sledećim primerima.

```
mov     al,cs:[bx]
mov     al,ds:[bp]
mov     al,es:[si]
mov     al,ss:[di]
```

2.3.4. Direktno adresiranje

Radi se o metodi adresiranja koja podrazumeva da joj je parametar adresa na kojoj se nalazi vrednost koju treba preuzeti. U mašinskom formatu adresa lokacije se nalazi kao neposredni operand i uvek je 16-bitna vrednost. Primeri :

```
mov     al,[0234h]
mov     [0045h],ax
```

Treba reći da se kod ove metode adresiranja kao segmentni registar podrazumeva DS, ali je to moguće predefinisati kao u narednim primerima.

```
mov     al,es:[0234h]
mov     ax,cs:[3452h]
```

2.3.5. Indeksirano adresiranje

Ova metoda adresiranja predstavlja varijantu koja povezuje registarsko indirektno i direktno adresiranje. Efektivna adresa se formira sabiranjem vrednosti registra i navedene konstante, pa se sa lokacije, koju određuje efektivna adresa, uzima podatak ili se on u nju smešta. Kod ove metode adresiranja indeks je u registrima SI i DI, a konstanta je u principu početna adresa bloka kome se pristupa.

```
mov     al,20h[si]  ili  mov        al,[si+20h]
mov     dl,78[di]  ili   mov        dl,[di+78]
```

Treba naglasiti da se prilikom računanja efektivne adrese koristi segmentni registar DS, što se može izmeniti ako se eksplicitno navedu segmentni registri kao u sledećim primerima.

```
mov     bh,es:[si+340]
mov     cl,ss:[di+120]
```

2.3.6. Bazno adresiranje

Razlika ovog adresiranja i indeksiranog adresiranja je u tome da bazni registar predstavlja početak bloka, a konstanta, koja se uz njega navodi relativan položaj elementa kome pristupamo. Kao bazni registri se mogu koristiti BX i BP.

```
mov     al,20h[bx] ili mov   al,[bx+20h]
mov     dl,120[bp] ili mov   dl,[bp+120]
```

Kada je bazni registar BX, za računanje efektivne adrese koristi se segmentni registar DS, dok se kod baznog registar BP koristi segmentni registar SS. Ovo se može izmeniti ako se segmentni registri navedu eksplicitno :

```
mov     bh,es:[bx+340]
mov     cl,cs:[bp+120]
```

2.3.7. Bazno indeksirano adresiranje

Ovo je podvarijanta indeksiranog adresiranja, s tom razlikom da više ne postoji konstanta koja određuje početak strukture kojoj se pristupa, već tu ulogu preuzima bazni registar. Kao i kod predhodnih metoda, važi ista napomena vezana za segmentne registre.

```
mov  al,ds:[bp][si]  ili  mov  al,ds:[bp+si]
mov  dl,es:[bx][di]  ili  mov  dl,es:[bx+di]
```

2.3.8. Bazno indeksirano relativno adresiranje

Ovo je najkompleksnija metoda adresiranja koja kombinuje prethodne dve vrste adresiranja, bazno i bazno indeksirano. U računanju efektivne adrese pored jednog baznog i jednog indeksnog registra učestvuje i konstanta pomeraja. Takođe važi napomena vezana za segmentne registre. Primeri za ovu metodu su slični predhodnim.

```
mov     al,ds:25h[bp][si]   ili
mov     al,ds:[bp+si+25h]

mov     dl,es:120[bx][di]   ili
mov     dl,es:[bx+di+120]
```

2.4. Instrukcije

Sada ćemo se upoznati sa repertoarom instrukcija procesora Intel 8086. Instrukcije ćemo podeliti u više logičkih grupa, i to po

njihovoj funkciji. Ovaj odeljak priručnika treba da posluži čitaocu da se upozna sa funkcionalnošću svake instrukcije kako bi bio u mogućnosti da prati ostatak priručnika.

2.4.1. Instrukcije za prenos podataka

Ova grupa sadrži instrukcije koje ćete najčešće upotrebljavati prilikom pisanja programa u asembleru. Podelićemo ih u četiri manje funkcionalne grupe.

2.4.1.1. Instrukcije prenosa opšte namene

U ovoj grupi se nalazi instrukcija **MOV**. Ona poseduje dva operanda, od kojih je prvi odredište, a drugi izvor podatka koji treba preneti. Moguće je vršiti prenos između registara, memorijskih lokacija, a i dodeliti vrednost konstante, ali sve kombinacije nisu moguće.

Instrukcija **LEA**, predstavlja specifičnu instrukciju prenosa. Ona služi za učitavanje efektivne adrese drugog operanda, tj. njene adrese unutar podrazumevanog segmenta. Prvi operand ove instrukcije mora biti neki od 16-bitnih registara izuzev segmentnih.

Učitavanje pointera se vrši uz pomoć instrukcija **LDS** i **LES**. Drugi operand predstavlja adresu na kojoj se nalazi zabeležen 32-bitni pointer. Prvi operand predstavlja registar u koji će biti smešten ofset na koji pokazuje pointer, dok se segment smešta u jedan od segmentnih registara (**DS**, kod instrukcije **LDS**, odnosno u **ES**, kod **LES**).

Moguća je i razmena sadržaja sa status registrom, i to pomoću instrukcija **LAHF** i **SAHF**. Ove instrukcije nemaju operanada, pa se podrazumeva **AH** kao prvi, a nižih 8-bita status registra kao drugi operand. Instrukcija **LAHF** smešta sadržaj nižih osam bita status registra u **AH**, dok instrukcija **SAHF** ima suprotno dejstvo.

Preostala instrukcija **XCHG** služi za zamenu sadržaja dva operanda. Jedan od operanada može biti i memorijska lokacija, dok drugi obavezno mora biti registar. Korisnost ove instrukcije je u tome što se izbegava korišćenje treće, pomoćne, promenljive prilikom izmene sadržaja.

2.4.1.2. Instrukcije za rukovanje stekom

Stek je jedna od procesorsko podržanih struktura podataka. Radi se o LIFO (Last In First Out) organizaciji, što znači da prvo očitavamo poslednje uneti podatak. Instrukcije ove grupe, kao implicitni operand, imaju memorijsku adresu na koju pokazuje registarski par **SS:SP**, što ujedno predstavlja vrh steka. Treba reći da se stek puni prema dole, tj. prema nižim adresama.

Instrukcija stavljanja podatka na stek je **PUSH**. Njen jedini operand određuje sadržaj koji treba staviti na stek. On može biti 16-bitni registar, uključujući i segmentne registre (osim **CS**), ili sadržaj neke memorijske lokacije.

Za skidanje sa steka koristi se instrukcija **POP**. Njen operand govori gde će podatak biti smešten, i to može biti 16-bitni registar, ili memorijska lokacija.

Postoje i instrukcije **PUSHF** i **POPF**, za koje se podrazumeva da je njihov operand status registar.

Stek se implicitno koristi i u nekim drugim instrukcijama, što će kasnije biti navedeno.

2.4.1.3. Instrukcije konverzije

Instrukcije ove grupe nemaju operande, a implicitno menjaju neke od registara.

Konverzija iz bajta u reč se obavlja instrukcijom **CBW**, pri čemu se vodi računa i o znaku datog broja. Intrukcija podrazumeva da se ulazni podatak nalazi u **AL**, dok izlaz predstavlja kompletan registar **AX**.

Slično tome imamo instrukciju **CWD**, koja služi za konverziju iz reči u duplu reč, pri čemu se podrazumeva da je ulaz registar **AX**, a izlaz registarski par **DX:AX**. Prilikom konverzije se očuvava znak.

Konverzija, uz upotrebu posebne tabele konverzije, može se izvršiti pomoću instrukcije **XLAT**. Podrazumeva se da se operand konverzije nalazi u registru **AL**, dok **BX** sadrži adresu početka tabele konverzije. Instrukcija kao izlaz vraća u registru **AL** vrednost pročitanu na lokaciji dobijenoj sabiranjem vrednosti registara **BX** i **AL**. Tabela konverzije je tabela uzastopnih vrednosti, čiji indeksi odgovaraju vrednostima koje se konvertuju u vrednosti navedene u tabeli.

2.4.1.4. Instrukcije za rukovanje periferijom

Pristup periferijama se vrši isključivo preko instrukcija iz ove grupe, osim ako nemamo računarski sistem sa memorijski preslikanim ulazom-izlazom, u kom slučaju se mogu koristiti i ostale instrukcije prenosa podataka.

Periferijama korenspondiraju posebne memorijske lokavije; nazvane portovi. Za očitavanje vrednosti na nekom portu koristimo instrukciju **IN**. Vrednost se smešta isključivo u akumulator, što je i prvi operand, dok drugi operand može biti 8-bitna konstanta, koja predstavlja adresu porta, ili registar **DX** koji sadrži 16-bitnu adresu porta kojeg čitamo.

Treba reći da prvi operand govori i koja je širina podatka koji se prenosi, **AL** 8-bitna, **AX** 16-bitna.

Slična je i instrukcija **OUT**, koja služi za slanje podatka na odgovarajući port. Operandi su isti, s tom razlikom da su im zamenjena mesta, tj. prvo adresa porta, a zatim podatak. Važe ista ograničenja.

2.4.2. Aritmetičke instrukcije

Aritmetičke operacije su upravo one za čije je izvršavanje, u osnovi, namenjen računar. Aritmetičko logička jedinica je 16-bitna.

Posebnu oblast čini BCD aritmetika. Ona se pojavila kao posledica potrebe izračunavanja vrednosti izraza sa svim tačnim ciframa, što se najviše koristilo prilikom računanja bankovnog salda. BCD brojevi se predstavljaju u formi u kojoj svaka četiri bita predstavljaju binarni kod jedne decimalne cifre. Postoje tzv. otpakovani i spakovani format. Otpakovani format ima za pretpostavku da se u jednom 8-bitnom registru nalazi samo jedna cifra, što je uvedeno radi lakše implementacije aritmetičkih operacija, dok spakovani format podrazumeva da se u 8-bitnom registru nalaze dve cifre, što je opet zgodno za čuvanje takvih brojeva u memoriji, jer su oni i onako predugački. Nevolja korišćenja brojeva u spakovanoj formi je u tome što se ne mogu vršiti operacije množenja i delenja direktno nad njima.

2.4.2.1. Instrukcije za sabiranje

Instrukcije za sabiranje poseduju samo dva operanda, oni sadrže sabirke, a rezultat sabiranja se smešta u prvi operand. Treba reći i da se sabiranje vrši na isti način i sa označenim i sa neoznačenim brojevima, te za to ne postoje posebne instrukcije.

Prva od instrukcija sabiranja je **ADD**, koja sabira dva operanda, a rezultat smešta u prvi. Operandi naravno moraju biti iste dužine, a dozvoljene su kombinacije registara, memorijskih lokacija, kao i konstante kao drugog operanda, što je i logično zbog smeštanja rezultata u prvi operand.

Često nam se, međutim, dešava situacija da treba sabirati brojeve iz opsega šireg od postojećeg 16-bitnog. Takvu situaciju rešavamo upotrebom instrukcije **ADC**, za sabiranje u višestrukoj preciznosti. Ona je, u principu, veoma slična instrukciji **ADD**, s tim da se na zbir dodaje i vrednost bita prenosa iz status registra, čija vrednost se, recimo, postavlja sabiranjem nižih delova brojeva, a onda koristi u sabiranju viših delova.

Prilikom pisanja programa u asembleru često se javlja potreba za uvećavanjem vrednosti nekog registra za jedan, što je najčešći slučaj u petljama. Iz razloga pojednostavljenja pisanja, kao i smanjenja dužine mašinskog koda, uvedena je instrukcija **INC**. Ona ima jedan operand koji može biti neki od 8-bitnih ili 16-bitnih registara (izuzimajući segmentne registre) ili memorijska lokacija.

Sabiranje brojeva u BCD otpakovanom formatu se takođe vrši sa instrukcijom **ADD**, ali je nakon njene upotrebe potrebno pozvati instrukciju **AAA** koja je zadužena da izvrši popravku rezultata i njegovo pretvaranje u BCD oblik. Ograničenje je u tome što se sabiranje mora vršiti kroz registar **AX**, jer je on podrazumevani operand ove instrukcije.

Sabiranje se može vršiti na identičan način i u spakovanoj formi, s tom razlikom da se nakon sabiranja mora pozvati instrukcija **DAA**, koja ima dejstvo kao i predhodna instrukcija, ali se radi o spakovanoj formi.

2.4.2.2. Instrukcije za oduzimanje

Postoji više instrukcija za oduzimanje.

Instrukcija **SUB** vrši duzimanje drugog operanda od prvog, a rezultat smešta u prvi operand. Operandi moraju biti istih dužina da bi se dobio i ispravan rezultat, bilo označen bilo neoznačen.

Kada nam 16-bita nije dovoljno da bismo izvršili smeštanje potrebnih vrednosti, koristimo instrukciju **SBB** za oduzimanje u višestrukoj preciznosti. Ona takođe ima dva operanda, ali u računu koristi i pozajmicu iz predhodnog oduzimanja nižih delova brojeva, koja se nalazi u bitu prenosa status registra.

Umanjivanje operanda za jediničnu vrednost je takođe često potrebna funkcija, te iz tog razloga imamo instrukciju **DEC**, koja vrši umanjivanje vrednosti jedinog svog operanda, koji može biti bilo registar bilo memorijska lokacija.

Rad sa BCD brojevima se vrši slično kao i kod sabiranja. Kada se koristi BCD aritmetika u otpakovanoj formi, nakon oduzimanja koristimo instrukciju **AAS**, da bismo ponovo vratili rezultat u odgovarajuću formu, a **DAS** kada radimo u spakovanoj formi. Slično kao kod sabiranja, i ovde važi ograničenje da se sabiranje mora vršiti kroz akumulator, jer je on imlicitni operand prethodne dve instrukcije.

2.4.2.3. Instrukcija za poređenje

Poređenje se zasniva na oduzimanju, s tom razlikom da se razlika zanemaruje, a postavljaju se samo bitovi status registra.

Instrukcija **CMP** omogućuje poređenje. Ona ima dva operanda koji se porede. Nakon ove instrukcije se uvek izvršava neka od instrukcija uslovnog skoka.

2.4.2.4. Instrukcija za komplement

Komplementiranje omogućuje instrukcija **NEG**. Ona poseduje jedan operand koji predstavlja vrednost čiji komplement 2 (negativnu vrednost) treba naći.

2.4.2.5. Instrukcije za množenje

Za množenje neoznačenih brojeva koristimo instrukciju **MUL**. Ona podrazumeva da se u akumulatoru nalazi jedan operand i zahteva da se eksplicitno navede samo drugi operand. Ako je operand dužine bajta, tada se množi sa **AL**, a rezultat smešta u **AX** registar. Ako je operand reč, tada se množi sa **AX**, a rezultat se smešta u duplu reč **DX:AX**.

Slično ovome, za množenje označenih brojeva koristimo instrukciju **IMUL**, koja radi na sličan način kao i predhodna.

Kada se radi sa BCD brojevima, moramo koristiti raspakovani format, da bismo bili u mogućnosti da radimo i operaciju množenja. Nakon izvršenog množenja potrebno je popraviti rezultat instrukcijom **AAM**, koja nema operanada, a podrazumevani je akumulator, što se poklapa sa izlaznim registrom instrukcije množenja.

2.4.2.6. Instrukcije za delenje

Za delenje neoznačenih bojeva koristimo instrukciju **DIV**. Ona podrazumeva da se u akumulatoru nalazi deljenik, i zahteva da se kao eksplicitni operand navede samo delilac. Ako je operand dužine bajta, tada je se vrednost registra **AX** deli sa operandom, rezultat smešta u **AL**, a ostatak delenja u **AH**. Kada je operand dužine reči, tada se **DX:AX** deli operandom, rezultat smešta u **AX**, a ostatak u **DX**.

Slično tome imamo instrukciju za delenje označenih brojeva, **IDIV**, za koju su operandi i njihovo tumačenje identnični.

Delenje BCD brojeva u otpakovanom formatu je takođe moguće, ali je nakon njega potrebno instrukcijom **AAD** prilagoditi rezultat delenja. Imlicitni operand ove instrukcije je akumulator, u kome se, ujedno, nalazi rezultat delenja.

2.4.3. Logičke instrukcije i instrukcije pomeranja i rotiranja

2.4.3.1. Logičke instrukcije

Instrukcijom **AND** izvršavamo logičku operaciju I nad operandima, a rezultat operacije se smešta u prvi od njih. Da podsetimo, rezultat ima jedinice na onim binarnim pozicijama na kojima ih istovremeno imaju i oba operanda, dok se na ostalim pozicijama nalaze nule. Dozvoljene kombinacije operanada su iste kao i kod aritmetičkih instrukcija.

Logičko ILI se postiže instrukcijom **OR**, koja ima isti oblik kao i predhodna, ali se ovde u rezultatu jedinica nalazi na onim binarnim pozicijama na kojima ih ima bar jedan od operanada, dok se na ostalim nalaze nule.

Operacija "logičko ekskluzivno ILI" se dobija upotrebom instrukcije **XOR**, a način upotrebe je identičan kao kod predhodnih. U rezultatu ove instrukcije se jedinica nalazi na onim binarnim mestima gde isključivo jedan od operanada ima jedinicu, dok se na ostalim pozicijama nalaze nule.

Logičko NE, odnosno negacija, se postiže instrukcijom **NOT** i, za razliku od predhodnih instrukcija, ima jedan operand, koji ujedno služi i kao izlaz operacije. Njeno dejstvo se svodi na prostu zamenu jednica nulama i obratno.

2.4.3.2. Instrukcija za rukovanje bitovima

Često imamo potrebu da proverimo vrednost nekog bita, a da tom prilikom ne izmenimo sadržaj registra, ili memorijske lokacije u kojoj se on nalazi. Taj problem se rešava instrukcijom **TEST**, koja, zapravo, vrši operaciju logičko I i samo postavlja bitove status registra. Posle ove instrukcije obično slede instrukcije uslovnog skoka.

2.4.3.3. Instrukcije za pomeranje

Postoje dve vrste pomeranja; logičko i aritmetičko. Razlika je u tome što instrukcije aritmetičkog pomeranja čuvaju znak operanda, dok kod logičkog pomeranja to nije slučaj.

Instrukcije logičkog i aritmetičkog pomeranja u levo su identične. Njihovi nazivi su **SHL** i **SAL** respektivno. Ove instrukcije kao prvi operand sadrže vrednost koju treba pomeriti, a drugi operand je broj 1, ili registar **CL**, koji sadrži podatak o zahtevanom broju pomeranja sadržaja prvog operanda. Treba reći da asembler omogućuje da se umesto broja jedan nađe bilo koji broj, jer se on pobrine da

se izvrši odgovarajući broj pomeranja. Pomeranje u levo je ekvivalent množenju sa dva. Prilikom pomeranja ubacuju se nula na najnižu poziciju, a bit sa najviše pozicije odlazi u bit prenosa status registra.

Kod pomeranja u desno se razlikuju operacije logičkog i aritmetičkog pomeranja. Ove operacije su ekvivalentne operacijama delenja sa dva.

Za operaciju logičkog pomeranja u desno instrukcija je **SHR**. Prilikom pomeranja bit sa najniže pozicije odlazi u bit prenosa status registra, dok na najvišu poziciju dolazi nula.

Instrukcija **SAR**, koja omogućuje aritmetičko pomeranje u desno se razlikuje od instrukcije **SHR**, u tome što se na najvišoj poziciji zadržava predhodna vrednost, tj, znak operanda.

2.4.3.4. Instrukcije za rotiranje

Rotacija u levo se obavlja instrukcijom **RCL**, uz preuzimanje zatečenog bita prenosa, i instrukcijom **ROL**, bez preuzimanja zatečenog bita prenosa. Kod obe instrukcije se vrednost sa najviše pozicije smešta u bit prenosa, s tom razlikom da se kod **RCL**, u najniži bit upisuje zatečena vrednost iz bita prenosa, a kod **ROL** vrednost sa najviše pozicije.

Kod rotacije u desnu stranu situacija je slična. Instrukcija **RCR** omogućuje rotiranje u desno uz preuzimanje zatečenog bita prenosa, i **ROR** bez preuzimanja zatečenog bita prenosa. Prilikom rotacije u desno, bit sa najniže pozicije odlazi u bit prenosa status registra, međutim, **RCR** instrukciju na najvišu poziciju dolazi zatečena vrednost bita prenosa, dok kod **ROR** instrukcije na to mesto dolazi vrednost sa najniže pozicije.

2.4.4. Upravljačke instrukcije

Upravljačke instrukcije omogućuju bezuslovnu i uslovnu izmenu redosleda izvršavanja instrukcija, obrazovanje programskih petlji i pozive potprograma.

2.4.4.1. Instrukcije bezuslovnog skoka

Kao osnovna instrukcija skoka, postoji instrukcija bezuslovnog skoka **JMP**. Ona poseduje jedan operand koji predstavlja adresu na koju će se izvršiti skok. Razlikujemo tri različita oblika instrukcije bezuslovnog skoka, kratki skok (**short**) koji obuhvata raspon od -128 do 127 bajta, skok u blizini (**near**) koji se odnosi na lokacije iz istog segmenta, i daleki skok (**far**) koji dozvoljava skok van tekućeg segmenta. Za kratki skok raspon se računa u odnosu na instrukciju skoka. Prilikom prevođenja asembler sam otkriva da li je skok unutar istog segmenta ili između dva segmenta. Kada je reč o kratkom skoku, to treba eksplicitno navesti stavljanjem ključne reči **short** ispred imena labele.

2.4.4.2. Instrukcije uslovnog skoka

Instrukcije uslovnog skoka se koriste nakon instrukcija koje utiču na sadržaj status registra. Obično se radi o aritmetičkim instrukcijama, najčešće poređenju, ili pak o nekoj iz grupe logičkih instrukcija. Njihov jedini operand omogućuje kratki skok u rasponu -128 do 127 bajta.

Instrukcija **JE** omogućuje utvrđivanje važenja relacije jednako. Ona važi za poređenje i označenih i neoznačenih brojeva. Instrukcija **JNE** omogućuje utvrđivanje relacije različito. I ona važi i za označene i za neoznačene brojeve.

Instrukcija **JA**, odnosno **JNBE**, se odnosi na neoznačene brojeve. Ona omogućuje utvrđivanje relacije veći, odnosno nije manji ili jednak.

Instrukcija **JBE**, odnosno **JNA**, se odnosi na neoznačene brojeve. Ona omogućuje utvrđivanje važenja relacije manje ili jednako, odnosno nije veće.

Za utvrđivanje važenja relacije veće ili jednako, odnosno nije manje, koristimo instrukciju **JAE**, odnosno **JNB**. I ova instrukcija važi za neoznačene brojeve.

Za utvrđivanje važenja relacije manje, odnosno nije veće ili jednako služi instrukcija **JB**, odnosno instrukcija **JNAE**. I ova instrukcija važi za neoznačene brojeve.

Za utvrđivanje važenja relacije veće, odnosno nije manje i jednako,za označene brojeve služi instrukcija **JG**, odnosno **JNLE**.

Za utvrđivanje važenja relacije manje ili jednako, odnosno nije veće, za označene brojeve služe instrukcije **JLE**, odnosno **JNG**.

Za utvrđivanje važenja relacije veće ili jednako, služi instrukcija **JGE**, a relacije nije manje instrukcija **JNL**. Obe se odnose na označene brojeve.

Za utvrđivanje važenja relacije manje za označene brojeve služi instrukcija **JL**, odnosno **JNGE**.

Pošto se brojačka konstanta obično smešta u registar **CX**, procesor podržava i instrukcija skoka ako je registar **CX** jednak nuli. To je instrukcija **JCXZ**.

Preostale instrukcije uslovnog skoka omogućuju skok ako je odgovarajući bit status registra jednak nuli odnosno jedinici. Instrukcija **JZ** omogućuje skok ako je bit **Z**

postavljen, a instrukcija **JNZ** omogućuje skok ako ovaj bit nije postavljen. Slično, instrukcija **JC** omogućuje skok kada je bit **C** postavljen, a instrukcija **JNC** omogućuje skok ako ovaj bit nije postavljen.

Što se tiče bita pariteta, instrukcija **JP**, odnosno **JPE** omogućuje skok kada je bit postavljen, a instrukcija **JPO**, odnosno **JNP**, kada on nije postavljen.

Slično za bit prekoračenja instrukcija **JO** omogućuje skok kada je bit **O** postavljen, a instrukcija **JNO** kada on nije postavljen.

Za bit predznaka instrukcija **JS** omogućuje skok kada je bit **S** postavljen, a instrukcija **JNS** kada on nije postavljen.

2.4.4.3. Instrukcije za rukovanje potprogramima

Pozivanje podprograma omogućuje instrukcija **CALL**, čiji operand sadrži adresu potprograma. Ova instrukcija implicitno koristi stek za smeštanje povratne adrese

Povratak iz potprograma omogućuje instrukcija **RET**.

2.4.4.4. Instrukcije za obrazovanje programskih petlji

Instrukcije za obrazovanje programskih petlji podrazumevaju da se brojačka konstanta nalazi u **CX** registru i da se njena vrednost u svakoj iteraciji smanjuje za jedan. Izlazak iz petlje se dešava onda kada registar **CX** postane jednak nuli, i on podrazumeva obavljenje skratkog skoka. Instrukcija **LOOP** je osnovna iz ove grupe, a tu su i **LOOPE** i **LOOPZ**, kao i **LOOPNE** i **LOOPNZ**, kod koji dolazi do izlaska iz petlje i ako je postavljen bit **Z** status registra, odnosno ako on nije postavljen.

2.4.4.5. Instrukcije za rukovanje prekidima

Prekidi se mogu softverski izazivati pomoću instrukcije **INT**, čiji je jedini operand vektor prekidne rutine. Postoji i instrukcija izazivanja prekida prekoračenja **INTO**, koja nema operanda, i izaziva prekid sa vektorom 4 ako je bit **O** status registra postavljen. Prilikom softverskog izazivanja prekida na steku se sačuvaju povratna adresa kao i zatečeni sadržaj status registra.

Povratak iz obrade prekida omogućuje instrukcija **IRET** koja skida sa steka povratnu adresu i sadržaj status registra.

2.4.5. Instrukcije za rukovanje nizovima

Instrukcije za rukovanje nizovima, kao implicitni operand koriste indeksne registre **SI** i **DI**, čija vrednost se nakon izvršenja instrukcije menja zavisno od sadržaja bita **D** status registra. Pritom je **SI** registar vezan za **DS** segmentni registar, a **DI** registar je vezan za **ES** segmentni registar. Sve instrukcije za rukovanje nizovima imaju dva oblika: jedan za rad sa nizovima bajtova, sa postfiksom **B**, i drugi za rad sa nizovima reči, sa postfiksom **W**.

Instrukcija **LODS** (**LODSB** za bajtove i **LODSW** za reči) služi za učitavanja elementa niza u akumulator. Ovde se implicitno menja samo **SI** registar, koji pokazuje na elemente niza iz koga se čitaju podaci.

Smeštanje u niz se vrši instrukcijom **STOS** (**STOSB** za bajtove i **STOSW** za reči). Vrednost se uzima iz akumulatora i smešta u element na koji pokazuje **DI** indeksni registar.

Za kopiranje elemenata nizova koristi se instrukcija **MOVS** (**MOVSB** za bajtove, odnosno **MOVSW** za reči). Kopiranje se vrši iz elementa na koji pokazuje **SI**, u element na koji pokazuje **DI**.

Instrukcija **SCAS** (**SCASB** za bajtove i **SCASW** za reči) omogućuje poređenje akumulatora sa elementima niza. Na elemente niza pokazuje **DI** registar.

Međusobno poređenje elemenata nizova omogućuje instrukcija **CMPS** (**CMPSB** za bajtove, i **CMPSW** za reči). Na prvi operand pokazuje **SI** registar, a na drugi **DI** registar.

Dodavanje prefiksa **REP** ispred imena instrukcije za rukovanje nizovima označava ponavljanje odgovarajuće instrukcije. Pri tome broj ponavljanja smeštamo predhodno u **CX** registar. Nakon svakog ponavljanja vrednost **CX** registra se umanjuje, a ponavljanje završi kada vrednost ovog pregistra padne na nulu. Kada prefiksi **REP**, **REPE** ili **REPZ** stoje ispred instrukcija **SCAS** ili **CMPS**, ponavljanje se završava i kada bit **Z** status registra nije postavljen. U slučaju prefiksa **REPNE** i **REPNZ** ponavljanje se završava i kada je bit **Z** status registra postavljen.

2.4.6. Instrukcije za rukovanje status registrom

Postavljenje bita **C** status registra omogućuje instrukcija **STC**, njegovo poništenje instrukcija **CLC**. Izmenu vrednosti ovog bita omogućuje instrukcija **CMC**.

Postavljanje bita **D** status registra omogućuje instrukcija **STD**, a njegovo poništavanje instrukcija **CLD**.

Instrukcija **STI** postavlja bit **I** status registra, a instrukcija **CLI** poništava ovaj bit.

2.4.7. Razne instrukcije

Instrukcija **NOP** troši procesorsko vreme pa omogućuje generisanje kratkih pauza, reda desetina taktova procesora.

Instrukcija **WAIT** omogućuje usklađivanje rada procesora i numeričkog koprocesora

Instrukcija **HLT** omogućuje zaustavljenje rada procesora.

3. Asemblerski jezik

3.1. Format asemblerske linije

Svaka asemblerska instrukcija ili direktiva zauzima jednu asemblersku liniju. I instrukciji i direktivi može da predhodi labela. Labela od instrukcije, pored razmaka, razdvaja i dvotačka. U asemblerskoj liniji se može nalaziti i komentar, koga najavljuje tačka zarez.

3.2. Segmenti

Za definisanje segmenata uvedene su posebne pretprocesorske direktive.

3.2.1. Programski modeli

Definicija programskog modela saopštava asembleru veličinu i broj segmenata koji će biti korišteni. Model definiše direktiva **.MODEL** koja uvodi naziv modela :

- **TINY** model sadrži jedan segment od 64 Kb u kome su i podaci i kod.
- **SMALL** model sadrži dva segmenta od po 64 Kb, jedan za podatke, a drugi za kod.
- **COMPACT** model sadrži dva segmenta. Jedan od 1 Mb je namenjen za podatke, a drugi od 64 Kb je namenjen za kod.
- **MEDIUM** model sadrži dva segmenta. Jedan od 64 Kb je namenjen za podatke, a drugi od 1 Mb za kod.
- **LARGE** model sadrži dva segmenta od po 1 Mb, jedan za podatke, a drugi za kod.
- **HUGE** model se razlikuje od **LARGE** modela po tome što dozvoljava definisanje nizova dužih od 64Kb.

U ovom priručniku će se koristiti **SMALL** model.

3.2.2. Kraj programa

Kraju svakog programa označava direktiva **END**. Nakon ove direktive može slediti labela koja ukazuje na lokaciju od koje započinje izvršavanje programa.

Asembler ignoriše tekst koji sledi iza ove direktive.

3.2.3. Segment koda

Direktiva **.CODE** označava početak segmenta koda, u kome se nalazi izvršni kod našeg programa. Posle ove direktive slede instrukcije asemblerskog jezika.

3.2.4. Segment podataka

Postoji više vrsta segmenata podataka. Svaki od njih najavljuje posebna direktiva:

- **.DATA** - označava početak segmenta podataka sa inicijalizovanim podacima, odnosno onim podacima čija početna vrednost je prilikom prevođenja poznata.
- **.DATA?** - označava početak segmenta podataka sa neinicijalizovanim podacima. Za ove podatke znamo samo koliku memoriju zauzimaju, a početne vrednosti nisu bitne.
- **.CONST** - definiše početak segmenta konstantnih podataka.

Segmenti **.DATA**, **.DATA?**, **.CONST** spadaju u grupu **DGROUP**, odnosno grupu segmenata podataka.

U nastavku sledi primer korišćenja direktiva.

```
.model  small
.data
.code
        mov     ax,@data
        mov     ds,ax
end
```

Jedine dve instrukcije iz predhodnog primera pune adresu segmenta podataka, koju označava ime **@data**, u **DS** registar.

Ime **@data** označava segment koji čine članovi **DGROUP**. Adresa nije smeštena direktno u segmentni registar, zbog ograničenja procesora pa se dodeljivanje vrši preko akumulatora.

3.2.5. Stek segment

Direktiva **.STACK** rezerviše 1 Kb memorije namenjene steku. Ukoliko je potrebno da stek bude duži, njegova dužina se navodi podle direktive.

Ovako definisani stek takođe spada u grupu **DGROUP** segmenata podataka.

Registri SS i SP se postavljaju automatski prilikom prevođenja, i to SS na početak stek segmenta, a SP na dužinu stek segmenta.

3.3. Konstante

Uz instrukciju se kao parametar mogu navesti kako registri i simboli tako i neposredne vrednosti, odnosno konstante. Njih možemo svrstati u više grupa koje se odlikuju svojom specifičnom upotrebom.

3.3.1. Numeričke i znakovne konstante

Dekadne konstante nastaju redanjem dekadnih cifara, koje se mogu završiti i znakom **d**.

Heksadecimalne konstante označava postfiks **h**, odnosno **H**. Pri tome, konstanta započinje cifrom od **0** do **9**, dok ostale cifre mogu biti od **0** do **F**.

Oktalne konstante označava postifks **q**, odnosno **o**. Dozvoljene cifare su od **0** do **7**.

Binarne konstante se sastoje od cifara **0** i **1**, iza kojih sledi znak **b**.

Stringovi (znakovne konstante) se navode između jednostrukih ili dvostrukih znakova navoda.

Direktiva **EQU** omogućuje davanje imena konstantama. Ispred direktive se navodi ime konstante, a nakon nje konstanta (broj ili string).

3.3.2. Predefinisane konstante

Pored imena **@data** za adresu segmenta podataka predefinisane su i sledeće konstante:

- **$** - adresa tekuće linije koda
- **@code** - adresa segmenta koda
- **@curseg** - adresa aktuelnog segmenta
- **??date** - tekući datum
- **@FileName** - ili **??filename** string sa nazivom asemblirane datoteke.
- **@Startup** - adresa početka koda.
- **??time** - string tekućeg vremena.

3.3. Zauzimanje memorije za promenljive

3.3.1. Skalarne promenljive

Direktiva **DB** omogućuje zauzimanje jednog bajta za promenljivu čije ime se navodi ispred direktive, a iza nje se po potrebi navodi početna vrednost promenljive. Ako se početna vrednost ne navodi, iza direktive sledi znak pitanja. U nastavku slede primeri direktive **DB**:

```
....
.data
x db 5
y db 2
z db ?
....
```

U predhodnom primeru se u segmentu podataka zauzimaju tri uzastopna bajta, u kojima će se nalaziti naše promenljive (x,y i z). Imena x,y i z predstavljaju adrese promenljivih.

Direktiva **DW** omoguæuje zauzimanje dva bajta, a direktiva **DD** omogućuje zauzimanje 4 bajta.

3.3.2. Pokazivači

Zauzeta memorijska lokacija može da sadrži pokazivač. To se opisuje oznakom **PTR** iza koje sledi opis pokazivane lokacije. Tako : **x dw ptr word** označava zauzimanje memorijske lokacije od dva bajta koja sadrži adresu lokacije od dva bajta.

3.3.3. Nizovi

Do zauzimanja niza lokacija dolazi ako se iza direktive za zauzimanje jedne navede broj elemenata niza, a zatim reč **DUP** iza koje u zagradi sledi vrednost na koju se inicijalizuju elementi niza :

```
....
.data
niz       db 100 dup(?)
....
```

U predhodnom primeru nam je definisan jedan niz od 100 bajtova, bez inicijalizacije.

Pristupanje elementu niza se označava kao **ime_niza[indeks]**.

3.4. Uslovno asembliranje

U nastavku su navedene direktive za uslovno asembliranje:

- **IF** - programski blok se asemblira u slučaju da je navedeni uslov ispunjen.
- **IFDEF** - programski blok se asemblira u slučaju da je navedeni simbol definisan.
- **IFNDEF** - asembliranje programskog bloka se vrši ako vrednost simbola nije definisana.

Nakon navođenja programskog bloka koji će se asemblirati ako je neki od uslova ispunjen, može slediti i programski blok, koji se asemblira u suprotnom slučaju. To označava direktiva **ELSE**.

Svaki programski blok za uslovno asembliranje se završava direktivom **ENDIF** :

```
.data
x dw 2
y dw 3
z dw ?
.code
    ifdef saberi
       mov ax,x
       add ax,y
       mov z,ax
    else
       mov z,0
    endif
```

U prethodnom primeru biće asembliran prvi programski blok (u kome se zbir promenljivih **x** i **y** smešta u **z**), ako je definisan simbol **saberi**. Inače će biti asembliran drugi programski blok (u kome se anulira **z**).

3.5. Makroi

Makro definicija se nalazi između direktiva **MACRO** i **ENDM**. Pre prve direktive je potrebno navesti jedinstveni naziv makroa, a nakon nje listu parametara koji će se u samom makrou koristiti. Druga direktiva označava kraj makro definicije i ograničava samo telo makroa. Sledi primer korišćenja ovih direktiva:

```
.data
x dw 2
y dw 3
z dw ?
.code

saberi macro a,b,c
       mov ax,a
       add ax,b
       mov c,ax
       endm

        ...

       saberi x,y,z
```

Nakon asembliranja predhodnog primera poziv makroa **saberi x,y,z** će biti zamenjen instrukcijama:

```
mov     ax,x
add     ax,y
mov     z,ax
```

Znači poziv makroa je zamenjen telom makroa u kome su parametri zamenjeni argumentima.

Direktiva **LOCAL** omogućuje uvođenje lokalnih labela u telu makroa. Jedan primer njene upotrebe je dat u sledećem primeru:

```
pauza macro a
    local umanji

    mov cx,a
umanji:
    dec cx
    jnz umanji
    endm

    ...

    pauza 100
    pauza 20
```

U predhodnom primeru labela **umanji** je proglašena lokalnom, pa višestruki pozivi makroa **pauza** neizazivaju probleme u asembliranju.

3.6. Potprogrami

Definicija prtprograma otpočinje direktivom **PROC** pre koje sledi njeno ime. Nakon ove direktive može da sledi lista parametara.

Potprogrami se pozivaju instrukcijom **CALL**.

Listu parametara najavljuje direktiva **ARG**. Nakon nje slede imena parametara. Za njih se podrazumeva da zauzimaju dva bajta. Iza parametra se može navesti njegov opis, koga najavljuje dvotačka.

Definicija potprograma se završava direktivom **ENDP**, kojoj predhodi naziv potprograma čiji kraj označava ova direktiva.

U nastavku sledi potpun primer asemblerskog programa sa definicijom i pozivom potprograma **saberi**:

```
.model small
.stack
.data
x dw 2
y dw 3
z dw ?
.code
saberi PROC NEAR
     ARG a,b,c : NEAR PTR WORD = ARGLEN
     push bp
     mov bp,sp

     mov ax,a
     add ax,b
     mov bx,c
     mov [bx],ax

     pop bp
     ret ARGLEN

saberi ENDP

start:
     mov ax,@data
     mov ds,ax

     lea ax,z
     push ax
     mov ax,y
     push ax
     mov ax,x
     push ax
     call saberi

     mov ax,4c00h
     int 21h
end start
```

U potprogramu **saberi** se podrazumeva da se argumenti prenose posredstvom steka. Zbog toga je potrebno korišćenje **BP** registra. Za sva tri parametra se podrazumeva da se odnose na lokacije steka velike po dva bajta. Za treći parametar je navedeno da sadrži adresu lokacije velike dva bajta. **ARGLEN** označava broj bajta potreban za smeštanje argumenata na

stek. On se koristi kao operand instrukcije **RET** da bi se stek ispraznio za pomenuti broj bajta. Poslednje dve instrukcije u asemblerskom programu omogućuju da se, po završetku izvršavanja asemblerskog programa, nastavi izvršavanje operativnog sistema.

4. Primeri programa

4.1. Dodela vrednosti

Prilikom računanja matematičkih izraza za privremeno čuvanje vrednosti obično koristimo neki od registara opšte namene, a najčešće akumulator. Da bi se vrednost registra inicijalizovala, odnosno postavila na nulu, koristi se sledeći izraz, konkretno za akumulator.

```
mov     ax,0
```

Međutim, to i nije najsrećnije rešenje. Eksplicitno dodeljivanje brojne vrednosti nekom registru zahteva postojanje neposrednog operanda u okviru mašinske reči. Kod dodeljivanja vrednosti različite od nula, to je neizbežno, međutim, ako je vrednost jednaka nuli to se može izvesti i upotrebom operacije "ekskluzivno ili" nad registrom, dok je i drugi parametar isti taj registar. Za akumulator to izgleda ovako.

```
xor     ax,ax
```

Kao što se može odmah primetiti, ova instrukcija ne zahteva nikakvu brojnu vrednost te je samim tim njen mašinski kod kraći. Ona se zbog svojih osobina mnogo češće sreće nego instrukcija pre nje.

Pre ili kasnije moramo pristupiti nekoj promenljivoj koja se nalazi u segmentu podataka. Naravno od toga kog je tipa podatak zavisi kako ćemo mu pristupiti. Treba obratiti pažnju na to da oba operanda moraju biti iste dužine. Vrednost skalarnoj promenljivoj se dodeljuje instrukcijom u sledećoj formi.

```
mov     ime_promenljive, vrednost
```

Operand, koji sadrži ime promenljive, se, prilikom asembliranja, zamenjuje adresom date promenljive unutar segmenta podataka (offset).

Dodela vrednosti jednom elementu niza ima sledeću formu:

```
mov     ime_niza[indeks], vrednost
```

Dodelu vrednosti svim elementima niza opisuje sledeći asemblerski program :

```
.model small
.stack
.data
niz   db 100 dup(?)
.code
start:
    mov ax,@data
    mov ds,ax

    mov al,5

    xor bx,bx
sledeci:
    mov niz[bx],al
    inc bx
    cmp bx,LENGTH niz
    jne sledeci

    mov ax,4c00h
    int 21h
end start
```

U primeru smo sve elemente niza od sto elemenata inicijalizovali na vrednost 5, koja se nalazi u akumulatoru. Kao što se moglo primetiti pristup svakom elementu je zahtevo računanje nove vrednosti indeksa niza, smeštenog u registru BX. Da smo imali elemente dužine dva bajta registar BX, bi morali uvećavati za dva. Oznaku **LENGTH** pre imena niza u asembliranju zamenjuje dužina niza.

Naravno, ovo nije jedini način za inicijalizaciju niza. Postoji i posebna grupa instrukcija za rad sa nizovima, koju smo imali prilike ranije da upoznamo, a naredni primer predstavlja ekvivalent predhodnog.

```
.model small
.stack
.data
niz   db 100 dup(?)
.code
start:
    mov ax,@data
    mov ds,ax
    mov es,ax

    mov al,5

    cld
    mov cx,LENGTH niz
    mov di,offset niz
    rep stosb

    mov ax,4c00h
    int 21h
end start
```

Pošto instrukcija **STOS** piše na poziciju označenu registarskim parom ES:DI morali smo pre početka inicijalizacije postaviti vrednost datog segmentnog registra da pokazuje na naš segment podataka. Korišćenje prefiksa **REP** pre instrukcije **STOS** uzrokuje nekoliko stvari. Tako registar CX koristimo za brojanje, jer je on implicitni parametar operacije repeticije. Osim što se vrednost brojačkog registra smanjuje, menja se i vrednost indeksnog registra odredišta, tj. DI, i to u onom smeru koji je određen stanjem bita smera status registra. Da bi brojanje bilo ka višim adresama morali smo postaviti taj bit na

nulu. Pri tome se podrazumeva da se za instrukciju **STOSB** vrednost **DI** registra umanjuje za jedan.

4.2. Kopiranje vrednosti

Kopiranje vrednosti se vrši u sledećoj formi:

```
mov     ax, ime_prom1   ili
mov     ime_prom2, ax
```

To naravno važi za slučaj kada se radi o skalaru dužine dva bajta, međutim, kada je u pitanju bajt potrebno je koristi neki od 8-bitnih registara kao pomoćni.

Moguće je iskopirati i vrednost promenljive manje dužine, u onu veće dužine. Da bi se to ostvarilo moramo koristiti i neku od instrukcija za konverziju, kao što je to pokazano u sledećem primeru.

```
.model small
.stack
.data
a     db 0
b     dw ?
c_nizi dw ?
c_visi dw ?
.code
start:
     mov ax,@data
     mov ds,ax

     mov al,a
     cbw
     mov b,ax
     cwd
     mov c_nizi,ax
     mov c_visi,dx

     mov ax,4c00h
     int 21h
end start
```

Ovaj primer nam pokazuje kako se vrednost dužine bajta upisuje u promenljive dužine dva ili četiri bajta. Treba naglasiti da je to ovde učinjeno za promenljive sa označenim vrednostima, jer instrukcije CBW i CWD proširuju vrednost na veću širinu obraćajući pažnju na znak. Ukoliko radimo sa neoznačenim vrednostima postupak je sledeći.

```
.model small
.stack
.data
a     db 0
b     dw ?
c_nizi dw ?
c_visi dw ?
.code
start:
     mov ax,@data
     mov ds,ax

    xor  ax,ax
     mov al,a

     mov b,ax

      xor dx,dx
      mov c_nizi,ax
      mov c_visi,dx

     mov ax,4c00h
     int 21h
end start
```

Za kopiranje nizova se može koristiti i instrukcija **MOVS**. Pri tome se podrazumeva da registarski par DS:SI sadrži adresu početka niza koji kopiramo, dok registarski par ES:DI sadrži adresu odredišnog niza.

```
.model small
.stack
.data
niz1 db 34,3,46,2,3,6,3,6,5,2
niz2 db 10 dup(0)
.code
start:
     mov ax,@data
     mov ds,ax
     mov es,ax

     cld
     mov cx,LENGTH niz2
     mov si,offset niz1
     mov di,offset niz2
     rep movsb

     mov ax,4c00h
     int 21h
end start
```

Kada se na ovaj način radi kopiranje nizova treba obratiti da će operatori **LENGTH**, a i **SIZE**, vratiti ispravnu dužinu jedino kod nizova koji su definisani pomoću **DUP** direktive. Nizovi definisani nabrajanjem elemenata se tumače tako da jedino prvom polju odgovara ime niza, a ostali se računaju kao neimenovani, pa iz tog razloga prevodilac vraća dužinu samo prvog elementa.

4.3. Računanje u duploj preciznosti

Kada za smeštanje vrednosti nije dovoljan registar dužine reči, koriste se registarski parovi koje ćemo posmatrati kao pojedinačne registre dužine duple reči. Najčešći registarski par koji se koristi za računanje, kao akumulator, je DX:AX. Podrazumeva se da se u registru AX nalazi niži deo, a u registru DX viši deo duple reči.

Aritmetička operacija sabiranja ima dva svoja oblika, sabiranje sa i bez bita prenosa. Upravo je to iskorišćeno za formiranje operacije sabiranja u duploj preciznosti.

```
add     ax, nizi_deo
```

```
adc     dx, viši_deo
```

Uvećavanje sadržaja za jedan, je takođe jedna vrsta sabiranja, međutim, pošto instrukcija **INC**, ne postavlja vrednost bita prenosa, moramo koristiti obično sabiranje kada radimo u duploj preciznosti.

```
add     ax, 1
adc     dx, 0
```

Sabiranje sa nulom, navedeno kao druga instrukcija, ima smisla jer se radi o sabiranju u kome se uzima vrednost bita prenosa iz nižeg dela.

Oduzimanje u duploj preciznosti liči na sabiranje u duploj preciznosti:

```
sub     ax, nizi_deo
sbb     dx, visi_deo
```

Dekrementiranje, odnosno umanjenje za jedan, slično inkrementiranju, u osnovi nemenja bit prenosa, pa se mora izvesti preko operacija oduzimanja. Za instrukciju **DEC** imamo sledeću implementaciju.

```
sub     ax, 1
sbb     dx, 0
```

Komplementiranje u duploj preciznosti se oslanja na instrukciju **NEG**, odnosno negaciju. Za ovo postoje dva načina na koji se može izvesti. Prvi predstavlja traženje drugog komplementa po definiciji, što podrazumeva traženje prvog komplementa, odnosno logičko ne, a zatim uvećanje vrednosti za jedan.

```
not     ax
not     dx
add     ax, 1
adc     dx, 0
```

Međutim, postoji i druga varijanta koja se zasniva na tome da instrukcija **NEG** menja bit prenosa ukoliko je potrebna pozajmica.

```
neg     dx
neg     ax
sbb     dx, 0
```

4.4. Asemblerski oblik IF-THEN-ELSE iskaza

Za sledeću sekvencu iskaza C programskog jezika :

```
if  ((x+4 != y*2)  && (x < 10))
  z = x + z;
else
  z = 0;
```

ekvivalentan asemblerski program može da izgleda ovako:

```
.model small
.stack
.data
x     dw 2
y     dw 6
z     dw ?
.code
start:
      mov ax,@data
      mov ds,ax

      mov ax,x
      add ax,4
      mov bx,y
      shl bx,1
      cmp ax,bx
      je  netacno
      mov ax,x
      cmp ax,10
      jge netacno
tacno:
      mov ax,x
      add ax,y
      mov z,ax
      jmp short kraj
netacno:
      mov z,0
kraj:

      mov ax,4c00h
      int 21h
end start
```

4.5. Asemblerski oblik SWITCH iskaza

Za sledeću sekvencu iskaza C programskog jezika :

```
switch (++x) {
  1 :
  2 :  z = x; break;
  3 :  z = z * 2; break;
default :
  z = 0;
}
```

ekvivalentni asemblerski program može da izgleda :

```
.model small
.stack
.data
x     dw 2
z     dw ?
.code
start:
      mov ax,@data
      mov ds,ax

      mov ax,x
      inc ax
```

```
     cmp ax,1
     je  jedan
     cmp ax,2
     je  dva
     cmp ax,3
     je  tri
     jmp short inace
jedan:
dva:  mov z,ax
     jmp short kraj
tri:  shl ax,1
     mov z,ax
     jmp short kraj
inace:
     mov z,0
kraj:

     mov ax,4c00h
     int 21h
end start
```

4.6. Asemblerski oblik FOR iskaza

Za sledeću sekvencu u C programskom jeziku:

```
for (a=0;a++;a <10)
  z = x++;
```

ekvivalentni asemblerski program može da izgleda:

```
.model small
.stack
.data
x      dw 3
z      dw ?
.code
start:
     mov ax,@data
     mov ds,ax

     mov cx,0
     jmp short provera

telo: mov ax,x
       mov z,ax
        inc x

        inc cx
provera:
     cmp cx,10
      jl  telo

     mov ax,4c00h
     int 21h
end start
```

Predhodni asemblerski program može se skratiti ako se koristi instrukcija **LOOP** (koja kao brojač iteracija koristi registar CX, a čiji operand je labela početka tela petlje):

```
.model small
.stack
.data
x      dw 3
.code
start:
     mov ax,@data
     mov ds,ax

     mov cx,10
telo:
     inc x

     loop telo

     mov ax,4c00h
     int 21h
end start
```

Ako se brojač iteracija ne smesti u registar, prethodni asemblerski program može da izgleda:

```
.model small
.stack
.data
x      dw 3
z      dw ?
a      dw ?
.code
start:
     mov ax,@data
     mov ds,ax

     mov a,0
     jmp short provera

telo: mov ax,x
       mov z,ax
        inc x

     inc a
provera:
     mov ax,a
     cmp ax,10
     jl  telo

     mov ax,4c00h
     int 21h
end start
```

4.7. Asemblerski oblik WHILE iskaza

Za sledeću sekvencu iskaza C programskog jezika:

```
while (x<z) x++;
```

ekvivalentni asemblerski program može da izgleda:

```
.model small
.stack
.data
x      dw 3
z      dw 6
.code
start:
     mov ax,@data
     mov ds,ax

provera:
     mov ax,x
     cmp ax,z
```

```
    jge kraj

telo: inc x
    jmp short provera

kraj:
    mov ax,4c00h
    int 21h
end start
```

4.8. Asemblerski oblik REPEAT-UNTIL iskaza

Za sledeću sekvencu iskaza C programskog jezika:

```
do {
  x++;
} while (x<z);
```

ekvivalentni asemblerski program može da izgleda:

```
.model small
.stack
.data
x     dw 3
z     dw 6
.code
start:
    mov ax,@data
    mov ds,ax

telo: inc x

provera:
    mov ax,x
    cmp ax,z
    jge kraj
    jmp short telo

    mov ax,4c00h
    int 21h
end start
```

4.9. Prenos argumenata preko registara

Argumenti se mogu slati preko registara u potprogram samo ako ima dovoljno registara. U primeru koji sledi argumenti potprograma se prosleđuju preko registara:

```
.model small
.stack
.code
pauza PROC
p   : dec cx
    jnz p
    ret
pauza ENDP

start:
    mov cx,1000
    call pauza

    mov ax,4c00h
    int 21h
end start
```

4.10. Vraćanje rezultata potprograma preko registara

U primeru asemblerskog programa koji sledi naveden je potprogram **saberi** koji u akumulatoru vraća sumu (koja je rezultat izvršavanja ovog potprograma).

```
.model small
.stack
.data
x  dw 2
y  dw 6
z  dw ?
.code
saberi PROC NEAR
     ARG a : WORD,b : WORD = ARGLEN

     push bp
     mov bp,sp

     mov ax,a
     add ax,b

     pop bp
     ret
saberi ENDP

start:
    mov ax,@data
    mov ds,ax

    mov  ax,y
    push ax
    mov  ax,x
    push ax
    call saberi
    mov  z,ax

    mov ax,4c00h
    int 21h
end start
```

4.11. Prekidne rutine

Prekidne rutine ili, kako se češće nazivaju, interapti (engl. interrupt), predstavljaju jednu posebnu vrstu potprograma. Postoje dve vrste prekidnih rutina.

Prva, po kojoj su i dobile ime, predstavlja rutine koje se pozivaju u slučaju nekog hardverskog događaja. Takve rutine se nazivaju i obrađivači prekida.

Drugu grupu čine tzv. servisi, odnosno sistemski podprogrami, koji su napisani kako bi se olakšala komunikacija između programera i operativnog sistema. Kod njih se komunikacija odvija preko registara, koji sadrže ili vrednosti potrebnih argumenata, ili pak adrese memorijskih zona gde bi servis trebao da vrati rezultat.

Ove rutine se nepozivaju po imenu, već po rednom broju, za što služi instrukcija INT, nakon koje se navodi broj prekidne rutine. Svaki servis može posedovati više različitih

funkcija, a one se biraju, najčešće, na osnovu sadržaja višeg dela akumulatora.

Na samom početku adresnog prostora ovog procesora se nalazi tabela prekida sa adresama prekidnih rutina. Svaka adresa zauzima 4 bajta, dva za ofset, a dva za segment.

5. Sistemski pozivi i obrada izuzetaka

5.1. Unutrašnji prekidi

Prekide iz ove grupe izaziva procesor. To se dešava kao posledica nekog internog događaja. Obično ih obrađuje operativni sistem.

Prekidi koje ćemo nabrojati su vezani za procesor 8086/8088 (kod procesora novijih generacija uvođeni su i dodatni prekidi).

INT 00h - Divide by Zero

Ovaj prekid izazivaju instrukcije za deljenje, ukoliko je deljenik jednak nuli. Zadatak operativnog sistema u tom slučaju je da prekine izvršenje programa, i da do znanja korisniku da je došlo do delenja nulom. Međutim, ovaj prekid može biti preuzet i od strane korisničkog programa, tako da on sam može da prijavi grešku i da se od nje oporavi u toku svog izvršenja.

INT 01h - Single Step

Ukoliko je postavljen trap bit status registra, doći će do izazivanja ovog prekida nakon svake izvršene instrukcije. Na ovaj način je omogućen rad dibagera, i izvršavanje instrukciju po instrukciju unutar njih.

INT 02h - NMI

Ovaj prekid je tzv. nemaskirajući te je nemoguće programski zabraniti njegovu pojavu, te se bit zabrane prekida status registra neodnosi na njega On se koristi za regulisanje kritičnih događaja za sam rad računara na hardverskom nivou.

Na samom procesoru postoji posebna nožica na koju se dovodi signal usled nekog kritičnog događaja. Kod PC računara ovaj prekid se generiše ukoliko dođe do problema sa memoriskim modulima, greška pariteta, ili kao posledica neispravnog rada koprocesora, ukoliko on postoji. Kod industrijskih računara služi za “general stop”, odnosno za zaustavljanje mašina u kritičnim situacijama.

INT 03h - Breakpoint

Ovaj prekid nam takođe služi za rad samog dibagera. Njega izaziva posebna naredba i tako prekida izvršavanje programa i predaje kontrolu dibageru.

INT 04h - Overflow

Izaziva ga instrukcija **INTO**, ukoliko je bit prekoračenja opsega status registra postavljen. On nam služi za kontrolu grešaka, nakon bilo kakvog niza aritmetičkih operacija. Za numeričke proračune je korisno napisati sopstvenu prekidnu rutinu koja će preuzeti ovaj prekid.

5.2. Vanjski prekidi

U ovoj grupi su opisani prekidi tipični za PC računare.

INT 08h - System timer

Ovaj prekid je vezan na IRQ0 nožicu kontrolera prekida, i na nju se dovodi signal sa tajmera. Ovaj prekid se dešava svakih 55ms, odnosno 18.2 puta u sekundi.

Kao rezultat svog rada postavlja u memoriji vrednost tajmera u otkucajima, koji se nalazi na adresu 0040h:006Ch i dužine je duple reči. On još poziva i prekid 1Ch koji je korisnički prekid i koji slobodno možemo menjati, dok se ne preporučuje menjanje tela obrađivača ovog prekida.

Od svih harderskih prekida ovaj ima najviši prioritet.

INT 09h - Keyboard data ready

Ovaj prekid se generiše prilikom pritiskanja ili otpuštanja tastera. Vrednosti pritisnutih tastera smešta u bafer. Ukoliko je došlo do pritiskanja kombinacije tastera Ctrl-Alt-Del tada dolazi do resetovanja računara. Ukoliko je pritisnut taster Print Screen izaziva se prekid 05h, a ukoliko je pritisnuta kombinacija Ctrl-Break izaziva prekid 1Bh

INT 0Bh - Serial port COM2 / COM4

Ovaj prekid se dešava kada dođe do prijema, odnosno slanja znaka preko serijskog porta COM2/COM4.

INT 0Ch - Serial port COM1 / COM3

Ovaj prekid se dešava kada dođe do prijema, odnosno slanja znaka preko serijskog porta COM1/COM3.

5.3. BIOS Servisi

Kao sastavni deo svakog PC računara nalazi se i BIOS, u okviru koga se nalazi inicijalizacioni program, kao i program namenjen omogućavanju podizanja operativnog sistema. Osim ovih rutina, BIOS preuzima kontrolu nad jednim brojem prekida, kako bi bio u stanju da pruža neke od osnovnih servisa za pristup hardveru računara.

Bez upotrebe ovih servisa rad sa računarom bi bio veoma komplikovan i mukotrpan, a programi razvijeni za jednu konfiguraciju računara se ne bi mogli izvršavati na drugoj. Ovom prilikom ćemo navesti samo one prekide koji se najčešće koriste i postoje u svim varijantama BIOS-a. Naravno, kako se tehnologija menja tako se menja i sadržaj BIOS-a, i dodaju se dodatni servisi za podržavanje novih mogućnosti.

5.3.1. Sistemski servisi

INT 19h - Bootstrap loader

Nakon što se izvrši testiranje memorije i kompletne opreme računara, i utvrdi da je sve u redu, vrši se pozivanje ovog servisa. Njegov zadatak je da u memoriju računara sa diskete ili fiksnog diska učita kod koji služi za učitavanje operativnog sistema, i pokrene ga.

INT 11h - Equipment check

Ovaj servis iz memorijske zone dodeljene BIOS-u čita podatak o tekućoj opremi računara i smešta je u AX registar. Sadržaj je spakovan u sledećoj formi.

bitovi	značenje
0	postoje disketne jedinice
1	koprocesor instaliran
3,2	broj 64K mem. blokova (samo XT)
5,4	video mod 00 - EGA,VGA ili PGA 01 - 40x25 color 10 - 80x25 color 11 - 80x25 mono
7,6	broj disketnih jednica - 1
8	rezervisano
11,10,9	broj serijskih portova
12	port za palicu instaliran
13	rezervisano
15,14	broj paralelnih portova

INT 12h - Memory size

Kao rezultat poziva ovog servisa dobijamo u akumulatoru dužinu osnovne memorije. Treba naglasiti da je maksimalna vrednost ovde 1 MB, a da se sve iznad toga tumači kao proširena memorija.

5.3.2. Video izlaz

INT 10h - Video services

Ovaj servis nudi niz funkcija koje omogućuju ispis na ekran.

AH = 00h - Set video mode

Kao prvu operaciju prilikom rada sa video izlazom, treba podesiti video mod. Na taj način ćemo biti sigurni da radimo ispravan ispis, odnosno iscrtavanje na ekran.

Ovde ćemo navesti neke osnovne grafičke modove koji postoje kod VGA kartica, jer pretpostavljamo da praktično svaki korisnik PC računara ima grafičku karticu ovog tipa, a preostale vrste su davno nestale sa tržišta.

AL	rezolucija	boja	vrsta
00h	40x25	2	tekst
01h	40x25	16	tekst
02h	80x25	16 sivih	tekst
03h	80x25	16	tekst
04h	320x200	4	grafika
05h	320x200	4 sive	grafika
06h	640x200	2	grafika
07h	80x25	mono	tekst
0Dh	320x200	16	grafika
0Eh	640x200	16	grafika
0Fh	640x350	mono	grafika
10h	640x350	16	grafika
11h	640x480	2	grafika
12h	640x480	16	grafika
13h	320x200	256	grafika

Kao što ste primetili ovde nema nekih većih rezolucija, razlog za to je što su ti modovi nestandardni i kod svakog proizvođača se nalaze na drugom mestu. Međutim postoji i VESA standard, koji propisuje način korišćenja takvih modova, bez obzira na proizvođača.

Što se tiče tekstualnih modova treba reći da kod njih za svaki karakter ispisan na ekranu postoji i atribut, koji definiše njegovu boju, i boju pozadine na kojoj je ispisan. U memoriji se početak tekstualnog ekrana nalazi u segmenu B800h.

AH = 01h - Set cursor type

Ponekad je u aplikacijama korisno izmeniti oblik tekstualnog kursora, ili ga potpuno isključiti. Ovo je upravo funkcija koja nam to nudi. Gde u CH registru navodimo početnu linuju kursora unutar karaktera, a u CL krajnju. U gornjem delu CH se nalazi i atribut vidljivosti koji je 0 za vidljiv, a preostali za nevidljiv kursor, mada se zbog kompatibilnosti uzima 2.

Evo procedura koje vrše izmenu oblika tekstualnog kursora, i odgovaraju radu u tekstualnoj rezoluciji 80x25. Prva isključuje kursor, druga ga vraća u normalu, dok treća čini da kursor postane veličine karaktera.

```
nocursor  PROC
        mov ah,1
        mov ch,20h
        mov cl,00h
        int 10h
        ret
nocursor  ENDP

normalcur PROC
        mov ah,1
        mov ch,06h
        mov cl,07h
        int 10h
        ret
normalcur ENDP

solidcur  PROC
        mov ah,1
        mov ch,00h
        mov cl,07h
        int 10h
        ret
solidcur  ENDP
```

AH = 02h - Set cursor position

Tekst je potrebno ispisati na neku konkretnu poziciju, što se postiže ovom funkcijom. Ona na ulazu zahteva da se vrsta nalazi u DH, a kolona u DL U BH se očekuje broj tekstualne stranice, obično se postavlja na nulu.

Sledeća procedura postavlja kursor na odgovarajuću poziciju, koju predhodno moramo staviti na stek.

```
gotoxy PROC
     ARG X : BYTE,Y : BYTE = ARGLEN

     push bp
     mov bp,sp

     mov ah,2
     xor bh,bh
     mov dh,Y
     mov dl,X
     int 10h
     pop bp
     ret ARGLEN
gotoxy ENDP
```

AH = 03h - Read cursor position

Ova funkcija vraća podešavanja od predhodne dve procedure. Tako da na osnovu registara CH i CL možemo utvrditi tekući izgled kursora, a na osnovu DH i DL poziciju kursora na ekranu.

```
getx PROC
    mov ah,3
    int 10h
    xor ax,ax
    mov al,dl
    ret
getx ENDP

gety PROC
    mov ah,3
    int 10h
    xor ax,ax
    mov al,dh
    ret
gety ENDP
```

Ove dve funkcije vraćaju u akumulatoru poziciju kursora, i to prva kolonu, a druga vrstu.

AH = 05h - Select active display page

U pojedinim modovima, zavisno od količine memorije na video kartici, postoji mogućnost paralelnog korišćenja više grafičkih strana. Pri radu u tekstualnom režimu ovo i nije bitno, tako da se nećemo previše zadržavati. Treba reći samo da broj stranice na koju želimo da pređemo treba sa se nalazi u AL registru.

AH = 06h - Scroll up window

Kada radimo u tekstualnom režimu, možemo definisati veličinu prozora koji želimo skrolovati, u ovom slučaju na gore. Ova funkcija se ponajpre koristi za brisanje ekrana.

Ona očekuje da na ulazu dobije u AL broj linija koliko treba skrolovati dati prozor, ako je nula, znači da se traži brisanje. A prozor se definiše pomoću dve tačke, gde su CH,CL koordinate gornjeg levog ugla, a DH,DL donjeg desnog ugla. U BH se stavlja atribut koji treba da se postavlja u prazne linije.

```
clrscr PROC
     mov ax,0600h
     mov bh,07h
     xor  cx,cx
     mov dh,24
     mov dl,79
     int 10h
     mov ah,2
     xor bh,bh
     xor dx,dx
```

```
    int 10h
    ret
clrscr ENDP
```

Navedena procedura briše sadržaj tekstualnog ekrana u standardnoj rezoluciji 80x25.

AH = 07h - Scroll down window

Slično predhodnoj funkciji, s tom razlikom da se sada skrolovanje radržaja vrši prema dole. Parametri poziva su identični predhodnoj funkciji.

AH = 08h - Read character and attribute

Očitava znak na trenutnoj poziciji kursora, kao i njegov atribut. Kao parametar poziva potrebno je postaviti u BH registru stranicu za koju se traži očitavanje karaktera. Kao izlaz dobijamo u AL karakter, a u AH atribut.

Atribut može se sastoji od dva dela, viši deo bajta je boja pozadine, a niži deo boja teksta.

Broj	Boja
0h	crna
1h	plava
2h	zelena
3h	plavo-zelena
4h	crvena
5h	ljubičasta
6h	braon
7h	svetlo siva
8h	tamno siva
9h	svetlo plava
Ah	svetlo zelena
Bh	svetlo plavo-zelena
Ch	svetlo crvena
Dh	svetlo ljubičasta
Eh	žuta
Fh	bela

AH = 09h - Write character and attribute

Naravno postoji i funkcija koja omogućava ispis znaka na ekranu i to sa određenim atributom. Ovaj put se karakter nalazi u AL registru, u BH broj tekstualne strane, a u BL atribut, koji formiramo kako je to ranije navedeno. U registar CX se stavlja broj ponavljanja ispisa datog karaktera.

Svi znakovi se prikazuju uključujući i kontrolne.

```
putcc PROC
    ARG c : BYTE,boja : BYTE = ARGLEN
    push bp
    mov bp,sp

    mov ah,09h
    mov al,c
    mov bl,boja
    xor  bh,bh
    mov cx,1
    int 10h

    pop bp
    ret ARGLEN
putcc ENDP
```

AH = 0Ah - Write character only

Ispisuje znak, slično prethodnom, s tom razlikom da se ne navodi atribut znaka koji se ispisuje. Svi preostali parametri su isti.

AH = 0Eh - Write text in teletype mode

Ispisuje znak, ali tako da se kontrolni znakovi interpretiraju na odgovarajući način. U AL je znak a u BH broj stranice.

```
putc PROC
    ARG c : BYTE = ARGLEN
    push bp
    mov bp,sp

    mov ah,0Eh
    mov al,c
    int 10h

    pop bp
    ret ARGLEN
putc ENDP
```

Procedura koju smo naveli na ekran ispisuje znak koji joj je prosleđen kao ulazni parametar.

AH = 0Fh - Get current video state

Pozivom ove funkcije dobijamo u registru AL redni broj video moda u kome se trenutno nalazimo, u BH aktivnu stranicu, a u AH broj kolona teksta koji se može prikazati u datom video modu.

AH = 13h - Write string

Ispisuje niz znakova koji se nalaze od adrese definisanoj registarskim parom ES:BP, i to na koordinate DH,DL. Dužinu stringa stavljamo u CX, a u AL se nalazi mod ispisa. Vrednost 0 je

za niz koji se sastoji samo od znakova, a u BL je atribut i ne vrši se pomeranje kursora, 1 za isto samo sa pomeranjem kursora. Dok vrednost 2 i 3 označavaju da se string sastoji od para znak, atribut, a kursor se ne pomera za 2, a pomera za 3.

Ovim smo obradili sve funkcije ovog servisa koje su nam potrebne da bi smo mogli raditi u tekstualnom režimu rada i imati ispis na ekranu pod punom kontrolom.

Sledeći primer služi za demonstraciju predhodno napisanih rutina, i potrebno je sve njih ubaciti u primer koji sledi, i to posle odmah direktive za započinjanje kodnog segmenta.

```
.model small
.stack
.data
.code
start :
        call clrscr
        mov ax,'A'
        push ax
        call putc
        mov ax,74h
        push ax
        mov ax,'B'
        push ax
        call putcc
        mov ax,10
        push ax
        push ax
        call gotoxy
        mov ax,'C'
        push ax
        call putc

        mov ax,4c00h
        int 21h
end start
```

Kao što ste verovatno i primetili, telo primera se sastoji samo od poziva predhodno napisanih procedura, koje možete slobodno koristiti i proširivati u svojim narednim programima.

5.3.3. Kontrola tastature

INT 16h - Keyboard services

Da bi smo našim programima omogućili komunikaciju sa korisnikom, moramo dozvoliti da neke od podataka korisnik unosi preko tastature.

AH = 00h - Wait for keystroke and read

AH = 10h - Wait for keystroke and read ext.

Ove funkcije će zaustaviti izvršenje programa dogod se ne pritisne neki taster, a zatim će kao izlaz dati u registru AL ASCII kod pritisnutog tastera, a u AH registru će se nalaziti tzv. scan kod, čiju tabelu možete naći u dodacima na kraju priručnika.

Druga funkcija se odnosi na tastature sa proširenim setom tastera, takva je većina današnjih tastatura.

AH = 01h - Get keystroke status

AH = 11h - Get keystroke status ext.

Ove funkcije će nam u bitu nule status registra vratiti jedinicu ako taster nije pritisnut, ili nulu ako jeste pritisnut i u akumulatoru će se nalaziti sadržaj kao i pri pozivu predhodnih funkcija. Razlika je u tome što ove funkcije ne smatraju znak očitanim, i samim tim ga ne uklanjaju iz bafera.

Druga funkcija se odnosi na tastature sa proširenim setom tastera.

AH = 02h - Get keyboard status

AH = 12h - Get keyboard status ext.

Ponekad nam je potrebno da proverimo status kontrolnih tastera na tastaturi. Upravo za to nam služe ove funkcije. Ona u registru AL vraća status u zapakovanom formatu.

Druga po redu funkcija je i ovde zadužena za rad sa tastaturana sa proširenim setom tastera, i kao povratnu informaciju vraća i podatak u registru AH.

AL

Bit	Opis
7	Insert aktivan
6	CapsLock aktivan
5	NumLock aktivan
4	ScrollLock aktivan
3	Alt pritisnut
2	Ctrl pritisnut
1	levi Shift pritisnut
0	desni Shift pritisnut

AH

Bit	Opis

7	SysReq pritisnut
6	CapsLock aktivan
5	NumLock aktivan
4	ScrollLock aktivan
3	desni Alt pritisnut
2	desni Ctrl pritisnut
1	levi Alt pritisnut
0	levi Ctrl pritisnut

5.4. DOS servisi

Svaki operativni sistem koristi servise za neki vid komunikacije aplikacionih programa sa njim. DOS, kao prvi operativni sistem, za PC računare, je uveo ovaj servis za komunikaciju. Mnoge od funkcija, nasledio je upravo od svog neposlednog prethodnika CP/M-a, čija je DOS poboljšana verzija. Naravno, i kao zaseban OS on je rastao i razvijao se, te su se pojavljivale i dodatne opcije. Ovde ćemo objasniti one koje se češće koriste, a izostaviti one čija je upotreba suviše komplikovana ili pak veoma retka.

5.4.1. Uništenje procesa

INT 21 - DOS Function Dispatcher

AH = 4Ch - Terminate process

Ovo je funkcija koju smo već koristili od samog početka priručnika, gde je i rečeno da će biti objašnjena i njegova konkretna uloga. DOS, kao operativni sistem, zahteva da svaki program koji radi pod njim bude završen pozivom ove funkcije, osim u slučajevima kada se radi o pisanju tzv. rezidentnih programa. Ovaj poziv je neophodan da bi se kontrola izvršavanja vratila komandnom interpreteru, tj. samom operativnom sistemu.

Kao argument poziva ove funkcije u AL registru se navodi tzv. povratni kod, koji standardno ima vrednost nula, ako je program legalno završen, odnosno sadrži kod greške, ukoliko je do nje došlo prilikom izvršenja programa.

5.4.2. Rad sa prekidima

AH = 25h - Set interrupt vector

Da bi sam operativni sistem imao uvida u to koje prekidne rutine sam korisnik, ili neka od aplikacija redefiniše tokom svog rada, on nudi ovu funkciju koja služi za njihovo definisanje.

Kao parametre poziva u registru AL navodimo broj prekida koji želimo da preuzmemo, a u registarski par DS:DX stavljamo adresu prekidne rutine.

AH = 35h - Get vector

Ukoliko nam je potrebno da naši prekidi budu radni samo tokom izvršenja našeg programa, ili želimo iz naše prekidne rutine da pozivamo prethodnu, moramo pre svega upotrebiti ovu funkciju.

Njen jedini parametar poziva je registar AL, u kome se nalazi broj prekida čiju adresu želimo, dok se sama adresa nalazi u registarskom paru ES:BX.

5.4.3. Rad sa konzolom

BIOS je imao veoma "grubu" komunikaciju sa korisnikom, te su se tvorci DOS-a pobrinuli za to da se funkcije za komunikaciju pojednostave za korišćenje, i da im se daju nove mogućnosti. Kod DOS-a, kao i kod mnogih drugih operativnih sistema, se ekran i tastatura posmatraju kao datoteke, ovde one nose naziv STDIN za tastaturu, i STDOUT za ekran. Naravno iz komandne linije se saobraćaj može preusmeriti, pa se za ulaz može koristiti prava datoteka, a i za izlaz takođe.

Osim ova dva postoje još i uređaji STDAUX, koji predstavlja serijsku liniju, i STDPRN, koji pokazuje na štampač. Naravno tu je i STDERR, koji je vezan za ekran, i služi za prikazivanje grešaka prilikom rada.

INT 21 - DOS Function Dispatcher

AH = 01h - Console input with echo

AH = 07h - Direct input without echo

AH = 08h - Console input without echo

Poziv ovih funkcije prevodi program u stanje čekanja, iz koga može izaći jedino pritiskom na neki od tastera na tastaturi, odnosno čin se na uređaju STDIN pojavi karakter.

U slučaju poziva prve od funkcija, primljeni znak će odmah biti poslat na uređaj STDOUT, dok druge dve to neće učiniti. U slučaju poziva druge funkcije, neće se reagovati na pritisak

Ctrl-Break kombinacije, dok će u preostalim pozivima to biti slučaj.

AH = 02h - Console output

Znak se naravno može prikazati, odnosno poslati standardnom izlaznom uređaju, i to upravo pozivom ove funkcije. Znak koji želimo ispisati potrebno je predhodno smestiti u DL registar. Njeno izvršenje je moguće prekinuti i to kombinacijom Ctrl-Break.

AH = 03h - Wait for auxiliary device input

AH = 04h - Auxiliary output

Sada je i upotreba serijskih portova daleko olakšana od strane operativnog sistema. Njegovim naredbama se podešavaju parametri komunikacije, a pozivima ove dve funkcije, koje u potpunosti odgovaraju onima sa konzole se vrši komunikacija.

AH = 05h - Printer output

Naravno, tu je i neizbežni štampač, kome se sada obraćamo preko ove funkcije, a znak koji želimo ispisati smeštamo u DL registar. Štampanje je moguće prekinuti kombinacijom Ctrl-Break.

AH = 06h - Direct console I/O

Funkcije direktnog pristupa konzoli, što podrazumeva nemogućnost prekida, objedinjene su u ovoj DOS funkciji.

Ukoliko prilikom poziva u DL registar smestimo vrednost FFh, vršiće se očitavanje ulaza, i ukoliko je bit nule status registra na nuli u registru AL će nam se nalaziti očitani znak.

U slučaju da se prilikom poziva u DL registru nalazi neka druga vrednost, taj znak će biti prosleđen na standardni izlaz.

AH = 09h - Print string

Bilo bi nam veoma nezgodno kada bismo dugačke poruke morali prikazivati znak po znak. To bi pre svega oduzimalo dosta procesorskog vremena, a veoma bi zamaralo programera. Ova funkcija upravo rešava takav problem. Pre poziva je potrebno u registarski par DS:DX postaviti adresu stringa koji želimo ispisati. Moramo naglasiti da se string mora završiti karakterom ”$”. Evo jednog primera koji to demonstrira.

```
.model small
.stack
.data
poruka db "Ovo je ispis stringa !!!",13,10,"$"
.code
start : mov ax,@data
        mov ds,ax

        mov ah,9
        mov dx,OFFSET poruka
        int 21h

        mov ax,4c00h
        int 21h
end start
```

AH = 0Ah - Buffered keyboard input

Takođe nam je potrebna i funkcija koja bi nam omogućila unošenje odgovora dužih od jednog znaka. Ona zahteva da se navede adresa bafera u memoriji gde će unešeni podaci biti smešteni. Prvi bajt u baferu, pre poziva mora sadržavati maksimalan broj znakova koji se u bafer mogu smestiti, dok će drugi bajt po povratku sadržavati broj unetih znakova do pritiskanja tastera ENTER. Preostali bajtovi bafera će sadržavati unete znakove.

Evo jednog primera upotrebe ove funkcije.

```
.model small
.stack
.data
poruka   db "Unesite string : $"
poruka2 db 10,13,"Uneli ste : $"
bafer    db 20,0,21 dup(?)
.code
start : mov ax,@data
        mov ds,ax

        mov ah,9
        mov dx,OFFSET poruka
        int 21h

        mov ah,0Ah
        mov dx,OFFSET bafer
        int 21h

        mov ah,9
        mov dx,OFFSET poruka2
        int 21h

        xor bx,bx
        mov bl,bafer[1]
        mov bafer[bx+2],"$"

        mov ah,9
        mov dx,OFFSET bafer+2
        int 21h

        mov ax,4c00h
        int 21h
end start
```

AH = 0Bh - Check standard input status

Ukoliko želimo da proverimo da li je došlo do pritiskanja nekog tastera na tastaturi, to možemo učiniti pozivom ove funkcije. Ona će nam kao odgovor dati u registru AL nula ukoliko nijedan taster nije pritisnut, a 0FFh ukoliko jeste pritisnut.

AH = 0Ch - Clear keyboard buffer

Ukoliko pre unosa želimo isprazniti bafer tastature od nepročitanih podataka, recimo kada se unosi neka šifra, to ćemo učiniti pozivom ove funkcije, tako što ćemo joj kao parametar proslediti , recimo, 0xFFH u AL registru. Ukoliko u njega smestimo broj neke od funkcija unosa, ona će biti pozvana po pražnjenju bafera.

5.5. Aplikacioni servisi

Preostale servise preuzimaju pojedini programi, oni nadopunjuju već postojeće servise ili formiraju nove. Većina tih servisa je nestandardna, a nije redak slučaj kada se poklapaju. Jedan od češće korišćenih dodatnih servisa predstavlja servis za opsluživanje miša.

INT 33 - Mouse service

AX = 00h - Mouse reset

Pre početka upotrebe drajvera miša, potrebno ga je inicijalizovati, što se čini pozivom ove funkcije. Ako ovaj drajver ne postoji u AX će nam biti vraćena vrednost nula, a ukoliko postoji AX će sadržati FFFFh, a u BX registru će se nalaziti 0002h ili FFFFh ako miš ima dva tastera, 0000h ili 0003h ukoliko ima tri.

AX = 01h - Show mouse cursor

Prikazuje pokazivač miša na ekranu. Ako smo u tekstualnom režimu, biće u obliku kvadratića, a ukoliko smo u nekom od standardnih grafičkih modova, biće prikazana strelica.

AH = 02h - Hide mouse cursor

Prilikom ispisa, je potrebno uklonit kursor sa ekrana, jer ukoliko se to neučini a dođe do skrolovanja teksta na ekranu, imaćemo ostatke pokazivača miša na ekranu. Ovom funkcijom se pokazivač privremeno uklanja sa ekrana.

AX = 03h - Get position and button status

Da bi naš program reagovao na poziciju pokazivača miša, moramo biti i u mogućnosti da očitamo trenutnu poziciju i status tastera.

Ova funkcija kao rezultat vraća u registru CX kolonu, a u DX vrstu u kojoj se nalazi pokazivač. Treba obratiti pažnju na to da se u tekstualnim režimima rada vraća pozicija u pikselima, a ne u pozicijama karaktera, te je stoga te veličine potrebno podeliti sa 8.

U registru BX se vraća status pritisnutih tastera, i to tako da je bit nula zadužen za levi, bit 1 za desni taster, a bit 2 eventualno za srednji taster miša.

AX = 04h - Set mouse cursor position

Pozicioniranje pokazivača miša može biti korisno kada želimo da korisniku predočimo koja od navedenih opcija se podrazumeva, ili za simulaciju kretanja pokazivača miša po ekranu.

Pozicija se navodi u registrima CX, kolona, i DX, vrsta, i to u pikselima, tako da je u tekstualnim modovima potrebno koordinate ekranske pozicije pomnožiti sa 8.

AX = 05h - Get mouse button press

Želimo li da očitamo poziciju na kojoj je neki od tastera pritisnut, to ćemo očitati pomoću ove funkcije. Njen jedini parametar je broj tastera u registru BX, i to 0 za levi, 1 za desni ili 2 za srednji.

Kao rezultat u AX registru dobijamo trenutno stanje tastera, u BX registru broj pritiskanja ovog tastera od prošlog pozivanja ove funkcije. U registrima CX i DX se nalaze koordinate na kojima se nalazio pokazivač miša kada je taster poslednji put pritisnut.

AX = 06h - Get mouse button release

Može se detektovati i otpuštanje tastera, a ova funkcija se koristi identično kao i predhodna, jedina razlika je u tome što se u BX registru vraća broj otpuštanja datog tastera od prošlog pozivanja ove funkcije, kao i koordinate na kojima je tada bio pokazivač.

AX = 07h - Define horizontal cursor range

Ograničavanje kretanja pokazivača miša može biti od koristi, kada je potrebno navesti korisnika da koristi samo određene opcije na ekranu. Definisanje opsega promene koordinata se vrši zasebno za svaku koordinatu.

Ova funkcija ograničava kretanje pokazivača miša po horizontalnoj osi, i to na opseg od CX do DX, izraženom u pikselima.

AX = 08h - Define vertical cursor range

Slično tome ova funkija ograničava kretanje pokoazivača po vertikalnoj osi, gde je u CX minimalna vrednost, a u DX maksimalna vrednost.

Sledeći primer demonstrira upotrebu nekih od dosad navedenih funkcija. Iz primera se izlazi pritiskom na oba tastera miša istovremeno.

```
.model small
.stack
.data
.code
start :
        xor ax,ax
        int 33h
        or  ax,ax
        jz  kraj

        mov ax,1
        int 33h

        mov ax,7
        mov cx,10*8
        mov dx,70*8
        int 33h

        mov ax,8
        mov cx,5*8
        mov dx,20*8
        int 33h
ponovo:
        mov ax,3
        int 33h
        cmp bx,3
        jne ponovo

kraj:
        mov ax,2
        int 33h

        mov ax,4c00h
        int 21h
end start
```

5.6. Pisanje obrađivača prekida

Kada razvijamo neku novu periferiju, ili želimo da bolje iskoristimo već postojeće uređaje, moramo se odlučiti da napišemo i softversku podršku na najnižem nivou. Obrađivači prekida reaguju na događaje koje prouzrokuje hardver. Tu spadaju prekidi iz grupe prekida na nivou računara, koje smo već objasnili na početku.

Evo na šta treba obratiti pažnju prilikom pisanja obrađivača. Pre svega treba imati na umu da se prekid može desiti u bilo kom trenutku, tako da je na početku potrebno sačuvati na steku sadržaj svih registara, koji će biti izmenjeni u telu obrađivača. Naravno, te vrednosti treba na kraju ponovo vratiti ali u obrnutom redosledu, zbog načina rada sa stekom.

Kada se prekid desi, procesor će sačuvati sadržaj status registra, kao i adresu povratka iz prekida na steku. Kada počne izvršavanje, automatski će se zabraniti dešavanje prekida, a ukoliko želimo da ih dozvolimo potrebno je da to eksplicitno učinimo, mada je to retko kad potrebno i može da dovede do pojave problema. To što je procesor sačuvao sadržaj status registra nam omogućava da možemo slobodnije da se koristimo njegovim sadržajem.

Telo obrđivača je potrebno da bude kratko i da sadrži samo operacije čitanja, odnosno pisanja, na port i smeštanje podataka u bafere u memoriji, ili pak njihovo isčitavanje. Ovo je naročito bitno ako je procesor spor, a prekidi se često dešavaju.

5.7. Pisanje servisa

Za razliku od pisanja obređivača prekida, ovde se ne mora preterano voditi računa o brzini izvršavanja tela prekida, jer se prekidi pozivaju na zahtev korisnika. Ovde treba pre svega obratiti pažnju da se ne preuzme neki od već postojećih servisa, a ako se to učini namerno, kada se odradi naš deo koda, potrebno je pozvati i prošli servis.

Servis obično podrazumeva postojanje više funkcija, tako da osnova tela servisa čini samo testiranje operanada i pozivanje procedura koje odrađuju ostatak posla.

Ovde nije neophodno voditi računa o čuvanju sadržaja registara, jer se pozivanje vrši programski, samo je neophodno u dokumentaciji navesti koji su to registri koje ovaj poziv menja.

A. Spisak instrukcija

Pre nego što počnemo sa predstavljanjem pojedinačnih instrukcija procesora 8086/8088 definisaćemo format njihovog opisa radi lakšeg tumačenja.

Opis svake instrukcije otpočinje njenim imenom i punim nazivom, na engleskom, zatim se navodi tabela sa načinima njenog korišćenja. Ovde postoji više vrsta operanada koji se mogu navesti uz instrukciju.

konstanta konstanta_8 konstanta_16	Operande iz ove grupe treba zameniti neposrednim operandom navedene dužine.
registar registar_8 registar_16 seg_registar akumulator	Slično prethodnom, s tim da se operandi zamenjuju registrom odgovarajuće dužine. Umesto "seg_registar" mogu se navoditi samo segmentni registri, a umesto "akumulator" registri AX (za 16-bitni operand) ili AL (za 8-bitni operand).
memorija memorija_8 memorija_16	Potrebno je navesti memorijsku lokaciju na kojoj se nalazi podatak odgovarajuće dužine, 8 ili 16 bita.
near_proc far_proc	Operand kod instrukcije poziva potprograma, predstavlja 16-bitnu ili 32-bitnu (u formatu segment:ofset) adresu.
mem_pointer_16 mem_pointer_32 reg_pointer_16	Adresa memorijske lokacije na kojoj se nalazi parametar. Ako je u pitanju “reg_pointer”, registar se koristi kao pokazivač na lokaciju gde se nalazi operand.
string string_dest string_src	Adresa početka stringa. Bez eksplicitnog “_dest” ili “_src” može se raditi bilo o izvornoj, bilo o odredišnoj adresi.
pomeraj	Relativni pomeraj u odnosu na trenutnu lokaciju.
short_labela near_labela far_labela	Kod bezuslovnih skokova razlikuju se relativna (u 8-bitnom ili 16-bitnom opsegu) i apsolutna varijanta. Operandi su zapravo konstante, ali se navode po imenu.

Posle toga se navodi kako izvršavanje instrukcije utiče na sadržaj statusnog registra.

	Prazna kućica znači da instrukcija ne utiče na dati fleg.
X	Označava da se fleg postavlja u zavisnosti od rezultata instrukcije.
?	Označava da uticaj na fleg nije definisan.
0	Postavlja fleg na nulu.
1	Postavlja fleg na jedinicu.
r	Vrednost se dobija čitanjem sa steka

Nakon ove tabele sledi pseudokod.

AAA ASCII Adjust after Addition

Korišćenje	AAA

O	D	I	T	S	Z	A	P	C
?				?	?	X	?	X

```
IF ((AL & 0fh) > 09h) OR (AF = 1) THEN
  AL <- (AL + 6) & 0fh
  AH <- (AH + 1) & 0fh
  AF <- 1
  CF <- 1
ELSE
  AL <- AL & 0fh
  AF <- 0
  CF <- 0
ENDIF
```

AAD ASCII Adjust before Division

Korišćenje	AAD

O	D	I	T	S	Z	A	P	C
?				X	X	?	X	?

```
AL <- AH * 10 + AL
AH <- 0
```

AAM ASCII Adjust after Multiply

Korišćenje	AAM

O	D	I	T	S	Z	A	P	C
?				X	X	?	X	?

```
AH <- AL / 10
AL <- AL MOD 10
```

AAS ASCII Adjust after Substraction

Korišćenje	AAS

O	D	I	T	S	Z	A	P	C
?				?	?	X	?	X

```
IF ((AL & 0fh) > 09h) OR (AF = 1) THEN
  AL <- (AL - 6) & 0fh
  AH <- (AH - 1) & 0fh
  AF <- 1
  CF <- 1
ELSE
  AL <- AL & 0fh
  AF <- 0
  CF <- 0
ENDIF
```

ADC Add with Carry

Korišćenje	ADC registar,registar ADC registar,konstanta ADC akumulator,konstanta ADC registar,memorija ADC memorija,registar ADC memorija,konstanta

O	D	I	T	S	Z	A	P	C
X				X	X	X	X	X

dest <- dest + source + CF

ADD Addition

Korišćenje	ADD registar,registar
	ADD registar,konstanta
	ADD akumulator,konstanta
	ADD registar,memorija
	ADD memorija,registar
	ADD memorija,konstanta

O	D	I	T	S	Z	A	P	C
X				X	X	X	X	X

dest <- dest + source

AND Logical AND

Korišćenje	AND registar,registar AND registar,konstanta AND akumulator,konstanta AND registar,memorija AND memorija,registar AND memorija,konstanta

O	D	I	T	S	Z	A	P	C
0				X	X	?	X	0

dest <- dest & source

CALL Call Procedure

Korišćenje	CALL near_proc CALL far_proc CALL mem_pointer_16 CALL reg_pointer_16 CALL mem_pointer_32

O	D	I	T	S	Z	A	P	C

```
IF FAR_pointer THEN
  PUSH CS
  CS <- dest_segment
ENDIF
PUSH IP
IP <- dest_offset
```

CBW Convert Byte to Word

Korišćenje	CBW

O	D	I	T	S	Z	A	P	C

```
IF (AL < 80h) THEN
  AH <- 00h
ELSE
  AH <- ffh
ENDIF
```

CLC Clear Carry Flag

Korišćenje	CLC

O	D	I	T	S	Z	A	P	C
								0

CF <- 0

CLD Clear Direction Flag

Korišćenje	CLD

O	D	I	T	S	Z	A	P	C
	0							

DF <- 0

CLI Clear Interrupt-Enable Flag

Korišćenje	CLI

O	D	I	T	S	Z	A	P	C
		0						

IF <- 0

CMC Complement Carry Flag

Korišćenje	CMC

O	D	I	T	S	Z	A	P	C
								X

CF <- -CF

CMP Compare

Korišćenje	CMP registar,registar
	CMP registar,konstanta
	CMP akumulator,konstanta
	CMP registar,memorija
	CMP memorija,registar
	CMP memorija,konstanta

O	D	I	T	S	Z	A	P	C
X				X	X	X	X	X

SET_FLAGS_FROM_RESULT(dest - source)

CMPSB Compare String by Byte

Korišćenje	CMPSB

O	D	I	T	S	Z	A	P	C
X				X	X	X	X	X

CMP (DS:SI),(ES:DI)
IF (DF = 0)
  SI <- SI + 1
  DI <- DI + 1
ELSE
  SI <- SI - 1
  DI <- DI - 1
ENDIF

CMPSW Compare String by Word

Korišćenje	CMPSW

O	D	I	T	S	Z	A	P	C
X				X	X	X	X	X

CMP (DS:SI),(ES:DI)
IF (DF = 0)
  SI <- SI + 2
  DI <- DI + 2
ELSE
  SI <- SI - 2
  DI <- DI - 2
ENDIF

CWD Convert Word to Doubleword

Korišćenje	CWD

O	D	I	T	S	Z	A	P	C

IF (AX < 8000h) **THEN**
  DX <- 0000h
ELSE
  DX <- ffffh
ENDIF

DAA Decimal Adjust after Addition

Korišćenje	DAA

O	D	I	T	S	Z	A	P	C
?				X	X	X	X	X

IF (AL & 0fh) > 9 **OR** (AF = 1) **THEN**
  AL <- AL + 6
  AF <- 1
ELSE
  AF <- 0
ENDIF
IF (AL > 9fh) **OR** (CF = 1) **THEN**
  AL <- AL + 60h
  CF <- 1
ELSE
  CF <- 0
ENDIF

DAS Decimal Adjust after Substraction

Korišćenje	DAS

O	D	I	T	S	Z	A	P	C
?				X	X	X	X	X

```
IF ((AL & 0fh) > 9) OR (AF = 1) THEN
  AL <- AL - 6
  AF <- 1
ELSE
  AF <- 0
ENDIF
IF (AL > 9fh) OR (CF = 1) THEN
  AL <- AL - 60h
  CF <- 1
ELSE
  CF <- 0
ENDIF
```

DEC Decrement

Korišćenje	DEC registar_8 DEC registar_16 DEC memorija

O	D	I	T	S	Z	A	P	C
X				X	X	X	X	

```
dest <- dest - 1
```

DIV Divide, Unsigned

Korišćenje	DIV registar_8 DIV registar_16 DIV memorija_8 DIV memorija_16

O	D	I	T	S	Z	A	P	C
?				?	?	?	?	?

```
IF BYTE_SIZE THEN
  AL <- AX / source
  AH <- AX mod source
ELSE
  AX <- DX:AX / source
  DX <- DX:AX mod source
ENDIF
```

HLT Halt the Processor

Korišćenje	HLT

O	D	I	T	S	Z	A	P	C

IDIV Integer Divide, Signed

Korišćenje	IDIV registar_8 IDIV registar_16 IDIV memorija_8 IDIV memorija_16

O	D	I	T	S	Z	A	P	C
?				?	?	?	?	?

```
IF BYTE_SIZE THEN
  AL <- AX / source
  AH <- AX mod source
ELSE
  AX <- DX:AX / source
  DX <- DX:AX mod source
ENDIF
```

IMUL Integer Multiply, Signed

Korišćenje	IMUL registar_8 IMUL registar_16 IMUL memorija_8 IMUL memorija_16

O	D	I	T	S	Z	A	P	C
X				?	?	?	?	X

```
IF BYTE_SIZE THEN
  AX <- AL * source
ELSE
  DX:AX <- AX * source
ENDIF
```

IN Input Byte or Word

Korišćenje	IN akumulator, konstanta_8 IN akumulator, DX

O	D	I	T	S	Z	A	P	C

```
IF BYTE_SIZE THEN
  AL <- (port)  ; port = konstanta_8 ili port =
DX
ELSE
  AX <- (port)
ENDIF
```

INC Increment

Korišćenje	INC registar_8 INC registar_16 INC memorija

O	D	I	T	S	Z	A	P	C
X				X	X	X	X	

dest <- dest + 1

INT Interrupt

Korišćenje	INT konstanta_8

O	D	I	T	S	Z	A	P	C
		0	0					

PUSHF
IF <- 0
TF <- 0
CALL FAR (INT*4)

INTO Interrupt on Overflow

Korišćenje	INTO

O	D	I	T	S	Z	A	P	C
		0	0					

IF (OF = 1) THEN
  PUSHF
  IF <- 0
  TF <- 0
  CALL FAR (10h)
ENDIF

IRET Interrupt Return

Korišćenje	IRET

O	D	I	T	S	Z	A	P	C
r	r	r	r	r	r	r	r	r

POP IP
POP CS
POPF

JA Jump if Above

JNBE Jump if Not Below or Equal

Korišćenje	JA pomeraj JNBE pomeraj

O	D	I	T	S	Z	A	P	C

IF (ZF = 1) **AND** (CF = 0) **THEN**
  JMP SHORT pomeraj
ENDIF

JAE Jump if Above

JNB Jump if Not Below

JNC Jump if Not Carry

Korišćenje	JAE pomeraj JNB pomeraj JNC pomeraj

O	D	I	T	S	Z	A	P	C

IF (CF = 0) **THEN**
  JMP SHORT pomeraj
ENDIF

JB Jump if Below

JNAE Jump if Not Above or Equal

JC Jump if Carry

Korišćenje	JB pomeraj JNAE pomeraj JC pomeraj

O	D	I	T	S	Z	A	P	C

IF (CF = 1) **THEN**
  JMP SHORT pomeraj
ENDIF

JBE Jump if Below

JNA Jump if Not Above

Korišćenje	JBE pomeraj JNA pomeraj

O	D	I	T	S	Z	A	P	C

IF (ZF = 1) **OR** (CF = 1) **THEN**
  JMP SHORT pomeraj
ENDIF

JCXZ Jump if CX register is Zero

Korišćenje	JCXZ pomeraj

O	D	I	T	S	Z	A	P	C

IF (CX = 0) **THEN**
  JMP SHORT pomeraj
ENDIF

JE Jump if Equal

JZ Jump if Zero

Korišćenje	JE pomeraj JZ pomeraj

O	D	I	T	S	Z	A	P	C

IF (ZF = 1) **THEN**
  JMP SHORT pomeraj
ENDIF

JG Jump if Greater

JNLE Jump if Not Less or Equal

Korišćenje	JG pomeraj JNLE pomeraj

O	D	I	T	S	Z	A	P	C

IF (ZF = 0) **AND** (SF = OF) THEN
  JMP SHORT pomeraj
ENDIF

JGE Jump if Greater or Equal

JNL Jump if Not Less

Korišćenje	JGE pomeraj JNL pomeraj

O	D	I	T	S	Z	A	P	C

IF (SF = OF) THEN
  JMP SHORT pomeraj
ENDIF

JL Jump if Less

JNGE Jump if Not Greater or Equal

Korišćenje	JL pomeraj JNGE pomeraj

O	D	I	T	S	Z	A	P	C

IF (SF <> OF) THEN
  JMP SHORT pomeraj
ENDIF

JLE Jump if Less or Equal

JNG Jump if Not Greater

Korišćenje	JLE pomeraj JNG pomeraj

O	D	I	T	S	Z	A	P	C

IF (SF <> OF) **OR** (ZF = 1) THEN
  JMP SHORT pomeraj
ENDIF

JNE Jump if Not Equal

JNZ Jump if Not Zero

Korišćenje	JE pomeraj JZ pomeraj

O	D	I	T	S	Z	A	P	C

IF (ZF = 0) **THEN**
  JMP SHORT pomeraj
ENDIF

JNO Jump if No Overflow

Korišćenje	JNO pomeraj

O	D	I	T	S	Z	A	P	C

IF (OF = 0) **THEN**
  JMP SHORT pomeraj
ENDIF

JNP Jump if No Parity

JPO Jump if Parity Odd

Korišćenje	JNP pomeraj JPO pomeraj

O	D	I	T	S	Z	A	P	C

IF (PF=0) THEN
  JMP SHORT pomeraj
ENDIF

JNS Jump if No Sign

Korišćenje	JNS pomeraj

O	D	I	T	S	Z	A	P	C

IF (SF = 0) **THEN**
  JMP SHORT pomeraj
ENDIF

JO Jump if Overflow

Korišćenje	JO pomeraj

O	D	I	T	S	Z	A	P	C

IF (OF = 0) **THEN**
  JMP SHORT pomeraj
ENDIF

JP Jump if Parity

JPE Jump if Parity Even

Korišćenje	JP pomeraj JPE pomeraj

O	D	I	T	S	Z	A	P	C

IF (PF = 1) **THEN**
  JMP SHORT pomeraj
ENDIF

JS Jump if Sign

Korišćenje	JS pomeraj

O	D	I	T	S	Z	A	P	C

IF (SF = 1) **THEN**
  JMP SHORT pomeraj
ENDIF

JMP Jump Unconditionally

Korišćenje	JMP short_labela JMP near_labela JMP far_labela JMP mem_pointer_16 JMP reg_pointer_16 JMP mem_pointer_32

O	D	I	T	S	Z	A	P	C

LAHF Load Register AH from Flags

Korišćenje	LAHF

O	D	I	T	S	Z	A	P	C

AH_0 <- CF
AH_2 <- PF
AH_4 <- AF
AH_6 <- ZF
AH_7 <- SF

LDS Load pointer using DS

Korišćenje	LDS registar_16, memorija

O	D	I	T	S	Z	A	P	C

```
reg <- (source)
DS  <- (source + 2)
```

LEA Load Effective Address

Korišćenje	LEA registar_16, memorija

O	D	I	T	S	Z	A	P	C

```
dest <- offset(source)
```

LES Load pointer using ES

Korišćenje	LES registar_16, memorija

O	D	I	T	S	Z	A	P	C

```
reg <- (source)
ES  <- (source + 2)
```

LODSB Load String by Byte

Korišćenje	LODSB

O	D	I	T	S	Z	A	P	C

```
AL <- (DS:SI)
IF (DF = 0) THEN
  SI <- SI + 1
ELSE
  SI <- SI - 1
ENDIF
```

LODSW Load String by Word

Korišćenje	LODSW

O	D	I	T	S	Z	A	P	C

```
AL <- (DS:SI)
IF (DF = 0) THEN
  SI <- SI + 2
ELSE
  SI <- SI - 2
ENDIF
```

LOOP Loop on Count

Korišćenje	LOOP pomeraj

O	D	I	T	S	Z	A	P	C

```
CX <- CX - 1
IF (CX <> 0) THEN
  JMP short_label
ENDIF
```

LOOPE Loop while Equal

LOOPZ Loop while Zero

Korišćenje	LOOPE pomeraj LOOPZ pomeraj

O	D	I	T	S	Z	A	P	C

```
CX <- CX - 1
IF (CX <> 0) AND (ZF =1) THEN
  JMP short_label
ENDIF
```

LOOPNE Loop while Not Equal

LOOPNZ Loop while Not Zero

Korišćenje	LOOPNE pomeraj LOOPNZ pomeraj

O	D	I	T	S	Z	A	P	C

```
CX <- CX - 1
IF (CX <> 0) AND (ZF =0) THEN
  JMP short_label
ENDIF
```

MOV Move

Korišćenje	MOV registar, registar
	MOV memorija, akumulator
	MOV akumulator, memorija
	MOV memorija, registar
	MOV registar, memorija
	MOV registar, konstanta
	MOV memorija, konstanta
	MOV seg_registar, registar_16
	MOV seg_registar, memorija
	MOV registar_16, seg_registar
	MOV memorija, seg_registar

O	D	I	T	S	Z	A	P	C

dest <- source

MOVSB Move String by Byte

Korišćenje	MOVSB

O	D	I	T	S	Z	A	P	C

```
(ES:DI) <- (DS:SI)
IF (DF = 0) THEN
  SI <- SI + 1
  DI <- DI + 1
ELSE
  SI <- SI - 1
  DI <- DI - 1
ENDIF
```

MOVSW Move String by Word

Korišćenje	MOVSW

O	D	I	T	S	Z	A	P	C

```
(ES:DI) <- (DS:SI)
IF (DF = 0) THEN
  SI <- SI + 2
  DI <- DI + 2
ELSE
  SI <- SI - 2
  DI <- DI - 2
ENDIF
```

MUL Multiply, Unsigned

Korišćenje	MUL registar_8 MUL registar_16 MUL memorija_8 MUL memorija_16

O	D	I	T	S	Z	A	P	C
X				?	?	?	?	X

```
IF BYTE_SIZE THEN
  AX <- AL * source
ELSE
  DX:AX <- AX * source
ENDIF
```

NEG Negate

Korišćenje	NEG registar NEG memorija

O	D	I	T	S	Z	A	P	C
X				X	X	X	X	X

dest <- - dest

NOP No Operation

Korišćenje	NOP

O	D	I	T	S	Z	A	P	C

```
XCHG ax,ax
```

NOT Logical NOT

Korišćenje	NOT registar NOT memorija

O	D	I	T	S	Z	A	P	C

dest <- **NOT** dest

OR Logical OR

Korišćenje	OR registar,registar OR registar,konstanta OR akumulator,konstanta OR registar,memorija OR memorija,registar OR memorija,konstanta

O	D	I	T	S	Z	A	P	C
0				X	X	?	X	0

dest <- dest | source

OUT Output to Port

Korišćenje	OUT konstanta_8, akumulator OUT DX, akumulator

O	D	I	T	S	Z	A	P	C

IF BYTE_SIZE **THEN**
(port) <- AL
ELSE
(port) <- AX
ENDIF

POP Pop a Word from Stack

Korišćenje	POP registar_16 POP seg_registar POP memorija

O	D	I	T	S	Z	A	P	C

dest <- (SS:SP)
SP <- SP + 2

POPF Pop Flags from Stack

Korišćenje	POPF

O	D	I	T	S	Z	A	P	C
r	r	r	r	r	r	r	r	r

CF <- $(SS:SP)_0$
PF <- $(SS:SP)_2$
AF <- $(SS:SP)_4$
ZF <- $(SS:SP)_6$
SF <- $(SS:SP)_7$
TF <- $(SS:SP)_8$
IF <- $(SS:SP)_9$
DF <- $(SS:SP)_{10}$
OF <- $(SS:SP)_{11}$
SP <- SP + 2

PUSH Push a Word onto Stack

Korišćenje	PUSH registar_16 PUSH seg_registar PUSH memorija

O	D	I	T	S	Z	A	P	C

SP <- SP - 2
(SS:SP) <- source

PUSHF Push Flags onto Stack

Korišćenje	PUSHF

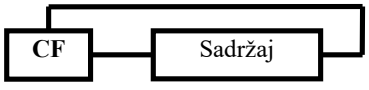
O	D	I	T	S	Z	A	P	C

SP <- SP - 2
$(SS:SP)_0$ <- CF
$(SS:SP)_2$ <- PF
$(SS:SP)_4$ <- AF
$(SS:SP)_6$ <- ZF
$(SS:SP)_7$ <- SF
$(SS:SP)_8$ <- TF
$(SS:SP)_9$ <- IF
$(SS:SP)_{10}$ <- DF
$(SS:SP)_{11}$ <- OF

RCL Rotate trough Carry Left

Korišćenje	RCL registar, 1 RCL registar, CL RCL memorija, 1 RCL memorija, CL

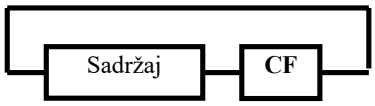
O	D	I	T	S	Z	A	P	C
X								X



RCR Rotate trough Carry Right

Korišćenje	RCR registar, 1 RCR registar, CL RCR memorija, 1 RCR memorija, CL

O	D	I	T	S	Z	A	P	C
X								X



REP Repeat

REPE Repeat while Equal

REPZ Repeat while Zero

Korišćenje	REP REPE REPZ

O	D	I	T	S	Z	A	P	C

```
WHILE (CX <> 0)
   string-operation
  CX <- CX - 1
  IF (ZF = 0) terminate_loop
END WHILE
```

REPNE Repeat while Not Equal

REPNZ Repeat while Not Zero

Korišćenje	REPNE REPNZ

O	D	I	T	S	Z	A	P	C

```
WHILE (CX <> 0)
   string-operation
  CX <- CX - 1
  IF (ZF <> 0) terminate_loop
END WHILE
```

RET Return from Procedure

Korišćenje	RET RET konstanta_16

O	D	I	T	S	Z	A	P	C

```
POP IP
IF FAR RETURN
  POP CS
ENDIF
IF (konstanta_16<>0)
  SP <- SP + konstanta_16
ENDIF
```

ROL Rotate Left

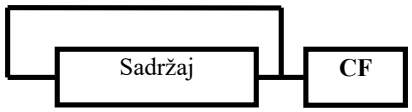
Korišćenje	ROL registar, 1 ROL registar, CL ROL memorija, 1 ROL memorija, CL

O	D	I	T	S	Z	A	P	C
X								X

ROR Rotate Right

Korišćenje	ROR registar, 1 ROR registar, CL ROR memorija, 1 ROR memorija, CL

O	D	I	T	S	Z	A	P	C
X								X



SAHF Store Register AH into Flags

Korišćenje	SAHF

O	D	I	T	S	Z	A	P	C
				X	X	X	X	X

CF <- AH_0
PF <- AH_2
AF <- AH_4
ZF <- AH_6
SF <- AH_7

SAL Shift Arithmetic Left

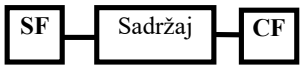
Korišćenje	SAL registar, 1 SAL registar, CL SAL memorija, 1 SAL memorija, CL

O	D	I	T	S	Z	A	P	C
X				X	X	?	X	X

SAR Shift Arithmetic Right

Korišćenje	SAR registar, 1 SAR registar, CL SAR memorija, 1 SAR memorija, CL

O	D	I	T	S	Z	A	P	C
X				X	X	?	X	X



SBB Sub with Borrow

Korišćenje	SBB registar, registar SBB registar, konstanta SBB akumulator, konstanta SBB registar, memorija SBB memorija, registar SBB memorija, konstanta

O	D	I	T	S	Z	A	P	C
X				X	X	X	X	X

dest <- dest - source - CF

SCASB Scan String by Byte

Korišćenje	SCASB

O	D	I	T	S	Z	A	P	C

CMP AL,(ES:DI)
IF (DF = 0) **THEN**
  DI <- DI + 1
ELSE
  DI <- DI - 1
ENDIF

SCASW Scan String by Word

Korišćenje	SCASW

O	D	I	T	S	Z	A	P	C

CMP AL,(ES:DI)
IF (DF = 0) **THEN**
  DI <- DI + 2
ELSE
  DI <- DI - 2
ENDIF

SHL Shift Logical Left

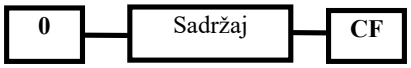
Korišćenje	SHL registar, 1 SHL registar, CL SHL memorija, 1 SHL memorija, CL

O	D	I	T	S	Z	A	P	C
X				X	X	?	X	X

SHR Shift Logical Right

Korišćenje	SHR registar, 1 SHR registar, CL SHR memorija, 1 SHR memorija, CL

O	D	I	T	S	Z	A	P	C
X				X	X	?	X	X



STC Set Carry Flag

Korišćenje	STC

O	D	I	T	S	Z	A	P	C
								1

CF <- 1

STD Set Direction Flag

Korišćenje	STD

O	D	I	T	S	Z	A	P	C
	1							

DF <- 1

STI Set Interrupt-Enable Flag

Korišćenje	STI

O	D	I	T	S	Z	A	P	C
		1						

IF <- 1

STOSB Store String by Byte

Korišćenje	STOSB

O	D	I	T	S	Z	A	P	C

(ES:DI) <- AL
IF (DF = 0) **THEN**
  DI <- DI + 1
ELSE
  DI <- DI - 1
ENDIF

STOSW Store String by Word

Korišćenje	STOSW

O	D	I	T	S	Z	A	P	C

(ES:DI) <- AL
IF (DF = 0) **THEN**
  DI <- DI + 2
ELSE
  DI <- DI - 2
ENDIF

SUB Substract

Korišćenje	SUB registar, registar
	SUB registar, konstanta
	SUB akumulator, konstanta
	SUB registar, memorija
	SUB memorija, registar
	SUB memorija, konstanta

O	D	I	T	S	Z	A	P	C
X				X	X	X	X	X

dest <- dest - source

TEST Test

Korišćenje	TEST registar, registar TEST registar, konstanta TEST akumulator, konstanta TEST registar, memorija TEST memorija, konstanta

O	D	I	T	S	Z	A	P	C
0				X	X	?	X	0

SET_FLAGS_FROM_RESULT(dest & source)
CF <- 0
OF <- 0

WAIT Wait

Korišćenje	WAIT

O	D	I	T	S	Z	A	P	C

XCHG Exchange Registers

Korišćenje	XCHG akumulator, registar_16 XCHG registar, registar XCHG memorija, registar

O	D	I	T	S	Z	A	P	C

dest <-> source

XLAT Translate

Korišćenje	XLAT

O	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**

AL <- (BX + AL)

XOR Logical XOR

Korišćenje	XOR registar,registar XOR registar,konstanta XOR akumulator,konstanta XOR registar,memorija XOR memorija,registar XOR memorija,konstanta

O	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**
0				X	X	?	X	0

dest <- dest ^ source

B. ASCII tabela

Ovde je data ASCII tabela, u kojoj se nalaze paralelno ispisane vrednosti u decimalnom, oktalnom, heksadecimalnom formatu, kao i sam prikaz ASCII karaktera, odnosno njegov naziv, ukoliko se radi o kontrolnom karakteru.

Dec	Octal	Hex	ASCII
0	000	00	NUL
1	001	01	SOH
2	002	02	STX
3	003	03	ETX
4	004	04	EOT
5	005	05	ENQ
6	006	06	ACK
7	007	07	BEL
8	010	08	BS
9	011	09	HT
10	012	0A	LF
11	013	0B	VT
12	014	0C	FF
13	015	0D	CR
14	016	0E	SO
15	017	0F	SI
16	020	10	DLE
17	021	11	DC1
18	022	12	DC2
19	023	13	DC3
20	024	14	DC4
21	025	15	NAK
22	026	16	SYN
23	027	17	ETB
24	030	18	CAN
25	031	19	EM
26	032	1A	SUB
27	033	1B	ESC
28	034	1C	FS
29	035	1D	GS
30	036	1E	RS
31	037	1F	US
32	040	20	SP
33	041	21	!
34	042	22	"
35	043	23	#
36	044	24	$
37	045	25	%
38	046	26	&
39	047	27	'
40	050	28	(
41	051	29	)
42	052	2A	*
43	053	2B	+
44	054	2C	,
45	055	2D	-
46	056	2E	.
47	057	2F	/
48	060	30	0
49	061	31	1
50	062	32	2
51	063	33	3
52	064	34	4
53	065	35	5
54	066	36	6
55	067	37	7
56	070	38	8
57	071	39	9
58	072	3A	:
59	073	3B	;
60	074	3C	<
61	075	3D	=
62	076	3E	>
63	077	3F	?
64	100	40	@
65	101	41	A
66	102	42	B
67	103	43	C
68	104	44	D
69	105	45	E
70	106	46	F
71	107	47	G
72	110	48	H
73	111	49	I
74	112	4A	J
75	113	4B	K
76	114	4C	L
77	115	4D	M
78	116	4E	N
79	117	4F	O
80	120	50	P
81	121	51	Q
82	122	52	R
83	123	53	S

Dec	Octal	Hex	ASCII
84	124	54	T
85	125	55	U
86	126	56	V
87	127	57	W
88	130	58	X
89	131	59	Y
90	132	5A	Z
91	133	5B	[
92	134	5C	\
93	135	5D	]
94	136	5E	^
95	137	5F	_
96	140	60	`
97	141	61	a
98	142	62	b
99	143	63	c
100	144	64	d
101	145	65	e
102	146	66	f
103	147	67	g
104	150	68	h
105	151	69	i
106	152	6A	j
107	153	6B	k
108	154	6C	l
109	155	6D	m
110	156	6E	n
111	157	6F	o
112	160	70	p
113	161	71	q
114	162	72	r
115	163	73	s
116	164	74	t
117	165	75	u
118	166	76	v
119	167	77	w
120	170	78	x
121	171	79	y
122	172	7A	z
123	173	7B	{
124	174	7C	|
125	175	7D	}
126	176	7E	~
127	177	7F	

C. Scan kodovi

Ovi kodovi se koriste u radu sa INT 16, BIOS podrškom za tastaturu. Podelićemo ih u više grupa radi lakšeg korišćenja.

C.1. Abeceda

Key	Norm.	+ Shift	+ Ctrl	+ Alt
A	1E61	1E41	1E01	1E00
B	3062	3042	3002	3000
C	2E63	2E42	2E03	2E00
D	2064	2044	2004	2000
E	1265	1245	1205	1200
F	2166	2146	2106	2100
G	2267	2247	2207	2200
H	2368	2348	2308	2300
I	1769	1749	1709	1700
J	246A	244A	240A	2400
K	256B	254B	250B	2500
L	266C	264C	260C	2600
M	326D	324D	320D	3200
N	316E	314E	310E	3100
O	186F	184F	180F	1800
P	1970	1950	1910	1900
Q	1071	1051	1011	1000
R	1372	1352	1312	1300
S	1F73	1F53	1F13	1F00
T	1474	1454	1414	1400
U	1675	1655	1615	1600
V	2F76	2F56	2F16	2F00
W	1177	1157	1117	1100
X	2D78	2D58	2D18	2D00
Y	1579	1559	1519	1500
Z	2C7A	2C5A	2C1A	2C00

C.2. Simboli

Key	Norm.	+ Shift	+ Ctrl	+ Alt
-	0C2D	0C5F	0C1F	8200
=	0D3D	0D2B		8300
[	1A5B	1A7B	1A1B	1A00
]	1B5D	1B7D	1B1D	1B00
;	273B	273A		2700
'	2827	2822		
`	2960	297E		
\	2B5C	2B7C	2B1C	2600
,	332C	333C		
.	342E	343E		
/	352F	353F		

C.3. Brojevi

Key	Norm.	+ Shift	+ Ctrl	+ Alt
1	0231	0221		7800
2	0332	0340	0300	7900
3	0433	0423		7A00
4	0534	0524		7B00
5	0635	0625		7C00
6	0736	075E	071E	7D00
7	0837	0826		7E00
8	0938	092A		7F00
9	0A39	0A28		8000
0	0B30	0B29		8100

C.4. Funkcijski tasteri

Key	Norm.	+ Shift	+ Ctrl	+ Alt
F1	3B00	5400	5E00	6800
F2	3C00	5500	5F00	6900
F3	3D00	5600	6000	6A00
F4	3E00	5700	6100	6B00
F5	3F00	5800	6200	6C00
F6	4000	5900	6300	6D00
F7	4100	5A00	6400	6E00
F8	4200	5B00	6500	6F00
F9	4300	5C00	6600	7000
F10	4400	5D00	6700	7100
F11	8500	8700	8900	8B00
F12	8600	8800	8A00	8C00

C.5. Specijalni tasteri

Key	Norm	+Shift	+Ctrl	+Alt
BackSpace	0E08	0E08	0E7F	0E00
Del	5300	532E	9300	A300
Down	5000	5032	9100	A000
End	4F00	4F31	7500	9F00
Enter	1C0D	1C0D	1C0A	A600
Esc	011B	011B	011B	0100
Home	4700	4737	7700	9700
Ins	5200	5230	9200	A200
Keypad 5		4C35	8F00	
Keypad *	372A		9600	3700
Keypad -	4A2D	4A2D	8E00	4A00
Keypad +	4E2B	4E2B		4E00
Keypad /	352F	352F	9500	A400
Left	4B00	4B34	7300	9B00
PgDn	5100	5133	7600	A100
PgUp	4900	4939	8400	9900
PrtSc			7200	
Right	4D00	4D36	7400	9D00
SpaceBar	3920	3920	3920	3920
Tab	0F09	0F00	9400	A500
Up	4800	4838	8D00	9800

Literatura

- "INTEL 8086 Programmers Reference Manual", Intel Corporation, 1987.
- Miroslav Hajduković, "Organizacija računara", drugo ispravljeno autorsko izdanje, Novi Sad, 1996.
- Miroslav Hajduković "Operacioni sistemi", Nuron - Tehnička serija, Novi Sad, 1999.
- Dušan Malbaški, Danilo Obradović "Osnovne strukture podataka", Tehnički fakultet "Mihajlo Pupin", Zrenjanin, 1995.
- Ralf Brow "Interrupt List, Release 53", 1997.