Praktikum računarskih vežbi

za predmet

Arhitektura računara

Autori:
Žarko Živanov
Ivan Nejgebauer
Lazar Stričević
Miroslav Hajduković

Novi Sad, 2019.

Sadržaj

1. Uvod u korišćenje *Linux* operativnog sistema

Linux [1] je jezgro operativnog sistema čiji je razvoj započeo Linus Torvalds 1991. godine kao student Univerziteta u Helsinkiju. Pored jezgra, za funkionisanje jednog operativnog sistema neophodni su sistemski alati i bibliotetke. *Linux* jezgro se najčešće koristi zajedno sa alatima i bibliotekama iz ***GNU*** projekta, pa se stoga ova kombinacija jednim imenom naziva *GNU/Linux* i u ovom trenutku predstavlja moderan i popularan operativni sistem. Često se u javnosti umesto punog imena koristi skraćeno - *Linux* kada može da se razume da li se govori o celokupnom operativnom sistemu ili samo o njegovom jezgru. U daljem tekstu ćemo zato najčešće koristiti ovakav skraćeni naziv.

Postoji mnoštvo različitih verzija *Linux*-a. Za razliku od komercijalnih operativnih sistema koje kontroliše određena matična kompanija, *Linux* je slobodan za distribuiranje i korišćenje. Tako je u *Linux* svetu situacija poprilično različita od onog što se dešava u svetu vlasničkog (*proprietary*) softvera: brojne kompanije, organizacije i pojedinci su razvili svoje sopstvene "verzije" *Linux* operativnog sistema, koje se još nazivaju i **distribucije** [3]. Postoje verzije *Linux*-a koje su razvijene npr. za računare koji kontrolišu obiman mrežni saobraćaj (npr. *web server*), zatim neke koje su razvijene za bezbedan rad sa poverljivim informacijama, itd. Neke verzije su osmišljene za različite hardverske platforme, kao što su npr. *ARM*, *PowerPC* ili *Sun UltraSPARC*.

1.1. Izbor platforme za rad

Na kursu iz predmeta Arhitektura računara ćemo se baviti arhitekturom PC platforme zasnovane na *Intel*-ovoj x86 familiji procesora, a kao sistemska osnova biće korišćena distribucija *Linux*-a namenjena ovoj platformi. Razlozi za izbor x86 PC za našu radnu platformu leži pre svega u njenoj opštoj dostupnosti.

Linux je izabran za operativni sistem na kome će kurs biti zasnovan zbog toga što zadovoljava sve zahteve koje nastava iz ovog predmeta podrazumeva, a drugi razlog je vezan za licencu pod kojom je dostupan. *Linux* je objavljen pod GPL (*General Public License*) licencom [2], što znači:

- da je korišćenje slobodno u nastavi i van nje,
- da je izvorni kôd svima dostupan i da se može slobodno koristiti,
- da se kopije mogu slobodno distribuirati, i
- da se kôd može menjati po potrebi i tako izmenjen dalje distribuirati.

1.2. Prijavljivanje na sistem

Na početku rada se od korisnika očekuje da unese svoje korisničko ime i lozinku, čime se prijavljuje na sistem (tzv. **logovanje**). Treba obratiti pažnju na to da *Linux* pravi razliku između malih i velikih slova (on je ***case sensitive***). Tako, na primer, imena "pera" i "Pera" označavaju dva različita korisnika. Nakon uspešnog prijavljivanja, operativni sistem je spreman za rad.

Na kraju rada neophodna je odjava sa operativnog sistema. To se obavlja izborom opcije *Logout* iz sistemskog menija. Odjavljivanje je neophodno, najviše zbog toga što je korisnicima pod *Linux*-om obično dozvoljeno da rukuju samo svojim fajlovima. Ako se računar ostavi bez odjavljivanja, nekom drugom se daje mogućnost da pristupi fajlovima neodjavljenog korisnika.

1.3. Rukovanje *shell*-om

Komunikaciju sa korisnikom putem komandne linije omogućuje program koji se zove ***shell***, i koji ima ulogu interpretera komandi. U *Linux* distribucijama najčešće korišćeni *shell* je ***bash*** (*Bourne Again Shell*), koji je obično podrazumevani *shell* i koji će se i ovde koristiti. Za

interaktivni rad sa shell-om je neophodan terminal, odnosno terminal emulator. Terminal emulator se pokreće tako što se klikne na njegovu ikonicu koja ima oblik terminala i koja se obično nalazi na jednom od ekranskih panela. Pokretanje terminal emulatora automatski dovodi do pokretanja *bash*-a.

Nakon pokretanja *bash*-a, ispisuje se njegov odziv, odnosno ***prompt***. Podrazumevani izgled *prompt*-a je:

```
user@computer:~/download$
```

Reč `user` označava korisničko ime, `computer` je naziv računara na kome je korisnik prijavljen, `~/download` je putanja do tekućeg direktorijuma, a `$` je oznaka da je prijavljen običan korisnik. Kada je prompt ispisan na ekranu, *bash* je spreman da prihvati komandu.

Važno je uočiti da *bash* pravi razliku između malih i velikih slova. *bash* pamti komande koje je korisnik ranije kucao (tzv. **istorija komandi**). Broj zapamćenih komandi se može podesiti. Sve ranije zapamćene komande se mogu pregledati pritiskom na kursorske tastere ↑ i ↓.

Ekran terminala se brzo popuni i tada se stari sadržaj gubi zbog pomeranja linija teksta na gore. Ukoliko je potrebno videti prethodne sadržaje ekrana, to se može postići pritiskom na *Shift*+*PgUp*, odnosno *Shift*+*PgDn*. Alternativno, mogu se koristiti i *Shift*+↑, odnosno *Shift*+↓.

Još jedna korisna mogućnost *bash*-a je **kompletiranje naziva**. Naime, dovoljno je otkucati prvih nekoliko slova neke komande i pritisnuti taster *Tab.* Nakon toga, ukoliko postoji samo jedna mogućnost za komandu, ona će biti ispisana, a ako postoji više, tada će biti ispisane alternative, pa korisnik može odabrati šta mu treba. Na primer, ako se otkuca “hi”, pa se pritisne *Tab*, na ekranu će se ispisati nekoliko komandi koje počinju slovima “hi”. Ukoliko se otkuca i slovo “s”, nakon pritiska na *Tab* će se na ekranu ispisati `history`, pošto je to jedina komanda koja počinje sa “his”. Pored toga što se može koristiti za nazive komandi i programa, kompletiranje naziva radi i sa nazivima direktorijuma i fajlova. Kada se kuca naziv komande (prva reč u liniji), pretražuju se podrazumevani direktorijumi (videti sledeće poglavlje), dok se prilikom kucanja parametara komandi pretražuje tekući direktorijum.

Postoji više načina da se završi rad *bash*-a. Jedan od najsigurnijih je zadavanje komande **exit**, a jedan od najkraćih je pritisak na *ctrl*+*d* u praznoj komandnoj liniji.

1.4. Organizacija direktorijuma

Osnovni direktorijum u *Linux*-ovom fajl sistemu je `root`, ili korenski direktorijum i označava se sa “`/`” (*slash*). Svi drugi direktorijumi se nalaze ispod njega. Od direktorijuma koji se nalaze u `root`-u, za obične korisnike je značajno nekoliko njih. Direktorijum `etc` sadrži razne konfiguracione fajlove, većinom u tekstualnom obliku. Direktorijum `bin` sadrži osnovne sistemske programe. Direktorijum `usr` sadrži korisničke programe i sve što je potrebno za njihovo korišćenje (slike, dokumentacija, razni pomoćni i konfiguracioni fajlovi).

Za razliku od npr. *Windows*-a, kod *Linux*-a ne postoje posebne oznake disk uređaja (`A:`, `C:`, itd.). Direktorijum `/mnt` (u novijim distribucijama direktorijum `/media`) u sebi sadrži putanje do uređaja koji se nalaze u sistemu, kao što su *floppy* diskovi, *CD-ROM*, hard diskovi, *flash* memorije... Ovo je samo najčešće korišćeni direktorijum za uvezivanje uređaja (*mount*-ovanje), pošto se uređaji mogu povezati na bilo koje mesto u hijerarhiji direktorijuma.

Za korisnike je ipak najbitniji direktorijum `/home` u kome se nalaze lični fajlovi za svakog od korisnika koji imaju nalog na sistemu. Svaki korisnik u ovom direktorijumu ima svoj poddirektorijum i obično je to jedino mesto u kome korisnici imaju pravo upisa, pored direktorijuma `/tmp`. Svi ostali direktorijumi se mogu ili samo čitati, ili im je pristup potpuno onemogućen.

U svakom direktorijumu se nalaze i dva specijalna poddirektorijuma: “`.`” označava tekući direktorijum, dok “`..`” označava nadređeni direktorijum (*parent*), koji se nalazi iznad tekućeg. Oba se često koriste u radu iz komandne linije. I korisnikov `home` direktorijum ima posebnu oznaku “`~`” (tilda), koja se može koristiti za pristup iz komandne linije.

Kada se navodi putanja do nekog direktorijuma, nazivi pojedinih poddirektorijuma se razdvajaju sa "/". Ovaj znak takođe mora stojati i između poslednjeg naziva poddirektorijuma i naziva fajla.

Kada se od *Linux*-a zatraži da pokrene neki izvršni fajl, ukoliko se ne navede tačna putanja do njega, pretražiće se podrazumevani direktorijumi za izvršne fajlove. Spisak ovih direktorijuma se nalazi u promenljivoj okruženja (*environment variable*) `PATH`. Treba napomenuti da se tekući direktorijum najčešće **ne nalazi** u ovom spisku, tako da, ako treba pokrenuti program koji se nalazi u tekućem direktorijumu (a koji se ne nalazi u `PATH`-u), potrebno je kucati

```
./naziv_programa
```

1.5. Osnovne komande *bash*-a

Kretanje po stablu direktorijuma i listanje fajlova

Komanda koja služi za kretanje po stablu direktorijuma je `cd` (*change directory*). Ukoliko se otkuca samo `cd` bez parametara, `home` direktorijum korisnika postaje tekući. Kao parametar se može navesti relativna ili apsolutna putanja do direktorijuma u koji se želi preći. Na primer, ako je tekući direktorijum

```
/home/pera/prvi/
```

a želi se preći u

```
/home/pera/drugi/
```

može se kucati apsolutna putanja

```
cd /home/pera/drugi/
```

ili relativna

```
cd ../drugi/
```

Treba napomenuti da se u svakom trenutku kucanja putanje može koristiti kompletiranje naziva pomoću tastera *Tab*, čime se vreme kucanja može znatno skratiti.

Ukoliko je potrebno saznati punu putanju do tekućeg direktorijuma, to se može postići kucanjem komande `pwd` (*print working directory*).

Prikaz sadržaja tekućeg direktorijuma (ili direktorijuma koji se navede kao parametar) omogućuje komanda `ls` (*list*). Ova komanda ima puno opciija, od kojih će ove biti pomenute samo neke. Opcija `-a` omogućuje da se prikažu i skriveni fajlovi i direktorijumi koji se standardno ne prikazuju (njihova imena počinju tačkom), a opcija `-l` (malo slovo L) omogućuje detaljniji prikaz podataka o fajlovima i direktorijumima. Ono što će prikazati komanda `ls` se može dodatno filtrirati upotrebom džoker znakova:

Džoker	***Značenje***
`*`	Zamenjuje bilo koji broj bilo kojih znakova
`?`	Zamenjuje tačno jedan bilo koji znak
`[]`	Slično džokeru `?`, samo što se zamnjuje tačno određeni skup znakova. Na primer, [a-dps] bi bila zamena za jedan od sledećih znakova: a,b,c,d,p,s

Tabela 1: Džoker znaci

Tako, na primer, ako treba izlistati sve fajlove u tekućem direktorijumu koji počinju sa "pr", komanda bi bila:

```
ls pr*
```

Ako treba izlistati sve fajlove u tekućem direktorijumu koji počinju sa “a”, “b” ili “c”, komanda bi bila:

```
ls [a-c]*
```

Upotreba džoker znakova nije ograničena samo na `ls` komandu. Oni se mogu koristiti sa bilo kojom komandom koja kao parametar prihvata naziv fajla ili direktorijuma.

Za većinu komandi se može dobiti spisak svih njenih opcija kucanjem komande i argumenta `--help`. Drugi način dobijanja (detaljnije) pomoći za neku komandu je komanda `man` (*manual*). Na primer, za komandu `ls` uputstvo se može dobiti kucanjem `man ls`. Pregledanje uputstva se vrši kursorskim tasterima, dok se izlazak postiže pritiskom na taster *q*. Ukoliko se u uputstvu traži neka posebna reč, treba otkucati */*, za pretragu unapred, odnosno *?* za pretragu unazad, pa nakon toga reč koja se traži. Ponovna pretraga za istom rečju se ostvaruje pritiskom na *n*, odnosno *Shift*+*n* za pretragu unazad, ili kucanjem */*, odnosno *?*, nakon čega treba pritisnuti *Enter*.

Rukovanje fajlovima i direktorijumima

Nazivi fajlova pod *Linux*-om se mogu sastojati od slova (velikih i malih), brojeva i specijalnih znakova, kao što su “`-`”, ”`_`” ili “`.`”. Iako se u nazivu mogu naći i ostali specijalni znakovi, kao što su razmak, “`;`”, “`&`”, “`?`”, “`|`”, “`*`”, “`[`”, “`]`”, “`<`” i “`>`”. Ipak upotreba ovih znakova se ne preporučuje, jer je korišćenje fajlova sa ovakvim nazivima otežano (npr. kada je potrebno uneti njihovo ime na tastaturi), ili čak nemoguće na nekim drugim sistemima. Ukoliko se takvi znakovi ipak koriste u nazivu, tada se naziv mora staviti unutar znakova navoda, ili, alternativno, ispred specijalnog znaka treba staviti “`\`” (*backslash*). Razlog za ovo je što mnogi specijalni znaci imaju posebno značenje i upotrebu (neki su već navedeni, a o nekima će biti reči kasnije), a stavljanjem znaka “`\`” ispred specijalnog znaka se kaže sistemu da ga tretira kao običan znak.

Jedna od osnovnih operacija pri radu sa fajlovima je njihovo kopiranje. To se postiže komandom `cp` (*copy*). Osnovni oblik ove komande je:

```
cp šta_se_kopira gde_se_kopira
```

Ukoliko se kopira samo jedan fajl, tada se umesto parametra `gde_se_kopira` može navesti drugo ime za fajl, a u suprotnom mora biti naziv direktorijuma. Na primer, ako treba kopirati sve fajlove sa ekstenzijom `.xyz` iz `home` direktorijuma u tekući direktorijum, komanda bi bila:

```
cp ~/*.xyz ./
```

Ukoliko treba obrisati jedan ili više fajlova, koristi se komanda `rm` (*remove*). Na primer, ako treba obrisati sve fajlove koji počinju sa `aaa`, komanda bi bila:

```
rm aaa*
```

Neretko se komanda `rm` koristi sa opcijom `-i`, čiji je efekat da se korisnik pita za brisanje svakog fajla ponaosob, kako bi se izbeglo slučajno brisanje fajlova. Ako takvih fajlova ima više, korisnik će biti pitan za svaki. Ako svi fajlovi sigurno treba da budu obrisani, može se dodati opcija `-f`. Međutim, sa ovom opcijom treba biti obazriv jer se lako može desiti da se obriše nešto što nije trebalo obrisati. Na primer, ako se greškom umesto `rm -f aaa*` otkuca `rm -f aaa *` (razmak između “`aaa`” i “`*`”), biće pobrisani **svi** fajlovi u direktorijumu.

Nekad je potrebno premestiti fajl sa jednog mesta na drugo (tj. iskopirati ga na drugo mesto, a zatim obrisati original). Ovo se postiže komandom `mv` (*move*), čije korišćenje je slično komandi `cp`. Komanda `mv` se može koristiti i za preimenovanje fajlova.

Ukoliko je potrebno, novi prazan fajl se može kreirati komandom `touch`. Osnovna namena ove komande je da, ako fajl postoji, postavi njegovo vreme poslednje izmene na tekuće vreme, a ukoliko fajl ne postoji, tada se kreira novi prazan fajl.

Novi direktorijum se može kreirati komandom `mkdir` (*make directory*), dok se postojeći direktorijum može obrisati komandom `rmdir` (*remove directory*). Obe komande kao parametar

prihvataju naziv direktorijuma. Treba obratiti pažnju na to da `rmdir` može obrisati samo prazan direktorijum.

Svaki fajl pod *Linux*-om ima pridružen skup atributa. Atributi predstavljaju prava koja korisnik može imati nad fajlom. Korisnik može imati prava čitanja (`r`, *read*), pisanja (`w`, *write*) i izvršavanja (`x`, *execute*). Atributi fajla npr. `fajl.txt` mogu da se vide uz pomoć komande `ls -l fajl.txt`

```
-rw-r--r-- 1 e12345 stud  42713 2009-04-15 15:20 fajl.txt
```

Komanda `chmod` služi za menjanje atributa fajla. Na primer, ako korisniku treba dati pravo izvršavanja nekog fajla, to se može postići sa:

```
chmod u+x naziv_fajla
```

Prikaz sadržaja fajla i pretraživanje

Za *Linux* se podrazumeva da termin standardni ulaz označava uređaj sa koga stižu podaci, a da termin standardni izlaz označava uređaj na kome se podaci prikazuju. Većina komandi, ukoliko se drugačije ne navede, kao standardni ulaz koristi specijalni fajl koji odgovara tastaturi, a kao standardni izlaz koristi specijalni fajl koji odgovara ekranu. Standardni ulaz i izlaz se mogu preusmeriti na druge fajlove.

Sadržaj tekstualnih fajlova se na ekranu može prikazati pomoću nekoliko komandi. Prva je komanda `cat` (*concatenate*) čija je osnovna namena da spoji više fajlova u jedan. Kao izlaz, `cat` koristi standardni izlaz, pa ako se napiše `cat naziv_fajla`, dobiće se ispis sadržaja fajla na ekran. Ukoliko je tekst iole veći, samo će preleteti preko ekrana.

Druga komanda, `less`, omogućava da se tekst lista u oba smera korišćenjem kursorskih tastera (izlazak je pomoću tastera *q*, a pretraga pomoću */*, *?*, *n* i *Shift*+*n*, kao kod `man` komande).

Često je potrebno u tekstualnom fajlu naći red u kome se nalazi neki tekst, naročito kada su programi u pitanju. Komanda koja radi upravo to je `grep`. Opšti oblik komande `grep` je:

```
grep sta_se_trazi gde_se_trazi
```

Na primer, ako treba naći sve linije u fajlovima `ispis1.c` i `ispis2.c` koje sadrže reč “printf”, komanda bi bila:

```
grep 'printf' ispis[1-2].c
```

Ukoliko se doda opcija `-i`, tada će se zanemariti razlika između velikih i malih slova. Ako se naziv fajla ne navede, tada se podrazumeva standardni ulaz.

Alias-i

Još jedna korisna mogućnost *bash*-a je definisanje ***alias***-a, odnosno drugih imena za postojeće komande. Na primer, ako se komanda `ls -l *.xyz` često koristi, može joj se dodeliti drugo ime, odnosno skraćenica kojom će se ta komanda pozivati:

```
alias xyz='ls -l *.xyz'
```

Sada, kada korisnik otkuca `xyz`, izvršiće se upravo `ls -l *.xyz`. Ako se zada samo `alias` bez parametara, na ekranu će se ispisati spisak svih trenutno postojećih zamena. Ako neku od zamena treba obrisati, to se može uraditi pomoću komande `unalias`.

Preusmeravanje standardnog ulaza, standardnog izlaza i pajpovi

Kada je potrebno da standardni ulaz ne bude tastatura, odnosno da standardni izlaz ne bude ekran, nego, na primer, neki fajl ili štampač, tada se koristi preusmeravanje. Simbol koji se koristi za preusmeravane standardnog ulaza je “`<`”, a za preusmeravanje standardnog izlaza je “`>`”, odnosno “`>>`”. Ono što se daleko češće koristi je preusmeravanje izlaza. Na primer, u direktorijumu `~/music/` se nalazi puno fajlova sa muzičkim sadržajem i ako je potrebno napraviti tekstualni spisak fajlova fajlova koji se tamo nalaze, to se može dobiti sa:

```
ls ~/music/ > spisak.txt
```

Ovime se u tekućem direktorijumu formira tekstualni fajl `spisak.txt` u kome se nalazi izlaz komande `ls ~/music/`. Sada bi bilo lepo u nastali spisak dodati datum i vreme nastanka. Datum i vreme možemo dobiti uz pomoć komande date. Komanda

```
date > spisak.txt
```

bi prebrisala prethodni sadržaj i upisala novi, što nije ono što nam treba. Da bi prethodni sadržaj ostao, a da se novi doda iza njega, koristi se simbol "`>>`":

```
date >> spisak.txt
```

Još jedna mogućnost koja postoji je da se standardni izlaz jedne komande direktno preusmeri na standardni ulaz druge komande. Ovo se naziva **pajp** (*pipe*) i označava se sa "|". Koristeći pajp, komanda `grep` se može koristiti i za filtriranje izlaza drugih komandi. Na primer, komanda `set` otkucana bez parametara prikazuje spisak svih promenljivih okruženja i njihovih vrednosti. Ukoliko treba izdvojiti podatak o nekoj posebnoj promenljivoj, recimo `PATH`, može se koristiti

```
set|grep PATH
```

Jedan komplikovaniji primer preusmeravanja bi mogao glasiti ovako: treba napraviti spisak svih tekstualnih fajlova iz nekog direktorijuma i svih njegovih poddirektorijuma, ali treba izostaviti one koji u svom nazivu imaju reč "proba". Traženi spisak se može dobiti, na primer, sa:

```
ls –R ~/tekstovi/|grep –i '.txt'|grep –v –i 'proba' >spisak.txt
```

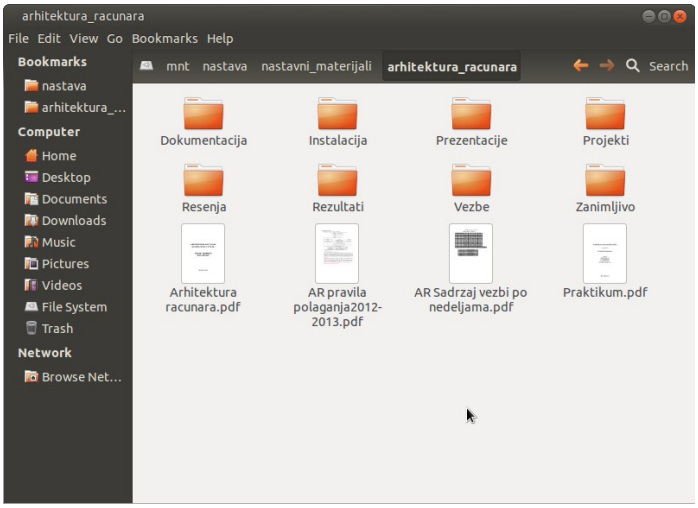
Opcija `-R` komande `ls` znači da se lista sadržaj direktorijuma i svih njegovih poddirektorijuma. Opcija `–v` komande `grep` znači da se izdvajaju sve linije koje **ne** sadrže zadatu reč. Gornje komande rade sledeće: prvo komanda `ls` izlista sve fajlove u zadatom direktorijumu i njegovim poddirektorijumima; izlaz komande `ls` se šalje prvoj `grep` komandi koja izdvaja samo one linije u kojima se nalazi tekst ".txt"; izlaz prve `grep` komande se šalje drugoj `grep` komandi koja izbacuje sve linije u kojima se pominje reč "proba", a zatim se tako prerađeni spisak upisuje u fajl `spisak.txt`.

Izvršavanje programa u pozadini

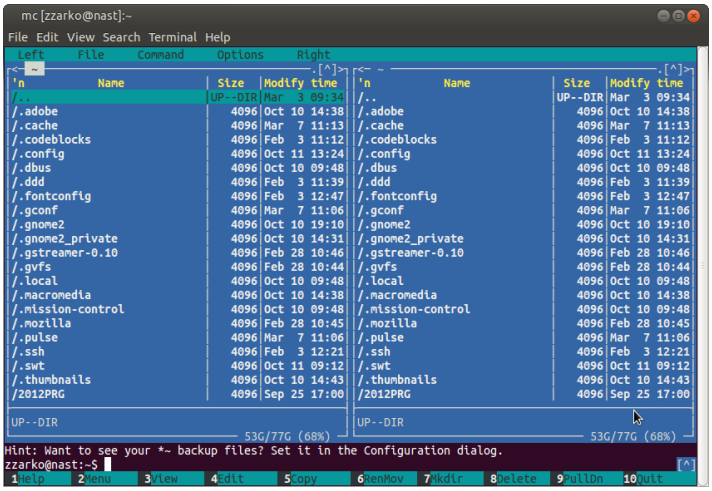
Ukoliko se na kraju komande stavi znak "`&`", komanda će se izvršavati u pozadini. Ovime *shell* i dalje ostaje dostupan za interaktivni rad.

1.6. Programi za rukovanje fajlovima

Pored rukovanja fajlovima iz komandne linije, moguće je baratanje fajlovima i iz grafičkog okruženja, korišćenjem miša. Postoji dosta programa za tu svrhu, a koji je podrazumevano instaliran zavisi od distribucije do distribucije. Na primer, uz grafičko okruženje *GNOME* stiže program *Nautilus*, dok uz *KDE* stiže *Dolphin*.



Slika 1: *Nautilus*



Slika 2: *Midnight Commander*

Druga vrsta programa je bazirana na postojanju dva zasebna dela prozora pri čemu svaki od njih nezavisno prikazuje sadržaj nekog od direktorijuma. Fajlovi se mogu kopirati, pomerati i sl. između ova dva dela prozora odnosno direktorijuma. U ovu grupu spada *Midnight Commander* koji se standardno isporučuje u skoro svim distribucijama i koji se izvršava u tekstualnom režimu. Postoji više programa tipa *Midnight Commander*-a koji se izvršavaju u grafičkom režimu, a neki od njih su *Krusader, Worker, Double Commander,* itd.

1.7. Tekst editori

Najčešće korišćeni tekst editori u *Linux* okruženju su *vim/gwim* i *emacs*. Pored njih postoji i mnoštvo drugih tekst editora, kao što su *nano, kwrite, nedit, gedit, xedit*...

Preporučeni editori za kurs iz Arhitekture računara su *kwrite* za *KDE* odnosno *gedit* za *Gnome* okruženje. Oba se izvršavaju u grafičkom okruženju i podržavaju standardne (*Windows*) prečice sa tastature za rad sa tekstom, tako da korisnici naviknuti na ovaj način rada mogu lako da se snađu.

1.8. Grafički programi

Što se porograma za crtanje tiče, za jednostavnije bitmapirane sličice se često koristi npr. *gpaint* ili *MyPaint*, dok se za ozbiljniju obradu slike preporučuje recimo *gimp*. Za obradu vektorskih crteža mogu da se koriste *inkscape* ili *karbon*. Program pod nazivom *dia* je namenjen crtanju dijagrama, dok je *BRL-CAD* namenjen za podršku računarski pomognutom dizajnu (*Computer Aided Design*).

1.9. Korisni linkovi

Više o *Linux*-u se može naći na stranicama *The Linux Documentation Project* [4], dok se detaljnije uputstvo za *bash* može naći na njegovoj zvaničnoj stranici [5]. Takođe, u okviru ovog praktikuma, u dodatku F, se može naći uvod u pisanje *shell* skriptova.

1. www.linux.org, www.kernel.org
2. www.gnu.org/licenses/licenses.html
3. www.ubuntu.com, www.debian.org, www.opensuse.org, www.knoppix.org, www.redhat.com, distrowatch.com
4. www.tldp.org

5. www.gnu.org/software/bash/bash.html
6. www.linuxlinks.com

2. Asemblerski programi

2.1. Zašto asemblersko programiranje

Poslednjih nekoliko godina se praktično došlo do gornje granice radne učestanosti na kojima rade savremeni mikroprocesori zasnovani na silicijumu (3-4GHz; veće brzine se mogu ostvariti samo korišćenjem ekstremno niskih temperatura, kao što je hlađenje tečnim azotom, na primer). To praktično znači da se više ne može reći “ako hoćeš da ti program brže radi, kupi brži procesor”. Delimično rešenje problema leži u korišćenju više procesorskih jezgara, međutim to zahteva pisanje programa koji mogu da iskoriste taj potencijal. Mnogi algoritmi se mogu samo delimično (ili nikako) paralelizovati, što dovodi do toga da se pisanju efikasnog kôda mora posvetiti veća pažnja. Današnji kompajleri su prilično napredovali po pitanju optimizacije, ali i njihove mogućnosti imaju granica i nekada je za brži kôd potrebno spustiti se na niži nivo, odnosno na asembler.

Optimalno iskorišćavanje mogućnosti kompajlera (i njegovih optimizacija) praktično podrazumeva poznavanje onoga što se dešava “ispod haube”, tj. kako će izgledati asemblerski, odnosno mašinski kôd koga će procesor izvršavati. Ovo je još bitnije kod programskih jezika koji se interpretiraju u virtuelnoj mašini (*Java* virtuelna mašina, *.NET* virtuelna mašina, itd.), pošto virtuelna mašina praktično izvršava neku vrstu mašinskog kôda za virtuelni procesor realizovan u njoj.

Ono što bi ovaj kurs trebalo da obezbedi je *osećaj* šta sve procesor mora da izvrši kada se izvršava iskaz napisan u višem programskom jeziku i da pruži osnovu za pisanje asemblerskih potprograma kojima se mogu ubrzati kritični delovi sporih programa.

2.2. Neophodno predznanje za praćenje vežbi

Da bi se vežbe iz Arhitekture računara mogle uspešno pratiti, neophodno je poznavanje programskog jezika *C*, pošto se prikazi mnogih algoritama u praktikumu, kao i objašnjenja pojedinih delova asemblerskog koda, zasnivaju na *C* programima. U neophodna znanja vezana za programski jezik *C* spadaju:

- definisanje i deklarisanje funkcija
- definisanje i korišćenje konstanti
- prenos parametara funkciji po vrednosti i po adresi
- osnovne operacije i rad sa nizovima (pristup elementima niza, pretraživanje niza, sortiranje, rad sa stringovima)
- osnovne operacije i rad sa pokazivačima (pokazivačka aritmetika, pristup nizovima pomoću pokazivača)
- poznavanje osnovnih C iskaza (`if`, `for`, `do`, `while`, `return`, `break`)

2.3. Programski model procesora *Intel 80386*

Intel 80386 (u daljem tekstu *80386*) je 32-bitni mikroprocesor *CISC* arhitekture i pripada familiji procesora x86. Ovaj procesor spada u *little endian* procesore. Unazad je kompatibilan sa procesorima iz familije x86, pa u sebi sadrži 8-bitne, 16-bitne i 32-bitne registre (slika 3).

Registri opšte namene (`eax`, `ebx`, ...) su specifični po tome što se kod njih može pristupiti njihovom manje značajnom delu kao posebnom 16-bitnom registru. Tako, na primer, manje značajnih 16 bita registra `eax` predstavlja 16-bitni registar `ax`. Dalje, kod registara `ax`, `bx`, `cx` i `dx` se može posebno posmatrati njihov manje značajan i više značajan bajt. Za registar `ax`, više

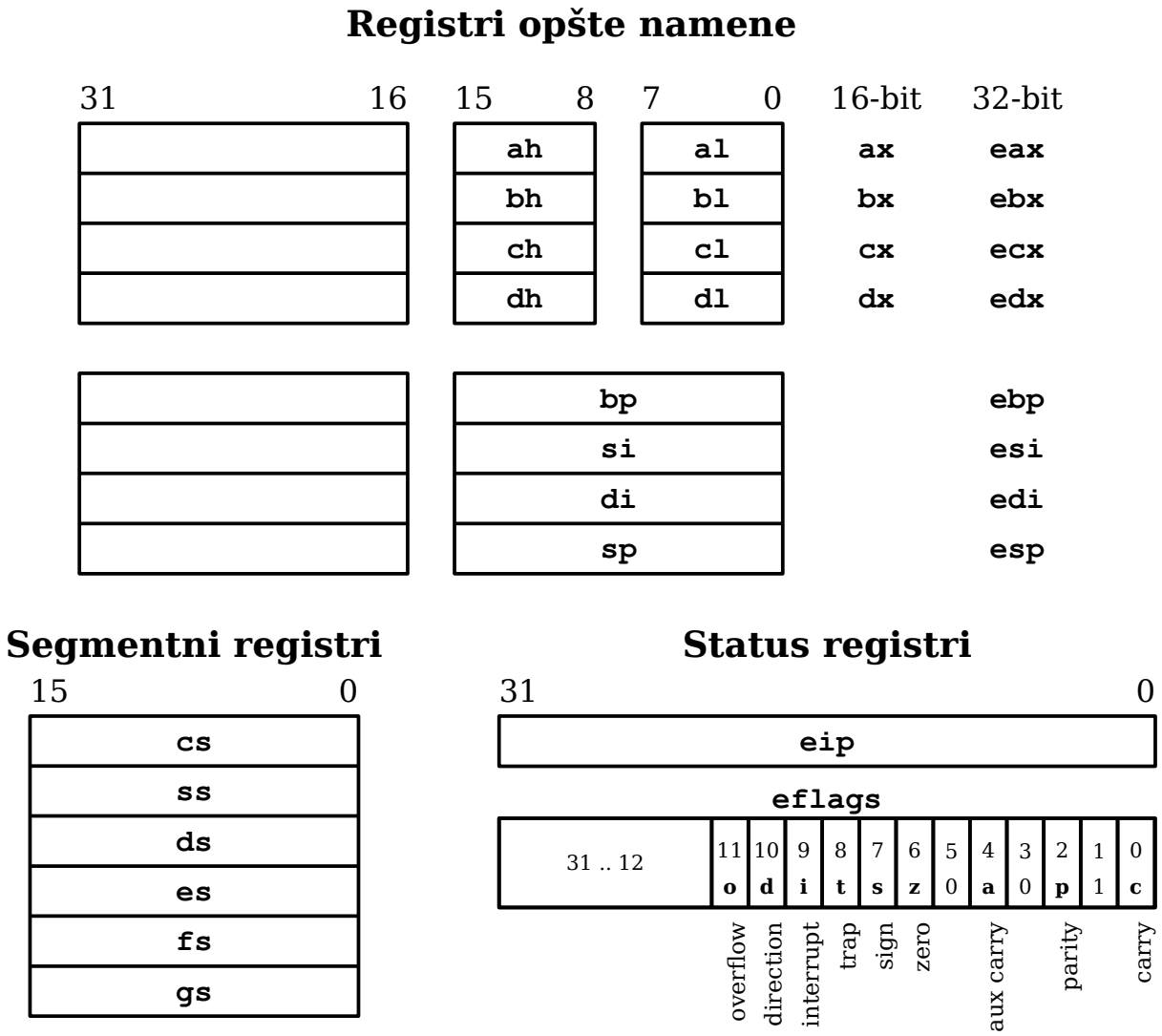
značajan bajt predstavlja registar `ah`, a manje značajan bajt predstavlja registar `al` (analogno za ostale registre).

Registri opšte namene `esi` (*source index*) i `edi` (*destination index*) imaju specijalnu namenu kod naredbi za rad sa nizovima, kao i za neka adresiranja.

Registri opšte namene `esp` (*stack pointer*) i `ebp` (*base pointer*) imaju specijalnu namenu kod naredbi koje koriste stek. Pošto su ova dva registra neophodna za rad sa potprogramima, u ovome kursu se **neće koristiti** ni za koju drugu namenu.

Segmentni registri u sebi sadrže bazne adrese segmenata memorije: `cs` (*code segment*) sadrži adresu početka segmenta naredbi, `ds` (*data segment*) sadrži adresu početka osnovnog segmenta podataka, `ss` (*stack segment*) sadrži adresu početka segmenta steka, dok su `es`, `fs` i `gs` dodatni segmentni registri koji se koriste za rukovanje podacima.

Statusni registri su dostupni samo pomoću specijalnih naredbi. Registar `eip` (*instruction pointer*) u sebi sadrži adresu naredbe (u okviru segmenta naredbi) koja će se sledeća izvršiti. Registar `eflags` u sebi sadrži indikatore (logičke promenljive, flegove) čije vrednosti zavise od rezultata izvršavanja naredbi. Od značaja će biti sledeći indikatori: `c` (*carry*), `o` (*overflow*), `z` (*zero*), `s` (*sign*), i `d` (*direction*; koristi se samo kod nekih specijalnih naredbi).



Slika 3: Osnovni registri procesora *Intel 80386*

Procesor *80386* je napravljen tako da mnoge njegove naredbe ne zavise od toga da li su im operandi označeni ili neoznačeni brojevi (što je posledica korišćenja komplementa 2 za označene brojeve). Da bi program koji se izvršava na procesoru mogao da koristi oba tipa podataka, a da pri tom ne dođe do zabune, mora se osloniti na indikatore. Posmatranjem vrednosti nekog od indikatora nakon obavljene operacije, program praktično daje značenje podatku koji se obrađuje (npr. da li je u pitanju označen ili neoznačen broj).

`c` (*carry*) je indikator prekoračenja za neoznačene brojeve. Postavlja se na `1` ukoliko je prilikom izvršavanja neke operacije došlo do prenosa ili pozajmce (rezultat se ovde posmatra kao neoznačen broj), a u suprotnom na `0`. Kada se obavlja neka aritmetička operacija, na primer sabiranje, procesor će odrediti rezultat u binarnom obliku i istovremeno postaviti sve indikatore koji mogu imati veze sa sabiranjem. Ako program radi sa neoznačenim brojevima, nakon sabiranja treba proveriti inidikator `c` kako bi se utvrdilo da li je rezultat u opsegu neoznačenih brojeva.

`o` (*overflow*) je indikator prekoračenja za označene brojeve. Postavlja se na `1` ukoliko je prilikom izvršavanja neke operacije došlo do prenosa ili pozajmce (rezultat se ovde posmatra kao označen broj), a u suprotnom na `0`. Ako program radi sa označenim brojevima, nakon aritmetičke operacije treba proveriti inidikator `o` kako bi se utvrdilo da li je rezultat u opsegu označenih brojeva.

`z` (*zero*) je indikator nule. Postavlja se na `1` ukoliko je prilikom izvršavanja neke operacije rezultat jednak nuli, a u suprotnom na `0`.

`s` (*sign*) je indikator znaka. Postavlja se na `1` ukoliko je prilikom izvršavanja neke operacije najviši bit rezultata jednak jedinici (odnosno, ako je rezultat, posmatran kao označeni broj, negativan), a u suprotnom na `0`.

Prilikom izvršavanja, program stanje ovih indikatora može testirati naredbama uslovnih skokova, a postoje i naredbe koje koriste vrednosti indikatora prilikom izračunavanja rezultata. Takođe, vrednosti nekih indikatora se mogu i direktno postavljati na `0` ili `1` (za detalje pogledati poglavlje 11, Pregled korišćenih naredbi).

Skup naredbi *80386* je **neortogonalan**: ne prihvataju sve naredbe sve vrste operanada. Ovo je velikim delom posledica činjenice da je *80386* (kao i svi ostali naslednici, uključujući i poslednje familije mikropeocesora) projektovan s ciljem da bude kompatibilan sa ranijim *Intel*-ovim mikroprocesorima iz x86 familije.

2.4. *AT&T* sintaksa

Za familiju procesora x86 postoje dve sintakse pisanja naredbi. Jedna je *Intel*-ova, a druga je *AT&T*. Osnovne karakteristike *AT&T* sintakse su sledeće:

- **Labela** se definiše navođenjem imena labele iza koga sledi dvotačka.
- **Neposredni operandi** imaju prefiks "`$`", nakon čega sledi broj ili naziv labele. Ukoliko se ispred konstante ili labele ne navede "`$`", tada se koristi direktno adresiranje.
- Podrazumevani brojni sistem je dekadni. Ako treba uneti broj u heksadecimalnom obliku, prefiks je "`0x`", za oktalne brojeve prefiks je "`0`", dok je za brojeve u binarnom obliku prefiks "`0b`".
- **Registarski operandi** imaju prefiks "`%`".
- Ukoliko naredba ima dva operanda, **prvo ide izvorni operand, a nakon njega odredišni**.

Sve naredbe koje barataju podacima imaju sufiks "`b`", "`w`" ili "`l`" (malo L), čime se definiše da li se naredba odnosi na bajt (*byte*, 8 bita), reč (*word*, 16 bita) ili dvostruku reč (*long*, 32 bita). Ukoliko se sufiks ne navede, biće određen na osnovu odredišnog operanda (ovo nije problem kod registarskog adresiranja, ali jeste kod memorijskog). Na primer, kod naredbe:

```
movw $10, %ax
```

slovo `w` u nazivu označava da je u pitanju naredba koja se odnosi na reč. Izvorni operand je neposredni (vrednost 10 dekadno), a odredišni je registarski (registar `ax`). Slično, u primeru:

```
movb %ch, %al
```

slovo `b` u nazivu označava da je u pitanju naredba koja se odnosi na bajt. Izvorni operand je registarski (registar `ch`), a odredišni operand je takođe registarski.

2.5. Neposredno i registarsko adresiranje

Procesor *80386* podržava više adresiranja (kompletan spisak se može naći u dodatku A). Na primer, kod naredbe koja vrši postavljanje registra **ax** na vrednost 10:

```
movw $10, %ax
```

prvi operand (`$10`) je naveden korišćenjem neposrednog adresiranja, što znači da on upravo predstavlja vrednost sa kojom se obavlja operacija. Drugi operand (`%ax`) je primer registarskog adresiranja, što znači da ovaj operand predstavlja registar nad kojim se obavlja operacija.

2.6. Izgled asemblerskog programa

Na početku asemblerskih programa, sa kakvima će se raditi u toku ovog kursa, treba da stoji sledeće:

```
#program radi to i to ...
#autor: Pera Peric, E1234
.section .data
.section .text
.globl main
main:
```

U prvih par linija u okviru komentara se nalazi kratak opis programa i par osnovnih podataka o fajlu (autor, datum i slično). Treba obratiti pažnju na to da se **ne prave komentari u stilu** `#1234` (broj koji se nalazi odmah nakon oznake "`#`"), pošto takve oznake asembler koristi interno prilikom asembliranja programa. Sledeća linija (`.section .data`) predstavlja **direktivu** (ili **pseudonaredbu**) asembleru i označava početak segmenta podataka. Nakon nje sledi direktiva koja označava početak segmenta naredbi (`.section .text`). U okviru segmenta naredbi se definiše ulazna naredba programa kao spoljašnja referenca (`.globl`) koja mora biti obeležena labelom `main`. Na kraju programa treba da se nalazi:

```
kraj:    movl $1, %eax
         movl $0, %ebx
         int $0x80
```

Ove dve linije omogućavaju da se program završi na regularan način, pozivom sistemske funkcije za završetak programa (o sistemskim pozivima će više reči u glavi 5). Da bi se videla stanja registara pre nego se program završi, korisno je postaviti prekidnu tačku na prvu od dve naredbe.

Napomena: većina programa u praktikumu je data bez ovih podrazumevanih linija.

2.7. Asembler (kompajler)

Asemblerski program se prvo piše tekst editorom i snima u fajl sa ekstenzijom "`S`" (veliko S). Za prevođenje asemblerskog programa u izvršni program će se koristiti `gcc` (*GNU Compiler Collection*):

```
gcc -m32 -g -o primer primer.S
```

Opcija `-g` omogućava dibagiranje programa, dok se opcijom `-o` zadaje naziv izvršnog programa. Ukoliko se opcija `-o` izostavi, izvršni program će imati naziv `a.out`. Prva navedena opcija, `-m32`, je neophodna za prevođenje 32-bitnih programa na 64-bitnim sistemima, pošto su svi programi predstavljeni u ovom Praktikumu 32-bitni.

Prevođenje asemblerskog programa u izvršni oblik praktično obuhvata tri koraka: makro pretprocesiranje, prevođenje asemblerskog programa u objektni fajl i linkovanje objektnog fajla sa programskim kodom koji omogućava pokretanje programa na operativnom sistemu. Svaki od koraka se može odraditi i zasebno (dodatak B), ukoliko je potrebno, međutim, to se neće koristiti u okviru ovog kursa.

2.8. Dibager

Dibager služi za kontrolisano izvršavanje programa, i unutar njega se mogu stalno pratiti i menjati sadržaji registara i indikatora i postavljati prekidne tačke. Dibager koji će se koristiti se zove ***DDD*** (*Data Display Debugger*) i predstavlja grafičko okruženje za *gdb* (*GNU Debugger*, osnovni dibager koji radi iz komandne linije). *DDD* se poziva komandom:

```
ddd primer
```

gde je `primer` naziv izvršnog fajla. Ukoliko se dibager pokrene iz komandne linije na ovaj način, dalja interakcija sa komandnom linijom neće biti moguća sve dok se dibager ne zatvori. Međutim, ukoliko se dibager pokrene na sledeći način:

```
ddd primer &
```

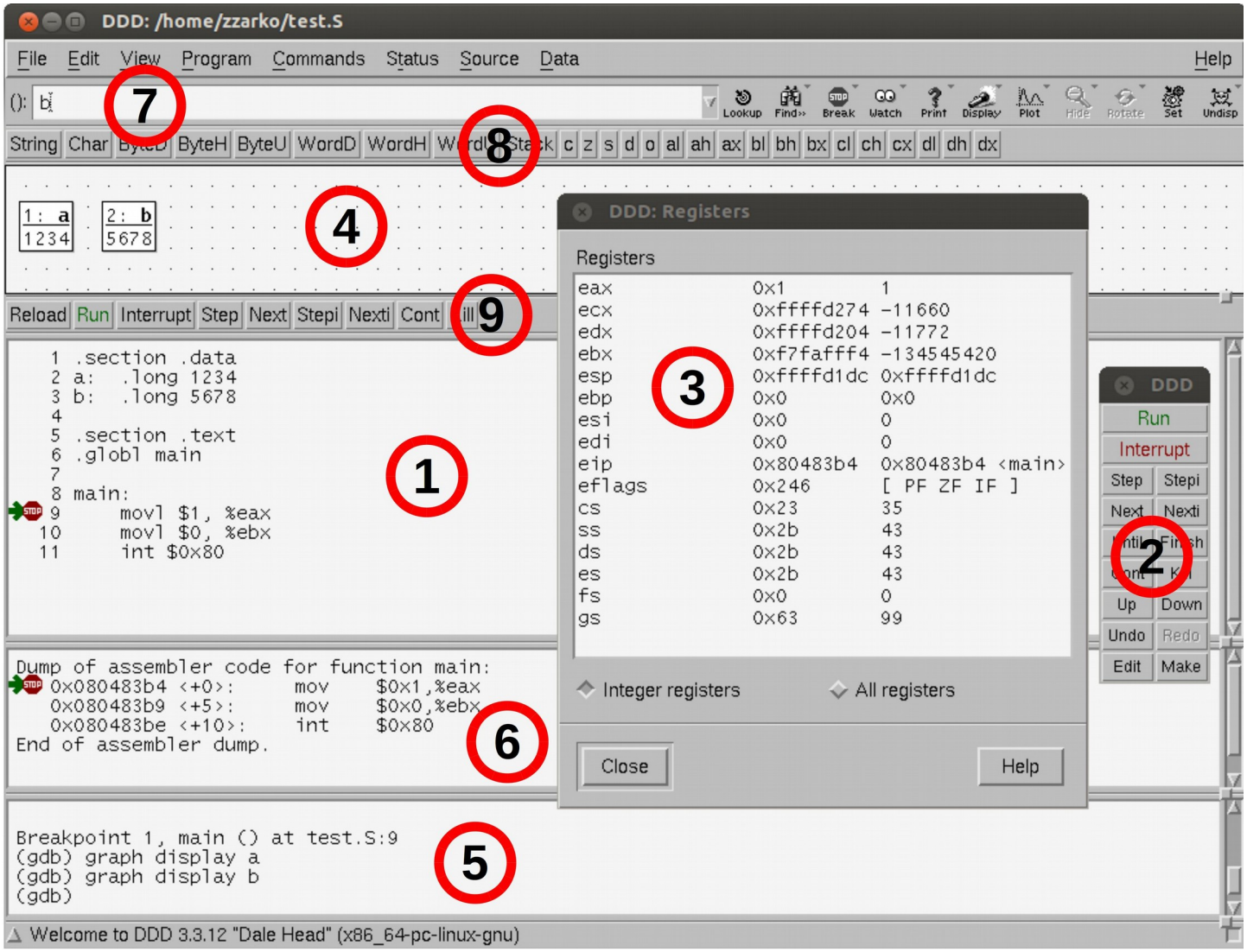
tada će komandna linija ostati slobodna za dalji rad i dok je dibager pokrenut. Na taj način se ona može dalje koristiti za prevođenje programa, bez potrebe da se dibager zatvara ili da se otvara nova komandna linija.

Ukoliko se desi da dibager prestane da se odaziva na komande, a ne može se zatvoriti klikom na dugme za zatvaranje, može se iskoristiti naredba `killall`:

```
killall -9 ddd
```

Nakon izvršenja ove naredbe, sve instance dibagera će biti zatvorene.

Na slici 4 je prikazan izgled glavnog prozora dibagera. Na ovom kursu se koristi **izmenjena verzija** *DDD* dibagera 3.3.11. Izmene su načinjene radi lakšeg rada sa asemblerskim programima. Glavni prozor je podeljen na nekoliko panela i prozora (koji se mogu prikazati, odnosno sakriti iz *View* i *Status* menija). Panel (1) sadrži izvorni kôd programa. Prozor (2) sadrži komande vezane za izvršavanje programa. Prozor (3) sadrži trenutne vrednosti registara. Panel (4) sadrži vrednosti koje je korisnik odabrao da prikazuje (kada takvih vrednosti nema, panel se ne prikazuje). Panel (5) sadrži komandnu liniju ***gdb***-a (tu se može videti tekstualni oblik svih komandi zadatih u *DDD*-u), dok panel (6) sadrži izgled programskog kôda u memoriji. Ukoliko nakom pokretanja dibagera prozor sa registrima nije uključen, treba ga uključiti odabirom opcije *Registers* menija *Status*.



Slika 4: DDD dibager

Pre pokretanja programa, prvo što treba uraditi je postaviti **tačku prekida** (*breakpoint*) na pretposlednju liniju u programu (trebalo bi da je `kraj: movl $1, %eax`). Tačka prekida predstavlja mesto na kome će se zaustaviti izvršavanje programa nakon što se on pokrene. Mesto do kog se stiglo u izvršavanju programa je obeleženo zelenom strelicom i predstavlja naredbu koja će se **sledeća** izvršiti. Tačka prekida se može postaviti na više načina:

- kursor se postavi na početak linije, pa se mišem klikne na dugme *Break* (ukoliko u toj liniji već postoji tačka prekida, dugme menja naziv u *Clear*, što znači uklanjanje tačke prekida)
- klikne se desnim dugmetom miša na početak linije, pa se iz menija odabere *Set Breakpoint*
- dvoklik mišem na početak linije

Tačka prekida se može isključiti ili preko dugmeta *Clear*, ili izborom opcije *Delete Breakpoint* iz menija koji se dobija desnim klikom miša na tačku prekida.

Prozor (2) sadrži više komandi koje se koriste prilikom izvršavanja programa, a od koristi će biti sledeće:

- *Run* pokreće program od početka. Izvršavanje se zaustavlja kad se naiđe na tačku prekida.
- *Stepi* i *Nexti* izvršavaju tekuću naredbu i nakon toga ponovo zaustavljaju izvršavanje programa. Koriste se za izvršavanje programa korak po korak. Razlika između ove dve komande se ogleda kod poziva potprograma: *Stepi* izvršava potprogram korak po korak, dok *Nexti* izvrši ceo potprogram i zaustavlja se na naredbi koja sledi naredbu poziva potprograma.

- *Cont* nastavlja izvršavanje programa do sledeće tačke prekida.
- *Interrupt* prekida izvršavanje pokrenutog programa (sa mogućnošću nastavka izvršavanja).
- *Kill* prekida izvršavanje pokrenutog programa (bez mogućnosti nastavka izvršavanja).

Ukoliko se klikne na neki od registara u *Registers* prozoru, njegovo ime se pojavljuje u liniji (7) koja se nalazi ispod menija. Na primer, ako se klikne na registar `eax`, u liniji (7) se pojavljuje tekst "*$eax*". Ako se sada klikne na dugme *Display*, u panelu (4) će se prikazati polje sa sadržajem registra `eax` u označenom dekadnom obliku. Ukoliko treba prikazati heksadecimalnu vrednost, pre klika na dugme *Display* treba dodati prefiks "*/x*", za prikaz binarnih vrednosti treba staviti "*/t*", dok za prikaz neoznačenih dekadnih vrednosti treba staviti "*/u*". Nazivi registara se u liniju (7) mogu unositi i ručno, samo ispred naziva registra treba staviti prefiks "*$*".

Prikazane vrednosti na panelu (4) se mogu obrisati klikom na vrednost, a zatim klikom na dugme *Undisp*.

Da bi se promenila vrednost u registru, u liniju (7) treba postaviti njegovo ime (zajedno sa prefiksom "*$*"; to se može uraditi, na primer, klikom na registar u prozoru *Registers*) i kliknuti na dugme *Set*. Nakon toga, u prozoru koji se otvori se može upisati vrednost koja će se upisati u registar. Ukoliko se vrednost navede bez prefiksa, smatraće se da je u dekadnom obliku, dok se prefiks "*0x*" koristi za heksadecimalne vrednosti.

Prikaz pojedinih indikatora iz `eflags` registra, kao i vrednosti 8-bitnih i 16-bitnih registara se može postići klikom na odgovarajuće dugme na panelu za prikaz podataka (8).

Napomena: vrednosti indikatora i registara prikazanih preko ovog panela se **ne mogu** menjati.

Panel (9) sadrži najčešće korišćene opcije prozora (2), kao i dugme za ponovno učitavanje programa (*Reload).*

Sva podešavanja *DDD*-a se čuvaju u skrivenom direktorijumu `.ddd` koji se nalazi u `home` direktorijumu korisnika, ako se drugačije ne naglasi. Ukoliko su podešavanja promenjena na takav način da se *DDD* ne ponaša kako se očekuje, potrebno je obrisati trenutna podešavanja, nakon čega će se automatski izgenerisati podrazumevana podešavanja (pri tome, *DDD* ne sme biti startovan). Podešavanja se mogu obrisati komandom:

```
rm -rf ~/.ddd
```

2.9. Primer: izračunavanje NZD

Na slici 5 prikazan je *C* program za izračunavanje najvećeg zajedničkog delioca za dva prirodna broja.

```
a = 12;
b = 8;
while (a != b)
     if a > b
          a -= b;
     else
          b -= a;
```

Slika 5: C program za izračunavanje NZD

Predstavljen kao niz koraka, algoritam bi glasio:

1. Smesti vrednost `12` u promenljivu `a`
2. Smesti vrednost `8` u promenljivu `b`
3. Uporedi `a` i `b`. Ako su jednaki, pređi na korak 9
4. Uporedi `a` i `b`. Ako je `a` veće od `b`, pređi na korak 7, inače pređi na sledeći korak

5. Oduzmi `a` od `b` i rezultat smesti u `b`
6. Pređi na korak 3
7. Oduzmi `b` od `a` i rezultat smesti u `a`
8. Pređi na korak 3
9. Kraj algoritma

Varijanta 1 *- Doslovan prevod algoritma u asembler*

Koriste se sledeće naredbe:

- `mov` - Smeštanje izvornog operanda u odredišni. U ovom programu se koriste registarski i neposredni operandi. Na primer, `movl $12, %eax` će smestiti broj `12` (neposredni operand, kodiran kao deo naredbe) u registar `eax`. Naredba `mov` ne utiče na indikatore.

- `cmp` - Poređenje operanada. Tehnički, ovom naredbom se od odredišnog operanda oduzima izvorni, ali se pri tom sadržaj odredišta ne menja, već se samo ažuriraju indikatori: u ovom slučaju zanimaju nas indikator `z` (*zero*), koji se postavlja ako je rezultat nula, i indikator `c` (*carry*), koji se postavlja u slučaju prenosa sa najznačajnije pozicije.
- `je`, `ja`, `jmp` - Relativni skokovi. Prva dva su uslovna. Do skoka `je` (*jump if equal*) dolazi ako važi `z`=1, a do skoka `ja` (*jump if above*) dolazi ako važi `c`=0 i `z`=0. Naredba `jmp` je bezuslovni skok. Naredbe skoka ne utiču na indikatore.
- `sub` - Oduzimanje izvornog operanda od odredišnog. Rezultat se smešta u odredište. Prilikom oduzimanja se ne uzima u obzir vrednost indikatora `c`. Naredba `sub` menja indikatore.

```
     asembler          objašnjenje           C program
-------------------------------------------------------------
     movl $12,%eax     # 12->eax             a = 12;
     movl $8,%ebx      # 8->ebx              b = 8;
uporedi:               #
     cmpl %ebx,%eax    #                     while (a != b)
     je kraj           # skok ako eax=ebx
     cmpl %ebx,%eax    #                           if (a > b)
     ja vece           # skok ako eax>ebx
     subl %eax,%ebx    # ebx-eax->ebx                    a = a - b;
     jmp uporedi       #
vece:                  #                           else
     subl %ebx,%eax    # eax-ebx->eax                    b = b - a;
     jmp uporedi       #
kraj:
     movl $1, %eax
     movl $0, %ebx
     int $0x80
```

Varijanta 2 *– Modifikacija varijante 1*

U ovoj varijanti se uzima u obzir činjenica da naredbe skoka ne utiču na indikatore, pa je drugo poređenje nepotrebno.

```
        movl $12,%eax
        movl $8,%ebx
uporedi:
        cmpl %ebx,%eax
        je kraj
        ja vece
        subl %eax,%ebx
        jmp uporedi
vece:
        subl %ebx,%eax
        jmp uporedi
kraj:
        movl $1, %eax
        movl $0, %ebx
        int $0x80
```

Varijanta 3 *- Optimizovana verzija*

Za algoritam nije bitno da li se veća vrednost nalazi u `eax` ili `ebx` (jer se to proverava u svakoj iteraciji). Zbog toga se u slučaju da je `ebx` veće od `eax`, sadržaj ovih registara može međusobno zameniti (kao rezultat zamene, `eax` će biti veće od `ebx`), i zatim uraditi oduzimanje `ebx` od `eax`.

Zamenu sadržaja radi naredba `xchg`. Ovde se koriste registarski operandi. Naredba ne utiče na indikatore.

```
        movl $12,%eax
        movl $8,%ebx
uporedi:
        cmpl %ebx,%eax
        je kraj
        ja vece
        xchgl %eax,%ebx  #eax<->ebx
vece:
        subl %ebx,%eax
        jmp uporedi
kraj:
        movl $1, %eax
        movl $0, %ebx
        int $0x80
```

2.10. Domaći zadaci

1. Napisati asemblerski program koji će sabrati prvih *n* prirodnih brojeva. Vrednost *n* se nalazi u registru `ecx`, dok zbir treba da bude u registru `eax`.
2. Napisati program koji poredi vrednosti u registrima `ebx` i `ecx`, i na osnovu rezultata poređenja u regaistar `eax` stavlja:
   `1` ako `ebx` > `ecx`
   `0` ako `ebx` = `ecx`
   `2` ako `ebx` < `ecx`

3. Rukovanje celim brojevima

3.1. Neoznačeni i označeni brojevi

Procesor *80386* omogućava rukovanje neoznačenim i označenim celim brojevima. Da li će se broj u nekom registru smatrati za označeni ili neoznačeni zavisi isključivo od programa i načina na koji se koriste indikatori. Veličina operanada kojima procesorske naredbe mogu direktno rukovati obično je ograničena kapacitetom mašinskih reči, tj. registara. Za operande u toj veličini se kaže da su u **jednostrukoj preciznosti**. Neoznačeni broj u jednostrukoj preciznosti kod *80386* procesora može imati vrednosti iz opsega od 0 do 2^{32}-1. Označeni brojevi se predstavljaju u komplement 2 predstavi, što znači da je opseg označenih brojeva u jednostrukoj preciznosti kod *80386* od -2^{31} do 2^{31}-1.

Ako program rukuje sa neoznačenim brojevima, tada indikator `c` (*carry*), ukazuje na prenos (izlazak van opsega) kod operacija sa neoznačenim brojevima. Kada program rukuje označenim brojevima, tada indikator `o` (*overflow*) ukazuje na izlazak van opsega.

3.2. Definisanje numeričkih podataka

Podaci se smeštaju u segment podataka. Za smeštanje različitih vrsta podataka se koriste različite direktive. Osnovne direktive za cele brojeve su prikazane u sledećem primeru:

```
.section .data                  #    sadržaj u memoriji u bajtovima
bajt:
     .byte 0xff, 100            #    0xff 0x64
rec:
     .word 0xee56, 2, 50        #    0x56 0xee 0x02 0x00 0x32 0x00
duga_rec:
     .long 0xabcd1234           #    0x34 0x12 0xcd 0xab
```

Iza odgovarajuće direktive se može navesti i više vrednosti odvojenih zarezima, čime se definiše niz vrednosti istog tipa. Ispred definicije podatka stoji labela, čime se omogućava da se podatku pristupa preko labele. Labela je samo **zamena za adresu** na kojoj se podatak nalazi. Treba imati na umu da procesor može sa bilo koje adrese da pročita podatak bilo koje podržane veličine, bez obzira na to kako je podatak definisan. Primeri:

```
movl duga_rec, %eax   #0xabcd1234 -> eax
movw %ax, rec         #sadržaj eax -> rec
movl $bajt, %eax      #adresa labele bajt -> eax
movw bajt, %ax        #0x64ff -> ax
movl rec+2, %eax      #0x320002 -> eax
movb duga_rec, %al    #0x34 -> al
movl bajt, %eax       #0xee5664ff -> eax
```

Konstante se definišu na sledeći način:

```
konstanta = vrednost
```

`gcc` dozvoljava osnovne aritmetičke operacije sa konstantama (moguće je pisati proizvoljne aritmetičke izraze sa konstantama bilo gde u programu gde se konstanta može naći):

```
const1 = 5
const2 = const1+4             #const2 ima vrednost 9
const3 = const2*2             #const3 ima vrednost 18
movl $const1+const2, %eax     #14 -> eax
movl $const3*4, %ebx          #72 -> ebx
movl $const1*%eax, %ebx       #GREŠKA - ne postoji u x86 arhitekturi
```

Treba imati na umu da ovakve operacije važe **samo za konstante**, i da će svaka drugačija upotreba dovesti do greške.

Labela se može definisati kao konstanta i tada joj se eksplicitno dodeljuje adresa. Ovakva labela se koristi ili kao neposredni operand ili kao početna vrednost kod promenljive:

```
max_vrednost = 20
vrednost:    .long max_vrednost
...
movl $max_vrednost, %eax
```

Mašinski format naredbi procesora *80386* dozvoljava da se u jednoj naredbi može obaviti najviše jedan pristup memoriji (kod naredbi sa dva operanda). To praktično znači da je neispravna naredba čija oba operanda zahtevaju pristupanje nekoj memorijskoj lokaciji, kao u primeru:

```
movb bajt, rec    #greška – u oba operanda postoji pristupanje memoriji
```

Treba obratiti pažnju na korišćenje konstanti i labela u programima. Ispravne i neispravne situacije i njihovo značenje su date u sledećem primeru:

```
a:   .long 10
b = 15
...
movl $a, %eax     # adresa promenljive a u %eax
movl a, %eax      # vrednost promenljive a u %eax
movl $b, %eax     # vrednost konstante b u %eax
movl b, %eax      # greška – pokušaj pristupa lokaciji sa adresom 15
```

Deklaracija promenljive uvek treba da sadrži i inicijalnu vrednost, jer u suprotnom ta promenljiva dobija adresu naredne promenljive. Ako su, na primer, promenljive definisane na sledeći način::

```
a:   .long
b:   .long 5
```

tada će promenljiva `a` imati istu adresu kao promenljiva `b`, što znači da će se njenim čitanjem dobiti vrednost 5, dok će upis u tu promenljivu menjati i verdnost promenljive `b`.

3.3. Prikaz promenljivih u dibageru

32-bitne promenljive se mogu prikazati tako što se u gornju liniju postavi njihovo ime (klikom na promenljivu, ili ručnim upisom imena), a zatim se klikne na dugme *Display*.

8-bitne promenljive se mogu prikazati postavljanjem njihovog imena u gornju liniju i klikom na odgovarajuće dugme na panelu za prikaz promenljivih: *ByteH* će vrednost prikazati kao heksadecimalnu, *ByteD* kao označenu dekadnu, a *ByteU* kao neoznačenu dekadnu. Analogno, za 16-bitne promenljive se koriste *WordH*, *WordD* i *WordU.*

Napomena: ovako prikazane 8-bitne i 16-bitne vrednosti se **ne mogu** menjati.

Promena 32-bitnih promenljivih se ostvaruje postavljanjem njihovog imena u gornju liniju i klikom na dugme *Set*. 8-bitne i 16-bitne promenljive se ne mogu menjati na ovaj način.

Ukoliko je potrebno prikazati sadržaj memorije koji je određen nekim tipom adresiranja, kao na primer:

```
movw nizrec(,%ecx,2), %ax
```

tada treba izraz `nizrec(,%ecx,2)` postaviti u gornju liniju (na primer, obeležiti ga mišem), a zatim odabrati opciju *Memory* iz menija *Data*. U dijalogu koji će se otvoriti treba odabrati broj elemenata koji se želi prikazati, brojni sistem u kome se žele prikazati vrednosti i veličinu jednog elementa (u poslednjem polju će se naći izraz koji određuje memorijsku lokaciju). Na primer,

ukoliko treba prikazati jedan podatak na koga ukazuje gornji izraz, u prvo polje bi se upisalo `1`, u drugom bi se odabrao brojni sistem, na primer `hex`, a u trećem bi se odabrala opcija `halfwords(2)` koja označava podatke veličine 2 bajta, pošto naredba `movw` radi sa 2-bajtnim vrednostima. Nakon klika na dugme *Display*, traženi podatak bi se prikazao u delu za prikaz podataka.

3.4. Množenje i deljenje pomoću sabiranja i oduzimanja

Množenje pomoću sabiranja je trivijalna operacija. Ukoliko treba pomnožiti dva broja *a* i *b*, prvo se rezultat postavi na nulu, a zatim se *b* puta na rezultat doda vrednost *a*. Deljenje pomoću oduzimanja je slično, treba izbrojati koliko puta se broj *b* može oduzeti od broja *a*:

```
R = 0;
a = 3;
b = 8;
for (;a > 0;a--)
     r += b;

     (a)
```

```
r = 0;
a = 8;
b = 3;
while (a >= b) {
     a -= b;
     r++;
}
     (b)
```

Slika 6: Množenje pomoću sabiranja (a) i deljenje pomoću oduzimanja (b)

3.5. Višestruka preciznost

Prilikom rada u višestrukoj preciznosti operacije se izvode u više koraka, od kojih svaki rukuje delovima operanada. Kod *80386*, jednostruka preciznost odgovara kapacitetu registara opšte namene (32 bita), a postoje i naredbe koje rade sa 8-bitnim i 16-bitnim vrednostima, kao i nekoliko njih koje kao operand ili rezultat koriste registarski par `edx:eax` (ova notacija znači da registar **edx** sadrži bite veće, a `eax` bite manje težine). *80386* spada u *little endian* procesore, što znači da se u memoriji nalazi prvo najniži, a posle viši delovi reči.

Brojevi u dvostrukoj preciznosti se u sekciji podataka mogu zadati na više načina. Jedan bi bio da se niži i viši deo zasebno, jedan za drugim, navedu u okviru `.long` direktive, a drugi je korišćenje `.quad` direktive:

```
a1: .long 20,10              #vrednost 42949672980₁₀ = A00000014₁₆
a2: .quad 0xA00000014        #vrednost 42949672980₁₀ = A00000014₁₆
```

Da bi se 64-bitna promenljiva prikazala u dibageru (bez obzira na to kako je definisana), potrebno je iz menija *Data* odabrati opciju *Memory* i popuniti polja na sledeći način: *Examine* treba postaviti na `1` (ili na više, ako se želi posmatrati niz 64-bitnih brojeva), sledeće polje treba postaviti na željeni brojni sistem (najčešće `hex`, `decimal` ili `unsigned`), naredno polje treba postaviti na `giants(8)` (posmatranje 8-bajtnih podataka), a u poslednjem polju treba upisati naziv promenljive sa znakom `&` ispred (za gornji primer bi bilo `&a1`, odnosno `&a2`). Nakon klika na *Display*, prikazaće se sadržaj 64-bitne promenljive kao 64-bitna vrednost.

Sabiranje u dvostrukoj preciznosti

80386 naredba za sabiranje je `add`. Ona sabira operande i rezultat smešta u odredišni operand, postavljajući pri tom indikatore na vrednosti koje odgovaraju rezultatu. Još jedna naredba za sabiranje je `adc`, koja pored operanada uzima u obzir i vrednost indikatora `c`, koristeći je kao početni prenos na najnižoj poziciji. Sabiranje u višestrukoj preciznosti se radi tako što se najmanje značajne reči operanada saberu pomoću `add`, dok se značajnije reči sabiraju pomoću `adc`, koristeći na taj način prenose iz prethodnih sabiranja. Naredba `inc` povećava odredišni operand za jedan (međutim, ne utiče na indikator `c`, pa treba biti obazriv kod korišćenja ove naredbe za rad u višestrukoj preciznosti).

Sabiranje neoznačenih brojeva

Operandi se u ovom primeru ne nalaze u registrima, već su zapisani u određene memorijske lokacije. Naredbe koje ih upotrebljavaju pristupaju im koristeći memorijsko ili direktno adresiranje. Rezultat će biti u registarskom paru `edx:eax`.

Prenos kod sabiranja nižih reči uzeće se u obzir prilikom sabiranja viših. Prenos kod konačnog sabiranja u ovom slučaju znači izlazak van opsega. Prema tome, stanje indikatora `c` posle svih operacija može se iskoristiti kao indikator greške. Ovde će se to upotrebiti za postavljanje promenljive `greska`, uz pomoć naredbe uslovnog skoka `jnc` (*jump if no carry*), kojoj je uslov za grananje `c`=0.

```
a:   .quad 0x8000      # a = 0x8000 (0x000080001)
b:   .long 0x8001,1    # b = 0x100080001
greska:  .byte 0
...
     movb $0, greska
     movl a, %eax
     addl b, %eax
     movl a+4, %edx
     adcl b+4, %edx
     jnc kraj
     incb greska
kraj:
     movl $1, %eax
     movl $0, %ebx
     int $0x80
```

Podrazumeva se da se prvi deo kôda nalazi u `data` sekciji, a drugi u `text` sekciji.

Sabiranje označenih brojeva

Postupak je veoma sličan prethodnom:

```
a:   .quad 0x8000      # a = 0x8000 (0x00008000)
b:   .long 0x8001,1    # b = 0x100080001
greska:      .byte 0
...
     movb $0, greska
     movl a, %eax
     addl b, %eax
     movl a+4, %edx
     adcl b+4, %edx
     jno kraj
     incb greska
kraj:
     movl $1, %eax
     movl $0, %ebx
     int $0x80
```

Razlika je u tome što se za detekciju izlaska van opsega koristi indikator `o` umesto indikatora `c`. Shodno tome, upotrebljena je naredba `jno` (*jump if no overflow*).

Postupak za oduzimanje u dvostrukoj preciznosti je sličan sabiranju. Za oduzimanje nižih delova se koristi naredba `sub`, dok se za oduzimanje viših koristi `sbb` (*subtraction with borrow*), koja radi isto što i `sub`, samo što u obzir uzima i vrednost indikatora `c` (dodaje ga na vrednost izvornog operanda). Naredba koja vrši umanjivanje opranda za 1 je `dec` (isto kao i naredba `inc`, ne utiče na indikator `c`).

3.6. Numerički tipovi podataka na *C*-u i asembleru

Sledeća tabela ilustruje osnovne tipove podataka programskog jezika *C* i odgovarajuće direktive za definisanje tipova podataka na asembleru. U tabeli je takođe data kolona sa nazivom tipa podatka u *DDD* dibageru.

Veličina (biti)	Opseg	C	Asembler	Dibager
8	-128 do +127	`char`	`.byte`	`bytes`
8	0 do 255	`unsigned char`	`.byte`	`bytes`
16	-32768 do +32767	`short int`	`.word`	`halfwords`
16	0 do 65535	`unsigned short int`	`.word`	`halfwords`
32	-2147483648 do 2147483647	`int`	`.long`	`words`
32	0 do 4,294,967,295	`unsigned int`	`.long`	`words`
64	-2^{63} do $+2^{63}-1$	`long long`	`.quad`	`giants`
64	0 do $+2^{64}-1$	`unsigned long long`	`.quad`	`giants`

Tabela 2: Numerički tipovi podataka na *C*-u i asembleru

3.7. Domaći zadatak

1. Napisati asemblerski program za poređenje dva označena broja (*a* i *b*) u dvostrukoj preciznosti. Rezultat rada programa treba da se nađe u registru `eax`, i to:
   - `0` ako je $a = b$
   - `-1` ako je $a < b$
   - `1` ako je $a > b$

4. Nizovi

4.1. Pristupanje elementima niza

Pristupanje elementima niza zahteva korišćenje **indeksnog** i/ili **indirektnog** adresiranja. Ukoliko se radi sa nizovima čiji elementi su veličine 1, 2, 4 ili 8 bajta, tada se može koristiti indeksno adresiranje, kao u primeru:

```
niz: .long -3,6,7,4,12
...
movl $2, %eax              # indeks u %eax
movl niz(,%eax,4), %ebx    # 7 u %ebx
```

U ovom primeru, registar `eax` je iskorišćen kao indeksni registar, a broj `4` označava veličinu jednog elementa niza. Adresa kojoj će se pristupiti se dobija kao zbir adrese labele `niz` i sadržaja indeksnog registra pomnoženog sa `4` (veličina jednog elementa niza je 4 bajta).

Koristeći indirektno adresiranje, istom elementu bi se moglo pristupiti na sledeći način:

```
movl $niz, %eax            # adresa niza u %eax
addl $8, %eax              # računanje adrese elementa sa indeksom 2
movl (%eax), %ebx          # 7 u %ebx
```

Ovde je registar `eax` iskorišćen kao indirektni registar. Adresa kojoj će se pristupiti predstavlja sadržaj registra `eax`.

Ova dva adresiranja se mogu i iskombinovati:

```
movl $niz, %ecx                # adresa niza u %ecx
movl $2, %eax                  # indeks u %eax
movl (%ecx, %eax, 4), %ebx     # 7 u %ebx
```

U ovom primeru, registar `eax` ima ulogu indeksnog registra, a registar `ecx` ima ulogu indirektnog registra. Adresa kojoj će se pristupiti se dobija kao zbir vrednosti registra `ecx` i sadržaja registra `eax` pomnoženog sa `4`. Ova varijanta je naročito pogodna ukoliko pristup elementima niza treba parametrizovati.

Vrednost kojom se množi indeksni registar može biti 1, 2, 4 ili 8. Za druge veličine elemenata niza se mora koristiti indirektno adresiranje, i tada se za svako pristupanje elementu niza mora izračunati njegova adresa.

DDD omogućava jednostavan prikaz sadržaja 32-bitnih nizova. Za 32-bitne nizove je dovoljno iza naziva niza staviti sufiks `@n`, gde je *n* broj članova niza koje treba prikazati. Na primer, ako se niz zove `nizrec` i ima 5 članova, treba kliknuti na naziv niza, zatim u gornjoj liniji iza naziva dodati `@5` (`nizrec@5`) i kliknuti na dugme *Display*. Za 8-bitne, 16-bitne i 64-bitne nizove treba iskoristiti opciju *Memory* iz menija *Data*, kako je ranije već opisano.

4.2. Nizovi brojeva

Nizovi brojeva će biti predstavljeni na primeru programa za sumiranje elemenata niza. Na slici 7 je prikazan C program za sumiranje elemenata niza čiji su elementi označeni celi brojevi:

```
#define NELEM 10
int niz[NELEM] = {1,2,3,4,5,6,7,8,9,10};
int suma = 0;
int i;

for (i = 0; i < NELEM; i++)
     suma = suma + niz[i];
```

Slika 7: C program za sumiranje elemenata niza

Varijanta 1 – indeksno adresiranje

```
NELEM    = 10
niz:     .long 1,2,3,4,5,6,7,8,9,10
suma:    .long 0

...

main:
         movl $0, %esi         #indeksni registar
         movl $0, %eax         #eax se koristi za računanje sume
petlja:
         addl niz(,%esi,4), %eax
         incl %esi
         cmpl $NELEM, %esi
         jl petlja
kraj:
         movl %eax, suma
         movl $1, %eax
         movl $0, %ebx
         int $0x80
```

Važno je napomenuti da, **ako** se traži da se rezultat na kraju nalazi u nekoj promenljivoj (`suma`), to treba i **uraditi** u programu. Program u toku svog rada koristi registar `eax` za računanje sume (x86 procesori ne mogu imati dva memorijska operanda u istoj naredbi), a na kraju programa se ta izračunata vrednost postavlja u promenljivu `suma`.

Varijanta 2 – indirektno adresiranje

Sekcija podataka je ista kao u prethodnoj varijnti, pa se ne prikazuje:

```
main:
         movl $niz, %esi       #registar za indirektno adresiranje
         movl $0, %eax         #eax se koristi za računanje sume
petlja:
         addl (%esi), %eax
         addl $4, %esi         #long zauzima 4 bajta
         cmpl $niz+NELEM*4, %esi
         jl petlja
kraj:
         movl %eax, suma
         movl $1, %eax
         movl $0, %ebx
         int $0x80
```

Ova varijanta programa je podesna za pretvaranje u potprogram, pošto se u glavnoj petlji nigde eksplicitno ne navodi adresa niza (ona je pre glavne petlje postavljena u registar `esi`).

Varijanta 3 – indirektno + indeksno adresiranje + optimizacija

Sekcija podataka je ista kao u prvoj varijnti, pa se ne prikazuje:

```
main:
        movl $niz, %esi           #bazni registar
        movl $NELEM-1, %edi       #indeksni registar
        movl $0, %eax             #eax se koristi za računanje sume
petlja:
        addl (%esi, %edi, 4), %eax
        decl %edi
        jns petlja
kraj:
        movl %eax, suma
        movl $1, %eax
        movl $0, %ebx
        int $0x80
```

Ova varijanta je takođe podesna za korišćenje u potprogramima. Dodatno, kod ove varijante nema eksplicitne provere da li se došlo do kraja niza. Inicijalno, indeksni registar je postavljen tako da pokazuje na poslednji element u nizu. U toku izvršavanja petlje, indeksni registar se smanjuje, a indikaciju završetka nam daje `s` (*sign*) indikator. Dok god je vrednost u indeksnom registru pozitivna ili nula nakon oduzimanja (što odgovara validnim indeksima elemenata u nizu), skakaće se na labelu `petlja`. Čim ova vrednost postane negativna (što nam označava indikator `s` na koga naredba `dec` utiče), ciklus se završava. Ovakva provera ima ograničenje da se ne može koristiti sa nizovima koji imaju više od 2^{31} elemenata (ako pogledamo binarnu reprezentaciju broja 2^{31}, a to je jedinica koju sledi 31 nula, videćemo da je za neoznačene vrednosti od 2^{31} do 2^{32}-1 najznačajniji bit postavljen na 1, što znači da će i `s` indikator biti postavljen na 1, pa bi se petlja završila odmah nakon obrade prvog elementa).

Varijanta 4 – indirektno + indeksno adresiranje + loop naredba

Sekcija podataka je ista kao u prvoj varijnti, pa se ne prikazuje:

```
main:
        movl $niz, %esi           #bazni registar
        movl $NELEM, %ecx         #indeksni registar
        movl $0, %eax             #eax se koristi za računanje sume
petlja:
        addl -4(%esi, %ecx, 4), %eax
        loopl petlja
kraj:
        movl %eax, suma
        movl $1, %eax
        movl $0, %ebx
        int $0x80
```

I ova varijanta je podesna za korišćenje u potprogramima. Kod ove varijante se koristi naredba `loop` za realizaciju petlje. U svojoj 32-bitnoj varijanti (`loopl`), ova naredba radi sledeće: umanjuje vrednost `ecx` registra za `1` i ukoliko je ta vrednost nakon umanjenja različita od nule, skače na labelu. Ako pogledamo vrednosti koje će uzimati indeksni registar (`ecx`), videćemo da će njihov opseg biti od `1` do `NELEM`, a ne od `0` do `NELEM-1`, kao što bi indeksno adresiranje zahtevalo. Iz tog razloga, opseg je pomeren za jednu 32-bitnu poziciju nadole dodavanjem konstante `-4` u izraz za adresiranje niza (praktično, sve adrese kojima se pristupa su pomerene za 4 bajta nadole, pa efektivno dobijamo opseg indeksa od `0` do `NELEM-1`).

Drugi primer – brojanje dana sa određenom temperaturom

Drugi primer se bavi nizom brojeva koji predstavljaju zabeležene temperature u toku jedne godine. Zadatak je da se u zadatom podintervalu prebroje dani kod kojih je temperatura unutar zadatog opsega.

Na slici 8 prikazan je *C* program za rešavanje ovog problema. Dužina niza `dani` određena je konstantom `BROJ_DANA`. Indeksiranje niza `dani` radi se sa osnovom nula (prvi element niza ima indeks nula), kao što je uobičajeno na *C*-u. Algoritam ispituje validnost ulaznih podataka:

- da li je indeks prvog dana pozitivan?
- da li je indeks poslednjeg dana manji od ukupnog broja dana?

- da li je indeks prvog dana manji ili jednak indeksu poslednjeg?

  Ako bilo koja od ovih provera ne uspe, postavlja se globalna promenljiva `greska`.

```
#define BROJ_DANA 365
int g_donja, g_gornja;
int d_prvi, d_poslednji;
int broj_d;
int greska;
int dani[] = { ... 365 vrednosti tempratura ...};
greska = 0;
if (d_prvi < 0 || d_poslednji >= BROJ_DANA
    || d_poslednji < d_prvi)
    greska++;
else {
    int i;
    broj_d = 0;
    for (i = d_prvi; i <= d_poslednji; i++)
        if (g_donja <= dani[i] && g_gornja >= dani[i])
        broj_d++;
}
```

Slika 8: C program za brojanje dana sa temperaturom u zadatim granicama

Asemblerski ekvivalent dat je u nastavku teksta. On se razlikuje od prethodnog programa po tome što je smanjen broj dana (na sedam), da asemblerski program ne bi bio predugačak (suština algoritma se ovime ne menja) i po tome što u asemblerskom programu nije odrađena većina provera.

```
broj_dana    = 7
g_donja:     .long -1
g_gornja:    .long 3
d_prvi:      .word 1
d_poslednji: .word 5
broj_d:      .word 0
greska:      .byte 0
dani:        .long 5, -1, 2, 4
             .long -2, -3, 3
...

main:
    movl $0, %ebx
    movw d_prvi, %bx
    cmpw $0, %bx
    jl i_greska
    movl $0, %ecx
    movw d_poslednji, %cx
    subl %ebx, %ecx
    incl %ecx
    movw $0, broj_djedan_dan:
    movl dani(,%ebx,4), %eax
    cmpl g_donja, %eax
    jl van_opsega
    cmpl g_gornja, %eax
    jg van_opsega
    incw broj_d
van_opsega:
    incl %ebx
    loopl jedan_dan
    jmp kraj
i_greska:
    incb greska
kraj:
    movl $1, %eax
    movl $0, %ebx
    int $0x80
```

Napomene uz primer:

- Pojedinačnim vrednostima niza pristupa se pomoću indeksnog adresiranja.
- Kao brojač iteracija glavne petlje programa (koja počinje labelom `jedan_dan`) koristi se registar `ecx`. Za njegovo dekrementiranje, testiranje i uslovno grananje moguće je koristiti specijalnu naredbu `loop` koja podrazumeva korišćenje samo ovog registra (zbog neortogonalnosti skupa naredbi *80386*). U ovom slučaju, `loop jedan_dan` bilo bi ekvivalentno nizu naredbi

```
decl %ecx
jnz jedan_dan
```

- Obratiti pažnju na korišćenje 16-bitnih i 32-bitnih registara (na primer, paralelno se koriste kako registar `bx`, tako i registar `ebx`).

4.3. Stringovi

Stringovi su nizovi znakova. Svaki znak je u računaru predstavljen celim brojem. Za kodiranje znakova koristi se ***ASCII*** standard (*American Standard Code for Information Interchange*) ili neka od njegovih proširenih varijanti. U osnovnom *ASCII* kodu, svaki znak zauzima jedan bajt. Za označavanje kraja stringa će se koristiti konvencija jezika *C*, a to je da se na kraju stringa nalazi ***ASCII NUL*** bajt, koji ima vrednost nula.

Direktiva `.ascii` služi za smeštanje *ASCII* teksta u segment podataka:

```
s:  .ascii "neki tekst\0"
```

Znaci "`\0`" na kraju stringa služe za definisanje *ASCII NUL* znaka (isto kao na *C*-u).

Za prikaz stringova se može iskoristiti dugme *String* sa panela za prikaz promenljivih. Rad sa stringovima će biti predstavljen na primeru izbacivanja znakova razmaka sa kraja i sa početka stringa.

Ukoliko je potrebno samo zauzeti prostor za string ili niz, može se iskoristiti direktiva `.fill`:

```
.fill broj, veličina, vrednost
```

Ova direktiva služi da se u memoriju upiše `broj` elemenata dužine `veličina` bajtova (može biti 1, 2 ili 4) sa vrednošću `vrednost`. Parametri `veličina` i `vrednost` su opcioni (podrazumevane vrednosti su 1 i 0, respektivno).

```
char s[] = "  neki tekst   \0"     //tekst
char *p = s;                      //pomoćni pokazivač
//izbacivanje razmaka sa kraja
while (*p) p++;                   //nalaženje kraja stringa
while (*(p-1) == ' ') p--;        //nalaženje prvog razmaka sa kraja
*p = 0;                           //postavljanje nule na taj razmak
//izbacivanje razmaka sa početka
while (*s == ' ') {               //dok god je na početku razmak
    p = s;                        //kreće se od početka stringa
    while (*(p+1)) {              //i ide se do kraja
            *p = *(p+1);          //niz[i] = niz[i+1]
            p++; }                //sledeći element
    *p = 0;                       //nula na kraju stringa
}
```

Slika 9: C program za izbacivanje razmaka sa početka i kraja stringa

Izbacivanje znakova razmaka sa kraja stringa se može obaviti jednostavnim ubacivanjem oznake kraja stringa na odgovarajuće mesto. Izbacivanje znakova razmaka sa početka stringa podrazumeva pomeranje sadržaja stringa za određeni broj pozicija ka početku.

Asemblerska verzija programa sadrži samo izbacivanje znakova razmaka sa kraja stringa. Za razliku od prethodnog primera, ovde se za pristupanje elementima niza koriste indeksno i indirektno adresiranje.

```
s: .ascii "  abcd   \0"
...
     movl $s, %eax          #početna adresa stringa u %eax
kraj_s:                     #određivanje kraja stringa
     cmpb $0, (%eax)        #da li se na tekućoj poziciji nalazi NUL?
     je razmaci
     incl %eax              #ako ne, uvećaj %eax
     jmp kraj_s
razmaci:                    #određivanje broja razmaka koje treba izbaciti
     cmpb $' ', -1(%eax)    #da li se na poziciji ispred tekuće nalazi
                            #znak različit od razmaka?
     jne izbaci
     decl %eax              #ako ne, umanji %eax
     jmp razmaci
izbaci:
     movb $0, (%eax)        #upisivanje NUL znaka iza poslednjeg znaka
                            #koji je različit od razmaka
kraj:
     movl $1, %eax
     movl $0, %ebx
     int $0x80
```

Adresiranje `-1(%eax)` znači da se posmatra lokacija sa adresom koja je za 1 manja od vrednosti `%eax` registra (kod stringova, jedan element niza zauzima 1 bajt). Ovakvo adresiranje je korisno ukoliko treba adresirati lokaciju koja je za fiksan broj mesta udaljena od tekuće, na primer, adresiranje prethodnog ili sledećeg elementa niza (pri tome treba voditi računa da se konstanta uskladi sa veličinom elemenata niza).

Ukoliko je potrebna *ASCII* vrednost nekog znaka, ona se može dobiti iz tabele *ASCII* znakova (nalazi se u dodatku) ili navođenjem znaka unutar apostrofa (na primer, `'a'` označava *ASCII* vrednost znaka `a`).

4.4. Naredbe za množenje i deljenje

Naredba za množenje neoznačenih operanada je `mul`. U zavisnosti od veličine operanda, naredba `mul` radi sledeće (operand može biti promenljiva/memorijska lokacija ili registar):

```
mulb operand     #al*operand -> ax
mulw operand     #ax*operand -> dx:ax
mull operand     #eax*operand -> edx:eax
```

Deljenje neoznačenih brojeva se može obaviti naredbom `div` koja, u zavisnosti od veličine operanada, radi sledeće (operand može biti promenljiva/memorijska lokacija ili registar):

```
divb operand     #ax/operand -> al, ostatak -> ah
divw operand     #dx:ax/operand -> ax, ostatak -> dx
divl operand     #edx:eax/operand -> eax, ostatak -> edx
```

Izvršavanje naredbe `div` može dovesti do pojave izuzetka. Ovo se može desiti u dva slučaja:

- ukoliko je vrednost operanda jednaka nuli
- ukoliko je dobijena vrednost prevelika da stane u odredište

Postoje i naredbe koje rade množenje (`imul`) i deljenje (`idiv`) označenih brojeva, a detalji o njima se mogu naći u glavi 11 (Pregled korišćenih naredbi).

4.5. Testiranje programa

Kada se napiše program, potrebno je naravno i proveriti da li on ispravno radi. Razmotrimo primer programa koji vrši sumiranje elemenata niza (Varijanta 1, na strani 24). Ukoliko je ulaz u program zadat sa:

```
NELEM    = 10
niz:     .long 1,2,3,4,5,6,7,8,9,10
suma:    .long 0
```

tada će rezultat rada programa biti vrednost 55 i ona će biti smeštena u promenljivu `suma`. Na osnovu ovog primera, mogli bismo reći kako je program ispravan. Međutim, testiranje ispravnosti programa se ne može zasnivati na samo jednom primeru. Program se mora istestirati za više različitih ulaza, i videti da li daje ispravan rezultat. Pretpostavimo da je ulaz u program zadat sa:

```
NELEM    = 10
niz:     .long 1,2,3,4,5,6,7,8,9,2147483632
suma:    .long 0
```

Šta će u ovom slučaju biti rezultat? Ako prihvatimo da asemblerski program radi sa označenim vrednostima (pošto tako radi i *C* program na osnovu koga je nastao asemblerski), vrlo lako se može utvrditi da će program kao rezultat dati vrednost -2147483618 koja nikako nije tačna. Zbog čega se desila greška?

Ako se pokuša izvršavanje programa korak po korak, može se videti da u trenutku kada se rezultatu doda vrednost 2147483632, on odjednom postane negativan. U tom istom trenutku se i indikator *overflow* postavlja na jedinicu, što nam daje do znanja da dobijeni rezultat više ne može da stane u 32 bita.

Kako se program može dopuniti da radi ispravnije? Na primer, tako što će se uvesti promenljiva `greska` i dodati provera prekoračenja nakon svakog sabiranja. Grešku na početku postavimo na `0`, a ako se desi prekoračenje, postavimo je na `1`. Ovako prošireni program može dati ispravan rezultat za mnogo širi opseg ulaza nego početna varijanta.

Što se program testira sa više različitih ulaza, to postoji veća sigurnost u njegovu ispravnost. Generalno gledano, program treba testirati sa dve grupe ulaza:

- ulazi koji treba da daju ispravan rezultat

- ulazi koji treba da izazovu grešku

U test primere uvek treba uvrstiti i vrednosti graničnih slučajeva, kao i vrednosti oko graničnih slučajeva. Na primer, ako testiramo program koji prebrojava koliko reči koje počinju slovom „a“ u nekom stringu, test primeri bi trebali da uključe prazan string, string od samo jednog znaka koji nije „a“, string od samo jednog znaka „a“, string sa više reči bez ijedne koja počinje sa „a“ ali ima slovo a u sredini ili na kraju, nekoliko varijanti stringova gde više reči počinje slovom „a“, string u kome sve reči počinju slovom „a“, i tako dalje.

4.6. Domaći zadaci

1. Napisati asemblerski program za sortiranje niza 16-bitnih vrednosti.
2. Prepraviti program za brojanje dana tako da se umesto indeksnog adresiranja za pristupanje elementima niza koristi indirektno adresiranje.

5. Sistemski pozivi

Program koji treba da komunicira sa okruženjem obraćajući se periferijskim uređajima može to da čini na dva načina: direktnim pristupom hardverskom interfejsu uređaja (na primer, portovima u ulazno/izlaznom adresnom prostoru koji odgovaraju hardverskim registrima), ili korišćenjem usluga operativnog sistema. Prvi način se zbog svoje složenosti koristi skoro isključivo kod programa specijalizovane namene, kao što su upravljački programi za mikrokontrolere ili drajveri. Drugi način podrazumeva obraćanje sistemu putem **sistemskih poziva**.

5.1. Sistemski pozivi

Sistemski pozivi omogućuju pozivanje operacija operativnog sistema. Kod sistema gde su korisnički i sistemski prostor strogo međusobno izolovani, oni su ujedno i jedini način za komunikaciju korisničkih programa sa okruženjem. Direktan pristup hardveru dopušten je samo sistemskim procesima. Čak i tamo gde ovakva ograničenja ne postoje, sistemski pozivi su osnovni nivo apstrakcije hardvera - oni omogućuju izvršavanje operacija bez poznavanja detalja fizičke arhitekture sistema.

Ilustracije radi, razmotrimo operaciju prikaza teksta na ekranu korišćenjem direktnog pristupa hardveru. Program koji želi da ispiše tekst direktno pristupajući video memoriji mora da zna koji se tip grafičke kartice nalazi u računaru, gde se nalazi početak video memorije, kako je ona organizovana, koja je rezolucija ekrana, koji su kodovi znakova i boja prilikom ispisa. Dalje, ako se prilikom upisivanja pređe ispod poslednje linije ekrana, sadržaj video memorije se mora pomeriti za jednu liniju naviše, itd. Korišćenjem sistemskih poziva, sve ovo pada na teret drajvera koji rukuje grafičkom karticom.

Pod *Linux*-om na procesoru *80386*, sistemskim pozivima se pristupa korišćenjem naredbe softverskog prekida, `int`, sa vektorom `0x80`. Procesor *80386* se u toku izvršavanja ove naredbe ponaša slično kao kod prijema hardverskog prekida: sačuva na steku sadržaj statusnog registra i programskog brojača, pronađe memorijsku lokaciju sa adresom obrađivača prekida i započne njegovo izvršavanje. Za razliku od hardverskog, softverski prekid se ne može zabraniti.

Pre poziva naredbe `int`, u registar `eax` treba postaviti broj sistemskog poziva, dok u registre `ebx`, `ecx`, `edx`, `esi` i `edi` treba postaviti eventualne argumente poziva (sadržaj ovih registara, kao i registra `ebp`, ostaje nepromenjen nakon povratka iz sistemskog poziva). Broj argumenata zavisi od sistemskog poziva koji se želi koristiti. Nakon izvršenja sistemskog poziva, registar `eax` sadrži informaciju o izvršenju poziva. Negativan broj označava grešku, dok nula ili pozitivan označava uspešno izvršenje.

5.2. Ulaz, izlaz i završetak programa

Svaki startovani program na raspolaganju ima 3 unapred otvorena fajla: *stdin* (standardni ulaz), *stdout* (standardni izlaz) i *stderr* (standardni izlaz za greške).

Sistemski poziv za ulaz (*read*) koristi sledeće argumente:

`eax` ← 3 (broj sistemskog poziva za ulaz)

`ebx` ← deskriptor fajla (za standardni ulaz, *stdin*, ova vrednost je 0)

`ecx` ← adresa ulaznog bafera

`edx` ← veličina bafera

Nakon povratka iz poziva, registar `eax` će sadržati broj pročitanih bajtova, odnosno znakova. Ukoliko je došlo do neke greške, ovaj broj će biti negativan. Treba imati u vidu da se kod unosa sa tastature, na kraju niza unetih znakova nalazi kôd 10 (kôd za taster *Enter*), kao i to da, ako se unese više od maksimalnog broja znakova, u bafer će se upisati samo onoliko koliko je specificirano registrom `edx`.

Sistemski poziv za izlaz (*write*) koristi sledeće argumente:

`eax` ← 4 (broj sistemskog poziva za izlaz)

`ebx` ← deskriptor fajla (za standardni izlaz, *stdout*, ova vrednost je 1,
dok je za standardni izlaz za greške, *stderr*, ova vrednost 2)

`ecx` ← adresa izlaznog bafera

`edx` ← veličina bafera

Nakon povratka iz poziva, registar `eax` će sadržati broj ispisanih bajtova, odnosno znakova. Ukoliko je došlo do neke greške, ovaj broj će biti negativan. Treba imati u vidu da će se ispisati onoliko znakova koliko se navede u `edx` registru, bez obzira na stvarnu veličinu bafera.

Za regularan završetak programa potreban je sistemski poziv *exit*, koji saopštava operativnom sistemu da je nastupio kraj izvršavanja programa. Sistemski poziv za završetak programa koristi sledeće argumente:

`eax` ← 1 (broj sistemskog poziva za završetak programa)

`ebx` ← izlazni kôd (kôd greške)

Po konvenciji, u registar `ebx` se stavlja 0 ukoliko je izvršavanje programa prošlo bez grešaka, odnosno broj različit od 0, ukoliko je došlo do neke greške.

U nastavku su dati primeri programa koji koriste pomenute sistemske pozive. Ovi primeri koriste naredbu `lea` (*Load Effective Address*). Naredba `lea` izračunava **adresu** izvornog operanda (koji mora biti memorijski) i smešta je u odredišni operand (koji mora biti registarski). Kod ove naredbe računanje adrese se obavlja u toku njenog izvršavanja (dinamički), dok se kod naredbe:

```
movl $dec_br, %edi
```

to radi u toku prevođenja programa (statički).

Primer 1:

```
#primer unosa stringa
str_max = 40
str: .fill string_max,1,42

movl $3, %eax
movl $0, %ebx
leal str, %ecx
movl $str_max, %edx
int $0x80

movl $1, %eax
movl $0, %ebx
int $0x80
```

Primer 2:

```
#primer ispisa stringa
str: .ascii “Neki tekst\0”
str_len = 11 #može i 10, \0 ionako nije znak koji se može ispiati

movl $4, %eax
movl $1, %ebx
leal str, %ecx
movl $str_len, %edx
int $0x80

movl $1, %eax
movl $0, %ebx
int $0x80
```

U drugom primeru se dužina stringa zadaje kao konstanta i programer je dužan da postavi njenu tačnu veličinu.

Primer 3:

Treći primer se od drugog razlikuje samo u jednom detalju, a to je način određivanja dužine **statičkog** stringa (stringa čija se dužina **zna** u vreme prevođenja programa):

```
#primer ispisa stringa
str: .ascii “Neki tekst\0”
str_len = . – string

movl $4, %eax
movl $1, %ebx
leal str, %ecx
movl $str_len, %edx
int $0x80

movl $1, %eax
movl $0, %ebx
int $0x80
```

Ovde je korišćena direktiva u obliku izraza

```
labela1 = .–labela2
```

Tačka u izrazu označava **brojač lokacija** koji predstavlja adresu do koje se trenutno stiglo pri asembliranju. Ovime se labeli `labela1` dodeljuje razlika pomenute adrese i adrese labele `labela2`, čime se efektivno izračunava broj bajtova koji se nalaze između te dve labele (u ovom primeru time se izračunava dužina stringa). Bitno je uočiti da korišćenje brojača lokacija zavisi od mesta na kome se obavlja računanje (računaje dužine stringa je **neophodno** da se obavi odmah nakon definicije stringa). Još jednom treba napomenuti da se ovako može izračunati dužina samo **statički** zadatih stringova, odnosno bilo kakvih drugih statički zadatih struktura podataka. Stringovima koji se genrišu tokom rada programa se na ovaj način **ne može** odrediti dužina.

Kratki segmenti kôda koji se često ponavljaju (kao što su sistemski pozivi) mogu biti veoma pogodni za pretvaranje u makro. Više o tome se može naći u dodatku D.

6. Potprogrami

U računarstvu se pod pojmom potprogram podrazumeva deo koda unutar većeg programa koji obavlja specifične zadatke i relativno je nezavisan od ostatka koda. Potprogram je obično tako napravljen da može da bude pokrenut (“pozvan”) više puta sa različitih mesta u toku izvršavanja programa, uključujući i pozive iz drugih potprograma. Po završetku potprograma izvršavanje se nastavlja od instrukcije neposredno posle poziva [Literatura, 10].

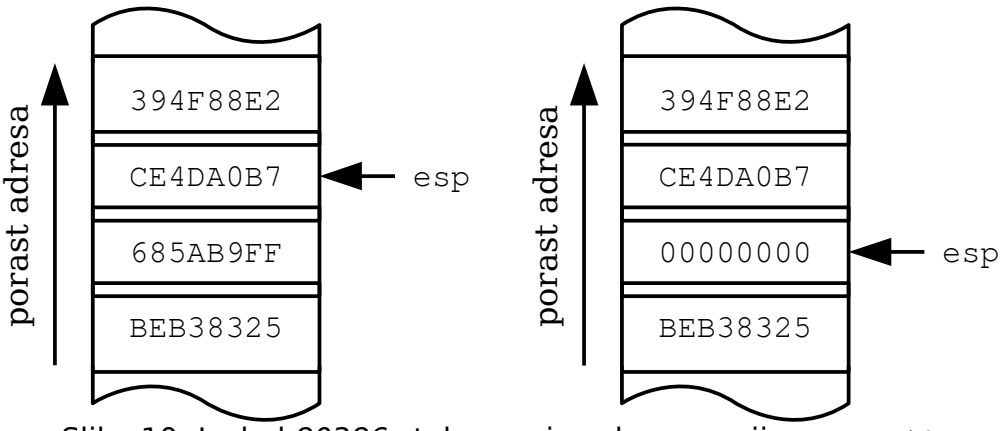
Kod procedurnih programskih jezika višeg nivoa potprogrami su osnovni način za modularizaciju kôda. Osim razbijanja programa na manje delove, po pretpostavci jednostavnije i lakše za razumevanje i održavanje, važna namena potprograma je **parametrizacija** problema: mogućnost da se za određenu grupu operacija odrede njeni parametri, a da se prilikom izvršavanja zadaju različite vrednosti parametara (argumenti) kojima će te operacije baratati.

Potprogrami koji kao rezultat izvršavanja vraćaju neku vrednost obično se nazivaju **funkcije**. Neki programski jezici striktno razlikuju ovakve potprograme (funkcije) od onih koji nemaju povratnu vrednost (procedure). S druge strane, *C* **sve** potprograme naziva funkcijama, s tim da se kao ekvivalent procedura koriste funkcije čiji je tip eksplicitno deklarisan kao `void`. Pozivi ovakvih funkcija ne smeju se pojaviti u izrazima.

Mikroprocesori po pravilu imaju operacije i memorijske strukture za implementaciju potprograma. *80386* nije izuzetak: za poziv potprograma postoji naredba `call`, povratak iz potprograma omogućava naredba `ret`, a da bi ove operacije automatski pamtile i preuzimale povratnu adresu (od koje se nastavlja izvršavanje po završetku potprograma), koristi se memorijska struktura zvana **stek** (*stack*).

Stek je tzv. *LIFO* (*Last In, First Out*) struktura: poslednji smešten element će biti prvi preuzet. Kod *80386*, `call` i `ret` koriste stek automatski, ali postoje i posebne operacije za smeštanje (`push`) i preuzimanje (`pop`) vrednosti. U 32-bitnom režimu rada, na stek se **uvek** smeštaju 32-bitne vrednosti (jedna ili više njih). Kao pokazivač na vrh steka koristi se registar `esp`.

Naredba `push` prvo smanjuje `esp` za 4, a zatim na lokaciju na koju pokazuje nova vrednost `esp` smešta izvorni operand, dok `pop` radi obrnuto: preuzima vrednost sa lokacije na koju pokazuje `esp`, smešta je u odredišni operand, i povećava `esp` za 4. Može se videti da prilikom punjenja stek **raste ka nižim adresama**. Ilustracije radi, efekat operacije `pushl $0` prikazan je na slici 10.



Slika 10: Izgled *80386* steka pre i posle operacije `pushl $0`

6.1. Konvencija poziva potprograma

Prenos argumenata u potprogram i rad sa lokalnim promenljivim (ako postoje) na asemblerskom jeziku je moguće obaviti na različite načine. Način na koji određeni viši programski jezik radi ovo prevođenje naziva se njegovom **pozivnom konvencijom** (*calling convention*). Pozivna konvencija opisuje interfejs pozvanog koda: gde se parametri

smeštaju (na stek ili u registre), redosled u kom se parametri alociraju, koji registri mogu da se koriste unutar potprograma, ko je odgovoran za dealociranje parametara (sam potprogram ili kod koji ga poziva).

Na ovom kursu ćemo koristiti pozivnu konvenciju programskog jezika *C* na x86 arhitekturi koja se naziva **cdecl**. Prema ovoj konvenciji potprograme ćemo pozivati u skladu sa sledećim pravilima:

- Kôd koji poziva potprogram je odgovoran za alociranje parametara, kao i za njihovo dealociranje. Ovo omogućava postojanje promenjive liste parametara kao što je ima npr. *C* funkcija *printf()*.
- Argumenti se stavljaju na stek od poslednjeg ka prvom (*right-to-left* redosled). Na ovaj način prvi argument je uvek na poznatoj lokaciji, neposredno iznad povratne adrese.
- Povratna vrednost potprograma tj. funkcije se smešta u registar `eax` (ukoliko je povratna vrednost 64-bitna, smešta se u registarski par `edx:eax`, gde `edx` predstavlja viših 32 bita, a `eax` nižih 32 bita vrednosti).
- Registri `eax`, `ecx` i `edx` su slobodni za korišćenje unutar potprograma. Ukoliko je neophodna upotreba još nekih registara, potrebno je da se obezbedi da vrednost tih registara po izlasku iz potprograma bude neizmenjena (mogu se u toku izvršavanja potprograma sačuvati na steku).

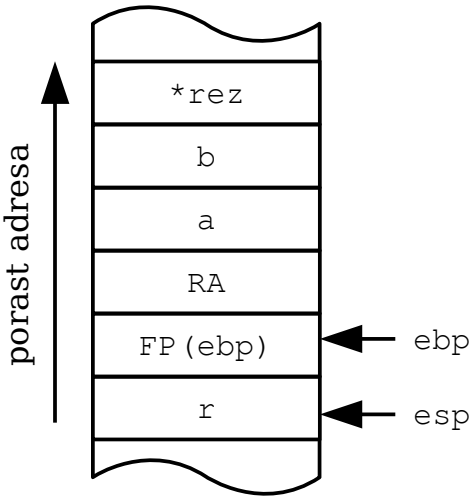
Svrha korišćenja ove konvencije je pre svega mogućnost povezivanja potprograma napisanih na asembleru sa programima pisanim na programskom jeziku *C*. Ovo je ponekad neophodno budući da je pojedine stvari (npr. optimalan pristup hardveru, optimizacija kritičnog dela koda, itd.) veoma teško ili čak nemoguće uraditi korišćenjem isključivo viših programskih jezika.

6.2. Stek frejm

Prostor za lokalne promenljive zauzima se na steku: pokazivač na vrh steka pomera se za odgovarajuću vrednost, tako da upotreba steka unutar funkcije (recimo za dalje pozive) ne utiče na sadržaj lokalnih promenljivih.

Za pristup argumentima i lokalnim promenljivama koristi se poseban pokazivački registar koji se zove **pokazivač frejma** (*frame pointer*). **Frejm** sadrži argumente, povratnu adresu, zatečenu vrednost pokazivača frejma i lokalne podatke funkcije. Zatečena vrednost pokazivača frejma se stavlja na stek na početku izvršavanja funkcije i restaurira pre povratka, čime se pozivajućoj funkciji osigurava pristup sopstvenom frejmu po nastavku njenog izvršavanja. *80386* kao pokazivač frejma koristi registar `ebp`.

```
void
nzd (int a, int b, int *rez)
{
      int r;
      while (a != b)
            if (a > b)
                  a -= b;
            else
                  b -= a;
      r = a;
      *rez = r;
}
```



(a) (b)

Slika 11: C funkcija: (a) izvorni kôd i (b) izgled frejma nakon poziva funkcije, a pre izvršavanja njenog tela

Na slici 11 prikazani su jedna *C* funkcija i izgled njenog frejma nakon inicijalizacije. **RA** (*return address*) je povratna adresa, a **FP** (*frame pointer*) prethodna vrednost pokazivača frejma (registar `ebp`).

Asemblerski kôd koji stvara frejm se nalazi na početku potprograma:

```
pushl %ebp            #čuvanje zatečenog FP-a
movl %esp, %ebp       #postavljanje novog FP-a
subl $4, %esp         #prostor za lokalnu promenljivu
```

Na kraju potprograma se nalazi asemblerski kôd koji uništava frejm:

```
movl %ebp, %esp       #brisanje lokalne promenljive
popl %ebp             #vraćanje prethodnog FP-a
ret
```

...

`12(%ebp)`	Drugi parametar funkcije
`8(%ebp)`	Prvi parametar funkcije
`4(%ebp)`	Povratna adresa (stari `eip`)
`(%ebp)`	Stari `ebp`
`-4(%ebp)`	Prva lokalna promenljiva
`-8(%ebp)`	Druga lokalna promenljiva

...

Tabela 3: Pristup parametrima i lokalnim promenljivama

Argumentima na steku se u okviru potprograma pristupa korišćenjem `ebp` registra i baznog adresiranja (tabela 3). Pošto `ebp` registar pokazuje na fiksnu tačku frejma, u odnosu na njega se može određivati relativna pozicija (*offset*) argumenata i lokalnih promenljivih:

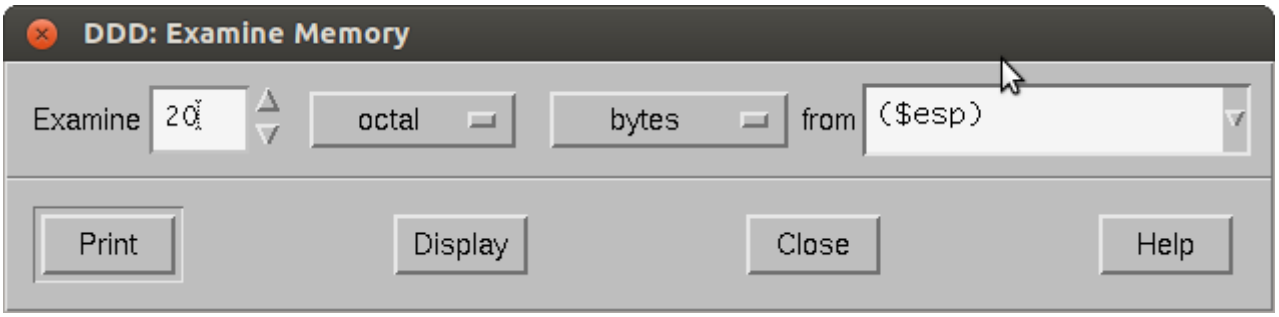
```
movl 12(%ebp), %ebx   #drugi parametar (b) -> ebx
movl -4(%ebp), %ecx   #prva lokalna promenljiva (r) -> ecx
```

Pošto pozivajuća funkcija stavlja argumente na stek, ona ih mora i skloniti sa steka. To se postiže izmenom sadržaja `esp` registra. Poziv funkcije sa slike 11 izveo bi se na sledeći način (pretpostavlja se da su argumenti `a`, `b` i `*rez` redom u registrima `eax`, `ebx` i `ecx`):

```
pushl %ecx        #treći argument -> stek
pushl %ebx        #drugi argument -> stek
pushl %eax        #prvi argiment -> stek
call nzd
addl $12, %esp    #oslobađanje prostora na steku
```

Svih 5 gornjih naredbi (smeštanje argumenata na stek, sâm poziv asemblerskog potprogama i oslobađanje zauzetog prostora na steku) predstavljaju ekvivalent poziva potprograma u višem programskom jeziku.

Dibager *DDD* može prikazati trenutni izgled steka korišćenjem opcije *Memory* iz menija *Data*. Opcije u prozoru koji se otvori treba podesiti na sledeće vrednosti:



Umesto broja 20 može se staviti broj vrednosti na steku koje se žele posmatrati, dok se umesto *octal* može staviti bilo koji drugi podržani brojni sistem.

6.3. Stek frejm i čuvanje registara

U situaciji kada je potrebno koristiti i registre koji po *cdecl* specifikaciji nisu slobodni za korišćenje (`ebx`, `esi` i `edi`), vrednosti tih registara se pre korišćenja moraju negde sačuvati, a pre izlaska iz potprograma ih treba restaurirati. Logično mesto za čuvanje njihovih vrednosti je stek. Na primer, ukoliko se u potprogramu koji ima tri lokalne promenljive, pored `eax`, `ecx` i `edx`, koriste i `ebx` i `esi`, tada bi početak potprograma trebao da izgleda ovako:

```
pushl %ebp            #čuvanje zatečenog FP-a
movl %esp, %ebp       #postavljanje novog FP-a
subl $12, %esp        #prostor za tri lokalne promenljive
pushl %ebx            #čuvanje sadržaja registra ebx
pushl %esi            #čuvanje sadržaja registra esi
```

Pre izlaska iz potprograma treba vratiti vrednosti sačuvanih registara u obrnutom redosledu:

```
popl %esi             #restauriranje sadržaja registra esi
popl %ebx             #restauriranje sadržaja registra ebx
movl %ebp, %esp       #brisanje lokalnih promenljivih
popl %ebp             #vraćanje prethodnog FP-a
ret
```

6.4. Definisanje potprograma na gcc asembleru

Definisanje potprograma ne zahteva nikakve dodatne direktive. Dovoljno je da se na ulaznu tačku potprograma postavi ulazna labela, i da se negde u telu potprograma nalazi naredba za povratak iz potprograma. Potprogrami se mogu definisati kako pre, tako i posle glavnog programa. Ukoliko je potrebno da se potprogram poziva iz *C* kôda, potprograme je potrebno deklarisati u skladu sa pravilima *C* jezika, kako bi *C* prevodioc znao broj i tipove parametara potprograma, kao i tip povratne vrednosti.

Imajući u vidu *C* pozivnu konvenciju, kada se piše asemblerski potprogram, najčešće ćemo uz njega davati i *C* deklaraciju funkcije, odnosno opis pozivanja tog potprograma iz *C*-a. Na primer, ako je potrebno napisati asemblerski potprogram za sabiranje dva broja, kod koga su prva dva parametra brojevi koji se sabiraju, dok je treći adresa rezultata, *C* deklaracija takvog potprograma bi izgledala:

```
void saberi (int a, int b, int *rez);
```

Iz ove deklaracije se može zaključiti više činjenica:

- potprogram treba da se zove `saberi`
- potprogram ne vraća vrednost (što znači da je sadržaj `eax` registra nebitan po izlasku iz potprograma)
- potprogram ima 3 parametra, prva dva se prenose po vrednosti, dok se treći prenosi po adresi
- nakon inicijalizacije stek frejma, prvi parametar će se nalaziti na `ebp+8`, drugi na `ebp+12`, a treći na `ebp+16`
- da bi se pristupilo vrednosti parametra prenetog po adresi, njegova adresa se prvo mora smestiti u neki registar, pa se treba iskoristiti neka od varijanti indirektnog adresiranja

6.5. Prenos argumenata po vrednosti i po adresi

Kada se potprogramu prosledi vrednost argumenta, ukoliko ona bude izmenjena u okviru potprograma, ta izmena se neće odraziti na originalni argument, pošto potprogram rukuje **lokalnom kopijom** vrednosti argumenta. Ovo odgovara ulaznim parametrima (prenošenju po vrednosti) kod viših programskih jezika.

Da bi se iz potprograma mogla menjati vrednost nekog argumenta, potprogramu treba proslediti njegovu **adresu**. Ovo odgovara ulazno-izlaznim parametrima (prenošenje po adresi ili referenci) kod viših programskih jezika. U primeru `nzd` (Slika 11) parametar `*rez` predstavlja adresu rezultata. Da bi se pristupilo vrednosti na koju `rez` pokazuje, treba iskoristiti neku varijantu indirektnog adresiranja, na primer:

```
movl 16(%ebp), %esi   #treći parametar (adresa rezultata) -> esi
movl $0, (%esi)       #postavljanje rezultata na 0
```

U *C* funkcijama je sasvim uobičajena situacija da se argument prenet po vrednosti modifikuje u toku izvršavanja. Na primer, ukoliko imamo:

```
void funkcija(int a, int b) {
     ...
     a += 4;
     if (a > 20)
     ...
}
```

to se u asembleru može odraditi tako što će se vrednost argumenta smestiti u neki registar, čija će se vrednost menjati u toku rada potprograma:

```
movl 8(%ebp), %ecx
...
addl $4, %ecx
cmpl $20, %ecx
...
```

Međutim, u situaciji kada nema dovoljno registara, vrednost argumenta se može koristiti i direktno, baš kao i na *C*-u:

```
...
addl $4, 8(%ebp)
cmpl $20, 8(%ebp)
...
```

Jedino ograničenje o kome treba voditi računa je, naravno, ograničenje *x86* procesora o maksimalno jednom pristupu memoriji u jednoj naredbi (pristup steku je takođe pristup memoriji).

***Primer 1** - Potprogram za sabiranje dva broja*

Ulaz u potprogram su dve vrednosti prenete preko steka, a izlaz je preko `eax` registra. Potprogram ne vrši proveru prekoračenja.

```
#potprogram za sabiranje dva broja
#int saberi(int a, int b);
a:   .long 123
b:   .long 456
saberi:                  #naziv potprograma
     pushl %ebp          #početak potprograma
     movl %esp, %ebp
     movl 8(%ebp), %eax  #telo potprograma
     addl 12(%ebp), %eax
     movl %ebp, %esp     #završetak potprograma
     popl %ebp
     ret
main:                    #glavni program
     pushl b             #drugi argument -> stek
     pushl a             #prvi argiment -> stek
     call saberi         #poziv potprograma
     addl $8, %esp       #oslobađanje prostora na steku
kraj:
     movl $1, %eax
     movl $0, %ebx
     int $0x80
```

***Primer 2** - Modifikacija prethodnog primera*

Ulaz u potprogram su dve vrednosti prenete preko steka, a rezultat se upisuje na adresu koja je preneta kao treći argument.

```
#potprogram za sabiranje dva broja
#void saberi(int a, int b, int *rez);
rezultat:    .long 0
...
saberi:                  #naziv potprograma
     pushl %ebp          #početak potprograma
     movl %esp, %ebp
     movl 8(%ebp), %eax  #telo potprograma
     addl 12(%ebp), %eax
     movl 16(%ebp), %ecx #adresa rezultata u %ecx
     movl %eax, (%ecx)   #upisivanje zbira na zadatu adresu
     movl %ebp, %esp     #završetak potprograma
     popl %ebp
     ret
main:                    #glavni program
     pushl $rezultat     #adresa rezultata -> stek
     pushl $123          #drugi argument -> stek
     pushl $456          #prvi argiment -> stek
     call saberi         #poziv potprograma
     addl $12, %esp      #oslobađanje prostora na steku
kraj:
     movl $1, %eax
     movl $0, %ebx
     int $0x80
```

6.6. Upotreba *gcc*-a za prevođenje i povezivanje (linkovanje) asemblerskih (pot)programa

Iako je moguće da se nađu u istoj izvornoj datoteci sa glavnim programom, potprogrami se obično nalaze u zasebnim izvornim datotekama i prevode (kompajliraju) se zasebno. Nakon prevođenja se povezuju (linkuju) sa glavnim programom kako bi se dobio izvršni oblik. Ovaj postupak omogućava da se jednom napisan i preveden kod upotrebljava u više različitih programa. Potprogrami slične namene se obično arhiviraju u kolekcije potprograma koje se nazivaju biblioteke. Ukoliko se poštuju pozivne konvencije jezika *C* prilikom pisanja potprograma, potprogram napisan na asembleru može da se poveže sa *C* programom.

Ukoliko se program sastoji iz, na primer, glavnog programa u jednom i nekih potprograma u drugom ili drugim fajlovima, izvršna verzija se može dobiti jednostavnim navođenjem naziva svih fajlova od kojih ona treba da se izgeneriše:

```
gcc -g -o program program.S potprogram.S pomocni.S
```

U ovom slučaju, izvršna verzija potprograma se dobija prevođenjem i povezivanjem kôda koji se nalazi u 3 fajla (`program.S`, `potprogram.S` i `pomocni.S`). Treba naglasiti da se u **samo jednom** od ovih fajlova sme naći direktiva `.globl main`, dok ostali treba da iza `.globl` direktive imaju labele koje treba da budu vidljive spolja, obično naziv potprograma. Na primer, ako je u nekom fajlu definisan potprogram pod nazivom `saberi` (samo taj potprogram), tada u tom fajlu ova direktiva treba da glasi:

```
.globl saberi
```

Ukoliko je asemblerski program ili potprogram potrebno prevesti bez povezivanja, kako bi se dobio tzv. objektni fajl, upotrebljava se opcija `-c`:

```
gcc -c program.S -o program.o
```

Asemblerski program se može povezati sa nekim već prevedenim kôdom, koji se u tom slučaju nalazi u objektnom fajlu. U tom slučaju je dovoljno samo navesti sve fajlove od kojih treba da se izgeneriše izvršna verzija programa:

```
gcc -g -o program program.S potprogram.o pomocni.S
```

U ovom primeru, fajl `potprogram.S` je unapred preveden i vrši se povezivanje već prevedenog potprograma sa ostatkom kôda. Ukoliko je `potprogram.S` preveden sa opcijom `-g`, tada će se moći vršiti dibagiranje i njegovog kôda.

Ukoliko je prevedeni program nastao povezivanjem više potprograma, postavlja se pitanje kako dibagirati kôd određenog potprograma, bez ulaženja u npr. glavni program. U tom slučaju, nakon pokretanja *DDD* dibagera, dovoljno je otvoriti odgovarajući fajl sa izvornim kôdom korišćenjem opcije *File/Open Source*.

Primer povezivanja asemblerskog potprograma i asemblerskog glavnog programa

Pretpostavimo da imamo dva fajla na disku, jedan koji sadrži potprogram za sabiranje dva broja i koji se zove `sabpp.S`:

```
#potprogram za sabiranje dva broja
#int saberi(int a, int b);
.section .text
.globl saberi
saberi:                    #naziv potprograma
    pushl %ebp             #početak potprograma
    movl %esp, %ebp
    movl 8(%ebp), %eax     #telo potprograma
    addl 12(%ebp), %eax
    movl %ebp, %esp        #završetak potprograma
    popl %ebp
    ret
```

i drugi koji u sebi sadrži glavni asemblerski program i koji se zove `glavni.S`:

```
#glavni program za sabiranje dva broja
#int saberi(int a, int b);
.section .data
a:   .long 123
b:   .long 456

.section .text
.globl main
main:                      #glavni program
    pushl b                #drugi argument -> stek
    pushl a                #prvi argiment -> stek
    call saberi            #poziv potprograma
    addl $8, %esp          #oslobađanje prostora na steku
kraj:
    movl $1, %eax
    movl $0, %ebx
    int $0x80
```

Ova dva fajla se mogu spojiti u jedan izvršni fajl sledećim pozivom `gcc`-a:

```
gcc –g –o saberi sabpp.S glavni.S
```

Kao rezultat će se dobiti izvršni fajl `saberi` koji će u sebi imati potprogram iz prvog fajla i glavni program iz drugog.

Primer povezivanja asemblerskog potprograma i C glavnog programa

Pretpostavimo da imamo isti potprogram kao u prethodnom primeru (`sabpp.S`), ali da ovaj put imamo glavni C program u fajlu `glavni.c`:

```
//glavni program za sabiranje dva broja
#include <stdio.h>

#deklaracija asemblerskog potprograma
int saberi(int a, int b);

int main()
{
        int rez;
        rez = saberi(8, 12);
        printf("Rezultat za 8 i 12 je %d\n", rez);
}
```

Pri početku *C* programa se nalazi deklaracija asemblerske funkcije (potprograma). Ova linija je neophodna kako bi se asemblerski potprogram na ispravan način pozivao iz *C* programa (tačnije, postoje situacije kada ovo nije neophodno uraditi, ali je mnogo bolja praksa da se deklaracija uvek napiše).

Slično kao u prethodnom primeru, ova dva fajla se mogu spojiti u jedan izvršni fajl sledećim pozivom `gcc`-a:

```
gcc –g –o saberi sabpp.S glavni.c
```

Kao rezultat će se dobiti izvršni fajl `saberi` koji će u sebi imati asemblerski potprogram iz prvog fajla i glavni C program iz drugog. Treba napomenuti da je ovo povezivanje moguće zato što je asemblerski potprogram napisan po **cdecl** deklaraciji za pozivanje potprograma.

Primer povezivanja asemblerskog potprograma i glavnog programa u objektnom fajlu

Pretpostavimo da imamo isti potprogram kao u prethodnim primerima (`sabpp.S`), ali da smo glavni asemblerski program iskompajlirali kao objektni fajl, na primer:

```
gcc –c glavni.S –o glavni.o
```

ili, alternativno, da smo glavni C program iskompajlirali kao objektni fajl:

```
gcc –c glavni.c –o glavni.o
```

Slično kao u prethodnim primerima, ova dva fajla se mogu spojiti u jedan izvršni fajl sledećim pozivom `gcc`-a:

```
gcc –g –o saberi sabpp.S glavni.o
```

Kao rezultat će se dobiti izvršni fajl `saberi` koji će u sebi imati asemblerski potprogram iz prvog fajla i glavni program iz drugog.

6.7. Rekurzija

Rekurzija predstavlja situaciju u kojoj potprogram poziva samog sebe (direktno, ili indirektno, preko drugog potprograma). Najlakše se može realizovati korišćenjem frejmova, odnosno steka, pošto u tom slučaju nije potreban nikakav dodatni kôd. Kod ovakvih potprograma treba biti pažljiv i treba dobro definisati uslov izlaska iz rekurzije, da se ne bi desilo da se stek prepuni (*stack overflow*).

Jedan primer rekurzivnog potprograma bi bila funkcija za izračunavanje faktorijela. Faktorijel nekog broja *n* (nije definisan za negativne brojeve) je po definiciji:

n!=n*(n-1)!

0!=1

Ako se usvoji da funkcija ima jedan parametar (*n*) tipa 32-bitnog neoznačenog broja koji se prenosi po vrednosti, a da se povratna vrednost vraća preko registra `eax`, program bi mogao izgledati ovako:

```
#int faktorijel(int n);
faktorijel:
    pushl %ebp
    movl %esp, %ebp
    movl 8(%ebp), %ecx
    andl %ecx, %ecx       #provera da li je n=0
    jz fakt_nula
    decl %ecx             #n-1
    pushl %ecx            #rekurzija
    call faktorijel
    addl $4, %esp
    mull 8(%ebp)          #n*(n-1)! -> eax
    jmp fakt_kraj
fakt_nula:
    movl $1, %eax         #slučaj n=0
fakt_kraj:
    movl %ebp, %esp
    popl %ebp
    ret
```

6.8. Domaći zadaci

1. Napraviti potprogram `int uvecaj(int x)` koji na argument dodaje 1 i vraća dobijeni broj.
2. Napraviti potprogram `int palindrom(char *str)` koji proverava da li je string palindrom. Ako jeste, vratiti 1, u suprotnom vratiti 0.
3. Prepraviti potprogram za faktorijel tako da se rezultat vraća preko argumenta prenetog po adresi. Registar `eax` iskoristiti za vraćanje vrednosti o grešci: ukoliko dođe do prekoračenja opsega od 32 bita prilikom množenja, u `eax` vratiti 1, inače vratiti 0.

7. Rukovanje bitima

7.1. Osnovne logičke naredbe

Za rukovanje bitima su zadužene logičke naredbe procesora *80386*. **Sve** logičke naredbe utiču na indikatore. Osnovne logičke naredbe su `and` (logičko I), `or` (logičko ILI), `xor` (ekskluzivno ILI) i `not` (logičko NE). Prve tri obavljaju logičku operaciju nad parovima korespondentnih bita izvornog i odredišnog operanda. Naredba `not` ima samo jedan operand i obavlja operaciju nad njegovim bitima.

Ove naredbe se često koriste i za tzv. **maskiranje**, odnosno proveravanje ili postavljanje pojedinih bita. Na primer, ako treba najniži bit registra `eax` postaviti na 0, to se može postići naredbom:

```
andl $0xfffffffe, %eax
```

Broj `0xfffffffe` se u ovom slučaju naziva **maska**, pošto služi za izdvajanje (maskiranje) najnižeg bita. Binarno, taj broj ima sve jedninice, osim na najmanje značajnoj poziciji. Na sličan način se naredba `or` može iskoristiti za postavljanje nekog bita na vrednost 1, dok se za promenu vrednosti nekog bita može iskoristiti naredba `xor`:

```
orlw $1, %ax      #postavljanje najnižeg bita registra ax na 1
xorb $1, %cl      #promena vrednosti najnižeg bita registra cl
```

Provera da li neki bit ima vrednost 0 ili 1 se može uraditi izdvajanjem tog bita pomoću naredbe `and` i provere dobijenog rezultata. Na primer, ako treba proveriti da li je najniži bit registra `eax` 0 ili 1, može se pisati:

```
andl $1, %eax
```

Nakon ovoga, ako se u registru `eax` nalazi 0, tada je vrednost najnižeg bita bila 0, a ako se u registru `eax` nalazi vrednost različita od 0, tada je vrednost najnižeg bita bila 1. Kod ove provere je nezgodno to što se vrednost odredišnog operanda menja. Upravo za ovakve provere postoji naredba `test`. Radi isto što i `and`, ali ne smešta rezultat u odredišni operand, nego samo postavlja indikatore u zavisnosti od dobijenog rezultata.

```
testw $0x8000, %cx #provera znaka označene vrednosti u registru cx
```

Komplement 2 nekog broja se može dobiti naredbom `neg`, koja predstavlja kombinaciju naredbe `not` i dodavanja jedinice rezultatu.

Negacija u dvostrukoj preciznosti

Negirati neku vrednost u ovom slučaju znači izračunati njen komplement 2. Za negiranje u jednostrukoj preciznosti 80386 ima naredbu `neg`, ali za slučaj dvostruke preciznosti bolje je postupiti po definiciji: komplement 2 neke vrednosti je njen komplement 1 uvećan za jedan:

```
notl %eax
notl %edx
addl $1, %eax
adcl $0, %edx
```

Navedeni program negira vrednost u registarskom paru `edx:eax`. Naredba `not` računa komplement 1 odredišnog operanda. U ovom slučaju je svejedno kojim će se redom komplementirati niža i viša reč 64-bitne vrednosti.

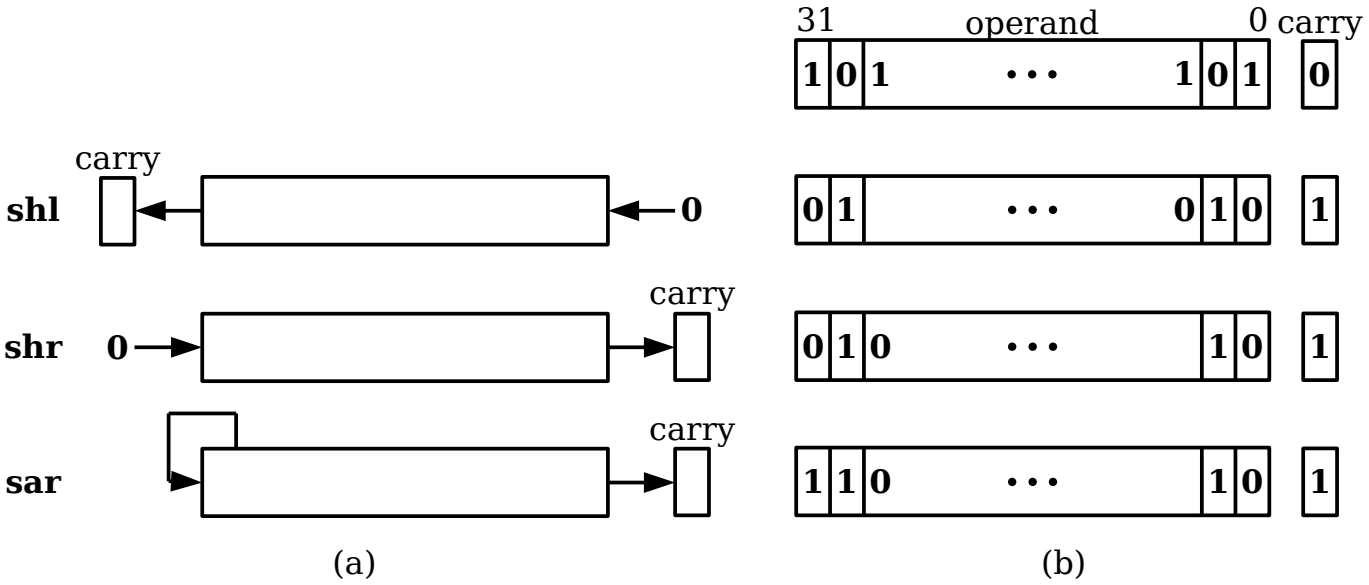
Korišćenjem naredbe `neg`, program bi se mogao napisati na sledeći način:

```
negl %edx
negl %eax
sbbl $0, %edx
```

Vrednost više reči se smanjuje za jedan ako se prilikom negiranja `eax` postavi indikator `c` (tj. ako `eax` promeni znak), što će se desiti za sve vrednosti `eax` osim nule. Ova varijanta jeste kraća, ali ima tu manu što ne postavlja indikator `o` u slučaju izlaska van opsega (do koga dolazi za vrednost 2^{63}).

7.2. Naredbe za pomeranje i rotiranje

Pomeranje neke vrednosti za *n* bita u levo ekvivalentno je množenju te vrednosti sa 2^n, dok je pomeranje u desno ekvivalentno celobrojnom deljenju sa 2^n. Pomeranje može biti logičko ili aritmetičko. Kod prvog, bit najmanje ili najveće težine (u zavisnosti od toga da li se pomera u levo ili desno) dobija vrednost nula. Kod drugog, pomeranje u levo je istovetno logičkom, a pomeranje u desno zadržava vrednost bita najveće težine. Ovime se čuva znak u slučaju rada sa označenim brojevima. Osnovne naredbe za pomeranje su `shl`, `shr` i `sar`.



Slika 12: Naredbe za pomeranje: princip rada (a) i primer pomeranja za 1 mesto (b)

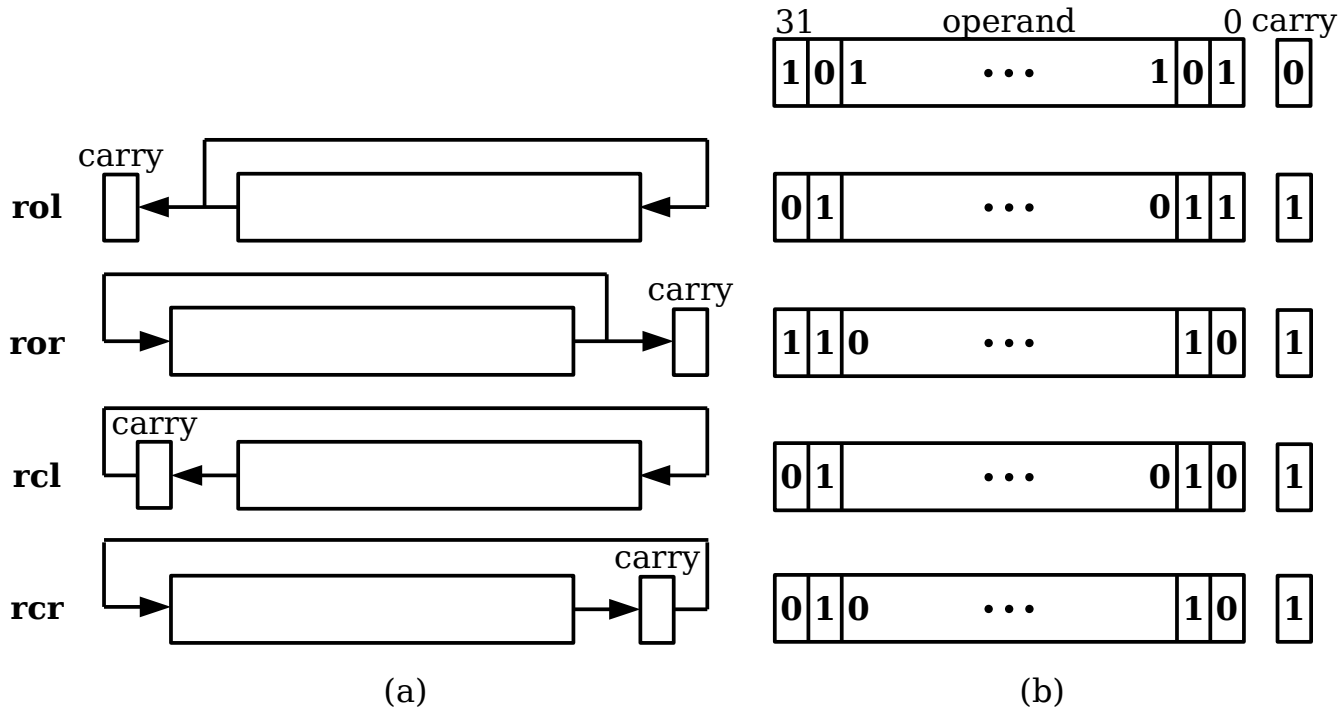
Na slici 12 prikazano je dejstvo *80386* naredbi za pomeranje. Odredišni operand može biti registar ili memorijska lokacija. Naredbe za pomeranje prihvataju još jedan operand, koji određuje broj pozicija za koje će pomeranje biti obavljeno. Ovaj operand je ili neposredni, ili se uzima iz registra `cl`. U slučaju pomeranja za više pozicija, indikator `c` će biti postavljen prema vrednosti poslednjeg istisnutog bita. Primeri:

```
shll $1, %eax     #pomeranje eax za 1 mesto u levo
shrw %cl, %bx     #pomeranje bx za cl mesta u desno
```

Prilikom pomeranja u levo, postavljen indikator `c` se može uzeti kao indikator izlaska van opsega odredišta za slučaj rada sa neoznačenim vrednostima. Ako se vrednosti tretiraju kao označene, treba proveravati stanje indikatora `o`.

Pomeranjem u desno ne može se prekoračiti kapacitet odredišta. Ovde postoje dva granična slučaja: logičko pomeranje neoznačene vrednosti i aritmetičko pomeranje pozitivne označene vrednosti će posle najviše *n* koraka (gde je *n* veličina odredišta u bitima) svesti vrednost na nulu, dok će kod aritmetičkog pomeranja negativne označene vrednosti krajnja vrednost biti -1.

Naredbe za rotiranje su prikazane na slici 13. `rol` vrši rotiranje u levo, a `ror` rotiranje u desno. Rotiranjem u levo se bit sa najviše pozicije u odredištu dovodi na najnižu, dok se svi ostali biti pomeraju za jednu poziciju u levo. Analogno tome se radi i rotiranje u desno. Sve što je rečeno za operande naredbi za pomeranje važi i ovde.



Slika 13: Naredbe za rotiranje: princip rada (a) i primer rotiranja za 1 mesto (b)

Uz prethodno navedene, postoje i dve naredbe za rotiranje kroz indikator `c`, `rcl` i `rcr`. Razlika u odnosu na `rol` i `ror` je u tome što se na najnižu ili najvišu poziciju dovodi bit iz indikatora `c`. Primeri:

```
rolw $1, %ax      #rotiranje ax za jedno mesto u levo
rcrl %cl, %ebx    #rotiranje ebx kroz c za cl mesta u desno
```

Pomeranje i rotiranje u dvostrukoj preciznosti

U tabeli 4 prikazani su programi koji pomeraju ili rotiraju registarski par `edx:eax`. Nisu prikazani jedino ekvivalenti operacija `rcl` i `rcr`.

Operacija	***Program***
`shl`	`shll $1, %eax` `rcll $1, %edx`
`shr`	`shrl $1, %edx` `rcrl $1, %eax`
`sar`	`sarl $1, %edx` `rcrl $1, %eax`

`rol`	`shll $1, %eax`
	`rcll $1, %edx`
	`rcrl $1, %eax`
	`roll $1, %eax`

`ror`	`shrl $1, %edx`
	`rcrl $1, %eax`
	`rcll $1, %edx`
	`rorl $1, %edx`

Tabela 4: Pomeranje i rotiranje u dvostrukoj preciznosti

Operacije rotiranja se mogu izvesti i drugačije. Na primer, program za rotiranje u levo mogao bi da glasi:

```
        shll $1, %eax
        rcll $1, %edx
        jnc nije_jedan
        orl $1, %eax
nije_jedan:
        ...
```

Ovime se eksplicitno testira stanje indikatora **c** i postavlja najniži bit `eax` ako je `c` postavljen. Implementacija iz tabele 4 je namerno urađena tako da ne sadrži naredbe uslovnog skoka.

7.3. Množenje pomoću sabiranja i pomeranja

Nešto komplikovaniji algoritam za množenje pomoću sabiranja je množenje pomoću sabiranja i pomeranja. Predstavlja vernu kopiju algoritma koji se koristi kod ručnog množenja dva binarna broja. Ako imamo vrednosti *a* i *b*, veličine *n* bita, njihov proizvod je

$$(a_{n-1}2^{n-1}+a_{n-2}2^{n-2}+\cdots+a_0)\cdot(b_{n-1}2^{n-1}+b_{n-2}2^{n-2}+\cdots+b_0)$$

gde je a_i vrednost bita na poziciji *i* (analogno tome i b_i). Proizvod se može zapisati i ovako:

$$b_0\cdot a+b_1\cdot 2\,a+\cdots+b_{n-1}\cdot 2^{n-1}a$$

Iz zapisa se vidi da je rezultat moguće izračunati pomoću sukcesivnih sabiranja. U svakom koraku posle prvog vrednost *a* se udvostručuje. Udvostručavanje vrednosti se u asembleru može izvesti:

- Pomeranjem svih bita vrednosti za jednu poziciju u levo; najniži bit dobija vrednost nula.
- Sabiranjem vrednosti sa samom sobom, što će se koristiti u ovom primeru s obzirom da je sabiranje u dvostrukoj preciznosti već opisano.

Na slici 14 prikazan je program na *C*-u za množenje brojeva na opisani način.

```
a = 60;
b = 30;
r = 0;
if (a != 0)
     while (b != 0) {
          if ((b & 1) != 0)
               r = r + a;
          a = a + a;
          b = b >> 1;
}
```

Slika 14: C program za množenje pomoću sabiranja i pomeranja

Postupak započinje samo ukoliko su oba činioca različita od nule. Izraz `(b & 1)` imaće vrednost jedan ako je najniži bit promenljive `b` jedinica (operator `&` označava bit-operaciju logičkog I). Promenljiva `b` se u svakom koraku pomera za jedan bit u desno (`b = b >> 1`). Na ovaj način će isti izraz poslužiti za testiranje bita rastuće težine.

Algoritam se može prevesti u sledeći asemblerski program:

```
        movl $60, %ebx
        movl $30, %eax
        movl %eax, %ecx
        xorl %eax, %eax
        xorl %edx, %edx
        xorl %esi, %esi
        andl %ebx, %ebx
        jz kraj
    proveri:
        jecxz kraj
        testl $1,%ecx
        jz uvecaj
        addl %ebx, %eax
        adcl %esi, %edx
    uvecaj:
        addl %ebx, %ebx
        adcl %esi, %esi
        shrl $1, %ecx
        jmp proveri
    kraj:
        movl $1, %eax
        movl $0, %ebx
        int $0x80
```

Napomene uz primer:

- Program prihvata 32-bitne neoznačene argumente u registrima `ebx` i `eax`, a rezultat je neoznačen 64-bitni broj u registarskom paru `edx:eax`. Do izlaska van opsega u rezultatu ne može doći, pa se taj slučaj i ne proverava.
- Vrednost koja se u toku algoritma udvostručuje je registarski par `esi:ebx`. Parametar `eax` se odmah prebacuje u registar `ecx` iz dva razloga: prvo, `eax` treba da sadrži nižu reč rezultata, i drugo, usled neortogonalnosti skupa naredbi *80386*, za `ecx` je vezana specijalna naredba `jecxz`, uslovni skok čiji je uslov za grananje `ecx`=0 i koji se u programu može dobro iskoristiti. Ova naredba postoji i za registar `cx` i njen naziv je `jcxz`.
- Ekskluzivno ILI neke vrednosti sa samom sobom (`xor %eax,%eax`) daje rezultat 0, što se može iskoristiti za anuliranje sadržaja registara. Ovo je ekvivalentno sa `mov $0,%eax`, s tim što se prva naredba brže izvršava.
- Logičko I neke vrednosti sa samom sobom (`and %ebx,%ebx`) neće promeniti tu vrednost, ali će *80386* postaviti indikator `z` ukoliko je rezultat nula (što se može desiti samo ukoliko je i sama vrednost jednaka nuli).
- Naredba `jz` (*jump if zero*) je alternativni naziv za `je` (*jump if equal*), koja je pomenuta ranije.

7.4. Deljenje pomoću oduzimanja i pomeranja

Jedan od algoritama za deljenje, slično algoritmu za množenje, je zasnovan na načinu na koji se ručno dele dva broja:

```
 110101 / 101 = 1010 (53 / 5 = 10, ostatak 3)
-101↓↓↓   (101 u 110 ide 1 puta, sledi oduzimanje)
 ------
   1↓↓↓   (ostatak nakon oduzimanja je 1)
   11↓↓   (spušta se sledeća cifra, 101 u 11 ide 0 puta)
   110↓   (spušta se sledeća cifra)
  -101↓   (101 u 110 ide 1 puta, sledi oduzimanje)
   ----
     1↓   (ostatak nakon oduzimanja je 1)
     11   (spušta se poslednja cifra, 101 u 11 ide 0 puta)
```

Ukoliko bi se izrazio C kodom, algoritam bi mogao da izgleda ovako:

```
unsigned int kol,ost,a,b,maska;
kol = 0;
ost = 0;
a = 53;
b = 5;
maska = 0x80000000;   //početna maska
while (maska) {       //dok se ne obrade svi biti
     ost <<= 1;
     if (maska & a)   //kopiranje bita iz a
          ost |= 1;   //u ostatak
     if (ost >= b) {  //ako se b sadrži u tekućem ost
          ost -= b;   //oduzmi b od tekućeg ostatka
          kol |= maska;//postavi 1 u količnik
     }
     maska >>= 1;
}
```

Slika 15: C program za deljenje pomoću oduzimanja i pomeranja

Za prolazak kroz bite broja `a` se koristi maska, a kada maska postane nula, obrađeni su svi biti. U svakom koraku petlje, tekućem ostatku se dodaje bit iz broja `a`, i ako je ostatak veći ili jednak broju `b`, vrši se oduzimanje i zapisvanje jedinice u količnik (na istoj poziciji na kojoj je tekući bit maske).

Algoritam se može prevesti u sledeći asemblerski program:

```
    movl $53, %esi          # a
    movl $5, %edi           # b
    movl $0, %eax           # količnik
    movl $0, %edx           # ostatak
    movl $0x80000000, %ecx  # maska
petlja:
    shll $1, %edx     # ost <<= 1
    testl %ecx, %esi  # kopiranje bita iz a
    jz oduzmi
    orl $1, %edx      # u ostatak
oduzmi:
    cmpl %edi, %edx   # da li je ostatak veći od b?
    jl kraj
    subl %edi, %edx   # odizmi b od ostatka
    orl %ecx, %eax    # upiši 1 u količnik
kraj:
    shrl $1,%ecx      # maska >>= 1
    jnc petlja        # ako je carry=1, obrađeni su svi biti
    movl %eax, kol
    movl %edx, ost

    movl $1, %eax
    movl $0, %ebx
    int $0x80
```

7.5. Domaći zadatak

1. Napisati potprogram koji kao argumente prima 3 vrednosti, *a*, *b* i *n* (0<=*n*<=32), a kao rezultat daje vrednost čijih *n* najnižih bita odgovara vrednosti *a*, a preostali biti odgovaraju vrednosti *b*:
   `int Izmesaj(int a, int b, int n)`

8. Kontrolna suma i paritet

Kada se podaci prebacuju sa jednog računara na drugi, postavlja se pitanje da li su oni ispravno preneti. Uobičajeni način za proveru je da se na podatke primeni neki algoritam koji će kao rezultat dati jedan ili više kontrolnih brojeva koji se prebacuju zajedno sa podacima. Kada se podaci prebace na drugi računar, na njih se primeni isti algoritam, pa se uporede tako dobijeni kontrolni brojevi sa prenetim kontrolnim brojevima. Ukoliko postoje razlike, došlo je do greške u prenosu.

Kontrolni brojevi se mogu računati za sve podatke zajedno (*checksum* - **kontrolna suma**), a mogu se računati i za svaki posebno. Postoji mnogo algoritama koji se koriste u ove svrhe, od kojih jednostavniji omogućavaju samo detekciju grešaka, dok komplikovaniji omogućavaju i ispravak grešaka u slučaju da ih nema previše (ovo zavisi od količine redundantnih podataka, odnosno kontrolnih brojeva).

Jedan algoritam koji radi sa svakim podatkom posebno se zasniva na sledećem: posmatra se binarna predstava svakog podatka posebno i prebrojava se koliko jedinica ima u broju (slika 16).

```
                    paritet
1.    00101001          1
2.    01010101          0
3.    01010100          1
4.    01100101          0
5.    00011111          1
```

Slika 16: Paritet

Ukoliko je broj jedinica paran, kontrolna suma (u ovom slučaju, kontrolni bit) se postavlja na 0, a u suprotnom se postavlja na 1. Ovakav kontrolni bit se naziva **paritet** (*parity*), a ponekad i **horizontalni paritet**. Što se ovog kursa tiče, termin **parnost** će označavati osobinu broja da je paran ili neparan (deljiv ili nedeljiv sa 2), dok će termin **paritet** predstavljati rezultat prebrojavanja bita u broju.

Kada se koristi paritet, uobičajeno je da se bit na najznačajnijoj poziciji koristi za smeštanje pariteta (u tom slučaju, podaci se moraju predstavljati sa jednim bitom manje). Ukoliko bi se bit pariteta ugradio u vrednosti sa slike 16, dobile bi se vrednosti kao na slici 17:

```
1.    10101001
2.    01010101
3.    11010100
4.    01100101
5.    10011111
```

Slika 17: Paritet na najznačajnijem bitu

Nedostatak ovakve kontrole podataka je što ne može otkriti grešku ukoliko je u broju promenjen paran broj bita.

Druga vrsta kontrolnih suma su one koje se odnose na podatke kao celinu. Jedan algoritam koji radi na taj način se zasniva na sledećem: posmatra se binarna predstava svih podataka zajedno i broji se koliko ima jedinica na svakoj težinskoj poziciji u svim brojevima (slika 18).

```
1.    00101001
2.    01010101
3.    01010100
4.    01100101
5.    00011111
Σ     01010010
```

Slika 18: Vertikalni paritet

Ukoliko je broj jedinica paran, na odgovarajućem mestu u kontrolnoj sumi se stavlja 0, a u suprotnom se stavlja 1. Ovakva kontrolna suma se naziva i **vertikalni paritet**. Očigledan nedostatak ovakve kontrolne sume je isti kao i kod pariteta, a to je da ne može detektovati grešku ako se na jednoj težinskoj poziciji javi paran broj grešaka.

Asemblerski program koji računa kontrolnu sumu za niz 32-bitnih brojeva je dat u nastavku teksta.

```
br_elem = 6
niz:.long 0b10101010100101010001010100111111
    .long 0b10101010110111111010101010001010
    .long 0b11111111110000000001111111000000
    .long 0b11101001010110100101101010101010
    .long 0b00010101010100101010101010100101
    .long 0b11000101001010001001000100101010
checksum:    .long 0
    xorl %eax, %eax           #checksum
    movl $1, %edx             #maska
sledeci_bit:
    xorl %ecx, %ecx           #brojač bita
    xorl %esi, %esi           #brojač elemenata niza
sledeci_el:
    movl niz(,%esi,4), %ebx
    andl %edx, %ebx           #provera vrednosti bita
    jz bit_nula
    incl %ecx
bit_nula:
    incl %esi
    cmpl $br_elem, %esi       #obrađeni svi elementi?
    jl sledeci_el
    shrl $1, %ecx             #parnost
    rcrl $1, %eax
    shll $1, %edx             #pomeranje maske
    jnc sledeci_bit           #obrađeni svi biti?
    movl %eax, checksum
kraj:
    movl $1, %eax
    movl $0, %ebx
    int $0x80
```

8.1. Domaći zadatak

1. Napisati program za proveru pariteta niza 8-bitnih brojeva (bit na najznačajnijoj poziciji je bit pariteta). Prvi podatak u nizu je broj elemenata niza (ne računajući taj podatak).

9. Konverzije brojeva iz internog oblika u znakovni oblik

Današnji računari interno (u registrima ili memorijskim lokacijama) predstavljaju brojeve u binarnom obliku. Brojevi su zapisani kao nizovi bita, čija dužina uglavnom odgovara kapacitetu memorijske lokacije. Za komunikaciju sa korisnikom se, međutim, broj iz interne predstave prevodi u znakovni niz čiji su elementi cifre. Takav znakovni zapis je promenljive dužine i može biti duži od internog. Na primer, za beleženje broja 31337_{10} u znakovnom obliku potrebno je najmanje pet bajtova (za svaku cifru po jedan), dok bi interni oblik u 16-bitnom registru zauzimao dva bajta.

Konverzija iz znakovnog oblika u interni oblik je neophodna prilikom zadavanja brojeva jer korisnik treba da ih zada u znakovnom obliku. Kada program izvrši obradu, rezultat treba prikazati u znakovnom obliku (koji je razumljiv korisniku), odnosno treba izvršiti konverziju iz internog oblika u znakovni oblik u nekoj osnovi.

U narednim primerima će se za znakovni zapis brojeva koristiti osnovni *ASCII* kôd, a svi brojevi će se tretirati kao neoznačeni. Znakovi za cifre dekadnog brojnog sistema imaju kodove od 48 do 57. Ako se radi sa brojnim osnovama od dva do deset, dobijanje vrednosti cifre na osnovu njenog *ASCII* koda se svodi na oduzimanje konstante 48 (koda za cifru nula) od *ASCII* koda cifre. Analogno tome se radi i obrnut postupak. Kod većih brojnih osnova kao cifre težine veće od devet koriste se mala ili velika slova, te je i postupak prilagođavanja složeniji.

9.1. Konverzija celih dekadnih brojeva iz internog oblika u znakovni oblik

Konverziju iz internog oblika u znakovni oblik celog broja određene brojne osnove najlakše je obaviti uzastopnim deljenjem ulazne vrednosti osnovom odredišnog brojnog sistema i pretvaranjem ostatka svakog deljenja u zankovni ekvivalent (tj. odgovarajuću cifru). Algoritam se završava kada se kao rezultat deljenja dobije nula. S obzirom da se na ovaj način dobija niz cifara zapisan obrnutim redosledom u odnosu na uobičajeni, elemente niza treba obrnuti da bi se dobio konačan i ispravan zapis.

U nastavku teksta je dat program za konverziju iz internog oblika u znakovni oblik celog dekadnog broja.

```
dec_br_max = 10
dec_br:  .fill dec_br_max,1,42
...
     movl $375000000, %eax     #broj za konverziju
     leal dec_br, %edi
     movl $10, %ebx            #baza
dec_cifra:
     xorl %edx, %edx           #neophodno zbog divl
     divl %ebx
     addb $'0', %dl
     movb %dl, (%edi)
     incl %edi
     andl %eax, %eax           #kraj algoritma?
     jnz dec_cifra
     movb $0, (%edi)           #kraj stringa
     decl %edi                 #obrtanje niza
     leal dec_br, %esi
```

```
obrni:
     cmpl %edi, %esi
     jae kraj
     movb (%esi), %ah
     movb (%edi), %al
     movb %al, (%esi)
     movb %ah, (%edi)
     decl %edi
     incl %esi
     jmp obrni
kraj:
     movl $1, %eax
     movl $0, %ebx
     int $0x80
```

9.2. Opšti slučaj konverzije razlomljenih brojeva iz internog oblika u znakovni oblik

Algoritam za konverziju iz internog oblika razlomljenog broja u znakovni oblik u bilo kojoj osnovi se može predstaviti sledećim nizom koraka:

1. Pomnoži broj sa bazom
2. Izdvoji ceo deo broja i zapiši ga kao cifru izlazne osnove
3. Ako je ceo deo broja različit od 0, oduzmi ga od broja
4. Ako je broj različit od 0, idi na korak 1, u suprotnom završi konverziju

Primer: konverzija vrednosti 0.3125_{10} u vrednost u oktalnom brojnom sistemu

$0.3125_{10} * 8 = 2.5 = 2 + 0.5$ => cifra izlazne osnove je 2, a u dalju obradu ide vrednost 0.5

$0.5_{10} * 8 = 4.0 = 4 + 0.0$ => cifra izlazne osnove je 4, postupak je završen, $0.3125_{10} = 0.24_8$

Ukoliko bi uzeli u obzir usvojenu internu predstavu brojeva (skaliranje sa 10^8), tada bi računica izgledala ovako:

$31250000 * 8 = 250000000 = 2*10^8 + 50000000$

$50000000 * 8 = 400000000 = 4*10^8 + 0$

Pri realizaciji ovog algoritma u asembleru, treba obratiti pažnju na to da se u ovom slučaju radi sa skaliranim brojevima, pa tome treba prilagoditi sve računske operacije. Takođe, pošto se može dobiti beskonačna periodičnost u izlaznom broju, neophodno je ograničiti broj njegovih cifara.

9.3. Rimski brojevi

Konverzija iz internog oblika u string koji predstavlja rimski broj se može lako odraditi korišćenjem nizova sa veličinama rimskih brojeva. U svakoj iteraciji spoljašnje petlje se poredi vrednost broja sa tekućom vrednošću rimske cifre, i ukoliko je cifra manja od broja, dodaje se u string, a broj se umanjuje za cifru. Pri ovome se prati koliko je mesta ostalo u izlaznom stringu.

```
#include <string.h>
int dec[] = {1000, 900, 500,400, 100, 90, 50,  40, 10,   9,  5,   4,  1};
char *rom[] =
{"M","CM","D","CD","C","XC","L","XL","X","IX","V","IV","I"};
int inter_to_roman(unsigned int broj, char *str, int duzina) {
    int i;
    for (i=0; i<13; i++)
        while (broj >= dec[i]) {
            if (strlen(rom[i]) > duzina-1)
                return 1;
            strcat(str,rom[i]);
            broj -= dec[i];
            duzina -= strlen(rom[i]);
        }
    return 0;
}
```

U asemblerskom rešenju za potprogram se mogu koristiti sledeći nizovi (smešteni u sekciju koda, te stoga read-only):

```
.section .text
.globl inter_to_roman
dec:    .long 1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4, 1
rom1:   .ascii "MCDCCXLXXIVII"
rom2:   .ascii " M D C L X V "
```

Niz `dec` sadrži vrednosti rimskih cifara, dok nizovi `rom1` i `rom2` sadrže prvi, odnosno drugi karakter rimske cifre. Ukoliko je drugi karakter razmak, ne treba se ubacivati u rezultujući string.

9.4. Domaći zadatak

1. Napisati program u asembleru za konverziju celog broja u dvostrukoj preciznosti iz internog oblika u znakovni dekadni oblik.

10. Konverzije brojeva iz znakovnog oblika u interni oblik

Prilikom konverzije iz znakovnog oblika u interni proverava se sledeće:

- Validnost pojedinačnih cifara, u skladu sa brojnom osnovom broja: cifra van dozvoljenog opsega predstavlja grešku.
- Kapacitet odredišta za interni oblik: prekoračenje kapaciteta je greška.
- Dužina izvornog niza cifara: prazan niz (onaj čiji je prvi element NUL bajt) se tretira kao greška.

Realni programi za konverziju imali bi veći broj provera. Na primer, uobičajeno je da se znakovi praznine (razmak i tabulator) pre prve cifre, kao i iza poslednje, ignorišu.

10.1. Konverzija celih dekadnih brojeva iz znakovnog oblika u interni oblik

Uopšte uzev, svaki broj od *n* cifara, zapisan u pozicionom brojnom sistemu sa osnovom *b*, može se predstaviti na sledeći način:

$$x_{n-1}b^{n-1}+x_{n-2}b^{n-2}+\cdots+x_1 b+x_0$$

gde je x_i cifra odgovarajuće težine. Pretpostavka je da se cifre pišu od najveće težine ka najmanjoj. Postupak konverzije ovako zapisanog broja u njegovu vrednost može se opisati sledećim nizom koraka:

1. Inicijalizuj rezultat konverzije na nulu i postavi tekuću poziciju u nizu cifara na prvu cifru
2. Pomnoži rezultat osnovom brojnog sistema
3. Dodaj na rezultat vrednost cifre na tekućoj poziciji
4. Ako ima još cifara, pomeri tekuću poziciju za jedno mesto u desno i pređi na drugi korak, u suprotnom završi konverziju

Program na *C*-u za konverziju celog neoznačenog broja iz dekadnog znakovnog oblika u interni oblik prikazan je na slici 19. Jednostavnosti radi, ovaj algoritam ne proverava izlazak van opsega rezultata.

```
#include <ctype.h>
char *broj = "82596"
char *d = broj;
int greska;
unsigned r;

greska = 0;
r = 0;
if (!*d)
    greska++;
else
    while (*d) {
        if (!isdigit(*d)) {
            greska++;
            break;
        }
        r *= 10;
        r += *d++ - '0';
    }
```

Slika 19: Konverzija znakovnog zapisa neoznačenog dekadnog broja u interni format

Asemblerska verzija ovog algoritma izgleda:

```
dec_br:  .ascii "3133734892\0"
greska:  .byte 0
...
     leal dec_br, %esi     #adresa prvog znaka -> esi
     movb $0, greska
     xorl %eax, %eax       #r=0
     xorl %ebx, %ebx
     movl $10, %ecx
dec_cifra:
     movb (%esi), %bl      #znak -> bl
     andb %bl, %bl         #provera kraja stringa
     jz kraj_dec
     subb $'0', %bl        #izdvajanje cifre i provere
     mull %ecx             #r*=10
     addl %ebx, %eax       #r=r+cifra
     incl %esi             #sledeći znak
     jmp dec_cifra
kraj_dec:
     cmpl $dec_br, %esi    #'\0' na početku stringa?
     jne kraj
i_greska:
     incb greska
kraj:
     movl $1, %eax
     movl $0, %ebx
     int $0x80
```

10.2. Konverzija razlomljenih dekadnih brojeva iz znakovnog oblika u interni oblik

Ako se usvoji skaliranje sa 10^9, tada se ulazni niz znakova može predstaviti kao skalirana vrednost. Zbog uvedenog načina predstavljanja razlomljenih brojeva, može se primetiti da je konverzija iz znakovnog dekadnog oblika razlomljenog broja u interni oblik identična prethodnom programu (konverzija celog broja). Jedini dodatni uslov je da rezultat ne sme biti veći ili jednak od 10^9.

```
dec_br:  .ascii "200000000\0"
greska:  .byte 0
...
     leal dec_br, %esi
     movb $0, greska
     xorl %eax, %eax
     xorl %ebx, %ebx
     movl $10, %ecx            #baza
dec_cifra:
     movb (%esi), %bl
     andb %bl, %bl             #kraj stringa?
     jz kraj_dec
     subb $'0', %bl            #dobijanje cifre
     jc i_greska
     cmpb $9, %bl
     ja i_greska
     mull %ecx
     andl %edx, %edx
     jnz i_greska
     addl %ebx, %eax
     jc i_greska
     cmpl $1000000000, %eax    #provera prekoračenja
     jae i_greska
     incl %esi                 #sledeći znak
     jmp dec_cifra
```

```
kraj_dec:
     cmpl $dec_br, %esi        #prazan string?
     jne kraj
i_greska:
     incb greska
kraj:
     movl $1, %eax
     movl $0, %ebx
     int $0x80
```

10.3. Opšti slučaj konverzije razlomljenih brojeva iz znakovnog oblika u interni oblik

Algoritam za konverziju razlomljenog broja u bilo kojoj osnovi u osnovu 10 (koja se, skalirana sa 10^9, koristila kao interna predstava u prethodnim primerima) se može predstaviti sledećim nizom koraka:

1. Postavi rezultat na 0 i pozicioniraj se na poslednju cifru u broju
2. Rezultat saberi sa cifrom i podeli sa bazom
3. Ukoliko ima još cifara, pređi na prethodnu cifru i idi na korak 2, u suprotnom završi konverziju

Primer: konverzija vrednosti 0.12_8 u vrednost u dekadnom brojnom sistemu

R = 0

R = (0 + 2) / 8 = 0.25_{10}

R = (0.25 + 1) / 8 = 0.15625_{10}

R = 0

R = (0 + 2*10^9) / 8 = 250000000

R = (250000000 + 1*10^9) / 8 = 156250000 = 0.15625 * 10^9

Kod implementacije ovog algoritma u asembleru, obratiti pažnju na to da se u tom slučaju radi sa skaliranim brojevima. Zbog toga što algoritam radi za bilo koju bazu, da bi se izbegla prekoračenja u nekim situacijama, za rad sa 32-bitnim vrednostima treba koristiti faktor skaliranja 10^8.

10.4. Rimski brojevi

Algoritam za konverziju iz stringa koji sadrži rimski broj u interni oblik se može definisati na sledeći način:

1. Indeks se postavi na prvi znak rimskog broja, a rezultat na nulu
2. Uporedi se vrednsot cifre na tekućoj i vrednost cifre na narednoj poziciji
   - Ako je vrednost tekuće cifre veća ili jednaka vrednosti naredne cifre, ta vrednost se dodaje na rezultat i prelazi se na sledeću cifru
   - Ako je vrednost tekuće cifre manja od vrednosti naredne cifre, na rezultat se dodaje vrednost naredne cifre umanjena za vrednost tekuće cifre i prelazi se na cifru iza naredne

Ovaj algoritam se može predstaviti sledećim C kodom:

```
#include <string.h>
int dec[] = {1000, 500, 100, 50, 10, 5, 1};
char *rom = "MDCLXVI";
```

```
unsigned int roman_to_inter(char* str, unsigned int* greska) {
    unsigned int r=0, i=0;
    char *c1,*c2;
    *greska = 0;
    while ((str[i]) && (!*greska)) {
        c1 = index(rom, str[i]);
        if (c1) {
            if (str[i+1]) {
                c2 = index(rom,str[i+1]);
                if (c2) {
                    if (c1<=c2)
                        r += dec[c1-rom];
                    else {
                        r += dec[c2-rom];
                        r -= dec[c1-rom];
                        i++;
                    }
                }
                else *greska = 1;
            }
            else r += dec[c1-rom];
        }
        else *greska = 1;
        i++;
    }
    return r;
}
```

U asemblerskom rešenju se mogu koristiti sledeći nizovi (smešteni u sekciju koda, te stoga read-only):

```
.section .text
.globl roman_to_inter
dec:    .long 1000, 500, 100, 50, 10, 5, 1
rom:   .ascii "MDCLXVI\0"
```

Niz `dec` sadrži vrednosti rimskih cifara, dok niz `rom` sadrži rimske cifre. Kod rešavanja zadatka smatrati da će redosled rimskih cifara biti korektan (odnosno da se neće unositi stringovi tipa XM), ali i dalje treba prijaviti grešku ukoliko se unese pogrešna cifra (npr. A).

10.5. Domaći zadaci

1. Napisati program za konverziju celog dekadnog broja u interni oblik u dvostrukoj preciznosti.
2. Napisati program koji će sve cifre heksadecimalnog celog broja koje su zadate malim slovima pretvoriti u velika slova.
3. Napisati program u asembleru za konverziju celog oktalnog broja u znakovnom obliku u binarni broj u znakovnom obliku.

11. Pregled korišćenih naredbi

U ovoj glavi je dat abecedni pregled asemblerskih naredbi koje su korišćene u programima u ovom praktikumu. Za svaku naredbu su navedeni operandi koji se uz nju očekuju, pri čemu oznaka *src* označava izvorni, *dst* označava odredišni operand, dok *cnt* označava brojački operand kod naredbi za pomeranje. Uz svaku naredbu je dat i tablarni pregled uticaja na pojedine indikatore, pri čemu je notacija za uticaj na pojedinačni indikator sledeća:

-	Nema promene
?	Vrednost je nedefinisana
M	Vrednost se menja u skladu sa rezultatom (ukoliko se drugačije ne naglasi)
T	Vrednost se testira tokom operacije
0 ili 1	indikator se postavlja na navedenu vrednost

Opisi naredbi u daljem tekstu važe za 32-bitni zaštićeni režim procesora *80386*.

adc *src*, *dst*

o	***d***	***s***	***z***	***c***
M	-	M	M	TM

Sabira izvorni i odredišni operand i rezultat smešta u odredišni operand. Prilikom sabiranja, zatečeni prenos se uzima u obzir.

add *src*, *dst*

o	***d***	***s***	***z***	***c***
M	-	M	M	M

Sabira izvorni i odredišni operand i rezultat smešta u odredišni operand. Prilikom sabiranja, zatečeni prenos se ne uzima u obzir.

and *src*, *dst*

o	***d***	***s***	***z***	***c***
0	-	M	M	0

Vrši operaciju logičkog I između korespondentnih bita izvornog i odredišnog operanda i rezultat smešta u odredišni.

call *dst*

o	***d***	***s***	***z***	***c***
-	-	-	-	-

Poziv potprograma. Prilikom poziva, na stek se smešta povratna adresa.

clc

o	***d***	***s***	***z***	***c***
-	-	-	-	0

Postavlja *carry* indikator na 0.

cld

o	***d***	***s***	***z***	***c***
-	0	-	-	-

Postavlja *direction* indikator na 0.

cmp *src, dst*

o	*d*	*s*	*z*	*c*
M	-	M	M	M

Oduzima izvorni operand od odredišnog, postavlja indikatore u skladu sa tim, ali ne menja odredišni operand.

dec *dst*

o	*d*	*s*	*z*	*c*
M	-	M	M	-

Oduzima 1 od odredišnog operanda.

div *src*

o	*d*	*s*	*z*	*c*
?	-	?	?	?

Neoznačeno deljenje. U zavisnosti od veličine izvornog operanda, ima sledeće dejstvo:

```
divb operand # ax/operand -> al, ostatak -> ah
divw operand # dx:ax/operand -> ax, ostatak -> dx
divl operand # edx:eax/operand -> eax, ostatak -> edx
```

`operand` – može biti registar ili memorija/promenljiva

Izvršavanje naredbe `div` može dovesti do pojave izuzetka. Ovo se može desiti u dva slučaja:

- ukoliko je vrednost operanda jednaka nuli
- ukoliko je dobijena vrednost prevelika da stane u odredište

idiv*src*

o	*d*	*s*	*z*	*c*
?	-	?	?	?

Označeno deljenje. U zavisnosti od veličine izvornog operanda, ima sledeće dejstvo:

```
idivb operand    # ax/operand -> al, ostatak -> ah
idivw operand    # dx:ax/operand -> ax, ostatak -> dx
idivl operand    # edx:eax/operand -> eax, ostatak -> edx
```

`operand` – može biti registar ili memorija/promenljiva

Izvršavanje naredbe `idiv` može dovesti do pojave izuzetka. Ovo se može desiti u dva slučaja:

- ukoliko je vrednost operanda jednaka nuli
- ukoliko je dobijena vrednost prevelika da stane u odredište

imul *op1[,op2[,op3]]*

o	*d*	*s*	*z*	*c*
M	-	?	?	M

Označeno množenje. U zavisnosti od veličine i broja operanada, ima sledeće dejstvo:

```
imulb operand    # al*operand -> ax
imulw operand    # ax*operand -> dx:ax
imull operand    # eax*operand -> edx:eax

imulw oper1, oper2 #oper1*oper2 -> oper2
imull oper1, oper2 #oper1*oper2 -> oper2

imulw const,oper1,oper2 #const*oper1 -> oper2
imull const,oper1,oper2 #const*oper1 -> oper2
```

Operandi imaju sledeća ograničenja:

`operand` – može biti registar ili memorija

`oper1` – može biti konstanta, registar ili memorija

`oper2` – može biti samo registar

`const` – može biti samo konstanta

inc *dst*

o	*d*	*s*	*z*	*c*
M	-	M	M	-

Dodaje 1 na odredišni operand.

int *dst*

o	*d*	*s*	*z*	*c*
-	-	-	-	-

Generiše softverski prekid.

jxx *dst*

o	*d*	*s*	*z*	*c*
-	-	-	-	-

Uslovni skok. Odredišni operand je rastojanje ciljne naredbe od naredbe koja sledi naredbu skoka (ovo rastojanje asembler računa automatski kada se kao operand stavi neka labela). Ovih naredbi ima više, a osnovne su date u sledećim tabelama:

Uslovni skokovi nakon poređenja neoznačenih brojeva		
`jxx`	**Skok ako je ...**	**odnosno ...**
`ja`	veće	**c** = 0 i **z** = 0
`jae`	veće ili jednako	**c** = 0
`jb`	manje	**c** = 1
`jbe`	manje ili jednako	**c** = 1 ili **z** = 1
`jna`	nije veće	**c** = 1 ili **z** = 1
`jnae`	nije veće ili jednako	**c** = 1
`jnb`	nije manje	**c** = 0
`jnbe`	nije manje ili jednako	**c** = 0 i **z** = 0
`je`	jednako	**z** = 1
`jne`	nije jednako	**z** = 0

Uslovni skokovi nakon poređenja označenih brojeva		
`jxx`	**Skok ako je ...**	**odnosno ...**
`jg`	veće	**z** = 0 i **s** = **o**
`jge`	veće ili jednako	**s** = **o**
`jl`	manje	**s** <> **o**
`jle`	manje ili jednako	**z** = 1 ili **s** <> **o**
`jng`	nije veće	**z** = 1 ili **s** <> **o**
`jnge`	nije veće ili jednako	**s** <> **o**
`jnl`	nije manje	**s** = **o**
`jnle`	nije manje ili jednako	**z** = 0 i **s** = **o**
`je`	jednako	**z** = 1
`jne`	nije jednako	**z** = 0

Ostali uslovni skokovi		
jxx	**Skok ako je ...**	**odnosno ...**
jc	prenos	**c** = 1
jnc	nije prenos	**c** = 0
jz	nula	**z** = 1
jnz	nije nula	**z** = 0
jo	prekoračenje	**o** = 1
jno	nije prekoračenje	**o** = 0
js	znak	**s** = 1
jns	nije znak	**s** = 0
jcxz	**cx** = 0	
jecxz	**ecx** = 0	

jmp *dst*

o	*d*	*s*	*z*	*c*
-	-	-	-	-

Bezuslovni skok.

lea *src*, *dst*

o	*d*	*s*	*z*	*c*
-	-	-	-	-

Smešta efektivnu adresu izvornog operanda u odredišni.

lods

o	*d*	*s*	*z*	*c*
-	-	-	-	-

U zavisnosti od veličine podrazumevanog operanda, ima sledeće dejstvo:

```
lodsb # (esi) -> al, esi+1 -> esi za d=0, odnosno esi-1 -> esi za d=1
lodsw # (esi) -> ax, esi+2 -> esi za d=0, odnosno esi-2 -> esi za d=1
lodsl # (esi) -> eax, esi+4 -> esi za d=0, odnosno esi-4 -> esi za d=1
```

Podrazumevani segmentni registar je `ds` (`ds:esi`).

loop *dst*

o	*d*	*s*	*z*	*c*
-	-	-	-	-

Smanjuje `cx` (`loopw`), odnosno `ecx` (`loopl`) za 1 i skače na ciljnu naredbu ako je rezultat različit od nule.

mov *src*, *dst*

o	*d*	*s*	*z*	*c*
-	-	-	-	-

Kopira izvorni operand u odredišni.

mul *src*

o	*d*	*s*	*z*	*c*
M	-	?	?	M

Neoznačeno množenje. U zavisnosti od veličine izvornog operanda, ima sledeće dejstvo:

```
mulb operand # al*operand -> ax
mulw operand # ax*operand -> dx:ax
mull operand # eax*operand -> edx:eax
```

`operand` – može biti registar ili memorija/promenljiva

neg *dst*

o	*d*	*s*	*z*	*c*
M	-	M	M	M

Negira odredišni operand po komplementu 2.

nop

o	*d*	*s*	*z*	*c*
-	-	-	-	-

Naredba bez efekta (troši nekoliko procesorskih ciklusa).

not *dst*

o	*d*	*s*	*z*	*c*
-	-	-	-	-

Negira odredišni operand po komplementu 1.

or *src, dst*

o	*d*	*s*	*z*	*c*
0	-	M	M	0

Vrši operaciju logičkog ILI između korespondentnih bita izvornog i odredišnog operanda i rezultat smešta u odredišni.

pop *dst*

o	*d*	*s*	*z*	*c*
-	-	-	-	-

Skida vrednost sa vrha steka i smešta je u odredišni operand, nakon čega se vrednost pokazivača na vrh steka poveća za 4.

push *src*

o	*d*	*s*	*z*	*c*
-	-	-	-	-

Smanji vrednost pokazivača na vrh steka za 4, a zatim izvorni operand smešta na novi vrh steka.

rcl *cnt, dst*

o	*d*	*s*	*z*	*c*
M/?	-	-	-	TM

Rotira odredišni operand kroz indikator *carry* u levo za navedeni broj mesta (neposredni operand ili registar `cl`). Ukoliko se rotira za više od jednog mesta, indikator *overflow* je nedefinisan.

rcr *cnt, dst*

o	*d*	*s*	*z*	*c*
M/?	-	-	-	TM

Rotira odredišni operand kroz indikator *carry* u desno za navedeni broj mesta (neposredni operand ili registar `cl`). Ukoliko se rotira za više od jednog mesta, indikator *overflow* je nedefinisan.

ret

o	*d*	*s*	*z*	*c*
-	-	-	-	-

Povratak iz potprograma. Povratna adresa se skida sa vrha steka, nakon čega se vrednost pokazivača na vrh steka poveća za 4.

rol *cnt, dst*

o	*d*	*s*	*z*	*c*
M/?	-	-	-	M

Rotira odredišni operand u levo za navedeni broj mesta (neposredni operand ili registar `cl`). Ukoliko se rotira za više od jednog mesta, indikator *overflow* je nedefinisan. Indikator *carry* ima vrednost poslednjeg bita upisanog na najmanje značajnu poziciju.

ror *cnt, dst*

o	*d*	*s*	*z*	*c*
M/?	-	-	-	M

Rotira odredišni operand u desno za navedeni broj mesta (neposredni operand ili registar `cl`). Ukoliko se rotira za više od jednog mesta, indikator *overflow* je nedefinisan. Indikator *carry* ima vrednost poslednjeg bita upisanog na najviše značajnu poziciju.

sar *cnt, dst*

o	*d*	*s*	*z*	*c*
M/?	-	M	M	M

Pomera odredišni operand u desno za navedeni broj mesta (neposredni operand ili registar `cl`). Naznačajniji bit zadržava svoju vrednost. Ukoliko se pomera za više od jednog mesta, indikator *overflow* je nedefinisan. Indikator *carry* ima vrednost poslednjeg bita istisnutog sa najmanje značajne pozicije.

sbb *src, dst*

o	*d*	*s*	*z*	*c*
M	-	M	M	TM

Oduzima izvorni operand od odredišnog i rezultat smešta u odredišni operand. Prilikom oduzimanja, zatečeni prenos se uzima u obzir.

shl *cnt, dst*

o	*d*	*s*	*z*	*c*
M/?	-	M	M	M

Pomera odredišni operand u levo za navedeni broj mesta (neposredni operand ili registar `cl`). Najmanje značajni bit dobija vrednost 0. Ukoliko se pomera za više od jednog mesta,

indikator *overflow* je nedefinisan. Indikator *carry* ima vrednost poslednjeg bita istisnutog sa najviše značajne pozicije.

shr *cnt, dst*

o	*d*	*s*	*z*	*c*
M/?	-	M	M	M

Pomera odredišni operand u desno za navedeni broj mesta (neposredni operand ili registar `cl`). Naznačajniji bit dobija vrednost 0. Ukoliko se pomera za više od jednog mesta, indikator *overflow* je nedefinisan. Indikator *carry* ima vrednost poslednjeg bita istisnutog sa najmanje značajne pozicije.

stc

o	*d*	*s*	*z*	*c*
-	-	-	-	1

Postavlja *carry* indikator na 1.

std

o	*d*	*s*	*z*	*c*
-	1	-	-	-

Postavlja *direction* indikator na 1.

stos

o	*d*	*s*	*z*	*c*
-	-	-	-	-

U zavisnosti od veličine podrazumevanog operanda, ima sledeće dejstvo:

```
stosb # al -> (edi), edi+1 -> edi za d=0, odnosno edi-1 -> edi za d=1
stosw # ax -> (edi), edi+2 -> edi za d=0, odnosno edi-2 -> edi za d=1
stosl # eax -> (edi), edi+4 -> edi za d=0, odnosno edi-4 -> edi za d=1
```

Podrazumevani segmentni registar je `es` (`es:edi`).

sub *src, dst*

o	*d*	*s*	*z*	*c*
M	-	M	M	M

Oduzima izvorni operand od odredišnog i rezultat smešta u odredišni operand. Prilikom oduzimanja, zatečeni prenos se ne uzima u obzir.

test *src, dst*

o	*d*	*s*	*z*	*c*
0	-	M	M	0

Vrši operaciju logičkog I između korespondentnih bita izvornog i odredišnog operanda, postavlja indikatore u skladu sa tim, ali ne menja odredišni operand.

xchg *src, dst*

o	*d*	*s*	*z*	*c*
-	-	-	-	-

Zamenjuje vrednosti izvornog i odredišnog operanda.

xor *src, dst*

o	*d*	*s*	*z*	*c*
0	-	M	M	0

Vrši operaciju ekskluzivnog ILI između korespondentnih bita izvornog i odredišnog operanda i rezultat smešta u odredišni.

Za detaljniji pregled naredbi procesora *80386*, čitalac se upućuje na knjigu *Intel 80386 Programmer's Reference Manual* (napomena za knjigu: sve je dato u *Intel*-ovoj sintaksi).

Dodatak A: Adresiranja procesora *Intel 80386*

Procesor *80386* podržava više adresiranja memorije: direktno, indirektno, indeksno, neposredno i registarsko, kao i kombinacije ovih adresiranja. Opšti format adresiranja memorije je:

```
adresa(%baza, %indeks, množilac)
```

pri čemu su sva polja opciona. Adresa memorijske lokacije se izračunava po formuli

```
adresa + baza + množilac*indeks
```

Polje `adresa` mora biti konstanta, `množilac` mora biti konstanta iz skupa {1,2,4,8}, dok `baza` i `indeks` moraju biti registri opšte namene (pri čemu se registar `esp` ne može koristiti kao `indeks`). Ukoliko se neko od polja ne navede, podrazumeva se da je 0, osim za `množilac` čija je podrazumevana vrednost 1. Treba obratiti i pažnju na činjenicu da kod naredbi sa dva operanda, samo jedan od njih može biti memorijski.

Direktno adresiranje ima samo polje `adresa`:

```
movl %eax, 0x1234          #eax -> lokacija 1234₁₆=4660₁₀
movl %eax, labela1         #eax -> lokacija labela1
```

Indirektno adresiranje sadrži samo polje `baza`:

```
movl %eax, (%ebx)          #eax -> lokacija čija je adresa u ebx
```

Bazno adresiranje sadrži polja `adresa` i `baza`:

```
movl %eax, 87(%ebx)        #eax -> lokacija čija je adresa ebx+87
```

Indeksno adresiranje sadrži polja `adresa`, `indeks` i eventualno `množilac`:

```
movl %eax, 1234(,%ecx)     #eax -> lokacija 1234+ecx
```

Vrednosti za `množilac` veće od 1 su pogodne za rad sa nizovima čiji su elementi veličine 2, 4 ili 8 bajtova. Na primer, ako je definisan niz `nizrec` čiji su elementi veličine 2 bajta, drugom elementu se može pristupiti sa:

```
movl $1, %ecx              #prvi element ima indeks 0
movw nizrec(,%ecx,2), %ax #vrednost sa lokacije nizrec + ecx*2 -> ax
```

Ukoliko veličina elementa niza nije 1, 2, 4 ili 8 bajtova, tada se za pristup elementima niza mora koristiti neka varijanta indirektnog adresiranja.

Dodatak B: Prevođenje programa korak po korak

Prilikom prevođenja programa korišćenjem gcc-a, praktično se obavljaju tri koraka: makro pretprocesiranje, prevođenje asemblerskog kôda u objektni i povezivanje objektnog kôda sa sistemskim bibliotekama. Svaki od ovih koraka se može odraditi i zasebno:

Za makro pretprocesiranje asemblerskog kôda se koristi *C* **pretprocesor** `gcc`-a:

```
gcc -m32 -E primer.S -o primer.s
```

Opcija `-E` omogućava da se dobije izlaz pretprocesora, dok opcija `-o` služi za zadavanje naziva izlaznog fajla. Ukoliko usluge pretprocesora nisu potrebne, ovaj korak se može preskočiti. Za asembliranje se koristi komanda `as` (*GNU* assembler) kojom se poziva **asembler**:

```
as --gdwarf2 primer.s -o primer.o
```

Opcija `--gdwarf2` omogućava dibagiranje programa. Ako je prevođenje prošlo bez grešaka, sistem će vratiti komandni prompt ne ispisujući nikakve dodatne poruke. Linkovanje obavlja program `ld` (*GNU* ***linker***):

```
ld primer.o -o primer
```

Linker će povezati fajl `primer.o`, nastao kao rezultat asembliranja, i kreirati izvršni fajl `primer`.

Dodatak C: Konverzije brojeva u binarnom znakovnom obliku

Konverzija celih binarnih brojeva iz internog oblika u znakovni oblik

Program na *C*-u za konverziju iz internog oblika u znakovni oblik celog binarnog broja i obrtanje rezultujućeg niza prikazan je na slici 20.

Kod asemblerske verzije ovog algoritma, za binarni zapis je predviđeno 33 bajta, što je dovoljno za broj u jednostrukoj preciznosti i završni *NUL*. Sâm program dat je u nastavku teksta.

Program koristi specifičnosti binarnog brojnog sistema, pa tako, deljenje sa 2 nije realizovano pomoću naredbe `div`, nego su iskorišćene naredbe za pomeranje i rotiranje kako bi se ostatak pri deljenju sa 2 direktno prebacio u registar `bl`.

```
#include <ctype.h>
char broj[33];
char *b = broj;

char *s, *t;

s = t = (char *) b;
do {
     *t++ = (r & 1) + '0';
     r >>= 1;
} while (r);
*t-- ='\0';
while (s < t) {
     char c = *s;
     *s++ = *t;
     *t-- = c;
}
```

Slika 20: Konverzija iz internog formata u binarni znakovni format

Asemblerska verzija programa izgleda:

```
bin_br_duz = 33
bin_br:  .fill bin_br_duz,1,0      #rezervisanje 33 bajta
...
     movl $31337, %eax             #broj koji se konvertuje
     leal bin_br, %edi
bin_cifra:
     shrl $1, %eax                 #prebacivanje najnižeg bita eax
     rclb $1, %bl                  #u bl
     andb $1, %bl
     addb $'0', %bl
     movb %bl, (%edi)              #upis znaka u string
     incl %edi
     andl %eax, %eax               #provera kraja algoritma
     jnz bin_cifra
     movb $0, (%edi)               #kraj stringa
     decl %edi                     #obrtanje dobijenog niza
     leal bin_br, %esi
obrni:
     cmpl %edi, %esi
     jae kraj
     movb (%esi), %ah
     movb (%edi), %al
     movb %al, (%esi)
     movb %ah, (%edi)
     incl %esi
     decl %edi
     jmp obrni
kraj:
     movl $1, %eax
     movl $0, %ebx
     int $0x80
```

Konverzija razlomljenih binarnih brojeva iz internog oblika u znakovni oblik

Razlomljen broj, u decimalnom zapisu oblika

$$x_{n-1}b^{n-1}+x_{n-2}b^{n-2}+\cdots+x_1 b+x_0$$

može se konvertovati u ekvivalentni binarni zapis po sledećem algoritmu:

1. Pomnoži broj sa dva
2. Ako je rezultat manji od jedan, odgovarajuća binarna cifra je 0; zapiši je i pređi na prvi korak
3. U suprotnom, zapiši binarnu cifru 1 i oduzmi jedinicu od broja
4. Ako je broj različit od nule, pređi na prvi korak, u suprotnom završi konverziju

Problem kod implementacije ovog algoritma u asembleru je što standardne procesorske naredbe barataju isključivo celobrojnim vrednostima. Ovo se može rešiti **skaliranjem** razlomljenog broja nekim stepenom broja 10: ako se implementacija zadrži na jednostrukoj preciznosti, skaliranje se može izvesti sa 10^9. Takvim postupkom bi, na primer, vrednost 0.2 bila predstavljena kao 200000000, a vrednost 0.0625 kao 62500000 ili 062500000. Drugi problem sa ovom konverzijom je mogućnost da se prilikom konverzije javi beskonačna periodičnost u rezultatu. Pošto je kapacitet memorije ograničen, mora se postaviti gornja granica broja cifara konvertovanog zapisa. Ako se skaliranje izvede na ranije opisan način, ograničavanje broja cifara na 32 će dati binarni razlomak uporedive preciznosti.

```
bin_br_duz = 33
bin_br:  .fill bin_br_duz,1,0
...
     movl $375000000, %eax     #broj za konverziju
     leal bin_br, %edi
     movl $32, %ecx            #max broj cifara
bin_cifra:
     movb $'0', %dl
     addl %eax, %eax
     cmpl $1000000000, %eax    #1 ili 0?
     jb nije_vece
     incb %dl
     subl $1000000000, %eax    #oduzimanje jedinice
nije_vece:
     movb %dl, (%edi)
     incl %edi
     andl %eax, %eax           #kraj algoritma?
     jz kraj_bin
     loopl bin_cifra           #ima još mesta u stringu?
kraj_bin:
     movb $0, (%edi)           #kraj stringa
kraj:
     movl $1, %eax
     movl $0, %ebx
     int $0x80
```

Konverzija razlomljenih binarnih brojeva iz znakovnog oblika u interni oblik

Binarni razlomak oblika

$$b_1 2^{-1} + b_2 2^{-2} + \cdots + b_n 2^{-n}$$

može se konvertovati u decimalni zapis ispitivanjem sukcesivnih bita i sumiranjem vrednosti 2^{-i} (počev od *1/2*) za svaku poziciju gde je odgovarajući bit jednak jedinici.

Neposredna implementacija ovog algoritma, u sprezi sa skaliranjem opisanim u prethodnom odeljku, značila bi korišćenje početne vrednosti 500000000 i njeno polovljenje u svakom narednom koraku. Ipak, za postizanje veće tačnosti rezultata bolje je primeniti alternativni postupak: ispitivanje binarnih cifara s kraja, s tim da se rezultat polovi pre ispitivanja, a dodaje mu se 500000000 ako se pokaže da je na tekućoj poziciji bit vrednosti jedan.

S obzirom na metod, broj binarnih cifara ne bi morao biti ograničen na 32 ukoliko bi se prvo utvrdila lokacija kraja niza i sa konverzijom krenulo ka njegovom početku. U ovoj verziji uvedeno je ograničenje radi jednostavnije konverzije binarnog znakovnog oblika u interni oblik, pošto se koristi registar kapaciteta 32 bita.

```
bin_br:  .ascii "1000100010110100\0"
greska:  .byte 0
...
     leal bin_br, %esi
     movb $0, greska
     xorl %ebx, %ebx
     xorl %ecx, %ecx
bin_cifra:
     cmpl $32, %ecx        #više od 32 cifre?
     ja i_greska
     movb (%esi), %al
     andb %al, %al         #kraj stringa?
     jz kraj_bin
     subb $'0', %al        #izdvajanje cifre
     jc i_greska
     cmpb $1, %al
     ja i_greska
     shrb $1, %al          #ubacivanje cifre u broj
     rcll $1, %ebx
     jc i_greska           #provera prekoračenja
     incl %esi
     incl %ecx
     jmp bin_cifra
i_greska:
     incb greska
     jmp kraj
kraj_bin:
     jecxz i_greska        #prazan string?
     xorl %eax, %eax
razl_poz:
     shrl $1, %eax         #rezultat/2 -> rezultat
     shrl $1, %ebx         #izdvajanje binarne cifre
     jnc nije_jedan
     addl $500000000, %eax
nije_jedan:
     loopl razl_poz
kraj:
     movl $1, %eax
     movl $0, %ebx
     int $0x80
```

Broj obrađenih binarnih cifara mora se pamtiti (`ecx`) jer se za proveru uslova kraja konverzije ne može iskoristiti nulta vrednost registra sa internim oblikom vrednosti, pošto su vodeće nule u razlomljenom broju bitne.

Dodatak D: Makroi

Iako su potprogrami osnovni i najčešće korišćen način za modularizaciju kôda, oni nisu uvek najefikasniji. Same operacije poziva i povratka traju nezanemarljivo vreme, koje produžava vreme obrade. Ako je potprogram pisan tako da poštuje neku pozivnu konvenciju, ovakav trošak vremena se povećava proporcionalno broju naredbi potrebnih da bi se konvencija implementirala. Ako se potprogramom izvodi relativno kratak skup operacija (kao što je, na primer, sistemski poziv), lako se može desiti da trošak vezan za poziv potprograma bude uporediv sa vremenom obrade. U vremenski kritičnim delovima kôda, ili u slučaju velikog broja poziva, ukupan trošak poziva potprograma može biti neprihvatljivo veliki.

Jedan od načina da se trošak potprogramskog poziva izbegne, a da izvorni kôd ne postane manje modularan, jeste korišćenje **makroa,** odnosno makro definicija i makro poziva. Makro definicije sadrže nizove naredbi koje zamenjuju makro poziv u tekstu programa.

Prosta zamena makro poziva naredbama iz makro definicije je korisna, ali relativno ograničene upotrebljivosti. Zbog toga većina makro-asemblera omogućava makro definicije sa parametrima (kao i makro pozive sa argumentima). Pored toga, podržane su i lokalne labele, specijalni operatori i konstrukcije za uslovno prevođenje.

Prilikom upotrebe makroa treba imati nekoliko stvari na umu. Prvo, svi iole složeniji makroi će raditi sa registrima. Ako se upotreba registara propisno ne dokumentuje, poziv makroa može da stvori nepoželjne sporedne efekte, na primer uništenje sadržaja registra koji se koristi u kasnijem toku programa. Ovakve greške nisu jednostavne za nalaženje. Drugo, preterana upotreba makroa može dovesti do toga da je kôd razumljiv samo autoru (a možda ni toliko). Makroe je zato najbolje koristiti umereno, i pažljivo ih dokumentovati.

Makro se definiše na sledeći način:

```
.macro naziv_makroa par1=pv1, par2=pv2, ...
  <telo makroa>
.endm
```

Parametri (par_i) su opcioni, kao i njihove podrazumevane vrednosti (pv_i). Kada u telu makroa treba iskoristiti neki parametar, njegovo ime se mora navoditi sa prefiksom "`\`", na primer `\par1`. Tekst koji se dobija nakon poziva makroa se može videti pozivom asemblera sa sledećim parametrima:

```
as –alm program.S
```

Primer 1 *- makro sistemskog poziva exit*

```
.macro kraj_rada greska=$0
    movl $1, %eax
    movl \greska, %ebx
    int $0x80
.endm
```

Pozivanje makroa `kraj_rada` bez parametara bi dalo sledeću sekvencu programa:

```
    movl $1, %eax
    movl $0, %ebx
    int $0x80
```

dok bi pozivanje istog makroa sa parametrom `$3` dalo sledeću sekvencu programa:

```
    movl $1, %eax
    movl $3, %ebx
    int $0x80
```

Ukoliko je u makrou potrebno definisati labelu, javlja se problem kada postoji više makro poziva u istom programu. Naime, isti naziv labele bi se pojavio dva ili više puta u tekstu

programa i asembler bi prijavio grešku. Zbog toga se u makroima koriste **lokalne labele**. Lokalnih labela ima 10 i definišu se ciframa od "`0`" do "`9`". Kada labelu treba iskoristiti kao operand neke naredbe, tada cifru labele dopunjava slovo "`b`" ili "`f`". Slovo "`b`" (*backwards*) se koristi kada definicija labele prethodi mestu korišćenja , dok se slovo "`f`" (*forwards*) koristiti kada definicija labele sledi posle mesta korišćenja. Lokalne labele se mogu koristiti kako u makroima, tako i u programu. Mogu se definisati proizvoljan broj puta i tada se njihovo pozivanje odnosi na poslednju definiciju labele (slovo "`b`"), odnosno na prvu sledeću definiciju labele (slovo "`f`").

***Primer 2** - makro za označeno celobrojno deljenje sa 10*

Proizvod 10x može se napisati i kao 8x + 2x. Pošto su 8 i 2 stepeni broja 2, množenje njima svodi se na uzastopno pomeranje u levo ili sabiranje operanda sa samim sobom. Sledeći makro množi označeni celobrojni operand u jednostrukoj preciznosti brojem deset koristeći takvo razlaganje.

```
.macro mnozi10o regmem    #registar (osim eax) ili mem. Lokacija
    pushl %eax
    movl \regmem, %eax
    pushl %ebx
    addl %eax, %eax
    jo 1f
    movl %eax, %ebx
    addl %eax, %eax
    jo 1f
    addl %eax, %eax
    jo 1f
    addl %ebx, %eax
1:
    popl %ebx
    movl %eax, \regmem
    popl %eax
.endm
```

Ovaj makro ostavlja rezultat množenja u registru ili memorijskoj lokaciji navedenoj kao argument (jedino se ne može koristiti registar `eax`). Sadržaj svih registara (osim odredišnog, ako je kao argument naveden registar) se čuva. U slučaju izlaska van opsega biće postavljen indikator `o`.

***Primer 3** - makro za određivanje tipa znaka*

Kod obrade stringova često je potrebno ispitivati tip pojedinih znakova. Na *C*-u su za to zadužene funkcije deklarisane u standardnom zaglavlju `<ctype.h>`. Na primer, `isspace` ispituje da li je znak delimiter (neki od znakova za razdvajanje), a `isdigit` proverava da li je znak cifra dekadnog brojnog sistema. Ove funkcije se često implementiraju kao makroi, da bi se kod ispitivanja velikog broja znakova izbegao trošak funkcijskog poziva za svaki obrađen znak.

Efikasan metod ispitivanja zahteva postojanje tabele čija veličina u bajtovima odgovara broju mogućih znakova. Za 8-bitni tip `char` ovo znači utrošak od 256 bajtova. Svaki bajt u tabeli je bit-maska nastala kombinovanjem (pomoću logičkog ILI) konstanti za klase kojima pripada odgovarajući znak. Konstante za pojedine klase su različiti stepeni broja 2. Lokacija bit-maske za određeni znak dobija se indeksiranjem tabele njegovim kodom, što je operacija složenosti O(1).

Tabela bi mogla da izgleda ovako:

```
_IS_SP  = 1
_IS_DIG = 2
_IS_UPP = 4
_IS_LOW = 8
_IS_HEX = 16
_IS_CTL = 32
_IS_PUN = 64
```

```
ctype:
     .fill 9,1,_IS_CTL
     .fill 5,1,_IS_CTL | _IS_SP
     .fill 18,1, _IS_CTL
     .fill 1,1, _IS_SP
     .fill 15,1, _IS_PUN
     .fill 10,1, _IS_DIG
     .fill 7,1, _IS_PUN
     .fill 6,1, _IS_UPP | _IS_HEX
     .fill 20,1, _IS_UPP
     .fill 6,1, _IS_PUN
     .fill 6,1, _IS_LOW | _IS_HEX
     .fill 20,1, _IS_LOW
     .fill 4,1, _IS_PUN
     .fill 1,1, _IS_CTL
     .fill 128,1, _IS_PUN
```

Asemblerski makroi, ekvivalenti za `isdigit` i `isspace` se sada mogu napisati kao:

```
.macro isclass character, class
    xorl %ebx, %ebx
    movb \character, %bl
    testl $\class, ctype(,%ebx,1)
.endm

.macro isspace character          #8-bitni registar ili memorija
    isclass \character, _IS_SP
.endm

.macro isdigit character          #8-bitni registar ili memorija
    isclass \character, _IS_DIG
.endm
```

Makroi menjaju sadržaj registra `ebx`. Rezultat izvršavanja je stanje indikatora posle naredbe `test`.

Domaći zadaci

1. Prepraviti program za množenje pomoću sabiranja u dvostrukoj preciznosti da koristi standardni ulaz i izlaz za komunikaciju sa korisnikom.
2. Napisati makro za sistemski poziv za ispis na ekran. Makro bi trebao da ima jedan argument, string koji treba ispisati na ekran, dok bi se određivanje dužine stringa radilo u samom makrou.

Dodatak E: Mašinska normalizovana forma

Primer mašinske normalizovane forme

Mašinska normalizovana forma je način predstavljanja realnih brojeva u računaru. Sastoji se od znaka, podešenog eksponenta i frakcije. Ukoliko se, na primer, usvoji da se za eksponent rezerviše 8 bita (*IEEE* standard za 32-bitnu mašinsku normalizovanu formu), tada se 32-bitna vrednost može posmatrati na sledeći način:

Z	Eksponent								Frakcija																							
1	1	0	0	0	0	0	1	1	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1
31	30							23	22																						0

Slika 21: Broj u mašinskoj normalizovanoj formi sa 8 bita za eksponent

Za znak je rezervisana najviša pozicija u broju i može imati vrednost 0 (pozitivno) ili 1 (negativno).

U sledećih 8 bita se nalazi podešeni eksponent. U ovom slučaju konstanta podešavanja iznosi 10000000_2, pa je vrednost upisanog eksponenta 11_2. Maksimalna vrednost podešenog eksponenta je 11111111_2, dok je minimalna 00000001_2 (vrednost 00000000_2 je rezervisana za predstavljanje nule).

Frakcija se predstavlja sa preostalih 23 bita, s tim da se najznačajnija cifra frakcije (koja je uvek 1) ne upisuje, čime se dobija veća tačnost. To znači da u ovom primeru vrednost frakcije u stvari iznosi $10111111100000000000011111_2$.

Izdvajanje delova mašinske normalizovane forme

Pre bilo koje operacije sa brojevima u mašinskoj normalizovanoj formi, potrebno je izdvojiti delove od kojih se oni sastoje: znak, eksponent i frakciju. Za to se koristi maskiranje i pomeranje. U primeru sa slike 21, znak se može izdvojiti maskiranjem najznačajnijeg bita. Eksponent se može izdvojiti maskiranjem bita na pozicijama od 23 do 30, pomeranjem dobijene vrednosti udesno za 23 mesta i oduzimanjem konstante podešavanja. Frakcija se može izdvojiti maskiranjem bita na pozicijama od 0 do 22 i dodavanjem podrazumevane jedinice na poziciju 23.

Za neke operacije je zgodno uključiti znak broja u mašinskoj normalizovanoj formi u izdvojenu frakciju, pa rukovanje znakom rezultata staviti u nadležnost aritmetičkih naredbi procesora. Takođe, posebno treba voditi računa i o vrednosti 0 (nula).

Operacije sa brojevima u mašinskoj normalizovanoj formi

Prilikom obavljanja operacija sa brojevima u mašinskoj normalizovanoj formi, treba imati na umu da je frakcija u stvari skaliran razlomljen broj. U primeru sa slike 21, konstanta skaliranja je 2^{23}. Znak se može tretirati ili posebno ili se može uključiti u frakciju.

Sabiranje i **oduzimanje** se svode na sledeće operacije:

- dovođenje eksponenata na istu vrednost (manji se dovodi na vrednost većeg, uz pomeranje frakcije)
- sabiranje, odnosno oduzimanje frakcija

Množenje se svodi na sledeće operacije:

- sabiranje eksponenata
- množenje frakcija
- deljenje rezultata konstantom skaliranja

Deljenje se svodi na sledeće operacije:

- oduzimanje eksponenata
- množenje rezultata konstantom skaliranja
- deljenje frakcija

Nakon obavljanja bilo koje od operacija, neophodno je obaviti **normmalizaciju rezultata**, odnosno obezbediti da se najznačajnija cifra u frakciji nalazi na odgovarajućoj poziciji (u primeru sa slike 21, to je pozicija 23). Ukoliko se prilikom normalizacije menja frakcija, neophodno je menjati i eksponent, tako da vrednost broja ostane ista.

Sastavljanje broja u mašinskoj normalizovanoj formi

Kada je rezultat operacije normalizovan, može se spakovati u 32-bitni broj postupkom obrnutim od izdvajanja delova. Pri tome treba voditi računa i o znaku izdvojene frakcije, ukoliko je on uključen u nju (u tom slučaju ga treba izdvojiti). Za primer sa slike 21, to znači da se iz frakcije obriše najznačajnija jedinica, a na eksponent se doda konstanta podešavanja i ta vrednost se pomeri ulevo za 23 pozicije. Ove vrednosti se zatim, upotrebom logičkih operacija, zajedno sa vrednošću znaka, iskombinuje u jedinstvenu 32-bitnu vrednost. Pri sastavljanju posebno treba obratiti pažnju na vrednost 0 (nula), pošto se ona drugačije tretira.

Domaći zadatak

1. Napisati potprogram za konverziju stringa koji predstavlja binarnu decimalnu vrednost oblika “1011.011” u vrednost u mašinskoj normalizovanoj formi sa 10 bita rezervisanih za eksponent.

   ```
   int BinStrToMNF(char* string, int* vrednost)
   ```

   Povratna vrednost potprograma je 0 ako nije bilo greške pri konverziji, odnosno 1 ako jeste.

Dodatak F: *Shell* skriptovi

***Shell* skriptovi** (*script*; u daljem tekstu skriptovi) su programi koji se izvršavaju u okviru *shell*-a. To može biti jednostavno niz komandi koje treba izvršiti jednu za drugom, a koji se relativno često ponavlja. Skriptovi mogu biti i znatno komplikovaniji programi, pošto *bash* podržava mnoge elemente strukturnog programiranja.

Jednostavni skriptovi

Skript je (skoro) običan tekstualni fajl koji se može otkucati u bilo kom tekst editoru. Na početku svakog skripta treba da stoji sledeći tekst:

```
#!/bin/bash
```

ili

```
#!/bin/sh
```

Prva dva znaka predstavljaju tzv. **magični broj** (*magic number*), specijalnu oznaku na počeku svakog izvršnog fajla koja određuje njegov tip. U ovom slučaju, reč je o skriptu. Ostatak linije je putanja do programa koji će izvršiti skript. Prvi primer kaže da će se koristiti *bash*, a drugi da će se koristiti *sh* (stariji *shell*, sa kojim je *bash* kompatibilan). Inače, oznaka "`#`" predstavlja oznaku za komentar, pa se sve posle "`#`" pa do kraja linije smatra komentarom i ne izvršava se. Ispod ove linije se standardno stavlja red sa komentarom koji opisuje šta skript radi, a nakon toga se stavljaju naredbe. Na primer, treba u jednom skriptu objediniti prelazak u home direktorijum i njegovo listanje na ekranu. Nazovimo ga `hl`:

```
#!/bin/bash
#prelazak u home direktorijum i njegovo listanje
echo Prelazak u home direktorijum i njegovo listanje
cd # prelazak u home direktorijum
ls -l #listanje direktorijuma
```

Skript se može kreirati, na primer, kucanjem `gedit hl`, nakon čega treba otkucati gornji program. Komanda `echo` služi za ispis teksta. Kada je skript otkucan i snimljen, može se preći na njegovo izvršavanje:

```
bash hl
```

Mnogo elegantniji način za izvršavanje je da se skript proglasi za izvršni fajl. To se postiže komandom:

```
chmod u+x hl
```

Sada se ne mora kucati `bash hl`, nego je dovoljno samo

```
./hl
```

Promenljive

Promenljive u skriptovima se jednostavno definišu. Dovoljno je samo u telu skripta napisati:

```
ime_promenljive=vrednost
```

Ono na šta treba obratiti pažnju je da ni sa leve ni sa desne strane znaka "=" ne sme biti razmaka. Kada se promenljiva želi koristiti, tada se piše "`$ime_promenljive`", čime se pristupa njenoj vrednosti. Znaci navoda nisu obavezni, ali se preporučuju. Ukoliko je potrebno da se odmah nakon naziva promenljive nađe neki tekst tada se koristi zapis "`${ime_promenljive}tekst`". Generalno, sve promenljive su string tipa, jedino što *bash* dozvoljava osnovne računske operacije ukoliko su svi znaci u stringu cifre (celi brojevi). Kod definisanja promenljivih, ukoliko string sadrži razmak, mora se staviti pod znake navoda. Primeri:

```
x=2
ime_i_prezime="Pera Peric"
ime=Pera
prezime="Peric"
```

Posebna vrsta promenljivih su one koje se u skriptu koriste kao ulazni parametri. Nazivaju se i **pozicioni parametri**, pošto im je vrednost vezana za poziciju prosleđenog argumenta. Njihova imena su `$0`, `$1`, ... `$9`. Na primer, ako se skript zove `xyz`, tada će prilikom poziva

```
xyz 1 "pera peric" mika 123 456
```

promenljive `$0` do `$5` imati sledeće vrednosti:

```
$0=xyz, $1=1, $2="pera peric", $3=mika, $4=123, $5=456
```

dok će ostale biti prazne. Postoji i posebna promenljiva `$#` koja sadrži broj prenetih argumenata, kao i `$*` koja u sebi sadrži sve prosleđene argumente. Koriste se isto kao i obične promenljive. Vrednosti pozicionih parametara (svih osim prvog) se mogu pomerati ulevo pomoću komande `shift`. Tako, nakon izvršenja komande `shift`, vrednosti promenljivih iz prethodnog primera bi bile:

```
$0=xyz, $1="pera peric", $2=mika, $3=123, $4=456
```

dok bi ostale bile prazne. Ovo je korisno ako bi skript trebao da prima nedefinisani broj parametara, da bi se svi mogli obraditi u nekoj petlji.

Upravljačke strukture

Osnovna upravljačka struktura je `if`, koja postoji u nekoliko varijanti. Najjednostavija varijanta je:

```
if <uslov>
then
  <naredbe>
fi
```

Opciono, može se dodati i `else` deo:

```
if <uslov1>
then
  <naredbe1>
else
  <naredbe2>
fi
```

Proširena verzija omogućava višestruke upite:

```
if <uslov1>
then
  <naredbe1>
elif <uslov2>
then
  <naredbe2>
elif ...
fi
```

Kao naredbe se mogu staviti bilo koje naredbe koje se mogu izvršiti u *bash*-u. Uslov takođe može biti bilo koja naredba, pri čemu se smatra da je uslov tačan ako je povratna vrednost naredbe 0 (nula), dok se bilo koja druga vrednost smatra za netačno. Uslov može imati i sledeći oblik:

```
[<razmak>operator<razmak>operand<razmak>] (za unarne operatore)
[<razmak>operand1<razmak>operator<razmak>operand2<razmak>] (za binarne)
```

Operandi mogu biti promenljive ili konstante (string ili brojevi). Neki od operatora su dati u sledećoj tabeli:

Operator	***Vrsta***	***Vraća vrednost "tačno" ako ...***
`-n`	unarni	operand nije prazan
`-z`	unarni	operand je prazan
`-d`	unarni	postoji direktorijum čije je ime operand
`-f`	unarni	postoji fajl čije je ime operand
`-eq`	binarni	su operandi celi brojevi i ako su isti
`-ne`	binarni	su operandi celi brojevi i ako su različiti
`=`	binarni	operandi su jednaki (posmatrani kao stringovi)
`!=`	binarni	operandi su različiti (posmatrani kao stringovi)
`-lt`	binarni	operand 1 je manji od operanda 2 (celi brojevi)
`-gt`	binarni	operand 1 je veći od operanda 2 (celi brojevi)
`-le`	binarni	operand 1 je manji ili jednak operandu 2 (celi brojevi)
`-ge`	binarni	operand 1 je veći ili jednak operandu 2 (celi brojevi)

Tabela 5: Operatori *bash*-a

Na primer, treba proveriti da li je skriptu prosleđen tačno jedan parametar koji je ceo broj i koji je veći od 3, pa ako jeste, vratiti tu vrednost uvećanu za 1:

```
#!/bin/bash
#primer za if
if [ "$#" -ne 1 ] #provera da li postoji tacno 1 parametar
then
  echo "Unesite tacno 1 argument"
  exit 1
elif [ "$1" -le 3 ] #provera da li je parametar veci od 3
then
  echo "Unesite parametar koji je veci od 3"
  exit 2
else
  x=$(($1 + 1))
  echo "Uneti broj uvecan za 1 iznosi $x"
fi
exit 0
```

Komanda `exit` služi da se OS-u vrati kôd greške. Po konvenciji, ako nema greške, sistemu se vraća kôd `0`, dok se u slučaju greške vraća kôd različit od `0`. Posebna promenljiva `$?` u sebi sadrži kôd greške poslednje izvršene naredbe. **Aritmetički izrazi** se navode u dvostrukim malim zagradama ispred kojih stoji znak `$`.

Petlje

Prva petlja koja će biti pomenuta je `for`. Njen osnovni oblik je:

```
for brojacka_promenljiva in lista_vrednosti
do
  <naredbe>
done
```

Lista vrednosti je lista stringova razdvojenih razmacima. U primeru:

```
#!/bin/bash
#for petlja
for boja in crvena zelena plava
do
  echo "Jedna od boja je $boja"
done
```

na ekranu će se ispisati:

```
Jedna od boja je crvena
Jedna od boja je zelena
Jedna od boja je plava
```

Argument u listi vrednosti može biti i naziv fajla ili direktorijuma, u kome se mogu naći i džoker znaci. U tom slučaju `for` komanda će za vrednost brojačke promenljive uzimati sve nazive fajlova koji se slažu sa argumentom. Na primer, treba napisati skript koji će pretražiti sve tekstualne fajlove u tekućem direktorijumu i ispisati koliko puta se zadata reč nalazi u svakom od njih:

```
#!/bin/bash
#for petlja, primer 2
for fajl in *.txt #prolazak kroz sve txt fajlove
do
  if [ -f "$fajl" ]
    n=$(grep -c -i "$1" "$fajl") #n dobija vrednost izlaza komande grep
    if [ "$n" -eq 0 ] #ako je n nula
    then
      echo "U falju $fajl rec '$1' se ne pojavljuje"
    elif [ "$n" -eq 1 ] #ako je n jedan
      echo "U falju $fajl rec '$1' se pojavljuje jednom"
    else #ako je n > 1
      echo "U falju $fajl rec '$1' se pojavljuje $n puta"
    fi
  fi
done
```

Konstrukcija `promenljiva=$(komanda)` se naziva **zamena komandi** (*command substitution*) i omogućava da se izlaz komande unutar zagrade dodeli promenljivoj. Opcija `-c` komande `grep` znači da će se vratiti broj pojavljivanja reči u fajlu, a onda će u skriptu taj broj biti dodeljen promenljivoj `n`. Dalje se proverava vrednost promenljive `n` i u zavisnosti od toga se ispisuje odgovarajuća poruka.

Druga petlja koja će biti pomenuta je `while`. Njen oblik je:

```
while <uslov>
do
  <naredbe>
done
```

Uslov se definiše isto kao kod `if` komande. Značenje je da se telo petlje izvršava dok god je uslov tačan. Na primer, ispis prvih *n* prirodnih brojeva bi izgledao ovako:

```
#!/bin/bash
#for petlja, primer 2
x=1
while [ "$x" -le "$1" ]
do
  echo -n "$x "
  x=$(($x+1))
done
```

Opcija `-n` komande `echo` znači da se nakon ispisa neće preći u novi red. Tako, na primer, ako bi se skript pozvao sa argumentom 5, izlaz bi bio “1 2 3 4 5”.

Specijalne shell promenljive

Promenljiva	***Značenje***
$0	naziv skripta
$1	pozicioni parametar 1
$2 - $9	pozicioni parametri 2 - 9
${10}	pozicioni parametar 10
$#	broj pozicionih parametara
"$*"	svi pozicioni parametrikao jedna reč (kao jedan parametar)
"$@"	svi pozicioni parametri (svaki zasebno)
${#*}	broj parametara prosleđenih skriptu
${#@}	broj parametara prosleđenih skriptu
$?	povratna vrednost poslednje izvršene komande
$$	pid (*Process ID*) skripta
$-	svi set indikatori prosleđeni skriptu
$_	poslednji argument prethodne komande
$!	pid (*Process ID*) poslednjeg procesa pokrenutog u pozadini

Tabela 6: Specijalne *shell* promenljive

Korisni linkovi

Na sledećim web stranicama se može naći više detalja vezanih za pisanje skriptova:

1. www.tldp.org/ldp/abs/html/
2. tille.xalasys.com/training/bash/
3. www.linuxtopia.org/online_books/advanced_bash_scripting_guide/refcards.html

Dodatak G: *ASCII* tabela

Decimal	Octal	Hex	Binary	Value
000	000	000	00000000	NUL (Null char.)
001	001	001	00000001	SOH (Start of Header)
002	002	002	00000010	STX (Start of Text)
003	003	003	00000011	ETX (End of Text)
004	004	004	00000100	EOT (End of Transmission)
005	005	005	00000101	ENQ (Enquiry)
006	006	006	00000110	ACK (Acknowledgment)
007	007	007	00000111	BEL (Bell)
008	010	008	00001000	BS (Backspace)
009	011	009	00001001	HT (Horizontal Tab)
010	012	00A	00001010	LF (Line Feed)
011	013	00B	00001011	VT (Vertical Tab)
012	014	00C	00001100	FF (Form Feed)
013	015	00D	00001101	CR (Carriage Return)
014	016	00E	00001110	SO (Shift Out)
015	017	00F	00001111	SI (Shift In)
016	020	010	00010000	DLE (Data Link Escape)
017	021	011	00010001	DC1 (XON/Device Control 1)
018	022	012	00010010	DC2 (Device Control 2)
019	023	013	00010011	DC3 (XOFF/Device Control 3)
020	024	014	00010100	DC4 (Device Control 4)
021	025	015	00010101	NAK (Negative Acknowledgement)
022	026	016	00010110	SYN (Synchronous Idle)
023	027	017	00010111	ETB (End of Trans. Block)
024	030	018	00011000	CAN (Cancel)
025	031	019	00011001	EM (End of Medium)
026	032	01A	00011010	SUB (Substitute)
027	033	01B	00011011	ESC (Escape)
028	034	01C	00011100	FS (File Separator)
029	035	01D	00011101	GS (Group Separator)
030	036	01E	00011110	RS (Request to Send)(Record Separator)
031	037	01F	00011111	US (Unit Separator)
032	040	020	00100000	SP (Space)
033	041	021	00100001	! (exclamation mark)
034	042	022	00100010	" (double quote)
035	043	023	00100011	# (number sign)
036	044	024	00100100	$ (dollar sign)
037	045	025	00100101	% (percent)
038	046	026	00100110	& (ampersand)
039	047	027	00100111	' (single quote)
040	050	028	00101000	((left/opening parenthesis)
041	051	029	00101001	) (right/closing parenthesis)
042	052	02A	00101010	* (asterisk)
043	053	02B	00101011	+ (plus)
044	054	02C	00101100	, (comma)
045	055	02D	00101101	- (minus or dash)
046	056	02E	00101110	. (dot)
047	057	02F	00101111	/ (forward slash)
048	060	030	00110000	0
049	061	031	00110001	1
050	062	032	00110010	2
051	063	033	00110011	3
052	064	034	00110100	4
053	065	035	00110101	5
054	066	036	00110110	6
055	067	037	00110111	7
056	070	038	00111000	8
057	071	039	00111001	9
058	072	03A	00111010	: (colon)
059	073	03B	00111011	; (semi-colon)
060	074	03C	00111100	< (less than)
061	075	03D	00111101	= (equal sign)
062	076	03E	00111110	> (greater than)

Decimal	Octal	Hex	Binary	Value
063	077	03F	00111111	? (question mark)
064	100	040	01000000	@ (AT symbol)
065	101	041	01000001	A
066	102	042	01000010	B
067	103	043	01000011	C
068	104	044	01000100	D
069	105	045	01000101	E
070	106	046	01000110	F
071	107	047	01000111	G
072	110	048	01001000	H
073	111	049	01001001	I
074	112	04A	01001010	J
075	113	04B	01001011	K
076	114	04C	01001100	L
077	115	04D	01001101	M
078	116	04E	01001110	N
079	117	04F	01001111	O
080	120	050	01010000	P
081	121	051	01010001	Q
082	122	052	01010010	R
083	123	053	01010011	S
084	124	054	01010100	T
085	125	055	01010101	U
086	126	056	01010110	V
087	127	057	01010111	W
088	130	058	01011000	X
089	131	059	01011001	Y
090	132	05A	01011010	Z
091	133	05B	01011011	[(left/opening bracket)
092	134	05C	01011100	\ (back slash)
093	135	05D	01011101	] (right/closing bracket)
094	136	05E	01011110	^ (caret/cirumflex)
095	137	05F	01011111	_ (underscore)
096	140	060	01100000	`
097	141	061	01100001	a
098	142	062	01100010	b
099	143	063	01100011	c
100	144	064	01100100	d
101	145	065	01100101	e
102	146	066	01100110	f
103	147	067	01100111	g
104	150	068	01101000	h
105	151	069	01101001	i
106	152	06A	01101010	j
107	153	06B	01101011	k
108	154	06C	01101100	l
109	155	06D	01101101	m
110	156	06E	01101110	n
111	157	06F	01101111	o
112	160	070	01110000	p
113	161	071	01110001	q
114	162	072	01110010	r
115	163	073	01110011	s
116	164	074	01110100	t
117	165	075	01110101	u
118	166	076	01110110	v
119	167	077	01110111	w
120	170	078	01111000	x
121	171	079	01111001	y
122	172	07A	01111010	z
123	173	07B	01111011	{ (left/opening brace)
124	174	07C	01111100	| (vertical bar)
125	175	07D	01111101	} (right/closing brace)
126	176	07E	01111110	~ (tilde)
127	177	07F	01111111	DEL (delete)

Dodatak H: Česte greške prilikom programiranja

Prilikom pisanja asemblerskih programa se mora voditi računa o tome da se kod takvog programiranja pristupa hardveru na veoma niskom nivou i da asemblerski jezik nema mogućnost provere raznih tipova grešaka koje mogu biti prijavljene prilikom rada sa programskim jezicima višeg nivoa.

Tokom godina uočeno je da studenti na vežbama često prave neke tipične greške prilikom pisanja asemblerskih programa. U ovom poglavlju će biti reči o ovim greškama, kao i o greškama koje mogu nastati prilikom korišćenja alata iz komandne linije.

Editovanje jednog, a prevođenje drugog programa – program "ne radi kako treba"

Pretpostavimo da postoje direktorijumi `~/vezba1/` i `~/vezba2/` i da se u oba nalazi fajl `program.S`. Ukoliko korisnik ima u editoru otvoren fajl `~/vezba1/program.S`, a komandna linija za prevođenje se nalazi u `~/vezba2/`, doći će do situacije da se program menja, ali "ne radi kako treba". Prilikom rada sa alatima iz komandne linije treba obratiti pažnju na to da se komande izvršavaju u ispravnoj putanji.

Prilikom prevođenja nije naveden naziv fajla iza `-o`

Primer ispravnog pozivanja kompajlera uz korišćenje opcije `-o` je:

```
gcc -g -o program program.S
```

Međutim, ako se kompajler pozove sa

```
gcc -g -o program.S program
```

ono što će se desiti je da se pokuša prevođenje fajla `program` i da se rezultat tog prevođenja na disk upiše kao `program.S`, čime će efektivno izvorni kod biti obrisan. Da se izvorni kod ne bi izgubio, korisno je uvek ga držati otvorenog u editoru, kako bi se mogao ponovo snimiti na disk u slučaju ovakve greške (editor je obično podešen da čuva prethodnu verziju teksta programa kao `program.S~`).

Pokretanje dibagera ukoliko je bilo grešaka prilikom prevođenja

Ako je prevođenje programa prošlo bez greške, tada će izlaz nakon pozivanja `gcc`-a biti prazan i tada ima smisla pokrenuti program u dibageru. Međutim, ako je bilo grešaka prilikom prevođenja (odnosno, ako je bilo nekog izlaza prilikom pozivanja `gcc`-a), tada izvršni fajl nije korektan, ili čak i ne postoji. U takvoj situaciji nema smisla pokretati dibager, nego je prvo potrebno ispraviti grešku/greške.

Pokretanje dibagera sa navođenjem izvornog umesto izvršnog fajla

Ukoliko postoji `program.S` i njegov izvršni oblik `program`, tada se dibager poziva sa:

```
ddd program
```

Međutim, ako se dibager pozove sa

```
ddd program.S
```

prozor dibagera će biti prazan, pošto `program.S` nije izvršni fajl.

Nerazlikovanje znakova l (malo L)i 1, odnosno O (veliko o) i 0

Osobe koje se tek upoznaju sa sintaksom asemblerskog jezika ponekad ne naprave razliku između znakova "l" (malo slovo L) i 1 (cifra 1), a takođe i između "O" (veliko slovo o) i "0" (cifra 0):

```
mov1 $5, %eax
movb $O, %ah
```

Naravno, ovakva greška neće proći nezapaženo od strane `gcc`-a:

```
Error: no such instruction: `mov1 $5,%eax'
```

odnosno:

```
/usr/lib/gcc/i686-linux-gnu/4.4.5/../../../../lib/crt1.o: In
function `_start':
(.text+0x18): undefined reference to `main'
/tmp/cccund9p.o:(.text+0x1): undefined reference to `O'
collect2: ld returned 1 exit status
```

Smeštanje naredbi u sekciju podataka – program neće da se pokrene

Ukoliko se napiše program kod koga se kôd nalazi u sekciji podataka:

```
.section .text
.section .data
.globl main
main:
a    .long 1
b    .long 2
movl a, %eax
addl b, %eax
```

prilikom prevođenja se neće prijaviti nikakva greška (pošto je u asembleru ovako nešto dozvoljeno uraditi), ali se nakon prevođenja takav program neće moći pokrenuti, pošto nema ni jednu naredbu u sekciji kôda.

Smeštanje promenljivih u sekciju kôda

Ukoliko se napiše program kod koga se promenljive nalaze u sekciji kôda:

```
.section .data
.section .text
.globl main
main:
a    .long 1
b    .long 2
movl a, %eax
addl b, %eax
```

prilikom prevođenja se neće prijaviti nikakva greška (pošto je u asembleru ovako nešto dozvoljeno uraditi), ali će se nakon prevođenja i pokretanja takvog programa najverovatnije javiti greška *Segmentation fault*, pošto će se pokušati izvršavanje ili menjanje naredbi nastalih prevođenjem podataka.

U programu ne postoji `.globl main` ili `main:`

Ukoliko u programu ne postoji direktiva `.globl main`, ili ako nema labele `main:`, prilikom prevođenja linker će prijaviti grešku:

```
/usr/lib/gcc/i686-linux-gnu/4.4.5/../../../../lib/crt1.o: In function
`_start':
(.text+0x18): undefined reference to `main'
collect2: ld returned 1 exit status
```

Korišćenje konstante bez znaka $

Ukoliko je u programu korištena konstanta bez znaka $, na primer:

```
movl 5, %eax
```

prilikom izvršavanja te linije, desiće se greška *Segmentation fault*, pošto je program pokušao da čita sa lokacije 5 (direktno adresiranje), umesto da radi sa konstantom (neposredno adresiranje).

Dibager se "zaglavio" - beskonačna petlja

Ukoliko u programu postoji beskonačna petlja (ili petlja koja se veoma dugo izvršava), može izgledati kao da dibager ne radi:

```
petlja:
     movl $0, %ecx
     ...
     incl %ecx
     cmpl $10, %ecx
     jne petlja
```

Međutim, ukoliko se pogleda donji desni ugao, može se videti trepćući zeleni kvadrat koji označava da se program trenutno izvršava (takođe se na trenutke može videti i strelica koja označava tekuću naredbu). Program koji je upao u beskonačnu petlju se može prekinuti komandom dibagera *Interrupt* (nalazi se na panelu sa komandama), pa se može izvršavati dalje korak po korak kako bi se otkrilo šta se u stvari desilo.

Istovremeno korišćenje registara koji se preklapaju za različite namene

Razmotrimo situaciju u kojoj imamo program koji nešto računa sa registrom `eax`, i još istovremeno koristi, na primer, registar `ax` za smeštanje neke provremene vrednosti:

```
     movl $0, %eax
     movw $1, %ax
     incl %eax
     cmpl $50, %eax
     jne L1
     movl $2, %ax
L1:
     ...
```

Čim se u `ax` upiše nova privremena vrednost, time će se pokvariti i registar `eax`, odnosno rezultat računanja. Takođe, ako se promeni vrednost registra `eax`, to se automatski odražava i na `ax`. U najvećem broju slučajeva ne treba istovremeno koristiti i 32-bitni/16-bitni registar i njegove niže delove (ukoliko postoje).

Korišćenje registra `ebp`, odnosno `esp` za smeštanje podataka

Može se napisati asemblerski program koji će registar `esp`, odnosno `ebp` koristiti za smeštanje podataka i koji će raditi kad se pokrene iz dibagera. Međutim, ova dva registra imaju i posebnu namenu prilikom rada sa stekom. Pošto je jedan od ciljeva ovog kursa pisanje asemblerskih potprograma koji se mogu uključiti u programe napisane u višim programskim jezicima, ovakva upotreba ovih registara samo može dovesti do neispravnog programa.

Prebacivanje svih argumenata u registre odmah na početku potprograma

Razmotrimo slučaj potprograma sa dva parametra koji se prenose po vrednosti i trećeg koji se prenosi po adresi, namenjenog za smeštanje rezultata. Pošto pristup trećem argumentu podrazumeva neku varijantu indirektnog adresiranja, on se pre korišćenja mora smestiti u registar. Pretpostavimo da su sva tri argumenta na početku potprograma smeštena u registre. Ukoliko je proračun složeniji, lako se može desiti da će u potprogramu ponestati slobodnih registara. U ovakvoj situaciji, smeštanje adrese rezultata u registar odmah na početku potprograma je pogrešno, pošto je ta adresa potrebna tek na kraju, kad se proračun završi i nema potrebe da zauzima registar tokom rada glavnog dela potprograma. Generalno pravilo bi bilo da se u registre smeštaju one vrednosti koje se najčešće koriste u potprogramu, i to neposredno pre njihove upotrebe. Ako je neki argument prenet po vrednosti, a koristi se na svega par mesta, treba videti da li se može koristiti direktno sa steka, bez smeštanja u registar.

Sekcija za kôd/podatke nazvana `text`/`data` (bez tačke)

Tačka u nazivima asemblerskih direktiva je obavezan deo imena, kao i bilo koje drugo slovo. Ako se izostavi, prevodilac će smatrati da je korisnik hteo da napravi sopstvenu sekciju sa podacima (što je dozvoljeno) pod tim imenom. Međutim, sekcije `.text` i `.data` su

podrazumevane sekcije prilikom pokretanja programa, i ako ne postoje, program se neće moći izvršavati kako je zamišljeno.

Nepotrebni skokovi

Razmotrimo nekoliko primera nepotrebnih skokova:

```
     cmpl $0, %eax
     je L1
L1:
     incl %eax
     ...
```

U gornjem primeru, ukoliko je uslov tačan, program će se izvršavati od naredbe `incl %eax`, a ako nije tačan, opet od te iste naredbe, odakle sledi da je skok suvišan (ili algoritam nije dobar).

```
     movl %eax, %ebx
     jmp L1
L1:
     incl %eax
     ...
```

U gornjem primeru postoji bezuslovni skok na naredbu odmah iza skoka, što je nepotrebno, pošto bi se ta naredba svakako sledeća izvršila i da nema bezuslovnog skoka.

```
     cmpl $0, %eax
     je L2
L1:
     incl %eax
     ...
L2:
     addl $5, %eax
     jmp L1
```

U gornjem primeru se privremeno iskače iz sekvencijalnog toka programa da bi se obavile neke naredbe, pa se vraća na mesto odmah ispod uslovnog skoka. Ovakva sekvenca naredbi se može mnogo razumljivije napisati korišćenjem suprotnog uslovnog skoka:

```
     cmpl $0, %eax
     jne L1
     addl $5, %eax
L1:
     incl %eax
     ...
```

Loše iskomentarisan kôd / odsustvo komentara

Pošto svi asemblerski programi veoma liče jedan na drugi na prvi pogled, ukoliko nisu iskomentarisani kako treba, veoma lako se posle nekog vremena može zaboraviti njihova namena, pa treba potrošiti vreme kako bi se otkrilo “šta je pisac hteo da kaže” (čak i kad je pisac ta ista osoba). Komentari u kôdu treba da u kratkim crtama opišu pojedine celine u programu, ili mesta za koja se smatra da nisu jasna na prvi pogled (obično nema smisla komentarisati svaku liniju). Komentari ne treba da opisuju očigledne stvari. Loši komentari samo mogu da doprinesu konfuziji prilikom kasnije analize.

```
#loše iskomentarisan kod
main:                                   #početak
     movl $niz, %esi                    #adresa od niz u esi
     movl $NELEM-1, %edi                #nelem-1 u edi
     movl $0, %eax                      #nula u eax
petlja:                                 #početak petlje
     addl (%esi, %edi, 4), %eax         #dodaj (%esi, %edi, 4) na eax
     decl %edi                          #smanji edi za 1
     jns petlja                         #skoči ako nije s
kraj:                                   #kraj programa
     movl %eax, suma                    #stavi eax u sumu
     movl $1, %eax                      #stavi 1 u eax
     movl $0, %ebx                      #stavi 0 u ebx
     int $0x80                          #pozovi int sa $0x80
```

Komentari u gornjem primeru ni na koji način ne doprinose razumevanju programa. Vrlo lako se može ispostaviti da je za početnike u asemblerskom programiranju prilično teško da pogode šta je svrha programa bez malo većeg udubljivanja u značenje asemblerskih naredbi. Jedan primer dobrog komentarisanja istog kôda je dat u sledećem listingu:

```
#dobro iskomentarisan kod
#program za računanje sume niza, varijanta 3
main:
     movl $niz, %esi                    #registar za bazu
     movl $NELEM-1, %edi                #indeks poslednjeg elementa
     movl $0, %eax                      #trenutna suma
petlja:                                 #od poslednjeg ka prvom elementu
     addl (%esi, %edi, 4), %eax
     decl %edi                          #kad se desi edi<0, s postaje 1
     jns petlja
kraj:
     movl %eax, suma                    #smeštanje sume u zadatu promenljivu
     movl $1, %eax                      #exit sistemski poziv
     movl $0, %ebx                      #nije bilo grešaka pri izvršavanju
     int $0x80
```

(Korisna) Literatura

1. *Intel 80386 Programmer's Reference Manual*, 1986
2. *Linux Assembly*, linuxassembly.org
3. *Programming from the Ground Up*, *Jonathan Bartlett*, 2003
4. *Professional Assembly Language*, *Richard Blum*, 2005
5. *The Linux Documentation Project*, www.tldp.org
6. *Linux for Programmers and Users*, *Graham Glass*, *King Ables*, 2006
7. *GNU Bash Manual*, www.gnu.org/software/bash/manual/bash.html
8. *Advanced Bash-Scripting Gude*, *Mendel Cooper*, 2002
9. *DDD - Data Display Debugger Manual*, www.gnu.org/manual/ddd/
10. *The Art of Computer Programming Volume 1: Fundamental Algorithms*, Addison-Wesley, 1997
11. *Write Great Code Volume 1: Understanding the Machine*, Randall Hyde, 2004
12. *Write Great Code Volume 2: Thinking Low-Level, Writing High-Level*, Randall Hyde, 2006

Indeks pojmova

Indeks slika

Indeks tabela