3.3 FORMATI ZAPISIVANJA INSTRUKCIJA BEZUSLOVNOG I USLOVNOG SKOKA

Format zapisivanja instrukcija bezuslovnog skoka *j* i *jal*. Podsjetimo da je simbolički kod zapisivanja instrukcija bezuslovnog skoka *j* i *jal* upotrebljavanih u MIPS asemblerskom kodu sljedeći:

j*/*jal **Address**

gdje Address predstavlja adresu instrukcije na koju se bezuslovno prelazi/skače (kod instrukcije *j*) ili adresu procedure/potprograma (adresu prve instrukcije procedure/potprograma) na koju se bezuslovno prelazi/skače. Dakle, prilikom zapisivanja mašinskog koda ovih instrukcija, potrebno je zapisati ***obilježja*** samo dva njihova simbolička dijela: ključne riječi/naziva instrukcije (*j*/*jal*) i Address-u.

Podsjetimo takodje se da su do sada definisana dva formata zapisivanja instrukcija: R-tip sa 6 polja za zapisivanje *obilježja* pojedinih simboličkih djelova instrukcije i I-tip sa 4 polja za zapisivanje *obilježja* pojedinih simboličkih djelova instrukcije. Pošto, kao što je gore već primijećeno, instrukcije bezuslovnog skoka *j* i *jal* zahtijevaju zapisivanje samo *obilježja* dva njihova simbolička dijela, R-tip i I-tip nijesu odgovarajući formati za zapisivanje ovih instrukcija. Stoga se uvodi novi (do sada nerazmatrani) format zapisivanja, nazvan **J-tip** (od Jump – po instrukcijama za čije zapisivanje se upotrebljava). Format mašinskog zapisivanja instrukcija *j* i *jal* grafički je prikazan na slici 6.

Polje:	op	address
Br. bitova:	6	26
Sadržaj:	2, 3	Address

Slika 6. Format mašinskog zapisivanja instrukcija bezuslovnog skoka *j* (za op=2) i *jal* (za op=3).

Primijetiomo da J-tip zapisivanja sadrži dva polja. 6-bitno op polje (Opcode od engleskih riječi *Operation Code*) upotrebljava se za mašinsko zapisivanje (odgovarajući niz jedinica i nula) ključne riječi/naziva instrukcije (op=2 za istrukciju *j*, op=3 za instrukciju *jal*), dok se 26-bitno address polje upotrebljava za zapisivanje adrese instrukcije, odnosno adrese procedure na koju se bezuslovno prelazi prilikom implementacije instrukcija *j* i *jal*, respektivno.

Format zapisivanja instrukcija uslovnog skoka/grananja *beq* i *bne*. Podsjetimo da je simbolički kod zapisivanja instrukcija uslovnog skoka/grananja *beq* i *bne* upotrebljavanih u MIPS asemblerskom kodu sljedeći:

beq*/*bne **$x, $y, L**

gdje $x i $y, x,y=0, 1, …, 31, predstavljaju obilježja registara čiji sadržaji se porede u odnosu na jednakost (kod instrukcije *beq*), odnosno u odnosu na nejednakost (kod instrukcije *bne*), dok je L obilježje (adresa) instrukcije na koju se prelazi/skače ukoliko je uslov jednakosti, odnosno nejednakosti zadovoljen. Dakle, prilikom zapisivanja mašinskog koda ovih instrukcija, potrebno je zapisati *obilježja* četiri njihova simbolička dijela: ključne riječi/naziva instrukcije (*beq*/*bne*), 2 registra ($x i $y) i obilježja L instrukcije na koju se uslovno prelazi/skače. Primijetimo da ovaj zahtjev ispunjava I-format mašinskog zapisivanja instrukcija, slično kao kod data-transfer instrukcija *lw* i *sw* i immediate instrukcija (*addi*, *muli*, *slti*, …). Format mašinskog zapisivanja instrukcija uslovnog skoka/grananja *beq* i *bne* grafički je prikazan na slici 7.

Polje:	op	rs	rt	immediate
Br. bitova:	6	5	5	16
Sadržaj:	4, 5	x	y	L

Slika 7. Format mašinskog zapisivanja instrukcija uslovnog skoka/grananja *beq* (za op=4) i *bne* (za op=5).

Primijetimo da se obilježje registra $x zapisuje u polju rs, a obilježje registra $y u polju rt I-formata mašinskog zapisivanja instrukcija, odnosno istim redom kojim se navode u simboličkoj/asemblerskoj formi instrukcija *beq*/*bne* (ove instrukcije ne uključuju rezultujući registar,

već samo operande koji se porede u odnosu na jednakost/nejednakost), dok bi obilježje L instrukcije na koju se uslovno prelazi/skače trebalo zapisati u 16-bitnom immediate polju.

Medjutim, 16-bitno immediate polje je više nego nedovoljno za zapisivanje memorijske adrese (obilježja L) na koju se uslovno prelazi/skače. Naime, 16-bitno immediate polje omogućava zapisivanje 2^{16} različitih memorijskih adresa ($0 \div (2^{16}-1)_{(10)}$), što je samo $(1/2^{16})$-ti, odnosno oko 65.5-hiljaditi dio ukupne memorije računara od 2^{32} lokacija ($0 \div (2^{32}-1)_{(10)}$), koja se može adresirati upotrebom instrukcije *jr*, kada se za zapisivanje adrese upotrebljava 32-bitni sadržaj registra. Sa druge strane, mora se obezbijediti adresiranje cjelokupne memorije prilikom implementiranja instrukcija uslovnog skoka/grananja. Drugim riječima, mora se pronaći način da se opisana situacija prevazidje.

U tom cilju, podsjetimo najprije da se instrukcije uslovnog skoka/grananja upotrebljavaju prilikom kreiranja if-upita i petlji. Uz to, primijetimo da if-upiti i petlje, po pravilu, ne zauzimaju značajne memorijske opsege (gotovo uvijek značajno manje memorijske opsege od 2^{16} lokacija). Sa druge strane, u PC registru nalazi se adresa trenutno izvršavane instrukcije. Drugim riječima, ukoliko se trenutno izvršava instrukcija uslovnog skoka/grananja *beq*/*bne*, te ukoliko se u 16-bitnom immediate polju mašinskog zapisa instrukcije *beq*/*bne*, slika 7, NE upiše adresa instrukcije na koju se uslovno prelazi/skače, već relativna udaljenost ove instrukcije u odnosu na tekući sadržaj PC registra, distanca uslovnog grananja/skoka može biti u opsegu od 2^{16} lokacija. Kao što je rečeno, ova distanca je, po pravilu, više nego dovoljna za kreiranje if-upita i petlji. Takodje, adresa instrukcije na koju se uslovno prelazi/skače dobija se sumom tekućeg sadržaja PC registra i distance grananja, tako da se na ovaj način omogućava kreiranje proizvoljne memorijske adrese (iz cjelokupnog memorijskog opsega), samo da je ta adresa iz opsega od 2^{16} od adrese lokacije na kojoj je zapisana instrukcija uslovnog skoka/grananja. Iz navedenih razloga, ovaj način adresiranja naziva se ***PC relativnim adresiranjem***.

ZAKLJUČAK: PC-relativno adresiranje upotrebljava se prilikom implementacije instrukcija uslovnog skoga/grananja *beq* i *bne*, pošto je nastovanje da buduće destinacije programa, nakon izvršavanja ovih instrukcija, ne budu udaljene više od 2^{16} lokacija.

NAPOMENA: Prilikom razmatranja poziva procedura i povratnog adresiranja, navedeno je da se sadržaj PC registra inkrementira za 4 (PC←PC+4) u samom početku izvršavanja svih MIPS instrukcija (time i instrukcija uslovnog skoka/grananja *beq* i *bne*). Drugim riječima, u trenutku odlučivanja da li će se izvršiti skok, i na koju adresu, prilikom implementacije instrukcija uslovnog skoka/grananja, u PC registru nalazi se sadržaj PC+4, odnosno adresa instrukcije koja, u memoriji računara, neposredno slijedi nakon instrukcije uslovnog skoka/grananja. Ova činjenica se mora uzeti u obzir prilikom odredjivanja relativne adrese L kod implementacije instrukcija *beq* i *bne*, tako da adresa buduće instrukcije (na koju se uslovno prelazi/skače nakon izvršavanja instrukcije *beq*/*bne*) treba da bude odredjena u odnosu na instrukciju koja se nalazi neposredno nakon instrukcije *beq*/*bne*, a ne u odnosu na samu instrukciju *beq*/*bne*, odnosno obilježje L treba da odgovara (PC+4)-realativnoj adresi, a ne PC-relativnoj adresi.

Primjer 16. Posmatrajmo ranije zapisani asemblerski kod while petlje:

```
Loop:  mult   $9, $19, $10
       lw     $8, Sstart($9)
       bne    $8, $21, Exit
       add    $19, $19, $20
       j      Loop
Exit:  .....
```

Napisati mašinski kod prethodno zapisane petlje, pretpostavljajući da je 80000 memorijska adresa obilježja Loop, a da je Sstart=1000.

Rješenje: Mašinski kod gore zapisane while petlje ima sljedeći izgled:

Add.	*Op*	*Rs*	*Rt*	*Rd*	*Shamt*	*Funct*
				Address/Immediate		
80000	0	19	10	9	0	24
80004	35	9	8	1000		
80008	5	8	21	8		
80012	0	19	20	19	0	32
80016	2	80000				
80020	.....					

Primijetimo najprije da se adrese susjednih instrukcije u memoriji računara razlikuju za 4, pošto je svaka od instrukcija zapisana u odgovarajućoj lokaciji, a adrese memorijskih lokacija se razlikuju za 4 byte-a (kod MIPS arhitektura, memorija je byte-adresabilna, a instrukcije i memorijske lokacije su 32-bitne). Koncentrišimo se, ipak, na zapisivanje mašinskog oblika insttrukcija *bne* i *j* (mašinske forme zapisivanja ostalih upotrijebljenih instrukcija R-tipa (*mult*, *add*) i data-transfer instrukcije *lw* ranije su razmatrane).

Opcode instrukcije *bne* je op=5, kako je već navedeno na slici 7, dok su 8 i 21 obilježja registara (operanada), čiji sadržaje se porede u odnosu na nejednakost. Brojna vrijednost 8 koja je zapisana u Immediate polju mašinskog koda instrukcije *bne* odgovara (PC+4)-relativnoj adresi obilježja Exit=80020. Naime, primijetimo da adresa 80020 odgovara obilježju Exit upotrijebljenom prilikom simboličkog/asemberskog zapisivanja while petlje, te da ova adresa ne može biti zapisana u 16-bitnom Immediate polju (maksimalna vrijednost koja u ovom polju može biti zapisana je $2^{16}-1=65535_{(10)}$). Prilikom izvršavanja instrukcije *bne*, u trenutku odredivanja adrese instrukcije koja sljedeća treba da se izvrši (nakon izvršavanja *bne* instrukcije), u PC registru će već biti upisana adresa 80012 (adresa instrukcije koja se nalazi u sljedećoj memorijskoj lokaciji upisuje se u PC registar na početku izvršavanja instrukcije *bne*). Brojna vrijednost 8 predstavlja **udaljenost** ili **rastojanje** (eng. *offset*) ciljne adrese 80020 (obilježja Exit) u odnosu na tekući sadržaj PC registra (80012) i 8 se upisuje u 16-bitno Immediate polje mašinskog koda instrukcije *bne*. Pojednostavljeno, 8 je (PC+4)-relativna adresa od adrese 80020, gdje je PC=80008 adresa instrukcije uslovnog skoka/grananja *bne*. Napomenimo da se isti postupak odredjivanja (PC+4)-relativne adrese upotrebljava kod svih instrukcija uslovnog skoka/grananja (*beq* i *bne*).

NAPOMENA 2: Prilikom mašinskog zapisivanja instrukcije *j*, za zapisivanje adrese instrukcije na koju se beuslovno prelazi/skače, raspolaženo sa 26-bitnim address poljem, slika 6.. Drugim riječima, prilikom zapisivanja *j* instrukcije raspolaže se sa 10 bitova više u odnosi na Immediate polje kojim raspolažu instrukcije uslovnog skoka/grananja *beq* i *bne*, tako da nema potrebe za PC relativnim adresiranjem, već se adresa instrukcije na koju se bezuslovno prelazi skače direktno zapisuje u 26-bitno address polje J-formata zapisivanja.

NAPOMENA 3: Prilikom razmatranja PC-relativnog adresiranja rečeno je da se njime vrši adresiranje kod instrukcija uslovnog skoka/grananja (*beq* i *bne*) i da se njime rješava problem adresiranja u slučajevima kada se adresa instrukcije na koju se uslovno prelazi skače nalazi na udaljenosti **ne većoj od 2^{16}** od adrese instrukcije uslovnog skoka/grananja. *Medjutim, postavlja se pitanje kako riješiti i da li je uopšte moguće riješti problem ukoliko je udaljenost izmedju ovih instrukcija veće od* 2^{16} *adresa, odnosno, pojednostavljeno, ukoliko se* (PC+4)*-relativna adresa instrukcije na koju se prelazi/skače ne može zapisati u 16-bitnom Immediate polju I-formata zapisivanja instrukcija beq i bne*?

Opisani problem moguće je riješiti implementiranjem komplementarne instrukcije uslovnog skoka/grananja (implementiranjem instrukcije *beq* umjesto *bne*, odnosno implementiranjem instrukcije *bne* umjesto *beq*) kojom se prelazi/skače, ukoliko je njen uslov granja zadovoljen, na veoma blisku memorijsku lokaciju, a u medjuvremenu, kada uslov grananja komplementarne instrukcije nije zadovoljen (primijetimo da tada mora biti zadovoljen uslov zadate instrukcije), vrši se prelazak/skok na zadatu adreasu i to implementacijom instrukcije bezuslovnog skoka *j*. Na ovaj način dobija se 10 bitova više za zapisivanje adrese instrukcije na koju se prelazi/skače (za zapisivanje adrese, na

raspolaganju je 26-bitno polje J-formata, umjesto 16-bitnog polja I-formata zapisivanja). Pojasnimo ovo rješenje na jednom prostom primjeru.

Primjer 17: Implementirati instrukciju uslovnog skoka/grananja

bne **$x, $y, L # skoči na instr. sa obilježjem L ako je $x≠$y**

pretpostavljajući da PC-relativna adresa (L) ne može biti smještena u 16-bitnom Immediate polju mašinskog koda instrukcije *bne* (instrukcija sa obilježjem L nalazi se na udaljenosti većoj od 2^{16} od adrese instrukcije uslovnog skoka/grananja *bne*).

Rješenje: Upotrijebimo najprije instrukciju komplementarnu zadatoj instrukciji *bne*, odnosno upotrijebimo instrukciju uslovnog skoka/grananja *beq*. Kada uslov grananja zadat instrukcijom *beq* nije zadovoljen (u tom slučaju, uslov zadat instrukcijom *bne* mora biti zadovoljen), izvršava se instrukcija bezuslovnog skoka na instrukciju sa obilježjem L,

```
        beq     $x, $y, L1 # skoči na instr. sa obilježjem L1 ako je $x=$y
        j       L          # skoči na instr. sa obilj. L (izvršava se za $x≠$y)
L1:     …
```

Analizirajmo prethodno zapisanim rješenje. Kako je u komentarima navedeno, prilikom izvršavanja instrukcije *beq* (komplementarne zadatoj instrukciji (*bne*)), uslovni skok na veoma blisku instrukciju (sa obilježjem L1) izvršava se ukoliko su identični sadržaji registara $x i $y. Ukoliko se sadržaji registara $x i $y razlikuju (odnosno, ukoliko nije ispunjen uslov instrukcije *beq*) izvršava se instrukcija koja neposredno slijedi nakon instrukcije uslovnog skoka/grananja *beq*, odnosno izvršava se beuslovni skok na instrukciju sa obilježjem L (implementiran instrukcijom: *j* L), što je, primijetimo, suštinski cilj zadate instrukcije *bne* $x, $y, L (vidjeti komentar zapisan uz ovu instrukciju). Primijetimo da ovim rješenjem na raspolaganju imamo 26-bitno polje za zapisivanje adrese L instrukcije na koju se prelazi/skače (odnosno 10 bitova više u odnosu na 16-bitno Immediate polje naredbe *bne*). Primijetimo, takodje, da je PC-relativna adresa (L1), neophodna prilikom zapisivanja mašinskog koda instrukcije *beq*, iznosi 4, odnosno PC-relativna adresa (L1)=4, pošto je 4 razlika/distanca izmedju adrese instrukcije koja neposredno slijedi nakon instrukcije *beq* i instrukcije sa obilježjem L1. Na koncu, notirajmo i da je instrukcija sa obilježjem L1 – instrukcija koja u memoriji računara slijedi neposredno nakon zadate naredbe *bne* (instrukcija koja se izvršava ukoliko su sadržaji registara $x i $y jednaki, odnosno ukoliko nije ispunjen uslov zadat instrukcijom *bne*).

NAPOMENA 4: Gore navedeno rješenje implementira se prilikom assembliranja, na slučan način kao u slučaju kreiranje velikih adresa/konstanti koje ne mogu biti zapisane u 16-bitnom Immediate polju instrukcija I-tipa, odnosno prilikom kreiranja pseudoinstrukcija *move* i *blt*.

NAPOMENA 5: Ukoliko nije dovoljno ni 26-bitno polje za smještanje adrese L instrukcije na koju se prelazi/skače, adresa L se može smjestiti u proizvoljni privremeni registar (recimo, $x), a potom upotrijebiti instrukcija *jr* $x (umjesto instrukcije *j* L), čime će se imati na raspolaganju makimalna 32 bita za zapisivanje adrese L (6 bitova više u odnosu na 26-bitno polje instrukcije *j*).

MIPS načini adrsiranja

Rekapitulirajmo, na jednom mjestu, četiri razmatrana načina adresiranja:

1. **Adresiranje registra.** Operand je sadržan u registru, a njegovo adresiranje vrši se sadržajem odredjenih polja instrukcije (polja rs i rt kod instrukcija R-tipa i instrukcija uslovnog skoka/grananja *beq* i *bne* i polja rs kod immediate instrukcija),
2. **Osnovno (displacement) adresiranje.** Operand je zapisan u memoriji računara, na lokaciji čija adresa se dobija sumom sadržaja indeks registra (obilježje ovog registra nalazi se u polju rs instrukcije) i adrese sačuvane u 16-bitnom adresnom polju instrukcije, Address=Reg(rs)+Instruction[15–0]. Operand se najprije mora premjestiti (eng. *displacement*) u registar označen poljem rt instrukcije, pa tek potom upotrijebiti u instrukcijama R-tipa, *beq*, *bne* ili immediate instrukcijama. Ovaj način adresiranja upotrebljava se kod memory-reference instrukcija *lw* i *sw*,

3. **Trenutno (immediate) adresiranje.** Operand je konstanta sačuvana u samoj instrukciji (immediate polju instrukcije). Operand je trenutno dostupan, po čemu je ovaj način adresiranja dobio svoj naziv. Upotrebljava se kod immediate instrukcija,
4. **PC-relativno adresiranje.** Adresira se instrukcija zapisana u memoriji računara, na lokaciji čija adresa se dobija sumom trenutnog sadržaja PC registra i konstante koja označava distancu (rastojanje ili ofset) zapisano u 16-bitnom immediatte polju instrukcije, Address=PC+ofset. Ovaj način adresiranja upotrebljava se kod instrukcija uslovnog skoka/grananja *beq* i *bne*.

NAPOMENA 6: Intrukcijama bezuslovnog i uslovnog skoka, prelazi se (skače se) na instrukcije koje sljedeće trebaju da se izvrše. Sve instrukcije (time i instrukcije na koje se prelazi/skače) zapisane su u pojedinačnim lokacijama memorije računara. Drugim riječima, prilikom adresiranja instrukcija, navode se adrese odgovarajućih lokacija u kojima su instrukcije zapisane. Sa druge strane, binarne adrese memorijskih lokacija, za razliku od adresa ostalih byte-ova, završavaju se sa $00_{(2)}$ (adresa prve memorijske lokacije je $0_{(10)}$=00...$000_{(2)}$, a adresa svake sljedeće memorijske lokacije dobija se dodavanje $4_{(10)}$=$100_{(2)}$ na adresu prethodne memorijske lokacije, tako da se sa $00_{(2)}$ moraju završavati binarne adrese svih memorijskih lokacija). Stoga, kada već obavezno moraju da postoje, ove dvije nule mogu biti *implicitno upamćene*, NE i zapisivane, prilikom zapisivanja (u mašinskom obliku) adrese instrukcije na koju se prelazi/skače kod instrukcija uslovnog i bezuslovnog skoka. Na taj način, u 16-bitnom Immediate polju instrukcija uslovnog skoka/grananja *beq* i *bne*, biće zapisana 18-bitna adresa, uz napomenu da se posljednje dvije nule ne zapisuju već se implicitno pamte (računa se sa njihovim postojanjem). Takodje, na isti način, u 26-bitnom address polju instrukcija bezuslovnog skoka *j* i *jal*, biće zapisana 28-bitna adresa, uz napomenu da se posljednje dvije nule ne zapisuju već se implicitno pamte (računa se sa njihovim postojanjem). Naravno, dvije nule koje nijesu zapisane prilikom kreiranja mašinskog koda instrukcija bezuslovnog i uslovnog skoka/grananja, formalno moraju biti zapisane (dodate) prilikom formiranja adrese instrukcije (u PC registru) koja sljedeća treba da se izvrši (na koju se prelazi/skače). One će biti dodate na veoma jednostavan način – pomeranjem sadržaja PC registra za 2 mjesta ulijevo (pomjeranjem/shift-ovanjem sadržaja registra ulijevo, napušteni bitovi registra popunjavaju se nulama, tako da se, nakon pomjeranja/shift-ovanja za dve mjesta ulijevo, napušteni bitovi registra popunjavaju sa dvije nule).

NAPOMENA 7: 28-bitna adresa, kreirana na prethodno opisani način prilikom formiranja adrese instrukcije na koju se prelazi skače implementacijom instrukcije *j*, zapisuje se u 32-bitnom PC registru. Medjutim, postavlja se pitanje što je sadržaj preostala 4 bita PC registra?

KONVENCIJA: ***j instrukcija popunjava (zamjenom prethodnog sadržaja)* 28 *nižih bitova PC registra, ostavljajući najviša 4 bita PC registra neprimijenjenim***. Posljedica toga je da loader i linker moraju strogo voditi računa da, prilikom učitavanja, programe učitaju u memoriju računara unutar granica adresiranja od 2^{28}=256 MB (oko 64 miliona instrukcija). U slučaju potrebe, ova granica se proširuje zamjenom *j* instrukcija sa *jr* instrukcijama, kada se na upotrebu (za adresiranje) stavljaju maksimalna 32 registra (vidjeti NAPOMENU 5).

3.4 PRIMJER CJELOKUPNO IZVRŠAVANOG PROGRAMA

Primjer 18. Napisati proceduru za zamjenu sadržaja dvije memorijske lokacije. Parametri procedure su početna adresa niza sačuvanog u memoriji računara i indeks elementa niza čiju zamjenu sa prvim sljedećim elementom niza treba izvršiti.

Rješenje: Prilikom zapisivanja programa u asemblerskoj formi, potrebno je izvršiti sljedeće korake:

1. Alokaciju registara,
2. Zapisivanje asemblerskog koda tijela procedure,
3. Čuvanje na stack i vraćanje sa stack-a sadržaja registara koji se modifikuju tokom izvršavanja tijela procedure.

Napomenimo da se čuvanje sadržaja registara na stack-u mora obaviti prije izvršavanja tijela procedure (u suprotnom, sadržaj ovih registara biće nepovratno izgubljen). Ipak, primijetimo da tek

zapisivanjem koda tijela procedure postajemo svjesni svih registara čiji sadržaj se mijenja tokom izvršavanja tijela procedure, odnosno svih registara čiji sadržaj je potrebno čuvati na stak-u. Stoga se najprije zapisuje asemblerski kod tijela procedure, ali uz ostavljanje prostora da se naknadno, ispred koda tijela procedure, dodaju instrukcije za čuvanje sadržaja registara na stack-u.

1. **Alokacija registara.** Naglasimo na početku da alokacija registara odgovara proceduri deklarisanja promjenljivih kod viših programskih jezika. Saglasno ranije razmatranim konvencijama za pozivanje procedura, parametri (ulazne varijable) procedure smještaju se u registre $4–$7, i to prvonavodjeni parametar u registar $4, drugonavodjeni parametar u registar $5, ... Parametri procedure, čije zapisivanje se traži, su početna adresa niza (označimo niz sa *v*) u memoriji računara i indeks elementa niza (označimo ga sa *k*). Shodno konvenijama za pozivanje procedura, navedeni parametri će biti zapisani (pronadjeni od strane pozvane procedure) u registrima $4 i $5, i to u registru $4 početna adresa niza *v* (adresa prvog elementa niza *v*) i $5←*k*.

   NAPOMENA: Zašto i kada kažemo da će parametri biti "zapisani", odnosno da će pozvana procedura "pronaći" parametre u registrima $4 i $5? Pozivajuća procedura *smješta*, odnosno *zapisuje* parametre procedure u registrima $4–$7 i stoga, s' aspekta pozivajuće procedure, kažemo da se parametri *zapisuju*, dok pozvana procedura *pronalazi* parametre, sa kojima će raditi, u ovim registrima ($4–$7).

2. **Kod tijela procedure.** Asemblerski kod tijela procedure, zapisan u nastavku, pojašnjen je komentarima dodatim uz svaku liniju koda:

   ```
   muli    $2, $5, 4      # $2←4×k (relativna udaljenost (u adresama) v[k] od v[0])
   add     $2, $4, $2     # $2←adresa (v[k])=adresa (v[0])+4×k
   lw      $15, 0($2)     # $15←v[k], jer se adresa (v[k]) nalazi u registru $2
   lw      $16, 4($2)     # $16←v[k+1], jer je adresa (v[k+1])=adresa (v[k])+4
   sw      $16, 0($2)     # $16←v[k], jer se adresa (v[k]) nalazi u registru $2
   sw      $15, 4($2)     # $15←v[k+1], jer je adresa (v[k+1])=adresa (v[k])+4
   ```

   NAPOMENA 1: Prvom upotrijebljenom instrukcijom (*muli*) pronalazimo relativnu udaljenost (u adresama) elementa niza *v*[*k*] u odnosu na prvi elemenat niza, *v*[0]. Naime, svaki elemenat niza *v* upisan je u odgovarajuću *memorijsku lokaciju*. Drugim riječima, elemenat niza *v*[*k*] udaljen je *k* memorijskih lokacija (odnosno 4×*k* adresa, pošto su memorije MIPS arhitektura byte-adresabilne, a svaka memorijska lokacija sadrži 4 byte-a) od prvog elementa niza, *v*[0]. Ovim su stvoreni uslovi da već narednom istrukcijom (*add*) pronadjemo adresu elementa niza *v*[*k*], sabiranjem adrese prvog elementa niza, pohranjene u registru $4, i relativne udaljenosti elementa niza *v*[*k*] u odnosu na prvi elemenat niza, *v*[0].

   NAPOMENA 2: Registri $15 i $16 upotrebljavaju se kao privremeni registri i u funkciji su obezbjedjivanja uslova za zamjenu sadržaja dvije memorijske lokacije. Naime, u MIPS asemblerskom kodu, direktna zamjena dvije memorijske lokacije, *v*[*k*]←*v*[*k*+1], odnosno *v*[*k*+1]←*v*[*k*], nije moguća, već se mora obaviti uz posredovanje privremenih registara (ovdje odabranih registara $15 i $16).

3. **Čuvanje/vraćanje sadržaja registara za vrijeme poziva procedure.** Upotrijebimo callee-save rutinu za čuvanje sadržaja registara upotrebljavanih u pozvanoj proceduri. U tom cilju, primijetimo da se tokom izvršavanja tijela procedure vrši izmjena sadržaja 3 registra: registra $2 (izvršavanjem instrukcija *muli* i *add*), registra $15 (izvršavanjem instrukcije ***lw* $15, 0($2)**) i registra $16 (izvršavanjem instrukcije ***lw* $16, 4($2)**). Notirajmo, pak, da se izvršavanjem instrukcije ***sw* $16, 0($2)**, odnosno instrukcije ***sw* $15, 4($2)** ne mijenjaju sadržaji registara $16 i $15, respektivno, već se navedenim instrukcijama sadržaj ovih registara samo kopira na odgovarajuće memorijske lokacije.

   Drugim riječima, sadržaji registara $2, $15 i $16 moraju biti sačuvani (od prebrisavanja) na stack-u prije izvršavanja tijela procedure i to implementacijom sljedećih koraka:

   - Obezbjedjivanjem mjesta na vrhu stack-a za smještanje sadržaja navedena 3 registra,

```
addi    $29, $29, –12    # Stack se uvećava za 3 lokacije (odnosno za 3×4=12 byte-a)
```

- Čuvanjem sadržaja navedena 3 registra u odgovarajućim dodatim lokacijama na vrhu stack-a,

```
sw      $2, 0($29)
sw      $15, 4($29)
sw      $16, 8($29)
```

Nakon izvršavanja tijela procedure, sadržaje registara sačuvane na stack-u potrebno je vratiti na odgovarajuće destinacije (u odgovarajuće registre), te pokazivač/pointer stack-a vratiti na staru vrijednost. Ovo se obavlja instrukcijama komplementarnim prethodno zapisanim instrukcijama za čuvanje sadržaja registara na stack-u,

```
lw      $2, 0($29)
lw      $15, 4($29)
lw      $16, 8($29)
addi    $29, $29, 12     # Stack se umanjuje za 3 lokacije (odnosno za 3×4=12 byte-a)
```

Na koncu, sa pozvane procedure se vraćamo na njen nad-nivo instrukcijom

```
jr      $31
```

Kompletna procedura:

```
Swap: addi    $29, $29, –12
      sw      $2, 0($29)
      sw      $15, 4($29)
      sw      $16, 8($29)
      muli    $2, $5, 4
      add     $2, $4, $2
      lw      $15, 0($2)
      lw      $16, 4($2)
      sw      $16, 0($2)
      sw      $15, 4($2)
      lw      $2, 0($29)
      lw      $15, 4($29)
      lw      $16, 8($29)
      addi    $29, $29, 12
      jr      $31
```

VJEŽBE

1. Koristeći potprogram *Copy* (sa prethodnih vježbi), napisati program *Copy_text* u MIPS asembleru, koji će niz od 10 stringova smještenih jedan iza drugog u memoriji, počev od lokacije $1000_{(10)}$ kopirati na novo mjesto u memoriji, počev od lokacije $3000_{(10)}$.

   Rješenje:

   Potprogram *Copy* služi za kopiranje 1 stringa sa jedne na drugu memorijsku lokaciju. Pozivanjem ovog potprograma 10 puta u okviru programa *Copy_text* ćemo iskopirati 10 stringova sa jedne na drugu memorijsku lokaciju.
   Potrebno je, prije svakog pozivanja potprograma, obezbijediti ispravne ulazne argumente, a to su adrese izvorišnog i odredišnog stringa (u registrima $4 i $5). Takođe, potrebno je voditi računa da tačno 10 puta pozovemo potprogram.

```
Copy_text:  addi  $10, $0, 10      # $10=10
            addi  $4, $0, 1000     # početna adresa sa koje se čita
            addi  $5, $0, 3000     # početna adresa na koju se kopira

Loop:       jal   Copy             # pozivanje procedure

            addi  $4, $4, 1        # pomjeranje na sljedeći string
            addi  $5, $5, 1        # pomjeranje na sljedeći string
            addi  $10, $10, -1     # dekrementiramo za svaki poziv
                                   # procedure
            bne   $10, $0, Loop    # dok god je $10≠0, dozvoljavamo
                                   # rad petlje; kad postane $10=0 znači
                                   # da smo iskopirali svih 10 stringova
```

NAPOMENA: Zašto je dovoljno uvećanje sadržaja registara $4 i $5 za 1 prije novog poziva procedure? Pogledajte proceduru *Copy*. U okviru same procedure, prilikom kopiranja jednog po jednog elementa iz izvorišnog stringa u odredišni string, stižemo do kraja stringa, odnosno do njegovog terminatora. To znači da je prvi naredni element nakon terminatora izvorišnog stringa prvi element narednog stringa.

2. Napisati potprogram *bfind* u MIPS asembleru. Potprogram uzima jedan argument u registru $4, koji je pokazivač na string. Potprogram treba da locira prvi 'b' karakter u stringu, i vrati njegovu adresu u registru $2. Ako ne postoji 'b' karakter u stringu *bfind* treba da vrati pokazivač na terminacioni karakter stringa. Na primjer, ako je string 'imbibe', tada povratna vrijednost treba da bude pokazivač na treći karakter u stringu.

   Rješenje:

   Svaki karakter ima odgovarajući ASCII kod (cjelobrojnu vrijednost), te se prilikom rada sa konkretnim karakterima ponašamo kao da radimo sa konstantnim vrijednostima. Karakter 'b' ima ASCII kod 98.

```
bfind:  addi    $29, $29, -8    # podešavanje pokazivača na vrh steka
        sw      $8, 0($29)      # memorisanje sadržaja pomoćnog registra
        sw      $9, 4($29)      # memorisanje sadržaja pomoćnog registra

        addi    $8, $0, 'b'     # $8='b'; nije neophodno koristiti stvarnu vrijednost
                                # ASCII koda
start:  lb      $9, 0($4)       # učitavanje elementa stringa
        beq     $9, $8, Lab     # nađeno prvo 'b'
        beq     $9, $0, Lab     # stigli smo do kraja stringa
        addi    $4, $4, 1       # pomjeramo adresu za 1
        j       start           # dok god ne nadjemo prvo 'b' ili stignemo do kraja
                                # stringa petlja radi
Lab:    add     $2, $4, $0      # prosljeđivanje adrese karaktera 'b' ili terminatora

        lw      $8, 0($29)      # vraćamo u pomoćni registar njegov
                                # originalni sadržaj
        lw      $9, 4($29)      # vraćamo u pomoćni registar njegov
                                # originalni sadržaj
        addi    $29, $29, 8     # podešavanje pokazivača na vrh steka
        jr      $31             # vraćanje u glavni program
```

NAPOMENA: U ovom slučaju su nam bila potrebna 2 pomoćna registra, pa smo pokazivač na vrh steka pomjerali za 2*4=8 lokacija.

3. Napisati potprogram *bcount* u MIPS asembleru. Ovaj potprogram uzima jedan argument u registru $4, a to je pokazivač na string. Potprogram vraća broj pojavljivanja karaktera 'b' u stringu, preko registra $2. Iskoristiti potprogram *bfind* iz prethodnog zadatka.

Rješenje:

Potprogram *bfind* pronalazi lokaciju prvog karaktera 'b' u zadatom stringu. Možemo ga iskoristiti za prebrojavanje ukupnog broja karaktera 'b' u stringu tako što ćemo ga pozivati nakon svakog pronađenog karaktera 'b' i pri tom mu proslijeđivati, kao početnu adresu (onu od koje počinje da traži) adresu prvog karaktera nakon nadjenog karaktera 'b'. Potprogram *bcount* će završiti pretragu kada dođe do terminacionog karaktera. Na primjer, ako je posmatrani string 'imbibe', nakon prvog poziva potprograma *bfind* imaćemo adresu prvog karaktera 'b', a brojač pojavljivanja karaktera 'b' će imati vrijednost 1. Potom, prije nego što opet pozovemo *bfind*, kao početnu adresu (onu od koje počinje da traži) ćemo proslijediti adresu drugog karaktera 'i'. Tada će *bfind* naći adresu drugog karaktera 'b', a brojač pojavljivanja karaktera 'b' će imati vrijednost 2. Potom, prije nego što opet pozovemo *bfind*, kao početnu adresu ćemo proslijediti adresu karaktera 'e'. U ovom slučaju *bfind* neće pronaći novi karakter 'b', već će vratiti adresu terminatora, a brojač će zadržati vrijednost 2.

Potrebno je voditi računa o činjenici da potprogram *bcount* poziva potprogram *bfind*. U ovakvim situacijama (kada potprogram poziva potprogram) moramo sačuvati na steku sadržaj registra $31, kako bismo sačuvali povratnu adresu.

```
bcount: addi    $29, $29, -8    # podešavanje pokazivača na vrh steka
        sw      $8, 0($29)      # memorisanje sadržaja pomoćnog registra
```

```
        sw      $9, 4($29)      # memorisanje sadržaja pomoćnog registra

        add     $8, $0, $0      # brojač inicijalizovati na 0

Loop:   addi    $29, $29, -4    # podešavanje pokazivača na vrh steka
        sw      $31, 0($29)     # memorisanje povratne adrese iz bcount u glavni
                                # program prije nego pozovemo bfind, jer će nakon
                                # poziva bfind smjestiti svoju povratnu adresu u
                                # $31 i obrisati prethodni sadržaj
        jal     bfind           # poziv procedure
        lw      $31, 0($29)     # bfind je odradila posao, vraćamo u $31 ono što
        addi    $29, $29, 4     # je tamo i bilo, i podešavamo pokazivač na vrh steka

        lb      $9, 0($2)       # u $2 je adresa ili karaktera 'b' ili terminatora
                                # pa provjeravamo šta je od to dvoje
        beq     $9, $0, Exit    # ako je terminator, pretraga je završena
        addi    $8, $8, 1       # u suprotnom je 'b', pa ga prebrojimo
        addi    $4, $2, 1       # prije ponovnog poziva bfind podesimo novu
                                # početnu adresu: adresa nadjenog 'b' plus 1
        j       Loop
Exit:   add     $2, $8, $0      # prosljeđujemo brojač
        lw      $8, 0($29)      # vraćamo u pom.registar originalni sadržaj
        lw      $9, 4($29)      # vraćamo u pom.registar originalni sadržaj
        addi    $29, $29, 8     # podešavanje pokazivača na vrh steka
        jr      $31             # vraćanje u glavni program
```

NAPOMENA: Uočite da sadržaj registra $8 (koji nam je služio kao brojač) kopiramo u izlazni registar $2 na kraju procedure *bcount*, kada smo završili pozivanje procedure *bfind*. Zašto? Zato što *bfind* koristi $2 da vraća svoj rezultat (adresu karaktera 'b' ili terminatora), pa ne smijemo da mu mijenjamo vrijednost, kako bi zadržali ispravnost rezultata.