3 ASEMBLERSKI PROGRAMSKI JEZIK – JEZIK RAČUNARSKOG SISTEMA –

Iako se od pojave prvih računara do danas, puno toga izdešavalo i promjenilo, ono što čini suštinu računara je ostalo isto. *Input* (ulazni podaci), *output* (izlazni podaci), *memorija*, *Datapath* (*hardware* potreban za rad sa podacima) i *kontrola* (upravljanje) u potpunosti karakterišu rad svakog računara, a kombinacija zadnje dvije komponente čini u stvari **procesor** koji omogućava rad sa podacima. Danas, čini se više nego ikad, razvoj računara je uslovio potrebu dizajnera i *hardware*-a i *software*-a da što više računar učine dostupnijim širokoj populaciji, tako što će olakšati njegovu upotrebu u najrazličitije svrhe. Stoga, možemo ustvrditi da moderan svijet u skoro svakoj oblasti življenja koristi računar kao praktično nezamjenjivi dio, ali i pored toga ipak mali broj ljudi ima spoznaju o tome kako računar razumije ono što mi želimo od njega. U cilju upravljanja računarom, odnosno hardverom samog računarskog sistema, čovjek mora upotrebljavati jezik kojim računar komunicira. Jezik računara (mašine) se naziva **mašinskim jezikom**. Riječi koje se upotrebljavaju u mašinskom jeziku, u cilju kreiranja određenih akcija, se nazivaju **instrukcijama** ili **naredbama**, dok se vokabular mašinskog jezika naziva **skupom instrukcija**.

Instrukcije zadaje čovjek u nekom od viših programskih jezika, na način koji je, zavisno od upotrebljavanog višeg programskog jezika, više ili manje sličan uobičajenom čovjekovom načinu komuniciranja. Sa druge strane, u računaru se instrukcije zapisuju u obliku niza jedinica i nula kojima se formiraju memorijske riječi. Drugim riječima, pojedinačne instrukcije se razlikuju u odnosu na položaj njihovog posmatranja:

- Imaju jedan izgled iz čovjekove perspektive – način na koji ih čovjek zadaje u nekom od viših programskih jezika, i
- Drugi izgled iz perspektive samog računara u kome se instrukcije zapisuju – način na koji ih računarski sistem čita.

Savremeni računari se baziraju na dva ključna principa:

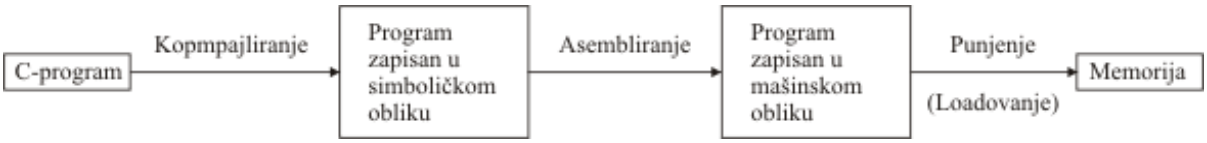
- Instrukcije se predstavljaju na isti način kao i binarni brojevi – nizom jedinica i nula,
- Programi se učitavaju u i čitaju iz memorije računara kao da se radi o brojnim veličinama.

Medjutim, postavlja se pitanja: kako se dolazi do zapisivanja naredbi, pisanih u nekom od viših programskih jezika, u obliku niza jedinica i nula? Program pisan u nekom od viših programskih jezika (C-jezik, Pascal, Fortran, ...) se najprije konvertuje u njegov simbolički oblik. Ovaj proces se obavlja kompajliranjem u odgovarajućem sistemskom software-u – compileru. Ulogu kompajlera može izvršavati i čovjek – programer u asemblerskom jeziku. Simbolička reprezentacija instrukcija se naziva reprezentacijom u *asemblerskom jeziku*. Numerički ekvivalent programa napisanog u simboličkom obliku se naziva *mašinskim kodom*. Transformacija programa napisanog u simboličkom obliku u mašinski jezik se naziva asembliranjem i izvršava se sistemskim software-om – assemblerom. Program napisan u mašinskom kodu se smješta u memoriju računara sistemskim programom nazivanim punjač (engl. loader). Ovaj program je potom spreman za izvršavanja od strane procesora samog računara (CPU).

Nakon ovih opservacija, iako smo na samom početku izlaganja, možemo uočiti osnovni cilj projektanata računara: *Pronalaženje asemblerskog jezika koji će omogućiti jednostavno kreiranje*

računara i njegovog kompajlera, jednovremeno postižući maksimalne performanse računara i njegovu minimalnu cijenu.

U ovoj glavi definišemo instrukcije u simboličkom obliku i njihove formate zapisivanja. Zajedno sa prezentiranjem skupa instrukcija kojim raspolaže asemblerski jezik računara otkrivamo način čuvanja programa u računaru. Skup instrukcija sa kojim ćemo raditi pripada MIPS skupu instrukcija koje se počeo razvijati 80-tih godina prošlog vijeka, a upotrebljava se u savremenim računarskim sistemima. Napomenimo da je MIPS skraćenica koja se sastoji od početnih slova engleskih riječi ***M**illions **I**nstructions **P**er **S**econd*, kojima se označava arhitektura kojom se omogućava izvršavanje nekoliko miliona instrukcija u jednoj sekundi.



Slika 1. Pricipijelna šema smještanja programskog koda, zapisanog u višem programskom jeziku (na slici je pretpostavljen C programski jezik) u memoriju računara.

3.1 JEZIK RAČUNARA – INSTRUKCIJE

Hardverom svakog računara mora biti omogućeno izvršavanja aritmetičkih i logičkih operacija. Najprije ćemo posmatrati aritmetičke instrukcije, dok će logičke instrukcije biti razmatrane prilikom projektovanja aritmetičko-logičke jedinice računara.

3.1.1 ARITMETIČKE INSTRUKCIJE

Format zapisivanja aritmetičkih instrukcija. Pođimo od instrukcije kojom se omogućava sabiranje operanada – operacija sabiranja. MIPS notacija instrukcije sabiranja je:

add a, b, c

Ovom instrukcijom se naređuje računaru da sabere promjenljive b i c i da dobjeni rezultat smjesti u promjenljivoj a (veoma brzo ćemo reći da su raspoloživi registri računarskog sistema – promjenljive hardvera računara). Primijetimo razliku u asemblerskom obliku zapisivanja instrukcije sabiranja u odnosu na njeno zapisivanje u nekom od viših programskih jezika: U asemblerskom jeziku su rezultujuća promjenljiva i promjenljive u kojima se smještaju operandi razdvojene zarezima i najprije se navodi rezultujuća promjenljiva, potom promjenljiva u kojoj je smješten prvi operand, te na koncu promjenljiva sa drugim operandom.

Navedena notacija za aritmetičku instrukciju sabiranja je stroga i odnosi se na sve aritmetičke instrukcije. Drugim riječima, sve aritmetičke instrukcije zadovoljavaju sljedeća pravila zapisivanja:

1. Najprije se navodi ime instrukcije (u našem slučaju *add* – imena instruncija će biti zapisivana kurzivom),
2. Instrukcije sadrže tačno tri promjenljive,
3. Promjenljive u instrukcijama se navode saglasno sljedećem redu: rezultujuća promjenljiva, promjenljiva koja sadrži prvi operand, promjenljiva koja sadrži drugi operand,
4. Promjenljive su razdvojene zarezima.

Primjer 1. Naći sumu promjenljivih b, c, d, e i dobijeni rezultat smjestiti u a.

Rješenje: Saglasno pravilima zapisivanja instrukcija u asemblerskom obliku (mogućnost sabiranja samo dvije promjenljive), za sumiranje četiri promjenljive potrebno je 3 puta zaredom upotrijebiti instrukciju sumiranja:

add a, b, c # $a = b + c$

add a, a, d # $a = a + d = b + c + d$

add a, a, e # $a = a + e = b + c + d + e$.

Napomenimo da se u asemblerskom jeziku upotrebljava znak # da bi se označilo mjesto za navođenje komentara od strane programera u asemblerskom obliku, kao i da se komentari nalaze na kraju linije zapisanog programa.

Primijetimo odmah razlike asemblerskog oblika zapisivanja instrukcija u odnosu na njihovo zapisivanje u drugim programskim jezicima:

1. Svaka linija programa pisanog u asemblerskom obliku može sadržati najviše jednu instrukciju,
2. Komentari se navode na kraju linije programa iza znaka #.

ZAPAŽANJE: Za instrukciju sabiranja i instrukcije slične njoj prirodan broj operanada je tri. Međutim, ovaj zahtjev se primjenjuje na sve aritmetičko-logičke instrukcije (ili tzv. instrukcije R-tipa) u cilju prilagođavanja filozofiji projektovanja jednostavnog hardvera. Naime, hardver za promjenljiv roj operanada je značajno komplikovaniji od hardvera sa tačno određenim brojem operanada (u našem slučaju – tri operanda).

Ovim zapažanjem dolazimo do **I principa projektovanja hardvera***: Jednostavnost favorizuje* (*ili čuva*) *regularnost* (*ili pravilnost*).

Primjer 2. Dat je segment programa napisan u C programskom jeziku:

$a = b + c;$

$d = a - e;$

Prebaciti naredbe napisane u C programskom kodu u MIPS asemblerski kod (drugim riječima, kompajlirati dio programa napisan u C jeziku). Predstaviti dobijeni asemblerski oblik dijela programa.

Rješenje: Simboličkim predstavljanjem ovih instrukcija (simboličkim načinom predstavljanja instrukcija, one se predstavljaju u asemblerskom obliku), uz uvođenje instrukcije *sub* (od *sub*straction – oduzimanje) dobijamo:

add a, b, c

sub d, a, e

Primjer 3. Dat je segment programa napisan u C programskom jeziku:

$f = (g + h) - (i + j);$

Napisati programski kod koji se za dati segment programa dobija na izlazu iz C kompajlera.

Rješenje: Mogući izlaz iz C kompajlera je:

add t0, g, h # u privremenoj promjenljivoj t0 smještamo sumu promjenljivih g i h

add t1, i, j # $t_1 = i + j$

sub f, t0, t1 # $f = t_0 - t_1$.

Primijetimo da smo predloženim kompajliranjem raširili program sa jedne (u C programskom jeziku) na tri linije (u asemblerskom kodu), uvođenjem privremenih promjenljivih t0 i t1. Ovo predstavlja cijenu zadržavanja stroge notacije, kojom se podrazumijeva postojanje tri promjenljive po jednoj instrukciji.

3.1.2 OPERANDI HARDVERA RAČUNARA

U višim programskim jezicima se operadni aritmetičkih instrukcija definišu promjenljivima. Prilikom kompajliranja programa, promjenljivima iz viših programskih jezika se moraju dodijeliti mjesta u računarskom hardveru. Drugim riječima, posmatrajući računarski hardver, promjenljive aritmetičkih instrukcija ne mogu biti proizvoljne promjenljive, kao što je to slučaj prilikom njihovog

definisanja u višim programskim jezicima, pošto bi se onda zahtijevao računarski hardver nemjerljivih dimanzija. One moraju biti iz konačnog broja specificiranih memorijskih lokacija – registara. Registri predstavljaju osnovne konstrukcione elemente računarskog sistema, male i veoma brze lokacija za smještanje podataka. Njihova osnovna najmena je omogućavanje pristupa računarskom sistemu od strane programera, što se ostvaruje upotrebom registara u različite svrhe. Kompajliranjem programa se promjenljive, deklarisane u višim programskim jezicima, pridružuju pojedinim registrima računarskog hardvera (stručnim rečnikom kazano: *kompajliranjem se vrši alokacija registara*). U slučaju MIPS arhitekture, uvode se sljedeća ograničenja u pogledu veličine i broja registara:

1. Veličina registara je 32 bita. Grupe od po 32 bita se u MIPS arhitekturi upotrebljavaju tako često (vidjet ćemo to i na primjeru memorija) da im se daje posebno ime - **memorijska riječ** ili jednostavno **riječ** (engleski word).
2. Broj registara je 32. Ovaj broj registara je različit i zavisi od upotrebljavane arhitekture. U MIPS arhitekturi je on 32, a kod drugih arhitekture njihov broj se kreće između 16 i 32. MIPS notacija za predstavljanje ovih registara je: $0, $1, $2, ..., $31.

Napomenimo da je ograničenjem 2. definisana osnovna razlika između promjenljivih viših programskih jezika i registara: *u višim programskim jezicima može biti definisan neograničen broj promjenljivih, ali se one mogu smjestiti samo u 32 registra računarskog hardvera*. Kasnije ćemo vidjeti da se sve ostale promjenljive (preko broja 32) smještaju u memoriji računara. Drugim riječima, zavisno od namjene i učestale upotrebe neke promjenljive, njena vrijednost se smješta u registru ili u memoriji računarskog hardvera. Kako je broj registara ograničen, jednostavno se može zaključiti da *efikasna upotreba registara predstavlja ključ za postizanje boljih performansi izvršavanog programa*.

Prilikom definisanja ograničenog broja registara (32 u MIPS arhitekturi) polazi se od II principa projektovanja hardvera, kojim se artikuliše disproporcionalnost veličine i brzina računarskog sistema: *Manji hardver je brži*.

Međutim, nakon definisanja ovog principa, nameće se logično pitanje: zašto je broj registara ograničen i zašto on iznosi 32? I: da li bi sistem sa 31 registrom bio brži od sistema koji upotrebljava 32 registra?

Odgovori na ova pitanja su jednostavni: Veliki broj registara povećava trajanje clock-ciklusa, pošto su električni signali sporiji kada prolaze kroz više elektronskih kola. Međutim, II princip projektovanja hardvera ne treba shvatiti u apsolutnom smislu: upotreba 31 registara ne mora povećavati performanse računara u odnosu na upotrebu 32 registra. Naime, performanse računara se mogu posmatrati iz dva ugla:

1. Programerskog i njihove žudnje za velikim brojem registara, u cilju omogućavanja konfornog programiranja, i
2. Dizajnerskog i njihove želje za "brzim clock-om".

Pravljenjem kompromisa između ovih suprostavljenih zahtjeva, dolazi se do optimalnog broja registara – 32 u MIPS arhitekturi.

Primjer 4. Posmatrajmo instrukciju zapisanu u C programskom jeziku:

$$f = (g + h) - (i + j);$$

Pretpostavljajući da se promjenljive f, g, h, i , j pridružuju registrima $16 – $20, respektivno, napisati MIPS asemblerski kod.

Rješenje: MIPS asemblerski kod za postavljeni program je:

```
add   $8, $17, $18      # $8 odgovara privremenoj promjenljivoj t0 iz primjera 3
add   $9, $19, $20      # $9 odgovara privremenoj promjenljivoj t1 iz primjera 3
sub   $16, $8, $9
```

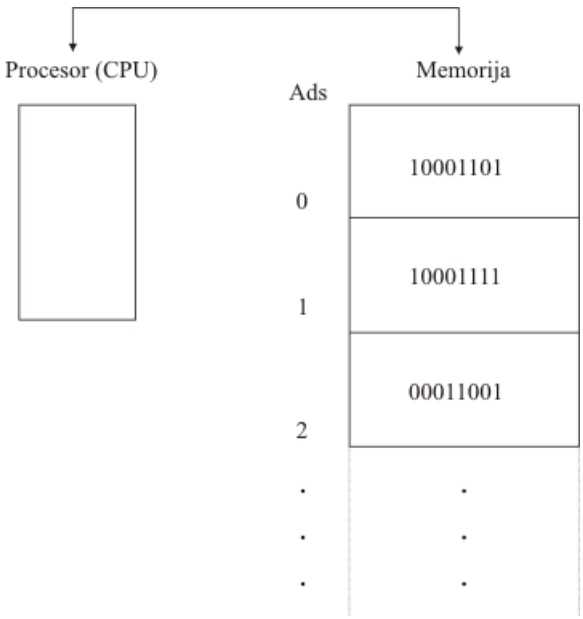
Dosad su posmatrane samo proste ili elementarne promjenljive koje sadrže jedinične podatke kao svoje elemente. Pored njih, postoje kompleksne strukture podataka koje mogu sadržati veliki broj

elemenata – često značajno veći broj elemenata os samog broja registara u sistemu. Takva struktura može biti *vektor* (ili *niz*) podataka. Pošto se u registrima računarskog sistema može smjestiti ograničen skup podataka (najviše 32 podatka), podaci iz kompleksne strukture se čuvaju u memoriji u kojoj se mogu smjestiti milioni ovakvih podataka.

Međutim, prilikom dizajniranja računara se uvodi ograničenje da se aritmetičke instrukcije mogu izvršavati samo nad operandima sadržanim u registrima računarskog sistema. Pošto se određene promjenljive mogu nalaziti i u memoriji računara (npr., promjenljive iz kompleksnih struktura podataka), MIPS skup instrukcija mora sadržati instrukcije koje omogućavaju prenos ili transfer podataka između registara i memorije računara – tzv. *data transfer instrukcije* ili *instrukcije za prenos* (*ili transfer*) *podataka*.

3.1.3 DATA-TRANSFER (MEMORY REFERENCE) INSTRUKCIJE

U cilju pristupa podacima čuvanim u memoriji rečunara, svaka memorijska lokacija (namijenjena čuvanju podataka) mora imati svoju adresu. Adresa memorijske lokacije omogućava izbor tačno određene memorijske lokacije, na isti način kao što adresa stana omogućava poštaru raznošenje pošte. Memorija se, stoga, može posmatrati kao veliki, jednodimenzioni niz, sa adresama koje predstavljaju indekse tog niza i elementima koji predstravljaju podatke sačuvane u memorijskim lokacijama, slika 1. Pretostavljajući 8-bitne lokacije memorije sa slike 1, može se zaključiti da se treći elemenat proizvoljne strukture podataka nalazi u lokaciji sa adresom 2, te da je u njoj sačuvan podatak 00011001. Drugim riječima: $\text{Memory}[2] = 00011001$.



Slika 1. Adrese momorijskih lokacija i njihov sadržaj.

Vidjeli smo dosad da se određeni podaci mogu čuvati u memoriji računara, ali da oni tada ne predstavljaju operande. Da bi se nad ovim podacima mogle izvršavati određene aritmetičke (i logičke) operacije, neophodno ih je prenijeti u neki od 32 registra računara. Takođe, lako se može zakaljučiti da je ograničeni broj registara nedovoljan za čuvanje svih rezultata i međurezultata aritmetičkih oparacija u njima, te da se isti čuvaju u memoriji računara. Iz ovih razloga, skupom instrukcija je neophodno obezbijediti dvije komplemetarne instrukcije: jednu koja će omogućiti prenos podataka iz pojedine memorijske lokacije u tačno određeni registar (i time od memorijske riječi napraviti operand hardvera računara), i drugu koja će omogućiti prenos podataka iz registra u memorijsku lokaciju (u cilju čuvanja dobijenog rezultata ili međurezultata određene aritmetičke instrukcije u memoriji računara). Operacija donošenja podataka iz memorije računara u neki od registara sistema se naziva *load* operacijom (ili punjenjem registra sadržajem određene memorijske lokacije), dok se operacija čuvanja sadržaja registra u memoriji računara naziva *store* operacijom. U MIPS arhitekturi ove instrukcije dobijaju posebna imena i oznake:

1. *load word* instrukcija u oznaci *lw*,
2. *store word* instrukcija u oznaci *sw*.

Format zapisivanja data-transfer instrukcija. Data-transfer instrukcijama se vrši jedna od dvije moguće akcije:

1. premještanje podatka iz memorije računara u određeni registar računarskog sistema (*lw* naredba), ili
2. premještanje (među)rezultata dobijen proizvoljnom aritmetičkom ili logičkom instrukcijom iz određenog registra u memoriju računara (*sw* naredba) u cilju njegovog čuvanja.

Memorijska lokacija sa koje se uzima podatak ili u koju se čuva dobijeni (među)rezultat, kao i registar u koji se podatak smješta ili iz kog se uzima dobijeni (među)rezultat su označeni zapisom data-transfer instrukcije. Format zapisivanja data-transfer instrukcije u asemblerskom obliku se sastoji iz nekoliko cjelina poređanih jedna za drugom:

1. ime instrukcije (*lw* ili *sw*);
2. obilježje registra koji se snadbijeva podatkom iz memorije računara (prilikom implementacije *lw* instrukcije) ili iz kojeg se uzima dobijeni (među)rezultat prethodne operacije (prilikom implementacije *sw* instrukcije);
3. početna adresa niza u memoriji računara u odnosu na koju se relativno posmatra adresa lokacije u kojoj se nalazi podatak koji se premješta u registar čije je obilježje dato u cjelini 2. (prilikom implementacije *lw* instrukcije) ili u kojoj se čuva (među)rezultat iz registra čije je obilježje dato u cjelini 2. (prilikom implementacije *sw* instrukcije);
4. obilježje registra čiji sadržaj je indeks elementa niza u memoriji računara (početna adresa ovog niza je navedena u tački 3.), čijim sadržajem se puni registar iz cjeline 2. instrukcije (prilikom implementacije *lw* instrukcije) ili indeks elementa niza u kom će se čuvati (među)rezultat iz registra iz cjeline 2. instrukcije (prilikom implementacije *sw* instrukcije).

Primijetimo da se apsolutna memorijska adresa elementa niza (memorijske lokacije sa koje se premješta podatak ili u koju se smješta (među)rezultat iz registra računara) formira sumiranjem početne adrese niza u memoriji računara (konstantni dio instrukcije – cjelina 3.) i sadržaja indeks registra čije obilježje se navodi u cjelini 4. instrukcije (tzv. indeks elementa niza).

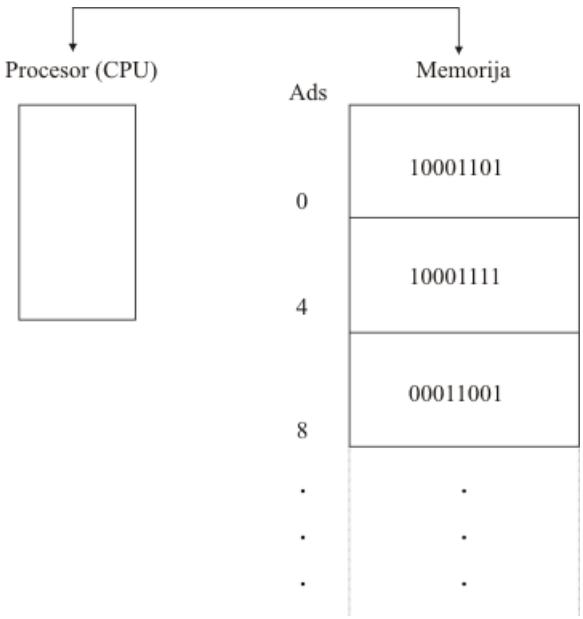
Primjer 5. Pretpostavimo da je A vektor sa 100 elemenata i da su kompajliranjem registrima \$17, \$18, \$19 pridružene promjenljive g, h, i. Pretpostavimo, takođe, da je vektor A smješten u memoriji računara počev od adrese Astart. Napisati u asemblerskom obliku sljedeću instrukciju pisanu u C programskom jeziku:

$$g = h + A[i];$$

Rješenje:

```
lw      $8, Astart($19) # u registaru $8 se smješta A[i]
add     $17, $8, $18            # g = h + A[i]
```

Pojašnjenje: Instrukcijom *lw* se, najprije, formira apsolutna adesa i-tog elementa niza A (u C programskom jeziku označenog sa A[i]) dodavanjem početne adrese niza A (oznečene sa Astart) indeksu i čija se veličina nalazi u registru \$19 (\$19 – index registar). Nakon toga procesor pročita podatak iz memorije računara, sa formirane apsolutne adrese lokacije, i smješta je u privremeni registar \$8. Upotreba privremenog registra u ovom primjeru je obavezna, pošto bi nemoguće bilo izvršiti aritmetičku operaciju sabiranja ukoliko se operand A[i] ne bi prethodno smjestion u neki od registara računarskog sistema. Registar \$8, kao privremeni registar, je proizvoljno odabran. Instrukcijom *add* sabiramo sadržaj registra \$18 (u kom je smještena vrijednost promjenljive h) sa sadržajem registra \$8 (u kome se nakon izvršavanja *lw* instrukcije nalazi A[i]) i dobijenu sumu smještamo u registar \$17, koji je pridružen promjenljivoj g.



Slika 2. Aktuelne MIPS adrese momorijskih lokacija i njihov sadržaj.

Primijetimo da se upotrebom data-transfer instrukcije zadaje početna adresa niza smještenog u memoriji računara. Postavlja se pitanje: kako računar prilikom izvršavanja ove instrukcije može da zna koja je to početna adresa niza u memoriji računara, da bi mogao izvršiti samu naredbu? Odgovor se, ponovo, pronalazi u kompajliranju. Kompajler, naime, najprije alocira registre i dodjeljuje ostalim promjenljivima (samim tim i kompleksnim strukturama podataka, time i vektorima) određena mjesta u memoriji računara. Stoga, kompajler potom može prepoznati i smjestiti početnu adresu niza u cjelinu 3. mašinskog koda data transfer instrukcije.

U mnogim programima se preferira adresiranja podataka u memoriji računara po byte-ima (skup od 8 bita, koji čine jednu cjelinu), tako da MIPS arhitektura, po pravilu, podrazumijeva adresiranja po byte-ima. Prisjećajući se da smo MIPS arhitekturom definisali 32-bitne memorijske riječi, jednostavno se može zaključiti da svaka riječ u memoriji računara (tj, memorijska lokacija) sadrži 4 byte-a, tako da je adresa lokacije tek jedna od četiri adrese pojedinih byte-a koje sadrži memorijska riječ zapisana u toj lokaciji. Sažeto kazano, adrese susjednih lokacija u memoriji računara se razlikuju za 4.

Upoređujući adresiranje lokacija memorije računara sa slika 1. i 2. lako se mogu uočiti razlike. Na slici 2. se vrši adresiranje memorijskih lokacija po byte-ima, tako da je adresa lokacije umnožak broja 4 (broja byte-a po 32-bitnoj memorijskoj riječi). Važno je napomenuti da ovako zapisane adrese memorijskih lokacija u binarnom obliku imaju 0 na posljednja dva binarna mjesta. Ova činjenica će biti upotrebljavana kasnije, prilikom dizajniranja hardware-a i njegovog adresiranja. Naravno, sadržaji memorijskih lokacija na slikama 1. i 2. ostaju neizmijenjeni.

Aktuelno MIPS adresiranje memorije po byte-ima utiče, pored ostalog, na reprezentaciju indeksa (i) elementa niza koji je sadržan u cjelini 4. data-transfer instrukcije i kojim je označen odgovarajući elemenat niza u memoriji računara. Preciznije rečeno, u cilju odgovarajućeg adresiranja prvog byte-a željene memorijske riječi, čiji je indeks sadržan u registru čije obilježje se nalazi u cjelini 4. data-transfer instrukcije (registar \$19 iz primjera 5), ovaj registar mora sadržati četvorostruku veličinu indeksa ($4\times i$). Tada suma sadržaja indeks registra (registar \$19 iz primjera 5) i početne adrese niza u memoriji računara (Astart iz primjera 5) ne bi selektovala elemenat niza A[i/4] u memoriji računara, već bi selektovala elemenat niza A[i].

Primjer 6. Pretpostavimo da su registri \$17 i \$18 pridruženi promjenljivima g, h, kao i da registar \$19 sadrži veličinu $4\times i$ (za razliku od primjera 5) u cilju korektnog adresiranja memorije računara po byte-ima. Predstaviti MIPS asemblerski kod sljedeće instrukcije, pretpostavljajući da je A vektor sa 100 elemenata i da je smješten u memoriji računara počev od adrese Astart.

$$A[i] = h + A[i];$$

Rješenje:

lw \$8, Astart(\$19)

add \$8, \$8, \$18

sw \$8, Astart(\$19)

Uočimo razliku ovog rješenja u odnosu na rješenje primjera 5: rezultat instrukcije sumiranja u drugoj liniji nije smješten u registru \$17 (pridružen promjenljivoj g), već u privremenom registru \$8.

NAPOMENA: U programima je veoma često veći broj upotrebljavanih promjenljivih od broja registara, tako da je kompajler prisiljen čuvati samo najčešće upotrebljavane promjenljive u registrima računarskog sistema, dok ostale promjenljive nastoji sačuvati u memoriji računara. Proces čuvanja rjeđe upotrebljavanih promjenljivih u memoriji računara se naziva rasipanjem registara (engl. spilling registers).

Naime, drugim principom projektovanja hardware-a se usvaja disproporcionalnost veličine i brzine računarskih komponenti, tako da se jednostavno može zaključiti da registri predstavljaju brže lokacije od memorije računara (registar odgovara jednoj memorijskoj lokaciji i ukupan broj registara je mnogostruko manji od broja memorijskih lokacija). Ovo predstavlja razlog čuvanja često upotrebljavanih podataka u registrima, kako bi time bila omogućena jednostavna manupulacija sa njima. Analizirajmo upotrebu registara u svrhu čuvanja često upotrebljavanih podataka u dosad razmatranim instrukcijama:

1. MIPS aritmetičke instrukcije:
   - ✓ Pristupaju sadržaju dva registra u cilju njihove upotrebe kao operanada;
   - ✓ Izvršavaju željenu operaciju nad ovim operandima;
   - ✓ Upisuju dobijeni rezultat operacije u rezultujući registar.
2. Data-transfer instrukcijama se, za razliku od prethodnog slučaja, samo može pročitati (iz memorije računara) ili upisati (u memoriju) jedan operand, bez mogućnosti vršenja bilo kakve operacije nad njima.

Drugim riječima, nad podacima koji se nalaze u registrima je moguće izvršiti instrukcijom zahtijevanu operaciju za vrijeme njenog trajanja, dok se nad podacima sadržani u memoriji računara ne može izvršavati oparacija do trenutka njihovog donošenja u neki od registara. Kada bi se omogućilo suprotno, data-transfer instrukcije bi bile značajno dužeg trajanja u poređenju sa drugim instrukcijama, odnosno došlo bi do značajne nestrazmjernosti u trajanju pojedinih instrukcija. Iz ovoga poizilazi zaključak da *u cilju postizanja najviših performansi računara, MIPS kompajleri moraju efikasno upotrebljavati postojeće registre. U njihovoj efikasnoj upotrebi, u stvari, leži tajna poboljšanja performansi računarskog sistema*.

NAPOMENA: U ovoj glavi će biti izostavljene pojedine naredbe iz potpunog skupa MIPS naredbi, tako da će, u stvari, biti prikazan potskup potpunog skupa MIPS naredbi, sa ciljem što jednostavnjeg shvatanja razmatranih problema. Na primjer, puni skup MIPS naredbi sadrži i naredbe za izdvajanje pojedinih byte-a iz riječi, kao i njhovo prenošenja load i store naredbama, omogućavajući time nekim programskim jezicima upotrebu byte-a za izvršavanje svojih instrukcija.

3.2 REPREZENTACIJA INSTRUKCIJA U RAČUNARU

U ovom poglavlju će biti predstavljena razlika između načina na koji čovjek zadaje naredbe i načina na koji računar vidi te naredbe. Podsjetimo se, naime, da se brojne veličine u računaru predstavljaju nizom niskih i visokih nivoa električnih signala, odnosno u binarnom brojnom sistemu. Osnovni oblik informacije je bit, koji može biti predstavljen u više različitih oblika:

- ✓ Visokog i niskog nivoa signala;

- ✓ On i Off stanja prekidača;
- ✓ $\top$ i $\bot$; ili
- ✓ 1 i 0.

3.2.1 INSTRUKCIJE R-TIPA

Instrukcije se, u računaru, takođe predstavljaju nizov visokih i niskih nivoa električnih signala i mogu biti presdstavljene nizom jedinica i nula. Preciznije rečeno, svaki dio instrukcije može biti posmatran kao pojedinačni broj (ili niz jedinica i nula koji odgovara odgovarajućem dekadnom broju), koji se u instrukciji nalaze poređani strana na stranu, jedan do drugoga. Na primjer, instrukcija

add $8, $17, $18

može biti predstavljena sljedećim nizom dekadnih brojeva, koji su u računaru predstavljeni u svom binarnom obliku:

0	17	18	8	0	32

Svaki element instrukcije se naziva *poljem*. Prvo i posljednje polje, kombinovano pokazuju da se ovom instrukcijom izvršava operacija sabiranja. U drugom polju se nalazi obilježje registra koji sadrži prvi operand ($17), dok se u trećem polju nalazi obilježje registra koji sadrži drugi operand ($18). U četvrtom polju se nalazi obilježje registra u kom se smješta rezultat operacije sabiranja ($8). U petom polju se smješta kod ili iznos eventualnog pomjeranja (shiftovanja) dobijenog rezultata. Napomenimo da nije potrebno vršiti shiftovanje da bi se izvršila operacija sabiranja, tako da se u ovom polju nalazi nulta vrijednost. Napomenimo, takođe, da se shiftovanja vrše prilikom implementacije množenja ili dijeljenja stepenom osnove brojanja binarnog brojnog sistema, oblika 2^n.

Originalni, binarni oblik ove instrukcije u računaru je sljedeći:

	000000	10001	10010	01000	00000	100000
Br. Bita:	6	5	5	5	5	6

Primijetimo da su drugo, treće i četvrto polje instrukcije 5-bita polja. Razlog tome je jednostavan: u ovim poljima se upisuju obilježja registara, mi u računarskom sistemu raspolažemo sa 32 registra koji se mogu opisati sa 5 bita. Takođe je i peto polje 5-bitno, pošto riječi 32-bitne dužine, tako da su i shiftovanja ove veličine jedina moguća.

Ovim je definisan R-tip formata zapisivanja aritmetičko-logičkih instrukcija u računaru. Napomenimo da ime formata – R potiče od **R**egistar, pošto za svoje izvršavanje ove instrukcije upotrebljavaju operande hardware-a računara sadržane u registrima. Ove instrukcije se zapisuju sa 6 polja, koja sadrže 32 bita ukupno (što ujedno predstavlja veličinu riječi u MIPS procesoru). Ovim je podržan prvi princip projektovanja hardware-a, a u prilog njegovom održanju dizajnirat ćemo i ostale MIPS instrukcije na 32-bitnu dužinu. Poljima R-tipa instrukcija se daju posebna imena zavisno od namjene kojoj služe, a mi ćemo ih definisati grafičkim putem, prikazom datim na sljedećoj slici:

Naziv polja:	*Op*	*Rs*	*Rt*	*Rd*	*Shamt*	*Funct*
Br. Bita:	6	5	5	5	5	6

Prvo polje instrukcoje se naziva **Op**eracionim, jer je njime definisana operacija koja se obavlja datom instrukcijom. *Op* polje će postojati kod svih formata MIPS instrukcija i uvijek je 6-bitno. Polja *Rs* i *Rt* sadrže obilježja registara čiji sadržaji predstavljaju operande nad kojima se izvršava instrukcijom zahtijevana operacija. U polju *Rd* (od engleskih riječi **R**egister **d**estination – odredišni registar) se nalazi obilježje registra u kom se upisuje dobijeni rezultat operacije. U polju Shamt (od engleskih riječi **Sh**ift **am**oun**t** – veličina sfiftovanja) se smješta veličina mogućeg shiftovanja. *Funct* (od **Funct**ion) se definiše oblik ili vrsta aritmetičke ili logičke operacije koja se obavlja izvršavanjem instrukcije. Postojanje *Funct* polja je vezano samo za instrukcije R-tipa, pošto je poljem *Op* definisan oblik instrukcije, ali ne i konkretna instrukcija, kojih je u slučaju instrukcija R-tipa veći broj. Na

primjer, instrukcije sabiranja (*add*) i oduzimanja (*sub*) imaju isti kod u *Op* polju (0), međutim različite kodove u *Funct* polju (32 i 34, respektivno). Sve ostale instrukcije će biti definisane jedino *Op* poljem.

3.2.2 Instrukcije I-tipa

Pokušajmo data-transfer instrukcije zapisati u prezentiranom formatu. Time se izvodi nekoliko jednostavnh zaključaka:

1. Neodgovarajuća je dužina pojedinih polja R-formata zapisivanja instrukcije, kao i samo njihovo postojanje. Na primjer, data-transfer instrukcije sadrže početnu adresu niza u memoriji računara (tzv. treća cjelina data-transfer instrukcije – konstantni dio apsolutne adrese memorijske lokacije). Ukoliko bi se zadržao R-format mašinskog zapisivanja instrukcije, početna adresa niza bi se zapisivala u *Shamt* ili *Funct* polju i za njeno predstavljanje bi se upotrijebilo, shodno tome, samo 5 ili 6 bita. Sa druge strane u memoriji računara se adresira veliki broj lokacija – značajno veći od 2^5=32 ili 2^6=64 lokacije (koliko ih može biti zapisano u 5-bitnom i 6-bitnom polju, respektivno);
2. Formatom zapisivanja data-transfer instrukcija neophodno je značiti dva registra (cjeline 2. i 4. ovih instrukcija), dok se R-formatom zapisivanja MIPS instrukcija označavaju tri registra;
3. Definicijom data-transfer instrukcija nije predviđeno shiftovanje podataka, tako da prilikom zapisivanja data-transfer instrukcija postojanje *Shampt* polja deplasirano.

Nakon donošenja ovih zaključaka, jednostavno se može zaključiti postojanje konflikta između dvije želje dizajnera računarskih sistema:

1. Zadržavanje iste dužine (32-bitne) MIPS instrukcija;
2. Zadržavanje jedinstvenog formata zapisivanja svih MIPS instrukcija.

Nastali konflikt se rješava kompromisim putem, saglasno **trećem principu projektovanja hardware-a**: *Dobar dizajn pretpostavlja pravljenje kompromisa*.

KOMPROMIS: Zadržava se jedinstvena dužina MIPS instrukcija – 32-bitna, zahtijevajući, stoga, različite formate zapisivanja instrukcija za različite tipove instrukcija. Napomenimo da različitim formatima zapisvanja instrukcija komplikujemo hardware računara. Ipak, zadržavanjem sličnih formata zapisivanja instrukcija – upotrebom zajedničkih sastavnih polja instrukcija – nastoji se redukovati njegova kompleksnost.

Ovim se dolazi do do drugog formata zapisivanja instrukcija – **I tipa** (**I**mmediate – trenutni), koji će biti upotrebljavan, između ostalih, za zapisivanja data-transfer instrukcija. I-format zapisivanja instrukcija će biti predstavljan grafičkim putem na sljedećoj slici:

Naziv polja:	*Op*	*Rs*	*Rt*	*Address*
Br. Bita:	6	5	5	16

Na primjer, intrukcija

lw $8, Astart($19)

se prikazuje I-formatom zapisivanja instrukcija, smještanjem pojedinih djelova instrukcije u sljedeća polja:

$19 – u *Rs* polju

$8 – u *Rt* polju

Astart (obilježje početne adrese niza u memoriji računara) – u *Address* polju.

Primjetimo da je I-format zapisivanja instrukcija zadržao 32-bitnu dužinu zapisivanja instrukcije, uz zadržavanje maksimalnih sličnosti sa R-formatom:

1. Zajednička tri polja – *Op*, *Rs*, *Rt*, sa različitom funkcijom polja *Rt*;

2. Dužina *Address* polja instrukcija I-tipa odgovara zbirnoj dužini posljednja tri polja instrukcija R-topa, čime se zadržava 32-bitna dužina oba tipa instrukcija.

Različita funkcija *Rt* polja u različitim tipovima instrukcija se ogleda u sljedećem:

1. U R-formatu zapisivanja instrukcija polje *Rt* se upotrebljava za predstavljanje obilježja registra čiji je sadržaj dugi operand MIPS instrukcije;
2. U I-formatu zapisivanja instrukcija polje *Rt* se upotrebljava za predstavljanje obilježja registra u kom se smješta podatak uzet iz memorije računara (prilikom implementacije *lw* instrukcije) ili obilježja registra u kom se nalazi podatak koji će biti sačuvan u memoriji računara (prilikom implementacije *sw* instrukcije).

NAPOMENA: Sada se nameće logično pitanje: Kako računar razlikuje o kom tipu instrukcije se radi? Odnosno, kako računar može da zna da li se radi o tri polja instrukcije R-tipa (*Rd*, *Shamt*, *Funct*) ili se radi o 16-bitnom *Address* polju I-tipa instrukcije?

Odgovor na ova pitanja se pronalazi u jednostavnoj činjenici da se svakom formatu zapisivanja instrukcija dodjeljuje određeni skup vrijednosti u *Op* polju, tako da dizajnirani hardware sam zna da li da posljednjih 16 bita instrukcije posmatra kao jednu cjelinu (kod I-tipa instrukcija – polje *Address*) ili kao tri cjeline (kod R-tipa instrukcija – polja *Rd*, *Shamt*, *Funct*). Stoga je razumljivo da se ovo polje naziva kodom izvršavane operacije (engl. Operation code ili skraćeno Opcode).

Sličnosti i razlike među analiziranim formatima zapisivanja instrukcija je vizuelno možda najlakše elaborirati zapisivanjem dosad analiziranih distribucija, datim u sljedećoj tabeli.

Instrukcija	Format	*Op*	*Rs*	*Rt*	*Rd*	*Shamt*	*Funct*	*Address*
add	R	0	Reg	Reg	Reg	0	32	NU
sub	R	0	Reg	Reg	Reg	0	34	NU
lw	I	35	Reg	Reg	NU	NU	NU	adresa
sw	I	43	Reg	Reg	NU	NU	NU	adresa

Reg – Obilježje registra (broj 0 – 31 za 32 različita registra u sistemu),

NU – **N**e**U**potrebljavano polje u posmatranom formatu zapisivanja instrukcije.

Primjer 7. Napisati MIPS mašinski kod za asemblerski oblik programa iz primjera 6. Za početnu adresu niza A u memoriji računara pretpostaviti Astart=1200_{10}=0000 0100 1011 0000_2.

Rješenje: Asemblerski kod programa iz primjera 6 je imao oblik:

lw $8, Astart($19)

add $8, $18, $8

sw $8, Astart($19)

U cilju jednostavnijeg vizuelnog shvatanja kreiranja mašinskog koda instrukcija, iste ćemo predstaviti u tabelarnoj formi. Napisat ćemo ih najprije u dekadnom, a potom u binarnom obliku.

Dekadni oblik zapisivanja instrukcija:

Op	*Rs*	*Rt*	*Rd*	*Shamt*	*Funct*
			Address		
35	19	8	1200		
0	18	8	8	0	32
43	19	8	1200		

Binarni oblik zapisivanja instrukcija:

Op	*Rs*	*Rt*	*Rd*	*Shamt*	*Funct*
			Address		
100011	10011	01000	0000 0100 1011 0000		
000000	10010	01000	01000	00000	100000
101011	10011	01000	0000 0100 1011 0000		

Primijetimo da su prva i treća instrukcija veoma slične – razlikuju se samo u trećem bitu. Kasnije ćemo zaključiti da iz sličnosti binarnih reprezentacija povezanih instrukcija slijedi jednostavnija hardware-ska implementacija sistema.

Primijetimo da je asemblerski način zapisivanja programa veliki napredak u odnosu na zapisivanje programa u mašinskom kodu – nizom jedinica i nula. Uz to, prilikom assembliranja programa, assembleru se omogućava da suštinski proširi skup naredbi kojima raspolaže compiler. Naime, hardwareom računara se implementiraju naredbe koje postoje u assemblerskom jeziku. Medjutim, moguće je definisati naredbe kojima ne raspolaže assemblerski jezik, a koje, sa druge strane, mogu biti ekvivalentirane postojećim naredbama, uz eventualnu pomoć prilikom alociranja registara. Assembler na ovaj način efektivno povećava broj instrukcija koje se stavljaju na upotrebu programerima u asempleru ili compilerima. Ove instrukcije nije potrebno hardwareski implementirati, ali njihovo postojanje, sa druge strane, pojednostavljuje programiranje. Stoga se instrukcije ovog tipa nazivaju *pseudoinstrukcijama* (fizički ili hardwareski nepostojećim, ali suštinski postojećim instrukcijama).

Na primjer, posmatrajmo pseudoinstrukciju *move*, kojom se premješta sadržaj jednog registra (neka se radi o registru $18) u drugi registar (pretpostavimo da je u pitanju registar $8), a koja ne postoji u skupu MIPS instrukcija. Njen simbolički način zapisivanja je:

move $8, $18

Kako instrukcija *move* ne postoji u MIPS skupu instrukcija, asembler može konvertovati gornju instrukciju u njen mašinski ekvivalent, asembliranjem njoj identične instrukcije:

add $8, $18, $0

Uz obezbjedjivanje nultog sadržaja registra $0 sadržana nula. Detaljnije kazano, u cilju omogućavanja kreiranja pojedinih pseudoinstrukcija, potrebno je obezbijediti postojanje nulte vrijednosti u sistemu. Dogovorom je regulisano da MIPS hardware uvijek postavlja sadržaj registra $0 na nuloj vrijednosti. Drugim riječima, kad god se vrši alokacija registara, compiler ili programer u asembleru dužni su se uzdržati upotrebe registra $0 i time obezbijediti postojanje nulte vrijednosti. Ovim je broj registara, koji se ostavljaju na upotrebu compileru, umanjen za jadan.

3.3 INSTRUKCIJE ZA KREIRANJE ODLUKE (INSTRUKCIJE GRANANJA ILI USLOVNOG SKOKA)

Sposobnost donošenja odluka razlikuje računar od jednostavnog kalkulatora. Naime, na osnovu ulaznih podataka i medjurezultata dobijenih izvršavanjem odredjenih naredbi programa, računar je u mogućnosti da donosi odluke o izvršavanju različitih instrukcija. Odluke tipa: »izvršavanje jedne od nekoliko instrukcija«, su karakteristika svih viših programskih jezika. Obično se, u svrhu njihovog kreiranja, upotrebljavaju *if* instrukcije, manje ili više lesto u kombinaciji sa *goto* naredbama i oznakama naredbi (labelama). Postoje dvije vrste instrukcija grananja:

1. Uslovne instrukcije grananja, i
2. Bezuslovne instrukcije grananja.

MIPS skup instrukcija uključuje dvije uslovne instrukcije grananja:

1. *bne* (engl. **B**e **N**ot **E**qual) – koja ispituje nejedanakost oparanada;
2. *beq* (engl. **B**e **Eq**ual) – koja ispituje jednakost operanada.

Napomenimo da su ove instrukcije slične instrukciji *if*, iz viših programskih jezika, kombinovanoj sa instrukcijom *goto*. Sintaksa zapisivanja ovih instrukcija u simboličkoj formi je:

```
bne     $x, $y, L1
beq     $x, $y, L1
```

Ovim naredbama se definiše uslovni skok na naredbu sa obilježjem L1, ukoliko je zadovoljen uslov nejednakosti sadržaja registara $x i $y (prilikom implementianja naredbe *bne*), odnosno uslov jednakosti sadržaja registara $x i $y (prilikom implementianja naredbe *beq*).

NAPOMENA: Instrukcije se čuvaju u memoriji računara, kao dio sačuvanog programa, tako da sve one imaju svoje memorijske adrese. Prilikom assembliranja programa obilježja instrukcija zapisanih u simboličkom obliku (lebele, u prethodnom tekstu označena sa L1) se u mašinskom kodu zamjenjuju binarnom adresom instrukcije u memoriji računara. Pojednostavljeno, u gore zapisanom simboličkom kodu, labela L1 odgovara adresi instrukcije na koju se prelazi prilikom implementacije naredbe uslovnog grananja.

Primjer 8. Pretpostavljajući da su promjenljive *f*, *g*, *h*, *i*, *j* pridružene registrima $16 – $20, respektivno, predstaviti u simboličkoj formi sljedeće naredbe, pisane u C-programskom jeziku:

```
    if (i == j) goto L1;
    f = g + h;
L1: f = f - i;
```

Rješenje: Slijedeći zapisivanja instrukcija u C-programskom jeziku, jednostavno se iste mogu zapisati u simboličkom obliku na sljedeći način:

```
    beq     $19, $20, L1
    add     $16, $17, $18
L1: sub     $16, $16, $19
```

Primijetimo da se druga naredba u nizu preskače ukoliko je zadovoljen uslov jednakosti sadržaja registara $19 i $20 (odnosno uslov jednakosti promjenljivih *i* i *j*). Uz to, treća naredba u nizu će biti obavezno izvršavana, bez obzira na zadovoljenje ovog uslova.

NAPOMENA: Assembler olakšava posao compileru, ili programeru u assemblerskom kodu, izračunavanjem adresa na koje se vrše grananja prilikom izvršavanja programa (na sličan način kao što je pronalazio početnu adresu niza u memoriji računara prilikom implementacije data-transfer instrukcija). Naime, compiler dodjeljuje memorijski prostor, i samim tim poznaje adresu lokacije u kojoj je sačuvana instrukcija označena labelom (npr. labelom L1 u prethodnim primjerima).

Compileri veoma često kreiraju grananja i označavaju labelama instrukcije na koje se vrše grananja i kada one nijesu prisutne u izvornom programskom kodu.

Primjer 9. Upotrebljavajući promjenljive i registre iz prethodnog primjera, napisati u simboličkom kodu sljadeću naredbu, pisanu u C-jeziku:

```
if (i==j) f=g+h; else f=g-h;
```

Rješenje: Ovom naredbom je predstavljena if-than-else programska struktura. Njen mogući simbolički oblik može biti predstavljen sljedećim naredbama:

```
    bne     $19, $20, Else
    add     $16, $17, $18
    j       Exit
```

```
Else:  sub   $16, $17, $18
Exit:
```

Naredbom *bne* preskačemo drugu i treću instrukciju u nizu ukoliko je zadovoljen uslov nejednakosti sadržaja registara $19 i $20 i prelazimo na izvršavanje instrukcije *sub*. Treća instrukcija u nizu (*j* od *jump* - instrukcija bezuslovnog skoka o kojoj će kasnije biti više riječi) se upotrebljava u cilju preskakanja instrukcije sa obilježjem Else: i izvršava se u slučaju jednakosti sadržaja registara $19 i $20.

Instrukcije grananja (ili donošenja odluke) mogu biti primijenjene za rješavanje više problema:

1. za izbor izmedju dvije moguće alternative (u ovom cilju se upotrebljava if struktura u višim programskim jezicima),
2. za iterativna izračunavanja (petlje u višim programskim jezicima; npr. for, while, switch).

U oba navedena slučaja se instrukcije grananja upotrebljavaju kao osnovni elementi simboličkog koda navedenih struktura.

Primjer 10. Posmatrajmo petlju napisanu u C-jeziku:

Loop: $g = g + A[i]$;

$i = i + j$;

$if\ (i != h)\ goto$ Loop;

Pretpostavimo da je A niz od 100 elemenata, te da se kompajliranjem registri $17 – $20 respektivno dodjeljuju promjenljivima *g*, *h*, *i*, j, kao i da je početna adresa niza A u memoriji računara – Astart. Pretpostavimo, takodje, da je u registru $10 već smještena brojna veličina 4 i da je MIPS skupom instrukcija obezbijedjena instrukcija *mult*, kojom se vrši multipliciranje operanada. Napisati MIPS assemblerski kod koji odgovara ovoj petlji.

Rješenje: Prisjetimo se da se u MIPS arhitekturi upotrebljava adresiranje byte-a, tako da je, u cilju adresiranja odgovarajućeg byte-a memorije računara, neophodno pomnožiti indeks *i* elementa niza sa 4 prije upotrebe data-transfer instrukcije. Na nivou našeg sadašnjeg poznavanja MIPS assemblerskog jezika, multipliciranje indeksa sa odgovarajućom brojnom veličinom će biti izvršeno upotrebom *mult* instrukcije uz pretpostavku postojanja brojne veličine u nekom od registara. Kasnije ćemo, prilikom uvodjenja trenutnih operanada, vidjeti da se ova akcija može izvršiti na mnogo jednstavniji način. Stoga bi MIPS simbolički kod mogao da bude sljedeći:

```
Loop:  mult   $9, $19, $10      # Sadržaj registra $9 će biti jednaka veličini 4i
       lw     $8, Astart($9)    # Sadržaj registra $8 je A[i]
       add    $17, $17, $8
       add    $19, $19, $20
       bne    $19, $18, Loop
```

NAPOMENA: Primijetimo da je oznaka petlje (labela Loop) ustvari adresa instrukcije *mult*. Naime, sa svakim prolaskom kroz petlju mijenja se vrijednost indeksa *i* i, shodno tome, prilikom svakog novog pristupa memoriji (u cilju odgovarajućeg adresiranja) nova vrijednost indeksa *i* mora biti pomnožena sa 4. Takodje, primijetimo da vrijednost 4*i* smješta u registru $9, kako bi originalna vrijednost indeksa *i* (koja se mijenja prilikom izvršavanja programa) ostala sačuvana u registru $19.

Primjer 11. Pretvoriti tradicionalnu while petlju iz C-jezika

$while(save[i] == k)$

$i = i + j$;

u assemlerski kod. Prilikom transliranja zadatog programskog koda iz C-jezika u assemblerski kod pretpostaviti da se promjenljive *i*, *j* i *k* nalaze u registrima $19 – $21, te da je početna adresa vektora save u memoriji računara Sstart, kao i da je u registru $10 smještena brojna veličina 4.

Rješenje: Mogući izgled MIPS assemblerskog koda bi bio sljedeći:

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

Na sličan način kao u prethodnom primjeru, u registru $9 se nalazi četvorostruka veličina indeksa vektora koji se nalazi u memoriji računara. Pošto se njegova vrijednost mijenja unutar tijela *while* petlje, originalna veličina indeksa i ostaje sačuvana u registru $19. Da bi se *i*-ti elemenat niza save mogao porediti sa promvenljivom *k*, neophodno ga je dovesti u neki od registara (u našem primjeru ulogu privremenog registra ima registar $8). U slučaju nejednakosti *i*-tog elementa niza save sa promjenljivom *k*, stvaraju se uslovi za izlazak iz *while* petlje. Ova akcija je u prezentiranom assemblerskom kodu implementirana upotrebom instrukcije *bne*. U suprotnom slučaju (jednakosti *i*-tog elementa niza save sa promjenljivom *k*) izvšava se instrukcija *add* i prelazi se na novu iteraciju izvršavanja *while* petlje, implementirnjem instrukcije bezuslovnog skoka *j*.

NAPOMENA: Programeri izbjegavaju pisanje petlji sa naredbom bezuslovnog skoka *goto*. Naime, jednostavno se može uočiti da se prilikom svakog prolaska kroz prezentirani assemblerski kod *while* petlje izvršavaju dvije instrukcije skoka:

1. instrukcija *bne* – instrukcija uslovnog grananja, i
2. instrukcija *j* – instrukcija bezuslovnog skoka.

Svi moderniji compileri izvršavaju samo jednu od ove dvije instrukcije skoka, čime se umanjuje broj ukupno izvršavanih operacija u hardwareu računara.

Primjer za vježbu. Reprogramirati prezentirani assemblerski kod taka da se *while* petlja iskompajlira upotrebom samo jedne od dvije naredbe skoka.

3.4 INSTRUKCIJA ZA POREDJENJE OPERANADA – *SLT* INSTRUKCIJA

Naredbama uslovnog skoka (*beq* i *bne*) se ispituje (ne)jednakost posmatranih operanada, dok se njihov relativni odnos (da li je jedan operand manji ili veći od onog drugog) ne odredjuje. Medjusobni odnos oparanada je često veoma značajan, na primjer prilikom hardwareske implementacije *for* ili *while* petlji čiji se uslov temelji na medjusobnom odnosu promjenljivih.

U cilju rješavanja navedenog problema, MIPS skup instrukcija sadrži *slt* (od engleskih riječi **S**et on **L**ess **T**han – setovati rezultat ukoliko je zadovoljen uslov *manji od*) instrukciju. Ovom instrukcijom se postavlja sadžaj rezultujućeg registra (prvonavedeni ragistar prilikom simboličkog zapisivanja instrukcija) na 1, ukoliko je sadržaj prvog operanda (drugonavedeni registar u simboličkoj formi zapisivanja instrukcije) manji od sadržaja drugog operanda (trećenavedeni registar u simboličkoj formi zapisivanja instrukcije). Kod ostalih relativnih odnosa operanada rezultujući registar se postavlja na 0.

Detaljnije kazano, simbolički oblik zapisivanja instrukcije

```
slt    $8, $19, $20
```

postavlja sadržaj rezultujućeg registra $8 na jednu od dvije moguće vrijednosti:

1. 1 – ukoliko je veličina sadržana u registru $19 manja od veličine sadržane u registru $20,

2. 0 – u svim ostalim slučajevima.

Ovim se stvaraju uslovi za kombinovanu primjenu instrukcija *slt*, *bne* (ili *beq*) i fiksirane nulte vrijednosti registra $0 u cilju kreiranja svih relativnih uslova.

Na primjer, posmatrajmo instrukcije koje omogućavaju grananje na instrukciju obilježenu lamelom Less ukoliko je promjenljiva *a* (sadržana, npr., u registru $16) manja od promjenljive *b* (sadržane, npr., u registru $17). Sagledavajući efekte koji se postižu primjenom instrukcija *slt* i *bne*, zahtijevano grananje se može izvršiti sljedećim parom instrukcija:

slt $1, $16, $17

bne $1, $0, Less

Naime, implementiranjem instrukcije *slt* sadržaj privremenog registra $1 se postavlja na 1 ukoliko je sadržaj registra $16 manji od sadržaja registra $17. Drugim riječima, prvom instrukcijom se ispituje zahtijevani uslov grananja, koje se izvršava instrukcijom *bne*. Naime, ukoliko je sadržaj privremenog registra $1 različit od 0 (sadržaj registra $0), u njemu se nalazi 1 (tj. $a<b$) i izvršava se grananje na instrukciju sa obilježjem Less.

U suštini, prezentiranim parom instrukcija vršimo grananje ukoliko je zadovoljen ispitivani medjusobni odnos promjenljivih. Ovim je implementirana instrukcija *blt* (čiji naziv potiče od engleskih riječi **B**ranch on **L**ess **T**han – izvršava grananje u slučaju zadovoljenja uslova "manji od"). Drugim riječima, MIPS assembler implementira *blt* instrukciju njenom konverzijom u kombinaciju instrukcija *slt* i *bne*. Time se, još jedanput, kreira (od strane assemblera) hardware-ski nepostojeća instrukcija – odnosno kreira se pseudoinstrukcija *blt*.

Nameće se pitanje: Zašto se blt instrukcija ne biimplementirala i hardware-ski? Odgovor na ovo pitanje se može naći u njenoj suviše komplikovanoj hardware-skoj implementaciji iz dva razloga:

1. Produžava trajanje clock-intervala, ili
2. Uvodi dodatni clock-ciklus po instrukciji.

Drugim riječima, dvije brze instrukcije (*slt* i *bne*) su upotrebljivije i uvijek se upotrebljavaju u ovoj formi.

NAPOMENA: U cilju implementiranja *blt* pseudoinstrukcije, assembler mora raspolagati jednim privremenim registrom (kod nas je to registar $1), bez straha od promjena programa, tj. bez straha da će isti registar biti neophodan za izvrašavanje drugih instrukcija. Ovo se obezjedjuje od strane compilera (ili programera u assemblerskom jeziku) – upotrebom registra $1 isključivo za svrhu privremenog registra. Naime, compiler (ili programer u assemblerskom jeziku) vrši alociranje postojećih registara i on se treba uzdržati od upotrebe nekog registra u cilju njegove kasnije upotrebe u odredjene svrhe.

3.4.1 *Switch* ISKAZ

Mnogi viši programski jezici posjeduju switch iskaz koji omogućava programeru da odabere jednu od nekoliko mogućih alternativa, zavisno od vrijednosti jedne promjenljive. U principu, postoje dva načina za njeno implementiranje:

1. Postavljanjem niza tastera kroz lanac *if-than-else* struktura, ili
2. Indeksiranjem tabele adresa pojedinih instrukcija na koje se vrši prelazak.

Potreban uslov za implementiranje načuna 2. je efikasno kodirana tabela adresa niza alternativnih instrukcija. Da bi se moglo preći na instrukciju sa zapisanom aresom u nekoj memorijskoj lokaciji (ili registru), MIPS skupom unstrukcija se uvodi *jr* instrukcija (od engleskih riječi ***j**ump **r**egister*) – bezuslovni skok na memorijsku lokaciju sa aderesom specificiranom sadržajem registra. Na primjer, instrukcijom *jr* $7 se prelazi na izvršavanje instrukcije koja se nalazi u memorijskoj lokaciji sa adresom specificiranom sadržajem registra $7.

Primjer 12. Dat je *switch* iskaz zapisan u C-jeziku:

$switch(k)\{$

case $0: f = i + j;\ break;$

case $1: f = g + h;\ break;$

case $2: f = g - h;\ break;$

case $3: f = i - j;\ break;$

$\}$

Pretpostavljajući da su promjenljive *f* do *k* pridružene registrima \$16 do \$21, respektivno, te da se u registru \$10 nalazi brojna veličina 4, napisati simbolički MIPS kod.

Rješenje: Pretpostavimo da 4 memorijske riječi, počev od lokacije JumpTable, sadrže adrese koje odgovaraju labelama L0, L1, L2 i L3 (za *k*= 0,1,2,3), respektivno. Mogući simbolički kod za implementaciju traženog *switch* iskaza može da izgleda na sljedeći način:

```
Swt:  mult  $9, $10, $21
      lw    $8, JumpTable($9)
      jr    $8
L0:   add   $16, $19, $20
      j     Exit
L1:   add   $16, $17, $18
      j     Exit
L2:   sub   $16, $17, $18
      j     Exit
L3:   sub   $16, $19, $20
Exit:
```

Prvom instrukcijom, označenom labelom Swt, u registru \$9 smještamo veličinu 4*k*. Naime, ova veličina se upotrijebljava kao pokazivač na adresu riječi u memorijskoj lokaciji, a kod MIPS arhitekture se, sa druge strane, upotrebljava adresiranja byte-a, a ne adresiranje memorijskih riječi (tako da se adrese memorijskih riječi razlikuju za 4). Uočimo, takodje, da se dobijena četvorostruka vrijednost *k* smješta u privremeni registar \$9, čime se omogućava ponovna upotreba promjenljive *k* u daljem toku izvršavanja *switch* iskaza ili samog programa (čiji je dio ovaj iskaz).

3.5 KORIŠĆENJE PROCEDURA

Procedure su posebne cjeline koje se izvrše kad ih program pozove. Nakon izvršenja procedure, program nastavlja rad (izvršava instrukcije koje slijede nakon poziva procedure). Naime, program pozove proceduru a istovremeno zapamti adresu slijedeće naredbe i to u registru *\$31*. To se postiže instrukcijom "*JUMP and link*" (**jal**) koja ima izgled

jal Procedure Address

Link je dio koji vraća izvršavanje instrukcije na pravu adresu, a imamo i instrukciju

jr \$31 – JUMP return

jal jednostavno inkrementira registar na sljedeću instrukciju prije čuvanja u *\$31*, a ***jr \$31*** kopira sadržaj iz *\$31* u taj registar. Procedura može pozivati i neku drugu proceduru i tu nam trebaju ***pointeri*** da pokažu gdje smo koju proceduru smjestili i kako ih pozvati. **STACK** – je izuzetno bitan

pojam a može se smatrati da su to rasuti registri u koje smiještamo podatke. Postavljanje podataka u **STACK** je *push* operacija, a uklanjanje iz njega *pop* operacija.

Primjer: Pretpostavimo da procedura A poziva proceduru B a ova proceduru C. Prije nego B pozove C sačuva se adresa povratka u **STACK**. **STACK** *pointer* je registar *$29* i on se podesi na novi početak **STACK**-a. Onda se pozove C i **jal** promijeni sadržaj registra *$31* da sadrži adresu povratka u C. Kad se C vrati u B, stara zapamćena adresa se vrati iz **STACK**-a u *$31*, a **STACK** *pointer* opet uzme staru vrijednost za početak **STACK**-a.

Neka *$29* sadrži *pointer* za početak **STACK**-a i neka, recimo *$24* ima vrijednost početka **STACK**-a. Možemo pisati:

```
A: ...
   ...
   jal B   # poziva B i čuva adresu povratka u $31
B: ...
   ...
   ...     #spreman za poziv C
```

ili:

```
   add $29,$29,$24  #podešavanje STACK-a za prostor za sljedeću vrijednost
   sw $31, 0 ($29)   #čuva adresu povratka
   jal C       #zove C i čuva adresu povratka u $31, vraća se sa C na sljedeću instrukciju
   lw $31, 0 ($29)   #ponovo učitava adresu vraćanja za B
   sub $29,$29,$24  #podešava STACK za "pop" za B adresu povratka
   ...
   jr $31              #povratak na dio koji poziva B
C: ...
   ...
   jr $31              #povratak na dio koji poziva C
```

Po pravilu se za upis parametara koriste registri *$4*,…,*$7*, ali nekad ih treba više pa su uvedene dvije konvencije:

1. **caller save** (pozivanje procedura čuva i vraća registre, i tada pozvana procedura može mijenjati sadržaj svih registara bez ograničenja);

2. **calee save** (pozvani je odgovoran za čuvanje i *restoring* registara koji se koriste, i tada pozivač može bez brige za restoring koristiti registre).

Znači, ovo se koristi kad imamo više od 4 parametra (*$4,...,$7*), i bitno je zapamtiti da **STACK**-ovi "idu" od viših ka nižim adresama. To znači da unosimo (*push*) nove vrijednosti u **STACK** oduzimanjem od početnog **STACK** *pointera*, a uklanjanje vrijednosti iz **STACK**-a ide dodavanjem **STACK** *pointeru*.

3.6 NAČINI ADRESIRANJA

Postoji još nekoliko načina pristupa operandima. Prvi način je za brži pristup malim konstantama, a drugi da "*BRANCH*" instrukcije učini što efikasnijima. Ovo je korisno kod iteracija, kao i za brojače ili indekse niza.

Primjer A: Inkrementirati sadržaj registra $29 za 4.

I način: Stavimo najprije u registar *$29* konstantu 4, a potom izvršimo inkrementiranje sadržaja registra $29:

lw $24,AddrConstant4($0) #$24=constant4

add $29,$29,$24 #$29=$29+$24 ($24=4)

II način: Koristeći **immediate** (konstantni) oblik naredbe sabiranja,

addi $29,$29,4

što simbolički izgleda kao što je prikazano na sljedećoj slici

op	rs	rt	immediate
8	29	29	4

ili binarno, kao što je prikazano na sljedećoj slici:

001000	11101	11101	0000000000000100

Često se registar *$0* koristi za poređenje sa "**immediate**", kao

slti $8,$18,10 # $8=1 za $18<10

Ovdje je bitan osnovni princip : **česte stvari učini brzima!**

Nekad su konstante duže od 16 bita pa koristimo **lui** (*load upper immediate*) koja težih (prvih) 16 bita setuje, a dozvoljava drugu instrukciju za nižih 16. Naprimjer:

lui $8,255

001111	00000	01000	0000000011111111

Nakon ove komande,sadržaj registra *$8* je

0000000011111111	0000000000000000

dakle, 16 nižih bita postavi se u nulu.

Primjer: Uzmimo vrijednost 0000 0000 0011 1101 0000 1001 0000 0000 , i ovaj 32-bitni zapis želimo smjestiti u registar *$16,* tada imamo:

lui $16,61 #61 decimal 0000 0000 0011 1101

Vrijednost *$16* je sada: 0000 0000 0011 1101 0000 0000 0000 0000

Sad treba dodati nižih 16 bita ovome, a to radimo kao

add $16,$16,2304 #2304=0000 1001 0000 0000

$\Rightarrow$ konačna vrijednost za *$16*: 0000 0000 0011 1101 0000 1001 0000 0000

Primjer B: Adresiranje naredbom j(ump) je najjednostavnije i tu se koristi *J-tip* (format) instrukcije koji se sastoji od 6 bita za polje operacije, a ostatak bita za adresu polja:

j 10000 #goto location 10000

2	10000
6 bitova	26 bitova

za razliku od uslovnog adresiranja, čiji format zapisivanja ćemo posmatrati na primjeru sljedeće instrukcije:

bne $8,$21,Exit #goto Exit if $8≠$21

5	8	21	Exit

6 bitova 5 bitova 5 bitova 16 bitova

Primjer B1: *loop: mult $9,$19,$10 #priv.$9=i*4*

lw $8,Sstart($9) #priv.$8= save[i]

bne $8,$21,Exit #za save[i] ≠k idi na Exit

add $19,$19,$20 #i=i+j

j loop #idi na loop

Exit:

Ako pretpostavimo da je *loop* na lokaciji 8000, a da je *Sstart* na 1000, mašinski kod će biti kao u tabeli

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...						

Pošto se radi o *bajt – adresiranju* ide se po 4 (prva kolona s lijeva), a "*JUMP*" ima početnu adresu 80 000 za *loop*.

Nekad su uslovi (*BRANCH*) blizu lokacije skoka, ali su često daleko (pa se ne može preko 16 bita pokazati i daljina i instrukcija), pa se tada ubacuje bezuslovni skok, kao u donjem primjeru.

Primjer B2

beq $18,$19,L1 se može napisati kao *bne $18,$19,L2*

j L1

L2:

Znači, u suštini imamo 4 tipa adresiranja:

1. *Register addressing*, gdje je operand registar,
2. *bazno adresiranje*, gdje je operand na lokaciji u memoriji čija je adresa=registar+adresa instrukcije,
3. *immediate adresiranje*, gdje je operand konstanta sadr\ana u samoj instrikciji,
4. *PC (program counter)* – relativno adresiranje gdje je adresa=PC+konstanta iz instrukcije.

3.7 ALTERNATIVNI PRISTUP MIPS INSTRUKCIJAMA

Težnja je smanjiti broj instrukcija koje radi program, i zato se uvode autoinkrementacija i autodekrementacija, izuzetno značajne kod nizova. Obično se učita *riječ* (*word*) i uveća indeks registra ka novoj *riječi*. Kod autoinkrementiranja, čim se *riječ* pročita indeks uzme novu vrijednost (*riječ*=4 bajta). Ovo možemo predstaviti sa:

lw $8,Sstart($19) #u $8 ide S[$19]

addi $19,$19,4 #$19=$19+4

a moglo bi se zamijeniti instrukcijom:

lw+$8,Sstart($19) #$8=S[$19] i $19=$19+4

koja ne postoji u MIPS-u. Nekad program, slično autoinkrementaciji, ide kroz memoriju u kontra pravcu i tada se radi o autodekrementaciji.

Dalje ušteda u vremenu izvršavanja instrukcija se postiže tako što se jedan operand smješta u memoriju, pa bi moglo biti:

Na primjer, sljedeća kombinacija instrukcija:

lw $8,Astart($19)

add $16,$17,$18

bi se mogla zamijeniti instrukcijom:

addm $16,$17,Astart($19) #$16=$17+Memory[*$19+Astart*]

ali bi izgleda ovo ipak zbog veličine adresa uvećalo dužinu instrukcija i broj takvih instrukcija, tako da je prva varijanta *isplativija* sa aspekta brzine..

Da bi se smanjio broj instrukcija imamo i "*increment-compare-and-BRANCH*" instrukciju kao zamjenu za "**for**" petlju.

Primjer. Inkrementirati sadrža *$19*, uporedimo ga sa sadržajem *$20*, i potom iozvršiti grananje dok je god je sadržaj *$19* manji od sadržaja *$20*. To možemo zapisati na sljedeći način:

Loop:

addi $19,$19,1 #$19=$19+1

slt $8,$19,$20 #$8=1 za $19<$20

bne $8,$0,Loop #$8≠0 znači a<b pa idi na loop

a moglo bi se i zamjeniti samo jednom moćnom instrukcijom:

icb $19,$20,Loop #$19=$19+1;if $19<$20 then goto Loop

Ova zamjena bi samo imala smisla kad je inkrementacija za 1 (a ne za 4 naprimjer) i kad imamo operaciju "<" a ne recimo "≤", pa je stoga nekad ipak bolji prvi kod.

3.8 OBJEDINIMO DOSADAŠNE ZNANJE

Pogledajmo primjer zamjene dvije lokacije u memoriji kroz proceduru "**swap**". U C-u bi ova procedura bila zapisana na sljedeći način:

```
swap (int v[ ], int k)
{
        int temp;
        temp=v[k];
        v[k]= v[k+1];
        v[k+1]=temp;
}
```

Ranije smo rekli da se parametri procedura smještaju u registrima *$4,$5,$6,$7*. Stoga parametre *v* i *k* procedure swap smjestimo u *$4* i *$5*. Lokalnu promjenljive *temp* smještamo u *$15*.

Za dio programa, zapisanog u **C**-u :

```
temp= v[k];
v[k]= v[k+1];
```

v[k+1]=temp;

odgovarajući asemblerski kod je:

```
add $2,$4,$5   # $2=v+k, $2 ima adresu za v[k]
lw  $15,0($2)  # $15 je temp=v[k]
lw  $16,1($2)  # $16= v[k+1] – sljedeći element od v
sw $16,0($2)   # v[k]=reg $16
sw  $15,1($2)  # v[k+1]=reg $15(temp)
```

Ovdje, međutim, moramo indeks *k* množiti sa 4, a takođe i *v[k]* inkrementirati sa 4 da dobijemo *v[k+1]* pa će ispravan zapis biti kako slijedi:

```
muli  $2,$5,$4   # $2=k*4
add   $2,$4,$2   # $2=v+(k*4), $2 ima adresu v[k]
lw    $15,0($2)  #$15(temp)= v[k]
lw    $16,4($2)  # $16= v[k+1]
sw    $16,0($2)  # v[k]=$16
sw    $15,4($2)  # v[k+1]=$15(temp)
```

Preostaje nam kod za čuvanje registara *$2,$15* i *$16* koje smo promijenili. Registar *$29* sadrži **STACK** *pointer*, i trebamo ga podesiti na 3*4=12 bajtova prije čuvanja ovih riječi:

addi $29,$29,-12 (jer "**STACK**" ide od većih ka nižim adresama).

Čuvanje starih vrijednosti registara na STACK-u se izvršava na sljedeći način:

```
sw  $2,0($29)    #sačuvaj $2 u STACK
sw  $15,4($29)   #sačuvaj $15 u STACK
sw  $16,8($29)   #sačuvaj $16 u STACK
```

a na kraju procedure ih vraćamo sa STACK-a u registre u kojima su i ranije bili zapisani. Primijetimo da gornje četiri komande idu na samom početku procedure, dok se povratak sadržaja registara sa STACK-a izvršava na samom kraju procedure.

```
lw   $2,0($29)   # restore $2 iz STACK-a
lw   $15,4($29)  #vrati $15 iz STACK-a
lw   $16,8($29)  #vrati $16 iz STACK-a
addi $29,$29,12  #restore STACK pointer
```

Kako procedura «**swap**» ne poziva druge procedure, povratak na pozivajući proceduru iz procedure swap se izvršava naredbom **jr *$31***.

```
jr $31_ #povratak na poziv procedure
```

Znači, proceduri zapisanoj sa 8 linija u **C**-u, odgovara 17 linija u asemblerskom kodu.

Pogledajmo sada i duži primjer koji poziva "**swap**" proceduru odozgo i sortira 10000 cijelih brojeva. U C-u bi kod bio zapisan kao što slijedi u nastavku:

```
int v[10 000];
sort (int v[ ], int n)
{
int  i, j;
```

```
for (i=0; i<n; i=i+1) {
    for (j=i-1; j>=0 && v[j]> v[j+1]; j=j-1) {
        swap (v, j);
    }
  }
}
```

U petlji **for** je prvo inicijalizacija *i=0*, pa test *i<n*, pa iteracija *i=i+1*.

Dva parametra procedure **sort** (***v*** i ***n***) su u registrima *$4* i *$5*, pa uzimamo *$19* za promjenljivu **i**, kao i registar *$17* za promjenljivu **j.**

Razvijmo sada kod od spolja ka sredini:

Prva **for** petlja:

for (i=0; i<n; i=i+1) $\Rightarrow$

```
add $19,$0,$0  # i=0,
```

a za inkrementaciju *addi $19,$19,1 #i=i+1*

Za test **i<n** $\Rightarrow$

```
for1tst: slt  $8,$19,$5  # $8=0 za $19≥$5(i≥n)
beg  $*,$0,exit1  #idi na exit1 za $19≥$5(i≥n)
```

znači "*set on less than*" setuje *$8*=1 za *$19<$5*, a *$8*=0 za drugo. Na dno petlje se vraćamo na test sa:

```
j  for1tst  #skok na test druge petlje (kruga)
Exit1:
```

Dakle sada cjelokupna **for** petlja izgleda ovako:

```
add  $19,$0,$0    # i=0
for1tst:  slt $8,$19,$5  # $8=0 za i≥n
bne  $8,$0,exit1  # idi na exit1 za i≥n
...
(tijelo druge petlje)
...
addi  $19,$19,1  # i=i+1
j  for1tst          #skok na test spoljne druge petlje (kruga)
Exit1:
```

Petlja po promjenljivoj **j**, izgleda slično samo sa 2 uslova:

```
addi  $17,$19,-1   # j=i-1
for2tst: slti $8,$17,0  # $8=1 za j<0
bne $8,$0,exit2  # idi na exit2 za j<0
muli  $15,$17,4   # $15=j*4
add   $16,$4,$15   # $16=v+j
lw  $24,0($16)      # $24= v[j]
lw  $25,4($16)      # $25= v[j+1]
```

```
slt  $8,$25,$24      #$8=0 za $25≥$24
beq $8,$0,exit2     # idi na exit2 za $25≥$24
...
(tijelo druge petlje)
...
addi  $17,$17,-1   # j=j-1
j  for2tst              # skok na unutrašnju petlju
exit2:
```

Ovdje je problem što "**sort**" procedura treba vrijednosti iz *$4* i *$5*, a "**swap**" isto treba smjestiti svoje parametre u te iste registre. Zato se ranije *$4* i *$5* kopiraju u *$18* i *$20* sa:

```
move $18,$4   # kopiraj parametar iz $4 u $18
move $20,$5   # kopiraj parametar iz $5 u $20
```

pa onda **swap** parametri idu u njih kako slijedi

```
move $4,$18   # prvi swap parametar je v
move $5,$17   # prvi swap parametar je j
```

Još je ostalo sačuvati i ponovo vratiti registre (**calee save**). Adresu povratka čuvamo u *$31* kad **sort** poziva drugu proceduru. Ostali registri koje koristimo su *$15, $16, $17, $18, $19, $20, $24 i $25*

```
sort:  addi $29,$29,-36  # pravljenje prostora u STACK-u za 9 registara
sw $15,0($29)      # čuvaj $15 u STACK
sw $16,4($29)      # čuvaj $16 u STACK
sw $17,8($29)      # čuvaj $17 u STACK
:     :       :                  :
sw $25,28($29)     # čuvaj $25 u STACK
sw $31,32($29)     # čuvaj $31 u STACK
```

pa ih na kraju procedure samo sa **lw** vratimo, i onda:

```
addi  $29,$29,36   # restore STACK pointer
jr   $31                # povratak na poziv rutine
```

Ovdje, znači, za 11 linija u **C**-u, imamo 44 u asembleru, pa je jasna ušteda!!!

3.9 NIZOVI ILI POINTERI?

Pogledajmo proceduru "čišćenja" dijela riječi iz memorije. Neka su parametri "*array*" i "*size*" u *$4* i *$5*, a neka "*i*" ide u *$2*. Inicijalizacija "*i*" za **for** petlju je:

```
move $2,$0    #i=0,
```

Sad stavimo da je *array[i]* = 0, a moramo mu dati i adresu pomnoženu sa 4 da dobijemo adresu bajta:

```
loop1: muli  $14,$2,4   # $14=i*4,
```

pa pošto je početna adresa niza u registru, dodajemo je indeksu

```
add  $3,$4,$14  # $3=adresa od array[i],
```

i sad mu dajemo vrijednost "0" sa

sw $0,0($3) # array[i]=0,

sad inkrementiramo *i*

addi $2,$2,1 # i=i+1,

pa test *slt $6,$2,$5 # $6=(i<size)*

bne $6,$0,loop1 #idi na loop1 za i<size

Posmatrajmo sada pažljivo sljedeće dvije procedure:

C kod: **clear1(int array[ ], int size)** → *koristi indekse niza*

{ *ova procedura radi dok je size>0*

```
   int i;
   for (i=0; i<size; i=i+1)
       array[i]=0;
}
```

clear2 (int * array, int size) → *koristi pointere*

```
{
    int  *p;
    for (p=&array[0]; p<&array[size]; p=p+1)
        *p=0;
}
```

"**&**"- indicira adresu varijable i odgovara objektu označenom sa "*****". Ovdje se prvi elemenat niza pridružuje *pointeru* **p**. Za cjelobrojni *pointer* inkrementacija je za 4

Druga procedura smješta "*array*" i "*size*" u *$4* i *$5*, a *pointer* "**p**" u *$2*. Prvo se **p**-u pridruži prva vrijednost niza:

move $2,$4 # p=adresa od array[0]

⇒ **for** petlja:*loop2: sw $0,0($2) # memory[p]= 0*

⇒ inkrementacija: *addi $2,$2,4 # p=p+4*

⇒ loop test: *muli $14,$5,4 # $14=size*4* - nalazi adresu zadnjeg člana niza i taj proizvod dodamo početnoj adresi niza da dobijemo adresu zadnjeg člana niza

⇒ *add $3,$4,$14 # $3=adresa od array[size]*

Sada vidimo da li je **p**<zadnjeg člana niza

slt $6,$2,$3 # $6=(p<array[size])

bne $6,$0,loop2 # za (p<array[size]) idi na loop2

Pretpostavka je da je *size*>0, pa cijela procedura izgleda:

```
    move   $2,$4  # p=adresa od  array[0]
loop2: sw $0,0($2)  # memory[p]= 0
    addi $2,$2,4   # p=p+4
    muli  $14,$5,4   # $14=size*4
    add $3,$4,$14  # $3=adresa od array[size]
    slt $6,$2,$3  # $6=(p<array[size])
```

```
        bne  $6,$0,loop2  # za p<array[size] idi na loop2
```

Ako izvadimo računanje adrese zadnjeg člana niza prije petlje, jer se ona ne mijenja dobijamo bolju varijantu iste procedure:

```
        move  $2,$4
        muli  $14,$5,4
        add  $3,$4,$14
loop2:  sw  $$0,0($2)
        addi $2,$2,4
        slt  $6,$2,$3
        bne  $6,$0,loop2
```

Ako je uporedimo sa prvom procedurom koja objedinjenja izgleda:

```
        move  $2,$0  # i=0
loop1:  muli  $14,$2,4  # $14=i*4
        add  $3,$4,$14  # $3=&array[i]
        sw  $0,0,($3)  # array[i]=0
        addi $2,$2,1 # i=i+1
        slt  $6,$2,$5  # $6=(i<size)
        bne  $6,$0,loop1  # za i<size idi na loop1
```

očigledno je da kako se "*i*" inkrementira i svaka adresa se mora ponovo računati, dok kod pointera imamo direktnu inkrementaciju, što smanjuje broj iteracija sa 6 na 4.

3.10 RIJEŠENI ZADACI ZA VJEŽBANJE

1. Date su sljedeće naredbe u programskom jeziku C:

   a) a = b–c–d;
   b) a = b;
   c) a = –a;
   d) a = 0.

   Napisati ove naredbe u MIPS asemblerskom jeziku pod pretpostavkom da su promjenljivima **a**, **b**, **c** i **d** dodijeljeni registri $16, $17, $18 i $19, respektivno.

   Rješenje:

   a) Izraz se ne može riješiti direktno, zbog toga što MIPS aritmetičke instrukcije imaju tačno 3 operanda. Stoga ćemo najprije oduzeti **c** od **b**, a onda od toga oduzeti **d**, tj.: a = (b–c)–d. Koristićemo privremeni registar $8 za smještanje međurezultata b-c.

```
sub $8, $17, $18        # u registar $8 smještamo b-c
sub $16, $8, $19        # u registar $16 (promjenljiva a) smještamo (b-c)-d.
```

   Prva razlika u odnosu na više programske jezike je ta što su operandi MIPS instrukcija registri, kojih u našem slučaju ima **32** (u oznaci $0, $1, ..., $31). S druge strane, operandi viših jezika su promjenljive, kojih može biti proizvoljno mnogo. Dužina registara, kao i dužina jedne memorijske riječi, je **32** bita.

Druga razlika u odnosu na većinu viših programskih jezika je ta što MIPS kod poštuje pravilo programske linije, tj. svaka instrukcija MIPS koda se nalazi u zasebnom redu. Komentari, ukoliko ih ima, se završavaju na kraju reda u kojem su počeli.

b) **a=b** ćemo napisati kao **a=b+0**. Vrijednost 0 dobijamo kao sadržaj registra $0 (programer ne može promijeniti sadržaj registra $0)

```
add $16, $0, $17        # u registar $16 smještamo vrijednost b+0=b.
```

Na ovaj način smo došli do instrukcije **move $16, $17** koja kopira sadržaj jednog registra u drugi. Iako nije hardverski implementirana, MIPS asembler prihvata ovu instrukciju. Ovakve instrukcije se nazivaju *pseudoinstrukcijama*.

c) **–a** ćemo napisati kao **0-a**, pa imamo:

```
sub $16, $0, $16        # u registar $16 smještamo vrijednost –a=0-a.
```

d) **a=0** ćemo dobiti kao **a=0+0**:

```
add $16, $0, $0 # u registar $16 smještamo vrijednost a=0+0.
```

2. Dat je dio koda u programskom jeziku C:

$$C[i] = A[i] – B[i]$$

a) Napisati ovu naredbu u MIPS asemblerskom jeziku pod pretpostavkom da su početne adrese nizova A, B i C: 1000, 1400 i 1800, respektivno. Pretpostaviti da registar $18 sadrži vrijednost 4*i. Kao privremene registre koristiti $19 i $20.

b) Kako izgleda format instrukcija dobijenih pod a)?

Rješenje:

a)
```
lw $19, 1000($18)       # i-ti element niza A u registar $19
lw $20, 1400($18)       # i-ti element niza B u registar $20
sub $20, $19, $20       # oduzimanje tih brojeva i rezultat u registar $20
sw $20, 1800($18)       # registar $20 u i-ti element niza C.
```

b) Izgled polja kod aritmetičkih instrukcija (R-tip) i instrukcija za prenos podataka (I-tip):

op operacija instrukcije;
rs registar prvog operanda;
rt registar drugog operanda;
rd registar u koji se smješta rezultat operacije;
shamt vrijednost za koju se shift-uje vrijednost registra (ovdje se ne koristi);
funct funkcijsko polje i ono bira varijantu operacije iz *op* polja.

Kod *lw* i *sw* index niza ide u *rs*, a registar u koji se smješta podatak iz memorije je *rt*.

lw $19, 1000($18) **35 18 19 1000** (dekadno)

1000 1110 0101 0011 0000 0011 1110 1000 (binarno)

sub $20, $19, $20 **0 19 20 20 0 34** (32 za sabiranje umjesto 34)
0000 0010 0111 0100 1010 0000 0010 0010

sw $20, 1800($18) **43 18 20 1800**
1010 1110 0101 0100 0000 0111 0000 1000

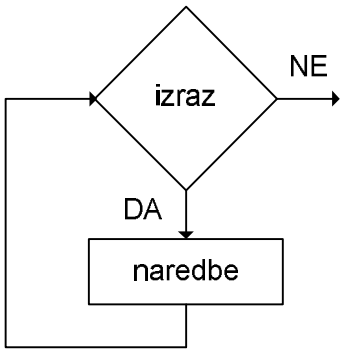
3. Dat je dio koda u programskom jeziku C kojim se vrši sabiranje prvih 100 prirodnih brojeva:

```
s=0;
i=1;
while (i<101)
{
      s=s+i;
      i=i+1;
}
```

Napisati ovaj dio koda u MIPS asemblerskom jeziku pod pretpostavkom da su promjenljivim **s** i **i** dodijeljeni registri $15 i $16, respektivno. Kao privremeni registar koristiti $8.

While petlja principijelno:

```
while(izraz)
{
      naredbe
}
```




```
        add  $15, $0, $0          # s=0;
        addi $16, $0, 1           # i=1;
Loop:   slti $8, $16, 101         # Ako je i<101 onda $8=1
        beq  $8, $0, Exit         # Ako je $8=0 izađi iz petlje
        add  $15, $15, $16        # s=s+i;
        addi $16, $16, 1          # i=i+1;
        j  Loop                   # go to na Loop
Exit:
```

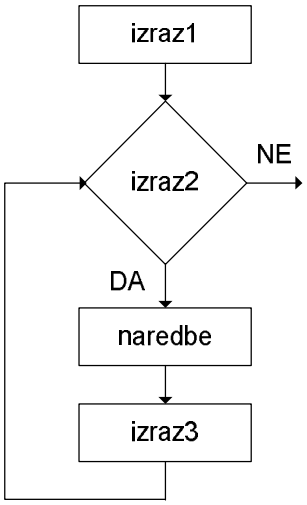
4. Dat je dio koda u programskom jeziku C koji kopira niz **A** u niz **B** i koji broji koliko ima elemenata u nizu A koji su različiti od 0. Dužina niza **A** je 20 elemenata.

```
br=0;
for (i=0; i<20; i++)
{
        B[i]=A[i];
        if (A[i]!=0)
                br=br+1;
}
```

Napisati ovaj dio koda u MIPS asemblerskom jeziku pod pretpostavkom da su početne adrese nizova **A** i **B** - 1000 i 1400, respektivno, kao i da registar $14 sadrži promjenljivu **br**, a registar $15 promjenljivu **i**. Kao privremene registre koristiti $8, $25 i $26.

For petlja principijelno:

```
for(izraz1; izraz2; izraz3)
{
        naredbe
}
```




```
        add  $14, $0, $0          # br=0;
        add  $15, $0, $0          # i=0;
Loop:   slti $8, $15, 20          # Ako je i<20 onda $8=1
        beq  $8, $0, Exit         # Ako je $8=0 ($8≠1) izađi iz petlje
        muli $25, $15, 4          # U $25 smještamo 4*i; memorija byte-adresibilna
        lw   $26, 1000($25)       # U $26 smještamo A[i]
        sw   $26, 1400($25)       # $26 smještamo u B[i], B[i]= A[i]
        beq  $26, $0, L1          # Ako je A[i]=0 skačemo na labelu L1
        addi $14, $14, 1          # br=br+1
L1:     addi $15, $15, 1          # i=i+1
        j Loop                    # go to na Loop
```

Exit:

5. Data je procedura u programskom jeziku C koja inicijalizuje članove niza cijelih brojeva **A**, dužine **N**, na 0. Napisati odgovarajući MIPS kod.

```
Clear(int A[], int N)
{
   int i;
   for(i=0; i<N; i++)
   A[i]=0;
}
```

Prilikom prevođenja koda iz programskog jezika C u MIPS kod pridržavamo se sljedeća tri generalna koraka:

I. Alokacija registara za programske promjenljive

Pošto se ulazni argumenti procedura prosljeđuju preko registara $4, $5, $6 i $7, a naša funkcija ima dva ulazna argumenta, onda su tim promjenljivima dodijeljeni registri $4 i $5, tj. A↔$4 i N↔$5. U okviru same procedure jedina deklarisana promjenljiva je **i**, pa njoj možemo dodijeliti registar $15.

II. Pisanje koda tijela procedure

```
        move $15, $0        # i=0;
Loop:   slt $8, $15, $5     # Ako je i<N onda $8=1
        beq $8, $0, Exit    # Ako je $8=0 izaći iz petlje
        muli $16, $15, 4    # $16=4*i - u $16 prava vrijednost pomjeraja
        add $16, $16, $4    # U $16 adresa i-tog člana niza A[i]
        sw $0, 0($16)       # A[i]=0
        addi $15, $15, 1    # i=i+1
        j Loop              # go to na Loop
Exit:
```

III. Čuvanje sadržaja registara tokom poziva procedura

U proceduri mijenjamo sadržaj registara $15, $8 i $16, pa se prije bilo kakve promjene njihov sadržaj mora smjestiti na stek radi čuvanja njihovih originalnih vrijednosti. Registar $29 sadrži pokazivač na stek, pa se on mora umanjiti za 3*4=12 bajta, kako bi se napravilo prostora za ova tri registra:

addi $29, $29, -12# Stek raste od viših adresa ka nižim i otuda -

Čuvanje starih vrijednosti u registrima $15, $8 i $16:

```
sw $15, 0($29)      # $15 na stek
sw $8, 4($29)       # $8 na stek
sw $16, 8($29)      # $16 na stek
```

Pošto naša procedura ne poziva nijednu drugu proceduru onda nema potrebe da se $31 smješta na stek.

Nakon završetka procedure vraćamo sa steka stare vrijednosti registara $15, $8 i $16 i usklađujemo pokazivač na stek:

```
lw $15, 0($29)      # sa steka u $15
lw $8, 4($29)       # sa steka u $8
lw $16, 8($29)      # sa steka u $16
addi $29, $29, 12   # usklađivanje steka
```

Posljednji korak je vraćanje u proceduru koja je pozvala našu proceduru, i to na instrukciju prvu posle poziva naše procedure:

```
        jr $31              # vraćanje u pozivajuću proceduru.
```

Dakle, **ukupan kod** bi bio:

```
Clear:  addi $29, $29, -12
        sw $15, 0($29)      # $15 na stek
        sw $8, 4($29)       # $8 na stek
        sw $16, 8($29)      # $16 na stek
        add $15, $0, $0     # i=0;
Loop:   slt $8, $15, $5     # Ako je i<N onda $8=1
        beq $8, $0, Exit    # Ako je $8=0 izađi iz petlje
        muli $16, $15, 4    # $16=4*i - u $16 prava vrijednost pomjeraja
        add $16,$16,$4      # U $16 adresa i-tog člana niza A[i]
        sw $0, 0($16)       # A[i]=0
        addi $15, $15, 1    # i=i+1
        j  Loop             # go to na Loop
Exit:   lw  $15, 0($29)     # sa steka u $15
        lw  $8, 4($29)      # sa steka u $8
        lw  $16, 8($29)     # sa steka u $16
        addi $29, $29, 12   # usklađivanje steka
        jr  $31             # vraćanje u pozivajuću proceduru.
```

Napomena: Korišćenje instrukcije **lw** za učitavanje **int** promjenljivih, koje obično zauzimaju 2 bajta. **lb**-load byte, **lh**-load halfword (2 bajta), **ld**-load double-word (8 bajta).

6. Data je procedura u programskom jeziku C koja broji koliko ima članova niza cijelih brojeva A koji su veći ili jednaki 10 i manji od 24. Dužina niza A je N.

```
int brojanje(int A[], int N)
{
   int i, br=0;
   for(i=0;i<N;i++)
      if(A[i]>=10 && A[i]<24)
         br++;
   return br;
}
```

I. Alokacija registara

Pošto se argumenti prosljeđuju preko registara $4, $5, $6 i $7, a funkcija ima dva ulazna argumenta, njima su pridruženi registri $4 i $5, tj. A↔$4 i N↔$5. Promjenljivima **i** i **br** su dodijeljeni registri $15 i $16, respektivno.

II. Kod tijela procedure

```
        add $16, $0, $0     # br=0;
        add $15, $0, $0     # i=0;
Loop:   slt $8, $15, $5     # Ako je i<N onda $8=1
        beq $8, $0, Exit    # Ako je $8=0 izađi iz petlje
        muli $8, $15, 4     # $8=4*i - u $8 prava vrijednost pomjeraja
        add $8, $8, $4      # U $8 adresa i-tog člana niza A[i]
```

```
        lw  $9, 0($8)          # U $9 član niza A[i]
        slti $10, $9, 10       # Ako je A[i]<10 onda $10=1
        bne $10, $0, Next      # Ako je $10=1 idi na Next
        slti $10, $9, 24       # Ako je A[i]<24 onda $10=1
        beq $10, $0, Next      # Ako je $10=0 idi na Next
        addi $16, $16, 1       # br=br+1
Next:   addi $15, $15, 1       # i=i+1
        j Loop                  # go to na Loop
Exit:   add $2, $16, $0        # u $2 vrijednost koju vraća funkcija – return br;
```

III. Čuvanje sadržaja registara tokom poziva procedura

U proceduri mijenjamo sadržaj registara $15, $16, $8, $9 i $10 pa se prije bilo kakve promjene njihov sadržaj mora smjestiti na stek radi čuvanja njihovih originalnih vrijednosti. Registar $29 sadrži pokazivač na posljednju zauzetu lokaciju steka, pa se on mora umanjiti za 4*5=20 B, kako bi se napravilo prostora za ovih 5 registara:

```
addi $29, $29, -20     # Stek raste od viših adresa ka nižim i otuda –
```

Čuvanje starih vrijednosti u registrima $15, $16, $8, $9 i $10:

```
sw  $15, 0($29)        # $15 na stek
sw  $16, 4($29)        # $16 na stek
sw  $8, 8($29)         # $8 na stek
sw  $9, 12($29)        # $9 na stek
sw  $10, 16($29)       # $10 na stek
```

Obzirom da procedura brojanje ne poziva nijednu drugu proceduru, nema potrebe da se $31 smješta na stek.

Nakon završetka procedure vraćamo sa steka stare vrijednosti registara $15, $16, $8, $9 i $10 i usklađujemo pokazivač na stek:

```
lw $15, 0($29)         # sa steka u $15
lw $16, 4($29)         # sa steka u $16
lw $8, 8($29)          # sa steka u $8
lw $9, 12($29)         # sa steka u $9
lw $10, 16($29)        # sa steka u $10
addi $29, $29, 20      # usklađivanje steka
```

Posljednji korak je vraćanje u proceduru koja je pozvala našu proceduru, i to na instrukciju prvu poslije poziva naše procedure:

```
jr $31                          # vraćanje u pozivajuću proceduru.
```

Dakle, **ukupan kod** bi bio:

```
brojanje: addi $29, $29, -20
          sw $15, 0($29)       # $15 na stek
          sw $16, 4($29)       # $16 na stek
          sw $8, 8($29)        # $8 na stek
          sw $9, 12($29)       # $9 na stek
```

```
        sw $10, 16($29)      # $10 na stek
        add $16, $0, $0      # br=0;
        add $15, $0, $0      # i=0;
Loop:   slt $8, $15, $5      # Ako je i<N onda $8=1
        beq $8, $0, Exit     # Ako je $8=0 iza]i iz petlje
        muli $8, $15, 4      # $8=4*i – U $8 prava vrednost pomeraja
        add $8, $8, $4       # U $8 adresa i-tog člana niza A[i]
        lw $9, 0($8)         # U $9 član niza A[i]
        slti $10, $9, 10     # Ako je A[i]<10 onda $10=1
        bne $10, $0, Next    # Ako je $10=1 idi na Next
        slti $10, $9, 24     # Ako je A[i]<24 onda $10=1
        beq $10, $0, Next     # Ako je $10=0 idi na Next
        addi $16, $16, 1     # br=br+1
Next:   addi $15, $15, 1     # i=i+1
        j Loop               # go to na Loop
Exit:   add $2, $16, $0      # u $2 vrijednost koju vraća funkcija – return br;
        lw $15, 0($29)       # sa steka u $15
        lw $16, 4($29)       # sa steka u $16
        lw $8, 8($29)        # sa steka u $8
        lw $9, 12($29)       # sa steka u $9
        lw $10, 16($29)      # sa steka u $10
        addi $29, $29, 20     # usklađivanje steka
        jr $31               # vraćanje u pozivajuću proceduru.
```

7. Data je procedura u programskom jeziku C koja za ulazne argumente ima string **S** i dva karaktera, **c1** i **c2**, i koja svaku pojavu tih karaktera u stringu S zamjenjuje karakterom ′+′.

```
void zamjena(char S[], char c1, char c2)
{
  int i=0;
 while (S[i]!='\0')
{
  if(S[i]==c1 || S[i]==c2)
     S[i]='+';
  i++;
  }
}
```

I. Alokacija registara

S↔$4, c1↔$5 i c2↔$6. Promjenljivoj **i** je dodijeljen registar $15.

II. Kod tijela procedure

```
        add $15, $0, $0      # i=0;
Loop:   add $17, $4, $15     # U $17 adresa A[i]
        lb $18, 0($17)       # U $18 karakter A[i]
        beq $18, $0, Exit    # Ako je $18=0 to je kraj stringa i izađi iz petlje
        beq $18, $5, L1      # Ako je A[i]==c1 idi na L1
        bne $18, $6, Next     # Ako je A[i]!=c2 idi na Next
L1:     addi $18, $0, '+'    # U $18 karakter '+'
```

```
            sb $18, 0($17)          # A[i]='+'
Next:       addi $15, $15, 1        # i=i+1
            j Loop                  # go to na Loop
Exit:
```

III. Čuvanje sadržaja registara tokom poziva procedura