3.1 HARDVERSKA PODRŠKA STRUKTURNOM PROGRAMIRANJU (UPOTREBI PROCEDURA – POTPROGRAMA)

Implementacija strukturnog programiranja obezbjedjuje se upotrebom procedura/potprograma. Na taj način postiže se rješavanje problema njegovom podjelom na više manjih problema koji se rješavaju unutar procedura/potprograma. Medjutim, ovo je samo iz programerske vizure sagledana prednost upotrebe procedura/potprograma. Hardverska prednost ogleda se u činjenici da se procedure/potprogrami u memoriji računara zapisuju jedanput, a upotrebljavaju se višestruko (koliko god puta je to potrebno). Alternativa ovom prilazu, predstavljalo bi višestruko zapisivanje procedura/potprograma u memoriji računara (kad god one trebaju da se izvrše), čime bi se višestruko rasipali resursi računara u poredjenju sa strukturnim programiranjem.

Procedure/potprogrami su posebne cjeline koje se izvršavaju kad god ih program (često nazivan glavnim programom) pozove. Nakon izvršavanja procedure/potprograma, program nastavlja sa izvršavanjem (izvršavanjem instrukcija koje slijede nakon poziva procedure/potprograma). U tom cilju moraju se ispuniti sljedeći hardverski zatjevi:

1. Definisati naredbu/instrukciju koja omogućava prelazak/skok na proceduru koja se poziva (pozvana procedura), ali i povratak nazad – na proceduru iz koje se vrši poziv (pozivajuća procedura) i to na instrukciju pozivajuće procedure koja neposredno slijedi nakon instrukcije poziva pozvane procedure,
2. Obezbjedjivanje mogućnosti poziva više procedura po dubini,
3. Protok parametara/argumenata izmedju procedura (pozivajuće i pozvane procedure).

Krenimo redom.

1. Naredba/instrukcija kojom se poziva procedura treba da omogući skok na pozvanu proceduru (na adresu njene prve naredbe/instrukcije), ali treba da omogući i povratak nazad (na pozivajuću proceduru), odnosno na naredbu pozivajuće procedure koja slijedi neposredno nakon instrukcije poziva pozvane procedure (ovim, instrukcija poziva procedure suštinski treba da sačuva vezu sa pozivajućom procedurom). Pojednostavljeno, ova naredba trena da omogući skok (eng. *jump*) na pozvanu proceduru i povezivanje (eng. *link*) da pozivajućom procedurom. Shodno tome, instrukcija se naziva ***jal***, kao skraćeni naziv od pojmova ***J****ump* ***A****nd* ***L****ink* (skoči i poveži) i njena sintaksa je:

   jal Procedure_Address

   gdje Procedure_Address označava adresu pozvane procedure (adresu njene prve instrukcije).

   Primijetimo da je sintaksom instrukcije *jal* ekplicitno označen JUMP dio instrukcije. Postavlja se pitanje: što je sa LINK dijelom naredbe? Link dio naredbe obezbjedjuje se čuvanjem adrese instrukcije pozivajuće procedure na koju treba izvršiti povratak (označimo je sa Add+4, gdje je Add adresa instrukcije *jal* Procedure_Address) u registru $31. Na ovaj način, registar $31 upotrebljava se isključivo za čuvanje *povratne adrese* na pozivajuću proceduru i zahtijeva se od kompajlera, odnosno od programera u asemblerskom obliku, uzdržanost u pogledu upotrebe registra $31, na isti način na koji se očekuje uzdržanost u pogledu upotrebe registra $1.

   Postavlja se, takodje, pitanje: *Kako računar može znati koju adresu treba da smjesti u registar* $31?

ODGOVOR: Pored 32 procesorska registra ($0–$31), MIPS računarski sistem raspolaže sa sistemskim registrom – PC registrom[1]. U njemu se čuva adresa trenutno izvršavane naredbe (prilikom izvršavanja naredbe *jal* Procedure_Address, to je adresa lokacije memorije računara u kojoj je ova naredba zapisana, a mi smo je označili sa Add). Sa druge strane, na kraju izvršavanja naredbe, u PC registru treba da bude upisana adresa naredbe koja se sljedeća izvršava, odnosno u slučaju posmatrane instrukcije *jal* Procedure_Address, na kraju njenog izvršavanja u PC registru treba da bude upisana adresa Procedure_Address. Napomenimo, medjutim, da se MIPS instrukcije izvršavaju u 3 do 5 koraka (ovo će detaljno biti elaborirano prilikom razmatranja višetaktne implementacije procesora), tako da adresa Procedure_Address mora biti upisana u PC registru najkasnije u posljednjem od ovih koraka. U medjuvremenu, u prvom koraku izvršavanja svih MIPS instrukcija (samim tim, i u prvom koraku izvršavanja instrukcije *jal* Procedure_Address) izračunava se adresa PC+4 (u slučaju instrukcije *jal* Procedure_Address, PC+4 uzima vrijednost Add+4), smješta se potom u PC registar[2] i može, po potrebi (ukoliko je riječ o izvršavanju instrukcije *jal*), biti prenijeta u registar $31.

Pojednostavimo krajnje prethodno izlaganje. U tom cilju, posmatrajmo procedure A i B i pretpostavimo da se procedura B poziva iz procedure A, te da B takodje oynačava i adresa prve instrukcije procedure B. Procedura B se stoga poziva (iz procedure A) instrukcijom

jal B

dok se instrukcijom

jr $31

koja se zapisuje na samom kraju procedure B (posljednja instrukcija procedure B), vraćamo na izvršavanje procedure A (od instrukcije koja neposredno slijedi poslije instrukcije *jal* B), pošto je, po svojoj funkciji, u registaru $31 sačuvana povratna adresa na proceduru A.

2. Apsolutno realna mogućnost je da se iz prvopozvane procedure (procedure koja je već pozvana sa svog nad-nivoa) poziva nova procedura. Stoga je neophodno obezbijediti mogućnost poziva više procedura po dubini.

   U cilju razmatranja ovog problema, posmatrajmo 3 procedure, A, B i C, pri čemu procedura A poziva proceduru B (A je, stoga, prvo-pozivajuća, a B – prvo-pozvana procedura), a potom, prilikom izvršavanja procedure B, iz procedure B se poziva procedura C. Drugim riječima, s' aspekta procedura A i B, A je pozivajuća, a B je pozvana procedura, dok je s'

---

[1] Naziv PC je dio istorijskog nasljedja, a predstavlja skraćenicu od engleskih riječi *Program Counter* (programski brojač na našem jeziku). Potiče iz vremena kada se program pravolinijski, odnosno sukcesivno izvršavao (naredba po naredbu, bez mogućnosti izvršavanja naredbi uslovnog i bezuslovnog skoka). S' današnjeg aspekta, ovom registru više bi odgovarao naziv Instruction Address Register. Ipak, MIPS arhitekture zadržale su naziv PC.

[2] Nakon izračunavanja, vrijednost PC+4 smješta se u PC registar, pošto se najčešće izvršavaju sukcesivne instrukcije (preko 95% svih izvršavanih instrukcija su sukcesivne instrukcije) i PC+4 će u tom slučaju predstavljati adresu sljedeće instrukcije. Uz to, sve i ukoliko nije riječ o instrukciji koja se sukcesivno izvršava (manje od 5% mogućnosti da je zaista tako), već o nekoj od instrukcija uslovnog ili bezuslovnog skoka, na raspolaganju su 2 do 4 sljedeća koraka da se napravljena greška (PC←PC+4) ispravi.

Takodje, ovim se postiže ušteda u vremenu izvršavanja instrukcija i to ušteda od jednog koraka po instrukciji. Naime, alternativa ovom prilazu bila bi potpuno izvršavanje instrukcije, a tek potom odredjivanje adrese sljedeće instrukcije, što bi uzelo čitav korak više. Drugim riječima, ušteda postignuta akcijom PC←PC+4, izvršenom na početku izvršavanja proizvoljne instrukcije iznosi od 20% vremena (u slučaju instrukcija koje za svoje izvršavanje zahtijevaju 5 koraka) do 33% vremena (u slučaju instrukcija koje za svoje izvršavanje zahtijevaju 3 koraka).

aspekta procedura B i C, B – pozivajuća, a C – pozvana procedura. Odnosno, saglasno protokolu poziva procedura (funkcionisanju instrukcije *jal*), opisanom pod 1., povratna adresa sa procedure B na proceduru A, ali i povratna adresa sa procedure C na proceduru B čuvaju se u istom registru (registru $31). Medjutim, čuvanjem povratne adrese sa procedure C na proceduru B u registru $31, iz istog registra će biti prebrisana povratna adresa sa procedure B na njen nad-nivo (proceduru A), odnosno biće onemogućen povratak na prvo-pozivajuću proceduru (proceduru A), te se ona neće moći izvršiti do kraja. Ovo je neželjena situacija koja mora biti prevazidjena po svaku cijenu.

U cilju njenog prevazilaženja, povratna adresa sa procedure B na proceduru A (sačuvana u registru $31) **MORA SE** sačuvati od prebrisavanja prije poziva procedure C. Ovo se obavlja pravovremenim čuvanjem (neposredno prije poziva procedure C) sadržaja registra $31 na memorijskoj strukturi nazvanoj ***stack***. Stack je suštinski dio memorije računara koji se sastoji od izvjesnog broja memorijskih lokacija. Stack ima svoje dno i vrh, ali ***poznata je samo adresa lokacije koja se nalazi na njegovom vrhu i sačuvana je u registru*** **$29** (iz ovog razloga, registar $29 obično se naziva *pokazivačem* (eng. *pointer*-om) na vrh stack-a). Shodno tome, **podaci se čuvaju na vrhu stack-a i to saglasno sljedećoj proceduri: stack se najprije uveća za jednu lokaciju (upotrebom instrukcije *addi*), a potom se podatak (koji je potrebno sačuvati) kopira u novododatu lokaciju (upotrebom instrukcije *sw*)**. Registar $29 se upotrebljava samo u svrhu čuvanj adrese vrha stack-s i preporučuje se da se ne upotrebljava u druge svrhe (slično registrima $1 i $31), odnosno od compilera i programera u asembelrkoj formi zahtijeva se da se uzdrže od njegove upotrebe (alokacije). Stack je veoma popularan u programiranju i u C programskom jeziku postoje posebne naredbe za čuvanje podataka, odnosno uzimanje podataka sa stack-a (push i pop naredbe, resspektivno).

VAŽNO ZAPAŽANJE: Shodno proceduri čuvanja podataka na stack-u, stack raste (prilikom dodavanja/čuvanja podataka na njemu) ODOZDO-NA-GORE. Podsjetimo da se, sa druge strane, memorija računara kreira na način da adrese njenih lokacija rastu u suprotnom smjeru (ODOZGO-NA-DOLJE). Shodno tome, adresa novog vrha stack-a (nakon registrovanja potrebe za dodavavanjem podatka na njemu) dobija se oduzimanjem konstante, koja pravi razliku od jedne lokacije u memoriji računara, od adrese starog vrha stack-a. Podsjetimo takodje da je, kao kod većine savremenih računarskih arhitektura, memorija MIPS arhitektura *bajt-adresabilna*, odnosno da se svaka njena 32-bitna lokacija sastoji od 4 bajta, te da konstanta 4 pravi razliku od jedne memorijske lokacije, kao i da, na isti način, konstanta 8 pravi razliku od 2 memorijske lokacije, konstanta 12 razliku od 3 memorijske lokacije, … Drugim riječima, prilikom uvećavanja stack-a za jednu memorijsku lokaciju, adresa novog vrha stack-a (pokazivač na vrh stack-a) dobija se ***oduzimanjem*** konstante 4 od adrese starog vrha stack-a, odnosno implementiranjem instrukcije

***addi* $29, $29, –4**

Primjer. Pretpostavimo da procedura A poziva proceduru B, a da potom, tokom svog izvršavanja, procedura B poziva proceduru C. Neka je B adresa procedure B (prve naredbe procedure B), a C adresa procedure C (prve naredbe procedure C). Napisati MIPS asemblerski kod za implementaciju protokola pozivanja više procedura po dubini.

Rješenje: Četiri su ključna trenutka tokom implementacije protokola poziva više procedura po dubini:

I. **Trenutak nakon pozivanja procedure B iz procedure A.** U ovom trenutku, u registru $31 smještena je povratna adresa sa procedure B na proceduru A (nazovimo je *Povratna adresa sa B na A*), a u registru $29 je pokazivač na vrh stack-a (adresa vrha stack-a).

II. **Trenutak prije nego što se iz procedure B pozove procedura C.** Ovaj trenutak se mora iskoristiti da se sačuva (od prebrisavanja iz registra $31) *Povratna adresa sa B na A*. Naime, ukoliko se propušti ovaj trenutak i dopušti se da se iz procedure B pozove procedura C, u registru $31 će biti upisana povratna adresa sa procedure C na proceduru B (nazovimo je *Povratna adresa sa C na B*) i ovim upisom će biti prepisana *Povratna*

adresa B na A. Procedura čuvanja sadržaja registra \$31 (njegov sadržaj je u ovom trenutku *Povratna adresa sa B na A*) na stack-u sastoji se iz sljedeća 2 dijela:

- Uvećavanja stack-a za jednu lokaciju u cilju stvaranja prostora na vrhu stack-a za čuvanje sadržaja registra \$31. U tu svrhu upotrebljavamo, kao što je prethodno već rečeno, instrukciju

  addi **\$29, \$29, –4**

- Čuvanja sadržaja registra \$31 (odnosno *Povratne adrese sa B na A*) na vrhu stack-a. U tu svrhu upotrebljavamo instrukciju

  sw **\$31, 0(\$29)**

  POJAŠNJENJE: Pojasnimo posljednju instrukciju i način njenog zapisivanja. Podsjetimo, u tom cilju, da je ***sw*****\$x, Astart(\$y)** sintaksa instrukcije *sw*, gdje je \$x registar čiji sadržaj je potrebno sačuvati u memoriji računara (u našem slučaju registar \$31), dok je sa Astart(\$y) zapisana adresa memorijske lokacije u kojoj se čuva sadržaj registra \$x. Primijetimo da se adresa ove memorijske lokacije, saglasno sintaksi instrukcije *sw*, sastoji iz 2 dijela: \$y – registara u kome je sačuvan indeks elementa proizvoljnog niza A koji je u memoriji računara sačuvan počev od adrese Astart. Drugim riječima, adresa lokacije u memoriji računara na kojoj se čuva sadržaj registra \$x dobija se sumom Astart+sadržaj(\$x). Medjutim, pokazivač na vrh stack-a (registar \$29) već sadrži adresu lokacije na kojoj treba sačuvati sadržaj registra \$x i samo je potrebno zapisati u formi Astart(\$y) i to tako da ovaj zapis rezultira sadržajem registra \$29. Shodno tome, adresu lokacije na kojoj čuvamo sadržaj registra \$x potrebno je zapisati u formi 0(\$29), pošto je 0+sadržaj(\$29)=sadržaj(\$29). Konačno, instrukcija ***sw*** **\$31, 0(\$29)** obezbjedjuje čuvanje sadržaja registra \$31 na vrhu stack-a.

III. **Trenutak nakon pozivanja procedure C iz procedure B.** U ovom trenutku, u registru \$31 smještena je *Povratna adresa sa C na B*, u registru \$29 je pokazivač na vrh stack-a (adresa vrha stack-a) izračunat u trenutku 2., a u memorijskoj lokaciji koja se nalazi na vrhu stack-a sačuvana je *Povratna adresa B na A*.

IV. **Trenutak nakon povratka sa procedure C na proceduru B.** Nakon izvršavanja procedure C i povratka na izvršavanje procedure B, *Povratna adresa sa C na B* više nije potrebna (obavila je svoju funkciju povratkom iz procedure C na proceduru B), već je potrebno, sa vrha stack-a, u registrar \$31 vratiti *Povratnu adresu sa B na A*, te nakon toga stack umanjiti za jednu memorijsku lokaciju. Ove akcije se obavljaju instrukcijama koje su komplementarne instrukcijama navedenim u trenutku 2. Shodno tome, procedura povratka *Povratne adrese sa B na A* sa stack-a u registar \$31 sastoji se iz sljedeća 2 dijela:

- Kopiranja *Povratne adrese sa B na A* sa vrha stack-a u registar \$31. U tu svrhu upotrebljava se instrukcija

  lw **\$31, 0(\$29)**

- Umanjivanja stack-a za jednu lokaciju. U tu svrhu upotrebljava se, kao što je prethodno već rečeno, instrukcija komeplementarna instrukciji ***addi*** **\$29, \$29, –4**, a to je instrukcija

  addi **\$29, \$29, 4**

  Primijetiomo da dodavanjem konstante +4 na stari vrh stack-a, suštinski umanjujemo stack za jednu memorijsku lokaciju. Naime, konstanta 4 pravi razliku od 4 bajta (odnosno razliku od jedne memorijske lokacije), dok njena pozitivna vrijednost doprinosi rastu memorijske adrese, odnosno uzrokuje umanjivanje stack-a (podsjetimo stack raste u suprotnom smjeru od porasta memorijskih adresa, tako da uvećavanje memorijske adrese odgovara umanjenju stack-a).

NAPOMENA: Kopiranjem *Povratne adrese sa B na A* sa vrha stack-a u registar $31, *Povratna andesa sa B na A* se ne briše sa memorijske lokacije u kojoj se nalazi. Medjutim, podešavanjem pokazivača na vrh stack-a, odnosno umanjivanjem stack-a za jednu lokaciju, memorijska lokacija u kojoj se nalazi *Povratna adresa sa B na A* neće više pripadati stack-u, tako da će biti prebrisana prvom sljedećom prilikom kada se bude ukazala potreba za čuvanjem registra $31 (ili moguće i nekog drugog registra) na stack-u.

Konačno, prethodno razmatrana procedura poziva više procedura po dubini, u MIPS asemblerskom kodu, ima sljedeći izgled:

A:	**...**		
	...		
	jal	B	# poziva se proc. B, a u reg. $31 čuva se *Povratna adresa B na A*
	...		

B:	...		
	...		# nakon ove instrukcije treba pozvati proceduru C
	addi	$29, $29, –4	# podešavanje $29/uvećava se stack za jednu memorijsku lokaciju
	sw	$31, 0($29)	# na vrhu steck-a sačuvan sadržaj reg. $31/*Povratna adresa sa B na A*
	jal	C	# poziva se proc. B, a u reg. $31 čuva se *Povratna adresa C na B*
	lw	$31, 0($29)	# $31←*Povratna adesa B na A* (sa vrha stack-a)
	addi	$29, $29, 4	# podešavanje $29/umanjuje se stack za jednu memorijsku lokaciju
	...		
	...		
	jr	$31	# posljednja naredba procedure B kojom se vraćamo na proceduru A

C:	...		
	...		
	jr	$31	# posljednja naredba procedure C kojom se vraćamo na proceduru B

3. Protok parametara/argumenata izmedju procedura (pozivajuće i pozvane procedure) obezbjedjuje se poštovanjem konvencija. U tom smislu, rezervišu[3] se registri $4–$7 za smještanje prva 4 parametra/argumenta koji se proslijedjuju pozvanoj proceduri (I parametar smješta se u registru $4, II parametar – u registru $5, III parametar – u registru $6 i IV parametar u registru $7), tako da pozivajuća procedura 'zna' u koje registere treba da smjesti prva 4 parametra sa kojima pozvana procedura treba da radi, a pozvana procedura – da 'zna' u kojim registrima se nalaze prva 4 parametra sa kojima ona treba da obavlja svoja izvršavanja. Eventualni ekstra parametri/argumenti (peti, šesti, ...) smještaju se na stak-u i indeksirani su putem stak pointera (pokazivača na vrh stack-a). Oni se upisuju (od strane pozivajuće procedure) i pronalaze na stack-u (od strane pozvane provedure) na isti način na koji se na staku čuva, odnosno vraća sadržaj registra $31.

[3] Registri $4–$7 rezervisani su za obezbjedjivanje protoka parametara/argumenata izmedju pozivajuće i pozvane procedure. Shodno tome, compileri i programeri u asemblerkom obliku treba da se uzdrže od upotrebe ovih registara u bilo koje druge svrhe, osim u svrhu obezbjedjivanja protoka parametara/argumenata izmedju pozivajuće i pozvane procedure.

Ukoliko imamo slučaj poziva više procedura po dubini, parametri/argumenti prvo-pozvane procedure biće čuvani na stack-u tokom izvršavanja drugo-pozvane procedure, dok se, istovremeno, u registrima \$4–\$7 nalaze parametri/argumenti drugo-pozvane procedure. Nakon povratka iz drugo-pozvane u prvo-pozvanu proceduru, u registre \$4–\$7 potrebno je vratiti (sa stack-a) parametre/argumente prvo-pozvane procedure. Procedura čuvanja i vraćanja parametara/argumenata odredjene procedure na stak-u, odnosno sa stack-a, prilikom poziva više procedura po dubini odgovara prethodno detaljno opisanoj proceduri čuvanja/vraćanja sadržaja registra \$31 na stack, odnosno sa stack-a.

Pored protoka patrametara izmedju procedura, poseban izazov predstavlja činjenica da se u opštem slučaju, tokom izvršavanja pozvane procedure, mogu modifikovati registri koje se upotrebljavaju od strane pozivajuće procedure. Sa druge strane, mora se obezbijediti nepromjenljivost sadržaja registara čiji sadržaj upotrebljava pozivajuća procedura i to nepromjenljivost sadržaja ovih registara nakon povratka iz pozvane u pozivajuću proceduru. Dvije su standardne konvencije za čuvanje sadržaja registara koji se upotrebljavaju od strane pozivajuće procedure, a čiji sadržaj se modifikuje tokom izvršavanja pozivane procedure:

i. **Caller-save rutina.** Ova rutina podrazumijeva da *pozivajuća (caller) procedura* treba da vodi računa o čuvanju (na stack) sadržaja svih registara koje sama upotrebljava I to prije poziva procedure, kao i o vraćanju sadržaja ovih registara (sa stack-a) nakon izvršavanja pozvane procedure. Nakon toga, pozvana (callee) procedura može upotrebljavati sve raspoložive registere, bez ograničenja i bez bojazni da će time modifikovati sadržaj nekog od registara koji upotrebljava pozivajuća procedura i čija se nepromjenljivost sadržaja (nakon povratka iz pozvane procedure) zahtijeva od strane pozivajuće procedure.

ii. **Callee-save rutina.** Ova rutina podrazimijeva da *pozvana (callee) procedura* treba da vodi računa o čuvanju (na stack) sadržaja svih registara koje sama upotrebljava, kao i o njihovom vraćanju (sa stack-a) nakon svog izvršavanja. Nakon toga, pozivajuća (caller) procedura ne treba da brine da li će sadržaj registara koje ona upotrebljava biti modifikovan tokom izvršavanja pozvane procedure.

Primijetimo da obje rutine (i caller-save i callee-save) imaju izvjesna organičenja s' aspekta optimalnog broja registara koje treba čuvati tokom izvršavanja pozvane provedure. Naime, pozivajuća (caller) procedura zna registre čiji sadržaj želi zadržati nepromjenljivim nakon povratka iz pozvane procedure, ali ne može znati da li će pozvana (callee) procedura upotrijebiti baš sve te registre i koliko od njih će uopšte upotrijebiti. Takodje, pozvana (callee) proceduda zna koje registre namjerava da upotrijebi tokom svog izvršavanja, ali ne može znati da li pozivajuća (caller) procedura upotrebljava baš sve ove registre i koliko od njih uopšte upotrebljava. Drugim riječima, implementacijom obje rutine, i caller-save i calle-save, može dovesti do nepotrebnog čuvanja sadržaja odredjenih registara na stack-u. Ukoliko se zna da *lw* i *sw* natedbe, upotrebljavane prilikom obraćanja memoriji računara (u cilju čuvanja sadržaja registara na stack-u, odnosno vraćanja njihovog sadržaja sa stack-a), predstavljaju vremenski najzahtjevnije instrukcije iz skupa MIPS instrukcije (što će biti dokazano kasnije, prilikom implementacije MIPS procesora), jednostavno se može zaključiti da implementacija caller-save i callee-save rutina može dovesti do značajnog rasipanja računarskih resursa.

Ipak, opisano rasipanje resursa značajno se može ublažiti. U tom cilju, razlikuju se 2 kategorije registara (caller-saved i callee-saved). Saglasno usvojenim konvencijama, u callee-saved registrima čuvaju se dugotrajno upotrebljavane promjenljive (promjenljive upotrebljavane u pozivajućoj proceduri i prije poziva procedure i nakon povratka iz pozvane procedure), dok se u caller-saved registrima čuvaju kratkotrajno upotrebljavane promjenljive (promjenljive upotrebljavane u pozivajućoj proceduri samo prije poziva procedure ili samo nakon povratka iz pozvane procedure). Shodno tome, tokom izvršavanja pozvane procedure, neophodno je čuvati (na stack-u) sadržaje samo callee-saved registara, dok je sadržaje caller-saved registara NEPOTREBNO čuvati, a čuvanje sadržaja callee-saved registara obavlja se callee-save rutinom (tokom izvršavanja i od strane pozvane procedure). Na ovaj način,

redukuje se potreba (brojno izražena) za čuvanjem sadržaja registara na stack-u, odnosno vraćanjem njihovog sadržaja sa stack-a.

3.2 ARITMETIČKO-LOGIČKE INSTRUKCIJE SA KONSTANTOM KAO OPERANDOM

U programiranju, veoma često se upotrebljavaju naredbe koje za jedan od svojim operanada imaju konstantu (prilikom inkrementiranja, odnosno dekrementiranja brojača iteracija u petljama, prilikom inkrementiranja/dekrementiranja indeksa vektora, prilikom inkrementiranja/dekrementiranja pokazivača na vrh stack-a, odnosno stack pointera, ...). U mnogim višim programskim jezicima, više od polovine aritmetičko-logičkih instrukcija za jedan od svojih operanada ima konstantu. Konkretno, u slučaju C compilera, 52% aritmetičko-logičkih instrukcija za jedan od svojih operanada ima konstantu, dok je taj procenat kod Spice programskog jezika čak 69%. Pošto je riječ o veoma čestom slučaju, ubrzavanje njegovog izvršavanja vodi poboljšavanju vremenskih performansi računarskog sistema. Tim problemom se bavi IV princip projektovanja hardvera računara:

IV princip projektovanja hardvera računara: *Česte slučajeve treba napraviti brzima!!*

Kao što smo naveli, upotreba konstantnih operanada u programiranju je veoma čest slučaj, te saglasno IV principu projektovanja hardvera računara, potrebno ga je što više ubrzati.

Jedan mogući prilaz implementaciji upotrebe konstantnih operanada uključivao bi *importovanje konstanti u memoriju računara* (na primjer, zajednom sa importovanjem/loadovanjem programa koji trebaju da se izvrše), *donošenje potrebne konstante* (upotrebom instrukcije *lw*) iz memorije u neki privremeni registar, te nakon toga *implementaciju zahtijevane aritmetičke ili logičke instrukcije*. Medjutim, ovaj prilaz bi zahtijevao ukupno vrijeme izvršavanja ne manje od vremena potrebnog za izvršavanja *lw* instrukcije uvećanog za vrijeme neophodno za izvršavanje zahtijevane aritmetičke ili logičke instrukcije. Kao što ćemo razmatrati kasnije, vrijeme izvršavanja koje bi odgovaralo ovom prilazu iznosilo bi 9 koraka, što je neprihvatljivo dugo vrijeme izvršavanja ukoliko se navede da aritmetičke i logičke instrukcije (instrukcije R-tipa) za svoje izvršavanje zahtijevaju samo 4 koraka. Uz to, kao što smo već naveli, upotreba instrukcija koje se obraćaju memoriji računara (kao što je instrukcija *lw*) značajno usporavaju rad računara i, u principu, preporučuje se izbjegavanje ovih instrukcija kad god je to moguće.

Iz gore navedenih razloga, u praksi se upotreba konstantnih operanada u aritmetičkim i logičkim instrukcija implementira uvodjenjem novog oblika aritmetičkih i logičkih instrukcija – *immediate instrukcija*[4], odnosno *instrukcija I-tipa*. Kod ovih instrukcija, jedan od operanada je konstanta, uz ograničenje da se konstanta (koja se u instrukcijama upotrebljava kao operand) čuva unutar same instrukcije koja se izvršava, a da njeno izvršavanje odgovara izvršavanju aritmetičkih i logičkih instrukcija (instrukcija R-tipa). Pošto odgovaraju instrukcijama R-tipa, immediate instrukcije imaju nazive koji odgovaraju nazivima odgovarajućih instrukcija R-tipa sa dodatim slovim i na kraju svog naziva, koje ukazuje da je riječ o immediate, a ne o instrukcijama R-tipa (na primjer, immediate instrukcije *addi*, *muli*, *slti* orgovaraju sljedećim instrukcijama R-tipa: *add*, *mult*, *slt*).

Sintaksa immediate instrukcija će biti ilustrovana na primjeru instrukcije *addi*. Sintaksa ove instrukcije je:

addi **$x, $y, C** **# $x=$y+C**

Primijetimo da se konstanta (konstantni operand) C čuva unutar same instrukcije i da se ona zapisuje, u simboličkom obliku instrukcije, na njenom kraju. Registar $x, x=1,...,31 označava registar u koji se upusuje rezultat sabiranja operanda sadržanog u registru $y, y=0,...,31 sa konstantom C, kao što je zapisano u komentaru gore navedene instrukcije.

---

[4] Naziv *immediate instrukcija,* odnosno instrukcija I-tipa, potiče iz činjenice da se na engleskom jeziku konstantni operandi nazivaju *immediate* ili *trenutnim* operandima (sačuvani su unutar same instrukcije i trenutno su dostupni).

Primijetimo, takodje, da format mašinskog zapisivanja instrukcije treba da uključi polja neophodna za zapisivanje sljedećih djelova instrukcije: ključne riječi/oznake instrukcije (*addi*), obilježja 2 registra (x i y), kao i konstante C. Drugim riječima, format zapisivanja immediate instrukcija treba da ima 4 polja, analogno formatu zapisivanja koji upotrebljavaju instrukcije lw i sw (Data Tranfer/Memory Reference naredbe) i dat je na sljedećem grafičkom prikazu:

Polje:	op	rs	rt	immediate
Br. bitova:	6	5	5	16
Sadržaj:	8	y	x	C

Slika 1. Format mašinskog zapisivanja immediate instrukcija.

Upotreba konstantnih (immediate) operanada prilikom poredjenja. Primjena konstantnih (immediate) operanada popularna je i u drugim slučajevima koji se mogu povezati sa aritmetičkim i logičkim instrukcijama (instrukcijama R-tipa). Konkretnije, instrukcija *slt* upotrebljava se prilikom poredjenja sadržaja registara i predtavlja instrukciju R-tipa. Immediate instrukcija *slti* predstavlja immediate oblik instrukcije *slt* uz sintaksu koja odgovara instrukcijama I-tipa, a ime sljedeći oblik:

slti **\$x, \$y, C** # $\$x = \begin{cases} 1, & \text{sadržaj } (\$y) < C \\ 0, & \text{sadržaj } (\$y) \geq C \end{cases}$

Mašinski kod *slti* instrukcije odgovara mašinskom kodu instrukcije addi, s tom razlikom što je Op=10 u slučaju slti instrukcije (kod addi instrukcije je Op=8).

NAPOMINA 1: Instrukcija *slti* kombinovana sa instrukcijama uslovnog skoka/grananja *beq* i *bne*, te uz upotrebu sadržaja registra \$0 (koji je uvijek 0), obezbjedjuje implementiranje svih vrsta grananja zahtijevanih nakon ispitivanja relativnih uslova koji mogu biti kreirani u odnosu na vrijednost konstantnog operanda C (manje od C, veće od C, manje ili jednako C, veće ili jednako C). Na primjer, par instrukcija

slti **\$x, \$y, C** # $\$x = \begin{cases} 1, & \text{sadržaj } (\$y) < C \\ 0, & \text{sadržaj } (\$y) \geq C \end{cases}$

bne **\$x, \$0, L** # skoči na naredbu sa obilježjem L ako je $\$x \neq 0$ (sadržaj (\$y) < C)

obezbjedjuje skok na instrukciju sa obilježjem L ukoliko je sadržaj registra \$y manji od konstante C, kao što se može zaključiti iz komentara napisanih pored svake od instrukcija. Ova mogućnost je u potpunosti analogna kreiranju pseudoinstrukcije *blt* implementiranjem instrukcija *stl* i *bne* (vidjeti uvodjenje instrukcije *slt*).

NAPOMENA 2: Osim instrukcija *addi* i *slti*, prilikom zapisivanja asemblerskog koda, veoma često se upotrebljava immediate instrukcija *muli*, koja se upotrebljava za implementaciju množenja sa konstantom i čija je sintaksa

muli **\$x, \$y, C** **# $\$x=\$y\times C$**

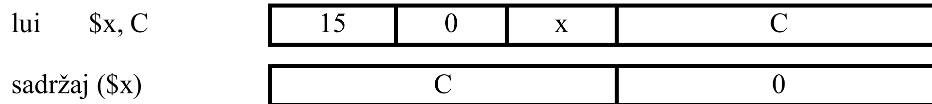
Primijetimo da sintansa *muli* instrukcije u potpunosti odgovara sintaksi immediate instrukcija *addi* i *slti*.

NAPOMENA 3: **Immediate instrukcija *lui*.** 16-bitno immediate polje upotrebljava se za zapisivanje i čuvanje konstantnih operanada prilikom zapisivanja immediate instrukcija. Shodno tome, maksimalna vrijednost konstante koja se upotrebljava kao operand u immediate instrukcijama odredjena je 16-bitnim immediate poljem. Medjutim, postavlja se pitanje što je sa konstantama (analogno tome, što je sa početnim adresama nizova kod lw i sw instrukcija) koje ne mogu biti zapisane u 16-bitnom immediate polju? Da li ove konstante mogu biti upotrebljavane i na koji način?

Navedeni problem se prevazilazi uvodjenjem immediate instrukcije *lui* (od engleskih riječi *Load Upper Immediate – napuni viši dio (rezultujućeg registra) trenutnim (konstantnim) operandom* na našem jeziku), čija je sintaksa

lui **\$x, C** **# $\$x=(C_{(10)},0_{(10)})$**

Primijetimo da sintaksa instrukcije *lui* od operanada uključuje samo konstantni operand C, dok je registar \$x – registar u kome će biti zapisan rezultat dobijen nakon izvršavanja instrukcije. Kako se immediate instrukcije zapisuju I-formatom zapisivanja, koji predvidja zapisivanje obilježja 2 registra (polja *rs* i *rt*, slika 1), prilikom zapisivanja mašinskog koda instrukcije *lui*, u polju *rs* zapisuje se 0 (nije uključen operand iz registara), a u polju *rt* obilježje rezultujućeg registra \$x, Slika 2. Viši 16-bitni dio reziltata odgovara konstanti C sačuvanoj unutar instrukcije *lui*, dok je niži 16-bitni dio rezultata $0_{(10)}$, kao što je naznačeno u komentaru gore navedene instrukcije i grafički prikazano na slici 2.



Slika 2. Format zapisivanja instrukcije *lui* i rezultat dobijen nakon izvršavanja.

Uvodjenjem instrukcije *lui*, te njenom kombinacijom sa instrukcijom *addi* može se kreirati velika konstanta čija vrijednost ne bi mogla biti zapisana u 16-bitnom immediate polju, odnosno stvoreni su uslovi za prevazilaženje problema istaknutog u NAPOMENI 3.

Primjer. Napisati MIPS asemblerski kod za smještanje broja

Z=0000 0000 0011 1101 0000 1001 0000 0000

u registar \$x.

Rješenje: Primijetimo najprije da navedeni broj predstavlja veliku konstantu, pošto ne može biti smješten u 16-bitnom immediate polju. Shodno tome, u cilju njegovog kreiranja i smještanja u registar \$x, potrebno je upotrijebiti immediate instrukciju *lui*. Ukoliko se broj podijeli na dva 16-bitna dijela, te svaki od djelova zapiše u dekadnom obliku, zadati broj Z može biti zapisan u obliku $Z=(61_{(10)},2304_{(10)})$.

Operacija zahtijevana ovim primjerom implementira se sljedećim parom instrukcija, uz napomenu da je pojašnjenje implementacije zapisano u komentarima zapisanim na kraju svake od instrukcija

lui	\$x, 61	# $\$x=(61_{(10)},0_{(10)})$
addi	\$x, \$x, 2304	# $\$x=(61_{(10)},2304_{(10)})$

Na koncu, zabilježimo da u praksi nema potrebe za kreiranjem velikih konstanti od strane compiler-a, odnosno od strane programera u asembleru, već da se one jednostavno navode u simboličkom (asemblerskom) kodu, a da je za njihovo kreiranje (gore navedenim parom instrukcija) zadužen assembler (prilikom zapisivanja mašinskog koda instrukcije koja u sebi uključuje konstantni operand velike vrijednosti). U tom slučaju, kao rezultujući registar (gore zapisan obilježjem \$x), assembler upotrebljava registar \$1, koji se inače **privremeni regista assembler-a**, kako smo to već naveli (prilikom razmatranja pseudoinstrukcije *blt*).

VJEŽBE

1. Napisati program u MIPS asembleru koji određuje broj elemenata većih od 12 i manjih od 24 u vektoru (niz cijelih brojeva) čija je početna adresa $500_{(10)}$. Dužina vektora se nalazi na memorijskoj lokaciji $60_{(10)}$. Smjestiti podatak o broju elemenata koji zadovoljavaju uslov na memorijsku lokaciju $64_{(10)}$.

   <u>Rješenje:</u>
   Potrebno je pročitati svaki element posmatranog niza (pristupiti memorijskoj adresi na kojoj se nalazi), i ispitati da li ispunjava postavljene uslove. Ukoliko element ne zadovoljava prvi uslov, drugi uslov se ne ispituje jer je neophodno da OBA uslova budu zadovoljena. Ispitivanje elemenata se vrši sve dok ne dodjemo do kraja niza, odnosno dok nam indeks niza ne postane jednak dužini niza.
   Obratite pažnju da je podatak o dužini niza smješten u memoriji, te ga je najprije neophodno pročitati i smjestiti u pomoćni registar. Takodje, uslov zadatka je da se broj elemenata koji zadovoljavaju postavljene uslove memoriše, odnosno smjesti na tačno određenu lokaciju.

```
        addi    $12, $0, 12        # $12=12
        add     $9, $0, $0         # indeks niza i
        add     $10, $0, $0        # brojač elemenata koji su zadovoljili uslov
        lw      $11, 60($0)        # čitamo dužinu niza
        beq     $11, $0, Exit      # provjeravamo da li je u pitanju prazan niz
Loop:   muli    $13, $9, 4         # 4i
        lw      $14, 500($13)      # $14=niz[i], čitamo element niza
        slti    $8, $14, 24        # $8=1 ako je niz[i]<24, $8=0 ako je niz[i]≥24
        bne     $8, $0, L1         # niz[i] ispunjava prvi uslov, idi na ispitivanje
                                   # drugog uslova
        j       L2                 # niz[i] ne ispunjava prvi uslov, drugi uslov ne
                                   # treba da provjeravaš (jer su ti potrebna OBA)
L1:     slt     $8, $12, $14       # $8=1 ako je 12<niz[i], $8=0 ako je 12≥ niz[i]
        beq     $8, $0, L2         # niz[i] ne ispunjava drugi uslov
        addi    $10, $10, 1        # niz[i] ispunjava oba uslova, prebrojiš
L2:     addi    $9, $9, 1          # i=i+1
        beq     $9, $11, Exit      # ako je i jednako dužini niza, znači da smo
                                   # ispitali sve elemente niza, pa idemo na izlaz
        j       Loop

Exit:   sw      $10, 64($0)        # memorišemo sadržaj brojača
```

2. Napisati potprogram (proceduru) *Copy* u MIPS asembleru. Potprogram uzima dva ulazna argumenta: pokazivač na izvorišni string u registru $4, i pokazivač na odredišni string u registru $5, i kopira sadržaj izvorišnog stringa u odredišni string. Potprogram vraća broj

prenesenih karaktera različitih od nule u registru $2. Napomena: string je niz karaktera i terminisan je nulom.

Rješenje:

Ulazni argumenti procedura se prosljeđuju preko registara $4, $5, $6 i $7. Naša procedura ima dva ulazna argumenta, pa su njima dodijeljeni registri $4 i $5. Takođe, izlazni argumenti procedura se prosljeđuju preko registara $2 i $3. U našem slučaju imamo jedan izlaz, koji ćemo ga proslijediti preko registra $2.

Procedure nemaju pravo mijenjanja sadržaja registara (osim onih registara koji su namijenjeni ulaznim i izlaznim argumentima procedure). S druge strane, pomoćni registri su neophodni proceduri za obavljanje različitih operacija. Rješenje ova dva suprostavljena zahtjeva je upotreba pomoćnih registara, pri čemu se njihov originalni sadržaj čuva na steku, i po završetku računanja (a prije vraćanja u glavni program) originalni sadržaj korišćenih registara se vraća sa steka. Na taj način je procedura obavila potrebna računanja, a glavni program "ne zna" da su njegovi registri korišćeni, jer je njihov sadržaj ostao nepromijenjen.

```
Copy:   addi    $29, $29, -4    # podešavanje pokazivača na vrh steka
        sw      $8, 0($29)      # smještanje sadržaja pomoćnog registra
        add     $2, $0, $0      # $2=0
Loop:   lb      $8, 0($4)       # učitavanje elementa stringa
        beq     $8, $0, Exit    # ako je element stringa =0, u pitanju je
                                # terminator → stigli smo do kraja stringa
        sb      $8, 0($5)       # kopiranje elementa stringa
        addi    $4, $4, 1       # sljedeći element izvorišnog stringa
        addi    $5, $5, 1       # sljedeći element odredišnog stringa
        addi    $2, $2, 1       # brojimo kopirane elemente
        j       Loop
        sb      $0, 0($5)       # dodajemo terminator novom stringu
Exit:   lw      $8, 0($29)      # vraćamo u pomoćni registar njegov
                                # originalni sadržaj
        addi    $29, $29, 4     # podešavanje pokazivača na vrh steka
        jr      $31             # vraćanje u glavni program
```

NAPOMENA: Prilikom pisanja procedura nije uvijek jednostavno procijeniti koliko će pomoćnih registara biti potrebno. Stoga je praksa da se najprije piše tijelo procedure, a da se smještanje na stek pomoćnih registara i vraćanje njihovih sadržaja (uokvireni djelovi koda) pišu naknadno. Imajte u vidu da stek raste ka nižim lokacijama, te da je zato neophodno pomjeriti pokazivač na vrh steka za 4*(broj pomoćnih registara) lokacija (svaki registar obuhvata 4B, otud množenje sa 4).