

MIPS procesor iz perspektive GNU asemblera

Amer Hasanović, Edin Pjanić i Selvin Fehrić

Tuzla, 2015

Amer Hasanović, Edin Pjanić, Selvin Fehrić
MIPS PROCESOR IZ PERSPEKTIVE GNU ASEMLERA

Izdavač

Izdavačka kuća Hamidović, Tuzla

Za izdavača

Rasim Hamidović

Glavni redaktor

Amer Hasanović

Recenzenti

Dr.sc. Damir Demirović, docent

Fakultet elektrotehnike Univerziteta u Tuzli

Dr.sc. Emir Mešković, docent

Fakultet elektrotehnike Univerziteta u Tuzli

Tiraž

200 primjeraka

CIP - Katalogizacija u publikaciji
Nacionalna i univerzitetska biblioteka
Bosne i Hercegovine, Sarajevo
004.272.3(075.8)

HASANOVIĆ, Amer

MIPS procesor iz perspektive GNU asemblera / Amer Hasanović, Edin Pjanić,
Selvin Fehrić. - Tuzla : "Hamidović", 2015. - VIII, 195 str. : graf. prikazi ; 24 cm

Bibliografija: str. 195.

ISBN 978-9958-833-07-6

1. Pjanić, Edin 2. Fehrić, Selvin

COBISS.BH-ID 22494214

Odlukom Senata Univerziteta u Tuzli br. 03-5285-9.7/15 od 07.10.2015. godine, odobrena je upotreba udžbenika "MIPS procesor iz perspektive GNU asemblera", autora Amera Hasanovića, Edina Pjanića i Selvina Fehrića, za potrebe izučavanja nastavnog predmeta "Arhitektura računara" na Fakultetu elektrotehnike Univerziteta u Tuzli.

Tuzli

Tuzli

Udžbenik "MIPS procesor iz perspektive GNU asemblera" je rezultat aktivnosti programa "Study of Excellence" u okviru projekta NORBOTECH, finansiranog od strane Ministarstva vanjskih poslova Kraljevine Norveške kroz HERD program, program za visoko obrazovanje, istraživanje i razvoj.

er Hasanović, Edin Pjanić,
graf. prikazi ; 24 cm

odobrena je upotreba udžbenika "MIPS
i Selvina Fehrića, za potrebe izučavanja
Univerziteta u Tuzli.

Sadržaj

<i>Predgovor</i>	vii
UVOD.....	1
Komponente računara	
MIPS procesor	
Kreiranje programa	
ALATI ZA KOMPILIRANJE I ANALIZU	13
Elk kompjajler	
Instalacija kompjajlerskog lanca	
Konfiguracija za kompajliranje	
Konfiguracija za debagiranje	
OSNOVE ASEMLERA	29
Kostur asembler programa	
Proces asembliranja i uvezivanja	
MIPS asembler instrukcije	
Debagiranje asembler programa	
ARITMETIČKO-LOGIČKE INSTRUKCIJE	45
Aritmetičke instrukcije	
Logičke instrukcije	
Šift instrukcije	
PRISTUP MEMORIJI	65
Memorijska slika programa	

Inicijalizacija globalnih varijabli

MIPS load i store instrukcije

PROMJENA TOKA PROGRAMA	81
Bezuslovni skokovi	
Instrukcija j (jump)	
Odgođeni slot	
Instrukcije jal i jr	
Uslovni skokovi	
Instrukcije slt i slti	
Realizacija petlji u asemblerskom kodu	
FUNKCIJE I STEK	117
Implementacija i pozivanje funkcije	
Stek	
Rekurzivne funkcije	
MAŠINSKI KOD	141
Načini adresiranja u MIPS procesoru	
Format MIPS instrukcija	
R format instrukcija	
I format instrukcija	
J format instrukcija	
UPRAVLJANJE PROJEKTIMA I UVEZIVANJE	177
Linker	
Makefile skripte	

Primarna načina organizacije strukture računala, zadržani komponenti, teta elektroinstalacija, ova knjiga je namenjena onisanju mikroprocesora. Za razumevanje i poznaje osnovne tehnike rada.

U ovoj knjizi su predstavljene i tehnike rada na računalima i poglavljaju se analizu programskog softvera i programata.

Ova knjiga je namenjena analizirane strukture računala, procesor. Razmatraju se koncepti rada na računalima.

Druge poglavljaje detaljno upoznaju čitaoca za pisanje i razvoj programskog softvera na jezicima C, C++ i Java.

Nakon teorijske i praktične izlaganja programera uveze se u programski jezik MIPS asembler. Način reprezentacije naredbi u programu, prilikom njegove izvođenja, i gramatički jezik pisanih naredbi.

Predgovor

81

Primarna namjena ove knjige je kao udžbenik na predmetu Arhitektura računara i dijelom na predmetima Operativni sistemi i Dizajn kompjajlera, koji se izučavaju na dodiplomskom studiju Fakulteta elektrotehnike Univerziteta u Tuzli. Osim toga, željeli smo da ova knjiga posluži i svima onima koji žele razumjeti način funkcionisanja mikroprocesora i nauče programirati u asemblerском jeziku. Za razumijevanje materije u ovoj knjizi, od čitaoca se očekuje da poznaje osnovne tehnike programiranja u programskom jeziku C.

kom kodu

117

nkcijske

141

procesoru

177

U ovoj knjizi se razmatra MIPS procesor, ali objašnjeni koncepti i tehnike vrijede za bilo koji mikroprocesor. Posebno vrijedna su i poglavlja u kojima su detaljno objašnjeni alati za kompjajliranje i analizu programskog koda kao i alati za automatizaciju kreiranja programa.

Ova knjiga se sastoji od 9 poglavlja. U uvodnom poglavlju su analizirane osnovne komponente računara sa posebnim osvrtom na procesor. Razmatrani su osnovni koncepti MIPS procesora, kao i koncepti računarskog programa.

Drugo poglavlje, nazvano *Alati za kompjajliranje i analizu*, sadrži detaljno uputstvo za instalaciju i konfiguraciju alata koji su potrebni za pisanje i debagiranje programa napisanih u programskim jezicima C, C++ i u asemblerском jeziku.

Nakon toga, u poglavlju *Osnove asemblera*, analizirana je struktura programa na najnižem nivou. Predstavljeni su najvažniji koncepti MIPS asemblera te je uspostavljena veza između izvršnih fajlova, binarnih reprezentacija kompjajliranih programa na disku i aplikacija prilikom njihovog izvršavanja. Nije zaobiđeno ni debagiranje programa pisanih u assembleru.

U poglavlju *Aritmetičko-logičke instrukcije* analizirane su istoime-ne instrukcije MIPS procesora kroz kompajliranje i analizu progra-ma pisanih u asemblerском језику.

Poglavlje *Pristup memoriji* se bavi analizom memorije sa stanovi-šta MIPS procesora, slikom programa u memoriji, te načinima pris-tupa memoriji od strane MIPS procesora.

U šestom poglavlju, *Promjena toka programa*, analizirane su asem-blerske instrukcije za realizaciju uslovnih i bezuslovnih skokova, kao i način realizacije petlji u asemblerским programima.

U poglavlju *Funkcije i stek* analiziran je koncept funkcije kao i način implementacije i pozivanja funkcija u asemblerском kodu. U vezi sa funkcijama, neizbjegjan je stek. Objasnjen je pojam aktiva-cijskog okvira sa detaljnom analizom njegove strukture kod MIPS procesora. Razmatrane su konvencije za očuvanje vrijednosti regis-tara prilikom pozivanja funkcija, način razmijene parametara funk-cije kao i način realizacije rekurzivnih funkcija.

Osmo poglavlje, nazvano *Mašinski kod*, bavi se načinom binarnog kodiranja instrukcija MIPS procesora. Detaljno su analizirani R, I i J formati MIPS instrukcija. Kroz primjere je analizirano i disasem-bliranje MIPS instrukcija.

U zadnjem poglavlju ove knjige, pod nazivom *Upravljanje projek-tima i uvezivanje*, razmatrani su alati kojim se reduciraju nepotrebna ponavljanja u procesu kreiranja programa. Konkretno, razmatrane su linker i Makefile skripte te kroz primjere dato detaljno uputstvo za njihovo korištenje.

Uvod

Pojam *računar* predstavlja relativno novu riječ u ljudskim jezicima. Ova riječ označava uređaj koji može da automatski izvrši proizvoljno dugu sekvencu jednostavnih operacija. Operacije koje izvršavaju računari nazivaju se instrukcije, a obično su aritmetičkog ili logičkog karaktera.

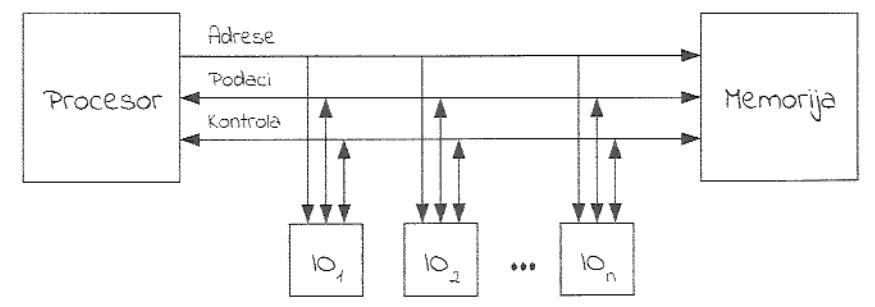
Procesom programiranja sekvence instrukcija koje izvršava računar može se izmjeniti, što omogućava primjenu računara u rješavanju različitih klasa problema iz ljudskog života.

Za kratko vrijeme računari su, od uređaja koji fizički zauzimaju cijele sobe i koje upotrebljavaju samo posebno obučeni istraživači, postali uređaji široke namjene koje koristimo svakodnevno u obliku kućnih računara, mobilnih telefona ili pametnih satova.

Bez obzira na formu, svi moderni računari dijele sličnu arhitekturu. Tokom ovog poglavlja analizirat ćemo osnovne komponente ove arhitekture. Posebnu pažnju posvetiti ćemo ključnoj komponenti koja se zove procesor. Dodatno, osvrnut ćemo se na proces programiranja i razmotriti različite nivoje sa kojih se može posmatrati program.

Komponente računara

Pojednostavljen dijagram koji uključuje osnovne komponente računara prikazan je na slici 1-1.



Slika 1-1. Komponente računara

Svi elementi na slici izrađuju se od digitalno-elektronskih komponenti i imaju različitu funkciju u računaru.

Procesor/CPU je ključna komponenta ove arhitekture. Njegov osnovni zadatak u računaru je da izvršava instrukcije.

Memorija/RAM je još jedna komponenta bez koje računar nema smisla. Zadatak memorije je da pohrani sekvence instrukcija, tj. programe koje će izvršavati procesor. Pored instrukcija, memorija pohranjuje i podatke koji se koriste u instrukcijama. Memoriju pojednostavljeni možemo posmatrati kao uređaj u kojem se nalazi ogroman broj kondenzatora. Količina informacija koju možemo pohraniti u memoriji direktno je proporcionalna broju kondenzatora. Svaki kondenzator u datom trenutku može biti u jednom od dva stanja: prazan ili napunjeno. Iz ovog zaključujemo da je osnovna jedinica za pohranu podataka u memoriji jedan bit informacije, tj. 0 za stanje praznog kondenzatora ili 1 za stanje punog kondenzatora. Kondenzatori obično formiraju grupe od osam elemenata. Svaka grupa, koja efektivno može da pohrani bajt informacije, sa sobom ima asociran broj koji predstavlja adresu u memoriji.

Iz ovakve organizacije memorije proizilazi da instrukcije i podaci koji se zapisuju u memoriji moraju biti prevedeni u binarni format, tj. moraju se predstaviti kao sekvence određenog broja bita.

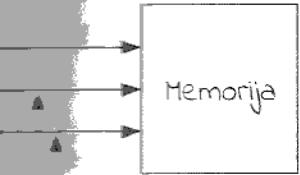
Prema
žica koj
ne tri m
ne magis
tj. da li se
mi određe

Ukoliko
na kontrol
tanje. Na
se nalazi t
koji se na
gistrale ša

Sličan s
moriju. T
koju se po
procesor
magistrali
podatka sa

Broj žic
koja se ko
žice za po
birnice, iz
čeno 4 ili
đuje veliči
ce imaju m

Pored i
tzv. uređa
daji mogu
zivaju ulaz
tor, miš, t
putem ko
dobija odr



o-elektronskih kom-

ove arhitekture. Njegov
strukcije.

bez koje računar nema
vence instrukcija, tj. pro-
strukcija, memorija po-
cijama. Memoriju pojed-
u kojem se nalazi ogro-
ja koju možemo pohra-
na broju kondenzatora.
že biti u jednom od dva
ujemo da je osnovna je-
dan bit informacije, tj. 0
anje punog kondenzato-
d osam elemenata. Svaka
t informacije, sa sobom
memoriji.

azi da instrukcije i podaci
vedeni u binarni format,
enog broja bita.

Procesor i memorija međusobno su povezani sa velikim brojem žica koje se organiziraju u magistrale (sabirnice). Na slici su prikazane tri magistrale: adresna, podatkovna i kontrolna. Putem kontrolne magistrale procesor memoriji određuje trenutni mod operacije, tj. da li se u datom trenutku od memorije očekuje da pročita ili snimi određeni podatak.

Ukoliko procesor treba da iz memorije pročita određeni podatak, na kontrolnoj magistrali postavit će mod operacije memorije na čitanje. Na adresnoj magistrali postavit će adresu u memoriji na kojoj se nalazi traženi podatak. Nakon čitanja stanja grupe kondenzatora koji se nalaze na zadanoj adresi, memorija putem podatkovne magistrale šalje procesoru pročitana stanja.

Sličan scenario događa se kada procesor upisuje podatak u memoriju. Tada procesor na adresnu magistralu postavlja adresu na koju se podatak snima. Podatak koji treba da se snimi u memoriju procesor postavlja na podatkovnu magistralu, dok na kontrolnoj magistrali procesor nareduje memoriji da izvrši operaciju snimanja podatka sa podatkovne sabirnice.

Broj žica koje čine sabirnice vezan je za tip procesora i memorije koja se koristi u računaru. Moderni računari obično koriste 32 ili 64 žice za podatkovnu sabirnicu. Dakle, spram širine podatkovne sabirnice, između memorije i procesora može biti maksimalno prebačeno 4 ili 8 bajta. Sa druge strane, broj žica adresne sabirnice određuje veličinu adresnog prostora računara. Moderne adresne sabirnice imaju minimalno 32 žice, čime dobijamo adresni prostor od 4GB.

Pored memorije i procesora, na sabirnice mogu biti povezani i tzv. uređaji, koji su na slici označeni kao IO_1 , IO_2 do IO_n . Ovi uređaji mogu obavljati funkciju ulaza, izlaza ili i ulaza i izlaza pa se nazivaju ulazno-izlazni uređaji računara. Neki od tih uređaja su monitor, miš, tastatura, disk itd. IO uređaji komuniciraju sa procesorom putem koncepta koji se naziva memorijsko mapiranje. Svaki uređaj dobija odredene adrese iz fizičkog adresnog prostora računara. Ka-

da procesor na adresnoj sabirnici emituje adresu asociranu sa nekim uređajem, komunikacija preko sabirnice odvija se sa datim uređajem, a ne sa memorijom. Memorijskim mapiranjem procesor ima identičnu vezu sa uređajima kao i sa memorijom.

MIPS procesor

Postoje dva pristupa za implementaciju procesora:

Reduced Instruction Set Computer - RISC

Jednostavni procesor koji je specijaliziran da izvršava mali skup pažljivo odabralih instrukcija.

Complex Instruction Set Computer - CISC

Kompleksan procesor koji direktno u hardveru implementira veliki broj instrukcija različitih namjena.

Pored razlike u broju instrukcija koje podržavaju, RISC i CISC procesori razlikuju se i u ostalim detaljima implementacije, kao npr. način formiranja instrukcija, način pristupa memoriji itd.

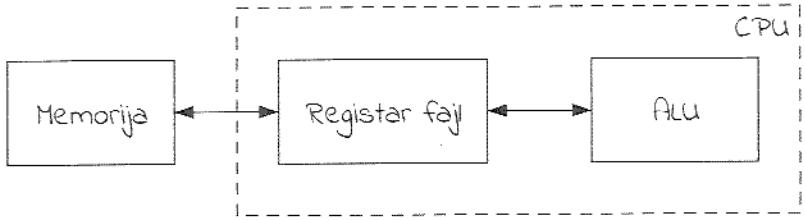
Bez obzira na korištenu filozofiju implementacije, od trenutka napajanja svi procesori izvršavaju instrukcije u petlji koja se sastoji od sljedećih koraka:

1. Preuzimanje instrukcije iz memorije.
2. Dekodiranje preuzete instrukcije.
3. Izvršenje dekodirane instrukcije.
4. Prelazak na izvršenje sljedeće instrukcije.

Dok izvršava određenu instrukciju, procesor mora voditi računa o lokaciji u memoriji na kojoj će pronaći sljedeću instrukciju iz programske sekvence koju izvršava.

MIPS procesor je nastao 1985 godine na Stanford Univerzitetu kao jedan od prvih pokušaja implementacije RISC arhitekture. Prvi

računari sa ovim procesorom služili su kao skupocjene grafičke ili serverske stanice. Ovaj procesor je danas u širokoj primjeni, od običnih mikrokontrolera, preko konzola za igru, pa do mrežnih usmjerivača i bežičnih baznih stanica. Kao i ostali RISC procesori, MIPS spada u tzv. load/store arhitekturu. Osnovni elementi ove arhitekture prikazani su na sljedećoj slici.



Slika 1-2. Load/store arhitektura

Sa slike možemo zaključiti da su ključne interne komponente MIPS procesora registar fajl i aritmetičko logička jedinica, označena kao ALU na slici.

Registar fajl je komponenta sastavljena od registara, uređaja za privremenu pohranu podataka direktno unutar samog procesora. Pojedinačni registri implementiraju se pomoću paralelnog spojenih flip-flopova. Registar fajl u MIPS procesoru ima 32 registra, a svaki registar individualno može da pohrani 32 bita informacije.

Registar fajl putem podatkovne sabirnice ima vezu sa memorijom. Unutar procesora, ova komponenta je povezana i sa aritmetičko-logičkom jedinicom u kojoj se, putem osnovnih logičkih kola, implementiraju operacije svih definiranih instrukcija iz domena MIPS procesorske arhitekture (MIPS ISA). Da bi se unutar ALU izvršila bilo koja operacija, npr. sabiranje, poređenje, šiftanje itd., podaci prvo moraju biti pripremljeni unutar registar fajla. MIPS ALU može da izvrši operaciju nad maksimalno dvije vrijednosti, koje mogu biti pročitane iz bilo koja dva registra unutar registar fajla, ili jedna vrijednost ALU operacije može biti zakodirana unutar instrukcije koju procesor izvršava u datom trenutku. Na-

kon što ALU obavi željenu operaciju, MIPS procesor treba da snimi rezultat u nekom od registara iz registar fajla. Destinacijski register definiran je u samoj instrukciji koju procesor trenutno izvršava. Nakon snimanja rezultata ALU operacije završava se proces izvršenja određene instrukcije a procesor može prijeći na preuzimanje sljedeće instrukcije.

MIPS procesor ne omogućava izvršenje ALU operacija nad vrijednostima pročitanim direktno iz memorije. Ti podaci se prvo putem specijalnih instrukcija za učitavanje (load) moraju kopirati iz memorije u registre. Tek nakon toga podatke koji su učitani u registar fajl je moguće obraditi bilo kojom od podržanih instrukcija. Ukoliko je potrebno snimiti podatke iz registar fajla u memoriju, MIPS procesor podržava specijalne instrukcije za pohranjivanje (store) vrijednosti pročitane iz bilo kojeg registra na neku adresu u memoriji. Transfer između registar fajla i memorije, u bilo kojem smjeru, može se odvijati u količini od:

- bajt — 8 bita,
- pola riječi (half word) — 16 bita,
- riječ (riječ) — 32 bita.

Pored registara u registar fajlu, MIPS procesor ima i druge registre, od kojih je najbitniji register pod imenom programski brojač. Kao i ostali registri, programski brojač je 32-bitan. Glavni zadatak ovog registra je da čuva adresu od koje počinje prvi bajt sljedeće instrukcije koja će se izvršavati u procesoru. Sve instrukcije MIPS arhitekture kodiraju se u 32 bita, tj. iznose 4 bajta, zbog čega je vrijednost programskog brojača u svakom trenutku djeljiva brojem 4.

MIPS procesor svaku instrukciju izvršava u tačno pet faza:

faza 1

Preuzimanje instrukcije iz memorije, sa adrese pročitane iz programskog brojača, i inkrementiranje vrijednosti programskog brojača za 4.

faza 2

Dekodiranje instrukcije i čitanje onih registara iz registar fajla koji su dati unutar dekodirane instrukcije.

faza 3

Obavljanje ALU operacije nad pročitanim vrijednostima iz registar fajla, ili na jednoj vrijednosti iz registar fajla a drugoj vrijednosti dobivenoj iz dekodirane instrukcije.

faza 4

Pristup memoriji radi obavljanja operacije pohranjivanja vrijednosti iz registra u memoriju u slučaju da je dekodirana instrukcija store.

faza 5

Zapisivanje rezultata obavljene instrukcije u destinacijski registar unutar registar fajla.

Za svaku od navedenih faza, MIPS procesor ima posebne hardverske komponente koje implementiraju datu fazu. Neke komponente smo već naveli, npr. registar fajl učestvuje u implementaciji faze 2, programski brojač aktivan je u fazi 1, dok ALU realizira fazu 3. Unutar MIPS procesora, komponente iz nabrojanih faza organizuju se u cjevovod (pipeline).

Operacije u cjevovodu možemo objasniti na praktičan način analizirajući proces koji se događa u manuelnoj praonici automobila. Da bi se izvršilo kompletno čišćenje, praonica na pojedinačnom automobilu treba izvršiti sljedeće korake:

1. nanošenje šampona,
2. pranje,
3. sušenje i usisavanje.

Ukoliko se radnici u praonici organiziraju u tri tima, svaki tim može da se posveti određenom koraku u procesu čišćenja automo-

bila. Na ovaj način u svakom trenutku pravilica efektivno može da procesira do tri automobila, uz optimalan angažman radnika. Pravilica je zapravo cjevovod koji ima tri faze, a svaku fazu implementira tim radnika. Automobil mora proći sve tri faze u sekvenci da bi bio očišćen.

Pojedinačna instrukcija u određenoj fazi MIPS cjevovoda tretira se kao automobil u pojedinačnoj fazi pravilice, tj. nakon procesiranja u datoј fazi prosljeđuje se sljedećoj. S obzirom na broj faza, MIPS procesor na ovaj način u svakom trenutku može da procesira paralelno do pet instrukcija, uz optimalan angažman hardvera u pojedinačnim fazama. Cjevovod značajno unapređuje performanse MIPS procesora, ali unosi i probleme za implementaciju. Jedan od problema rješava se uvođenjem tzv. odgodenog slota nakon svake instrukcije kojom se vrši promjena toka programa, o čemu ćemo govoriti u narednim poglavljima.

Kreiranje programa

Već smo naveli da su programi različite sekvence instrukcija čijim se izvršenjem na procesoru rješavaju problemi iz bilo kojeg domena realnog života. Programiranje je proces formulacije konkretnе sekvene instrukcija, tj. programa. Instrukcije za bilo koji procesor formulišu se u binarnom formatu, koji se još zove i mašinac. Ovaj format nije pogodan za manipulisanje od strane programera, prvenstveno zbog nepreglednosti programa i nedostatka ekspresivnosti na nivou mašinskih instrukcija. Zbog toga se programi pišu u raznim programskim jezicima a procesom kompajliranja prevode u sekvenčne instrukcije za željenu procesorsku arhitekturu.

Proces kompajliranja, prikazan na slici 1-3, odvija se u dva koraka:

1. Putem kompajlera, program napisan u nekom programskom jeziku prevodi se u asembler formu.

efektivno može da
radnika. Prav
fazu implemen
faze u sekvenci da bi

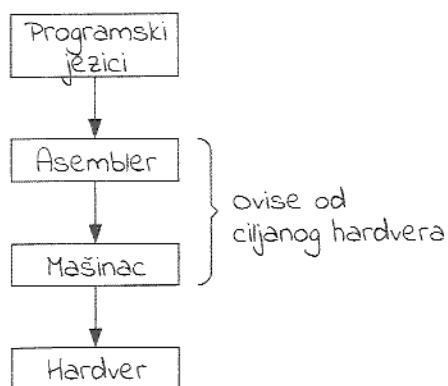
MIPS cjevovoda tretira
tj. nakon procesiranja na broj faza, MIPS
može da procesira paralelno hardvera u pojedini
red je performanse MIPS
konfiguraciju. Jedan od problema
nakon svake instruk
ćemu ćemo govoriti u

sekvene instrukcija čijim
čim iz bilo kojeg domena
formulacije konkretnе sek
za bilo koji procesor for
zove i mašinac. Ovaj for
programera, prvens
statka ekspresivnosti na
programi pišu u raznim
ranja prevode u sekven
eksturu.

3, odvija se u dva kora

u nekom programskom

- Iz dobijene asembler forme program se, putem asemblera, prevodi u binarnu formu.



Slika 1-3. Proces kompajliranja programa

Kompajler i asembler su također programi u mašinskoj formi, nastali procesom kompajliranja.

Dok je u formi određenog programskog jezika, program nije specijaliziran za konkretnu procesorsku platformu te za njega kažemo da je portabilan, tj. da se može prevesti u mašinsku formu za određenu procesorsku platformu, sve pod uslovom da imamo odgovarajući kompajler i asembler. U asembler formi, program je već prevoden za specifični procesor. Sekvenca odgovarajućih instrukcija već je generirana i prisutna u asembler fajlu. Međutim, ovaj fajl je u tekstuallnom formatu a pojedinačne instrukcije iz sekvence programa reprezentirane su njihovim imenima. Parametri za pojedinačne instrukcije u asembler fajlu su također čitljivi, i to u formi brojeva ili slova. Ovo možemo ilustrirati jednostavnim primjerom.

Prepostavimo da je dat sljedeći segment programa napisanog u programskom jeziku C.

```
int foo(int a, int b) {  
    return (a + b) / 8;  
}
```

Čak i za programere neupućene u sintaksu programskog jezika C, gornji kod je čitljiv u smislu da se može zaključiti da segment obavlja aritmetičku operaciju sabiranja između varijabli a i b, te dijeli dobivenu sumu brojem 8. Za upućenije, jasno je da segment koda definira funkciju pod imenom foo, koja uzima dva parametra a i b a vraća rezultat pomenute aritmetičke operacije.

Kada izvršimo kompajliranje za MIPS procesor, dobijamo sljedeću asembler formu:

```
foo:  
    addu $1, $5, $4  
    sra $2, $1, 31  
    srl $2, $2, 29  
    addu $1, $1, $2  
    jr $ra  
    sra $2, $1, 3
```

Kompajliranjem istog C segmenta za Intel platformu, dobijamo sljedeći rezultat:

```
foo:  
    movl 8(%esp), %ecx  
    addl 4(%esp), %ecx  
    movl %ecx, %eax  
    sarl $31, %eax  
    shrl $29, %eax  
    addl %ecx, %eax  
    sarl $3, %eax  
    retl
```

Poređenjem rezultata kompajliranja za dvije platforme možemo zaključiti sljedeće:

- Iako čitljivi u tekstualnoj formi, asembler programi su manje razumljivi u odnosu na C formu.
- Bez obzira na ciljanu platformu, asembler programi su duži s obzirom na to da je C sintaksa značajno ekspresivnija u odnosu na jednostavne procesorske instrukcije.
- Asembler programi za Intel i MIPS platformu u potpunosti se razlikuju jer procesori podržavaju različite instrukcije.

Dod...
segment
istih, se...
je da pu...
tijeva ne...

Ukola...
obavim...
ju, za M...
cija zapis...

0: ee
4: ee
8: ee
c: ee
10: ee
14: ee

Isto tak...
tel proces...

0: 8b
4: 03
8: 89
a: c1
d: c1
10: 01
12: c1
15: c3

Poređe...
zaključi...

- Sval...
plat...
bajt...
- Pro...
noz...
maš...
- Pro...
čitlj...

Dodatno, proces kompajliranja nikad nije jednoznačan, tj. jedan segment C koda je moguće prevesti u više različitih, ali funkcionalno istih, sekvenci asembler koda za neki procesor. Zadatak kompajlera je da pokuša da pronađe onu od svih dozvoljenih sekvenca koja zahtijeva najmanje vremena za izvršenje na procesoru.

Ukoliko nastavimo dalje sa transformacijom koda iz primjera, tj. obavimo asembleriranje iz asembler forme u mašinsku reprezentaciju, za MIPS procesor dobijamo sljedeći segment mašinskih instrukcija zapisanih u heksadecimalnoj formi:

```
0: 00a40821  
4: 000117c3  
8: 00021742  
c: 00220821  
10: 03e00008  
14: 000110c3
```

Isto tako, transformacija koda iz primjera na mašinski nivo za Intel procesor rezultira sljedećom sekvencom mašinskih instrukcija:

```
0: 8b 4c 24 08  
4: 03 4c 24 04  
8: 89 c8  
a: c1 f8 1f  
d: c1 e8 1d  
10: 01 c8  
12: c1 f8 03  
15: c3
```

Poređenjem rezultata asembleriranja za dvije platforme možemo zaključiti sljedeće:

- Svaka MIPS instrukcija kodira se u četiri bajta, dok za Intel platformu to nije slučaj i instrukcije su promjenljivog broja bajta.
- Proces asembleriranja, bez obzira na platformu, uvijek je jednoznačan, tj. za jednu asembler instrukciju dobijamo jednu mašinsku instrukciju.
- Programi na nivou mašinaca, bez obzira na platformu, su nečitljivi i nerazumljivi za programere.

Iz prethodne analize možemo zaključiti da je programe moguće pisati na bilo kojem nivou. Sa stanovišta čitljivosti i ekspresivnosti, programski jezici imaju značajnu prednost u odnosu na asembler ili mašinsku formu. Obzirom na sposobnost današnjih kompjajlera da produciraju optimalne sekvene instrukcija u procesu kompjiranja, teško je naći opravdanje za programiranje direktno u asembleru. Međutim, sa stanovišta razumijevanja funkcionalnosti i implementacije procesora ili definiranja novih programske jezika i kreiranja njihovih kompjajlera, apsolutno je neophodno poznavanje asemblera i mašinca, kao i alata koji sudjeluju u procesu kompjiranja.

Poglavlja koja slijede dat će nam uvid u funkcionalnost MIPS procesora, i to sa stanovišta pisanja i analize programa na nivou asemblera i mašinca za ovaj procesor.

Proces
koji se
zirom
da izla
za dru
nom —

Nat
će koris

- g
- c
- v

Sa s
sorske
Micro
tekstu,
tačnije
jekta C

Iako
optimiz
racije i

-
1. <https://>
 2. <http://>
 3. <https://>

Alati za kompajliranje i analizu

Proces kreiranja programa zahtjeva skup različitih alata (aplikacija) koji se pokreću u različitim stadijima kompajliranja programa. Obzirom da se pokretanje ovih aplikacija odvija u lancu, na način da izlazni podaci iz jedne aplikacije predstavljaju ulazne podatke za drugu aplikaciju, ti alati se često označavaju zajedničkim imenom — *kompajlerski lanac*.

Na tržištu postoje različiti kompajlerski lanci, od kojih su najčešće korišteni:

- *gcc* — lanac otvorenog koda projekta GNU¹
- *clang* — lanac otvorenog koda projekta LLVM²
- *Visual studio* — lanac kompanije Microsoft³

Sa stanovišta poštovanja standarda i podrške za različite procesorske platforme, *gcc* i *clang* su značajno bolje opcije u odnosu na Microsoft lanac. Kompajlerski lanac *Elk*, koji ćemo koristiti dalje u tekstu, nastao je fuzijom komponenti iz dva lanca otvorenog koda, tačnije kompajlera iz projekta LLVM, te asemblera i linkera iz projekta GNU.

Iako je moguće izvršiti instalaciju *Elka* na Microsoft platformi, optimalno okruženje za ovaj kompajlerski lanac je Linux. Konfiguracije i primjeri koji slijede odnose se na 64-bitnu Ubuntu distribu-

1. <https://gcc.gnu.org>

2. <http://clang.llvm.org>

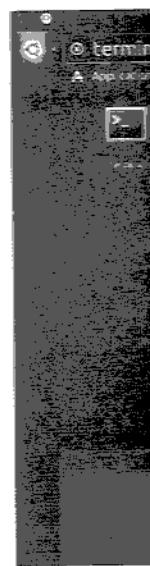
3. <https://www.visualstudio.com/>

Elk uključuje kompajliranje i analizu kod-a. Uključujući podršku za mandne linije komandne vrline, podršku za izvršavanje programova na različitim operativnim sistemima, podršku za tipke Alt+F1 do F12 i mogućnost izvođenja aplikacija.

Instalacija

Kompajlirani programi su dostupni u formi mandne linije komandne vrline, podržavajući izvršavanje programova na različitim operativnim sistemima. Tipke Alt+F1 do F12 omogućuju izvođenje aplikacija.

Da bi se moglo koristiti Elk, je potrebno instalirati kompajler na računaru sa Intel procesorima i koristiti komandne opcije za kompajliranje programova.



Slika 2-1. Prikaz terminala s izvođenjem komandе 'elk'.

Nakon instalacije, možete da pokrenete komandu 'elk' u terminalu.

ciju Linuxa, koju je potrebno instalirati direktno na računar ili u virtualnu mašinu unutar nekog drugog operativnog sistema. U vrijeme pisanja ovog teksta aktuelna verzija Ubuntu distribucije je 15.04. Bilo koja verzija Ubuntu sistema mogla bi se koristiti na sličan način.

Elk kompajler

Kompajlerski lanac Elk¹ ima niz mogućnosti koje su interesantne sa stanovišta izučavanja procesorske arhitekture, procesa kompajliranja i izvršavanja programa. Elk se instalira na standardnim desktop računarima sa Intel procesorima i koristi se za kompajliranje programova pomoću komandnih opcija koje su u potpunosti preuzete od gcc kompajlера.

Lanac dodaje i nove komandne opcije koje nemamo na raspolaganju sa drugim lancima. Posebno interesantna je opcija target, koja omogućava kreiranje programa za druge procesore i druge operativne sisteme. Upotrebom ove komande pri kompajliranju, na raspolaganju imamo sljedeći izbor procesora: ARM, PowerPC, X86, AMD64 ili MIPS, a kao ciljana okruženja imamo izbor između Linux i Microsoft operativnih sistema. Elk staticki uvezuje C ili C++ standardne biblioteke sa kompajliranim programima, čime je omogućeno izvršavanje na različitim distribucijama operativnih sistema neovisno od njihovih instaliranih verzija standarnih biblioteka.

Kompajlirani programi za Linux operativni sistem mogu se izvršavati i debagirati na standardnim personalnim računarima, i to bez obzira na procesor odabran opcijom target prilikom kompajliranja. Ovo je posljedica činjenice da Elk uključuje i aplikaciju pod imenom Qemu², koja može da simulira različite procesorske platforme.

-
1. <http://elcc.org>
 2. <http://qemu.org>

na računar ili u virtuelnom sistemu. U vrijeme distribucije je 15.04. mogu se koristiti na sličan način.

Ukoliko su interesantne sa vremenom procesa kompajliranja na standardnim desktop sistemima za kompajliranje programi potpuno preuzete od

Kako nemamo na raspolaganju je opcija target, koja omogućava kompajliranju, na različitim procesore i drugi operativnim sistemima. Takođe imaju kompajliranju, na različitim procesorima, PowerPC, X86, AMD64 i ARM. Među Linux i Microsoft C ili C++ standardne kompajlere je omogućeno izvršavanje u svim sistemima neovisno o bibliotekama.

Na ovom sistemu mogu se izvršiti kompajlirati na računarama, i to bez prilikom kompajliranja. Ako želimo da instaliramo aplikaciju pod imenom ELLCC na ovom sistemu, treba je odabrati procesorske platforme.

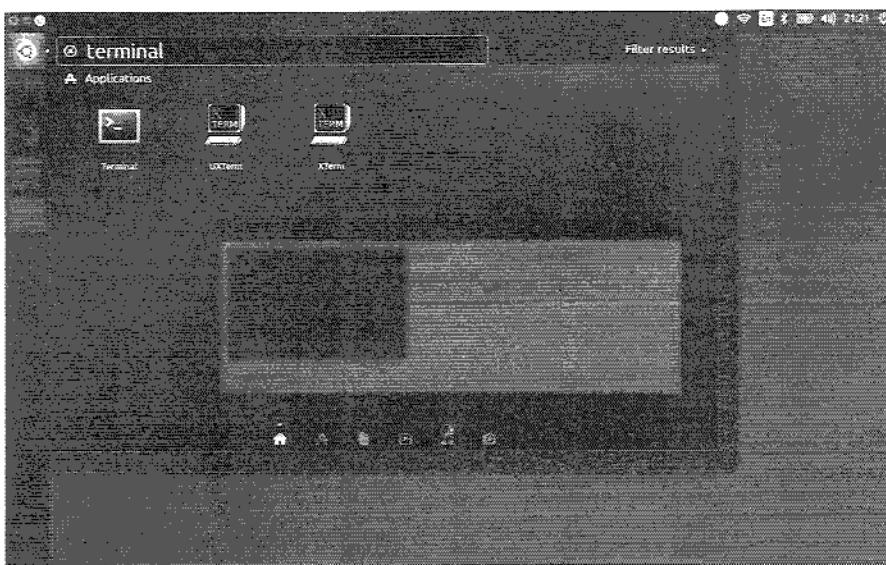
U ovom poglavlju će se detaljnije govoriti o kompajliranju i analizu programskih kodova.

Elk uključuje i alate korisne za analizu kompajliranih programa, npr. *objdump* i *readelf*, koje ćemo koristiti intenzivno u dalnjem tekstu.

Instalacija kompajlerskog lanca

Kompajliranje programa Elk kompajlerom odvija se putem bash komandne linije unutar terminala. Unutar Ubuntu okruženja, pritisnuvši na tipku **Alt**, koja se na tastaturi obično nalazi pored lijeve tipke **Win**, aktiviramo Ubuntu Dash nakon čega možemo unijeti ime aplikacije koja je instalirana u sistemu a koju želimo pokrenuti.

Da bi aktivirali terminal unosimo tekst kao na sljedećoj slici, pratioći ga sa tipkom **Enter**.



Slika 2-1. Pokretanje terminala

Nakon pokretanja terminala možemo nastaviti sa instalacijom Elk kompajlerskog lanca unosom sljedećeg teksta u terminal:

```
moj@komp$ mkdir -p $HOME/programi ❶  
moj@komp$ wget -O - \ ❷  
> http://ellcc.org/releases/ellcc-x86_64-linux-eng-0.1.14.tgz \ ❸  
> | tar zxf - -C $HOME/programi ❹
```

moj@komp: ~ > "exp... > >> S... moj@komp: ~

Napomena Ispis se u t Med... mina... Sa > kom... će da



Jednu bash komandu možemo razdijeliti u više linija terminala unosom karaktera \ praćeno sa tipkom ↵. Sistem će očekivati nastavak unosa teksta komande u novoj liniji terminala, a nakon karaktera >.

Linijom ❶ kreiramo direktorij programi u kojeg instaliramo kompjuterski lanac. Pomoću aplikacije wget u liniji ❷ preuzimamo Elk verziju 0.1.14 za 64-bitne Linux sisteme u obliku komprimirane tgz arhive sa adrese ❸.

Nastavljanjem komande u liniji ❹ zahtijevamo otpakivanje arhive paralelno sa njenim preuzimanjem, i to u novokreirani direktorij programi.

Po izvršenju gornje komande, čije trajanje ovisi od brzine Internet veze, instalirali smo kompjuter. Da bismo mogli adekvatno koristiti alate iz Elk lanca, potrebno je da bin folder kompjutera dodamo na komandnu stazu. To činimo promjenom bash varijable PATH na sljedeći način:

```
moj@komp$ export PATH=$HOME/programi/ellcc/bin:$PATH  
moj@komp$
```

Napomena

Dodavanjem direktorija na bash stazu, programi iz datog direktorija postaju dostupni bilo gdje u terminalu, bez obzira na trenutni direktorij.

Unos gornje komande morali bismo raditi prilikom svakog novog pokretanja terminala. Da bismo promjenu staze učinili trajnom, komandu je potrebno dodati na kraj .bashrc skripte, fajla čiji sadržaj bash izvršava prilikom svakog pokretanja terminala.

To možemo uraditi pomoću alata echo koji u terminalu vrši ispisivanje njemu proslijedenog stringa.

❶ Pre...
❷ Pre...
❸ Izl...

Alternativno...
program...
dit, koji...
ženja ka...
ruka je i...
savladiv...
gedit. In...
njem sek...

sudo ap...
sudo ap...

❶ Os...
lo...
❷ In...

vršenje u terminala uno-
staviti očekivati nastan-
jenje zara nakon karaktere-

kojeg instaliramo
liniji ② preuzimamo
obliku komprimira-

evamo otpakivanje arhi-
u novokreirani direktorij

anje ovisi od brzine Inter-
smo mogli adekvatno ko-
n folder kompjajlera doda-
jenom bash varijable PATH

bin:\$PATH

grami iz datog direktorija
zbzira na trenutni direk-

aditi prilikom svakog no-
jenju staze učinili trajnom,
rc skripte, fajla čiji sadržaj
terminala.

koji u terminalu vrši ispi-

```
moj@komp$ echo \  
> "export PATH=$HOME/programi/ellcc/bin:$PATH" \  
> >> $HOME/.bashrc ②  
moj@komp$ exit ③
```

Napomena

Ispis iz bilo kojeg pokrenutog programa u opštem slučaju prikazuje se u terminalu.

Međutim, putem operatora > ili >> izlaz iz programa, umjesto u terminal, možemo proslijediti u neku destinacijsku datoteku.

Sa > kreiramo novu datoteku ili već postojeći datoteci mijenjamo kompletan sadržaj, dok sa >> novi sadržaj dodajemo na kraj postojeće datoteke.

- ① Predstavlja zadati string koji echo treba da ispiše.
- ② Preusmjeravamo izlaz u datoteku .bashrc.
- ③ Izlazak iz terminala.

Alternativno, komandu smo mogli dodati u .bashrc fajl pomoću programerskog editora. Ubuntu dolazi sa instaliranim editorom gedit, koji se ponaša slično editorima iz integriranih razvojnih okruženja kao što su Visual studio ili Eclipse. Međutim, naša preporuka je instalacija i korištenje editora vim koji, iako kompleksniji za savlađivanje, dugoročno nudi značajno bolje okruženje u odnosu na gedit. Instalacija vim editora moguća je iz komandne linije unošenjem sekvence komandi:

```
sudo apt-get update ①  
sudo apt-get install vim-gtk ②
```

- ① Osvježava listu paketa koji su na raspolaganju za instalaciju na lokalni sistem.
- ② Instalira paket pod imenom vim-gtk, koji sadrži editor vim.

Napomena

sudo izvršava datu komandu sa administratorskim privilegijama. Da bi izvršenje ove komande bilo dozvoljeno, korisnik koji poziva sudo mora biti na listi tzv. sudo korisnika. Obzirom da rezultat izvršenja proslijedene komande potencijalno unosi promjene u kompletan sistem, korisnik mora izvršiti potvrdu unosom svoje šifre.

Konfiguracija za kompajliranje

Da bismo provjerili da je instalacija Elka obavljena uspješno, možemo kreirati i kompajlirati jednostavni C program.

```
moj@komp$ mkdir -p ~/test ❶
moj@komp$ cd ~/test ❷
moj@komp$ vim foo.c ❸
moj@komp$ gcc -o bar foo.c ❹
moj@komp$ ./bar ❺
$=0
$=1
$=2
$=2
$=1
```

- ❶ Kreiramo direktorij za pokusne programe.
- ❷ Prelazimo u kreirani direktorij.
- ❸ Editorom vim u trenutnom direktoriju kreiramo datoteku `foo.c`.
- ❹ Kompajliranjem datoteke `foo.c` pravimo program pod imenom `bar`.
- ❺ Izvršavamo program `bar` iz trenutnog direktorija.

Napomena

Karakter ~ u komandoj liniji ima isto značenje kao varijabla \$HOME, tj. predstavlja home direktorij trenutnog korisnika.
Elk kompajler je aplikacija pod imenom gcc i podržava iste komandne opcije kao gcc kompajler.
Komandna opcija -o određuje ime izlaznog fajla.
Linux programi obično nemaju nikakvu ekstenziju.

Sadržaj datoteke foo.c iz prethodnog primjera, unesen putem vim editora, dat je u sljedećim linijama:

```
#include <stdio.h>
int main() {
    int i = 0;
    while ( i < 5 ) {
        printf("i=%d\n",i);
        ++i;
    }
    return 0;
}
```

U prethodnom primjeru, obzirom da nismo koristili opciju target, kompajler je proizveo program za Linux platformu i Intel 64-bitni procesor, tj. ono okruženje unutar kojeg se vrši kompajliranje programa. Da bismo proizveli program za 32-bitni MIPS procesor, na rasplaganju imamo sljedeće četiri target opcije:

mipsel-linux-eng

Procesor u malom endian modu sa hardverskom podrškom za rad sa realnim brojevima.

mipsel-linux-engsf

Procesor u malom endian modu bez hardverske podrške za rad sa realnim brojevima.

mips-linux-eng

Procesor u velikom endian modu sa hardverskom podrškom za rad sa realnim brojevima.

mips-linux-engsf

Procesor u velikom endian modu bez hardverske podrške za rad sa realnim brojevima.

Obzirom da je naša ciljana platforma MIPS procesor u malom endian modu sa koprocesorom za FPU operacije, svi primjeri u daljem tekstu koristit će komandnu opciju `mipsel-linux-eng`.

Ukoliko proizvedemo program za MIPS platformu, za njegovo izvršenje bismo trebali imati odgovarajući hardver sa MIPS procesorom i instaliranim Linux operativnim sistemom, što je čest slučaj sa jeftinim kućnim mrežnim usmjerivačima. Čak i za slučaj da posjedujemo takav hardver, prebacivanje programa sa računara na kojem je vršeno kopajliranje i njegovo izvršenje na kućnom usmjerivaču nije pretjerano praktično. Zbog toga ćemo koristiti već navedenu mogućnost koju pruža Elk — da na Intel platformi, putem Qemu aplikacije, za MIPS program simuliramo potrebno okruženje. Npr:

```
moj@komp$ cd ~/test/
moj@komp$ ls
bar foo.c ①
moj@komp$ gcc -target mipsel-linux-eng \ ②
> -o mips_bar foo.c
moj@komp$ ls
bar foo.c mips_bar ③
moj@komp$ ./bar ④
1=0
1=1
1=2
1=3
1=4
moj@komp$ ./mips_bar ⑤
bash: ./mips_bar: cannot execute binary file: Exec format error
moj@komp$ qemu-mipsel mips_bar ⑥
1=0
1=1
1=2
1=3
1=4
```

Napomena

Aplikacija `ls` prikazuje sadržaj direktorija. Pozvana bez ikakvih ulaznih parametara, ispisuje sadržaj trenutnog direktorija.

- ❶ Sadržaj po...
- ❷ Kompajli...
- ❸ Folder sa...
- ❹ Pokretanj...
- ❺ Neuspješ...
- ❻ tel platform...
- ❾ Pokretanj...

Da bismo re...

opcijom, može...

u `.bashrc`:

```
alias mipsec...
```

Napomena

alias omogu...

U našem pri...

pokretati, a `-ta...`

na komandna o...

Nakon nove...

žemo kompajli...

```
moj@komp$ cd ~
moj@komp$ mips...
moj@komp$ qemu...
1=0
1=1
1=2
1=3
1=4
```

Konfiguracija za k...

MIPS procesor u malom
svi primjeri u dalj-
mipsel-linux-eng.

platformu, za njegovo iz-
hardver sa MIPS proceso-
mem, što je čest slučaj sa
a. Čak i za slučaj da posje-
rama sa računara na kojem
e na kućnom usmjerivaču
no koristiti već navedenu
platformi, putem Qemu apli-
bno okruženje. Npr:

Le: Exec format error

Pozvana bez ikakvih ulaz-
direktorija.

alati za kompajliranje i analizu

- ❶ Sadržaj pokusnog direktorija prije kreiranja novog programa.
- ❷ Kompajliranje programa za MIPS platformu.
- ❸ Folder sadrži novu datoteku pod imenom `mips_bar`.
- ❹ Pokretanje prethodno kompajliranog programa za Intel plat-
formu.
- ❺ Neuspješan pokušaj izvršenja MIPS programa direktno na In-
tel platformi.
- ❻ Pokretanje MIPS programa unutar emulatora daje željeni re-
zultat.

Da bismo reducirali tipkanje komande `ecc` sa željenom `target` opcijom, možemo uvesti alias komandu dodavanjem sljedeće linije u `.bashrc`:

```
alias mipsecc="ecc -target mipsel-linux-eng"
```

Napomena

alias omogućava pokretanje bilo kojeg programa sa djelimično una-
prije definiranim komandnim opcijama, i to unošenjem *alias* strin-
ga.

U našem primjeru, `mipsecc` je *alias string*, `ecc` program koji će se
pokretati, a `-target mipsel-linux-eng` je jedina unaprijed definira-
na komandna opcija.

Nakon nove modifikacije `.bashrc` datoteke, mips programe mo-
žemo kompajlirati kao u sljedećem primjeru.

```
moj@komp$ cd ~/test
moj@komp$ mipsecc -o prog foo.c
moj@komp$ qemu-mipsel prog
i=0
i=1
i=2
i=3
i=4
```

Konfiguracija za debagiranje

Kao što je ranije navedeno, kompjuterski lanac dolazi sa različitim alatima za analizu programa. Ovi alati se u opštem slučaju dijele u dvije skupine:

- Alati koji operiraju direktno na izvršnoj datoteci, npr. radi konverzije u druge izvršne formate, disasemliranje segmenata programa itd.
- Alati koji rade dinamičku analizu programa u izvršenju, npr. radi otklanjanja grešaka, procjene i unapređenja performansi itd.

Debageri spadaju u drugu skupinu alata. Omogućavaju zaustavljanje programa na određenim lokacijama u kodu ili nakon izvršenja svake komande u programu. Po zaustavljanju programa daju mogućnost uvida u stanje memorije i procesora.

Primarna namjena debagera je otklanjanje grešaka u programima koje nisu mogle biti uhvaćene u procesu kompajliranja. Međutim, mogućnost observacije posljedica izvršenja individualnih procesorskih instrukcija čini ove alate idealnim za učenje procesorskih arhitektura.

Elk uključuje aplikaciju pod imenom `ecc-gdb`, standardni `gdb` debager za različite procesorske arhitekture preuzet iz GNU projekta. Radi intimnog upoznavanja sa radom `mips` procesora, u narednim poglavljima koristit ćemo `gdb` u kombinaciji sa simulatorom `qemu`.

Debagere obično koristimo za analizu programa na lokalnom računaru. Međutim, `gdb` uključuje i opciju debagiranja udaljenog računara preko mreže, a `qemu` može na mreži da glumi računar koji dozvoljava kontrolu izvršenja programa putem debagera. Dakle, odabirom ovih opcija putem `gdb` debagera, možemo pratiti stanje `mips` procesora i memorije dok se program izvršava unutar `qemu` simulatora.

Očigledno je da za ovakvu konfiguraciju obje aplikacije moraju da rade paralelno, zbog čega moramo dva puta pokrenuti terminal.

U prvoj instanci terminala aktiviramo qemu u posebnom modu tako da, umjesto neposrednog izvršenja cjelokupnog proslijeđenog programa, čeka na instrukcije debagera na određenom mrežnom portu.

```
moj@komp$ cd ~/test  
moj@komp$ mpsecc -g -o prog foo.c ①  
moj@komp$ qemu-mipsel -g 1234 prog ②
```

Napomena

Komandna opcija -g za ecc nalaže kompajleru da u program doda simbole za debagiranje, tako da debager može koristiti imena varijabli i ostalih definicija iz programa.

Komandna opcija -g za qemu predstavlja broj mrežnog porta na kojem simulator čeka na konekciju debagera radi kontrole toka izvršenje proslijeđenog programa.

- ① Nanovo kompajliramo program, ovaj put sa uključenim simbolima za debager.
- ② Pokrećemo qemu u modu čekanja na debager sesiju, i to na portu broj 1234.

U drugom terminalu pokrećemo debager.

```
moj@komp$ cd ~/test/  
moj@komp$ ecc-gdb -q prog  
Reading symbols from prog...done.
```

Ovim počinje sesija u koju unosimo sljedeće gdb komande:

```
(gdb) target remote :1234 ①  
Remote debugging using :1234  
0x80400160 in _start ()  
(gdb) break main ②  
Breakpoint 1 at 0x400208: file foo.c, line 3.  
(gdb) continue ③  
Continuing.
```

```

Breakpoint 1, main () at foo.c:3
3     int i = 0;
(gdb) list ④
4         #include <stdio.h>
5     int main() {
6         int i = 0;
7         while ( i < 5 ) {
8             printf("i=%d\n",i);
9             ++i;
10        }
11    }
12
(gdb) break 6 ⑤
Breakpoint 2 at 0x400240: file foo.c, line 6.
(gdb) print i ⑥
$1 = 0
(gdb) c ⑦
Continuing.

Breakpoint 2, main () at foo.c:6
6         ++i;
(gdb) n ⑧
7         while ( i < 5 ) {
(gdb) n
8             printf("i=%d\n",i);
(gdb) print i
$2 = 1
(gdb) n

Breakpoint 2, main () at foo.c:6
6         ++i;
(gdb) quit ⑨
A debugging session is active.

Inferior 1 (Remote target) will be killed.

Quit anyway? (y or n) y

```

Da bismo izvršili povezivanje sa qemu programom koji je u stanju čekanja u drugom terminalu, u liniji ① dajemo instrukciju za gdb da se poveže sa udaljenim programom za debagiranje na portu 1234 lokalnog mrežnog interfejsa.

Iz odgovora na prethodnu komandu, možemo zaključiti da Linux programi počinju od funkcije iz standardne biblioteke, pod imenom `_start`, koja je implicitno uvezana sa našim programom u procesu kompajliranja programa. Ova funkcija će u nekom trenutku pozvati funkciju `main`, definiranu u našem kodu. Da bi program stao sa iz-

vođenjem na željenom mjestu, u liniji ❷ postavljamo tačku prekida u programu pomoću komande `break`. U liniji ❸ puštamo program da nastavi sa izvođenjem do prve tačke prekida.

S obzirom na to da je jedina tačka prekida u našem programu postavljena u funkciji `main`, program staje u prvoj liniji te funkcije, gdje je definirana varijabla `i`.

Pomoću komande `list` u bilo kojem trenutku možemo prikazati kod iz programa kojeg debagiramo. Pozvana bez parametara, kao u liniji ❹, ova komanda ispisuje 10 linija koda iz fajla čija se funkcija trenutno izvršava.

Pored imena funkcije, komanda `break` kao parametar može uzeti broj linije u kodu gdje želimo da postavimo novu tačku prekida, kao što je to uradeno u liniji ❺.

Bitna karakteristika debager okruženja je mogućnost ispisivanja trenutnih vrijednosti varijabli. Jedna od komandi koja služi ovoj namjeni je komanda `print`, putem koje ispisujemo trenutnu vrijednost varijable `i` u liniji ❻.

Gdb prihvata i skraćena imena za komande, kao u liniji ❼ gdje je unešen karakter `c` umjesto komande `continue`, ili u liniji ⪻ gdje karakter `n` predstavlja komandu `next`. Primjećujemo da komanda `next` izvršava tačno jednu liniju koda.

Konačno, u liniji ⪯, komandom `quit` prekidamo sesiju debagiranja, što će imati za posljedicu i gašenje `qemu` instance u drugom terminalu.

Pokretanje i koordinacija dva terminala tokom debager sesije, kao u prethodnom primjeru, nije praktična obicom da se sva interakcija odvija u terminalu gdje se izvršava `gdb`. Da bismo istovremeno jednom komandom pokrenuli obje aplikacije u istom terminalu, možemo napisati posebnu `bash` komandu, u obliku funkcije

`run_mips`, koja će pokretati qemu tiho u pozadini i dati kontrolu terminala debageru.

Slijedi kod funkcije `run_mips`, koji je potrebno dodati na kraj `.bashrc` skripte. Funkcija uzima jedan paramater, a to je program koji se debugira.

```
function run_mips() {  
    qemu-mipsel -g 1234 $1 &> /dev/null & ❶  
    ecc-gdb -q $1 ❷  
}
```

Pokrećemo simulator sa istim komandama kao i ranije. Simbol `$1` će biti zamijenjen imenom programa kojeg debugiramo u trenutku pozivanja funkcija `run_mips`. Bilo kakav ispis iz pokrenutog programa, putem operatora `&>` preusmjeravamo na uređaj `/dev/null`, koji zanemaruje proslijeđene podatke. Upotrebom karaktera `&` na kraju komande program pokreće se u pozadini, što nam omogućava unos novih komandi u terminalu paralelno sa izvršavanjem simulatora.

❷ Pokrećemo i debager sa `$1`, tj. imenom programa kojeg debugiramo.

Nakon definiranja funkcije `run_mips`, programe za mips platformu možemo analizirati u jednom terminalu pomoću samo jedne komande. Tako bismo prethodni primjer debugiranja mogli započeti na sljedeći način:

```
moj@komp$  
moj@komp$ cd ~/test  
moj@komp$ run_mips prog  
[1] 18219  
Reading symbols from prog...done.  
(gdb) target remote :1234  
Remote debugging using :1234  
0x00400568 in _start ()
```

za dati i dati kontrolu ter-

prebno dodati na kraj
parametar, a to je program

ama kao i ranije. Sim-
programa kojeg debagiramo
mips. Bilo kakav ispis iz
preusmjeravamo
proslijedene podatke.
ende program pokreće-
nos novih komandi u
simulatora.

programa kojeg deba-

gramme za mips platfor-
pomoću samo jedne ko-
debagiranja mogli započeti

U ovom poglavlju smo debagiranje i kontrolu toka izvršenja pro-
grama napisanog u programskom jeziku C vršili na simuliranom
mips računaru. Ako bismo isti program kompajlirali nanovo za In-
tel platformu a zatim pokrenuli u debager sesiji sa identičnim ko-
mandama, ovaj put bez simulatora, dobili bismo identičan rezultat,
bez ozbira što se program izvršava na različitom procesoru.

Tek na nivou asemblera, debagiranje i analiza programa može
dati uvid u instrukcije koje se daju različitim procesorima. Ovu či-
njenicu potvrdit ćemo kroz naredna poglavlja u kojim ćemo anali-
zirati programe na nivou asemblera za mips platformu.

Osnove asemblera

Programiranje na nivou asemblera se konceptualno ne razlikuje mnogo u odnosu na pisanje programa u ostalim programskim jezicima. Asembler programi pišu se u tekstualnim fajlovima, obično sa ekstenzijom `s` ili `asm`, i čitljivi su upućenim programerima. Ključna razlika u odnosu na programske jezike visokog nivoa je u činjenici da asembler ne nudi nikakvu mogućnost apstrakcije, tj. prilikom programiranja smo ograničeni na onaj skup instrukcija i tipova podataka koje podržava procesor za kojeg pišemo program.

Tokom ovog poglavlja upoznat ćemo se sa strukturom programa na najnižem nivou. Uspostaviti ćemo vezu između izvršnih fajlova, binarnih reprezentacija kompajliranih programa na disku i aplikacija u izvršenju, tj. programa dok okupiraju memoriju. S obzirom na to da Elk integrira asemblere i linkere iz binutils¹ komponente GNU projekta, svi primjeri će biti prezentirani u posebnoj notaciji koju koristi `gas` asembler.

Kostur asembler programa

Početni primjeri u knjigama koje tretiraju problematiku programiranja u bilo kojem programskom jeziku obično uključuju jednostavnu aplikaciju koja prilikom pokretanja ispisuje pozdravnu poruku. Takav program na nivou asemblera suviše je kompleksan da bi služio kao uvodni primjer. Stoga će naš prvi asemblerski program, oko kojeg ćemo formirati kostur za sve buduće programe, biti takav da

1. <http://www.gnu.org/software/binutils/>

je ekvivalentan najmanjem C kodu od kojeg je moguće proizvesti izvršnu datoteku, tj. program

```
int main() {
    return 0;
}
```

Gornji program moguće je kompajlirati, ali njegovo pokretanje ne bi proizvelo nikakav rezultat. Isti program, napisan u mips asembleru, ima sljedeću strukturu:

```
.section .text
.set noreorder
.global main
main:
    addi $v0, $0, 0
    jr $ra
    nop
```

Program pokazuje neke bitne osobine gäs koda. Analizom njegove strukture, možemo zaključiti da asembler fajlovi mogu imati tri tipa elemenata:

direktive

Počinju karakterom . i predstavljaju informacije ili upute koje asembler koristi u procesu asembliranja.

oznake

Završavaju se karakterom : i asociraju pozicije (lokacije) u asembler kodu sa proizvoljnim imenima.

instrukcije

Ne spadaju niti u jednu od prethodnih kategorija, a predstavljaju instrukcije datog procesora koje će da čine program.

GNU asembler podržava veliki broj direktiva. Određene direktive, kao npr. .section, .text ili .global iz primjera, prisutne su u gotovo svim asembler fajlovima, dok se ostale direktive koriste rjeđe. Zbog toga ćemo se prvenstveno fokusirati na bitne directive bez kojih asembler programi nemaju smisla.

Proces asembleriranja i uvezivanja

radi objašnjenja direktive `section`, potrebno je da se prvo osvrnu na proces nastanka izvršnih fajlova (datoteka). Ove datoteke predstavljaju programe u formi za direktno izvršenje na računarima i distribuiraju se u različitim binarnim formatima povezanim sa konkretnim operativnim sistemima, npr. ELF¹ format za Unix sisteme, ili PE² format Microsoft Windows OS-a.

Zajednička osobina svih binarnih formata je da se sastoje od *sekacija*, u kojima se nalaze različiti strukturni elementi programa koji su neophodni za njegovo normalno funkcionisanje. Kada korisnik unutar nekog operativnog sistema pokrene aplikaciju, prije nego što počne njen izvršenje, određene sekcije iz izvršnog fajla asocirane sa aplikacijom operativni sistem kopira u memoriju, nakon čega naređuje procesoru da počne izvršavati prvu instrukciju iz upravo kopiranog segmenta u memoriji sa lokacije gdje se nalazi funkcija koja je označena kao ulazna tačka u program (entry point). Na osnovu primjera debagiranja programa u prethodnom poglavlju znamo da funkcija `_start` predstavlja ulaznu tačku za Linux programe. Ova funkcija, nakon što obavi sve neophodne korake za inicijalizaciju programa, vrši pozivanje funkcije `main`.

GNU asembler može proizvoditi izlazne datoteke u različitim binarnim formatima. Podržani formati specificiraju različite tipove i imena sekcija za objektne datoteke. Međutim, dvije sekcije neizostavne su u svim formatima, i to: sekcija u kojoj se nalazi sekvenca instrukcija koje čine program, i sekcija u kojoj se nalaze globalne variable korištene u programu. S obzirom na to da je naša ciljana platforma Linux, slijedi opis sekcija koje se obično nalaze u binarnim fajlovima ELF formata:

1. <http://man7.org/linux/man-pages/man5/elf5.html>

2. <https://msdn.microsoft.com/en-us/windows/hardware/gg463119.aspx>

text

Sekcija u kojoj se nalaze sve mašinske instrukcije nastale asembleriranjem i uvezivanjem fajlova koji čine program. Sekcija često nosi i oznaku `code` sekcijske.

data

Sekcija u kojoj se nalaze vrijednosti inicijaliziranih globalnih varijabli.

bss

Sekcija rezervirana za neinicijalizirane globalne varijable.

rodata

Sekcija u kojoj se nalaze konstantne vrijednosti koje se koriste u programu.

Pored navedenih, izvršni fajl može sadržavati i druge sekcijske, npr. ukoliko se u program uključe simboli za debagiranje izvršna datoteke će sadržavati više sekcijske koje počinju riječju `debug`. Debager koristi podatke iz ovih sekcijskih tokom debagiranja programa.

Sadržaj sekcijskog izvršnog fajla definira se u asembleriskom kodu. Na slici 3-1 slikovito je prikazan pojednostavljen proces kreiranja izvršne datoteke pod imenom `foo`, koja je nastala procesom asembleriranja i uvezivanja tri asembleriska fajla: `f1.s`, `f2.s` i `f3.s`.

Asembleriranjem svakog pojedinačnog `.s` fajla nastaje odgovarajući binarni objektni fajl, pod istim imenom ali sa drugom ekstenzijom, obično `.o`. Objektni fajl sadržavat će sekcijske koje su definisane u asembler fajlu putem direktive `.section`. Poslije svake `.section` direktive u asembler fajlu slijedi direktiva kojom se daje ime odredene sekcijske iz objektnog fajla. Nakon toga, pa sve do nove `.section` direktive, asembler popunjava imenovanu sekcijsku objektnog fajla interpretacijom elemenata iz asembleriskog fajla koji slijede nakon ove direktive.

Slika 3-1. Pri-

Asembler
sekcijske objektni
procesorske
bss i data sekcije
prostor u kojem
odvojen u različite
binarnog fajla
nih varijabli

Asembler
različite načine
kodu od kojih
se nastavlja
ostalih naredbi
od kojih počinje

sekcijske nastale asemblerom program. Sekcija često

znači uvezivanje globalnih varijabli.

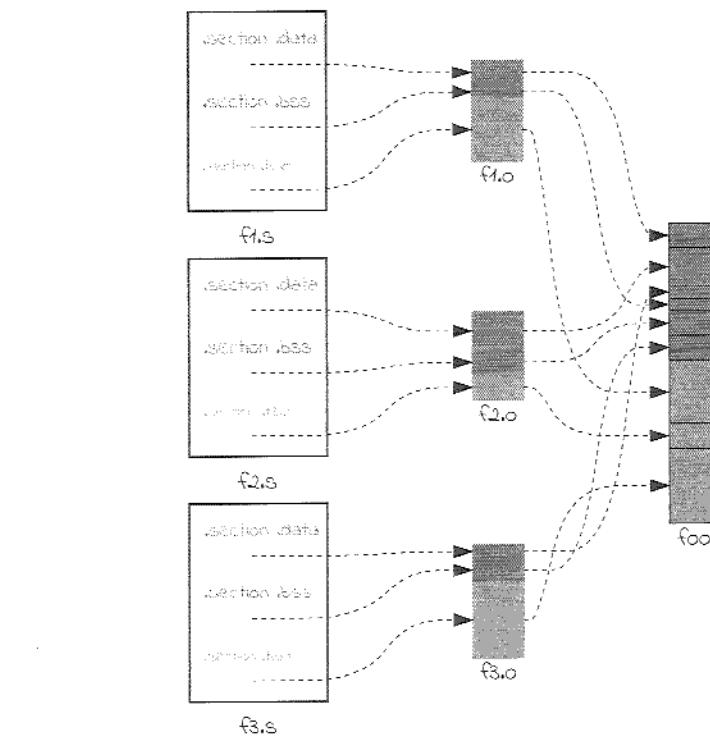
globalne varijable.

sekcije koje se koriste u

text sekcijski i druge sekcije, npr. učitavanje i izvršavanje izvršna datoteke ili učitavanje debug. Debager koristi za učitavanje programa.

u asemblerском kodu. Učitavanje proces kreiranja izvršnog programa procesom asemblera. f1.s, f2.s i f3.s.

u objektnog fajla nastaje odgovarajući objektni fajl sa drugom ekstenzijom. Sekcije koje su definirane u asembleru svake .section direktyve se daje ime određene sekcijske oznake do nove .section direktive. Sekciju objektnog fajla inicijalizacija koja slijede nakon ove



Slika 3-1. Primjer asembliranja i uvezivanja

Asemblerski elementi koji se koriste za kreiranje sadržaja text sekcijske objektnog fajla obično su sekvence instrukcija u skladu sa procesorskom platformom za koju se program piše. Sekcije rodata, bss i data su sastavljene od niza asembler direktiva kojima se alocira prostor u korespondirajućim sekcijskim objektnog fajla. Prostor odvojen u data sekcijsku popunjava se vrijednostima odgovarajućeg binarnog formata, i to u skladu sa željenom inicijalizacijom globalnih varijabli.

Asemblerski elementi oznake se koriste u svim sekcijskim, ali za različite namjene. Unutar text sekcijske, oznake markiraju lokacije u kodu od kojih počinju prve instrukcije funkcija ili lokacije od kojih se nastavlja izvršenje programa nakon obavljenog grananja. Unutar ostalih nabrojanih sekcijskih, oznake se povezuju sa lokacijama u sekcijskim od kojih počinju određene varijable. Ukoliko se asemblerška oznaka

unutar određenog asembler fajla, putem direktive `global` ili `globl`, označi kao globalna, ta oznaka će biti dostupna u drugim asemblerima fajlovima čiji se objektni fajlovi uvezuju u zajednički izvršni fajl.

Nakon što se izvrši asembliranje slijedi proces uvezivanja u kojem linker kreira sadržaj programa sastavljanjem sekcija iz pojedinačnih objektnih fajlova. Redoslijed procesiranja i lokacije na kojima će se sekcije iz pojedinačnih objektnih fajlova završiti u izvršnom fajlu ovise o linker skripti koju linker konsultuje tokom uvezivanja. Svaki kompjuterski lanac dolazi sa predefinisanom linker skriptom, pomoću koje se vrši uvezivanje korisničkih aplikacija na datom operativnom sistemu. Linker obično objedinjuje pojedinačne sekcije iz različitih objektnih fajlova tako da čine jednu veliku zajedničku sekciju u izvršnom fajlu, kao što je prikazano različitim bojama na slici 3-1. Tokom uvezivanja linker mora vršiti korekciju adresa dobivenih na osnovu oznaka iz pojedinačnih asemblera fajlova jer se njihove adrese mijenjaju objedinjavanjem u zajedničkom izvršnom fajlu.

Nakon objašnjenja direktiva `section` i `text` nastavimo sa asembliranjem našeg primjera sljedećom sekvencom bash komandi:

```
moj@komp$ cd ~/test
moj@komp$ ls
prog.s
moj@komp$ cat prog.s ①
.section .text
.set noreorder
.global main
main:
    addi $v0, $a1, 0
    jr $ra
    nop
moj@komp$ mipsecc -g -o prog prog.s ②
moj@komp$ ls
prog  prog.s
moj@komp$ qemu-mipsel prog ③
moj@komp$
```

Napomena

Program `cat` ispisuje sadržaj proslijeđene datoteke.

directive global ili globl, u drugim asemblerima za zajednički izvršni fajl.

proces uvezivanja u kojem sekcija iz pojedinog sekcija i lokacije na koji će se završiti u izvršnom programu tokom uvezivanja. Iznosom linker skriptom, aplikacija na datom operativnom sistemu uključuje pojedinačne sekcije iz jednu veliku zajedničku sekciu različitim bojama na slici s korekciju adresa dobivenim asemblera fajlova jer se njihova zajedničkom izvršnom fajlu.

i text nastavimo sa asemblerom bash komandi:

datoteke.

- ① Ispisujemo sadržaj datoteke prog.s koja je prethodno kreirana putem editora.
- ② Asemblerimo fajl prog.s i uvezujemo sa standardnom bibliotekom u izvršni fajl prog.
- ③ Kao što je očekivano, program pokrenut u simulatoru ne proizvodi ispis.

GNU asembler distribuira se u obliku programa pod imenom as, a linker pod imenom ld. Ove aplikacije prisutne su u bin direktoriju Elk, ali pod različitim imenima, i to: mips-elf-as kao asembler za mips platformu, i ecc-ld kao linker koji se može koristiti za bilo koju procesorsku platformu.

Međutim, prilikom kreiranja izvršnog fajla u prethodnom primjeru nismo eksplicitno koristili asembler i linker, već ecc kompajler. Pozvan na način kao u liniji ②, ecc kompajler prvo vrši asembleriranje proslijedene datoteke u odgovarajuću objektnu datoteku, nakon čega kompajler kreira izvršnu datoteku uvezivanjem sa odgovarajućim fajlovima iz standardne biblioteke koja se nalazi u posebnom direktoriju kompajlerskog lanca. Moguće je tražiti od kompajlera da prikaže sve izvedene korake prilikom asembleriranja, tako što dodavamo opciju -v:

```
moj@komp$ cd ~/test
moj@komp$ mipsecc -v -g -o prog prog.s
ecc 9.1.14 based on clang version 3.7.0 (based on LLVM 3.7.0svn)
Target: mipsel-ellcc-linux
Thread model: posix
"/home/moj/programi/ellcc/bin/mips-elf-as" -o /tmp/prog-d48241.o \
    -f prog.s ①
"/home/moj/programi/ellcc/bin/ecc-ld" -nostdlib ② \
    -L/home/moj/programi/ellcc/lib/mipsel-linux-ung ③ \
    -m elf32linux -build-id --eh-frame-hdr -o prog -e _start ④ \
    -static \
    /home/moj/programi/ellcc/tlbecc/lib/mipsel-linux-eng/crti.o ⑤ \
    /home/moj/programi/ellcc/tlbecc/lib/mipsel-linux-eng/crtbegin.o ⑤ \
    /tmp/prog-d48241.o ⑥ \
    -lc -lcompiler-rt -) ⑦ \
    /home/moj/programi/ellcc/lib/mipsel-linux-eng/crtend.o ⑤
moj@komp$
```



Bitne komandne opcije za linker:

-L

Dodaje direktorij u listu direktorija u kojim linker traži objektne fajlove ili biblioteke koji nemaju apsolutnu stazu.

-e

Postavlja ulaznu tačku u programu.

-l

Dodaje biblioteku sa kojom se vrši uvezivanje, pri čemu se u imenu biblioteke izostavlja prefiks `lib` i ekstenzija `.a`.

-m

Definira platformu za koju se vrši uvezivanje.

-
- ❶ Asemblira `prog.s` putem `mips-elf-as` u malom endian modeu (opcija `-EL`), nakon čega se proizvodi objektni fajl `prog-d40241.o` u direktoriju `/tmp`.
 - ❷ Poziv linkera `ecc-ld` sa komandnim opcijama u više linija.
 - ❸ Specificira linkeru direktorij u kojem se nalaze C standardne biblioteke.
 - ❹ Između ostalog, za linker definira ime izvršne datoteke `prog` i postavlja da je ulazna tačka u programu `_start`.
 - ❺ Dodaje za uvezivanje objektne fajlove `crt1.o`, `crtbegin.o` i `crtend.o`, sa kojim se uvezuje svaka aplikacija za Linux OS.
 - ❻ Dodaje za uvezivanje asemblirani fajl `prog-d40241.o`.
 - ❼ Dodaje za uvezivanje biblioteke `libc.a` i `libcompiler-rt.a`.

PS asset
aktivitet
funkcija
asembler
na GNU
r. Posjet
između raz
Lokacije
je asembler
mpr. main i
luju lokaci
koje su im
no da se fu
standardne
čega je pot
glasimo glo
asembler fa

MIPS in
čina. Najče

ime_instru

Upotreba

addi \$v0
jr \$ra
nop ❸

MIPS asemb

Bitno je naglasiti da, pored raznih komandnih opcija, linker očekuje listu objektnih fajlova i biblioteka koje će učestvovati u uvezivanju u konačni izvršni fajl. Eksplisitno pozivanje linkera i asemblera odradit ćemo kasnije u posebnom poglavlju. Za sada je bitno konceptualno razumijevanje asembliranja i uvezivanja putem komajler komande ecc.

MIPS asembler instrukcije

Direktive i oznake su bitni elementi unutar asemblerских datoteka. Međutim, ključnu ulogu imaju instrukcije, putem kojih se na nivou asemblera zapravo i kreiraju programi. Za razliku od ostalih elemenata GNU asembler koda, instrukcije su vezane za konkretni procesor. Posljedica ove činjenice je da asembler programi nisu portabilni između različitih procesorskih platformi, za razliku od C programa.

Lokacije u kodu gdje zatičemo sekvene instrukcija su text sekcije asemblerских fajlova. Asemblerске oznake u ovim sekcijama, kao npr. `main` iz našeg osnovnog asembler primjera, obično predstavljaju lokacije od kojih počinju prve instrukcije iz asembler funkcija koje su imenovane putem asemblerских oznaka. Već ranije je rečeno da se funkcija `main` poziva iz funkcije `_start` koja je sastavni dio standardne biblioteke, tj. definirana u drugom asembler fajlu, zbog čega je potrebno da oznaku `main`, pomoću direktive `global`, proglašimo globalnom, a time i raspoloživom za korištenje u drugim asembler fajlovima.

MIPS instrukcije u asembler fajlovima formatiraju se na više načina. Najčešći je sljedeći format:

ime_instrukcije operand1, operand2, operand3

Upotrebu ovog formata već smo vidjeli u prethodnom primjeru:

```
addi $v0, $0, 0 ①  
jr $ra ②  
nop ③
```

Većina instrukcija podržava tri operanda, kao npr. instrukcija addi u liniji ①. Međutim, postoje instrukcije koje uzimaju manje operanada, kao npr. instrukcija jr u liniji ②, koja uzima jedan operand, ili instrukcija nop u liniji ③, koja uopšte ne uzima operande.

Operandi u instrukcijama mogu biti: brojevi, registri ili oznake.

Brojevi, ukoliko se koriste u instrukcijama, pojavljuju se obično kao treći operand, kao npr. 0 u liniji ①. Brojevi mogu biti zadani u decimalnom ili heksadecimalnom formatu, sa ili bez predznaka, odnosno na isti način kao u programskom jeziku C.

Operandi registri počinju karakterom \$ i mogu biti referencirani simboličkim imenima, npr. \$v0 ili \$ra u liniji ① i ②, ili njihovim indeksom u registar fajlu MIPS procesora, npr. \$0 u liniji ③.

MIPS procesor u registar fajlu ima 32 registra koji se mogu koristiti kao operandi u instrukcijama. Slijedeća tabela daje pregled svih registara iz registar fajla:

Tabela 3-1. MIPS registri iz registar fajla

Indeks	Simboličko ime	Opis
0	zero	vrijednost 0
1	\$at	za asembler upotrebu
2-3	v0 – v1	povratne vrijednosti funkcija
4-7	a0 – a3	argumenti funkcija
8-15	t0 – t7	privremene vrijednosti
16-23	s0 – s7	snimljene vrijednosti
24-25	t8 – t9	privremene vrijednosti
26-27	k0 – k1	rezervirano za OS

Pretevi, registri ili oznake.

Uzimajući u obzir da su pojavljuju se obično **čvorovi**, mogu biti zadani u **čvorovima** sa ili bez predznaka, odnosno u **čvorovima** na jeziku C.

U liniji ❶ i ❷, ili njihovim in-
a, npr. \$0 u liniji ❶.

2 registra koji se mogu koris-
ćeca tabela daje pregled svih

Indeks	Simboličko ime	Opis
0	\$gp	globalni pointer
1	\$sp	stek pointer
2	\$fp	frejm pointer
3	\$ra	povratna adresa

Iako navedeni registri za odredene kontekste imaju specifične namjene, u skladu sa opisima u tabeli, u opštem slučaju instrukcije ih mogu slobodno mijenjati. Tokom svog izvršenja MIPS instrukcije mogu čitati vrijednosti iz maksimalno dva registra. Nakon obavljene operacije nad vrijednostima iz operanada, instrukcije mogu promijeniti sadržaj samo jednog registra.

Dok traje izvršenje odredene instrukcije poseban MIPS registar označen kao `pc`, koji se ne nalazi u registar fajlu, uvijek sadrži adresu naredne instrukcije za izvršenje u programskoj sekvenci. Dakle, nusprodukt izvršenja svake instrukcije je promjena vrijednosti ovog registra. Posebne instrukcije za grananje mogu mijenjati vrijednost `u` ovom registru na način da se preusmjeri tok izvršenja programa na drugu lokaciju u asemblerskom kodu.

Navedene karakteristike zajedničke su za većinu instrukcija navedenog formata. Detalje o pojedinim instrukcijama predstaviti ćemo u narednim poglavljima.

Debagiranje asembler programa

Debagiranje programa na nivou asemblera daje nam mogućnost uvida u stanje svih registara MIPS procesora, prije i poslije izvršenja pojedinačnih instrukcija u programu.

Ovo je najbolje praktično prikazati debagiranjem programa programom putem komande `run mips` kao u sljedećem primjeru:

```

moj@komp$ cd ~/test/
moj@komp$ ls
prog prog.s
moj@komp$ run_mips prog
[1] 4197
Reading symbols from prog...done.

```

Nakon čega počinje sljedeća gdb sesija.

```

(gdb) target remote :1234
Remote debugging using :1234
0x00400180 in _start ()
(gdb) print $pc ❶
$1 = 4194688
(gdb) print/x $pc ❶
$2 = 0x400180
(gdb) p/t $pc ❶
$3 = 100000000000000110000000
(gdb) print _start ❷
$4 = {<text variable, no debug info>} 0x400180 <_start>
(gdb) print main ❸
$5 = {<text variable, no debug info>} 0x4001f4 <main>
(gdb) x/i *main ❹
    0x4001f4 <main>:      addi      v0,zero,0
(gdb) break *main
Breakpoint 1 at 0x4001f4
(gdb) c
Continuing.

Breakpoint 1, 0x004001f4 in main ()
(gdb) p/x $pc ❺
$6 = 0x4001f4
(gdb) disp/i $pc ❻
1: x/i $pc
=> 0x4001f4 <main>:      addi      v0,zero,0
(gdb) ni ❼
0x004001f8 in main ()
1: x/i $pc
=> 0x4001f8 <main+4>:      jr      r31
    0x4001fc <main+8>:      nop
(gdb) ni ❼
0x0040043c in __libc_start_main (main=0x4001f4 <main>, argc=1,
    argv=0x7fffff894) at src/env/_libc_start_main.c:73
73      src/env/_libc_start_main.c: No such file or directory.
1: x/i $pc
=> 0x40043c <__libc_start_main+564>:      jal      0x40044c <exit>
    0x400440 <__libc_start_main+568>:      move    $0,v0
(gdb) q
A debugging session is active.

Inferior 1 (Remote target) will be killed.

Quit anyway? (y or n) y

```

Izvršenje programa, kao i u slučaju C programa, počinje od lokacije u kodu asociranom sa oznakom `_start`. Ova oznaka definirana je u objektnom fajlu iz standardne biblioteke.

Kao što je prikazano u linijama označenim sa ①, sadržaj registara možemo ispisivati putem komande `print`, pri čemu karakter koji sledi nakon / diktira format ispisa. Ukoliko se izostavi format, ispis je decimalan. Dozvoljeni formati ispisa za ovaj kontekst su:

- t binarni
- d decimalni
- x heksadecimalni

Upotreboom komande `print` na asembler oznakama, kao u linijama markiranim sa ②, konstatujemo da su oznake asocirane sa memorijskim lokacijama koje su prikazane u ispisu. Registar \$pc ima vrijednost kao i oznaka `_start`, što je i očekivano jer registar \$pc pokazuje na memorijsku lokaciju sa koje se preuzima sljedeća instrukcija u programu, a program je trenutno zaustavljen na prvoj instrukciji funkcije `_start`.

Potem komande x možemo izvršiti inspekciju sadržaja memorijih lokacija. Komandom u liniji ③, sa kontrolnim parametrima /1i, zahtijevamo ispis jedne instrukcije koja se nalazi na memorijskoj lokaciji asociranoj sa oznakom `main`. Potvrđujemo da je to instrukcija `addi`, identično kao u našem izvornom asembler fajlu.

Nakon što postavimo tačku prekida u funkciji `main`, te nastavimo sa izvođenjem programa, u liniji ④ ponovnom inspekcijom vrijednosti registra \$pc konstatujemo da je program zaista stao na lokaciji asociranoj sa oznakom `main`.

```
1f4 <main>, argc=1,  
at_main.c:73  
such file or directory.  
  
jal      0x40044c <exit>  
move    b0, v0  
  
e killed.
```

Debager, putem komande `disp`, može vršiti ispisivanje registara ili memorijskih lokacija automatski nakon svakog zaustavljanja programa. Putem komandnih parametara `/1` i u liniji **6**, pri svakom novom zaustavljanju zahtijevamo ispis sljedeće instrukcije za izvršenje u programu.

Nastavljamo izvršenje programa u liniji **6**, i to za samo jednu instrukciju, putem komande `ni`. Primjećujemo da će u sljedećem koraku procesor, umjesto jedne, izvršiti dvije instrukcije. Ovo je posljedica činjenice da MIPS procesor ima cjevovod. Zbog specifične implementacije cjevovoda, sve instrukcije kojim se mijenja tok programa ne izvršavaju se samostalno, već skupa sa instrukcijom koja ih slijedi. Instrukcija u sekvenci nakon instrukcije za promjenu toka nalazi se u tzv. *odgođenom slotu*. U našem primjeru, instrukcija `nop` je u odgođenom slotu instrukcije `jr`.

Nakon što u liniji **7** nanovo pokrenemo komandu `ni`, procesor izvršava dvije navedene instrukcije čime se okončava izvršenje našeg koda. Konačno, sesiju debagiranja terminiramo komandom `q`.

Ukoliko se u asembler kodu, putem direktive `set`, postavi opcija `noreorder`, tada svi odgođeni slotovi nakon direktive postaju direktno dostupni u asembler kodu. Programer ima dužnost da ove slobove popuni odgovarajućim instrukcijama.

Ukoliko se izostavi direktiva sa opcijom `noreorder` ili ako se pomocu `set` eksplicitno postavi opcija `reorder`, odgođeni slotovi ne bi bili vidljivi u asembler kodu. Tokom asembliranja asembler ih popunjava na način da pokuša pronaći neovisnu instrukciju u kodu prije instrukcije grananja čije pomjeranje u odgodeni slot ne bi utjecalo na ispravnost izvođenja programa kao cjeline. Ukoliko ne uspije pronaći navedenu instrukciju, asembler popunjava odgođeni slot sa instrukcijom `nop`, koja troši procesorsko vrijeme ne radeći ništa.

ispisivanje registara pri svakog zaustavljanja programi, pri svakom naredbi instrukcije za izvršenje

to za samo jednu instrukciju da će u sljedećem kočiti instrukcije. Ovo je potonuvod. Zbog specifične načina se mijenja tok programa sa instrukcijom koja koči instrukcije za promjenu toka u programu, instrukcija nop je

komandu ni, procesor se ne končava izvršenje naredbe, izlazimo komandom q.

direktive set, postavi opciju direktno, direktive postaju direktivama dužnost da ove slo-

moreorder ili ako se postavi, odgođeni slotovi ne bi smatrani za izvršavanja asembler ih postavlja u istu instrukciju u kodu jer odgođeni slot ne bi utjecao na cijeline. Ukoliko ne uspije popunjava odgođeni slot u vrijeme ne radeći ništa.

Ukoliko bismo prethodni primjer preformulisali bez opcije no moreorder, dobili bismo sljedeći asemblerski program:

```
section .text
global main
main:
    addi $v0, $0, 0
    jr $ra
```

Naredna debager sesija prethodnog programa otkriva da je asembler konstatovao neovisnu instrukciju prije instrukcije jr i izvršio popunjavanje odgođenog slota sa tom instrukcijom u objektnom fajlu.

```
(gdb) target remote :1234
Remote debugging using :1234
0x00400100 in _start ()
(gdb) x/2i *main
0x4001f4 <main+0>      jr      ra
0x4001f5 <main+1>      addi   v0, zero, 0
```

U ovom poglavlju detaljno smo analizirali sve elemente minimalnog asembler programa za MIPS platformu. Detalje o osnovnim aritmetičko-logičkim instrukcijama, u koje spada i instrukcija addi iz našeg primjera, predviđaćemo u narednom poglavlju.

Aritmetičko-logičke instrukcije

U ovom poglavlju ćemo analizirati osnovne aritmetičko-logičke instrukcije mips procesora kroz kompajliranje i debagiranje jednostavnih programa pisanih u asembleru.

Kao što je već navedeno u prethodnim poglavljima, mips procesor ima 32 registra. Da bi aritmetičko-logička jedinica izvršila neku operaciju, podaci moraju biti u registrima. Opšti oblik, odnosno format, aritmetičko-logičke instrukcije za mips procesor je:

```
ime_instrukcije destinacija izvor1 izvor2
```

Ime instrukcije je neka razumljiva skraćenica operacije poput instrukcije addi koja je korištena u prethodnom poglavlju i obavlja operaciju sabiranja. Destinacija predstavlja neki od mips registara iz registar fajla u koji se snima rezultat operacije. Naredna dva parametra su izvori koji su operandi za obavljanje operacije. Prvi izvor uvijek predstavlja registar, dok drugi može biti registar ili neka konstanta.

Aritmetičke instrukcije

U aritmetičke instrukcije spadaju instrukcije za osnovne aritmetičke operacije, poput operacija sabiranja, oduzimanja, množenja i dijeljenja. Postoji više tipova ovih instrukcija, a sve osnovne aritmetičke instrukcije mips procesora su date u tabeli ispod.

Tabela 4-0. Osnovne aritmetičke instrukcije mips procesora

Sintaksa	Opis
add d,s,\$t	sabiranje
addu d,s,\$t	sabiranje bez predznaka
sub d,s,\$t	oduzimanje
subu d,s,\$t	oduzimanje bez predznaka
addi d,s,C	sabiranje sa konstantom
addiu d,s,C	sabiranje bez predznaka sa konstantom
mult s,t	množenje
multu s,t	množenje bez predznaka
div s,t	dijeljenje
divu s,t	dijeljenje bez predznaka

Kao što vidimo iz prethodne tabele, instrukcije koje imaju sufiks 'u' predstavljaju operacije na brojevima bez predznaka. Prilikom njihovog izvršavanja se ne generira iznimka prilikom prekoračenja. Instrukcije koje ne sadrže sufiks 'u' generiraju iznimku prilikom prekoračenja i program blokira.

Također, možemo primjetiti da instrukcije koje imaju sufiks 'i' predstavljaju instrukcije čiji je zadnji parametar konstanta koja predstavlja operand. Važno je naglasiti da konstanta mora biti u opsegu od -32768 do 32767. Također je bitno primjetiti da ne postoje instrukcije subi te subiu iz razloga što su te operacije pokrivene instrukcijama addi i addiu jer iste uzimaju i negativne konstante.

Instrukcije za množenje i dijeljenje imaju nešto drugačiji format od instrukcija za sabiranje i oduzimanje iz razloga što rezultat ovih

operacija ne možemo smjestiti u jedan registar, tako da ne postoji destinacijski registar kao u slučaju operacija sabiranja i oduzimanja. Rezultat ovih operacija smješta se u dva regista koji su zapravo specijalni registri za smještanje rezultata operacija množenja i dijeljenja i označavaju se kao LO i HI. U registar LO se smješta donjih 32 bita rezultata, a u registar HI se smješta gornjih 32 bita rezultata u slučaju operacije množenja, dok se u slučaju operacije dijeljenja u registar LO smješta rezultat dijeljenja, a ostatak dijeljenja se smješta u registar HI.

Postoje posebne instrukcije koje pomjeraju vrijednosti iz ovih specijalnih registara u registre opšte namjene. To su sljedeće dvije instrukcije:

mfhi \$d

Pomjera vrijednost iz HI regista u registar opšte namjene \$d.

mflo \$d

Pomjera vrijednost iz LO regista u registar opšte namjene \$d.

Kao što vidimo, ove instrukcije uzimaju samo jedan parametar, tj. destinacijski registar opšte namjene.

Iako smo u prethodnom poglavlju vidjeli da mips ima 32 registra u registar fajlu, u narednim primjerima ćemo koristiti registre za privremene i povratne vrijednosti. Upotreba ostalih registara će biti objašnjena u narednim poglavljima.

U narednom primjeru ćemo predstaviti korištenje različitih vrsta instrukcija sa sabiranje i oduzimanje.

```
.section .text
.set noreorder
.global main
main:
    addi $t0, $0, 100 ①
    add $t1, $t0, $t0 ②
    sub $t2, $t1, $t0 ③
    lui $t8, 0x7fff ④
```

Aritmetičke instrukcije

47

```
ori $t8, $t8, 0xffff ⑤
addu $t3, $t8, $t8 ⑥
add $t4, $t8, $t8 ⑦
addi $v0, $0, 0
jr $ra
nop
```

U liniji ① ćemo prvo u registar \$t0 da stavimo vrijednost 100 koristeći addi instrukciju tako što ćemo sabrati proslijedenu konstantu sa registrom \$0. Zatim ćemo iskoristiti instrukciju add da bismo u registar \$t1 smjestili rezultat sabiranja vrijednosti registra \$t0 sa samim sobom. U liniji ③ ćemo u registar \$t2 smjestiti rezultat oduzimanja vrijednosti registra \$t1 i \$t0. U linijama ④ i ⑤ učitavamo u registar \$t8 veliku vrijednost kako bi mogli testirati prekoračenje (pročitati napomenu). U liniji ⑥ u registar \$t3 smještamo rezultat sabiranja dva velika broja prethodno smještena u registar \$t8, te isto ponavljamo u liniji ⑦ sa instrukcijom add da bismo vidjeli razliku između ove dvije instrukcije koja je opisana prethodno.

Napomena

U liniji ④ gornjeg primjera korištena je instrukcija lui koja učitava datu konstantu u gornjih 16 bita destinacijskog registra. Ova instrukcija se koristi da bi bili u mogućnosti učitati vrijednosti veće od 32767 koliko je maksimalna veličina za konstantu unutar instrukcije. Najčešće se koristi u kombinaciji sa ori instrukcijom kao u liniji ⑤ gdje smo učitali donjih 16 bita u destinacijski registar.

Debaging sesija prethodnog primjera se nalazi ispod.

```
moj@kempS:~/tmp$ gdb sabiranje_odeuzimanje
[2] 3125
Reading symbols from sabiranje_odeuzimanje...done.
(gdb) target remote :1234
Remote debugging using :1234
0x00400180 in __start ()
(gdb) b main
Breakpoint 1 at 0x1001f4
(gdb) c
Continuing.

Breakpoint 1, 0x004001f4 in main ()
(gdb) disp/i $pc
1: n/i $pc
=> 0x4001f4 <main>          addi    t0,zero,100
```

mo vrijednost 100 ko-
proslijedenu konstan-
strukciju add da bismo
vrijednosti registra \$t0 sa
smjestiti rezultat odu-
zima ④ i ⑤ učitavamo
testirati prekoračenje
ar \$t3 smještamo rezultat
stena u registar \$t8, te is-
da bismo vidjeli razliku
na prethodno.

strukcija lui koja učitava
ikog registra. Ova instruk-
vrijednosti veće od 32767
u unutar instrukcije. Naj-
kcijom kao u liniji ⑥ gdje
gistar.

nalazi ispod.

done.

zero, 100

Aritmetičko-logičke instrukcije

```
(gdb) ni ②
>0x4001f8 in main ()
l: x/i $pc
=> 0x4001f8 <main+4>:      add      t1,t0,t0
(gdb) p $t0 ③
$1 = 100
(gdb) ni ④
>0x4001fc in main ()
l: x/i $pc
=> 0x4001fc <main+8>:      sub      t2,t1,t0
(gdb) p $t1
$2 = 200
(gdb) ni ⑤
>0x400200 in main ()
l: x/i $pc
=> 0x400200 <main+12>:     lui      t8,0x7fff
(gdb) p $t2
$3 = 100
(gdb) ni ⑥
>0x400204 in main ()
l: x/i $pc
=> 0x400204 <main+16>:     ori      t8,t8,0xffff
(gdb) p $t8 ⑦
$4 = 2147418112
(gdb) p/x $t8 ⑧
$5 = 0x7ffff0000
(gdb) ni
>0x400208 in main ()
l: x/i $pc
=> 0x400208 <main+20>:     addu   t3,t8,t8
(gdb) p/x $t8 ⑨
$6 = 0x7fffffff
(gdb) ni ⑩
>0x40020c in main ()
l: x/i $pc
=> 0x40020c <main+24>:     addu   t4,t8,t8
(gdb) p $t3 ⑪
$7 = -2
(gdb) p/x $t3 ⑫
$8 = 0xffffffff
(gdb) ni ⑬
Remote connection closed. ⑯
(gdb) q
[1]- Aborted
[2]+ Exit j
(cored dumped) qemu-mipsel -g 1234 $1 &> /dev/null
qemu-mipsel -g 1234 $1 &> /dev/null
```

Nakon standardne inicijalizacije debaging sesije, stavljanja tačke prekida na main, te postavljanja prikaza sljedeće instrukcije u liniji ①, krećemo sa izvršavanjem instrukciju po instrukciju. Kao što vi-
dimo u liniji ②, pozivamo komandu ni koja predstavlja skraćenicu za komandu nexti, tj. next instruction. Za razliku od komande n,

Aritmetičke instrukcije

49

tj. komande `next` koja izvršava jednu po jednu liniju koda, komanda `ni` izvršava jednu instrukciju.

Nakon što smo u liniji ❶ postavili prikaz svake sljedeće instrukcije, ispod nam se ispisuje sljedeća instrukcija, tj. prva iz našeg primjera. Nakon poziva instrukcije `ni` u liniji ❷ debager izvršava tu instrukciju i prikazuje sljedeću instrukciju. U liniji ❸ printamo vrijednost registra `$t0` da se uvjerimo da je prethodna instrukcija izvršena i vidimo da je u ovom registru očekivana vrijednost, tj. vrijednost operacije sabiranja nultog registra sa konstantom. Postupak ponavljamo dalje. U liniji ❹ izvršavamo sljedeću instrukciju i ispisujemo vrijednost registra `$t1`, koji nakon toga treba da sadrži duplu vrijednost registra `$t0`. Pozivom sljedeće `ni` instrukcije izvršavamo instrukciju u liniji ❺ te zatim printamo vrijednost registra `$t2` koji sadrži rezultat operacije oduzimanja prethodna dva registra.

U liniji ❻ ćemo pozvati izvršavanje instrukcije `lui` a rezultat ove operacije vidimo u linijama ❼ i ➋, tj. učitali smo gornja 2 bajta u register `$t8`. U narednoj instrukciji učitavamo donja 2 bajta u isti register te njegovu cjelokupnu vrijednost vidimo u liniji ⩿. Na kraju su nam ostale dvije različite vrste sabiranja, bez provjere prekoračenja i sa generisanjem iznimke. Prvo ćemo u liniji ⩾ pozvati instrukciju `addu` gdje vidimo da je program nastavio i da smo dobili prekoračenje. Vrijednost registra `$t3` printamo u linijama ⩾ i ⩼ u decimalnom i heksadecimalnom formatu. U liniji ⩽ pozivamo instrukciju `add` koja generiše iznimku i naš program je prekinut. Poruku da je naš program prekinuo izvršavanje vidimo u liniji ⩾.

U narednom primjeru ćemo predstaviti korištenje osnovnih instrukcija za množenje i dijeljenje.

```
.section .text
.set noreorder
.global main
main:
    lui $t0, 0x9 ❶
    ori $t0, $t0, 0xc405 ❷
    addi $t1, $t0, 32000 ❸
```

• **sljednju koda, komanda**

• sljedeće instrukcije prva iz našeg primjera. Debager izvršava tu instrukciju ③ printamo vrijednost na instrukcija izvršena jednost, tj. vrijednost sa kom. Postupak ponavljamo i ispisujemo tako da sadrži duplu vrijednost. U sljedećoj izvršavamo instrukciju ④ množenje registra \$t2 koji sadrži dva registra.

U sljedećoj instrukciji lui a rezultat ove operacije smo gornja 2 bajta u redoslijedu donja 2 bajta u isti redoslijed u liniji ⑨. Na kraju bez provjere prekoračenja liniji ⑩ pozvati instrukciju i da smo dobili preko linijama ⑪ i ⑫ u decimatu. Liniji ⑬ pozivamo instrukciju je prekinut. Poruku da je u liniji ⑭.

koristenje osnovnih ins-

```
mult $t0, $t1 ④
mfhi $t2 ⑤
mflo $t3 ⑥

addi $t4, $0, 20000
div $t1, $t4 ⑦
mfhi $t5 ⑧
mflo $t6 ⑨

addi $v0, $0, 0
jr $ra
nop
```

U linijama ①, ② i ③ učitavamo (proizvoljne) vrijednosti u registre \$t0 i \$t1. U liniji ④ izvršili smo množenje te dvije vrijednosti, a u linijama ⑤ i ⑥ rezultat ove operacije prebacujemo iz specijalnih registara HI i LO u privremene registre opšte namjene.

U liniji ⑦ vršimo operaciju dijeljenja vrijednosti iz registara \$t1 i \$t4, te na isti način kao u slučaju množenja pomjeramo vrijednosti iz specijalnih registara u registre opšte namjene.

Tok prethodnog programa ćemo prikazati u sljedećoj debager sesiji:

```
[jelkomp$ run_mips mnozenje_dijeljenje
l 3908
Reading symbols from mnozenje_dijeljenje...done.
(gdb) target remote :1234
Remote debugging using :1234
0x00400180 in _start ()
(gdb) b main
Breakpoint 1 at 0x4001f4
(gdb) c
Continuing.

z breakpoint 1, 0x004001f4 in main ()
(gdb) disp/i $pc
t: >/i $pc
-> 0x4001f4 <main>:      lui          t0, 0x9
(gdb) ni
0x004001f8 in main ()
t: >/i $pc
-> 0x4001f8 <main+4>:      ori          r5, t0, 0xc405
(gdb) ni
0x004001fc in main ()
t: >/i $pc
-> 0x4001fc <main+8>:      addi         t1, zero, 32000
```

```

(gdb) p/x $t0 ❶
$1 = 0x9c405
(gdb) p/d $t0 ❶
$2 = 640005
(gdb) ni
0x00400200 in main ()
1: x/i $pc
=> 0x400200 <main+12>:      mult      t0,t1
(gdb) p/d $t1 ❷
$3 = 32000
(gdb) ni ❸
0x00400204 in main ()
1: x/i $pc
=> 0x400204 <main+16>:      mfhi     t2
(gdb) ni ❹
0x00400208 in main ()
1: x/i $pc
=> 0x400208 <main+20>:      mflo     t3
(gdb) ni ❺
0x0040020c in main ()
1: x/i $pc
=> 0x40020c <main+24>:      addi    t4,zero,20000
(gdb) p/x $t2 ❻
$4 = 0x4
(gdb) p/x $t3 ❾
$5 = 0xc4b67100
(gdb) ni ❿
0x00400210 in main ()
1: x/i $pc
=> 0x400210 <main+28>:      bnez    t4,0x40021c <main+40>
                                div     zero,t1,t4
(gdb) p/d $t4 ❽
$6 = 20000
(gdb) x/10i $pc ❾
=> 0x400210 <main+28>:
    0x400214 <main+32>:
    0x400218 <main+36>:
    0x40021c <main+40>:
    0x400220 <main+44>:
    0x400224 <main+48>:
    0x400228 <main+52>:
    0x40022c <main+56>:
    0x400230 <main+60>:
    0x400234 <main+64>:
(gdb) ni
0x0040021c in main ()
1: x/i $pc
=> 0x40021c <main+40>:      li      at,-1
(gdb) ni
0x00400220 in main ()
1: x/i $pc
=> 0x400220 <main+44>:      bne    t4,at,0x400234 <main+64>
                                lui    at,0x8000
(gdb) ni
0x00400224 in main ()
1: x/i $pc
=> 0x400224 <main+48>:      lui    at,0x8000
(gdb) ni
0x00400234 in main ()
1: x/i $pc

```

bnez	t4,0x40021c <main+40>
div	zero,t1,t4
break	bx7
li	at,-1
bne	t4,at,0x400234 <main+64>
lui	at,0x8000
bne	t1,at,0x400234 <main+64>
nop	
break	bx6
mflo	t1

- Ispisujemo u heksadecimalnom pa u decimalnom formatu vrijednost učitanu u registar \$t0
 - Provjeravamo vrijednost učitanu u registar \$t1
 - Izvršavamo operaciju množenja vrijednosti iz registara \$t0 i \$t1
 - Pomjeramo gornjih 32 bita rezultata množenja u registar \$t2
 - Pomjeramo donjih 32 bita rezultata množenja u registar \$t3
 - Provjeravamo vrijednosti rezultata množenja
 - Učitavamo i provjeravamo vrijednost registra \$t4

- Primjećujemo da prikaz sljedeće instrukcije ne prikazuje našu instrukciju dijeljenja već dvije instrukcije zbog instrukcije grananja. Kako nismo u svom kodu imali instrukciju grananja, narednom komandom prikazujemo sljedećih 10 instrukcija za
- ❸ analizu generisanih instrukcija. Možemo primijetiti da je naša instrukcija dijeljenja zamijenjena sa 10 instrukcija u kojima se vrši provjera vrijednosti djelioca, iz razloga što kompjuter sam dodaje provjere za dijeljenje ako mu se drugačije ne naglasi. Zatim smo sa nekoliko **ni** komandi ispratili tok programa.
 - ❹ Pomjeramo vrijednost ostatka pri dijeljenju iz specijalnog registra u registar opšte namjene i provjeravamo rezultat
 - ❺ Pomjeramo vrijednost rezultata pri dijeljenju u registar opšte namjene i provjeravamo rezultat.

Napomena

U liniji ❸ prethodne gdb sesije, gdje smo prikazali instrukcije koje su zamijenile našu instrukciju dijeljenja, pojavljuju se nove instrukcije, bnez i bne, koje predstavljaju instrukcije grananja. O njima ćemo detaljnije u narednim poglavljima.

Logičke instrukcije

Kao i u slučaju aritmetičkih instrukcija, tako i logičke instrukcije obuhvataju sve osnovne logičke operacije.

Tabela 4-0. Osnovne logičke instrukcije mips procesora

Sintaksa	Opis
and d,s, \$t	logička operacija I
andi d,s, C	logička operacija I sa konstantom
or d,s, \$t	logička operacija II

očnije ne prikazuje našu
vježbu zbog instrukcije grana-
nja. U instrukciju grananja,
te u sljedećih 10 instrukcija za-
nimemo primijetiti da je naša
vježba instrukcija u kojima se
zadatka pogodno što kompjuter sam
uvede drugačije ne naglasi.
izmislili tok programa.

u vježbi iz specijalnog re-
zultata učitavamo rezultat

u vježbi u registar opšte

očnije su u vježbi učinili instrukcije koje su
uvedene u sebe nove instrukcije,
u vježbi. O njima ćemo de-

u vježbi i logičke instrukcije

Sintaksa	Opis
<code>lui d.s, C</code>	logička operacija ILI sa konstantom
<code>nor d.s, \$t</code>	logička operacija negirano ILI
<code>nor d.s, \$t</code>	logička operacija ekskluzivno ILI
<code>nor d.s, C</code>	logička operacija ekskluzivno ILI sa konstantom

Isto kao i za aritmetičke instrukcije, i za logičke instrukcije važi
da sufiks 'i' označava instrukcije čiji je drugi operand konstanta
umjesto registra. Sve gore nabrojane logičke instrukcije rade na ni-
vou bita. Ove instrukcije su korisne za operacije maskiranja i seto-
vanja što ćemo vidjeti u narednom primjeru.

```
section .text
    .cfq noreorder
    .globl main
    .text
        lui $t0, 0xfffff ①
        ori $t0, $t0, 0xffff00 ②
        andi $t2, $t0, 0xabcd ③
        ori $t3, $t2, 1024 ④
        xor $t4, $t3, -1 ⑤
        addi $t5, $0, 1024 ⑥
        xor $t5, $t5, -1 ⑦
        and $t6, $t0, $t5 ⑧
        addi $v0, $0, 0
        jr $ra
        nop
```

U linijama ① i ② ćemo u registar \$t0 učitati vrijednost 0xffffffff00
koju ćemo koristiti kao masku u liniji ③. Operacija logičkog I vrši
maskiranje bita, tj. svi biti gdje maska ima vrijednost 1 će zadržati
svoju vrijednost, dok će biti gdje maska ima vrijednost 0 dobiti vri-
jednost 0.

U liniji ④ vršimo setovanje desetog bita u registru \$t2 koristeći operaciju ILI. Operacijom ILI se vrši setovanje onog bita rezultata koji odgovara setovanom bitu konstante u instrukciji. Ostali biti ostaju nepromijenjeni. Instrukcija xor u liniji ⑤ radi zamjenu vrijednosti bita, tj. biti koji su imali vrijednost 1 postaju 0, i obrnuto.

U zadnje 3 označene linije, ⑥, ⑦ i ⑧, vršimo resetovanje određenog bita. U odnosu na liniju ④, ovo je obrnut proces. U ovom slučaju na mjesto desetog bita postavljamo 0 bez obzira na prethodnu vrijednost. Prvo u registar \$t5 učitamo konstantu koja ima vrijednost 1 samo na desetom bitu. Zatim tu vrijednost invertujemo, tako da dobijemo konstantu koja ima samo 0 na desetom bitu, te na kraju izvršimo operaciju logičkog I koja, kako smo već vidjeli, radi maskiranje. Na ovaj način će svi biti u registru \$t0, osim desetog bita kojeg smo željeli nulirati, zadržati svoju vrijednost.

Rezultati izvršavanja prethodno navedenih instrukcija su prikazani u sljedećoj debager sesiji:

```
maj@kompS:~/ua_mips/logicke$ ./logicke
(gdb) target remote :1234
Remote debugging using :1234
0x00400162 in _start ()
(gdb) b main
Breakpoint 1 at 0x4001f4
(gdb) c
Continuing.

Breakpoint 1, 0x004001f4 in main ()
(gdb) disp/i $pc
1: x/i $pc
=> 0x4001f4 <main>:      lui      t0,0xffff
(gdb) ni
0x004001f8 in main ()
1: x/i $pc
=> 0x4001f8 <main+4>:      ori     t0,t0,0xffff00
(gdb) p/x $t0
$t0 = 0xfffff0000
(gdb) ni
0x004001fc in main ()
1: x/i $pc
=> 0x4001fc <main+8>:      andi    t2,t0,0xabcd
(gdb) p/x $t0
```

registru \$t2 koristeći se na onog bita rezultata prethodne instrukcije. Ostali biti osim t2 radi zamjenu vrijednosti postaju 0, i obrnuto.

U sljedećem resetovanju određenih bita u registru \$t0 bez obzira na prethodnu konstantu koja ima vrijednost invertujemo, tako da na desetom bitu, te na kraju kako smo već vidjeli, radi mjerljivog registru \$t0, osim desetog bita nisu vrijednost.

Navedenih instrukcija su prikazane u sledećoj tabeli:

t6,0xffff

t6,t6,0xffff00

t2,t6,0xabcd

```
gdb> s $t2=0xffffffff00
(gdb) ni ②
$3=0x455290 in main ()
l: + 1 $spc
-> 0x455290 <main+12>:    orl      $3,12,0x400
(gdb) p/x $t2 ②
$2 = 0x3000
(gdb) ni ③
$3=0x455294 in main ()
l: + 1 $spc
-> 0x455294 <main+16>:    lsl      $t,-1 ⑤
(gdb) p/x $t3 ③
$2 = 0xE000
(gdb) p/t $t2 ④
$5 = 0310101100000000
(gdb) p/t 0x400 ④
$6 = 10000000000
(gdb) p/t $t3 ④
$7 = 1310111100000000
(gdb) ni ⑤
$3=0x455298 in main ()
l: + 1 $spc
-> 0x455298 <main+20>:    xorl      $4,t3,$t
(gdb) ni ⑥
$3=0x45529c in main ()
l: + 1 $spc
-> 0x45529c <main+24>:    addl      $5,zero,1024
(gdb) p/t $t4 ⑥
$8 = 111111111111111010000011111111
(gdb) ni
$3=0x4552a0 in main ()
l: + 1 $spc
-> 0x4552a0 <main+28>:    lsl      $t,-2
(gdb) p/t $t5
$9 = 10000000000
(gdb) ni
$3=0x4552a4 in main ()
l: + 1 $spc
-> 0x4552a4 <main+32>:    xorl      $5,$t5,$t
(gdb) ni ⑦
$3=0x4552a8 in main ()
l: + 1 $spc
-> 0x4552a8 <main+36>:    andl      $t6,t8,$t5
(gdb) p/t $t5 ⑧
$10 = 111111111111111111110111111111
(gdb) ni ⑨
$3=0x4552ac in main ()
l: + 1 $spc
-> 0x4552ac <main+40>:    addl      $6,zero,0
(gdb) p/t $t6 ⑨
$11 = 11111111111111111111011000000000
(gdb) ni
$3=0x4552d0 in main ()
l: + 1 $spc
-> 0x4552d0 <main+44>:    jf      $t0
-> 0x4552d4 <main+48>:    pop
```

```

(gdb) ni
0x00400464 in __libc_start_main (main=0x4000f4 <main>, argc=1, argv=0x7f
    at src/env/_libc_start_main.c:73
73          src/env/_libc_start_main.c: No such file or directory.
1: x/i $oc
=> 0x400464 <__libc_start_main+564>:      jal      0x400474 <exit>
    0x400468 <__libc_start_main+568>:      move    $0,v0
(gdb) ni
[Inferior 1 (Remote target) exited normally]
(gdb) q
[1]+ Done                  gnu-mipsel -g 1234 $1 &> /dev/null

```

Provjeravamo vrijednost učitane maske u registar **\$t0** nakon

- ➊ što smo pojedinačno učitali gornjih i donjih 16 bita koristeći kombinaciju instrukcija **lui** i **ori**.

Izvršavamo instrukciju logičkog I, te prikazujemo i provjera-

- ➋ vamo rezultat. Koristeći masku, zadržali smo vrijednost gornjih 24 bita u zadatoj konstanti **0xabcd**, te kao rezultat dobili vrijednost **0xab00**.

Izvršili smo instrukciju logičkog ILI, te prikazali dobiveni re-

- ➌ sultat. U ovom slučaju izvršili smo operaciju ILI između vrijednosti **0xab00** i **0x400**, te dobili vrijednost **0xaf00**.

Prikazujemo vrijednosti prethodnih operanada i rezultata u

- ➍ binarnom formatu da bismo pokazali kako se izvršava instrukcija **ori**.

Sljedeća instrukcija iz našeg koda je predstavljena kao dvije

- ➎ instrukcije jer smo pokušali da učitamo vrijednost koja je veća od maksimalne vrijednosti koja može stati u instrukciju. Kako vidimo, korištena je instrukcija **li** koja je zapravo pseudo-instrukcija i predstavlja kombinaciju već pomenutih instrukcija **lui** i **ori**.

Instrukcija **xor** je izvršila invertovanje svih bita iz zadanog operanda jer smo izvršili tu operaciju nad konstantom **-1**, koja

- ➏ predstavlja vrijednost **0xffffffff**. Ako uporedimo vrijednost registra **\$t3** iz linije ➍, te vrijednosti registra **\$t4** kojeg ispisujemo u ovoj liniji, vidimo da su biti invertovani.

Nakon što smo u registar \$t5 učitali vrijednost koja ima jedinicu na desetom bitu, u ovoj liniji vršimo invertovanje tih bita na isti način kao u prethodnom primjeru kako bismo dobili vrijednost 0 samo na desetom bitu.

Prikazujemo novu invertovanu vrijednost koja ima nulu samo na mjestu desetog bita. Ovu vrijednost ćemo koristiti kao masku.

Izvršavamo operaciju logičkog I kojom ćemo resetovati deseti bit iz vrijednosti registra \$t0, te na kraju prikazujemo rezultat. Početna maska, koja je bila smještena u ovom registru, sada na poziciji desetog bita ima vrijednost 0.

Šift instrukcije

Koristeći šift instrukcije vršimo pomjeranje vrijednosti unutar registra za određeni broj bita ulijevo ili udesno. Osnovne šift instrukcije su date u tabeli ispod.

Tabela 4-0. Osnovne šift instrukcije mips procesora

Sintaksa	Opis
sll d,s,C	pomjeranje bita ulijevo za konstantan broj bita
srl d,s,C	pomjeranje bita udesno za konstantan broj bita
sra d,s,C	pomjeranje bita udesno sa predznakom za konstantan broj bita
sllv d,s,\$t	pomjeranje bita ulijevo
srlv d,s,\$t	pomjeranje bita udesno
sraw d,s,\$t	pomjeranje bita udesno sa predznakom

Instrukcija **sll** pomjera vrijednost registra **\$s** za **C** bita na način da se sa desne strane vrijednost puni nulama. Ovom operacijom se

zapravo vrši množenje te vrijednosti sa 2^C . Instrukcija `srl` radi slično instrukciji `sll` s razlikom što se vrijednost pomjera udesno, tj. dijeli se sa 2^C . Ove dvije instrukcije rade logičko pomjeranje bita, dok instrukcija `sra` radi aritmetičko pomjeranje bita. Razlika između aritmetičkog i logičkog pomjeranja bita udesno je u tome što se kod aritmetičkog pomjeranja udesno slobodna mjesta sa lijeve strane popunjavaju bitom predznaka binarnog broja, odnosno vrijednosti registra, nad kojim se vrši operacija, dok se u slučaju logičkog pomjeranja bita slobodna mjesta slijeva uvijek popunjavaju nulom.

Kao što vidimo u drugom dijelu prethodne tabele, svaka od navedenih instrukcija ima svoju 'v' varijantu, tj. u registar `$t` se smješta broj mjesta za pomjeranje ukoliko isti nije prethodno poznat. Na ovaj način se broj mjesta za pomjeranje čita iz registra, a ne iz konstante, dok je funkcionalnost potpuno ista.

Za praktično objašnjenje raznih varijanti šest instrukcija koristit ćemo sljedeći primjer:

```
.section .text
.set noreorder
.global main
main:
    addi $t0, $0, 128
    sll $t1, $t0, 3 ①
    srl $t2, $t0, 4 ②
    sra $t3, $t0, 4 ③

    addi $t0, $0, -128
    sll $t4, $t0, 3 ④
    srl $t5, $t0, 4 ⑤
    sra $t6, $t0, 4 ⑥

    addi $t3, $0, 5
    sllv $t4, $t0, $t3 ⑦
    srav $t5, $t0, $t3 ⑧
    srav $t6, $t0, $t3 ⑨

    addi $v0, $0, 0
    jr $ra
    nop
```

Logičko pomjeranje bita, razlike između udesno je u tome što se mesta sa lijeve strane broja, odnosno vrijednosti dok se u slučaju logičkog unjek popunjavaju nulom.

prethodne tabele, svaka od narednih linija, tj. u registar \$t se smještiti nije prethodno poznat. Nakon toga se čita iz registra, a ne iz konstanta.

Ujanti šift instrukcija koristit

U linijama ❶, ❷ i ❸ ćemo prikazati kako funkcionišu instrukcije ~~z~~eranja na pozitivnoj vrijednosti, dok ćemo u linijama ❹, ❺ i ❻ ~~z~~lati iste te instrukcije kada se izvrše nad operandom negativ-~~g~~ predznaka. Na kraju ćemo prikazati primjer upotrebe šift ins-~~trukcija~~ koje ne koriste vrijednost pomjeraja iz konstante već iz re-~~zultata~~ kojeg smo prethodno napunili.

Gdb sesija prethodnog primjera je prikazana ispod:

```

0x00400204 in main ()
1: x/i $pc
=> 0x400204 <main+16>:      addi      t0,zero,-128
(gdb) p $t3
$3 = 8
(gdb) ni
0x00400208 in main ()
1: x/i $pc
=> 0x400208 <main+20>:      sll      t1,t0,0x3
(gdb) p $t0
$9 = -128
(gdb) ni
0x0040020c in main ()
1: x/i $pc
=> 0x40020c <main+24>:      srl      t2,t0,0x4
(gdb) p $t1
$10 = -1024
(gdb) ni
0x00400210 in main ()
1: x/i $pc
=> 0x400210 <main+28>:      sra      t3,t0,0x4
(gdb) p $t2
$11 = 268435448
(gdb) p/t $t1
$12 = 111111111111111111110000000000
(gdb) p/t $t2
$13 = 111111111111111111111111111111000
(gdb) ni
0x00400214 in main ()
1: x/i $pc
=> 0x400214 <main+32>:      addi      t3,zero,5
(gdb) p/t $t3
$14 = 111111111111111111111111111111000
(gdb) p $t3
$15 = -8
(gdb) ni
0x00400218 in main ()
1: x/i $pc
=> 0x400218 <main+36>:      sllv     t4,t0,t3
(gdb) p $t3
$16 = 5
(gdb) ni
0x0040021c in main ()
1: x/i $pc
=> 0x40021c <main+40>:      sriv     t3,t0,t3
(gdb) p $t4
$17 = -4096
(gdb) ni
0x00400220 in main ()
1: x/i $pc
=> 0x400220 <main+44>:      srlv     t6,t0,t3
(gdb) p $t5
$18 = 134217723
(gdb) ni
0x00400224 in main ()
1: x/i $pc

```

```

... 0x400224 <main+48>:    addi      v0,zero,0
(gdb) p $t6
$17 = -4
(gdb) ni
1:00000228 in main ()
1: >i $pc
=> 0x400228 <main+52>:    jr      ra
    0x40022c <main+56>;    nop
(gdb) ni
2:0000046c in __libc_start_main (main=0x4001f4 <main>, argc=1, argv=0x7ffff661)
    st src/env/_libc_start_main.c:73
2: >src/env/_libc_start_main.c: No such file or directory.
2: >i $pc
=: 0x40046c <__libc_start_main+564>;    jal      0x40047c <exit>
    0x400470 <__libc_start_main+568>;    move   $0,v0
(gdb) ni
[Inferior 1 (Remote target) exited normally]
(gdb) q
[1]- Done
qemu-mipsel -p 1234 $1 &> /dev/null

```

Prikazujemo vrijednost učitanu u registar $$t0$ u različitim

- ❶ formatima, kako bi lakše shvatili različite vrste operacija šiftanja.

Izvršavamo operaciju logičkog pomjeranja uljevo za 3 mesta te prikazujemo dobiveni rezultat u binarnom i decimalnom

- ❷ formatu. Kao što vidimo, biti su pomjereni uljevo a prazna mesta su popunjena nulama, dok u decimalnom formatu primjećujemo da je vrijednost pomnožena sa 8, tj. 2^3 .

Na isti način izvršavamo i operaciju logičkog pomjeranja

- ❸ udesno, ali u ovom slučaju za 4 mesta. Kada prikažemo rezultat vidimo da je početna vrijednost podijeljena sa 16, tj. 2^4 .

Kada izvršimo instrukciju aritmetičkog pomjeranja udesno

- ❹ vidimo da je rezultat isti kao i za logičko pomjeranje jer je u ovom slučaju vrijednost koju pomjeramo pozitivna, pa je bit predznaka 0.

Nakon što smo učitali negativnu vrijednost u registar nad ko-

- ❺ jim izvršavamo operacije šiftanja, prvo ćemo izvršiti instrukciju za logičko pomjeranje bita uljevo. Kao što vidimo, rezul-

tat je očekivan, tj. dobivamo broj -1024 koji je rezultat pomje- ranja broja -128 ulijevo za 3 pozicije.

Kada izvršimo operaciju logičkog pomjeranja udesno na ne-

- ⑥ negativnom broju, prazna mjesta slijeva se popunjavaju nulama, tako da broj koji dobivamo u ovom slučaju nije pravi rezultat dijeljenja.

U slučaju aritmetičkog pomjeranja bita udesno, bit predznaka

- ⑦ popunjava prazna mjesta, tako da dobivamo kao rezultat -8, tj. dijeljenje broja -128 sa 16, tj. 2^4 .

Na primjeru instrukcija koje za treći argument šift instrukcije koriste registar za zadavanje broja pomjeranja bita vidimo da

- ⑧ je ponašanje ovih instrukcija isto kao u prethodnim slučajevima. Ovdje smo pomjeranje vršili za 5 bita, jer se u registru \$t3 nalazi vrijednost 5.

Pristup memoriji

MIPS procesor posjeduje specijalizirane instrukcije za komunikaciju sa memorijom, koje spadaju u jednu od dvije kategorije:

load

Za transfer određenog broja bajta iz memorije u neki registar unutar registar fajla.

store

Za transfer određenog broja bajta iz nekog registra unutar registar fajla na neku adresu u memoriji.

Sem instrukcija iz ove dvije kategorije, niti jedna druga instrukcija nema mogućnost pristupa memoriji, bilo za čitanje ili pisanje podataka.

Load i store instrukcije podržavaju transfere između memorije i registar fajla sa sljedećom količinom podataka:

1. bajt — Transfer od jednog bajta može se događati sa bilo koje adrese u memoriji.
2. pola riječi — Istovremeni transfer od dva bajta može se događati samo na adresama koje su djeljive sa dva.
3. riječ — Istovremeni transfer od četiri bajta može se događati samo na adresama koje su djeljive sa četiri.

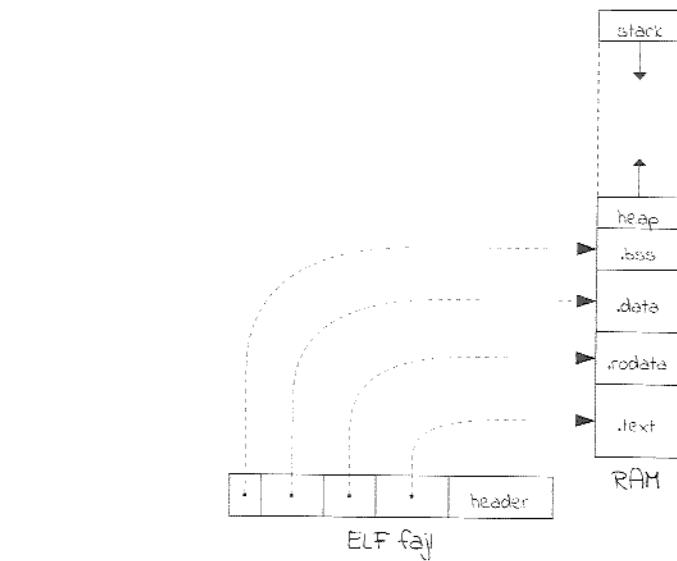
Za transfer od riječi ili pola riječi kažemo da mora biti poravnat na određene adrese. Ukoliko neka aplikacija pokuša inicirati transfer podataka sa neporavnate adrese u memoriji tada MIPS procesor

generira iznimku na hardverskom nivou, koju na adekvatan način tretira operativni sistem.

Prije nego što prijeđemo na formulaciju load i store instrukcija analizirat ćemo sliku programa u memoriji nakon njegovog pokretanja.

Memorijska slika programa

Izvršna datoteka nakon obavljenog kompajliranja, asembleriranja i uvezivanja u elf format obično sadrži sekcije: `text`, `data`, `bss` i `rodata`, čija je funkcija objašnjena u poglavlju 3. Kada je u ovoj formi, program se može pokretanuti u terminalu unošenjem imena elf datoteke, kao što je demonstrirano u prethodnim primjerima. Operativni sistemi za ovu svrhu uključuju posebnu komponentu, obično označenu kao *loader*, koji ima zadatak da, prije nego što počne izvršenje programa, u memoriji formira sve potrebne segmente programa.



Slika 5-1. Formiranje memorijske slike

na adekvatan način

red i store instrukcija
njegovog pokre-

ranja, asembliranja i
text, data, bss i ro-
da. Kada je u ovoj formi,
šenjem imena elf da-
primjerima. Opera-
komponentu, obično
nešto počne izvr-
ne segmente progra-



Proces učitavanja programa u memoriju prikazan je na slici 5.1. Početku elf datoteke nalazi se zaglavje (header), koje opisuje sadržaj koji slijedi u fajlu. Putem zaglavlja možemo doći do informacija o počecima i veličinama svih sekcija u elf datoteci koje su nastale djelovanjem kompjuterskog lanca. Operativni sistem direktno iz fajla u memoriju kopira kompletan sadržaj sekcija text, rodata i data, dok za bss sekciju, obzirom da ista sadrži neinicijalizirane globalne varijable, OS samo odvaja potrebnii prostor kojeg popunjava slobodna mesta.

Kopiranjem sekcija iz elf fajla formiran je samo dio memorijiske strukture programa. U memoriju je do tada učitan programski kod i inicijaliziran segment memorije za konstante, kao i za sve globalne varijable. Program za normalno funkcionisanje zahtijeva još i prostor za lokalne varijable koje se pojavljuju u memoriji dok se izvršavaju pojedinačni blokovi programa, te prostor za dinamički alocirane varijable nastale pozivanjem C funkcije malloc ili putem C++ operatora new. Operativni sistem za dinamičke varijable odvaja prostor, označen kao heap, i to odmah iznad segmenta memorije koji je nastao kopiranjem sadržaja iz elf datoteke. Prostor za lokalne varijable, označen kao stack, operativni sistem postavlja na vrh adresnog prostora ostavljenog za aplikaciju. Oba segmenta mogu da rastu ili da se smanjuju po potrebi, i to tako da stack raste prema nižim memorijskim adresama, a heap prema višim. Stack ćemo detaljno tretirati u poglavljju 7, u kojem ćemo obraditi asemblerske funkcije. Daljnji fokus u ovom poglavljju usmjerit ćemo na asemblerske elemente kojima se formiraju sadržaji data, rodata i bss sekcija unutar elf fajla.

Inicijalizacija globalnih varijabli

Radi kreiranja globalnih varijabli u asembleru potrebno je koristiti dva tipa asembler elemenata, i to:

oznake

Putem kojih se imenuju globalne varijable i označavaju adrese u memoriji odakle počinju njihove vrijednosti.

direktive

Putem kojih se alocira, i po potrebi popunjava određenim vrijednostima, prostor u trenutnoj sekciji, obično nakon asembler označke.

Već ranije je naglašeno da direktivom `global` ili `globl` određenu oznaku proglašavamo vidljivom u drugim asembler fajlovima. Sa stanovišta C koda, `global` direktiva je potrebna za kreiranje svih funkcija ili globalnih varijabli. Za one globalne varijable ili funkcije koje su prilikom definiranja modificirane sa ključnom riječi `static`, upotreba `global` direktive u asemblerском kodu nije potrebna jer su iste vidljive samo unutar fajla u kojem su definirane.

Sljedeća tabela daje pregled često korištenih asembler direktiva za alokaciju prostora u trenutnoj sekciji, te njeno popunjavanje binarnim vrijednostima određenog formata:

Tabela 5-1. Asembler data direktive

Direktiva	Opis	Primjer
<code>.byte</code>	8-bitna vrijednost	<code>.byte 10, 0xab, 'B'</code>
<code>.half</code>	16-bitna vrijednost	<code>.hword 5, 0xffa0</code>
<code>.short</code>	isto kao <code>.half</code>	<code>.short 100</code>
<code>.word</code>	32-bitna vrijednost	<code>.word 25438, 0x11223</code>
<code>.int</code>	za MIPS isto kao <code>.word</code>	<code>.int 18</code>
<code>.long</code>	za MIPS isto kao <code>.int</code>	<code>.long 25483</code>

Direktiva	Opis	Primjer
.ascii	ascii kodiran niz karaktera	.ascii "Ovo je tekst"
.asciiz	null terminiran niz karaktera	.asciiz "Niz"
.string	isto kao .asciiz	.string "abcdefg"
.float	32-bitni broj IEEE FP format	.float 3.14
.single	isto kao .float	.single 18.45
.double	64-bitni broj IEEE FP format	.double 2.2
.zero	popunjava n bajta nulama	.zero 32
.fill	popunjava n puta, m bajta, vrijednost p	.fill 9, 1, 0xfa
.align	poravnava sljedeću direktivu na sljedeću adresu djeđiju sa 2^n	.align 4

Gornje direktive najčešće se koriste u `data` i `rodata` sekcijama. Osnovna razlika je da se u `.data` sekciju smještaju vrijednosti globalnih varijabli, dok `rodata` sekciju sačinjavaju vrijednosti konstanti korištenih u kodu. U sljedećem asembler kodu dat je primjer upotrebe nekih od navedenih direktiva:

```
.data ①
    var1: byte 1, 2, 3 ②
    var2: ascii "Moj string"
    var3: word 0xdeadbeaf
    .align 3 ③
    var4: half 0xccbb
    .align 5
    var5: byte 5
    var6: word var2 ④
.text ⑤
    .global main
    main:
        addi $v0, $0, 0
        jr $ra
```

Sve oznake u prethodnom primjeru, sem `main`, su lokalnog karaktera. Na osnovu linija ① i ⑤ vidimo da, pri određenim definicijama sekcija, možemo izostaviti `section` direktivu.

U liniji ② vidimo da direktive primaju niz vrijednosti koje će u memoriji biti poredane sukcesivno, u skladu sa navedenim formatom. Za ovaj slučaj oznaka `var2` korelirat će sa adresom prvog elementa u nizu.

Align direktiva u liniji ③ odnosi se na poravnanje varijable `var4`, koja slijedi nakon te direktive.

Vrijednosti koje se koriste pri inicijalizaciji alociranog prostora mogu biti i asemblerски izrazi. Naprimjer, u liniji ④ je specificirano da će memorija na lokaciji asociranom sa varijablu `var6` biti popunjena adresom varijable `var2`.

Za analizu sadržaja elf fajlova kompajlerski lanac sadržava više alata, od kojih je najkorisniji `objdump`. Ovaj alat ćemo koristiti za analizu sadržaja objektnog fajla nastalog asembliranjem prethodnog primjera, kao i izvršne datoteke nastale uvezivanjem dobivenog objektnog fajla sa objektnim fajlovima iz standardne biblioteke.

```
moj@komp$ cd ~/test/
moj@komp$ mpsecc -c -o mem1.o mem1.s ①
moj@komp$ mpsecc -o prog_mem1 mem1.s ②
moj@komp$ ls
mem1.o mem1.s prog_mem1
moj@komp$ ecc-objdump -h mem1.o ③

mem1.o:      file format elf32-littles_endian

Sections:
Idx Name      Size    VMA      LMA      File off  Align
 0 .text     00000006  00000000  00000000  00000034  2**2
              CONTENTS, ALLOC, LOAD, READONLY, CODE
 1 .data     00000020  00000009  00000009  00000040  2**3
              CONTENTS, ALLOC, LOAD, READONLY, DATA
 2 .bss     00000000  00000000  00000000  00000060  2**0
              ALLOC
 3 .reginfo   00000016  00000000  00000000  00000060  2**2
              CONTENTS, READONLY, LINK_ONCE_SAME_SIZE
 4 .MIPS.abtflags 00000016  00000000  00000000  00000078  2**3
```

5 .pdr
6 .gnud
moj@komp\$
4 .text
5 .fini
6 .rodata
7 .ctors
8 .dtors
9 .data
10 .sbss
11 .bss
moj@komp\$

- Napomena**
- ① Koman posred Sadržaj C++ ili objektr
 - ② Pute taje o
 - ③ Vršin totek
 - ④ Ispis sljedu ne rij skup

lokalnog k-a
definicija

nečnosti koje će u
mavedenim forma-
cijama prvog ele-

e varijable var4,

ulaznog prostora
specificirano
6 biti popu-

sadržava više
koristiti za
prethodnog
dobivenog
fajla biblioteke.

FF 4104
34 2**2
E 2**3
068 2**8
0068 2**2
E_SIZE
000676 2**3

je 5, Pristup memoriji

```
CONTENTS, ALLOC, LOAD, READONLY, DATA, LINK_ONCE  
5 .pur 00000000 00000000 00000000 00000090 2**2  
CONTENTS, READONLY  
6 .gnu.attributes 00000010 00000000 00000000 00000090 2**0  
CONTENTS, READONLY  
moj@komp$ ecc-objdump -h prog_mem1 | grep -A 15 text ④  
4 .text 00000c94 00400180 00400180 00000000 2**2  
CONTENTS, ALLOC, LOAD, READONLY, CODE  
5 .fini 00000048 00400114 00400114 00000034 2**2  
CONTENTS, ALLOC, LOAD, READONLY, CODE  
6 .rodata 00000027 00400e5c 00400e5c 0000005c 2**2  
CONTENTS, ALLOC, LOAD, READONLY, DATA  
7 .ctors 00000008 00401000 00401000 00001000 2**2  
CONTENTS, ALLOC, LOAD, DATA  
8 .dtors 00000008 00401008 00401008 00001008 2**2  
CONTENTS, ALLOC, LOAD, DATA  
9 .data 00000020 00401010 00401010 00001010 2**3  
CONTENTS, ALLOC, LOAD, DATA  
10 .sbss 00000008 00401030 00401030 00001030 2**2  
ALLOC  
11 .bss 00000188 00401028 00401038 00001038 2**2  
ALLOC  
moj@komp$
```

Napomena

Komandna opcija kompjlera **-c** zaustavlja proces kompajliranja ne-
posredno prije uvezivanja u izvršni fajl.
Sadržaj ulaznog fajla koji se procesira ovom komandom može biti C,
C++ ili asemblerSKI kod, a kao rezultat procesiranja dobija se jedan
objektni fajl.

- ➊ Putem **-c** samo prevodimo asembler fajl u mašinac, čime nastaje objektni fajl **mem1.o**.
- ➋ Vršimo asembliranje i uvezivanje, čime dobijamo izvršnu datoteku **prog_mem1**.
- ➌ Ispisujemo sve sekciJE objektnog fajla **mem1.o**.
Ispisujemo sve sekciJE izvršnog fajla **prog_mem1.o**. Ispis prosljeđujemo programu **grep**, koji pretražuje tekst spram ključne riječi **text**, te ispisuje liniju u kojoj pronađe ključnu riječ skupa sa narednih 15 linija teksta.

Opcijom `-h` ispisujemo bitne informacije o sekcijama unutar elf fajla, i to:

1. Size — veličina sekcije u bajtima.
2. VMA i LMA — virtuelna i fizička adresa u memoriji na kojoj sekcija očekuje da bude učitana.
3. File off — Pomak u bajtima odakle počinje sekcija unutar fajla.

Na osnovu ispisa koji dobijemo kao rezultat naredbe **③**, zaključujemo da u pojedinačnim objektnim fajlovima adrese u svakoj sekciji počinju od 0. Nakon uvezivanja u izvršni fajl adrese sekcija koriguju se spram uputa u linker skripti za dati operativni sistem, što je vidljivo u ispisu nakon linije **④**. Analizom ovog ispisa možemo još zaključiti da objektni fajlovi iz standardne biblioteke u izvršni fajl dodaju i druge sekcije koje nisu prisutne u objektnom fajlu kojeg smo asemblirali u primjeru.

Informacije o pojedinačnim simblima, tj. oznakama, unutar elf fajla možemo dobiti putem `-t` komandne opcije.

```
moj@komp$ ecc-objdump -t mem1.o ①
```

```
mem1.o:      file format elf32-littlemips
```

```
SYMBOL TABLE:
```

00000000 l	d .text	00000000	.text
00000000 l	d .data	00000000	.data
00000000 l	d .bss	00000000	.bss
00000000 l	d .reginfo	00000000	.reginfo
00000000 l	d .MIPS.abiflags	00000000	.MIPS.abiflags
00000000 t	d .pdr	00000000	.pdr
00000000 t	d .gnu.attributes	00000000	.gnu.attributes
00000000 l	.data	00000000	var1
00000003 l	.data	00000000	var2
00000010 l	.data	00000000	var3
00000018 l	.data	00000000	var4
0000001a l	.data	00000000	var5
0000001c l	.data	00000000	var6
00000000 g	0 .text	00000000	main

```
moj@komp$ ecc-objdump -t prog_mem1 | grep -A 5 var1 ②
```

00401010 l	.data	00000000	var1
00401013 l	.data	00000000	var2

```
00401020 l    .data      00000000 var3  
00401028 l    .data      00000000 var4  
0040102e l    .data      00000000 var5  
0040103c l    .data      00000000 var6
```

Aplicirana na objektnom fajlu, opcija **-t** daje ispis nakon linije ❶. Svaki red ispisa predstavlja informaciju o nekom simbolu iz objektog fajla, od čega su najbitnije: adresa, tip simbola, sekција kojoj simbol pripada i oznaka simbola.

Obzirom da je prvi u data sekciji, simbol `var1` počinje od adrese 0, nakon čega slijedi simbol `var2` itd. Ispis simbola u izvršnoj datoteci `prog_mem1`, linija ❷, pokazuje da je adresa simbola `var1` iz našeg asembler fajla, nakon uvezivanja, pomjerena na adresu `0x401010`.

Pojedinačne bajte tj. sadržaj elf fajla možemo dobiti komandnom opcijom **-s**, pri čemu se možemo ograničiti samo na određenu sekciju unutar fajla primjenom komandne opcije **-j**.

```
moj@komp$ ecc-objdump -s -j .data mem1.o  
  
mem1.o:     file format elf32-littlerios  
  
Contents of section .data:  
0000 0102634d 6f6a2073 7472696e 67000000 ...Moj string... ❶  
0010 afbeadde 00000000 bbae0500 03000000 ..... ❷  
moj@komp$ ecc-objdump -s --start-address=0x401010 \  
> -j .data prog_mem1  
  
prog_mem1:     file format elf32-littlemacos  
  
Contents of section .data:  
401010 0102634d 6f6a2073 7472696e 67000000 ...Moj string... ❸  
401020 afbeadde 00000000 bbae0500 13104000 ..... ❹  
amer@x201:~/test$ cat mem1.s  
.data  
    var1: .byte 1, 7, 3  
    var2: .ascii "Moj string"  
    var3: .word 0xdeadbeaf  
    .align 3  
    var4: .half 0xcccbb  
    var5: .byte 5  
    var6: .word var2  
.text  
    global main  
main:
```

Dobiveni ispis u linijama ❶, ❷, ❸ i ❹, prikazuje u svakom redu adresu, praćenu sa 12 bajta nakon te adrese iz sekcije .data u heksadecimalnom formatu, a zatim i u tekstualnom formatu. Primjećujemo da asembler vrši poravnanja spram tipa podatka čija vrijednost se unosi u elf fajl.

Varijabla var3 u objektnom fajlu mem1.o, obzirom da je tipa .word, poravnata je na adresu 0x10, tj. prvu lokaciju u memoriji nakon zadnjeg bajta varijable var2 koja je djeljiva sa 4.

Varijabla var4 počinje od lokacije 0x18 zbog prethodne align direktive, koja zahtijeva da adresa sljedećeg simbola bude djeljiva sa 8.

Na osnovu linije ❷, vrijednost varijable var6 unutar mem1.o je 3, obzirom da je to adresa prvog bajta u nizu karaktera var2.

Nakon uvezivanja u izvršni fajl, analizom ispisa u linijama ❸ i ❹ možemo zaključiti da su redoslijed i vrijednosti varijabli iz objektnog fajla mem1.o zadržani u izvršnoj datoteci prog_mem1, uz odgovarajuće korekcije adresa. Jedina promijenjena vrijednost je sadržaj varijable var6, i to zbog činjenice da je nova adresa varijable var2 nakon uvezivanja u izvršni fajl 0x040113.

Dosad navedene direktive alociraju i popunjavaju prostor unutar izvršne datoteke. Sadržaj alociranog prostora kopira se u memoriju kada program počne sa izvršenjem.

Asembler podržava i posebne direktive comm i lcomm, koje umjesto da odvajaju prostor unutar elf fajla, samo bilježe informaciju o količini potrebnog prostora u memoriji tokom izvršenja programa. Razlika u direktivama je što comm odvaja prostor globalno, a lcomm lokalno samo za korištenje u jednom objektnom fajlu. Ove direktive

priznaje u svakom redu sekcije .data u heksadecimatu. Primjećuje se podatka čija vrijednost

obzirom da je tipa sekciju u memoriji način na 4.

prethodne align direkcie bude djeljiva sa 8.

unutar mem1.o je 3, a variable var2.

sa u linijama ③ i ④ varijabli iz objekta mem1, uz odgovarajuću vrijednost je sadržaj unosa variable var2

prostor unutar fajla se u memoriju

kom, koje umjesto informaciju o koštenja programa. globalno, a lcomm fajlu. Ove direktive

Slavje 5, Pristup memoriji

iste se obično pri definiranju globalnih neinicijaliziranih varijabli u asembler kodu.

Primjenu direktive comm prikazat ćemo na sljedećem primjeru:

```
moj@komp$ cd ~/test/
moj@komp$ cat mem2.s
.data
    fill 100000 ①
.text
    global main
main:
    addi $v0, $0, 0
    jr $ra
moj@komp$ mpsecc -o mem_prog2 mem2.s
moj@komp$ cat mem3.s
.bss
    comm buf, 100000 ②
.text
    global main
main:
    addi $v0, $0, 0
    jr $ra
moj@komp$ mpsecc -o mem_prog3 mem3.s
moj@komp$ ls -lh mem_prog2 mem_prog3 ③
-rwxrwxr-x 1 amer amer 121K Jul 18 15:28 mem_prog2
-rwxrwxr-x 1 amer amer 24K Jul 18 16:20 mem_prog3
```

- ① Program mem2.s u data sekciji, putem direktive fill, za varijablu buf odvaja 100KB prostora.
- ② Program mem3.s u bss sekciji, putem direktive comm, za varijablu buf odvaja 100KB prostora.
- ③ To je posljedica činjenice da direktiva comm ne odvaja prostor, već samo bilježi informaciju o potrebnom prostoru.

MIPS load i store instrukcije

Za pristup memoriji MIPS procesor koristi instrukcije u sljedećem formatu:

instrukcija a, c(b)

Pri tome a i b mogu biti bilo koji registri iz registar fajla, a c može biti bilo koji 16-bitni broj sa predznakom. Adresa u memoriji sa kojom komuniciramo putem instrukcija u ovom formatu računa se sabiranjem vrijednosti registra b sa konstantom c.

Za slučaj da je instrukcija tipa load, sadržaj memorije pročitan sa dobivene adrese snima se u registar a. Ako je instrukcija tipa store, vrijednost registra a snima se u memoriju na izračunatu lokaciju na osnovu registra b i konstante c.

Slijedi lista podržanih load i store instrukcija spram količine informacija koje se razmjenjuju između registar fajla i memorije:

1. bajt — load instrukcija lb, store instrukcija sb,
2. half — load instrukcija lh, store instrukcija sh,
3. word — load instrukcija lw, store instrukcija sw.

Transferi u količini word i half moraju biti poravnati na adrese djeljive sa 4, odnosno 2. U suprotnom, procesor generira hardversku iznimku.

Instrukcije lb i lh djelimično popunjavaju sadržaj registra informacijama dobijenim iz memorije. Pri tome lb čita jedan bajt iz memorije i popunjava najniži bajt u destinacijskom registru, dok lh radi istu operaciju, ali sa pola riječi, tj. sa dva bajta.

Preostali biti u destinacijskom registru nakon izvršenja instrukcija lb ili lh popunjavaju se na osnovu predznaka vrijednosti koja je prenesena iz memorije u registar, i to cifrom 1 u slučaju negativnog predznaka, ili cifrom 0 u slučaju pozitivnog predznaka. MIPS procesor još podržava i instrukcije lbu i lhu, koje uvijek popunjavaju preostale bite cifrom 0.

Ukoli
nake iz p
koja je fo

la reg,

Ovim
oznakom

Ukoli
tada aser
suprotne
vatno ispr

Da bi
tup mem

.data
ver1:
ver2:
ver3:
.text
.glob
ver1:
add
la
la
la
tb
lbu
sw
sw
jr

① Du
pu
② U
t2

Ukoliko je potrebno da u neki registar učitamo adresu neke oznake iz programa, to možemo uraditi putem pseudo instrukcije `la` koja je formata:

```
la reg, L
```

Ovim se sadržaj registra `reg` ispunjava adresom asociranom sa oznakom `L`.

Ukoliko adresa oznake `L` predstavlja broj koji može stati u 16 bita, tada asembler mijenja pseudo instrukciju `la` instrukcijom `addiu`. U suprotnom su potrebne dvije instrukcije, `lui` i `addiu`, da bi se adekvatno ispunio zadatak pseudo instrukcije `la`.

Da bismo demonstrirali upotrebu navedenih instrukcija za pristup memoriji, koristit ćemo sljedeći asembler fajl:

```
.data
var1: .byte 0xff ①
var2: .word 0 ①
var3: .word 0 ①
.varL
.global main
main:
    addi $v0, $0, 0
    la $t1, var1 ②
    la $t2, var2 ②
    la $t3, var3 ②
    lb $t5, 0($t1) ③
    lbu $t6, 0($t1) ③
    sw $t5, 0($t2) ④
    sw $t6, 0($t3) ⑤
    jr $ra
```

- ① Definiramo tri globalne varijable `var1`, `var2` i `var3`, koje popunjavamo vrijednostima -1, 0 i 0.
- ② Učitavamo adrese varijabli `var1`, `var2` i `var3` u registre `t1`, `t2` i `t3`.

- Sa `lb` učitavamo vrijednost varijable `var1` u registar `t5`, a sa
- ③ lbu radimo istu operaciju bez uzimanja u obzir predznaka, a rezultat snimamo u registar `t6`.
 - ④ Mijenjamo vrijednost varijable `var2` u memoriji sadržajem registra `t5`.
 - ⑤ Mijenjamo vrijednost varijable `var3` u memoriji sadržajem registra `t6`.

Tok gornjeg programa možemo propratiti izvršavanjem programa u debageru, instrukciju po instrukciju.

```
(gdb) x/12i *main ①
0x4001f4 <main+>;      addi    v0,zero,0
0x4001f8 <main+4>;     lui     t1,0x40
0x4001fc <main+8>;     addiu   t1,t1,4112
0x400200 <main+12>;    lui     t2,0x40
0x400204 <main+16>;    addiu   t2,t2,4116
0x400208 <main+20>;    lui     t3,0x40
0x40020c <main+24>;    addiu   t3,t3,4120
0x400210 <main+28>;    lb      t5,0(t1)
0x400214 <main+32>;    lbu    t6,0(t1)
0x400218 <main+36>;    sw     t5,0(t2)
0x40021c <main+40>;    jr     r2
0x400220 <main+44>;    sw     t6,0(t3)

(gdb) b *main
Breakpoint 1 at 0x4001f4
(gdb) c
Continuing.

Breakpoint 1, 0x004001f4 in main ()
(gdb) disp/i $pc
1: x/i $pc
=> 0x4001f4 <main+>;      addi    v0,zero,0
(gdb) ni
0x004001f8 in main ()
1: x/i $pc
=> 0x4001f8 <main+4>;     lui     t1,0x40
(gdb) p/x $t1 ②
$1 = 0x0
(gdb) p/x &var1 ②
$2 = 0x401010
(gdb) ni
0x004001fc in main ()
1: x/i $pc
=> 0x4001fc <main+8>;     addiu   t1,t1,4112
(gdb) ni
0x00400200 in main ()
```

registar t5, a sa
predznaka, a

memoriji sadržajem

memoriji sadržajem

čitanjem programa

```
1: x/i $pc
=> 0x400200 <main+12>:      lui      t2,0x40
(gdb) p/x $t1 ③
$t1 = 0x401010
(gdb) ni
0x00400204 in main ()
1: x/i $pc
=> 0x400204 <main+16>:      addiu   t2,t2,4116
(gdb) ni
0x00400208 in main ()
1: x/i $pc
=> 0x400208 <main+20>:      lui      t3,0x40
(gdb) ni
0x0040020c in main ()
1: x/i $pc
=> 0x40020c <main+24>:      addiu   t3,t3,4120
(gdb) ni
0x00400210 in main ()
1: x/i $pc
=> 0x400210 <main+28>:      lb      t5,0(t1)
(gdb) p/x $t5 ④
$t5 = 0x0
(gdb) ni
0x00400214 in main ()
1: x/i $pc
=> 0x400214 <main+32>:      lw      t6,0(t1)
(gdb) p/x $t5 ④
$t5 = 0xffffffff
(gdb) p/x $t6 ⑤
$t6 = 0x0
(gdb) ni
0x00400218 in main ()
1: x/i $pc
=> 0x400218 <main+36>:      sw      t5,0(t2)
(gdb) p/x $t6 ⑤
$t6 = 0xff
(gdb) p/x var2 ⑥
$var2 = 0x0
(gdb) ni
0x0040021c in main ()
1: x/i $pc
=> 0x40021c <main+40>:      jf      t6,0(t2)
     0x400220 <main+44>:      sw      t6,0(t2)
(gdb) p/x var2 ⑥
$var2 = 0xffffffff
(gdb) p/x var3 ⑦
$var3 = 0x0
(gdb) ni
0x00400240 in __libc_start_main (main=0x4001f4 <main>, argc=1, argv=0x
73 src/env/_libc_start_main.c: No such file or directory.
1: x/i $pc
=> 0x400460 <__libc_start_main+564>:      jal      0x400470 <exit>
     0x400464 <__libc_start_main+568>:      move    #0,v0
(gdb) p/x var3 ⑦
$var3 = 0xff
(gdb) q
```

Do sada smo učili metičke instrukcije. Pomoći će nam u tome pravolinjano način rada korisnog

U ovom poglavlju ćemo promatrati gramatičke jezicima, uključujući while, unless i if.

Za pomoći će nam — skupovi, promjene tipa skupova, itd.

Kao primjer možemo koristiti drugim vježbama redbe sa

MIPS load

- ① Primjećujemo da je svaka pseudo instrukcija la unutar objektnog fajla zamijenjena sa dvije instrukcije, lui i addiu.
- ② Ispisujemo inicijalni sadržaj registra t1 i adresu varijable var2.
- ③ Nakon izvršenja lui i addiu, registar t1 sadrži adresu varijable var2.
- ④ Sadržaj registra t5, prije i poslije operacije lb.
- ⑤ Sadržaj registra t6, prije i poslije operacije lbu.
- ⑥ Sadržaj varijable var2, prije i poslije prve sw operacije.
- ⑦ Sadržaj varijable var3, prije i poslije prve sw operacije.

Promjena toka programa

Do sada smo razmatrali samo asemblerske instrukcije koje vrše aritmetičke i logičke operacije, tzv. aritmetičko-logičke instrukcije, i instrukcije za komunikaciju sa memorijom, tzv. load/store instrukcije. Pomoću ovih instrukcija možemo pisati programe koji imaju pravolinijsku strukturu, ali ne i programe koji imaju uslovna grananja ili pozive funkcija, što je neophodno u bilo kojem realnom i korisnom programu.

U ovom poglavlju upoznaćemo se sa asemblerskim instrukcijama za promjenu toka programa, odnosno realizaciju funkcija i programske strukture selekcije i iteracije, koje se u višim programskim jezicima najčešće implementiraju pomoću naredbi if, switch, for, while, until itd.

Za promjenu toka programa često se koristi i kraći termin — *skok*. U bilo kojem programskom jeziku, koji ima mogućnost promjene toka programa, možemo primijeti dva suštinski različita tipa skokova:

- bezuslovni skokovi i
- uslovni skokovi.

Kao primjere ovih programskih struktura, u ovom poglavlju ćemo koristiti njihovu implementaciju u programskom jeziku C. U drugim višim programskim jezicima koriste se gotovo identične naredbe sa vrlo sličnom sintaksom.

④ jer je
"presko

Nak
to nare
mjenu t
redbu €
naredne
skok na
izvršava
ponavlja

Kad
funkcije
program
koje je r
gradama
sve do r
znak da
zvana o
je važio
hanizma
omeđen
konteks
u asemb
poziv fu
kod.

Prije
za realiz
gram i k

Rekli
nom ob
šavaju u

Bezuslovni skokovi

Pod terminom *bezuslovni skok* podrazumijevamo promjenu toka programa koja će se izvršiti uvijek kad god programski tok dode do odgovarajuće naredbe ili instrukcije. Ovaj skok nije uslovjen nikakvim uslovom.

U programskom jeziku C bezuslovni skokovi se vrše u slučajevima kada program nađe na:

- naredbu `goto`,
- poziv funkcije,
- naredbu `return`,
- kraj bloka u kojem je implementirana funkcija.

Naredbom `goto` vrši se bezuslovni skok na naredbu koja se nalazi neposredno nakon oznake u C programu iza koje slijedi znak `:`. Ove oznake se često nazivaju labele, od engleskog naziva *label*. U narednom listingu je dat primjer korištenja `goto` naredbi u C programu.

```
#include <stdio.h>
int main()
{
    unsigned int i;
    i=0;
    goto ovdje; ①
    i = i + 1; ②

    ovdje: ③
    i = i + 100; ④

    i = i + 10; ⑤
    i = i + 1;
    printf("i je %d\n", i);
    goto ovdje; ⑥

    return 0; ⑦
}
```

Naredbom ① vrši se bezuslovni skok na oznaku (labelu) koja je označena sa `ovdje` (③). U stvari, efektivno se skok vrši na naredbu

● jer je to prva naredba nakon labele ovdje. Naredba ● će se uvijek "preskočiti" i nikad se neće izvršiti.

Nakon ovog skoka izvršiće se redom sve naredbe do naredne goto naredbe označene sa ●. Ova naredba će izazvati bezuslovnu promjenu toka programa na prvu naredbu nakon labele ondje, tj. naredbu ●. Nakon naredbe ●, izvršavaju se ponovo sve naredbe do naredne goto naredbe, tj. naredbe ●, koja opet uzrokuje bezuslovni skok na ● i tako unedogled, dok se ne isključi računar na kojem se izvršava ovaj program ili dok se program nasilno ne prekine. Ovo ponavljanje je poznato pod terminom *beskonačna petlja*.

Kad C program pri izvršavanju naide na poziv funkcije (ime funkcije iza kojeg slijede zagrade () sa eventualnim parametrima), programski tok se prebaci na prvu naredbu unutar tijela funkcije, koje je realizovano kao imenovani blok koda omeđen vitičastim zagrada {} . Potom se izvršavaju naredbe u kontekstu te funkcije sve do naredbe return ili zagrade } na kraju tijela funkcije. Ovo je znak da programski tok treba da se "vrati" na mjesto odakle je pozvana ova funkcija i da se nastavi izvršavati dalje, u kontekstu koji je važio prije poziva ove funkcije. Ovo je krajnje uprošten opis mehanizma "poziva" funkcije u C jeziku. Pojmovi kao što su blok koda omeđen vitičastim zagrada {} i izvršavanje programskog koda u kontekstu funkcije su apstrakcije na vrlo visokom nivou kojih nema u asembleru. Čak i "skok" koji se izvrši pri goto naredbi, a pogotovo poziv funkcije, je suviše velika apstrakcija u odnosu na asemblerski kod.

Prije nego što krenemo sa objašnjanjem asemblerskih instrukcija za realizaciju "skokova", prisjetimo se iz prvog poglavlja šta je program i kako se izvršava u procesoru.

Rekli smo da je program niz tzv. mašinskih instrukcija u binarnom obliku koje su smještene u memoriji te se jedna po jedna izvršavaju u procesoru. U ovom trenutku, dobro pitanje bi bilo: "Kako

instrukcija uopšte dođe u procesor i kako se određuje redoslijed izvršavanja instrukcija?"

Prisjetimo se posebnog 32-bitnog registra u procesoru koji se zove *programski brojač*, čija je oznaka PC od engleskog naziva *program counter*. Razlog postojanja ovog registra je da u svakom trenutku čuva adresu naredne instrukcije programa. Dakle, registar PC sadrži adresu **naredne instrukcije** koja će se dohvatiti i izvršiti u procesoru. Prilikom dohvatanja naredne instrukcije sadržaj ovog registra CPU pošalje ka memoriji sa kontrolnim signalom za čitanje. Memorija odgovara slanjem 32-bitne instrukcije sa adrese primljene od strane procesora. Procesor prihvata ovu instrukciju, programski brojač se inkrementira tako da sadrži adresu naredne instrukcije programa i krene se sa izvršavanjem instrukcije. Ova adresa u registru PC će biti spremna za naredni ciklus dohvatanja instrukcije iz memorije. Kažemo da se programski brojač inkrementira, ali to u stvarnosti znači da se vrijednost programskog brojača, odnosno adrese koju on sadrži, poveća za 4 jer su sve MIPS instrukcije veličine 32 bita, odnosno 4 bajta:

$$PC \leftarrow PC + 4$$

Program se izvršava tako što se na svakom sinhronizacionom signalu u procesoru dešava gore opisani proces: pošalje se sadržaj programskog brojača u memoriju, dohvati se instrukcija, PC se inkrementira a instrukcija izvršava. Sinhronizacioni signali se generišu sa periodom koji je dovoljan da se izvrše odgovarajuće operacije u procesoru i to je karakteristika elektroničkih komponenti od kojih je izgrađen procesor, odnosno računar.

Sada možemo govoriti o pomenutim "skokovima". Jasno je da ništa, ništa i nigdje ne skakuće. Ono što nazivamo skokom ili promjenom toka programa je izvršavanje naredne instrukcije koja se nalazi na nekoj drugoj lokaciji, a ne instrukcije koja se nalazi neposredno u memoriji nakon instrukcije koja se trenutno izvršava.

U MIPS procesoru se skokovi implementiraju tako što se sadržaj programskog brojača (registra PC) postavi na adresu željene instrukcije. Ovaj koncept je vrlo jednostavan. Međutim, MIPS procesor ne omogućava direktno mijenjanje registra PC. Za generisanje bezuslovnih skokova u standardnom MIPS asembleru koriste se tri instrukcije: *j*, *jr* i *jal*.

Instrukcija *j* (*jump*)

Ova instrukcija se koristi za bezuslovni skok na instrukciju označenu asemblerском oznakom (labelom). Sintaksa ove instrukcije je:

j OZNAKA

Navedeni oznaka (labela) u *j* instrukciji može biti bilo koja labela koja se odnosi na instrukciju u asemblerском fajlu u kojem se nalazi *j* instrukcija o kojoj je riječ, a može biti i labela iz drugog fajla ako je označena kao globalna (.global). S obzirom da labela ima značenje memorijске adrese, efekat instrukcije *j* je postavljanje registra PC na adresu koja odgovara navedenoj labeli.

Sa stanovišta programiranja u C jeziku, efekat *j* instrukcije je identičan goto naredbi. U primjeru koji slijedi, prethodni primjer korištenja goto naredbe u C jeziku je napisan u asemblerском kodu. Skokovi realizovani pomoću goto naredbe u C jeziku, u asemblerском kodu se realizuju *jump* instrukcijama na oznake (labele) istog naziva. Ostale instrukcije su označene kao *i1*, *i2*, *i3* itd. da bismo se skoncentrisali na realizaciju skokova.

```
labeled:  
    i1  
    j ovdje  
    i2  
ovdje:  
    i3  
    i4  
used:  
    i5  
    i6
```

U vezi sa instrukcijom *j* treba obratiti pažnju na naizgled čudno ponašanje ove instrukcije: instrukcija koja se u memoriji nalazi neposredno nakon *j* instrukcije će se uvjek izvršiti uz instrukciju *j*, i to neposredno prije instrukcije na koju se vrši skok. Ovakvo ponašanje je karakteristično za sve instrukcije za promjenu toka programa. Zašto se to dešava?

Odgodeni slot

Sve MIPS instrukcije za realizaciju skokova unose tzv. odgođeni slot, o kome je bilo govora u prethodnim poglavljima. Ovaj problem je prisutan u svim procesorima kod kojih je izvršavanje instrukcija realizovano u višefaznom cjevovodu (*pipeline*) radi veće efikasnosti. Kod jednofaznih procesora, kod kojih se izvršavanje cijele instrukcije odvija u jednoj fazi a izvršavanje naredne počinje tek kad se završi prethodna instrukcija, ovaj problem nije izražen.

Postavlja se pitanje zašto se onda uopšte koristi arhitektura cjevovoda ako unosi probleme. Pa, cjevovod ima mnogo više koristi. Uveden je da bi se ubrzalo izvršavanje instrukcija koje traju relativno dugo. Cjevovod omogućava da se u svakom radnom ciklusu procesora učita po jedna instrukcija. Takođe omogućava da se u svakom radnom ciklusu procesora završi izvršavanje jedne instrukcije, čak i ako izvršavanje instrukcije traje duže od jednog ciklusa. Kako to? Sjetimo se pravice automobila iz prvog poglavlja. Slično je i u procesoru. Izvršavanje instrukcije se vrši u različitim fazama, koje su nabrojane u poglavlju 1 ove knjige. Dakle, dok se u jednoj fazi jedna instrukcija učitava, u drugoj fazi se instrukcija koja je ranije učitana izvršava. To je moguće jer je svaka faza cjevovoda realizovana u posebnom fizičkom dijelu procesora te se rezultat iz jedne faze prosljeđuje u narednu na kraju svakog radnog ciklusa. Zato se i zove cjevovod. Međutim, ovakva arhitektura unosi i određene nuspojave, kao što je odgođeni slot, o kome ovdje govorimo.

Probl...
zultata r...
instrukc...
grama je...
izvršava...
program...
ma, adre...
kroz cijel...
nakon t...
tim, u m...
riji nala...
ma, tako...
koju je i...
pomalo...
tavan p...

Za ilu...
vanje na...

.sectio...
.set n...
.globa...
.main:
no...
ad...
L1: ad...
j ad...
ad...
ad...

Pretp...
učitan...
listingu...
morijsk...

Potpuno shvatanje ovog problema zahtijeva poznavanje implementacije procesora, što ne spada u domen ove knjige. Međutim, ovdje ćemo dati objašnjenje koje je dovoljno za razumijevanje tog problema radi pisanja programa u asemblerskom kodu.

Problem odgođenog slota je posljedica problema dostupnosti rezultata nekih vrsta instrukcija kada se taj rezultat koristi u narednoj instrukciji (tzv. *hazard*). Rezultat instrukcije za promjenu toka programa jeste adresa instrukcije na koju se vrši skok, koja se u procesu izvršavanja *jump* instrukcije u procesoru treba izračunati i upisati u programski brojač. U slučaju instrukcija za promjenu toka programa, adresa će moći biti spremljena u PC tek pošto instrukcija prode kroz cijeli cjevovod, odnosno sve faze izvršavanja instrukcije. Tek nakon toga se može učitati instrukcija na koju se vrši skok. Međutim, u međuvremenu je u cjevovod ušla instrukcija koja se u memoriji nalazi neposredno nakon instrukcije za promjenu toka programa, tako da se kao posljedica izvrši i ta instrukcija, a instrukcija na koju je izvršen skok se izvrši tek nakon te instrukcije. Možda zvuči pomalo komplikovano, međutim u suštini je to poprilično jednostavan proces.

Za ilustraciju objašnjenja odgođenog slota analiziraćemo izvršavanje narednog asemblerskog koda:

```
.section text
.set noreorder
.global main
main:
    nop ①
    add $v0, $0, $0 ②
    addi $v0, $v0, 1 ③
    j L1 ④
    addi $v0, $v0, 2 ⑤
    addi $v0, $v0, -9 ⑥
```

Prepostavimo da je gornji asemblerski program asembliran i učitan u memoriju od adrese 0x4000, kako je prikazano u sljedećem listingu u kojem je za svaku instrukciju data i heksadecimalna memorijska adresa na kojoj se instrukcija nalazi:

```

0x4000:    nop ①
0x4004:    add $2, $0, $0 ②
0x4008: L1: addi $2, $2, 1 ③
0x400c:    j L1 ④
0x4010:    addi $2, $2, 2 ⑤
0x4014:    addi $2, $2, -9 ⑥

```

Prisjetimo se objašnjenja faza izvršavanja instrukcije u MIPS procesoru iz poglavlja 1. Instrukcija se izvršava u 5 faza. Međutim, radi jednostavnosti, nećemo posmatrati svaku fazu odvojeno. Smatraćemo da se proces izvršavanja instrukcije odvija u dva dijela. Neka se u prvom dijelu odvija faza 1, u kojoj se vrši preuzimanje instrukcije iz memorije sa adrese pročitane iz programskog brojača i inkrementiranje programskog brojača, a u drugom dijelu neka se odvijaju sve ostale faze koje uključuju dekodiranje instrukcije, učitanje operanada iz registara, izvršavanje ALU operacije i upisivanje rezultata u odredišni registar.

Objasnićemo sa dovoljno detalja proces izvršavanja gornjeg programa u cjevovodu MIPS procesora. Za svaki od navedena dva dijela procesa izvršavanja instrukcije navećemo instrukcije koje se obrađuju u odgovarajućoj fazi cjevovoda. Pri tome ćemo obratiti pažnju na vrijednosti registara PC i \$v0. Odredićemo i prikazati vrijednost registra PC na početku faze 1, na kraju faze 1 i poslije zadnje faze izvršavanja instrukcije. Za registar \$v0, tj. \$2, vrijednost ćemo dati samo na kraju ciklusa izvršavanja instrukcije jer se opšti registri mogu mijenjati jedino u fazi 5.

Clk	Faza 1		Faze 2, 3, 4 i 5			Poslije	
	PC	Instrukcija	PC (novo)	Instrukcija	Opis (efekat)	\$v0	PC
1	4000	nop	4004	?	?	?	4004
2	4004	add \$2, \$0, \$0	4008	nop		?	4008
3	4008	addi \$2, \$2, 1	400c	add \$2, \$0, \$0	V0 <= 0+0	0	400c
4	400c	j L1	4010	addi \$2, \$2, 1	V0 <= v0+1	1	4010
5	4010	addi \$2, \$2, 2	4014	j L1	PC <= 4008	1	4008
6	4008	addi \$2, \$2, 1	400c	addi \$2, \$2, 2	V0 <= v0+2	3	400c
7	400c	j L1	4010	addi \$2, \$2, 1	V0 <= v0+1	4	400c
8	4010	addi \$2, \$2, 2	4014	j L1	PC <= 4008	4	4008
9	4008	addi \$2, \$2, 1	400c	addi \$2, \$2, 2	V0 <= v0+2	6	400c
10	400c	j L1	4010	addi \$2, \$2, 1	V0 <= v0+1	7	4010

Slika 6-1. Prvih 10 ciklusa izvršavanja instrukcija iz primjera

Opisani proces je ilustrovan u tabeli koja je prikazana na slici 6-1. Značenje kolona tabele na toj slici je sljedeće:

Prva kolona, **Clk**, predstavlja redni broj ciklusa izvršavanja instrukcije, relativno u odnosu na ciklus učitavanja instrukcije ①.

Slijede tri kolone **faze 1** cjevovoda MIPS procesora. Kolona **PC** predstavlja vrijednost programskog brojača (PC) na početku te faze. Asemblerska **instrukcija** prikazana u istoimenoj koloni se u fazi 1 učitava sa adrese pročitane iz registra PC. Vrijednost programskog brojača na kraju faze 1, odnosno nakon inkrementiranja njegove vrijednosti je prikazan u koloni **PC (novo)**.

U naredne dvije kolone je navedena asemblerska instrukcija koja se u trenutnom ciklusu izvršava u ostalim fazama cjevovoda MIPS procesora i simbolički opis njenog efekta.

Zadnje dvije kolone predstavljaju vrijednosti registara \$v0 i PC na kraju ciklusa izvršavanja instrukcije. Vrijednosti registara koje se formiraju do kraja ciklusa izvršavanja instrukcije će se koristiti u nadrednom ciklusu.

U prethodnom asemblerskom listingu možemo odmah zaključiti da ćemo imati beskonačnu petlju zbog *jump* instrukcije koja programski tok uvijek vraća dvije instrukcije unazad, na lokaciju L1. Međutim, koje instrukcije će se uopšte izvršiti i kojim redoslijedom?

U svakom trenutku se u procesoru izvršava neka instrukcija. Prije učitavanja instrukcije ①, koja se nalazi na adresi 0x4000, u cjevovodu procesora je već učitana neka instrukcija. U tabeli je ta instrukcija označena sa ?. Neka programski brojač prije prvog ciklusa ima vrijednost 0x4000.

U prvom ciklusu ($Clk = 1$), u prvu fazu cjevovoda učitava se instrukcija sa adrese koja se pročita iz registra PC. To je instrukcija ①.

Nakon učitavanja instrukcije ① vrijednost registra PC se inkrementira, tako da će sada sadržavati vrijednost 0x4004. Za to vrijeme u ostalim fazama cjevovoda se izvršava instrukcija označena sa ?, a koja je učitana u prethodnom ciklusu. Na kraju prvog ciklusa izvršavanja instrukcije vrijednost registra \$v0 je označena sa ? jer vrijednost tog registra zavisi od prethodnih instrukcija, koje u ovom trenutku ne znamo. Vrijednost registra PC će biti 0x4004.

Kad nastupi drugi signal za učitavanje naredne instrukcije (Clock=2), iz memorije se učitava instrukcija sa adresi koja se nalazi u registru PC (0x4004). To je instrukcija ❷. Kad se ta instrukcija učita, vrijednost registra PC se inkrementira (poveća za 4). Sad PC sa-drži vrijednost 0x4008. Istovremeno se u ostalim fazama izvršava instrukcija ❸, koja je proslijedena iz faze 1. Ova instrukcija (nop) ne proizvodi nikakav efekat već samo troši procesorsko vrijeme, od-nosno zauzima odgovarajuće faze. Na kraju ovog ciklusa i dalje ne možemo znati vrijednost registra \$v0. Vrijednost programskog bro-jača određena je u fazi 1 i iznosi 0x4008.

Nastupa signal za učitavanje naredne instrukcije čija je adresa u PC (0x4008). Učitava se instrukcija $\textcircled{2}$ a PC se inkrementira, tj. postavlja na vrijednost 0x400c. Kroz ostale faze, za to vrijeme, prolazi instrukcija $\textcircled{2}$ učitana u prethodnom ciklusu. Ova instrukcija postavlja registar \$v0 na nulu. Na kraju ovog ciklusa, kao rezultat izvršavanja instrukcije $\textcircled{2}$, vrijednost registra \$v0 je 0. Vrijednost registra PC je 0x400c.

Nastupa novi signal za učitavanje naredne instrukcije (Clk=4) čija je adresa u PC (0x400c). Učitava se instrukcija ④ (instrukcija j), a PC se inkrementira i postavlja na vrijednost 0x4010. U ostalim fazama se u tom vremenu dekodira i izvršava instrukcija ③, učitana u prethodnom ciklusu. Ova instrukcija povećava vrijednost registra \$v0 za 1. Na kraju ciklusa register \$v0 će imati vrijednost 1 a PC će imati vrijednost 0x4010.

Na početku postavljenu adresu. Sastavljene su u ostalim redovima učitane u program. U cijeli je postavljanju vrijednosti u jednost 0x40000000 učitana instrukcija se ovdje razmatra. Ovo je vrlo slična rana a učitana instrukcija.

Naredni
instrukcija
krementira
1 cjevovod
va vrijednc
vrijednost

U sedmoj
je u PC (0x40)
vrijednost
cija je vrijednost

Tek ovde
instrukcije
proizvodi se
instrukcije
instrukcije

Na početku narednog ciklusa ($\text{Clk}=5$), PC ima vrijednost 0x4010, postavljenu u prethodnom ciklusu. Učitava se instrukcija ⑤ sa te adrese. Sasvim prirodno, PC se inkrementira na vrijednost 0x4014. U ostalim fazama se dešava dekodiranje i izvršavanje instrukcije učitane u prethodnom ciklusu. To je instrukcija ④, tačnije instrukcija j, koja vrši promjenu toka programa na labelu L1. Ova labela u datom slučaju ima značenje adrese 0x4008. Efekat instrukcije ④ je postavljanje registra PC na vrijednost 0x4008. Na kraju ciklusa će vrijednost registra \$v0 ostati nepromijenjena, tj. 1, a PC će imati vrijednost 0x4008. Uočimo da je u ovom ciklusu u fazu 1 cjevovoda učitana instrukcija neposredno nakon instrukcije ④ bez obzira što se ovdje radi o instrukciji j koja vrši skok na lokaciju L1. Razlog za ovo je vrlo jednostavan — instrukcija ④ još uvijek nije bila dekodirana a vrijednost registra PC na početku ciklusa je bila 0x4010 pa je i učitana instrukcija sa te adrese.

Naredni ciklus ($\text{Clk}=6$). Procesor se ponaša normalno. Učitava se instrukcija ③ sa adrese pročitane iz registra PC (0x4008) i PC se inkrementira. U ostalim fazama se izvršava instrukcija ⑤ koja je u fazu 1 cjevovoda učitana u prethodnom ciklusu. Ova instrukcija uvećava vrijednost registra \$v0 za 2. Na kraju ciklusa će registar \$v0 imati vrijednost 3 a PC vrijednost 0x400c.

U sedmom ciklusu ($\text{Clk}=7$) učitava se instrukcija sa adrese koja je u PC (0x400c). To je instrukcija ②. PC se inkrementira pa će imati vrijednost 0x4010. U ostalim fazama se dekodira i izvršava instrukcija ③, učitana u prethodnom ciklusu.

Tek ovdje možemo reći da se desio skok na labelu L1 zahtijevan instrukcijom j L1 jer tek u ovom ciklusu instrukcija sa lokacije L1 proizvodi svoj efekat. Osim toga, iako je instrukcija ③ fizički nakon instrukcije j, ta instrukcija se već izvršila u tzv. odgođenom slotu instrukcije ④ kao posljedica cjevovoda.

U ovom ciklusu se registar \$v0 povećava za 1 pa će poprimiti vrijednost 4. Vrijednost registra PC na kraju sedmog ciklusa će imati vrijednost kao na kraju ciklusa 4, pa možemo zaključiti da će se nadalje učitavati i izvršavati instrukcije kao u koracima 5, 6, 7, itd.

Za nekoliko narednih koraka, opis svih faza sa vrijednostima registara \$v0 i PC su dati u tabeli na slici 6-1. Primijetimo da se instrukcija ⑥ nikad neće izvršiti. Instrukcije koje se nikad neće izvršiti pojedini kompjajleri i asembleri mogu prepoznati i izbaciti ih iz izvršnog koda sa ciljem optimizacije.

U asemblerском programu koji smo analizirali, putem asembler-ska direktive `set` postavljena je opcija `noreorder`. Ova opcija govori assembleru da prilikom asembliranja programa, odnosno prevodenja u izvršni mašinski kod, zadrži redoslijed asemblerских instrukcija navedenih u izvornom kodu i prevede ih jednu po jednu u mašinski kod. To znači da programer zna šta radi i da je svjestan koje posljedice može izazvati odgodeni slot kod instrukcija za promjenu toka programa.

Ako se izostavi opcija `noreorder`, asembler podrazumijeva opciju `reorder`. Ova opcija govori assembleru da su instrukcije u asemblerском kodu date u logičkom redoslijedu kojim programer želi da se instrukcije izvrše. Asembler je slobodan pri asembliranju mijenjati konačan fizički redoslijed instrukcija u memoriji, uzimajući u obzir i odgođeni slot, sa ciljem postizanja zadanog redoslijeda izvršavanja instrukcija u izvornom asemblerском kodu. Osim izmjene redoslijeda instrukcija, asembler ima i dozvolu dodavanja `nop` instrukcija tamo gdje je to potrebno, u svrhu postizanja istog cilja.

Za ilustraciju ove opcije, neka je u prethodnom primjeru izostavljena direktiva `.set noreorder` ili neka je postavljena opcija `.set reorder`:

```
.section text
.set reorder
.global main
```

```
main:  
    nop  
    add $v0, $0, $0  
    l1: addi $v0, $v0, 1  
    j L1  
    addi $v0, $v0, 2  
    addi $v0, $v0, -9
```

Ovo bi značilo da je u asemblerском коду да logički redoslijed i da programer želi da se nakon instrukcije `j` zaista desi skok na lokaciju `L1` kako bi se izvršila instrukcija na toj lokaciji a ne neka druga. Nakon asembleriranja, rezultujući izvršni kod bi odgovarao sljedećem asemblerском programu:

```
0x4000:      nop  
0x4004:      add $2,$0,$0  
0x4008:  L1: addi $2,$2,1  
0x400c:      j L1  
0x4010:      nop  
0x4014:      addi $2,$2,2  
0x4018:      addi $2,$2,-9
```

Ubačena je `nop` instrukcija koja u odgođenom slotu instrukcije neće izvršavati nikakvu operaciju već će samo trošiti procesorsko vrijeme, odnosno tokom izvršavanja instrukcije zauzimati odgovarajuće faze cjevovoda. Ovo nije jedino rješenje ali je najjednostavnije — nakon svake instrukcije za promjenu toka programa postavi se `nop` koji u odgođeni slot neće učitati ni jednu drugu instrukciju.

Druge rješenje, program koji bi generisao neki pametniji asembler, bi odgovarao sljedećem asemblerском programu:

```
0x4000:      nop  
0x4004:      add $2,$0,$0  
0x4008:  L1: j L1  
0x400c:      addi $2,$2,1
```

U ovom slučaju je instrukcija `addi $2, $2, 1` ubaćena u odgođeni slot instrukcije `j` pa bismo imali isti efekat, s tim što bi se instrukcija `addi $2, $2, 1` izvršavala u svakom drugom ciklusu, što je dosta brže u odnosu na prethodnu verziju, gdje bi se ova instrukcija izvr-

šavala u svakom trećem ciklusu. Osim toga, izbačene su instrukcije koje se ionako ne bi nikad izvršile.



Sve instrukcije za promjenu toka programa imaju tzv. odgođeni slot.

Instrukcije `jal` i `jr`

Instrukcije `jal` (*jump and link*) i `jr` (*jump register*) se takođe koriste za realizaciju bezuslovnih skokova na sličan način kao i instrukcija `j`, ali sa vrlo bitnim razlikama. Najčešće se ove dvije instrukcije koriste za realizaciju koncepata koji se u višim programskim jezicima zovu funkcije, procedure, podrutine itd.

Kao što je rečeno ranije, svaka asemblerska instrukcija, osim pseudo-instrukcija, odgovara jednoj mašinskoj instrukciji procesora. U asembleru, kao programskom jeziku, koncept funkcije ne postoji, pa je nemoguće "pozvati funkciju" u onom smislu kako smo navikli u C jeziku. Pa kako se onda u procesoru izvrši poziv funkcije definisane u C jeziku? Detalji u vezi sa realizacijom onoga što nazivamo funkcijama će biti objašnjeni u poglavlju 7 ove knjige. Ovdje ćemo se upoznati sa mehanikom instrukcija koje se pri tome koriste.

Instrukcija `jal`

Sintaksa ove instrukcije je:

`jal OZNAKA`

Ova instrukcija se koristi za bezuslovni skok na asemblersku oznaku (labelu) datu u instrukciji. Pri tome se vrijednost programskog brojača u trenutku izvršavanja (adresa naredne instrukcije) kopira u registar `$ra`. Otuda i naziv ovom registru — *return address*. Instrukcija `jal` se koristi za ono što bismo mogli nazvati poziv funkcije a mjesto na koje se iz te funkcije treba vratiti, tj. adresa instrukcije koja se treba izvršiti poslije izvršavanja funkcije, se upisuje u registar `$ra`. Upravo zbog toga što ova instrukcija

ja sač
truk
Efeka
\$ra ←

Instrukcije

Sinta

jr

Instr
zi u r
opera
proce
Efeka
PC ←

Treb
slot.

Primjer

Kao
lovanja

- N
- X
- po
- ge
- mi
- no
- je
- N
- b1
- je

ja sačuva adresu mjesta gdje se programski tok treba vratiti, instrukcija je i dobila svoj naziv *jump and link*.

Efekat ove funkcije možemo opisati kao:

$$\$ra \leftarrow PC, PC \leftarrow \text{OZNAKA}$$

Instrukcija jr

Sintaksa ove instrukcije je:

$$jr \text{ registar}$$

Instrukcija *jr* se koristi za bezuslovni skok na adresu koja se nalazi u registru navedenom kao jedini operand ove instrukcije. Kao operand se može koristiti bilo koji registar opšte namjene MIPS procesora, ali se najčešće koristi *\\$ra*, zbog prirode instrukcije *jal*.

Efekat instrukcije *jr* možemo opisati kao:

$$PC \leftarrow \text{registar}$$

Trebamo i ovdje upamtiti da i ove instrukcije imaju odgođeni slot.

Primjer 6-1. Primjer korištenja instrukcija jal i jr

Kao ilustraciju korištenja instrukcija *jal* i *jr*, kao i njihovog djeđovanja, napisaćemo dio asemblerorskog programa koji radi sljedeće:

- Neka su zadane .word vrijednosti na globalnim lokacijama *a*, *b*, *x* i *y*,
- potrebno je na lokaciju *x* upisati vrijednost izraza *a1*b0+a1*, gdje je sa *a1* označena vrijednost varijable *a* kod koje je bit na mjestu 1 postavljen na vrijednost 1, a sa *b0* je označena vrijednost varijable *b* kod koje je bit na mjestu 0 postavljen na vrijednost 0.
- Nakon toga je potrebno na lokaciju *y* upisati vrijednost izraza *b1*a0+b1*, gdje je sa *b1* označena vrijednost varijable *b* kod koje je bit na mjestu 1 postavljen na vrijednost 1, a sa *a0* je označe-

na vrijednost varijable a kod koje je bit na mjestu 0 postavljen na vrijednost 0.

Prepostaviti da pri množenju i ostalim operacijama ne dolazi do prekoračenja 32-bitnog rezultata.

Dakle, dio programa koji treba napisati je ekvivalentan narednom C kodu:

```
x = (a | 2) * (b & 0xffffffff) + (a | 2);
y = (b | 2) * (a & 0xffffffff) + (b | 2);
```

Da bismo realizovali navedenu funkcionalnost potrebno je da napišemo asemblerski kod koji će učitati vrijednosti sa lokacija a i b, izvršiti odgovarajuće operacije i upisati rezultate na lokacije x i y. Ovo nije mali broj instrukcija. Za vježbu čitaocu, realizujte traženu funkcionalnost pomoću instrukcija koje smo do sada naučili! Rješenje do kojeg ćemo doći u nastavku teksta je nešto kraće po broju instrukcija od onog koje ćete Vi napisati, ali ima i još neke prednosti. Analizirajmo problem dalje!

Kada bismo ovaj dio programa pisali u C jeziku, uzimajući u obzir da se u obje linije vrše iste operacije, vjerovatno bismo ove dvije linije koda zamijenili linijama:

```
x = funkcija(a, b);
y = funkcija(b, a);
```

A negdje u izvornom C kodu definisali funkciju:

```
int funkcija (int param1, int param2)
{
    return (param1 | 2) * (param2 & 0xffffffff) + (param1 | 2);
}
```

Ovom izmjenom bi ne samo program učinili preglednijim, nego bismo i smanjili mogućnost greške u pisanju koda koji se ponavlja.

postavljen
ne dolazi do
nared-
trebno je da
sa lokacija a i
lokacije x i y.
ite traženu
naucili! Rje-
će po broju
ke prednos-
ajući u ob-
mo ove dvije
z);
ednjim, nego
se ponavlja.
na toku programa

Međutim, u asemblerskom kodu nema funkcija, ili ih mi još uvi-
jek ne znamo realizovati. Srećom, imamo na raspolaganju instruk-
cije `jal` i `jr` koje možemo iskoristiti da realizujemo traženu funkci-
onalnost na približno elegantan način kao i u gornjem kodu.

Najprije da razmotrimo koje su nam operacije, odnosno instruk-
cije, potrebne za izračunavanje vrijednosti izraza u `return` nared-
bi C funkcije. Pretpostavimo da se vrijednosti parametara `param1` i
`param2` već nalaze u nekim registrima, npr. `$a0` i `$a1`. Izračunajmo
kompletan izraz i rezultat smjestimo u neki treći registar, npr. `$v0`.
Asemblerski kod koji vrši ovaj račun bi bio:

```
    .quad:  
        ori $a0, $a0, 2 ①  
        addi $t0, $0, -2 ②  
        and $a1, $a1, $t0 ③  
        mult $a0, $a1 ④  
        mflo $v0 ⑤  
        add $v0, $v0, $a0 ⑥
```

- ① U registru `$a0` (prvi parametar) postavljamo drugi bit sdesna
na vrijednost 1.

U ovoj i narednoj liniji postavljamo prvi bit sdesna registra
`$a1` (drugi parametar) na vrijednost 0. Da bismo to postigli,
u ovoj liniji smo najprije u pomoći registar `$t0` upisali vri-
jednost `0xfffffffffe`. To smo uradili tako što smo vrijednost
u registru `$t0` sabrali sa vrijednošću -2 korištenjem instrukcije
`addi`. Ova instrukcija će konstantu datu u instrukciji tretirati
kao označenu tako da će ova konstanta zapravo predstavljati
32-bitnu binarnu vrijednost `0xfffffffffe`.

Logičkom operacijom AND nad svim bitima postavljamo prvi
bit registra `$a1` na 0 koristeći kao masku vrijednost registra
③ `$t0`, koju smo postavili na `0xfffffffffe` u prethodnoj instruk-
ciji.

- Vršimo množenje vrijednosti u registrima \$a0 i \$a1. 64-bitni
- ④ rezultat množenja dva 32-bitna broja ide u pomoćne 32-bitne registre HI i LO.

Uzimamo samo donjih 32 bita i smještamo u registar \$v0 (rezultat). Pretpostavili smo ranije da neće doći do rezultata koji je veći od 32 bita.

- Na proizvod dodajemo vrijednost registra \$a0 u kojem se načini rezultat. ⑥ Lazi vrijednost prvog operanda sa setovanim drugim bitom. Ovo je rezultat cijelog računa.

Ovaj dio asemblerskog koda bismo mogli iskoristiti tako što funkcionalnost koja je tražena realizujemo prema sljedećem algoritmu:

1. Učitamo vrijednosti sa lokacija a i b.
2. Vrijednost registara \$a0 i \$a1 postavimo na vrijednosti koje smo učitali sa lokacija a i b.
3. Izvršimo gornji dio koda koji počinje od labele racun (izvršimo bezuslovni skok).
4. Rezultat, tj. vrijednost iz registra \$v0 upišemo na lokaciju x.
5. Vrijednost registara \$a0 i \$a1 sada postavimo na vrijednosti sa lokacija b i a.
6. Ponovo izvršimo gornji dio koda koji počinje od labele racun (bezuslovni skok).
7. Sada rezultat, koji je ponovo u registru \$v0, upišemo na lokaciju y.

Gornjim algoritmom bismo postigli traženu funkcionalnost. Međutim, kako da se vratimo na naredni korak opisanog algoritma jednom kad izvršimo skok na labelu racun? Isti problem imamo svaki put kad izvršimo skok na taj dio koda (koraci 3 i 6). Rješenje ovog

problema leži u osmišljene instrukciji jal.

Dio asemblera u kojem je implementirana ovog primjera je u sljedećem nosno koristi.

```
la $t0, a
lw $s0, 0($t0)
la $t0, b
lw $s1, 0($t0)
```

```
add $a0, $s0
add $a1, $s1
jal racun
la $t0, x
sw $v0, 0($t0)
```

```
add $a0, $s1
add $a1, $s0
jal racun
la $t0, y
sw $v0, 0($t0)
```

Negdje u ovom kodu je implementiran cijeli račun,

```
racun:
ori $t0, $t0, $0
add $t0, $t0, $1
and $t0, $t0, $f
mul $t0, $t0, $t0
mflo $v0
add $t0, $t0, $1
jr $t0
```

Od instrukcija u ovom kodu vauju se vrijednosti a i b, a u rezultatu će nam ove vrijednosti biti u registru \$v0.

U instrukciji jal se učita vrednost u registru \$t0, a u rezultatu cun. U tom rezultatu je u registru \$v0.

problema leži u upotrebi instrukcija `jal` i `jr` jer su ove instrukcije i osmišljene radi funkcionalnosti koja nam treba.

Dio asemblerskog programa koji realizuje traženu funkcionalnost je dat u narednom listingu. Za dati kod i kod koji slijedi do kraja ovog primjera, instrukcije su napisane u logičkom redoslijedu, odnosno korištena je `reorder` opcija asemblera.

```
...  
la $t0, a ①  
lw $s0, 0($t0)  
la $t0, b  
lw $s1, 0($t0) ②  
  
add $a0, $0, $s0  
add $a1, $0, $s1  
jal racun ③  
la $t0, x ④  
sw $v0, 0($t0) ⑤  
  
add $a0, $0, $s1 ⑥  
add $a1, $0, $s0  
jal racun  
la $t0, y  
sw $v0, 0($t0) ⑦  
...
```

Negdje u istom asemblerskom fajlu bismo imali dio koda koji vrši cijeli račun, sa instrukcijom `jr` dodatom na kraju:

```
label:  
    ori $a0, $a0, 2  
    addi $t0, $0, 0xffffe  
    and $s1, $a1, $t0  
    mult $a0, $s1  
    mflo $v0  
    add $v0, $v0, $a0  
    jr $ra ⑧
```

Od instrukcije označene sa ① do instrukcije označene sa ② učitavaju se vrijednosti sa lokacija `a` i `b` i smještaju u registre `$s0` i `$s1` jer će nam ove vrijednosti trebati više puta.

U instrukciji označenoj sa ③ vrši se bezuslovni skok na labelu `racun`. U tom dijelu asemblerskog koda smo realizovali proračun koji

smo dobili kao zadatak. S obzirom na to da smo u tom proračunu pretpostavili da će vrijednosti parametara za račun biti u registrima \$a0 i \$a1, u dvije instrukcije prije instrukcije ③ ćemo vrijednosti iz registara \$s0 i \$s1 kopirati u registre \$a0 i \$a1.

Zašto u instrukciji ③ koristimo `jal` a ne `j`? Da ponovimo, instrukcija `jal` se razlikuje od instrukcije `j` samo po tome što se prije skoka adresa naredne instrukcije smjesti u registar \$ra. Konkretno, u ovom slučaju će se, prije skoka, u registru \$ra sačuvati adresa instrukcije ④.

Za šta nam treba ova adresa? Nakon što se izvrši skok na lokaciju `racun`, izvrše odgovarajuće instrukcije i rezultat smjesti u registar \$v0, imamo instrukciju ③. Ova instrukcija vrši bezuslovni skok na adresu koja se nalazi u registru \$ra. To je upravo adresa instrukcije ④. Ako ste pomislili kako Vas ovakav tok programa neodoljivo podsjeća na poziv funkcije u instrukciji ③ i povratak iz funkcije u instrukciji ④, potpuno ste u pravu. Koncept funkcije i poziva funkcije se realizuje upravo na ovaj način.

Da nastavimo sa objašnjenjem programa. Po povratku u prvobitni dio koda imamo instrukcije ④ i ⑤. Ovim instrukcijama se rezultat, koji je u registru \$v0, smješta na lokaciju x, kako je traženo.

Od instrukcije ⑥ do instrukcije ⑦ imamo gotovo identičan asemblerski kod kao u prethodnih 5 instrukcija. Razlika je u tome što se ovaj put, prije skoka na lokaciju `racun`, u registar \$a0 smjesti vrijednost sa lokacije b, koja je sačuvana u registru \$s1, a u registar \$a1 se smjesti vrijednost sa lokacije a, koja je sačuvana u registru \$s0, jer račun trebamo izvršiti sa takvim vrijednostima kao parametrima "funkcije". Rezultat, koji je ponovo u \$v0, smještamo na lokaciju y.

Gornji dijelovi koda su napisani u logičkom redoslijedu instrukcija, što zahtijeva opciju `reorder` asemblera. Za vježbu čitaocu, napišite ekvivalentan asemblerski kod gornjem kodu uz korištenju op-

ciju no
prikaži
će se, k
ra, u re
uvećan
ki nepo

Uslovni

Za razli
ako je is
ma često
od uslov
ternativ

U pro
naredbe
jezik je
lov unut

U ase
je samo
ukoliko j
instrukci
ovih inst
kasnije, r

Obje i

beq r1,
bne r1,

gdje su
a OZNAKA j
ukoliko je
fizički iza

ciju noreorder. Izvršite analizu izvršavanja instrukcija po fazama i prikažite je u tabeli, kao što je to učinjeno za instrukciju `j`. Uočite da će se, kao rezultat izvršavanja instrukcije `jal` u cjevovodu procesora, u registar \$ra kao povratna adresa upisati adresu instrukcije `jal` uvećana za 8, a da će se u odgođenom slotu izvršiti instrukcija fizički neposredno nakon instrukcije `jal`.

Uslovni skokovi

Za razliku od bezuslovnih skokova, uslovni skokovi se vrše samo ako je ispunjen odgovarajući uslov. Uslovnu promjenu toka programa često nazivamo i *grananje* (engleski *branching*) jer se, u zavisnosti od uslova, programski tok može preusmjeriti na jednu od dvije alternativne grane.

U programskom jeziku C bismo u ovu kategoriju mogli svrstati naredbe `if`, `for`, `while`, `do-while` i `switch`. Zadatak ovih naredbi C jezika je da se izvrši odgovarajući blok koda ako je odgovarajući uslov unutar naredbe ispunjen.

U asemblerskom jeziku koncept bloka koda ne postoji. Moguće je samo izvršiti *skok* na instrukciju koja se nalazi na nekoj adresi ukoliko je odgovarajući uslov ispunjen. Ovdje ćemo razmotriti dvije instrukcije za uslovnu promjenu toka programa, `beq` i `bne`. Pomoću ovih instrukcija, uz korištenje instrukcije `slt` koja će biti objašnjena kasnije, moguće je realizovati bilo koju programsku logiku.

Obje instrukcije imaju istu sintaksu:

```
beq r1, r2, OZNAKA  
bne r1, r2, OZNAKA
```

gdje su `r1` i `r2` oznake registara čije se binarne vrijednosti porede a `OZNAKA` je asemblerska oznaka lokacije (labela) na koju se vrši skok ukoliko je uslov zadovoljen. Lokacija na koju se vrši skok može biti fizički iza ili ispred instrukcije grananja iz koje se inicira skok. Uslov

koji treba da bude zadovoljen definisan je samom instrukcijom kako slijedi:

beq (branch if equal)

Kao što sam naziv instrukcije kaže, grananje na navedenu lokaciju će se izvršiti samo ako su vrijednosti u navedenim registrima jednake.

Primjer upotrebe:

```
...
beq $a0, $a1, L1
... ovaj kod će se izvršiti
... ako $a0 nije isto kao $a1

L1:
... grananje ide ovdje ako je $a0 isto kao $a1
```

Ekvivalentni C kod, uz pretpostavku da se vrijednosti iz registara \$a0 i \$a1 nalaze u varijablama a i b, bi bio:

```
...
if(a == b) goto L1
/* ovaj kod će se izvršiti
ako a nemá istu vrijednost kao b
*/
...
L1: /* grananje ide ovdje ako je a == b */
```

bne (branch if not equal)

Naziv instrukcije sve govori i za ovu instrukciju. Grananje na lokaciju navedenu u instrukciji će se izvršiti samo ako su vrijednosti u navedenim registrima različite.

Primjer upotrebe:

```
...
bne $a0, $a1, L1
... ovaj kod će se izvršiti
... ako je $a0 isto kao $a1

L1:
... grananje ide ovdje ako $a0 nije isto kao $a1
```

Ekvivalentni C kod, i ovaj put uz pretpostavku da se vrijednosti iz registara \$a0 i \$a1 nalaze u varijablama a i b, bi bio:

```

...
if(a != b) goto L1
/* ovdje kod će se izvršiti
ako a ima istu vrijednost kao b
*/
...
L1: /* grananje ide ovdje ako je a != b */
...

```

Ako se u instrukcijama za grananje ne izvrši skok na labelu navedenu u instrukciji jer uslov zahtijevan instrukcijom nije ispunjen, izvršiće se naredna instrukcija, u skladu sa normalnim programskim tokom.

Treba i ovdje napomenuti da sve instrukcije za promjenu programskog toka, pa i instrukcije grananja, imaju odgođeni slot jer u suštini tek u zadnjoj fazi cjevovoda postavljaju programski brojač na adresu koja odgovara labeli navedenoj u instrukciji.

Primjer 6-2. Primjer korištenja instrukcija beq i bne

Najbolje se uči na konkretnom primjeru. Napišimo asemblerски program u skladu sa sljedećom specifikacijom:

- Neka su date označke lokacija x, y i z na kojima su 32-bitni cijeli brojevi (word).
- Ako su vrijednosti na lokacijama x i y jednake, na lokaciju z treba upisati vrijednost 0xffffffff. U suprotnom, na lokaciju z treba upisati zbir vrijednosti koje se nalaze na x i y.

Ovaj problem i nije tako komplikovan tako da bismo ga vrlo lako mogli napisati direktno u asemblerском kodu. Međutim, ovdje ćemo pokazati metodologiju pisanja bilo kojeg algoritma, odnosno programa, u asemblerском kodu na način da krenemo od odgovarajućeg, dosta preglednijeg C koda kojim rješavamo dati problem i u nizu koraka dođemo do konačnog rješenja, odnosno asemblerског koda.

Metodologija se sastoji iz četiri koraka:

1. Napišemo algoritam u C programskom jeziku uz korištenje svih elemenata jezika za strukturirano programiranje, kao što su if, for, while itd.
2. C kod iz prethodnog koraka transformišemo u primitivnu verziju C programskog jezika u kome nema elemenata strukturiranog programiranja kao što su petlje, if-else, switch itd. Pri tome koristimo samo pravolinijsku programsku strukturu, promjenju programskog toka vršimo sa goto a grananje možemo vršiti samo if naredbom tipa
`if(uslov) goto Lokacija`
3. Izvršimo mapiranje korištenih varijabli u registre MIPS procesora za one variable za koje je potrebno.
4. Na osnovu koda u "primitivnom C-u", napišemo asemblerски kod.

U skladu sa **prvim korakom** opisane metodologije, naš konkretni problem možemo opisati jednostavnim i preglednim C kodom

```
if(x == y)
    z = -1;
else
    z = x + y;
```

Drugi korak je transformacija ovog koda u primitivni C:

```
if(x == y) goto jednako; ①
z = x + y; ②
goto kraj; ③
jednako:
    z = -1; ④
kraj:
```

Za **treći korak** nemamo mnogo varijabli. Vrijednosti varijabli ćemo čuvati u registrima kao u tabeli:

Varijabla

x

y

z

Prelazim
redbu iz tre
odgovaraju
odnosu na
kraju upisat
koristimo p
ciju koristim

la	\$t0
lw	\$t1
la	\$t0
lw	\$t2
beq	\$t1
add	\$t3
jr	Kraji
jednako:	
addi	\$t3
kraj:	
la	\$t0
sw	\$t3

Posmatra
koda iz dru
instrukciju.
koda iz C ko

Na počet
trukciji ① as
cije označeno

Zatim ins
gistru \$t0, u

Uсловni skokovi

Varijabla	Registar
x	\$t1
y	\$t2
z	\$t3

Prelazimo na **četvrti korak**. U ovom koraku ćemo svaku naredbu iz trećeg koraka zamijeniti najčešće jednom a nekada sa više odgovarajućih asemblerских instrukcija. Moraćemo još, dodatno u odnosu na C kod, učitati vrijednosti sa lokacija x i y u registre i na kraju upisati vrijednost na lokaciju z. Za učitavanje adrese lokacije koristimo pseudoinstrukciju la a za upisivanje vrijednosti na lokaciju koristimo instrukciju lw.

```

la    $t0, x ⑪
lw    $t1, 0($t0) ⑫
la    $t0, y
lw    $t2, 0($t0)

beq  $t1, $t2, jednako ⑬
add  $t3, $t1, $t2 ⑭
j     kraj ⑮
        addi $t3, $0, -1 ⑯
        la    $t0, z ⑰
        sw    $t3, 0($t0) ⑱

```

Posmatrajmo prethodni asemblerски kod napisan na osnovu C koda iz drugog koraka. Objasnićemo detaljno svaku asemblersku instrukciju. Vidjećemo koliko je jednostavno pisanje asemblerskog koda iz C koda napisanog na način kao u drugom koraku.

Na početku trebamo učitati vrijednosti sa lokacija x i y. U instrukciji ⑪ asemblerskog koda u registar \$t0 učitavamo adresu lokacije označene sa x.

Zatim instrukcijom ⑫ sa lokacije x, odnosno adrese koja je u registru \$t0, učitavamo 32-bitnu (word) vrijednost i smještamo je u

register \$t1, u skladu sa mapiranjem varijabli iz prethodnog koraka. U naredne dvije instrukcije vrijednost sa lokacije y na isti način učitavamo u register \$t2.

Instrukcija ❶ je doslovno prevedena linija ❶ C koda u asembler-ski kod. U navedenim linijama poredimo vrijednosti u registrima \$t1 i \$t2 (varijabli x i y u C kodu) i ako su jednake vršimo skok na lokaciju jednakoj.

U liniji ❷ vršimo sabiranje vrijednosti registara \$t1 i \$t2 a rezultat smještamo u register \$t3. Ovo doslovno odgovara liniji ❷ C koda.

Linije ❸ i ❹ su doslovno prevedene linije ❸ i ❹.

U linijama ❺ i ❻ asemblerskog koda upisujemo dobijenu vrijednost na lokaciju z.

Ovaj problem smo mogli realizovati i obrnutom logikom:

```
if(x != y) goto nije_jednako;
z = -1;
goto kraj;
nije_jednako:
z = x + y;
```

```

Rezultujući asemblerski kod bi tada bio:

```
la $t0, x
lw $t1, 0($t0)
la $t0, y
lw $t2, 0($t0)

bne $t1, $t2, nije_jednako
addi $t3, $0, -1
j kraj
nije_jednako:
add $t3, $t1, $t2
kraj:
la $t0, z
sw $t3, 0($t0)
```

## Instrukcije slt i slti

U mnogim slučajevima je potrebno izvršiti poređenje dvije vrijednosti na način da li je jedna vrijednost manja od druge i u zavisnosti od toga izvrši odgovarajući dio koda. Ovo ne možemo postići korištenjem samo instrukcija `beq` i `bne` iz prostog razloga što se sa ove dvije instrukcije provjerava da li su dvije vrijednosti jednake ili različite a ne i koja je veća.

Za utvrđivanje da li je vrijednost koja se nalazi u nekom registru manja od vrijednosti koja se nalazi u nekom drugom registru koristi se instrukcija `slt` (*set on less than*).

Sintaksa ove instrukcije je:

```
slt rd, rs, rt
```

U slučaju da je vrijednost u registru navedenom na mjestu `rs`, manja od vrijednosti u registru navedenom na mjestu `rt`, odredišni registar, naveden na mjestu `rd`, će se postaviti na vrijednost 1. U ostalim slučajevima, registar `rd` se postavlja na vrijednost 0. Prilikom poređenja, binarne vrijednosti u registrima koji se porede tretiraju se kao označene (*signs*).

Efekat ove instrukcije bismo mogli opisati C kodom kao:

```
rd = rs < rt;
```

ili

```
rd = (rs < rt) ? 1 : 0;
```

Za poređenje vrijednosti u registru sa konstantom koristi se instrukcija `slti` (*set on less than immediate*). Njena sintaksa je:

```
slti rd, rs, C
```

Značenje se već može prepostaviti na osnovu instrukcije `slt`. U slučaju da je vrijednost u registru navedenom na mjestu `rs`, manja od vrijednosti konstante `C`, odredišni registar, naveden na mjestu `rd`, će se postaviti na vrijednost `1`. U ostalim slučajevima, registar `rd` se postavlja na vrijednost `0`. Ovdje treba napomenuti da se u ovoj instrukciji kao konstanta može koristiti samo 16-bitni označeni binarni broj.

Za poređenje vrijednosti na način da se svi operandi tretiraju kao neoznačeni binarni brojevi koriste se instrukcije `sltu` (*set on less than unsigned*) i `sltiu` (*set on less than immediate unsigned*). Sintaksa i efekat ove dvije instrukcije je ista kao odgovarajuće prethodne dvije instrukcije, osim što se operandi tretiraju kao neoznačeni pozitivni cijeli brojevi.

#### Primjer 6-3. Primjer korištenja `slt` instrukcija

Posmatrajmo naredni asemblerski kod:

```
addi $t0, $0, 5 ①
addi $t1, $0, -1 ②
slt $t2, $t0, $t1 # 0 ③
sltu $t3, $t0, $t1 # 1 ④
slti $t4, $t0, -2 # 0 ⑤
sltiu $t5, $t0, -2 # 1 ⑥
```

- ① U registar `$t0` smještamo vrijednost  $5_{10}$ .

② U registar `$t1` smještamo vrijednost `0xffffffff`. Ne zaboravimo, instrukcija `addi` proširuje 16-bitnu konstantu na 32-bitnu vrijednost tako što 32 puta uljevo kopira bit predznaka konstante (bit na mjestu 15).

③ Instrukcija `slt` posmatra operative u registrima kao označene cijele brojeve. Vrijednost u registru `$t0` je `0x00000005` ( $5_{10}$ ) a u registru `$t1` je `0xffffffff` ( $-1_{10}$ ). Odredišni registar `$t2` će dobiti vrijednost `0` jer  $5$  nije manje od  $-1$ .

Kao  
if(x  
{  
/\*  
}

Instrukci

Instrukcija `slt` posmatra operande u registrima kao neoznačene cijele brojeve. Vrijednost u registru  $\$t_0$  je  $0x00000005$  ( $5_{10}$ ). Međutim, binarna vrijednost u registru  $\$t_1$  je nepromijenjena i iznosi  $0xffffffff$ , ali ovaj put ta vrijednost ima značenje  $4294967295_{10}$ . Odredišni register  $\$t_3$  će dobiti vrijednost 1 jer 5 jestе manje od  $4294967295$ .

Instrukcija `slti` ima isti efekat kao instrukcija `slt` s tim što prije samog poređenja 16-bitnu konstantu datu u instrukciji proširi predznakom (vrijednost bita na poziciji 15) na 32-bitni

- ④ broj. Vrijednost u registru  $\$t_0$  je  $0x00000005$  ( $5_{10}$ ) a drugi operand će biti  $0xffffffff$  ( $-2_{10}$ ). Odredišni register  $\$t_4$  će dobiti vrijednost 0 jer 5 nije manje od -2.

Ovdje se operandi posmatraju kao neoznačeni cijeli brojevi. 16-bitna konstanta data u instrukciji na mjestu drugog operanda se proširuje nulama tako da će konstanta  $-2_{10}$  ( $0xffffe$ )

- ⑤ predstavljati vrijednost  $0x0000ffff$  ( $65534_{10}$ ). Register  $\$t_5$  će nakon ove instrukcije imati vrijednost 1 jer je 5 manje od 65534.

Uz pomoć instrukcija `slt` familije, uz instrukcije `beq` i `bne`, možemo realizovati bilo koju programsku logiku. Grananje za slučaj jednakosti ili nejednakosti dvije vrijednosti smo vidjeli ranije. Grananje za slučaj da je jedna vrijednost manja od druge se vrši tako što se prvo, pomoću instrukcija familije `slt`, ispita da li je jedna vrijednost manja od druge i postavi vrijednost nekog pomoćnog registra na 0 ili 1. Nakon toga se, korištenjem `beq` ili `bne` vrši grananje u zavisnosti od vrijednosti u tom pomoćnom registru.

Kao primjer posmatrajmo C kod sa jednostavnom logikom:

```
if(x < y)
{
 /* radi nešto */
}
```

```

else
{
 /* radi nesto drugo */
}

```

Ovaj kod prevodimo u primitivni C:

```

temp = x < y;
if(temp != 0) goto nesto;
/* radi nesto drugo */
goto kraj;
nesto:
/* radi nesto */
kraj:

```

Uočimo kako je izvršena transformacija uslova

```
if(x < y)
```

Najprije je u temp postavljena vrijednost logičkog izraza  $x < y$ . Ako je ovaj izraz tačan, temp će imati vrijednost 1. U suprotnom će temp imati vrijednost 0. Dakle, temp će imati vrijenost ili 0 ili 1. U slučaju da temp ima vrijednost 1 ( $x$  jeste manje od  $y$ ) treba izvršiti skok na lokaciju nesto. To je razlog zašto skok na lokaciju nesto vršimo ako je temp različito od nule — ako temp nije 0 onda je sigurno 1 a to znači da je  $x < y$ .

Sada nije teško ovaj kod prevesti u asemblerski kod. Prepostavimo pri tome da se vrijednosti varijabli  $x$  i  $y$  već nalaze u registrima \$s0 i \$s1:

```

slt $t0, $s0, $s1
bne $t0, $0, nesto
... radi nesto drugo
j kraj
nesto:
... radi nesto
kraj:

```

## Realizacija petlji u asemblerskom kodu

Postoji više načina za implementaciju petlji napisanih u C programskom jeziku. Prije same implementacije u asemblerskom kodu potrebno je izvršiti transformaciju petlji u odgovarajući oblik sa if i goto naredbama, na sličan način kao što je pokazano u prethodnom dijelu ovog poglavlja.

Petlja do-while je možda najjednostavnija za implementaciju i može služiti kao polazni osnov za realizaciju ostalih petlji C jezika. Stoga ćemo prvo vidjeti kako ovu petlju implementiramo u asemblerskom kodu.

Neka imamo do-while petlju oblika

```
do{
 Tijelo
} while(Uslov)
```

Prije implementacije u MIPS asemblerski kod, ovu petlju možemo transformisati u:

```
TIJELO:
 Tijelo
 if(Uslov) goto TIJELO;
```

Petlja while se razlikuje od do-while petlje po tome što se prvo provjeri uslov. Nadalje se obje ponašaju identično. Dakle, ako u priлагodenoj C verziji do-while petlje u dolasku do petlje odmah skočimo do uslova imaćemo while petlju. Dakle, petlja:

```
while(Uslov){
 Tijelo
}
```

se može transformisati u:

```
 goto USLOV;
TIJELO:
 Tijelo
```

```
USLOV:
if(Uslov) goto TIJELO;
```

Petlja for ima tri dijela ali je transformacija, nakon gornjih petlji, sada jednostavna.

```
for(Init; Uslov; Izraz){
 Tijelo
}
```

Gornju petlju možemo transformisati u petlju oblika:

```
Init
goto USLOV;
TIJELO:
 Tijelo
 Izraz
USLOV:
 if(Uslov) goto TIJELO;
```

Nakon odgovarajuće transformacije nije teško napisati ekvivalentni MIPS asemblerski kod za implementaciju bilo koje petlje.

#### Primjer 6-4. Primjer realizacije petlje

---

Dat je niz od 10 32-bitnih označenih cijelih brojeva. Sabrati sve elemente niza i rezultat smjestiti na lokaciju suma. Najmanji element smjestiti na lokaciju najmanji!

Odakle da krenemo? Riješimo ovaj problem u C jeziku kako najbolje znamo! Radi jednostavnijeg debagiranja, napišimo cijeli program. To ćemo uraditi na kraju i za asemblerski kod. Postavimo niz, suma i najmanji kao globalne. Upisaćemo i neke vrijednosti u niz da bismo mogli provjeriti radi li nam program onako kako je zahtijevano.

**Korak prvi:** napišimo program u C jeziku!

```
int niz[10]={10,2,3,4,-5,0,-1,2,3,-9};
int suma, najmanji;

int main()
```

```

{
 int i;
 suma = 0;
 najmanji = niz[0];
 for(i=0; i<10; ++i)
 {
 suma += niz[i];
 if(najmanji > niz[i])
 najmanji = niz[i];
 }
 return 0;
}

```

Gornji kod u narednom koraku transformišemo u C kod koji je blizak asemblerском kodу. Pri tome ћемо if и for transformisati kako smo ranije pokazali а приступ низу ћемо raditi помоћу показиваčа jer asembler ne poznaje низове као концепт. Uslov > transformišemo у одговарајући uslov sa <.

Rezultantni C код у овом кораку је:

```

int niz[10]={10,2,3,4,-5,0,-1,2,3,-9}; ①
int suma, najmanji; ②

int main()
{
 int i;

 suma = 0; ③
 najmanji = *niz; ④
 i=0; ⑤
 goto uslov; ⑥
petlja:
{
 int* adr = niz + i; ⑦
 elem = *adr; ⑧
 suma += elem; ⑨
 if(!(elem < najmanji)) goto nije_najmanji; ⑩
 najmanji = elem; ⑪
nije_najmanji:
 ++i; ⑫
}
uslov:
 if(i<10) goto petlja; ⑬
/* Na zaboraviti spremiti sumu i najmanji u memoriju */ ⑭
return 0; ⑮
}

```

Za pristup  $i$ -tom elementu niza, u gornjem kodu smo koristili pokazivač `adr` koji je tipa `int*`. Vrijednost tog pokazivača, odnosno adresu elementa niza, smo dobili izrazom `adr = niz + i`. Kod ovakvih operacija moramo biti vrlo oprezni prilikom prevođenja ovog koda u asemblerski kod.

Naime, aritmetika sa pokazivačima u C jeziku se razlikuje od aritmetike sa pokazivačima u asembleru. C kompjuter prilikom sabiranja pokazivača sa cijelim brojem vodi računa o tipu pokazivača i adresu inkrementira u koracima veličine podatka na koji taj pokazivač pokazuje. Za razliku od toga, u MIPS asembleru ne postoji posebna aritmetika sa pokazivačima. S obzirom da se za adresiranje memorije koriste registri opšte namjene, adresa je samo binarni broj kao i svaki drugi koji se nalazi u registru. Zbog toga moramo odrediti tačnu adresu elementa posmatrajući memoriju kao niz bajta. S obzirom na to da su elementi našeg niza 32-bitni brojevi (4 bajta), da bismo odredili tačnu adresu elementa, indeks i množimo sa 4 i taj broj dodamo na adresu početka niza `niz`.

Dakle, ako bismo željeli gornji C kod za pristup elementu niza zaista prilagoditi MIPS asembleru što je moguće bliže, onda bismo to mogli napisati kao:

```
char* adr;
adr = (char *)niz + i*4; ⑦
elem = *((int*)adr); ⑧
```

Međutim, ovaj C kod je komplikovaniji od asemblerskog za ono što želimo da postignemo pa u praksi rijetko idemo do ovih detalja. Radije ćemo kod za pristup preko pokazivača ostaviti onako kako je uobičajeno u C jeziku ali moramo biti svjesni na koji način se to treba prevesti u asemblerski kod.

Množenje u MIPS procesoru zahtijeva korištenje pomoćnih registara `L0` i `HI` i kopiranje rezultata u neki registar opšte namjene za daljnju upotrebu. To zahtijeva bar dvije instrukcije. Umjesto instrukcije za množenje, u slučaju potrebe za množenjem sa stepenom

broja 2 možemo koristiti operaciju šiftanja pa se izraz  $i * 4$  može jednostavnije izračunati kao  $i \ll 2$ . Sve ovo ćemo uzeti u obzir pri transformisanju gornjeg C koda u asemblerski kod.

Da bi prevođenje iz C koda u asemblerski kod bilo jasnije, instrukcije u asemblerskom kodu će biti označene identičnim oznakama kao i odgovarajuće linije u gornjem C kodu. Ako je za implementaciju neke linije C koda potrebno više asemblerskih instrukcija, odgovarajuća oznaka će biti postavljena samo uz prvu instrukciju a ostale instrukcije bez oznake koje slijede pripadaju toj implementaciji.

U trećem koraku vršimo mapiranje varijabli korištenih u C kodu u registre MIPS procesora. Za varijable iz gornjeg C koda koristićemo registre kao u narednoj tabeli. Naravno, moglo je i drugče.

| Varijabla | Registar |
|-----------|----------|
| niz       | \$s0     |
| suma      | \$s1     |
| najmanji  | \$s2     |
| i         | \$t1     |
| adr       | \$t2     |
| elem      | \$v0     |

Koristićemo i druge pomoćne registre prilikom implementacije pojedinih linija koje zahtijevaju više instrukcija, kao što su linija 7 za formiranje adrese elementa i uslovi u if.

Ekvivalentni asemblerski kod bi mogao biti:

```
.section .data
.align 2
```

```

niz: .word 10 2 3 4 -5 0 -1 2 3 -9 ①
suma: .word 0 ②
najmanji: .word 0

.section .text
.set reorder
.global main
main:
 la $s0, niz ④
 addi $s1, $0, 0 ③
 lw $s2, 0($s0) ⑤
 addi $t1, $0, 0 ⑥
 j uslov ⑦
petlja:
 sll $t2, $t1, 2 ⑧
 add $t2, $s0, $t2
 lw $v0, 0($t2) ⑨
 add $s1, $s1, $v0 ⑩
 slt $t3, $v0, $s2 ⑪
 beq $t3, $0, nije_najmanji
 add $s2, $0, $v0 ⑫
 rj_najmanji:
 addi $t1, $t1, 1 ⑬
uslov:
 slti $t3, $t1, 10 ⑭
 bne $t3, $0, petlja
 la $t0, suma ⑮
 sw $s1, 0($t0)
 la $t0, najmanji
 sw $s2, 0($t0)
 add $v0, $0, $0 ⑯
 jr $ra

```

Ovo je samo jedna od mogućih implementacija datog problema. U gornjem asemblerskom kodu su instrukcije napisane u logičkom redoslijedu kako bi cijelokupan kod i način prevođenja C koda u asemblerski kod bio jasniji. Zbog toga je korištena `reorder` opcija asemblera. Fizički redoslijed instrukcija u izvršnom kodu ne mora i neće odgovarati redoslijedu instrukcija u gornjem kodu, kako je to objašnjeno ranije u ovom poglavlju.

Prilikom  
obilazno  
nja funk  
metara, p  
toliko pr  
način su

Sve re  
da se blo  
program  
program

Kako  
jelovi ra  
više put  
zvati fu

Funk  
U ovom  
implem  
funkciju  
jednost

U im  
zove st  
način n

## Funkcije i stek

Prilikom pisanja programa u C ili C++ programskom jeziku nezaobilazno je korištenje funkcija. U stvari, programiranje bez korištenja funkcija je nezamislivo. Pozivanje funkcije, proslijđivanje parametara, povratna vrijednost funkcije su pojmovi koji su nam postali toliko prirodni da neki nikada i ne razmišljaju, niti saznaju, na koji način su te apstrakcije u stvari implementirane.

Sve rečeno važi i za druge programske jezike visokog nivoa, s tim da se blokovi koda, koji se u C-u nazivaju funkcije, u nekim drugim programskim jezicima nazivaju procedure, rutine, podrutine, podprogrami, metodi.

Kako god da ih zovemo, u suštini su to imenovani nezavisni dijelovi računarskog programa koje želimo koristiti, odnosno izvršiti, više puta. U dalnjem tekstu ćemo takve imenovane dijelove koda zvati funkcije jer se tako zovu u programskom jeziku C.

Funkcije mogu primati parametre a mogu i vraćati vrijednost. U ovom poglavlju ćemo pokazati kako se u asemblerском kodu implementiraju funkcije, na koji način se proslijeduju parametri u funkciju, kako funkcija prihvata parametre, kako funkcija vraća vrijednost. Nećemo zaobići ni rekurzivne funkcije.

U implementaciji funkcija važnu ulogu igra dio memorije koji se zove stek. Stoga će u ovom poglavlju biti objasnjena uloga steka i način njegovog korištenja.

## Implementacija i pozivanje funkcije

Prilikom korištenja funkcije u jeziku visokog nivoa, kao naprimjer u C-u, navikli smo da "pozovemo" funkciju tako što na odgovarajući način navedemo ime funkcije sa eventualnim parametrima. Pri tome funkcija mora biti "definisana", što znači da negdje treba biti napisan kod za definiciju funkcije u višem programskom jeziku.

Nakon poziva očekujemo da se programski tok prebaci u funkciju, odnosno prvu naredbu unutar njene definicije, a da se nakon okončanja izvršavanja funkcije programski tok vратi na mjesto sa kojeg je funkcija pozvana i da se program nastavi izvršavati dalje. Detalji implementacije na nivou procesora ostaju skriveni od programera.

Međutim, ako želimo implementirati funkciju u asemblerском kodu, moramo voditi računa o svim detaljima i eksplicitno navesti korištenje svakog registra, kao i eksplicitno i konkretno navesti svaku promjenu toka programa.

Kao što je rečeno ranije, asembler ne poznaje pojам funkcije. Ono što nazivamo funkcijom ili pozivom funkcije u C-u, ili bilo kojem drugom višem programskom jeziku, u asemblerском kodu moramo implementirati pomoću odgovarajućih skokova, odnosno instrukcija za promjenu toka programa. Srećom, MIPS ISA posjeduje odgovarajuće instrukcije koje olakšavaju implementaciju ove apstrakcije. Pri tome postoje i određene konvencije kojih se treba pridržavati ukoliko želimo da se naša funkcija implementirana u asemblerском kodu može pozivati i iz C koda i obratno.

Kako implementirati i pozvati funkciju u asemblerском kodu? Za početak — na prvu instrukciju u asemblerском kodu koji implementira logiku funkcije postavimo asemblersku oznaku (labelu) sa željenim imenom funkcije a poslije zadnje instrukcije navedemo instrukciju

```
jr $ra
```

Ovom instrukcijom se programski tok vraća na prvu instrukciju nakon instrukcije `jal` kojom je funkcija pozvana. To znači da asemblerski kod svake funkcije odgovara kodu:

```
jal _ime_funkcije.
... ovdje ide kod za implementaciju funkcije
jr $ra
```

a poziva se instrukcijom `jal`:

```
jal ime_funkcije
```

Međutim, kao što znamo, funkcija u C-u može primati parametre a može i vratiti neku vrijednost. U prethodnom kodu se ne vidi mogućnost za prosljeđivanje parametara ili povratne vrijednosti. Tu već počinju komplikacije.

Standard za implementaciju C funkcija u asemblerskom kodu za MIPS procesor ne postoji. Međutim, postoje određene konvencije kojih se treba pridržavati. Te konvencije opet zavise od kompjuterskog lanca koji se koristi. S obzirom da u ovoj knjizi koristimo GNU kompjuterski lanac, ograničićemo se na konvencije koju isti koristi.

Za početak ćemo navesti dva osnovna pravila za prosljeđivanje parametara u funkciju i povratne vrijednosti iz funkcije:

1. Ako funkcija ima do četiri cijelobrojna argumenta, prvi se prosljeđuje u registru `$a0`, drugi u registru `$a1`, treći u `$a2` i četvrti u `$a3`.
2. Funkcija može vratiti do dvije cijelobrojne vrijednosti. Ako funkcija vraća jednu ili dvije vrijednosti, prva povratna vrijednost se postavlja u `$v0` a druga, ako postoji, u registru `$v1`.

Primijetimo da MIPS implementacija funkcija, u skladu sa konvencijom, može vratiti dvije vrijednosti iz funkcije. Naravno, dvije povratne vrijednosti nije moguće iskoristiti u C-u već samo jednu, onu koja je u registru `$v0`. Međutim, ako povratna vrijednost odgo-

vara C tipu `long long`, odnosno 64-bitnom cijelom broju, onda se za povratnu vrijednost koriste oba v registra.

Navedene konvencije ne kontroliše niti asembler, niti procesor. Ovo su samo dogovorne konvencije za veću kompatibilnost koda.

U primjeru 6-1 prethodnog poglavlja detaljno je analiziran primjer korištenja instrukcija `jal` i `jr`. Dio koda koji je pri tome izdvojen iza labele `racun` može se smatrati implementacijom funkcije `racun` jer zadovoljava sve konvencije koje smo naveli u pogledu prosljeđivanja parametara i povratne vrijednosti iz funkcije. Navedena implementacija funkcije se čak može pozivati i iz C programskog jezika. Njen potpis bi bio:

```
int racun(int, int);
```

Kao drugi primjer implementacije i pozivanja funkcije posmatrajmo sljedeći kod u programskom jeziku C i njegovu implementaciju u asemblerском kodu:

```
int abs(int a){
 if(a<0)
 a = - a;
 return a;
}
```

Asemblerска implementacija gornje funkcije koja vraća apsolutnu vrijednost cijelog broja je:

```
abs: ①
 slt $t0, $a0, $zero ②
 beq $t0, $zero, kraj_abs
 sub $a0, $zero, $a0
kraj_abs:
 or $v0, $a0, $zero ③
 jr $ra
```

Poziv gornje funkcije u C-u:

```
temp = abs(-8);
```

odgovara asemblerskom kodu:

```
addi $a0, $zero, -8 ❶
jal abs
add $t0, $v0, $zero ❷
```

- ❶ Početak implementacije funkcije u asemblerskom kodu smo označili imenom funkcije, odnosno istoimenom labelom.
- ❷ Argument funkcije očekujemo u **\$a0**. Provjeravamo da li je argument negativan i vršimo odgovarajuće operacije.
- ❸ Povratnu vrijednost funkcije smještamo u **\$v0**.
- Prije pozivanja funkcije, odnosno skoka na labelu **abs**, argument funkcije **-8** postavljamo u **\$a0** jer funkcija očekuje argument u tom registru.
- Po povratku iz funkcije, povratna vrijednost je u registru **\$v0**.
- ❾ Sad možemo tu vrijednost upotrijebiti kako želimo. U primjeru je povratna vrijednost kopirana u registar **\$t0**.

Pridržavanje konvencija za razmjenu parametara olakšava i ubrzava pisanje i izvođenje programa. S istim ciljem je projektovana i instrukcija **jal** — povratna adresa se smjesti u registar **\$ra** prije nego što se izvrši skok na funkciju, što pojednostavljuje povratak iz funkcije jer se na osnovu adrese u registru **\$ra** programski tok jednostavno vrati skokom na tu adresu.

Međutim, šta ako pozvana funkcija poziva neku drugu funkciju. Npr. pozvana je funkcija **f** koja poziva funkciju **g**:

```
int main(){
 ...
 f();
 ...
}

void f(){
 ...
 g();
}
```

```
...
 return;
}

void g(){
...
 return;
}
```

što odgovara asemblerskom kodu:

```
main:
...
 jal f ①
L1:
...
f:
...
 jal g ②
L2:
...
 jr $ra ④
g:
...
 jr $ra ③
```

Redoslijed bezuslovnih skokova je sljedeći:

- ① Poziva se funkcija *f*, tj. vrši se skok na labelu *f*. Pri tome se u *\$ra* sačuva povratna adresa, tj. adresa koja odgovara labeli L1.
- ② Poziva se funkcija *g*. Pri tome se u *\$ra* sačuva povratna adresa, tj. adresa koja odgovara labeli L2.
- ③ Vrši se skok na adresu koja je u *\$ra*. To je adresa labele L2.
- ④ Vrši se skok na adresu koja je u *\$ra*. To je ponovo adresa labele L2. Zar u *\$ra* nije adresa labele L1? Ne, otkuda bi bila ta adresa? U *\$ra* je adresa labele L2.

U liniji ②, instrukcijom *jal*, koja u registar *\$ra* uvijek upisuje adresu naredne instrukcije, upisaće se adresa koja odgovara labeli L2

i zauvijek će se izgubiti povratna adresa koja je u \$ra sačuvana instrukcijom ❶ (adresa koja odgovara labeli L1). Tako će pri izvršavanju instrukcije ❷ u registru \$ra biti adresa koja odgovara labeli L2. To vjerovatno nije ono što bismo željeli da se desi.

Da bismo spriječili ovaj problem, stanje registra \$ra prije pozivanja funkcije g, odnosno prije instrukcije ❸, treba na neki način sačuvati i restaurirati ga prije izvršavanja instrukcije ❹. Time bismo postigli željeni programski tok.

U gornjem primjeru je zanemarena činjenica da je funkcija main obično pozvana iz neke druge funkcije (npr. \_start). To znači da bismo i prije pozivanja funkcije f, u gornjem primjeru, trebali sačuvati stanje registra \$ra pa ga restaurirati kad se programski tok vrati iz funkcije f. Sve rečeno za funkciju f i dalje stoji.

S obzirom na to da se pozivanje funkcija i povratak iz istih vrši po LIFO (*last in first out*) redoslijedu, jer će zadnja pozvana funkcija prva okončati svoje izvršavanje pa onda predzadnja itd., za čuvanje povratne adrese, odnosno vrijednosti registra \$ra, koristi se posebno organizovani stek. Za ovaj stek se koriste i nazivi: pozivni stek, izvršni stek, kontrolni stek, mašinski stek... Međutim, najčešće se naziva samo *stek*.

## Stek

U osnovi, za rad sa stekom koriste se dvije operacije: *push* — kojom se elementi ubacuju u stek, i *pop* — kojom se elementi uklanjuju iz steka. Ovdje se očekuje da čitalac poznaje osnovne principe steka (stoga) kao strukture podataka.

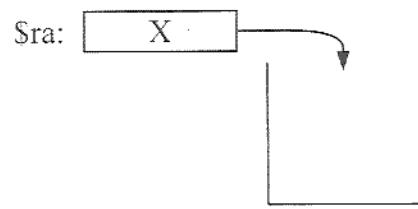
Dakle rješenje gornjeg problema bi odgovaralo narednom pseudokodu:

```
main:
 ...
 push L1
 call f
 add $ra, L2
 ...
```

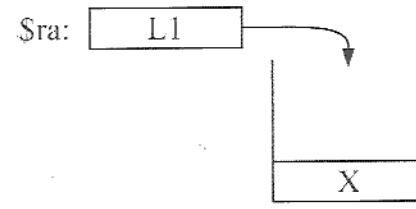
```

push $ra ①
jal f ②
l1:
...
pop $ra ③
jr $ra ④
l2:
...
push $ra ⑤
jal g ⑥
l3:
...
pop $ra ⑦
jr $ra ⑧
l4:
...
jr $ra ⑨

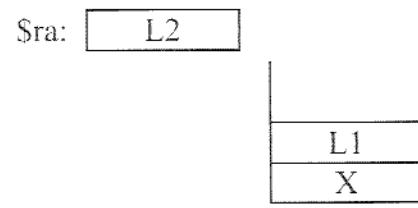
```



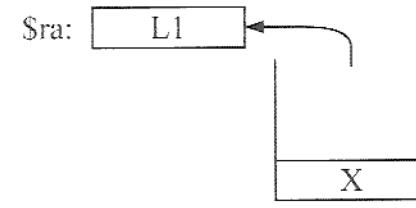
Slika 7-1. Push operacija na steku prije poziva funkcije f



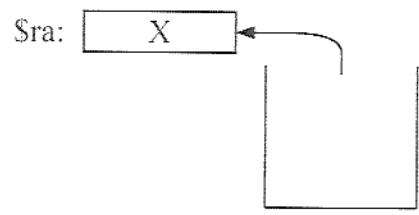
Slika 7-2. Push operacija na steku prije poziva funkcije g



Slika 7-3. Stanje steka i registra \$ra nakon poziva funkcije g



Slika 7-4. Pop operacija na steku prije povratka iz funkcije f



Slika 7-5. Pop operacija na steku  
prije povratka iz funkcije main

S obzirom da je funkcija main pozvana iz neke funkcije (npr. \_start), u registru \$ra je povratna adresa koja će nam trebati

- ❶ za instrukciju ⑨. Označimo adresu koja se trenutno nalazi u \$ra sa X. Pije poziva funkcije f smještamo tu adresu na stek koji je trenutno prazan. Ova operacija je ilustrovana na slici 7-1.

Vrši se "poziv funkcije" f. Pri tome će adresa lokacije L1, na

- ❷ koju se treba vratiti nakon izvršavanja funkcije f, biti upisana u register \$ra.

S obzirom da ćemo pozivom funkcije g izgubiti adresu koja je trenutno u registru \$ra, ponovo vrijednost iz registra \$ra,

- ❸ ovoga puta je to adresa lokacije L1, smještamo na stek za kasniju upotrebu. Stanje steka kao i operacija ubacivanja trenutne vrijednosti registra \$ra na stek je ilustrovana na slici 7-2.

- ❹ Vrši se poziv funkcije g. U register \$ra se upisuje adresa lokacije L2.

Instrukcija jrc se koristi za vraćanje iz pozvane funkcije. Nakon završetka izvršavanja tijela funkcije g vrši se skok na

- ❺ adresu koja je u registru \$ra. To je adresa lokacije L2, upravo ondje u funkciji f gdje se treba nastaviti programski tok nakon poziva funkcije g. U funkciji g ne diramo pozivni stek koji je ilustrovan na slici 7-3.

a na steku  
na steku  
e f  
unkcije i stek

- Na kraju funkcije `f` se trebamo vratiti na mjesto odakle je ona pozvana, tj. na lokaciju `L1`. Međutim, u registru `$ra` je adresa lokacije na koju se trebalo vratiti iz funkcije `g` a ne iz funkcije `f`. Srećom, vrijednost registra `$ra` nakon pozivanja funkcije `f` smo sačuvali na steku. Sada tu vrijednost uzmemo sa steka i smjestimo u registar `$ra` tako da u registru `$ra` imamo spremnu adresu na koju se programski tok treba vratiti nakon izvršavanja funkcije `f`. Ova operacija je ilustrovana na slici 7-4.
- 7 Vrši se skok na lokaciju koja je u registru `$ra`. To je lokacija `L1`.
- Prije povratka iz funkcije `main` trebamo restaurirati stanje registra `$ra`. Uzećemo vrijednost sa steka i smjestiti u `$ra`. Ova operacija je ilustrovana na slici 7-5.
- 8 Povratak iz funkcije `main` vršimo skokom na lokaciju koju smo sa steka smjestili u registar `$ra`.

Dakle, neko opšte pravilo je da ako iz neke funkcije `A` pozivamo drugu funkciju `B` onda prije poziva te druge funkcije vrijednost registra `$ra` trebamo postaviti na stek jer ćemo tu vrijednost trebati uzeti sa steka prije povratka iz funkcije `A` da bi se programski tok mogao vratiti na mjesto odakle je funkcija `A` pozvana. Ako funkcija ne poziva drugu funkciju, to nije potrebno raditi jer se vrijednost registra `$ra` ne mijenja, pa ćemo imati sačuvanu adresu lokacije na koju se treba vratiti. Funkciju koja ne poziva drugu funkciju nazivamo list-funkcijom (engl. *leaf*) jer je zadnja, kao list na grani, i dalje se ne ide nikuda.

Osnovna zadaća steka je čuvanje povratne adrese pri pozivu funkcija. Međutim, stek se koristi i za čuvanje lokalnih varijabli, čuvanje vrijednosti registara koji će se mijenjati, za prosljeđivanje parametara u funkciju itd.

## OČUVANJE VRIJEDNOSTI REGISTARA

Radi očuvanja konteksta izvršavanja neke funkcije, vrlo važno pravilo, odnosno konvencija, je raspodjela odgovornosti za očuvanje vrijednosti registara između pozvane funkcije (engl. *callee*) i funkcije koja ju je pozvala, odnosno funkcije pozivatelja (engl. *caller*):

1. Pozvana funkcija (*callee*), što znači svaka funkcija, se mora pobrinuti da određeni registri, sa stanovišta pozivatelja, ostanu netaknuti. To su s registri (\$s0 do \$s7), \$gp, \$sp, \$fp i \$ra.
2. Funkcija pozivatelj (*caller*) treba da se pobrine da ostali registri koje ona koristi ostanu nepromijenjeni nakon pozivanja neke funkcije.

To se obezbeđuje tako što se vrijednosti odgovarajućih registara postave na stek prije njihove izmjene a onda se ti registri restauriraju vrijednostima sa steka kad je to potrebno.

## STRUKTURA STEKA I AKTIVACIJSKI OKVIR

Stek se u memoriji postavlja na vrh adresnog prostora i raste prema nižim adresama. Vrh steka se kod MIPS procesora prati u registru \$sp koji se zove *pokazivač na stek* (*stack pointer*). To znači da ovaj register u svakom trenutku sadrži adresu vrha steka.

Kod nekih procesora postoje posebne instrukcije za *push* i *pop* operacije, kao u gornjem primjeru u kojem smo ilustrovali princip korištenja steka. Te instrukcije automatski inkrementiraju i dekrementiraju pokazivač na stek.

Kod MIPS procesora takve instrukcije ne postoje. Pristup steku se vrši load/store instrukcijama, kao i prema bilo kojem drugom dijelu memorije, s tim da se kao bazna adresa koristi adresa u registru \$sp. Osim toga, register \$sp se mora eksplicitno inkrementirati i dekrementirati u skladu sa veličinom prostora koji je potreban funkciji.

Zauzimanje prostora na steku se obično vrši pri ulazu u funkciju. Pri ulazu u funkciju se obavljuju i sve ostale pripreme za izvršavanje funkcije. Ovaj dio funkcije (procedure) se naziva prolog.

Dio funkcije koji obavlja obradu pri izlazu iz funkcije naziva se epilog. U ovom dijelu se obavlja svo pospremanje, koje uključuje restauraciju svih registara za koje je zadužena pozvana procedura a koje je mijenjala, kao i oslobođanje zauzetog prostora na steku.

Prostor na steku se zauzima tako što se vrh steka pomjeri ( $\$sp$  se dekrementira) za potreban broj bajta. Npr. ako funkciji treba 64 bajta na steku, taj prostor se zauzme tako što pomjerimo vrh steka za 64 bajta nadole. Ne zaboravimo da stek raste prema nižim adresama tako da je  $\$sp$  potrebno umanjiti. Memorijске adrese tretiramo kao neoznačene brojeve pa koristimo instrukciju addiu:

```
addiu $sp, $sp, -64
```

Ovaj segment memorije na steku koji koristi jedna funkcija naziva se *aktivacijski okvir*. U engleskoj terminologiji se koriste nazivi *activation frame*, *activation record*, *stack frame*. Za čuvanje adrese početka aktivacijskog okvira za aktivnu funkciju koristi se register  $\$fp$  (*frame pointer*), kao što je ilustrovano na slici 7-6.

Ako se u implementaciji funkcije pristup podacima na steku vrši korištenjem registra  $\$fp$ , sama funkcija se mora brinuti o vrijednosti ovog registra. Međutim, u većini implementacija funkcija pristup podacima na steku vrši se isključivo korištenjem registra  $\$sp$ , tako da se register  $\$fp$  može slobodno koristiti kao register opšte namjene ali samo pod uslovom da se na kraju funkcije restaurira na vrijednost koju je ovaj register imao prije poziva te funkcije.

Dakle, mogli bismo reći da se u stvari stek sastoji od aktivacijskih okvira gdje se u svakom aktivacijskom okviru nalaze podaci o stanju odgovarajuće funkcije. Aktivacijski okviri su poredani na steku onim redoslijedom kako su pozivane funkcije. Veličina aktivacij-

funkciju.  
ršavanje

naziva se  
uključuje  
procedura a  
eku.

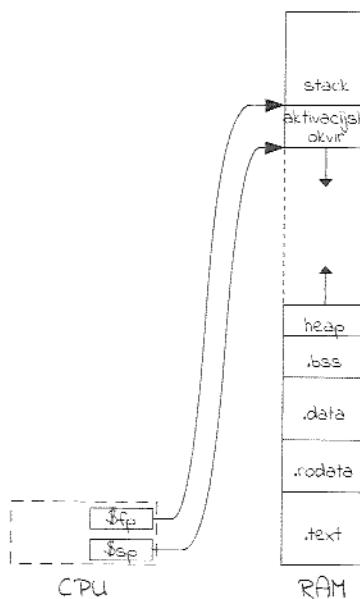
ri (\$sp se  
a 64 baj-  
steka za  
dresama  
amo kao

kacija na-  
te nazivi  
rese po-  
jistar \$fp

teku vrši  
ijednosti  
n pristup  
\$sp, tako  
e namje-  
a na vri-

vacijskih  
o stanju  
na steku  
aktivacij-

cije i stek



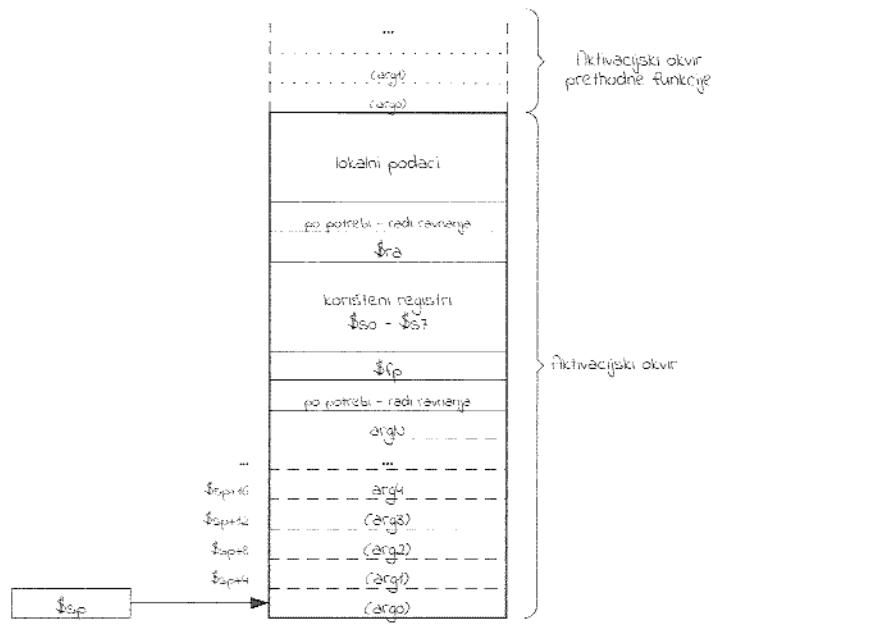
Slika 7-6. Memorijska slika programa

skog okvira mora biti u inkrementima od 8 bajta, odnosno djeljiva sa 8, kako bi 64-bitni podaci imali ravnjanje na adrese djeljive sa 8. Posljedica ovoga je da i \$sp uvijek mora biti djeljiv sa 8.

Aktivacijski okvir funkcije u opštem slučaju može sadržavati:

- Prostor za argumente koje funkcija proslijedi drugim funkcijama koje poziva,
- Prostor za očuvanje s registara (\$s0 do \$s7).
- Mjesto za čuvanje adrese na koju će se funkcija vratiti (register \$ra).
- Prostor za lokalne podatke.

Slika 7-7 ilustruje organizaciju generalizovanog aktivacijskog okvira.



Slika 7-7. Generalizovani aktivacijski okvir funkcije

Na ilustraciji aktivacijskog okvira mogu se uočiti sljedeći dijelovi:

1. Sekcija za argumente koje će funkcija koristiti za proslijedivanje parametara svim funkcijama koje će pozivati. Ova sekcija je organizovana iz 32-bitnih riječi. Prve 4 riječi su rezervisane za prva četiri argumenta ali ih trenutna funkcija nikad ne koristi jer se prva četiri argumenta proslijeduju preko registara \$a0 do \$a3. Međutim, ovaj prostor može koristiti pozvana funkcija za čuvanje prva četiri parametra u slučaju potrebe. Ostatak prostora, \$sp+16, \$sp+20 itd., u ovoj sekciji se koristi za smještanje ostalih argumenata redom kako su definisani u funkciji. Minimalna veličina ove sekcije je 16 bajta a ukupna veličina u bajtima treba biti djeljiva sa 8. Zbog toga je moguće da na kraju ove sekcije bude jedna riječ koja se ne koristi a služi radi ravnanja na adresu djeljivu sa 8. Veličina ove sekcije se određuje na osnovu najvećeg broja parametara među svim funkcijama koje poziva aktivna funkcija.

2. Nakon sekcije sa argumentima ide jedna riječ predviđena za registar \$fp.
3. Sekcija za s registre služi za čuvanje registara \$s0 do \$s7 koje će trenutna funkcija mijenjati. Na početku funkcije ovi se registri sačuvaju na steku a prije povratka iz trenutne funkcije ovi registri se restauriraju sačuvanim vrijednostima. Na taj način funkcija koja je pozvala trenutnu funkciju ne vidi promjenu s registara. S obzirom da će i funkcije koje poziva trenutna funkcija sačuvati vrijednost ovih registara u slučaju da ih koriste, u ove registre se obično smještaju vrijednosti koje želimo da nam ostanu u tim registrima i nakon poziva drugih funkcija. Na kraju ove sekcije se može nalaziti i vrijednost registra \$gp u slučaju da trenutna funkcija ima potrebu da ga mijenja.
4. Nakon sekcije sa s registrima dolazi riječ predviđena za povratnu adresu, odnosno registar \$ra.
5. Ponovo dolazi jedna nekorištena riječ radi ravnjanja na adrese djeljive sa 8 u slučaju potrebe, tj. ako je u prethodne tri sekcije sačuvan neparan broj registara.
6. Sekcija za lokalne podatke sadrži lokalne (automatske) varijable smještene od kraja aktivacijskog okvira prema vrhu steka. U ovoj sekciji se čuvaju i ostali registri koje koristi trenutna funkcija a koje ne želimo da se promijene u slučaju poziva drugih funkcija. Ako trenutna funkcija ne poziva druge funkcije onda nema potrebe za čuvanjem ovih registara na steku. S obzirom da ukupna veličina steka treba biti djeljiva sa 8, na kraju ove sekcije ponovo može doći jedna prazna riječ.

Ako funkcija ne koristi neku od navedenih sekcija onda se ta sekcija izostavi iz aktivacijskog okvira. S obzirom na navedeno, možemo zaključiti da funkcija-list koja ne mijenja s registre niti ostale registre koje u skladu sa konvencijom mora očuvati, uopšte neće alocirati prostor na steku.

Predči dijelovi:

prosljeđivanje. Ova sekcija rezervisane kad ne koristi registara \$a0 zvana funkcije. Ostatak registri za smješteni u funkciji. na veličina u te da na kraju sačuji radi ravne se određuje funkcijama

U primjerima koji slijede biće pokazana tehnika za korištenje steka u MIPS asemblerskom kodu.

#### Primjer 7-1. Jednostavna funkcija-list

Posmatrajmo jednostavnu C funkciju:

```
int suma(int x, int y){
 return x + y;
}
```

Ova funkcija ne poziva drugu funkciju, ne koristi s registre niti bilo koji drugi registar koji mora očuvati. Nema potrebe da manipulišemo stekom. S obzirom na konvencije za parametre funkcije i povratnu vrijednost, implementacija ove funkcije u asemblerskom kodu je vrlo jednostavna:

```
sum:
 add $v0, $a0, $a1
 jr $ra
```

#### Primjer 7-2. Funkcija sa lokalnim podacima koja koristi s registre i poziva drugu funkciju

Posmatrajmo narednu C funkciju:

```
int f(int x, int y, int z) {
 int a[20];
 ... neki proracun
 a[5] = g(y,x,a[2]);
 ... neki proracun
 a[0] = a[5] + h(x, y, a[1], &a[2], z);
 ... neki proracun
 return a[0];
}
```

Ova funkcija zahtijeva prostor za lokalni niz. Ovaj niz ćemo alocirati na steku u sekciji predviđenoj za lokalne podatke. Dodatno, funkcija poziva drugu funkciju, tako da ćemo trebati sačuvati i registar \$ra na steku. Osim toga, pretpostavimo da se u implementa-

ciji ove funkcije koriste registri \$s0 i \$s3, pa ćemo i ove registre trebati očuvati sa stanovišta funkcije koja poziva funkciju f.

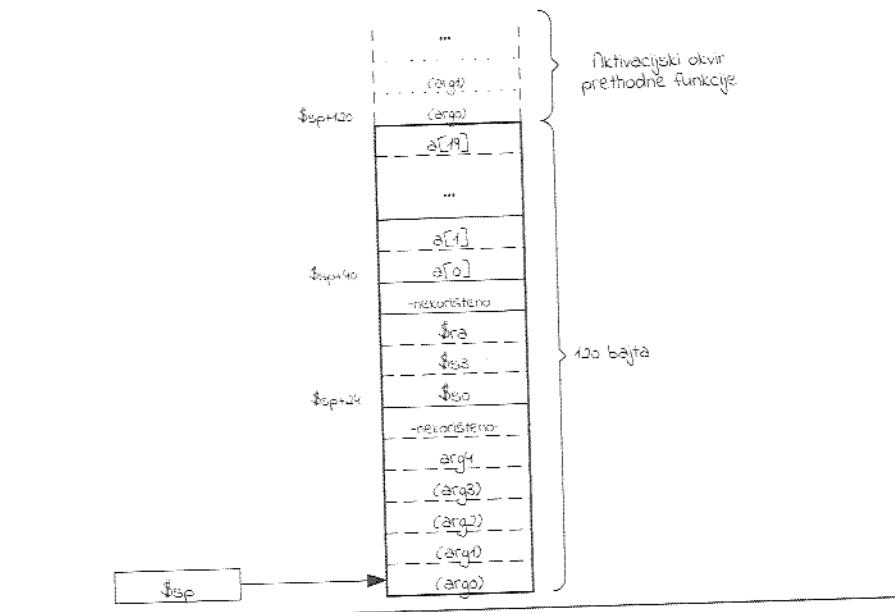
Aktivacijski okvir funkcije f će se sastojati od sljedećih sekcija:

1. Sekcija za argumente funkcija koje poziva funkcija f. U ovoj sekciji nam je potreban prostor za pet argumenata jer to zahtijeva funkcija h. Ova sekcija treba imati veličinu u bajtima djeđljivu sa 8. S obzirom da imamo pet parametara sa po 4 bajta trebaćemo na kraju ove sekcije predvidjeti jednu praznu riječ. To znači da će ukupna veličina ove sekcije biti 24 bajta.
2. Sekcija za s registre će imati dvije riječi, po jednu za registre \$s0 i \$s3.
3. Sačuvaćemo registar \$ra.
4. U prve tri sekcije nam ukupno treba  $24+8+4=36$  bajta. Sekcija sa lokalnim podacima treba imati ravnjanje na 8 bajta. Zbog toga ćemo i na ovom mjestu predvidjeti jednu riječ koja se neće koristiti.
5. U sekciji sa lokalnim podacima nam treba prostor za niz a. Ovaj niz se smješta od 40. bajta pa do kraja aktivacijskog okvira. 20 elemenata veličine 4 bajta ukupno iznosi 80 bajta. Ne-ma potrebe za dodavanjem prazne riječi na kraju aktivacijskog okvira.

Na slici 7-8 je ilustrovan aktivacijski okvir trenutne funkcije.

Ukupna veličina aktivacijskog okvira će biti 120 bajta. Rezervaciju potrebnog prostora, odnosno dekrementiranje vrijednosti regis-tra \$sp, vršimo u dijelu funkcije koji nazivamo prolog. U prologu ćemo sačuvati i sve registre koje ova funkcija mijenja a dužna je očuvati (\$s0, \$s3 i \$ra).

Nakon koda koji implementira logiku funkcije, vratićemo vrijednost regista \$sp na vrijednost koju je imao prije poziva ove funk-



Slika 7-8. Aktivacijski okvir funkcije f iz primjera

cije. Prije toga ćemo restaurirati vrijednosti registara \$s0, \$s3 i \$ra. To vršimo u dijelu funkcije kojeg nazivamo epilog.

Asemblerски kod za implementaciju funkcije f bi bio:

```

... ovdje pocinje prolog
addiu $sp, $sp, -120 ①
sw $ra, 32($sp)
sw $s3, 28($sp)
sw $s0, 24($sp)
... ovdje zavrsava prolog a pocinje tijelo funkcije

... ovdje ide neki proracun

sw $a0, 120($sp) ②
sw $a1, 124($sp)
sw $a2, 128($sp)

add $t0, $a0, $0 ③
add $a0, $a1, $0
add $a1, $t0, $0
lw $a2, 48($sp) ④
jal g ⑤
sw $v0, 60($sp) ⑥

```

... ovdje ide neki proračun

```
lw $a0, 120($sp) ⑦
lw $a1, 124($sp)
lw $a2, 40($sp)
add $a3, $sp, 44 ⑧
lw $t0, 128($sp) ⑨
sw $t0, 20($sp)
jal h ⑩
```

```
lw $t0, 60($sp) ⑪
add $v0, $v0, $t0 ⑫
sw $v0, 40($sp)
```

... ovdje ide račun

```
lw $v0, 40($sp) ⑬
```

... ovdje se zavrsava tijelo funkcije i pocinje epilog

```
lw $s0, 24($sp) ⑭
lw $s3, 28($sp)
lw $ra, 32($sp)
addiu $sp, $sp, 120 ⑮
```

```
jr $ra ⑯
```

Pomjeramo pokazivač steka za 120 bajta ka nižim memorijskim adresama. 120 bajta je veličina potrebnog aktivacijskog okvira funkcije f, odnosno pravimo prostor na steku za trenutni aktivacijski okvir. Ovdje ujedno počinje i prolog funkcije. U naredne tri instrukcije ćemo sačuvati vrijednosti registara \$ra, \$s3 i \$s0 na steku jer ih ova funkcije mijenja. U epilogu ćemo restaurirati originalne vrijednosti ovih registara sa steka.

S obzirom da ćemo pozivati funkcije g i h sa argumentima koji nisu isti kao argumenti funkcije f moraćemo promijeniti vrijednosti registara \$a0 do \$a3. Zbog toga na stek smještamo vrijednosti argumenata koji su proslijedeni u aktivnu funkciju putem ova tri registra jer će nam te vrijednosti trebati kasnije. U skladu sa konvencijom, prostor za ove argumente je već rezervisala funkcija koja je pozvala aktivnu funkciju (funkciju f) unutar svog aktivacijskog okvira, pa ćemo vrijednosti tih re-

i \$ra.

stek

Stek

135

gistara sačuvati na tom mjestu. To ćemo izvesti u ovoj i naredne dvije instrukcije.

Pripremamo se za poziv funkcije  $g$ . U ovoj i narednoj instrukciji u registre  $\$a0$  i  $\$a1$  smještamo vrijednosti prva dva parametra za funkciju  $g$ . Ti parametri odgovaraju C varijablama  $y$  i  $x$ .

④ Treći argument funkcije  $g$  (element niza  $a[2]$ ) smještamo u registar  $\$a2$ .

⑤ Pozivamo funkciju  $g$ .

⑥ Povratnu vrijednost funkcije smještamo u šesti element niza ( $a[5]$ ).

Argumenti za poziv funkcije  $h$  su varijable  $x$ ,  $y$ ,  $a[0]$ , adresa elementa  $a[1]$  i  $z$ . U skladu sa konvencijom, prva četiri argumenta smještamo u  $a$  registre. Prva dva argumenta ćemo učitati sa steka jer smo ih prethodno sačuvali (②), naredni učitavamo iz niza (element  $a[0]$ ).

Četvrti parametar je adresa drugog elementa niza. Adresu formiramo koristeći pokazivač steka kao referentnu adresu. Drugi element se nalazi 44 bajta udaljen od vrha steka. Četvrti parametar funkcije se prenosi putem registra  $\$a3$ .

Peti i svaki naredni parametar prosljeđuje se putem steka na lokacijama  $\$sp+16$ ,  $\$sp+20$  itd. Peti parametar za funkciju  $h$  je u stvari treći parametar aktivne funkcije koji je proslijeđen putem registra  $\$a2$ . Vrijednost registra  $\$a2$  je u međuvremenu promijenjena zbog pozivanja funkcije  $g$ . Međutim, ovu vrijednost smo sačuvali na steku (②) pa ćemo tu vrijednost učitati sa steka i smjestiti na mjesto petog parametra u sekciji za argumente funkcije koju pozivamo. To vršimo u ovoj i narednoj instrukciji.

⑩ Pozivamo funkciju  $h$ .

⑪ U pomoćni registar učitavamo vrijednost elementa niza  $a[5]$ .

U ovoj i narednoj instrukciji računamo vrijednost izraza  $a[5]$

- ⑫  $+ h(x, y, a[1], a[2], z)$  i rezultat smještamo u element niza  $a[0]$ .

U registar  $\$v0$ , koji služi za povratnu vrijednost iz funkcije, učitavamo vrijednost elementa niza  $a[0]$ . Ako između instrukcija ⑪ i ⑫ ne bi bilo drugih instrukcija, onda ne bi bilo potrebe za ovo učitavanje jer je povratna vrijednost instrukcijom ⑪ već smještena u registar  $\$v0$ .

U ovoj i naredne dvije instrukcije restauriramo vrijednosti snimljenih registara. Pri tome u registar  $\$r1$  vraćamo adresu lokacije na koju se treba vratiti iz trenutne funkcije.

- ⑬  $sh$  sadrži adresu lokacije na koju se treba vratiti iz trenutne funkcije.

Vraćamo vrh steka na stanje koje je bilo prije poziva trenutne funkcije tako što povećamo vrijednost registra  $\$sp$  za veličinu aktivacijskog okvira. Ovo odgovara operaciji *pop* aktivacijskog okvira trenutne funkcije. Time se pokazivač steka vraća na aktivacijski okvir funkcije koja je pozvala funkciju  $f$ .

- ⑭ Povratak u funkciju koja je pozvala trenutnu funkciju.

## Rekurzivne funkcije

Rekurzivne funkcije su funkcije koje pozivaju same sebe. Studen-tima često, na početku, asemblerska implementacija rekurzivnih funkcija izgleda zastrašujuća. Međutim, implementacija rekurzivnih funkcija u asemblerskom kodu nije ništa složenija nego implementacija "običnih" funkcija koje pozivaju neku drugu funkciju. Ako nam rekurzivna funkcija izgleda zastrašujuće, zamislimo samo da ta funkcija ne poziva samu sebe nego `neku_drugu_funkciju` i sve će izgledati mnogo lakše.

S obzirom da rekurzivna funkcija vrši poziv funkcije, u njenoj implementaciji će se koristiti stek. S tim u vezi, sve što je rečeno za funkcije koje pozivaju neku drugu funkciju važi i za rekurzivne funkcije.

Ipak, ima jedna sitna razlika u odnosu na obične funkcije. Rekursivne funkcije imaju tzv. osnovni slučaj za koji se prekida rekurzija. Kad se rekursivna funkcija pozove za osnovni slučaj, ona je u stvari funkcija-list jer ne poziva drugu funkciju. U tom slučaju često nije potrebno koristiti stek. Međutim, nije greška ako rekursivnu funkciju implementiramo tako da uvijek u prologu rezervišemo potreban stek a u epilogu ga dealociramo. Najčešće bismo takvu funkciju usmjerili da u svakom slučaju funkcija završava na jednom mjestu u kodu sa jednom `jr` instrukcijom za povratak u funkciju koja je pozvala ovu funkciju.

Moguć je i drugačiji dizajn, da stek alociramo samo za slučaj da je potrebno izvršiti rekursivni poziv ili poziv neke druge funkcije, a za osnovni slučaj da ne koristimo stek.

Pokažimo na primjeru dva moguća dizajna rekursivne funkcije. Neka je potrebno napisati rekursivnu funkciju koja sabira sve prirodne brojeve do proslijedjenog broja. U C jeziku bi funkcija glasila:

```
int suma_n(int n)
{
 if(n==0) return 0;
 return suma_n(n-1) + n;
}
```

S obzirom da je funkcija jednostavna, odmah ćemo je implementirati u asemblerskom kodu. Funkciju ćemo posmatrati kao običnu funkciju koja nije list i rezervisaćemo potreban prostor na steku odmah na početku. Dealokaciju zauzetog prostora na steku ćemo realizovati u epilogu.

Trebamo odlučiti koliki aktivacijski okvir nam je potreban:

- S obzirom da pozivamo funkciju sa jednim parametrom ćemo sekciju sa argumentima: 4 riječi, tj. 16 bajta.
- Nema potrebe za korištenjem `s` registara.

- Trebaće nam prostor za \$ra jer pozivamo "drugu" funkciju. To je još jedna riječ, tj. 4 bajta.
- Više nam ništa ne treba na steku. Moramo rezervisati još jednu riječ koju nećemo koristiti radi ravnjanja na 8 bajta.

Dakle, veličina aktivacijskog okvira ove funkcije je 24 bajta. Asemblerска implementacija gornje funkcije bi bila:

```

 .text
 addiu $sp, $sp, -24
 sw $ra, 16($sp)

 bne $a0, $0, rekurzija
 add $v0, $0, $0
 jr epilog

rekurzija:
 sw $a0, 24($sp)
 addi $a0, $a0, -1
 jal suma_n
 lw $a0, 24($sp)
 addu $v0, $v0, $a0

 lw $ra, 16($sp)
 addiu $sp, $sp, 24

 jr $ra

```

Implementacija bi mogla biti i takva da stek alociramo samo u slučaju rekurzije a ne odmah na početku. Time ćemo malo uštedjeti stek pri osnovnom slučaju i ubrzati program za nekoliko radnih taktova:

```

 .text
 bne $a0, $0, rekurzija
 add $v0, $0, $0
 jr $ra

rekurzija:
 addiu $sp, $sp, -24
 sw $ra, 16($sp)
 sw $a0, 24($sp)
 addi $a0, $a0, -1
 jal suma_n
 lw $a0, 24($sp)
 add $v0, $v0, $a0

```

```
epi log:
 lw $ra, 16($sp)
 addiu $sp, $sp, 24

 jr $ra
```

Ovo je bila vrlo jednostavna funkcija za koju, na kraju ovog poglavlja, zaista nije potrebno komentarisati pojedine linije koda.

Složenija rekurzivna funkcija će imati samo složeniju programsku strukturu ali principi pri implementaciji ostaju isti.

## Mašinski kod

U prethodnim poglavljima smo procesorske instrukcije pisali korištenjem tzv. asemblerских mnemonika, kao što su add, lw, bne itd. Pri tome smo za označavanje registara kao operanada u instrukcijama koristili njihova imena, kao npr. \$v0, \$t5, \$s4, \$sp, \$ra itd., ili njihove brojčane oznake \$0 do \$31. Koristili smo i labele za označavanje memorijskih lokacija za podatke ili za mjesta na koje će se programski tok preusmjeriti korištenjem *branch* ili *jump* instrukcija. Programme pisane u ovom obliku procesor ne može direktno izvršavati.

Da bi se unijeli u memoriju i izvršili od strane MIPS procesora, takvi asemblerски programi se moraju konvertovati u niz 32-bitnih binarnih instrukcija koje razumije MIPS procesor. Ovakve instrukcije napisane u binarnom, a najčešće u heksadecimalnom obliku, nazivaju se *mašinske instrukcije*. Ovdje je bitno naglasiti da se apsolutno sve MIPS instrukcije kodiraju kao 32-bitni binarni brojevi. To je jedna od karakteristika RISC<sup>1</sup> procesora, u koju spada i MIPS procesor.

Dio kompajlerskog lanca koji program napisan u asemblerском jeziku, tzv. asemblerски kod, pretvara u niz mašinskih instrukcija, tzv. mašinski kod, naziva se asembler. Jedna asemblerска instrukcija odgovara jednoj mašinskoj instrukciji, osim za pseudoinstrukcije, koje nemaju svoj mašinski ekvivalent. Upravo iz tog razloga, u ovom poglavlju nećemo razmatrati pseudoinstrukcije.

---

<sup>1</sup> RISC - Reduced Instruction Set Computer



Svaka MIPS instrukcija ima fiksnu veličinu binarnog zapisa: 32 bita.

## Načini adresiranja u MIPS procesoru

U procesoru se izvršavaju različite operacije nad operandima koji mogu dolaziti iz različitih izvora u procesoru ili izvan procesora. Ovdje se pod operandima misli kako na ulazne podatke koji se obrađuju nekom instrukcijom, tako i na izlazne podatke koji se dobiju kao rezultat izvršavanja neke instrukcije. Adresiranje označava način na koji se zadaje lokacija operanada unutar neke instrukcije, ali i način određivanja lokacije instrukcije, odnosno njene memorijske adrese.

Postoji mnogo načina adresiranja. Mnogi procesori podržavaju modifikacije nekih standardnih načina adresiranja a neki podržavaju neke načine adresiranja samo pod određenim uslovima. U mnogim slučajevima je jedan način adresiranja moguć samo sa određenim skupom instrukcija. Nije rijedak ni slučaj da je neki način adresiranja u nekim procesorima rezervisan samo za jednu jedinu instrukciju. Takođe, često neke instrukcije mogu kombinovati dva ili više načina adresiranja, zavisno od operanada.

Ovdje će biti nabrojani samo oni načini adresiranja koji su nam bitni za razumijevanje binarnog kodiranja instrukcija MIPS procesora.

#### **registarsko adresiranje**

Kod registarskog adresiranja, podaci koji se adresiraju se nalaze u registrima. Registr na koji se odnosi instrukcija se zadaje kao dio instrukcije. Zavisno od vrste instrukcije i od broja registara koje podržava ta instrukcija, u instrukcijskoj riječi se rezerviše odgovarajući broj bita za adresiranje željenog registra. Npr. instrukcija za sabiranje sadržaja dva registra čiji rezultat se smješta u treći register kod procesora koji ima 8 registara bi imala po 3 bita (dakle ukupno 9 bita) rezervisanih za adresiranje ta tri registra.

Registersko adresiranje je veoma važno u procesorima gdje se operandi za aritmetičke operacije uvijek ili uglavnom uzimaju iz registara umjesto iz memorije, kao što je slučaj kod MIPS procesora. Osim toga, ovo je vrlo brz način adresiranja jer se po podatak ne ide u memoriju.

### neposredno adresiranje

U engleskoj terminologiji se ovaj način adresiranja naziva *immediate addressing*. Kod ove vrste adresiranja, operandi (ne lokacija operanada) su zadani unutar same instrukcije ili u posebnoj riječi iza instrukcije. Ovaj način adresiranja omogućava korištenje konstanti kao operanada za različite instrukcije.

Ako je konstanta koja se koristi pri neposrednom načinu adresiranja dovoljno mala da stane unutar instrukcije (obično pola ili manje od pola instrukcijske riječi) onda procesor ima sve potrebne podatke u samoj instrukciji, što dodatno ubrzava izvršavanje same instrukcije jer nema potrebe podatke dobavljati iz memorije. Ako konstanta nije dovoljno mala onda se kod nekih procesora ta konstanta smješta iza instrukcijske riječi i obično se mora dohvatiti u narednom instrukcijskom ciklusu.

Neki procesori, kao što je MIPS procesor, imaju fiksnu širinu instrukcijske riječi pa je nemoguće koristiti konstante koje imaju više bita nego što je predviđeno za ovaj način adresiranja.

### direktno adresiranje

Ako se podatak nalazi u memoriji, adresa tog podatka kod nekih procesora može biti zadana kao konstanta unutar same instrukcije. MIPS procesor ne podržava ovaj način adresiranja operanda.

### pseudo-direktno adresiranje

Kod ovog načina adresiranja u samoj instrukciji je zadan jedan dio memorijske adrese, dok se drugi dio određuje na neki drugi način. Najčešće se jedan dio bita binarnog zapisa memorijske adrese uzima iz nekog registra. Ovaj način adresiranja je u MIPS procesoru podržan u instrukciji *j*, o čemu će biti riječi kasnije u ovom poglavljju.

### **indirektno registarsko adresiranje**

Kod indirektnog registarskog adresiranja podatak koji se adresira je smješten u memoriji a adresa memorijske lokacije na kojoj je smješten taj podatak se nalazi u registru. Neki procesori imaju posebnu grupu registara rezervisanu za čuvanje adrese, tzv. adresni registri, dok drugi procesori imaju registre opšte namjene koji mogu čuvati kako vrijednosti tako i adrese podataka. Indirektno registarsko adresiranje je zbog dva razloga jedan od najvažnijih načina adresiranja u mnogim procesorima. Prvo, omogućava jednostavan i prirodan rad sa nizovima podataka, što je veoma često u mnogim aplikacijama. Drugo, indirektno registarsko adresiranje je efikasno sa stanovišta instrukcijskog seta procesora jer omogućava moćno i fleksibilno adresiranje zauzimajući relativno mali broj bita u instrukcijskoj riječi.

Postoji mnogo varijacija ovog načina adresiranja. Jedna od njih je i bazno registarsko adresiranje, koje je podržano u MIPS procesoru.

### **bazno registarsko adresiranje**

Ovo je vrsta indirektnog registarskog adresiranja. U okviru instrukcije se zadaje register u kojem se nalazi bazna memorijska adresa, te se zadaje pomjeraj u odnosu na baznu adresu. Ovaj pomjeraj se zadaje kao konstanta. Podatak (operand) će se dohvatiti sa adrese koja se dobije kao zbir bazne adrese (sadržaj registra zadanog u instrukciji) i pomjeraja (zadanog kao konstanta u instrukciji).

### **PC-relativno adresiranje**

Relativno adresiranje u odnosu na programski brojač se u MIPS procesoru koristi samo za uslovna grananja. Adresa naredne instrukcije se računa kao zbir sadržaja programskog brojača i konstante koja je data u okviru instrukcije. Kod nekih drugih procesora se ovaj način adresiranja koristi i za dohvatanje operanda koji je u memoriji smješten blizu koda koji se izvršava.

Ostale načine adresiranja, koji nisu navedeni, MIPS procesor ne podržava.

## Format MIPS instrukcija

S obzirom na način zadavanja operanada unutar instrukcije, a sa tim i način binarnog kodiranja, MIPS instrukcije grupišemo u tri skupine:

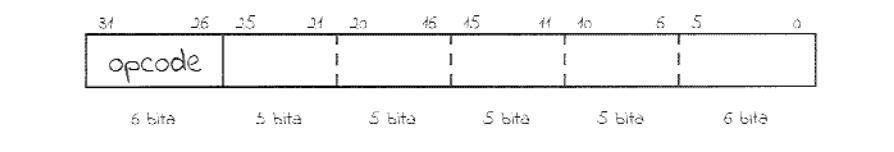
- **Instrukcije R formata** — to su instrukcije kod kojih se vrijednost svih operanada nalaze u registrima. Neke od instrukcija R formata su add, or, sll i druge.
- **Instrukcije I formata** — to su instrukcije kod kojih je vrijednost jednog od operanada data u okviru same instrukcije. Neke od instrukcija I formata su addi, lw, beq i druge.
- **Instrukcije J formata** — to su instrukcije za generisanje bezuslovnih skokova na lokaciju koja je data u okviru same instrukcije. U ovu skupinu spadaju samo instrukcije j i jal.

Ponekad umjesto termina R, I ili J *format* instrukcija koristimo i termine R, I ili J *tip* instrukcija. Koji god termin od ovih da koristimo, misli se na specifičan način binarnog kodiranja instrukcije a ne na njenu funkcionalnost.

MIPS instrukcija je kodirana na način da se 32 bita dijeli na polja odgovarajuće veličine koja sa sobom nose segment informacije o instrukciji, kao što su operacija koju ALU treba da izvrši, način dojavljanja operanda ili operanada operacije, lokacija rezultata operacije itd.

Unutar 32-bitne MIPS instrukcije, prvih 6 bita slijeva (biti 31-26) nose informaciju o tipu instrukcije. Ovo polje nazivamo kod operacije ili skraćeno *opcode* ili samo *op*, od engleskog naziva *operation code*. Preostalih 26 bita se kodira na način specifičan za taj tip instrukcija.

Radi lakšeg pamćenja, mogli bismo reći da je opšti format MIPS instrukcije: 6 – 5 – 5 – 5 – 6, pri čemu navedeni brojevi označavaju veličinu polja u bitima, kao na slici 8-1.



Slika 8-1. Opšti format MIPS instrukcije

Svih 6 polja prikazanih na navedenoj slici su prisutna samo u instrukcijama R tipa. Kod instrukcija I i J tipa pojedina polja označena na slici 8-1 su grupisana u jedno polje sa više bita, što će biti detaljno objašnjeno u narednim sekcijama ovog poglavlja.

Kao primjer izgleda binarnog zapisa MIPS instrukcije, uzmimo instrukciju

add \$v0, \$t1, \$t2

Istu instrukciju bismo mogli napisati korištenjem brojčanih oznaka registara, umjesto njihovih naziva:

add \$2, \$8, \$9

U ovom trenutku, bez objašnjenja, molimo čitaoca da prihvati kao činjenicu da je binarni zapis ove instrukcije:

00000001000010010001000000100000

Ovaj binarni zapis je ekvivalentan zapisu u heksadecimalnom brojnom sistemu:

0x01091020

Iako je heksadecimalni zapis dosta kraći i lakši za upoređivanje i eventualno pamćenje (ako neko uopšte ima potrebu za pamćenjem

binarnog zapisa instrukcije), zapis u čistom binarnom obliku je pregledniji i jasniji sa stanovišta razumijevanja svih elemenata instrukcije.

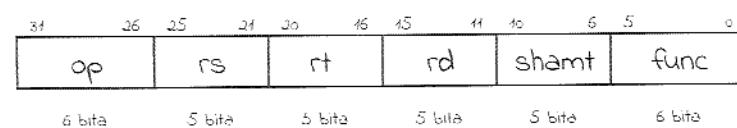
Ako u binarnom zapisu gornje instrukcije bite grupišemo kao na slici 8-1, binarni zapis instrukcije bi bio:

000000 01000 01001 00010 00000 100000

Površnom analizom binarnog zapisa date instrukcije mogu se u 2., 3. i 4. grupi binarnih cifri primijetiti da su vrijednosti 5-bitnih binarnih brojeva 8, 9 i 2, što odgovara oznakama registara iz instrukcije napisane u asembleru. Ovo se ne može na lak način primijetiti u heksadecimalnom zapisu iste instrukcije. Ove činjenice će biti jasne u narednim sekcijama ovog poglavlja, u kojima će biti dat pregled načina kodiranja za sva tri tipa instrukcija.

## R format instrukcija

Instrukcije MIPS procesora kod kojih se operandi nalaze u registrima (otuda naziv R), ili koje uopšte nemaju operanda, ubrajamo u instrukcije R tipa, odnosno R formata. Ove instrukcije se prepoznaju po tome što je vrijednost prvog 6-bitnog polja u instrukciji kodirano kao **000000**. Ove instrukcije imaju tri polja za specifikaciju registara koji se koriste pri izvršavanju date instrukcije, jedno polje za zadavanje broja pomjeranja za shift operaciju (*shift*), odnosno operaciju pomjeranja bita, i jedno polje za zadavanje operacije koja će se izvršiti nad operandima.



Slika 8-2. R format MIPS instrukcije

Kao što se vidi na slici 8-2, instrukcije R formata imaju 6 polja. Značenje tih polja je sljedeće:

- *op* — *operation code*, za R tip instrukcije, ovo polje ima vrijednost 0,
- *rs* — *source register*, registar u kojem se nalazi prvi operand (ili izvorni registar za neke instrukcije),
- *rt* — *target register*, registar u kojem se nalazi drugi operand (ili ciljni registar za neke instrukcije),
- *rd* — *destination register*, registar u koji se smješta rezultat operacije (ili odredišni registar za neke instrukcije),
- *shamt* — *shift amount*, broj bita za šift operacije,
- *func* — *function code*, za instrukcije R tipa, ovo polje je to koje određuje o kojoj instrukciji je riječ.

MIPS procesor posjeduje 32 registra opšte namjene. Svaki registar ima svoju brojčanu oznaku, od 0 do 31. Da bi se u binarnom brojnom sistemu napisala brojčana oznaka bilo kojeg MIPS registra, dovoljno je 5 bita. S obzirom na to, polja koja specificiraju registre zadaju se tako da se brojčana oznaka registra kodira kao 5-bitni binarni broj. Tako će npr. registar \$v0, čija je brojčana oznaka \$2, biti kodiran kao 00010, registar \$s5, čija je brojčana oznaka \$21, će biti kodiran kao 10101 a registar \$ra, čija je brojčana oznaka \$31, biće kodiran kao 11111.

U slučaju šift instrukcije, broj pomjeranja bita za šift operaciju se kodira kao 5-bitni binarni broj.

Neke instrukcije R tipa ne koriste sva navedena polja. U tom slučaju su nekorištena polja kodirana na način da svi biti takvih polja imaju vrijednost 0.

Vrijednosti *func* polja za instrukcije R tipa su date u narednoj tabeli.

Tabela 8-1. Vrijednosti func polja za pojedine instrukcije R formata

| Instrukcija | Korišteni operani | func (bin) | func (dec) |
|-------------|-------------------|------------|------------|
| add         | rd, rs, rt        | 100000     | 32         |
| addu        | rd, rs, rt        | 100001     | 33         |
| and         | rd, rs, rt        | 100100     | 36         |
| break       |                   | 001101     | 13         |
| div         | rs, rt            | 011010     | 26         |
| divu        | rs, rt            | 011011     | 27         |
| jalr        | rd, rs            | 001001     | 9          |
| jr          | rs                | 001000     | 8          |
| mfhi        | rd                | 010000     | 16         |
| mflo        | rd                | 010010     | 18         |
| mthi        | rs                | 010001     | 17         |
| mtlo        | rs                | 010011     | 19         |
| mult        | rs, rt            | 011000     | 24         |
| multu       | rs, rt            | 011001     | 25         |
| nop         |                   | 000000     | 0          |
| nor         | rd, rs, rt        | 100111     | 39         |
| or          | rd, rs, rt        | 100101     | 37         |
| sll         | rd, rt, shamt     | 000000     | 0          |
| sllv        | rd, rt, rs        | 000100     | 4          |
| slt         | rd, rs, rt        | 101010     | 42         |
| sltu        | rd, rs, rt        | 101011     | 43         |
| sra         | rd, rt, shamt     | 000011     | 3          |

| Instrukcija | Korišteni operani | func (bin) | func (dec) |
|-------------|-------------------|------------|------------|
| sraw        | rd, rt, rs        | 000111     | 7          |
| srl         | rd, rt, shamt     | 000010     | 2          |
| srlv        | rd, rt, rs        | 000110     | 6          |
| sub         | rd, rs, rt        | 100010     | 34         |
| subu        | rd, rs, rt        | 100011     | 35         |
| syscall     |                   | 001100     | 12         |
| xor         | rd, rs, rt        | 100110     | 38         |



Za instrukcije R tipa:

- op polje je kodirano kao: 000000,
- func polje određuje o kojoj operaciji se zapravo radi.

Na prvi pogled, kodiranje/dekodiranje MIPS instrukcija može izgledati komplikovano. Međutim, u stvarnosti je to vrlo jednostavan proces. U nastavku ćemo dati nekoliko karakterističnih primjera koji će pomoći u boljem razumijevanju načina kodiranja MIPS instrukcija.

#### Primjer 8-1. Binarni zapis instrukcije and

Potrebno je odrediti binarni i heksadecimalni zapis instrukcije:

and \$v0, \$t0, \$s2

Ova instrukcija vrši binarnu operaciju I nad bitima vrijednosti u registrima \$t0 i \$s2 a rezultat te operacije smješta u register \$v0.

Odredićemo pojedinačna polja unutar binarnog zapisa instrukcije na osnovu operacije i registara. Pri određivanju polja func consultovaćemo tabelu 8-1:

**Polje op:** za instrukcije R tipa ovo polje uvijek ima vrijednost 0<sub>10</sub>, odnosno 000000.

**Polje func:** s obzirom da se radi o operaciji and, operacija I nad bitima, ovo polje će imati vrijednost 36<sub>10</sub>, odnosno 100100.

**Polje rs:** prvi operand se nalazi u registru \$t0. Brojčana oznaka ovog registra je \$8, pa će polje rs imati vrijednost 8<sub>10</sub>, odnosno 01000.

**Polje rt:** drugi operand se nalazi u registru \$s2. Brojčana oznaka ovog registra je \$18, pa će polje rt imati vrijednost 18<sub>10</sub>, odnosno 10010.

**Polje rd:** rezultat operacije se smješta u registar \$v0. Brojčana oznaka ovog registra je \$2, pa će polje rd imati vrijednost 2<sub>10</sub>, odnosno 00010.

**Polje shamt:** operacija and ne zahtijeva pomjeranje bita pa se za ovu operaciju polje shamt ne koristi. Zbog toga se ovo polje kodira kao 0<sub>10</sub>, odnosno 00000.

Konačno, za instrukciju:

and \$2, \$8, \$18

možemo sklopiti njen binarni zapis, kako je to prikazano na slici 8-3.

U svim softverskim alatima, radi jednostavnosti, instrukcija se prikazuje u heksadecimalnom brojnom sistemu. Da bismo jednostavnije izvršili konverziju naše instrukcije iz binarnog u heksadecimalni brojni sistem, njen binarni zapis ćemo napisati u pogodnijem obliku — grupisaćemo po 4 binarne cifre, tzv. tetrade. Nakon to-

Pri  
cijalnog  
je zada  
izvorni

Odr  
je na o

Pol  
odnos

Pol  
nosno

Pol  
imati

Po  
imati

Po  
odred  
gista  
410, C

Pe  
ovu  
kao

K

mfl

k

| op | rs | rt | rd | shamt | func |
|----|----|----|----|-------|------|
| 0  | 8  | 18 | 2  | 0     | 36   |
| 31 | 26 | 25 | 24 | 20    | 16   |
| 15 | 11 | 10 | 6  | 5     | 0    |

6 bita      5 bita      5 bita      5 bita      5 bita      6 bita

000000 01000 10010 00010 00000 100100

Slika 8-3. Binarni zapis instrukcije and

ga svaku tetrudu zapisujemo kao jednu heksadecimalnu cifru. Ovaj postupak je ilustrovan na slici 8-4.

|                        |       |       |       |       |        |    |    |    |   |   |   |
|------------------------|-------|-------|-------|-------|--------|----|----|----|---|---|---|
| 31                     | 26    | 25    | 24    | 20    | 16     | 15 | 11 | 10 | 6 | 5 | 0 |
| 000000                 | 01000 | 10010 | 00010 | 00000 | 100100 |    |    |    |   |   |   |
| 0' 1' 1' 2' 1' 0' 2' 4 |       |       |       |       |        |    |    |    |   |   |   |

Slika 8-4. Primjer 1 — heksadecimalni zapis instrukcije

Sada imamo i heksadecimalni zapis date instrukcije: 0x01121024.

#### Napomena

Cijeli broj se iz binarnog brojnog sistema pretvara u heksadecimalni tako što, krećući sdesna od najmanje težinske vrijednosti, svaku grupu od 4 bita pretvaramo u jednu heksadecimalnu cifru. Dobijene heksadecimalne cifre su cifre ekvivalentnog heksadecimalnog broja. Pretvaranje broja iz heksadecimalnog u binarni brojni sistem se vrši obrnutim postupkom, tj. svaka heksadecimalna cifra se zamjenjuje sa odgovarajućim 4-bitnim binarnim brojem, uključujući vodeće nule. Konačan binarni broj se dobije jednostavnim spajanjem binarnog zapisa svih 4-bitnih grupa, tzv. tetrada.

#### Primjer 8-2. Binarni zapis instrukcije mfhi

Potrebno je odrediti heksadecimalni zapis instrukcije:

mfhi \$a0

Pri izvršavanju instrukcije `mfhi` (*move from Hl*), vrijednost iz specijalnog registra `hi` kopira se u registar `$a0`. U okviru ove instrukcije je zadan odredišni registar `$a0`, a implicitno se koristi registar `hi` kao izvorni registar pri kopiranju podataka.

Odredićemo pojedinačna polja unutar binarnog zapisa instrukcije na osnovu operacije i upotrijebljenih registara:

**Polje op:** za instrukcije R tipa ovo polje uvijek ima vrijednost  $0_{10}$ , odnosno `000000`.

**Polje func:** za datu instrukciju ovo polje ima vrijednost  $16_{10}$ , odnosno `010000`.

**Polje rs:** za instrukcije `mfhi` i `mflo` ovo polje se ne koristi pa će imati vrijednost  $0_{10}$ , odnosno `00000`.

**Polje rt:** za instrukcije `mfhi` i `mflo` ovo polje se ne koristi pa će imati vrijednost  $0_{10}$ , odnosno `00000`.

**Polje rd:** za instrukcije `mfhi` i `mflo` ovdje dolazi brojčana oznaka odredišnog registra opšte namjene. U konkretnom slučaju, to je registar `$a0` čija je brojčana oznaka `$4`, pa će polje `rd` imati vrijednost  $4_{10}$ , odnosno `00100`.

**Polje shamt:** ova instrukcija ne zahtijeva pomjeranje bita pa se za ovu operaciju polje `shamt` ne koristi. Zbog toga se ovo polje kodira kao  $0_{10}$ , odnosno `00000`.

Konačno, za instrukciju:

`mfhi $4`

konstatujemo njen binarni zapis, kao na slici 8-5.

Prisjeti  
blerske na  
asembler  
cesa asem  
trukcije n

| op                                                                                    | rs      | rt      | rd      | shamt | func  |         |
|---------------------------------------------------------------------------------------|---------|---------|---------|-------|-------|---------|
| 0                                                                                     | 0       | 0       | 4       | 0     | 16    |         |
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | 0000000 | 0000000 | 0000000 | 00100 | 00000 | 0100000 |

6 bita      5 bita      5 bita      5 bita      5 bita      6 bita

Slika 8-5. Primjer 2 — binarni zapis instrukcije mfhi \$a0

Iz binarnog zapisa sa slike 8-5, grupisanjem po 8 bita, sa istom efikasnošću kao i grupisanjem po 4 bita, možemo dobiti heksadecimalni zapis instrukcije jer su uočljive binarne tetrade.

bin: 00000000 00000000 00100000 00010000  
hex: 0 0 0 0 2 0 1 0

Dakle, heksadecimalni zapis date instrukcije je 0x00002010.

#### Napomena

Registri hi i lo su specijalni 32-bitni registri u koje se smješta rezultat instrukcija za cjelobrojno množenje i dijeljenje: mult, multu, div i divu.

Kod instrukcija mfhi i mflo (*move from HI* i *move from LO*), polje rd (*destination register*) predstavlja odredišni registar instrukcije, tj. registar opšte namjene u koji će se kopirati vrijednost iz registra hi odnosno lo.

Kod instrukcija mthi i mtlo (*move to HI* i *move to LO*), polje rs (*source register*) predstavlja izvorni registar instrukcije, tj. registar opšte namjene iz kojeg će se kopirati vrijednost u registar hi odnosno lo.

#### Primjer 8-3. Dekodiranje instrukcije R formata

U ovom primjeru je potrebno dekodirati mašinsku instrukciju datu u heksadecimalnom obliku:

0x00a01020

Prisjetimo se, proces određivanja mašinske instrukcije od asemblerске naziva se asembliranje. Dekodirati instrukciju znači odrediti asemblersku instrukciju od mašinske, što je obrnut proces od procesa asembliranja. Zbog toga se postupak dekodiranja mašinske instrukcije naziva još i *disasembliranje*.

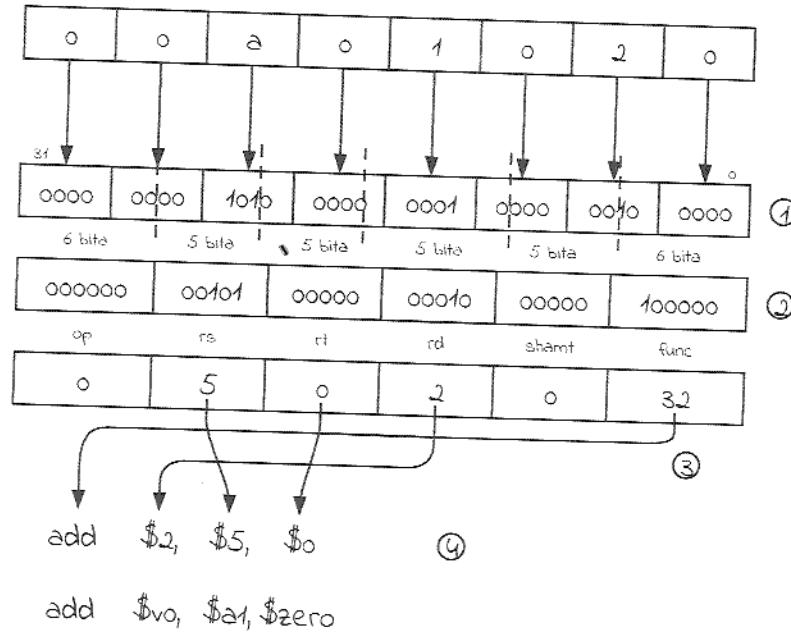
Proces disasembliranja instrukcije uključuje naredne korake:

1. Zapisati mašinsku instrukciju u binarnom obliku.
2. Grupisati bite unutar instrukcije na 6-5-5-5-6 bita. Ovako grupisana polja možemo posmatrati direktno u binarnom obliku, ali je u ovom trenutku praktičnije pretvoriti ih u decimalni brojni sistem i kao takve analizirati.
3. Na osnovu polja op i func odrediti asemblerски mnemonik. Za instrukcije R tipa možemo se poslužiti tabelom 8-1.
4. Zapisati vrijednosti polja koja su specifična za datu instrukciju. U slučaju instrukcija R formata, to su registri opšte namjene koji učestvuju u instrukciji i/ili iznos za šiftanje.

Postupak određivanja polja mašinske, odnosno asemblerске, instrukcije je detaljno ilustrovan na slici 8-6.

Na slici se mogu vidjeti svi koraci opisanog postupka.

U prvom koraku heksadecimalni kod mašinske instrukcije pretvaramo u binarni brojni sistem. To radimo tako što svaku heksadecimalnu cifru nezavisno zapisujemo kao binarnu tetradu. Dobili smo binarni ekvivalent heksadecimalnog broja kojim je zadana mašinska instrukcija.



Slika 8-6. Primjer — ilustracija disasembliranja mašinske instrukcije

Naredni korak je formiranje polja mašinske instrukcije R tipa grupisanjem bita (sjetimo se: 6-5-5-5-6). Dobili smo vrijednosti svih polja MIPS instrukcije:

- $op = 0 \Rightarrow$  radi se o instrukciji R tipa. Polje `func` nam govori o kojoj se operaciji radi.
- $func = 32 \Rightarrow$  radi se o instrukciji `add` (konsultovati tabelu 8-1). Ovu instrukciju bismo mogli opisati kao  $rd = rs + rt$ , odnosno zbir vrijednosti iz registara `rs` i `rt` (ulazni operandi instrukcije) smješta se u registar `rd` (rezultat instrukcije). Iz polja `rs`, `rt` i `rd` saznaćemo koji konkretno registri učestvuju u ovoj instrukciji.
- $rs = 5 \Rightarrow$  prvi operand operacije `add` je registar `$5`, odnosno registar `$a1`.
- $rt = 0 \Rightarrow$  drugi operand operacije `add` je registar `$0`, odnosno registar `$zero`.

- $rd = 2 \Rightarrow$  rezultat se smješta u registar \$2, odnosno registar \$v0.

Konačno, za instrukciju čiji je heksadecimalni kod mašinske instrukcije 0x00a01020, asemblerски ekvivalent je

add \$2, \$5, \$0

odnosno

add \$v0, \$a1, \$zero

## I format instrukcija

Kod ovog tipa MIPS instrukcija vrijednost jednog od operanada je sadržana u samoj instrukciji. U engleskoj terminologiji, ovakav operand je označen kao *immediate value* ili skraćeno *immediate*. Otuda naziv I za instrukcije koje se kodiraju na ovaj način.

U vezi sa načinima adresiranja, I format instrukcija se u MIPS procesoru koristi za instrukcije koje implementiraju neposredno adresiranje, bazno registarsko adresiranje i PC-relativno adresiranje.

Konkretno, to su:

- aritmetičke i logičke instrukcije koje imaju konstantu za jedan operand, kao npr. addi, ori i sl.
- load i store instrukcije, odnosno instrukcije za komunikaciju sa memorijom, kao npr. lw, sw i sl.
- instrukcije za uslovna grananja, kao npr. beq, bne i sl.

Format instrukcija I tipa slijedi format koji smo vidjeli na početku ovog poglavlja, s tim da je za konstantu koja se zadaje unutar same instrukcije rezervisano 16 bita s kraja 32-bitne riječi, odnosno od bita 15 do bita 0, kao što je ilustrovano na slici 8-7.

Tabela 8-1  
strukcije I f

Instrukcija

addi

addiu

andi

beq

bne

lb

lbu

lh

lhu

lui

lw

lwcl

ori

sb

slti

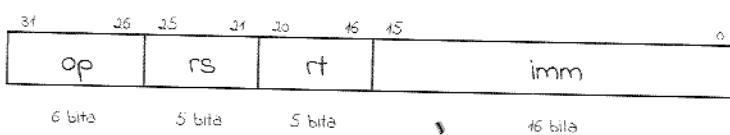
sliu

sh

sw

swcl

xori



Slika 8-7. I format MIPS instrukcije

Značenje pojedinih polja unutar 32-bitne instrukcijske riječi je:

- **op** - predstavlja instrukcijski kod. Za I tip instrukcija ovo polje unikatno identificuje instrukciju, za razliku od instrukcija R tipa, gdje je to određivalo func polje. U instrukcije I tipa spadaju sve instrukcije osim onih čija vrijednost op polja odgovara vrijednostima: 000000, 00001x i 0100xx. Instrukcijski kodovi instrukcija I formata su dati u tabeli 8-2.
- **rs** - ovo polje predstavlja izvorni registar za sve instrukcije I tipa, osim za load i store instrukcije. Kod load/store instrukcija ovo je bazni registar za određivanje memorijске adrese operanda korištenjem baznog registarskog adresiranja.
- **rt** - ovo polje predstavlja ciljni registar za sve instrukcije I tipa. U slučaju load/store instrukcija ovo je registar u/iz kojeg se upisuje/čita vrijednost (za load: registar u koji se upisuje vrijednost pročitana iz memorije; za store: registar iz kojeg se čita vrijednost koja se upisuje u memoriju).
- **imm** - ovo polje sadrži 16-bitnu konstantu koja je sadržana unutar same instrukcije. Konstanta može imati vrijednost od -32768 do 32767 ili od 0 do 65535, zavisno od instrukcije, odnosno od tretiranja bita najveće težinske vrijednosti u ovom polju. Značenje ove konstante za sve instrukcije I tipa nije isto, pa će to biti objašnjeno detaljnije u nastavku.

Tabela 8-2. Instrukcijski kodovi za neke instrukcije I formata

| Instrukcija | Korišteni operandi | op (bin) | op (dec) |
|-------------|--------------------|----------|----------|
| addi        | rt, rs, immediate  | a001000  | 8        |
| addiu       | rt, rs, immediate  | 001001   | 9        |
| andi        | rt, rs, immediate  | 001100   | 12       |
| beq         | rs, rt, label      | 000100   | 4        |
| bne         | rs, rt, label      | 000101   | 5        |
| lb          | rt, immediate(rs)  | 100000   | 32       |
| lbu         | rt, immediate(rs)  | 100100   | 36       |
| lh          | rt, immediate(rs)  | 100001   | 33       |
| lhu         | rt, immediate(rs)  | 100101   | 37       |
| lui         | rt, immediate      | 001111   | 15       |
| lw          | rt, immediate(rs)  | 100011   | 35       |
| lwc1        | rt, immediate(rs)  | 110001   | 49       |
| ori         | rt, rs, immediate  | 001101   | 13       |
| sb          | rt, immediate(rs)  | 101000   | 40       |
| slti        | rt, rs, immediate  | 001010   | 10       |
| sliu        | rt, rs, immediate  | 001011   | 11       |
| sh          | rt, immediate(rs)  | 101001   | 41       |
| sw          | rt, immediate(rs)  | 101011   | 43       |
| swc1        | rt, immediate(rs)  | 111001   | 57       |
| xori        | rt, rs, immediate  | 001110   | 14       |

## ARITMETIČKE I LOGIČKE INSTRUKCIJE I TIPO

Ove instrukcije u asemblerskom kodu možemo prepoznati po njihovom formatu:

```
op rt, rs, C
```

gdje su:

- op — asemblerski mnemonik instrukcije, odnosno aritmetičko-logičke operacije (addi, addiu, ori, lui, itd.),
- rt — odredišni registar, tj. registar u koji se smješta rezultat operacije
- rs — registar u kojem se nalazi prvi ulazni operand operacije,
- C — konstanta koja predstavlja drugi ulazni operand operacije.

Kod aritmetičkih i logičkih instrukcija I tipa, prvi ulazni operand je dat u registru rs (registarsko adresiranje) a polje imm predstavlja drugi ulazni operand adresiran neposrednim adresiranjem. Rezultat se smješta u registar naveden u polju rt. Izuzetak od ovog formata je instrukcija lui, kod koje se ne navodi registar rs jer se ne koristi.

Način izvođenja aritmetičkih i logičkih instrukcija I formata se ne razlikuje od izvršavanja odgovarajućih instrukcija R formata, osim u tome što je drugi operand naveden unutar same instrukcije i ograničen je na 16 bita.

### Primjer 8-4. Binarno kodiranje instrukcije ori

Kao ilustrativan primjer I formata ovakvih instrukcija, određimo binarni i heksadecimalni zapis instrukcije ori (*OR immediate*):

```
ori $v0, $a1, 8
```

Ova instrukcija ekvivalentna je instrukciji

`ori $2, $5, 8`

Ova instrukcija vrši binarnu operaciju ILI nad bitima operanda koji se nalazi u registru  $\$a1$  i 16-bitne konstante vrijednosti 8, koja je data u okviru instrukcije. Rezultat te operacije smješta se u register  $\$v0$ .

Odredićemo pojedinačna polja unutar binarnog zapisa instrukcije u skladu sa formatom instrukcija I tipa:

**Polje op** : vrijednost ovog polja ćemo uzeti iz tabele 8-2. Za instrukciju `ori` vrijednost ovog polja je  $13_{10}$ , odnosno 001101.

**Polje rs**: prvi ulazni operand operacije ILI se nalazi u registru  $\$a1$ . Brojčana oznaka ovog regista je  $\$5$ , pa će polje **rs** imati vrijednost  $5_{10}$ , odnosno 00101.

**Polje rt**: za instrukcije I tipa u ovom polju je dat ciljni registar, odnosno registar u koji se smješta rezultat operacije. U konkretnom slučaju je to registar  $\$v0$ , čija je brojčana oznaka  $\$2$ , pa će polje **rt** imati vrijednost  $2_{10}$ , odnosno 00010.

**Polje imm**: ovo 16-bitno polje sadrži konstantu koja predstavlja drugi ulazni operand za operaciju ILI. U konkretnom slučaju je to broj  $8_{10}$ , odnosno 0000000000001000.

Konačno, za instrukciju:

`ori $2, $8, 8`

možemo sklopiti njen binarni zapis, kako je to prikazano na slici 8-8.

Iz binarnog koda je sada lako odrediti heksadecimalni zapis instrukcije:

| op                                       | rs    | rt    | imm              |
|------------------------------------------|-------|-------|------------------|
| 0                                        | 5     | 2     | 8                |
| 31      26    25    24    20    16    15 |       |       | 0                |
| 001101                                   | 00101 | 00010 | 0000000000001000 |

6 bita      5 bita      5 bita      16 bita

Slika 8-8. Binarni zapis instrukcije ori \$v0, \$a1, 8

0011 0100 1010 0010 0000 0000 0000 1000  
 3      4      a      2      0      0      0      8

Dakle, heksadecimalni kod instrukcije je: 0x34a20008.

### 32-bitna konstanta kao operand

Vidjeli smo na koji način možemo kao operand koristiti 16-bitnu konstantu. Međutim, postavlja se pitanje kako u istom kontekstu koristiti 32-bitnu konstantu. Konkretno, ako bismo željeli da u 32-bitni registar \$s4 upišemo konstantu datu kao 32-bitni broj čiji je heksadecimalni ekvivalent 0xDEADBEEF, da li bismo mogli koristiti instrukciju

~~addi \$s4, \$0, 0xDEADBEEF~~

Odgovor je potvrđan ali pod uslovom da se koristi asembler koji će ovu instrukciju umjesto nas pretvoriti u dvije odgovarajuće mašinske instrukcije koje obavljaju taj posao. Naime, navedena instrukcija postoji kao pseudoinstrukcija i ta instrukcija nema svoj mašinski ekvivalent iz razloga što konstanta unutar mašinske instrukcije ne može biti veličine 32 bita već mora biti kodirana kao 16-bitni binarni broj. Dakle, 32-bitnu konstantu je nemoguće upisati u registar u jednom instrukcijskom ciklusu.

Da bi se 32-bitna konstanta upisala u registar potrebna su najmanje dva koraka. Postoji više načina na koji to možemo uraditi ali uobičajeno je da u prvom koraku upišemo gornjih (MSB) 16

bita konstante u gornjih 16 bita registra a u drugom koraku donjih (LSB) 16 bita te konstante u donjih 16 bita istog registra.

Za upisivanje 16-bitne konstante u gornjih 16 bita nekog registra koristi se instrukcija I tipa *lui* (*load upper immediate*) čiji je opšti oblik:

*lui rt, konstanta*

Ova instrukcija u gornjih 16 bita navedenog ciljnog registra *rt* upisuje 16-bitnu konstantu navedenu u instrukciji. Pri tome se donjih 16 bita registra *rt* postavlja na 0. Ovu zadnju činjenicu početnici često zaborave te je potrebno обратити dodatnu pažnju na то jer se instrukcija *lui* obično koristi u пару са још неком instrukcijom па је redoslijed izvršavanja tih instrukcija vrlo bitan.

Za konkretan primjer, upisivanje 32-bitne konstante 0xDEADBEEF у регистар \$s4 бисмо могли обавити у две инструкције:

*lui \$s4, 0xdead  
ori \$s4, \$s4, 0xbeef*

Prvom инструкцијом у горњих 16 бита регистра \$s4 убацијемо 16-bitну константу 0xDEAD. При томе се донђих 16 бита регистра \$s4 поставља на 0. Након ове инструкције стање регистра \$s4 ће бити 0xDEAD0000, без обзира која је vrijednost била у том регистру прије извршења инструкције. Друга инструкција врши бинарну операцију ILI над trenutnom vrijedношћу регистра \$s4 и константе 0xBEEF те резултат смјешта у регистар \$s4:

|             |  |
|-------------|--|
| DEAD0000    |  |
| OR 0000BEEF |  |
| -----       |  |
| DEADBEEF    |  |

Ефективно, спајају се горњих 16 бита регистра \$s4 и донђих 16 бита константе дате у инструкцији *ori*.

Vratimo se ponovo na instrukciju `lui`. Primijetimo da se, za razliku od instrukcije `ori`, u instrukciji `lui` navodi samo ciljni registar. Da li je ovo uredu i kako se kodira ova instrukcija? Naravno da je ovo uredu jer je svrha ove instrukcije da se jedan registar postavi na željenu vrijednost pa ova instrukcija uopšte ne zahtijeva drugi registar. Što se tiče binarnog kodiranja instrukcije, sjetimo se šta smo rekli na početku ovog poglavlja za polja koja se ne koriste u instrukciji: postavljaju se na 0.

Odredimo binarne kodove gornje dvije instrukcije!

Za instrukciju:

`lui $s4, 0xDEAD`

vrijednosti polja u ovoj instrukciji I formata će biti:

- **op:** za instrukciju `lui` kod operacije je  $15_{10}$ , odnosno 001111.
- **rs:** ovo polje se ne koristi pa se kodira kao 00000.
- **rt:** u ovo polje se upisuje brojčana oznaka registra koji se postavlja na željenu vrijednost. Brojčana oznaka registra `$s4` je `$20` pa se ovo polje postavlja na 10100.
- **imm:** ovo polje se postavlja na željenu konstantu. U našem slučaju je to `0xDEAD`, odnosno 1101 1110 1010 1101.

Konačno, binarni kod date instrukcije je:

001111 00000 10100 1101111010101101

U heksadecimalnom obliku je to:

0011 1100 0001 0100 1101 1110 1010 1101  
3 c 1 4 d e a d

Za instrukciju:

`ori $s4, $s4, 0xBEEF`

vrijednosti polja će biti:

- **op:** za instrukciju `ori` kod operacije je 1310, odnosno 001101.
- **rs:** prvi operand je u registru \$s4 čija je brojčana oznaka \$20 pa se ovo polje kodira kao 10100.
- **rt:** u ovo polje se upisuje brojčana oznaka ciljnog registra koji se postavlja na željenu vrijednost, dakle 10100.
- **imm:** ovo polje se postavlja na željenu konstantu. U našem slučaju je to 0xBEEF, odnosno 1011 1110 1110 1111.

Konačno, binarni kod ove instrukcije je

001110 10100 10100 101111011101111

a u heksadecimalnom obliku je:

0011 0110 1001 0100 1011 1110 1110 1111  
3 6 9 4 b e e f

Dakle, dio asemblerskog programa:

```
lui $s4, 0xdead
ori $s4, $s4, 0xbeef
```

se kodira u dvije mašinske instrukcije:

0x3c14dead  
0x3694beef

## LOAD I STORE INSTRUKCIJE

Kod load i store instrukcija koristi se bazno registarsko adresiranje za određivanje lokacije operanda u memoriji. Polje `imm` predstavlja pomjeraj u odnosu na baznu adresu koja se nalazi u registru navedenom u polju `rs`. Konačna memorijska adresa operanda se dobije kao suma vrijednosti koja se nalazi u baznom registru datog u polju `rs` i konstante date u polju `imm`. Polje `imm` se kod load i store instrukci-

da se, za raz-  
jni registar.  
Naravno da je  
postavi na  
drugi re-  
se šta smo  
te u instruk-

001111.

koji se pos-

\$s4 je \$20

našem slu-

ja tretira kao označeni 16-bitni broj što znači da se može adresirati memoriska lokacija ispred ili iza bazne adrese.

Binarno kodiranje load i store instrukcija pokažimo na primjeru instrukcije

`lw $v0, 8($t0)`

koja je ekvivalentna instrukciji

`lw $2, 8($5)`

Vrijednosti polja MIPS instrukcije će biti:

- **op:** za instrukciju `lw` kod operacije je  $35_{10}$ , odnosno 100011.
- **rs:** bazni registar, odnosno registar u kojem se nalazi bazna adresa, je  $\$t0$ . Brojčana oznaka registra  $\$t0$  je  $\$5$  pa se ovo polje kodira kao 00101.
- **rt:** u ovo polje se upisuje oznaka odredišnog registra, odnosno registra u koji će se upisati vrijednost pročitana iz memorije. U ovom primjeru to je registar  $\$v0$ , odnosno  $\$2$ , pa se ovo polje kodira kao 00010.
- **imm:** ovo polje predstavlja pomjeraj u odnosu na baznu adresu. U našem primjeru pomjeraj je 8 bajta pa se ovo polje kodira kao 0000000000001000.

Konačno, binarni kod date instrukcije je:

`100011 00101 00010 0000000000001000`

a u heksadecimalnom obliku je to:

`1000 1100 1010 0010 0000 0000 0000 1000  
8 c a 2 0 0 0 8`

## INSTRUKCIJE GRANANJA

U slučaju instrukcija grananja (branch), odnosno tzv. uslovnih skokova, polje **imm** služi za određivanje adrese naredne instrukcije korištenjem PC-relativnog adresiranja za slučaj da je uslov ispunjen. Ovo polje predstavlja pomjeraj koji se izražava u broju instrukcijskih riječi u odnosu na trenutnu vrijednost programskog brojača. Nova adresa se dobije kao zbir trenutne vrijednosti programskog brojača i konstante date u ovom polju pomnožene sa 4 jer je instrukcija veličine 4 bajta. Ova konstanta se tretira kao označeni 16-bitni broj.

Ovdje treba napomenuti da se u programskom brojaču nalazi adresa instrukcije koja se fizički nalazi neposredno nakon branch instrukcije koja se trenutno izvršava jer se programski brojač MIPS procesora inkrementira odmah nakon učitavanja instrukcije koja treba da se izvrši. S obzirom na to da se polje **imm** kod branch instrukcija tretira kao označeni 16-bitni broj, uslovno grananje može se vršiti na instrukcije koje su ili ispred ili nakon trenutne instrukcije. Dakle, u suštini, polje **imm** kod branch instrukcija govori koliko instrukcija unaprijed ili unazad treba skočiti.

### Primjer 8-5. Kodiranje instrukcija za grananje

Način kodiranja branch instrukcija je najbolje ilustrovati na primjeru. Neka je dat listing dijela MIPS asemblerorskog programa u kojem ćemo se skoncentrisati samo na branch instrukcije. Iz tog razloga su ostale instrukcije označene kao **i1, i2, i3, itd.**

```
...
L1: i1
 i2
 beq $t0, $a2, L2
 i3
 i4
L2: i5
 bne $t1, $a3, L1
 i6
...

```

I format instrukcija

U gornjem listingu su dvije instrukcije grananja. Prva instrukcija (`beq — branch if equal`) će u slučaju da su jednake vrijednosti u registrima `$t0` i `$a2`, izazvati promjenu programskog toka (skok) na asemblersku oznaku `L2`. Druga instrukcija u primjeru (`bne — branch if not equal`) izaziva uslovni skok na asemblersku oznaku `L1` u slučaju da vrijednosti u registrima `$t1` i `$a3` nisu jednake. Pretpostavimo da su ostale instrukcije u ovom primjeru aritmetičke ili logičke instrukcije i da nema uslovnih, niti bezuslovnih skokova. O kojim se konkretno instrukcijama radi, sa stanovišta analize kodiranja branch instrukcija, apsolutno je nebitno. Odredićemo binarni i heksadecimalni kod obje instrukcije grananja iz primjera.

Posmatrajmo instrukciju

```
beq $t0, $a2, L2
```

Odredićemo sva polja I formata instrukcije:

- `op`: za instrukciju `beq` kod operacije je  $4_{10}$ , odnosno 000100.
- `rs`: izvorni register, odnosno prvi operand operacije poređenja jednakosti, je `$t0` čija je brojčana oznaka `$5` pa se ovo polje kodira kao 00101.
- `rt`: ciljni register, odnosno register sa kojim se prvi register poredi je register `$a2`, odnosno `$6`, pa se ovo polje kodira kao 00110.
- `imm`: ovo polje predstavlja lokaciju instrukcije na koju se treba preusmjeriti programski tok. Izražava se u broju instrukcija u odnosu na trenutnu vrijednost programskog brojača PC. U trenutku izvršavanja ove instrukcije PC sadrži adresu naredne instrukcije, tj. adresu instrukcije `i3`. Polje `imm` će sadržavati udaljenost izraženu u broju instrukcija (instrukcijskih riječi) između asemblerske iznake `L2` i trenutne vrijednosti programskog brojača PC. U konkretnom primjeru, oznaka `L2` je *vije instrukcije* *alje* u odnosu na trenutnu vrijednost registra PC.

Dakle, vrijednost ovog polja će biti  $2_{10}$ , odnosno 0000000000000010.

Mašinski kod ove instrukcije u binarnom obliku je:

000100 00101 00110 0000000000000010

a u heksadecimalnom obliku je to:

0001 0000 1010 0110 0000 0000 0000 0010  
1 0 a 6 0 0 0 2

Sličan je postupak određivanja mašinskog koda druge instrukcije grananja iz primjera, s tim što trebamo voditi računa da se uslovni skok vrši unazad u odnosu na programski brojač:

bne \$t1, \$a3, L1

Odredićemo sva polja I formata i ove instrukcije:

- **op:** za instrukciju bne kod operacije je  $5_{10}$ , odnosno 000101.
- **rs:** izvorni registar, odnosno prvi operand operacije poređenja nejednakosti, je \$t1 čija je brojčana oznaka \$6 pa se ovo polje kodira kao 00110.
- **rt:** ciljni registar, odnosno registar sa kojim se prvi registar poredi je registar \$a3, odnosno \$7, pa se ovo polje kodira kao 00111.
- **imm:** s obzirom da je lokacija oznake L1 *seam* instrukcija *unazad* u odnosu na trenutnu vrijednost programskog brojača PC, koji trenutno pokazuje na instrukciju i6, vrijednost polja imm će biti -7. Ovaj broj trebamo napisati kao 16-bitni označeni binarni broj u formatu drugog komplementa: 111111111111001.

Pišemo mašinski kod ove instrukcije u binarnom obliku:

000101 00110 00111 111111111111001

Mašinski kod date instrukcije u heksadecimalnom obliku određujemo na poznati način:

```
0001 0100 1100 0111 1111 1111 1111 1001
1 4 c 7 f f f 9
```

#### Primjer 8-6. Disasembliranje instrukcija

Uradimo još jedan primjer, ovoga puta disasembliranje instrukcija. Potrebno je disasemblirati naredne dvije instrukcije:

```
0x02d83822
0xa6d5fff4
```

Odredićemo binarni zapis pa kod operacije op prve instrukcije:

```
hex: 0 2 d 8 3 8 2 2
bin: 0000 0010 1101 1000 0011 1000 0010 0010
op: 000000
```

Vidimo da je vrijednost koda operacije (polje op) 000000 što znači da je ova instrukcija R formata. Odredićemo sada sva polja ove instrukcije tako što binarne cifre grupišemo po 6-5-5-5-6 bita. Na osnovu funkc polja ćemo odrediti o kojoj se asemblerskoj instrukciji radi. Cijeli postupak je ilustrovan na slici 8-9.

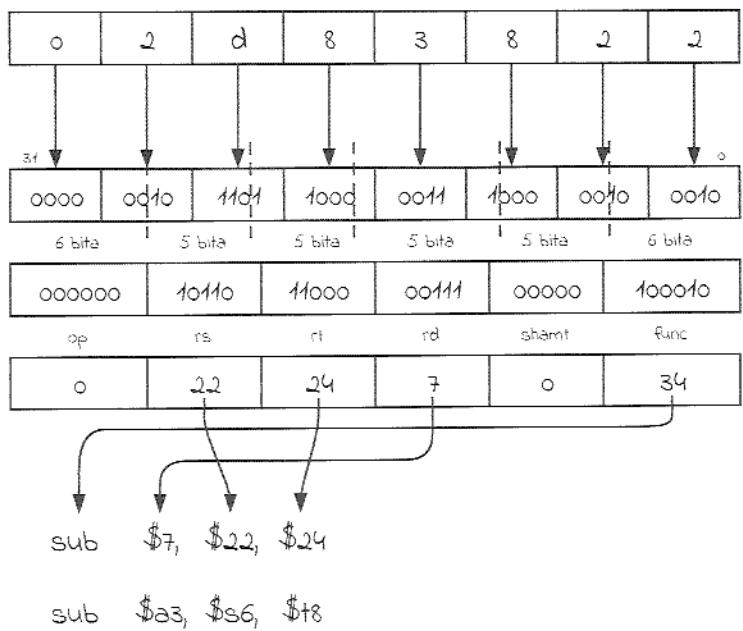
Dakle, mašinska instrukcija odgovara asemblerskoj instrukciji

```
sub $a3, $s6, $t8
```

Isti postupak ćemo obaviti i za drugu mašinsku instrukciju. Odredićemo binarni zapis pa kod operacije op:

```
hex: a 6 d 5 f f f 4
bin: 1010 0110 1101 0101 1111 1111 1111 0100
op: 101001
```

Ovoga puta vrijednost koda operacije (polje op) je 101001 što znači da je ovo instrukcija I formata. Odredićemo sva polja i ove instrukcije ali ovoga puta tako što binarne cifre grupišemo po 6-5-5-16



Slika 8-9. Ilustracija disasembliranja mašinske instrukcije R formata

bita. Kod instrukcija I formata polje op govori o kojoj se asembler-skoj instrukciji radi. Cijeli postupak je ilustrovan na slici 8-10.

Asemblererska instrukcija koja odgovara zadanoj mašinskoj instrukciji je

```
sh $s5, -10($s6)
```

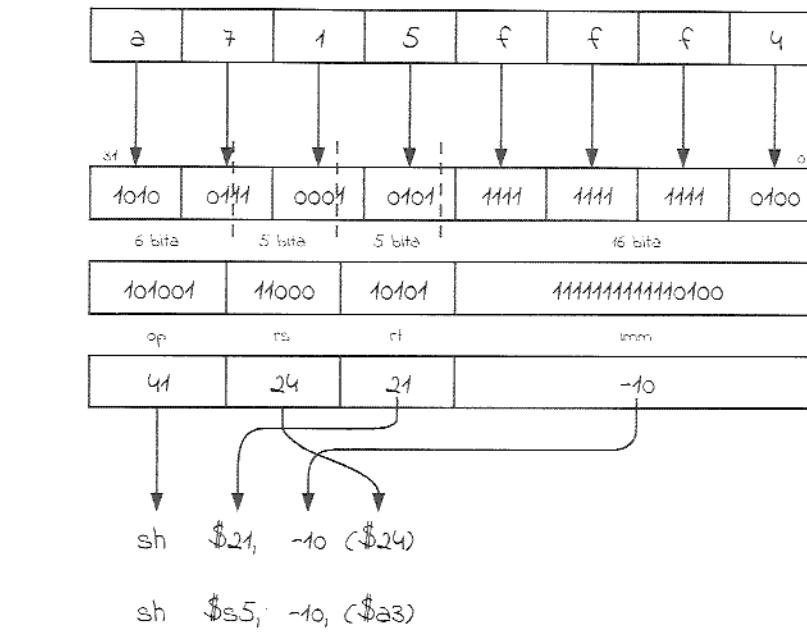
Konačno, zadani dio programa od dvije instrukcije u mašinskom kodu:

```
0x02d83822
0xa6d5fff4
```

odgovara dijelu programa u asemblerском kodu:

```
sub $a3, $s6, $t8
sh $s5, -10($a3)
```

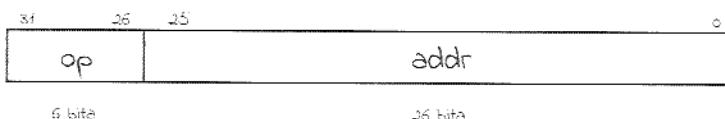
Polj  
 vršiti  
 u nast  
**KODIR**  
 Za za  
 trebn  
 ravne  
 jer in  
 nju :  
 $2^{22}$   
 svoj  
 ned



Slika 8-10. Ilustracija disasembliranja mašinske instrukcije I formata

### J format instrukcija

Ovaj format instrukcija se koristi za kodiranje mašinskih instrukcija bezuslovnih skokova na fiksnu adresu koja se zadaje u okviru instrukcije. U ovu skupinu instrukcija spadaju samo dvije instrukcije, *j* i *jal*. Mašinska instrukcija J formata se sastoji samo od dva polja, kao što je ilustrovano na slici 8-11.



Slika 8-11. J format MIPS instrukcije

Polje op: u ovom polju se nalazi 6-bitni instrukcijski kod. Za instrukciju *j* to je  $2_{10}$  (000010) a za *jal* je to  $3_{10}$  (000011).

Polje `addr`: u ovom polju je zadana ciljna adresa na koju će se izvršiti bezuslovni skok. Način kodiranja ovog polja će biti objašnjen u nastavku.

### KODIRANJE CILJNE ADRESE

Za zadavanje tačne adrese na koju će se izvršiti bezuslovni skok potrebno nam je 32 bita, kolika je veličina programskog brojača. Naučno, svih 32 bita iz 32-bitne instrukcije nemamo na raspolaganju jer instrukcijski kod (polje `op`) zauzima 6 bita. Dakle, na raspolaganju nam je 26 bita. Sa 26 bita bismo mogli adresirati samo prvih  $2^{27}$  bajta memorije, odnosno prvih 64 MB. Naši pradjedovi su se u svoje vrijeme možda i mogli zadovoljiti tim opsegom ali danas je to nedovoljno. Kako adresirati veći opseg memorije?

Razmislimo malo, MIPS instrukcije su veličine 32 bita, odnosno 4 bajta. Svaka instrukcija mora biti smještena na adresu koja je djeljiva sa 4. To znači da nam za adresiranje instrukcija (skokova) uopšte ne trebaju adrese koje nisu djeljive sa 4, a to su adrese čije su zadnje dvije cifre u binarnom zapisu 01, 10 i 11. Drugim riječima, korisne su nam samo adrese koje se završavaju na 00. Konstruktori MIPS procesora su ovu činjenicu iskoristili i izostavili zadnje dvije cifre iz kodiranja adrese unutar polja `addr` instrukcija J formata. Zapisuju se ostali biti a dvije zadnje nule se podrazumijevaju. Zapisuje se 26 bita koji u stvari označavaju broj od 28 bita. To je nešto slično kao izostavljanje prve jedinice u mantisi binarnih brojeva sa početnim zarezom (*floating point*). Efektivno, zadaje se redni broj instrukcije a ne njena adresa. Time smo dobili četiri puta veći opseg, tj. opseg od  $2^{29}$  bajta, odnosno 256 MB.

Međutim, možemo li adresirati više memorije jer je 256 MB i daleko pre malo? Možemo li iz ovih 26 bita izvući još nešto zajedničko pa uštedjeti još koji bit? Nažalost ne možemo, ali konstruktori MIPS procesora sa Stanford univerziteta su i ovdje bili domišljati. Pri generisanju ciljne adrese u instrukcijama J formata, 4 bita najveće težinske vrijednosti uzimaju se iz programskog brojača. Ovo je vrlo

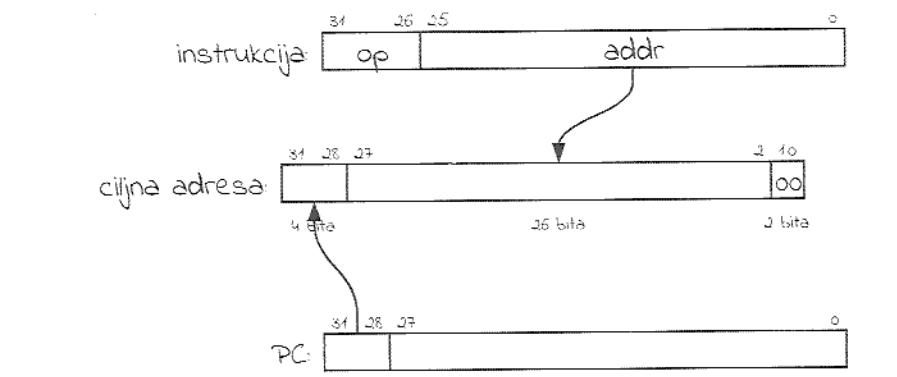
praktično jer se skokovi unutar programa uglavnom i vrše na lokacije koje su relativno blizu lokacije odakle se generiše skok. Prijetimo se, skokovi se u višim programskim jezicima, kao što je C, generišu naredbama za realizaciju programskih struktura selekcije (if, switch itd.) i iteracija (for, while itd.), te pozivima funkcija koje obično nisu daleko u memoriji od mjesta pozivanja jer se i lokacija funkcija može optimizirati.

Na ovaj način, ciljna adresa za instrukcije J formata se dobija spajanjem tri binarne sekvene:

$$[\text{ciljna adresa}] \Leftarrow \text{PC}_{31:28} :: \text{addr} :: 00$$

Kao što je ilustrovano i na slici 8-12, 32-bitna ciljna adresa se formira na sljedeći način:

- prvih 4 bita su biti 31 do 28 iz programskog brojača,
- narednih 26 bita predstavljaju vrijednost polja addr iz mašinske instrukcije J tipa,
- dva bita najmanje težinske vrijednosti su uvijek 00.



Slika 8-12. Određivanje ciljne adrese kod *jump* instrukcija

Jasno je da se na ovakav način mogu vršiti skokovi samo u okviru segmenta programske memorije veličine 256 MB koji odgovara trenutnoj vrijednosti programskog brojača. Ukoliko je potrebno izvr-

še na lokok. Pri-  
što je C,  
selekcije  
kcija koje  
i lokacija  
bija spa-  
a se for-  
mašin-  
okvиру  
ara tre-  
o izvr-  
nski kod

šiti skok na udaljeniju lokaciju, to nije moguće izvršiti u jednoj instrukciji. Međutim, moguće je izvršiti niz sukcesivnih skokova na način da se vrši skok na lokaciju na kojoj je ponovo neka instrukcija za uslovni ili bezuslovni skok i tako redom dok se ne dođe na željenu instrukciju.

Način kodiranja MIPS instrukcije J formata najbolje je shvatiti kroz primjer.

#### Primjer 8-7. Disasembliranje i tumačenje instrukcije J formata

Ako registar PC ima vrijednost 0x488000a0 u trenutku izvršanja instrukcije čiji je mašinski kod 0x0e000400, šta će biti efekat te instrukcije?

Analizirajmo datu mašinsku instrukciju! Najprije iz heksadeci-malnog zapisa odredimo binarni zapis i kod operacije (op):

|      |      |      |      |      |      |      |      |
|------|------|------|------|------|------|------|------|
| hex: | 0    | e    | 0    | 0    | 0    | 4    | 0    |
| bin: | 0000 | 1110 | 0000 | 0000 | 0000 | 0100 | 0000 |

op: 000011

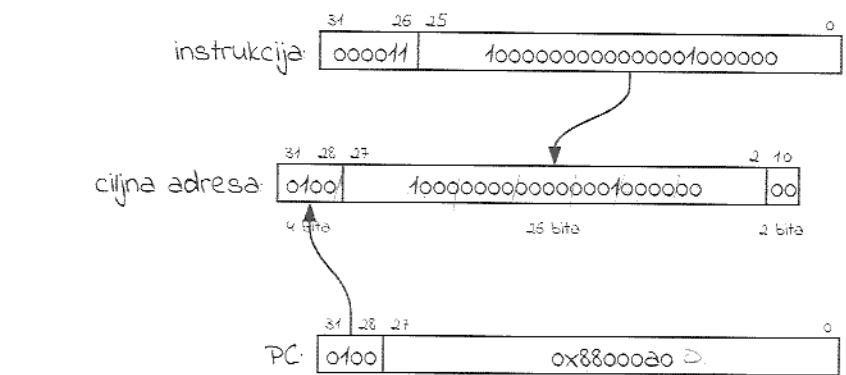
Instrukcijski kod je 000011, što predstavlja instrukciju J formata. Radi se o instrukciji jal. Polje addr ima vrijednost:

10000000000000001000000

Odredimo ciljnu adresu instrukcije jal:

- Prvih 4 bita ciljne adrese uzimamo iz registra PC: 0100.
- Za drugih 26 bita uzimamo polje addr, čija je vrijednost 10000000000000001000000.
- Zadnja dva bita su uvijek 00.

Ciljnu adresu dobijemo spajanjem ovih bita, kao što je ilustrova-no na slici 8-13.



Slika 8-13. Određivanje ciljne adrese instrukcije J formata

Odredimo adresu u heksadecimalnom formatu:

bin: 0100 1000 0000 0000 0001 0000 0000  
hex: 4 8 0 0 1 0 0

Dakle, bezuslovni skok će se izvršiti na adresu 0x4800100, odnosno programski brojač PC će dobiti vrijednost 0x4800100 prije narednog ciklusa učitavanja instrukcije iz memorije. S obzirom da se radi o instrukciji *jal* (*jump and link*), koja se koristi za pozivanje funkcija, prije nego registar PC poprimi ciljnu adresu, njegov sadržaj se kopira u registar \$ra da bi se sačuvala vrijednost adrese na koju se treba vratiti nakon izvršavanja funkcije. To znači da ova instrukcija u stvari radi *link and jump*, ali se ipak zove *jump and link*. To treba zapamtiti!

#### Napomena

Instrukcija *jal* se koristi za pozivanje funkcija. Vrijednost povratne adrese se pri tome sačuva u registru \$ra.

## Upravljanje projekta i uvezivanje

Uvezivanje predstavlja posljednji korak kojeg izvodi kompjuterski lanac, a sprovodi se putem komponente koja se označava kao linker. Cilj uvezivanja je kreiranje konačne ELF datoteke programa u kojoj se nalaze sve funkcije i podaci preuzeti iz pojedinačnih ELF objektnih fajlova koji su nastali u prethodno obavljenim koracima kompjuiranja i asembleriranja. Uvezivanje se kontrolira putem posebne skripte u kojoj linker pronađi neophodna uputstva za kreiranje sekcija u ELF datoteci programa, kao i određene adrese u memoriji na koje se kreirane sekcije postavljaju.

Moderne aplikacije koriste veliki broj funkcija i podatkovnih struktura definiranih u različitim bibliotekama i modulima koje su često sastavljene od nekoliko desetina datoteka sa C ili asembler kodom. Kompajliranje ovakvih projekata obično se upravlja putem posebnih alata kojima se reduciraju nepotrebna ponovna kompjuiranja ili asembleriranja pojedinačnih modula, te se automatizira cijelokupan proces kreiranja programa.

U ovom poglavlju razmatrat ćemo dva tipa skripti i to:

### **linker skripte**

Koje koristi linker za kreiranje programa na osnovu ulaznih objektnih fajlova.

### **Makefile skripte**

Koje koristi standardni alat za automatizaciju Unix projekata pod nazivom make.

J format instrukcija

## Linker

Prilikom predstavljanja osnova asemblera u poglavlju 3, asembliranje i uvezivanje obavljano je u jednom koraku jednostavnim pozivanjem Elk kompjajlera. Upotreba kompjajlerskog lanca na ovaj način od krajnjeg korisnika sakriva korak uvezivanja te prepostavlja da će kreirani program biti korišten pod unaprijed definiranim uslovima na tačno određenom operativnom sistemu, obzirom da se u sakrivenom koraku aplikacija uvezuje sa standardnim bibliotekama putem skripte koja je integrirana u linker.

Unaprijed definirana linker skripta dostaatna je za većinu programera koji se bave razvojem običnih korisničkih aplikacija za dati operativni sistem. Međutim, kompjajliranje programa za neku drugu platformu zahtijeva eksplicitno pozivanje svih komponenti u kompjajlerskom lancu, uključujući i linker. Kao primjer analizirat ćemo aplikaciju koja se kompjajlira za korištenje bez operativnog sistema, a koja se sastoji od sljedećih elemenata:

```
moj@komp$ cd ~/test/
moj@komp$ ls -l
total 16
-rwxr-x-- 1 amer amer 276 Aug 28 20:09 ams.c ①
-rwxr-x-- 1 amer amer 207 Aug 28 20:00 lib.c ②
-rw-rw-r-- 1 amer amer 104 Aug 28 20:00 lib.h ③
-rwxr-x-- 1 amer amer 346 Aug 28 20:35 link_script ④
-rwxr-x-- 1 amer amer 115 Aug 28 20:07 main.c ⑤
moj@komp$
```

- ① Modul aplikacije napisan u asembleru koji služi za inicijalizaciju BSS sekcije programa i pokretanje funkcije `main`.
- ② Biblioteka koju upotrebljava aplikacija, a koja je napisana u programskom jeziku C.
- ③ Zaglavljje biblioteke.
- ④ Linker skripta koja sadrži upute za uvezivanje programa.
- ⑤ Fajl u kojem se nalazi funkcija `main`.

Radi pojednostavljenja primjera uvest ćemo sljedeće prepostavke o okruženju u kojem će se izvršavati program:

1. MIPS procesor će biti konfiguriran da počne preuzimati instrukcije sa nulte adrese.
2. Memorijski modul sistema će imati minimalni kapacitet od 8KB.
3. Aplikacija će na neki način iz ELF fajla u kojem se distribuira biti učitana u memoriju na nultu adresu.

Analizu pojedinačnih modula aplikacije započinjemo od komponente `asm.s`, koja ima sljedeći kod:

```
.abicalls ①
.option pic0 ②
.text
.global boot
.set noreorder
.boot: ③
 la $t0, _bss_start ④
 la $t1, _bss_end ⑤
 la $sp, _sp ⑥
 .bss "clear": ⑦
 beq $t0, $t1, _start ⑧
 nop
 sb $0, 0($t0)
 j bss_clear
 addiu $t0, $t0, 1
_start:
 jal main ⑨
 nop
L:
 j L ⑩
 nop
.data
.globl br ⑪
br:
 .byte -25
```

Komponenta je kreirana tako da se počne izvršavati od linije ③ i to prije bilo kojeg drugog elementa aplikacije, a sa primarnim zadatkom da se stvori adekvatno okruženje za ostatak aplikacije koji je napisan u programskom jeziku C. Petlja definirana oznakom u liniji ⑦ inicijalizira .bss sekciju programa u memoriji nulama. Adrese

početka i kraja .bss sekcije određene su putem simbola `_bss_start` i `_bss_end`, čije su vrijednosti učitane u registre `t0` i `t1` u linijama ④ i ⑤. Vrh steka postavljen je u liniji ⑥ na adresu definiranu putem simbola `_sp`. Nakon što se inicijalizira kompletna sekcija .bss, iz linije ⑦ vrši se skok u liniju ⑧, gdje se poziva funkcija odredena simbolom `main`. Po povratku iz funkcije `main`, program ulazi u beskonačnu petlju u liniji ⑨.

Simboli `_bss_start`, `_bss_end`, `_sp` i `main`, obzirom da nisu definirani kao oznaće u komponenti, predstavljaju nerazjašnjene simbole koji moraju biti definirani u nekom drugom modulu koji će se uvezivati sa gornjom komponentom ili putem linkera. Komponenta definira dva globalna simbola u linijama ⑩ i ⑪ koji se mogu koristiti u drugim objektnim fajlovima sa kojim se komponenta uvezuje. Da bi se asembliranje obavljalo bez opcije generiranja koda neovisnog od lokacije učitavanja, korištene su asembler direktive ⑫ i ⑬.

Biblioteka definirana u fajlu `lib.c`, sastoji se od samo jedne funkcije:

```
// lib.c

#include "lib.h"
char* byteToString(int8_t num, char* str) {
 *(str+8) = '\0';
 uint8_t mask = 0x80;
 while (mask) {
 *str++ = !(mask & num)?'0';
 mask >>= 1;
 }
 return str;
}
```

Funkcija `byteToString` uzima 8-bitni broj `num`, a njegovu binarnu reprezentaciju u obliku niza karaktera zapisuje na lokaciju u memorijskoj sekciji koja je određena putem drugog parametra `str`.

Sama aplikacija definirana je u fajlu `main.c` na sljedeći način:

```
// main.c
```

```
#include "lib.h"
char byteToString(int8_t num, char* str) {
 extern int main();
 byteToChar(num, str);
 return main();
}

int main() {
 byteToString(0x80, &buf[8]);
 return 0;
}
```

Ko  
komma

moje  
moje  
BSP  
moje  
moje  
moje  
moje  
BSP  
moje  
> -

①  
②  
③

```

#include "lib.h"
char binbr[9]; ①
extern int8_t br; ②

int main() {
 byteToString(br,binbr); ③
 return 0;
}

```

Program se sastoji od samo jednog poziva funkcije `byteToString` u liniji ③, čime se konvertuje vrijednost globalne varijable `br` u njenu binarnu reprezentaciju u obliku niza karaktera `binbr`. Obzirom da je deklarirana sa ključnom riječi `extern` u liniji ②, varijabla `br` mora biti definirana u nekom drugom fajlu ovog projekta, u našem slučaju to je obavljeno putem globalnog simbola u `.data` sekciji fajla `asm.s`. Niz karaktera `binbr` definiran u liniji ③, obzirom da nije eksplicitno inicijaliziran, nakon uvezivanja završit će u `.bss` sekciji programa.

Kompajliranje projekta možemo sprovesti sljedećom sekvencom komandi:

```

moj@komp$ cd ~/test/
moj@komp$ ls
asm.s lib.c lib.h link_script main.c
moj@komp$ ecc -target mipsel-linux-eng -c main.c -o main.o ①
moj@komp$ ecc -target mipsel-linux-eng -c lib.c -o lib.o ②
moj@komp$ mips-elf-as -EL asm.s -o asm.o ③
moj@komp$ ls
asm.o asm.s lib.o lib.c lib.h link_script main.c main.o ④
moj@komp$ ecc-ld -m elf32elmp -o prog -T link_script \ ⑤
> -e boot main.o lib.o asm.o ⑥

```

- ① Kompajliranje i asembleriranje fajla `main.c`.
- ② Kompajliranje i asembleriranje fajla `lib.c`.
- ③ Asembleriranje fajla `asm.s` putem asemblera `mips-elf-as` i to sa opcijom `-EL` kojom se vrši asembleriranje za mali endian mod.

Nakon kompajliranja i asembleriranja dobiveni su pojedinačni objektni fajlovi prikazani u liniji ❶. Uvezivanje objektnih fajlova u program predstavljen datotekom prog, vrši se u linijama ❷ i ❸ eksplisitim pozivom linkera `gcc -ld`. Putem linker opcije `-m` specificirana je MIPS odredišna platforma i ELF izlazni format, a putem opcije `-T` linkeru se eksplisitno daje skripta putem koje će vršiti uvezivanje svih ulaznih objektnih fajlova pobrojanih u liniji ❹.

Korištena linker skripta `link_script`, ima sljedeći sadržaj:

```
SECTIONS
{
 _sp = 0x2000; ❶
 . = 0 ; ❷
 .text : ❸
 {
 asm.o(.text)
 *(.text)
 *(.rodata)
 }
 . = ALIGN(0x1000); ❹
 .data :
 {
 (.data)
 }
 _bss_start = . ; ❺
 .bss : ❻
 {
 *(.bss)
 *(.scommon)
 *(COMMON)
 }
 _bss_end = . ; ❻
 /DISCARD/ :
 {
 *(.MIPS.abiflags) *(.gnu*)
 *(.pdr) *(.reginfo)
 *(.eh_frame) *(.comment)
 }
}
```

Gornja skripta sadrži uputstva koja linker koristi pri procesiranju liste ulaznih objektnih fajlova. Blok `SECTIONS` definira sve sekcije koje će biti prisutne u izlaznom ELF fajlu programa. Pojedinačna sekcija definira se u formatu:

```
.ime_sekcije : {
 sadrzaj sekcije
}
```

Na osnovu linija **3**, **5** i **7**, program će se sastojati od sekcija porедanih u sljedećem redoslijedu: .text, .data i .bss. Sadržaj svake sekcije izlaznog fajla nastaje na osnovu sadržaja sekcija pojedinačnih ulaznih objektnih fajlova. Na primjer, sadržaj sekcije .text u izlaznom fajlu definiran je izrazom:

```
asm.o(.text)
*(.text)
*(.rodata)
```

Gornje linije interpretiraju se na sljedeći način. Simbol \* ispred zagrada označava bilo koji objektni fajl koji je kao ulaz dat linkeru. Unutar zagrade se imenuju sekcije iz fajla čiji se sadržaj preuzima. Tokom kreiranja sadržaja sekcije .text izlaznog fajla, iz svakog ulaznog objektnog fajla preuzimat će se sadržaji sekcija .text i .rodata. Obzirom na postavljeni redoslijed, izlazna sekcija .text prvo će se popuniti sadržajem sekcije .text iz fajla asm.o, nakon čega će se redati sadržaji sekcija .text iz svih ostalih ulaznih fajlova. Proces će se ponoviti sa sadržajima svih pojedinačnih .rodata sekcija koji će se umetnuti na kraj sekcije .text izlaznog fajla. Redoslijed procesiranja pojedinačnih objektnih fajlova putem operatora \* usklađen je sa redoslijedom ulaznih fajlova koji je dat prilikom pozivanja linkera.

Simbol . u linker skripti je brojač koji predstavlja memorijsku lokaciju na koju će se dodavati sljedeći sadržaj. U liniji **2** brojač je eksplicitno postavljen da počinje od nulte memorijske adrese. Pri svakom dodavanju sadržaja iz određenog objektnog fajla, brojač se inkrementira za vrijednost koja predstavlja količinu bajta koja je umetnuta u izlazni fajl. Linker pri uvezivanju modificira sve instrukcije i simbole preuzete iz ulaznih objektnih fajlova spram njihove nove lokacije u izlaznom fajlu koja je određena sa aktuelnom vrijednošću brojača. Prilikom eksplicitnog postavljanja vrijednosti

brojača moguće je koristiti linker direktivu ALIGN kojom se nova vrijednost za brojač poravnava na najbližu adresu koja je djeljiva sa nekim brojem. Na primjer, putem direktive ALIGN u liniji ❸ sekcija .data izlaznog fajla uvezuje se na prvu adresu iznad sadržaja sekcije .text koja je djeljiva brojem 4096.

Linker putem skripte može definirati vrijednosti za određene simbole korištene u objektnim fajlovima koji se uvezuju. Tako je u liniji ❶ simbol \_sp postavljen na vrijednost 8192, dok su u linijama ❷ i ❸ vrijednosti simbola \_bss\_start i \_bss\_end definirane spram lokacija početka i kraja sekcije .bss izlaznog fajla. U liniji ❹ putem posebne sekcije DISCARD sadržaji pobrojanih sekcija ulaznih objektnih fajlova eksplisitno se izostavljaju u izlaznom fajlu.

Linker tokom uvezivanja formira izlazni fajl koji je sastavljen od sekcija koje su definirane u linker skripti. Sa druge strane, linker tokom uvezivanja u izlaznom fajlu bilježi i informacije o programskim segmentima. Segment predstavlja jedan kontinuirani blok u memoriji čiji se sadržaj definira putem sekcija ELF fajla, a formira se sa prvenstvenom namjerom da se pojednostavi proces učitavanja programa u memoriju. Za svaki segment unutar izvršne datoteke obezbjeduju se sljedeće informacije koje su relevantne za učitavanje datog segmenta u memoriju:

- lokacija početka i količine sadržaja segmenta unutar ELF fajla
- lokacija gdje segment očekuje da bude učitan u memoriju
- veličina segmenta u memoriji.

Na samom početku formiranog ELF fajla programa nalazi se isti string (magic string) koji je prisutan u svim ELF fajlovima. String je sastavni dio ELF zaglavljia (header) koji ima fiksnu veličinu. Unutar zaglavljia mogu se naći osnovne informacije o izvršnoj datoteci kao i lokacije u fajlu od kojih počinju druga zaglavljia koja opisuju pojedinačne programske segmente. Putem programa `readelf` moguće je pročitati razne informacije pohranjene unutar ELF fajlova. Na pri-

mjer, apliciranjem komande `ecc-readelf -h prog` dobijamo sljedeće informacije iz zaglavlja ELF datoteke programa iz primjera.

```
moj@komp$ ecc-readelf -h prog
ELF Header:
 Magic: 7F 45 4C 46 01 01 01 00 00 00 00 00 00 00 00 00 ①
 Class: ELF32 ②
 Data: 2's complement, little endian
 Version: 1 (current)
 OS/ABI: UNIX - System V
 ABI Version: 1
 Type: EXEC (Executable file) ③
 Machine: MIPS R3000 ④
 Version: 9xt
 Entry point address: ⑤
 Start of program headers: ⑥
 Start of section headers: ⑦
 Flags: 32 (bytes)
 Size of this header: 32 (bytes)
 Size of program headers: 3 ⑧
 Number of program headers: 40 (bytes)
 Size of section headers: 7
 Number of section headers: 4
 Section header string table index: 4
```

- ① String na samom početku svake ELF datoteke.
- ② Forma izvršne datoteke.
- ③ Procesorska arhitektura za koju je program kompajliran.
- ④ Lokacija od koje počinje prva instrukcija u programu.
- ⑤ Lokacija unutar ELF fajla izražena u bajtima od koje počinju zaglavljia programskih segmenata.
- ⑥ Veličina pojedinačnog programskog zaglavlja.
- ⑦ Broj programskih segmenata unutar ELF fajla.

Na osnovu informacije u liniji ⑦ zaključujemo da program ima tri segmenta koja trebaju biti učitana u memoriju da bi program bio spremjan za izvršenje. Dodatne informacije o ovim segmentima mogu se dobiti čitajući njihova zaglavljia na osnovu informacija datih u linijama ⑤ i ⑥. Program `readelf` putem opcije `-l` može ispisati sadržaj

žaje zaglavljaju svih raspoloživih programskih segmenata iz ELF izvršnih datoteka:

```
moj@komp$ ecc-readelf -l prog
Elf file type is EXEC (Executable file)
Entry point 0x8
There are 3 program headers, starting at offset 52

Program Headers:
Type Offset VirtAddr PhysAddr FileSiz MemSiz
LOAD 0x001000 0x00000000 0x00000000 0x00128 0x00128 ①
LOAD 0x002000 0x000001000 0x000001000 0x00001 0x00002 ②
GNU_STACK 0x0000000 0x00000000 0x00000000 0x0000000 0x0000000 ③

Section to Segment mapping: ④
Segment Sections...
00 .text ⑤
01 .data .bss ⑥
02 ⑦
```

Ispis iz programa `readelf` u ovom slučaju podijeljen je na dva dijela. Prvi dio ispisa u linijama ①, ② i ③ sadrži informacije relevantne za učitavanje raspoloživih programskih segmenata u memoriju. Drugi dio ispisa nakon linije ④ prikazuje raspored sekcija ELF fajla unutar programskih segmenata. Na osnovu linija ⑤, ⑥ i ⑦ zaključujemo da se sekcija `.text` nalazi u prvom segmentu, sekcije `.data` i `.bss` u drugom segmentu, dok je treći segment programa prazan.

Polja `PhysAddr` i `MemSiz` u zaglavljima programskog segmenta određuju adresu u memoriji na koju treba učitati segment i količinu memorije koja će se zauzeti nakon učitavanja segmenta. Kao što je prikazano u linijama ① i ②, prvi segment učitava se na nultu adresu, dok se drugi segment učitava na adresu `0x1000` zbog poravnanja početne adrese `.data` sekcijske koje je dano u linker skripti. Polja `Offset` i `FileSiz` u zaglavljima programskog segmenta sadrže informacije o lokaciji unutar ELF fajla na kojoj počinje sadržaj segmenta i veličini segmenta u bajtima. Linija ② prikazuje da drugi segment ELF fajla ima veličinu od 1B, a u memoriji njegova veličina je 10B. Ovo je posljedica činjenice da se drugi segment sastoji od sekcija `.data` i `.bss`, pri čemu se u `.data` sekciji nalazi varijabla `br` za koju je u fajlu deklaracija:

lu `asm.s` odvojen jedan bajt memorije, dok je u `.bss` sekciji smješten niz `bin\br` za kojeg je u fajlu `main.c` odvojen prostor od devet karaktera. Niz `bin\br` prilikom definiranja nije inicijaliziran, te se za njega unutar ELF fajla ne odvaja prostor, a pri učitavanju programa u memoriju svi elementi niza inicijaliziraju se nulama, obzirom da se sadržaj niza nalazi u sekciji `.bss`.

## Makefile skripte

Kreiranje programa koji je kao primjer korišten u prethodnoj sekciji zahtijeva kompajliranje, asembleriranje i uvezivanje tri pojedinačne komponente. Izmjena sadržaja neke komponente zatijeva unos komandi za njeno procesiranje, nakon čega slijedi ponovno uvezivanje nove verzije programa. Ručni unos komandi nije praktičan čak ni za situaciju u kojoj se projekat sastoji od tri komponente, a postaje nemoguće za realistične projekte koji često uključuju stotine programskih datoteka i veliki broj biblioteka. Zbog toga se uvode alati za upravljanje softverskim projektima, od kojih je najčešće u upotrebi programski alat `make`.

Da bi mogao upravljati nekim projektom, `make` očekuje skriptu pod imenom `Makefile` snimljenu u direktorij projekta u kojem se nalaze programske datoteke. Skripta je običan fajl tekstualnog formata u kojem se nalaze pravila za procesiranje komponenti projekta. Pravila su u sljedećem formatu:

```
cilj: preduslovi
 recept
```

Gdje:

- `cilj` — može biti nova akcija koja se definira ili ime fajla koji treba kreirati.
- `preduslovi` — predstavlja listu fajlova ili akcija koje treba preduzeti kao preduslov za procesiranje cilja.

- **recept** — lista konkretnih koraka koji se obavljaju u okviru pravila.

Slijedi jednostavna Makefile skripta koja za primjer iz prethodne sekcije ima dva pravila:

```
Makefile

prog: main.c lib.c asm.s ①
 gcc -target mipsel-linux-eng -c main.c -o main.o
 gcc -target mipsel-linux-eng -c lib.c -o lib.o
 mips-elf-as -EL asm.s -o asm.o
 gcc-ld -m elf32elmp -o prog -T link_script \
 -e boot main.o lib.o asm.o

clean: ②
 rm -f *.o
 rm -f prog
```

Pravilo kao cilj postavlja formiranje datoteke `prog` u trenutnom direktoriju. Preduslov za izvršavanje recepta definiranog

- ❶ pravila je prisustvo datoteka `main.c`, `lib.c` i `asm.s`. Recept pravila je sekvenca komandi koja je ručno unošena u prethodnoj sekciji.

Pravilo definira akciju `clean` koja nema preduslova. Recept

- ❷ pravila je sekvenca od dvije komande kojim se iz trenutnog direktorija brišu sve datoteke sa ekstenzijom `.o` i fajl `prog`.

Za izvođenje komandi koje čine recept određenog pravila potrebno je pozvati program `make` skupa sa imenom pravila. Na primjer:

```
moj@komp$ cd ~/test
moj@komp$ make prog ①
gcc -target mipsel-linux-eng -c main.c -o main.o
gcc -target mipsel-linux-eng -c lib.c -o lib.o
mips-elf-as -EL asm.s -o asm.o
gcc-ld -m elf32elmp -o prog -T link_script \
 -e boot main.o lib.o asm.o
moj@komp$ make prog ②
make: 'prog' is up to date.
moj@komp$ make clean ③
```

```
rm -f *.o
rm -f prog
```

Nakon što se u liniji ❶ pozove program `make` sa pravilom `prog`, izvodi se recept koji rezultira programskom datotekom `prog` u trenutnom direktoriju. Ponovno pokretanje istog pravila u liniji ❷ ne rezultira ponovnim pokretanjem komandi koje čine recept, obzirom da je `make` konstatovao da je datoteka `prog` već kreirana, a datoteke koje čine preduslov za recept `prog` u međuvremenu nisu mijenjane. Pozivom akcije `clean` u liniji ❸ skupa sa datotekom `prog` brišu se svi objektni fajlovi u trenutnom direktoriju.

Uvođenjem `Makefile` skripte proces kompajliranja je automatiziran, a ponovno kompajliranje neće se moći pokrenuti sve dok se ne promijeni sadržaj makar jedne datoteke koja je sastavni dio preduslova pravila `prog`. Međutim, skripta ima značajan nedostatak. Promjena sadržaja jednog od fajlova koji čine preduslov pri ponovnom pokretanju pravila `prog` pokretat će proces ponovnog kompajliranja svih komponenti programa, bez obzira što je promijenjen sadržaj samo jedne komponente. Promjenu sadržaja neke datoteke možemo simulirati putem programa `touch` koji u fajl sistemu mijenja vremenski pečat zadnje izmjene ulazne datoteke spram trenutka u kojem je program pozvan. Na primjer:

```
moj@komp$ cd ~/test/
moj@komp$ make clean
rm -f *.o
rm -f prog
moj@komp$ make prog ❶
gcc -target mipsel-linux-eng -c main.c -o main.o
gcc -target mipsel-linux-eng -c lib.c -o lib.o
mips-elf-as -EL asm.s -o asm.o
gcc-ld -m elf32elmp -o prog -T link_script \
-e boot main.o lib.o asm.o
moj@komp$ make prog ❷
make: 'prog' is up to date.
moj@komp$ touch lib.c ❸
moj@komp$ make prog ❹
gcc -target mipsel-linux-eng -c main.c -o main.o ❺
gcc -target mipsel-linux-eng -c lib.c -o lib.o
mips-elf-as -EL asm.s -o asm.o ❻
gcc-ld -m elf32elmp -o prog -T link_script \
-e boot main.o lib.o asm.o
```

Nakon kompajliranja programa u liniji ❶, pravilo prog je nanovo pokrenuto u linijama ❷ i ❸. Pri pokretanju pravila u liniji ❷, komande koje čine recept pravila nisu ponovo izvršene obzirom da sadržaji ulaznih datoteka nisu u međuvremenu mijenjani. Nakon što je u liniji ❹ simulirana izmjena sadržaja datoteke lib.c, ponovno pokretanje pravila prog u liniji ❺ rezultira izvršavanjem svih komandi koje čine recept pravila, uključujući i komande u linijama ❻ i ❼ u kojim se procesiraju fajlovi koji nisu mijenjani. Neselektivno kompajliranje fajlova koje se sprovodi u skripti postaje značajan problem ukoliko se broj ulaznih datoteka u procesu kompajliranja poveća.

Da bismo riješili navedeni problem, u direktoriju projekta definiramo novu skriptu pod nazivom Makefile1 koja ima sljedeći sadržaj:

```
Makefile1

prog: main.o lib.o asm.o ❶
 gcc -ld -m elf32elmp -o prog -T link_script\
 -e boot main.o lib.o asm.o

main.o: main.c ❷
 gcc -target mipsel-linux-eng -c main.c -o main.o

lib.o: lib.c ❸
 gcc -target mipsel-linux-eng -c lib.c -o lib.o

asm.o: asm.s ❹
 mips-elf-as -EL asm.s -o asm.o

clean:
 rm -f *.o
 rm -f prog
```

Skripta je izmjenjena u odnosu na originalnu na način da su dodata nova pravila u linijama ❷, ❸ i ❹, dok je pravilo prog u liniji ❶ izmjenjeno. Da bi se kreirao program spram pravila u liniji ❶ potrebno je da u direktoriju projekta postoje objektni fajlovi main.o, lib.o i asm.o. Pravila za kreiranje pojedinačnih objektnih fajlova koji zahtijevaju kompajliranje i asembliranje izvornih C datoteka data su u linijama ❷ i ❸, dok je u liniji ❹ dato pravilo za nastanak

objektnog fajla na osnovu asembler datoteke `asm.s`. Ovim je efektivno razdvojeno procesiranje pojedinačnih modula u projektu, dok je u receptu pravila za nastanak programa, u liniji ❶, ostala samo komanda za uvezivanje. Ukoliko korištenjem skripte `Makefile1` ponovimo komande iz prethodnog primjera, dobijamo sljedeći rezultat:

```
moj@komp$ cd ~/test/
moj@komp$ make -f Makefile1 clean
rm -f *.o
rm -f prog
moj@komp$ make -f Makefile1 prog
gcc -target mipsel-linux-eng -c main.c -o main.o
gcc -target mipsel-linux-eng -c lib.c -o lib.o
mips-elf-ld -EL asm.s -o asm.o
gcc-ld -m elf32elmp -o prog -T link_script \
-e boot main.o lib.o asm.o
moj@komp$ make -f Makefile1 prog
make: [prog] is up to date.
moj@komp$ touch lib.c
moj@komp$ make -f Makefile1 prog ❶
gcc -target mipsel-linux-eng -c lib.c -o lib.o ❷
mips-elf-ld -m elf32elmp -o prog -T link_script ❸
-e boot main.o lib.o asm.o
```

Pokretanje pravila `prog` u liniji ❶, nakon simulirane izmjene sadržaja datoteke `lib.c`, ovaj put dovodi do ponavljanja samo neophodnih komandi kojim se u liniji ❷ kompajlira i asemblira izmijenjena datoteka, nakon čega se u liniji ❸ vrši ponovno uvezivanje ELF datoteke programa. U prethodnom primjeru postoji izmjena i u načinu upotrebe programa `make`, obzirom da se koriste pravila iz skripte koja nije nazvana `Makefile`, što zahtijeva upotrebu komandne opcije `-f` skupa sa imenom datoteke u kojoj se nalaze nova pravila za program `make`.

Skripta `Makefile1` značajno unapređuje performanse kompajliranja cjelokupnog projekta. Međutim, sa stanovišta fleksibilnosti skriptu je moguće unaprijediti upotrebom varijabli kao što je prikazano u sljedećoj verziji skripte:

```
Makefile2
CFLAGS = -target mipsel-linux-eng ❶
LDFLAGS = -m elf32elmp -T link_script -e boot
```

```

ASFLAGS = -EL
CC = ecc
AS = mips-elf-as
LD = ecc-ld ②

prog: main.o lib.o asm.o
 $(LD) $(LDFLAGS) -o prog main.o lib.o asm.o ③

main.o: main.c
 $(CC) $(CFLAGS) -c main.c -o main.o ④

lib.o: lib.c
 $(CC) $(CFLAGS) -c lib.c -o lib.o ⑤

asm.o: asm.s
 $(AS) $(ASFLAGS) asm.s -o asm.o ⑥

clean:
 rm -f *.o
 rm -f prog

```

Sa stanovišta funkcionalnosti nova verzija skripte identična je prethodnoj. Međutim, putem operatora `=` u linijama ① do ② definirane su varijable čije se vrijednosti u linijama ③, ④, ⑤ i ⑥, putem operatora `$()` koriste za formiranje komandi u odgovarajućim pravilima skripte. Na ovaj način, eventualne izmjene opcija za kompajliranje projekta centralizirane su na jednoj lokaciji u `make` skripti. Na primjer, ukoliko bi se u projektu tražila izmjena kompjajlera, dovoljno bi bilo promijeniti definiciju varijable `CC` te postaviti odgovarajuće opcije za novi kompjajler promjenom vrijednosti varijable `CFLAGS`.

Skriptu dalje možemo izmijeniti uvođenjem automatskih varijabli u receptima pravila, kao u sljedećoj varijanti skripte:

```

Makefile3

CFLAGS = -target mipsel-linux-eng
LDFLAGS = -m elf32elmp -T link_script -e boot
ASFLAGS = -EL
CC = ecc
AS = mips-elf-as
LD = ecc-ld

prog: main.o lib.o asm.o
 $(LD) $(LDFLAGS) -o $@ $? ①

main.o: main.c
 $(CC) $(CFLAGS) -c $< -o $@ ②

```

```

lib.o: lib.c
 $(CC) $(CFLAGS) -c $< -o $@ ③

asm.o: asm.s
 $(AS) $(ASFLAGS) $< -o $@ ④

clean:
 rm -f *.o
 rm -f prog

```

Automatske varijable korištene u linijama ①, ②, ③ i ④ imaju sljedeće značenje:

- \$@ — Predstavlja ime cilja aktuelnog pravila, npr. u liniji ③ vrijednost varijable je `prog`, dok u liniji ④ varijabla ima vrijednost `asm.o`.
- \$< — Predstavlja ime prvog preduslova u aktuelnom pravilu, npr. u liniji ① vrijednost varijable je `main.o`, dok je u liniji ④ vrijednost varijable `asm.s`.
- \$? — Predstavlja listu svih preduslova u aktuelnom pravilu, npr. u liniji ① vrijednost varijable je `main.o lib.o asmo.o`.

Inspekcionim linija ② i ③ primjećujemo ponavljanje istog recepta u dva korištena pravila. Ponavljanje se može reducirati upotrebom pravila baziranih na uzorcima, kao u sljedećoj verziji skripte:

```

Makefile4

CFLAGS = -target mipsel-linux-eng
LDFLAGS = -m elf32elmp -T link_script -e boot
ASFLAGS = -EL
CC = ecc
AS = mips-elf-as
LD = ecc-ld

OBJS = main.o lib.o asmo.o ①

%.o: %.c ②
 $(CC) $(CFLAGS) -c $< -o $@

%.o: %.s ③
 $(AS) $(ASFLAGS) $< -o $@

prog: $(OBJS) ④

```

```
$(LD) $(LDFLAGS) -o $@ $?
clean:
rm -f *.o
rm -f prog
```

Pravila uvedena u linijama ❷ i ❸ sadrže recepte za kreiranje bilo kojeg objektnog fajla u trenutnom direktoriju, pri čemu se recept u liniji ❶ aplicira na fajlove sa ekstenzijom .c, dok se recept u liniji ❷ aplicira na fajlove sa ekstenzijom .s. Varijabla OBJS definirana u liniji ❶ sadrži listu objektnih fajlova koji su neophodni da bi se, na osnovu pravila u liniji ❷, proizveo program prog. Traženi fajlovi main.o i lib.o proizvest će se na osnovu pravila ❷, obzirom da trenutni direktorij sadrži datoteke main.c i lib.c, dok će se za kreiranje fajla asm.o koristiti pravilo u liniji ❸, jer trenutni direktorij sadrži datoteku asm.s. Eventualno dodavanje novog modula napisanog u C-u ili asembleru zahtijevalo bi samo modifikaciju varijable OBJS spram imena objektnog fajla od nove komponente.

Pravila u linijama ❷ i ❸ spadaju u grupu implicitnih pravila koja su unaprijed ugradena u program make, tako da ih je moguće izostaviti. Uzimajući u obzir navedeno, konačna verzija skripte uz zadržavanje iste funkcionalnosti ima sljedeći reducirani sadržaj:

```
Makefile
CFLAGS = -target mipsel-linux-eng
LDFLAGS = -m elf32elmp -T link_script -e boot
ASFLAGS = -EL
CC = ecc
AS = mips-elf-as
LD = ecc-ld

OBJS = main.o lib.o asm.o

prog: $(OBJS)
 $(LD) $(LDFLAGS) -o $@ $?

clean:
rm -f *.o
rm -f prog
```