

Arhitektura računara 1

skripta za vežbe sa zadacima

Oktoibar 2023

Sadržaj

1	Uvod	1
1.1	Registri	2
1.2	Prosledjivanje argumenata	3
1.3	GDB debugger	5
1.4	Hello world program	5
1.4.1	Intelova sintaksa	6
1.4.2	Sekcija sa podacima	6
1.4.3	Tekst i global	7
1.4.4	Enter, leave i ret	8
1.4.5	Instrukcije mov i lea	11
1.5	Osnovne aritmetičke operacije	12
1.5.1	Sabiranje i oduzimanje	12
1.5.2	Množenje i deljenje	13
1.5.3	Potencijalni problemi	14
1.6	Nizovi	15
1.6.1	Nizovi i množenje - hack	16

1 Uvod

U ovoj skripti akcenat će biti na 64-bitnoj varijanti x86-arhitekture, x64 i operativnom sistemu Linux.

x64 je generičko ime za 64-bitnu ekstenziju Intelovih i AMD-ovih skupova instrukcija 32-bitne x86-arhitekture. AMD je predstavio prvu verziju x64, inicijalno nazvanu x86-64, a kasnije su je preimenovali u AMD64. Intel je svoju implementaciju nazvao IA-32e, a potom EMT64. Postoje neznatne razlike između ove dve verzije, ali većina koda radi dobro u obe.

Asembler je bilo koji programski jezik niskog nivoa sa jakim vezom između instrukcija u samom jeziku i instrukcija mašinskog jezika koje odgovaraju samoj arhitekturi. Svaki assembler ima različitu sintaksu. Čak i između assemblera

koji imaju "Intelovu sintaksu" odnosno koji su kompatibilni sa Intelovim procesorima, postoji razlika. Na primer, za rad sa x86 procesorima možemo da koristimo sledeće asemblere: MASM, NASM, YASM, itd.

GNU assembler, poznatiji kao gas ili as je assembler razvijan od strane GNU projekta. On je podrazumevani "deo u pozadini" GCC kompajlera. Takođe je i cross-platform, što znači da može da se izvršava na velikom broju različitih arhitektura. Na početku je podržavao samo AT&T sintaksu, a od verzije 2.10 podržava i Intelovu sintaksu. Na ovom kursu radićemo GNU assembler i Intelovu sintaksu.

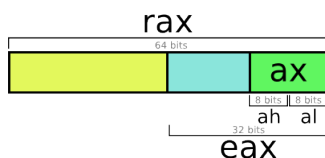
1.1 Registri

x86 ima 8 registara opšte namene (eax, ebx, ecx, edx, ebp, esp, esi, edi). Pošto je x86 32-bitna arhitektura, to znači da adrese u memoriji možemo da zapišemo u 32 bita, odnosno da ukupno možemo da zapišemo 4GB RAM-a. Takođe, veličina registara je 32 bita. U x64 ovi registri su prošireni do 64 bita. Novodobi-jeni registri imaju isti naziv osim što je umesto slova 'e' prvo slovo 'r' i njihova donja polovina (niži bitovi) su odgovarajući x86 registri. Npr. prošireni registar rax u x64 ima kao svoju donju polovinu eax, dok prošireni registar rbx ima kao svoju donju polovinu rbx.

U x86 nisu svi registri zaista bili registri opšte namene, jer su se neki implicitno koristili u specifične svrhe. Npr. registri ebp i esp za rad sa stekom. U x64, njima odgovarajući registri rsp i rbp zadržavaju tu namenu.

Detaljnije o registrima:

- **rax** - Povratne vrednosti funkcije se skladište u njemu, niža polovina mu je eax, dok je niža polovina eax-a ax. Registar od samo 16 bitova ax, dovoljan je da se u njemu smesti npr. podatak tipa short int, i ima dve svoje polovine - višu ah i nižu al, koje su po 8 bitova i u njima može da stane podatak tipa char.



- **rbx** - Registar koji mora da se sačuva pre upotrebe. Nakon što završimo rad sa njim, mora da ponovo ima vrednost koju je imao pre nego što smo ga koristili. Niža polovina mu je ebx, čija je niža polovina bx. Registar bx ima višu polovinu bh i nižu bl.
- **rcx** - Tipični registar opšte namene. Često se koristi kao brojač. Niža polovina mu je ecx, čija je niža polovina cx. Registar cx ima višu polovinu ch i nižu cl.

- **rdx** - Registar opšte namene. Niža polovina mu je edx, čija je niža polovina dx. Registar dx ima višu polovinu dh i nižu dl.
- **rsp** - Registar koji pokazuje na vrh steka (detalji u nastavku). Niža polovina mu je esp, čija je niža polovina sp. Niža polovina registra sp je spl. Kada koristimo ovaj registar moramo da budemo veoma pažljivi. Nije preporučljivo koristiti ga za druge namene, jer onda dolazi do grešaka u radu sa stekom.
- **rbp** - Ponekad se koristi da čuva staru vrednost steka ili "bazu". Preciznije, kada smo u pozivu funkcije (što podrazumeva da je na stek stavljen okvir funkcije i da je rsp promenjen), moramo da zapamtimo gde smo stali na steku kada se funkcija završi i ukloni sa steka. Za to nam služi rbp. Niža polovina mu je ebp, čija je niža polovina bp. Niža polovina registra bp je bpl. Nije preporučljivo koristiti ga za druge namene, jer onda dolazi do grešaka u radu sa stekom.
- **rsi** - Registar opšte namene. Koristi se za prenos drugog argumenta funkcije. Niža polovina mu je esi, čija je niža polovina si. Niža polovina registra esi je sil.
- **rdi** - Registar opšte namene. Koristi se za prenos prvog argumenta funkcije. Niža polovina mu je edi, čija je niža polovina di. Niža polovina registra edi je dil.
- **r8, r9, r10, r11** - Registri opšte namene. Niža polovina r8 je r8d. Niža polovina r8d je r8w. Niža polovina r8w je r8b. Sufiksi d,w,b su skraćenice od dword, word, byte i opisuju koliko memorije zauzimaju. Slično se određuju delovi registara r9, r10 i r11.
- **r12, r13, r14, r15** - Registri koje bi trebalo sačuvati, odnosno ako radimo sa njima, trebalo bi ih nakon što završimo vratiti na vrednosti koje su imali pre nego što smo počeli rad sa njima. Njihovi delovi se određuju slično kao za registar r8.

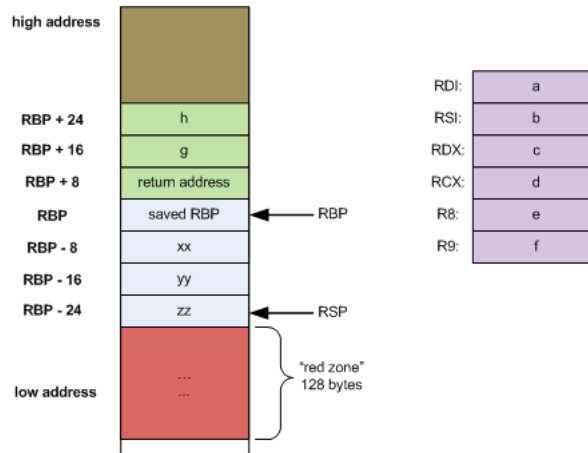
1.2 Prosledjivanje argumenata

Prvih šest celobrojnih argumenata funkcije (od kojih su neki potencijalno pokazivači, jer pokazivači čuvaju adrese (koje su celi brojevi) i zauzimaju isti prostor kao neki celobrojni podatak) prenose se redom u registrima rdi, rsi, rdx, rcx, r8 i r9. Ako imamo više od šest ovakvih argumenata onda se oni prenose preko steka. Najbolje je da pogledamo na primeru. Za ovu funkciju stek izgleda na sledeći način: Ne smemo da pokvarimo rbp zato što nam rbp određuje poziciju na steku iznad koje ne smemo da diramo podatke. Preciznije, mi znamo da se na adresi rbp+8 nalazi adresa povratka iz funkcije. Ako to "pokvarimo" nećemo znati gde treba da se vratimo iz funkcije. Takodje, pošto smo imali osam argumenata funkcije, a ne šest, dva su morala da budu preneti preko steka. Oni se redom nalaze iznad adrese povratka, na adresama rbp+16, rbp+24, itd. (**Ovde**

```

long myfunc(long a, long b, long c, long d,
            long e, long f, long g, long h)
{
    long xx = a * b * c * d * e * f * g * h;
    long yy = a + b + c + d + e + f + g + h;
    long zz = utilfunc(xx, yy, xx % yy);
    return zz + 20;
}

```



treba biti pažljiv. U pitanju su bili podaci tipa `long` i oni zauzimaju 8 bajtova, da je bio u pitanju podatak tipa `int` zauzeo bi samo 4 bajta, i ne bi išao na adresu `rbp+16`, nego na `rbp+12`, jer bi bio 4 bajta udaljen od adrese povratka koja je na `rbp+8`).

Ostaje da kažemo nešto i o crvenoj zoni koja je jedan vid optimizacije. Možemo da pretpostavimo da 128 bajtova ispod `rsp` neće biti narušene raznim signalima i prekidima, te možemo da koristimo taj prostor za medjuračunanja, bez da eksplicitno pomeramo vrh steka. Treba imati na umu da ova zona može da bude pogodjena pozivima drugih funkcija, te je najbolje da je koristimo u funkcijama koje ne sadrže pozive drugih funkcija u sebi. Pogledajmo kako "radi" crvena zona na primeru funkcije koja u sebi nema pozive drugih funkcija: Izgled

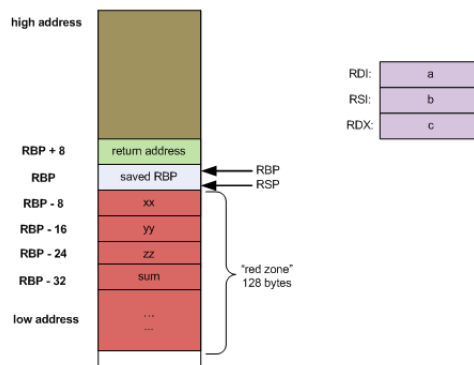
```

long utilfunc(long a, long b, long c)
{
    long xx = a + 2;
    long yy = b + 3;
    long zz = c + 4;
    long sum = xx + yy + zz;

    return xx * yy * zz + sum;
}

```

steka za ovu funkciju dat je na slici ispod. U funkciji `utilfunc`, nema prenosa promenljivih po steku, jer argumenata funkcije ima manje od šest. Takođe, zbog nedostatka poziva ka drugim funkcijama, moguće je da se rezultati nekih



izračunavanja (ovde sve promenljive za medjukorake zauzimaju manje jednako od 128 bajtova) sačuvaju u crvenoj zoni.

1.3 GDB debugger

Ako smo napisali naš program 1.s koristeći assembler Intel64 arhitekture (x64), taj program ćemo prevesti sledećim dvema naredbama:

```
as -gstabs -o 1.o 1.s
```

```
gcc -g -Wall -pedantic -no-pie -o 1 1.o
```

Opcije -g i -gstabs odnose se na formate za debugovanje. Opcija -g podrazumeva naivni format za debugovanje koji pruža operativni sistem i ovakav format za debugovanje podržan je od strane GDB debagera koji ćemo koristiti. Što se tiče -gstabs opcije, na nekim sistemima GNU assembleri zahtevaju ovu opciju. Opcija -pedantic uključuje dodatne ekstenzije i generiše detaljnija upozorenja. Opcija -no-pie omogućava da se ne generiše poziciono nezavisan izvršni fajl. Pie je bezbednosni preduslov da svaki put kernel učitava binarni fajl i zavisnosti u proizvoljnu adresu virtuelne memorije. Često zahteva dodatno programiranje.

GDB debugovanje pokrećemo u terminalu sa: `gdb ./1` gde je 1 generisani izvršni program. Ako želimo da pokrenemo program potrebno je da unesemo komandu `start`. Ako želimo da se pomerimo na sledeću mašinsku instrukciju koristimo komandu `stepi`. Ako želimo da udjemo u poziv funkcije i proverimo šta se u njoj dešava koristimo instrukciju `step`. Za štampanje vrednosti registara npr. registra `rax` kucamo `info registers $rax`.

1.4 Hello world program

Započnimo naš put ka učenju GNU assemblera najjednostavnijim programom - Hello world. U stanju smo da sa slike 1 identifikujemo nekoliko važnih činjenica koje ćemo postupno izložiti.

1.4.1 Intelova sintaksa

U prvoj liniji hello world programa vidimo da stoji navedeno **.intel_syntax noprefix**. Ova jednostavna komanda nam govori da je sintaksa koju ćemo koristiti Intelova sintaksa bez prefiksa, a ne AT&T sintaksa koju podrazumevano koristi GCC kompajler. Neke od razlika su sledeće:

- U AT&T na nazive registara dodaje se prefiks %, a na nazive konstanti prefiks #
- Redosled operandu u instrukcijama je različit. U Intelovoj sintaksi prvi operand je odredišni (engl. *destination*), a drugi izvorni (engl. *source*), dok je u AT&T sintaksi obrnuto.
- Kada u Intelovoj sintaksi želimo da pristupimo vrednosti koja se nalazi na adresi zapisanoj u registru rax, to činimo na sledeći način sa uglastim zagradama: [rax]. U AT&T sintaksi, zapis bi bio (%rax).
- U AT&T sintaksi, nazivi instrukcija dobijaju odgovarajući sufiks, koji zavisi od veličine operandu. Npr. ako želimo da u Intelovoj sintaksi prebacimo neku vrednost iz registra al u bl, ili iz registra ax u bx, bez obzira na to što al i bl zauzimaju različit broj bitova od ax i bx, naziv instrukcije bi bio isti - mov. Zapisali bismo te instrukcije na sledeći način:

mov bl, al

mov bx, ax

Sa druge strane, u AT&T sintaksi, prva instrukcija bi bila:

movb al, bl

jer su al i bl veličine jednog bajta, i kao što smo ranije naučili, redosled operandu je obrnut od onog u Intelovoj sintaksi. Druga bi bila:

movw ax, bx

jer su ax i bx veličine 16 bitova. Oznaka 'w' je za broj bitova, odnosno za word - 4 bajta. Za long vrednosti - 8 bajtova, koristi se sufiks 'l'.

1.4.2 Sekcija sa podacima

U jednostavnom hello world programu prikazana su dva načina da se definiše promenljiva koja će u sebi sadržati tekst "Hello world". Drugi način je stavljen pod komentar koristeći oznaku #. U zavisnosti od toga da li dozvoljavamo menjanje promenljivih koje definišemo, možemo da se opredelimo za stavljanje te promenljive u sekciju za read-only podatke (podatke koji se samo mogu čitati, ali se njihove vrednosti ne mogu menjati), datu sa **.section .rodata** ili u sekciju sa podacima koji se mogu menjati **.data**. Podaci u read-only odeljku se moraju

```

1.intel_syntax noprefix
2
3.section .rodata
4Hello: .string "Hello world"
5
6# .data
7# x:      .int 0
8# message: .asciz "Hello world!\n"
9
10.text
11
12.global main
13
14main:
15    enter 0,0
16
17    lea rdi, Hello
18
19    call printf
20
21    leave
22    ret

```

Figure 1: Caption

inicijalizovati, dok je za podatke u odeljku `.data`, preporuka da se inicijalizuju. Promenljive se definišu imenom iza koga sledi dvotačka, tip i vrednost. Tip promenljive može da bude `.string`, `.ascii`, `.asciiz`, `.double`, `.float`, `.byte`, `.word`, `.int`, `.long`, itd. Direktive `.ascii`, `.asciz` i `.string` na prvi pogled mogu delovati identično, ali to nije u potpunosti tačno. Naime, intuitivno, `.ascii` je slično `char[]`, te se kada mu dodelimo npr. reč "hello" to prevodi na sledeći kod `0x48 0x65 0x6c 0x6c 0x6f`. Sa `.asciz` je isto, osim što se dodaje nul na kraj, pa dobijamo sekvencu `0x48 0x65 0x6c 0x6c 0x6f 0x00`. Što se tiče `.string`, kao i većina drugih gas direktiva, može da ima različita značenja u zavisnosti od toga na kom sistemu radimo. Tako na nekim sistemima `.string` je isto što i `.ascii`, a na nekim isto što i `.asciz`.

1.4.3 Tekst i global

Kao i `.data` i `.section .rodata`, `.text` i `.global` su asemblerske direktive zadužene za kontrolu sekcija. Direktiva `.text` nam govori da počevši od nje kreće kod. Direktiva `.global` zadužena je da u fazi linkovanja, ukoliko budemo asemblersku funkciju `main` pozivali iz nekog drugog programa (npr. c-ovskog 1.c) taj program zna da je definicija funkcije `main` u našem fajlu 1.s, odnosno `.global` čini funkciju `main` vidljivu drugim fajlovima koji se linkuju sa našim. Nakon **`.global main`** vidimo labelu **`main`** koja nam govori odakle počinje naša funkcija `main`. Labele možemo koristiti i za druga mesta koja po našem mišljenju čine neku celinu i na njih možemo skakati naredbama uslovnih i bezuslovnih skokova.

Ono što je posebno važno da imamo na umu jeste da unutar jednog asemblerskog fajla možemo imati i više funkcija, i svaku od njih možemo "proglasiti"

```

push rbp
mov rbp, rsp
sub rsp, numVars

```

Figure 2: Caption

za global.

1.4.4 Enter, leave i ret

Svaka funkcija koju budemo napisali u GNU assembleru počinje instrukcijom `enter`, a završava se instrukcijama `leave` i `ret`. Zato ćemo u ovom odeljku posebnu pažnju posvetiti objašnjenju kako one rade. Krenimo od `enter`. Instrukcija **`enter varNum, 0`** je zamena za niz instrukcija na slici 2.

U registru `rbp` čuva se adresa na steku iznad koje ne smemo ništa da diramo, jer ta memorija ne pripada nama. Kada neka funkcija (recimo `main`) poziva našu funkciju (recimo `prime`), na steku se formira novi stek okvir za nas. Funkcija `main` ima svoju vrednost `rbp`-a, odnosno ona "zna" gde se nalazi memorija koja njoj ne pripada. U trenutku kada ona pozove `prime`, `prime` ne sme da menja memoriju koja pripada `main`-u i ne sme da izgubi informacije koje su zbog nje stavljene na stek (parametre koji nisu mogli da se prenesu preko registara, adresu povratka...). Tako da i `prime` treba da postavi svoju vrednost `rbp`-a na trenutnu vrednost vrha steka koja se nalazi u `rsp`. Međutim, ako to odmah uradi, izgubiće vrednost `rbp`-a funkcije `main`, i po povratku u funkciju `main` nakon što se `prime` završi može se dogoditi da funkcija `main` pristupi memoriji kojoj ne bi smela. Zato je važno da pre nego što promenimo vrednost registra `rbp` u adresu na steku koja je nama potrebna, sačuvamo stari `rbp` na stek. Nakon što ga stavimo na stek, `rsp` se automatski ažurira i pokazuje na novi vrh steka. Nakon toga možemo da `rbp`-u dodelimo tu vrednost, što je za nas signal da funkcija `prime` neće promeniti vrednosti na steku i u memoriji koje ne bi smela.

Treća instrukcija na slici 2 služi da "odvoji" memoriju na steku za naše lokalne promenljive. Izrazito je korisna ukoliko nam je potrebno da imamo niz fiksne veličine. Konstanta `numVars` je broj bajtova koji nam je potreban da se odvoji i važno je da na x64 arhitekturi to bude broj koji je parni umnožak od osmice, kako bi stek bio poravnat (o tome u nastavku skripte).

Pogledajmo primer na slici 3. Kada u funkciju udjemo sa **`enter 16,0`**, to znači da smo alocirali 16 bajtova za naše promenljive. Ovde ćemo da stavimo niz od dva elementa. Naš `rsp` je bio jednak `rbp`-u, a potom se od njega oduzelo 16, te je on ustvari postao `rbp-16`. Nulti član niza se upisuje na adresu `rsp`, prvi član niza na adresu `rsp+8`. Da smo ih imali više upisivali bismo ih redom na adrese `rsp+16`, `rsp+24`, `rsp+32`, itd. Ovo može delovati da je u suprotnosti sa pravilom opadajućeg steka gde novi podaci dobijaju manje adrese, međutim to nije tačno. Naime, moramo ispoštovati princip stavljanja nizova u memoriju, a to je da se niz pakuje u memoriju u jednom komadu, i da je element `a[i]` na adresi `a+i`. Odnosno, nulti član je na adresi `a`, prvi na `a+1`, drugi na `a+2`, itd.


```

1.intel_syntax noprefix
2
3.section .rodata
4Hello: .string "Hello world\n"
5
6.section .data
7    x: .int 0
8    message: .asciz "Hello world!\n"
9    fmt: .asciz "%d\n"
10.text
11
12.global main
13
14main:
15    enter 16,0
16
17    mov r8, rsp
18
19    mov qword ptr [r8],100
20    mov qword ptr [r8+8], 2
21
22    push r8
23    push r8
24    lea rdi, Hello
25    call printf
26    pop r8
27    pop r8
28    mov rax, [r8]
29
30
31    lea rdi, fmt
32    mov rsi, rax
33    call printf
34
35    mov rax, [rsp+8]
36
37|
38    lea rdi, fmt
39    mov rsi, rax
40    call printf
41
42    leave
43    ret

```

Figure 3: Caption

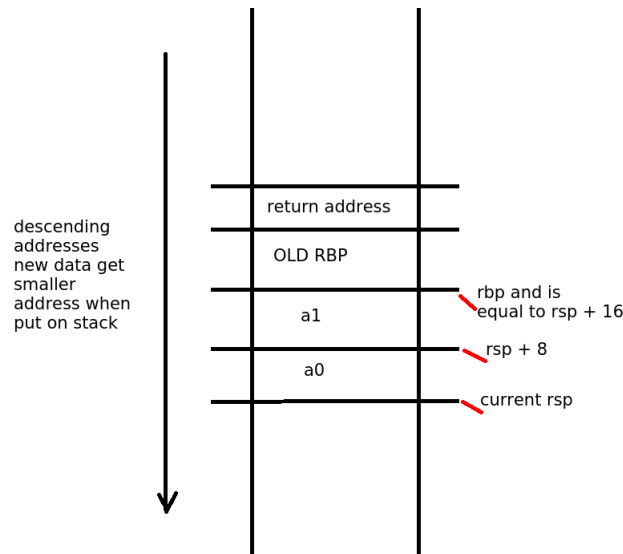


Figure 4: Caption

gde je uvećavanje za 1 zapravo pokazivačka aritmetika, odnosno uvećavanje za jedan zapravo predstavlja uvećavanje za broj bajtova koji promenljiva zauzima što je u našem slučaju 8. Više o tome možemo videti na slici 4.

Objasnim detaljnije kod sa slike 3. Čitaocu savetujemo da se na ovo objašnjenje vrati nakon što pročita deo o `leave` i `ret`. Kako će se vrednost registra `rsp` menjati u toku programa, najbolje bi bilo da tu vrednost zapamtimo u nekom drugom registru, npr. `r8` i to činimo u 17. liniji instrukcijom `mov`. Instrukcije u linijama 19. i 20. su karakteristične zato što opisuju donošenje podatka u memoriju koji može da bude zapisan sa više ili manje bajtova (kada napišemo broj 100, možemo da mislimo na `int` ili na `long` ili na `short`), te je zato potrebno da preciziramo tačan broj bajtova koji naš podatak zauzima. Ukoliko to ne uradimo dobićemo sledeći tip greške: *ambiguous operand size for instruction mov*. Problem se može rešiti na dva načina. Možemo da pre 19. i 20. linije stavimo 100 i 2 u registre, čime se oni zapisuju u onoliko bajtova koliko taj registar zauzima, i da onda na adresu zapisanu u `r8` donesemo vrednost tog registra. Npr.

```
mov rax, 100
mov [r8], rax
```

Drugi način je da se ispred adrese precizira i veličina podatka koji ćemo doneti. U ovom slučaju želimo da podatak zapišemo na osam bajtova pa ćemo koristiti `qword`. Za 4 bajta bi bilo `dword`, za dva bajta `word`, i za jedan byte. Ne zaboravimo da nakon toga napišemo `ptr`, što je skraćenica za pokazivač. Odnosno, `r8` je pokazivač na taj memorijski blok, veličine koju smo odredili.

Može se dogoditi da se u pozivu funkcije `printf` pokvari vrednost registra `r8`, te je zato potrebno sačuvati taj registar pre ovog poziva i dohvatiti ga nakon poziva. Stavili smo dva `push-a`, jer želimo da nam adrese na steku stalno budu poravnate (u nastavku skripte više o tome), odnosno da stalno budu parni umnožak osmice.

`Leave` i `ret` su zamena za instrukcije na slici 5. Prva komanda na toj slici nam govori o tome da mi najpre brišemo sve lokalne promenljive. Potom vraćamo stari `rbp` jer se pripremamo da se vratimo u funkciju čiji je to `rbp` bio i želimo da ona ima informaciju o tome čiju memoriju ne sme da pokvari. Nakon što smo stari `rbp` skinuli sa steka, `rsp` nam pokazuje na adresu povratka. Instrukcija `ret` uzima adresu sa steka na koju pokazuje `rsp` i vodi nas na nju. Nakon toga se nastavlja izvršavanje funkcije koja je pozvala našu.

```
mov rsp, rbp
pop rbp
ret
```

Figure 5: Caption

1.4.5 Instrukcije `mov` i `lea`

Naziv instrukcije `lea` potiče od engleske rečenice Load Effective Address i to je ujedno opis onoga što ona radi. Naime, instrukcija `lea` učitava pokazivač na stavku koju adresiramo. Za razliku od nje `mov` učitava vrednost stavke koju adresiramo. Primeri upotrebe:

- `mov rax, [broj]` - broj je promenljiva definisana u sekciji sa podacima. Smisao ove instrukcije jeste da će se u registar `rax` doneti vrednost promenljive broj. Odnosno, `mov` je "protumačio" ovaj zapis na sledeći način: U registar `rax` doneti ono što se nalazi na adresi promenljive broj.
- `mov rax, broj` - broj je promenljiva definisana u sekciji sa podacima. Smisao ove instrukcije jeste da će se u registar `rax` doneti vrednost promenljive broj. Odnosno, `mov` je "protumačio" ovaj zapis na sledeći način: U registar `rax` doneti vrednost promenljive broj.
- `lea rax, [broj]` - broj je promenljiva definisana u sekciji sa podacima. Smisao ove instrukcije jeste da će se u registar `rax` doneti adresa promenljive broj. Odnosno, `lea` je "protumačila" ovaj zapisa na sledeći način: U registar `rax` doneti adresu promenljive broj.
- `lea rax, broj` - broj je promenljiva definisana u sekciji sa podacima. Smisao ove instrukcije jeste da će se u registar `rax` doneti adresa promenljive broj. Odnosno, `lea` je "protumačila" ovaj zapisa na sledeći način: U registar `rax` doneti adresu promenljive broj.
- `mov rax, [r8]` - Smisao ove instrukcije jeste da se u registar `rax` donese ono što je zapisano na adresi čija se vrednost nalazi u `r8`.

- *mov rax, r8* - Smisao ove instrukcije jeste da se u registar rax donese vrednost koja se nalazi u r8.
- *mov [rax], r8* - Smisao ove instrukcije jeste da se na adresu čija se vrednost nalazi u rax-u upiše ono što je zapisano u registru r8.
- *mov [rax], [r8]* - Ovakva instrukcija nije moguća, zato što operandi mov-a ne mogu oba da budu memorijska.
- *lea rax, [rdi * 8]* - U registar rax upisuje se efektivna adresa rdi*8, a ne ono što je na toj adresi nadjeno. Umesto broja 8 nije mogao da stoji proizvoljan broj. Više o ovome pročitati u pododeljku 1.6.1.

1.5 Osnovne aritmetičke operacije

Ovaj odeljak posvećen je izvršavanju osnovnih aritmetičkih operacija u GNU assembleru.

1.5.1 Sabiranje i oduzimanje

Instrukcije ADD i SUB su redom instrukcije zadužene za sabiranje i oduzimanje 8-bitnih, 16-bitnih, 32-bitnih i 64-bitnih operanada.

Instrukcijom ADD je moguće:

- dodati vrednost jednog registra na drugi (npr. *add rax, rbx* u značenju $rax = rax + rbx$)
- dodati vrednost registra na vrednost iz memorijske adrese (npr. *add [rax], rbx* u značenju da se na memorijskoj lokaciji zapisanoj u rax-u čuva suma prethodne vrednosti sa te memorijske lokacije i vrednosti koja se nalazi u registru rbx)
- dodati vrednost sa memorijske lokacije na vrednost registra (npr. *add rax, [rbx]* u značenju da nova vrednost registra rax postane suma stare vrednosti registra rax i vrednosti koja se nalazi na memorijskoj lokaciji zapisanoj u registru rbx)
- dodati konstantu na registar (npr. *add rax, 4* u značenju $rax = rax + 4$)
- dodati konstantu na memorijsku lokaciju (npr. *add qword ptr [rax], 4* u značenju da se na vrednost koja se nalazi na memorijskoj lokaciji zapisanoj u registru rax dodaje broj četiri zapisan na 8 bajtova)

Analogno se upotrebljava instrukcija SUB. Treba primetiti da nije moguće da oba operanda instrukcija ADD i SUB budu memorijske lokacije. Osim ove dve instrukcije postoje i instrukcije INC (npr. *inc rax*) i DEC (npr. *dec rax*) koje redom uvećavaju i smanjuju vrednost u registru za 1.

1.5.2 Množenje i deljenje

U zavisnosti od toga da li se radi o označenim ili neoznačenim celim brojevima, razlikujemo instrukcije za označeno množenje i deljenje (IMUL i IDIV) i instrukcije za neoznačeno množenje i deljenje (MUL i DIV).

Za različite veličine operanada procesi množenja i deljenja su različiti. Razmotrimo najpre množenje:

- **množenje dva operanda koji su oba veličine jedan bajt** - Podrazumevano, prvi činilac mora biti smešten u registar al, dok je drugi činilac ili registar ili vrednost sa memorijske lokacije. Viših osam bitova proizvoda se upisuju u ah, a nižih u al. Primer množenja 4 sa 2:

mov al, 4

mov bl, 2

mul bl

- **množenje dva operanda koji su oba veličine dva bajta** - Podrazumevano, prvi činilac mora biti smešten u registar ax, dok je drugi činilac ili registar ili vrednost sa memorijske lokacije. Viša polovina bitova proizvoda se upisuje u dx, a niža u ax.
- **množenje dva operanda koji su oba veličine četiri bajta** - Podrazumevano, prvi činilac mora biti smešten u registar eax, dok je drugi činilac ili registar ili vrednost sa memorijske lokacije. Viša polovina bitova (32-bitova) proizvoda se upisuje u edx, a niža u eax.
- **množenje dva operanda koji su oba veličine osam bajtova** - Podrazumevano, prvi činilac mora biti smešten u registar rax, dok je drugi činilac ili registar ili vrednost sa memorijske lokacije. Viša polovina bitova (64-bitova) proizvoda se upisuje u rdx, a niža u rax.

Deljenje je nešto složenije:

- **deljenje dva operanda koji su oba veličine jedan bajt** - Podrazumevano, deljenik mora biti smešten u registar al, a u registru ah moraju biti smešteni bitovi znaka deljenika. Ukoliko je neoznačeno deljenje u pitanju to znači da se registar ah mora ispuniti nulama. Medjutim, ako nismo sigurni da li je deljenje označeno ili neoznačeno, moramo iskoristiti instrukciju cbw koja proverava kakav je znak broja u al i ispunjava ah odgovarajućim bitovima znaka. Delilac je ili registar odgovarajuće veličine (8-bitova) ili vrednost sa memorijske lokacije. Nakon deljenja, količnik se nalazi u al, a ostatak u ah.
- **deljenje dva operanda koji su oba veličine dva bajta** - Podrazumevano, deljenik mora biti smešten u registar ax, a u registru dx moraju biti smešteni bitovi znaka deljenika. Ukoliko je neoznačeno deljenje u

```
mov al,-48
cbw
mov bl,5
idiv bl
```

Figure 6: Caption

pitanju to znači da se registar dx mora ispuniti nulama. Međutim, ako nismo sigurni da li je deljenje označeno ili neoznačeno, moramo iskoristiti instrukciju `cwd` koja proverava kakav je znak broja u `ax` i ispunjava `dx` odgovarajućim bitovima znaka. Delilac je ili registar odgovarajuće veličine (16-bitova) ili vrednost sa memorijske lokacije. Nakon deljenja, količnik se nalazi u `ax`, a ostatak u `dx`.

- **deljenje dva operanda koji su oba veličine četiri bajta** - Podrazumevano, deljenik mora biti smešten u registar `eax`, a u registru `edx` moraju biti smešteni bitovi znaka deljenika. Ukoliko je neoznačeno deljenje u pitanju to znači da se registar `edx` mora ispuniti nulama. Međutim, ako nismo sigurni da li je deljenje označeno ili neoznačeno, moramo iskoristiti instrukciju `cdq` koja proverava kakav je znak broja u `eax` i ispunjava `edx` odgovarajućim bitovima znaka. Delilac je ili registar odgovarajuće veličine (32-bita) ili vrednost sa memorijske lokacije. Nakon deljenja, količnik se nalazi u `eax`, a ostatak u `edx`.
- **deljenje dva operanda koji su oba veličine osam bajtova** - Podrazumevano, deljenik mora biti smešten u registar `rax`, a u registru `rdx` moraju biti smešteni bitovi znaka deljenika. Ukoliko je neoznačeno deljenje u pitanju to znači da se registar `rdx` mora ispuniti nulama. Međutim, ako nismo sigurni da li je deljenje označeno ili neoznačeno, moramo iskoristiti instrukciju `cqto` koja proverava kakav je znak broja u `rax` i ispunjava `rdx` odgovarajućim bitovima znaka. Delilac je ili registar odgovarajuće veličine (64-bita) ili vrednost sa memorijske lokacije. Nakon deljenja, količnik se nalazi u `rax`, a ostatak u `rdx`.

1.5.3 Potencijalni problemi

Ponekad, iako nam se čini da smo množenje i deljenje sproveli tačno onako kako je trebalo, dobijamo neočekivan rezultat. Potencijalni razlog za to može da bude činjenica da upisivanje broja u manji deo registra za neke instrukcije ispunjava ostatak registra nulama, a za neke ne. Takodje, ako je broj koji upisujemo negativan, ispunjavanje ostatka registra nulama verovatno nije ponašanje koje želimo. Iz tog razloga, potrebne su nam instrukcije koje će za nas ili da ispune nulama ostatak registra ili da ga ispune bitovima znaka, u zavisnosti od toga kakav efekat želimo da postignemo. Primeri:

- `movsxd rax, eax` - ova instrukcija proširuje bitovima znaka registar `eax` do registra `rax`. Umesto `rax` i `eax`, možemo da koristimo i drugi par jednog 64-bitnog i jednog 32-bitnog registra. Sufiks `d` u nazivu instrukcije naglašava da je u pitanju proširivanje 32-bitnog registra (32-bita je `dword`) do 64-bitnog, a `sx` je skraćenica za `sign-extendion`.
- `movsxw eax, ax` - ova instrukcija proširuje bitovima znaka registar `ax` do registra `eax`. Umesto `eax` i `ax`, možemo da koristimo i drugi par jednog 32-bitnog i jednog 16-bitnog registra. Sufiks `w` u nazivu instrukcije naglašava da je u pitanju proširivanje 16-bitnog registra (16-bita je `word`) do 32-bitnog, a `sx` je skraćenica za `sign-extendion`.
- `movsxb ax, al` - ova instrukcija proširuje bitovima znaka registar `al` do registra `ax`. Umesto `ax` i `al`, možemo da koristimo i drugi par jednog 16-bitnog i jednog 8-bitnog registra. Sufiks `b` u nazivu instrukcije naglašava da je u pitanju proširivanje 8-bitnog registra (8-bita je `byte`) do 16-bitnog, a `sx` je skraćenica za `sign-extendion`.
- Sve iznad navedene varijante dolaze i sa nastavkom `zx`, i tada se proširivanje vrši nulama, te tako imamo instrukcije `movzxd`, `movzxw`, `movzxb`.
- `cltq` - instrukcija koja proširuje `eax` bitovima znaka do `rax` (pogledati i druge ovakve instrukcije u odeljku koji se odnosi na množenje i deljenje)

1.6 Nizovi

U radu sa višim programskim jezicima naučili smo da je naziv promenljive koja je tipa niz, ujedno i adresa početka tog niza, odnosno njegovog nultog elementa. Elementi niza su u memoriji spakovani u jednom komadu, i dodavanjem određenog broja bajtova na adresu početka niza (u zavisnosti od tipa podataka u nizu) prelazimo na odgovarajuće elemente niza. Na primer, zamislimo da imamo niz `a` u kome se nalaze elementi tipa `long` (podatak ovog tipa zauzima 8 bajtova). Nulti član niza počinje na adresi `a`, prvi na adresi `a+8`, drugi na adresi `a+16`, treći na adresi `a+24`, itd. Zaključujemo da se `i`-ti član niza `a` nalazi na adresi `a+8*i`.

U GNU assembleru dozvoljen je pristup memorijskoj lokaciji zapisanoj u sledećem obliku **baza+index*skaliranje + pomeraj**. Baza i indeks mogu da budu bilo koji registari od 64-bita. Skaliranje je veličina podatka koji pokušavamo da dohvatimo i može da bude 1 (za 1 bajt - `byte`), 2 (za 2 bajta - `word`), 4 (za 4 bajta - `dword`) i 8 (za 8 bajtova - `qword`). Pomeraj je samo konstanta koja se dodaje na ostatak adrese i može da ne postoji ili da bude 8, 16, 32 ili 64 bita.

U ovakvom zapisu memorijske lokacije prepoznavamo i zapis nizova. Naime baza nam je ovde adresa početka niza, indeks je pozicija na kojoj tražimo određen element, a skaliranje zavisi od tipa elemenata u nizu. Na primer:

```
mov [rdi + 8 * rax], rbx
```

Ova instrukcija postavlja vrednost elementa pod rednim brojem `rax` u nizu na adresi `rdi`, elemenata koji su veličine 8 bajtova, na vrednost registra `rbx`.

1.6.1 Nizovi i množenje - hack

U prethodnom delu skripte naučili smo kako radi `lea`, kako se množe operandi, i kako se pristupa elementima u nizu. Sada je vreme da to iskoristimo i posvetimo ovaj deo skripte jednom zanimljivom optimizacionom triku. Naime, zamislamo da vrednost registra `rdi` treba da pomnožimo sa 8 i da taj rezultat stavimo u `rax`. To možemo učiniti uz pomoć naredne instrukcije:

```
lea rax,[rdi * 8]
```

Broj osam je pažljivo odabran i on je u ovoj instrukciji skaliranje, `rdi` je indeks a baze i pomeraja nema. Instrukcija `lea` radi sa adresama i ona postavlja vrednost `rax` na vrednost adrese `rdi*8`. Problem je samo jedna sitnica, a to je da u `rdi` nije adresa, već običan broj. Medjutim, to u stvari i nije problem, zato što smo mi svesni da u `rdi` nije adresa, kao ni u `rax`-u, te tako nećemo ni pristupati vrednostima koje se nalaze na broju zapisanom u `rax` i `rdi`. Na slici se nalazi

```
.intel_syntax noprefix

.data
    fmt: .asciz "%d\n"
.text
.global main
main:
    enter 0,0
    mov rdi, 3
    lea rax,[rdi+rdi*8]

    lea rdi, fmt
    mov rsi, rax
    call printf
    leave
    ret
```

Figure 7: Caption

asemblerski kod koji prikazuje način da se vrednost registra `rdi` pomnoži sa 9 i da se ta vrednost ubaci u registar `rax`. Jasno je da ne može adresa niza i indeks člana tog niza da budu iste vrednosti, ali `rdi` nije adresa niza i nije indeks u tom nizu, i mi smo ti koji to znamo. Zato nikada nećemo probati da pristupimo vrednosti na adresi zapisanoj u `rdi`.

Ako bismo želeli da `rdi` pomnožimo na primer sa 6 mogli bismo da ga pomnožimo sa 3, pa sa dva, upotrebom dve instrukcije.

```
lea rax,[rdi + rdi * 2]
lea rax,[rax * 2]
```

Ovo nije samo trik koji štedi vreme i trud programera. Naime, on je i vremenska ušteda u assembleru, jer se brže izvršava nego niz komandi potreban za standardni način množenja.