Sveučilište u Zagrebu

Fakultet elektrotehnike i računarstva

Ime Prezime

Seminarski rad iz predmeta "Prevođenje programskih jezika"

Zadatak broj 2072

Zagreb, siječanj 2010.

Seminarski rad iz predmeta Prevođenje programskih jezika

Student: Ime Prezime

Matični broj studenta: 0036438727

Zadatak broj 2072: Programski ostvariti jezični **procesor koji prevodi aritmetički izraz zadan u postfix notaciji u strojni kôd**. Aritmetički izrazi obuhvaćaju operatore +, -, * i /. Samostalno odabrati neku od postojećih računalnih arhitektura i odgovarajući strojni kôd.

Sadržaj

1. Uvod 3
2. Naredbe 6
3. Generiranje naredbi 7
4. Prilozi 12
5. Problemi 13
6. Zaključak 13
7. Literatura 13

1. Uvod

Na početku ovog seminarskog želim opisati sve pojmove iz svog zadatka koji su specifični za ovaj kolegij kako bi Vam lakše objasnio što sam radio i zašto.

Prvo se spominje **jezični procesor**. *Jezični procesor je središnji proces prevođenja korisničkog programa u izvodivi strojni program.* [1] U mom slučaju je to program koji će uzeti aritmetički izraz (kojeg korisnik upisuje) i dati strojni kod.

Strojni kod je niz naredbi (u heksadekadskom ili binarnom obliku) koje su izvodive na određenoj arhitekturi. Odabrao sam arhitekturu ARM7 jer sam sa njom upoznat na kolegiju *Arhitektura računala 1*, a naprednija je od FRISC-a koji nema složenije naredbe poput množenja i dijeljenja (na žalost, ni ARM nema dijeljenje). Također, ARM7 je load-store arhitektura, što znači da neću morati slati podatke u akumulator nego ću moći vršiti operacije izravno iz registara opće namjene. Za ovakav zadatak je inače pogodnija stogovna arhitektura zbog postfiks notacije.

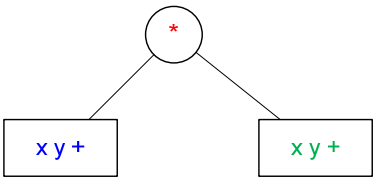
Jedini preostali pojam iz zadatka je **postfiks notacija**. Postfiks notacija je zapis u kojem se prvo navedu dva operanda pa jedan operator s time da bilo koji od ta dva operanda može biti također sastavljen od dva operanda nakon kojeg ide operator. *Postfiksni sustav oznaka nastaje obilaskom čvorova sažetog sintaksnog stabla po dubini.* [1] Ta rečenica je važna jer se iz nje može zaključiti da nećemo imati nikakvih zagrada te da nećemo morati paziti na prioritet (jer je sažeto sintaksno stablo hijerarhijsko).

Primjer postfiksnog zapisa je:

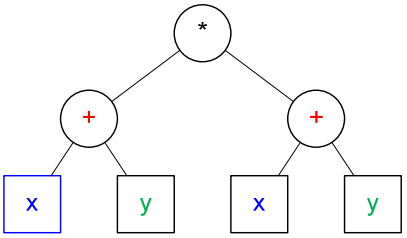
x y + x y + *

Kako čitati taj zapis? Kao što sam rekao, prvo se navode dva operanda pa operator, dakle pročitajmo prvi operand pa drugi pa operator. Nakon toga, pogledamo sadrži li operand u sebi operator i dva operanda. Zadnja slika je zapravo sažeto sintaksno stablo.

x y + x y + * (x y +) * (x y +)



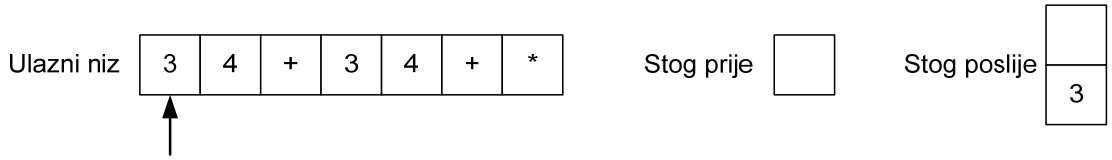
x y + x y + * (x + y) * (x + y)



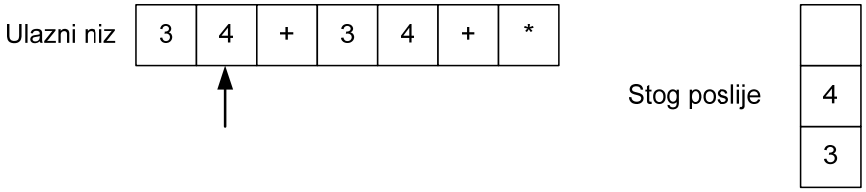
Taj način čitanja je intuitivan i ljudski. Čovjek lako prepoznaje operande dok će računalo ipak raditi malo drugačije koristeći stog. Krenuti će od početka te stavljati operande na stog dok ne dođe do operatora. Kada dođe do operatora onda će uzeti zadnja dva operanda sa stoga, izvršiti operaciju koju operand predstavlja te će rezultat pohraniti na stog i tako dok ne dođe do kraja niza. Zato sam i rekao da je pogodnija stogovna arhitektura jer su naredbe napravljene upravo tako da uzimaju operande sa stoga te pohranjuju rezultat na stog.

Primjer: 34+34+*

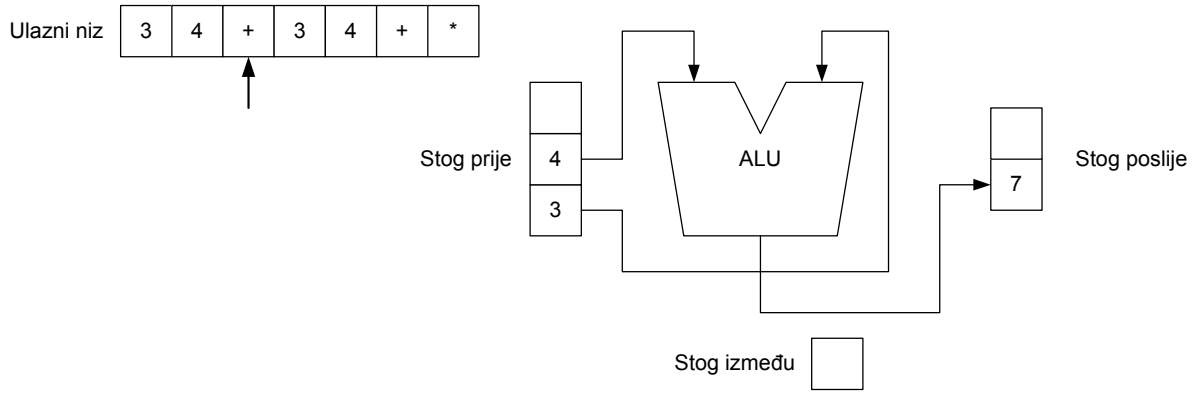
Stog je na početku prazan, a kazaljka pokazuje na početak niza. Kazaljku stoga nisam crtao jer nije ni bitna. Može pokazivati na zadnji učitani operand ili na prvo prazno mjesto iza zadnjeg operanda. Pročita se operand i stavi na stog (Slika 1.1). Slijedi također operand pa se i on stavi na stog (Slika 1.2). Nakon toga dolazimo do operatora. Uzimaju se zadnja dva operanda sa stoga, vrši se zadata operacija te se pohranjuje rezultat na stog (Slika 1.3). Isto se tako nastavlja sa čitanjem operanda (Slika 1.4 i Slika 1.5), pa sa čitanjem operatora (Slika 1.6 i Slika 1.7).



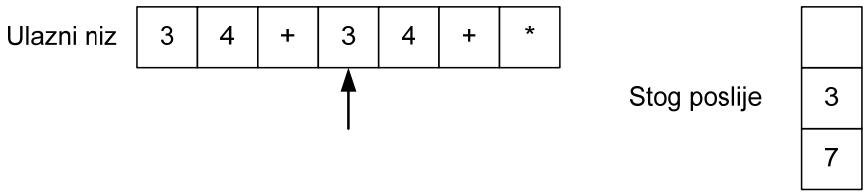
Slika 1.1 Čita se prvi operand



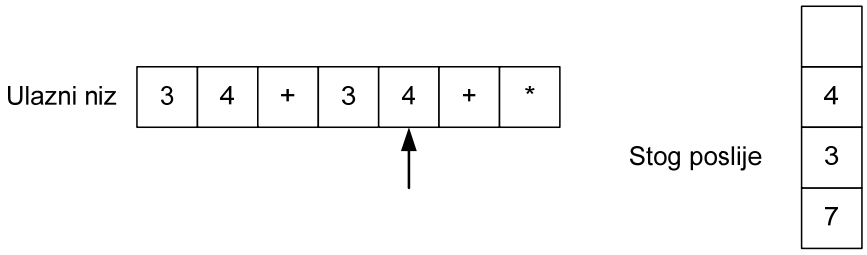
Slika 1.2. Čita se drugi operand i stavlja se na stog



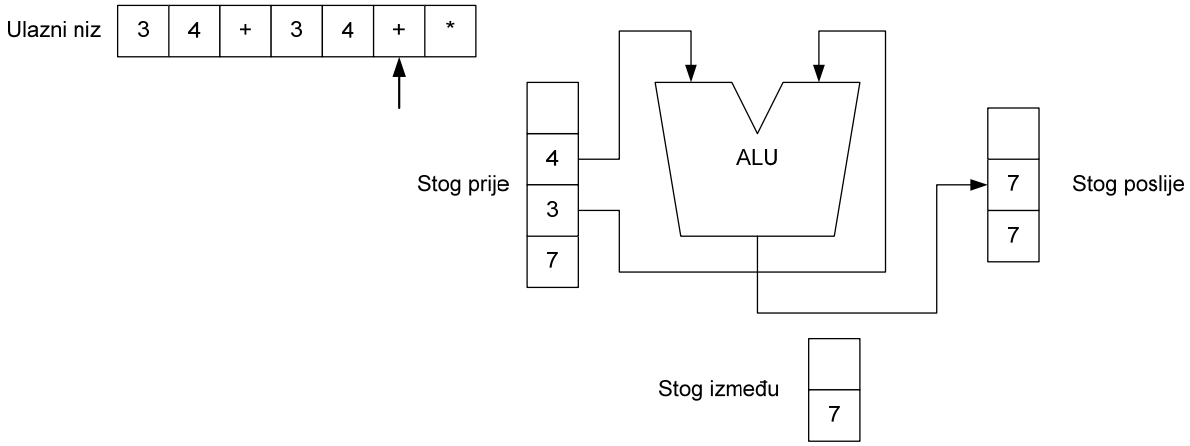
Slika 1.3. Čita se operator, uzimaju se brojevi sa stoga te pohranjuje rezultat na vrh stoga



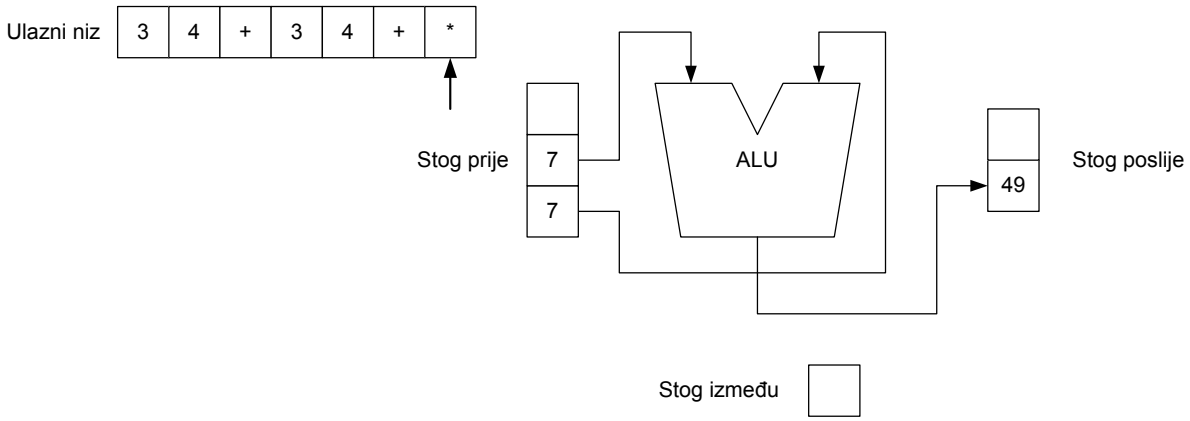
Slika 1.4. Čita se operand 3 i stavlja na stog



Slika 1.5. Čita se operand 4 i stavlja na stog



Slika 1.6. Čita se operator + i uzimaju se dva operanda sa stoga
te se vrši zbrajanje nakon kojeg se rezultat pohranjuje na stog



Slika 1.7. Čita se operator * i uzimaju se dva operanda sa stoga
te se vrši množenje i rezultat se sprema na stog

U primjeru su korišteni brojevi umjesto labela jer sve varijable sadrže neku svoju vrijednost i ALU radi operacije nad tim vrijednostima.

2. Naredbe

Naredbe u strojnom kodu su heksadekadski zapisi koji procesor analizira i vrši određene operacije. Nažalost, zbog nedostatka vremena i zbog nedostupnosti točnih heksadekadskih oblika odgovarajućih naredbi, moj program generira mnemonički oblik naredbi koje Atlas može prevesti u strojni kod.

Naredbe koje se generiraju su sljedeće:

```
`ORG 0          ; početak programa, tj. sljedeće naredbe će biti zapisane na toj lokaciji

MOV R11, #1  ; sprema se konstanta 1 u registar R11

MOV R11, R14          ; sadržaj registra R14 sprema u registar R11

LDR R0, #1     ; pohranjuje se konstanta 1 u registar R0

LDR R1, vibi   ; pohranjuje se podatak na adresi koja je obilježena labelom vibi

LDR R1, [R13, #-4]!   ; R13 se smanji za 4 te se učitava podatak sa adrese iz R13

STR R0, [R13, #4]!     ; R13 se poveća za 4 te se pohranjuje podatak na adresu R13

STMIA R13!, {R0, R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11, R12}

;pohranjivanje registara na stog

LDMDB R13!, {R0, R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11, R12}

; skidanje podataka sa stoga

ADD R0, R1, R12; zbrajaju se registri R1 i R12 i pohranjuje se u R0 (R0 = R1 + R12)

SUB R0, R0, R1; od registra R0 se oduzima sa R1 te se pohranjuje u R0 (R0 = R0 - R1)

MUL R0, R0, R1; množe se registri R0 i R1 te se rezultat sprema u R0 (R0 = R0 * R1)

CMP R12, R11; osvježava zastavice oduzimajući R12 i R11

BHI KRAJ; ako zastavice pokazuju da je prethodno oduzimanje bilo takvo da je rezultat
          pozitivan, onda se skače na mjesto koje sadrži labelu KRAJ

B DIV         ; bezuvjetni skok na labelu DIV
```

3. Generiranje naredbi

Program (koji je priložen i napisan u c-u) će čitati ulazni niz od početka do kraja te prepoznavati operande i operatore. Operand je niz znakova maksimalne duljine 10 ili broj. Maksimalna duljina niza znakova je 10 zato jer smo iz *Arhitekture računala 1* učili da kada FRISC ima labelu unutar asemblerskog koda, da se prepoznaju samo prvih 10 znakova dok se ostali zanemaruju. Iako se za ARM nije spominjalo nikakvo ograničenje, nije se spominjalo ni da ga nema pa zbog opreza preporučujem da ne upisujete više od 10 znakova. Programski nisam limitirao duljinu labele jer neće doći do greške u programu ni pri generiranju koda. Također, zbog jednostavnosti i nedostatka vremena, operand može sadržavati samo slova i brojke, s tim da ako započinje sa slovom je varijabla, a ako započinje brojkom (a onda mora nastaviti i završiti brojkom) je konstanta. Svi operandi iza sebe moraj imati točno jedan razmak, a operatori iza sebe ne smiju imati razmak.

Na početku će moj prevoditelj uvijek generirati slijedeću naredbu (uvijek u datoteku):

```
`ORG 0                  ; označava početak
```

Tek nakon toga će započeti sa čitanjem ulaznog niza. Prvo pročita prvi znak i zapamti je li bilo slovo ili znamenka. Ako je bilo slovo, sve do razmaka će tretirati kao varijablu, a ako je bila znamenka, sve do razmaka će tretirati kao broj. Uz to, ako je prvi učitani znak slovo, puniti će se polje koje će pamtiti ime varijable, a ako je bila učitana znamenka, onda se to polje neće puniti. Bitno je da operandi iza sebe imaju točno jedan razmak jer razmak označava kraj varijable te pokreće određene naredbe. Prva od tih naredba je uvećanje varijable koja se brine da se ne popune svi registri. Druga je pohranjivanje (ako se radilo o varijabli) imena varijable u polje te upisivanje 0 (kao neki graničnik između imena varijabli). Treća je generiranje jedne od ove dvije asemblerske naredbe:

```
LDR R%d, %s     ; gdje je %d broj prvog slijedećeg praznog registra, a %s ime varijable

LDR R%d, #%s    ; gdje je %d broj prvog slijedećeg praznog registra, a %s broj
```

Naredbe se razlikuju samo u # koji govori asembleru da se radi o konstanti. Kao što se vidi iz asemblerskih naredbi, podaci se spremaju u registar. To je zato jer ARM nema stogovnu nego load-store arhitekturu. Kad bi kod ARM-a podatke htio stavljati na stog, trebao bi prvo odrediti početak stoga (odabrati dovoljno daleku memorijsku adresu), pohraniti ga u registar (npr. R13) i svaki put koristiti ne jednu (kao gore) nego dvije asemblerske naredbe. Jednu koja stavlja podatak u registar, a drugu koja iz tog registra stavlja na stog (također bi se pojavila i asemblerska naredba viška kada bi htio izvršiti neku operaciju nad podatkom jer se operacije mogu vršiti samo nad podacima u registru). Te dvije asemblerske naredbe ne rade nikakav koristan posao a troše 2 ciklusa (zapravo 4 ako se gleda protočna struktura) pa sam odlučio iskoristiti činjenicu da ARM ima 15 registara koje mogu koristiti. Programski sam uredio da se pohranjuju operandi na stog tek ako se učitalo 13 operanda, a do tada se podaci pohranjuju u registrima od R0 do R12. Registar R13 će mi služiti kao stog (u njemu će biti pohranjena početna adresa stoga) ali samo kao nužda kada se popuni tih 13 registara. Tu se javlja problem jer ne znam koliko će mi program biti dugačak pa ne znam koliko će memorijskih lokacija zauzeti. Zato tijekom programa prilikom generiranja svake asemblerske naredbe uvećam varijablu R13 za 4.

Pseudo-kod algoritma za učitavanje operanda je sljedeći:

```
radi
{
    učitaj operand
    uvećaj broj operanda
    ako je broj operanda djeljiv sa 12 i veći od 0
    {
        generiraj posebnu asemblersku naredbu koja će spremiti registre na stog
    }
    inače
    {
        generiraj asemblersku naredbu koja će spremiti operand u registar koji ima broj
        jednak ostatku dijeljenja sa 13
    }
}do kraja učitanog niza
```

Posebna asemblerska naredba koja se generira, koja sprema registre na stog je:

```
STMIA R13!, {R0, R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11, R12}
```

Sada kada je opisan način učitavanja operanda i ekstrema, mogu krenuti na opisivanje rada programa kada dođe do operatora. Iako su operatori različiti, sličan algoritam se izvodi. Prvo treba pogledati ekstremne slučajeve. Prvi je kada se učita 13 operanda te se pohrane na stog. Potrebno je prvo vratiti sadržaj natrag u registre asemblerskom naredbom:

```
LDMDB R13!, {R0, R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11, R12}
```

pa tek onda generirati asemblersku naredbu (ovisno o operatoru):

zbrajanje:

```
ADD R11, R11, R12
```

oduzimanje:

```
SUB R11, R11, R12
```

množenje:

```
MUL R11, R11, R12
```

Drugi ekstrem je nešto kompliciraniji jer se mijenja redoslijed operanda. Da pojasnim, registre punimo redom od 0 do 12 pa ih stavimo na stog. Nakon toga nastavljamo s učitavanjem opet od registra R0. Ako nakon 14. operanda (ili 27. itd) dođe operator, trebati ćemo zadnji operand sa stoga. Učitati ćemo ga u registar R1 i time ćemo promijeniti redoslijed operanda.

```
LDR R1, [R13, #-4]  ; učitavamo operand u registar R1 sa adrese iz R13 – 4 bajta

SUB R0, R1, R0      ; oduzimao od registra R1 registar R0 te pohranjujemo rezultat u R0

STR R0, [R13]       ; pohranjujemo rezultat nazad na stog
```

R13 nismo mijenjali jer smo jedan operand uzeli i jedan vratili pa nakon ovih naredbi i dalje pokazuje na prvu praznu lokaciju nakon podataka. Umjesto naredbe SUB može biti naredba ADD ili MUL, ali jedino kod oduzimanja je važno koji je operand prvi a koji drugi. Uzmimo aritmetički izraz u postfiks notaciji: xy-. On označava: x – y. x mora biti prvi, a y drugi operand.

Ako nema ekstrema, dakle, ako su nam svi operandi u registrima, kada dođe operator, izvršiti će se naredba ovisno o operatoru:

```
ADD Rx, Rx, Ry ; x = broj učitanih operanda % 13 – 1, y = broj učitanih operanda %13

SUB Rx, Rx, Ry ; x = broj učitanih operanda % 13 – 1, y = broj učitanih operanda %13

MUL Rx, Rx, Ry ; x = broj učitanih operanda % 13 – 1, y = broj učitanih operanda %13
```

Pseudokod bi mogao ovako izgledati:

```
radi
{
    učitaj operator;
    ako je broj operanda djeljiv sa 12 i veći od 0
    {
            generiraj asemblersku naredbu koja će vratiti sadržaj registara;
            ovisno o operatoru generiraj asemblersku naredbu sa ispravnim redoslijedom registara;
    }
    inače ako je broj operanda veći od 1 i daje ostatak 1 pri dijeljenju s 12
    {
            učitaj operand sa stoga u registar R1;
            ovisno o operatoru generiraj asemblersku naredbu sa obrnutim redoslijedom registara;
            vrati operand na stog iz registra R0;
    }
}do kraja učitanog niza;
```

Operand +, - i * su ovdje, ali nedostaj operand /. To je zato jer sam entuzijastično se prihvatio ARM-a zaboravljajući da ARM nema naredbu za dijeljenje. Mislio sam da je ima (u sjećanju mi je ostala asemblerska naredba DIV), ali se čini da je to ipak bilo sjećanje iz srednje škole kad smo radili Motorolu 6800. Dakle, za tu naredbu je trebalo napraviti mali algoritam. Sjećate se da sam rekao da ARM ima na raspolaganju 15 registara? R13 sam odabrao za stog jer je to uobičajena praksa (*...registar R13 uobičajeno se koristi kao pokazivač na vrh stoga*...[2]), a registre od R0 do R12 za podatke. Preostao mi je još jedan registar R14 koji će mi sada poslužiti kao spremište rezultata dijeljenja. Asemblerski algoritam je standardni algoritam dijeljenja uzastopnim oduzimanjem i ide ovako:

```
; x = popunjenost_registara % 13 – 1

; y = popunjenost_registara % 13

        MOV R14, #0  ; prvo inicijaliziramo registar R14 (damo mu vrijednost 0)

DIV     ; labela na koju će se pozvati naredba za skok

        CMP Rx, Ry    ; uspoređuje registre Rx i Ry tako da oduzme od Rx Ry i osvježi zastavice

        BHI KRAJ      ; ako je Rx manji od Ry, dijeljenje je gotovo pa skačemo na labelu KRAJ

        SUB Rx, Rx, Ry          ; od Rx oduzimamo Ry i spremamo u Rx

        ADD R14, R14, #1      ; R14 uvećamo za 1

        B DIV             ; skok na labelu DIV

KRAJ

        MOV Rx, R14             ; u Rx spremamo rezultat dijeljenja
```

Taj cijeli algoritam se umeće u moj program tamo gdje inače dođe jedna asemblerska naredba za zbrajanje, oduzimanje ili množenje. Također treba paziti kod drugog ekstrema (kada je prvi operand na stogu a drugi u registu, pa se prvi prebaci u registar) da je Rx R1, a Ry R0, a ne obrnuto. Treba napomenuti da taj algoritam ne radi sa negativnim brojevima.

Kada se pročitaju svi operandi i operatori, na kraju se generira još jedan red koji označava kraj programa.

```
kraj    SWI 123456;
```

Poslije tog reda se trebaju sad navesti sve labele koje su bile spominjane u programu. Tome služi polje koje je pamtilo imena svih varijabli. Moj korisnički program će Vas pitati i da unesete vrijednost za te labele, pa ako ste imali varijablu vibi i dali ste joj vrijednost 13, generirati će se red:

```
vibi   DW %D13      ; vibi je labela, a %D označava da je 13 broj u dekadskoj bazi

; DW označava da će na toj memorijskoj lokaciji biti smještena jedna riječ (od 32 bita)
```

Na samom smo kraju i sad se samo treba sjetiti da R13 i dalje nema početnu vrijednost. U drugom redu asemblerskog programa treba dodati naredbu MOV i to je to.

```
        MOV R13, #%Dx       ; x je broj u dekadskoj bazi
```

Primjer jednog generiranog programa za ulaz: *vibi 1 /tri 8* *+

```
        `ORG 0
        MOV R13, #%D88
        LDR R0, vibi
        LDR R1, #1
        MOV R14, #0
DIV
        CMP R0, R1
        BHI KRAJ
        SUB R0, R0, R1
        ADD R14, R14, #1
        B DIV
KRAJ
        MOV R0, R14
        LDR R1, tri
        LDR R2, #8
        MUL R1, R1, R2
        ADD R0, R0, R1
kraj    SWI    123456
vibi    DW %D13
tri     DW %D4
```

4. Prilozi

U ZIP-u se nalaze datoteke:

- izraz_u_strojni_kod.c
- PPJ.exe
- primjeri.txt
- Vibor_Pokupec_0036438727.pdf
- Vibor_Pokupec_0036438727.docx (za svaki slučaj, ako pdf nije sasvim korektan)

Upute:

Pokrenite program PPJ.exe dvoklikom i upišite neki od primjera iz datoteke primjeri.txt. Svi primjeri rade, a možete i upisati svoje primjere. Generirati će Vam se dvije datoteke. Jedna je sk.a, a druga je sk1.a. sk.a sadrži ulazni niz, ali nema druge naredbe koja inicijalizira stog. sk1.a nema ulazni niz, ali ima inicijalizaciju stoga te je pravi asemblerski program (onaj koji se uzima u obzir).

Ako vam se slučajno pojavi sljedeći redak u asemblerskom kodu:

LDR R0,

pazite da niste upisali razmak iza operatora. Ako je iza operatora razmak, onda će se ovakav red sigurno dogoditi.

Ako ste upisali varijablu, onda će vas program na kraju i pitati za vrijednost te varijable. Na žalost, ako ste dvaput upisali istu varijablu, dvaput će vas pitati za vrijednost i neće generirati ispravan asemblerski kod. (Ispravan asemblerski kod nema dvije labele s istim imenom, a generirani asemblerski program će imati).

5. Problemi

Prilikom pisanja programa morao sam napraviti puno kompromisa zbog nedostatka vremena (generiranje ciljnog programa smo obradili u petak 8.1.2009.), zbog ograničenja ARM arhitekture i zbog svoje nesnalažljivosti u tome da nađem točne heksadekadske oblike asemblerskih naredbi.

Zbog nedostatka vremena nisam uspio napraviti provjeru za upisani niz. Dakle, kada se upisuje neki niz, mora se paziti da je u postfiks notaciji jer inače program neće ispravno raditi pa će generirati neupotrebljivi asemblerski program. Također trebate paziti da iza operanda obavezno ide razmak, a da iza operatora nema razmaka. Iskreno, malo me je sram predati Vam takav program, ali to je najbolje što sam stigao u tako kratkom vremenu. Do petka 8.1. nisam ni znao što točno trebam raditi. Tek je taj dan prof. dr. sc. Srbljić predavao generiranje ciljnog programa te posebno *Generiranje ciljnog programa na temelju postfiksnog sustava oznaka*.

Problem je predstavljala i druga asemblerska naredba. Nisam znao kako dodati 2. red (onaj u kojem upisujemo vrijednost adrese stogu), pa sam to riješio prepisivanjem cijele datoteke u drugu tako da bi „zastao“ sa prepisivanjem na drugom redu te upisao svoj red te nastavio sa prepisivanjem.

Pošto sam često testirao program, a cijelu datoteku na kraju i ovako prepisujem, onda sam odlučio u prvu datoteku staviti u prvom redu, na početku samog asemblerskog programa ulazni niz, tako da se može usporediti generirani program i zadani niz.

Najveći problem (uz nedostatak vremena) je bila nedostupnost podataka o asemblerskim naredbama, o arhitekturi i o strojnom kodu nekog procesora. Na internetu nisam mogao nigdje naći detaljan opis funkcije pojedinih naredbi niti strojni kod tih naredbi, a to mi je najviše trebalo.

I na kraju, nisam vam mogao predati strojni kod (ne prevoditelj, nego sam kod) jer Atlas ne radi u Visti i nisam imao sa čime prevesti asemblerski oblik u strojni.

6. Zaključak

Iako je izrada ovog SSP-a (zbog svih navedenih problema) bila jako stresna, uživao sam raditi prevoditelj algebarskog izraza u strojni kod. Uživao sam zato je sam konačno počeo shvaćati povezanost višeg jezika, asemblerskog jezika i strojnog jezika. Žao mi je što nisam upotrijebio regularne izraze za prepoznavanje varijabli, brojeva i operatora, jer da sam krenuo tim postupkom, program bi sigurno bio kvalitetniji pa ne bi ste trebali paziti na razmake prilikom upisivanja. Također mi je žao što znam samo C, vjerujem da u nekom drugom jeziku bi mogao uspoređivati imena labela tako da se ne ponavljaju iste labele te da program ne pita dvaput za istu labelu.

7. Literatura

[1] Siniša Srbljić, *Prevođenje programskih jezika,* Zagreb, 2007.
[2] M. Kovač / D. Basch, OSNOVE PROCESORA ARM, Zagreb, veljača 2004.