Univerzitet u Novom Sadu
Fakultet tehničkih nauka
Novi Sad
Departman za računarstvo i automatiku
Odsek za računarsku tehniku i računarsks ekomunikacije

RT-RK
COMPUTER BASED SYSTEMS

Projektni zadatak

MAVN Prevodilac

Mentor:
Miloš Subotić

Kandidat:
Isidora Poznanović
RA 163/2019

Novi Sad, jun, 2021.

Sadržaj

1 Uvod

U ovom radu bavimo se realizacijom MAVN prevodioca koji prevodi programe sa višeg asemblerskog jezika na osnovni MIPS 32bit asemblerski jezik. Prva faza prevođenja je leksička analiza koja izdvaja tokene iz ulaznog fajla i radi na principu *state machine*. Sledeća faza je sintaksna analiza koja implementira gramatiku jezika. Nakon koje se formiraju liste promenljivih, labela, funkcija i instrukcija. Naredna faza je analiza životnog veka promenljivih, koju sledi formiranje grafa smetnji. Što je neophodno realzovati radi dodele raspoloživih resursa.

2 Analiza problema

Potrebno je realizovati MAVN prevodilac koji prevodi programe sa višeg asemblerskog jezika na osnovni MIPS 32bit asemblerski jezik. Pri čemu vodimo računa o konceptu registarske promenljive koju uvodi MAVN jezik. Registarska promenljiva znatno olakšava pisanje asemblerskog koda, jer umesto pravog resursa (registra) koristimo promenljivu. Zadatak je dodeliti resurse za registarske promenljive sa ograničenjem na 4 registra.
Osnovna stvar koju ćemo morati da uradimo jeste da listu tokena koju dobijamo iz leksičke analize raspodelimo na osnovu tipa na memoriske promenljive, registarske promenljive i instrukcije.
Naš MAVN prevodilac podržava sledeće MIPS instrukcije:

- b - bezuslovni skok
- add - sabiranje
- lw - čitanje jedne memorijske reči
- addi - sabiranje sa konstantom
- sra - aritmetički pomeraj udesno
- bltz - skok ako je registar manji od nule
- jr - skok na adresu iz registra
- not - bitska negacija
- sw - upis jedne memorijske reči
- la - učitavanje adrese u registar

Nakon što se definišu osnovni pojmovi kao što su promenljive koje se u određenoj instrukciji koriste, one koje se definišu, prethodne instrukcije i instrukcije naslednice, sprovodimo analizu životnog veka. Kada odradimo analizu životnog veka sledeći korak je formiranje grafa smetnji. Nakon toga uprošćavamo graf smetnji stavljanjem promenljivh na stek po određenom algoritmu. Na kraju bojimo čorove grafa po redosledu utvrđenom na steku i napravimo izlazni fajl.

3 Koncept rešenja i programsko rešenje

U ovom poglavlju ćemo proći kroz ceo VSC projekat i navesti neke najbitnije klase i funkcije, kao i samu hijerarhiju projekta.

3.1 Leksička analiza

Leksička analiza izdvaja tokene iz ulaznog fajla i radi na principu *state machine*. U ovom projektu koristimo već pripremljenu leksičku analizu iz fajlova *LexicalAnalysis.cpp* i *LexicalAnalysis.h*. Uz to dobili smo heder fajl *Constants.h* u kome su definisane sve konstante koje se koriste u projektu. U ovom fajlu smo morali da promenimo broj stanja konačne mašine stanja NUM_STATES koju smo takođe dobili implementiranu i to u fajlovima *FiniteStateMachine.cpp* i *FiniteStateMachine.h*.

Konačna mašina stanja se koristi za pravljenje tokena, tako što se u zavisnosti od trenutnog stanja skače na sledeće stanje dok se ne naiđe na terminalni simbol. U fajl *Token.cpp* dodata su tri nova tokena T_SRA, T_JR i T_NOT, koje smo prethodno dodali u enumeraciju za tokene *TokenType*.

Sve enumeracije korišćene u programu se nalaze u fajlu *Types.h*.
Za registre koristimo *Regs* enumeraciju koja sadrži četiri podržana registra i inicijalno stanje. Za funkcije i Labele koristimo *LabelFunctionType* koja u sebi sadrži stanje funkcije, labele i inicijalno stanje. Slično za promenljive koristimo *VariableType* enumeraciju koja promenljive deli na memorijske i registarske. Pritom sadrži i inicijalno stanje. Pored toga postoji i tip *InstructionType* u koga smo dodali tri nove instrukcije I_SRA, I_JR i I_NOT.

3.2 Sintaksna analiza

Sintaksna analiza (klasa *SyntaxAnalysis*) je zadužena za proveru gramatike MAVN jezika, čiji su neterminalni simboli su Q, S, L i E. Svaki od ovih simbola iziskuje jednu funkciju u algoritmu sa rekurzivnim spuštanjem koji ćemo koristiti za realizaciju sintaksnog analizatora.

3.2.1 Gramatika MAVN jezika

$$Q \longrightarrow S; L$$

$$\begin{aligned} S &\longrightarrow \_mem\, mid\, num \\ S &\longrightarrow \_reg\, rid \\ S &\longrightarrow \_func\, id \\ S &\longrightarrow id : E \\ S &\longrightarrow E \end{aligned}$$

$$\begin{aligned} L &\longrightarrow eof \\ L &\longrightarrow Q \end{aligned}$$

$$\begin{aligned} E &\longrightarrow add\, rid, rid, rid \\ E &\longrightarrow addi\, rid, rid, num \\ E &\longrightarrow la\, rid, mid \\ E &\longrightarrow lw\, rid, num(rid) \\ E &\longrightarrow sw\, rid, num(rid) \\ E &\longrightarrow b\, id \\ E &\longrightarrow bltz\, rid, id \\ E &\longrightarrow not\, rid, rid \\ E &\longrightarrow jr\, rid \\ E &\longrightarrow sra\, rid, rid, num \end{aligned}$$

3.2.2 Algoritam

Kao što je već rečeno u ovom projektu koristili smo algoritam sa rekurzivnim spuštanjem koji je prilagođen navedenoj gramatici. U klasi *SyntaxAnalysis* imamo funkciju $bool\, Do()$ koja pokreće sintaksnu analizu i poziva funkciju $void\, Q()$ koja je najviša funkcija u rekurzivnom spuštanju. Funkcije Q, S, L i E ispituju pravila gramatike i prijavljuju sintaksnu grešku ukoliko naiđu na pravilo koje ne postoji. Funkcija $void\, eat()$ dobija token i ona je ta koja proverava da li se token očekuje ili se prijavljuje greška.

3.3 Kreiranje promenljivih, funkcija i labela

Po završetku sintakne analize formiramo liste promenljivih, funkcija i labela. U programu sve se odvija u fajlovima *IR.cpp* i *IR.hpp*, gde su definisane klase *Variable* i *LabelFunction*. Svaka klasa pored polja i konstruktora sadrži getere, setere i preklopljen operator ispisa. Pri formiranju promenljivih, labela i funkcija sve vreme koristimo globalno definisane liste.

Kreiranje promenljivih, funkcija i labela se vrši preko funkcije *void createVariablesLablesandFunctions(LexicalAnalysis& lex)*. Ova funkcija prolazi kroz sve tokene i razvrstava ih na registarske promenljive, memorijske promenljive, funkcije i labele. Za svaku proverava da li je već definisana i da li joj je dodeljen korektan format naziva. Takođe prolaskom kroz tokene povećavamo i globalni brojač linije koda pri nailasku na dvotačku ili tačku-zarez. Kada prođemo kroz sve tokene pozivamo funkciju koja iz liste promenljivih izdvaja registarske.

3.4 Kreiranje instrukcija

Pri kreiranju instrukcija ponovo prolazimo kroz listu tokena i u zavisnosti od toga na koji token naiđemo izvršavamo određenu akciju. Pre svega resetujemo globalni brojač linija na jednicu i kasnije ga povečavamo ukoliko naiđemo na odgovarajuće tokene. Ako, pri prolaženju kroz tokene, pronađemo neki koji predstavlja instrukciju pravimo novi pokazivač na objekat klase *Instruction* i dodajemo ga na globalnu listu instrukcija.

Klasa *Instruction* se razlikuje od prethodne dve po tome što ima više polja, značajnih za dalji tok prevodjenja.
Samo kreiranje instrukcija vrši funkcija *void createInstructionList(LexicalAnalysis& lex)*, koja kao što je naznačeno prolazi kroz tokene i kada dođe do odgovarajućeg tokena popunjava destinaciju i izvor promenljivama kao sto je naznačeno u gramatici. Zatim formira string koji odgovara instrukciji u asemblerskom kodu sa tilda karakterima koji će nam označiti mesta na koja kasnije stavljamo registre, labele i memorijske promenljive. Kada prođemo kroz sve tokene pozivamu funkciju za popunjavanje skpova prethodnika i sledbenika, kao i funkciju za popunjvanje *use* i def skupova.

3.5 Analiza životnog veka

Analizu životnog veka obavljamo u funkciji *void printInstructions()*. Njen zadatak je da popuni *in* i *out* liste za svaku instrukciju. Korišćen je algoritam sa vežbi. U principu imamo flag koji nam ukazuje na to kada prestajemo sa prolaskom kroz instrukcije što se dešava kada dodjemo do toga da su prethodno i sledeće stanje pokazivača na in i out liste isti. A do tad ubacujemo nove promenljive u *in* i *out* liste, tako što u *out* listu ubacujemo sve promenljive iz *in* liste njenog sledbenika a u *in* listu se ubace sve promenljive iz njene *out* liste bez elemenata koji nisu u njenoj def listi i ubace se još i elementi *use* liste.

3.6 Dodela resursa

Sve faze do sada su podrazumevale da u ciljanoj platformi imamo neograničen broj registara. U ovoj fazi dodele resursa taj neograničen broj registara moramo svesti na broj raspoloživih.

3.6.1 Faza formiranja

Formira se graf smetnji na osnovu analize životnog veka promenljivih. To se odvija u funkciji *void doInterferenceGraph()*. Princip je sledeći: za svaku instrukciju gledamo šta se definiše i šta je na izlazu

čvora. U matricu smetnji zapisujemo sve što je živo na izlazu a ne definiše se tj. smetnju između toga i ovoga što se definiše.

3.6.2 Faza uprošćavanja

U ovoj fazi na stek stavljamo čvorove grafa smetnji po algoritmu opisanom u funkciji *bool doSimplification(int regNum)* kojoj kao parametar prosleđujemo konstantu $\_\_REG_NUMBER\_\_$. Funkcija vraća *true* ako je uspela da formira graf smetnji, u suprotnom naš program će se završiti uz ispis greške.

3.6.3 Faza prelivanja

Ova faza bi se sprovodila ako se graf ne može uprostiti. Pošto nemamo dovoljano resursa morali bismo da prebacujemo nešto u memoriju. Ovu fazu nismo sprovodili jer to od nas nije bilo zahtevano.

3.6.4 Faza izbora

Nakon što je graf uspešno uprošćen potrebno je obojiti čvorove. To implementira funkcija *bool doResourceAllocation()*.

3.7 Pravljenje izlaznog fajla

Globalne promenljive koje su korišćene za ovaj deo programa su g_output_globl, g_output_data i g_output_data. One su mape čiji je ključ pozicija zadate promenljive, funkcije ili labele a vrednost je string koji predstavlja ekvivalentnu instrukciju. Funkcija *string makeInstructionString(Instruction* i)* od pokazivača na instrukciju pravi string ekvivalentan zadatoj instrukciji. Izlazni fajl se kreira na kraju funkcije *void printOutputString(string out)* kojoj se kao parametar zadaje ime izlaznog fajla bez ekstenzije.

4 Verifikacija

Na primeru *simple.mavn* datoteke, još jednom ćemo proći kroz čitav proces i na kraju ćemo navesti izlaze za jos par primera.

```
_mem m1 6;
_mem m2 5;

_reg r1;
_reg r2;
_reg r3;
_reg r4;
_reg r5;

_func main;
        la          r4,m1;
        lw          r1, 0(r4);
        la          r5, m2;
        lw          r2, 0(r5);
        add         r3, r1, r2;
```

Datoteka 4.1: simple.mavn

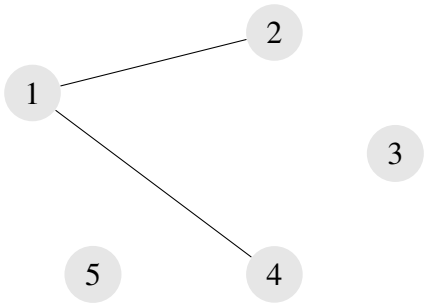
- Pri pokretanju programa, potrebno je kao argumente proslediti naziv ulaznog i izlaznog fajla bez ekstenzija (nije potrebno prethodno formirati izlazni fajl). U ovom radu nećemo navoditi sve ispise programa već samo izlazni fajl.
- Leksička i sintaksna analiza prolaze bez grešaka kao što je i očekivano, uz propratne ispise tokena, koje kao što je već rečeno nećemo navoditi ovde. Posle toga su kreirane sve funkcije, labele, promenljive i instrukcije, nakon čega prelazimo na analizu životnog veka.
- Nakon analize životnog veka dobijamo ponovni ispis svih instrukcija. Ručno izračunavamo tabelu prethodnika, sledbenika, use, def, in i out skupova, kao na vežbama. Ona se poklapa sa ispisima programa i izgleda ovako:

	pred	succ	use	def	in	out
1	/	2	/	r_4	/	/
2	1	3	r_4	r_1	r_4	r_1
3	2	4	/	r_5	r_1	r_1 r_5
4	3	5	r_5	r_2	r_5 r_1	r_1 r_2
5	4	/	r_1 r_2	r_3	r_1 r_2	/

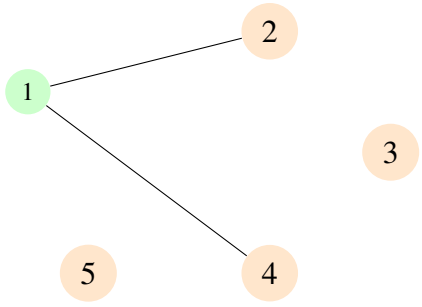
- Zatim određujemo matricu smetnji na osnovu koje crtamo graf smetnji. Ispostavlja se da je u pitanju ista matrica smetnji koju program ispisuje. U preseku onih vrsta i kolona za promenljive između kojih postoje smetnje unosimo 1, u suptornom 0:

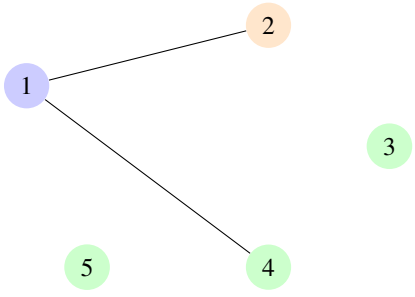
	r_1	r_2	r_3	r_4	r_5
r_1	0	1	0	0	1
r_2	1	0	0	0	0
r_3	0	0	0	0	0
r_4	0	0	0	0	0
r_5	1	0	0	0	0

- Graf smetnji:



Primećujemo da se graf smetnji može obojiti na više načina, na primer:





- Naše bojenje grafa odgovara prvoj slici, gde je graf obojen sa dve boje, dok je u postavci zadatka dato rešenje u kome je obojen graf na drugi prikazan način. Oba načina su korektna. U nastavku teksta prilažemo naš izlazni fajl kao i još par ukratko objašnjenih primera.

```
.globl main

.data
m1:     .word 6
m2:     .word 5

.text
main:
    la      $t3, m1
    lw      $t2, 0($t3)
    la      $t3, m2
    lw      $t3, 0($t3)
    add     $t3, $t2, $t3
```

Datoteka 4.2: outSimple.s

4.0.1 Još neki primeri

Primeri dati u ovom delu nemaju nikakav algoritamski značaj. Njihova svrha je bila jedino testiranje ispravnosti programa.

U ovom primeru je dat još jedan primer koda koji se uspešno prevodi na osnovni MIPS 32bit asemblerski jezik.

```
_mem m1 6;
_mem m2 5;
_mem m3 0;
_mem m4 1;

_reg r1;
_reg r2;
_reg r3;
_reg r4;
_reg r5;
```

```
_reg r6;
_reg r7;
_reg r8;

_func main;
        la              r4,m1;
        lw              r1, 0(r4);
        la              r5, m2;
        lw              r2, 0(r5);
lab:
    la      r6, m3;
    addi    r7, r6, 1;
    bltz    r8, lab;
```

Datoteka 4.3: test.mavn

```
.globl main

.data
m1:     .word 6
m2:     .word 5
m3:     .word 0
m4:     .word 1

.text
main:
    la      $t2, m1
    lw      $t2, 0($t2)
    la      $t2, m2
    lw      $t2, 0($t2)
lab:
    la      $t2, m3
    addi    $t2, $t2, 1
    bltz    $t3, lab
```

Datoteka 4.4: outTest.s

U ovom primeru, naš algoritam u fazi dodele resursa ne uspeva da uprosti i oboji graf smetnji sa

samo četiri registra. Ispisuje grešku u kojoj navodi da ne podržava prelivanje jer to nije ni traženo u ovom zadatku.

```
_mem m1 6;
_mem m2 44;
_mem m3 36;
_mem m4 24;
_reg r1;
_reg r2;
_reg r3;
_reg r4;
_reg r5;
_reg r6;
_reg r7;
_reg r8;
_reg r9;
_func main;
    la      r2, m1;
    sra     r1, r2, 2;
    la      r4, m2;
    sra     r3, r4, 2;
    la      r6, m3;
    sra     r5, r6, 2;
    la      r8, m4;
    sra     r7, r4, 2;
    add     r9, r1, r3;
```

Datoteka 4.5: testSpill.mavn

Sledeći primer ne uspeva da prevede fajl zato što koristimo nedefinisanu memorijsku promenljivu *m*_4, što nam program i ispisuje.

```
_mem m1 6;
_mem m2 5;
_mem m3 0;

_reg r1;
_reg r2;
```

```
_reg r3;
_reg r4;
_reg r5;
_reg r6;
_reg r7;
_reg r8;

_func main;
        la              r1, m1;
        lw              r2, 0(r1);
        la              r3, m2;
        lw              r4, 0(r3);
        la              r5, m3;
        la              r6, m4;

lab:
        la              r6, m3;
        addi    r5, r5, 1;
        bltz    r7, lab;

        la              r8, m3;
        sw              r6, 0(r8);
```

Datoteka 4.6: testError.mavn

Literatura

[1] Miroslav ukić. *Materijali sa vežbi i predavanja*. Available from: `https://www.rt-rk.uns.ac.rs/`.

[2] Miroslav Popović Vladimir Kovačević. *Sistemska programska podrška u realnom vremenu 1*. FTN Izdavaštvo, 2013.