UNIVERZITET U NOVOM SADU
FAKULTET TEHNIČKIH NAUKA
U NOVOM SADU

Projekat - MAVN prevodilac

Autor: Filip Čonić
Broj Indeksa: RA 126/2022

Predmet: Osnovne paralelnog programiranja i softverski alati

Novi Sad, jun, 2024.

Sadržaj:

1. Uvod

Tema projekta je MAVN (MIPS Asembler Visokog Nivoa) prevodilac, odnosno alat koji uzima izvornu tekst datoteku i nad njom vrši leksičku i sintaksnu analizu, a potom ga, ukoliko je napisan u obliku koji prati sintaksu MAVN jezika, pretvara u asemblerski kod namenjen za MIPS32 arhitekturu.

Najbitnija stvar koju uvodi generalizacija MIPS32 asemblerskog koda na MAVN asemblerski jezik je uvođenje registarskih promenljivih. Registarske promenljive jesu one promenljive koje su tokom celog programa čuvane u jednom registru dok su od koristi. Uvođenjem registarskih promenljivih pisanje koda za MIPS32 arhitekturu je olakšano jer nije potrebno pratiti koja promnljiva koristi koji registar u određenom delu programa, već je moguće dodeliti posebne oznake neograničenom broju promenljivih. Međutim, problem može predstavljati korišćenje više od ograničenog broja registara, bilo to zbog fizičkog ograničenja arhitekture ili zbog veštačkog ograničenja datog u programu.

Iako MIPS32 arhitekutra podržava veliki broj instrukcija, MAVN prevodilac je ograničen na manji skup tih instrukcija. U slučaju ovog projekta veličina skupa tih instrukcija je 13, a tih 13 instrukcija su:

- add - sabiranje
- addi - sabiranje sa konstantom
- sub - oduzimanje
- la - učitavanje memorijske lokacije
- li - pomeranje konstante u registar
- lw - učitavanje vrednosti sa memorijske adrese
- sw - stavljanje vrednosti na memorijsku adresu
- b - bezuslovni skok na labelu
- bltz - skok na labelu u slučaju da je vrednost u datom registru manja od nula
- nop - prazna operacija
- and - logičko i između parova bitova iz dva prosleđena registra
- or - logičko ili između parova bitova iz dva prosleđena registra
- not - logičko ne na svakom bitu prosleđenog registra

2. Analiza problema

Svrha prevodioca je da analizira leksičku, semantičku i sintaksnu tačnost ulazne tekst datoteke i dalje je pretvori u asemblerski kod, odnosno tekst datoteku koja predstavlja asemblerski kod za MIPS32 arhitekturu.

Prvi deo čitanja datoteke je pretvaranje napisanih znakova u simbole koji predstavljaju zapisane reči kao objekte koje program razume. Na primer, reč *_reg* se pretvara u simbol *T_REG* što znači da se na toj poziciji nalazi simbol koji predstavlja komandu za kreiranje registra. U suprotnom, razmaci nemaju konkretno značenje i oni se ne pretvaraju u simbole.

Jezik MAVN ima određenu strukturu, odnosno sintaksu koju mora da prati. Ova struktura se naziva gramatika programskog jezika. Gramatika programskog jezika prati određena pravila. Na primer, nakon linije koda potrebno je da se nalazi simbol *;* i da u instrukciji u kojoj pravimo registar moramo dati ime registru. Gramatika izgleda kao drvo koje se širi dok ne stigne do terminalnih simbola, odnosno dok rastavljanje programa ne stigne do simbola koji nemaju dalje razlaganje i imaju jedinstveno značenje. Na primer, *eof* simbol predstavlja kraj datoteke i on se ne pretvara u dalje simbole. Suprotno tome je simbol za liniju koda Q koja može značiti više stvari i dalje se razlaže na moguće simbole.

Gramatika MAVN programskog jezika izgleda ovako:

Q -> S *;* L

Q -> *comment* L

L -> Q

L -> *eof*

S -> *_func id*

S -> *_reg reg_id*

S -> *_mem mem_id num*

S -> *id :* E

S -> E

E -> *add reg_id , reg_id , reg_id*

E -> *addi reg_id , reg_id , num*

E -> *sub reg_id , reg_id , reg_id*

E -> *la reg_id , mem_id*

E -> *lw reg_id , num (reg_id)*

E -> *li reg_id , num*

E -> *sw reg_id , num (reg_id)*

E -> *b id*

E -> *bltz reg_id , id*

E -> *nop*

E -> *and reg_id , reg_id , reg_id*

E -> *or reg_id , reg_id , reg_id*

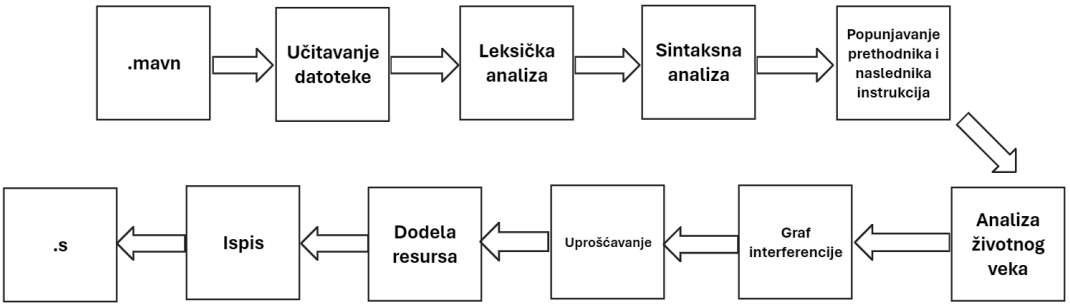
E -> *not reg_id , reg_id*

Terminalni simboli su:
, : ; ( )
comment eof _func _reg _mem id num reg_id mem_id
add addi sub la lw li sw b bltz nop and or not

3. Rešenje problema

Postoje određeni koraci kroz koje prevodilac prolazi da bi pretvorio ulaznu *.mavn* datoteku u izlaznu asemblersku *.s* datoteku.

Prvi korak jeste učitavanje datoteke i leksičke analize gde se pravi lista simbola i proverava da li svi simboli pripadaju listi mogućih simbola. Dalje, sintaksnom analizom od liste simbola se kreiraju instrukcije i proverava sintaksna tačnost ulazne datoteke. Nakon sintaksne analize se svakoj instrukciji odrede njeni prethodnici i naslednici, odnosno redosled izvršavanja instrukcija. Takođe, u ovom koraku se popunjavaju liste promenljivih koje se koriste i liste onih koje se definišu u određenoj instrukciji. Nakon pripreme instrukcija, vrši se analiza životnog veka gde se određuju promenljive koje su ulazi i izlazi instrukcija, odnosno da li je potrebno čuvati vrednosti određene promenljive tokom jedne instrukcije radi daljeg korišćenja u nekoj drugoj instrukciji. Kada su određeni ulazi i izlazi svake instrukcije gleda se da li dolazi do preplitanja korišćenja promenljivih, i za svako preplitanje u graf koji sadrži sve promeljive kao čvorove se dodaje veza između čvorova promenljivih koje se prepliću. Koristeći graf se pravi simplifikacioni stek ili stek uprošćavanja koji sadrži promenljive poređane redom od onih kojima se može najjednostavnije dodeliti neki registar, imaju najmanje preplitanja, do onih kojima je teže, imaju više preplitanja. Sa steka se uzimaju jedna po jedna promeljiva s vrha i dodeljuju im se slobodni registri. Kad su dodeljeni svi registri svim promenljivama vrši se ispis u *.s* datoteku u vidu asemblerskog koda gde se imena svih registarskih promenljivih menjaju sa registrima koji su im dodeljeni.



4. Programsko rešenje

4.1. main

Program se izvršava počevši od ove datotke. U njoj su date ulazne datoteke i prave se objekti koji vrše obradu podataka. Pomoću try-catch bloka vrši se provera greške. U slučaju neke greške koja nosi manje informacija podiže se *runtime_error* iz standardne C++ biblioteke. Kod sintaksne analize koriste se posebne greške koje nose više informacija.

4.2. Token

Klasa token predstavlja omotač za jedan simbol.

Polja:

- *TokenType tokenType* - tip simbola koji se nalazi u objektu
- *string value* - ime simbola ili vrednost kao broj u slučaju simbola broja

Metodi:

- *void makeToken(int begin, int end, std::vector<char>& program, int lastFiniteState)* - metod koji uzima vektor karaktera, početak simbola unutar vektora i koji je bilo prethodno stanje programa, i popunjava polja tokena sa vrednostima pročitanim iz vektora karaktera
- *void makeErrorToken(int pos, std::vector<char>& program)* - postavalja polja tokena na simbol greške u slučaju pogrešnog simbola i čuva unutar polja *value* vrednost koja je predstavljala grešku
- *void makeEofToken()* - postavlja polja tokena na vrednosti vezane za kraj datoteke
- *void printTokenInfo()* - ispisuje vrednosti koje se nalaze u poljima tokena
- *void printTokenValue()* - ispisuje vrednost koja se nalazi u polju *value*
- *string tokenTypeToString(TokenType t)* - vraća tip tokena u obliku string-a

4.3. LexicalAnalysis

Klasa LexicalAnalysis kreira objekat unutar kojeg se vrši leksička analiza datoteke i priprema se lista simbola za dalju analizu. Koristi konačan automat stanja da čita karakter po karakter i reči pretvara u simbole, ako prelazak stanja u konačnom automatu nije ispravan pravi se simbol greške.

Polja:

- *ifstream inputFile* - ulazna datoteka
- *vector<char> programBuffer* - vektor svih karaktera iz datoteke
- *unsigned int programBufferPosition* - trenutna pozicija čitanja vektora
- *FiniteStateMachine fsm* - konačan automat stanja
- *TokenList tokenList* - lista pročitanih tokena
- *Token errorToken* - token u kojem se nalazi token greške u slučaju da se dogodi greška

Metodi:

- *void initialize()* - priprema sam objekat i konačan automat
- *bool Do()* - metod koji pokreće leksičku analizu

- *bool readInputFile(string fileName)* - učitava ulaznu datoteku
- *Token getNextTokenLex()* - čita sledeću reč na redu i vraća simbol
- *void printTokens()* - ispisuje sve simbole na terminal
- *void printLexError()* - ispisuje simbol sa grečkom u slučaju greške
- *void printMessageHeader()* - ispisuje poglavlje ispisa pre *printTokens()* metoda

4.4. IR

Unutar datoteke *IR.h* se nalaze dve klase, klasa *Variables* i klasa *Instructions*. Ove dve klase unutar sebe drže sve potrebne informacije za dalju analizu, obradu i ispis.

Variable

Enumeracija *VariableType*:

- MEM_VAR - memorijska promenljiva koja u sebi čuva vrednost na memorijskoj lokaciji
- REG_VAR - registarska promenljiva koja je dodata u MAVN prevodiocu
- LABEL_VAR - promenljiva koja čuva ime labele
- CONST_VAR - promenljiva koja unutar sebe drži konstantnu vrednost, odnosno broj
- NO_TYPE - tip u slučaju da nije dodeljen ni jedan drugi promenljivoj

Polja:

- *static unsigned counter* - brojač na nivou cele klase koji broji koliko registarskih promenljivih postoji i koristi se za postavljanje pozicije regisitarske promenljive u korist grafa interferencije
- *int value* - vrednost koja se čuva unutar promenljive, takođe je koristi label promenljiva da pokaže da li postoji i da li je valido da neka druga label promenljiva pokazuje na nju
- *VariableType m_type* - tip promenljive
- *string m_name* - ime promenljive
- *int m_position* - pozicija registarske promenljive u grafu interferencije
- *Regs m_assignment* - registar dodeljen registarskoj promenljivoj

Metodi:

- *friend std::ostream& operator<<(std::ostream& out, Variable& var)* - prijateljska funkcija koja ispisuje promenljivu na terminal
- *friend void print(std::list<Variable*>& vars)* - prijateljska funkcija koja ispisuje listu promenljivih na terminal
- *string printType()* - ispisuje tip promenljive
- *void printTable()* - ispisuje ceo sadržaj promenljive na terminal, formatirano

Instruction

Enumeracija *InstrucitonType*:

- I_NO_TYPE - instrukcija bez tipa, takođe se koristi za instrukciju koja čuva funkciju
- I_ADD - *add* instrukcija
- I_ADDI - *addi* instrukcija
- I_SUB - *sub* instrukcija
- I_LA - *la* instrukcija

- I_LI - *li* instrukcija
- I_LW - *lw* instrukcija
- I_SW - *sw* instrukcija
- I_BLTZ - *bltz* instrukcija
- I_B - *b* instrukcija
- I_NOP - *nop* instrukcija
- I_AND - *and* instrukcija
- I_OR - *or* instrukcija
- I_NOT - *not* instrukcija

Polja:

- *static unsigned counter* - brojač instrukcija na nivou cele klase, koristi se za dodeljivanje broja narednoj novoj instrukciji
- *Variable* label* - pokazivač na labelu u slučaju da funkcija ima labelu
- *int m_position* - redni broj instrukcije u programu
- *InstructionType m_type* - tip instrukcije
- *Variables m_dst* - lista odredišnih registarskih promenljivih
- *Variables m_src* - lista izvornih registarskih promenljivih
- *Variables m_use* - lista promenljivih koje instrukcija koristi
- *Variables m_def* - lista promenljivih koje instrukcija definiše
- *Variables m_in* - lista ulaznih promenljivih
- *Variables m_out* - lista izlaznih promenljivih
- *list<Instruction*> m_succ* - lista instrukcija koje su naslednici date instrukcije
- *list<Instruction*> m_pred* - lista instrukcija koje su prethodnici date instrukcije

Metodi:

- *Variables getSuccIns()* - vraća listu svih ulaznih promenljivih instrukcija koje su naslednici date instrukcije, koristi se za životnu analizu
- *Variables getUseWithOutWithoutDef()* - vraća listu promenljivih koja predstavlja uniju ulaznih promenljivih sa izlaznim promenljivama koje nisu takođe u listi definisanih, koristi se za životnu analizu
- *friend Instruction* findInstructionWithLabel(Variable* lab, list<Instruction*>& ins)* - pronalazi instrukciju sa određenom labelom
- *friend Instruction* findInstructionAfterFunc(Instruction* in, list<Instruction*>& ins)* - pronalazi instrukciju koja dolazi posle instrukcije koja predstavlja funkciju
- *friend void addEachother(Instruction& successor, Instruction& predecessor)* - dodeljuje dvema instrukcijama jedna drugu kao naslednika i sledbenika
- *bool isFunc()* - vraća da li je instrukcija funkcija, odnosno da li nema tip i ima labelu
- *string toString()* - pretvara tip instrukcije u string
- *friend std::ostream& operator<<(std::ostream& out, Instruction& in)* - operator za ispis u izlaznu datoteku koji formatira instrukciju da bude ispravna za MIPS32 arhitekturu
- *friend void print(std::list<Instruction*>& ins)* - ispisuje listu instrukcija na terminal
- *void printTable()* - ispisuje ceo sadržaj instrukcije na terminal, formatirano

4.5. SyntaxAnalysis

Klasa SyntaxAnalysis vrši sintaksnu analizu prateći gramatiku jezika MAVN. Dobija listu simbola od napravljene tokom leksičke analize. Unutar ove klase se nalazi iterator liste simbola koji pokazuje na početak liste simbola. Ovaj iterator predstavlja trenutno gledan simbol i uzima ga jedan po jedan (jede ga metodom *eat()*, odnosno vrši se provera da li je očekivani simbol taj koji je na redu). Ako su išli redom očekivani metodi proizvodi se instrukcija koja predstavlja napisanu liniju. Unutar sintaksne analize se takođe nalazi enumeracija za sintaksne greške i sve greške u sintaksnoj analizi podižu greške ovog tipa. Ovo je rađeno radi veće preciznosti u tipu greške koja je nastala.

Polja:

- *LexicalAnalysis& lex* - referenca na prethodno odrađenu leksičku analizu
- *TokenList::iterator currentToken* - iterator koji pokazuje na trenutno gledan simbol
- *Instructions instrs* - lista instrukcija
- *Variables reg_vars* - lista registraskih promenljivih
- *Variables mem_vars* - lista memorijskih promenljivih
- *Variables label_vars* - lista labela
- *Variables const_vars* - lista konstanta koje se čuvaju unutar promenljivih
- *bool err* - povratna vrednost koja predstavlja da li je došlo do greške tokom sintaksne analize
- *bool eof* - povratna vrednost koja predstavlja da li je pročitan simbol koji predstavlja kraj datoteke
- *bool next_instruction_has_label* - vrednost koja vraća tačno ako je pročitana labela i ako je potrebno narednoj instrukciji dodeliti labelu

Metodi:

- *bool Do()* - pokreće sintaksnu analizu, vraća da li je sintaksna analiza uspešno izvršena
- *void printInstructions()* - ispisuje sve instrukcije na terminal
- *void printVariables()* - ispisuje sve promenljive na terminal
- *void eat(TokenType token)* - proverava da li je trenutno gledan simbol onaj koji je prosleđen metodu i podiže grešku ako nije, u suprotnom, pomera iterator da gleda na sledeći token
- *void glance(TokenType token)* - proverava da li je trenutno gledan simbol onaj koji je prosleđen metodu i podiže grešku ako nije
- *void regVariableExists(std::string& name)* - proverava da li već postoji registarska promenljiva sa prosleđenim imenom
- *void memVariableExists(std::string& name)* - proverava da li već postoji memorijska promenljiva sa prosleđenim imenom
- *void labelExists(std::string& name)* - proverava da li već postoji labela sa prosleđenim imenom
- *Variable* createVariable()* - uzima trenutno gledan simbol i pretvara ga u promenljivu odgovarajućeg tipa
- *Variable* findVariable()* - traži promenljivu istog imena kao trenutno gledan simbol
- *Variable* constVariable(int value)* - traži konstantnu promenljivu i u slučaju da ne postoji prosleđena data vrednost kreira novu sa tom vrednošću
- *Variable* findVariable(std::string& name)* - traži promenljivu sa prosleđenim imenom
- *Variable* findLabel(std::string& name)* - traži labelu sa prosleđenim imenom

- void checkLabels() - proverava da li postoje sve kreirane labele
- *void Q()* - čita liniju koda
- *void L()* - čita kraj linije koda
- *void S()* - čita sadržaj linije koda
- *void E()* - čita izraz

4.6. LivenessAnalysis

Klasa LivenessAnalysis radi više manjih delova prevodioca za kraj. Prvo vrši životnu analizu, gde određuje ulaze i izlaze svake instrukcije. Životna analiza se vrži iterativno, dok nisu ulazi i izlazi iz *n*-te iteracije jednaki ulazima i izlazima iz *(n-1)* iteracije, za svaku instrukciju. Izlazi treba da budu jednaki uniji ulaza svih naslednika instrukcije, dok su ulazi unija korišćenih promenljiivih i onih izlaznih koje nisu u listi definisanih. Sledeći korak jeste popunjavanje grafa interferencije. Za svaku definisanu promenljivu u instrukciji dodaje se interferencija sa svakom drugom promenljivom u listi izlaznih promenljivih. Potom, kreira se simplifikacioni stek. U simplifikacioni stek se preuzimaju čvorovi sa grafa, po pravilu da se čvor sa najvećim rangom manjim od broja registara prvi uklanja sa grafa. Poslednji korak je bojenje, odnosno dodela registara registarskim promenljivama. Promenljive se sa simplifikacionog steka skidaju jedna po jedna i dodeljuju im se slobodni registri, tako da se ne prepliću ni sa jednom drugom registarskom promenljivom tokom rada.

Polja:

- bool err - povratna vrednost koja predstavlja da li je došlo do greške
- Variables& reg_vars - referenca na listu registarskih promenljivih
- Variables& mem_vars - referenca na listu memorijskih promenljivih
- Variables vars - lista promenljivih koja se koristi za čuvanje dodeljenih registara
- Instructions& instrs - referenca na listu instrukcija
- vector<vector<int>> interferenceGraph - dvodimenzionalni vektor koji predstavlja graf interferencije, ako je 0 na određenoj poziciji onda se dva registra ne prepliću, ako je 1 onda se prepliću

Metodi:

- bool Do() - funkcija koja izvršava životnu analizu, popunjavanje grafa interferencije i alokaciju resursa, vraća da li je došlo do greške tokom nekog koraka
- void printRegisters() - ispisuje sve registarske promenljive na terminal
- void printGraph() - ispisuje graf interferencije na terminal
- void writeToFile(string& nameOfOutputFile) - ispisuje asemblerski kod *.s* u datoteku sa prosleđenim imenom
- void setPredAndSucc() - popunjava liste naslednika i prethodnika svih instrukcija
- void setUseAndDef() - popunjava korišćene i definisane promenljive svih instrukcija
- void liveness() - izvršava životnu analizu
- void setInterference(int x, int y) - pravi vezu između promenljivih registarskih promenljivih na pozicijama x i y
- void setGraph() - popunjava graf sa interferencijama
- stack<Variable*> createSimplificationStack() - proizvodi simplifikacioni stek za već popunjenu matricu interferencije
- int getColor(Variable* var) - vraća vrednost validnog registra u poređenju sa ostalim zauzetim registrima, odnosno u odnosu na već alocirane registarske promenljive
- void resourceAllocation() - vrši alokaciju registara

5. Primer

Data su nam dva primera *.mavn* fajlova i ovo je kako izgledaju njihova rešenja.

simple.mavn

```
_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;
```

simple.s

```
.globl main

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

.text
main:
        la $t0, m1
        lw $t0, 0($t0)
        la $t2, m2
        lw $t1, 0($t2)
        add $t0, $t0, $t1
```

multiply.mavn

```
_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);
        li      r5, 1;
        li      r6, 0;
lab:
        add     r6, r6, r2;
        sub     r7, r5, r4;
        addi    r5, r5, 1;
        bltz    r7, lab;

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

multiply.s

```
.globl main

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

.text
main:
        la $t0, m1
        lw $t3, 0($t0)
        la $t0, m2
        lw $t4, 0($t0)
        li $t1, 1
        li $t2, 0
lab:
        add $t2, $t2, $t3
        sub $t0, $t1, $t4
        addi $t1, $t1, 1
        bltz $t0, lab
        la $t0, m3
        sw $t2, 0($t0)
        nop
```

Postoji problem sa *multiply.mavn* datotekom ovakvom kakva jeste, jer je nije moguće prevesti u asemblerski kod sa samo 4 registra, jer je potrbno da se u jednom momentu čuva 5 vrednosti. Ovo se lako može ispraviti dodavanjem još jednog registra.

6. Zaključak

Rešavanje problema prevodioca na prvi pogled možda izgleda veoma zastrašujuće, ali prevodilac nije ništa drugo sem parser jedne datoteke i potraga za određenim izlazom koji prati određena pravila. U ovom slučaju ta pravila koja naš program mora da prati su leksička, semantička i sintaksna analiza. Mnogi ljudi su se već bavili ovim problemima i oni nam daju primere s kojih možemo učiti. Kreiranje sopstvenog prevodioca nije lak zadatak, ali zadatak koji zahteva rad sa velikim objektima, strukturama i algoritmima, što je sve što unapređuje naš rad i naše znanje kao programera.