УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА

УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА
НОВИ САД
Департман за рачунарство и аутоматику
Одсек за рачунарску технику и рачунарске комуникације

ПРОЈЕКТНИ РАД

Кандидат: Ана Гавриловић
Број индекса: RA 65/2018

Предмет: Системска програмска подршка у реалном времену I
Тема рада: MAVH преводилац

Ментор рада: др Миодраг Ђукић

Нови Сад, јун, 2020.

Sadržaj

1. Uvod

Tema projekta je realizacija MAVN prevodioca koji prevodi programe sa MIPS 32bit višeg asemblerskog programskog jezika na osnovni asemblerski jezik MIPS arhitekture. Program kao ulaz dobija kod na MAVN jeziku, a kao izlaz daje ispravan asemblerski kod koji se može izvršavati na MIPS 32bit arhitekturi. Ako postoje greške u ulaznom fajlu, leksičke, sintaksne, semantičke ili bilo kojih drugih koje se dogode tokom prevođenja, one će biti prijavljene kako bi se lakše uočile i ispravile.

2. Analiza problema

Instrukcije MAVN programskog jezika i asemblerskog jezika su iste. Bitna razlika između ova dva jezika je ta, što MAVN jezik uvodi koncept regisarskih promenljivih. Time je programeru olakšan posao pisanja koda jer ne mora da vodi računa o slobodnim registrima računara, već samo koristi promenljive koje je prethodno definisao. Problem je u tome što se, pri pisanju koda, može desiti da se koristi više registarskih promenljivih nego što ima stvarnih registara. Prevodilac treba da obezbedi dodelu stvarnih registara registarskim promenljivama ukoliko je to moguće. Ako nije, treba da prijavi grešku. Potrebno je izgenerisati izlaznu datoteku, u kojoj će, pored ispravnog koda na osnovnom asemblerskom jeziku, sve registarske promenljive iz ulazne biti zamenjene stvarnim registrima.

Takođe se uvodi koncept „funkcija“, koje se prevode kao globalne labele u MIPS asemblerskom jeziku.

Instrukcije koje podržava MAVN jezik:

- add – (addition) sabiranje
- addi – (addition immediate) sabiranje sa konstantom
- b – (unconditional branch) bezuslovni skok
- bltz – (branch on less than zero) skok ako je registar manji od nule
- la – (load address) učitavanje adrese u registar
- li – (load immediate) učitavanje konstante u registar
- lw – (load word) čitavanje jedne memorijske reči
- nop – (no operation) instrukcija bez operacije
- sub – (subtraction) oduzimanje
- sw – (store word) upis jedne memorijske reči
- and – (logical and) logičko „i“ sadržaja dva registra
- bne – (branch if not equal) skok ako sadržaji 2 registra nisu ista
- sll – (shift left logical) logički shift u levo
- srl – (shift right logical) logički shift u desno

3. Koncept rešenja

Prevodilac se realizuje kroz više zavisnih faza koje kao rezultat daju ispravan asembleksi kod ili prijavljuju grešku:

1. Leksička analiza
2. Sintaksna analiza
3. Semantička analiza i pravljenje insturkcija
4. Analiza životnog veka promenljivih
5. Dodela resursa
6. Generisanje izlazne datoteke

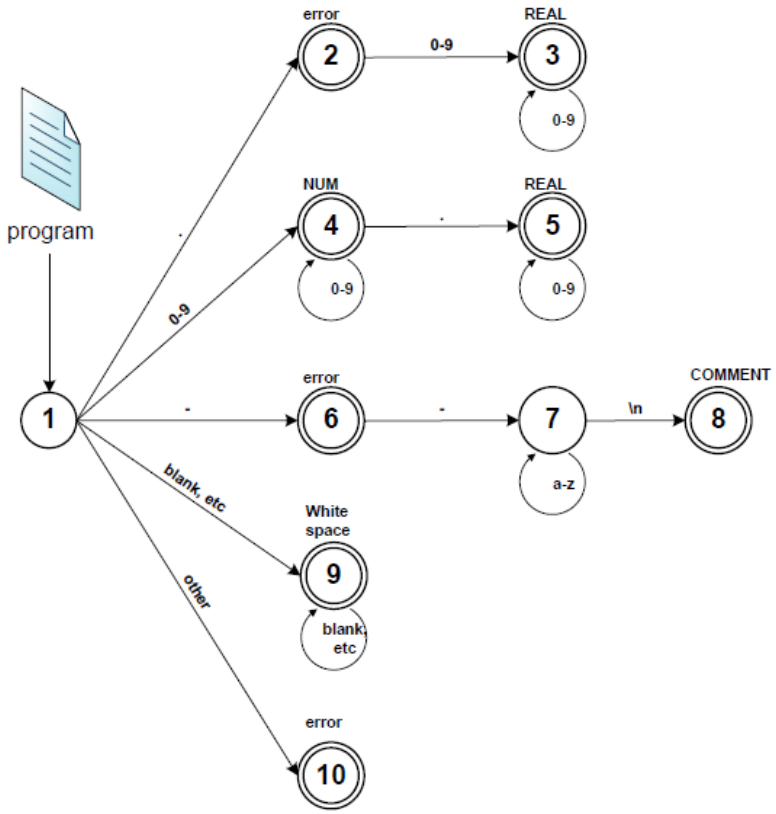
3.1. Leksička analiza

Cilj leksičke analize je izvršavanje provere postoji li u ulaznom fajlu nepodržanih reči od strane MAVN jezika i pretvaranje niza karaktera, odnosno simbola, iz koda koji treba da se prevede u niz tokena (leksički simboli).

Leksički analizator je realizovan pomoću automata sa konačnim brojem stanja. Definisano je početno stanje od kojeg se kreće. Simboli se prepoznaju tako što se čita karakter po karakter iz ulaznog koda i onda se, u zavisnosti od pročitanih znakova prelazi iz stanja u stanje sve dok se ne dođe do finalnog stanja. U tom stanju se prepoznaje simbol i definiše token. Prelazak iz stanja u naredno stanje je jasno definisan i zavisi od narednog simbola. Ako je naredni simbol takav da je reč koja se čita nepodržana od strane MAVN jezika, prijavljuje se leksička greška. Izlaz iz leksičkog analizatora je lista pročitanjih tokena.

Leksička analiza samo ukazuje na neispravnost samih reči, ukoliko takve postoje, ne proverava smisao.

Primer jednog takvog automata:



3.2. Sintaksna analiza

Ulaz u sintaksni analizator je lista tokena iz leksičke analize. Sintaksni analizator ima zadatak da proveri da li procitani tokeni imaju smisleno značenje, odnosno odgovaraju li formalnoj gramatici jezika.

Sintaksa MAVN jezika je opisana gramatikom:

$Q \rightarrow S\ ;\ L$

$L \rightarrow eof$

$L \rightarrow Q$

$S \rightarrow \_mem\ mid\ num$

$S \rightarrow \_reg\ rid$

$S \rightarrow \_func\ id$

$S \rightarrow id:\ E$

$S \rightarrow E$

$E \rightarrow add\ rid,\ rid,\ rid$

$E \rightarrow addi\ rid,\ rid,\ num$

$E \rightarrow sub\ rid,\ rid,\ rid$

$E \rightarrow la\ rid,\ mid$

$E \rightarrow lw\ rid,\ num(rid)$

$E \rightarrow li\ rid,\ num$

$E \rightarrow sw\ rid,\ num(rid)$

$E \rightarrow b\ id$

$E \rightarrow bltz\ rid,\ id$

$E \rightarrow nop$

$E \rightarrow and\ rid,\ rid,\ rid$

$E \rightarrow bne\ rid,\ rid,\ id$

$E \rightarrow sll\ rid,\ rid,\ num$

$E \rightarrow srl\ rid,\ rid,\ num$

Sintaksni analizator se realizovan je pomoću algoritma sa rekurzivnim spuštanjem koji poseduje po jednu funkciju za svaki neterminalni simbol (Q, S, L, E).

3.3. Semantička analiza i pravljenje instrukcija

Kroz ovu fazu se proveravaju semantičke greške, odnosno greške koje su sintaksno ispravne ali ne daju mogućnost uspešnog izvršavanja programa. Semantičke greške su na primer dvostruko definisanje neke memorijske ili registarske promenljive, dvostruko definisanje neke labele ili funkcije, skok na nepostojeću labelu ili funkciju, korišćenje memorijske ili registarske promenljive koja nije definisana.

Sve ove provere vrše se prolaskom kroz listu tokena koja je sintaksno ispravna (prošla je sintaksnu analizu) i tada se prikupe sve memorijske i registarske promenljive, labele i funkcije, a zatim je moguće formirati listu funkcija jer je, ukoliko nije prijavljena semantička greška, pokazana semantička ispravnost.

3.4. Analiza životnog veka promenljivih

Cilj ove faze je da se odredi životni vek promenljivih. To je period koji započinje definicijom promenljive, a završava se njenom poslednjom upotrebom. Ovaj korak je priprema za dodelu resursa (stvarnih registara) promenljivama i neophodan je da bi promenljive koje nisu istovremeno u upotrebi mogle da dele isti resurs.

Na kraju analize životnog veka svaka instrukcija će imati sledeće informacije:

- pred – instrukcije koje joj prethode (prethodnici)
- succ – intrukcije koje joj slede (sledbenici)

- def – skup promenljivih koje ta intrukcija definiše (isto što i destination)
- use – skup promenljivjih koje ta instrukcija koristi (isto što i source)
- in – promenljive koje su žive na ulazu instrukcije
- out – promenljive koje su žive na izlazu instrukcije

Pred, succ, use i def se popunjavaju na osnovu ulaznog koda i već postojećih informacija u listi instrukcija, a in i out se popunjavaju rešavajući jednačine:

$$out[n] \leftarrow U_{s \in succ[n]} in[s]$$

$$in[n] \leftarrow use[n] \cup (out[n] - def[n])$$

To se realizuje algoritmom koji je dat u pseudo kodu, iterirajući od poslednje instrukcije ka prvoj jer to dovodi do bržeg izvršavanja:

```
for each n
        in[n] ← {}; out[n] ← {}
repeat
        for each n
                in'[n] ← in[n]; out'[n] ← out[n]
                out[n] ← U_{s∈succ[n]} in[s]
                in[n] ← use[n] ∪ (out[n] - def[n])
until in'[n] = in[n] and out'[n] = out[n] for all n
```

3.5. Dodela resursa

Cilj ove faze je dodela konkretnih resursa koji su nam dostupni registarskim promenljivama koje su definisane u programu. Sastoji se iz 3 podfaze:

3.5.1. Pravljenje grafa smetnji

Formira se na osnovu analize životnog veka promenljivih. Graf smetnji je kvadratna matrica kojoj je dimenzija jednaka broju promenjivih iz koda. Određena vrednost koja označava postojanje smetnje se postavlja na preseku određene vrste i kolone promenljivih između kojih

postoji smetnja. Smetnja se postavlja između svake promenljive koja je definisana u instrukciji (def) i promenljive koja je živa na izlazu tog čvora (out), a da nije ona sama.

Smetnja onemogućava dodelu istog registra dvema promenljivama koje imaju smetnju, dok se promenljivama koje nemaju smetnju slobodno može dodeliti isti resurs.

3.5.2. Uprošćavanje grafa

Da bi se resursi mogli dodeljivati, graf se prvo mora uprostiti. To se radi tko što se nalazi čvor koji ima najviše smetnji ali tako da broj smetnji ne bude veći od *broj registara-1*. Taj čvor se zatim gura na stek i uklanja iz grafa i takođe se uklanjaju smetnje koje je taj čvor imao sa drugim čvorovima. Zatim se postupak ponavlja dok se svi čvorovi ne budu nalazili na steku. Ukoliko nije moguće pronaći čvor sa prethodno definisanim brojem smetnji to znači da graf ne može da se uprosti i potrebno je vršiti fazu prelivanja u memoriju, ali za potrebe ovog zadatka se samo prijavljuje greška prilikom uprošćavanja.

3.5.3. Dodela resursa

Na kraju je potrebno svakoj promenljivoj dodeliti resurs. Resurs se dodeljuje promenljivoj sa vrha steka, dakle to se vrši obrnutim redosledom od onog kojim su gurani na stek. Dodeljuju se slobodni registri ili registri promenljive sa kojom trenutna promenljiva nije u smetnji.

3.6. Generisanje izlazne datoteke

Na osnovu svih prethodnih analiza i podataka, vrši se ispis asemblerskog koda u tekstualnu datoteku sa ekstenzijom „.S“ . Sve funkcije slede iza „.globl“, sve memorijske promenljive se pisu u sekciju „.data“, a instrukcije i labele u sekciju „.text“ poštujući pravila asemblerskog jezika.

4. Programsko rešenje

Constants.h

```
const int IDLE_STATE = 0;
const int START_STATE = 1;
const int INVALID_STATE = -2;
const int NUM_STATES = 53;
const int NUM_OF_CHARACTERS = 47;

const int __INTERFERENCE__ = 1;
const int __EMPTY__ = 0;

const int __REG_NUMBER__ = 4;
```

- Finalno stanje leksičke analize reči
- Početno stanje leksičke analize
- Stanje neuspešnog čitanja reči
- Broj stanja automata iz leksičke analize
- Broj podržanih znakova leksičke analize

- Smetnja izmedju registara u matrici smetnji
- Oznaka da nema smetnji između registara

- Broj registara

Types.h

Zaglavlje koje definiše tipove (tokena, instrukcija, registara i promenljivih).

```
enum TokenType
{
	T_NO_TYPE, T_ID, T_M_ID, T_R_ID, T_NUM, T_WHITE_SPACE, T_MEM, T_REG, T_FUNC,
	T_ADD, T_ADDI, T_SUB, T_LA, T_LI, T_LW, T_SW, T_BLTZ, T_B, T_NOP, T_AND, T_BNE,
	T_SLL, T_SRL, T_COMMA, T_L_PARENT, T_R_PARENT, T_COL, T_SEMI_COL, T_COMMENT,
	T_END_OF_FILE, T_ERROR
};

enum InstructionType
{
	I_NO_TYPE = 0, I_ADD, I_ADDI, I_SUB, I_LA, I_LI, I_LW, I_SW, I_BLTZ, I_B, I_NOP,
	I_AND, I_BNE, I_SLL, I_SRL
};

enum Regs
{
	no_assign = -1, t0, t1, t2, t3
};


enum VariableType
{
	MEM_VAR, REG_VAR, NO_TYPE
};
```

IR.h

Zaglavlje koje definiše klase za instrukcije, promenljive i graf smetnji i neke metode vezane za njih.

- Klasa Variable

Polja:

```
VariableType m_type;                    - Tip
std::string m_name;                     - Naziv
int m_position;                         - Pozicija u grafu smetnji
Regs m_assignment;                      - Registar koji joj je dodeljen
std::string m_const;                    - Vrednost memorijske promenljive
int m_interference;                     - Sa koliko drugih promenljivih ima smetnju
```

Metode:

```
Variable() : m_type(NO_TYPE), m_name(""), m_position(-1), m_assignment(no_assign)
{}
Variable(VariableType type, std::string name, int pos) : m_type(type),
m_name(name), m_position(pos), m_assignment(no_assign), m_interference(0) {}

void setAssignment(Regs r)

void setConst(std::string c)
```

- Klasa Function

Polja:

```
FunctionType m_type;                    - Tip
std::string m_name;                     - Naziv funkcije
```

Metode:

```
Function() : m_type(FUNC_TYPE), m_name("") {}
Function(std::string name) : m_type(FUNC_TYPE), m_name(name) {}

std::string getName()
```

- Klasa Label

Polja:

```
LabelType m_type;                       - Tip
std::string m_name;                     - Naziv labele
```

Metode:

```
Label() : m_type(LABEL_TYPE), m_name("") {}
Label(std::string name) : m_type(LABEL_TYPE), m_name(name) {}

std::string getName()
```

- Klasa Instruction

Polja:

```
int m_position;                    - Pozicija u grafu smetnji
InstructionType m_type;            - Tip instrukcije
std::string m_const;               - Za instrukcije koje imaju konstantu
std::string m_myLabel;             - Kojoj labeli pripada instrukcija
std::string m_branchLabel;         - Za skokove (labela na koju treba skočiti)

Variables m_dst;                   - Destination promenljive
Variables m_src;                   - Source promenljive
Variables m_use;                   - Use promenljive
Variables m_def;                   - Def promenljive
Variables m_in;                    - In promenljive
Variables m_out;                   - Out promenljive
std::list<Instruction*> m_succ;    - Instrukcije naslednici
std::list<Instruction*> m_pred;    - Instrukcije prethodnici
```

Metode:

```
Instruction () : m_position(0), m_type(I_NO_TYPE) {}
Instruction (int pos, InstructionType type, Variables& dst, Variables& src,
std::string label) :
m_position(pos), m_type(type), m_dst(dst), m_src(src), m_myLabel(label) {}

void setConst(std::string c)
std::string getConst()
std::string getLabel()
void setBranchLabel(std::string s)
```

- Struktura InterferenceGraph

```
Variables* variables;              - Lista svih registarskih promenljivih
int** values;                      - Vrednost u matrici
int size;                          - Dimenzija kvadratne matrice
```

- Lista promenljivih

```
typedef std::list<Variable*> Variables;
```

- Lista funkcija

```
typedef std::list<Function*> Functions;
```

- Lista labela

```
typedef std::list<Label*> Labels;
```

- Lista instrukcija

```
typedef std::list<Instruction*> Instructions;
```

Token.h

Klasa koja modeluje token.

Polja:

```
TokenType tokenType;
```
- tip tokena

```
std::string value;
```
- vrednost tokena

Metode:

```
Token() : tokenType(T_NO_TYPE), value("") {}

TokenType getType();
void setType(TokenType t);
string getValue();
void setValue(std::string s);


void makeToken(int begin, int end, std::vector<char>& program, int
lastFiniteState);
void makeErrorToken(int pos, std::vector<char>& program);
void makeEofToken();
void printTokenInfo();
void printTokenValue();
std::string tokenTypeToString(TokenType t);
```

FiniteStateMachine.h / FiniteStateMachine.cpp

Modeluje automat sa konačnim brojem stanja, za leksičku analizu.

```
void initStateMachine();
```

LexicalAnalysis.h / LexicalAnalysis.cpp

Klasa za leksičku analizu.

Polja:

```
typedef std::list<Token> TokenList;
FiniteStateMachine fsm;
TokenList tokenList;
Token errorToken;
```
- Lista tokena

Metode:

```
bool Do();
bool readInputFile(std::string fileName);
```
- Pokretanje leksičke analize
- Učitavanje ulazne datoteke

- Čita naredni token iz fajla

```
Token getNextTokenLex();

void printTokens();
```

SyntaxAnalysis.h / SyntaxAnalysis.cpp

Klasa za sintaksnu analizu.

Polja:

```
LexicalAnalysis& lexicalAnalysis;
TokenList::iterator tokenIterator;
```
- Iterator na listu tokena koja predstavlja izlaz leksičke analize

```
Token currentToken;
```
- Trenutni token

```
bool errorFound;
```
- Indikator greške

Metode:

```
SyntaxAnalysis(LexicalAnalysis& lex);
```

- Obavlja sintaksnu analizu

```
bool Do();
```

- Ispisuje poruku o grešci

```
void printSyntaxError(Token token);
```

- Ispisuje informacije o tokenu

```
void printTokenInfo(Token token);
```

- Metoda koja služi za proveru sintaksne ispravnosti trenutnog tokena

```
void eat(TokenType t);
```

- Vraća sledeći token iz liste tokena

```
Token getNextToken();
```

- Neterminalni simboli: Q, S, L, E

```
void Q();

void S();

void L();

void E();
```

InstructionList.h / InstructionList.cpp

Klasa za obavljanje semantičke analize i pravljenje liste instrukcija.

Polja:

```
Instructions instrList;
```
- Lista pokazivača na instrukcije koja se kreira metodom Do()

```
Variables memList;
```
- Lista pokazavača na memorijske promenljive

```
Variables regList;
```
- Lista pokazivača na registarske promenljive

```
Functions funcListFirst;
```
- Lista funkcija koje postoje u kodu

```
Labels labelListFirst;
```
- Lista labela koje postoje u kodu

```
Functions funcListSecond;
```
- Lista funkcija na koje treba skociti (za proveru semantičke ispravnosti)

```
Labels labelListSecond;
```
- Lista labela na koje treba skočiti (za proveru semantičke ispravnosti)

Metode:

```
InstructionList(LexicalAnalysis& lex);
```

- Obavlja semantičku analizu i formiranje liste instrukcija

```
bool Do();

~InstructionList();
```

- Obavlja proveru za memorijske promenljive

```
void memAnalysis();
```

- Obavlja proveru za registarske promenljive

```
void regAnalysis();
```

- Obavlja proveru za funkcije

```
void funcAnalysis();
```

- Obavlja proveru za labele

```
void labelAnalysis();

void addAnalysis();
void addiAnalysis();
void subAnalysis();
void laAnalysis();
void lwAnalysis();
void liAnalysis();
void swAnalysis();
void bAnalysis();
void bltzAnalysis();
void nopAnalysis();
void andAnalysis();
void bneAnalysis();
void sllAnalysis();
void srlAnalysis();
```

```
void printSemanticErrorReg(Token token, int num);
void printSemanticErrorMem(Token token, int num);
void printSemanticErrorFunc0(Token token);
void printSemanticErrorFunc1(string function);
void printSemanticErrorLabel0(Token token);
void printSemanticErrorLabel1(string label);
void printTokenInfo(Token token);
```

- Vraća sledeći token iz liste tokena

```
Token getNextToken();
```

- Proverava koji je sledeći token iz liste tokena (korisno kada ispitujemo da li je neki id zapravo labela)

```
Token checkNextToken();
```

- Metoda koja prolazi kroz listu registarskih promenljivih i vraća pokazivač na željenu registarsku promeljivu

```
bool findVariableReg(string value, Variable*& var);
```

- Metoda koja prolazi kroz listu memorijskih promenljivih i vraća pokazivač na željenu memorijsku promeljivu

```
bool findVariableMem(string value, Variable*& var);
```

- Metoda koja proverava da li medju labelama na koje treba skočiti ima nepostojećih

```
bool labelExists(Label& label);
```

- Metoda koja proverava da li medju funkcijama na koje treba skočiti ima nepostojećih

```
bool functionExists(Function& func);
```

LivenessAnalysis.h / LivenessAnalysis.cpp

Klasa za analizu životnog veka.

Polja:

```
InstructionList& instructionList;
```

Metode:

```
LivenessAnalysis(InstructionList& instr);
```

- Poziva metode za pravljenje grafa toka upravljanja i analize životnog veka

```
void Do();
```

- Funkcija koja pravi graf toka upravljanja

```
void makeFlowControlGraph();
```

- Za definisanje liste USE promenljivih

```
void defineUse();
```

- Za definisanje liste DEF promenljivih

```
void defineDef();
```

- Definiše listu prethodnika i sledbenika
```
void definePredSucc();
```

- Obavlja analizu životnog veka
```
void livenessAnalysis(Instructions instructions);
```

- Proverava da li postoji određena promenljiva u listi promenljivih
```
bool variableExists(Variable* var, Variables varList);
```

- Vraća pokayivač na prvu instrukciju posle labele sa prosleđenim nazivom
```
Instruction* findInstruction(string label);
```

- Ispisuje use, def, pred, succ, in i out za svaku instrukciju
```
void printInstructions();
```

ResourceAllocation.h / ResourceAllocation.cpp

Programsko rešenje dodele resursa.

- Za pravljenje grafa smetnji
```
void makeInterferenceGraph(InterferenceGraph* ig, Instructions& instrList,
Variables& varList);
```

- Štampa graf smetnji
```
void printInterferenceGraph(InterferenceGraph* ig);
```

- Uprošćavanje grafa smetnji
```
bool doSimplification(Stack& s, InterferenceGraph* ig, int regNumber);
```

- Pronalazi promenljivu na određenoj poziciji
```
bool findVariable(int position, Variables& varList, Variable*& var);
```

- Funkcija za dodelu resursa
```
bool doResourceAllocation(Stack* simplificationStack, InterferenceGraph* ig);
```

- Štampa listu promenljivih (naziv + broj registra koji joj je dodeljen
```
void printVariables(Variables& varList);
```

- Brisanje dinamički zauzete memorije
```
void deleteInterferenceGraph(InterferenceGraph* ig);
```

WritingOutputFile.h / WritingOutputFile.cpp

```
bool writeInFile(InstructionList& instrList);

- Ispisuje .globl sekciju
void writeFunctions(Functions& funcList, ofstream& file);

- Ispisuje .data sekciju
void writeMemVariables(Variables& memList, ofstream& file);

- Ispisuje .text sekciju
void writeInstructions(Instructions& instrList, ofstream& file);

void writeAdd(Instruction* instr, ostream& file);
void writeAddi(Instruction* instr, ostream& file);
void writeSub(Instruction* instr, ostream& file);
void writeLi(Instruction* instr, ostream& file);
void writeLa(Instruction* instr, ostream& file);
void writeLw(Instruction* instr, ostream& file);
void writeSw(Instruction* instr, ostream& file);
void writeBltz(Instruction* instr, ostream& file);
void writeB(Instruction* instr, ostream& file);
void writeNop(Instruction* instr, ostream& file);
void writeAnd(Instruction* instr, ostream& file);
void writeBne(Instruction* instr, ostream& file);
void writeSll(Instruction* instr, ostream& file);
void writeSrl(Instruction* instr, ostream& file);
```

main.cpp

U main-u su pravljeni svi objekti klasa koje realizuju odredjene faze, pozivane metode koje pokreću te faze i rađeni ispisi poruka o (ne)uspešnosti svake faze.

5. Verifikacija rešenja

Program na terminal ispisuje poruke o stanju svake faze, da li je uspešno prošla ili se javila greška, za neke faze ispisuje i koja je greška u pitanju, takođe ispisuje na primer stanje informacije o listi instrukcija nakon obavljene analize životnog veka, graf smetnji u kome su prikazane smetnje između registara i koji je registar dodeljen kojoj promenljivoj nakon dodele resursa.

Praćenjem svih ovih poruka, namernim izazivanjem grešaka, povećavanjem i smanjenjem broja registara i dodavanjem i brisanjem raličitih instrukcija u ulazne datoteke izvršena je provera ispravnosti.

Za proveru su korišćeni „simple.mavn“ , „simple.mavn“ dopunjen sa dodatne 4 instrukcije i „multiply.mavn“.

- simple.mavn (broj registara - 4)

  ulazna datoteka:

simple.mavn | multiply.mavn

```
1  _mem m1 6;
2  _mem m2 5;
3
4  _reg r1;
5  _reg r2;
6  _reg r3;
7  _reg r4;
8  _reg r5;
9
10 _func main;
11     la      r4, m1;
12     lw      r1, 0(r4);
13     la      r5, m2;
14     lw      r2, 0(r5);
15     add     r3, r1, r2;
16
```

izlazna datoteka:

simple.mavn | multiply.mavn | izlazna.s

```
1  .globl main
2
3  .data
4  m1:     .word 6
5  m2:     .word 5
6
7  .text
8  main:
9      la      $t0, m1
10     lw      $t1, 0($t0)
11     la      $t0, m2
12     lw      $t0, 0($t0)
13     add     $t0, $t1, $t0
14
15
```

- multiply.mavn (broj registara - 5)

  ulazna datoteka:

simple.mavn | multiply.mavn | izlazna.s

```
1  _mem m1 6;
2  _mem m2 5;
3  _mem m3 0;
4
5  _reg r1;
6  _reg r2;
7  _reg r3;
8  _reg r4;
9  _reg r5;
10 _reg r6;
11 _reg r7;
12 _reg r8;
13
14 _func main;
15     la      r1, m1;
16     lw      r2, 0(r1);
17     la      r3, m2;
18     lw      r4, 0(r3);
19     li      r5, 1;
20     li      r6, 0;
21 lab:
22     add     r6, r6, r2;
23     sub     r7, r5, r4;
24     addi    r5, r5, 1;
25     bltz    r7, lab;
26
27     la      r8, m3;
28     sw      r6, 0(r8);
29     nop;
30
```

Izlazna datoteka:

simple.mavn | multiply.mavn | izlazna.s

```
1  .globl main
2
3  .data
4  m1:     .word 6
5  m2:     .word 5
6  m3:     .word 0
7
8  .text
9  main:
10     la      $t0, m1
11     lw      $t3, 0($t0)
12     la      $t0, m2
13     lw      $t4, 0($t0)
14     li      $t1, 1
15     li      $t2, 0
16 lab:
17     add     $t2, $t2, $t3
18     sub     $t0, $t1, $t4
19     addi    $t1, $t1, 1
20     bltz    $t0, lab
21     la      $t0, m3
22     sw      $t2, 0($t0)
23     nop
24
```

Namerno izazivanje semantičke greške:

simple.mavn | multiply.mavn | izlazna.s | new 1

```
1  _mem m1 6;
2  _mem m2 5;
3
4  _reg r1;
5  _reg r2;
6  _reg r3;
7  _reg r4;
8  _reg r5;
9
10 _func main;
11     la      r4, m1;
12     lw      r1, 0(r4);
13     la      r5, m2;
14     lw      r2, 0(r5);
15     add     r3, r1, r2;
16 labela:
17     and     r1, r3, r2;
18     sll     r1, r1, 10;
19     b       skoci;
20     srl     r1, r1, 10;
21     bne     r1, r1, main;
```

```
[T_COMMA]                                   ,
[T_ID]                                   main
[T_SEMI_COL]                                ;
[T_END_OF_FILE]                           EOF
Syntax analysis finished successfully!



Semantic analysis: START
        Semantic error! Label: skoci not defined.

Exception! Semantic analysis failed!

Press any key to continue . . .
```

Namerno izazivanje faze prelivanja koja u ovom rešenju nije dozvoljena (ulazna datoteka multiply.mavn):

```
r8      0       0       0       0       0

/**
 * Use Exception! Spill detected!
 */
const i Press any key to continue . . .

/**
 * Number of regs in processor.
 */
const int __REG_NUMBER__ = 4;

/**
```