

MAVN compiler

Osnovi paralelnog programiranja i softverski alati

Papp Tamás
RA4/2022

MAVN compiler to translate higher level assembly Mips
to regular Mips32 assembly



Faculty of Technical Sciences
University of Novi Sad
Serbia
29. maj 2024.

Sadržaj

1	Uvod	2
2	Analiza problema	2
2.1	MAVN jezik	2
3	Koncept i programsko rešenje	3
3.1	Lexical analysis	3
3.2	Syntax analysis	3
3.3	Labele, Funkcije, Memory vredonsti i Variable	4
3.4	Kreiranje instrukcija	4
3.5	Analiza životnog veka	5
3.6	Dodela resursa	5
3.7	Pravljanje izlaznog datoteke	5
4	Arhitektura projekta i verifikacija	6
4.1	Arhitektura projekta	6
4.2	Verifikacija	7

1 Uvod

U ovom projektu zadatak je da realizuje MAVN prevodilac koji prevodi programe sa višeg asemblerskog jezika na osnovi MIPS 32bit asemblerski jezik. Prevodilac prvo pročita tekst iz datoteke i pretvori u tokene sa leksičkom analizom na principu state machine. Sledeće faza je sintaksna analiza gde se formira gramatika programskog jezika. Posle formiraju liste promenljivih, labele, funkcije, instrukcije. Zadnja faza je analiza životnog veka promenljivih, koju sledi formiranje grafa smetnji.

Naš zadatak je da dodajemo 3 nova instrukcija u naš prevodilac i da realizujemo sve delove prevodioca, ali nam za start je dato leksička analiza.

2 Analiza problema

MAVN(Mips Assembler Visokog Nivoa) ja alat koji prevodi program napisan na MIPS 32bit asembleru. Viši MIPS 32 bit asemblerski jezik koji uvodi koncept registarske promenjive. Olakšava pisanje asemblerskog koda jer pravi abstrakciju na promenjive umesto pravih resursa i ne treba da pratimo fizičkih gde su nam snimljeni podaci.

2.1 MAVN jezik

U instrukcijama ne smeju da se koriste pravi resurse već isključivo promenjive. MAVN jezik podržava 10 MIPS instrukcija, a to su:

- add - addition
- addi - addition immediate
- b - unconditional branch
- bltz - branch on less than zero
- la - load address
- li - load immediate
- lw - load word
- nop - no operation
- sub - subtraction
- sw - store word

Još 3 naša dodata instrukcija

- mult - multiply
- not - not
- j - unconditional jump

3 Koncept i programsko rešenje

3.1 Lexical analysis

Leksička analiza učitava karaktere iz ulaznog datoteke i u jedan *finite state machine* pogleda šta bi trebao da bude sledeći karakter u instrukciji. U ovom projektu je nam dato

- *Constansts.h* gde su definisane broj stanja i broj karaktera u jeziku.
- *Token.h* & *Token.cpp* & *Types.h* gde su definisane tipove tokene i metode za korišćenje.
- *FiniteStateMachine.h* & *FiniteStateMachine.cpp* gde je definisan matrix mašine
- *LexicalAnalysis.h* & *LexicalAnalysis.cpp* gde čitanje datoteke i konvertovanje u tokene se dešava

Ako dodamo još 3 instrukcija onda treba da dodamo u *Token.cpp* tri novi tokene

T_OR, T_NOR, T_J

Sve enumeracije korišćene u projektu se nalazi u *Types.h* TODO

3.2 Syntax analysis

Sintaksna analiza je faza koja proverava da li niz tokena generisanih leksičkom analizom odgovara gramatici jezika. Ova klasa koristi rezultate leksičke analize i generiše odgovarajuće uputstva za dalje korake obrade kao što su generisanje međureprezentacije (IR) i analiza životnog veka. U MAVN jeziku *Q, S, L, E*, svaki simbol ima svoju funkciju u klasu *SyntaxAnalysis*

$Q \rightarrow S; L$	$S \rightarrow _mem\ mid\ num$	$L \rightarrow eof$	$E \rightarrow add\ rid,\ rid,\ rid$
	$S \rightarrow _reg\ rid$	$L \rightarrow Q$	$E \rightarrow addi\ rid,\ rid,\ num$
	$S \rightarrow _func\ id$		$E \rightarrow sub\ rid,\ rid,\ rid$
	$S \rightarrow id: E$		$E \rightarrow la\ rid,\ mid$
	$S \rightarrow E$		$E \rightarrow lw\ rid,\ num(rid)$
			$E \rightarrow li\ rid,\ num$
			$E \rightarrow sw\ rid,\ num(rid)$
			$E \rightarrow b\ id$
			$E \rightarrow bltz\ rid,\ id$
			$E \rightarrow nop$

Funkcije *Q, S, L, E* proveravaju sintaksnu grešku i sa funkcijom *eat(Token)* idu od tokena do token i proveravaju redosled tokena u jedan naredbi. U *SyntaxAnalysis* klasu još popunimo liste za *Memory, Register, Function i Label* i kreiramo *Variable* za korišćenje a iz *variable* pravimo odgovarajuće *Instrukcije*.

3.3 Labele, Funkcije, Memory vredonsti i Variable

Ako u funkciji *S* našli smo jedan od odgovarajućih tokena *T_MEM*, *T_REG*, *T_FUNC*, *T_ID* onda pozovemo funkcije koje stavljaju tokene u STL listi. Za sačuvanje podatke koristimo klasu u *IR.h* datoteku

Variable sa polje:

- int m_value - vredonst
- VariableType m_type - tip variabla
- std::string m_name - ime
- int m_position - poziciju u programu
- Regs m_assignment - koji registar koristi

Kao i konstruktori, getter i setter funkcije i operator overload za "<<".

Labele imaju svoju klasu u *Labels.h* sa polje:

- std::string name - ime
- int position - poziciju u programu

3.4 Kreiranje instrukcija

Pre svega ponovo prolazimo kroz liste tokena ali sada tražimo tokene koji predstavljaju instrukcije. Ako smo našli odgovarajući token onda pravimo jedan *Instruction* class gde čuvamo više lista variable. Još u *SyntaxAnalysis.cpp* fajlu ima funkcija *instructionFactory(type, destination, source)* koji smesti koja kao što je naznačeno prolazi kroz tokene i kada dođe do odgovarajućeg tokena popunjava destinaciju i izvor promenljivama kao što je naznačeno u gramatici **Instruction** klasa ima sledeće polje:

- int m_position - pozicija u programu
- InstructionType m_type - tip instrukcije
- Variables m_dst - destination variables, gde upišemo vrednosti
- Variables m_src - source variables, od ovih variable čitamo vredonsti
- Variables m_use - skup za blok sadrži sve promenljive koje su korišćene u bloku pre nego što su dodeljene (definisan u tom bloku).
- Variables m_def - skup sadrži sve promenljive koje su definisane unutar bloka
- Variables m_in - skup za svaki osnovni blok sadrži promenljive koje su žive na početku tog bloka.
- Variables m_out - skup sadrži promenljive koje su žive na kraju bloka
- std::list<Instruction*>m_succ - sledeće instrukcije u listu
- std::list<Instruction*>m_pred - predhodne instrukcije u listu

Kada smo formirali instrukcije pozivamo funkciju za popunjavanje skupova prethodnika i sledbenika, kao i funkciju za popunjavanje *use* i *def* skupova.

3.5 Analiza životnog veka

Analiza životnog veka (engl. Liveness Analysis) je važna tehnika u optimizaciji kompajlera koja se koristi za određivanje perioda u kojem promenljive (ili registri) u programu "žive", odnosno kada su njihove vrednosti relevantne i mogu biti korišćene. Ova analiza pomaže kompajleru da optimizuje upotrebu resursa, posebno registara, smanjujući potrebu za prekomernim upisivanjem i čitanjem iz memorije. Da uradimo LivenessAnalysis mi u programu treba da popunimo *Successor* i *Predecessor* liste variable a posle da uradimo analizu sa listama variable. Algoritam za **LivenessAnalysis** je sledeći:

```
out[n] ←  $\bigcup_{s \in succ[n]} in[s]$ 
in[n] ← use[n] ∪ (out[n] − def[n])

for each n
  in[n] ← {}; out[n] ← {}
repeat
  for each n
    in'[n] ← in[n]; out'[n] ← out[n]
    out[n] ←  $\bigcup_{s \in succ[n]} in[s]$ 
    in[n] ← use[n] ∪ (out[n] − def[n])
until in'[n] = in[n] and out'[n] = out[n] for all n
```

3.6 Dodela resursa

Do sada smo koristili registarske promenljive, koji može da imamo beskonačno mnogo, ali u arhitekturi računara imamo konačan broj registar. Moramo da mapiramo registarske promenljive na realne registre

Formira se graf smetnji na osnovu analize životnog veka promenljivih. To se odvija u funkciji *buildGraph()*. 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. Na primer u ovom zadatku dobijamo:

	r0	r1	r2	r3	r4
r0	0	1	0	0	1
r1	1	0	0	0	0
r2	0	0	0	0	0
r3	0	0	0	0	0
r4	1	0	0	0	0

Posle formiranja matrice možemo da uradimo *doResourceAllocation()*. Dodeljivanje registara se često posmatra kao problem bojenja grafa Svaki čvor (promenljiva) se "boji" koristeći jednu od dostupnih boja (registara). Čvorovi povezani granom (interferiraju) ne smeju imati istu boju.

3.7 Pravljanje izlaznog datoteke

MIPS jezik ima tri dela *.globl*, *.data*, *.text* u koji se naš program treba različite stvari da piše

1. **.globl** se koristi za deklarisanje globalnih simbola. Ovi simboli (funkcije ili promenljive) mogu biti vidljivi i dostupni iz drugih fajlova koji su povezani tokom faze linkovanja
2. **.data** označava početak segmenta podataka. U ovom segmentu se definišu statički podaci, kao što su globalne promenljive, inicijalizovani podaci itd
3. **.text** označava početak segmenta koda. U ovom segmentu se definišu instrukcije programa, tj. kod koji će se izvršavati

4 Arhitektura projekta i verifikacija

Da bi verifikovali tačnost programa uzimamo primere. Pre toga treba da znamo šta tražimo sa kompajlerom.

4.1 Arhitektura projekta

Za tipa tokene koristimo enum:

```
enum TokenType
{
    T_NO_TYPE,

    T_ID,                // abcd...
    T_M_ID,              // m123...
    T_R_ID,              // r123...
    T_NUM,               // 123...
    T_WHITE_SPACE,

    // reserved words
    T_MEM,               // _mem
    T_REG,               // _reg
    T_FUNC,              // _func
    T_ADD,               // add
    T_ADDI,              // addi
    T_SUB,               // sub
    T_LA,                // la
    T_LI,                // li
    T_LW,                // lw
    T_SW,                // sw
    T_BLTZ,              // bltz
    T_B,                 // b
    T_NOP,               // nop

    // added token types
    T_OR,                // or
    T_NOR,               // nor
    T_J,                 // j

    // operators
    T_COMMA,             // ,
    T_L_PARENTH,         // (
    T_R_PARENTH,         // )
    T_COL,               // :
    T_SEMI_COL,          // ;

    // utility
    T_COMMENT,
    T_END_OF_FILE,
```

```

        ERROR,
    };

```

I za tipovi instrukcije i registri

```

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,
    //added instructions
    I_OR,
    I_NOR,
    I_J
};

/**
 * Reg names.
 */
enum Regs
{
    no_assign = 0,
    t0,
    t1,
    t2,
    t3
};

```

Svoje klase koji smo implementirali i koji koristimo u projektu:

```

LexicalAnalysis lex;
SyntaxAnalysis Syl;
Variable var;
Instruction instr;
Labels labs;
LivenessAnalysis liv;
InterferenceGraph infg;

```

4.2 Verifikacija

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 ;

```

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;

```