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

ИСПИТНИ РАД

Кандидат: **Алексић Никола**
Број индекса: **RA 24/2018**

Предмет: **Системска програмска подршка у реалном времену II**
Тема рада: **Преводилац**

Ментор рада: **др Миодраг Ђукић**

Нови Сад, јун, 2020.

Sadržaj:

1. Uvod

2. Analiza problema

3. Koncept rešenja

4. Opis rešenja

4.1 Ulazna tačka programa

4.2 Sintaksna analiza

4.3 Skladištenje ulaznog koda

4.4 Određivanje skupova prethodnika I naslednika

4.5 Određivanje in I out skupova

4.6 Dodela resursa

4.6.1 Kreiranje grafa interferencije

4.6.2 Kreiranje steka simplifikacije

4.6.3 Dodela registara (bojenje)

5. Testovi

6. Dodatak

7. Zaključak

1.Uvod

U okviru projektnog zadatka potrebno je realizovati MAVN prevodilac koji prevodi programe sa višeg asemblerskog jezika na osnovni MIPS asemblerski jezik. MAVN jezik u odnosu na osnovni asemblerski jezik sadrži koncept registarske promenjive, odnosno korisnik ne mora da vodi računa o korišćenju pravih resursa nego samo manipuliše promenjivama. Takođe MAVN jezik podržava sledećih 13(10 zadatih i 3 **dodate**) instrukcija:

add - sabiranje

addi - sabiranje sa konstantom

and - logičko i

andi - logičko i sa konstantom

lb - učitavanje bajta sa adrese u registar

la - učitavanje adrese u registar

lw - učitavanje jedne memorijske reči

li - učitavanje konsante u registar

nop - instrukcija bez operacije

sub - oduzimanje

b - bezuslovni skok

bltz - skok ako je registar manji od nule

sw - upis jedne memorijske reči

Mavn prevodilac je potrebno implementirati tako da izveštava o potencijalnim greškama (leksičkim, sintaksičkim, semantičkim). Izlaz iz prevodioca treba da sadrži adekvatan asemblerski kod koji je moguće izvršavati na simulatoru MIPS 32-bitne arhitekture.

2. Analiza problema

Ako se zanemari šablonski deo zadatka (unapred data leksička analiza, sintaksička analiza, kreiranje liste instrukcija zajedno sa skupovima koji ih određuju, sam ispis u .s fajl....) suština problema se svodi na dodelu realnih resursa registarskim promenjivim. Odnosno formiranje, uprošćavanje i bojanje grafa smetnji ukoliko je to uopšte i moguće uraditi bez implementacije faze prelivanja koja je neophodna u slučaju nemogućnosti bojanja grafa sa zadatim brojem registara koje imamo na raspolaganju. Fazu prelivanja zbog njene preobimnosti nećemo implementirati, a u slučaju nemogućnosti bojanja grafa prijavljujemo grešku.

3. Koncept rešenja

Radi lakšeg pregleda rešenje je podeljeno u etape:

1. izrada leksičkog analizatora (data kao već implementirana u projektu)
   -unutar koga smo dopisali 3 instrukcije
2. izrada sintaksnog analizatora
3. grupisanje ulaznog koda u celine i memorisanje istih zarad lakše obrade
   -formiranje liste instrukcija
   -formiranje lista promenjivih
   -formiranje liste labela
   -formiranje liste funkcija
4. određivanje skupova prethodnika i naslednika instrukcija
5. određivanje in/out skupova
6. dodela resursa
   -kreiranje grafa interferencije
   -kreiranje steka simplifikacije
   -bojenje grafa(dodela realnih resursa registarskim promenjivama)
7. ispis u .s fajl odnosno kreiranje asemblerskog koda spremnog za dalju obradu

Svaka od navedenih etapa prevođenja ispraćena je kontrolnim ispisom konzole radi lakšeg praćenja toka prevođenja MAVN jezika. U slučaju leksičkih, sintaksičkih ili pak semantičkih grešaka program će obustaviti rad I prijaviti grešku u vidu ispisa u konzoli.

4. Opis rešenja

4.1 ulazna tačka programa (MainProgram)

int main(int argc,char* argv[])

U početnom delu našeg programa kreiramo kontejnere za skladištenje celina ulaznog MAVN koda. Zatim, ostatak main-a pokriven je try-catch blokom da bismo u slučaju grešaka obustavili rad programa i ispisali podatke o nastaloj grešci. Unutar try bloka pozivaju se funkcije koje izvršavaju prethodno navedene etape prevođenja i koje ćemo u nastavku posebno obrađivati, izuzevši leksički analizator koji nam je dat kao već implementiran. Svaka etapa praćena je zasebnim .h(header) i .cpp(source) fajlom u kom smo iskoristili mogućnost objektno orijentisanog programiranja i kreirali klase radi preglednosti i lakšeg grupisanja koda. Obratiti pažnju da zbog dinamičke alokacije memorije unutar kontejnera imamo u catch, a i na kraju try bloka pozvanu metodu nad objektom klase *CreateList .doList(Instructions& listaInstrukcija, Variables& listaRegistarskihPromenjivih, Variables& listaMemorijskihPromenjivih, Labels& labele,Functions& funkcije)* koja će obrisati svu dinamički alociranu memoriju i postarati se da ne dođe do curenja memorije(memory leak). Razlog zbog kog na oba mesta pozivamo funkciju koja briše dinamički alociranu memoriju je čisto bezbednosni. Ako u slučaju izuzetka odnosno greške program obustavi svoj normalni tok i udje u catch blok mi takođe moramo obrisati dinamički alociranu memoriju isto kao i kad se program izvrši bez pojavljivanja greške.

4.2 sintaksna analiza

Sintaksnu analizu vršimo kreiranjem objekta klase *SyntaxAnalysis* kojem prilikom kreiranja kao parameter konstruktora prosledimo objekat klase *LexicalAnalysis* koji u sebi sadrži listu tokena ulaznog MAVN jezika neophodnu za dalju obradu. Pozivanjem funkcije *.Do()* nad objektom klase *SyntaxAnalysis* vršimo sintaksnu analizu prativši gramatiku MAVN jezika. Funkcija realizuje sintaksnu analizu pozivajući funkcije: *SyntaxAnalysis::Q(),SyntaxAnalysis::S(), SyntaxAnalysis::L(),SyntaxAnalysis::E().* Koje su imenovane i izvršavaju se po istoimenim simbolima I pravilima gramatike. Takođe prilikom dolaska do terminalnog simbola pozivamo funkciju *SyntaxAnalysis::eat(TokenType t)* koja će nam u slučaju neregularostni (pokušaja da

pojedemo neočekivani token) ulazne liste tokena izbaciti grešu. Nakon završene sintaksne analize funkcija *.Do()* vraća bool vrednost na osnovu koje ispisujemo poruku o uspešnoj odnosno neuspešnoj sintaksnoj analizi.

4.3 skladištenje ulaznog koda

Skladištenje ulaznog koda u celine vršimo kreiranjem objekta klase *CreateList* kojem prilikom kreiranja kao parameter konstruktora prosledimo objekat klase *LexicalAnalysis* koji u sebi sadrži listu tokena ulaznog MAVN jezika neophodnu za dalju obradu. Zatim pozivanjem funkcije *doList(Instructions& listaInstrukcija,Variables& listaRegistarskihPromenjivih, Variables& listaMemorijskihPromenjivih,Labels& labele,Functions& funkcije)* nad objektom klase kreiramo liste koje su nam u nastavku potrebne radi dalje analize ulaznog koda. Takođe unutar funkcije *.doList()* koristimo i dodatne funkciju *CreateList::compareStrings(*`std::string str1, std::string str2`*)* koja nam vraća bool vrednost "true" u slučaju da smo joj prosledili 2 ista stringa. Realizovana je statički da bismo je mogli koristiti i u drugim fajlovima bez da prethodno kreiramo objekat klase *CreateList.* Prilikom učitavanja koda u kontejnere korišćeni su predloženi typedef-ovi iz unpared implementiranog "*IR.h*" kao i već predložene klase radi bolje čitljivosti koda.

4.4 određivanje skupova prethodnika i naslednika

Određivanje skupova prethodnika i naslednika vršimo kreiranjem objekta klase *LifeAnalysis* nad kojim pozivamo funkciju *createPredAndSucc(Instructions& listaInstrukcija, Labels& listaLabela)* kojoj prosleđujemo prethodno kreiranu listu instrukcija i labela. Listu labela koristimo u slučaju instrukcija skoka kako bismo utvrdili na koju labelu instrukcija skače. Kada odredimo prethodnika ili naslednika koristimo redom *Instructions::pushPred(Intruction* i)* i *Instructions::pushSucc(Intruction* i)* metode koje su naknadno dodate radi lakšeg dodavanja instrukcija u skupove prethodnika/naslednika.

4.5 određivanje in i out skupova instrukcije

Određivanje in i out skupova vršimo pozivanjem funkcije *.livenessAnalysis(Instructions& listInstrukcija)* nad već kreiranim objektom klase *LifeAnalysis*. Algoritam određivanja skupova zbog efikasnosti prolazi kroz listu instrukcija u obrnutom redosledu i dodeljuje clanove po pravilu:

out[n] ← U in[s] , s ← succ[n]
in[n] ← use[n] U (out[n]-def[n])

sve dok se u 2 uzastopne iteracije ne poklope vrednosti svih in i out skupova.

Takođe u toku određivanja skupova koristimo funkciju *LifeAnalysis::variableExists(Variable*& var, Variables& listVar)* koja u slučaju da je promenjiva sadržana u listi promenjivih vraća bool vrednost *true* u suprotnom *false*.

4.6 dodela resursa

4.6.1 kreiranje grafa interferencije

Kreiranje/menjanje grafa interferncije vršimo pozivanjem funkcija nad objektom klase *InterferenceGraph:*

- *initInterferenceGraph(InterferenceGraphStruct& ig,int size)*

  -Graf je realizovan kao niz nizova int vrednosti, gde je svaki niz veličine-size, koje na početku sve bivaju inicijalizovane na 0

- *deleteInterferenceGraph(InterferenceGraphStruct& ig, int size)*

  -Brišemo dinamički alociranu memoriju koju smo alocirali pri kreiranju grafa

- *createInterferenceGraph(InterferenceGraphStruct&ig,Instructions&listaInstrukcija)*

  -Za svaku definiciju promenljive u čvoru koji nije MOVE, u graf interferencije stavljamo 1 između date promenljive i svih promenljivih koje žive na izlazu čvora

- *printInterferenceGraph(InterferenceGraphStruct&ig,Variables&listaPromenjivih)*

  -ispis grafa interferencije sa oznacenim poljima 0 u slučaju nepostojanja interferencije odnosno 1 u suprotnom.

4.6.2 kreiranje steka simplifikacije

Stek simplifikacije kreiramo pozivom funkcije .*CreateStack(Stack& simplificationStack, Variables& listaPromenjivih, InterferenceGraphStruct& ig)* nad objektom klase *ResourceAllocation.* Stek se kreira primenom sledeće heuristike: Pretpostavimo da čvor M grafa G ima manje od K suseda. Neka je G` graf koji se dobija uklanjanjem čvora M, G` = G – {M}. Očigledno je da ako je moguće obojiti G` onda je moguće obojiti i G, pošto kad se čvor M doda obojenom grafu G`, njegovi susedi mogu biti obojeni sa najviše K-1 boja, tako da uvek postoji slobodna boja za čvor M. Ovo prirodno vodi ka rekurzivnom algoritmu bojenja grafa, u kom se čvorovi čiji je rang manji od K postupno uklanjaju iz grafa smetnji i guraju na stek. Svako uprošćavanje dovodi do snižavanja ranga čvora susednih čvoru koji se uklanja, čime se stvara

mogućnost za dalje uprošćavanje. U slučaju nemogućnosti bojenja grafa odnosno kreiranja steka simplifikacije program prijavljuje grešku i prestaje sa radom. U našem konkretnom slučaju ako nakon uprošćavanja i dalje postoje čvorovi sa 4 ili više suseda znači da nije moguće kreirati stek simplifikacije. Tada se u praksi prelazi na fazu prelivanja promenjivih u memoriju ali kao što je ranije rečeno zbog preobimnosti posla tu fazu preskačemo. Unutar funkcije kreiranja steka koristimo još 2 funkcije:

1)*ResourceAllocation::pushToStack(int x, Variables& listaPromenjivih, Stack& simplificationStack)*

-smešta i-tu promenjivu iz liste promenjivih na stek

2)ResourceAllocation::removeFromVector(int x, std::vector<std::vector<int>>& vektor)

-uklanja i-ti int iz vektor vektora int-ova pri čemu ima bool povratnu vrednost koja nam ukazuje da li smo stigli do kraja algoritma

4.6.3 dodela registara

Poslednja faza dodele resursa nakon već odrađenog steka simplifikacije predstavlja šablonski posao, odnosno skidanje jedne po jedne promenjive sa steka i u slučaju interferencije sa nekom prethodno skinutom sledi dodela novog registra ili u suprotnom dodela istog. Postižemo je pozivom funkcije *.doResourceAllocation(Stack* simplificationStack, InterferenceGraphStruct& ig)* nad prethodno napravljenim objektom klase *ResourceAllocation.* Povratna vrednost funkcije dodele registara je bool koji u slučaju neuspele dodele vraća vrednost *false* i ispisuje da dodela nije moguća.

4.7 ispis u fajl

Poslednja faza samog programa ispisuje uspešno preveden kod u obliku razumljivom Mips asemblerskom jeziku. Ispis vršimo pozivajući funkciju *.printIntoFile(Instructions& listaInstrukcija, Variables& listaRegistarskihPromenjivih, Variables& listaMemorijskihPromenjivih,Functions& listaFunkcija)* nad objektom klase *File* kojem pri kreiranju kao parameter konstruktora prosleđujemo listu labela neophodnu za njen pravilan ispis u fajl. Takođe unutar funkcije ispisa koristimo funkciju *File::printLabel(int x)* kojoj

prosleđujemo broj funkcije na kojoj se nalazimo i ukoliko pre nje postoji labela, funkcija će je upisati u fajl.

5. testovi

Program je testiran za niz grešaka koje korisnik može da napravi, neke od njih su:

-2 put deklarisana ista promenjiva

- korišćenje nedeklarisane promenjive

-2 put deklarisana ista labela

-2 put deklarisano isto ime funkcije

-nemogućnost bojenja registara za arhitekturu od 4 realna registra

6. dodatak

Program pored osnovnih stvari sadrži i kontrolni ispis koji možemo dobiti pozivajući funkcije nad objektom klase *Print* :

- printVariableslist(Variables& listaPromenjivih)
- printInstructionList(Instructions& listaInstrukcija)
- printInterferenceGraph(InterferenceGraphStruct &ig, int size)

Takođe u program su dodate gore pomenute and,andi i lb instrukcije tako što smo unutar *FiniteStateMachine* dodali 3 terminalna stanja i namestili da date instrukcije dovode do njih. Osim toga dopunili smo odgovarajućim enumeracijama već date enumeracije kako bi nam program raspoznavao dodate instrukcije.

7. Zaključak

Napravljen je i verifikovan MAVN prevodilac koji prevodi sa MAVN asemblerskog jezika višeg nivoa na MIPS asemblerski jezik. Verifikacija je rađena kroz brojne test primere koji su potvrđivali mogućnost programa na adekvatno prijavljivanje odnosno ispisivanje greške. Ono čime se još mogao dopuniti uradjen prevodilac pored svakako očiglednog nedostatka faze prelivanja je i paralelizacija određenih nezavisnih delova koda. Konkretno sintaksne analize i skladištenja ulaznog koda u kontejnere.

.