УНИВЕРЗИТЕТ У НОВОМ САДУ

ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА

НОВИ САД

ИСПИТНИ РАД

Кандидат: Марко Његомир

Број индекса: sw-38-2018

Предмет: Системска програмска подршка

Тема рада: MAVH преводилац за МИПС асемблерски језик

Нови Сад, Јун, 2020.

1. Увод

МАВН(Мипс асемблер високог нивоа) је алат који преводи програм написан на вишем асемблерском језику на нижи 32бит језик. Постоје значајне сличности између превођења са виших програмских језика на асемблерски језик и превођења са вишег асемблерског језика на основни.

Због тога је потребно применити исте кораке као и код превођења са виших програмских језика. При томе треба имати на уму да је неке кораке могуће упростити јер је код већ доста сличан коду који треба да се добије у овом поступку.

Мој лични циљ је био да у овом пројекту уведем оптимизације које ће умањити број регистара који је потребан да би се програм превео.

2. Анализа проблема

Реч је о 32бит програмском језику чија приложена граматика подржава само целе бројеве. То значајно олакшава превођење јер се елеминише велики број инструкција које раде са вредностима у двострукој прецизности.

Потребно је имплементирати 10 датих инструкција, и 3 по избору. Тако мали број инструкција ограничава број проблема који могу да настану приликом превођења, и отвара могућност за агресивније оптимизације кода.

Граматика МАВН језика дозвољава да више функција буду за редом а да немају ни једну инструкцију која им припада, док лабеле морају имати бар једну инструкцију.

Регистарске променљиве се третирају као и обичне променљиве у вишим програмским језицима, с тим да се овде не води рачуна о типу података јер увек садрже целобројну вредност.

Постоји само једна инструкција за смештање садржаја у меморију, и то отвара пут даљим оптимизацијама кода које су касније објашњене.

Величина непосредног операнда је 16 бита, што омогућава да се неке вредности из меморије потенцијално пребаце у непосредни операнд у инструкцији која мења стару инструкцију.

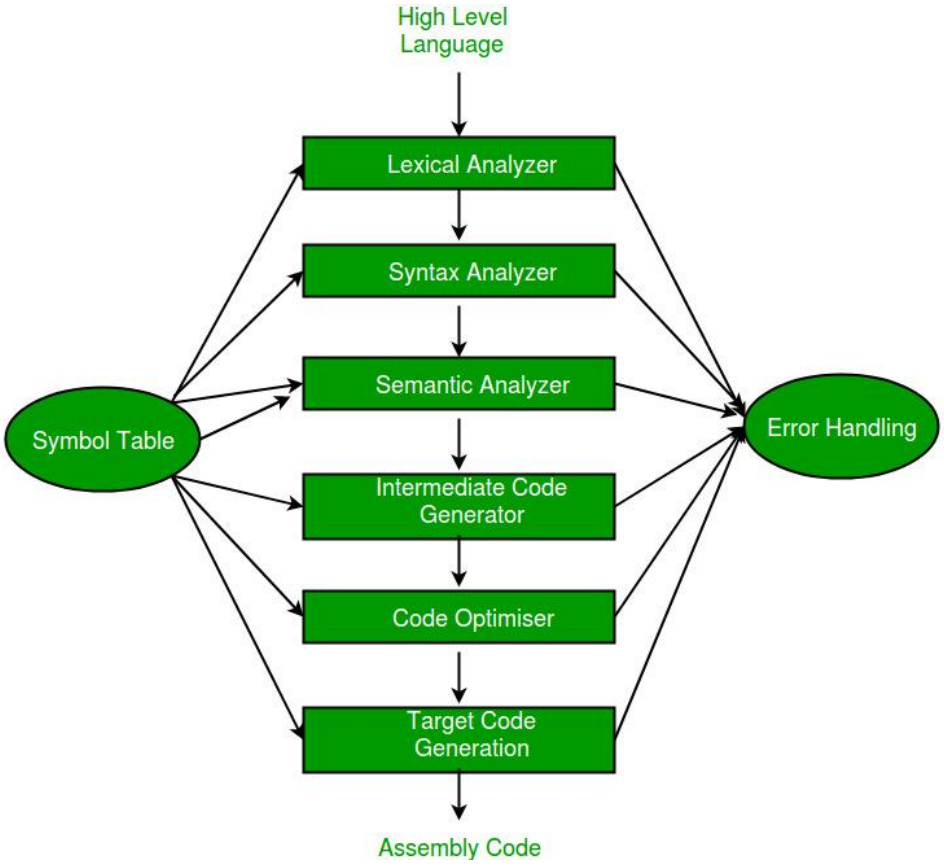
Треба водити рачуна да sw инструкција и инструкције за сранање воје операнде третирају као изворне, иако један од њих садржи адресу одредишта.

Због свега наведеног један приступ за решавање овог проблема је да се прате модификовани кораци за превођење програма високог нивоа на асемблерски код, почевши од лексичке анализе а завршавајући са доделом регистара и оптимизацијама.

3. Концепт решења

Идеја је да се прође кроз следеће фазе приликом превођења:

1. Лексичка анализа
2. Синтаксна анализа
3. Семантичка анализа која није засебно издвојена већ је део других фаза
4. Фаза анализе животног века
5. Фаза доделе регистара која обухвата
    i. Формирање графа сметњи
    ii. Процес упрошћења графа
    iii. Процес бојења графа
    iv. Фаза преливања уколико је то потребно
6. Додатне оптимизације које су део претходно наведених корака



Слика 1 Фазе приликом превођења (преузето са geeksforgeeks сајта)

Решење је осмишљено тако да се раздвоји већина функција у посебне модуле, а да се подаци сместе у један модул који је овде представљен табелом симбола.

Због тога сам се овде одлучио да применим шаблон “посетилац” који наглашава овакву поделу података и функција који раде са тим подацима.

Направљене су апрстрактне класе Program Data и Visitor. Прва имплементира чисту виртуалну методу accept која има један апраметар а то је конст референца на посетиоца. Табела симбола ће наследити класу ProgramData. Метода виситор са друге стране има чисту виртуалну методу visit која у овом случају има један параметар а то је конст референца на табелу симбола.

Шаблон функционише тако што класа која садржи податке помоћу методе прихвати класу која посетиоца која има методе које раде над тим подацима. Она затим позове методу висит од прихваћене класе, и проследи јој референцу на саму себе, и на тај начин омогући да класа посетиоц може да ради са подацима.

Пошто табела симбола садржи скоро све релевантне податке у програму, она је задужена и за ослобађање динамички заузете меморије.

Осим табеле симбола и класа које су укључене у фазе превођења, постоје и класа која ради са фајловима, класа која моделује елементе МИПС програмског језика и друге помоћне класе.

Реаговање на грешке и прослеђивање порука о грешкама приликом превођења је дсмишљено да се ради помоћу изузетака.

Програм треба да омогући и да се више пута обави фаза преливања уколико је то потребно, али и да не упадне у бесконачну петљу, већ да заврши са преливањем уколико су сви регистри већ преливени у меморију.

Приликом фазе преливања је најпогодније применити агресивне оптимизације са циљем да се смањи број потребних регистара.

Коначно, када се све фазе успешно окончају потребно је генерисати асемблерски код.

Додатна функционалност програма коју сам из радозналости имплементирао је да може да учитава код из имена фајлова. Ти фајлови су заправо празни, и пријављена величина им је 0 бајтова. Али сав код се налази или у бинарном или у хексадецималном запису у њиховим именима. Сваки од њих због тога име и идентификациони број, да би се могао одредити редослед којим их треба преводити.

4. Програмско решење

4.1 Лексичка анализа

У лексичкој анализи се користи детерминистички аутомат са коначним бојем стања. Сви валидни токени морају проћи кроз њега. Неки токени неће изазвати грешку али се неће ни прикупљати.

Уколико се анализа успешно обави, листа токена се прослеђује следећој фази а то је фаза синтаксне анализе.

4.2 Синтаксна анализа

Имплементирана је синтаксна анализа са рекурзивним спуштањем. Она у потпуности мора да поштује правила дефинисана граматиком, и да пријављује синтаксне грешке.

Битна разлика у односу на неке друге преводиоце је та што се овде не генерише стабло синтаксне анализе које би касније требало да обрађено у фази доделе инструкција, већ се овде директно додељују одговарајуће инструкције.

Пре него што се пређе на следећу фазу, а то је анализа животног века, потребно је повезати инструкције. И при томе је могуће извршити и проверу да ли постоје све лабеле које су мета инструкција за гранање. Ово је урађено тек након што се у фази синтаксне анализе прикупе све информације о променљивама, фунцкијама и лабелама у програму. Тако је омогућено да нека ранија инструкција скаче на лабелу која се среће касније у коду.

Повезивање инструкција је реализовано у класи табела симбола.

Табела симбола поседује осим тога још и функције за додавање инструкција, за додавање и проверу променљивих, лабела, фунцкија и других структура. Неке од провера као што су провере имена користе регуларне изразе.

4.3 Фаза анализе животно века

Ова фаза за циљ има да одреди које су променљиве живе линији која повезује инструкције. То значи да се одређују ин и аут скупови променљивих.

Алгоритам је имплементиран тако што се пролази кроз инструкције од назад, и затим се попуњава прво аут скуп па тек затим ин скуп, и при томе ин скуп користи управо ажурирани аут скуп. На тај начин се убрзава ова фаза.

Попуњавање скупова се обавља по формули:

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

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

Слика 2 Формула за попуњавање ин и оут скупова у фази анализе животног века (материјали са вежби)

Процес попуњавања скупова се понавља све код се не деси да у две сукцесивне итерације није било промене у садржају скупова.

Све до ове фазе није било могућности преливања, а од следеће фазе постоји и та могућност. Али ова фаза јесте захваћена фазом преливања због тога што када се преливање деси, читав процес повезивања инструкција и анализе животног века мора такође да се понови.

4.4 Фаза доделе ресурса

Као што је раније напоменуто, ова фаза се састоји од 3 обавезне фазе, и постоји могућност одрађивања додатне фазе преливања ако за то буде потребе.

4.4.1 Фаза одређивања графа сметњи

За сваку дефиницију променљиве се додаје сметња са сваком променљивом која живи на излазу чвора. Резултат се смешта у матрицу сметњи која се корсти у наредним фазама доделе ресурса.

4.4.2 Фаза упрошћавања

У овој фази се скидају променљиве са графа сметњи, и то само оне чији је ранк мањи од задатог броја регистара. Увек се бира чвор највећег ранка који задовољава тај услов.

Сваки скинути чвор ставља се на помоћну структуру стек.

Ако се деси да се не могу скинути сви чворови, активираће се фаза преливања.

4.4.3 Фаза бојења графа (доделе регистара)

У овој фази се пролази кроз стек са променљивама и то у обрнутом редоследу од онога на који су додавани. Затим се свакој променљивој покуша доделити прва слободна боја коју не заузимају њени суседи.

Ако је могуће обојити граф, онда је ова фаза успешно обављена и може се генерисати асемблерски код.

Уколико није могуће свакој променљивој доделити боју, онда се одлази у фазу преливања.

4.4.4 Фаза преливања и додатних оптимизација

У фази преливања имплементирана је хеуристика која тражи променљиву која има највише сметњи у графу сметњи. Идеја је да се та променљива пребаци у меморију тако што се пре сваке инструкције која користи ту променљиву додају две инструкције за учитавање из меморије, а након сваке инструкције која дефинише ту променљиву се додају две инструкције које ту вредност премештају у меморију. За ту потребу се прво направи меморијска променљива која ће држати вредност те променљиве.

Прва додатна оптимизација се може одрадити пре претходно описаног поступка и она за циљ има да декомпонује инструкцију sub на две инструкције neg и add од којих се инструкција add касније може заменити инструкцијом addi која премешта вредност из меморије у непосредни операнд и на тај начин умањује број променљивих које се користе у инструкцији.

Друга додатна оптимизација се односи управо на премештање променљивих из меморије у непосредне операнде и адекватном изменом инструкција.

Да би ово било безбедно у овој ограниченој граматици са само једном инструкцијом која смешта вредности у меморију потребно је прво осигурати да се не користе offset вредности у sw инструкцијама. Ако се примети да се бар на једном месту појављује офсет, одма се одустаје од ове оптимизације.

Затим треба проверити да ли су све адресе које се користе у sw инструкцијама неизмењене, тј да ли од тренутка учитавања адресе меморијске локације нема редефинисања тог регистра који садржи ту адресу а пре употребе у sw инструкцији. Ако се ово детектује, одма се одустаје од ове оптимизације.

Након тога се проверава да ли се уопште може нека од меморијских променљивих може заменити са непосредним операндом. Провери се да ли је опсег вредности од -32768 до 32767. Затим се још и провери да ли нека sw инструкција пише у ту меморијску локацију, и ако не пише, тј та вредност у меморији се не мења у току целог програма, она онда постаје кандидат за замену са непосредним операдном.

Затим се пролази кроз листу свих кандидата за замену, и гледа се да ли се можу заменити инструкције LW и ADD које их користе са инструкцијама LI и ADDI.

Ако успе ова замена, онда је тако додатно поједностављен код умањивањем броја регистарских променљивих, па то и омогућава да се преведу програми у асемблерски код који користи само 2 регистра.

Резултујући код ће имати много више инструкција, али ће бити испуњен циљ да се користи најмањи могући број регистара.

4.5 Прављење асемблерског кода

Напослетку, потребно је генерисати асемблерски код и записати га у излазни фајл.

Улазни параметри програма су редом:

1. путања улазног .mavn фајла
2. Путања излазног фајла

Други сет параметара је:

1. Путања улазног .mavn фајла
2. Путања излазног фајла
3. hex ili bin

Овај трећи параметар ће проузроковати да се приликом учитавања програма из улазног фајла, створи копија тог програма која ће помоћу одабране вредности записати тај код у имена фајлова у предодређеном фолдеру. Ови фајлови су величине нула бајтова.

Трећи сет параметара је:

1. Путања улазног фолдера
2. Путања излазног фајла
3. hex или bin
4. Zerobytes

Овако позван програм ће из задатог фолдера учитати програм и претворити га у обичан низ асци карактера, који ће онда превести у асемблерски код.

Занимљива ствар је што ови фајлови са кодом у називу, осим што имају нула линија кода у себи, приказују и да су величине нула бајтова.

Када сам при тестирању пребацио велику количину тих фајлова на клауд, десило се да се ни тамо није повећала заузетост простора.

За сваки фајл на клауду је такође писало да заузима нула бајтова.

5. Верификација

Постоји више тестних .mavn фајлова који демонстрирају неке од наведених концепата и служе за проверавње исправности добијених решења. Решење се може проверити покретањем програма QtSpim. Он пријави грешку ако код није ваљан. А исправност кода се онда може додатно проверити итерирањем кроз инструкције уз помоћ истог алата.

Генерисани су и фолдери и фајлови који демонстрирају рад zeorbytes програма и могуће је проверити и њихову исправност помоћу претходно поменутог алата.

Постоје и тестни случајеви који приказују успешно превођење програма који поседује новододате инструкције.