



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

ПРОЈЕКТНА ДОКУМЕНТАЦИЈА

Кандидат: Нађа Гвозденац
Број индекса: SW 10/2018

Предмет: Системска програмска подршка 1
Тема рада: Пројектни задатак

Ментор рада: проф. др Миодраг Ђукић

Нови Сад, јун, 2020.

САДРЖАЈ

1. Увод	1
1.1 МАВН - Преводаилац.....	1
1.2 Задатак.....	2
2. Анализа проблема	3
3. Концепт решења	4
4. Опис решења	6
4.1 Модули и њихов опис.....	6
4.1.1 Модул токена (Token).....	6
4.1.2 Модул реализације променљивих и инструкција (IR).....	6
4.1.3 Модул коначног детерминистичког аутомата (FiniteStateMachine).....	6
4.1.4 Модул за рад са функцијама (FuncManager).....	6
4.1.5 Модул за рад са лабелама (LblManager).....	7
4.1.6 Модул за рад са променљивама (VarManager).....	7
4.1.7 Модул лексичке анализе(LexicalAnalysis).....	7
4.1.8 Модул синтаксне анализе(SyntaxAnalysis).....	7
4.1.9 Модул анализе животног века(LivenessAnalysis).....	8
4.1.10 Модул доделе ресурса и графа сметњи (InterferenceGraph).....	8
5. Верификација	9
6. Закључак	10

СПИСАК СЛИКА

Слика 1: Улазна датотека (лево) и излазна датотека (десно).....	2
Слика 2: Виши асемблерски језик (лево) и MIPS 32bit асемблерски језик (десно).....	9

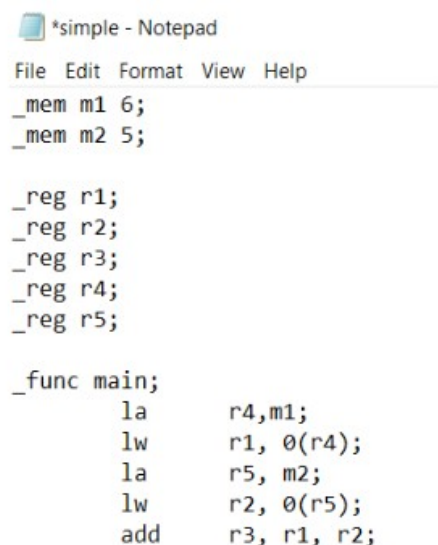
1. Увод

1.1 МАВН - Преводацац

МАВН (Мипс Асемблер Високог Нивоа) је алат који преводи програм написан на вишем MIPS 32bit асемблерском језику на основни асемблерски језик. Виши MIPS 32bit асемблерски језик служи лакшем асемблерском програмирању јер уводи концепт регистарске променљиве. Регистарске променљиве омогућавају програмерима да приликом писања инструкција користе променљиве уместо правих ресурса. Ово знатно олакшава програмирање јер програмер не мора да води рачуна о коришћеним регистрима и њиховом садржају.

1.2 Задатак

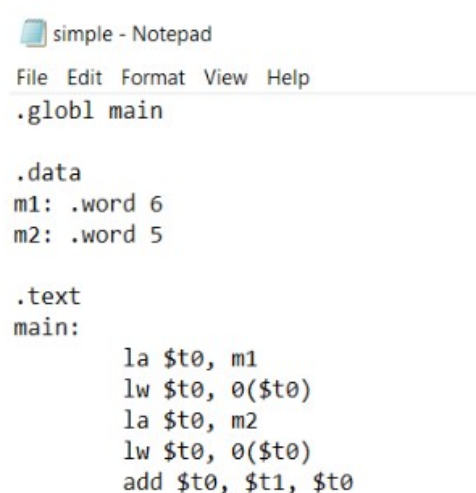
У оквиру пројектног задатка потребно је реализовати МАНН преводац који преводи програме са вишег асемблерског језика на основни MIPS 32bit асемблерски језик. Преводац треба да подржава детекцију лексичких, синтаксних и семантичких грешака као и генерисање одговарајућих извештаја о евентуалним грешкама. Излаз из преводиоца треба да садржи коректан асемблерски код који је могуће извршавати на MIPS 32bit архитектури (симулатору). Пример улазне .mavn и излазне .s датотеке су приказани на слици Слика 1.



```
*simple - Notepad
File Edit Format View Help
_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 - Notepad
File Edit Format View Help
.globl main

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

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

Слика 1: Улазна датотека (лево) и излазна датотека (десно)

Програм на МАНН језику из улазне .mavn датотеке је потребно учитати.

Над учитаним подацима потребно је извршити лексичку анализу. Уколико је лексичка анализа успешно завршена, односно лексичке грешке нису пронађене, потребно је извршити синтаксну анализу. Приликом синтаксне анализе врши се обрада података („токена“) које је претходно креирао лексички анализатор и креирају се инструкције.

Након успешно завршене синтаксне анализе потребно је извршити анализу животног века променљивих и доделу ресурса користећи граф сметњи.

У излазну .s датотеку је потребно уписати резултат превођења, односно програм на MIPS 32bit асемблерском језику.

Потребно је реализовати још три произвољне инструкције од којих барем једна није из аритметичко-логичког скупа инструкција.

2. Анализа проблема

Главни проблеми задатка су генерисање одговарајућег MIPS 32bit асемблерског кода, детекција лексичких, синтаксних и семантичких грешака и генерисање одговарајућих извештаја о истим.

Потребно је предвидети и пазити на све могуће врсте грешака које се могу појавити приликом превођења.

Како би детекција грешака омогућила адекватно исправљање истих, потребно је за сваки увиђени тип грешке генерисати одговарајући извештај који ће исправљање грешака олакшати.

Потребно је реализовати још три произвољне инструкције од којих барем једна није из аритметичко-логичког скупа инструкција. У овом решењу пројектног задатка су реализоване инструкције *and*, *or* и *lb*.

3. Концепт решења

Да бисмо програм успешно превели, потребно је проћи неколико фаза анализе полазног програмског кода и генерисања излазног програмског кода.

Прва фаза је лексичка анализа програма помоћу лексичког анализатора. Лексички анализатор је алат који улазни низ карактера претвара у низ токена – лексичких симбола који одговарају речима неког програмског језика. Реализован је помоћу коначног детерминистичког аутомата чији је циљ да препозна све симболе у програму. Из датотеке се редом читају карактери и шаљу у коначни аутомат, одакле се, у зависности од карактера, прелази у наредно стање. Приликом откривања симбола се користи правило најдуже речи. Под овим правилом подразумева се читање знакова из програма и прелазак из стања у стање аутомата све док се не пријави грешка лексичког анализатора. Излаз из лексичког анализатора је листа токена прочитаних из улазне датотеке.

Након успешно завршене лексичке анализе, прелази се на синтаксну анализу. Да би се сазнало да ли је нека реченица граматички исправна одређује се њено порекло – обавља се извођење. Извођење се обавља тако што се креће од почетног симбола, након чега се у итерацијама уместо нетерминалних симбола смењују њихове продукције. Синтаксни анализатор је реализован помоћу алгорита са рекурзивним спуштањем. Алгоритам са рекурзивним спуштањем поседује по једну функцију за сваки нетерминални симбол и по један услов за сваку продукцију. Приликом извођења, једна од функција која је у употреби је функција *eat*. Функцији *eat* прослеђује се симбол који се очекује да ће бити следећи у

програму и колико се очекивани симболне поклапа са новооткривеним симболом потребно је пријавити синтаксну грешку.

Приликом синтаксне анализе се обавља и фаза генерисања инструкција.

Након успешно завршене синтаксне анализе, потребно је извршити анализу животног века променљивих. Да би се омогућило да различите променљиве које нису истовремено у употреби деле исти ресурс, неопходна је информација о животном веку променљиве. Животни век променљиве започиње дефиницијом променљиве (упис почетне вредности), а завршава се последњом употребом (последње читавање садржаја променљиве). Променљива је у неком сегменту жива ако садржи вредност која ће бити коришћена касније у току извршења програма. Први корак у анализи је конструисање графа тока управљања који се конструише тако што се свака инструкција представи чвором графа, а затим се за свака два чвора x и y , такве да x претходи y , повлачи стрелица од x ка y . Променљива је жива на стрелици графа ако постоји усмерена путања од те стрелице до чвора који користи ту променљиву, а да при томе путања не прелази преко иједног чвора који дефинише исту променљиву. Променљива је жива на улазу чвора ако је жива на било којој улазној стрелици чвора, а жива је на излазу у случају да је жива на било којој излазној стрелици чвора. Информација о животу променљивих се може изразити на основу скупова променљивих које чвор дефинише и користи.

Након извршене анализе животног века променљивих следи фаза доделе ресурса. Проблем који онемогућава да се исти регистар додели двома привременим променљивама назива се сметња. Најчешћи узрок сметње је преклапање опсега животног века променљивих. Граф сметњи се често у рачунару представља матрицом сметњи и правимо га на основу резултата анализе животног века променљивих.

Фазе у компајлеру које претходе фази доделе ресурса подразумевају да на циљаној платформи постоји неограничен број регистара (и ресурса уопште) у које се смештају променљиве. Фаза доделе ресурса има задатак да тај неограничени број регистара сведе на тачно онај број регистара колико има на циљаној платформи. Проблем доделе ресурса се често своди на проблем бојења графа. Код бојења графа потребно је сваком чвору доделити неку боју, али тако да суседни чворови нису обојени истом бојом. У најпростијој форми проблема, циљ је обојити произвољни граф са најмањим бројем боја. У овом случају нам је граф који желимо да бојимо управо граф сметњи, а боје конкретни ресурси циљне архитектуре.

4. Опис решења

4.1 Модули и њихов опис

4.1.1 Модул токена (Token)

Овај модул садржи класу токена.

Класа садржи одговарајуће `get` и `set` методе као и методе за креирање и испис токена различитог типа.

4.1.2 Модул реализације променљивих и инструкција (IR)

Овај модул садржи класу инструкција, променљивих и инструкција које садрже токен број.

Модул садржи одговарајуће `get` и `set` методе као и методе за креирање и испис променљивих и инструкција.

4.1.3 Модул коначног детерминистичког аутомата (FiniteStateMachine)

Модул у ком је реализован коначни детерминистички аутомат. Срж аутомата је матрица стања помоћу које се врши прелазак у наредно стање на основу прочитаног симбола.

Матрица прелаза стања:

```
static const int stateMatrix[D_NUM_STATE][D_NUM_OF_CHARACTERS];
```

4.1.4 Модул за рад са функцијама (FuncManager)

Модул који садржи класу `FuncManager`.

Класа садржи методе за додавање нове функције и проверу да ли је функција дефинисана.

```
bool checkFunc(string name) – враћа true ако функција са именом name већ постоји  
void addFunc(Token& t, int position) – додаје нову функцију
```

4.1.5 Модул за рад са лабелама (LblManager)

Модул који садржи класу LblManager и класу Label.

Класа садржи методе за додавање нове лабеле, проверу да ли је лабела дефинисана и методу за добављање позиције лабеле.

bool checkLabel(string name) – враћа true ако лабела са именом name већ постоји

void addLabel(Label l) – додаје нову лабелу

int getPosFromLabel(string name) – враћа позицију лабеле

4.1.6 Модул за рад са променљивама (VarManager)

Модул који садржи класу VarManager.

Класа садржи методе за додавање нове променљиве, проверу да ли је променљива дефинисана и методу за креирање стринга секције података.

Методе за проверу да ли је променљива са именом name већ постоји:

Variable* chackMemVar(string name)

Variable* chackRegVar(string name)

Методе додавање променљивих:

void addMemVar(Token& t, Token& v)

void addRegVar(Token& t)

4.1.7 Модул лексичке анализе(LexicalAnalysis)

Модул који садржи лексички анализатор и врши лексичку анализу.

Ослања се на модуле токена и аутомата коначног стања.

Добављање токена у зависности од тренутног стања аутомата:

Token LexicalAnalysis::getNextTokenLex()

4.1.8 Модул синтаксне анализе(SyntaxAnalysis)

Модул који садржи синтаксни анализатор и врши синтаксну анализу и креирање инструкција.

Ослања се на модул лексичке анализе и модуле за рад са променљивама, лабелама и функцијама.

Приликом реализације задатка користити методу:

void eat(TokenType t)

која служи за проверу синтаксне исправности тренутног токена. Уколико тренутни токен није очекиваног типа (прослеђеног типа), метода исписује поруку о синтаксној грешци.

Ма метода за креирање нове инструкције:

void createInstruction(InstructionType type, vector<Token>& dst, vector<Token>& src)

4.1.9 Модул анализе животног века(LivenessAnalysis)

Модул који врши анализу животног века променљивих.

Садржи функцију која обавља поменућу анализу:

```
void livenessAnalysis(Instructions instructions)
```

4.1.10 Модул доделе ресурса и графа сметњи (InterferenceGraph)

Модул који врши доделу ресурса.

Садржи класу `InterferenceGraph` која садржи граф (матрицу) сметњи и стек променљивих. Такође садржи одговарајуће методе за иницијализацију и попуњавање графа.

Метода за креирање графа:

```
void buildGraph(Instructions& instructions)
```

Метода за креирање стека променљивих:

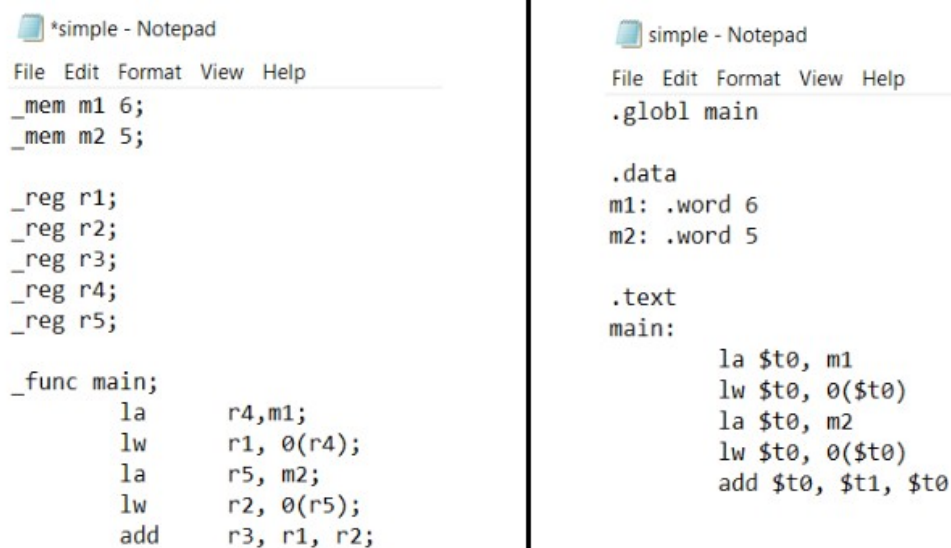
```
void buildVarStack()
```

Метода која врши доделу ресурса:

```
bool doResourceAllocation()
```

5. Верификација

Преводаилац је тестиран користећи датотеке које садрже програм написан на MABH асемблерском језику. Излазни програмски код је проверен коришћењем QtSpim симулатора. На следећој слици је приказана улазна датотека и њој одговарајућа излазна датотека:



```
*simple - Notepad
File Edit Format View Help
_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 - Notepad
File Edit Format View Help
.globl main

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

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

Слика 2: Виши асемблерски језик (лево) и MIPS 32bit асемблерски језик (десно)

У случају да дође до грешке, једна од следећих порука ће бити исписана на екран:

- Exception! Lexical analysis failed! - грешка приликом лексичке анализе
- Exception! Syntax analysis failed! - грешка приликом синтаксне анализе
- Exception! Resource allocation failed! - грешка приликом доделе ресурса

У сваком од ових случајева се исписује узрок грешке у зависности од случаја.

6. Закључак

Направљен је и верификован МАВН преводац који преводи програме са вишег асемблерског језика на основни MIPS 32bit асемблерски језик.

Верификација секвенцијалног програма урађена је помоћу других софтверски алата попут QtSpim симулатора. Успешним извршавање програма у излазној датотеци потврђена је исправност МАВН преводиоца.