



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

ДОКУМЕНТАЦИЈА

Кандидат: Павле Вуковић
Број индекса: RA135/2019

Предмет: Системска програмска подршка у реалном времену I
Тема рада: Пројекат из СППуРВ I

Нови Сад, јун, 2021.

САДРЖАЈ

1. Анализа проблема.....	3
2. Концепт решења.....	4
2.1 Лексичка анализа.....	4
2.2 Синтаксна анализа.....	5
2.3 Избор инструкција.....	6
2.4 Анализа животног века променљивих.....	6
2.5 Додела ресурса.....	7
3. Програмско решење.....	8
3.1 Синтаксна анализа.....	8
3.2 Анализа животног века променљивих.....	9
3.3 Додела ресурса.....	9
4. Верификација.....	11

1. АНАЛИЗА ПРОБЛЕМА

Циљ овог пројекта јесте да на основу почетног, унапред датог улазног фајла, који представља програм написан на вишем асемблерском језику, добијем програм преведен на нижи, тј. основни МИПС 32-битни асемблерски језик. Самим тим јасно је да је реч о имплементацији својеврсног компајлера. Да бих ово и постигао, неопходно је да прођем кроз низ различитих фаза, као што су: анализа почетног кода (која је подељена на: лексичку, синтаксну и семантичку анализу), формирање граматике програмског језика, избор инструкција, на основу предходно имплементираних елемената, анализа животног века променљивих, која се наслања на избор инструкција, те напоследку додела ресурса, као логичан наставак анализе животног века. Када сам извршио све ове фазе, сада могу исписати преведени код у одговарајућем асемблерском језику.

2. КОНЦЕПТ РЕШЕЊА

Као што је горе споменуто, решење се састоји из више фаза. Свака од фаза се наставља једна на другу и када се једном успешно изврши њихово повезивање, долазимо до жељеног резултата. Прва од фаза је Лексичка анализа, стога одмах прелазимо на објашњење исте.

2.1 Лексичка анализа

Лексичка анализа је део решења који је већ претходно имплементиран, тј. добио сам га уз поставку задатка. Извршио сам благе измене како бих добио одговарајући испис на конзоли. Осим тога, све је већ било написано. Она је имплементирана у оквиру заглавља *"LexicalAnalysis.h"* и изворног фајла *"LexicalAnalysis.cpp"*. Покреће се из функције: *main()*, позивом конструктора лексичке анализе: *LexicalAnalysis lex;*, након тога следи функција: *initialize()*, која служи да иницијализујем лексичку анализу и коначни аутомат: *FiniteStateMachine*. Овај аутомат је реализован у заглављу: *FiniteStateMachine.h* и изворном фајлу *FiniteStateMachine.cpp*, а користи се како бих препознао све симболе у програму. Наиме, циљ лексичке анализе јесте да она прође кроз цео улазни фајл, и да све симболе на које наилази пребаци у листу токена, коју ће проследити синтаксној анализи на проверу исправности исте, при чему се умногоме ослања на горе

напоменути аутомат. Аутомат је такође прошао кроз благе промене, наиме додата су поједина стања , као последица нових инструкција које постоје у мојој граматичи. Лексичка анализа се покреће функцијом ***Do()***, чијим се извршавањем формира листа токена, а враћа се одговарајућа ***bool*** вредност ***true*** или ***false***, у зависности од тога да ли је извршење функције било успешно или не.

2.2 Синтаксна анализа

Синтаксна анализа је осмишљена тако да, на основу граматике, прође кроз листу токена, која јој је прослеђена од стране лексичке анализе и утврди да ли је програм написан у складу са граматиком. У мом решењу, због једноставности , спојио сам синтаксну анализу са избором инструкција, тј. прескочио сам фазу: Формирање стабла међукода, те сам одмах приликом провере синтаксне исправности кода, читавао променљиве и формирао инструкције.

$Q \rightarrow S ; L$	$S \rightarrow _mem \ mid \ num$	$L \rightarrow eof$	$E \rightarrow and \ rid, \ rid, \ rid$
	$S \rightarrow _reg \ rid$	$L \rightarrow Q$	$E \rightarrow b \ id$
	$S \rightarrow _func \ id$		$E \rightarrow add \ rid, \ rid, \ rid$
	$S \rightarrow id: \ E$		$E \rightarrow lw \ rid, \ num(rid)$
	$S \rightarrow E$		$E \rightarrow abs \ rid, \ rid$
			$E \rightarrow sw \ rid, \ num(rid)$
			$E \rightarrow xor \ rid, \ rid, \ rid$
			$E \rightarrow bltz \ rid, \ id$
			$E \rightarrow or \ rid, \ rid, \ rid$
			$E \rightarrow la \ rid, \ mid$

Слика 1. Граматика коју је требало имплементирати

Синтаксна анализа се имплементирана у заглављу: *SyntasAnalysis.h*, као и изворном фајлу: *SyntaxAnalysis.cpp*. Њено се извршавање такође позива из функције *main()*. Користи се функције *Do()*, на веома сличан начин као и код лексичке анализе. Важно је знати да се њеном конструктору обавезно прослеђује објекат класе Лексичка анализа. У зависности од тога, да ли је синтаксна анализа извршена успешно или не, имам одговарајући испис на конзоли.

2.3 Избор инструкција

Као што је претходно споменуто, избор инструкција је реализован у синтаксној анализи, због једноставности. Наиме, ја приликом проласка кроз инструкције, које су описане у граматичи, могу да ако се испостави да је инструкција написана у складу са граматиком, позивом функције: *napraviInstrukciju(vector<Token>& destination, vector<Token>& source, InstructionType tip_instrukcije)*, у зависности од типа инструкције, направим инструкцију и убацим је на листу инструкција, која представља својеврстан излаз из синтаксне анализе.

2.4 Анализа животног века променљивих

Ово је фаза, у оквиру које, на основу листе инструкција, треба подесити одговарајућа поља, која су одлика сваке инструкције, као што су листе променљивих које се користе у инструкцији (*m_use*), које се дефинишу (*m_def*), које су претходнице (*m_pred*) и

следбенице инструкције(*m_succ*). Главни део приче је подешавање променљивих које су живе на улазу(*m_in*) и излазу (*m_out*) из инструкције. Цео овај процес се назива анализа животног века променљивих и имплементиран је у заглављу: ***IR.h***, које је само прилагођено појединим функцијама, као и изворном фајлу: ***LivenessAnalysis.cpp***. Извршење фазе се позива у функцији: *main()*, а уколико је све адекватно прошло десиће се испис листе инструкција са свих подешеним параметрима.

2.5 Додела ресурса

Ово је последња фаза у компајлирању. Њен је циљ да регистарским променљивама, које су претходно учитане у оквиру синтаксне анализе, додели које ће ресурсе, тј. регистре да користе. Наиме на асемблеру вишег нивоа је могуће да се уместо регистара користе константе и променљиве уместо правих регистара, а сада се свакој променљивој додељује посебан регистар. Да бих извршио доделу ресурса, неопходни су ми: граф сметњи, који се прави на основу анализе животног века, стек који се прави на основу поједностављења графа сметњи. Треба утврдити да ли је могуће свим променљивама доделити К регистара, а да не дође до промене понашања програма. Уколико то није могуће долази до преливања. Додела ресурса је имплементирана у заглављу: ***InterferenceGraph.h***. У функцији: *main()*, позивом функције: *doResourceAllocation(stack<Variable*>* simplificationStack,*

InterferenceGraph ig*), извршавам последњу фазу доделе ресурса у зависности од чије успешности исписујем да ли је дошло до преливања или је додела успешно завршена. Након овога би требало исписати на конзолу преведени асемблерски код, што ја нажалост нисам стигао урадити.

3. ПРОГРАМСКО РЕШЕЊЕ

Поједини делови, као нпр. лексичка анализа су унапред одрађени, тако да ћу њих изоставити, базираћу се на ономе што сам сам имплементирао, почевши са синтаксном анализом.

3.1 Синтаксна анализа

Направљена су два нова фајла, заглавље: *SyntaxAnalysis.h* и изворни фајл *SyntaxAnalysis.cpp*. Њихов костур сам прекопирао из одговарајућег домаћег задатка. Имплементирао сам функције: *Q()*, *S()*, *L()* и *E()*, које сам попунио тако да проверавају да ли се код поклапа са граматиком. У функцији *S()*, користио сам функције: *ubaciMemorijskuVarijablu(Token& token)*, као и њој сличну *ubaciRegistarskuVarijablu(Token& token)*, чији је смисао да на листу свих променљивих које користим у програму *Variables variable* убацам одговарајуће променљиве, које претходно направим.

У функцији *E()*, након што установим синтаксну исправност неке инструкције, зовем функцију *napraviInstrukciju(vector<Token>&*

destination, vector<Token>& source, InstructionType tip_isntrukcije), која на основу параметара, вектора токена који ће се касније превести у листу одговарајућих променљивих и типа инструкције, прави инструкцију, која се убацује на листу свих инструкција. Тиме сам извршио попуњавање листе инструкција, што је задатак фазе избора инструкција.

3.2 Анализа животног века променљивих

Имплементирана је у оквиру изворног фајла: *LivenessAnalysis.cpp*. Нове функције: *prethodnici(Instructions& instructions)*, *sledbenici(Instructions& instructions)*, *ubaciVariable(Instructions& instructions)*, се користе да бих иницијализовао листе *m_pred*, *m_succ*, *m_use* и *m_def* респективно. Затим ту је и функција: *livenessAnalysis(Instructions& instructions)*, која подешава листе *m_in* и *m_out*. Након анализе животног века, свака инструкција у листи ће бити измењена, тј. одговарајуће, малочас споменуте листе ће бити освежене. Такође у овом фајлу је и функција: *ispisiInstrukcije(Instructions& instructions)*, коју ћу користити за испис листе инструкција, пре и после анализе животног века.

3.3 Додела ресурса

Имплементирана је у заглављу *InterferenceGraph.h*. Имплементирао сам све четири фазе доделе. Прва, формирање графа сметњи, се извршава позивом функције *doInterferenceGraph(Instructions& instructions, Variables&*

registarske_promenljive). У оквиру ње се врши и иницијализација и формирање матрице сметњи, као саставног дела графа сметњи. Граф је представљен класом *InterferenceGraph*, која као поља има инструкције које се користе, регистарске променљиве које се користе, матрицу сметњи и димензију исте.

Друга фаза, фаза упрошћавања је направљена у оквиру функције *doSimplification()*, која је метода класе *Stack*. У оквиру ње, вршим поједностављивање графа сметњи, као и пребацивање променљивих на стек, самим тим и формирање стека.

Трећа фаза је само провера да ли долази до преливања, где преливање није реализовано.

Четврта фаза је фаза избора. У оквиру ње бојим чворове, тј. на основу стека, скидам променљиве са истог и додељујем им боју, тј. регистар, тако да се за сваке две променљиве које су међусобно повезане додели различит регистар. Ово је написано у оквиру функције *doResourceAllocation(stack<Variable*>* simplificationStack, InterferenceGraph* ig)*.

Након овога, требало је написати функције које ће извршити испис преведеног асемблерског кода на конзолу, али то нисам стигао урадити.

4. ВЕРИФИКАЦИЈА

Како бих верификовао решење, тј. део решења који сам урадио, користио сам четири улазна фајла:

1. sintaksnagreksa.mavn

```
1  _mem error 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;
```

Слика 2 . Изглед улазног фајла

Идеја тог улаза је била да изазовем синтаксну грешку, тј. да убацивањем речи **error**, у првој линији наведем да ми конзола испише синтаксну грешку. Са слике 3. видимо да је ми је то и пошло за руком.

```
C:\Users\Pavle\Desktop\SPPURVARA\Projekat5\src\Debu...
[T_R_PARENT]      )
[T_SEMI_COL]      ;
[T_LA]            la
[T_R_ID]           r5
[T_COMMA]          ,
[T_M_ID]           m2
[T_SEMI_COL]      ;
[T_LW]            lw
[T_R_ID]           r2
[T_COMMA]          ,
[T_NUM]            0
[T_L_PARENT]      (
[T_R_ID]           r5
[T_R_PARENT]      )
[T_SEMI_COL]      ;
[T_ADD]           add
[T_R_ID]           r3
[T_COMMA]          ,
[T_R_ID]           r1
[T_COMMA]          ,
[T_R_ID]           r2
[T_SEMI_COL]      ;
[T_END_OF_FILE]   EOF
_mem
Syntax error! Token: error unexpected
Syntax error! Token: error unexpected
Syntax analysis has finished unsuccessfully!

ISPIS INSTRUKCIJA NAKON UCITAVANJA...
```

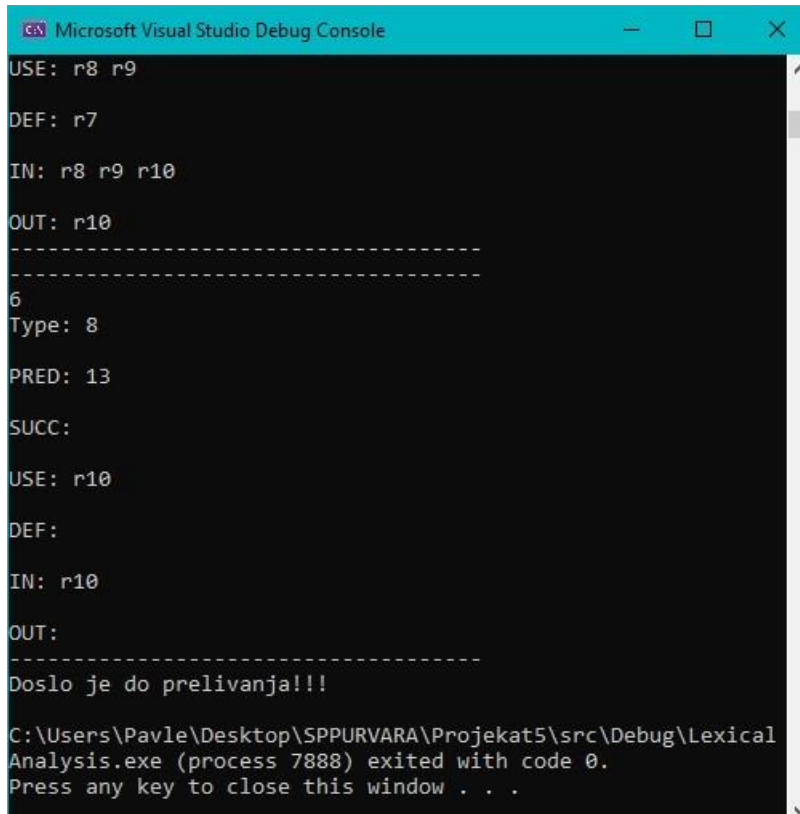
Слика 3. Испис за пример 1.

2. *prelivanje.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  _reg r6;
10 _reg r7;
11 _reg r8;
12 _reg r9;
13 _reg r10;
14
15 _func main;
16     la      r4,m1;
17     lw      r1, 0(r4);
18     la      r5, m2;
19     lw      r2, 0(r5);
20     add     r3, r1, r2;
21     xor     r7,r8,r9;
22     bltz    r10,labela
```

Слика 4. Изглед улазног фајла

Идеја овог улаза је била да створим ситуацију, у којој ћу имати недовољан број регистара за све променљиве, ислед чега долази до преливања.



```
Microsoft Visual Studio Debug Console
USE: r8 r9
DEF: r7
IN: r8 r9 r10
OUT: r10
-----
6
Type: 8
PRED: 13
SUCC:
USE: r10
DEF:
IN: r10
OUT:
-----
Doslo je do prelivanja!!!

C:\Users\Pavle\Desktop\SPPURVARA\Projekat5\src\Debug\Lexical
Analysis.exe (process 7888) exited with code 0.
Press any key to close this window . . .
```

Слика 5. Испис за пример 2.

3.primer.mavn

```
_mem m1 6;
_mem m2 5;

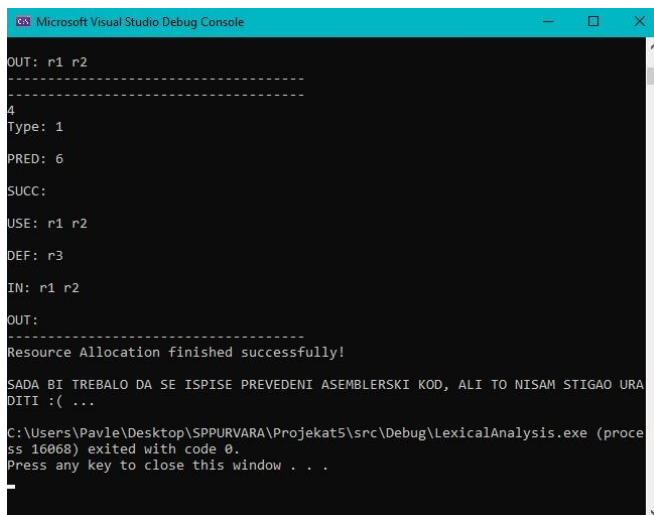
_reg r1;
_reg r2;
_reg r3;
_reg r4;
_reg r5;

_func main;
    and    r1,r2,r3;
    b      labela;
    add    r1,r2,r3;
    lw     r1, 0(r4);
    abs    r4,r5;
    sw     r1, 0(r4);
    xor     r5,r4,r3;
    bltz   r4,labela;
    or     r2,r3,r4;
    la     r5, m2;
```

Слика 6. Изглед улазног фајла

Идеја овог примера је да учитам све новоимплементиране инструкције, што ми је и успело, али због дужине исписа, изостављам ту слику.

4. simple.mavn



```
Microsoft Visual Studio Debug Console

OUT: r1 r2
-----
4
Type: 1
PRED: 6
SUCC:
USE: r1 r2
DEF: r3
IN: r1 r2
OUT:
-----
Resource Allocation finished successfully!

SADA BI TREBALO DA SE ISPISE PREVEDENI ASEMBLERSKI KOD, ALI TO NISAM STIGAO URA
DITI :( ...

C:\Users\Pavle\Desktop\SPPURVARA\Projekat5\src\Debug\LexicalAnalysis.exe (proce
ss 16068) exited with code 0.
Press any key to close this window . . .
```

Слика 7. Испис за пример 4.

То је улазни фајл дат у тексту пројекта, те због тога слику истог изостављам. Излаз је представљен на слици 7.