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: **Лука Матић**
Број индекса: **РА16/2018**

Предмет: **Системска програмска подршка у реалном времену I**
Тема рада: **MAVH преводилац**

Ментор рада: **др Миодраг Ђукић**

Нови Сад, јун, 2020.

САДРЖАЈ

1. Увод

1.1 МАВН - Преводилац

За почетак, осврнућемо се на то шта је уопште МАВН преводилац и како он и МАВН језик функционишу. 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. Битно је напоменути и правило да се у МАВН језику не смеју користити прави ресурси већ искључиво променљиве.

1.1.1 Инструкције МАВН језика

МАВН језик подржава 13 MIPS инструкција, а то су:

add – (addition) сабирање

addi – (addition immediate) сабирање са константом

b – (unconditional branch) безусловни скок на лабелу унутар тренутне функције

bltz – (branch on less than zero) скок на лабелу унутар тренутне функције ако је регистар мањи од нуле

la – (load address) учитавање адресе у регистар

li – (load immediate) учитавање константе у регистар

lw – (load word) учитавање једне меморијске речи

nop – (no operation) инструкција без операције

sub – (subtraction) одузимање

sw – (store word) упис једне меморијске речи

jal – (jump and link) безусловни скок на функцију

and – (and) логичко „и“

andi – (and immediate) логичко „и“ са константом.

1.1.2 Синтакса МАВН језика

Синтакса МАВН језика описана је граматиком:

Q → S ; L

S → _mem mid num

S → _reg rid

S → _func id E

S → id: E

S → E

L → eof

L → Q

E → add rid, rid, rid

E → addi rid, rid, num

E → sub rid, rid, rid

E → la rid, mid

E → lw rid, num(rid)

E → li rid, num

E → sw rid, num(rid)

E → b id

E → bltz rid, id

E → nop

E → jal id

E → and rid, rid, rid

E → andi rid, rid, num

Терминални симболи МАВН језика су:

: ; , ( )

_mem, _reg, _func, num id, rid, mid, eof, add, addi, sub, la, lw, li, sw, b, bltz, nop, jal, and, andi

У наставку се напомиње на који начин се врши декларисање функција, меморијских и регистарских променљивих:

• Декларација функције: _func funcName

funcName – мора почети словом, у наставку може бити било који низ слова и бројева.

Пример:

```
_func foo
```

• Декларација меморијске промељиве: _mem varName value

varName – мора почети малим словом m у наставку може бити било који број.

Пример:

```
_mem m4 55
_mem m78 12
```

• Декларација регистарске променљиве: _reg varName

varName – мора почети малим словом r у наставку може бити било који број.

Пример:

```
_reg r1
_reg r9
```

1.2 Задатак

Задатак је да имплементирамо MAVH преводиоц који ће да учита улазну датотеку писану на MAVH језику и преведе њен код на MIPS 32bit асемблер. Ограничићемо се на само једну улазну датотеку која ће имати екстензију „.mavn“.

Приликом превођења, потребно је да преводилац:

1. Додели ресурсе за регистарске променљиве – ограничићемо се на 4 регистра: t0, t1, t2 и t3 из MIPS архитектуре
2. Генерише све меморијске променљиве у секцију за податке - .data водећи рачуна о синтакси асемблерског језика
3. Све инструкције смести у програмску секцију - .text
4. Име функције генерише као глобални симбол - .globl и као лабелу на прву њену инструкцију
5. Генерише додатне инструкције за исправан рад излазног кода у случају коришћења инструкције jal у улазном коду
6. Генерише излазну датотеку са екстензијом “.s” која садржи преведен и коректан MIPS 32bit асемблерски језик полазног програма.

2. Анализа проблема

С обзиром на то да је потребно превести код са једног језика на други, мораћемо да прођемо кроз све фазе превођења, осим фазе избора инструкција (јер ће се вршити превођење инструкције на инструкцију, па нам она није потреба). То значи да морамо прво да обавимо лексичку, потом синтаксну анализу, као и анализу животног века промењљивих и на крају фазу доделе ресурса. За поседње две фазе ће нам бити потребно да направимо одређене објекте инструкција и промењљивих у које ћемо сместити податке које смо добили из улазне датотеке и онда ћемо, на основу тих објеката, моћи да прођемо кроз наведене фазе, формирамо излазни код и испишемо га у излазну датотеку.

3. Концепт решења

Прво ћемо помоћу модула FiniteStateMachine и LexicalAnalysis обавити учитавање улазне датотеке и лексичку анализу над улазном датотеком, а потом кроз модул SyntaxAnalyisis и синтаксну. Уколико програм наиђе на синтаксну грешку, пријавиће је и обуставиће даље процесирање.

Након тога, издвојићемо листе лабела и функција помоћу модула LabelsAndFunctions, а потом и листе меморијских и регистарских променљивих помоћу модула IR. Листу инструкција ћемо генерисати и повезати са променљивама које се односе на њих, као и са лабелама у којима се те инструкције налазе користећи модуле IR и InstructionGenerator. Уколико програм наиђе на семантичку грешку, пријавиће је и обуставиће даље процесирање. InstructionGenerator модул ће такође и формирати листе претходника и наследника за сваку инструкцију.

Када имамо формирану листу инструкција која садржи све неопходне податке, можемо је искористити да одрадимо анализу животног века променљивих помоћу модула LivenessAnalysis. Излаз из тог дела програма ће бити исправљена листа инструкција са попуњеним листама променљивих које су живе на улазу и променљивих које су живе на излазу сваке инструкције.

То ће нам омогућити да извршимо доделу ресурса унутар модула ResourceAllocation. Што се тиче фазе доделе ресурса, прво ћемо конструисати граф сметњи на основу прерађених инструкција. Потом ћемо одрадити фазу упрошћавања и на стек ћемо наређати променљиве на основу ранга, односно броја пресека (сметњи) које свака променљива има са другим променљивама унутар графа сметњи. Стек ћемо искористити како би одрадили коначну доделу ресурса и заменили регистарске променљиве са правим

регистрима. Уколико дође до преливања, пријавићемо грешку и нећемо обрађивати преливање, већ ћемо обуставити читав процес.

У случају успешне доделе ресурса, на основу листи меморијских променљивих, листи фукнција, листи лабела које се налазе унутар сваке функције и листи инструкција које се налазе унутар сваке лабеле, генерисаћемо излазни код. При томе ћемо водити рачуна о томе да на појединим местима морамо изгенерисати додатне инструкције које се односе на регулисање понашања инструкције jal.

Новоформирани код ћемо исписати у излазну датотеку чији ћемо назив одредити на основу имена улазне дадотеке. Излазна датотека ће се звати исто као и улазна, са разликом у томе што јој екстензија неће бити „.mavn“ него „.s“. Сачуваћемо излазну датотеку у истом фолдеру у ком се налази и улазна датотека и тиме ћемо завршити програм.

4. Опис решења

4.1 Модул главног програма (main)

```
int main();
```

Функција „main“ јесте главна функција програма. Приказује основне информације о програму и увезује све блокове. Такође, врши и генерисање излазне датотеке након извршених процеса свих блокова.

4.2 Модул LexicalAnalysis

4.2.1 Метода readInputFile

```
bool LexicalAnalysis::readInputFile(string fileName);
```

Учитава улазну датотеку са прослеђеним именом у бафер и враћа true уколико у томе успе, а false ако не.

4.2.2 Метода Do

```
bool LexicalAnalysis::Do();
```

Врши лексичку анализу над кодом учитаним у бафер и враћа true уколико је лексичка анализа успешно извршена, а false ако није..

4.2.3 Метода getNextTokenLex

```
Token LexicalAnalysis::getNextTokenLex();
```

Враћа следећи токен који добавља на основу тренутног уз помоћ модула FiniteStateMachine.

4.3 Модул FiniteStateMachine

4.3.1 Метода getNextState

```
int FiniteStateMachine::getNextState(int currentState, char transitionLetter);
```

Добавља следеће стање из матрице стања на основу тренутног стања и прелазног карактера.

4.4 Модул LabelsAndFunctions

4.4.1 Функција generateLabelsAndFunctions

```
void generateLabelsAndFunctions(Labels& labels, Functions& functions,
                                LexicalAnalysis& lex)
```

Генерише прослеђене листе лабела и функција на основу прослеђене лексичке анализе.

4.4.2 Функција functionExists

```
bool functionExists(Functions& functions, string funcName);
```

Враћа true уколико се функција са прослеђеним именом налази у прослеђеној листи функција, а false уколико не.

4.4.3 Функција labelExistsInFunction

```
bool labelExistsInFunction(Function* func, string labelName);
```

Враћа true уколико се лабела са прослеђеним именом налази у листи лабела прослеђене функције, а false уколико не.

4.5 Модул IR

4.5.1 Функција variableExists

```
bool variableExists(string variableName, Variables variables);
```

Враћа true уколико се промнљива са прослеђеним именом налази у прослеђеној листи променљивих, а false уколико не.

4.5.2 Функција getVariablePosition

```
int getVariablePosition(string name, Variables variables);
```

Враћа int који представља позицију променљиве уколико се променљива са прослеђеним именом налази у прослеђеној листи променљивих, а пријављује грешку уколико не.

4.5.3 Функција getVariableType

```
Variable::VariableType getVariableType(string name, Variables variables);
```

Враћа `Variable::VariableType` који представља тип променљиве уколико се променљива са прослеђеним именом налази у прослеђеној листи променљивих, а пријављује грешку уколико не.

4.5.4 Функција generateVariables

```
void generateVariables(Variables& memVariables, Variables& regVariables,
                       LexicalAnalysis& lex);
```

Генерише прослеђене листе меморијских и регистарских променљивих на основу прослеђене лексичке анализе.

4.6 Модул InstructionGenerator

4.6.1 Функција generateInstructions

```
void generateInstructions(Instructions& instructions, LexicalAnalysis& lex,
                          Functions& functions, Labels& labels, Variables& regVariables,
                          Variables& memVariables);
```

Генерише прослеђену листу инструкција на основу прослеђене лексичке анализе и прослеђених листа фукнција, лабела, регистарских и меморијских променљивих.

4.6.2 Функције за уређивање инструкција

```
void generateUse(Variables& use, InstructionType type, Variable* src1, Variable* src2);
void generateDef(Variables& use, InstructionType type, Variable* dst1);
```

Ове функције служе за попуњавање скупова (листи) променљивих које одређена инструкција користи или дефинише.

```
void generatePred(Instructions& instructions);
void generateSucc(Instructions& instructions, Labels& labels);
```

Ове функције служе за попуњавање скупова (листи) претходника и наследника.

4.7 Модул LivenessAnalysis

4.7.1 Функција livenessAnalysis

```
void livenessAnalysis(Instructions& instructions);
```

Врши анализу животног века над прослеђеном листом инструкција. Користи се функцијом `bool checkEnd(std::vector<bool>& done);` за проверу краја анализе животног века.

4.8 Модул ResourceAllocation

4.8.1 Метода build

```
void InterferenceGraph::build(Instructions instructions);
```

Гради граф интерференције на основу прослеђених инструкција.

4.8.2 Метода simplify

```
void InterferenceGraph::simplify(stack<Variable*>& simplificationStack);
```

Врши фазу упрошћавања графа и поставља променљиве на прослеђени стек уколико је могуће обојити граф 4 боје (због 4 регистра која имамо на располагању), а у супротном пријављује грешку и обуставља даљи процес.

4.8.3 Функција doResourceAllocation

```
void doResourceAllocation(stack<Variable*>* simplificationStack, InterferenceGraph*
                          interferenceGraph);
```

Користећи прослеђени стек и граф сметњи, врши коначну доделу ресурса уколико је то могуће, а у супротном пријављује грешку и обуставља даљи процес.

5. Тестирање

Тестирање је спровођено тако што је на основу улазног кода, програм генерисао излазни, ако је то могуће, или пријављивао грешку уколико није. Уколико се код успешно превео, пократање је вршено уз помоћ „QtSpim” алата.

Посматраћемо 4 тестна случаја.

5.1 Тестни случај 1: simple.mavn

Посматрамо понашање програма када му се проследи једноставан „.mavn“ код који изгледа овако:

```
_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;
```

Излазни код је:

```
.globl main

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

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

Излазни код је тестиран помоћу „QtSpim“ алата и програм је извршен без проблема.

5.2 Тестни случај 2: multiply.mavn

Посматрамо понашање програма када му се проследи „.mavn“ код који не може да се преведе на асемблерски код архитектуре са 4 регистра без преливања. Он изгледа овако:

```
_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;
```

Излазни код није формиран јер је програм пријавио да у овом случају долази до преливања, а пошто тај случај не покривамо, ту се процес и завршио.

5.3 Тестни случај 3: allIntructions.mavn

Посматрамо понашање програма када му се проследи „.mavn“ код који садржи све MIPS инструкције које MABH подржава, а који изгледа овако:

```
_mem m1 17;
_mem m2 4;
_mem m3 1;
_reg r1;
_reg r2;
_reg r3;
_reg r4;
_func main;
 la r4, m3;
 li r1, 0;
 sw r1, 0(r4);
 li r1, 0;
 la r2, m2;
 lw r3, 0(r2);
 add r1, r1, r3;
 addi r1, r1, 7;
 sub r1, r1, r3;
 jal f1;
 sub r1, r1, r3;
 and r1, r1, r3;
_func f1;
 and r1, r1, r3;
 andi r1, r1, 0;
 b lab;
 la r2, m2;
lab:
 la r4, m3;
 li r1, 7;
 sw r1, 0(r4);
 sub r1, r1, r3;
 nop;
 bltz r1, f1;
```

Излазни код је:

```
.globl main
.globl f1

.data
m1:           .word 17
m2:           .word 4
m3:           .word 1

.text
main:
       la             $t0, m3
       li             $t2, 0
       sw             $t2, 0($t0)
       li             $t2, 0
       la             $t0, m2
       lw             $t1, 0($t0)
       add            $t2, $t2, $t1
       addi           $t2, $t2, 7
       sub            $t2, $t2, $t1
       addi           $sp, $sp, -4
       sw             $ra, ($sp)
       jal            f1
       lw             $ra, ($sp)
       addiu          $sp, $sp, 4
       sub            $t2, $t2, $t1
       and            $t2, $t2, $t1
       jr             $ra
f1:
       and            $t2, $t2, $t1
       andi           $t2, $t2, 0
       b              lab
       la             $t0, m2
lab:
       la             $t0, m3
       li             $t2, 7
       sw             $t2, 0($t0)
       sub            $t2, $t2, $t1
       nop
       bltz           $t2, f1
       jr             $ra
```

Излазни код је тестиран помоћу „QtSpim“ алата и програм се извршава без проблема.

5.4 Тестни случај 4: multiJal.mavn

Посматрамо понашање програма када му се проследи „.mavn“ код који има дубину позива функција већу од 1 (унутар прве се позива друга, унутар друге трећа):

```
_mem m1 3;
_mem m2 4;
_reg r1;
_reg r2;
_func main;
  la r1, m1;
  lw r2, 0(r1);
  addi r1, r2, 4;
```

```
      jal f2;
      add r1, r2, r2;
    _func f1;
      la r1, m1;
      lw r2, 4(r1);
      addi r1, r2, 4;
    _func f2;
      la r1, m1;
      lw r2, 0(r1);
      addi r1, r2, 4;
      jal f1;
      addi r1, r2, 7;
```

Излазни код је:

```
.globl main
.globl f1
.globl f2

.data
m1:           .word 3
m2:           .word 4

.text
main:
      la            $t0, m1
      lw            $t0, 0($t0)
      addi          $t0, $t0, 4
      addi          $sp, $sp, -4
      sw            $ra, ($sp)
      jal           f2
      lw            $ra, ($sp)
      addiu         $sp, $sp, 4
      add           $t0, $t0, $t0
      jr            $ra
f1:
      la            $t0, m1
      lw            $t0, 4($t0)
      addi          $t0, $t0, 4
      jr            $ra
f2:
      la            $t0, m1
      lw            $t0, 0($t0)
      addi          $t0, $t0, 4
      addi          $sp, $sp, -4
      sw            $ra, ($sp)
      jal           f1
      lw            $ra, ($sp)
      addiu         $sp, $sp, 4
      addi          $t0, $t0, 7
      jr            $ra
```

Излазни код је тестиран помоћу „QtSpim“ алата и програм се извршава без проблема.

5.5 Додатни тестни случајеви

Поред наведених кодова, тестирани су још неки како би се проверила пријава одређених синтаксних и семантичких грешака. Неке од њих су: изостављање знака „;“ на крају линије, двострука декларација променљивих, функција и лабела, скок на лабелу која не припада тренутној функцији помоћу инструкција „b“ и „bltz“, скок на лабелу која није функција помоћу инструкције „jal“. Сви наведени случајеви пријављују грешку и обустављају даљи процес превођења.

6. Закључак

Прошли смо кроз један концепт за решење датог проблема. Реализовали смо потребне фазе превођења и извршили одређене семантичке провере. Верификација рада програма одрађена је генерисањем излазних датотека које су извршаване уз помоћ „QtSpim“ алата, уколико их је било могуће изгенерисати, а уколико није, програм је пријављивао грешку и обустављао даљи процес.