UNIVERZITET U NOVOM SADU
FAKULTET TEHNIČKIH NAUKA
NOVI SAD

Departman za računarstvo i automatiku
Odsek za računarsku tehniku i računarske komunikacije

---

# PROJEKTNI ZADATAK

---

Kandidat: Nenad Gvozdenac
Broj indeksa: RA 133/2021
Predmet: Osnove paralelnog programiranja i softverski alati
Mentor rada: Kenjić Dušan

Novi Sad, jun, 2023

# Sadržaj

# 1. Uvod

## 1.1 Opis problema

Potrebno je realizovati MAVN prevodilac koji prevodi programe sa višeg asemblerskog jezika na osnovni MIPS 32bit asemblerski jezik. Prevodilac treba da podržava detekciju leksičkih, sintaksnih i semantičkih grešaka kao i generisanje odgovarajućih izveštaja o eventualnim greškama. Izlaz iz prevodioca treba da sadrži korektan asemblerski kod koji je moguće izvršavati na MIPS 32bit arhitekturi, odnosno simulatoru.

MAVN (MIPS Asembler Visokog Nivoa) je alat kojeg izdvaja činjenica da uvodi registarske promenljive, koje omogu´cavaju korišćenje promenljivih umesto registara, odnosno pravih resursa.

# 2. Analiza problema

Proces prevodenja iz višeg u osnovni asemblerski jezik nije komplikovan ali jeste kompleksan. Da bi kod bio funkcionalan, neophodno je uraditi sledeće korake:

1. Leksička analiza
2. Sintaksna analiza
3. Kreiranje instrukcija
4. Analiza životnog veka
5. Kreiranje grafa smetnja
6. Kreiranje simplificiranog steka registara
7. Simplifikacija registara
8. Alokacija resursa
9. Ispisivanje u .S fajl

Jezik podržava 14 instrukcija: add, addi, b, bltz, la, li, lw, nop, sub, sw. Sa bilo kojom pojavom greške, program se prekida i greška se ispisuje. Ako nema grešaka, odnosno program se uspešno izvrši, generiše se novi ”.s” fajl.

Imali smo zadatak da dodamo tri instrukcije, dve aritmetičko-logičke i jednu. Ja sam dodao addu, or i lb.

# 3. Realizovano rešenje

Rešenje se može podeliti na više koraka. Krajnje rešenje je .S fajl sa svim instrukcijama prevedenim u asemblerski jezik.

1. Prvi korak predstavlja leksičku analizu, koja generiše listu tokena. Leksička analiza predstavlja automat koji sve tokene vraća u neki STL kontejner i definiše lekseme jezika.
2. Sintaksna analiza, koja na osnovu leksičke analize implementira gramatiku jezika i formira liste instrukcija, labela i promenljivih. Promenljive su realizovane kao REGISTRI i MEMORIJSKE promenljive.
3. Kreiranje instrukcija se svodi na to da tokom sintaksne analize, on pronalazi sve labele i promenljive, te ih sve stavlja u različite STL kontejnere. Nakon što pronađe instrukcije, sve STL kontejnere proverava i ubacuju u instrukciju, tako da objekat instrukcije ima sve podatke potrebne.
4. Analiza životnog veka se realizuje kao jedna funkcija. Sve varijable koje su registri se proveravaju da li će „živeti“ do neke naredne instrukcije, ili će se osloboditi radi preuzimanja neke druge varijable, tog registra.
5. Kreiranje grafa prolazi kroz sve promenljive koje su prošle analizu životnog veka, pa na osnovu njih proverava da li se one preklapaju ili ne.
6. Kreiranje simplifikacionog steka registara predstavlja sve registre koje su stavljene u STL kontejner varijable. On je na početku prazan, te se nakon simplifikacije (koraka 7) on popunjava.
7. Tokom simplifikacije, popunjava se simplifikacioni stek, tako što se prolazi kroz graf smetnji i proverava da li su varijable u interferenciji. Proveravaju se susedi samih varijabli, i ukoliko su oni manji od broja registara, znači da se on može zameniti sa nekim drugim registrom. Stavlja se u stek.
8. Alokacija resursa se radi kao bojanje grafa, tako što se prolazi kroz simplificirani stek, i na osnovu njega proveravamo koje registre možemo da zamenimo kojim.
9. Ispisivanje fajla se vrši samo ukoliko su svi koraci bili uspešni. Ukoliko se to ne desi, puca program i greška je ispisana na konzolni ekran.

# 4. Programski koncept rešenja

Rešenje smo izveli u jeziku C++. Programsko rešenje se izvelo po prethodno navedenim koracima. Prvo smo otvorili ulazni fajl i proverili da li je moguće otvoriti fajl. Ukoliko jeste, nastavljamo dalje na leksičku analizu.

## 4.1 Klasa SyntaxAnalysis

Konstruktor klase SyntaxAnalysis se sastoji od referenci na objekat klase *LexicalAnalysis*, i objekte klasa *Variables, Instructions, Labels.* U početku, svi ti elementi su prazni STL kontejneri.

Bitno je napomenuti metode koje se realizuju po šemi algoritma: *Q(), S(), E(), L()*. One implementiraju gramatiku jezika te proveravaju da li je sintaksično dobro postavljen ceo ulazni kod.

Klasa sadrži sve reference na ove elemente, te kada uradimo sintaksnu analizu, svi ti elemeni se popunjavaju. Bitno je napomenuti da ponavljanje varijabli ili labela nije dozvoljeno po asemblerskom jeziku, te se to smatra kao sintaksna greška.

Metoda *S()* se poziva kada se dodaje memorijska promenljiva, registarska promenljiva, labela i ulazna funkcija.

Ceo ovaj proces se obavlja pozivom *doSyntaxAnalysis()* metode objekta klase sintaksne analize.

U slučaju *E()* metode, kreiraju se instrukcije, kao i čitaju izvorni i destinacioni registri koji se upisuju u sam kontejner koji čiva sve registre. Postoje instrukcije koje čitaju brojeve, te se one čuvaju kao posebno polje koje se čuva u vrednosti *m_number*. Takođe, neke instrukcije mogu da dodaju labele, te je pre dodavanja proveriti da li ta labela uopšte postoji; te je onda dodati.

Ovaj gornji proces se radi preko *findPredAndSucc()* metode objekta klase sintaksne analize.

Nakon generisanja svih ovih instrukcija, varijabli i labela, mora se proveriti postojanje njenih parametara. Međukorak nakon sintaksne analize, a pre analize životnog toka, bitno je proveriti da li postoje prethodnici i sukcesori svake instrukcije.

Specijalne instrukcije koje su *I_B* i *I_BLTZ*, mogu da imaju više sukcesora i prethodnika, te je njih proveravati bitno posebno.

Svaka instrukcija ima sukcesora, narednu instrukciju. Svaka naredna instrukcija ima prethodnika, prethodnu instrukciju.

Svaka instrukcija ima *m_def, m_use, m_in, m_out* STL kontejnere varijabli, nazvanih *Variables*.

*def* i *use* se popunjavaju tokom poziva konstruktora instrukcije.

## 4.2 LivenessAnalysis

LivenessAnalysis nema svoju klasu, te je to samo jedna funkcija, *doLivenessAnalysis()*.

Napomenuti je bitno da se prilikom poziva ovoga, nalaze *m_in* i *m_out*, na osnovu atributa klase *m_def* i *m_use.* Ovi atributi, kao prethodno napomenuto, su ustvari lista registarskih varijabli.

## 4.3 Klasa InterferenceGraph

Pre nego što ćemo napraviti graf smetnji, potrebno je da se analiza životnog veka uspešno završi.

Za svaku varijablu mi proveravamo da li je to registarska varijabla. Ako je registarska varijabla, onda postavljamo njenu poziciju na poziciju koja se uvek uvećava od nule, te stavljamo to u STL kontejner.

Atributi klase su *InterferenceMatrix*, koji je ustvari vektor vektora instrukcija. Ovo treba da bude kvadratna matrica.

Nakon što se pronađu varijable, bitno je napraviti interferencni graf. Ovo se realizuje metodom *makeInterferenceGraph()*. Kroz svaku instrukciju se prolazi, i kroz svaki *m_def* i *m_out* se prolazi, konsekutivno. Nakon toga, proverava se da li je graf u interferenciji, tako što se prvo proveri da li su pozicije različite. Ukoliko jesu, postavljaju se oba elementa u interferenciju.

Bitno je napomenuti da je graf, po dijagonali, simetričan. Ukoliko je registar1 u interferenciji sa registrom2, onda je i registar2 u interferenciji sa registrom1.

## 4.4 SimplificationStack

Prvo se kreira simplifikacioni stek kao prazna promenljiva, te se nakon izrade funkcije *doSimplification()*.

Funkcija *doSimplification()* proverava da li su broj registara u interferenciji veći od broja suseda samog registra, kao i da li je taj broj registara manji od dozvoljenog broja registara koji smo prosledili. Ukoliko jeste, stavlja poziciju tog registra kao najveću poziciju.

Nakon toga, prolazi kroz sve varijable i proverava da li je pozicija te varijable ista kao i najveća pozicija. Ukoliko jeste, stavlja je na stek.

Nakon ovoga, stavlja tu poziciju kao „zauzetu“ i ne koristi je dalje.

## 4.5 Alokacija resursa

Traži se bojanje grafa, te se prolazi kroz ceo simplifikacioni stek i proverava se da li je u interferenciji element koji se u interferencnom grafu nalazi sa flagom 1. Ukoliko da, ne možemo mu dati prvi registar, te mu prosleđujemo jedan od narednih registara. U našem zadatku je rečeno da broj registara mora biti 4, ali, sa dogovorom sa mentorom, sam taj broj zamenio na pet. Razlog zamene broja registara, je što u ulaznom fajlu *multiply.mavn*, dolazi do spillovanja kada imamo 4 registra.

## 4.6 Normalizacija promenljivih

Nakon alokacije resursa, došlo je do problema jer su mi varijable koje su se nalazile u instrukcijama, bile drugačije od varijabli koje sam izmenio. Nakon utvrđivanja problema, solucija je bila da se mapiraju nove varijable, na varijable u instrukcijama. Ovo se radilo primenom funkcije *normalizeAssignmentsToVariables()*.

## 4.7 Klasa FileWriter

Ova klasa nije bila neophodna, moglo se uraditi i preko jedne metode. Međutim, zbog čitkosti, ovo sam realizovan preko klase.

Prilikom metode *write()*, otvara se izlazni stream u fajl. Ukoliko se ne uspe otvoriti izlazni stream, izbacuje se greška.

Prvo se ispisuje *.globl* i entry point, što je prva labela koja je pronađena.

Nakon toga, ispisuju se sve memorijske varijable kao *.word* varijable.

Nakon svega ovoga, ispisuju se labele i korespodentne instrukcije. Registri se zamenjuju sa realnim registrima, a memorijske promenljive sa realnim memorijskim promenljivama.

# 5. Rad na primerima

## 5.1 Primer #1

```
_mem m1 6;
_mem m2 5;
_mem m3 3;

_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;
    addu        r3, r2, r1;
    or      r3, r2, r1;
    lb      r5, 4(r4);

lab:
    la      r4, m3;
    or      r4, r4, r4;
    lb      r3, 8(r4);
    bltz        r2, lab
```

*Lista 1: Ulazni fajl simple.mavn sa dodate tri instrukcije*

---

```
.globl main

.data
m1: .word 6
m2: .word 5
m3: .word 3

.text
main:
    la $t1, m1
    lw $t3, 0($t1)
    la $t0, m2
    lw $t2, 0($t0)
    add $t0, $t3, $t2
    addu $t0, $t2, $t3
    or $t0, $t2, $t3
    lb $t0, 4($t1)
lab:
    la $t1, m3
    or $t1, $t1, $t1
    lb $t0, 8($t1)
    bltz $t2, lab
```

*Lista 2: Izlazni fajl simple.S sa dodate tri instrukcije*

## 5.2 Primer #2

```
_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;
```

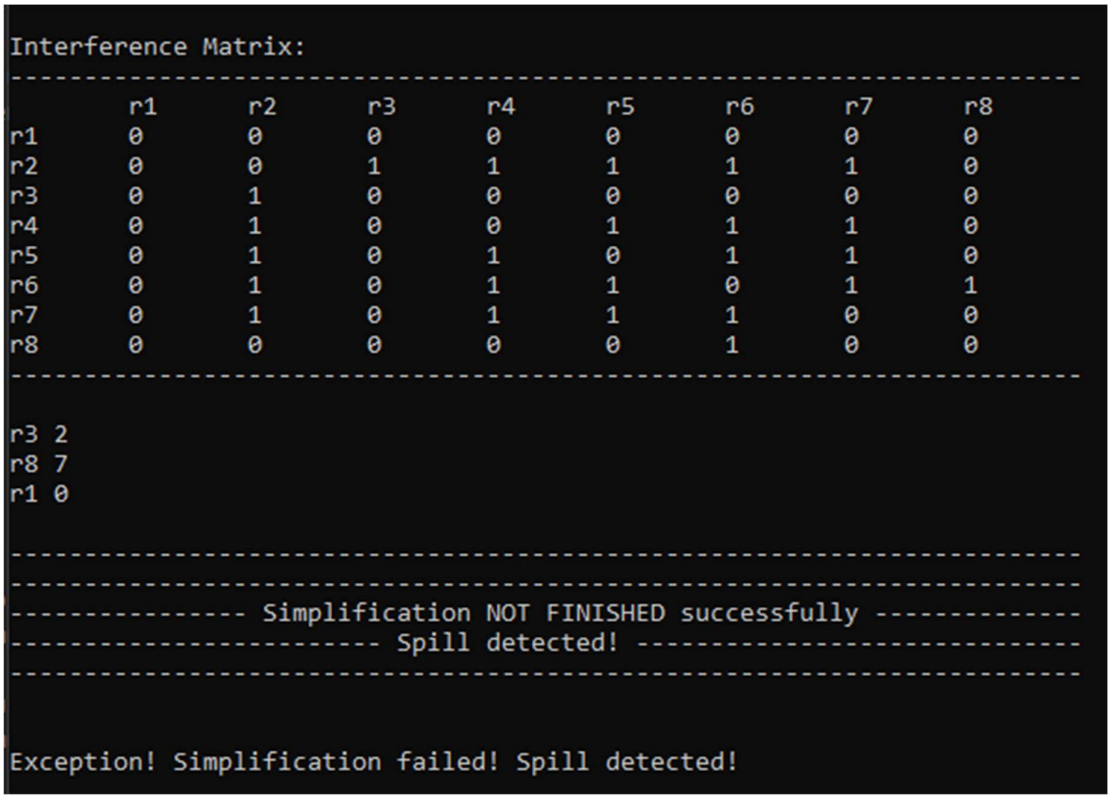
*Lista 3: Ulazni fajl multiply.mavn*

---

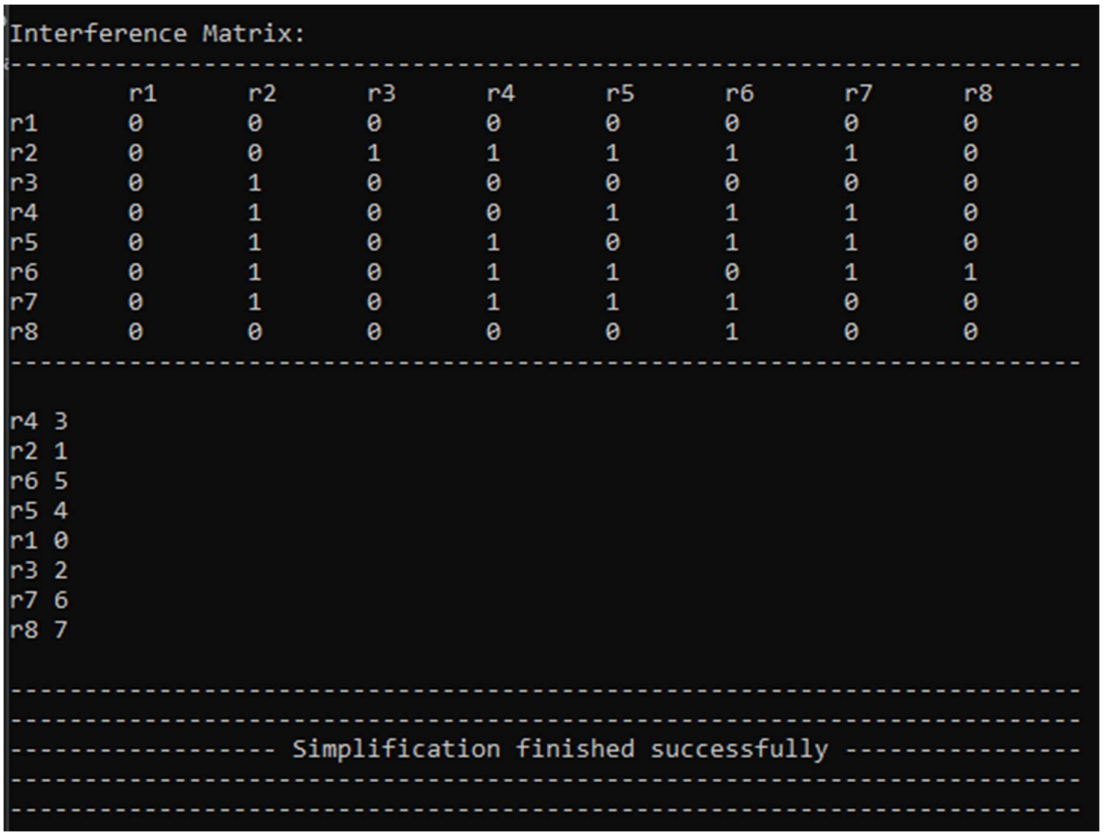
```
.globl main

.data
m1: .word 6
m2: .word 5
m3: .word 0

.text
main:
    la $t0, m1
    lw $t3, 0($t0)
    la $t0, m2
    lw $t4, 0($t0)
    li $t1, 1
    li $t2, 0
lab:
    add $t2, $t2, $t3
    sub $t0, $t1, $t4
    addi $t1, $t1, 1
    bltz $t0, lab
    la $t0, m3
    sw $t2, 0($t0)
    nop
```

*Lista 4: Izlazni fajl multiply.S*

## 5.3 Provera spillovanja




*Slika 1: Spillovanje ukoliko se koriste 4 registra*




*Slika 2: Spillovanje ukoliko se koriste 5 registra*

# 6. Zaključak

Zašto se koristi smanjivanje broja registara? MAVN jezik može da koristi beskonačan broj registara, a nama treba samo određeni broj registara u ASM jeziku.

Sam kompajler se koristi u dve velike faze:

1. Prevod koji se koristi u leksičkoj i sintaksnoj analizi
2. Smanjivanje broja registara na 4, što se radi u analizi životnog veka, grafu interferencije, simplifikaciji kao i bojanju grafa.

Pokazali smo da se neki zadaci poput *simple.mavn*, mogu svesti na korišćenje samo 2 registra. Dok se neki zadaci koji koriste osam registara u *multiply.mavn*, može svesti na samo 5 registara, ali ne može ispod 5.