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

ЗАВРШНИ (BACHELOR) РАД

Кандидат: **Томислав Ковач**
Број индекса: **Е12717**

Тема рада: **Додавање нових типова података у компајлерску структуру за процесоре са ограниченим ресурсима**

Ментор рада: **Проф. др Мирослав Поповић**

Нови Сад, јул, 2012.

УНИВЕРЗИТЕТ У НОВОМ САДУ ● **ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА**
21000 НОВИ САД, Трг Доситеја Обрадовића 6

КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА

Редни број, **РБР**:		
Идентификациони број, **ИБР**:		
Тип документације, **ТД**:		Монографска документација
Тип записа, **ТЗ**:		Текстуални штампани материјал
Врста рада, **ВР**:		Завршни (Bachelor) рад
Аутор, **АУ**:		**Томислав Ковач**
Ментор, **МН**:		**Проф. др Мирослав Поповић**
Наслов рада, **НР**:		**ДОДАВАЊЕ НОВИХ ТИПОВА ПОДАТАКА У КОМПАЈЛЕРСКУ СТРУКТУРУ ЗА ПРОЦЕСОРЕ СА ОГРАНИЧЕНИМ РЕСУРСИМА**
Језик публикације, **ЈП**:		Српски / латиница
Језик извода, **ЈИ**:		Српски
Земља публиковања, **ЗП**:		Република Србија
Уже географско подручје, **УГП**:		Војводина
Година, **ГО**:		**2012**
Издавач, **ИЗ**:		Ауторски репринт
Место и адреса, **МА**:		Нови Сад; трг Доситеја Обрадовића 6
Физички опис рада, **ФО**: (поглавља/страна/ цитата/табела/слика/графика/прилога)		**7/43/0/3/0/0/0**
Научна област, **НО**:		Електротехника и рачунарство
Научна дисциплина, **НД**:		Рачунарска техника
Предметна одредница/Кqучне речи, **ПО**:		**Компајлер**
УДК		
Чува се, **ЧУ**:		У библиотеци Факултета техничких наука, Нови Сад
Важна напомена, **ВН**:		
Извод, **ИЗ**:		У овом раду приказано је решење проширења постојеће компајлерске структуре подршком за нови тип података, шездесетчетворобитни целобројни тип лонг лонг.
Датум прихватања теме, **ДП**:		
Датум одбране, **ДО**:		
Чланови комисије, **КО**:	Председник:	др Иштван Пап, доц.
	Члан:	др Јелена Ковачевић, доц.
	Члан, ментор:	др Мирослав Поповић, ред. проф.

Потпис ментора

UNIVERSITY OF NOVI SAD ● **FACULTY OF TECHNICAL SCIENCES**
21000 NOVI SAD, Trg Dositeja Obradovića 6

KEY WORDS DOCUMENTATION

Accession number, **ANO**:			
Identification number, **INO**:			
Document type, **DT**:		Monographic publication	
Type of record, **TR**:		Textual printed material	
Contents code, **CC**:		Bachelor Thesis	
Author, **AU**:		**Tomislav Kovač**	
Mentor, **MN**:		**PhD Miroslav Popović**	
Title, **TI**:		**Adding new data types in compiler structure with limited resources**	
Language of text, **LT**:		Serbian	
Language of abstract, **LA**:		Serbian	
Country of publication, **CP**:		Republic of Serbia	
Locality of publication, **LP**:		Vojvodina	
Publication year, **PY**:		2012	
Publisher, **PB**:		Author's reprint	
Publication place, **PP**:		Novi Sad, Dositeja Obradovica sq. 6	
Physical description, **PD**: (chapters/pages/ref./tables/pictures/graphs/appendixes)		**7/43/0/3/0/0/0**	
Scientific field, **SF**:		Electrical Engineering	
Scientific discipline, **SD**:		Computer Engineering, Engineering of Computer Based Systems	
Subject/Key words, **S/KW**:		**Compiler**	
UC			
Holding data, **HD**:		The Library of Faculty of Technical Sciences, Novi Sad, Serbia	
Note, **N**:			
Abstract, **AB**:		**Adding new data types in compiler structure with limited resources is described in this paper.**	
Accepted by the Scientific Board on, **ASB**:			
Defended on, **DE**:			
Defended Board, **DB**:	President:	Dr. Ištvan Pap, doc	
	Member:	Dr. Jelena Kovačević, doc	Menthor's sign
	Member, Mentor:	Phd Miroslav Popović	

Zahvalnost

Svom mentoru prof. dr Miroslavu Popoviću, želeo bih odati zahvalnost na izboru interesantne teme, znanju i iskustvu koje sam stekao tokom izrade diplomskog rada.

Za nesebičnu posvećenost i pomoć prilikom realizacije rada, posebnu zahvalnost dugujem asistentu Miodragu Đukiću.

U Novom Sadu, 25.juna 2012. godine Tomislav Kovač

SADRŽAJ

SPISAK TABELA

SKRAĆENICE

IR - *Intermediate Representation*, međukod

EDG - *Edison Design Group*

GPL - *GNU General Public License*

1. Uvod

U ovom radu realizovano je rešenje proširenja postojeće kompajlerske strukture podrškom za novi tip podataka kao što je šezdesetčetvorobitni celobrojni tip long long. Izazov je bilo uvođenje novog tipa podataka na platformu sa ograničenim resursima, kao i nedostatak fizičke podrške za pomenuti tip unutar ciljnog procesora.

Za izradu diplomskog rada u programskom jeziku C korišteno je okruženje Microsoft Visual Studio 2010, a za podršku pisanja i prevođenja funkcija u asembleru CLIDE 6.5 (Cirrus Logic Integrated Development Environment).

Motivacija za odabiranje teme bila je želja za upoznavanjem koncepata kompajliranja i realne industrijske kompajlerske strukture.

Cilj rada bio je dobiti jedno skalabilno programsko rešenje, koje će se moći dalje nadograđivati. Kvaliteno rešenje koje će biti robustno i otporno na greške.

2. Teorijske osnove

2.1 Kompajler

Kompajler (eng. compiler) je računarski program koji prevodi izvorni kod koji je napisan u nekom od viših programskih jezika na aseblerski ili mašinski jezik procesora. Često se za njega koristi reč programski prevodilac. Međutim, kod kompajlera za razliku od *interpretera,* koji takođe spada u programske prevodioce, izvorni program i izvršni program su potpuno odvojeni i pri izvođenju nezavisni**.**

Najčešći razlog za prevođenje izvornog koda jeste pravljenje izvršnog programa. U slučaju kompajlera izvršni program se može izvršavati bez postojanja izvornog programa, pa se korisniku najčešće i predaje samo izvršna datoteka programa.

Proces kompajliranja se obično razlaže na dve etape:

- *analizu izvornog koda*, u kojoj se izvorni program prevodi u određenu posrednu reprezentaciju pogodnu za dalje manipulisanje i
- *sintezu objektnog koda,* u kojoj se iz posredne reprezentacije dobija objektni kod.

Kompajler se sastoji od prednjeg dela (eng. front-end) i zadnjeg dela(eng. back-end).

Prednji deo kompajlera analizira izvorni kod da bi napravio unutrašnju reprezentaciju programa koja se zove *međukod* (eng. intermediate representation). Realizovan je u nekoliko faza:

1. **Rekonstrukcija linije** - predstavlja fazu pre parsiranja u kojoj se ulazna sekvenca karaktera konvertuje u kanonsku formu spremnu za sintaksnu analizu. U ovoj fazi uklanjaju se komentari i nepotrebne praznine.

2. **Leksička analiza (skeniranje)** - vrši podelu niza znakova osnovnog jezika na kome je program napisan, u osnovne elemente jezika koji se nazivaju **tokeni**. Token može biti *identifikator*, *literal* ili *terminal*. U ovoj fazi formiraju se *tabele literala*, *identifikatora* i *uniformnih simbola*.
3. **Pretprocesiranje** - ova faza je naročito značajna kod programskih jezika, kao što je C, koji podržavaju uslovno kompajliranje i korišćenje makro-a.
4. **Sintaksna analiza (parsiranje)** - obuhvata parsiranje sekvence tokena da bi se identifikovala struktura programa. Zadatak sintaksne analize je da se utvrdi da li je program u skladu sa gramatičkim pravilima programskog jezika u kome je pisan. Ova faza pravi stablo sintaksne analize koje zamenjuje linearne sekvence tokena strukturom nastalom prema pravilima formalne gramatike koja definiše sintaksu jezika.
5. **Semantička analiza -** koristi stablo sintaksne analize i informacije iz tabele simbola da bi napravio semantičke provere koda.
6. **Faza prevođenja u međukod -** stablo sintaksne analize se u ovoj fazi prevodi u neku od međureprezentacija.

Zadnji deo kompajlera može se logički podeliti u tri faze:

1. **Analiza**: u ovoj fazi se skupljaju se programske informacije iz međukoda. Tipične analize su: *analiza grafa toka upravljanja*, *analiza zavisnosti*, *analiza životnog veka programskih objekata*, *alijas analiza* itd.
2. **Optimizacija**: međukod se transformiše u funkcionalno ekvivalentne, ali brže forme. Popularne optimizacije su *inlajn proširenje*, *izbacivanje suvišnih instrukcija*, *propagacija konstanti*, *propagacija kopiranja*, *ekstrakcija konstanti*, *transformacije petlji*, *alokacija registara* itd.
3. **Generisanje koda**: međukod se prevodi u izlazni jezik, obično u prirodni mašinski jezik sistema.

2.2 Međukod

Međukod (eng. intermediate representation) nije vezan ni za jedan određeni programski jezik, niti za određenu ciljnu platformu. Po svom obliku međukod najviše podseća na asembler nekog hipotetičkog procesora.

Dobar međukod zadovoljava sledeće osobine:

- pogodan je za davanje značenja čvorovima sintaksne analize,
- pogodan je za prevođenje u mašinski jezik bilo kog odredišnog procesora
- čvorovi stabla međukoda treba da imaju jasna značenja da bi se optimizacione transformacije međukoda mogle lako specificirati i realizovati.

2.3 Troadresni kod

U kompajleru CCC za koji je potrebno proširiti postojeću kompajlersku strukturu podrškom za nove tipove podataka, za međukod se koristi **troadresni kod.**

Troadresni kod predstavlja niz instrukcija oblika ***x = y op z***, gde je *op* kod operacije, x i y dva ulazna operanda i z odredišni operand u koji se smešta rezultat.

Na sledećem primeru može se videti kako se složeni izraz može opisati u troadresnom kodu:

složeni izraz	troadresni kod

`a = b*c + b*d;`	`_t1 = b*c;`
	`_t2 = b*d;`
	`_t3 = _t1+_t2;`
	`a = _t3;`

Može se primetiti, da su za potrebe izračunavanja složenog izraza stvorene privremene promenljive (eng. temps) za smeštanje privremenih rezultata aritmetičkih operacija.

Na kraju treba istaći neke od prednosti korišćenja troadresnog međukoda:

- kod se razbija u jednostavne instrukcije kojima je lako upravljati
- omogućuje se mašinski nezavisna optimizacija
- pojednostavljuje se prelaz na drugu platformu
- pojednostavljuje se problem korišćenje više arhitektura

Iako instrukcije troadresnog koda na na višem nivou mogu imaju najviše tri operanda, na nižem nivou to pravilo ne mora da važi i one se često mogu sastojati i od više od tri operanda.

3. Koncept rešenja

Postojeći prednji sloj kompajlera ima podršku za long long tip, ali se prema njemu odnosi kao prema tridesetdvobitnom tipu. Trebalo bi podesiti veličinu long long tipa i omogućiti njegovo regularno prenošenje u zadnji deo CCC (Cirrus C Compiler) kompajlera preko rtir datoteke.

Potrebno je dodati novi tip, definisati njegovu veličinu i zauzimanje memorije.

Za učitavanje šezdesetčetvorobitnih konstanti ideja je učitavati samo polureči koje nose informaciju tj. koje su različite od nule. Zbog veličine long long-a, taj tip će se jedino moći smestiti u označeni ili neoznačeni dugi akumulator.

Da bi se uveo long long tip trebalo bi da se omogući konverzija između postojećih tipova (celobrojni tipovi i tipovi nepokretnog zareza) i tipa long long-a.

Neke od operacija kao što su množenje, deljenje, ostatak pri deljenju, aritmetičko i logičko pomeranje neophodno je implemetirati za slučaj long long tipa.

Kako su ranije sve celobrojne konstante bile iste veličine, potrebno je omogućiti rukovanje memorijom u slučaju učitavanja i upisivanja šezdesetčetvorobitnog celobrojnog tipa.

Emisija konstanti do sada je bila realizovana samo za tridesetdvobitne celobrojne konstante, trebalo bi omogućiti emisiju long long tipa konstanti u sličajevima korišćenja memorijskih zona podataka x, y i xy, kao i programske memorije p.

Generisanje efikasnijeg koda moguće je postići ekstrakcijom konstanti i izračunavanjem aritmetičkih izraza unapred.

U slučaju ekstrakcije konstanti ideja je celobrojne konstante koje su veće veličine smestiti u memoriju i po potrebi je iz nje učitavati. Prednost se dobija zahvaljujući činjenici da je moguće jednom instrukcijom iz memorije dobaviti 32 bita, a inače je za učitavanje šezdesetčetvorobitne konstante iz koda u dugi akumulator potrebna po jedna instrukcija za svakih 16 bita.

Druga mogućnost povećanja efikasnosti koda je sračunavanje konstantnih izraza. Time se omogućuje da se pri narednoj propagaciji konstante rezultat izraza iskoristi direktno kao ulazni operand instrukcije koja ga koristi, a instrukcija aritmetike kosntantih izraza postoje nepotrebna i uklanja se u procesu eliminacije mrtvog koda.

Broj instrukcija, takođe je, moguće smanjiti ako bi se unapred eliminisali nepotrebni izrazi korišćenjem neutralnih elemenata aritmetičkih operacija u kojima je jedan operand konstanta, a drugi promenljiva. Na osnovu osobine neutralnog elementa moguće je unapred znati vrednost rezultata izraza. Analogno slučaju sračunavnju izraza, dolazi do uklanjanja nepotrebnih instrukcija.

4. Programsko rešenje

4.1 Podešavanje prednjeg dela kompajlera

Prednji deo kompajlera (eng. frontend) već je imao podršku za celobrojni tip long long. Međutim, bilo je potrebno u modulu za rukovanje celobrojnim konstatama koji se nalazi u datoteci *const_ints.c* podesiti njegovu podrazumevanu veličinu i poravnanje (eng. alignment), jer je do tada long long bio tretiran kao tridesetdvobitni tip podataka.

Funkciju za zapis celobrojne konstante u rtir datoteku, *write_rtir_constant* iz datoteke *write_rtir.c,* trebalo je prilagoditi, jer se po prvi put javila potreba za prenosom celobrojne konstante koja je imala 64 bita. Sve celobrojne konstante zadržale su svoje veličine, ali je njihovo prenošenje u rtir datoteku omogućeno funkcijom *fwrite* preko veličine long long konstante, jer je ona bila u stanju da sem svoje vrednosti predstavi i vrednosti svih drugih tipova.

Za čitanje rtir datoteke zadužena je klasa *CEdgParser* prednjeg dela kompajlera. Pošto je sada celobrojna konstanta u rtir datoteku prenošena u šesdesetčetvorobitnoj veličini, funkcija *loadExpression,* klase *CEdgParser,* morala je biti spremna da primi bite u istom formatu u kojima su oni zapisani u datoteku. Za učitavanje konstante iz datoteke iskorištena je već postojeća funkcija *loadLongLong* klase *CRtirFile*. Učitanoj konstanti je na osnovu pročitanih informacija iz datoteke valjalo dodeliti tip, veličinu i memoriju, za šta je bilo potrebno modifikovati funkciju *createIntConst* i klasu *CIntConstOperand*.

4.2 Dodavanje novog tipa

4.2.1 klasa CIntConst

Za potrebe uvođenja šezdesetčetvorobitnih celobrojnih konstanti bilo je potrebno izmeniti postojeću klasu IntConstOperand. Njeno polje *m_value*, koje služi za smeštanje same vrednosti konstante, trebalo je omogućiti korišćenje nove celobrojne konstante koja je po svojoj veličini bila veća od prethodno podržanih. Postojalo je više rešenja da se implementira postavljeni zahtev, ali izabrano je ono najjednostavnije, tim pre, što je pratilo do tada primenjivanu praksu prilikom uvođenja novih tipova konstanti. Ideja je da se za smeštanje vrednosti celobrojne konstante koristiti tip najveće konstante, jer će tada ona umeti predstaviti i vrednosti svih manjih celobrojnih konstanti. Pored same izmene tipa polja *m_value*, bilo je potrebno prilagoditi postojeće funkcije za postavljanje i korišćenje tog polja.

4.2.2 klase CIntegerType i CCoyoteTypeSize

U postojećoj klasi celobrojnih tipova CIntegerType omogućeno je korišćenje long long tipa dodavanjem nove podržane veličine od 64 bita za cele brojeve.

Pomoću klase *CoyoteTypeSize* specificirana je veličina memorije koju će u svojoj upotrebi koristiti:

memorijska zona	x	y	p	xy
veličina	2	2	2	1

Tabela 4.1 Način zauzimanja memorijskih zona za tip long long

Pomoću datoteke *TypeSize.h* definisano je da će long long za svoju upotrebu koristiti dve tridesetdvobitne reči.

```
#define LONG_LONG_INT_SIZE 2
```

4.3 Učitavanje konstanti

klase CIntConstToUnsigLongAccum i CIntConstToSigLongAccum

Da bi se vrednosti konstanti koje postoje u kodu koristile, potrebno je prethodno konstante učitati. Za slučaj long long tipa, zbog njegove veličine moguće ga je smestiti samo u označeni ili neoznačeni dugi akumulator (eng. long accumulator). Za to prebacivanje iz koda u dugi akumulator potrebna je po jedna instrukcija za svakih 16 bita. Međutim, često je slučaj da samu suštinu informacije ne nose baš svi segmenti, pa je tako i uz manji broj od četiri instrukcije moguće u potpunosti učitati vrednost konstante, a da se time ne poremeti njena vrednost. Ideja je učitavati samo polureči koje su različite od nule.

Ukoliko konstanta nosi informaciju u najviših 16 bita, moguće je učitati pomoću instrukcije **fixed16(a0),** u slučaju označenih brojeva, ili **ufixed16(a0)** u slučaju neoznačenih brojeva. Sledećih 16 bita se učitava pomoću instrukcije **lo16(a0).** U slučaju da najviših 16 bita ne postoji, gornju reč moguće je učitati uz pomoć samo jedne instrukcije **halfword(a0).** Nakon završenog učitavanja više reči, na bitima niže reči se nalaze nule. To je veoma značajno u slučaju da su viša ili niža polureč donje reči jednake nuli, jer tada nije potrebna instrukcija za njihovo učitavanje. U suprotnom, za više bite koristi se instrukcija **fixed(a0l**), a za niže **lo16(a0l)**.

Na primer:

1)
```
long long c1 = 0x0001222200003333LL
fixed16(a0) = 0x0001
lo16(a0) = 0x2222
lo16 (a0l) = 0x3333
```

2)
```
long long c2 = 0x0000FFFFFFFF0000LL
halfword(a0) = 0xFFFF
fixed16(a0l) = 0xFFFF
```

4.4 Učitavanje iz memorije i pisanje u memoriju

Operacije učitavanja iz memorije u pisanja u memoriju realizovane su za dve memorijske zone podataka X i Y i jednu programsku memorijsku zonu P. Postupak učitavanja i upisivanja analogan je za sve tri vrste memorije, tako da će u nastavku biti dati postupci objedinjeno. Upisivanje tipa long long u memoriju

klase CLowerStoreLongLongToXMemFromAccum, CLowerStoreLongLongTo - YMemFromAccum i CLowerStoreLongLongToPMemFromAccum

Korišćenjem indeksnog adresiranja na određenu lokaciju u memoriju se upisuje sadržaj gornje reči, a na susednu lokaciju sadržaj donje reči akumulatora.

```
xmem[i0] = a0h
i0 += 1
xmem[i0] = a0l
```

4.4.1 Upisivanje tipa long long u memoriju preko ofseta

klase CLowerStoreLLongToXMemFromAccumViaOffset, CLowerStoreLLongTo - YMemFromAccumViaOffset i CLowerStoreLLongToPMemFromAccumViaOffset

Korišćenjem neposrednog adresiranja na susedne lokacije u memoriji se po redu upisuju gornja i donja reč akumulatora.

```
xmem[_x + 0] = a0h
xmem[_x + 1] = a0l
```

4.4.2 Učitavanje long long tipa iz memorije

klase CLowerLoadLongLongFromXMemToAccum, CLowerLoadLongLongFrom - XMemToAccum i CLowerLoadLongLongFromXMemToAccum

Učitavanje celobrojnog tipa long long vrši se indeksnim adresiranjem kada se sadržaj susednih lokacija u memoriji smešta redom u gornju pa u donju reč akumulatora.

```
a0 = xmem[i0]
i0 += 1
a0l = xmem[i0]
```

4.4.3 Učitavanje long long tipa iz memorije preko ofseta

klase CLowerLoadLongLongFromXMemToAccumViaOffset, CLowerLoadLong - LongFromXMemToAccumViaOffset i CLowerLoadLongLongFromXMemToAccumVia - Offset

Sadržaj susedna 64 bita iz memorije se koršćenjem neposrednog adresiranja smešta u sedamdesetdvobitni akumulator.

```
a0 = xmem[_x + 0]
a0l = xmem[_x + 1]
```

4.5 Emisija celobrojnih globalnih promenljivih

Postojeća realizacija funkcije za emisiju celobrojnih globalnih konstanti `intConstToString` `klase` *CCoyoteEmitGlobalVariable* omogućavala je emisiju svih tipova veličine 32 bita. Uvođenjem tipa koji ima 64 bita trebalo je obezbediti klasifikovanje konstanti pre procesa emisije i sam format emisije šezdesetčetvorobitnih celobrojnih konstanti.

Funkcija za emitovanje globalnih celobrojnih konstanti u ulaznim parametrima prima konstantni operand koji je potrebno emitovati i njegovu memorijsku zonu u kojoj će se nalaziti. U zavisnosti od njegovog pokazatelja veličine tipa i deklarisane memorijske zone ona, konvertujući vrednosti konstanti u stringove koji se upisuju u izlaznu datoteku, generiše četiri moguće opcije formata emisije.

1) za tridesetdvobitne tipove (char, short, int i long)

a) memorijske zone x i memorijska zona y:

```
_normal_length_memx
  .dw   (0x12345678)
```

b) memorijske zone xy:

```
_normal_length_memxy
  .dw   (0x12345678), (0x0)
```

2) za šezdesetčetvorobitni tip (long long)

a) memorijske zone x i memorijska zona y:

```
_llong_length_memy
  .dw   (0x76543210)
  .dw   (0x54321098)
```

b) memorijske zone xy:

```
_llong_length_memxy
  .dw   (0x12345678), (0x12345678)
```

Kao što se to iz primera emisije iznad može videti, prilikom emitovanja konstanti dužine 64 bita, prvo se emituje gornja reč, a zatim donja reč. U slučaju emitovanja nizova, takođe se prati prethodni postupak, a članovi niza se emituju u smeru od najnižeg indeksa ka najvišem.

4.6 Konverzije između celobrojnih tipova

<table>
<tr><th colspan="2"></th><th>a0g</th><th colspan="2">a0h</th><th>a0l</th></tr>
<tr><td>signed</td><td rowspan="2">int</td><td>s</td><td>s</td><td>31 bit</td><td>0</td></tr>
<tr><td>unsigned</td><td>0</td><td colspan="2">32 bita</td><td>0</td></tr>
<tr><td>signed</td><td rowspan="2">long long int</td><td>s</td><td>s</td><td colspan="2">63 bita</td></tr>
<tr><td>unsigned</td><td>0</td><td colspan="3">64 bita</td></tr>
</table>

Tabela 4.2 Celobrojni tipovi i njihovi formati u akumulatorskom registru

4.6.1 Konverzija iz int tipa koji se nalazi u akumulatoru u long long tip

4.6.1.1 klasa CMoveSigLongLongSigInt

Da bi se uradila konverzija iz označenog int-a u označeni long long potrebno je bite više reči akumulatora a0h prebaciti na bite niže reči a0l i uraditi znakovno proširenje više reči i zaštitnih bita akumulatora.

4.6.1.2 klasa CMoveSigLongLongUnsigInt

Za konverziju iz neoznačenog int-a u označen **long long** biti više reči se prebacuju na mesto bita niže reči akumulatora, a viša reč i zaštitni deo se popunjavaju nulama.

4.6.1.3 klasa CMoveUnsigLongLongSigInt

Konverzija označenog int-a u neoznačeni long long zahteva da se viša reč prebaci na mesto bita niže reči, da se uradi znakovno proširenje bita više reči i da se na mesto zaštinih bita akumulatora postavi nula.

4.6.1.4 klasa CMoveUnsigLongLongUnsigInt

U slučaju konverzije iz neoznačenog int-a u neoznačeni long long, više reči akumulatora se prebacuju na mesto nižih, a zaštitni biti i biti više reči se popunjavaju nulama.

4.6.2 Konverzija iz int tipa koji se nalazi u registru u long long tip

4.6.2.1 klasa CMoveSigLongLongIntReg

Za konverziju int-a iz registra u long long tip koji se nalazi u akumulatoru, niža reč akumulatora se popunjava bitima iz tridesetdvobitnog registra, a na ostatku bita akumulatora se vrši proširenje znaka.

4.6.2.2 klasa CMoveUnsigLongLongIntReg

U slučaju konverzije iz registra u neoznačeni akumulator potrebno je sadržaj tridesetdvobitnog registra smestiti na bite niže reči akumulatora, a ostale bite akumulatora popuniti nulama.

4.6.3 Konverzija iz long long tipa u int tip koji se nalazi u akumulatoru

4.6.3.1 klasa CMoveSigIntSigLongLong

Za konverziju iz označenog tipa long long u označeni int potrebno je samo prebaciti bite niže reči akumulatora na mesto bita više reči.

4.6.3.2 klasa CMoveSigIntUnsigLongLong

Postupak je istovetan kao u prethodnom slučaju.

4.6.3.3 klasa CMoveUnsigIntSigLongLong

U slučaju konverzije iz označeno long long-a u neoznačeni int potrebno je bite niže reči akumulatora premestiti na mesto bita više reči i anulirati zaštitni deo akumulatorskog registra.

4.6.3.4 klasa CMoveUnsigIntUnsigLongLong

Postupak je isti kao u prethodnom slučaju.

4.6.4 Konverzija iz long long tipa u int tip koji se nalazi u registru

4.6.4.1 klasa CMoveIntRegLongLong

Konverzija je ista bez obzira na oznake tipova koji učestvuju u konverziji. U tridesetdvobitan registar podataka se smeštaju biti niže reči akumulatorskog registra.

4.6.5 Konverzije između različitih oznaka long long tipa

4.6.5.1 CMoveSigLongLongSigLongLong

Za premeštanje označenog tipa long long iz jednog akumulatora u drugi koristi se operacija sedamdesetdvobitnog pomeranja između registara (eng. bitwise accumulator move).

```
a0 = +a0
```

4.6.5.2 CMoveUnsigLongLongSigLongLong

U slučaju pretvaranja označenog long long-a u neoznačeni long long vrši se sedamdesetdvobitno pomeranje između registara i anuliranje zaštitnog dela akumulatora odredišta.

```
a0=+a0
a0g=(0x0)
```

4.6.5.3 CMoveSigLongLongUnsigLongLong

Za pretvaranje neoznačenog long long-a u označeni long long dovoljno je izvršiti operaciju sedamdesetdvobitnog pomeranja između registara.

4.6.5.4 CMoveUnsigLongLongUnsigLongLong

Premeštanje neoznačenog long long iz jednog akumulatora u drugi vrši se takođe pomoću instrukcije sedamdesetdvobitnog pomeranja.

4.7 Konverzije između long long tipa i tipova nepokretnih zareza

<table>
<tr><td colspan="2"></td><td colspan="2">a0g</td><td colspan="2">a0h</td><td>a0l</td></tr>
<tr><td>signed</td><td rowspan="2">fract</td><td colspan="2">s</td><td>s</td><td>31 bit</td><td>0</td></tr>
<tr><td>unsigned</td><td colspan="2">0</td><td>0</td><td>31 bit</td><td>0</td></tr>
<tr><td>signed</td><td rowspan="2">long fract</td><td colspan="2">s</td><td>s</td><td colspan="2">63 bita</td></tr>
<tr><td>unsigned</td><td colspan="2">0</td><td>0</td><td colspan="2">63 bita</td></tr>
<tr><td>signed</td><td rowspan="2">accum</td><td>s</td><td>8 bita</td><td colspan="2">63 bita</td><td>0</td></tr>
<tr><td>unsigned</td><td colspan="2">9 bita</td><td colspan="2">31 bit</td><td>0</td></tr>
<tr><td>signed</td><td rowspan="2">long accum</td><td>s</td><td>8 bita</td><td colspan="3">63 bita</td></tr>
<tr><td>unsigned</td><td colspan="2">9 bit</td><td colspan="3">63 bita</td></tr>
</table>

Tabela 4.3 Tipovi nepokretnog zareza i njihovi formati u akumulatorskom registru

4.7.1 Konverzija iz tipa long long int u tip accum

Long long ima veći raspoloživi opseg celih brojeva od accum tipa. U slučaju da se broj koji želimo konvertovati uklapa u određeni opseg accum-a, postupak za sva četiri naredna tipa konverzije je isti. Najnižih 9 bita se postavlja na 8 bita zaštitnog dela i na prvi bit gornje reči akumulatora. Ostali deo bita se nulira. U suprotnom, ako vrednost koju je potrebno konvertovati ne može stati u opseg accum-a, dolazi do saturacije tj. zasićenja. U narednim paragrafima akcenat će biti upravo dat na formiranju granica i postupku rešavanja saturacije, jer je to zapravo jedino ono u čemu se ove konverzije razlikuju.

4.7.1.1 klasa CMoveSigAccumSigLongLong

U slučaju konverzije iz označenog long long tipa u označeni accum tip, brojevi veći od 255, tj. najvećeg broja koga je moguće prikazati sa 8 bita, se zamenjuju sa maksimalnom vrednošću accum-a koja približno iznosi 256. Isto važi i za brojeve manje od -255 koji se zamenjuju saturisanom vrednošću najmanjeg broja koga je moguće prikazati.

4.7.1.2 klasa CMoveSigAccumUnsigLongLong

Za konverziju iz neoznačenog long long tipa u označeni tip accum-a, nije potrebno postavljati donju granicu jer neoznačeni broj koji želimo pretvoriti ne može biti negativan. Gornja granica ista je kao u prethodnom slučaju.

4.7.1.3 klasa CMoveUnsigAccumSigLongLong

Kod konverzije iz označenog long long-a u neoznačeni accum, gornja granica postavljena je na najveći broj koji neoznačeni accum tip može prikazati sa 9 bita (512), dok je donja granica saturacije postavljena na nulu, jer neoznačeni broj koji se dobija kao produkt konverzije mora biti pozitivan ili jednak nuli.

4.7.1.4 klasa CMoveUnsigAccumUnsigLongLong

Neoznačeni long long tipovi nemaju proveru za donju granicu, jer ne mogu biti negativni. Neoznačeni accum tip, ima 9 bita za prikaz broja, pa je gornju granicu saturacije potrebno postaviti na 511.

4.7.2 Konverzija iz long long-a u long accum

Kao i u slučaju konverzije između tipova long long int i accum, i u ovom slučaju se javlja potreba za korišćenjem tehnike zasićenja jer je opseg celih brojeva long long tipa daleko veći od opsega koji je moguće prikazati tipom long accum. Zapravo, iako long long accum koristi znatno više bita za predstavljanje brojeva od tipa accum, oni imaju isti opseg predviđen za smeštanje celog dela broja. To se može jasno uočiti u tabeli 4.3 u kojoj su prikazani tipovi nepokretnog zareza.

U slučaju da se broj koji želimo konvertovati uklapa u opseg tipa long accum predviđen za smeštanje celih delova broja, postupak za naredne konverzije je isti. Najniža 9 bita je potrebno postaviti na 8 bita zaštinog dela akumulatora i na prvi bit gornje reči akumulatora. Svi ostali biti se postavljaju na vrednost nula.

4.7.2.1 klasa CMoveSigLongAccumSigLongLong

Za konverziju iz označenog tipa long long int-a u označeni tip long accum, celi brojevi koji se ne mogu prikazati uz pomoć 8 bita se zamenjuju vrednošću saturacije. Vrednosti saturacije predstavljaju najmanji i najveći broj koji je moguće predstaviti uz pomoć 8 bita za celi broj i 63 bita za razlomljeni deo. To se zapravo i krije jedina razlika u konverziji u odnosu na tip accum, koji je za saturisani deo koristio 31 bit.

4.7.2.2 klasa CMoveSigLongAccumUnsigLongLong

Konverzija iz neoznačenog long long-a u označeni long accum nema potrebu za proverom donje granice, jer neoznačeni broj ne može biti negativan. Gornja granica određena je mogućnošću 8 bita koji služe za prikaz celog broja, tako da su svi brojevi veći od 255 prikazani pomoću saturisane vrednosti koja je približno jednaka 256.

4.7.2.3 klasa CMoveUnsigLongAccumSigLongLong

U slučaju konverzije iz označenog long long int-a u neoznačeni long accum, donja granica saturacije postavljena je na nulu, jer neoznačeni broj mora biti pozitivan ili jednak jedinici. Gornja granica zasićenja postavljena je na 512, pošto neoznačeni brojevi imaju 9 bita za smeštanje celog dela broja.

4.7.2.4 klasa CMoveUnsigLongAccumUnsigLongLong

Konverzija neoznačenog long long-a u neoznačeni long accum proverava samo da li je broj koji želimo konvertovati veći od 512. Ukoliko jeste, u akumulator se postavlja vrednost saturacije čiji svi biti su jedinice.

4.7.3 Konverzija iz tipa long long u tip fract

Tip nepokretnog zareza fract nema bite ostavljene za predstavu celih brojeva. Označeni fract ima opseg (-1,1), dok je opseg neoznačenog fract-a [0,1).

4.7.3.1 klasa CMoveSigFractSigLongLong

Za slučaj konverzije iz označenog tipa long long int u označen tip fract, koristi se konverzija iz označenog long long-a u označeni long accum. Biti niže reči se postavljaju na nula, a umesto zaštitinih bita radi se proširenje znaka.

4.7.3.2 klasa CMoveUnsigFractSigLongLong

Kod pretvaranja označenog long long-a u neoznačeni fract, ukoliko je broj veći ili jednak jedan u akumulator se na poziciji bita više reči osim prvog bita smeštaju jedinice, a na sve ostale bite nule. Ako je broj manji od jedan, akumulator se popunjava nulama.

4.7.3.3 klasa CMoveSigFractUnsigLongLong

Konverzija neoznačenog long long-a u označeni fract koristi rezultat konverzije neoznačenog long long int-a u označeni long accum, s tim što se nakon konverzije niža reč i zaštitni biti postavljaju na nula.

4.7.3.4 klasa CMoveUnsigFractUnsigLongLong

Postupak pretvaranja neoznačenih long longa u neoznačeni fract je identičan pretvaranju označenog long long-a u neoznačeni fract.

4.7.4 Konverzija iz long long-a u long fract

Kao i u slučaju fract-a, long fract ne poseduje bite za prikaz celih brojeva. Dakle, brojevi mogu uzeti sledeće vrednosti: za označeni fract nulu ili neku od saturisanih vrednosti -1 ili 1; a za neoznačeni nulu ili vrednost zasićenja koja je približno jednaka 1.

4.7.4.1 klasa CMoveSigLongFractSigLongLong

U slučaju konverzije iz označenog long long-a u označeni fract koristi se rezultat konverzije iz označenog long long-a u long accum, samo se zaštitni biti akumulatora u slučaju da je broj pozitivan postavljaju na nula, a ako je negativan na sve jedinice.

4.7.4.2 klasa CMoveUnsigLongFractSigLongLong

Kod pretvaranja označenog long long-a u neoznačeni long fract, ukoliko je broj veći ili jednak jedan u akumulator se smešta sve jedinice izuzev bita niže reči, zaštitnog bita i prvog bita više reči na kome se uvek kod neoznačenog tipa fract nalaze nule. U suprotnom rezultat pretvaranja se dobija tako što se na sve bite akumulatora postavi nula.

4.7.4.3 klasa CMoveSigLongFractUnsigLongLong

Za slučaj konverzije iz neoznačenog long long-a u označeni fract koristi se rezultat konverzije iz neoznačenog long long-a u long accum, samo se zaštitni biti akumulatora postavljaju na nula.

4.7.4.4 klasa CMoveUnsigLongFractUnsigLongLong

Pretvaranje neoznačenih long long-a u neoznačeni long fract je identično pretvaranju označenog long long-a u neoznačeni long fract.

4.7.5 Konverzija iz tipova nepokretnog zareza u celobrojni tip long long

Za razliku od konverzija u suprotnom smeru, konverzije iz svih tipova pokretnog istih označavanja (eng. signification) moguće je uraditi potuno na isti način. U poglavljima ispod dati su postupci u slučajevima konvertovanja neoznačenih i označenih fiksnih tipova u celobrojni šezdesetčetvorobitni tip long long.

4.7.5.1 klase CMoveLongLongUnsigAccum, CMoveLongLongUnsigLongAccum i CMoveLongLongUnsigLongFract

U slučaju neoznačene konverzije potrebno je pomeriti bite zaštitnog dela akumulatora i prvi bit gornje reči na poziciju bita donje reči. Ostali biti se postavljaju na vrednost nula.

```
AnyReg(a0l, a0h)
AnyReg(a0h, a0g)
a0 = a0 << 1
AnyReg(a0l, a0h)
uhalfword(a1l) = (511)
a0 = a0 & a1
a0g = (0x0)
```

4.7.5.2klase CMoveLongLongSigAccum, CMoveLongLongSigLongAccum, CMoveLongLongFract i CMoveLongLongSigLongFract

Za konverziju označenih tipova nepokretnog zareza u tip long long koristi se konverzija za neoznačene brojeve, tako što joj se pošalje apsolutna vrednost broja kojeg je potrebno konvertovati, a ona nakon što primi povratnu vrednost od funkcije ako je broj negativan vrši negaciju bita akumulatora.

```
a2 =+ a0
a0 = |a0|
call __ulaccum_to_longlong
a2 & a2
if (a >= 0) jmp >_end
a0 =- a0
```

4.8 Ekstrakcija konstanti

(funkcija checkIfSuitableForExtraction klase CConstantExtraction)

Sam postupak otkrivanja da li bi ekstrakcija konstanti donosila dobitak vrši se procesom maskiranja pri čemu se otkriva da li određena polureč od 16 bita nosi informaciju. Nakon što se provere vrednosti svih polureči, vrši se računica eventualnog dobitka u vidu tasa na vagi, na kojoj se sa jedne strane nalazi potreban broj instrukcija za čitanje konstante iz koda, a sa druge strane potreban broj instrukcija za čitanje konstante iz memorije.

4.9 Izračunavanje izraza

funkcija attemptConstantArithmetic klase CConstantPropagation

4.9.1 Aritmetika konstanti

Izračunavanje izraza predstavlja mašinski nezavisnu optimizaciju u kojoj dolazi do prethodnog izračunavanja rezultata instrukcija koje sadrže aritmetičke operacije između dve konstante. Postojeća instrukcija zamenjuje se pomeračkom (eng. move) instrukcijom sa istim odredištem dok ulazni operand predstavlja novoformiranu konstantu čija vrednost odgovara rezultatu aritmetičke operacije.

Propagacija izraza sama po sebi ne donosi dobit, ali je u kombinaciji sa propagacijom konstanti rezultat aritmetičkih operacija moguće staviti direktno u ulazni operand instrukcije koja ga koristi. Izvorna instrukcija postaje nepotrebna i moguće ju je ukloniti.

Postojeći kompajler imao je samo jednu veličinu za podršku svim celobrojnim konstantama. Uvođenjem nove veličine, bilo je potrebno postojeću propagaciju izraza prilagoditi novoj veličini celobrojne konstante.

Uslov za postojanje mogućnosti sračunavanja celobronih izraza je da su oba operanda celobrojne konstante i da kod operacije odgovara nekom od kodova aritmetičkih operacija.

U slučaju sabiranja, odredištu instrukcije se dodeljuje nova konstanta čija je vrednost jednaka zbiru sabiraka ulaznih operanada instrukcije. U slučaju da se sabiraju dve neoznačene konstante tridesetdvobitne dužine za rezultat će se uzeti samo najmanje značajnih 32 bita.

U slučaju da kompajler naiđe na operaciju oduzimanja, odredištu će se dodeliti konstanta čija je vrednost jednaka razlici umanjenika koji je predstavljen prvim ulaznim operandom i umanjioca, tj. drugog ulaznog operanda. Neoznačeno oduzimanje tridesetdvobitnih brojeva je zaštićeno, tako da se u slučaju oduzimanja manjeg broja od većeg, odredištu dodeljuje samo vrednost donjih 32 bita.

Za slučaj množenja pravilo je analogno onom za sabiranje i oduzimanje. Odredištu se dodeljuje proizvod ulaznih operanada, a u slučaju da je došlo do prekoračenja u množenju neoznačenih tridesetdvobitnih konstanti, tada se uzima samo niža 32 bita.

U slučaju deljenja, ukoliko je delilac jednak nuli, kako deljenje nulom nije definisano u odredište se sprema odgovarajuća vrednost koja bi ukazala na grešku. Ako je delilac različit od nule, tip oznake deljenika određuje da li će se upotrebiti označeno ili neoznačeno deljenje. Količnik se smešta u odredišni operand.

Za aritmetičke operacije "I" (eng. and), "ILI" (eng. or) i ekskluzivno "ILI" (eng. xor) postupak je jednostavan. Izračuna se rezultat aritmetičke operacije i u odredište se smešta konstata čiji tip odgovara tipu ulaznih operanada.

Logičko i aritmetičko pomeranje dele isti kod operacije, a koje će pomeranje biti realizovano zavisi isključivo od tipa konstante koja je smeštena u prvi operand. U slučaju da je ona označena radi se o aritmetičkom pomeranju, a ako je neoznačena tada je pomeranje logičko. Obe vrste pomeranja, i logičko i aritmetičko se razlikuju u zavisnosti od tipa celobrojne konstante koju je potrebno pomeriti u levo ili u desno.

U slučaju pomeranja u levo, u odredište instrukcije se smešta rezultat logičkog ili aritmetičkog pomeranja ako vrednost drugog operanda nije veća ili jednaka broju bita celobrojne konstante. U suprotnom, u odredište se smešta kosntanta čija je vrednost jednaka nuli.

Kod pomeranja u desno, se ako drugi operand nije jednak ili veći od broja bita za predstavu konstante, u odredišni operand instrukcije smešta rezultat logičkog ili aritmetičkog pomeranja u desno. U suprotnom, kod logičkog pomeranja se u odredište smešta nula, a kod aritmetičkog pomeranja vrednost konstante koja ukazuje da se desila greška.

Za sve prethodne slučajeve, tip konstante koja se smeštala u akumulator odgovarao je tipu prvog ulaznog operanda instrukcije koje je bila menjana.

Propagacija izraza realizovana je i za operacije poređenja "jednako", "veće", "manje", "veće ili jednako", "manje ili jednako" i "nije jednako", kako za označene tako i za neoznačene brojeve. U odredište instrukcije se smešta konstanta tridesetdvobitne veličine čija je vrednost jedan, u slučaju da je iskaz poređenja istinit ili nula ako poređenje nije istinito. Može se primetiti, u ovim slučajevima veličina konstante ne zavisi od tipova ulaznih operanada.

4.9.2 Neutralni elementi aritmetičkih operacija

Broj instrukcija, takođe je, moguće smanjiti ako bismo se unapred rešili nepotrebnih izraza iskoristivši neutralne elemente aritmetičkih operacija u kojima je jedan operand konstanta, a drugi promenljiva. Na osnovu osobine neutralnog elementa moguće je unapred znati vrednost izraza, smestiti je u odredište i zameniti postojeću instrukciju operacije.

Ako je jedan od operanada jednak nuli u slučajevima množenja, aritmetičkog "I" i logičkog "I", rezultat će takođe biti jednak nuli, pa se u odredište smešta konstanta čija je vrednost jednaka nuli.

Ako je konstanta jednaka jedinici rezultat operacije logičko "ILI" će sigurno biti jednak jednici.

U slučaju aritmetičkog "ILI", rezultat će biti jednak vrednosti konstante u slučaju da je svaki bit konstante jedinica.

Ukoliko su vrednosti oba ulazna operanda jednake, rezultat operacije oduzimanja ili ekskluzivnog "ILI" će biti jednak nuli, dok će u slučaju deljenje biti jednak jedinici. Odgovarajuća vrednost konstante se smešta u odredište i ta instrukcija zamenjuje instrukciju operacije.

4.10 Aritmetičko i logičko pomeranje u desno

Prilikom uvođenja šezdesetčetvorobitnog celobrojnog tipa long long za logičko i aritmetičko pomeranje u levo moguće je iskoristiti postojeći mehanizam za tridesetdvobitne brojeve. To, međutim, nije slučaj sa pomeranjima u desno koje ima određene specifičnosti koje je potrebno sprovesti prilikom implementacije.

Pomeranje u desno implementirano je za tri slučaja:

- pomeranje za vrednost promenljive
- pomeranje za najčešće vrednosti konstanti
- pomeranje za nestandardne vrednosti konstanti

4.10.1 Pomeranje za vrednost promenljive

(klasa CBitwiseLongLongShiftRightVariableValue)

Aritmetičko i logičko pomeranje u desno za vrednost promenljive realizovano je preko funkcija napisanih u asembleru. Razlikuju se u tome što se pre svih elementarnih pomeranja ako je u pitanju logičko pomeranje u zaštitni deo akumulatora stavlja nula. Dalje funkcionišu po istom principu: preko ulaznih parametara funkcije se uzima broj koje je potrebno pomeriti i zatim se hardverska petlja sa instrukcijom jediničnog pomeranja izvršava onoliko puta koliko je definisano drugim operandom instrukcije pomeranja.

4.10.2 Pomeranje za najčešće vrednosti konstanti

(klase CBitwiseShiftRightSigLongLongStdVal1, CBitwiseShiftRightSigLongLongStd-Val2, CBitwiseShiftRightUnsigLongLongStdVal1 i CBitwiseShiftRightUnsigLongLong-StdVal2)

Za konstante koje imaju vrednost jedan i dva, pomeranje je realizovano veoma jednostavno, pomoću jedne odnosno dve elementarne funkcije pomeranja. Razlika između aritmetičkog i logičkog pomeranja je samo u tome što se kod logičkog pomeranja anulira zaštitni deo akumulatora koji se pomera.

4.10.3 Pomeranje za ostale nestandardne vrednosti

(klasa CBitwiseLongLongShiftRightConstNonstandardValue)

Pomeranje za ostale vrednosti koje konstanta može imati je slično pravilu pomeranja za vrednost promenljive. Dakle, u slučaju logičkog pomeranja u zaštitni deo akumulatora se stavlja nula. Zatim se pomoću fizički podržane petlje vrši pomeranje onoliko puta kolika je vrednost konstante.

4.11 Množenje celobrojnih šezdesetčetvorobitnih brojeva

4.11.1 CLowerMulUnsignedLongLong

Množenje neoznačnih šezdesetčetvorobitnih brojeva realizovano je pozivanjem funkcije napisane u asembleru. Ideja je prikazati množenje dva neoznačena šezdesetčetvorobitna broja kao proizvod zbirova gornje i donje reči dva broja.

$(Z_H * 2^{32} + Z_L) = (X_H * 2^{32} + X_L) * (Y_H * 2^{32} + Y_L)$

Ukoliko se izmnože ta dva binoma dobija se:

$(Z_H * 2^{32} + Z_L) = (X_H * Y_H)\, 2^{64} + (X_H * Y_L + X_L * Y_H) * 2^{32} + X_L * Y_L$

Za smene: $A = X_L * Y_L$, $B = X_H * Y_L + X_L * Y_H$ i $C = X_H * Y_H$,

izraze je moguće grupisati na sledeći način:

$X_L * Y_L = A = A_H * 2^{32} + A_L$
$(X_H * Y_L + X_L * Y_H) * 2^{32} = B * 2^{32} = (B_H * 2^{32} + B_L) * 2^{32} = B_H * 2^{64} + B_L * 2^{32}$
$(X_H * Y_H) * 2^{64} = C * 2^{64}$

Smenom dobijenih izraza u početnu jednačinu, dobija se sistem od tri jednačine sa dve nepoznate:

$0 * 2^{64} = B_H * 2^{64} + C * 2^{64}$	=>	$0 = C + B_H$
$Z_H * 2^{32} = A_H * 2^{32} + B_L * 2^{32}$		$Z_H = A_H + B_L$
$Z_L = A_L$		$Z_L = A_L$

Dakle, gornju reč je moguće izračunati kao zbir gornja 32 bita proizvoda nižih reči činioca i donja 32 bita unakrsnog proizvoda nižih i viših reči činioca.

$Z_H = A_H + B_L = \text{high32}(X_L * Y_L) + \text{low32}(X_H * Y_L + Y_H * X_L)$

Donja reč se može dobiti kao donja 32 bita proizvoda nižih reči činioca.

$Z_L = A_L = \text{low32}(X_L * Y_L)$

Kako neoznačeni long long tip koristi 64 bita sedamdesetdvobitnog akumulatora, gornja 32 bita unakrsnog proizvoda i proizvod gornjih reči činioca se odbacuju.

$C + B_H = X_H * Y_H + (\text{high32}(X_H * Y_L + Y_H * X_L) = 0$

Na kraju se dobija formula za proizvod množenja dva neoznačena šezdesetčetvorobitna long long-a:

$Z_H * 2^{32} + Z_L = (\text{low32}(X_H * Y_L) + \text{low32}(Y_H * X_L)) * 2^{32} + (\text{high32}(X_L * Y_L) + \text{low32}(X_L * Y_L))$

Na mesto zaštitnih bita akumulatora(eng. guard bits) se stavljaju nule.

4.11.2 CLowerMulSignedLongLong

Množenje označenih brojeva tipa long long realizovano je preko množenja neoznačenih long long brojeva. Funkciji se prosleđuju apsolutne vrednosti brojeva koje je potrebno pomnožiti. Ukoliko su oba broja istog znaka (pozitivna ili negativna), proizvod dva označena broja isti je kao i u slučaju neoznačenih brojeva. U suprotnom, ukoliko su brojevi različitog predznaka, biti povratne vrednosti funkcije se invertuju da bi se dobio proizvod koji je negativan.

4.12 Deljenje brojeva tipa long long

4.12.1 CLowerDivUnsignedLongLong

Deljenje dva neoznačena broja najjednostavnije bi bilo moguće implementirati na način da se izračuna koliko puta se od deljenika može oduzeti delilac.

Međutim, ideja koja će biti izneta obavlja istu funkcionalnost ali znatno efikasnije. Na početku se računa koliko je iteracija potrebno za izračunavanje količnika tako što se u svakoj od iteracija trenutna vrednost delioca množi sa dva sve dok je deljenik veći od delioca. Zatim se za izračunati broj iteracija pravi fizički podržana petlja. Na početku petlje trenutna vrednost rezultata se množi sa dva. Ukoliko je vrednost deljenika veća od delioca, od deljenika se oduzima delilac i poslednji bit količnika se postavlja na vrednost jedan. Na kraju petlje se trenutna vrednost deljenika množi sa dva. Ceo proces zaštićen je zastavicom koja pokazuje da li je eventualno došlo do prenosa u toku neke od operacija, kako se ne bi dobio pogrešan rezultat za vrednost količnika.

Algoritam koji je primenjen, analogan je već postojećoj implementaciji neoznačenog deljenja za tridesetdvobitne cele brojeve.

```
cnt = carry = r = 0;
while (((y>>63)&1)==0 && ( x > y ) )
{
	y += y;
	cnt++;
}
for (i=0; i<=cnt; i++)
{
	r += r;
	if (x >= y || carry)
	{
		x -= y;
		r |= 1;
	}
	x += x;
	carry = ((x >> 63) & 1);
}
```

4.12.2 CLowerDivSignedLongLong

Deljenje označenih brojeva vrši se na način da se pošalju funkciji za deljenje neoznačenih brojeva apsolutne vrednosti brojeva koje je potrebno podeliti. U slučaju da su oba broja istog znaka rezultat deljenja predstavlja povratna vrednost pozvane funkcije. U suprotnom, potrebno je invertovati bite količnika dobijenog neoznačenim deljenjem.

4.13 Ostatak pri deljenju brojeva tipa long long

4.13.1 CLowerModuleLLongUnsigned

CLowerModuleLLongUnsigned klasa obezbeđuje podršku računanja ostatka pri deljenju neoznačenih brojeva tipa long long. Ideja je iskoristiti funkcije realizovane za množenje i deljenje neoznačenih long long brojeva. Postupak je vrlo jednostavan: od deljenika se oduzima proizvod delioca i količnika deljenja. Računanje ostatka pri deljenju neoznačenih long long tipova realizovano je preko funkcije napisane u asembleru. Ispod je dato kako bi funkcija izgledala da je pisana u c programskom jeziku:

```
long long _umod_llong(unsigned long long x, unsigned long long y)
{
     long long r = _udiv_llong(x, y);
     return x - r * y;
}
```

4.13.2 CLowerModuleLLongSigned

Klasa za računanje ostatka pri deljenju označenih brojeva radi po istom principu kao što je to slučaj sa neoznačenim brojevima.

5. Ispitivanje i verifikacija

Sam proces testiranja počeo je i tekao uporedo sa fazom razvoja programskog rešenja. Mehanizam koji je korišten u tom periodu bilo je pisanje C koda sa ciljem testiranja određenog modula, a donošenje zaključaka rukovođeno je analizom dobijenog programskog koda u asembleru. Uočene greške i nedostaci većinom su bili otklanjani u hodu.

Nakon završetka faze kodiranja usledila je i formalna verifikacija. Za testiranje su korišteni gcc testni skupovi (eng. gcc test suite) koji su sadržali izvršive testne slučajeve koje je, dakle, bilo moguće prevesti, povezati i pokrenuti. Ovi DejaGnu testni skupovi su pokretani iz osnovne skripte, nazvane *runtest.* Kriterijum pokretanja bio je definisan u konfiguracionoj datoteci. Cilj korišćenja DejaGnu testova bio je obezbediti jedinstveni prednji deo za sve testove. Svaki od pojedinačnih testova licenciran od strane GPL-a zasnivao se na očekivanjima i ukoliko ta očekivanja nisu bila ispunjena test se proglašavao neuspešnim.

Dodavanje novog tipa podataka zahtevao je dosta izmena na samoj kompajlerskoj strukturi, tako su se problemi javljali i u testovima koji nisu prvenstveno pisani za testiranje operacija sa long long tipom.

Ukupan broj korištenih testnih slučajeva bio je 1215, od kojih je neposredno za tip long long pisano 55.

Svi testovi su ocenjeni kao uspešni.

U toku procesa ispitivanja i verifikacije dobijeno programsko rešenje kompajlera pokazalo se kao logički ispravno, stabilno i robusno.

6. Zaključak

U ovom radu realizovano je rešenje dodavanja novog tipa long long u postojeću kompajlersku strukturu sa ograničenim resursima. Zbog zaista prevelikog obima broja klasa i funkcija, akcenat je stavljen na to da se na razumljiv i precizan način objasni sama bit njihove funkcionalnosti. Za uvođenje novog tipa long long u kompajlersku strukturu bilo je potrebno:

- napraviti određena podešavanja na prednjem delu kompajlera,
- precizirati veličinu i memoriju koju će koristiti novi tip,
- omogućiti učitavanje šezdesetčetvorobitnih konstanti iz koda,
- obezbediti rukovanje memorijom za novi tip podataka,
- omogućiti konverzije između novog tipa i postojećih tipova podataka,
- omogućiti poboljšanje efikasnosti koda korišćenjem modula za ekstrakciju konstanti i izračunavanje izraza i
- implemetirati operacije koje nisu bile podržane za šezdesetčetvorobitni celobrojni tip kao što su logičko i aritmetičko pomeranje, množenje, deljenje i računanje ostatka pri deljenju.

Najteže pri izradi rada bilo je predvideti i pokriti što više karakterističnih slučajeva koji se mogu susresti prilikom obrade izvornog koda.

Dobijeno rešenje pokazalo se tokom faze testiranja kao logički ispravno, robusno i otporno na greške. Dodavanjem novog tipa u kompajlersku strukturu zadržana je skalabilnost samog kompajlera, što će prvenstveno biti značajno prilikom uvođenja tipova pokretnog zareza.

Treba istaći da postoji mogućnost unapređenja programskog rešenja, povećanjem efikasnosti čitanja i upisivanja u memoriju koristeći četvrtu memorijsku zonu xy. Istovremenim pristupom memorijskim zonama x i y, omogućilo bi se dobavljanje iz memorije ili smeštanje u memoriju šezdesetčetvorobitne reči u samo jednoj instrukciji.

7. Literatura

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman: Compilers : principles, techniques, and tools

[2] Steven S. Muhnick - Advanced Compiler Design Implementation

[3] Vladimir Kovačević, Miroslav Popović: Sistemska programska podrška u realnom vremenu, Univerzitet u Novom Sadu, Fakultet Tehničkih Nauka, 2002

[4] ISO/IEC 9899:TC3 : C Language Standard - C99