Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Jakob Erzar

Prilagoditev LLVM-ja za sistem SIC/XE

DIPLOMSKO DELO

UNIVERZITETNI ŠTUDIJSKI PROGRAM
PRVE STOPNJE
RAČUNALNIŠTVO IN INFORMATIKA

Mentor: doc. dr. Boštjan Slivnik

Ljubljana, 2019

COPYRIGHT. Rezultati diplomske naloge so intelektualna lastnina avtorja in Fakultete za računalništvo in informatiko Univerze v Ljubljani. Za objavo in koriščenje rezultatov diplomske naloge je potrebno pisno privoljenje avtorja, Fakultete za računalništvo in informatiko ter mentorja.

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Prilagodite zadnji del prevajalnika, ki temelji na sistemu LLVM, za hipotetični procesor SIC/XE. Prilagoditev naj omogoča prevajanje programov, ki so napisani v programskem jeziku C, s prevajalnikom Clang, v strojno kodo za procesor SIC/XE. Zagotovite, da bo prilagojeni zadnji del prevajalnika omogočal optimizacijo vmesne in strojne kode na enak način, kot jo omogoča za druge procesorje. Opišite morebitne omejitve, ki jih boste morali uvesti pri prilagoditvi na SIC/XE procesor.

Zahvaljujem se mentorju doc. dr. Boštjanu Slivniku za vse nasvete, pomoč in potrpežljivost. Zahvalil bi se tudi moji družini in prijateljem, ki so me pri študiju ves čas podpirali. Posebna zahvala gre še Karin, ki mi je stala ob strani, mi pomagala in me spodbujala.

Kazalo

Povzetek

Abstract

1 Uvod **1**

2 Sistem SIC/XE **3**

2.1 Pomnilnik 3

2.2 Registri 3

2.3 Formati podatkov 4

2.4 Oblike ukazov 4

2.5 Načini naslavljanja 6

2.6 Ukazi 7

3 LLVM **11**

3.1 Zgodovina 11

3.2 Uporabniki LLVM-ja 12

3.3 Arhitektura 13

3.4 Primerjava s klasičnimi prevajalniki 13

3.5 Vmesna koda LLVM 14

3.6 Sprednji deli 18

3.7 Uporabljena orodja 19

4 Dodajanje podpore za arhitekturo SIC/XE v LLVM **21**

4.1 Domensko specifičen jezik TableGen 22

4.2 Proces generiranja kode 25
4.3 Splošne informacije o ciljni arhitekturi 27
4.4 Informacije o registrih 29
4.5 Nižanje in legalizacija DAG-ov 32
4.6 Definicija izbora ukazov 34
4.7 Izbiranje ukazov 38
4.8 Izpis zbirne kode 40
4.9 Integracija v LLVM 41
4.10 Omejitve dodane podpore za SIC/XE 42

5 Evaluacija **43**

5.1 Prevajanje LLVM-ja 43
5.2 Primer rekurzivnega programa 44
5.3 Primer iterativnega programa 51
5.4 Primer programa s tabelo in kazalci 55
5.5 Izvajanje v simulatorju 60

6 Sklepne ugotovitve **63**

Literatura **66**

Seznam uporabljenih kratic

kratica	angleško	slovensko
SIC	Simplified Instructional Computer	Poenostavljen ukazni računalnik
SIC/XE	SIC/Extra Equipment	SIC z dodatno opremo
LLVM IR	LLVM Intermediate Representation	Vmesna koda LLVM
GCC	GNU Compiler Collection	Zbirka prevajalnikov GNU
SSA	Static Single Assignment	Enkratno statično prirejanje
DAG	Directed Acyclic Graph	Usmerjen acikličen graf
ELF	Executable and Linkable Format	Datoteka, pripravljena za izvajanje ali povezovanje
SVC	Supervisor Call	Nadzorniški sistemski klic

Povzetek

Naslov: Prilagoditev LLVM-ja za sistem SIC/XE

Avtor: Jakob Erzar

Ogrodje LLVM omogoča lažji razvoj prevajalnikov, saj ga v prevajalniku lahko uporabimo za generiranje strojne kode več različnih arhitektur. V tem diplomskem delu vanj dodamo podporo za generiranje strojne kode arhitekture SIC/XE, ki je namenjena poučevanju sistemske programske opreme. S tem omogočimo prevajanje v strojno kodo sistema SIC/XE iz vseh programskih jezikov, ki se lahko prevedejo v LLVM-jevo vmesno kodo. Podporo za arhitekturo SIC/XE integriramo tudi v prevajalnik Clang, ki je namenjen predvsem prevajanju jezikov C in C++. Delovanje pokažemo s prevajanjem treh programov v C-ju in primerjamo število ukazov glede na stopnjo optimizacije.

Ključne besede: LLVM, prevajalnik, SIC, SIC/XE.

Abstract

Title: LLVM backend for SIC/XE

Author: Jakob Erzar

The LLVM compiler framework can be used as a compiler backend that supports generating machine code for multiple computer architectures. In this thesis we implemented an LLVM backend for the SIC/XE architecture, which is used for teaching system software. This enables generation of SIC/XE machine code from every language that can be compiled to LLVM's intermediate representation. We also added SIC/XE support to the Clang compiler, which is used for compiling C-family programming languages. We demonstrated our work with compilation of three different C programs and compared the amount of instructions with different optimization levels.
Keywords: LLVM, compiler, SIC, SIC/XE.

Poglavje 1

Uvod

Dandanes je velika večina programov napisana v višjenivojskih programskih jezikih, iz katerih se nato prevedejo v strojni jezik ciljnega sistema. Za prevod programov torej potrebujemo program, ki bo to storil – prevajalnik. Programskih jezikov je veliko, različnih arhitektur, za katere lahko program prevedemo, pa prav tako (čeprav so nekatere popularnejše kot druge). Da bi lahko pokrili vse kombinacije programskih jezikov in računalniških arhitektur, bi morali napisati veliko prevajalnikov.

Sodobni prevajalniki so zato pogosto ločeni na sprednji in zadnji del, med njima pa je vmesni jezik – sprednji del program prevede iz programskega v vmesni jezik, zadnji del pa iz vmesnega jezika v strojno kodo. Ta princip uporablja tudi projekt LLVM, ki je zbirka modularnih orodij in tehnologij, namenjenih razvoju prevajalnikov [16]. Tako nam LLVM omogoča, da za poljubni programski jezik implementiramo le sprednji del, LLVM pa poskrbi za pretvorbo iz LLVM-jeve vmesne kode (LLVM IR) v strojno kodo ali obratno – za poljubno računalniško arhitekturo lahko implementiramo le zadnji del, s tem pa nam LLVM omogoči prevajanje iz večjega števila programskih jezikov v strojno kodo poljubne arhitekture.

SIC/XE je računalnik, ki ga je Leland Beck predstavil v svoji knjigi *System Software: An Introduction to Systems Programming* [3]. Namenjen je poučevanju sistemske programske opreme, spoznali pa smo ga tudi na FRI

pri predmetu Sistemska programska oprema. Zanj je že na voljo prevajalnik iz C-ja [11], vendar ne omogoča prevajanja iz drugih jezikov in raznih optimizacij, ki jih lahko izvede LLVM.

Namen te diplomske naloge je v zadnji del LLVM-ja dodati podporo za sistem SIC/XE, da bo omogočal prevajanje iz več jezikov v strojno kodo sistema SIC/XE. Tako bo mogoče za sistem SIC/XE pisati v več jezikih, poleg tega pa bo koda tudi bolj optimizirana. Implementacija bo poučna za tiste, ki so SIC/XE že spoznali pri učenju sistemske programske opreme in bodo želeli videti, kako v LLVM implementirati podporo za novo arhitekturo. Podobno je bila v LLVM dodana podpora za sistem TriCore [9].

V drugem in tretjem poglavju najprej spoznamo računalnik SIC/XE in projekt LLVM. Temu v četrtem poglavju sledi razlaga delov implementacije. Na koncu je v petem poglavju predstavljen še primer programov v C-ju in njihov prevod v zbirno kodo sistema SIC/XE.

Poglavje 2

Sistem SIC/XE

SIC oziroma *Simplified Instructional Computer* je hipotetični računalnik, ki ga je Leland Beck predstavil v njegovi knjigi *System Software: An Introduction to Systems Programming* [3]. Ker so sodobni procesorji postali precej kompleksni, je z njimi težko poučevati programiranje sistemske programske opreme. Zato je bil SIC ustvarjen z namenom, da je čim bolj preprost in jasen in tako nadomešča kompleksnejše sisteme pri poučevanju. Poleg standardne različice je Beck predstavil tudi sistem SIC/XE, kjer XE pomeni *extra equipment* ali *extra expensive*.

2.1 Pomnilnik

Bajti so 8-bitni, zaporedje treh pa sestavlja besedo (angl. *word*). V sistemu SIC lahko naslavljamo 2^{15} bajtov, v razširjeni različici, SIC/XE, pa 2^{20} oziroma 1 megabajt.

2.2 Registri

V osnovni različici je na voljo le pet 24-bitnih registrov:

- A – akumulator, namenjen aritmetičnim operacijam,

- X – indeksni register, namenjen indeksnem naslavljanju,
- L – povezavni register, namenjen shranjevanju programskega števca pri skokih v procedure,
- PC – programski števec, ki vsebuje naslov naslednjega ukaza,
- SW – statusni register, ki shranjuje različne informacije, med drugim tudi rezultat primerjanja (*Condition Code* oziroma CC).

SIC/XE doda še štiri registre:

- B – bazni register, namenjen baznem naslavljanju,
- S – namenjen splošni uporabi,
- T – namenjen splošni uporabi,
- F – 48-bitni akumulator za delo s plavajočo vejico.

2.3 Formati podatkov

Cela števila so 24-bitna in predznačena, za njihov zapis pa se uporablja dvojiški komplement. Znaki uporabljajo 8-bitni ASCII format. Števila s plavajočo vejico so podprta samo v SIC/XE, uporabljajo pa 48 bitov, kjer je 1 bit namenjen za predznak, 11 za eksponent in 36 za mantiso.

2.4 Oblike ukazov

2.4.1 SIC

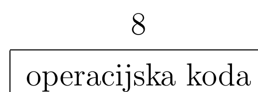
Vsi ukazi za standardno različico SIC imajo enako obliko, prikazano na sliki 2.1. Vsak ukaz je dolg 3 bajte oziroma 24 bitov, vsebuje pa operacijsko kodo ukaza, bit x, ki določa uporabo indeksnega naslavljanja in naslov.

Slika 2.1: Format SIC.

2.4.2 SIC/XE

SIC/XE pozna še dodatne štiri različne oblike oziroma formate ukazov, ki se razlikujejo po dolžini in zastavicah:

- Format 1 je dolg 8 bitov in vsebuje le operacijsko kodo, brez operandov.

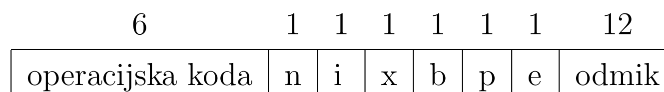


Slika 2.2: Format 1.

- Format 2 je dolg 16 bitov in poleg operacijske kode ukaza vsebuje še številki dveh registrov. Namenjen je predvsem operacijam, ki delujejo nad dvema registroma, v nekaterih primerih pa se *r2* uporablja tudi kot konstanta.

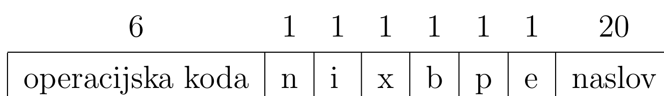
Slika 2.3: Format 2.

- Format 3 je dolg 24 bitov in vsebuje operacijsko kodo, bite n, i, x, b, p in e ter 12-bitni odmik. Bit e ima vedno vrednost 0.



Slika 2.4: Format 3.

- Format 4 je podoben formatu 3, le da namesto odmika vsebuje 20-bitni naslov in ima bit e vedno vrednost 1.



6	1	1	1	1	1	1	20
operacijska koda	n	i	x	b	p	e	naslov

Slika 2.5: Format 4.

2.5 Načini naslavljanja

Načine naslavljanja v sistemu SIC/XE lahko ločimo glede na način izračuna uporabnega naslova (UN) in glede na način uporabe uporabnega naslova.

2.5.1 Izračun uporabnega naslova

Po načinu izračuna uporabnega naslova ločimo med tremi različnimi načini:

Neposredno naslavljanje – bita b in p sta oba nastavljena na 0, UN pa je kar naslov.

Bazno-relativno naslavljanje – bit b je nastavljen na 1, p pa na 0. UN izračunamo tako, da seštejemo odmik in vsebino registra B.

PC-relativno naslavljanje – bit b je nastavljen na 0, p pa na 1. UN izračunamo s seštevanjem odmika in vsebine programskega števca.

Vse izmed naštetih načinov lahko kombiniramo še z **indeksnim naslavljanjem**, ki ga določimo z bitom x. V tem primeru naslovu UN prištejemo še vsebino registra X.

V standardni različici arhitekture SIC lahko uporabimo le neposredno in indeksno naslavljanje.

2.5.2 Način uporabe naslova

Po načinu uporabe izračunanega uporabnega naslova ločimo med tremi različnimi načini:

Enostavno naslavljanje – bita n in i sta oba nastavljena na 0 (format SIC) ali 1 (format 3 ali 4), UN pa je lokacija operanda.

Posredno naslavljanje – bit n je nastavljen na 1, i pa na 0. UN predstavlja naslov, na katerem se nahaja naslov operanda. V zbirni kodi ga označimo s simbolom `@`.

Takojšnje naslavljanje – bit n je nastavljen na 0, i pa na 1. V tem primeru je UN že sam operand. V zbirni kodi pred operandom navedemo simbol `#`.

2.6 Ukazi

V tabeli 2.1 so predstavljeni ukazi, formati, v katerih so na voljo, njihove operacijske kode in učinki. Od naštetih SIC/XE podpira vse, SIC pa le tiste, ki so označeni z formatom SIC.

ukaz	format	koda	učinek
`ADD m`	SIC/3/4	18	A ← (A) + (m..m+2)
`ADDF m`	3/4	58	F ← (F) + (m..m+5)
`ADDR r1,r2`	2	90	r2 ← (r2) + (r1)
`AND m`	SIC/3/4	40	A ← (A) & (m..m+2)
`CLEAR r1`	2	B4	r1 ← 0
`COMP m`	SIC/3/4	28	(A) : (m..m+2)
`COMPF m`	3/4	88	(F) : (m..m+5)
`COMPR r1,r2`	2	A0	(r1) : (r2)
`DIV m`	SIC/3/4	24	A ← (A) / (m..m+2)
`DIVF m`	3/4	64	F ← (F) / (m..m+5)
`DIVR r1,r2`	2	9C	r2 ← (r2) / (r1)
`FIX`	1	C4	A ← (F)
`FLOAT`	1	C0	F ← (A)
`HIO`	1	F4	Zaustavi V/I kanal številka (A)
`J m`	SIC/3/4	3C	PC ← m
`JEQ m`	SIC/3/4	30	PC ← m, če je CC nastavljen na =
`JGT m`	SIC/3/4	34	PC ← m, če je CC nastavljen na >
`JLT m`	SIC/3/4	38	PC ← m, če je CC nastavljen na <

`JSUB m`	SIC/3/4	48	L ← (PC); PC ← m
`LDA m`	SIC/3/4	00	A ← (m..m+2)
`LDB m`	3/4	68	B ← (m..m+2)
`LDCH m`	SIC/3/4	50	A[najbolj desni bajt] ← (m)
`LDF m`	3/4	70	F ← (m..m+5)
`LDL m`	SIC/3/4	08	L ← (m..m+2)
`LDS m`	3/4	6C	S ← (m..m+2)
`LDT m`	3/4	74	T ← (m..m+2)
`LDX m`	SIC/3/4	04	X ← (m..m+2)
`LPS m`	3/4	D0	Naloži status procesorja z naslova m
`MUL m`	SIC/3/4	20	A ← (A) * (m..m+2)
`MULF m`	3/4	60	F ← (F) * (m..m+5)
`MULR r1,r2`	2	98	r2 ← (r2) * (r1)
`NORM`	1	C8	F ← (F)[normaliziran]
`OR m`	SIC/3/4	44	A ← (A) | (m..m+2)
`RD m`	SIC/3/4	D8	A[najbolj desni bajt] ← podatek z naprave (m)
`RMO r1,r2`	2	AC	r2 ← (r1)
`RSUB`	SIC/3/4	4C	PC ← (L)
`SHIFTL r1,n`	2	A4	r1 ← vrednost r1 rotirana v levo za n bitov
`SHIFTR r1,n`	2	A8	r1 ← aritmetični pomik v vrednosti r1 za n bitov
`SIO`	1	F0	Začni V/I kanal številka (A)
`SSK m`	3/4	EC	Zaščitni ključ za naslov m ← (A)
`STA m`	SIC/3/4	0C	m..m+2 ← (A)
`STB m`	3/4	0C	m..m+2 ← (B)
`STCH m`	SIC/3/4	54	m ← (A)[najbolj desni bajt]
`STF m`	3/4	80	m..m+5 ← (F)
`STI m`	3/4	D4	Časovni interval ← (m..m+2)
`STT m`	3/4	84	m..m+2 ← (T)
`STX m`	SIC/3/4	10	m..m+2 ← (X)
`SUB m`	SIC/3/4	1C	A ← (A) - (m..m+2)
`SUBF m`	3/4	5C	F ← (F) - (m..m+5)

`SVC n`	2	B0	Generiraj SVC prekinitev
`TD m`	SIC/3/4	E0	Testiraj napravo (m)
`TIO`	1	F8	Testiraj V/I kanal številka (A)
`TIX m`	SIC/3/4	2C	X ← (X) + 1; (X) : (m..m+2)
`TIXR r1`	2	B8	X ← (X) + 1; (X) : (r1)
`WD m`	SIC/3/4	DC	Naprava (m) ← (A)[najbolj desni bajt]

Tabela 2.1: Ukazi, ki so na voljo v arhitekturi SIC/XE.

Poglavje 3

LLVM

Projekt LLVM je zbirka modularnih orodij in tehnologij, namenjenih razvoju prevajalnikov [16]. Sprva je bila LLVM le kratica, ki je pomenila *low level virtual machine*, torej nizkonivojski navidezni stroj, a se je zaradi obširnosti projekta spremenila – zdaj LLVM predstavlja polno ime projekta. Koda projekta LLVM je na voljo pod odprtokodno licenco Univerzitete Illinois/N-CSA [18].

3.1 Zgodovina

LLVM se je začel kot raziskovalni projekt leta 2000 na Univerzi Illinois z namenom raziskave tehnik statičnega in dinamičnega prevajanja poljubnih programskih jezikov.

Leta 2005 je Apple najel glavnega avtorja LLVM-ja, Chrisa Lattnerja, in ustvaril posebno ekipo za razvoj in integracijo LLVM-ja [1]. S tem je LLVM postal eden izmed ključnih delov Applovih razvojnih orodij.

S časom se je obseg projekta povečal, zato zdaj projekt LLVM vsebuje več podprojektov, med glavnimi pa so: knjižnice LLVM (ki omogočajo optimizacijo in generiranje kode), Clang (prevajalnik za C/C++/Objective-C), LLDB (razhroščevalnik) in libc++ (standardna knjižnica C++).

Slika 3.1: Logotip LLVM. Zmajem pripisujemo moč, hitrost in inteligenco, poleg tega pa se pogosto znajdejo na naslovnicah knjig s tematiko prevajalnikov [19]. (The LLVM Compiler Infrastructure Project [19])

3.2 Uporabniki LLVM-ja

LLVM uporablja več komercialnih in odprtokodnih projektov, pogosto pa se uporablja tudi v akademskih raziskavah [14].

Med komercialnimi uporabniki lahko najdemo Adobe Systems, ki so ga med drugim uporabili za svoj programski jezik Hydra, ki je namenjen procesiranju slik, Sony, ki ga uporablja kot prevajalnik za PlayStation 4, več podjetij, npr. NVIDIA, ga uporablja za implementacijo ogrodja OpenCL. Med večjimi uporabniki in podporniki je tudi Apple, ki ga uporablja v svojih operacijskih sistemih (iOS, macOS, tvOS, watchOS), razvojnih okoljih (Xcode), razvojnih orodjih programskih jezikov Swift, C, C++ in Objective-C ter svojih implementacijah OpenCL, OpenGL in gonilnikih za grafiko.

Obstaja tudi veliko število odprtokodnih projektov, ki med drugim uporabljajo LLVM za statično analizo kode ali pa kot zadnji del prevajalnika.

3.3 Arhitektura

Večina tradicionalnih prevajalnikov je razdeljenih na tri dele: sprednji, vmesni in zadnji del. Tako so deljeni tudi prevajalniki, ki uporabljajo LLVM kot vmesni in zadnji del.

Sprednji del prevede kodo iz izvornega jezika v vmesno kodo, ob tem pa poroča morebitne napake na sintaktični in semantični ravni. Pred prevedbo v vmesno kodo lahko sprednji del kodo najprej prevede tudi v druge oblike, npr. abstraktno sintaksno drevo, in opravi preverjanje napak in optimizacije, ki so specifične za ta jezik.

Vmesni del poskrbi za optimizacije, ki so neodvisne od izvornega jezika in ciljne arhitekture, kot je npr. odstranjevanje nepotrebnih operacij.

Zadnji del je zadolžen za prevedbo iz vmesne kode v zbirni jezik oz. objektno datoteko ciljne arhitekture. Poleg pravilne prevedbe mora poskrbeti, da prevedena koda upošteva posebne lastnosti, ki jih lahko ima ciljna arhitektura.

3.4 Primerjava s klasičnimi prevajalniki

Glavna razlika med LLVM-jem in klasičnim prevajalnikom, kot je npr. GCC, je v modularnosti in vmesni kodi.

LLVM-jeva vmesna koda je natančno določena in vsebuje vse informacije, ki jih LLVM potrebuje za optimiziranje in generiranje kode. To pomeni, da sta sprednji in zadnji del lahko povsem ločena, kar omogoča lažji razvoj posameznih komponent. Razvijalci sprednjega dela prevajalnika tako ne potrebujejo znanja o zadnjem delu, kar olajša razvoj prevajalnikov za nove programske jezike. Podobno velja za razvijalce zadnjih delov, saj ne potrebujejo znanja o sprednjem delu, kar lahko zahteva npr. GCC [2]. Ker je vmesno kodo mogoče brez težav izpisati v datoteko, mora razvijalec prevajalnika le poskrbeti, da lahko svoj jezik prevede v LLVM-jevo vmesno kodo ali prevede iz nje v ustrezno ciljno arhitekturo.

Druga pomembna razlika je, da LLVM ni monoliten prevajalnik, temveč

zbirka knjižnic. Vsaka analiza ali transformacija, ki se izvede nad kodo pri optimizaciji, je samostojna ali pa ima eksplicitno navedene analize oz. transformacije, ki se morajo izvesti pred njo. To omogoča, da si vsako orodje, ki uporablja LLVM, lahko izbere, katere optimizacije naj se nad kodo izvedejo in katere ne, uredi vrstni red njihove izvedbe, lahko pa doda tudi svoje. Poleg olajšanega razvoja orodij na podlagi LLVM-ja modularnost omogoča tudi testiranje po manjših enotah – testiramo lahko le eno transformacijo, ne da bi izvedli še ostale.

3.5 Vmesna koda LLVM

LLVM-jeva vmesna koda (angl. *LLVM Intermediate Representation*, krajše LLVM IR) je ključnega pomena pri modularnosti LLVM-ja. Namenjena je analizam, transformacijam in pretvorbo med poljubnim jezikom in ciljno arhitekturo. Zato mora biti po eni strani čim bolj nizkonivojska, po drugi pa omogočati izražanje višjenivojskih idej.

Najdemo jo v treh oblikah, ki so med seboj ekvivalentne:

- tekstovna oblika (.ll), ki je namenjena predvsem branju in razumevanju,
- binarna oblika (.bc), ki je namenjena shranjevanju v datoteko in procesiranju med programi,
- kot podatkovna struktura v pomnilniku, nad katero se izvajajo optimizacije in analize.

V izseku kode 3.1 je prikazan izpis LLVM-jeve vmesne kode v tekstovni obliki pri prevajanju preproste rekurzivne funkcije za izračun fakultete.

Koda 3.1: Funkcija za izračun fakultete v LLVM-jevi vmesni kodi.

```
define i32 @_Z9factoriali(i32 %n) #0 {
entry:
    %retval = alloca i32, align 4
    %n.addr = alloca i32, align 4
    store i32 %n, i32* %n.addr, align 4
    %0 = load i32, i32* %n.addr, align 4
    %cmp = icmp eq i32 %0, 1
    br i1 %cmp, label %if.then, label %if.else

if.then:
    store i32 1, i32* %retval, align 4
    br label %return

if.else:
    %1 = load i32, i32* %n.addr, align 4
    %2 = load i32, i32* %n.addr, align 4
    %sub = sub nsw i32 %2, 1
    %call = call i32 @_Z9factoriali(i32 %sub)
    %mul = mul nsw i32 %1, %call
    store i32 %mul, i32* %retval, align 4
    br label %return

return:
    %3 = load i32, i32* %retval, align 4
    ret i32 %3
}

define i32 @main() #1 {
entry:
    %retval = alloca i32, align 4
    store i32 0, i32* %retval, align 4
    %call = call i32 @_Z9factoriali(i32 5)
    ret i32 %call
}
```

3.5.1 Spremenljivke

Kot je mogoče opaziti v izseku 3.1, imena lokalnih spremenljivk vsebujejo predpono `%`, imena globalnih pa predpono `@`.

3.5.2 Funkcije

Funkcije so definirane v obliki `define <tip> <ime> (<parametri>) <atributi>`.

3.5.3 Oznake

Znotraj funkcij opazimo tudi oznake blokov, ki jih program lahko uporablja za vejitve, označene pa so z dvopičji.

3.5.4 Tipi

LLVM-jeva vmesna koda ima močan sistem tipov, ki omogoča lažje branje kode in optimizacije, ki sicer ne bi bile mogoče [13].

Med prvorazrednimi tipi (ki so lahko rezultati ukazov) lahko najdemo najbolj pogoste, prikazane v tabeli 3.1, vsebovanih pa je tudi nekaj tipov, ki so posebni za določene arhitekture, kot npr. x86_fp80, ppc_fp128 in x86_mmx.

Tabela 3.1: Nekateri izmed tipov v LLVM-jevi vmesni kodi.

Sintaksa	Primer	Opis
iN	i1, i8, i16, i32...	Celo število širine N
half, float, double, fp128		Števila s plavajočo vejico
label	label %if.then	Oznaka
<tip>*	i32*, float*	Kazalec
<<#elem>x<tipelem>>	<4 x i32>, <4 x i64*>	Vektor
[<#elem>x<tipelem>]	[40 x i32], [3 x [4 x i32]]	Polje
{ <seznam tipov>}	{ i32, i32, i32 }	Struktura

3.5.5 Ukazi

Ukazi v LLVM-jevi vmesni kodi spominjajo na tiste iz zbirnih jezikov RISC procesorjev in so pogosto v obliki s tremi naslovi. Vsi LLVM programi uporabljajo enkratno statično prirejanje oziroma SSA (angl. *static single assignment*), kar pomeni, da je vrednost vsakega navideznega registra definirana le enkrat. Pri tem velja upoštevati, da LLVM dopušča neomejeno število registrov, zapisani pa so v obliki `%n`, torej `%0`, `%1` in tako naprej.

V izseku kode 3.2 je predstavljen prevod stavka `if (n == 1)`. V 2. in 10. vrstici je primer ukaza `load`, ki za operanda dobi tip vrednosti in naslov, na katerem se nahaja, in vrednost dodeli danemu registru (v primeru torej 0 oziroma 3). V 3. vrstici je predstavljen ukaz `icmp eq`, torej preverjanje enakosti, in sicer med registrom 0 in konstanto 1. Drug operand bi lahko bil tudi register, vendar bi v tem primeru morali pred tem vanj naložiti 1. V 4. vrstici lahko opazimo ukaz `br`, ki dobi tri operande: pogoj, oznako, na katero skoči, če je vrednost pogoja enaka `true`, in oznako, na katero skoči, če je vrednost pogoja enaka `false`.

Koda 3.2: Preverjanje vrednosti parametra n pri izračunu fakultete.

```
1      ...
2      %0 = load i32, i32* %n.addr, align 4
3      %cmp = icmp eq i32 %0, 1
4      br i1 %cmp, label %if.then, label %if.else
5  if.then:
6      ...
7  if.else:
8      ...
9  return:
10     %3 = load i32, i32* %retval, align 4
11     ret i32 %3
```

3.5.6 Odvisnost od ciljne arhitekture

Izpis LLVM-jeve vmesne kode ni povsem neodvisen od ciljne arhitekture, saj že vsebuje podatke, ki se med različnimi arhitekturami lahko razlikujejo. To so npr. poravnanost, privzeta širina tipa celih števil, širina kazalcev in morebitni posebni tipi, ki so na voljo le na določenih arhitekturah. Iz tega sledi, da mora te podatke poznati že sprednji del prevajalnika in datoteke ni mogoče najprej prevesti v vmesno kodo, ki cilja na eno arhitekturo, potem pa jo prevesti v strojno kodo druge ciljne arhitekture. Iz tega razloga v okviru te diplomske naloge nismo testirali prevajanja iz več jezikov, temveč le iz C/C++.

3.6 Sprednji deli

3.6.1 Clang

Clang je prevajalnik za jezike, kot so C, C++, Objective C/C++, OpenCL, CUDA in RenderScript [5]. Za zadnji del uporablja LLVM, s tem pa lahko igra tudi vlogo sprednjega dela za naštete jezike v drugih programih, ki prav tako uporabljajo LLVM. Skupaj z drugimi podprojekti LLVM-ja izhaja od verzije 2.6 naprej, pred tem pa je LLVM za prevajanje iz C-ja uporabljal GCC [20] [21].

Med njegovimi glavnimi cilji so:

- hitro prevajanje in nizka poraba pomnilnika,
- boljše poročanje napak,
- boljša integracija z razvojnimi okolji,
- licenca, ki dovoljuje uporabo v komercialnih izdelkih.

3.6.2 Drugi prevajalniki

Poleg Clanga obstaja še veliko število prevajalnikov, ki kot zadnji del uporabljajo LLVM, vendar so od samega projekta ločeni:

- prevajalnik jezika Rust, ki je namenjen pisanju varne sistemske programske opreme [24],
- prevajalnik jezika Julia, ki je namenjen znanstvenemu računanju [10],
- prevajalnik jezika Crystal, ki ima sintakso, podobno jeziku Ruby, vendar ima cilj biti hitrejši [8],
- prevajalnik jezika Swift, ki poskuša nadomestiti jezike, ki so zasnovani na jeziku C, popularen pa je predvsem za razvoj na platformah iOS in macOS [27],
- Kotlin Native, eden izmed uradno podprtih prevajalnikov jezika Kotlin, ki poskuša nadomestiti Javo [12],
- Scala Native, eden izmed uradno podprtih prevajalnikov jezika Scala, ki združuje objektno usmerjeno in funkcijsko programiranje [25].

Vsi ti prevajalniki so relativno novi, kažejo pa, da je LLVM zaradi svojih optimizacij in podprtih arhitektur privlačna izbira za zadnje dele novih prevajalnikov. Večina navedenih omogoča izpis LLVM-jeve vmesne kode, kar pomeni, da bi morda lahko z nekaj popravki, kot omenjeno v podpoglavju 3.5.6, kodo prevedli v zbirno kodo sistema SIC/XE.

3.7 Uporabljena orodja

Ob prevajanju LLVM-ja dobimo več orodij, ki jih lahko uporabimo pri delu z LLVM-jevo vmesno kodo, le nekaj pa je ključnih za razvoj zadnjega dela. To so:

- Clang, s katerim lahko prevedemo izvorno datoteko v vmesno kodo ali pa neposredno v zbirno kodo,
- llc, ki lahko vmesno kodo prevede v zbirno,
- llvm-dis, ki pretvori binarno obliko vmesne kode v tekstovno,
- llvm-as, ki pretvori tekstovno obliko vmesne kode v binarno,
- lli, ki lahko vmesno kodo izvede, če se arhitektura gostitelja ujema z njeno ciljno arhitekturo,
- tblgen, ki iz opisa ciljne arhitekture (datotek .td) ustvari C++ kodo.

Poglavje 4

Dodajanje podpore za arhitekturo SIC/XE v LLVM

V tem poglavju bodo opisani deli implementacije podpore prevajanja za novo ciljno arhitekturo (SIC/XE) v LLVM. Veliko algoritmov in privzetih nastavitev nam LLVM že ponuja v obliki splošnih razredov in vmesnikov, ki pa jih lahko nadomestimo s svojimi implementacijami in uporabo dedovanja. Zaradi kompleksnosti vseh teh razredov pa je pogosto namesto pisanja svojih implementacij od začetka lažje vzeti podporo za že obstoječo arhitekturo in jo prilagoditi za svojo. V našem primeru to ni bilo tako enostavno, saj je sistem SIC/XE zaradi nabora ukazov, uporabe akumulatorja in majhnega števila registrov od že podprtih manjših arhitektur precej drugačen, večje arhitekture pa niso bile privlačne že zaradi količine kode. Zato smo se namesto podobnosti raje osredotočili na arhitekture, ki so bolje dokumentirane, in si pomagali z vodičem za arhitekturo Cpu0 [7], ki pa je podobna arhitekturi MIPS.

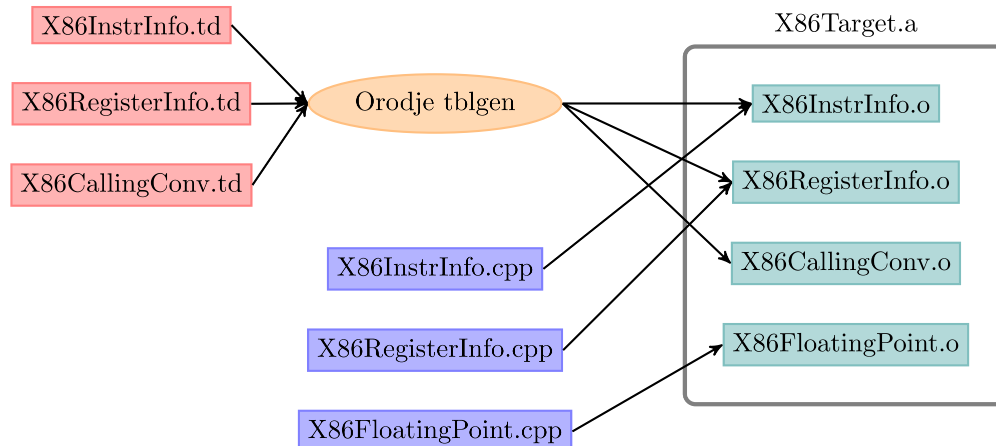
Večina implementacij za posamezne arhitekture se nahaja v podmapah mape `lib/Target`, v primeru te diplomske naloge torej znotraj `lib/Target/SIC`.

4.1 Domensko specifičen jezik TableGen

V tem podpoglavju bo predstavljen domensko specifičen jezik TableGen. Večina LLVM-ja je napisana v programskem jeziku C++, a je za opis arhitektur primernejši jezik TableGen, saj omogoča pisanje fleksibilnih opisov, poleg tega pa zmanjša ponavljanja enakih lastnosti v opisih. Trenutno ga najbolj uporabljata LLVM-jev generator kode in diagnostični del Clanga [15].

Da lahko LLVM-jev generator kode deluje neodvisno od različnih arhitektur, mora poznati osnovne specifikacije ciljne arhitekture, kot so registri, nabor ukazov in njihove omejitve. Ker mu vsaka arhitektura te podatke poda na enak način, so lahko določeni algoritmi (npr. dodeljevanje registrov) deljeni med več arhitekturami, iz česar sledi tudi manjše število podvojene kode in lažje vzdrževanje.

4.1.1 Namen in uporaba



Slika 4.1: Poenostavljen opis podpore za arhitekturo x86.

Za opis lastnosti arhitektur vsaka arhitektura opiše svoje lastnosti v deklarativnem, domensko specifičnem jeziku TableGen. Datoteke s končnico `.td` nato uporabi orodje tblgen in iz njih ustvari C++ kodo, ki jo lahko uporabimo v drugih datotekah. Tako lahko, kot je prikazano na sliki 4.1, dele

arhitekture opišemo z jezikom TableGen, kjer to ni mogoče, pa uporabimo C++.

4.1.2 Opis jezika

Zapisi v TableGenu se delijo na razrede in definicije. Oboji imajo edinstveno ime, seznam nadrazredov in seznam vrednosti, ki jih tblgen uporabi pri generiranju nove kode.

Razredi so abstraktni zapisi, s katerimi lahko sestavljamo druge zapise. Omogočajo nam, da skupne lastnosti več zapisov zapišemo le enkrat. Ustvarimo jih s ključno besedo `class`.

Definicije so konkretni zapisi, ki nimajo več neznanih vrednosti, ustvarimo pa jih s ključno besedo `def`.

Za združevanje več razredov lahko uporabimo tudi `multiclass`, ki omogoča definicijo več objektov naenkrat.

Koda 4.1: Primer definiranja registrov v jeziku TableGen.

```
1  // Iz datoteke include/llvm/Target.td
2  class Register<string n, list<string> altNames = []> {
3      string Namespace = "";
4      string AsmName = n;
5      ...
6      // Zapis, ki se uporablja pri kodiranju registra v ukazih
7      bits<16> HWEncoding = 0;
8  }
9
10 // Iz datoteke lib/Target/SIC/SICRegisterInfo.td
11 class SICReg<bits<16> Enc, string n> : Register<n> {
12     let Namespace = "SIC";
13     let HWEncoding = Enc;
14 }
15
16 def A: SICReg<0, "A">;
17 def X: SICReg<1, "X">;
```

V izseku kode 4.1 je prikazana definicija registrov `A` in `X` prek razreda `SICReg`, ki je podrazred razreda `Register`. Obema registroma nastavimo `Namespace` na niz `SIC`, vrednosti `AsmName` in `HWEncoding` pa sta ustrezni glede na register. Vrednost `AsmName` smo nastavili prek parametrov razreda `Register`, vrednosti `AsmName` in `HWEncoding` pa z uporabo ključne besede `let`. Preden smo lahko vrednosti spremenili z uporabo `let`, smo jih morali v razredu `Register` definirati z zapisom tipa in imena (ter pomožne začetne vrednosti).

Koda 4.2: Razširjene definicije primera 4.1.

```
def A { // Register SICReg
    string Namespace = "SIC";
    string AsmName = "A";
    ...
    bits<16> HWEncoding = 0;
}

def X { // Register SICReg
    string Namespace = "SIC";
    string AsmName = "X";
    ...
    bits<16> HWEncoding = 1;
}
```

Po branju datotek orodje tblgen definicije razširi, kot je prikazano v izseku kode 4.2, potem pa jih lahko izpiše za namene razhroščevanja ali pa iz njih generira C++ kodo.

4.1.3 Ključne datoteke

V primeru 4.1 smo vzeli poenostavljen primer razreda `Register` iz datoteke `include/llvm/Target.td`, razred `SICReg` ter definiciji `A` in `B` pa smo napisali sami. V datotekah `include/llvm/*.td` se torej nahajajo razredi, ki jih tblgen pri generiranju kode išče, saj jih glede na namen različno obrav-

nava. Tako bo npr. pri generiranju informacij o registrih iskal vse definicije, ki implementirajo razred `Register`, pri generiranju informacij o ukazih pa vse, ki implementirajo razred `Instruction` (ki se prav tako nahaja v `Target.td`).

Te ključne razrede in nekaj takih, ki lahko pridejo v pomoč vsem arhitekturam, lahko torej najdemo v naslednjih datotekah mape `include/llvm`:

- `Target.td`,
- `TargetCallingConv.td`,
- `TargetItinerary.td`,
- `TargetSchedule.td`,
- `TargetSelectionDAG.td`.

4.2 Proces generiranja kode

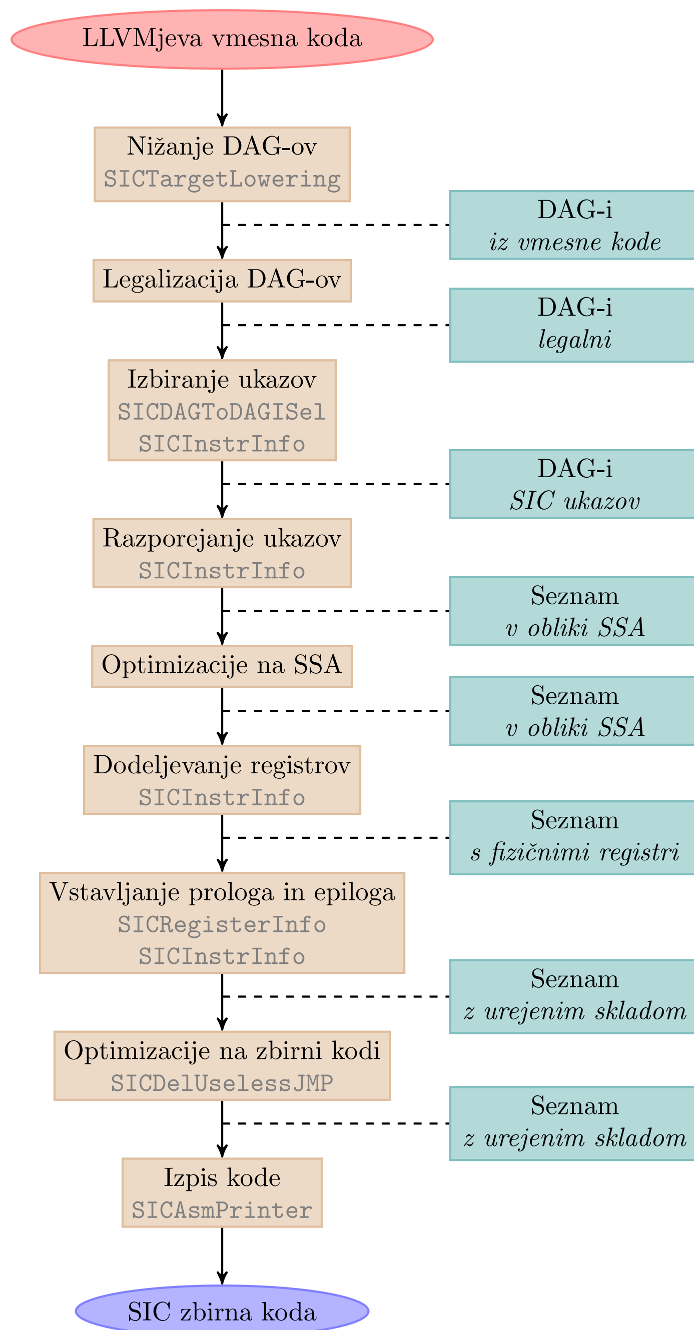
Koda gre v zadnjem delu skozi več faz. Na sliki 4.2 so prikazane najpomembnejše. Pri vsaki je zapisano ime, imena razredov, v katerih se nahaja koda, ki na to fazo vpliva, ter podatkovna struktura, ki je njen rezultat in ob enem vhod v naslednjo fazo.

Nižanje DAG-ov (angl. *DAG Lowering*) – LLVM IR se zniža (spreminja v strukturo, ki je bolj nizkonivojska) v usmerjene aciklične grafe (angl. *Directed Acyclic Graph* oziroma *DAG*).

Legalizacija DAG-ov (angl. *DAG Legalization*) – operacije in tipi, ki jih ciljna arhitektura ne podpira, se ustrezno pretvorijo.

Izbiranje ukazov (angl. *Instruction Selection*) – DAG-i se nadomestijo z ustreznimi ukazi za dano arhitekturo, še vedno pa so v obliki grafov.

Razporejanje ukazov (angl. *Scheduling*) – ukazom se dodeli vrstni red izvajanja, iz grafov pa se spremenijo v seznam.



Slika 4.2: Najpomembnejše faze, skozi katere gre vhodna koda v generatorju kode.

Optimizacije na osnovi SSA (angl. *SSA-based Optimizations*) – izvedejo se optimizacije kode, ki so možne nad ukazi v obliki enkratnega statičnega dodeljevanja (angl. *Static Single Assignment* oziroma *SSA*).

Dodeljevanje registrov (angl. *Register Allocation*) – navideznim registrom se dodeli fizične registre, ki so na voljo v arhitekturi.

Vstavljanje prologa in epiloga (angl. *Prolog/Epilog Code Insertion*) – funkcijam se doda prolog in epilog, torej kodo, ki se izvede pred in po kodi funkcije, ponavadi pa opravlja ustrezne operacije nad skladom.

Optimizacije na strojni kodi (angl. *Late Machine Code Optimizations*) – izvedejo se še zadnje optimizacije, ki se lahko izvedejo na kodi pred izpisom.

Izpis kode (angl. *Code Emission*) – koda se izpiše.

4.3 Splošne informacije o ciljni arhitekturi

4.3.1 Lastnosti arhitekture

Vsaka arhitektura mora narediti svoj razred, ki deduje iz razreda `LLVMTargetMachine` [17]. To je glavni razred, prek katerega lahko v LLVM-ju dostopamo do naše arhitekture. Na njem definiramo funkcije za pridobivanje razreda ukazov, registrov, informacij o klicnih zapisih ipd. Lahko tudi dodamo faze, skozi katere bo vmesna koda (oziroma grafi, seznam ukazov) šla pred izpisom.

Zahtevana je tudi definicija razporeditve podatkov (angl. *Data Layout*), ki LLVM-ju pove, kakšne podatkovne tipe naša arhitektura podpira in kako jih zapiše. Definicija razporeditve se mora ujemati s tisto, ki jo je v vmesno kodo zapisal sprednji del, zato moramo v večini primerov tudi sprednjemu delu dodati podporo za našo arhitekturo. Zapisana je v obliki niza, ki je sestavljen iz več delov, ločenih z pomišljajem. V primeru SIC/XE je to `"E-p:24:32-i8:8:32-i16:16:32-i24:32-n24-S64"`. `E` (namesto `e`) nam

pove, da za zapis bajtov uporabljamo pravilo debelega konca (angl. *big endian*), sledijo pa mu deli, ki so še naprej ločeni z dvopičjem. Črka označuje tip (`p` za kazalec, `i` za celo število), sledi širina tipa, potem pa še poravnanost. Za poravnanost se pri vseh tipih uporablja 32, saj mora biti potenca števila 2, kar pomeni, da število 24 ni primerno. Sledi še `n24`, ki pomeni, da so na voljo 24-bitni registri, ter `S32`, ki označuje, da je sklad poravnan s številom 32.

V primeru naše implementacije lahko najdemo podatke v razredu `SICTargetMachine`, ta pa se nahaja v datotekah `SICTargetMachine.h` in `SICTargetMachine.cpp`

4.3.2 Informacije o podarhitekturah

Vsaka arhitektura lahko vsebuje več podarhitektur, kar omogoča uporabo različnih razredov za različne podarhitekture. Tako npr. MIPS loči med Mips16 in MipsSE. Definiramo lahko tudi več opcij, ki jih uporabnik izbere pri prevajanju. Tako lahko podpremo npr. novejše procesorje in novejše ukaze. Na ta način bi lahko dodali tudi podporo za osnovni sistem SIC, vendar bi bilo zaradi manjšega števila registrov in funkcionalnosti prevajanje za SIC precej težje in ga zato nismo podprli.

4.3.3 Registracija arhitekture

Da lahko druga orodja LLVM našo podporo za arhitekturo SIC/XE najdejo, jo moramo registrirati. Registracija omogoča, da orodja prek imena (npr. kot ga podamo pri prevajanju) pridobijo instanco našega razreda SICTargetMachine. Primer iz naše implementacije je prikazan v izseku kode 4.3.

Koda 4.3: Registracija arhitekture SIC/XE.

```
// SICTargetMachine.cpp
extern "C" void LLVMInitializeSICTarget() {
    RegisterTargetMachine<SICTargetMachine> X(TheSICTarget);
}


// SICTargetInfo.cpp
Target llvm::TheSICTarget;
extern "C" void LLVMInitializeSICTargetInfo() {
    RegisterTarget<Triple::sic,
    /*HasJIT=*/false> X(TheSICTarget, "sic", "SIC");
}
```

4.4 Informacije o registrih

4.4.1 Definicija registrov in razredov registrov

V izseku kode 4.4 je prikazan način definicije registrov za sistem SIC/XE v jeziku TableGen. Na enak način, kot sta definirana registra A in X, so definirani še vsi ostali registri, ki jih vsebuje SIC/XE.

Koda 4.4: Definicija registrov za SIC/XE.

```
class SICReg<bits<16> Enc, string n> : Register<n> {
    let HWEncoding = Enc;
    let Namespace = "SIC";
}

def A: SICReg<0, "A">;
def X: SICReg<1, "X">;
...
```

Poleg registrov potrebujemo tudi definicijo razredov registrov, poimenovano `RegisterClass`. Kot parametre moramo podati štiri argumente: ime prostora, v katerem so definirani (angl. *Namespace*), seznam tipov, poravnanost in seznam registrov.

Naša implementacija razredov registrov je prikazana v izseku kode 4.5. Vsi razredi so si v argumentih precej podobni, razlikujejo pa se v seznamu registrov. Razred `GPROut` vsebuje registre, s katerimi lahko izvajamo ukaze formata 2, razred `CPURegs` pa poleg tega vsebuje še register SW. Vsebino SW je namreč mogoče uporabiti kot operand pri ukazih prek shranjevanja v pomnilnik, ni pa mogoče vanj neposredno shraniti vsebine. Navedeni so še razredi `SR`, `RetRegs` in `ACCs`, ki vsebujejo po le en register, uporabni pa so pri določenih ukazih, ki so omejeni na točno določen register.

Koda 4.5: Razredi registrov za SIC/XE.

```
1  def CPURegs : RegisterClass<"SIC", [i24], 32, (add A, S, T,
2     B, X, L, SW)>;
3
4  def GPROut : RegisterClass<"SIC", [i24], 32, (add (sub
      CPURegs, SW))>;
5
6  def SR      : RegisterClass<"SIC", [i24], 32, (add SW)>;
7
8  def RetRegs : RegisterClass<"SIC", [i24], 32, (add L)>;
9
10 def ACCs    : RegisterClass<"SIC", [i24], 32, (add A)>;
11
12 def RegsNoA : RegisterClass<"SIC", [i24], 32, (sub
13    CPURegs, A)>;
```

Do težave je prišlo pri ukazih formata 2 in akumulatorju, registru A. Ker se rezultat pri več operacijah znajde v registru A, je lahko prišlo do tega, da sta se oba operanda nahajala v razredu `ACCs`, kar pa je predstavljalo težave za dodeljevalnik registrov. Najenostavnejša rešitev je bila, da smo dodali še razred `RegsNoA`, ki ne vsebuje registra A, in tako pri ukazih formata 2

zahteva, da se vsaj eden izmed operandov ne nahaja v tem registru. S tem popravkom je LLVM-jev generator kode pravilno ugotovil, da mora v takih situacijah vstaviti SIC ukaz `RMO` in s tem premakniti vrednost iz registra A v drug register.

Vse te definicije lahko najdemo v datoteki `SICRegisterInfo.td`, ki se ob prevajanju LLVM-ja prevede v datoteko `SICGenRegisterInfo.inc`. Ker so vsi registri definirani v imenskem prostoru `SIC`, se v C++ kodi nanje lahko nanašamo prek tega imenskega prostora – register A zapišemo kot `SIC::A`.

4.4.2 Druge informacije o registrih

Poleg registrov in registrskih razredov je treba določiti še nekaj posebnih funkcionalnosti, ki smo jih določili registrom. To smo storili v datoteki `SICRegisterInfo.cpp`.

Register L smo določili kot register, ki ga mora shraniti klicana funkcija (angl. *callee-saved register*), registra X in PC pa določili kot rezervirana, saj ne želimo, da jima LLVM-jev generator kode sam spreminja vrednost. Oboje smo implementirali v obliki funkcije, in sicer `getCalleeSavedRegs` in `getReservedRegs`.

Implementirati je bilo treba tudi funkcijo `eliminateFrameIndex`, ki ukaz, ki vsebuje le indeks spremenljivke na skladu, pretvori v ukaz z registrom in odmikom. Ker SIC nima registra SP, torej kazalca na sklad, smo v njegovo vlogo postavili register X, saj omogoča indeksno naslavljanje. Tako lahko lokalne spremenljivke iz sklada nalagamo in shranjujemo z le enim ukazom, ki vsebuje register X in odmik spremenljivke od njegove vrednosti (npr. `LDA 4, X`).

4.5 Nižanje in legalizacija DAG-ov

4.5.1 Legalizacija ukazov in tipov

Ker sistem SIC/XE ne podpira vseh ukazov in podatkovnih tipov, ki so na voljo v LLVM-jevi vmesni kodi, smo morali za nekatere določiti, da jih ne podpiramo. To smo med drugim storili za ukaze, ki štejejo bite (`BSWAP`, `CTTZ`, `CTLZ`, `CTPOP`, `BITREVERSE`), za nekatere načine množenja (`MULHS`, `MULHU`, `SMUL_LOHI`, `UMUL_LOHI`), nekatere skoke (`BR_JT`, `BR_CC`), združene ukaze za deljenje in ostanek (`SDIVREM`, `UDIVREM`) ter rotacije bitov v desno (`ROTR`).

Koda 4.6: Navodila generatorju kode za nepodprte ukaze.

```
1 setOperationAction(ISD::SREM, MVT::i24, Expand);
2 setOperationAction(ISD::UREM, MVT::i24, Expand);
```

Kot je prikazano v primeru 4.6, smo to storili tudi za ukaza `SREM` in `UREM` (ostanek pri deljenju z predznačenimi oziroma nepredznačenimi števili). Za ta primer lahko enostavno pogledamo v datoteko `LegalizeDAG.cpp` in vidimo, kaj generator kode stori s takimi ukazi. Iz izseka kode 4.7 lahko razberemo, da generator kode preveri, če je na voljo ukaz `SDIVREM` oziroma `UDIVREM`. Ker ni na voljo, gre naprej v pogoj `else if`, kjer preveri, če je na voljo deljenje. Ker SIC/XE deljenje podpira, lahko torej ostanek nadomesti z odštevanjem, množenjem in deljenjem. Tako smo lahko brez posebnega truda podprli operacijo, ki sicer na SIC/XE ni na voljo, saj je bila pretvorba že na voljo v LLVM-ju. Žal pri vseh primerih ni tako enostavno, saj seveda niso pokriti vsi ukazi in pomanjkljivosti vseh arhitektur, zato morajo arhitekture nekatere primere obravnavati same; namesto `Expand` v primeru 4.6 lahko uporabijo `Custom` in implementirajo funkcijo, ki ta primer pokrije.

Poleg `Expand` in `Custom` je na voljo tudi `Promote`, ki ga lahko uporabimo za tipe, ki jih ne podpiramo neposredno. Tako več arhitektur za tip `i1` uporabi `Promote`, kar pomeni, da se bo tip obravnaval kot večji tip, ki je v arhitekturi podprt, npr. `i8`.

V naši implementaciji podpore za arhitekturo SIC/XE sta legalizacija in nižanje DAG-ov implementirana v datotekah `SICISelLowering.h` in `SICISelLowering.cpp`.

Koda 4.7: Legalizacija ukazov za ostanek.

```
1  EVT VT = Node->getValueType(0);
2  bool isSigned = Node->getOpcode() == ISD::SREM;
3  unsigned DivOpc = isSigned ? ISD::SDIV : ISD::UDIV;
4  unsigned DivRemOpc = isSigned ? ISD::SDIVREM : ISD::UDIVREM;
5  Tmp2 = Node->getOperand(0);
6  Tmp3 = Node->getOperand(1);
7  if (TLI.isOperationLegalOrCustom(DivRemOpc, VT)) {
8    SDVTList VTs = DAG.getVTList(VT, VT);
9    Tmp1 = DAG.getNode(DivRemOpc, dl, VTs, Tmp2, Tmp3)
10             .getValue(1);
11   Results.push_back(Tmp1);
12 } else if (TLI.isOperationLegalOrCustom(DivOpc, VT)) {
13   // X % Y -> X-X/Y*Y
14   Tmp1 = DAG.getNode(DivOpc, dl, VT, Tmp2, Tmp3);
15   Tmp1 = DAG.getNode(ISD::MUL, dl, VT, Tmp1, Tmp3);
16   Tmp1 = DAG.getNode(ISD::SUB, dl, VT, Tmp2, Tmp1);
17   Results.push_back(Tmp1);
18 }
```

4.5.2 Klicne konvencije

Sistem SIC/XE nima točno določene klicne konvencije, zato smo si izbrali svojo, ki je precej podobna MIPS-ovi:

- register X (SP) vedno kaže na najnižjo točko v skladu, saj lahko uporabljamo le pozitivne odmike, kar omogoča manjše število ukazov,
- na vrhu klicnega zapisa se nahajajo vrednosti registrov, ki bi se morda spremenile ob klicanju funkcij (register L), sledijo lokalne spremenljivke, nato shranjene vrednosti ob pomanjkanju registrov, za tem pa

sledi še prostor, namenjen shranjevanju argumentov pred klicem funkcije,

- vse argumente funkciji podamo prek sklada, nobenega prek registrov (zaradi enostavnosti in pomanjkanja registrov),
- funkcija vrne rezultat v registru S.

Podpora za celotno delovanje se nahaja v več datotekah:

- `SICCallingConv.td`,
- `SICFrameLowering.h`,
- `SICFrameLowering.cpp`,
- `SICISelLowering.h`,
- `SICISelLowering.cpp`.

4.6 Definicija izbora ukazov

4.6.1 Formati ukazov

Definirali smo štiri različne formate ukazov, ki so na voljo v sistemu SIC/XE. Vsi so definirani v datoteki `SICInstrFormats.td`. V izseku kode 4.8 je prikazan razred za ukaze formata 2 v TableGenu.

Razred `SICInst` je skupen vsem ukazom, ki se pojavljajo v arhitekturi SIC/XE. Iz podanih parametrov nastavi vse osnovne lastnosti, ki so skupne vsem ukazom. Kot ključne velja izpostaviti:

- `InOperandList` in `OutOperandList`, DAG-a, ki predstavljata vhodne oziroma izhodne operande,
- `AsmString`, niz, ki predstavlja zbirno kodo tega ukaza,

Koda 4.8: Splošen razred za SIC ukaze in razred za format 2. Drugi formati so bili zaradi dolžine izpuščeni.

```
1  class SICInst<bits<8> op, dag outs, dag ins, string asmstr,
2    list<dag> pattern, InstrItinClass itin, Format f> :
       Instruction
3  {
4      field bits<32> Inst;
5      field bits<32> SoftFail = 0;
6      Format Form = f;
7      bits<4> FormBits = Form.Value;
8      bits<6> Opcode = op{7-2};
9
10     let Namespace          = "SIC";
11     let DecoderNamespace = "SIC";
12     let OutOperandList     = outs;
13     let InOperandList      = ins;
14     let Size               = Form.Value;
15     let AsmString          = asmstr;
16     let Pattern            = pattern;
17     let Itinerary          = itin;
18     let TSFlags{3-0}       = FormBits;
19 }
20
21 // SIC format 2
22 class F2<bits<8> op, dag outs, dag ins, string asmstr,
23   list<dag> pattern, InstrItinClass itin>:
24 SICInst<op, outs, ins, asmstr, pattern, itin, Frm2>
25 {
26   bits<4> r1;
27   bits<4> r2;
28
29   let Inst{15-8} = op;
30   let Inst{7-4}  = r1;
31   let Inst{3-0}  = r2;
32 }
```

- `Pattern`, seznam vzorcev, s katerimi bo generator kode ob ujemanju vzorca izbral ta ukaz.

Poleg posredovanja parametrov razredu SICInst razred `F2` definira še dve lastnosti, in sicer registra `r1` in `r2`, ki sta prisotna v ukazih tega formata, poleg tega pa nastavi ustrezne bite v lastnosti `Inst`, da generator kode lahko ukaz zapiše v binarni obliki.

Koda za druge formate je podobna, vendar namesto registrov definira odmik oziroma naslov, poleg tega pa uporablja še dodatne bite, ki določijo način naslavljanja in uporabe končnega naslova. Najdemo jo lahko v `SICInstrFormats.td`.

4.6.2 Ukazi

V izseku kode 4.9 je predstavljen zapis aritmetičnih ukazov `ADDR`, `SUBR`, `MULR` in `DIVR`, ki kot operanda uporabljajo dva registra. Tako kot v 4.8 registra poimenujemo z `r1` in `r2`, označimo pa tudi, da se mora `r1` nahajati v razredu registrov `RegsNoA`, `r2` pa v razredu `CPURegs`. Ta opis lahko najdemo v vrstici 3, kjer sta v grafu podana kot argumenta za parameter `ins` v razredu `F2`, vrstici 4, kjer je v nizu zapisan način izpisa v zbirni kodi, in vrstici 5, kjer sta uporabljena v vzorcu za izbor ukazov. Kot rezultat definiramo še register `r3`, ki pripada razredu registrov `GPROut`, dodeljevalnik registrov pa mora upoštevati, da je ta register enak registru `r2`, saj SIC/XE shrani rezultat v register `r2`. V 8. vrstici še ustrezno nastavimo komutativnost ukazov, kar lahko dodeljevalniku registrov olajša delo.

Koda 4.9: Definicija nekaterih aritmetičnih ukazov formata 2.

```
1 class ArithLogicR<bits<8> op, string instr_asm,
2   SDNode OpNode, RegisterClass RC, RegisterClass RD,
3   bit isComm = 0> :
4 F2<op, (outs GPROut:$r3), (ins RC:$r2, RD:$r1),
5   !strconcat(instr_asm, "\t$r1, $r2"),
6   [(set GPROut:$r3, (OpNode RC:$r2, RD:$r1))], IIAlu>
```

```
7  {
8    let Constraints = "$r2 = $r3";
9    let isCommutable = isComm;
10 }
11
12 def ADD : ArithLogicR<0x90, "ADDR", add, CPURegs, RegsNoA,
       1>;
13 def SUB : ArithLogicR<0x94, "SUBR", sub, CPURegs, RegsNoA,
       0>;
14 def MUL : ArithLogicR<0x98, "MULR", mul, CPURegs, RegsNoA,
       1>;
15 def DIV : ArithLogicR<0x9C, "DIVR", sdiv, CPURegs, RegsNoA,
       0>;
```

Kot lahko vidimo, nam TableGen res omogoča, da nadvse zmanjšamo količino kode. Priročno je tudi, da se informacije za več faz generiranja kode nahajajo na enem mestu, saj smo definirali vzorec za fazo izbiranja ukazov, operande in razrede registrov za analizo aktivnosti (angl. *liveness analysis*) in dodeljevalnik registrov in vzorec za izpis ukaza.

Na podoben način so definirani tudi ostali ukazi, vsi ti opisi pa se nahajajo v datoteki `SICInstrInfo.td`.

4.6.3 Dodatne funkcije

V razredu `SICInstrInfo` smo implementirali še:

- funkciji `storeRegToStack` in `loadRegFromStack`, ki vrneta ukaze za shranjevanje oziroma nalaganje registrov na sklad, ko jih zmanjka,

- funkcijo `adjustStackPtr`, ki zviša oziroma zniža kazalec na sklad (register X) za določeno velikost,

- funkcijo `copyPhysReg`, ki vrne ukaz za premik vrednosti iz enega registra v drugega, torej `RMO`, razen v primeru kopiranja iz registra SW. Vrednost registra SW je potrebna za nekatere operacije primerjanja, saj po primerjanju dveh števil vsebuje rezultat (večje, manjše, je enako).

Ker RMO ne podpira kopiranja iz SW, smo dodali svoj psevdo ukaz, ki je sestavljen iz dveh ukazov: +STSW MEMREG in +LD$r1 MEMREG. S tem ukazom vrednost registra SW najprej shranimo v pomnilnik na mesto, označeno z MEMREG, potem pa jo prek njega naložimo v poljuben register r1, torej destinacijo premika vrednosti.

Implementacije teh funkcij se nahajajo v datoteki SICInstrInfo.cpp.

4.7 Izbiranje ukazov

Večina ukazov je izbranih s pomočjo vzorcev, definiranih ob definiciji ukaza, kot v izseku kode 4.9. Za nekatere ukaze pa smo v SICInstrInfo.td definirali še dodatne vzorce.

4.7.1 Nalaganje konstant v registre

Koda 4.10: Vzorci za neposredno nalaganje števil v registre.

```
1  // Vzorci na listih grafa, s predikati
2  def immZExt12 : PatLeaf<(imm),
3    [{ return isUInt<12>(N->getZExtValue()); }]>;
4  def immZExt20 : PatLeaf<(imm),
5    [{ return isUInt<20>(N->getZExtValue()); }]>;
6  def immSExt24 : PatLeaf<(imm),
7    [{ return isInt<24>(N->getSExtValue()); }]>;
8
9  // Vzorci za nalaganje konstant, Pat<vzorec, rezultat>
10 def : Pat<(i24 immZExt12:$in), (LDi imm:$in)>;
11 def : Pat<(i24 immZExt20:$in), (LDi4 imm:$in)>;
12 def : Pat<(i24 immSExt24:$in), (LDiL imm:$in)>;
```

V izseku kode 4.10 so prikazani vzorci, ki smo jih definirali za nalaganje števil v registre. Med vzorci so razlike v razponu števil in vrnjenih ukazih, prikazane pa so v tabeli 4.1.

LDi in LDi4 sta najbolj smiselna za nalaganje števil v registre, saj zavzameta le en ukaz dolžine treh oziroma štirih bajtov, vendar ne podpirata

Razpon števila	Ime definicije	Format	Velikost	Primer ukaza
0 ... 4095	LDi	3	3	LDA #42
0 ... 1048575	LDi4	4	4	+LDA #42
-16.777.215 ... 16777214	LDiL	3	3 (+ 3)	LDA =42

Tabela 4.1: Izbira ukaza glede na velikost števila.

negativnih števil. Zato smo dodali še `LDiL`, ki uporablja literale, zmožnost zbirnika, da številu dodeli mesto v pomnilniku, iz katerega lahko SIC/XE to število naloži v register. To pomeni, da poleg ukaza, ki vzame tri bajte, zavzame tri še število, ki je shranjeno v pomnilniku, vendar v primeru ponovne uporabe istega števila v isti funkciji to ni potrebno, saj lahko zbirnik ponovno uporabi isti naslov. Tako `LDiL` zavzame šest (oziroma včasih tri) bajtov v pomnilniku in en ukaz pri izvajanju, kar pa je boljše od alternative, da z `LDi` ali `LDi4` naložimo pozitivni del števila, ki ga nato odštejemo od števila 0, saj bi ta rešitev zavzela vsaj šest bajtov v pomnilniku in dva ukaza pri izvajanju.

4.7.2 Bitna operacija XOR

Nabor ukazov sistema SIC/XE ne vključuje operacije bitne XOR ali negacije, kar pomeni, da ga moramo nadomestiti z drugimi ukazi. Zgledovali smo se po enostavni definiciji te operacije: ekskluzivni or vrne 1, ko je vsaj en operand enak 1, vendar ne, ko imata vrednost 1 oba. Zato smo ga nadomestili z operacijami OR, AND in SUB, kot je prikazano v izseku kode 4.11.

Koda 4.11: Vzorec za nadomestitev operacije XOR.

```
1 def XOR: Pat<(xor ACCs: $lhs, RegsNoA: $rhs),
2               (SUB (OR ACCs: $lhs, RegsNoA: $rhs),
3                    (AND ACCs: $lhs, RegsNoA: $rhs))>;
```

4.7.3 Bitni premiki

LLVM pozna več variacij bitnih premikov (angl. *bitwise shifts*), ki so predstavljene v tabeli 4.2. V zadnjem stolpcu je prikazan ekvivalenten SIC/XE ukaz. Rotacijo v desno smo že nadomestili v legalizaciji, kot omenjeno v podpoglavju 4.5.1, za premik v levo in logični premik v desno pa smo dodali vzorce, ki te ukaze nadomestijo s kombinacijo ukazov `SHIFTL`, `SHIFTR` in `AND`.

Ime	TableGen DAG	Novi biti	SIC/XE
Premik v levo	shl	0	?
Rotacija v levo	rotl	Enaki odstranjenim	SHIFTL
Logični premik v desno	srl	0	?
Aritmetični premik v desno	sra	Enaki predznaku	SHIFTR
Rotacija v desno	rotr	Enaki odstranjenim	?

Tabela 4.2: Različne oblike bitnih premikov.

Poleg nadomeščanja ukazov z drugimi je pri sistemu SIC/XE še dodatna omejitev – zamiki so mogoči le s števili, ki so prevajalniku znana že ob prevajanju ter so večja od 1 in manjša od 17, saj so pri ukazih `SHIFTL r1, n` in `SHIFTR r1, n` za število `n` na voljo le štirje biti.

4.8 Izpis zbirne kode

Za izpis kode smo v datotekah `SICAsmPrinter.h` in `SICAsmPrinter.cpp` dodali razred `SICAsmPrinter`, podrazred razreda `AsmPrinter`. V njem smo implementirali naslednje funkcije:

- `EmitInstruction`, ki izpiše dan ukaz,
- `EmitFunctionEntryLabel`, ki izpiše oznako na začetku funkcije,
- `EmitStartOfAsmFile`, s katero na vrh datoteke dodamo dodatne ukaze: `+LDX #0xFFFFC`, da v X naložimo začetno vrednost, `JSUB main`, da

skočimo na začetek kode, in `halt J halt`, da se simulator po izvajanju funkcije `main` ustavi, ker zazna neskončno zanko,

- `EmitEndOfAsmFile`, s katero na konec datoteke dodamo `MEMREG RESW 1`, ki nam omogoča uporabo nekaterih psevdo ukazov, kot je `STSW`, omenjen v podpoglavju 4.6.3,
- `EmitJumpTableInfo`, ki se kliče po izpisu vsake funkcije in je ustrezno mesto za izpis direktive `EQU *`, s katero lahko zbirniku označimo konec funkcije, in `LTORG`, ki zbirniku pove, naj sem izpiše literale, ki smo jih omenili v podpoglavju 4.7.1.

4.9 Integracija v LLVM

4.9.1 Registracija ciljne trojke

Ciljna trojka (angl. *target triple*) je kombinacija arhitekture, proizvajalca in operacijskega sistema, vsebuje pa lahko še okolje. Z njo prevajalniku podamo informacije o željeni arhitekturi. Ker smo razvijali na Linuxu, smo med razvoj uporabljali trojko sic-unknown-linux-gnu.

Da LLVM sistem SIC/XE lahko najde, smo ga morali dodati na dve mesti v LLVM-ju, in sicer v datoteko `llvm/ADT/Triple.h` in `lib/Support/-Triple.cpp`.

4.9.2 Integracija s Clangom

Posebno podporo za SIC/XE smo dodali tudi v Clang, saj mora Clang imeti informacije o ciljni razporeditvi podatkov in širinah tipov. To smo storili v datoteki `tools/clang/lib/Basic/Targets.cpp`, kjer smo dodali razred `SICTargetInfo`, ki nastavi razporeditev podatkov tako, kot je bilo omenjeno v podpoglavju 4.3.1, širino kazalcev in celih števil pa na 24. Poleg tega smo podporo za 24-bitna števila dodali tudi na več mest, povezanih z LLVM-jevo vmesno kodo.

S to integracijo Clang generira vmesno kodo, ki že vsebuje 24-bitna cela števila namesto 32-bitnih, lahko pa tudi prevaja neposredno v zbirno kodo sistema SIC/XE.

4.10 Omejitve dodane podpore za SIC/XE

Ker cilj te naloge ni bila popolna podpora sistema SIC/XE, ki nima tako bogatega nabora ukazov kot nekatere druge arhitekture in vsebuje nekaj posebnosti, kot so 24-bitni registri in 48-bitna števila s plavajočo vejico, smo se zadovoljili že z nepopolno podporo, ki pa je kljub temu zmožna prevajanja marsikaterih programov, kot je prikazano v poglavju 5.

Nekatere omejitve naše podpore so:

- ni podpore za standardno knjižnico, ker bi zahtevala posebno implementacijo za SIC/XE,
- ni podpore za števila s plavajočo vejico, ker SIC uporablja 48-bitna števila in vsebuje le en register za delo z njimi,
- ni pravega prevajanja v objektno datoteko, kot je definirana za SIC/XE, saj se razlikuje od standardnih, kot je ELF,
- ni celotne podpore za nekatere LLVM IR ukaze, ki nastanejo redkeje ali so rezultat optimizacij,
- ni podpore za SIC/XE zbirno kodo v C-ju,
- izhodna koda ni vedno najbolj optimalna.

Poglavje 5

Evaluacija

V tem poglavju je predstavljeno prevajanje C programov v zbirno kodo SIC/XE z razširjenim LLVM-jem. Predstavljeni C programi so prikazani v izvorni kodi, LLVM-jevi vmesni kodi in zbirni kodi SIC/XE. Bralec naj razume, da narava dela narekuje prikaz rezultatov v obliki zbirne kode, čeprav je ta morda težko berljiva.

5.1 Prevajanje LLVM-ja

Za prevajanje LLVM-ja se uporablja orodje CMake [6], uporabili pa smo tudi že prevedeno različico prevajalnika Clang [5], povezovalnik gold [4] in orodje ninja [23].

V skripti 5.1 so prikazani ukazi, ki smo jih uporabili za prevajanje iz korenske mape izvorne kode, `sic-llvm`. V 6. vrstici je za še hitrejše prevajanje in samo delovanje mogoče namesto `Debug` izbrati `Release`, vendar izgubimo možnost razhroščevanja. V 11. vrstici je določena omejitev maksimalnega števila paralelnih izvajanj povezovalnikov, saj lahko posamezen proces zavzame tudi do 4 GiB pomnilnika in ne želimo prekoračiti pomnilnika, ki ga imamo na razpolago. V 13. vrstici je ukaz `ninja` izveden brez dodatnih opcij in zastavic, saj že privzeto uporabi vsa procesorska jedra, ki so na voljo, poleg tega pa prevede vsa LLVM-jeva orodja. Če si želimo le prevajanja orodja

llc in clang, lahko to storimo z ukazom `ninja llc clang`.

Po izvajanju skripte 5.1 so v mapi `build/bin` na voljo programi `clang`, `llc` in `llvm-dis`, s katerimi lahko prevajamo C kodo v zbirno kodo SIC/XE.

Koda 5.1: Ukazi za prevod LLVM-ja s podporo za SIC/XE.

```
cd ..
mkdir build
cd build
cmake
  -G Ninja
  -DCMAKE_BUILD_TYPE=Debug
  -DLLVM_TARGETS_TO_BUILD="SIC"
  -DCMAKE_C_COMPILER=clang
  -DCMAKE_CXX_COMPILER=clang++
  -DLLVM_USE_LINKER=gold
  -DLLVM_PARALLEL_LINK_JOBS=2
  ./../sic-llvm/
ninja
```

5.2 Primer rekurzivnega programa

V tem podpoglavju je predstavljeno prevajanje programa, ki z uporabo rekurzije izračuna deseto Fibonaccijevo število.

Koda 5.2: Primer programa v C-ju, ki izračuna deseto Fibonaccijevo število.

```
int fib(int n) {
    if (n <= 2) return 1;
    else return fib(n - 1) + fib(n - 2);
}

int main() {
    return fib(10);
}
```

5.2.1 Brez optimizacij

Izvorno kodo 5.2 smo najprej z ukazom `./clang -target sic-unknown-linux-gnu -S -emit-llvm fib.c -o fib.ll` prevedli v LLVM-jevo vmesno kodo 5.3.

Koda 5.3: Program v LLVM-jevi vmesni kodi, ki izračuna deseto Fibonaccijevo število. Za namen manjše porabe prostora smo nekatere metapodatke, ki niso ključnega pomena, izpustili.

```
1  ; ModuleID = 'fib.c'
2  source_filename = "fib.c"
3  target datalayout = "E-p:24:32-i8:8:32-i16:16:32-i24:32-n24-
      S32"
4  target triple = "sic-unknown-linux-gnu"
5
6  ; Function Attrs: nounwind
7  define i24 @fib(i24 %n) #0 {
8  entry:
9    %retval = alloca i24, align 4
10   %n.addr = alloca i24, align 4
11   store i24 %n, i24* %n.addr, align 4
12   %0 = load i24, i24* %n.addr, align 4
13   %cmp = icmp sle i24 %0, 2
14   br i1 %cmp, label %if.then, label %if.else
15
16 if.then:  ; preds = %entry
17   store i24 1, i24* %retval, align 4
18   br label %return
19
20 if.else:  ; preds = %entry
21   %1 = load i24, i24* %n.addr, align 4
22   %sub = sub nsw i24 %1, 1
23   %call = call i24 @fib(i24 %sub)
24   %2 = load i24, i24* %n.addr, align 4
25   %sub1 = sub nsw i24 %2, 2
26   %call2 = call i24 @fib(i24 %sub1)
27   %add = add nsw i24 %call, %call2
```

```
28    store i24 %add, i24* %retval, align 4
29    br label %return
30
31  return:    ; preds = %if.else, %if.then
32    %3 = load i24, i24* %retval, align 4
33    ret i24 %3
34  }
35
36  ; Function Attrs: nounwind
37  define i24 @main() #0 {
38  entry:
39    %retval = alloca i24, align 4
40    store i24 0, i24* %retval, align 4
41    %call = call i24 @fib(i24 10)
42    ret i24 %call
43  }
```

V vmesni kodi 5.3 lahko najprej v vrsticah 1–4 zasledimo metapodatke o izvorni datoteki in ciljni arhitekturi, v vrsticah 7–34 pa sledi prevedena rekurzivna funkcija `fib`. Razdeljena je na štiri bloke:

- `entry`, kjer se naložijo vrednosti iz podanih parametrov in se preverja vrednost pogoja if stavka,
- `if.then`, kjer se v rezultat shrani vrednost `1`,
- `if.else`, kjer se nahaja izračun argumentov in rekurzivna klica funkcije `fib`,
- `return`, kjer se vrne rezultat funkcije.

V vrsticah 37–43 je izpisana prevedena funkcija `main`, ki le pokliče funkcijo `fib` z vrednostjo `10` in vrne rezultat.

Poleg zapisane razporeditve podatkov za arhitekturo SIC/XE lahko opazimo tudi, da se je tip `int` prevedel v `i24`, za kar smo poskrbeli z integracijo podpore za SIC/XE v prevajalnik Clang in dodajanjem tipa za 24-bitna cela števila v LLVM-jevo vmesno kodo.

Iz vmesne kode 5.3 smo program nato z ukazom `./llc -O0 fib.ll -o fib_llc.sic.asm` prevedli v zbirno kodo 5.4. Opcija `-O0` prevajalniku pove, naj ne uporablja optimizacij.

Koda 5.4: Program v zbirni kodi SIC/XE, ki izračuna deseto Fibonaccijevo število. Za lažje razumevanje so dodani komentarji.

```
1           .text
2           +LDX    #0xFFFFC
3           JSUB    main
4  halt     J       halt
5           .file   "fib.ll"
6  fib                          . @fib
7  . BB#0:                      . %entry
8           LDT     #32         . Prolog - znizanje X (SP)
9           SUBR    T, X
10          STL     28, X       . Shrani L (stari PC)
11          LDA     32, X       . Shrani n, podan kot argument
12          RMO     A, S        . v lokalno spremenljivko
13          STA     20, X
14          LDT     #2          . Primerjaj n z 2
15          COMPR   A, T
16          STS     16, X
17          JGT     _BB0_2      . Ce je n > 2, skoci v else
18 . BB#1:                      . %if.then
19          LDA     #1          . Shrani 1 v spremenljivko
20          STA     24, X       . retval, ki hrani rezultat
21          J       _BB0_3      . Skoci v return
22 _BB0_2                       . %if.else
23          LDA     20, X       . Izracunaj n - 1
24          ADD     =-1
25          RMO     X, S        . Shrani n - 1 kot argument
26          +STS    MEMREG
27          +STA    @MEMREG
28          JSUB    fib         . Poklici fib
29          LDA     20, X       . Izracunaj n - 2
30          ADD     =-2
31          RMO     X, L        . Shrani n - 2 kot argument
```

```
32          +STL    MEMREG
33          +STA    @MEMREG
34          STS     12, X       . Shrani rezultat fib(n-1)
35          JSUB    fib         . Poklici fib
36          LDA     12, X       . Sestej rezultata klicov
37          ADDR    S, A
38          STA     24, X       . Shrani rezultat v retval
39  _BB0_3                      . %return
40          LDS     24, X       . Nalozi retval v register S
41          LDL     28, X       . Nalozi L (stari PC)
42          LDT     #32         . Epilog - zvisanje X (SP)
43          ADDR    T, X
44          RSUB                . Vrni se
45  _func_end0
46          EQU     *           . Shrani naslov konca funkcije
47          LTORG               . Izpisi literale
48
49  main                        . @main
50  . BB#0:                     . %entry
51          LDT     #16         . Prolog - znizanje X (SP)
52          SUBR    T, X
53          STL     12, X       . Shramo L (stari PC)
54          CLEAR   A
55          STA     8, X
56          RMO     X, S        . Nalozi 10 kot argument
57          LDA     #10
58          +STS    MEMREG
59          +STA    @MEMREG
60          JSUB    fib         . Poklici fib
61          LDL     12, X       . Nalozi L (stari PC)
62          LDT     #16         . Epilog - zvisanje X (SP)
63          ADDR    T, X
64          RSUB                . Vrni se
65  _func_end1
66          EQU     *           . Shrani naslov konca funkcije
67          LTORG               . Izpisi literale
68
69  MEMREG  RESW    1           . Mesto za operacije nad spominom
```

Zbirno kodo 5.4 je veliko lažje primerjati z vmesno kodo 5.3 kot pa z izvorno kodo 5.2. Večina ukazov iz vmesne kode se namreč pretvori v enega ali več ukazov v končni zbirni kodi, podobnost pa je opaziti tudi pri organizaciji kode, saj je koda enako ločena na bloke.

Koda 5.4 ni optimizirana, saj je s tem lazje pokazati preslikave, do katerih pride med prevajanjem.

5.2.2 Z optimizacijami

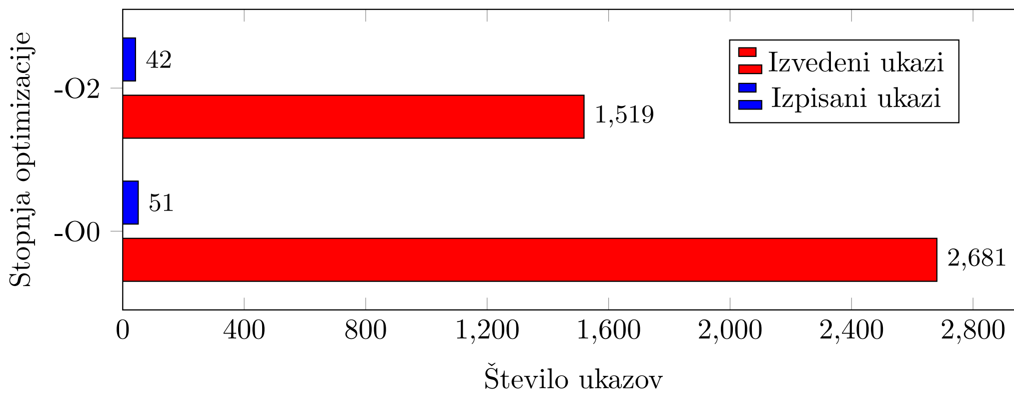
Tokrat smo uporabili stopnjo optimizacije `-O2`, ki smo jo navedli pri prevajanju s prevajalnikoma Clang in llc. Poskusili smo tudi s stopnjama `-O1` in `-O3`, a smo prišli do enakih rezultatov kot pri stopnji `-O2`. Optimizacije so zmanjšale uporabo pomnilnika, poleg tega pa so enega izmed rekurzivnih klicev nadomestile z zanko. Ker je koda ponekod podobna tisti brez optimizacij, smo v zbirni kodi 5.5 izpostavili le funkcijo `fib`, saj vsebuje optimizacijo rekurzije.

Koda 5.5: Funkcija v LLVM-jevi vmesni kodi, ki z uporabo optimizacij izračuna dano Fibonaccijevo število.

```
1  fib                             . @fib
2  . BB#0:                         . %entry
3          LDT     #24
4          SUBR    T, X
5          STL     20, X           . 3-byte Folded Spill
6          LDT     #1
7          LDA     24, X
8          LDS     #3
9          COMPR   A, S
10         JLT     _BB0_3
11 . BB#1:                         . %if.else.preheader
12         LDT     #1
13 _BB0_2                          . =>Loop Header: Depth=1
14         STA     16, X           . A vsebuje n drugega klica
15         STT     12, X           . T predstavlja akumulator
```

```
16          LDA     16, X       . Izracunaj n - 1
17          ADD     =-1
18          +STX    MEMREG      . Shrani n - 1 kot argument
19          +STA    @MEMREG
20          JSUB    fib         . Poklici fib
21          LDA     16, X
22          LDT     12, X       . Rezultat klica pristej
23          ADDR    S, T        . akumualtorju
24          ADD     =-2         . Zmanjsaj n
25          LDS     #2
26          COMPR   A, S        . Primerjaj n z 2
27          JGT     _BB0_2      . Skoci nazaj, ce je n > 2
28 _BB0_3                       . %return
29          RMO     T, S        . Premakni akumualtor v rezultat
30          LDL     20, X
31          LDT     #24
32          ADDR    T, X
33          RSUB
```

5.2.3 Primerjava



Slika 5.1: Število izpisanih in izvedenih ukazov za program 5.2 glede na stopnjo optimizacije.

S pomočjo orodja SicTools smo izvedli programa ter prešteli število izpi-

sanih in izvedenih ukazov. Rezultate štetja ukazov za kodo 5.4, ki z opcijo `-O0` ni uporabljala optimizacij, in kodo 5.5, ki z opcijo `-O2` te je uporabljala, smo predstavili na grafu 5.1.

V preštetih izpisanih ukazih razlika ni tako velika, saj nadomestitev nalaganja argumentov in klica funkcije z zanko ni prihranila največ izpisanih ukazov, je pa zato razlika očitna v številu izvedenih ukazov. Z manjšim številom klicev namreč prihranimo na več ukazih, saj poleg nekaterih delov funkcije ni potrebno dodatno izvajanje prologa in epiloga funkcije.

5.3 Primer iterativnega programa

V tem podpoglavju je predstavljeno prevajanje programa, ki deseto Fibonaccijevo število namesto uporabe rekurzije izračuna na iterativen način.

Koda 5.6: Primer programa v C-ju, ki na iterativen način izračuna deseto Fibonaccijevo število.

```
1  int main() {
2      int n = 10;
3      int prev1 = 1;
4      int prev2 = 1;
5      int result = 1;
6      for (int i = 3; i <= n; i++) {
7          result = prev1 + prev2;
8          int tmp = prev1;
9          prev1 = result;
10         prev2 = tmp;
11     }
12     return result;
13 }
```

5.3.1 Brez optimizacij

Tokrat smo program prevedli neposredno v zbirno kodo z uporabo ukaza `./clang -target sic-unknown-linux-gnu -S fib_iter.c -o fib_iter_`

clang.sic.asm.

Koda 5.7: Program v zbirni kodi SIC/XE, ki na iterativen način izračuna deseto Fibonaccijevo število.

```
1           .text
2           +LDX     #0xFFFFC
3           JSUB     main
4  halt     J        halt
5           .file    "fib_iter.c"
6  main                            . @main
7  . BB#0:                         . %entry
8           LDT      #32
9           SUBR     T, X
10          CLEAR    A
11          STA      28, X
12          LDA      #10
13          STA      24, X
14          LDA      #1
15          STA      20, X
16          STA      16, X
17          STA      12, X
18          LDA      #3
19          STA      8, X
20 _BB0_1                          . %for.cond
21                                 . =>Loop Header: Depth=1
22          LDA      8, X
23          LDS      24, X
24          COMPR    A, S
25          JGT      _BB0_4
26 . BB#2:                         . %for.body
27          LDA      20, X
28          LDS      16, X
29          ADDR     S, A
30          STA      12, X
31          LDA      20, X
32          STA      4, X
33          LDA      12, X
34          STA      20, X
```

```
35          LDA     4, X
36          STA     16, X
37  . BB#3:                         . %for.inc
38          LDA     8, X
39          ADD     #1
40          STA     8, X
41          J       _BB0_1
42  _BB0_4                          . %for.end
43          LDS     12, X
44          LDT     #32
45          ADDR    T, X
46          RSUB
47  _func_end0
48          EQU     *
49          LTORG
50
51  MEMREG  RESW    1
```

5.3.2 Z optimizacijami

Za prevajanje v zbirno kodo z uporabo optimizacij smo uporabili ukaz `./clang -O2 -target sic-unknown-linux-gnu -S fib_iter.c -o fib_iter_clang _O2.sic.asm`. Tako kot v rekurzivnem programu smo do enakih rezultatov prišli tudi z uporabo stopenj optimizacije `-O1` in `-O3`.

Ključna je 8. vrstica, saj kot rezultat funkcije kar neposredno naloži število 55. LLVM je torej kodo optimiziral tako, da je celoten algoritem za izračun števila nadomestil kar z rezultatom.

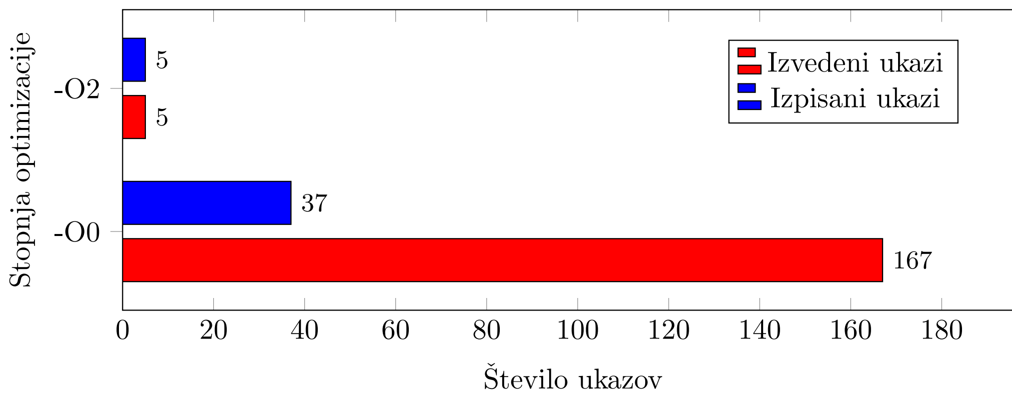
Koda 5.8: Program v zbirni kodi SIC/XE, ki z uporabo optimizacij na iterativen način izračuna deseto Fibonaccijevo število.

```
1           .text
2           +LDX    #0xFFFFC
3           JSUB    main
4   halt    J       halt
5           .file   "fib_iter.c"
6   main                                        . @main
```

```
7   . BB#0:                                          . %entry
8            LDS      #55
9            RSUB
10  _func_end0
11           EQU      *
12           LTORG
13
14  MEMREG   RESW     1
```

5.3.3 Primerjava

Čeprav je štetje ukazov v optimizirani zbirni kodi 5.8 trivialno, smo oba programa izvedli ter prešteli število izpisanih in izvedenih ukazov z uporabo orodja SicTools.



Slika 5.2: Število izpisanih in izvedenih ukazov za program 5.6 glede na stopnjo optimizacije.

Rezultati, prikazani na grafu 5.2, kažejo, kako učinkovite so lahko optimizacije v LLVM-ju. S pomočjo optimizacij tako lahko prevajalnik že izračuna marsikatero vrednost, ki jo sami ob pisanju morda spregledamo. So tudi dober razlog, da se odločimo za dodajanje podpore željene arhitekture v LLVM oziroma GCC namesto pisanja svojega prevajalnika, saj se ni lahko kosati z vsem delom, ki je že šlo v razvoj teh prevajalnikov.

5.4 Primer programa s tabelo in kazalci

V tem podpoglavju bo prikazan še program, ki uredi tabelo z desetimi števili in pri tem uporablja funkcijo `swap`, ki zamenja dve vrednosti z uporabo kazalcev.

Zaradi dolžine kode je v tem podpoglavju izpisana le izvorna in optimizirana zbirna koda. Za prevajanje smo uporabili podoben ukaz kot v podpoglavju 5.3.2.

Koda 5.9: Primer programa v C-ju, ki uredi tabelo z desetimi števili.

```
1  #define LEN 10
2  int arr[] = {7, 8, 9, 1, 2, 3, 5, 4, 0, 6};
3
4  void swap(int* a, int* b) {
5      int temp = *a;
6      *a = *b;
7      *b = temp;
8  }
9
10 int main() {
11     for (int i = 0; i < LEN; i++) {
12         for (int j = i + 1; j < LEN; j++) {
13             if (arr[i] > arr[j]) {
14                 int* a = &arr[i];
15                 int* b = &arr[j];
16                 swap(a, b);
17             }
18         }
19     }
20     return arr[LEN - 1];
21 }
```

Koda 5.10: Program v zbirni kodi SIC/XE, ki z uporabo optimizacij uredi tabelo z desetimi števili.

```
1          .text
2          +LDX     #0xFFFFC
```

```
3              JSUB      main
4    halt      J         halt
5              .file     "selection_sort_02.c"
6    swap                                 . @swap
7    . BB#0:                              . %entry
8              LDS       4, X
9              LDT       0, X
10             +STT      MEMREG
11             +LDA      @MEMREG
12             +STS      MEMREG
13             +LDB      @MEMREG
14             +STT      MEMREG
15             +STB      @MEMREG
16             +STS      MEMREG
17             +STA      @MEMREG
18             RSUB
19   _func_end0
20             EQU       *
21             LTORG
22
23   main                                 . @main
24   . BB#0:                              . %entry
25             LDT       #32
26             SUBR      T, X
27             STL       28, X            . 3-byte Folded Spill
28             CLEAR     B
29             +LDA      #arr
30             STA       0, X             . 3-byte Folded Spill
31             ADD       #4
32             STA       4, X             . 3-byte Folded Spill
33   _BB1_1                               . %for.body
34                                        . =>Loop Header: Depth=1
35             RMO       B, A
36             ADD       #1
37             LDS       #9
38             STA       8, X             . 3-byte Folded Spill
39             COMPR     A, S
40             JGT       _BB1_7
```

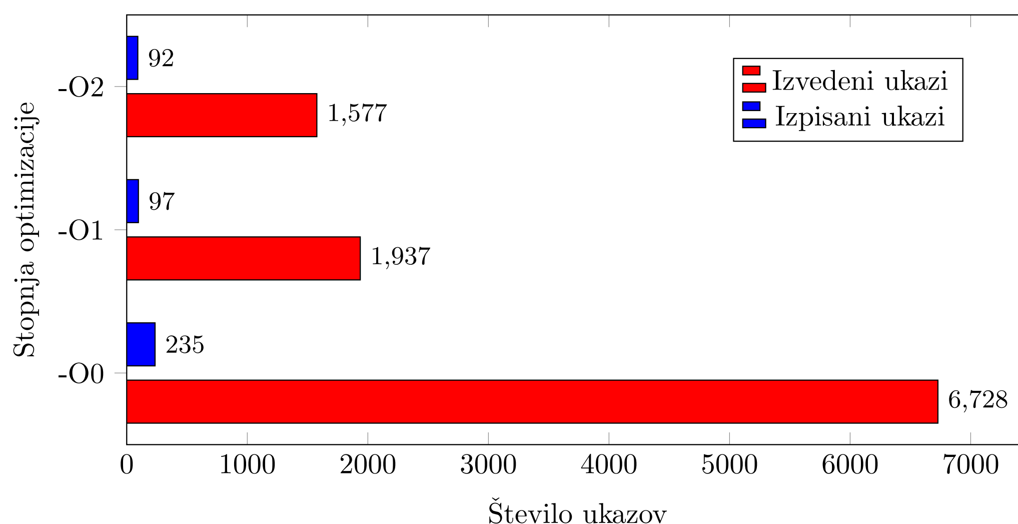
```
41  . BB#2:                         . %for.body4.lr.ph
42          RMO     B, A
43          SHIFTL  A, 2
44          AND     =-4
45          RMO     A, S
46          LDA     0, X            . 3-byte Folded Reload
47          ADDR    A, S
48          STS     16, X           . 3-byte Folded Spill
49          LDA     4, X            . 3-byte Folded Reload
50          RMO     A, T
51          LDS     #9
52          RMO     S, A
53          STA     24, X           . 3-byte Folded Spill
54          STB     12, X           . 3-byte Folded Spill
55  _BB1_3                          . %for.body4
56                                  . =>Loop Header: Depth=2
57          +STT    MEMREG
58          +LDS    @MEMREG
59          LDB     16, X           . 3-byte Folded Reload
60          +STB    MEMREG
61          +LDL    @MEMREG
62          COMPR   L, S
63          +STSW   MEMREG
64          +LDA    MEMREG
65          ADD     #128
66          SHIFTR  A, 7
67          AND     #1
68          COMP    #0
69          JGT     _BB1_5
70  . BB#4:                         . %if.then
71          LDB     16, X           . 3-byte Folded Reload
72          +STB    MEMREG
73          +STS    @MEMREG
74          +STT    MEMREG
75          +STL    @MEMREG
76  _BB1_5                          . %for.inc
77          RMO     T, A
78          ADD     #4
```

```
79          STA     20, X       . 3-byte Folded Spill
80          LDA     24, X       . 3-byte Folded Reload
81          ADD     =-1
82          LDB     12, X       . 3-byte Folded Reload
83          STA     24, X       . 3-byte Folded Spill
84          COMPR   A, B
85          LDA     20, X       . 3-byte Folded Reload
86          RMO     A, T
87          JEQ     _BB1_7
88  . BB#6:                     . %for.inc
89          J       _BB1_3
90  _BB1_7                      . %for.cond.loopexit
91          LDA     4, X        . 3-byte Folded Reload
92          ADD     #4
93          STA     4, X        . 3-byte Folded Spill
94          LDA     8, X        . 3-byte Folded Reload
95          COMP    #10
96          +STSW   MEMREG
97          +LDA    MEMREG
98          ADD     #64
99          SHIFTR  A, 7
100         COMP    #0
101         LDA     8, X        . 3-byte Folded Reload
102         RMO     A, B
103         JGT     _BB1_1
104 . BB#8:                     . %for.cond.cleanup
105         +LDA    #arr
106         ADD     #36
107         +STA    MEMREG
108         +LDS    @MEMREG
109         LDL     28, X       . 3-byte Folded Reload
110         LDT     #32
111         ADDR    T, X
112         RSUB
113 _func_end1
114         EQU     *
115         LTORG
116
```

```
117         .data
118 arr
119         WORD    7           . 0x7
120         RESB    1
121         WORD    8           . 0x8
122         RESB    1
123         WORD    9           . 0x9
124         RESB    1
125         WORD    1           . 0x1
126         RESB    1
127         WORD    2           . 0x2
128         RESB    1
129         WORD    3           . 0x3
130         RESB    1
131         WORD    5           . 0x5
132         RESB    1
133         WORD    4           . 0x4
134         RESB    1
135         WORD    0           . 0x0
136         RESB    1
137         WORD    6           . 0x6
138         RESB    1
139
140 MEMREG  RESW    1
```

5.4.1 Primerjava optimizirane in neoptimizirane zbirne kode

Tudi ta program smo izvedli v optimizirani in neoptimizirani obliki in prešteli število ukazov, rezultati pa so prikazani na grafu 5.3. Stopnja optimizacije `-O3` je bila stopnji `-O2` ponovno enaka, zato je bila iz primerjave izpuščena, stopnja `-O1` pa je tokrat v primerjavi z `-O2` privedla do nekaj ukazov več. Obe optimizirani različici sta v številu izpisanih ukazov več kot dvakrat krajši od neoptimizirane oblike, posledično pa je več kot trikrat manjše tudi število izvedenih ukazov.

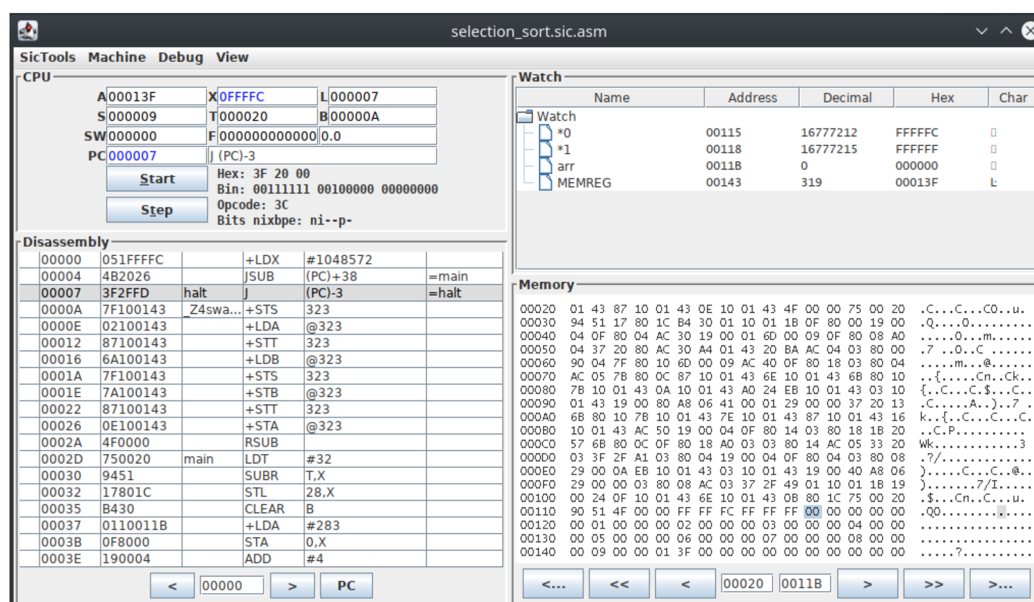


Slika 5.3: Število izpisanih in izvedenih ukazov za program 5.9 glede na stopnjo optimizacije.

5.5 Izvajanje v simulatorju

Prevedene programe je mogoče izvajati v simulatorju SicTools [22] [26]. Za prevajanje smo uporabili ukaz `./clang -target sic-unknown-linux-gnu -O2 -S selection_sort.cpp -o selection_sort.sic.asm`, nato pa datoteko odprli v simulatorju z ukazom `java -jar sictools.jar selection_sort.sic.asm`. Simulator lahko zaženemo z gumbom start, in počakamo, da se ustavi, saj zazna neskončno zanko, ki smo jo ustvarili z dodatkom ukaza `halt J halt`.

Na sliki 5.4 je prikazan simulator SicTools po zagonu kode iz poglavja 5.4. Prek simulatorja lahko v registru S vidimo končni rezultat, število 9, na naslovu `0x11B` pa se nahaja sortirana tabela.



Slika 5.4: Simulator SicTools po zagonu kode 5.10.

Poglavje 6

Sklepne ugotovitve

Z razširitvijo LLVM-ja smo v tej diplomski nalogi želeli omogočiti prevajanje v strojno kodo sistema SIC/XE iz več jezikov. Želeli smo izkoristiti LLVM-jeve optimizacije kode, saj bi za njihove implementacije v svojem prevajalniku potrebovali kar nekaj dela.

Ugotovili smo, da dodajanje podpore za novo arhitekturo v tako obsežen projekt, kot je LLVM, ni enostavno, saj se moramo spoznati s posebnim jezikom, novimi koncepti in prebrati veliko že napisane kode. Prvi izziv predstavlja prevajanje LLVM-ja, ki je zaradi velikosti projekta lahko zelo dolgotrajno, dokler ne najdemo pravih nastavitev. Treba se je soočiti tudi s pomanjkanjem dokumentacije, ki je v primerjavi z dokumentacijo za pisanje sprednjih delov ni tako veliko, zato je pogosto treba pogledati v izvorno kodo ali pa si pomagati s kodo drugih implementacij.

Naši cilji so bili pretežno doseženi, saj nam je uspelo prevajanje osnovnih programov iz C-ja. Za podporo drugih jezikov, katerih prevajalniki omogočajo prevajanje v LLVM-jevo vmesno kodo, bi morali te prevajalnike še ustrezno prilagoditi za specifike sistema SIC/XE. Pri prevajanju lahko uporabimo tudi LLVM-jeve optimizacije, ki pogosto omogočajo precej boljšo kodo, kot bi jo napisali sami ali pa ustvarili s svojim prevajalnikom. Zaradi enostavnosti arhitekture SIC/XE je naša implementacija primerna tudi kot učno gradivo za tiste, ki si želijo spoznati generiranje kode v LLVM-ju ali pa

dodati podporo za novo arhitekturo.

Naša podpora za prevajanje ima določene omejitve, saj ne podpira števil s plavajočo vejico, uporabe standardne knjižnice in prevajanja nekaterih ukazov LLVM-jeve vmesne kode. Kot nadaljnje delo je poleg implementacije podpore za naštete omejitve mogoče izpostaviti tudi dodajanje prevajanja v objektne datoteke, saj se te v sistemu SIC/XE razlikujejo od bolj znanih, kot je ELF.

Celotna izvorna koda naše prilagoditve je na voljo na spletnem naslovu `https://github.com/jakoberzar/sic-llvm`.

Literatura

[1] Qt4-preview-feedback Archive, February 2005 mkspecs and patches for LLVM compile of Qt4. Dosegljivo: https://web.archive.org/web/20111004073001/http://lists.trolltech.com/qt4-preview-feedback/2005-02/msg00691.html, 2019. [Dostopano: 16. 1. 2019].

[2] The Architecture of Open Source Applications. Dosegljivo: http://www.aosabook.org/en/llvm.html#fig.llvm.rtc, 2018. [Dostopano: 13. 6. 2018].

[3] Leland L Beck. *System software: an introduction to systems programming*. Addison-Wesley, 1997.

[4] Binutils - GNU Project. Dosegljivo: https://www.gnu.org/software/binutils/, 2019. [Dostopano: 29. 1. 2019].

[5] Clang C Language Family Frontend for LLVM. Dosegljivo: http://clang.llvm.org/, 2019. [Dostopano: 17. 1. 2019].

[6] CMake. Dosegljivo: https://cmake.org/, 2019. [Dostopano: 29. 1. 2019].

[7] Table of Contents - Tutorial: Creating an LLVM Backend for the Cpu0 Architecture. Dosegljivo: http://jonathan2251.github.io/lbd/index.html, 2019. [Dostopano: 2. 2. 2019].

[8] The Crystal Programming Language. Dosegljivo: https://crystal-lang.org/, 2019. [Dostopano: 17. 1. 2019].

[9] Christoph Erhardt and Dipl-Inf Fabian Scheler. *Design and implementation of a tricore backend for the llvm compiler framework*. Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), 2010.

[10] The Julia Language. Dosegljivo: `https://julialang.org/`, 2019. [Dostopano: 17. 1. 2019].

[11] Klemen Košir. Prevajalnik za programski jezik c za računalnik sic/xe. Diplomska naloga, Fakulteta za računalništvo in informatiko, Univerza v Ljubljani, 2015.

[12] Kotlin Programming Language. Dosegljivo: `http://kotlinlang.org/`, 2019. [Dostopano: 17. 1. 2019].

[13] LLVM Language Reference Manual – LLVM 7 Documentation. Dosegljivo: `https://releases.llvm.org/7.0.1/docs/LangRef.html`, 2019. [Dostopano: 16. 1. 2019].

[14] LLVM Users. Dosegljivo: `https://llvm.org/Users.html`, 2019. [Dostopano: 16. 1. 2019].

[15] TableGen – LLVM 7 Documentation. Dosegljivo: `https://releases.llvm.org/7.0.1/docs/TableGen/index.html`, 2019. [Dostopano: 22. 1. 2019].

[16] The LLVM Compiler Infrastructure. Dosegljivo: `http://llvm.org/`, 2019. [Dostopano: 16. 1. 2019].

[17] Writing an LLVM Backend – LLVM 7 Documentation. Dosegljivo: `https://releases.llvm.org/7.0.1/docs/WritingAnLLVMBackend.html`, 2019. [Dostopano: 24. 1. 2019].

[18] LLVM Developer Policy – LLVM 7 documentation. Dosegljivo: `https://releases.llvm.org/7.0.1/docs/DeveloperPolicy.html#license`, 2019. [Dostopano: 16. 1. 2019].

[19] LLVM Logo. Dosegljivo: https://llvm.org/Logo.html, 2018. [Dostopano: 13. 6. 2018].

[20] LLVM Download Page. Dosegljivo: https://releases.llvm.org/download.html#2.6, 2019. [Dostopano: 17. 1. 2019].

[21] llvmgcc - LLVM C front-end. Dosegljivo: https://releases.llvm.org/1.6/docs/CommandGuide/html/llvmgcc.html, 2019. [Dostopano: 31. 1. 2019].

[22] J Mihelič and Tomaz Dobravec. Sicsim: A simulator of the educational sic/xe computer for a system-software course. *Computer Applications in Engineering Education*, 23(1):137–146, 2015.

[23] Ninja, a small build system with a focus on speed. Dosegljivo: https://ninja-build.org/, 2019. [Dostopano: 29. 1. 2019].

[24] Rust programming language. Dosegljivo: https://www.rust-lang.org, 2019. [Dostopano: 17. 1. 2019].

[25] The Scala Programming Language. Dosegljivo: https://www.scala-lang.org/, 2019. [Dostopano: 17. 1. 2019].

[26] jurem/SicTools: System software and tools for SIC/XE hypothetical computer. Dosegljivo: https://github.com/jurem/SicTools, 2019. [Dostopano: 26. 1. 2019].

[27] Swift.org - Welcome to Swift.org. Dosegljivo: https://swift.org/, 2019. [Dostopano: 17. 1. 2019].