Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Peter Žužek

C prevajalnik za procesor HIP

diplomsko delo

univerzitetni študijski program prve stopnje
računalništvo in matematika

Mentor: prof. dr. Dušan Kodek

Ljubljana 2013

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Univerza *v Ljubljani*
Fakulteta *za računalništvo in informatiko*

Tržaška 25
1001 Ljubljana, Slovenija
telefon: 01 476 84 11
01 476 83 87
faks: 01 426 46 47
01 476 87 11
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 00034/2013
Datum: 09.04.2013

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko ter Fakulteta za matematiko in fiziko izdaja naslednjo nalogo:

Kandidat: **PETER ŽUŽEK**

Naslov: **C PREVAJALNIK ZA PROCESOR HIP**
C COMPILER FOR THE HIP PROCESSOR

Vrsta naloge: Diplomsko delo univerzitetnega študija prve stopnje

Tematika naloge:

Na FRI se pri predmetu Arhitektura računalniških sistemov kot učni pripomoček uporablja procesor HIP. Za ta procesor je bilo v preteklosti razvitih več programskih orodij, med katerimi pa ni prevajalnika za višji programski jezik. Na osnovi odprtokodnega orodja GCC razvijte in izdelajte prevajalnik za programski jezik C, ki prevaja v zbirni jezik procesorja HIP. Prevajalnik preizkusite na tipičnih programih in ovrednotite njegovo delovanje.

Mentor:
prof. dr. Dušan Kodek

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Dekan Fakultete za računalništvo in informatiko:
prof. dr. Nikolaj Zimic

Dekan Fakultete za matematiko in fiziko:
akad. prof. dr. Franc Forstnerič

UNIVERZA V LJUBLJANI • Fakulteta za matematiko in fiziko • 1

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Peter Žužek, z vpisno številko **63100012**, sem avtor diplomskega dela z naslovom:

C prevajalnik za procesor HIP

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom prof. dr. Dušana Kodeka,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne 11. julij 2013 Podpis avtorja:

Rad bi se zahvalil mentorju doktorju Dušanu Kodeku, doktorju Boštjanu Slivniku, Jaku Močniku iz podjetja XLab in doktorju Damjanu Šoncu za vso pomoč, ki so mi jo nudili med izdelavo te diplomske naloge.

Kazalo

Seznam uporabljenih kratic

- `AST` - Abstract Syntax Tree
- `EPC` - Exception Program Counter
- `FPGA` - Field-Programmable Gate Array
- `GCC` - GNU Compiler Collection
- `GNU` - GNU's Not Unix
- `GMP` - GNU Multiple Precision Arithmetic Library
- `GPL` - GNU General Public License
- `MD` - Machine Description
- `MinGW` - Minimalistic GNU for Windows
- `MPC` - GNU Multiple Precision Complex Numbers Library
- `MPFR` - GNU Multiple Precision Floating-Point Reliably
- `MSYS` - Minimal System
- `PC` - Program Counter
- `QImode` - Quarter Integer mode (8-bit)
- `RISC` - Reduced Instruction Set Computing
- `SImode` - Single Integer mode (32-bit)

- `VHDL` - VHSIC Hardware Description Language
- `VHSIC` - very-high-speed integrated circuits

Povzetek

Glavni cilj te diplomske naloge je bilo spisati prevajalnik iz programskega jezika C v zbirni jezik procesorja HIP. Preden je to možno izvesti, je potrebno dobro poznati postopek prevajanja programskega jezika ter ciljno arhitekturo. V tem diplomskem delu so sprva predstavljene osnove prevajanja. Predstavljen je prevajalnik GCC, ki je osnova C prevajalnika za HIP. Sledi opis delovnega okolja in programskih orodij, ki so bila uporabljena za prevajanje in testiranje. Pisanje C prevajalnika ni enostavno opravilo, zato so bile upoštevane določene omejitve, česa v tem diplomskem delu ni moč pričakovati. Sledi še pregled arhitekture procesorja HIP.

Preostanek diplomskega dela se ukvarja z dejansko implementacijo prevajalnika. Sprva je predstavljeno, kako se prevajalnik GCC prevede za želeno ciljno arhitekturo in kako se registrira novo arhitekturo. Sledi najobsežnejši del, ki prikazuje izseke kode iz zadnjega dela prevajalnika GCC, prilagojenega arhitekturi HIP. S primeri za procesor HIP je prikazana notranja sestava prevajalnika GCC. Na koncu je prikazanih še par primerov, kakšno kodo ustvari končna izvedba C prevajalnika za HIP.

Ključne besede: prevajalnik, procesor, arhitektura, sklad, HIP, GCC, zadnji del, vmesna koda, zbirnik

Abstract

The main goal of this thesis was to write a C compiler for the HIP processor. In order to be able to do that, one has to be well acquainted with the compilation process and the target computer architecture. This thesis introduces the basics of compilation. It begin with the description of the GCC compiler, which is the basis for the C compiler for HIP. This is followed by the workspace description - which software tools were used for compilation and testing. Writing a C compiler is not a simple task, therefore some restrictions of what not to expect from this thesis were put into place. The next chapter is an overview of the HIP processor architecture.

The rest of the thesis deals with the actual implementation of the compiler. It starts with an overview of how GCC is compiled for some target architecture and how to add a custom target architecture. This is followed by code excerpts from the HIP-specific GCC back end. Code examples are also used to reveal some of the GCC internals. At the end there are examples of the assembly code, generated by the HIP C compiler.

Keywords: compiler, processor, achitecture, stack, HIP, GCC, back end, intermediate code, assembler

Poglavje 1

Uvod

Na vajah pri predmetu Arhitektura računalniških sistemov je asistent doktor Damjan Šonc izrazil željo, da bi se razvil prevajalnik za programski jezik C za procesor HIP. Na vajah se namreč ta procesor uporablja kot učni procesor. Študentje se učijo programirati v zbirnem jeziku. Včasih so navodila vaj napisana v programskem jeziku C, ki naj bi bil študentom bolj domač, nakar morajo študentje spisati ustrezno zbirno kodo. V tej diplomski nalogi je bila želja poenostaviti to pretvorbo za čimvečjo množico programov, spisanih v jeziku C.

Spisati C prevajalnik ni enostavno opravilo. Prevajalniki so že sami po sebi obsežni, programski jezik C le še doda h kompleksnosti.

Ker so prevajalniki pogosto zgrajeni po delih - prisotna je vsaj ločitev na prednji in zadnji del - se lahko za osnovo vzame že obstoječ prevajalnik in se mu priredi zadnji del. Za ta namen je bil izbran odprtokodni prevajalnik GCC. Čeprav ima GCC obsežno dokumentacijo, pisanje zadnjega dela zanj še vedno ni enostavno. Pri tem mi je pomagal Jaka Močnik iz podjetja XLab, ki je že sodeloval pri razvoju zadnjega dela za GCC. Glavni nasvet je bil, da se vzame že spisan zadnji del za procesor, podoben HIPu, in se ga priredi.

Morda se popravljanje obstoječe kode sliši enostavno, vendar je potrebno najprej spoznati mnogo podrobnosti, na kakšen način zadnji del prevajalnika GCC sploh deluje. Potrebno je definirati prave makre na pravi način, se

naučiti GCCjevega lastnega jezika za opis procesorja in spisati pomožno kodo v programskem jeziku C. Pri testiranju pravilnosti se pojavi marsikakšna malenkost, ki jo je potrebno popraviti.

V tej diplomski nalogi so sprva predstavljene osnove: o prevajalnikih na splošno, o prevajalniku GCC ter o delovnem okolju. Nadaljuje se s predogledom arhitekture procesorja HIP, kaj se lahko pričakuje od C prevajalnika za HIP in kaj ne. Na koncu so še konkretni primeri programske kode, ki pomagajo pri sprotni razlagi notranjega delovanja prevajalnika GCC.

Poglavje 2

Prevajalnik GCC

GCC je odprtokodna zbirka prevajalnikov - angleško se imenuje GNU Compiler Colection. Podpira velik nabor programskih jezikov in velik nabor procesorskih arhitektur. To poglavje je spisano deloma splošno glede prevajalnikov in ponekod nekoliko bolj natančno za GCC.

V osnovi prevajalnik dobi kot vhod izvorno kodo, spisano v višjem programskem jeziku, in vrne kodo nižjega programskega jezika, najpogosteje strojno ali zbirno kodo ciljne procesorske arhitekture. Zaradi zmanjšanja kompleksnosti se prevajanje izvede v več fazah. V grobem ločimo dve glavni fazi. Prva se imenuje prednji del (front end) in je odvisna le od vhodnega programskega jezika, ne od ciljne arhitekture. Druga faza je ravno obratna, odvisna le od ciljne arhitekture in ne od vhodnega jezika. V resnici je po navadi vmes prisotna še ena faza, srednji del, ki je neodvisna tako od vhodnega jezika kot od ciljne arhitekture. Namen srednjega dela je po navadi optimizacija vmesne kode.

2.1 Faze prevajanja

2.1.1 Leksikalna analiza

Ta faza prebere vhodni program znak za znakom in ustvari zaporedje osnovnih simbolov (token). Najpogosteje se tukaj izloči prazne znake (whitespace),

kar poenostavi nadaljnjo obravnavo.

2.1.2 Sintaksna analiza

Za vhod dobi zaporedje osnovnih simbolov in preveri, če je zaporedje veljavno za vhodni programski jezik. Zgradi abstraktno sintaksno drevo (Abstract Syntax Tree, AST), drevesno predstavitev osnovnih simbolov vhodnega programa z dodanimi metapodatki. Drevesna predstavitev omogoča enostavnejšo obdelavo, saj je iz nje bolje razvidna odvisnost med osnovnimi simboli.

2.1.3 Semantična analiza

Preveri AST za pravilne pojavitve imen spremenljivk in podprogramov ter za skladnost med uporabljenimi tipi podatkov.

2.1.4 Generator vmesne kode

Prevede AST v prevajalniku lastno vmesno kodo (Intermediate Code, IMC). Vmesna koda je nekoliko bližje strojnemu jeziku. Zasnovana je tako, da je neodvisna od vhodnega jezika in ciljne arhitekture ter poenostavi optimizacijo. S stališča te diplomske naloge je to pomembnejši del.

2.1.5 Optimizacija kode

Odstranitev odvečnega kopiranja podatkov, nadomestitev izrazov s konstantami, zmanjševanje števila klicev podprogramov, itd.

2.1.6 Generator kode

Zaobjema tlakovanje (tiling) vmesne kode s strojnimi ukazi ciljne arhitekture ter dodeljevanje registrov začasnim spremenljivkam. Pri tlakovanju po navadi prevajalnik želi uporabiti čimmanj ukazov, lahko pa se ravna tudi po drugih smernicah, recimo ceni posameznega ukaza. Poskrbeti mora tudi, da se lahko vse vmesne vrednosti izrazov shranijo v omejeno množico registrov.

2.2 Pomožni programi

Prevajalnik lahko vsebuje tudi dodatne programe, ki skrbijo za pretvorbo kode.

2.2.1 Predprocesor

Pri določenih jezikih (C, C++) je potrebno pred vsemi ostalimi fazami opraviti nekatere osnovne operacije, ki razširijo kodo. To vključuje zamenjavo pojavitev makrov s konstantami ali programsko kodo, vključitev drugih datotek, preverjanje pravilnosti konstant in podobno. Rezultat je spremenjen vhodni program, kjer so makri zamenjani z vrednostmi, izbrane le določene vrstice izvorne kode in je več datotek združenih v eno.

2.2.2 Zbirnik

Prevajalnik običajno tvori zbirno kodo ciljne arhitekture in posreduje rezultat zbirniku. Zbirnik je precej bolj enostaven program od prevajalnika, saj le pretvori zbirno kodo v strojno kodo. Zbirna koda ima zelo podoben pomen kot strojna koda, le da je namesto ničel in enic program zapisan človeku bolj razumljivo. Zbirnik ima sicer tudi dodatne naloge, recimo da zagotovi prostor v pomnilniku za spremenljivke ali da upošteva psevdo-ukaze (običajno so to makri). Izhodna datoteka zbirnika je ali izvršljiva datoteka ali objektna datoteka.

2.2.3 Povezovalnik

Kot vhod dobi eno ali več objektnih datotek, kot izhod vrne izvršljivo datoteko. Objektna datoteka vsebuje strojno kodo z dodanimi podatki o neodvisnosti od položaja v pomnilniku ter sklice na zunanje podprograme, običajno na funkcije iz standardne knjižnice. Povezovalnik te sklice iz več objektnih datotek poveže v delujoč program.

2.2.4 Nalagalnik

Izvršljivo datoteko v pomnilnik naloži program, imenovan nalagalnik. Nalagalnik je večinoma del operacijskega sistema, ki zagotavlja pravilno delovanje programa oziroma ga lahko prekine, če se začne obnašati nepredvidljivo (začne dostopati do pomnilniškega prostora ostalih programov).

Poglavje 3

Delovno okolje

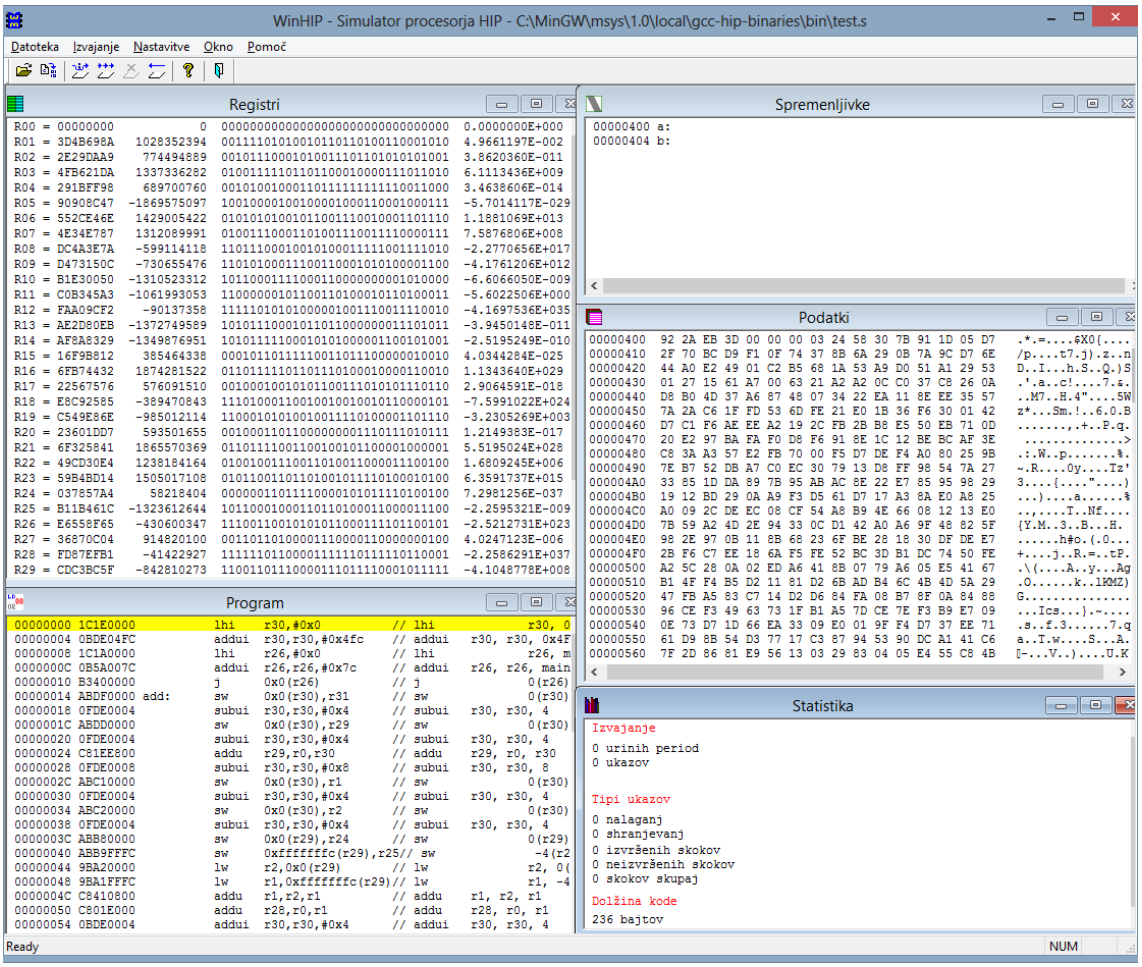
3.1 Simulator WinHIP

Za namen prevajanja zbirne kode in njenega izvrševanja pri predmetu Arhitektura računalniških sistemov je asistent doktor Damjan Šonc spisal simulator procesorja HIP, imenovan WinHIP.

WinHIP je osnovan na programu WinMIPS64 - HIP je po osnovni zgradbi namreč precej podoben arhitekturi MIPS. WinHIP vsebuje grafični vmesnik, ki omogoča naložitev datoteke z zbirno kodo, njen prevod in izvajanje dobljene strojne kode.

Slika 3.1 prikazuje simulator WinHIP, kjer so vidna naslednja podokna:

- Registri - prikazuje trenutne vrednosti registrov v šestnajstiškem, desetiškem, dvojiškem ter IEEE 754 zapisu.
- Program - prikazuje naslove, strojne kode in zbirne kode prevedenih ukazov.
- Spremenljivke - prikazuje naslov v pomnilniku, ime in vrednost globalnih spremenljivk.
- Podatki - prikazuje simuliran pomnilnik računalnika s procesorjem HIP: položaje ter šestnajstiške in znakovne (ASCII) vrednosti pomnilniških celic na teh položajih.



Slika 3.1: Simulator WinHIP

- Statistika - nekaj statistike o številu izvršenih ukazov ter potrebnih urinih period.

Poleg prevoda in zagona celotnega programa je moč programe izvajati tudi po le en ukaz naenkrat za boljši pregled nad dogajanjem. Na voljo je tudi različica simulatorja WinHIP, ki podpira cevovodno izvedbo procesorja HIP. V njej je večji poudarek na prikazu dogajanja v cevovodu med izvajanjem.

3.2 MinGW

Uporabljen je bil prevajalnik GCC različice `4.6.2`. V času začetka pisanja te diplomske naloge (april 2013) je bila sicer najnovejša stabilna različica prevajalnika GCC `4.7.3`, vendar so bile posredi dodatne omejitve. Ker je GCC v prvi vrsti spisan za operacijski sistem UNIX - in sisteme, osnovane na UNIXu - v operacijskem sistemu Windows ne deluje privzeto, saj se zanaša na določene sistemske klice. Za delovanje v Windows je potreben dodaten vmesnik. Za potrebe te naloge je bil kot vmesnik izbran prevajalnik MinGW in okolje MSYS.

MinGW je okrajšava za Minimalistic GNU for Windows. Je implementacija prevajalnika GCC s pomočjo knjižnic operacijskega sistema Windows. Vendar za lasten prevod prevajalnika GCC to ni dovolj. Ob namestitvi MinGW je potrebno namestiti tudi okolje MSYS. MSYS je simulacija ukazne lupine operacijskega sistema UNIX. Priskrbi programe, ki so potrebni ob prevajanju izvorne kode prevajalnika GCC.

Aprila 2013 je bila najnovejša različica MinGW `4.7.2`, vendar sem na domačem računalniku uporabljal starejšo, `4.6.2`, zato je bila izbrana enaka različica tudi za C prevajalnik za HIP. MinGW izdaje sicer nekoliko zaostajajo. Junija 2012 je bila najnovejša različica MinGW še vedno `4.7.2`, medtem ko je bila na voljo že koda za GCC različice `4.8.1`. Če se zgradba prevajalnika GCC v prihodnje ne bo preveč spremenila, bi moral C prevajalnik za HIP pravilno delovati tudi na novejših izdajah.

Na ostalih operacijskih sistemih C prevajalnik za HIP ni bil ne preveden, ne preizkušen. Najverjetneje bi bil postopek na operacijskem sistemu Linux enostavnejši, kajti Linux je osnovan na UNIXu - tako ne bi bilo potrebno nameščati posebnega UNIX vmesnika.

3.3 GitHub

Prevajalnik GCC je izdan pod odprtokodno licenco GPL, ki zahteva, da so tudi vse spremembe te kode izdane pod enako licenco. Posledično je izvorna koda C prevajalnika za HIP na voljo na spletni strani upravljalca datotek GitHub [1].

Poglavje 4

Omejitve C prevajalnika za HIP

Pri procesorju HIP se pojavi težava, da nima določenega zapisa objektne datoteke. To ni tako zelo nenavadno, kajti obliko objektne datoteke po navadi določi operacijski sistem, ki tudi skrbi za pravilno nalaganje programa v pomnilnik in njegovo izvajanje. Trenutno noben operacijski sistem ni prilagojen za procesor HIP.

Poseben operacijski sistem načeloma niti ni potreben, saj bi se dalo spisati povezovalnik in nalagalnik, ki bi simulirala delovanje procesorja HIP - podobno kot to počne zbirnik in simulator izvajanja WinHIP. To je izvedljivo, vendar precej obsežno in bi najverjetneje zahtevalo dodatno diplomsko nalogo. Glavni cilj te naloge je ustvariti prevajalnik, ki zna enostavno kodo programskega jezika C prevesti v zapis, primeren za študente predmeta Arhitektura računalniških sistemov. Pri predmetu se obravnava zbirna koda in prevajanje zbirne kode v strojno kodo. Pri predmetu se ne obravnava objektne datoteke za HIP (ker ni določena) in posledično ne povezovalnika. Ker povezovalnik skrbi za povezavo z zunanjimi podprogrami, v končnem prevajalniku ni možnosti rabe standardne knjižnice programskega jezika C. Zaradi teh razlogov se končna izvedba prevajalnika te naloge (končni prevajalnik) omeji na generiranje zbirne kode.

Prav tako je prevajalnik omejen le na programski jezik C. GCC sicer podpira velik nabor programskih jezikov v prednjem delu, prav tako se v tej

nalogi obravnava le zadnji del, ki naj bi bil neodvisen od izvornega programskega jezika. Vendar ta neodvisnost ne drži povsem. Primer je jezik C++, ki potrebuje posebno obravnavo izjem (exception). Končni prevajalnik te naloge izjem in prekinitev ne obravnava.

Poglavje 5

Arhitektura procesorja HIP

HIP je okrajšava za hipotetični procesor. Zasnoval ga je profesor doktor Dušan Kodek za namene poučevanje računalniških arhitektur. Podrobno je razdelan v knjigi Arhitektura in organizacija računalniških sistemov [6].

Spada med RISC računalnike: je 3-operandni registrsko-registrski procesor z relativno velikim naborom registrov in relativno majhnim številom ukazov. Velikost pomnilniške besede je 32 bitov, besede so shranjene po pravilu debelega konca (Big Endian). Čeprav je procesor hipotetičen, vseeno obstaja implementacija vezja v VHDL, ki omogoča zapis na vezje FPGA. Te čipe uporabijo študentje Arhitekture računalniških sistemov kot primer praktične rabe zbirnega jezika - konkretno se s pomočjo procesorja HIP naslavlja LCD zaslon, da izpiše besedilo.

5.1 Registri

HIP vsebuje 32 32-bitnih splošnonamenskih registrov. Vrednost registra 0 je vedno enaka 0 (`r0 = 0`).

Poleg splošnonamenskih so na voljo še posebni registri:

- Prvi je 32-bitni Program Counter oziroma `PC`. Ta vsebuje pomnilniški naslov, kjer se nahaja naslednji ukaz za izvršitev.

- Drugi je 32-bitni Exception Program Counter oziroma `EPC`. Ko v procesorju pride do prekinitve, se vrednost `PC` zapiše v `EPC`.

- Da so prekinitve omogočene, mora biti nastavljena vrednost 1 v 1-bitnem registru `I`. Ker je I le 1-biten, to pomeni, da je brez dodatne logike na voljo le en nivo prekinitev.

5.2 Formati ukazov in načini naslavljanja

Na voljo sta dva formata ukazov. Vsi ukazi so 32-bitni in 3-operandni.

5.2.1 Format 1

Prvih 6 bitov je operacijska koda ukaza, nato sledi 5 bitov za izvorni register, 5 bitov za ciljni register in 16 bitov za številsko konstanto. "Izvorni register"pomeni, da se v ukazu prebere njegova vrednost, "cilji" pa da se v ta register piše.

Format 1 omogoča dva načina naslavljanja.

Prvo je **takojšnje naslavljanje**, kjer je številska konstanta takojšnji operand. Primer je ukaz

```
1 addui r1, r2, 30
```

ki vrednosti registra `r2` prišteje konstanto `30` in rezultat shrani v register `r1`. Ukazi, ki uporabljajo takojšnje naslavljanje, vsebuje na koncu mnemonika črko `i`.

Drugo je **bazno naslavljanje**, kjer se izračuna pomnilniški naslov tako, da se vzame vrednost baznega registra in se ji prišteje vrednost številske konstante - konstanta torej deluje kot odmik. Primer je ukaz

```
1 lw  r1, 16(r2)
```

ki izračuna naslov v pomnilniku tako, da vrednosti registra `r2` prišteje konstanto `16`, nato iz tega naslova prebere eno besedo in jo shrani v register `r1`.

5.2.2 Format 2

Prvih 6 bitov je ponovno operacijska koda, nato sledi 5 bitov za prvi izvorni register, 5 bitov za drugi izvorni register in 5 bitov za ciljni register. Zadnjih enajst bitov je na voljo kot možna razširitev obstoječega nabora ukazov. Ta format omogoča **neposredno registrsko naslavljanje**, kjer se izvrši operacija nad izvornima registroma in se rezultat shrani v ciljni register. Primer je ukaz

```
1 and  r1, r2, r3
```

ki izvede logično konjunkcijo nad registroma `r2` in `r3` ter rezultat shrani v register `r1`.

Pri pogojnih skokih se uporablja **relativno naslavljanje** - zbirnik prevede odmik relativno glede na trenutno vrednost `PC`. Primer je ukaz

```
1 beq  r1, L1
```

ki primerja vrednost registra `r1` s konstanto `0` - če je dosežena enakost, skoči na naslov, označen z labelo `L1`.

5.3 Ukazi

HIP vsebuje naslednje osnovne skupine ukazov: prenos podatkov, aritmetične operacije, logične operacije, vejitve in sistemske ukaze. Prenos podatkov omogoča naslavljanje posameznih bajtov (8 bitov), polbesed (16 bitov, halfword) ali besed (32 bitov, word), vendar morajo biti polbesede obvezno poravnane na 2 bajta in besede na štiri bajte. Neporavnanost sproži past. HIP omogoča le celoštevilsko aritmetiko, tudi ni množenja in deljenja. Sistemski ukazi se ukvarjajo s prekinitvami.

Ukaz za klic shrani trenutno vrednost `PC` v ciljni register. Primer klica funkcije fib:

```
1 call r1, fib(r0)
```

Tukaj se `PC` shrani v register `r1`.

5.4 Sklad in klicna konvencija

Sledeča pravila niso del arhitekture procesorja HIP, temveč le dogovor o uporabi registrov ter pomnilnika pri klicih podprogramov. Dogovor pri procesorju HIP je, da sklad raste v smeri padajočih naslovov. Skladovni kazalec kaže na prvo prosto mesto na skladu.

Uporaba registrov v zvezi s skladom:

- `r24` - prvi parameter z leve
- `r25` - drugi parameter z leve
- `r26` - bazni register za dolge skoke
- `r27` - bazni register za dolge klice
- `r28` - vračana vrednost
- `r29` - kazalec na okvir
- `r30` - skladovni kazalec
- `r31` - povratni naslov

Shranitev registra na sklad (push) se izvede z dvema ukazoma:

```
1 sw  0(r30), rN
2 subui r30, r30, 4
```

Podobno se obnovitev registra s sklada (pop) izvede z dvema ukazoma:

```
1 addui r30, r30, 4
2 lw  rN, 0(r30)
```

Prva dva parametra podprograma se prenašata prek registrov `r24` in `r25`. Ostali parametri se prenašajo prek sklada v obratnem vrstnem redu - najbolj desni parameter se prvi naloži na sklad.

Ukaz za klic shrani `PC` v register `r31`. To je povratni naslov, torej kam se mora klican podprogram vrniti ob koncu lastnega izvajanja. Zaključek podprograma je potem tak:

```
1 j  0(r31)
```

Ker je odmik skoka oziroma klica omejen na predznačenih šestnajst bitov, se v primeru, da je odmik večji od 16 bitov, za skok uporabi register `r26` in za klic `r27`. To so tako imenovani dolgi skoki oziroma klici. Na tem mestu je potrebno odmik naložiti v register, za kar sta potrebna dva ukaza:

```
1 lhi  r27, fib
2 addui r27, r27, fib
```

Klic se nato izvrši z ukazom

```
1 call r31, 0(r27)
```

Ker je velikost takojšnjega operanda 16-bitna, labele pa lahko kažejo na naslove, ki potrebujejo več kot 16 bitov (do 32 bitov), je potrebno take operande nalagati v dveh korakih. Najprej se uporabi ukaz `lhi`, ki takojšnji operand zapiše v zgornjih 16 bitov ciljnega registra, spodnjih 16 bitov nastavi na 0. Nato se ciljnemu registru prišteje isto konstanto, kar vpliva le na spodnjih 16 bitov registra. Zbirnik običajno prevaja labele v 16-bitne naslove tako, da vzame spodnjih 16 bitov 32-bitnega naslova - izjema je ukaz `lhi`, pri katerem vzame zgornjih 16 bitov.

C prevajalnik za HIP ne uporablja registrov `r26` in `r27` za dolge skoke oziroma klice. Razlog za to odločitev je lažja implementacija in možnost večje učinkovitosti, če se GCC sam odloči, kateri register bo uporabil kot bazni register. Če bazni register ni predhodno določen, lahko GCC uporabi katerega izmed tistih registrov, v katere se je že pisalo.

Klicani podprogram mora na sklad shraniti vse registre, ki jih spreminja. Edine izjeme so:

- `r0` - je vedno enak `0`, vanj ni možno pisati
- `r24`, `r25` - prenašata prva dva parametra podprograma, njuna ohranitev ni pričakovana
- `r28` - vrača vrednost podprograma, zato se mora spremeniti

- `r30` - skladovni kazalec se nadzorovano spreminja, vanj se ne sme prosto pisati

Pri klicih podprogramov je pomembno, kaj se zgodi ob začetku podprograma (prolog) in kaj na koncu (epilog):

- V prologu se najprej shrani register `r31`, nato je potrebno shraniti kazalec okvira na sklad, register `r29`.

  Skladovni kazalec `r30` nato prepiše kazalec okvira `r29` z ukazom:

```
1 addu r29, r0, r30
```

  Na ta način je omogoče dostop do parametrov in do lokalnih spremenljivk s pomočjo odmika od kazalca na okvir Prva lokalna spremenljivka se nahaja na odmiku `0`, druga na `-4`, tretja na `-8` in tako naprej. Podobno se parametri nahajajo na odmikih `12`, `16` in višjih.

  Za lokalne spremenljivke je potrebno rezervirati prostor na skladu, kar se naredi z ukazom:

```
1 subui r30, r30, N
```

  Tukaj je `N` število bajtov, ki ga zasedejo lokalne spremenljivke.

  Na koncu prologa je še shranjevanje vseh registrov, v katere podprogram piše.

- V epilogu je potrebno izvesti obratno zaporedje kakor v prologu, torej najprej obnoviti vse shranjene registre, premakniti kazalec okvira v skladovni kazalec ter obnoviti registra `r29` in `r31`. Premik kazalca okvira v skladovni kazalec hkrati tudi pobriše lokalne spremenljivke s sklada. Sledi še skok na povratni naslov.

Poglavje 6

Prevajanje prevajalnika GCC

6.1 Priprava

Podrobna navodila za prevod prevajalnika GCC so podana na spletu [8]. Tukaj so izpostavljene le najbolj osnovne nastavitve.

V operacijskem sistemu Windows je potreben prevajalnik MinGW. GCC za prevod potrebuje knjižnice `GMP`, `MPFR` in `MPC`. MinGW vsebuje upravljalec s paketi, zato se lahko v ukazni vrstici operacijskega sistema Windows požene naslednje ukaze:

```
1 mingw-get install gmp
2 mingw-get install mpfr
3 mingw-get install mpc
```

Od te točke naprej se v sistemu Windows za vnašanje ukazov uporablja okolje `MSYS`.

6.2 Skripta za prevod

Koda 6.1: Skripta za prevod izvorne kode prevajalnika GCC

```
1  #!/bin/bash
2  # build-gcc-incremental.sh
3  export TARGET=$1
4  export PREFIX=/usr/local/gcc-$TARGET-binaries/
5  export PATH=$PATH:$PREFIX/bin
6  cd ./build-gcc-$TARGET
7  rm ./gcc/gtype.state
8  ../gcc/configure -C --quiet
9  --target=$TARGET --prefix=$PREFIX
10 --disable-nls --enable-languages=c --without-headers
11 make -j4 all-gcc --quiet
12 make install-gcc --quiet
```

Skripto 6.1 se požene v mapi, ki naj se za lažjo razlago imenuje `src`, čeprav se lahko imenuje drugače. Skripta predpostavi, da se v mapi `src` nahajata podmapi `gcc` in `build-gcc-$TARGET`. Pri tem `$TARGET` nadomestimo s ciljno arhitekturo, v našem primeru `hip`. Dodatna predpostavka je, da že obstaja mapa `/usr/local/gcc-$TARGET-binaries/`, kamor se namesti končni prevajalnik.

Skripto 6.1 se najlažje požene z ukazom:

```
1  ./build-gcc-incremental.sh hip 2> build-gcc-error.out
```

Tukaj je podan parameter hip kot ciljna arhitektura. Za lažje razhroščevanje je priporočljivo napake prevajanja zapisati v datoteko, kot primer je datoteka `build-gcc-error.out`.

Skripta 6.1 je zasnovana, da so nadaljnji prevodi čim hitrejši. To nakazuje več zastavic. Primer je `-C`, ki omogoči predpomnjenje. Sledijo `--disable-nls --enable-languages=c --without-headers`, ki povedo, da se prevede le majhen del prevajalnika. Pomembna zastavica je tudi `-j4`, ki pove, naj se med prevajanjem uporabijo štiri logična procesorska jedra. Če ima upora-

bljeni računalnik na razpolago več procesorskih jeder, jih je zelo priporočljivo vsa uporabiti. GCC je namreč precej obsežen prevajalnik, ki lahko potrebuje dolgo časa za prevajanje.

Upoštevati pa je potrebno, da kasnejša prevajanja ne potrebujejo celotne skripte 6.1. Vrstice `7` do `10` (vključujoč `7` in `10`) so potrebne le pri spreminjanju nastavitev prevajalnika GCC, sicer se jih lahko izpusti.

Poglavje 7

Dodajanje lastnega zadnjega dela prevajalniku GCC

7.1 Procesor Moxie

Pri snovanju lastnega zadnjega dela je priporočljivo uporabiti že obstoječega. Izbrati je potrebno procesor, ki je čimbolj podoben našemu. Kljub tej poenostavitvi namreč še vedno ostane precej veliko dela.

HIP je arhitekturno precej podoben MIPS [2] - to podrobnost izkorišča simulator WinHIP, ki je osnovan na WinMIPS64. Vendar je zadnji del prevajalnika GCC za MIPS vse prej kot enostaven - MIPS je možno kupiti v mnogo različnih izvedenkah in GCC mora to vse pokriti, kar se odraža v obsežni kodi z mnogo datotekami.

Precej bolj enoten procesor je MMIX [5]. Ta je podoben HIPu v smislu, da je prav tako hipotetičen, učni procesor. Zasnoval ga je Donald Knuth kot moderno nadomestitev prejšnjega procesorja MIX. Vendar se MMIX od HIP arhitekturno precej razlikuje. MMIX ima 256 ukazov, 256 64-bitnih splošnonamenskih registrov, 32 posebnih registrov, ukaze za množenje, plavajočo vejico, vektorske podatke in mnogo več. Takoj je razvidno, da je to precej bolj obsežen procesor od HIP. V tej diplomski nalogi je sprva bil uporabljen MMIX, kajti zadnji del za GCC ne vsebuje veliko datotek in je

posledično dokaj razumljiv. Vseeno pa je še vedno precej obsežen in se je ponekod kar težko znajti.

Po približno mesecu dela s procesorjem MMIX sem zadnji del raje začel snovati na procesorju **Moxie** [3]. Moxie je zasnoval Anthony Green. Trenutno je verjetno najmanjša javna implementacija zadnjega dela GCC. Tudi Moxie ima nekaj očitnih razlik glede na HIP, vendar jih je bilo lažje odpraviti kot pri MMIX.

Moxie ima le 16 32-bitnih splošnonamenskih registrov. Ukazi so dolgi 16 bitov, vendar nekaterim ukazom sledi še 32-bitni operand. Večina ukazov je dvo-operandnih, primer:

```
1 and $r1, $r2
```

V zgornjem primeru sta registra `$r1` in `$r2` izvorna, rezultat pa se shrani v register `$r1`, ki je torej tudi ciljni.

Zadnji del za procesor Moxie vsebuje tudi dodatne možnosti, ki pri HIP zaradi pomanjkanja povezovalnika trenutno niso na voljo, recimo ELF objektno datoteko ter knjižnico za računanje v plavajoči vejici (*soft floating point*).

Ostale razlike med HIP in Moxie bodo dodatno obrazložene na konkretnih primerih programske kode.

7.2 Registracija zadnjega dela

Naj tudi tukaj veljajo predpostavke, da se naša delovna mapa imenuje `src`, v njej se nahajata skripta za prevod ter mapa `gcc` z izvorno kodo prevajalnika GCC.

Izvorna koda različnih zadnjih delov prevajalnika GCC se nahaja v mapi `src/gcc/gcc/config`. Znotraj se nahaja mnogo podmap. Podmape nosijo ime arhitekture, ki se nahaja v njih. V našem primeru gre za `src/gcc/gcc/config/hip`. Znotraj te arhitekturne podmape se morajo nahajati vsaj obvezne tri datoteke: `hip.h`, `hip.c` in `hip.md`. Za druge arhitekture se te datoteke ustrezno drugače imenujejo.

Te tri datoteke ne smejo biti prazne. Ni mi uspelo ugotoviti, kakšne so minimalne zahteve za delujoč lasten zadnji del. Čeprav je GCC dokumentacija [4] obsežna, tega podatka ni enostavno najti. Ni namreč čisto jasno, kateri ukazi morajo biti podprti. Za makre je na voljo obsežen seznam, ki vsakega podrobno opiše. Običajno v opisu makra piše ‘‘`If this macro is not defined ...`’’ ali ‘‘`Default value of this macro is ...`’’ - to je v veliko pomoč, vendar bi bil priročen seznam najnujnejših makrov.

Ko so datoteke `hip.h`, `hip.c` in `hip.md` pripravljene, je potrebno še popraviti nastavitve prevajalnika GCC.

Najprej je potrebno popraviti datoteko `src/gcc/config.sub`.

Koda 7.1: Registracija procesorja HIP v datoteki config.sub.

```
1 ...
2   | hip \
3   | moxie \
4 ...
```

V izseku 7.1 je prikazan dodan zapis za procesor HIP. Znak | označuje, da gre za logično disjunkcijo - ta zapis je namreč del logičnega izraza, ki pove, katere arhitekture so podprte.

Ostane še datoteka `src/gcc/gcc/config.gcc`. HIP ne potrebuje nobenih dodatnih nastavitev, zato je nastavitev enostavna.

Koda 7.2: Registracija procesorja HIP v datoteki config.gcc.

```
1  ..
2  hip) cpu_type=hip
3    ;;
4  moxie*) cpu_type=moxie
5    ;;
6  ...
7  hip-*-*)
8    ;;
9  moxie-*-elf)
10   gas=yes
```

```
11    gnu_ld=yes
12    tm_file="dbxelf.h elfos.h newlib-stdint.h ${tm_file}"
13    extra_parts="crti.o crtn.o crtbegin.o crtend.o"
14    tmake_file=
15      "${tmake_file} moxie/t-moxie moxie/t-moxie-softfp soft-fp/t-softfp"
16    ;;
17  ...
```

V izseku izvorne kode 7.2 je za primerjavo prikazana tudi ena izmed možnosti nastavitev za procesor Moxie. Na tem mestu se lahko poda dodatne možnosti, morda vključi kakšno knjižnico, vendar to pri HIP ni potrebno.

7.3 hip.h

Datoteka `hip.h` je namenjena definiciji makrov, ki jih potrebuje prevajalnik GCC.

Navedeni so izseki programske kode z njihovo obrazložitvijo.

7.3.1 Shranjevanje podatkov v pomnilniku

Koda 7.3: Primeri opisa shranjevanja podatkov.

```
1  #define SHORT_TYPE_SIZE  16
2  #define INT_TYPE_SIZE  32
3  #define DEFAULT_SIGNED_CHAR 1
4  #define UNITS_PER_WORD  4
5  #define BYTES_BIG_ENDIAN 1
6  #define WORDS_BIG_ENDIAN 1
7  #define STRICT_ALIGNMENT 1
```

Pregled pomena kode 7.3 po vrsticah:

- Prva in druga vrstica sporočata prevajalniku GCC, da je tip **short** na procesorju HIP velik 16 bitov in da je tip **int** na HIP velik 32 bitov.

- `DEFAULT_SIGNED_CHAR` v tretji vrstici sporoča, da naj se tip **char** obravnava predznačeno.
- Makro `UNITS_PER_WORD` pove, da pomnilniška beseda (word) vsebuje 4 bajte.
- Vrstici 5 in 6 povesta, da se podatki v pomnilniku shranjujejo po pravilu debelega konca.
- `STRICT_ALIGNMENT` sporoča v vrstici 7, da je poravnanost pomnilniških naslovov obvezna.

7.3.2 Izpis zbirne kode

Koda 7.4: Primeri opisa načina izpisa zbirne kode.

```
1 #define ASM_COMMENT_START  ";"
2 #define DATA_SECTION_ASM_OP  "\n\t.data\n"
3 #define ASM_OUTPUT_FUNCTION_LABEL(stream, name, decl) \
4   fprintf(stream, "%s:\n", name)
5 #undef TARGET_ASM_INTEGER
6 #define TARGET_ASM_INTEGER  hip_assemble_integer
```

- Prva vrstica kode 7.4 označuje, kako se začenjajo komentarji.
- Druga vrstica vrne niz, ki se izpiše ob začetku podatkovnega segmenta.
- Tretja in četrta vrstica vrneta niz, ki se uporabi kot labela podprograma. Hkrati to prikazuje makro s parametri - take makre je priporočljivo definirati s funkcijami.
- Vrstici 5 in 6 prikazujeta, da je nekatere makre najprej potrebno onemogočiti.

7.3.3 Opis registrov

Pomemben makro je `FIRST_PSEUDO_REGISTER`, definiran tako:

```
#define FIRST_PSEUDO_REGISTER 37
```

`FIRST_PSEUDO_REGISTER` pove prevajalniku GCC, koliko registrov je fizičnih. S tem si GCC pomaga pri začasnem dodeljevanju spremenljivk registrom.

V številki `37` je zajetih 32 splošnonamenskih, `PC`, `EPC`, `I` ter skladovni kazalec `QFP` in register `QAP`, ki vsebuje kazalec na položaj na skladu, kjer se nahaja prvi parameter podprograma.

`QFP` in `QAP` nista fizična, temveč se uporabljata le med prevajanjem, kasneje se ju nadomesti. Prevajalnik GCC register `QFP` med dodeljevanjem registrov nadomesti z registrom, ki je bil določen kot skladovni kazalec - pri HIP je to register `r30`. `QAP` se uporablja pri podprogramih, ki nimajo vnaprej določenega števila parametrov (`varargs`) - primer je funkcija `printf` v programskem jeziku C. Podpora za `QAP` je prisotna pri zadnjem delu za procesor Moxie, pri HIP pa ni bila testirana, čeprav je podpora ostala. Pri procesorju Moxie se register `QAP` nadomesti s fizičnim registrom, dejanski parametri se nato prenašajo prek sklada.

Prav tako niso v splošni rabi posebni registri `PC`, `EPC` in `I`. Register `PC` se uporablja med prevajanjem, medtem ko bi bila registra `EPC` in `I` potrebna le pri obravnavanju prekinitev. Vseeno njuna prisotnost ne škodi, tako je koda bolje pripravljena na morebitne razširitve.

Koda 7.5: Primeri opisa skupin registrov

```
enum reg_class {
  NO_REGS,
  GENERAL_REGS,
  SPECIAL_REGS,
  ALL_REGS,
  LIM_REG_CLASSES
};
```

```
8  #define REG_CLASS_CONTENTS              \
9  { { 0x00000000, 0x00000000 }, /* No registers */       \
10   { 0xFFFFFFFF, 0x00000003 }, /* General registers: r0 to r31, qfp, qap */ \
11   { 0x00000000, 0x0000001C }, /* Special registers: pc, epc, i */    \
12   { 0xFFFFFFFF, 0x0000001F } /* All registers */       \
13   /* { 31 <- 0, 63 <- 32 } */          \
14 }
```

- `enum reg_class` v kodi 7.5 je obvezno oštevilčenje, ki poda skupine registrov. Prevajalnik GCC potrebuje te skupine pri dodeljevanju registrov.

  V primeru procesorja HIP je to enostavno, ker nima veliko različnih skupin registrov, le splošnonamenske (`GENERAL_REGS`) in pet posebnih (`SPECIAL_REGS`).

  Za pravilno delovanje morajo biti v pravem vrstnem redu navedene še tri skupine.

  `NO_REGS` se običajno uporablja kot sporočilo, da neke operacije ni možno izvršiti v registrih.

  `ALL_REGS` je skupina vseh fizičnih registrov. Ni sicer nujno, da so vsi res fizično prisotni v arhitekturi, zajema pa vse registre, ki so oštevilčeni nižje kot `FIRST_PSEUDO_REGISTER`.

  Na koncu je še vrednost `LIM_REG_CLASSES`, ki pove število različnih skupin registrov.

- Makro `REG_CLASS_CONTENTS` v kodi 7.5 določi, v katere skupine spadajo posamezni registri. V tem makru so števila zaradi boljše preglednosti predstavljena šestnajstiško. Vsako število, ki je 32-bitno, hrani podatek o 32 registrih: če je `i`-ti bit števila enak 1, register z zaporedno številko `i` spada v to skupino, sicer ne spada.

  Ta makro upošteva nekoliko nenavadno zaporedje registrov. Za več kot 32 registrov je potrebnih več števil, ločenih z vejico, obdanih z zavitimi

oklepaji. Vendar se znotraj števila registri številčijo z desne proti levi: najbolj desni bit v zapisu števila predstavlja najnižjo številko registra. To je prikazano v zakomentirani vrstici 13.

Koda 7.6: Primer opisa namena registrov

```
1 #define FIXED_REGISTERS {    \
2   1, 0, 0, 0, 0, 0, 0, 0, /* r0 - r7*/ \
3   0, 0, 0, 0, 0, 0, 0, 0, /* r8 - r15*/ \
4   0, 0, 0, 0, 0, 0, 0, 0, /* r16 - r23*/ \
5   0, 0, 0, 0, 0, 0, 1, 0, /* r24 - r31*/ \
6   1, 1, 1, 1, 1   /* special */ }
```

- Makro `FIXED_REGISTERS` v kodi 7.6 za posamezen fizičen register, ali je fiksen. Fiksen register pomeni, da ga prevajalnik GCC ne sme uporabiti za pisanje. Za pisanje v te registre - če je to sploh dovoljeno - skrbi snovalec zadnjega dela. Pri procesorju HIP so kot fiksni označeni vsi posebni registri ter registra `r0` in `r30`.

  Register `r0` je konstantno 0, zato je potrebno preprečiti pisanje vanj. `r0` se bo uporabljal pri nekaterih ukazih, sicer pa ga GCC niti ne potrebuje.

  Register `r30` je skladovni kazalec. Ker se njegova vrednost nadzorovano spreminja, naj se ne piše vanj.

- Podoben makro je `CALL_USED_REGISTERS`, ki pove, kateri registri se lahko zamažejo (se piše vanje) med klicem podprograma. Če se v register ne sme pisati, je v makru na pravem mestu 0. Če se v tak register med izvajanjem podprograma piše, ga je potrebno ob vstopu v podprogram shraniti na sklad in ob izstopu obnoviti.

  Pri procesorju HIP se enica (vrednosti registra ni potrebno ohranjati) pojavi pri registrih `r0`, `r24`, `r25`, `r28`, `r30` in posebnih registrih.

Pri registrih `r0`, `r30` in posebnih registrih je utemeljitev podobna kot pri `FIXED_REGISTERS`, namreč da je snovalec zadnjega dela odgovoren za njihovo dodeljevanje.

Registra `r24` in `r25` prenašata prva dva parametra podprograma in ju po klicni konvenciji ni potrebno ohraniti. Register `r28` vrača vrednost podprograma, zato se mora zamazati med klicem.

7.3.4 Opis sklada

Koda 7.7: Primer makrov za opis sklada

```
1 #define STACK_PUSH_CODE POST_DEC
2 #define FRAME_GROWS_DOWNWARD 1
3 #define FUNCTION_ARG_REGNO_P(r) (r >= 24 && r <= 25)
4 #define STACK_POINTER_REGNUM 30
```

Koda 7.7 vsebuje nekaj makrov, ki se nanašajo na sklad pri procesorju HIP:

- `STACK_PUSH_CODE` in `FRAME_GROWS_DOWNWARD` povesta, da sklad raste navzdol. Zmanjšanje skladovnega kazalca se zgodi šele po shranitvi registra na sklad (push).
- `FUNCTION_ARG_REGNO_P(r)` sporoča, kateri registri so namenjeni prenosu parametrov.
- `STACK_POINTER_REGNUM` obvesti prevajalnik GCC, kateri fizični register se uporablja kot skladovni kazalec.

7.4 hip.md

Datoteka `hip.md` je ključna pri tlakovanju vmesne kode prevajalnika GCC z ukazi ciljne arhitekture. Ko želi prevajalnik GCC ukaz vmesne kode pretvoriti v zbirni/strojni ukaz, pregleda datoteko `hip.md` za morebitne ujemajoče vzorce.

Ta datoteka se imenuje `GCC Machine Description` [7] oziroma `MD` in uporablja posebno sintakso, ki je podobna programskemu jeziku Lisp in omogoča drevesno sestavo kode. Datoteka se pregleda dvakrat: prvič za splošno ujemanje in enostavno tlakovanje, drugič za natančnejše ujemanje in morebitno optimizacijo.

Če se med tlakovanjem noben vzorec ne ujema z vmesno kodo, se lahko zgodi več možnosti:

- Nekateri ukazi so obvezni že ob prevajanju prevajalnika GCC, torej se brez njih ne bo uspel prevesti.

- Drugi ukazi so obvezni ob izvajanju prevajalnika, torej prevajanju programske kode v ciljno strojno kodo. Ti ukazi se uporabijo šele ob ustrezni vmesni kodi - če se med prevajanjem taka vmesna koda ne tvori, ni težav. V nasprotnem primeru prevajalnik javi napako ‘‘`Internal compiler error`’’.

  Dokumentacija [4] žal ni preveč jasna, kateri ukazi so obvezni.

- Za ostale ukaze lahko GCC nekoliko spremeni vmesno kodo in ponovno poskusi najti vzorce. Ta korak običajno uspe, ni pa to zagotovljeno. Tudi tukaj se v primeru neuspeha javi napaka.

7.4.1 define_insn

`define_insn` je najpomembnejša vrsta vzorca, ker definira ukaz. Pri tlakovanju se tak vzorec uporablja neposredno za primerjavo z vmesno kodo.

Koda 7.8: Primer definicije ukaza addsi3

```
1  (define_insn "addsi3"
2    [(set
3      (match_operand:SI 0 "register_operand" "=r, =r, =r")
4      (plus:SI
5        (match_operand:SI 1 "register_operand" "r, r, %r")
6        (match_operand:SI 2 "hip_reg_or_int" "I, N, r")
7      )
8    )]
9    ""
10   "@
11   addui %0, %1, %2
12   subui %0, %1, %n2
13   addu %0, %1, %2"
14 )
```

Koda 7.8 prikazuje primer ukaza v datoteki `hip.md`, ki nazorno prikaže sestavne dele vzorca. Navadni oklepaji predstavljajo poddrevesa vmesne kode.

Vzorec `define_insn` vsebuje ime vzorca (prva vrstica), vzorec vmesne kode (v oglatih oklepajih, vrstice `2` do `8`), dodatne pogoje v vrstici `9` ter možne strojne ukaze (vrstice `10` do `13`).

Sestavni deli kode 7.8 natančneje:

- Ime vzorca je lahko prazno. Prevajalnik GCC v prvi vrsti opazuje vzorce vmesne kode. Vseeno so nekatera imena obvezna, kajti strojne ukaze se lahko tvori tudi s klici funkcij v jeziku C. To se opravi s klicem funkcije `gen_`***`insn`***, kjer ***`insn`*** predstavlja ime vzorca.

- Vzorec vmesne kode ima drevesno sestavo in lahko vsebuje več možnosti

prepoznave. Pogosto je oblike `(set (t1 ...) (t2 ...))`, kar nakazuje, da se rezultat izraza `t2` shrani v `t1`.

- Operande se najprej preverja v vzorcu vmesne kode, vendar se lahko poda dodatne pogoje v obliki C kode, ki vrača logično vrednost. V primeru 7.8 dodatnih pogojev ni.

- Podanih mora biti toliko strojnih ukazov, kolikor je različnih vzorcev, ki jih ta vzorec vmesne kode definira. Znak `@` na začetku pove, da nastopa več kot en ukaz.

  V primeru 7.8 so navedeni trije strojni ukazi, ker ta vzorec vmesne kode pokrije tri možnosti operandov.

Namesto niza znakov pri generiranju strojnih ukazov se lahko piše tudi C kodo, obdano z zavitimi oklepaji, ki vrača niz znakov. V vzorec tipa `define_insn` vstopajo operandi v obliki polja (*array*) izrazov vmesne kode `RTX`. To je posebej koristno pri rabi C kode za generiranje strojnih ukazov. V tem primeru se lahko do `i`-tega operanda dostopa z `operands[i]`. Za prepoznavo, za katero možnost gre, se v jeziku C uporablja številska spremenljivka `which_alternative`.

Do sedaj še ni bilo razčiščeno glede različnih možnosti generiranja strojne kode v enem vzorcu. Za ta namen se lahko podrobneje pregleda vzorec vmesne kode iz primera 7.8:

- `match_operand` se uporablja za prepoznavo operandov. Podano mora imeti velikost - v tem primeru je `SI`. `SI` je okrajšava, ki jo prevajalnik GCC uporablja za 32-bitna cela števila, *Single Integer*.

- Sledi zaporedna številka operanda.

- Naslednji del je predikat, ki v prvem prehodu ugotavlja, ali je operand ustrezen.

- Nazadnje je prisoten še seznam različnih možnosti operandov, ločenih z vejicami. To so dodatne omejitve predikatu, vendar se preverjajo šele v drugem prehodu.

Nekatere omejitve so standardne, možno je dodajati lastne. Ker je spisek vseh precej obsežen [4], je tukaj le kratek pregled:

- `r` označuje splošnonamenski register.
- `I` in `N` sta določena posebej za HIP, in sicer `I` označuje celoštevilsko konstanto na intervalu `[0, 65535]`, `N` pa celoštevilsko konstanto na intervalu `[-65535, 0]`.
- Omejitve imajo lahko dodatne možnosti. Pri prvem operandu (operand z indeksom `0`) v kodi 7.8 je pri vseh alternativah enačaj `=`, ki pove, da se prejšnja vrednost operanda izgubi in se jo nadomesti z novo.
- Znak `%` pri omejitvi pomeni, da je dovoljeno prevajalniku zamenjati trenutni operand z naslednjim - da velja komutativnost. To je lahko koristno pri optimizacijah.
- Omejitve se lahko dodaja z `define_constraint`. Koda 7.9 prikazuje dodajanje prej omenjene omejitve `I`, ki zahteva celo število na intervalu `[0, 65535]`.

Koda 7.9: Primer dodajanja lastne omejitve

```
1 (define_constraint "I"
2   "A 16-bit constant [0..65535]"
3   (and
4     (match_code "const_int")
5     (match_test "ival >= 0 && ival <= 65535")
6   )
7 )
```

Za večjo razumljivost je v kodi 7.10 prikazan še konkreten primer vmesne kode, ki bi ustrezal vzorcu 7.8.

Koda 7.10: Primer vmesne kode, ki ustreza vzorcu 7.8

```
(set
  (reg 1)
  (plus
    (reg 2)
    (reg 3)
  )
)
```

Vmesna koda 7.10 bi se s pomočjo vzorca 7.8 pretvorila v naslednji ukaz:

```
addu r1, r2, r3
```

7.4.2 define_expand

Tip vzorca `define_expand` je zasnovan splošneje kakor `define_insn`. Lahko pokrije večji nabor vmesne in lahko tvori osnovo večim `define_insn` vzorcem - ni pa to nujno. V osnovi ima dve možnosti delovanja:

- `define_expand` lahko tvori podlago za nadaljne `define_insn`. S tem lahko zagotovi izpolnjenost nekaterih pogojev za več vzorcev hkrati ali pa le za en vzorec, vendar zajema splošnejše pogoje. Primer je, da tip operanda ni določen s pomočjo MD kode, temveč se ga preverja s pomočjo ukazov v jeziku C. To je prikazano v kodi 7.11:

Koda 7.11: Primer `define_expand` vzorca `movqi`

```
1  (define_expand "movqi"
2    [(set
3      (match_operand:QI 0 "general_operand" "")
4      (match_operand:QI 1 "general_operand" "")
5    )]
6    ""
7  {
8    if(MEM_P(operands[0]))
9      operands[1] = force_reg(QImode, operands[1]);
10 }
11 )
```

Koda 7.11 prikazuje ukaz `movqi`, ki premakne en bajt (`QI`, Quarter Integer) iz drugega (indeks `1`) v prvi operand (indeks `0`). Če gre za pisanje v pomnilnik, mora biti drugi operand register.

Sestava `define_expand` je zelo podobna `define_insn`. Pomembna razlika je, da se lahko C kodo v zavitih oklepajih (vrstice `7` do `10`) popolnoma izpusti. Prav tako ta C koda ne vrača ničesar - za razliko od `define_insn`, ki mora na tem mestu vrniti niz znakov.

- `define_expand` lahko deluje samostojno, brez nadaljnih `define_insn` vzorcev. S pomočjo C kode je moč pisati v končno datoteko z zbirno

kodo, kar prikazuje koda 7.12. `epilogue` je standardno ime vzorca. Ko C koda opravi potrebne operacije, je potrebno le še sporočiti, da tega vzorca ni potrebno razširjati, čemur služi makro `DONE`. Koda 7.12 prikazuje v drugi vrstici še ukaz vmesne kode `(return)`, ki sporoča prevajalniku konec prevajanja podprograma.

Koda 7.12: Primer `define_expand` vzorca `epilogue`

```
1 (define_expand "epilogue"
2   [(return)]
3   ""
4 {
5   hip_expand_epilogue();
6   DONE;
7 }
8 )
```

Poleg `DONE` obstaja tudi makro `FAIL`, ki sporoči, da nekaj ni bilo v redu. GCC posledično vzorca ne uporabi vzorca, kjer je naletel na `FAIL`. V tem primeru poskusi najti drug vzorec - če ne uspe, javi napako.

7.5 hip.c

Ta datoteka vsebuje pomožne funkcije. Čeprav je možno v makrih in opisih ukazov pisati polno C kodo, postane koda bolj pregledna, če se jo porazdeli na funkcije. Vendar je potrebno funkcijam, ki se bodo rabile izven te datoteke, definirati prototipe v datoteki `hip-protos.h`.

Koda 7.13: Funkcija hip_assemble_integer

```
1  static bool
2  hip_assemble_integer(rtx x, unsigned int size, int aligned_p) {
3    char name[6];
4    switch(size) {
5      case 1:
6        strcpy(name, "byte"); break;
7      case 2:
8        strcpy(name, "word16"); break;
9      case 4:
10       strcpy(name, "word"); break;
11     case 8:
12       strcpy(name, "word64"); break;
13     default:
14       fprintf(stderr, "Size of integer cannot be %u", size);
15       return false;
16   }
17   fprintf(asm_out_file, "\t.%s\t", name);
18   hip_print_operand(asm_out_file, x, 0);
19   fprintf(asm_out_file, "\n");
20   return true;
21 }
```

Koda 7.13 prikazuje funkcijo, ki je zadolžena za izpis zbirne kode, ki globalni spremenljivki določi začetno vrednost. Iz kode je razvidno, da je kazalec na izhodno zbirno datoteko `asm_out_file`.

Poglavje 8

Izvajanje C prevajalnika za HIP

Končni C prevajalnik za HIP se s skripto 6.1 namesti v mapo `/usr/local/gcc-hip-binaries/`. Izvršljiva datoteka `hip-gcc.exe` se nahaja v podmapi `bin`. V tej podmapi `bin` se morajo nahajati tudi datoteke, ki jih želimo prevesti (razen če se ta podmapa doda v okoljsko spremenljivko `PATH`). Ta navodila se nanašajo na ukazno vrstico operacijskega sistema Windows, ne na okolje `MSYS` (čeprav recimo v sistemu Linux ne bi bilo dosti drugače). Polna pot do podmape `bin` je običajno `C:\MinGW\msys\1.0\local\gcc-hip-binaries\bin`.

Predpostavimo, da se v podmapi `bin` nahaja datoteka `test.c`, ki vsebuje izvorno kodo v programskem jeziku C. C prevajalnik za HIP se požene z ukazom:

```
1 hip-gcc.exe -S test.c
```

Ta ukaz vrne datoteko z zbirno kodo `test.s`. Lahko se uporablja večina zastavic prevajalnika GCC, vendar je najpomembnejša zastavica `-S`, ki vrača zbirno kodo. Brez nje bi GCC poskušal tvoriti strojno kodo, kar zaradi manjkajočega povezovalnika ni možno, zato bi GCC javil napako.

8.1 Primer klica podprograma

Koda 8.1: Primer enostavnega klica podprograma

```
1 int a;
2 int b = 5;
3 int sestej(int a, int b, int c) {
4   return a + b + c;
5 }
6 int main() {
7   a = 3;
8   return sestej(a, b, 2);
9 }
```

Koda 8.1 vsebuje zelo enostaven program v jeziku C, ki sešteje tri števila prek klica funkcije `add`. Spremenljivki `a` je potrebno pred klicem še nastaviti vrednost.

Koda 8.2: Prevod enostavnega programa 8.1

```
1   .text
2   lhi  r30, 0x100000
3   addui r30, r30, 0x100000
4   lhi  r26, main
5   addui r26, r26, main
6   j  0(r26)
7   .data
8 a: .space 4
9 ; b
10 b: .word 5
11   .text
12 ; sestej
13 sestej:
14   sw  0(r30), r31
15   subui r30, r30, 4
16   sw  0(r30), r29
```

```
17    subui r30, r30, 4
18    addu r29, r0, r30
19    subui r30, r30, 8
20    sw  0(r30), r1
21    subui r30, r30, 4
22    sw  0(r30), r2
23    subui r30, r30, 4
24    sw  0(r29), r24
25    sw  −4(r29), r25
26    lw  r2, 0(r29)
27    lw  r1, −4(r29)
28    addu r2, r2, r1
29    lw  r1, 12(r29)
30    addu r1, r2, r1
31    addu r28, r0, r1
32    addui r30, r30, 4
33    lw  r2, 0(r30)
34    addui r30, r30, 4
35    lw  r1, 0(r30)
36    addu r30, r0, r29
37    addui r30, r30, 4
38    lw  r29, 0(r30)
39    addui r30, r30, 4
40    lw  r31, 0(r30)
41    j  0(r31)
42  ; main
43  main:
44    sw  0(r30), r31
45    subui r30, r30, 4
46    sw  0(r30), r29
47    subui r30, r30, 4
48    addu r29, r0, r30
49    sw  0(r30), r1
```

```
50  subui r30, r30, 4
51  sw  0(r30), r2
52  subui r30, r30, 4
53  sw  0(r30), r3
54  subui r30, r30, 4
55  addui r1, r0, 3
56  sw  a(r0), r1
57  lw  r2, b(r0)
58  lw  r1, a(r0)
59  addui r3, r0, 2
60  sw  0(r30), r3
61  subui r30, r30, 4
62  addu r25, r0, r2
63  addu r24, r0, r1
64  call r31, sestej(r0)
65  addui r30, r30, 4
66  addu r1, r0, r28
67  addu r28, r0, r1
68  addui r30, r30, 4
69  lw  r3, 0(r30)
70  addui r30, r30, 4
71  lw  r2, 0(r30)
72  addui r30, r30, 4
73  lw  r1, 0(r30)
74  addu r30, r0, r29
75  addui r30, r30, 4
76  lw  r29, 0(r30)
77  addui r30, r30, 4
78  lw  r31, 0(r30)
79  j  0(r31)
```

Zbirna koda 8.2, ki je rezultat prevoda kode 8.1, je zelo obsežna za tako enostaven program. Kode se je najbolje lotiti po delih:

- Vrstice `1` do `6` so del inicializije. Ta koda se doda na začetek vsakega programa, da se vzpostavi sklad ter skoči na naslov glavne funkcije `main`.

- Vrstice `7` do `10` predstavljajo globalni spremenljivki `a` in `b`. Zanimiva je predvsem deveta vrstica, ki vsili odvečno labelo za spremenljivko `b`. Vzrok tega mi je neznan, vendar je moč to zaobiti z dodatnim komentarjem (tega ni potrebno dodajati ročno). Ta težava se pojavlja tudi pri labelah podprogramov.

- Vrstice `12` do `41` so prevod funkcije `sestej`. Razvidni so glavni sestavni deli: prolog (`14` do `23`), telo (`24` do `31`) in epilog (`32` do `41`) podprograma.

  V telesu podprograma so zanimive vrstice `24` do `27`, kjer GCC najprej shrani registra s prvima dvema parametroma (`r24` in `r25`) na njuno predvideno mesto na skladu, nato pa ju takoj prebere v registra `r1` in `r2`. Po konvenciji se ta dva registra sploh ne bi smela shraniti na sklad. Njuno predvideno mesto sta prvi dve mesti, namenjeni lokalnim spremenljivkam - vrstica `19` namreč na skladu rezervira prostor za lokalne spremenljivke, v tem primeru 8 bajtov.

  Vrstice `25` do `31` so bolj v skladu s pričakovanji, saj se naloži še tretji parameter s sklada (odmik `12` od kazalca na okvir) in se seštejejo tri števila.

- Vrstice `42` do `79` predstavljajo glavno funkcijo `main`, predvsem zanimivo je telo funkcije v vrsticah `55` do `67`.

  Najprej se zapiše konstanta `3` v spremenljivko `a` (vrstici `55` in `56`). Nato se naložijo parametri za klic podprograma. Spremenljivki `a` in `b` se zapišeta v registra `r24` in `r25`. Konstanta `2`, ki se prenaša prek sklada, se potisne na sklad v vrsticah `59` do `61`.

  V vrstici `64` je klic podprograma `sestej`. Takoj v naslednji vrstici je razvidno, da se dodatni parametri pobrišejo s sklada z enostavnim

povečanjem skladovnega kazalca. Vrstici `66` in `67` premakneta rezultat klicanega podprograma v začasen register, nato pa še glavna funkcija `main` vrne isto vrednost - tukaj je očitno preveč kopiranja istih podatkov.

Izsek kode 8.2 je precej obsežnejši, kot bi bilo potrebno. Nekaj je nepotrebnega kopiranja podatkov in verjetno bi se dalo program 8.1 implementirati z manjšim naborom registrov. Tukaj je v veliko pomoč dejstvo, da je GCC razširjen prevajalnik: ima namreč dobro podporo za optimizacije. Čeprav v izvorni kodi zadnjega dela za HIP ni nikjer izrecno zapisano, kakšne naj bodo optimizacije (kar je sicer možno narediti), se lahko vseeno vključi optimizacije z dodatkom zastavice `-O1`.

Koda 8.3: Prevod enostavnega programa 8.1 z optimizacijo `-O1`

```
1  sestej:
2    sw  0(r30), r31
3    subui r30, r30, 4
4    sw  0(r30), r29
5    subui r30, r30, 4
6    addu r29, r0, r30
7    addu r24, r24, r25
8    lw  r28, 12(r29)
9    addu r28, r24, r28
10   addu r30, r0, r29
11   addui r30, r30, 4
12   lw  r29, 0(r30)
13   addui r30, r30, 4
14   lw  r31, 0(r30)
15   j  0(r31)
16 main:
17   sw  0(r30), r31
18   subui r30, r30, 4
19   sw  0(r30), r29
20   subui r30, r30, 4
```

```
21    addu r29, r0, r30
22    addui r24, r0, 3
23    sw  a(r0), r24
24    addui r25, r0, 2
25    sw  0(r30), r25
26    subui r30, r30, 4
27    lw  r25, b(r0)
28    call r31, sestej(r0)
29    addui r30, r30, 4
30    addu r30, r0, r29
31    addui r30, r30, 4
32    lw  r29, 0(r30)
33    addui r30, r30, 4
34    lw  r31, 0(r30)
35    j  0(r31)
36    .data
37 b: .word 5
38 a: .space 4
```

Zbirna koda 8.3 je že precej krajša in bolj razumljiva. Zaradi nekoliko večje preglednosti ni vključena obvezna začetna inicializacija, ki ostane ista, prav tako so odstranjene zakomentirane vrstice.

Nekaj ključnih razlik:

- Področje spremenljivk se je premaknilo čisto na konec (od `36` naprej). V tem primeru to ni v pomoč, škodi pa tudi ne. Na tak način bi se pri večjih programih verjetno bolje izkoristila lokalnost podatkov.

- V obeh podprogramih sta se skrajšala prolog in epilog, ki obnavljata le še registra `r29` in r̃31. Za operacije so namreč na voljo registri `r24`, `r25` in `r28`, ki jih ni potrebno obnavljati.

- Podprogram `sestej` ne vsebuje več shranjevanja prvih dveh parametrov na sklad, zato tudi ni potrebno rezervirati prostora zanju. Tretji

parameter se uporablja enako kot prej, saj je bil na enak način potisnjen na sklad.

- V glavni funkciji `main` se je prav tako zmanjšalo število ukazov zaradi boljše izrabe registrov. Registra `r28` se sploh ne uporablja, kajti funkcija `main` vrača isto vrednost kakor klicani podprogram `sestej`.

GCC podpira tudi naprednejše optimizacije prek zastavic `-O2` in `-O3`. Zastavica `-O2` le odpravi klic podprograma `sestej` v glavni funkciji `main`, kar je razvidno v zbirni kodi 8.4, kjer vrstici `9` in `10` poskrbita za izračun rezultata. GCC lahko nekatera števila izračuna že med prevajanjem. Ker ni klica podprograma, tudi ni potrebno pripravljati parametrov klica. Podprogram `sestej` ostane enak.

Zanimivo je, da zastavica `-O3` ne povzroči prav nobenih sprememb v zbirni kodi glede na `-O2`. Možno je, da bi taka stopnja optimizacije prišla v poštev šele pri večjih programih.

Koda 8.4: Glavna funkcija `main` programa 8.1 z optimizacijo `-O2`

```
1  main:
2    sw  0(r30), r31
3    subui r30, r30, 4
4    sw  0(r30), r29
5    subui r30, r30, 4
6    addu r29, r0, r30
7    addui r24, r0, 3
8    sw  a(r0), r24
9    lw  r28, b(r0)
10   addui r28, r28, 5
11   addu r30, r0, r29
12   addui r30, r30, 4
13   lw  r29, 0(r30)
14   addui r30, r30, 4
15   lw  r31, 0(r30)
16   j  0(r31)
```

8.2 Primer izvajanja zanke

Koda 8.5: Primer zanke s pomočjo Fibonaccijevega zaporedja

```
1  int fib(int n) {
2    int n2 = 0;
3    int n1 = 1;
4    if(n < 2)
5      return n;
6    int rezultat, i;
7    for(i = 2; i <= n; ++i) {
8      rezultat = n1 + n2;
9      n2 = n1;
10     n1 = rezultat;
11   }
12   return rezultat;
13 }
14 int main() {
15   return fib(6);
16 }
```

Program 8.5 je nekoliko obsežnejši kakor 8.1. Funkcija `fib` s pomočjo zanke vrne `n`-to število v Fibonaccijevem zaporedju. Tukaj je bolj zanimiva zanka kakor klic podprograma, zato bo prikazana le funkcija `fib`. Zaradi večje preglednosti bo prikazana optimizirana koda (`-O2`). Čeprav v tem primeru ni velike razlike med optimizirano in neoptimizirano kodo, optimizacija vseeno pripomore k jasnosti.

Koda 8.6: Podprogram `fib` programa 8.5 (optimizacija `-O2`)

```
1  fib:
2    sw  0(r30), r31
3    subui r30, r30, 4
4    sw  0(r30), r29
5    subui r30, r30, 4
6    addu r29, r0, r30
```

```
7     sw  0(r30), r1
8     subui r30, r30, 4
9     sw  0(r30), r2
10    subui r30, r30, 4
11    sw  0(r30), r3
12    subui r30, r30, 4
13    sgti r1, r24, 1
14    xori r1, r1, 1
15    bne  r1, L2
16    addui r2, r0, 2
17    addui r25, r0, 1
18    addui r3, r0, 0
19  L3:
20    addu r28, r25, r3
21    addui r2, r2, 1
22    addu r3, r0, r25
23    addu r25, r0, r28
24    slt  r1, r24, r2
25    xori r1, r1, 1
26    bne  r1, L3
27    addu r24, r0, r28
28  L2:
29    addu r28, r0, r24
30    addui r30, r30, 4
31    lw  r3, 0(r30)
32    addui r30, r30, 4
33    lw  r2, 0(r30)
34    addui r30, r30, 4
35    lw  r1, 0(r30)
36    addu r30, r0, r29
37    addui r30, r30, 4
38    lw  r29, 0(r30)
39    addui r30, r30, 4
```

```
40  lw  r31, 0(r30)
41  j  0(r31)
```

Poglavje 9

Sklepne ugotovitve

Želji po pisanju programov v programskem jeziku C je bilo vsaj delno zadoščeno. Končni C prevajalnik za HIP, osnovan na prevajalniku GCC, zna prevajati precej obsežnejše programe od tistih, ki se jih obravnava pri predmetu Arhitektura računalniških sistemov. Osnovni prevod je včasih nekoliko nepregleden, vendar prevajalnik GCC omogoča optimizacijo končne zbirne kode. Prva stopnja optimizacije (zastavica –O1) pogosto vrne zbirno kodo, ki je programerju že bolj razumljiva. Druga stopnja optimizacije lahko odpravi nepotrebne klice podprogramov in izračuna konstante med prevajanjem. Tako optimizacijo bi programer lahko spregledal.

Kljub temu C prevajalnik za HIP ne more prevesti poljubnih programov, spisanih v jeziku C. Največjo težavo predstavlja standardna knjižnica. Ta ni podprta zaradi manjkajočega povezovalnika za procesor HIP.

Pisanje zadnjega dela za prevajalnik GCC ni enostavno opravilo. Dokumentacija je sicer zelo obsežna, vendar neprijazna programerjem, ki se šele spoznavajo z notranjo sestavo prevajalnika GCC. V veliko pomoč so dokumenti, podobni tej diplomski nalogi, kjer je namen podoben: spisati zadnji del prevajalnika GCC za doslej nepodprto procesorsko arhitekturo.

Ko je programer dovolj seznanjen z delovanjem prevajalnika GCC, poteka dodajanje zadnjega dela relativno tekoče. Zelo pogost pristop je namreč, da se vzame že spisan zadnji del za procesor, podoben obravnavanemu. V

vsakem primeru je potrebno dobro poznati arhitekturo procesorja ter uporabljene klicne konvencije.

Pravilnost končne zbirne kode je potrebno nenehno preverjati. V tej diplomski nalogi se je za ta namen uporabljal simulator procesorja HIP, WinHIP. Težavno je, da je potrebno za vsak popravek zadnjega dela prevesti celoten prevajalnik GCC, kar je trajalo vsaj pol minute, običajno celo štiri minute, kljub vključenim možnim pospešitvam prevajanja.

Končni prevajalnik gotovo vsebuje še kakšnega hrošča, ki je bil doslej spregledan. Za večino primerov pa bi moral C prevajalnik za HIP relativno dobro delovati. Moja želja je, da bi prevajalnik koristil čimveč ljudem, ki bi spoznavali delovanje procesorja HIP ali notranjo sestavo prevajalnika GCC.

Literatura

[1] Izvorna koda gcc hip. https://github.com/ProGTX/GCC-HIP, June 2013.

[2] Mips instruction reference. http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html, June 2013.

[3] Moxie logic wiki. http://moxielogic.org/wiki/index.php/Architecture, June 2013.

[4] Inc. Free Software Foundation. Gnu compiler collection internals. http://gcc.gnu.org/onlinedocs/gccint/, June 2013.

[5] Donald Knuth. Mmix 2009. http://www-cs-faculty.stanford.edu/~knuth/mmix.html, June 2013.

[6] Dušan Kodek. *Arhitektura in organizacija računalniških sistemov*. Šenčur, 2008.

[7] Savithri H. Venkatachalapathy. Porting gcc to x32v architecture. Master's thesis, Oregon State University, 2004.

[8] OSDev Wiki. Gcc cross-compiler. http://wiki.osdev.org/GCC_Cross-Compiler, June 2013.